Politecnico di Torino

Politecnico di Torino

Master's Degree in Quantum Engineering

A.a. 2024/2025

Graduation Session: October 2025

First Experimental Validation of the *Lagrange* On-Premise IQM Spark Superconducting Quantum Processor

Supervisors:
Prof. Bartolomeo MONTRUCCHIO
Dr. Emanuele DRI
Dr. Giacomo VITALI

Candidate:
Francesca ZAPPULLA

Abstract

This thesis presents an experimental benchmarking study of the IQM Spark, an on-premise five-qubit superconducting quantum device recently installed at Politecnico di Torino. The main objective is to provide a first validation of the quantum computer through practical experiments, in order to understand its capabilities and limitations.

In order to better evaluate the results obtained from the quantum machine, the defined set of experiments has also been performed using classical emulation, both noiseless and noisy. In particular, this stage made it possible to validate the workflow and to prepare the circuits in advance. Then, all experiments have been repeated on the actual IQM Spark quantum computer, and the results have been compared with those obtained on a fake backend. The comparison has provided a measure of consistency between the emulator and the physical quantum processor.

With the aim of providing an overview as wide as possible, both fault-tolerant and non fault-tolerant algorithms have been identified and implemented. More in detail, the study focuses on three applications. The first addresses a fault-tolerant version of the Deutsch-Jozsa algorithm, implemented using the error detection code [[4,2,2]]. This case shows how a simple quantum algorithm can be protected against noise by encoding logical qubits into a small code, and it highlights the trade-off between error detection and hardware resources. The second application investigates the Single-Impurity Anderson Model (SIAM), studied with a Variational Quantum Eigensolver (VQE) and, again, leveraging the [[4,2,2]] code to provide partial detection. For both the Deutsch-Jozsa algorithm and the SIAM problem, the experiments quantify the reduction of errors achieved by encoded implementations compared to their unencoded counterparts. The third application explores Quantum Reservoir Computing (QRC), a framework that leverages quantum systems for tasks such as time-series processing, classification and control. In this work, QRC is specifically applied to temporal signal prediction, with a model trained and tested on the `NARMA10` sequence, a widely used nonlinear benchmark in time-series prediction, consisting of a tenth-order Nonlinear Autoregressive Moving Average system, which probes the ability of the model to capture complex temporal dependencies. This implementation offers insight into the expected performance of such a class of algorithms on Noisy Intermediate-Scale Quantum (NISQ) devices.

Although the algorithms presented in this thesis have already been investigated on other quantum platforms based on different technologies, such as trapped ions and neutral-atom systems, they have never before been implemented on the IQM Spark quantum processor. The study required careful adaptation of the implementations to the hardware architecture and constraints, and it delivers a benchmark that highlights both the capabilities and the limitations of the device at the algorithmic level.

In conclusion, this thesis shows that the IQM Spark quantum computer can already serve as a platform for preliminary implementations of error detection, variational algorithms and quantum machine learning models. Although the results are limited in scale, they establish the first experimental reference point for future research on the IQM Spark processor.

Acknowledgements

Colgo l'occasione per ringraziare il Professor Bartolomeo Montrucchio, relatore di questa tesi, per l'entusiasmo con cui ha accolto la mia proposta e per il prezioso supporto che ne ha reso possibile la realizzazione.

Desidero inoltre esprimere la mia più sincera gratitudine ai ricercatori del team di Quantum Computing della *Fondazione LINKS* per avermi offerto l'opportunità di svolgere il tirocinio e la tesi presso la loro sede. Sono particolarmente riconoscente per aver avuto accesso prioritario al computer quantistico, esperienza che ha arricchito in maniera significativa il mio percorso di formazione e ricerca. Un ringraziamento speciale va ai miei due correlatori, Emanuele Dri e Giacomo Vitali, per la preziosa guida fornita nello sviluppo delle implementazioni e nella realizzazione di questo lavoro. La loro competenza e immensa disponibilità hanno rappresentato per me un riferimento costante durante tutte le fasi del progetto. Ringrazio anche Paolo Viviani per la costante disponibilità e per il supporto tecnico offerto con grande pazienza e tempestività, fondamentale per la buona riuscita del lavoro.

Rimanendo in ambito accademico, desidero esprimere la mia profonda gratitudine ai miei colleghi di corso, insieme ai quali ho intrapreso questo percorso nuovo, nato due anni fa. Ciò ha reso ogni nostro passo un atto di pionierismo e anche di coraggio. Le difficoltà non sono mancate, e spesso esse hanno messo alla prova non solo noi, ma anche i professori, chiamati a sperimentare insieme a noi un cammino ancora inedito. In questo contesto, la solidarietà reciproca e la collaborazione hanno rappresentato il sostegno costante che ci ha permesso di affrontare le sfide e avanzare insieme. Con consapevolezza e un pizzico di orgoglio, possiamo dire di aver gettato le basi di questa magistrale al Politecnico di Torino, tracciando i primi segni di un percorso che altri seguiranno. Magari non troveremo subito lavoro, ma almeno ne usciamo impavidi, temerari e con qualche aneddoto (più o meno divertente) da raccontare.

Ringrazio profondamente i miei genitori, fonte costante di incoraggiamento: nulla sarebbe stato possibile senza il loro sostegno, sia finanziario sia emotivo. Loro mi hanno insegnato il senso del dovere e del sacrificio e, cosa non da poco, mi hanno trasmesso la fiducia in me stessa necessaria per affrontare ogni sfida con coraggio e determinazione, anche e soprattutto nei momenti di maggior sconforto.

Ringrazio, inoltre, tutte le persone che hanno dimostrato di volermi bene: dagli amici di lunga data, con cui da bambini scambiavamo cartoline durante le vacanze,

a quelli che ho incontrato più tardi lungo il cammino. Ho avuto modo di toccare con mano ciò che fino a quel momento conoscevo solo attraverso libri o consigli di persone sagge, e constatare quanto fosse vero. Le scoperte più inattese e più preziose si nascondono spesso nei momenti più ordinari: in un incontro di un gruppo di lettura in biblioteca, nella frustrazione di un esame di Analisi 1 quando la piattaforma di ateneo non funziona, tra i banchi di un'aula colma di studenti spaesati dopo una lezione di Analisi 2, durante il semplice gesto di andare a ritirare la smartcard al Politecnico, o in quell'istante in cui una richiesta di aiuto su un assignment particolarmente complesso di Elettromagnetismo Applicato si trasforma in un'occasione di dialogo e confronto sulle proprie esperienze di vita. *Amicum an nomen habeas, aperit calamitas* (Publilio Siro). Se tu abbia un amico, o solo uno che si dice tale, lo chiarirà la sventura.

Vorrei infine esprimere la mia più profonda riconoscenza a una persona speciale incontrata nella parte finale di questo percorso, la cui presenza costante e il sostegno affettuoso hanno saputo illuminare i miei giorni più difficili e riaccendere in me la speranza, con un calore, un livello di partecipazione emotiva e una dolcezza che non avevo mai conosciuto prima e che, quindi, non credevo possibili.

Table of Contents

List of Tables

List of Figures

Acronyms

AIM
Anderson Impurity Model

API
Application Programming Interface

AWG
Arbitrary Waveform Generator

CPTP
Completely Positive Trace-Preserving

DMFT
Dynamical Mean-Field Theory

DTW
Dynamic Time Warping

HEMT
High-Electron-Mobility Transistor

HTTP
Hypertext Transfer Protocol

JSON
JavaScript Object Notation

MERA
Multiscale Entanglement Renormalization Ansatz

MQM
Macroscopic Quantum Model

NARMA
Nonlinear Autoregressive Moving Average

NIBP
Noise-Induced Barren Plateau

NISQ
Noisy Intermediate-Scale Quantum

NMSE
Normalized Mean Square Error

PDU
Power Distribution Unit

QAOA
Quantum Approximate Optimization Algorithm

QPU
Quantum Processing Unit

QRC
Quantum Reservoir Computing

SIAM
Single-impurity Anderson Model

SPC
Single-Parity Check

SQUID
Superconducting Quantum Interference Device

TWPA
Traveling Wave Parametric Amplifier

UPS
Uninterruptible Power Supply

VQA
Variational Quantum Algorithm

VQE
Variational Quantum Eigensolver

Part I

Theoretical Framework

Chapter 1

Introduction

Before delving into the heart of this thesis, it is appropriate to briefly recap the basic building blocks of quantum computing. From cryptography and optimization to materials science and machine learning, the list of potential applications goes far beyond our current imagination. Quantum technology is no longer the exclusive playground of theoretical speculation; there have been significant achievements in hardware, software, and algorithmic complexity.

The objective of this chapter is to offer a clear and concise introduction to these essential concepts, setting a solid foundation for the technical discussions that will follow.

1.1 Quantum Computing

As a rapidly advancing domain of study and innovation, quantum computing introduces a revolution in the way information is manipulated and processed. By exploiting the principles of quantum mechanics - such as superposition, entanglement, and quantum interference - quantum computers have the potential to tackle challenges beyond the reach of classical systems.

This section will first introduce the concept of qubits, followed by an overview of single and multi-qubit gates. Then, some fundamental quantum and hybrid algorithms will be presented.

1.1.1 Quantum Bit

Quantum computing relies on quantum bits, or *qubits*, which are two-level quantum systems represented by vectors in a two-dimensional Hilbert space. In classical computing, information is encoded in bits that can exist in one of two definite states: 0 or 1. In contrast, a qubit can be in a state of superposition of the two basis states $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \alpha \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{1.1}$$

The coefficients α and β belong to the set of complex numbers, i.e., $\alpha, \beta \in \mathbb{C}$, and must satisfy the normalization condition $|\alpha|^2 + |\beta|^2 = 1$. The set $\{|0\rangle, |1\rangle\}$ is known as the *computational basis* and forms an orthonormal basis for the two-dimensional Hilbert vector space.

Although a qubit can exist in a superposition of the basis states $|0\rangle$ and $|1\rangle$, performing a projective measurement in the computational basis $\{|0\rangle, |1\rangle\}$ causes the state $|\psi\rangle$ to collapse into one of the two basis states. The probability of obtaining outcome $|0\rangle$ is given by $P(0) = |\langle 0|\psi\rangle|^2 = |\alpha|^2$, and the probability of measuring $|1\rangle$ is $P(1) = |\langle 1|\psi\rangle|^2 = |\beta|^2$. These correspond to the squared magnitudes of the projections of $|\psi\rangle$ onto $|0\rangle$ and $|1\rangle$, respectively. By definition of probability, the total must satisfy $P(0) + P(1) = 1$, which implies that the state vector must be normalized. If it is not, one can normalize it by applying the following operation:

$$|\psi'\rangle = \frac{|\psi\rangle}{\| |\psi\rangle \|^2} \tag{1.2}$$

By relying on the normalization condition on the coefficients α and β, the state $|\psi\rangle$ can be conveniently expressed using polar coordinates. This allows rewriting the qubit as:

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\varphi}\sin\left(\frac{\theta}{2}\right)|1\rangle \tag{1.3}$$

where θ and φ represent the polar and azimuthal angles on the Bloch sphere, shown below.

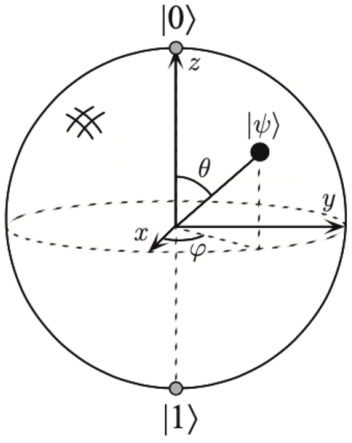


Figure 1.1: Representation of a qubit in the Bloch Sphere [1].

As already mentioned before, in quantum mechanics the state of a single qubit is described by a vector in a two-dimensional complex Hilbert space. More generally, particularly when dealing with statistical mixtures or partial information, the formalism of the density matrix ρ is employed. A density matrix is a positive semi-definite, Hermitian operator with unit trace. For a pure state, it can be written as $\rho = |\psi\rangle\langle\psi|$, while a mixed state is expressed as a convex combination of pure states:

$$\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|, \quad \text{with } p_i \geq 0, \quad \sum_i p_i = 1. \tag{1.4}$$

Any single-qubit state can be conveniently represented using the Bloch sphere formalism, in which the density matrix takes the form:

$$\rho = \frac{1}{2}\left(\mathbb{I} + \vec{r}\cdot\vec{\sigma}\right), \tag{1.5}$$

where $\vec{r} = (r_x, r_y, r_z)$ is the Bloch vector and $\vec{\sigma} = (\sigma_x, \sigma_y, \sigma_z)$ denotes the Pauli matrices. The identity operator $\mathbb{I}$ and the Pauli matrices form a basis for all Hermitian 2×2 operators.

The Pauli matrices are an essential set of Hermitian and unitary operators given by:

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{1.6}$$

The purity of a qubit state is determined by the norm of the Bloch vector. If $|\vec{r}| = 1$, the state is pure and lies on the surface of the Bloch sphere. If $|\vec{r}| < 1$, the state is mixed and represents a probabilistic ensemble of pure states.

For composite systems, such as two qubits, quantum states are classified as either separable or entangled. A bipartite state ρ_{AB} is separable if it can be written as a convex combination of product states:

$$\rho_{AB} = \sum_k p_k \, \rho_A^{(k)} \otimes \rho_B^{(k)}, \quad \text{with } p_k \geq 0, \quad \sum_k p_k = 1. \tag{1.7}$$

If no such decomposition exists, the state is said to be entangled. Entangled states exhibit correlations that cannot be explained by any classical probabilistic model.

A well-known example is the Bell state:

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right). \tag{1.8}$$

This state cannot be expressed as a product of two single-qubit states. Assuming a separable form:

$$|\Phi^+\rangle = (a|0\rangle + b|1\rangle) \otimes (c|0\rangle + d|1\rangle), \tag{1.9}$$

would lead to the expression:

$$|\Phi^+\rangle = ac|00\rangle + ad|01\rangle + bc|10\rangle + bd|11\rangle. \tag{1.10}$$

Equating this to the original Bell state imposes the following conditions:

$$ac = \frac{1}{\sqrt{2}}, \quad bd = \frac{1}{\sqrt{2}}, \quad ad = 0, \quad bc = 0.$$

These equations cannot be satisfied simultaneously unless some coefficients vanish, which contradicts the requirement that both ac and bd be non-zero. Therefore, the Bell state cannot be decomposed into product form and is entangled. The term *entanglement* refers to the fact that a measurement of the first qubit on the computational basis immediately determines the outcome of the second qubit: in the case of the Bell state $|\Phi^+\rangle$, if the first qubit is found in state $|0\rangle$, the second is also in $|0\rangle$; if the first is in $|1\rangle$, so is the second. Formally, conditional probabilities such as $P_B(1 \mid A = 1) = 1$ hold in the ideal case, illustrating the perfect correlation between the outcomes.

Entanglement is a non-classical type of correlation in which the measurement of one qubit instantaneously determines the state of the other, regardless of the spatial separation between the two subsystems. Thanks to this property, it finds application in quantum teleportation, superdense coding, and secure quantum communication. Moreover, it is essential for achieving computational advantages in various quantum algorithms.

1.1.2 Quantum Circuit

A quantum circuit is a mathematical model used to describe quantum computations. As in classical computing, a circuit is composed of wires and logic gates. In the quantum case, the wires represent qubits, and the gates correspond to unitary operations that manipulate quantum information. The circuit evolves a quantum state $|\psi\rangle$ from an initial input to a final output through a sequence of quantum gates.

For instance, let the system be initialized in a computational basis state $|x_1x_2\ldots x_n\rangle$, where $x_j \in \{0,1\}$. A quantum circuit applies a sequence of unitary operations $U_1, U_2, \ldots, U_k$ to this state, resulting in the transformation:

$$|\psi_{\text{out}}\rangle = U_k \ldots U_2U_1 \, |x_1x_2\ldots x_n\rangle \, . \tag{1.11}$$

Each U_j is a unitary matrix of dimension $2^n \times 2^n$, possibly acting only on a subset of qubits, and extended to the full Hilbert space via the tensor product with identity operators.

Unlike classical gates, quantum gates must be reversible (even more specifically, they must be unitary):

$$U^\dagger U = UU^\dagger = \mathbb{I}. \tag{1.12}$$

Before introducing more advanced logic operations, single-qubit gates must be defined: they act on a two-dimensional Hilbert space and are represented by 2×2 unitary matrices. They perform rotations and reflections on the Bloch sphere and are the building blocks of more complex quantum operations.

Pauli Gates (X, Y, Z): These gates correspond to π-rotations around the x, y, and z axes of the Bloch sphere. They can be expressed in matrix formalism:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{1.13}$$

Hadamard Gate (H): The action of the Hadamard gate creates an equally weighted superposition of all states in the computational basis $|x_1, \ldots, x_n\rangle$, with $x_j \in \{0,1\}$.

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \tag{1.14}$$

This gate is essential for creating quantum interference.

Phase Gate (S and T): Phase gates modify the phase of the $|1\rangle$ component of a state. They are defined as:

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \qquad T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}. \tag{1.15}$$

Both gates preserve the computational basis but apply complex phases, which are essential for interference.

Arbitrary Rotational Gates: Single-qubit rotations about the Bloch sphere axes are defined using the exponential of Pauli operators:

$$R_x(\theta) = e^{-i\frac{\theta}{2}X} = \cos\left(\frac{\theta}{2}\right)\mathbb{I} - i\sin\left(\frac{\theta}{2}\right)X, \tag{1.16}$$

$$R_y(\theta) = e^{-i\frac{\theta}{2}Y} = \cos\left(\frac{\theta}{2}\right)\mathbb{I} - i\sin\left(\frac{\theta}{2}\right)Y, \tag{1.17}$$

$$R_z(\theta) = e^{-i\frac{\theta}{2}Z} = \cos\left(\frac{\theta}{2}\right)\mathbb{I} - i\sin\left(\frac{\theta}{2}\right)Z. \tag{1.18}$$

These gates allow arbitrary rotations on the Bloch sphere.

At this point, it is possible to generalize the action of logical gates from one to multiple qubits.

Controlled operations apply a given unitary transformation only when all the control qubits are in state $|1\rangle$.

CNOT Gate (Controlled-NOT): The controlled-NOT gate acts on two qubits and flips the target qubit when the control qubit is $|1\rangle$. Its matrix in the basis $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ is expressed as:

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{1.19}$$

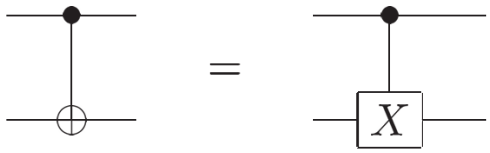


Figure 1.2: Two different representations for the controlled-NOT gate [1].

CZ Gate (Controlled-Z): The controlled-Z gate applies a Z operation to the target qubit conditioned on the control qubit being $|1\rangle$. It introduces a relative phase of -1 to the $|11\rangle$ state, leaving all others unchanged. Its matrix in the computational basis is expressed as:

$$\text{CZ} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}. \tag{1.20}$$

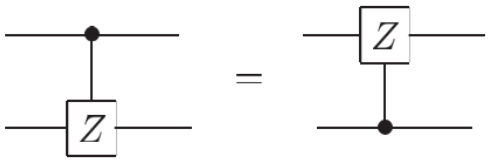


Figure 1.3: Symmetry of the controlled-Z gate [1].

This gate is symmetric (i.e., it commutes with qubit exchange) and is diagonal in the computational basis. When applied to the state $|+\rangle|+\rangle$, it produces the

entangled Bell state:

$$\mathrm{CZ}\left(\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)\right) = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle - |11\rangle) = |\Phi^-\rangle\,. \tag{1.21}$$

SWAP Gate: The SWAP gate is a two-qubit gate that exchanges the states of the two qubits. It is used to rearrange qubit positions in a quantum circuit without altering the information contained in the individual qubits. In the computational basis $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$, the SWAP gate is represented by the following 4×4 matrix:

$$\mathrm{SWAP} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{1.22}$$

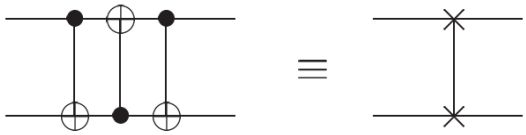


Figure 1.4: Representation of the SWAP gate [1].

Toffoli Gate (CCNOT): The Toffoli gate applies a NOT operation to the third (target) qubit only if the first two (controls) are both in state $|1\rangle$. It is represented by an 8×8 unitary matrix and is universal for classical reversible computation.

$$\mathrm{Toffoli} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}. \tag{1.23}$$

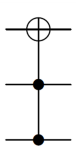


Figure 1.5: Representation of the Toffoli gate [1].

Finally, the measurement operation is applied at the end of a quantum circuit to extract classical information from the qubits, resulting in classical bits.

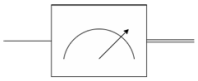


Figure 1.6: Representation of the measurement symbol [1].

Gate	Name	Matrix / Output	Effect on the state
$\boxed{\mathbf{X}}$	Pauli-X	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	Bit-flip: swaps amplitudes α and β
$\boxed{\mathbf{Y}}$	Pauli-Y	$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$	Bit and phase flip: adds $\pm i$ phase depending on state
$\boxed{\mathbf{Z}}$	Pauli-Z	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	Phase-flip: changes sign of $\lvert 1\rangle$ component
$\boxed{\mathbf{H}}$	Hadamard	$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$	Creates superpositions from basis states
$\boxed{R_x(\theta)}$	Rotation around X	$\begin{bmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$	Rotates state vector around X-axis by angle θ
$\boxed{R_y(\theta)}$	Rotation around Y	$\begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$	Rotates state vector around Y-axis by angle θ
$\boxed{R_z(\theta)}$	Rotation around Z	$\begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix}$	Adds phase shift by rotating around Z-axis
	CNOT	$\lvert a, b\rangle \mapsto \lvert a, b \oplus a\rangle$	Flips target qubit if control qubit is 1 (entangling gate)
	CZ	$\lvert a, b\rangle \mapsto (-1)^{ab} \lvert a, b\rangle$	Adds phase -1 when both qubits are 1
	SWAP	$\lvert a, b\rangle \mapsto \lvert b, a\rangle$	Exchanges the states of two qubits
	Toffoli	$\lvert a, b, c\rangle \mapsto \lvert a, b, c \oplus ab\rangle$	Flips target if both controls are 1

Table 1.1: Summary of the main logical quantum gates. The table lists commonly used single- and multi-qubit gates, including their symbolic representation, matrix form, and effect on the quantum state. These gates form the building blocks for most quantum circuits and algorithms.

1.1.3 Quantum Algorithms and Hybrid Algorithms

In quantum computing, a quantum algorithm is defined as a sequence of computational steps that can be implemented on a quantum computer, typically modeled using the quantum circuit framework. While classical algorithms consist of finite

sets of instructions executable by classical computers, quantum algorithms follow a similar step-by-step structure, with operations that exploit the principles of quantum mechanics. The term *quantum algorithm* is generally used to describe procedures that leverage distinctly quantum phenomena, such as superposition and entanglement, to achieve tasks that may be inefficient or infeasible using classical approaches.

At the core of many quantum algorithms lies the concept of *oracle*. The term originates from ancient mythology, referring to the Oracle of Delphi, a prophetic figure believed to provide cryptic yet insightful answers to important questions. The Oracle of Delphi was an important sanctuary in ancient Greece, located on Mount Parnassus. It was dedicated to the god Apollo, and its priestess, known as the Pythia, was the sole person authorized to pronounce prophecies on his behalf. Oracles were consulted for political, personal, and religious matters, and the sanctuary was considered the spiritual center of the Hellenic world.

In quantum computing, an oracle is a special type of quantum subroutine or function that behaves like a black box. It provides information about a computational problem by encoding part of the solution in the quantum state of the system. The oracle typically operates by marking (or identifying) certain basis states that satisfy a particular condition, without revealing any internal mechanism or logic used to perform this identification. This enables quantum algorithms to explore the solution space more efficiently, as the oracle can be queried in superposition, allowing the simultaneous evaluation of multiple inputs. Moreover, on a quantum computer, since operator must be both unitary and reversible, if a function takes n qubits as input and produces m qubits as output, the corresponding quantum gate requires an additional $m - n$ qubits as input, referred to as *ancillae* qubits. This ensures that the input can be uniquely determined from the output.

Quantum oracles can be generally classified into two main categories: *Boolean oracles* and *phase oracles*. Each type encodes information differently and plays distinct roles in quantum algorithms. A Boolean oracle is a unitary operator O_f that encodes a Boolean function

$$f : \{0,1\}^n \to \{0,1\}. \tag{1.24}$$

The oracle acts on two registers: an input register $|x\rangle$ with n qubits, and an ancilla qubit $|y\rangle$. Its action is defined as

$$O_f|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle, \tag{1.25}$$

In other words, the oracle flips the ancilla qubit if and only if $f(x) = 1$, otherwise it leaves it unchanged.

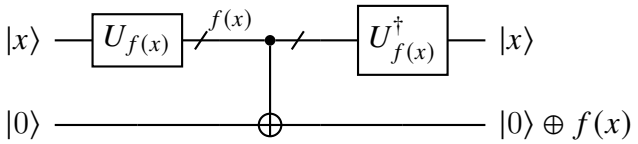
A phase oracle also encodes a Boolean function $f : \{0,1\}^n \to \{0,1\}$, but instead of flipping an ancillary qubit, it applies a phase factor to the input state. Its action on the input register alone is

$$O_f|x\rangle = (-1)^{f(x)}|x\rangle. \tag{1.26}$$

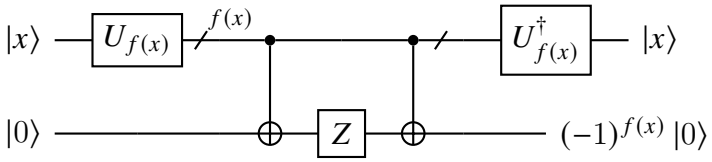
This means that the oracle multiplies the basis state $|x\rangle$ by a phase of -1 if $f(x) = 1$, and leaves it unchanged if $f(x) = 0$.

Phase oracles are useful in many quantum algorithms because they enable interference effects by modifying the relative phases of quantum states [2].

Such oracles play a fundamental role in algorithms like Grover's search and Deutsch-Jozsa, where their combination with quantum interference[1] leads to computational speedup.



(a) Boolean oracle



(b) Phase oracle

Figure 1.7: Comparison between the construction of a Boolean oracle (top) and of a phase oracle (bottom). In both circuits, $U_{f(x)}$ and $U^\dagger_{f(x)}$ refer to the unitary transformation implementing the function $f(x)$ and its conjugate, respectively. In the bottom circuit, the oracle operates as follows: the first CNOT transfers the value of $f(x)$ into the ancilla qubit. The Z gate then applies a phase shift of -1, but only when the ancilla qubit is flipped to $|1\rangle$ (i.e., when $f(x) = 1$). Finally, the second CNOT restores the ancilla qubit to $|0\rangle$, leaving the phase shift intact after being applied [2].

On the other hand, hybrid quantum-classical algorithms combine the strengths of both quantum and classical computing to solve problems that are challenging for either approach alone. These algorithms typically use a quantum processor to prepare and manipulate quantum states, while a classical computer optimizes parameters based on measurement results. One of the most prominent examples of such algorithms are *Variational Quantum Eigensolver* (VQE) and *Quantum Approximate Optimization Algorithm* (QAOA). In particular, the VQE is designed

[1]Quantum interference occurs when probability amplitudes associated to different quantum paths combine, either reinforcing or cancelling each other, which is essential to amplify the correct solutions marked by the oracle.

to find the lowest eigenvalue (ground state energy) of a Hamiltonian, which is a fundamental problem in quantum chemistry and materials science.

Deutsch-Jozsa Algorithm

The Deutsch-Jozsa algorithm was first proposed by David Deutsch and Richard Jozsa in 1992. It was intentionally constructed to be efficiently solvable by a quantum algorithm, while remaining difficult for any deterministic classical algorithm.

In the Deutsch-Jozsa problem, one is given access to a black-box oracle, which evaluates a function f taking binary inputs of length n, and for each such input, returns either 0 or 1. It is guaranteed in advance that the function is either constant (i.e., producing the same output for every input) or balanced (i.e., returning 1 for exactly half of the possible inputs and 0 for the other half). The goal is to determine, by querying the oracle, whether the function f is constant or balanced.

Let n be the number of bits in the input to the function f. In the worst case, a classical deterministic algorithm would require $2^{n-1}+1$ evaluations of f. This is because, in order to confirm that f is constant, it is necessary to check just over half of all possible inputs and verify that the output is the same in each case. Given the promise that f is either constant or balanced (but never something in between), this number of evaluations is sufficient to reach a conclusion. In the best case, where the function is balanced and the first two outputs are different, the determination can be made in only two evaluations.

In contrast, the Deutsch-Jozsa quantum algorithm requires only a single evaluation of f and always produces the correct result with certainty.

It is now appropriate to examine the specific steps of the algorithm, both from a mathematical perspective and in terms of circuit construction.
The derivations and explanations in this section are largely based on the presentation in Nielsen and Chuang's work [1].

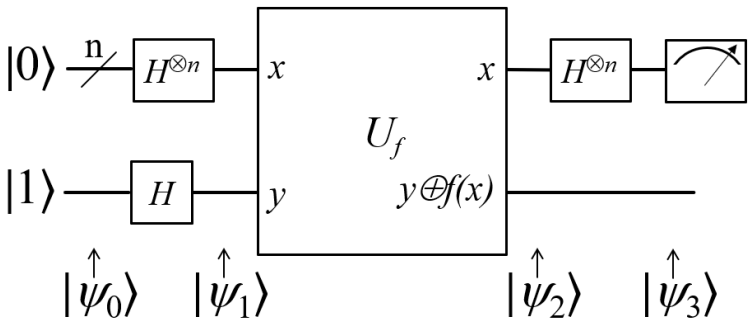


Figure 1.8: Quantum circuit implementing the Deutsch-Jozsa algorithm [1].

By relying on Fig. 1.8, it is possible to write the mathematical expression of the evolution of the state at each step.

First of all, the input state is characterized by n input qubits initialized in the $|0\rangle$ state and an additional ancilla qubit initialized in the $|1\rangle$ state:

$$|\psi_0\rangle = |0\rangle^{\otimes n}|1\rangle \tag{1.27}$$

After the action of the Hadamard gates, the query register becomes a superposition of all possible values:

$$|\psi_1\rangle = \left(\frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} |x\rangle\right) \otimes \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) \tag{1.28}$$

In the next step, the function f is evaluated by means of the unitary transformation $U_f : |x, y\rangle \mapsto |x, y \oplus f(x)\rangle$ and the state evolves as:

$$|\psi_2\rangle = \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} (-1)^{f(x)}|x\rangle \otimes \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) \tag{1.29}$$

In this precise step of the algorithm, the result of the evaluation of f is coded in the amplitude of the qubit superposition state. It is then necessary to apply a Hadamard transform to the query register to interfere the terms in the superposition.

To understand the effect of the Hadamard transform, it is useful to first analyze its action on a single qubit state $|x\rangle$. Considering the two cases $x = 0$ and $x = 1$ separately, one finds that

$$H|x\rangle = \frac{1}{\sqrt{2}} \sum_{z=0}^{1} (-1)^{xz}|z\rangle. \tag{1.30}$$

Extending this to n qubits, the transformation is given by

$$H^{\otimes n}|x_1, \ldots, x_n\rangle = \frac{1}{\sqrt{2^n}} \sum_{z_1,\ldots,z_n\in\{0,1\}} (-1)^{x_1z_1+\cdots+x_nz_n}|z_1, \ldots, z_n\rangle. \tag{1.31}$$

This result can be more compactly expressed as

$$H^{\otimes n}|x\rangle = \frac{1}{\sqrt{2^n}} \sum_{z\in\{0,1\}^n} (-1)^{x\cdot z}|z\rangle, \tag{1.32}$$

where $x \cdot z$ denotes the bitwise inner product modulo 2 of the bitstrings x and z.

Thus, one can obtain:

$$|\psi_3\rangle = \sum_z \sum_x \frac{(-1)^{x\cdot z+f(x)}}{2^n}|z\rangle \left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right] \tag{1.33}$$

At this point, the measurement of the query register is performed. It is noteworthy that the amplitude corresponding to the state $|0^{\otimes n}\rangle$ is given by $\sum_x \frac{(-1)^{f(x)}}{2^n}$. One proceeds by analyzing two distinct scenarios (where f is constant, and where f is balanced) based on the outcome.

Should f be a constant function, the amplitude for $|0\rangle^{\otimes n}$ will be either +1 or −1, depending on the fixed value assumed by f. Given that the state $|\psi_3\rangle$ possesses unit norm, it necessarily follows that all other amplitudes are zero. Consequently, an observation will invariably result in all qubits in the query register being measured as 0.

Conversely, if f is a balanced function, the constructive and destructive interferences within the amplitude for $|0^{\otimes n}\rangle$ will lead to a cancellation, resulting in an amplitude of zero. This implies that any measurement will necessarily produce a non-zero outcome for at least one qubit within the query register.

In summary, a measurement yielding all 0s indicates that the function is constant; otherwise, the function is balanced.

Bernstein-Vazirani Algorithm

Another oracle-based quantum algorithm is the Bernstein-Vazirani algorithm, invented in 1992 by Ethan Bernstein and Umesh Vazirani. It is very similar to the Deutsch-Jozsa algorithm, but this time the aim is to find an unknown or secret bitstring. Given a black-box function $f : \{0,1\}^n \rightarrow \{0,1\}$, the algorithm is designed to find s, such that $f(x) = s \cdot x \mod 2$.

As in the case of the Deutsch-Jozsa algorithm, solving this problem classically would require a check of each value one bit at a time, while thanks to quantum algorithms it is possible to find the secret bitstring s with a single query.

The circuit to be constructed to implement the Bernstein-Vazirani algorithm is depicted in Fig. 1.9.

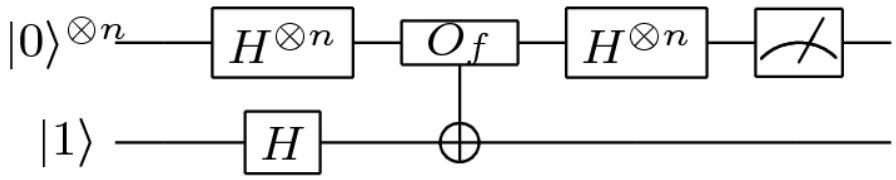


Figure 1.9: Quantum circuit implementing the Bernstein-Vazirani algorithm [3].

The Bernstein-Vazirani algorithms implements the same steps as the Deutsch-Jozsa algorithm [4]:

- All n input qubits are initialized to the ground state $|0\rangle$;
- The ancilla qubit is initialized to the excited state $|1\rangle$;

- Hadamard gates are applied to all input qubits and to the ancilla qubit to create superposition;
- The oracle is created with the aim of applying a phase shift based on the secret bistring s through CNOT gates;
- To create interference among the terms in the superposition, a Hadamard transform is applied to the query register;
- The input qubits are measured to obtain the secret bitstring s.

Grover's Algorithm

Grover's algorithm stands as a powerful quantum algorithm that provides a quadratic speedup for unstructured search problems. Its primary goal is to efficiently locate a unique "marked" item within an unsorted database of N elements, typically requiring about $O(\sqrt{N})$ queries to an oracle, a substantial improvement over the $O(N)$ queries needed by any classical algorithm. In essence, the search problem involves finding the input x_0 for which a boolean function $f(x)$ outputs 1, and 0 otherwise, with the algorithm operating on an n-qubit quantum register where $N = 2^n$.

The process begins by preparing all qubits in a uniform superposition state, achieved by applying the Hadamard operator to each qubit of the initial $|0\rangle^{\otimes n}$ state:

$$|\psi_0\rangle = H^{\otimes n}|0\rangle^{\otimes n} = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle \tag{1.34}$$

This state ensures that every possible item has an equal amplitude. Following this, the quantum oracle (U_f) comes into play, serving as a unitary operator that "marks" the target item by flipping the phase of the state corresponding to x_0, while leaving all other states untouched. Formally, its action is defined by:

$$U_f|x\rangle = (-1)^{f(x)}|x\rangle \tag{1.35}$$

Specifically, if $f(x_0) = 1$ and $f(x) = 0$ for $x \neq x_0$, then the oracle acts as:

$$U_f|x\rangle = \begin{cases} -|x\rangle & \text{if } x = x_0 \\ |x\rangle & \text{if } x \neq x_0 \end{cases} \tag{1.36}$$

After the oracle's application, the amplitude of the marked state becomes negative. This is where the Grover diffusion operator, also known as the diffuser (G_D or D), becomes crucial. This operator is often referred to as *inversion about the mean* because it amplifies the amplitude of the marked state while simultaneously reducing the amplitudes of all other states (see Fig. 1.10). The diffuser is defined as:

$$G_D = 2|\psi_0\rangle\langle\psi_0| - I \tag{1.37}$$

where I is the identity operator. Its effect can be geometrically visualized as a reflection around the average amplitude vector. By inverting the amplitudes relative to their mean, the marked state's amplitude, which had its phase flipped by the oracle, experiences a significant increase, while the amplitudes of the unmarked states proportionally decrease. A single Grover iteration consists of applying the oracle followed by the diffusion operator:

$$G = G_D U_f \tag{1.38}$$

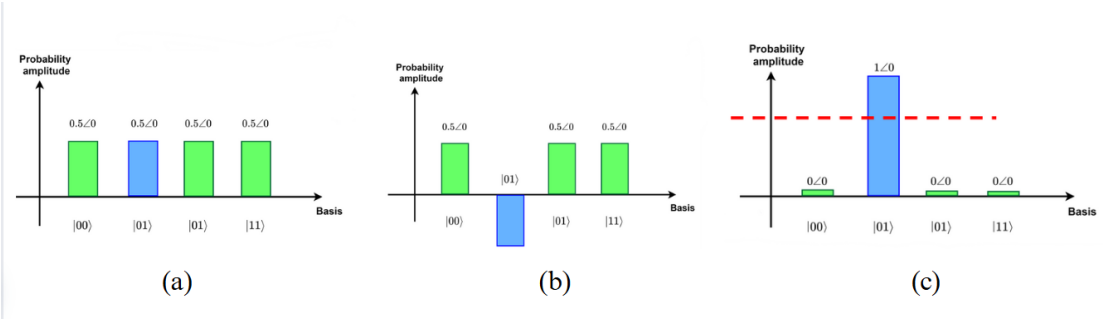


Figure 1.10: Inversion about the mean (example with 2 qubits) [2]. (a) States amplitude after $H^{\otimes n}$ operator; (b) States amplitude after the action of U_f; (c) State amplitudes after a Grover iteration $G = G_D U_f$.

The algorithm's power lies in repeating this iteration for an optimal number of iterations, approximately:

$$R \approx \frac{\pi}{4}\sqrt{\frac{N}{M}} - \frac{1}{2}, \tag{1.39}$$

where N is the number of elements in the dataset and M is the number of solutions. After R iterations, the amplitude of the marked state will be very close to 1, making the measurement of x_0 almost certain. Geometrically, each Grover iteration can be interpreted as a rotation within a two-dimensional subspace spanned by the marked state $|x_0\rangle$ and the uniform superposition of all unmarked states. Each subsequent Grover iteration increases the amplitude of $|x_0\rangle$ and decreases that of the non-solutions, systematically steering the system's state towards $|x_0\rangle$. This process beautifully illustrates quantum computing's capability to solve specific problems with remarkable efficiency compared to classical methods.

Variational Quantum Eigensolver (VQE)

The Variational Quantum Eigensolver (VQE) is a hybrid quantum-classical algorithm belonging to the family of Variational Quantum Algorithms (VQAs) with

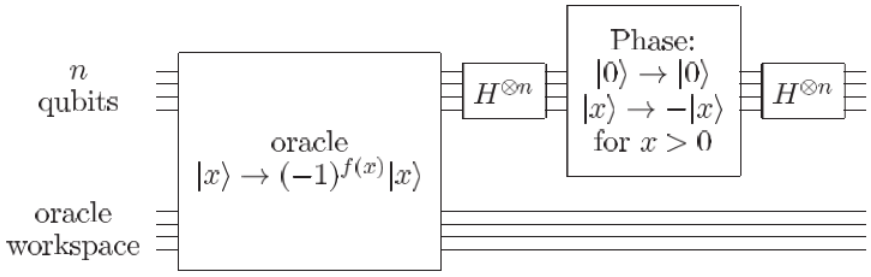


Figure 1.11: Schematic circuit of a single Grover iteration [1].

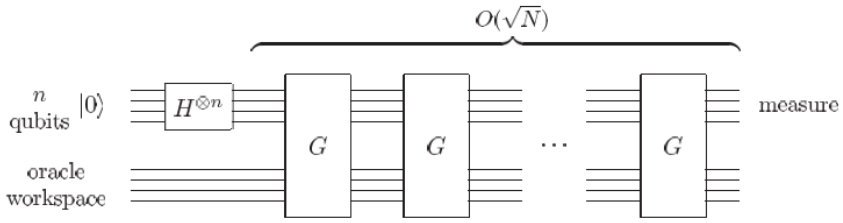


Figure 1.12: Schematic circuit of the Grover's search algorithm [1].

possible applications in optimization, quantum simulations and quantum chemistry. It was developed to approximate the ground state energy of quantum systems described by a Hamiltonian $\hat{H}$. The VQE algorithm is particularly well suited to Noisy Intermediate-Scale Quantum (NISQ) devices for several reasons. First, it requires quantum circuits of shallow depth, which minimizes the accumulation of noise and decoherence (two major limitations in current quantum hardware). Secondly, the hybrid structure of VQE means that the quantum computer is used only for state preparation and expectation value estimation, while the optimization loop is carried out on classical hardware. This division exploits the current strengths of both platforms: the quantum processor performs quantum mechanical operations that are exponentially costly on classical computers, while the classical processor handles optimization tasks that would be challenging on quantum hardware.

As described in a very detailed way in the work by Tilly et al. [5], the algorithm is based on the variational principle, which guarantees that the energy expectation value over any normalized trial wavefunction $|\psi(\boldsymbol{\theta})\rangle$, with $\boldsymbol{\theta}$ denoting a set of parameters taking values in $(-\pi, \pi]$, gives an upper bound to the ground state energy E_0 of $\hat{H}$:

$$E_0 \leq \frac{\langle\psi(\boldsymbol{\theta})|\hat{H}|\psi(\boldsymbol{\theta})\rangle}{\langle\psi(\boldsymbol{\theta})|\psi(\boldsymbol{\theta})\rangle}. \tag{1.40}$$

In practice, the trial wavefunction is generated by applying a parameterized

unitary operator $U(\boldsymbol{\theta})$ to a known initial state $|0\rangle^{\otimes N}$:

$$|\psi(\boldsymbol{\theta})\rangle = U(\boldsymbol{\theta})\,|0\rangle^{\otimes N}\,. \tag{1.41}$$

The core objective of the VQE is thus to solve the following optimization problem:

$$E_{\mathrm{VQE}} = \min_{\boldsymbol{\theta}} \langle 0|\, U^{\dagger}(\boldsymbol{\theta})\hat{H}U(\boldsymbol{\theta})\,|0\rangle\,. \tag{1.42}$$

The Hamiltonian $\hat{H}$ is typically written in second quantization as a linear combination of tensor products of Pauli operators (Pauli strings), after appropriate mapping (e.g., Jordan-Wigner or Bravyi-Kitaev transformation):

$$\hat{H} = \sum_{a=1}^{|\mathcal{P}|} w_a \hat{P}_a, \quad \hat{P}_a \in \{I, X, Y, Z\}^{\otimes N}, \quad w_a \in \mathbb{R}. \tag{1.43}$$

The expectation value is computed term by term:

$$E_{\mathrm{VQE}}(\boldsymbol{\theta}) = \sum_{a=1}^{|\mathcal{P}|} w_a \,\langle 0|\, U^{\dagger}(\boldsymbol{\theta})\hat{P}_a U(\boldsymbol{\theta})\,|0\rangle\,. \tag{1.44}$$

The classical optimizer seeks the parameter set $\boldsymbol{\theta}^*$ that minimizes $E_{\mathrm{VQE}}(\boldsymbol{\theta})$.

In order to accomplish the task of the optimization, the VQE algorithm relies on the construction of the *ansatz*, a parameterized quantum circuit that is used to construct a trial wavefunction $|\psi(\boldsymbol{\theta})\rangle$ for the quantum system under study. The expressivity of the ansatz determines how accurately it can approximate the true ground state of the Hamiltonian, while its structure influences the trainability and depth of the circuit. A well-designed ansatz should strike a balance between expressiveness (the ability to represent complex states) and feasibility (the ability to optimize it efficiently on noisy hardware).

For what concerns the classical optimizers, they fall into two major families, described in the following.

- **Gradient-based optimizers** are frequently used within the VQE framework to iteratively update the ansatz parameters $\boldsymbol{\theta}$ in the direction of steepest descent of the cost function $E(\boldsymbol{\theta})$. These methods require the evaluation of gradients with respect to the parameters. In quantum computing, gradients of the cost function with respect to variational parameters can often be computed analytically using the *parameter-shift rule*. Let $f(\boldsymbol{\theta})$ denote the expectation value of an observable measured on the output state of a parameterized quantum circuit. The function f depends on the vector of parameters $\boldsymbol{\theta} = (\theta_1, \theta_2, \dots)$, and can be regarded as a *quantum function*. In many common cases, particularly when the parameterized gates are generated by Hermitian operators with two distinct eigenvalues (such as Pauli operators), the partial derivative $\partial f/\partial\theta_k$ can be exactly expressed as a linear combination of values

of f evaluated at shifted parameter configurations. Specifically, the derivative with respect to θ_k can be written as:

$$\frac{\partial f}{\partial \theta_k} = \frac{1}{2}\left[f(\boldsymbol{\theta}_k^+) - f(\boldsymbol{\theta}_k^-)\right], \tag{1.45}$$

where $\boldsymbol{\theta}_k^\pm = \boldsymbol{\theta} \pm \frac{\pi}{2}\mathbf{e}_k$, and $\mathbf{e}_k$ is the unit vector in the k-th direction. Importantly, both $f(\boldsymbol{\theta}_k^+)$ and $f(\boldsymbol{\theta}_k^-)$ can be evaluated using the same quantum circuit structure as the original function, differing only by a deterministic shift in one parameter. This allows for efficient and exact gradient computation without requiring access to ancillae qubits or full tomography of the quantum state.

First-order optimizers, such as stochastic gradient descent (SGD), RMSProp, and Adam, are based solely on first-order derivative information. These methods employ adaptive learning rates and momentum-like terms to stabilize convergence in the presence of shot noise and hardware imperfections.

Second-order methods attempt to accelerate convergence by incorporating curvature information of the cost landscape. A prominent example is the *Broyden-Fletcher-Goldfarb-Shanno* (BFGS) algorithm, which approximates the inverse Hessian matrix to guide parameter updates. While BFGS offers faster convergence in convex regions of the cost function, its computational cost and memory requirements scale poorly with the number of parameters, which can become prohibitive for high-dimensional variational circuits. For this reason, limited-memory variants (such as L-BFGS) are sometimes preferred. The performance of gradient-based optimizers in VQE can be significantly affected by noise, barren plateaus, and optimization landscape ruggedness. As such, careful calibration with gradient-free methods are often explored in practical implementations.

- **Gradient-free optimizers** These include:
  - **COBYLA (Constrained Optimization By Linear Approximations):** builds a simplex around the current parameter vector and approximates the cost landscape locally.
  - **Nelder-Mead:** uses a heuristic search among a set of candidate solutions, iteratively reflecting, expanding, or contracting the simplex.
  - **Bayesian Optimization:** constructs a surrogate probabilistic model of the cost function and selects new samples based on expected improvement.
  - **POWELL:** proceeds by optimizing one parameter (i.e., one dimension of the search space) at a time. During each step, all parameters are fixed except for one, which is varied to minimize the objective function along that direction. This process is repeated cyclically for all dimensions. The algorithm employs a hill climbing strategy along each coordinate direction to iteratively refine the solution.

These optimizers are useful when gradients are too noisy to be useful or unavailable due to hardware limitations.

A major issue in VQE optimization is the barren plateau phenomenon, where gradients vanish exponentially with the number of qubits, making optimization intractable. The barren plateau phenomenon in VQE can be formally characterized by the following probabilistic bound: given a cost function

$$E(\boldsymbol{\theta}) = \langle \psi(\boldsymbol{\theta}) | \hat{H} | \psi(\boldsymbol{\theta}) \rangle ,$$

for any parameter $\theta_i \in \boldsymbol{\theta}$ and any $\epsilon > 0$, there exists a constant $b > 1$ such that:

$$\Pr\left(\left| \frac{\partial E(\boldsymbol{\theta})}{\partial \theta_i} \right| \geq \epsilon \right) \leq \mathcal{O}\left(\frac{1}{b^N} \right), \tag{1.46}$$

where N is the number of qubits. This inequality implies that the probability of observing a gradient with magnitude greater than an arbitrary threshold ϵ decays exponentially with system size. As a result, for sufficiently large N, gradients become exceedingly small with high probability, rendering gradient-based optimization practically infeasible without mitigation strategies.

The possible sources for barren plateaus are listed below:

- **Ansatz expressibility:** highly expressive circuits lead to uniform gradient landscapes.
- **Random initialization:** causes the parameters to fall into flat regions of the cost function.
- **Circuit depth:** deeper circuits increase the likelihood of gradient vanishing.
- **Quantum noise:** decoherence and gate imperfections can introduce noise-induced barren plateaus (NIBP).

For the sake of clarity, a description of the general pipeline of the VQE algorithm is provided (see Fig. 1.13).

- **Pre-processing:**
  - *Hamiltonian representation:* The system Hamiltonian is expressed as a quantum observable, using a predefined basis set that captures the electronic structure of the problem.
  - *Encoding:* The Hamiltonian is mapped into a form compatible with quantum hardware by converting fermionic operators into spin operators acting on qubits, using appropriate encoding techniques.
  - *Measurement grouping:* The resulting operators are organized into groups of commuting terms, enabling simultaneous measurement. This typically involves applying basis rotation gates within a group to allow simultaneous diagonalization.

  - *State initialization:* The initial quantum state is prepared as the reference state to which the variational ansatz will be applied.

- **VQE loop:**
  - *Ansatz application:* A parameterized quantum circuit is applied to the initial state to prepare a trial wavefunction. The parameters are initialized either randomly or using heuristics.
  - *Measurement:* The trial state is rotated into the appropriate basis and measured to obtain expectation values of the observables.
  - *Cost evaluation:* Expectation values are classically combined (typically via weighted summation) to compute the value of the cost function.
  - *Parameter update:* The cost function is minimized by updating the ansatz parameters through a classical optimization routine, initiating the next iteration of the loop.

- **Post-processing:**
  - *Error mitigation:* Techniques are applied to reduce the impact of quantum noise, either on the raw measurement results or directly on the quantum state before measurement.

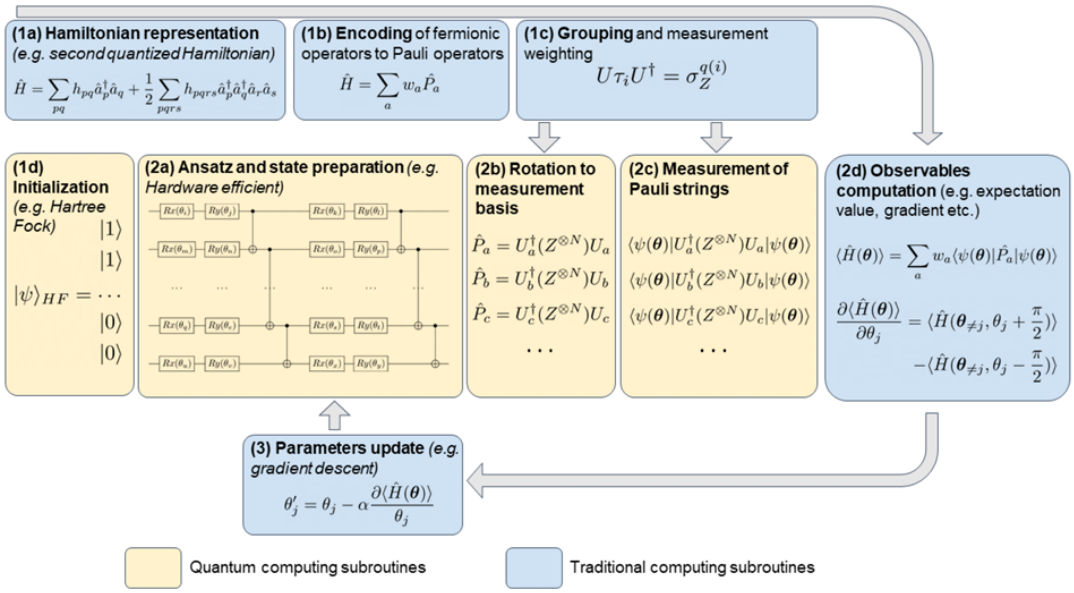


Figure 1.13: The VQE pipeline with an indication of the steps solved classically and quantum [5].

1.1.4 Quantum Fourier Transform

The Quantum Fourier Transform (QFT) is a linear, unitary transformation that plays a fundamental role in several quantum algorithms that outperform their classical counterparts. It can be seen as the quantum analogue of the classical Discrete Fourier Transform (DFT), and generalizes the Hadamard transformation to higher-dimensional systems. In particular, it is central to algorithms like Shor's factoring algorithm and those designed to solve periodicity problems.

The content in this section is based on the lecture notes by Professor Nilanjana Datta from the University of Cambridge [6].

Definition and Matrix Form

Let N be a positive integer, and consider a Hilbert space $\mathcal{H}_N$ with an orthonormal basis $\{|0\rangle, |1\rangle, \ldots, |N-1\rangle\}$ labeled by elements of $\mathbb{Z}_N$. The QFT modulo N, denoted QFT_N, is defined on the computational basis as follows:

$$\mathrm{QFT}_N |x\rangle = \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} \exp\left(\frac{2\pi i x y}{N}\right) |y\rangle , \tag{1.47}$$

for all $x \in \mathbb{Z}_N$. This transformation is unitary and maps each basis vector into an equally weighted superposition, where the amplitudes encode phase information that depends linearly on x and y.

The matrix representation of QFT_N is given by:

$$[\mathrm{QFT}_N]_{j,k} = \frac{1}{\sqrt{N}} \exp\left(\frac{2\pi i j k}{N}\right), \quad 0 \le j, k < N. \tag{1.48}$$

Letting $\omega = e^{2\pi i/N}$ be the N-th primitive root of unity, each matrix element becomes a power of ω, and the matrix is symmetric and unitary.

Unitarity Proof. To show that QFT_N is unitary, consider the product $\mathrm{QFT}_N^\dagger \mathrm{QFT}_N$. The inner product of the j-th and k-th rows yields:

$$\left(\mathrm{QFT}_N^\dagger \mathrm{QFT}_N\right)_{jk} = \frac{1}{N} \sum_{\ell=0}^{N-1} \exp\left(\frac{2\pi i \ell (k-j)}{N}\right). \tag{1.49}$$

This is a finite geometric series that sums to N when $j = k$, and 0 otherwise, due to the properties of roots of unity. Hence, $\mathrm{QFT}_N^\dagger \mathrm{QFT}_N = I_N$, confirming unitarity.

Efficient Implementation for Powers of Two

When $N = 2^n$, the QFT can be efficiently implemented as a quantum circuit acting on n qubits. The transformation can be rewritten as:

$$\mathrm{QFT}_{2^n} |x\rangle = \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} \exp\left(\frac{2\pi i x y}{2^n}\right) |y\rangle . \tag{1.50}$$

Let $x = x_{n-1}x_{n-2}\ldots x_0$ be the binary expansion of x. It turns out that the output state can be expressed as a tensor product of n single-qubit states with phase rotations depending on binary fractions:

$$\mathrm{QFT}_{2^n} |x\rangle = \bigotimes_{j=0}^{n-1} \frac{1}{\sqrt{2}} \left(|0\rangle + e^{2\pi i \cdot 0.x_j x_{j-1}\ldots x_0} |1\rangle\right). \tag{1.51}$$

This factorisation allows an efficient circuit construction using:

- n Hadamard gates (one per qubit);
- $\frac{n(n-1)}{2}$ controlled-R_k phase shift gates, where each R_k acts as:

$$R_k = \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{pmatrix}. \tag{1.52}$$

- $n/2$ SWAP gates to reverse the qubit order.

The total gate count scales as $\mathcal{O}(n^2)$, making the QFT one of the few non-trivial quantum operations implementable in polynomial time.

In Fig. 1.14, the circuit for $n = 3$ is depicted as an example.

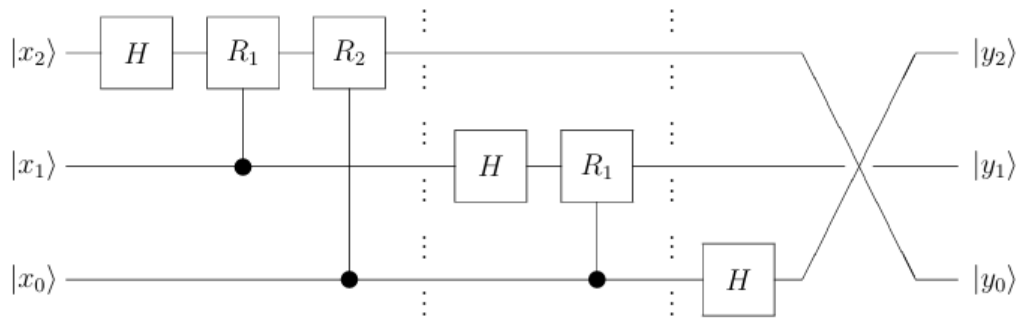


Figure 1.14: Circuit scheme of the Quantum Fourier Transform for $n = 3$ [6].

Chapter 2

Quantum Error Correction

2.1 Introduction and Motivation

This chapter draws conceptual inspiration from the doctoral thesis of Daniel Gottesman [7], which represents the foundation for the theory of stabilizer codes and modern quantum error correction.

In practice, quantum systems are never perfectly isolated. When qubits are stored in memory, transmitted through communication channels, or manipulated via quantum gates, they inevitably interact with their surrounding environment. This interaction introduces noise and alters the system's evolution, typically degrading the coherence and integrity of the quantum information. An initially pure state $|\psi\rangle$ may, as a consequence of entanglement with the environment, evolve into a mixed state described by a density matrix ρ. Such noise can arise from various sources, including thermal fluctuations, electromagnetic interference, or imperfections in hardware components. Even in the absence of active operations, idle qubits are subject to decoherence over time.

To model this noisy evolution mathematically, one uses the formalism of quantum channels. A quantum channel is represented by a linear map $\mathcal{E}$ acting on density matrices. To ensure that this map produces valid quantum states as output, it must satisfy two essential properties:

- **Complete positivity (CP):** A map $\mathcal{E}$ is completely positive if, when extended to any larger system by tensoring with the identity map on an ancillary space, it still maps positive operators to positive operators. This condition guarantees that even if the system of interest is entangled with an external reference system, the evolution remains physically valid.

- **Trace preservation (TP):** The trace of the density matrix is preserved under $\mathcal{E}$: $\mathrm{Tr}(\mathcal{E}(\rho)) = \mathrm{Tr}(\rho) = 1$. This condition ensures that the total probability is conserved, i.e., no part of the quantum state is "lost" or "created" during the process.

A map satisfying both properties is called a *completely positive trace-preserving* (CPTP) map. CPTP maps form the most general class of transformations that can describe physical evolutions of quantum states, including noise, decoherence, and imperfect operations.

Any CPTP map admits a representation known as the operator-sum or Kraus decomposition:

$$\mathcal{E}(\rho) = \sum_i E_i \rho E_i^\dagger, \tag{2.1}$$

where the operators E_i are called *Kraus operators*, and satisfy the completeness condition:

$$\sum_i E_i^\dagger E_i = I. \tag{2.2}$$

This condition is what ensures that the map $\mathcal{E}$ is trace-preserving. The Kraus operators describe different possible ways the quantum system may evolve due to interaction with the environment, each associated with a certain probability. These operators are not required to be unitary; they may represent irreversible processes such as amplitude damping or phase noise.

From the perspective of quantum error correction, it is sufficient to construct a code that is capable of correcting the effect of each individual Kraus operator. If this condition is met, the code can correct the entire action of the noisy channel $\mathcal{E}$, regardless of whether the output state is pure or mixed. In particular, a mixed state can be understood as a statistical ensemble of pure states, and if all pure components of the ensemble can be individually corrected, then the full state is effectively recovered.

For simplicity, the following analysis will focus on pure input states affected by arbitrary (possibly non-unitary) error operators. This abstraction provides a general and powerful foundation for the development of quantum error-correcting codes, without loss of generality regarding the physical noise processes involved.

2.2 Classical Error Correction Fundamentals

Before delving into the core of quantum error correction, it is beneficial to establish a solid algebraic foundation by formalizing key definitions and concepts from classical error correction. This section follows the framework presented in *Protecting Information: From Classical Error Correction to Quantum Cryptography* by Loepp and Wootters [8].

2.2.1 Linear Block Codes

Let A be a finite set and $n \geq 1$ an integer. A code C of length n is any subset of A^n. In this setting, A^n is called the *codespace* and the elements of C are called *codewords*.

Moreover, C is a *linear code* if

- C is not empty
- $\forall \vec{c}_1, \vec{c}_2 \in C$, $\vec{c}_1 + \vec{c}_2 \in C$ (closed under addition)
- $\forall \vec{c} \in C, \forall \alpha \in \mathbb{F}_2$, $\alpha \vec{c} \in C$ (closed under scalar multiplication)

Generator Matrix

A linear code C can be uniquely specified by a $k \times n$ matrix G, known as the *generator matrix*, whose k rows form a basis for C. The generator matrix is typically presented in a systematic form, $G = [I_k|P]$, where I_k is the $k \times k$ identity matrix and P is a $k \times (n-k)$ matrix.

Weight and Distance

The *Hamming weight* of a vector $\vec{v} \in \mathbb{F}_2^n$, denoted $w(\vec{v})$, is defined as the number of non-zero components in $\vec{v}$. This is equivalent to the number of 1s in the vector.

The *Hamming distance* between two vectors $\vec{u}, \vec{v} \in \mathbb{F}_2^n$, denoted $d(\vec{u}, \vec{v})$, is the number of positions in which they differ. The *minimum distance* of a linear code C, denoted $d(C)$ or simply d, is a critical parameter quantifying its error-correcting capability. It is defined as the smallest Hamming distance between any two distinct codewords:

$$d = \min\{d(\vec{u}, \vec{v}) \mid \vec{u}, \vec{v} \in C, \vec{u} \neq \vec{v}\} \tag{2.3}$$

For a linear code, this simplifies significantly: the minimum distance is equal to the minimum weight of any non-zero codeword:

$$d = \min\{w(\vec{c}) \mid \vec{c} \in C, \vec{c} \neq \vec{0}\} \tag{2.4}$$

A linear code with minimum distance d is capable of detecting up to $d-1$ errors and correcting up to $t = \lfloor (d-1)/2 \rfloor$ errors. This fundamental relationship is often

summarized by stating that a code is an $[n, k, d]$ code. k represents the *dimension* of C.

Parity-Check Matrix

A linear code C can also be defined as the set of all vectors $\vec{x} \in \mathbb{F}_2^n$ that are orthogonal to the rows of a specific $(n-k) \times n$ matrix H. This matrix H is called the *parity-check matrix* of C. Formally, a vector $\vec{x} \in \mathbb{F}_2^n$ is a codeword not affected by errors if and only if:

$$H\vec{x}^T = \vec{0}^T \tag{2.5}$$

where $\vec{0}$ is the zero vector of length $n-k$. If the generator matrix G is in systematic form $G = [I_k|P]$, then the corresponding parity-check matrix H can be constructed as $H = [-P^T|I_{n-k}]$.

Dual Code

The *dual code* $C^\perp$ of a code C is defined as the set of all vectors $\vec{v} \in \mathbb{F}_2^n$ that are orthogonal to every codeword in C under the standard dot product (modulo 2):

$$C^\perp = \{\vec{v} \in \mathbb{F}_2^n \mid \vec{v} \cdot \vec{c} = 0 \text{ for all } \vec{c} \in C\} \tag{2.6}$$

If C is an $[n, k]$ linear code, then its dual $C^\perp$ is an $[n, n-k]$ linear code. Crucially, if G is a generator matrix for C, then G is a parity-check matrix for $C^\perp$. Conversely, if H is a parity-check matrix for C, then H is a generator matrix for $C^\perp$.

A code C is said to be *self-orthogonal* if $C \subseteq C^\perp$. If, in addition, $C = C^\perp$, the code is called *self-dual*. For a self-dual code, it must hold that $k = n-k$, implying n must be an even number and $k = n/2$.

Syndrome Decoding

Upon receiving a potentially erroneous vector $\vec{y} \in \mathbb{F}_2^n$, the decoding process begins by computing the *syndrome* $\vec{s} = H\vec{y}^T$. If $\vec{s} = \vec{0}^T$, then $\vec{y}$ is a codeword not affected by errors. If $\vec{s} \neq \vec{0}^T$, an error has occurred. The syndrome uniquely corresponds to a correctable error pattern. If $\vec{e}$ is the error vector such that $\vec{y} = \vec{c} + \vec{e}$ for some codeword $\vec{c}$, then $\vec{s} = H(\vec{c} + \vec{e})^T = H\vec{c}^T + H\vec{e}^T = \vec{0}^T + H\vec{e}^T = H\vec{e}^T$. Thus, the syndrome reveals information about the error vector $\vec{e}$. Syndrome decoding involves pre-calculating the syndrome for all possible correctable error patterns (those with weight up to t) and then, upon computing a syndrome $\vec{s}$, identifying the most likely error pattern $\vec{e}$ that produces this $\vec{s}$ (i.e., $H\vec{e}^T = \vec{s}$ and $w(\vec{e})$ is minimal). The corrected codeword is then $\vec{c} = \vec{y} - \vec{e}$.

Cosets and Coset Leaders

For a linear code $C \subseteq \mathbb{F}_2^n$, a *coset* of C with respect to a vector $\vec{x} \in \mathbb{F}_2^n$ is the set:

$$C + \vec{x} = \{\vec{c} + \vec{x} \mid \vec{c} \in C\} \tag{2.7}$$

Each coset partitions the entire vector space $\mathbb{F}_2^n$ into disjoint subsets. Importantly, two vectors $\vec{x}_1, \vec{x}_2 \in \mathbb{F}_2^n$ belong to the same coset if and only if their difference $\vec{x}_1 - \vec{x}_2 \in C$, which implies $H(\vec{x}_1 - \vec{x}_2)^T = \vec{0}^T$, or $H\vec{x}_1^T = H\vec{x}_2^T$. This means that all vectors within the same coset share the identical syndrome.

A *coset leader* is the vector of minimum Hamming weight within a given coset. If there are multiple vectors with the same minimum weight in a coset, any one of them can be chosen as the coset leader. If a received vector $\vec{y}$ has syndrome $\vec{s}$, one can find the coset leader $\vec{e}$ associated with that syndrome. The decoded codeword is then $\vec{c} = \vec{y} - \vec{e}$.

Example: The Hamming Code A well-known example of a linear code is the binary *Hamming code* $\mathrm{Ham}(r, 2)$, where $r \geq 2$. It is an $[n, k, d]$ code with parameters $n = 2^r - 1$, $k = 2^r - 1 - r$, and $d = 3$. This means a Hamming code can correct any single bit error. Its parity-check matrix H is constructed by taking as its columns all distinct non-zero vectors in $\mathbb{F}_2^r$. For instance, for $r = 3$, the Hamming code is a [7,4,3] code. Its parity-check matrix H has dimensions $(n-k) \times n = 3 \times 7$, and its columns are all non-zero binary vectors of length 3:

$$H = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Suppose a vector $\vec{y} = (1011100)$ is received. To determine if $\vec{y}$ is a valid codeword or if it has been affected by an error, its syndrome is computed:

$$\vec{s} = H\vec{y}^T = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Since the computed syndrome $\vec{s} = (100)^T$ is not the zero vector, the received

vector $\vec{y}$ is not a valid codeword. This indicates that a detectable error occurred during transmission for this particular received vector.

By computing the coset leader corresponding to the syndrome $\vec{s} = (100)^T$, one knows that an error in the 5th bit has occured. So, the correct codeword is $\vec{c} = (1011000)$.

In this example, every coset possesses a unique coset leader. This is not universally the case: if a coset contains more than one coset leader, unique identification of the error pattern becomes impossible. In such scenarios, accurate correction of errors by a linear code designed to correct t errors cannot be guaranteed. Instead, the code may only be able to detect that the number of errors exceeds its correction capability.

2.3 General Quantum Error-Correcting Codes

A quantum code that encodes k logical qubits into n physical qubits defines a subspace of the 2^n-dimensional Hilbert space. This subspace, denoted $\mathcal{T}$, consists of 2^k basis codewords that correspond to the computational basis states of the logical qubits. Any linear combination of these basis codewords remains a valid codeword, preserving the linear structure of quantum states.

To determine whether a code can correct a given set of errors, it is sufficient to consider a basis for the space of all possible errors. A common choice is the group $\mathcal{G}_n$ of n-fold tensor products of the single-qubit operators $\{I, \sigma_x, \sigma_y, \sigma_z\}$, including possible global phases such as ± 1 and $\pm i$. The *weight* of an operator is defined as the number of qubits on which it acts non-trivially (i.e., not as the identity).

For a code to distinguish between two correctable errors E_a and E_b acting on potentially different codewords $|\psi_i\rangle$ and $|\psi_j\rangle$, it is required that

$$\langle \psi_i | E_a^\dagger E_b | \psi_j \rangle = 0 \quad \text{for } i \neq j. \tag{2.8}$$

This condition ensures that the errors do not cause confusion between distinct logical states.

Furthermore, in order to preserve superpositions and avoid acquiring information about the encoded state during error detection, it is also necessary that

$$\langle \psi_i | E_a^\dagger E_b | \psi_i \rangle = \langle \psi_j | E_a^\dagger E_b | \psi_j \rangle \quad \forall i, j. \tag{2.9}$$

Both conditions can be combined into the more general requirement:

$$\langle \psi_i | E_a^\dagger E_b | \psi_j \rangle = C_{ab} \delta_{ij}, \tag{2.10}$$

where C_{ab} is a Hermitian matrix independent of the codeword indices i and j.

The condition in Eq. (2.10) is both necessary and sufficient for a code to correct the set of errors $\{E_a\}$. By diagonalizing the Hermitian matrix C_{ab}, one can obtain

an orthonormal error basis $\{F_a\}$ such that

$$\langle\psi_i|F_a^\dagger F_b|\psi_j\rangle = \delta_{ab}\delta_{ij} \quad \text{or} \quad 0, \tag{2.11}$$

depending on the nature of the errors. Errors of the second type may annihilate the code space, making their occurrence detectable and ignorable. The other type of errors always results in orthogonal states, allowing the identification of the specific error through an appropriate measurement, from which the location and nature of the disturbance within the code can be precisely detected.

A code for which the matrix C_{ab} is not full-rank is called *degenerate*, while it is *non-degenerate* if C_{ab} has full rank.

2.3.1 Code Distance and Error Correction Capability

The weight of the smallest $E = E_a^\dagger E_b \in \mathcal{G}_n$ that violates Eq. (2.10) represents the distance d of the code. A code capable of correcting up to t arbitrary errors must satisfy $d \geq 2t + 1$. A quantum code encoding k qubits into n physical qubits with distance d is denoted $[n, k, d]$. In literature, the notation $[[n, k, d]]$ is sometimes used to avoid confusion with classical codes.

Quantum error correction can also be adapted to different error models:

- To detect (but not correct) up to s errors, a code must have distance at least $s + 1$.
- If the positions of up to r errors are known (e.g., in a quantum erasure channel), the code needs a minimum distance of $r + 1$ to correct them.
- A code that corrects t arbitrary errors, r known-location errors, and detects s additional errors must have $d \geq 2t + r + s + 1$.

It is assumed that errors occur independently across different qubits and that single-qubit errors are uniformly distributed among the Pauli operators σ_x, σ_y, and σ_z. Under the assumption of a small error probability ϵ per qubit, the probability of more than t errors is $O(\epsilon^{t+1})$ and can typically be neglected.

Some systems experience *leakage errors*, which cause the system to leave the computational subspace. Examples include atomic transitions to unintended energy levels or photon loss. These errors can be identified by measurements distinguishing computational from non-computational states. Once detected, such errors may be reinterpreted as located errors and corrected accordingly.

Correlated errors acting on multiple qubits simultaneously present another challenge. Nevertheless, if the probability of such correlated errors decays exponentially with their weight, they can be treated within the same framework.

In practice, error models often deviate from the uniform assumption. For example, in ion-trap qubits, spontaneous emission tends to generate specific error types more frequently. Such amplitude damping channels can produce errors like

$\sigma_x + i\sigma_y$ with probability ϵ, while other errors, such as $I - \sigma_z$, occur with probability $O(\epsilon^2)$. These asymmetries suggest that tailoring error-correcting codes to realistic noise models can improve efficiency. Hence, a careful characterization of the physical error processes is essential for the design of practical quantum error correction schemes.

``latex

2.4 The Stabilizer Codes

Stabilizer codes constitute a broad class of quantum codes that are naturally described in group-theoretical terms. The central idea is to define the codespace $\mathcal{T} \subseteq \mathbb{C}^{2^n}$ as the common $+1$ eigenspace of a subgroup of the n-qubit Pauli group $\mathcal{G}_n$. The operators belonging to this subgroup are called *stabilizers* because they *stabilize* the codespace: every state $|\psi\rangle \in \mathcal{T}$ remains invariant under their action,

$$S|\psi\rangle = |\psi\rangle \qquad \forall\, S \in \mathcal{S}, \tag{2.12}$$

where $\mathcal{S} \subset \mathcal{G}_n$ denotes the stabilizer group.

Each element of the Pauli group $\mathcal{G}_n$ is unitary and either Hermitian or anti-Hermitian. Any two elements $A, B \in \mathcal{G}_n$ either commute, $[A, B] = 0$, or anticommute, $\{A, B\} = 0$. A stabilizer code is defined by a subgroup $\mathcal{S} \subset \mathcal{G}_n$ (the stabilizer group) that satisfies the following conditions:

- $\mathcal{S}$ is Abelian, ensuring that all its elements commute pairwise and admit a common set of eigenvectors.
- $\mathcal{S}$ does not contain the elements $-I$, iI, $-iI$.

Given an $[[n, k]]$ stabilizer code, the codespace $\mathcal{T}$ has dimension 2^k, and the stabilizer group $\mathcal{S}$ contains 2^{n-k} elements. Although stabilizer codes are often used as quantum error-correcting codes, this is not always the case. In particular, codes of distance $d = 2$ cannot correct arbitrary single-qubit errors, yet they are still described within the stabilizer formalism.

It is useful to recall some algebraic properties of Pauli operators. For single-qubit Pauli matrices $\sigma_x, \sigma_y, \sigma_z$ one has $\sigma_x^2 = \sigma_y^2 = \sigma_z^2 = I$, so every element of $\mathcal{G}_n$ squares to either $+I$ or $-I$. Pauli matrices acting on the same qubit anticommute, while those acting on different qubits commute. Since σ_y has imaginary matrix elements, whereas σ_x and σ_z are real, the parity of the number of σ_y factors in a stabilizer element determines whether a simultaneous eigenbasis can be chosen with real coefficients. Nevertheless, it has been proven that for any stabilizer code defined over the complex numbers there exists a real representation with identical parameters, so one may often restrict attention to real codes without loss of generality.

For the sake of clarity, an explicit example is provided for the two-qubit Bell state $|\Phi^+\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$. This state is a simultaneous $+1$ eigenstate of the commuting

operators $X \otimes X$ and $Z \otimes Z$. Indeed,

$$\begin{aligned}(X \otimes X)|\Phi^+\rangle &= \tfrac{1}{\sqrt{2}}\big(X|0\rangle \otimes X|0\rangle + X|1\rangle \otimes X|1\rangle\big) \\ &= \tfrac{1}{\sqrt{2}}\big(|1\rangle \otimes |1\rangle + |0\rangle \otimes |0\rangle\big) \\ &= |\Phi^+\rangle,\end{aligned} \tag{2.13}$$

and

$$\begin{aligned}(Z \otimes Z)|\Phi^+\rangle &= \tfrac{1}{\sqrt{2}}\big(Z|0\rangle \otimes Z|0\rangle + Z|1\rangle \otimes Z|1\rangle\big) \\ &= \tfrac{1}{\sqrt{2}}\big(|0\rangle \otimes |0\rangle + (-1)(-1)|1\rangle \otimes |1\rangle\big) \\ &= |\Phi^+\rangle.\end{aligned} \tag{2.14}$$

Since $[X \otimes X,\ Z \otimes Z] = 0$, their common $+1$ eigenspace is one-dimensional and spanned precisely by $|\Phi^+\rangle$.

Furthermore, using the identity $Y = iXZ$, one finds

$$Y \otimes Y = (iXZ) \otimes (iXZ) = -(X \otimes X)(Z \otimes Z), \tag{2.15}$$

so that

$$(Y \otimes Y)|\Phi^+\rangle = -(X \otimes X)(Z \otimes Z)|\Phi^+\rangle = -|\Phi^+\rangle. \tag{2.16}$$

Therefore, the stabilizer group of $|\Phi^+\rangle$ is

$$\mathcal{S} = \langle X \otimes X,\ Z \otimes Z\rangle = \{\, I,\ X \otimes X,\ Z \otimes Z,\ -Y \otimes Y \,\}. \tag{2.17}$$

The group contains $2^{2-0} = 4$ elements, corresponding to a one-dimensional codespace ($k = 0$).

2.4.1 Error Detection and Correction

An error $E \in \mathcal{G}_n$ is detectable by the code if it anticommutes with at least one generator of $\mathcal{S}$. In that case, the error flips the sign of at least one stabilizer measurement, which signals that an error has occurred. More precisely, for any state $|\psi\rangle \in \mathcal{T}$ and any stabilizer generator $M \in \mathcal{S}$, if $\{E, M\} = 0$, then:

$$ME|\psi\rangle = -E|\psi\rangle,$$

which implies that $E|\psi\rangle$ lies outside of $\mathcal{T}$ and can therefore be detected.

In addition, some errors commute with all elements of the stabilizer but are not themselves in the stabilizer. These errors do not take states out of $\mathcal{T}$, but they can transform one logical state into another. The set of all such operators forms the *centralizers* $C(\mathcal{S})$ of $\mathcal{S}$ in $\mathcal{G}_n$. Due to the algebraic properties of the stabilizer group $\mathcal{S}$ and the Pauli group $\mathcal{G}_n$, the centralizer of $\mathcal{S}$ in $\mathcal{G}_n$ coincides with the *normalizer* $\mathcal{N}(\mathcal{S})$ of $\mathcal{S}$ in $\mathcal{G}_n$.

The quotient group $\mathcal{N}(\mathcal{S})/\mathcal{S}$ describes how these commuting errors act non-trivially within the codespace $\mathcal{T}$. Logical operations on the encoded qubits are represented by elements of this quotient group. For a code encoding k logical qubits, a standard choice is to define logical Pauli operators $\overline{X}_1, \ldots, \overline{X}_k$ and $\overline{Z}_1, \ldots, \overline{Z}_k$, which satisfy the same algebraic relations as their physical counterparts:

$$[\overline{X}_i, \overline{X}_j] = 0, \tag{2.18}$$

$$[\overline{Z}_i, \overline{Z}_j] = 0, \tag{2.19}$$

$$[\overline{X}_i, \overline{Z}_j] = 0 \quad \text{for } i \neq j, \tag{2.20}$$

$$\{\overline{X}_i, \overline{Z}_i\} = 0. \tag{2.21}$$

2.4.2 Error Syndromes

To determine which error has occurred, one measures the eigenvalues of each generator of $\mathcal{S}$. This yields a binary vector known as the *syndrome*, which depends on the commutation properties between the error and the generators. For each stabilizer generator M_i, one can define:

$$f_{M_i}(E) = \begin{cases} 0 & \text{if } [M_i, E] = 0, \\ 1 & \text{if } \{M_i, E\} = 0. \end{cases}$$

The full syndrome is the vector $f(E) = (f_{M_1}(E), \ldots, f_{M_{n-k}}(E))$. For nondegenerate codes, each correctable error yields a distinct syndrome, allowing the error to be identified and corrected. In degenerate codes, different errors may produce the same effect on all codewords, making them indistinguishable but functionally equivalent.

The error operator can always be assumed to lie within the Pauli group $\mathcal{G}_n$, since the code is defined with respect to this error basis. All elements of $\mathcal{G}_n$ are unitary, and therefore invertible, which ensures that any correctable error can be reversed.

Once the error is identified (up to equivalence under multiplication by elements of the stabilizer group $\mathcal{S}$), an appropriate inverse operator can be applied to restore the state to the codespace. Even if the original error is a non-trivial linear combination of Pauli operators, the act of measuring the syndrome will project the system onto a specific error component within the Pauli basis.

2.4.3 Encoding and Decoding of Stabilizer Codes

To use a stabilizer code in practice, it is necessary to specify how logical qubits are encoded into physical qubits. This involves identifying a set of logical operators and constructing an encoding circuit that prepares codewords from standard basis inputs.

Let $\mathcal{S}$ be the stabilizer group for an $[[n, k]]$ stabilizer code. The codespace $\mathcal{T}$ is the subspace of $\mathbb{C}^{2^k}$ consisting of all quantum states that are stabilized by every

element of $\mathcal{S}$, that is,

$$\mathcal{T} = \{|\psi\rangle \in \mathbb{C}^{2^k} \mid M|\psi\rangle = |\psi\rangle, \quad \forall M \in \mathcal{S}\}.$$

To define the logical structure of the code, one must select a set of $2k$ Pauli operators $\overline{X}_1, \ldots, \overline{X}_k, \overline{Z}_1, \ldots, \overline{Z}_k$ that satisfy the usual Pauli commutation relations. These operators act on the codespace in the same way that X_i and Z_i act on unencoded qubits, allowing quantum gates and algorithms to be carried out directly on the encoded information.

Let us assume that the code is specified in the binary symplectic representation. The Pauli group on n qubits, denoted $\mathcal{G}_n$, has a natural binary representation via the map:

$$X^{\mathbf{u}} Z^{\mathbf{v}} \mapsto (\mathbf{u}|\mathbf{v}) \in \mathbb{F}_2^{2n},$$

where $\mathbf{u}, \mathbf{v} \in \mathbb{F}_2^n$ are binary vectors indicating the positions where X and Z act, respectively.

To characterize commutation relations in this representation, one introduces a bilinear form known as the *symplectic product*. Given two elements $(\mathbf{u}_1|\mathbf{v}_1), (\mathbf{u}_2|\mathbf{v}_2) \in \mathbb{F}_2^{2n}$, their symplectic product is defined as:

$$[[(\mathbf{u}_1|\mathbf{v}_1), (\mathbf{u}_2|\mathbf{v}_2)]] := \mathbf{u}_1 \cdot \mathbf{v}_2 + \mathbf{v}_1 \cdot \mathbf{u}_2 \pmod 2. \tag{2.22}$$

This operation determines whether two Pauli operators commute or anticommute:

- If the symplectic product is 0, the corresponding Pauli operators commute.
- If the symplectic product is 1, the operators anticommute.

A set of vectors in $\mathbb{F}_2^{2n}$ is called a *symplectic subspace* if it is closed under addition and its elements satisfy specific symplectic orthogonality constraints. Stabilizer codes rely on the fact that the generators of the stabilizer group must pairwise commute, and this condition is equivalent to requiring their binary representations to be symplectically orthogonal. Each Pauli operator on n qubits can be represented by a binary vector $(\mathbf{u}|\mathbf{v}) \in \mathbb{F}_2^{2n}$, where $\mathbf{u}$ indicates the positions of X operators and $\mathbf{v}$ the positions of Z operators. The stabilizer is then generated by $n-k$ linearly independent vectors $(\mathbf{u}_i|\mathbf{v}_i)$, corresponding to the generators $M_i = i^{\lambda_i} X^{\mathbf{u}_i} Z^{\mathbf{v}_i}$, where $\lambda_i \in \{0,1,2,3\}$ determines the global phase.

To simplify the encoding process, it is useful to bring the stabilizer matrix into a standard form. This is done by applying row operations and permuting qubit labels so that:

$$M = \begin{pmatrix} I_r & A_1 & B & C_1 \\ 0 & A_2 & D & C_2 \end{pmatrix},$$

where I_r is an identity block, and the remaining blocks are arbitrary binary matrices. Here, r is the rank of the X part of the stabilizer matrix.

Logical operators $\overline{X}_i$ and $\overline{Z}_i$ can be chosen in such a way that they commute with all stabilizer generators and with each other as required. For each logical qubit i, the operator $\overline{X}_i$ can be chosen to act as an X on the $(n-k+i)$-th physical qubit, together with other operations that ensure commutation with $\mathcal{S}$. A similar construction is used for $\overline{Z}_i$.

Once the stabilizer and logical operators are in standard form, an encoding circuit can be constructed.

To implement the encoding procedure of a general stabilizer code, all n qubits are first initialized in the computational basis state $|0\rangle^{\otimes n}$. The goal is to apply a unitary operation that transforms this product state into a valid codeword of the stabilizer code:

$$|\overline{c_1 \cdots c_k}\rangle \to \overline{X}_1^{c_1} \cdots \overline{X}_k^{c_k} \sum_{M \in \mathcal{S}} M \, |0\rangle^{\otimes n} \, . \tag{2.23}$$

Let r be the number of stabilizer generators that contain Pauli X-type terms. The encoding process can be organized into the following steps:

- **Eliminating trivial Z-type actions from logical operators.** Logical X-type operators, when brought into standard form, act as Pauli Z operators on the first r qubits and as Pauli X operators on the remaining $n-k-r$ qubits. Since $Z\,|0\rangle = |0\rangle$, the action of Z on $|0\rangle$ is trivial and can be ignored in the encoder. The non-trivial contributions come from the X-type parts, which are implemented using CNOT gates. In general, this step is necessary if $r < n-k$.

- **Creating superpositions with Hadamard gates and applying controlled M_i operators.** The first r stabilizer generators include X-terms, which require superpositions to be created. This is done by applying Hadamard gates to the first r qubits, turning them into $|+\rangle$ states. Then, each stabilizer generator M_i is applied conditionally, controlled on the i-th qubit. If M_i includes only phase-type operators (such as Z or Y) on the target qubits, these can be applied without interfering with other operations. If M_i includes a Z on the control qubit, it introduces only a phase flip, which can be applied after the Hadamard gate directly.

- **Omitting purely Z-type stabilizers.** When $r < n-k$, there exist Z-only stabilizers that leave the initial state unchanged. For any such generator M, acting on $|0\rangle^{\otimes n}$ produces no effect, because each Z leaves $|0\rangle$ invariant. These stabilizers also commute with other generators and logical operators, so their effect is encoded indirectly. Therefore, they can be omitted from the encoder.

- **Resource estimate.** The implementation of logical X-operators (ignoring trivial Z-parts) may involve up to $n-k-r$ CNOT gates. The preparation of r qubits in the $|+\rangle$ state requires r Hadamard gates, possibly followed by additional single-qubit Z gates for sign corrections. Each of the r stabilizer

generators may act on up to $n-1$ other qubits, contributing up to $r(n-1)$ two-qubit gates. Thus, the total number of two-qubit gates is bounded by:

$$k(n-k-r)+r(n-1) \leq (k+r)(n-k) \leq n(n-k). \tag{2.24}$$

This provides a meaningful upper bound for the encoding circuit complexity, based on the code parameters (n,k,r).

Decoding consists of reversing the encoding procedure, either to extract the logical information or as a preliminary step in measurement-based algorithms.

To correct errors, the stabilizer generators are measured, producing a syndrome vector $s \in \mathbb{F}_2^{n-k}$ that identifies the equivalence class of the error. A recovery operator R_s is then applied such that for any $|\psi\rangle \in \mathcal{T}$ and any error E consistent with the syndrome s, one has:

$$R_s E |\psi\rangle \in \mathcal{T}. \tag{2.25}$$

In a non-degenerate code, the syndrome uniquely identifies the error up to stabilizer equivalence. In a degenerate code, different errors may correspond to the same syndrome but have the same action on the codespace, making explicit distinction unnecessary.

The recovery must preserve logical information, meaning that for any two logical states $|\psi_i\rangle, |\psi_j\rangle \in \mathcal{T}$, the following condition holds:

$$\langle\psi_i| R_s E |\psi_j\rangle = \delta_{ij}. \tag{2.26}$$

This guarantees that the corrected state retains the correct logical content and that the error correction procedure introduces no additional disturbance.

Together, the encoding and decoding mechanisms define the operational framework through which stabilizer codes are applied to protect quantum information from noise and decoherence.

An important connection between classical and quantum coding theory is the quantum Hamming bound, which can be seen as the quantum analogue of the classical sphere-packing bound.[1]

Consider a quantum stabilizer code with parameters $[\![n,k,d]\!]$. Since each Pauli error acts non-trivially on i qubits with 3^i possibilities (excluding identity on

[1] In classical coding theory, a binary code with parameters $[n,k,d]$ maps 2^k messages to n-bit codewords. The code can correct up to $t = \left\lfloor \frac{d-1}{2} \right\rfloor$ errors if Hamming balls of radius t centered at each codeword are disjoint. The number of distinct error patterns within such a ball is $\sum_{i=0}^{t} \binom{n}{i}$, so the sphere-packing bound requires that

$$2^k \sum_{i=0}^{t} \binom{n}{i} \leq 2^n. \tag{2.27}$$

remaining qubits), the number of errors up to weight t is:

$$\sum_{i=0}^{t} 3^i \binom{2^n}{i}. \tag{2.28}$$

To ensure that each of these errors yields a distinct syndrome (i.e., is detectable and correctable), it is required that

$$2^n \geq 2^{n-k} \cdot \sum_{i=0}^{t} 3^i \binom{2^n}{i} \tag{2.29}$$

This is the quantum Hamming bound. If one aims at finding the smallest number of physical qubits needed to protect a single qubit from single erros, by setting $t = 1$ and $n - k = 1$, the inequality becomes:

$$2(3n + 1) \leq 2^n \tag{2.30}$$

Solving this inequality, the smallest integer n for which it holds is $n = 5$. Thus, at least five physical qubits are required to protect one logical qubit against all single-qubit errors. Notably, there exists a [[5,1,3]] stabilizer code that saturates this bound. It encodes one logical qubit into five physical qubits, has distance $d = 3$, and corrects any arbitrary single-qubit error. Such a code is called *perfect*, since equality is achieved in the quantum Hamming bound. In analogy with classical codes, a quantum code is defined to be perfect if the total number of correctable errors exactly matches the upper bound given by the quantum Hamming bound [1]. More in detail, the generators of the [[5,1,3]] stabilizer code exhibit symmetry under cyclic permutation of the qubits. An explicit set of stabilizer generators is given by:

$$S_1 = XZZXI \tag{2.31}$$
$$S_2 = IXZZX \tag{2.32}$$
$$S_3 = XIXZZ \tag{2.33}$$
$$S_4 = ZXIXZ \tag{2.34}$$

The automorphism group of the code corresponds to the dihedral group of order 10. A natural graphical representation of the code, respecting the structure of the encoder, takes the form of a pentagon with an additional central input node. This is the unique quantum stabilizer code with parameters [[5,1,3]] up to equivalence. In fact, any transversal stabilizer code with these parameters and distance must necessarily coincide with the five-qubit code. The code is also sometimes referred to as the *DiVincenzo–Shor code*, in reference to a study that analyzed its syndrome extraction circuitry. Further technical details and background can be found in the Error Correction Zoo website [9].

2.4.4 CSS Codes

Calderbank-Shor-Steane (CSS) codes form a distinguished subclass of quantum stabilizer codes that can be constructed from pairs of classical linear binary codes. These codes are especially important due to their algebraic transparency, their support for transversal logical gates, and the simplification they bring to the syndrome extraction process. The general framework is well described in [10].

Definition and Construction Let $C_1, C_2 \subseteq \mathbb{F}_2^n$ be two classical binary linear codes such that $C_2 \subseteq C_1$ and $C_1, C_2^\perp$ are both capable of correcting up to t errors. Then, a CSS code constructed from the pair (C_1, C_2) encodes

$$k = \dim C_1 - \dim C_2 \tag{2.35}$$

logical qubits into n physical qubits. The code distance is given by

$$d = \min \left\{ w(v) : v \in (C_1 \setminus C_2) \cup (C_2^\perp \setminus C_1^\perp) \right\}, \tag{2.36}$$

where $w(v)$ denotes the Hamming weight of v.

Stabilizer Formalism The stabilizer group $S \subset \mathcal{G}_n$ associated with the CSS code is generated by two sets of Pauli operators:

- Z-type stabilizers derived from the generator matrix G_2 of C_2,
- X-type stabilizers derived from the generator matrix $G_1^\perp$ of the dual code $C_1^\perp$.

Explicitly, the stabilizer group is given by:

$$S = \langle Z^g : g \in C_2 \rangle \cup \left\langle X^h : h \in C_1^\perp \right\rangle, \tag{2.37}$$

where the notation Z^g indicates the application of Pauli-Z on qubits indexed by the support of g, and analogously for X^h. The orthogonality condition

$$\forall g \in C_2, \forall h \in C_1^\perp, \quad g \cdot h = 0 \tag{2.38}$$

guarantees that all generators commute, as required in the stabilizer formalism.

Matrix Representation In terms of the binary stabilizer matrix formalism (using symplectic representation), the stabilizer group can be written as a binary matrix $G \in \mathbb{F}_2^{r \times 2n}$, where:

$$G = \begin{bmatrix} 0 & H_{C_2} \\ H_{C_1^\perp} & 0 \end{bmatrix}, \tag{2.39}$$

with H_{C_2} being the parity-check matrix of C_2 and $H_{C_1^\perp}$ the parity-check matrix of $C_1^\perp$. This structure reflects that all stabilizers are composed either purely of Pauli-Z

or purely of Pauli-X, and not mixed Y-type operators. This property significantly simplifies fault-tolerant circuit design and syndrome extraction.

Logical Operators Logical Pauli operators are constructed as representatives of the quotient spaces:

$$\bar{X}_i \in C_1 \setminus C_2, \quad \bar{Z}_i \in C_2^\perp \setminus C_1^\perp, \tag{2.40}$$

for $i = 1, \ldots, k$, such that the canonical commutation relations are satisfied:

$$\bar{X}_i \bar{Z}_j = (-1)^{\delta_{ij}} \bar{Z}_j \bar{X}_i. \tag{2.41}$$

Encoding Let $\{u_i\} \subset C_1$ be coset representatives of C_1/C_2. Each logical state corresponds to a uniform superposition over the associated coset:

$$|u_i\rangle_L = \frac{1}{\sqrt{|C_2|}} \sum_{c \in C_2} |u_i + c\rangle. \tag{2.42}$$

Example: The [[4,2,2]] CSS Code

The [[4,2,2]] code is the smallest qubit CSS stabilizer code capable of detecting any single-qubit error. It is defined by the classical codes:

$$C_1 = [4,3,2] \text{ (SPC code)}, \quad C_2 = [4,1,4] \text{ (Repetition code)}. \tag{2.43}$$

A single-parity-check (SPC) code is a linear $[n, n-1, 2]$ code defined over $\mathbb{F}_2$, where each codeword satisfies a single linear constraint: the sum (modulo 2) of all bits must be zero. Formally, the code consists of all vectors $\vec{x} \in \mathbb{F}_2^n$ such that $\sum_{i=1}^n x_i = 0$. The minimum distance is 2, allowing the detection (but not correction) of any single-bit error. Its parity-check matrix is a single row of all ones.

The binary repetition code of length n is a linear $[n,1,n]$ code over $\mathbb{F}_2$, consisting of only two codewords: the all-zero vector and the all-one vector. It encodes one bit of information redundantly and achieves the maximum possible minimum distance n, enabling correction of up to $\lfloor \frac{n-1}{2} \rfloor$ errors. The generator matrix is a row vector of all ones, and the parity-check matrix has $n-1$ linearly independent rows.

The generator matrices are:

$$G_1 = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}, \qquad G_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}. \tag{2.44}$$

The corresponding stabilizer matrix in binary symplectic form is:

$$G = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{2.45}$$

Its codewords span a four-dimensional Hilbert space with logical basis states:

$$|00\rangle_L = \frac{1}{\sqrt{2}}(|0000\rangle + |1111\rangle), \tag{2.46}$$

$$|01\rangle_L = \frac{1}{\sqrt{2}}(|1100\rangle + |0011\rangle), \tag{2.47}$$

$$|10\rangle_L = \frac{1}{\sqrt{2}}(|0101\rangle + |1010\rangle), \tag{2.48}$$

$$|11\rangle_L = \frac{1}{\sqrt{2}}(|0110\rangle + |1001\rangle). \tag{2.49}$$

Structure and Subcodes This code can be interpreted as a concatenation of a two-qubit bit-flip code with a two-qubit phase-flip code. Additionally, the code contains a subcode known as the Leung-Nielsen-Chuang-Yamamoto (LNCY) code that encodes one logical qubit with the basis:

$$|0\rangle_{L'} = \frac{1}{\sqrt{2}}(|0000\rangle + |1111\rangle), \tag{2.50}$$

$$|1\rangle_{L'} = \frac{1}{\sqrt{2}}(|0101\rangle + |1010\rangle). \tag{2.51}$$

The [[4,2,2]] code is unique up to local Clifford equivalence for its parameters and admits a graphical representation as a square lattice (planar surface code with open boundaries) [9]. The surface code is defined on a square lattice of size $L \times L$, where qubits reside on the edges. For analytical convenience, periodic boundary conditions are often assumed. The code is specified by its stabilizer group, generated by two types of operators: vertex stabilizers A_v, which are tensor products of four Z operators around each vertex, and plaquette stabilizers B_p, tensor products of four X operators around each face. These stabilizers commute because each pair intersects on an even number of qubits.

All X-type stabilizers correspond to loops on the lattice formed by plaquettes; their products form closed loops due to cancellation of X operators on overlapping edges. Similarly, Z-type stabilizers correspond to loops on the dual lattice, where vertex stabilizers become plaquette-like operators. Thus, both types of stabilizers are represented by loops, either on the primal or dual lattice. Errors create excitations (measured syndromes) only at the boundaries of error strings, which appear in pairs and move as the error string increases. When an error string forms

a loop, excitations vanish, i.e., loops always commute with all stabilizers. For further details, refer to the bibliography [11].

2.5 The Gottesman-Knill Theorem

Before stating the Gottesman-Knill theorem, it is appropriate to define a key class of quantum operations known as *Clifford gates*.

Definition. *The Clifford group on n qubits is defined as the normalizer of the n-qubit Pauli group $\mathcal{G}_n$ within the unitary group $U(2^n)$. A unitary operator U belongs to the Clifford group if*

$$U\mathcal{G}_n U^\dagger \subseteq \mathcal{G}_n.$$

Equivalently, U is a Clifford operation if for every Pauli operator $P \in \mathcal{G}_n$, the conjugated operator $UPU^\dagger$ is also a Pauli operator.

The Clifford group is generated by the following gates:

- The Hadamard gate H, which maps $X \leftrightarrow Z$;
- The phase gate $P = \mathrm{diag}(1, i)$, which maps $X \to Y$;
- The controlled-NOT (CNOT) gate, which maps tensor products of Pauli operators to other such products.

Theorem (Gottesman-Knill). *Any quantum computation that involves only the following operations can be efficiently simulated on a classical computer:*

- *Initialization of qubits in computational basis states;*
- *Application of Clifford gates (Hadamard, phase, and CNOT);*
- *Application of Pauli gates (X, Y, Z);*
- *Measurements of observables in the Pauli group;*
- *Classical control conditioned on previous measurement outcomes.*

Each Clifford operation corresponds to a deterministic update of the stabilizer generators. For instance, if a Hadamard gate is applied to qubit i, each generator M is updated via conjugation: $M \mapsto H_i M H_i^\dagger$.

Each such update can be performed in $O(n^2)$ time, where n is the number of qubits. Hence, a circuit consisting of m Clifford operations can be simulated classically in time $O(n^2 m)$.

The Gottesman-Knill theorem illustrates that the presence of quantum entanglement is not sufficient to guarantee exponential computational advantage. Several important quantum protocols, including teleportation and superdense coding, are implementable within the Clifford framework and therefore admit efficient classical simulation. Exponential speedup over classical computation becomes possible only when operations outside the Clifford group are incorporated.

2.6 Fault-Tolerant Quantum Computation

A quantum operation is said to be *fault-tolerant* if a single physical error during its execution causes at most one error in each encoded block. This guarantees that if the number of errors per block remains within the error-correcting capacity of the code, the original quantum information can still be recovered.

More formally, consider a quantum code that encodes k logical qubits into n physical qubits, capable of correcting up to t errors. A fault-tolerant circuit ensures that any single fault leads to at most one error per block, preserving the ability of the code to correct up to t errors.

Error correction must be implemented in a fault-tolerant manner in order to prevent the propagation of errors during the measurement of stabilizer generators. Consider the task of measuring a multi-qubit Pauli operator, for example $M = Z_1 Z_2$. A straightforward implementation employs a single ancilla qubit and two CNOT gates, with control lines from the data qubits to the ancilla. This configuration is vulnerable to fault propagation: a phase error on the ancilla may propagate backward to both data qubits, potentially introducing multiple errors within the same code block.

To avoid this problem, a fault-tolerant approach uses a specially prepared ancilla in an entangled state (e.g., a cat state). This ancilla is verified prior to use, and only valid ancilla states are employed in the syndrome extraction process. By coupling the verified ancilla to the data block in a controlled way, it is possible to extract the error syndrome without directly entangling the data qubits.

If verification fails, the ancilla is discarded and a new one is prepared. This procedure reduces the risk of introducing correlated errors during syndrome extraction.

The result of the measurement yields a classical syndrome vector $s \in \mathbb{F}_2^{n-k}$, which identifies the error up to stabilizer equivalence.

Universal quantum computation requires the ability to implement at least one gate that lies outside $N(\mathcal{G}_n)$. Examples include the Toffoli gate and the T gate (i.e., the $\pi/8$ phase rotation gate). Such gates cannot in general be implemented transversally in stabilizer codes.

A standard method for implementing non-Clifford gates fault-tolerantly is based on *gate teleportation*. This approach consists of the following steps:

1. Preparation of a special ancilla state encoding the desired gate,

2. Coupling of the data block and ancilla via a Clifford subcircuit,

3. Measurement of the ancilla and application of classically controlled correction operators to the data block.

Ancilla verification is essential to ensure that faults do not propagate during the interaction, preserving the overall fault-tolerant structure of the computation.

2.6.1 Propagation of Errors in Quantum Gates

In quantum circuits, certain gates can propagate errors between qubits. For example, in a controlled-NOT (CNOT) gate, a bit-flip (X) error on the control qubit propagates forward to the target, while a phase-flip (Z) error on the target propagates backward to the control. This is summarized as follows:

- $X \otimes I \rightarrow X \otimes X$ (bit-flip propagates forward),
- $I \otimes Z \rightarrow Z \otimes Z$ (phase-flip propagates backward).

A common strategy to achieve fault tolerance is the use of *transversal gates*. A transversal gate acts independently on corresponding qubits in different blocks. For example, applying a single-qubit gate U transversally to a block means applying U to each physical qubit in the block individually. Since each qubit only interacts with its counterpart, a single physical fault cannot propagate within a block, making the operation inherently fault-tolerant. However, not all gates in a universal gate set can be implemented transversally.

Following the overview of the fault-tolerant mechanism, it is instructive to analyze a representative example of a fault-tolerant implementation, namely that of the CNOT gate [1].

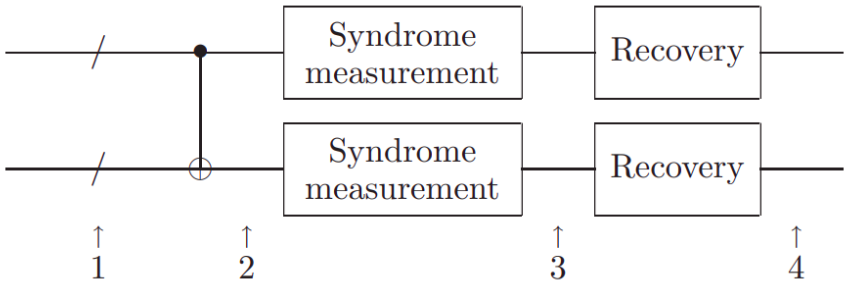


Figure 2.1: Logical structure of the fault-tolerant protocol, with explicit inclusion of error-correction stages [1].

By looking at Fig. 2.1, it is possible to notice the following:

- A single pre-existing error may be present at step 1 in each encoded block. Although this error originates outside the current fault-tolerant gadget, it can propagate through the circuit and induce multiple errors. If all operations up to this point have been performed in a fault-tolerant manner, the probability that such a pre-existing error arises from the prior syndrome extraction or recovery steps is upper bounded by $c_0 p$ per block. Assuming independent errors on both blocks, the joint probability of such a configuration is at most $c_0^2 p^2$, where c_0 quantifies the number of possible fault locations during syndrome measurement and recovery.

- A single error may enter one of the blocks at step 1 (either from an earlier stage or due to residual noise), and a separate fault may occur during the execution of the fault-tolerant CNOT gate. If these two errors are sufficiently correlated or lie in locations that cause them to propagate and combine destructively, the result may be two or more errors in the output block. The joint probability of this event is bounded by $c_1 p^2$, where c_1 represents the number of pairs of failure locations.

- Two independent faults may occur during the execution of the encoded CNOT itself. In a fault-tolerant construction, the gate is typically implemented transversally or via an ancilla-assisted method, but simultaneous faults in distinct locations can lead to a breakdown of the fault-tolerance condition. The total number of such failure pairs gives a probability upper bound of $c_2 p^2$, where c_2 is determined by the combinatorics of fault locations within the CNOT implementation.

- A fault may occur during the CNOT gate, and another during the following syndrome measurement. If the syndrome measurement yields an incorrect outcome due to the latter fault, the recovery procedure may introduce additional errors instead of correcting them. The logical error only manifests if the syndrome is incorrect; thus, the total probability of this combined event is $c_3 p^2$, where c_3 counts combinations of failure points across gate execution and syndrome extraction.

- Two or more faults may happen during syndrome extraction itself. Although individual faults are typically corrected, multiple faults can corrupt the syndrome data beyond the code's error-correcting capacity. This type of event occurs with probability at most $c_4 p^2$, where c_4 reflects the number of fault pairs within the syndrome measurement circuit.

- One fault may occur during syndrome measurement, and a second fault during recovery. If both events occur in a manner that leads to an incorrect outcome (either by misinterpreting the syndrome or introducing an error during recovery) the system may end up with an uncorrectable state. The joint probability of this situation is bounded by $c_5 p^2$, with c_5 depending on the circuit depth and interaction points between measurement and recovery stages.

- Two or more faults may occur during the recovery process itself. Since recovery involves conditional operations based on the measured syndrome, multiple faults in this phase can directly lead to logical errors. The probability of such a case is upper bounded by $c_6 p^2$, where c_6 is the number of relevant fault pairs in the recovery circuitry.

The total probability of introducing two or more errors into the output of the

encoded block is therefore bounded by

$$P_{\text{fail}} \leq cp^2, \quad \text{with} \quad c = c_0^2 + c_1 + c_2 + c_3 + c_4 + c_5 + c_6. \tag{2.52}$$

This shows that, provided the physical error rate p is sufficiently small (for instance, $p < 10^{-4}$), the overall logical error rate can be suppressed quadratically, demonstrating the effectiveness of fault-tolerant procedures in reducing error propagation within quantum circuits with respect to unencoded implementations.

In general, in order to achieve universal fault-tolerant computation for a given code, the initial step involves generating the encoded CNOT for that code. For the most general stabilizer code, this process necessitates a four-qubit operation utilizing two ancilla qubits, followed by two measurements. Within a CSS code, this procedure simplifies considerably, requiring only a single transversal operation.

Subsequently, to implement one-qubit operations, one ancilla qubit is needed, along with a CNOT operation and a measurement. This requirement persists even for the most general CSS code. However, if the code exhibits the property where $C_1 = C_2$ (thereby implying $C_1^{\perp} \subseteq C_1$), then the σ_x generators adopt the same form as the σ_z generators. Consequently, a transversal Hadamard rotation is also a valid fault-tolerant operation. Furthermore, if the parity check matrix of C_1 contains a multiple of four 1s in each row, then the transversal phase P similarly becomes a valid fault-tolerant operation.

For a general CSS code that satisfies these conditions, these operations will execute a multiple-qubit gate on the qubits encoded within a single block. Nonetheless, if each block exclusively encodes a single qubit, the $\bar{X}$ and $\bar{Z}$ operators can be conveniently chosen such that a transversal Hadamard performs an encoded Hadamard rotation, and a transversal P performs an encoded P or $P^{\dagger}$. To attain universal computation, the Toffoli gate or another gate external to $N(G)$ will also be indispensable, and this will almost invariably demand a more intricate construction. For further information, refer once again to the bibliography [7].

Basis Operations	
Logical Basis	**Physical Basis**
$X_1 \otimes I_2$	$X \otimes I \otimes X \otimes I$
$I_1 \otimes X_2$	$X \otimes X \otimes I \otimes I$
$Z_1 \otimes I_2$	$Z \otimes Z \otimes I \otimes I$
$I_1 \otimes Z_2$	$Z \otimes I \otimes Z \otimes I$
$H_1 \otimes H_2$	$H \otimes H \otimes H \otimes H$
$CNOT_{12}$	$SWAP_{12}$
$CNOT_{21}$	$SWAP_{13}$

Table 2.1: Fault-tolerant basis operations for the [[4,2,2]] code [12].

Chapter 3

Theory of Superconductivity

This section provides an overview of superconductivity, as the quantum processor employed for all the experiments (IQM Spark) is based on superconducting technology.

This chapter draws conceptual inspiration from the work of Neil W. Ashcroft and N. David Mermin [13], with additional references taken from personal course materials [14, 15, 16].

3.1 Basic experimental evidences

A brief description of the general properties of superconductors is provided below.

3.1.1 Perfect conductivity

The electrical resistance of a superconductor vanishes completely when the material is cooled below a characteristic critical temperature T_c. This phenomenon was first discovered by Kamerlingh Onnes in 1911 during experiments with mercury.

Very pure samples exhibit a sharp superconducting transition, while impure or "dirty" samples show a broadened transition (see Fig. 3.1). Notably, even the best samples of high-T_c superconductors do not display an extremely sharp transition, with a typical transition width ΔT_c exceeding 0.3 K.

3.1.2 Perfect diamagnetism (Meissner effect)

Superconductors differ from perfect conductors in their magnetic response. While a perfect conductor only preserves the magnetic flux already present when it becomes resistanceless, a superconductor always expels magnetic flux upon cooling below T_c.

This phenomenon, discovered by Meissner and Ochsenfeld in 1933, proves that superconductivity is a distinct state of matter. The magnetic field penetrates only to a finite depth, the *penetration depth* λ, typically a few tens of nanometers.

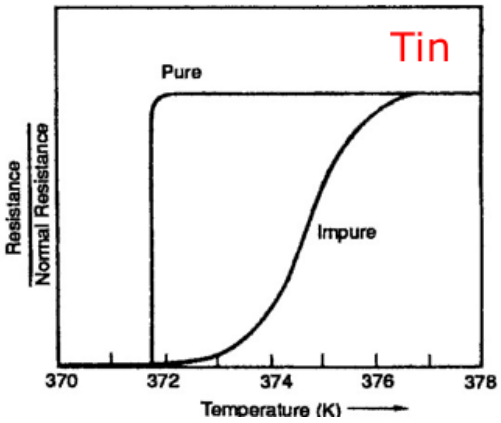


Figure 3.1: Comparison of resistance vs.temperature for pure and impure superconductors [14, 15].

Any claim of a new superconductor requires demonstration of both zero resistance and perfect diamagnetism.

3.1.3 Critical magnetic field

Superconductivity is destroyed above a critical magnetic field H_c, related to the condensation energy of the superconducting state. For type-I superconductors, the temperature dependence is well approximated by:

$$H_c(T) \approx H_c(0)\left[1 - \left(\frac{T}{T_c}\right)^2\right].$$

Type-II superconductors, which include most alloys and compounds, display two critical fields H_{c1} and H_{c2}, defining a mixed state in which flux partially penetrates the material (Fig. 3.2).

A critical transport current density J_c also exists, beyond which superconductivity breaks down (Silsbee effect). The interplay of temperature, field, and current defines a *critical surface* (Fig. 3.3).

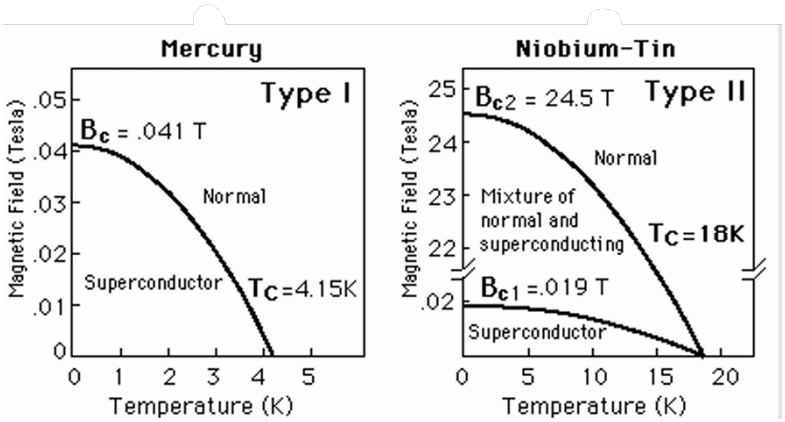


Figure 3.2: Temperature dependence of the critical magnetic field in type-I and type-II superconductors [15].

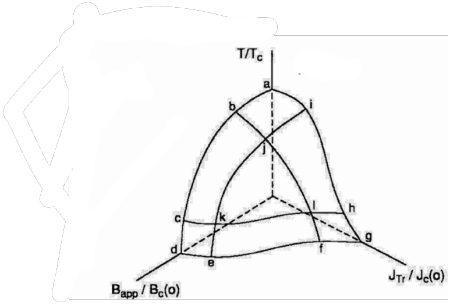


Figure 3.3: Critical surface describing the superconducting-to-normal transition [15].

3.1.4 Energy gap

Experiments reveal the existence of an excitation gap in the superconducting state:

- The exponential decay of the specific heat at low temperatures.
- The threshold photon energy observed in absorption spectra.
- Tunneling spectroscopy of the density of states.

The gap has magnitude on the order of $2{\times}1.5\,k_B T_c$, confirming that superconducting charge carriers exist only for excitations below this threshold.

3.2 The Macroscopic Quantum Model

In 1935, the London brothers introduced equations that successfully described zero resistance and the Meissner effect, though without microscopic justification.

Fritz London later emphasized that superconductivity must be understood as a *macroscopic quantum phenomenon*, analogous to the coherence of light in a laser. The key assumption is the existence of a collective quantum wavefunction:

$$\Psi(\mathbf{r}, t) = \Psi_0(\mathbf{r}, t)e^{i\theta(\mathbf{r},t)},$$

where $|\Psi|^2$ represents the density of superconducting carriers.

Within this framework:

- The supercurrent depends on the phase gradient and vector potential, linking superconductivity to electromagnetic fields.
- The London equations emerge naturally, predicting magnetic field expulsion over the penetration depth λ.
- Fluxoid quantization arises, showing that magnetic flux is quantized in units of
$$\Phi_0 = \frac{h}{2e} \approx 2.07 \times 10^{-15}\,\mathrm{Tm}^2.$$

Flux quantization provides direct evidence that the charge carriers in superconductors are Cooper pairs with effective charge $2e$.

3.2.1 The Josephson Effect and the DC Josephson Effect

The Josephson effect refers to the quantum mechanical tunneling of Cooper pairs between two superconductors separated by a thin insulating barrier (see Fig. 3.4). This process arises from the overlap of the superconducting wavefunctions in the barrier region, leading to a coherent supercurrent even in the absence of an applied voltage.

For a junction composed of two superconductors with respective macroscopic wavefunctions $\Psi_1 = |\Psi_1|e^{i\theta_1}$ and $\Psi_2 = |\Psi_2|e^{i\theta_2}$, the resulting supercurrent across the junction is given by:

$$\mathbf{J}_s = \mathbf{J}_c \sin(\theta_1 - \theta_2), \tag{3.1}$$

where $\mathbf{J}_c$ denotes the critical current density, and $\theta_1 - \theta_2$ is the phase difference across the junction. This relation is known as the *Josephson current-phase relation*, and it represents the DC Josephson effect: it shows that the supercurrent flowing through a Josephson junction varies sinusoidally with the phase difference across it.

The critical current density can be expressed in terms of the Cooper pair parameters and the geometry of the junction. For tunnel junctions, the current decays exponentially with increasing thickness $2a$ of the barrier:

$$J_c \propto \sinh^{-1}(2c/\zeta) \approx \frac{1}{2}\exp(-2a/\zeta), \tag{3.2}$$

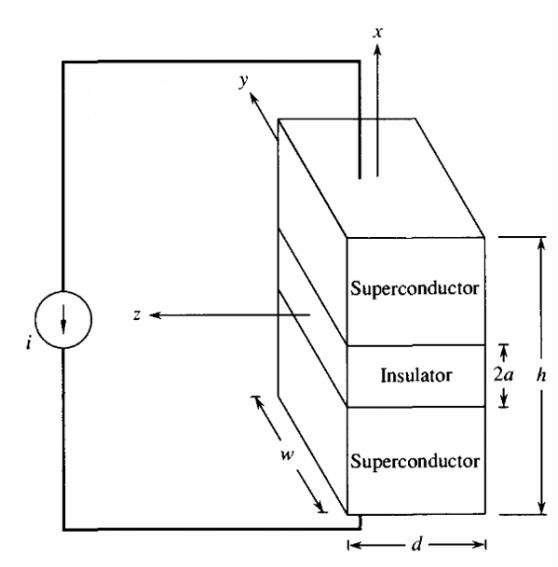


Figure 3.4: Schematic representation of a Josephson Junction [15].

where ζ denotes the characteristic decay length of the wavefunction in the insulating layer.

Unlike conventional electron tunneling, this effect occurs without requiring an applied bias voltage and persists as long as the current remains below the critical threshold. This dissipationless current is a direct result of phase coherence between the superconducting condensates.

Overall, The fundamental equations governing the dynamics of a Josephson junction are the following:

1. **Current-phase relation**:

$$\mathbf{J}_s(\mathbf{r}, t) = \mathbf{J}_c(y, z, t) \sin \varphi(y, z, t) \tag{3.3}$$

2. **Gauge-invariant phase difference**:

$$\varphi(y, z, t) = \theta_1(y, z, t) - \theta_2(y, z, t) - \frac{2\pi}{\Phi_0} \int_{\mathbf{r}_1}^{\mathbf{r}_2} \mathbf{A}(\mathbf{r}, t) \cdot d\mathbf{l} \tag{3.4}$$

3. **Voltage-phase relation**:

$$\frac{\partial \varphi(y, z, t)}{\partial t} = \frac{2\pi}{\Phi_0} \int_{\mathbf{r}_1}^{\mathbf{r}_2} \mathbf{E}(\mathbf{r}, t) \cdot d\mathbf{l} \tag{3.5}$$

These relations were initially derived for tunnel junctions but remain applicable to a wider class of weak-link structures in which the superconducting wavefunctions

overlap via their evanescent tails. Representative examples include point-contact junctions, microbridges, and constrictions.

3.2.2 Basic lumped Junctions and the AC Josephson Effect

When a constant voltage V_0 is applied across the junction, the phase difference evolves in time according to the voltage-phase relation:

$$\frac{d\varphi(t)}{dt} = \frac{2\pi}{\Phi_0} V_0, \tag{3.6}$$

where $\varphi(t) = \theta_1(t) - \theta_2(t)$ is the gauge-invariant phase difference.

Integrating this relation yields:

$$\varphi(t) = \varphi_0 + \frac{2\pi V_0}{\Phi_0} t, \tag{3.7}$$

which, when substituted into the current-phase relation, leads to a time-dependent supercurrent:

$$I(t) = I_c \sin\left(\varphi_0 + \frac{2\pi V_0}{\Phi_0} t\right). \tag{3.8}$$

This phenomenon is referred to as the *AC Josephson effect*, and it implies that a DC voltage induces an AC supercurrent oscillating at the *Josephson frequency*:

$$f_J = \frac{2e}{h} V_0 \approx 483.6\,\mathrm{GHz/mV}. \tag{3.9}$$

This effect demonstrates that a Josephson junction acts as an ideal voltage-to-frequency converter. The relationship is used in metrology to establish primary voltage standards, exploiting the precision of frequency measurements.

Furthermore, when the junction is driven by both DC and AC voltage sources (e.g., $V(t) = V_0 + V_s \cos(\omega_s t)$), the resulting current includes frequency mixing terms, leading to phenomena such as Shapiro steps. These are discrete values of V_0 for which a DC component in the current is observed, given by:

$$V_0 = \frac{nhf_s}{2e}, \quad n \in \mathbb{Z}, \tag{3.10}$$

which again confirms the quantum nature of the phase dynamics.

Finally, the dynamic response of Josephson junctions also reveals their non-linear inductive behavior. By differentiating the current-phase relation and using the voltage-phase relation, one obtains:

$$\frac{dI}{dt} = \frac{I_c 2\pi}{\Phi_0} \cos(\varphi) V(t), \tag{3.11}$$

which is equivalent to the response of a non-linear inductor with Josephson inductance:

$$L_J(\varphi) = \frac{\Phi_0}{2\pi I_c \cos(\varphi)}. \tag{3.12}$$

This kinetic inductance originates from the inertia of Cooper pairs rather than from magnetic energy storage and is a key design parameter in superconducting circuit models.

3.3 Implementation of Superconducting Qubits

Superconducting qubits are artificial two-level quantum systems implemented using nonlinear electrical circuits. These circuits rely on Josephson junctions to introduce anharmonicity into otherwise harmonic resonators, allowing quantum information to be stored and manipulated within discrete energy levels.

Moreover, the quantity $\hbar\omega/k_B$ defines the characteristic energy scale associated with quantum excitations in the system. Thermal fluctuations become strongly suppressed when the operating temperature satisfies $T \ll \hbar\omega/k_B$.

For a typical superconducting circuit operating at microwave frequencies, such as $\omega = 2\pi \times 8\,\mathrm{GHz}$, the corresponding thermal energy scale is approximately:

$$\frac{\hbar\omega}{k_B} \approx 0.38\,\mathrm{K}. \tag{3.13}$$

The excitation probability of the qubit due to thermal fluctuations decreases exponentially with temperature. Representative values are summarized below:

Temperature (mK)	**Excitation Probability**
400	38%
100	2%
50	0.04%
10	$\sim 10^{-15}$%

Therefore, operating the quantum circuit inside a dilution refrigerator capable of reaching temperatures in the 20–50 mK range ensures that thermal excitation probabilities are negligible, preserving the qubit in its ground state with high fidelity.

The simplest realization is the *charge qubit*, consisting of a small superconducting island connected to a ground plane via a Josephson junction, and capacitively coupled to a voltage source through a gate capacitor. The lumped elements in the circuit include:

- a Josephson junction with critical current I_c,
- a total capacitance C to ground (including the junction's intrinsic capacitance),
- a gate capacitance C_g,
- a voltage source V.

The quantization procedure follows that of the linear LC resonator, but the current-flux relation of the junction introduces a nonlinearity:

$$I_a = I_c \sin\left(\frac{\phi_a}{\varphi_0}\right), \tag{3.14}$$

where ϕ_a is the superconducting phase difference and $\varphi_0 = \Phi_0/2\pi$ is the reduced flux quantum.

The Lagrangian of the system leads to a Hamiltonian of the form:

$$\hat{H} = 4E_C(\hat{n} - n_g)^2 - E_J \cos\hat{\varphi}, \tag{3.15}$$

where $E_C = e^2/2C_\Sigma$ is the charging energy, $E_J = I_c\varphi_0$ is the Josephson energy, $\hat{n}$ is the Cooper pair number operator, and $\hat{\varphi}$ is the phase operator conjugate to $\hat{n}$. The offset charge is defined as $n_g = C_g V/2e$.

Another variant is the *phase qubit* that operates in the regime $E_J \gg E_C$, where the Josephson energy dominates over the charging energy. In this limit, the superconducting phase $\hat{\varphi}$ becomes a well-defined quantum variable, while the charge $\hat{n}$ undergoes strong quantum fluctuations. The potential energy landscape takes the form of a tilted cosine potential:

$$U(\varphi) = -E_J \cos\varphi - I_{\text{bias}}\varphi_0\varphi, \tag{3.16}$$

where I_{bias} is an externally applied current bias. For suitable values of I_{bias}, the potential forms an asymmetric well in which discrete quantized levels appear. Quantum state manipulation is achieved via microwave excitation resonant with the transition frequency ω_{01}. Readout typically exploits quantum tunneling from the excited state out of the potential well.

The *flux qubit* is based on a superconducting loop interrupted by one or more Josephson junctions and biased by an external magnetic flux Φ_{ext}. Its dynamics are governed by the flux Φ threading the loop, which plays the role of the canonical coordinate. Here the potential forms a double-well structure in the phase space. The two lowest eigenstates, corresponding to clockwise and counter-clockwise circulating persistent currents, define the computational basis states. Tunneling between the wells gives rise to quantum coherence, and the energy splitting is tunable via Φ_{ext}.

The scheme in Fig. 3.5 summarizes the different implementations of superconducting qubits based on the Josephson junction, each corresponding to a distinct regime of circuit parameters and quantum variables.

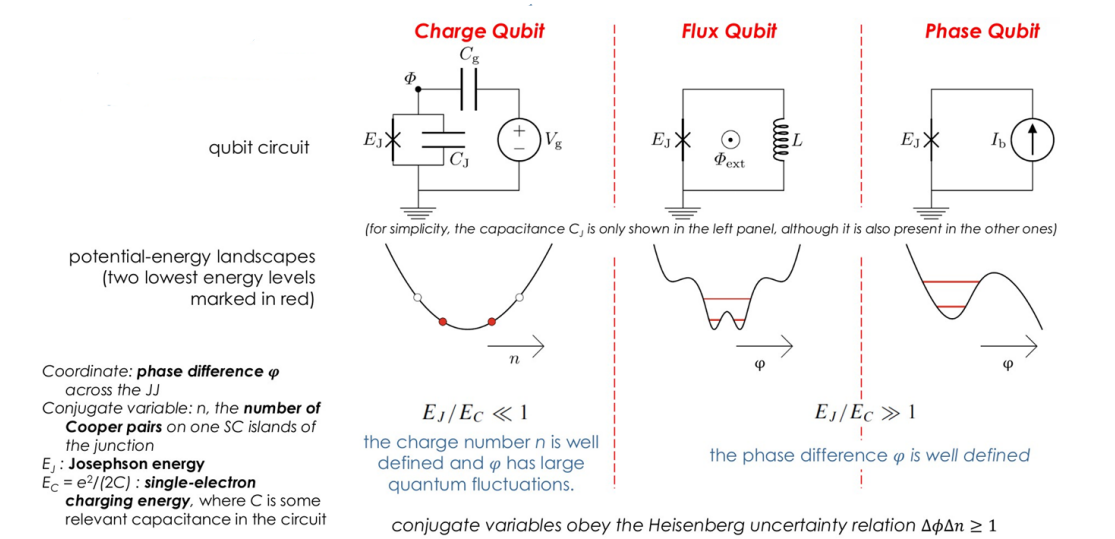


Figure 3.5: Scematic representation and summary of the three basic Josephson-Jucntion qubits [16].

In the regime $E_C \gg E_J$, the energy spectrum of a superconducting qubit becomes highly sensitive to background charge fluctuations. This charge noise significantly limits coherence times. To mitigate this sensitivity, the *transmon qubit* was developed by shunting the Josephson junction with a large additional capacitance. Charge and transmon qubits can be described by the same circuit, what changes is the parameters regime. This design moves the system into the opposite regime $E_J \gg E_C$, in which the qubit becomes exponentially less sensitive to offset charge, while still preserving sufficient anharmonicity to function effectively as a two-level system.

In this transmon configuration, the cosine potential associated with the Josephson junction becomes weakly anharmonic, resulting in an energy level spectrum that is nearly harmonic but with a small anharmonicity defined as:

$$\alpha = \omega_{12} - \omega_{01}, \tag{3.17}$$

where ω_{ij} denotes the transition frequency between energy levels $|i\rangle$ and $|j\rangle$. This anharmonicity ensures that selective excitation of the $|0\rangle \leftrightarrow |1\rangle$ transition is possible without inadvertently populating higher energy levels.

To enable in-situ tunability of the Josephson energy E_J, the single junction is often replaced by a Superconducting Quantum Interference Device (SQUID). A SQUID consists of a superconducting loop interrupted by one or more Josephson junctions. When placed in a magnetic field, the effective Josephson energy becomes a function of the external magnetic flux $\mathbf{\Phi}$ threading the loop:

$$E_J(\Phi) = E_{J,\max} \left| \cos\left(\frac{\pi\Phi}{\Phi_0}\right) \right|, \tag{3.18}$$

where $\Phi_0 = h/2e$ is the flux quantum. This tunability provides dynamic control over the qubit transition frequency, enabling functionalities such as frequency multiplexing, qubit-qubit detuning, and flux-based gate operations.

There are two principal types of SQUIDs used in superconducting circuits:

- **RF-SQUID**: This configuration consists of a single Josephson junction embedded in a superconducting loop. The loop behaves as a nonlinear inductor, and the system is typically driven by an external radio-frequency magnetic flux. The RF-SQUID supports a flux-tunable potential landscape and can be used as a qubit in the flux regime. It features a relatively simple design but offers limited tunability and is generally more sensitive to flux noise.

- **DC-SQUID**: The DC-SQUID consists of two Josephson junctions connected in parallel within a superconducting loop. This arrangement enables precise modulation of the effective critical current (and hence the Josephson energy) by means of an externally applied static magnetic flux. The DC-SQUID provides symmetric tuning of E_J and is widely employed in transmon-type architectures due to its higher flux tunability and compatibility with planar circuit layouts.

Fig. 3.6 shows the representations of the RF-SQUID and the DC-SQUID.

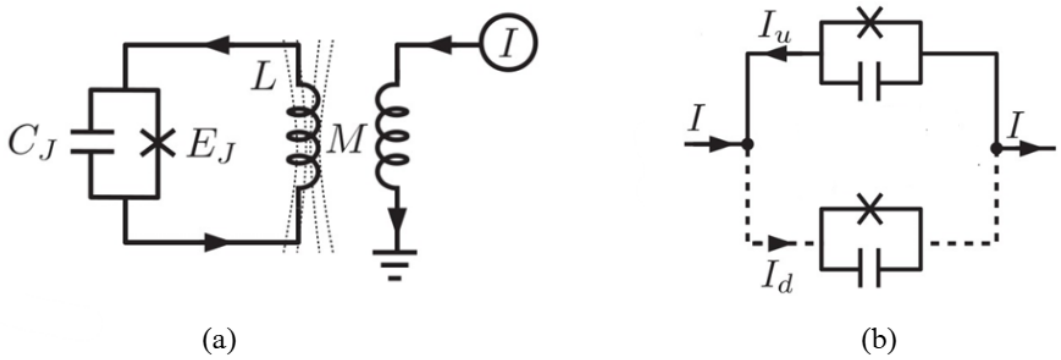


Figure 3.6: Schematic representation of (a) the RF-SQUID and (b) the DC-SQUID [16].

3.4 The IQM Spark Quantum Computer

In this section, a brief description of the IQM Spark quantum computer is provided, by summarizing the information reported in the paper by Rönkkö et al. [17].

3.4.1 Overview

The Quantum Processing Unit (QPU) in this system consists of five data qubits arranged in a star configuration. A central qubit is linked to the four outer qubits through tunable couplers. The layout is created using `KQCircuits`, an open-source extension for `KLayout`, which allows for circuit design, simulation, and export for fabrication.

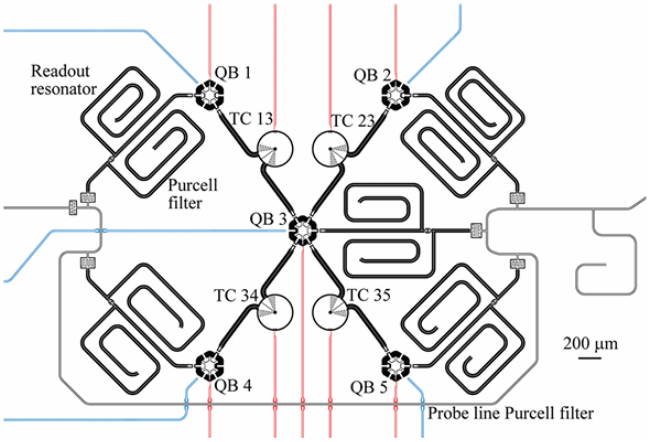


Figure 3.7: Layout of the 5-qubit superconducting quantum processor, featuring five qubits (QB) interconnected by four tunable couplers (TC). Regions shaded in black represent areas where the superconducting film has been selectively etched to reveal the substrate. Flux control lines are shown in red, while microwave drive lines are depicted in blue [17].

3.4.2 Qubit Type

The quantum processor utilizes transmon qubits, which are nonlinear superconducting oscillators engineered for stability and coherence. Structurally, each transmon consists of a SQUID (two Josephson junctions connected in parallel) shunted by a large capacitance. This configuration ensures that the Josephson energy dominates over the charging energy by a factor of few tens, thereby reducing sensitivity to charge and flux noise.

The qubit capacitor is realized through a metallic island, patterned with six-fold rotational symmetry, and separated from the ground plane by etched gaps that

expose the underlying dielectric. Each sector of the island contributes to the total capacitance, with carefully varied dimensions allowing tailored capacitive coupling to neighboring qubits, resonators, and ground.

A narrow strip between adjacent sectors minimizes unintended capacitive cross-coupling between couplers. The SQUID provides tunability of the qubit frequency via externally applied magnetic flux, enabling frequency selection and gate implementation with high precision.

3.4.3 Qubit Control

Each qubit is equipped with two independent control lines: a flux line for frequency tuning and a drive line for state manipulation. Both types of lines are implemented as coplanar waveguides integrated on the chip.

The flux control line is terminated to ground near the SQUID loop of the transmon, creating a mutual inductance that allows magnetic flux to thread the loop when a current is applied. This flux modifies the phase across the Josephson junctions, effectively tuning the Josephson energy and, consequently, the qubit transition frequency. Flux tuning is also used to configure interaction rates and implement native Z and CZ gates.

The drive line, in contrast, is left open at the end near the qubit and capacitively couples to the qubit's charge island. Applying microwave pulses to this line enables coherent transitions between energy levels. The strength of this coupling is carefully engineered to ensure that qubit control remains efficient while minimizing radiative losses (i.e., Purcell decay) to the environment. The drive allows the implementation of arbitrary single-qubit rotations $R(\theta, \phi)$, as well as access to higher excited states of the transmon when needed.

3.4.4 Readout Mechanism

Qubit readout is performed using the dispersive regime of circuit quantum electrodynamics, wherein each qubit is coupled to a dedicated microwave resonator. This interaction leads to a qubit-state-dependent frequency shift of the resonator, known as the *dispersive shift*, which enables indirect measurement of the qubit state (for more details, refer to the bibliography [18].).

To avoid radiative energy loss (Purcell decay) through the readout resonators, each resonator is connected not directly to the external transmission line but via a dedicated Purcell filter. These bandpass filters are designed to suppress signal components near the qubit frequency while allowing efficient transmission at the resonator frequency.

All readout resonators are coupled to a common probe line, which is driven by a frequency comb that excites all resonators simultaneously (see Fig. 3.8). The transmitted signal is analyzed to extract the amplitude and phase response at each resonator frequency. By comparing these values to calibrated thresholds, the state of each qubit can be inferred with high fidelity.

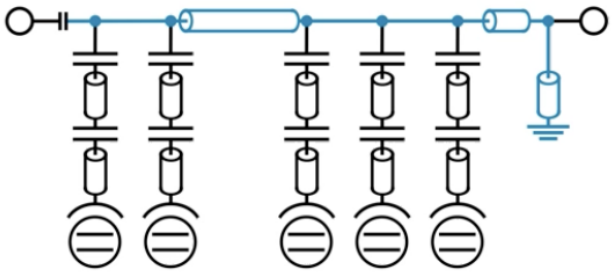


Figure 3.8: Schematic representation of the readout circuit modeled as a quasi-lumped element network. Each qubit is coupled to a dedicated readout resonator, which is in turn connected to a Purcell filter. These elements interface with a shared probe line that incorporates a distributed Purcell filtering structure. Qubits are symbolized as circles intersected by two horizontal lines [17].

3.4.5 Tunable Couplers

Tunable couplers are employed to implement high-fidelity two-qubit gates by dynamically controlling the effective interaction strength between neighboring qubits. These components are based on superconducting circuits similar in structure to transmon qubits, and are placed between data qubits to mediate interactions through a controllable coupling mechanism.

In the architecture considered, the couplers are arranged in such a way that their interaction with the qubits is mediated by waveguide extenders (see Fig. 3.9).

The strength of the effective ZZ interaction (g_{ZZ}) between any two qubits can be modulated by applying a magnetic flux through the coupler's SQUID loop. By tuning the coupler frequency appropriately, the system can span a wide range of interaction strengths, including both positive and negative values. Crucially, there exists a flux bias point where $g_{ZZ} = 0$, which is used when the quantum processor is idle to ensure minimal residual interaction.

During gate execution, the coupler is pulsed to a specific frequency using baseband square-shaped flux signals. The resulting interaction between the selected qubit pair enables the implementation of entangling gates such as CZ, iSWAP, or more general fermionic simulation gates, depending on the relative detuning between the qubit transition frequencies.

3.4.6 QPU Packaging

The chip is mounted in a copper carrier with gold-plated surfaces to enhance thermal and electrical properties. This assembly is connected to a cryogenic environment and enclosed in magnetic shielding to ensure stable and isolated operation.

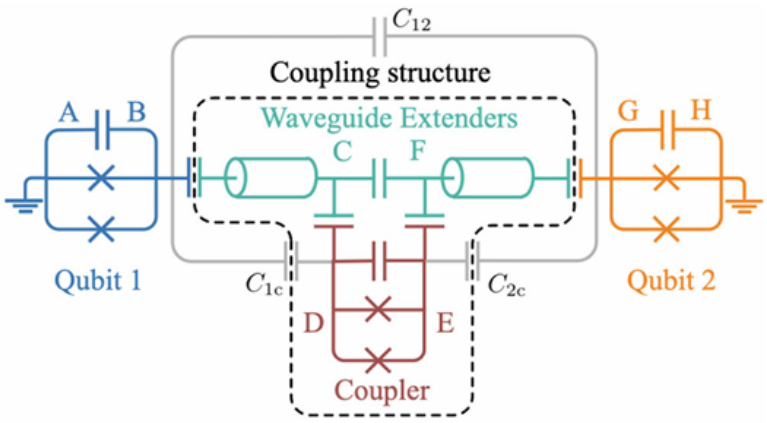


Figure 3.9: Quasi-lumped element schematic illustrating the coupling between two transmon qubits (depicted in blue and orange) mediated by a tunable coupling structure. The coupler consists of a floating qubit (shown in red) and interconnecting waveguide extenders (in turquoise). Electrical nodes are labeled with capital letters. Grey elements represent effective coupling paths introduced by the waveguide extenders [17].

3.4.7 Refrigerator

The quantum processor is operated at cryogenic temperatures using a dilution refrigerator designed to maintain millikelvin-range conditions. The QPU, mounted within its thermally anchored carrier, is affixed to the mixing chamber plate (the coldest stage of the cryostat) ensuring efficient thermal contact and temperature stabilization.

The refrigeration system is based on a combination of a pulse tube cooler and a dilution unit. The upper stages of the cryostat are pre-cooled to a few kelvin via the pulse tube, while the dilution circuit reaches base temperatures below 30 mK.

3.4.8 Signal Inputs and Outputs

All microwave signals required for controlling and reading out the qubits, including drive pulses, flux tuning, readout probes, and amplifier pumps, are routed from room temperature down to the quantum processing unit (QPU) using coaxial SCuNi cables. These cables include attenuation at various thermal stages and low-pass filters at the base temperature to ensure noise suppression.

DC signals, particularly for flux biasing, are delivered using twisted pair wiring. These lines are also filtered at room temperature, at the 3 K stage, and at the base stage to prevent thermal and electrical noise from reaching the QPU.

For readout amplification, a Traveling Wave Parametric Amplifier (TWPA) is used as the first amplification stage, providing near-quantum-limited performance.

The amplified signal is passed through superconducting NbTi coaxial cables to a High-Electron-Mobility Transistor (HEMT) amplifier at approximately 3 K. The signal is then routed through silver-plated coaxial lines to the top plate of the cryostat, where it can be digitized and processed.

3.4.9 QPU Control Electronics

The microwave control pulses for qubit operations are generated by AC-coupled Arbitrary Waveform Generators (AWGs), which operate in the qubit frequency range. The flux control pulses, necessary for tuning the tunable couplers and qubit frequencies, are generated by baseband DC-coupled AWGs.

Readout tones are generated and digitized by a quantum analyzer operating at the resonator frequencies. The electronics are designed to support fast feedback protocols, where the result of a measurement can condition a control operation within the qubit coherence time.

DC sources used for flux tuning and TWPA biasing are connected through low-pass filtered lines, which include inline resistors to ensure stable current delivery.

All control and readout equipment is housed in electronics racks, which also contain a main Linux host for control software, a Windows machine for dilution refrigerator management, various specialized Linux machines for hardware interfacing, a reference clock, power supplies, a remotely controllable power distribution unit (ePDU), and an Uninterruptible Power Supply (UPS) for stable operation.

3.4.10 Software

The software stack is organized into three main layers, each offering a different level of abstraction and control over the quantum computer.

Cortex is the high-level interface designed for end users to define and run quantum circuits. It supports circuit definitions in standard languages such as Qiskit, Cirq, and OpenQASM 2.0.

EXA is a Python-based framework used for calibration, control, and experimentation. It allows the execution of both pre-defined and custom experiments. Users can create modular experiment units that combine data acquisition, analysis, and visualization. It supports both Jupyter notebooks and standalone Python scripts.

IQM Station Control manages low-level hardware interactions. It houses device parameters and drivers, and provides an HTTP (Hypertext Transfer Protocol)-based JSON (JavaScript Object Notation) API (Application Programming Interface) that Cortex and EXA use to execute commands. This service abstracts hardware-level details from higher layers and is not typically accessed directly by users.

The following scheme allows to visualize the layers and modules of the control stack described above.

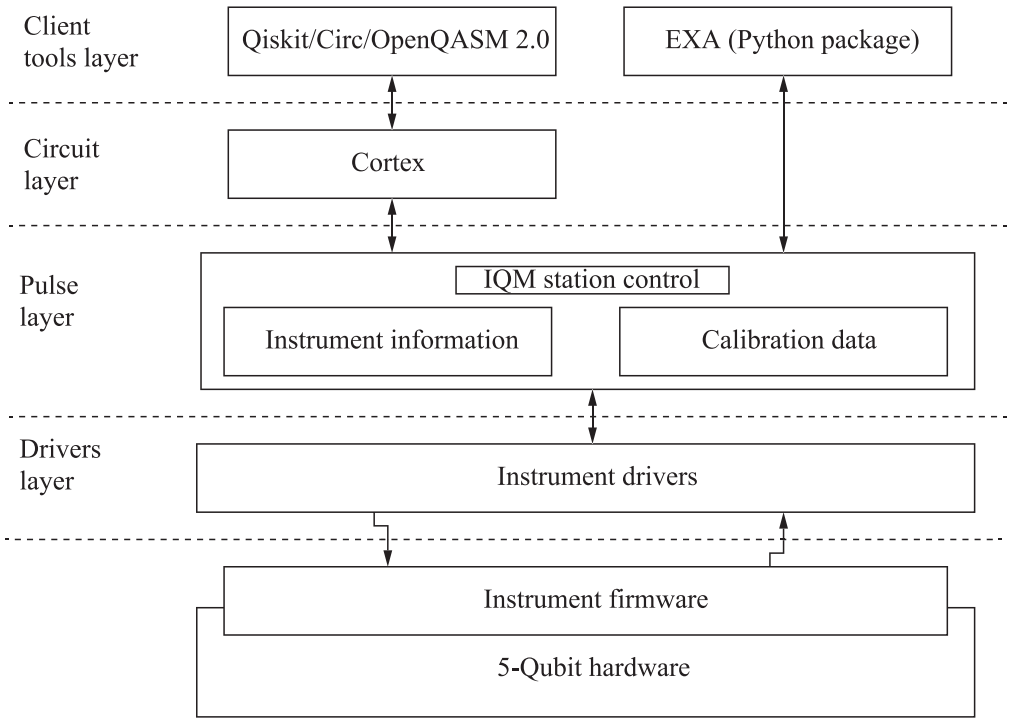


Figure 3.10: The software layers and modules of IQM Spark quantum computer control software stack (source: [17]).

Part II

Benchmarking and Experimental Validation

As a preliminary step in the benchmarking process, a series of tests on several quantum algorithms was conducted by employing the `IQMFakeAdonis()` backend, an emulated environment reproducing the noise characteristics of IQM's 5-qubit Adonis architecture. It does not aim to exactly reproduce the operational characteristics of the IQM Spark quantum processor. Instead, *IQMFakeAdonis* provides the closest available model in terms of topology and native gate set. Its use still allowed the tests to yield meaningful insights into the performance that could reasonably be expected on real hardware, thereby mitigating the uncertainty inherent in purely theoretical analyses. Operating within this emulated framework offered several advantages. First, it enabled the validation of the entire workflow, from quantum circuit compilation to execution, without the overhead and constraints associated with limited hardware availability. Second, the noise model integrated into the fake backend provided the opportunity to explore noise-aware behaviors, such as the susceptibility of certain algorithms to specific error channels. This made it possible to identify early-stage challenges related to noise sensitivity, limitations imposed by the connectivity graph, and the non-trivial decomposition of high-level quantum gates into the native gate set of the device.

The benchmarking phase focused on two distinct implementations of quantum algorithms based on the [[4,2,2]] quantum error-detecting code (see Section 2.4.4). This particular code was selected as a pragmatic compromise between fault-tolerance principles and the constraints imposed by the hardware. Specifically, the IQM Spark device features a star-shaped qubit connectivity pattern, which, combined with a relatively limited number of physical qubits, poses a non-trivial challenge for implementing more sophisticated codes. While more advanced error-correcting codes such as the [[5,1,3]] or even larger stabilizer codes offer superior error resilience and error-correction, their deployment on the IQM Spark architecture would require a number of qubits and connectivity configurations beyond the available resources.

Moreover, the benchmarking comprised implementations with physical circuits, implemented directly at the level of hardware qubits without any form of quantum error encoding. These served as reference baselines to assess algorithmic performance under native device conditions.

As a final remark, in the theoretical foundations (Section 2.6), fault-tolerant quantum computation was defined in its rigorous and scalable sense: a quantum operation is fault-tolerant if a single physical fault results in at most one error per encoded block, thereby preserving the error-correcting capability of the code and enabling arbitrarily long computations with active error correction.

In the present chapter, however, the term *fault-tolerant* is employed in the more restricted sense introduced by Gottesman for small-scale experiments [19]. According to this criterion, an implementation is considered fault-tolerant if, for a given family of circuits, the encoded version consistently achieves a lower error rate than the corresponding unencoded circuit when executed on the same hardware or backend. In practice, this is realized through error detection and post-selection: single-qubit errors that would alter the logical outcome are identified by the code

structure, and the corresponding runs are discarded. The logical results are then reconstructed from the subset of post-selected data.

It should be noted that the [[4,2,2]] code can only detect single errors, but it cannot correct them. As a result, the scheme is unable to protect against all possible faults. The results should therefore be interpreted as a *fault-tolerant demonstration in the sense of Gottesman (2016)*, and not as an implementation of universal, large-scale fault-tolerant quantum computation.

Chapter 4

Encoded Deutsch-Jozsa

4.1 Introduction

Implementing textbook quantum algorithms in a fault-tolerant manner remains a crucial benchmark on the road to scalable quantum computation. The Deutsch-Jozsa algorithm (see Section 1.1.3), due to its simplicity and reliance solely on Clifford gates, is particularly suitable for fault-tolerant experiments. In this work, I have closely followed the methodology and circuits presented by Singh and Prakash [20], where the authors demonstrate a fully fault-tolerant implementation of the Deutsch-Jozsa algorithm using the smallest error-detecting code, namely the [[4,2,2]] stabilizer code. Their work shows that, by preparing specific logical states and performing transversal operations, a statistically significant noise reduction can be achieved using existing commercial hardware without ancilla qubits.

4.2 Methods

The [[4,2,2]] code is an error-detecting code defined by the stabilizer generators $S_1 = XXXX$ and $S_2 = ZZZZ$. The logical basis states used are:

$$
\begin{aligned}
|+0\rangle &= \frac{1}{\sqrt{2}}(|0000\rangle + |1111\rangle), \\
|+1\rangle &= \frac{1}{\sqrt{2}}(|1100\rangle + |0011\rangle), \\
|-0\rangle &= \frac{1}{\sqrt{2}}(|1010\rangle + |0101\rangle), \\
|-1\rangle &= \frac{1}{\sqrt{2}}(|0110\rangle + |1001\rangle).
\end{aligned}
$$

A logical $|++\rangle$ state can be defined as:

$$|++\rangle = \frac{1}{2}(|0000\rangle + |0011\rangle + |1100\rangle + |1111\rangle) = \frac{1}{2}(|00\rangle + |11\rangle)(|00\rangle + |11\rangle) \quad (4.1)$$

and it serves as the input of this specific implementation that can be prepared using the following fault-tolerant circuit:

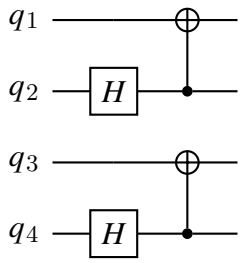


Figure 4.1: Fault tolerant preparation of the state $|{+}{+}\rangle$ in the $[[4, 2, 2]]$ error-detecting code [20].

The authors could implement a fault-tolerant simultaneous measurement of $X \otimes I$ and $I \otimes Z$ without using any ancillae by measuring each of the four physical qubits in the computational basis. If a single X error occurs on any physical qubit, the total number of measured 1's will be odd, leading to discard that particular bitstring. Conversely, a single Z error on any physical qubit, while undetectable, will not affect the measurement results and can therefore be safely disregarded.

To understand the fault tolerance of this circuit, consider that an error in one of the CNOT gates could induce a two-qubit error on one of the Bell states. Such an error can be written as a linear superposition of Pauli errors. However, by using the identity $U_1 \otimes U_2(|00\rangle + |11\rangle) = 1 \otimes U_2 U_1^T(|00\rangle + |11\rangle)$, any two-qubit Pauli error on the Bell state is effectively equivalent to a single-qubit error. It is important to note that preparing other logical states, such as $|+\rangle|0\rangle$, would generally necessitate the use of an ancilla qubit.

Regarding a fault-tolerant implementation of the Deutsch-Jozsa algorithm, Alice provides Bob with an oracle that implements one of four binary functions. The authors assume Alice commits to providing a fault-tolerant version of this oracle, utilizing a pre-specified dictionary. Bob then applies a fault-tolerant encoding of the circuit implementing the Deutsch-Jozsa algorithm to this oracle. Crucially, Bob remains unaware of Alice's chosen oracle, meaning his circuit must maintain fault tolerance regardless of which of the four oracles Alice provides. The authors also permit Alice and Bob to individually simplify their respective quantum circuits. However, the oracles and the Deutsch-Jozsa circuit must be prepared and executed independently. Therefore, the authors stipulate that no simplification should occur that combines gates from both Alice's and Bob's circuits.

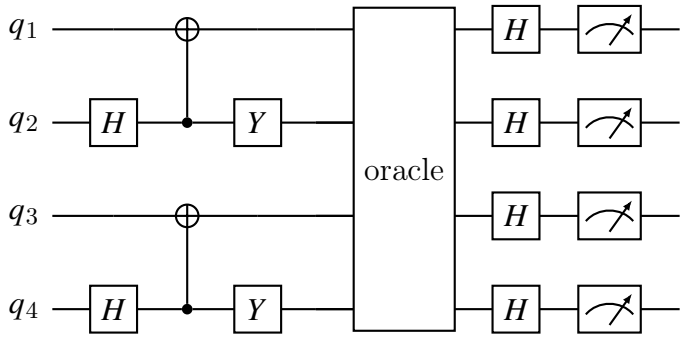


Figure 4.2: Fault-tolerant encoding of the Deutsch-Jozsa algorithm in the [[4,2,2]] code. In the last step, the authors swap the logical qubits before measurement, in order to measure X rather than Z on the logical qubit of interest [20].

Due to the limited number of transversal gates associated with the $[[4, 2, 2]]$ code, it is not immediately apparent how to implement the Deutsch-Jozsa algorithm without ancillae using this code. Nevertheless, a fault-tolerant implementation of the Deutsch-Jozsa algorithm is indeed possible. This is critically achieved by leveraging the ability to fault-tolerantly prepare the $|++\rangle$ state and measure one of the qubits in the X eigenbasis.

Each of the four possible oracles $f(x) \in \{0, x, 1 \oplus x, 1\}$ is implemented fault-tolerantly with transversal operations as indicated in Table 4.1.

The correctness of the implementation is verified through post-selection on valid codewords (even parity bitstrings), which correspond to the logical qubit outcomes. It is necessary to define a decoding step that maps the 4-bit output bitstring produced by the encoded circuit to the corresponding 2-bit output bitstring of the bare circuit. The decoding proceeds by aggregating the probabilities of the valid codewords into logical states as:

$$R_{00} = R'_{0000} + R'_{1111}, \tag{4.2}$$
$$R_{01} = R'_{1100} + R'_{0011}, \tag{4.3}$$
$$R_{10} = R'_{1010} + R'_{0101}, \tag{4.4}$$
$$R_{11} = R'_{0110} + R'_{1001}. \tag{4.5}$$

The logical measurement probabilities are then:

$$R_0 = R_{00} + R_{01}, \quad R_1 = R_{10} + R_{11}$$

In order to compare the results between bare and encoded circuits, the *statistical distance* has been used as a performance metric for small fault-tolerant experiments. This metric compares the observed probability distribution of outcomes for a given circuit with the ideal probability distribution expected for that circuit in the absence

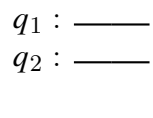
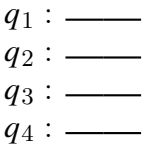
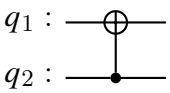
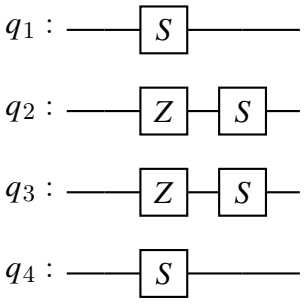
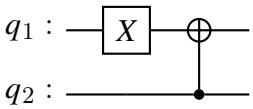
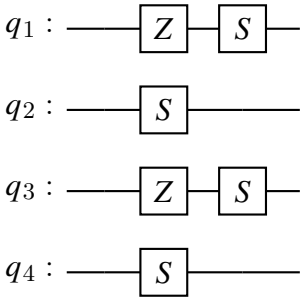
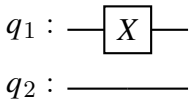
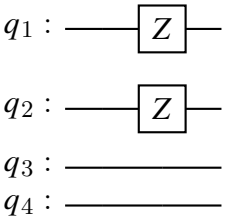
Binary Function	Bare Oracle	Encoded Oracle
$f(x) = 0$		
$f(x) = x$		
$f(x) = 1 \oplus x$		
$f(x) = 1$		

Table 4.1: Comparison between bare and encoded oracles for all the oracle functions [20].

of noise. While a complete characterization of the noise reduction associated with a fault-tolerant scheme should ideally include detailed tomography studies, for the purpose of evaluating a fault-tolerant implementation of a particular quantum algorithm, the statistical distance is the most natural performance metric to use.

Let P_i denote the theoretical probability of outcome labeled i in an ideal quantum circuit, Q_i denote the observed probability of the outcome i in the bare (non-fault-tolerant) quantum circuit, and R_i the observed probability in the encoded (fault-tolerant) circuit.

In the bare circuit implementing the Deutsch-Josza algorithm the output is

obtained by measuring the second qubit only. If the oracle implements a constant function, then $P_0 = 0$ and $P_1 = 1$. Conversely, if the oracle implements a balanced function, then $P_0 = 1$ and $P_1 = 0$. When the algorithm is executed, the measurement of the second qubit leads to obtain probabilities Q_0 and Q_1. These can be compared to the ideal probabilities P_0 and P_1 via the statistical distance, that is defined as:

$$D_{\text{bare}} = \frac{1}{2}(|P_0 - Q_0| + |P_1 - Q_1|) \tag{4.6}$$

The statistical distance for the encoded circuit is similarly defined as:

$$D_{\text{enc}} = \frac{1}{2}(|P_0 - R_0| + |P_1 - R_1|) \tag{4.7}$$

The fraction of runs that were not discarded is referred to as the *post-selection ratio*. The percentage noise reduction is calculated as:

$$\text{Percentage Noise Reduction} = \frac{(D_{\text{enc}} - D_{\text{bare}})}{D_{\text{bare}}} \times 100\% \tag{4.8}$$

As indicated in Eq. 4.8, a negative value of noise reduction denotes that the encoding procedure effectively mitigates the impact of noise, whereas a positive value implies that the encoding instead amplifies it.

The complete source code is available in Appendix 7.

4.3 Results

The results of the fault-tolerant Deutsch-Jozsa algorithm are presented in this section.

4.3.1 AerSimulator Results: The Noiseless Baseline

The analysis begins with noiseless simulations on the *AerSimulator*, which serve as a validation tool for the logical design and as a baseline against which noisy implementations can later be compared.

For each oracle, 150 independent repetitions were performed, with 1024 shots per run. Fig. 4.3 shows the outcomes for the four oracle functions: $f(x) = 0$, $f(x) = x$, $f(x) = 1 + x$, and $f(x) = 1$. Each panel reports three metrics: the bare distance (`D_bare`), the encoded distance (`D_enc`), and the postselection ratio (`Postselection`). In addition, the logical probability distributions $R_{00}, R_{01}, R_{10}, R_{11}$ are displayed.

In the noiseless case, the computation proceeds under idealized conditions: all quantum gates are applied as exact unitary operations, without over-rotations, crosstalk, or calibration errors, and measurement deterministically projects each

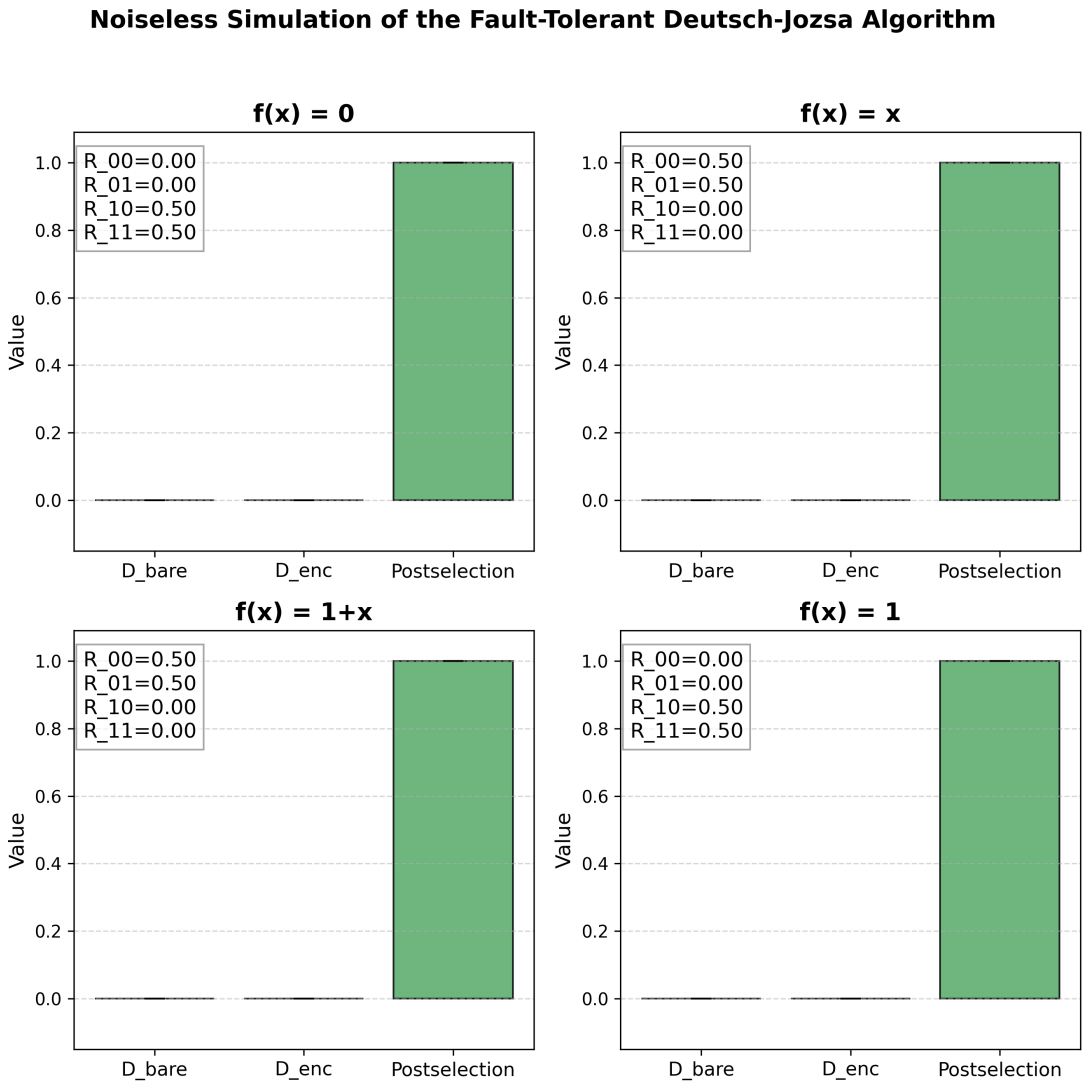


Figure 4.3: Noiseless simulation of the fault-tolerant Deutsch-Jozsa algorithm showing average `D_bare`, `D_enc`, and `post-selection ratio` over 150 repetitions per oracle, with 1024 shots per run. Note that the noiseless case is not physically meaningful, as the encoding offers no real advantage without errors to detect. It only serves as a consistency check to ensure that the overall workflow is correct.

qubit onto the correct computational basis state. As a result, the decoded distributions R_{ij} exactly match the theoretical predictions for each oracle: only the expected logical outcomes occur with nonzero probability, and spurious contributions are completely absent. This agreement confirms that the encoding and decoding pipeline is implemented correctly.

Consequently, the distance metrics (which quantify deviations between observed and target distributions) are identically zero. Any nonzero distance would indicate a fault in the logical design or in the encoding procedure. The post-selection ratio similarly reaches 1.00 for all oracle functions, confirming that the logical computation is perfectly preserved under encoding.

It is important to stress, however, that the analysis of encoded circuits in a noiseless scenario does not represent a physically meaningful setting. In the absence of noise, the additional resources introduced by the encoding provide no real benefit, since there are no errors to detect or mitigate. The noiseless case therefore serves only as a logical consistency check: it ensures that the encoding preserves the intended computation and that the decoding correctly maps the four-qubit codewords to the two-qubit logical outcomes, without introducing artificial deviations.

In summary, the noiseless simulation provides a rigorous baseline: any deviation observed on noisy simulators or real quantum hardware can be attributed to the specific noise profile of the backend, rather than to flaws in the logical design or in the implementation of the encoding scheme.

4.3.2 IQMFakeAdonis Results: Performance on a Noisy Backend

To assess the effectiveness of the fault-tolerant scheme under realistic noise conditions, the Deutsch-Jozsa algorithm was executed on the *IQMFakeAdonis* backend. For this experiment, 1000 independent repetitions were performed for each oracle, with 8192 shots per run.

Fig. 4.4 reports the average noise reduction observed across the four different oracles.

The error bars in Fig. 4.4 were obtained by combining two complementary sources of uncertainty. First, within each of the 1000 independent runs, the statistical uncertainties of the bare and encoded statistical distances were propagated to obtain the uncertainty on the noise reduction. This was achieved through explicit error propagation formulas.

Then, in order to consistently report the overall uncertainty, the average of the per-run errors was combined in quadrature with the variability of the mean values across the 1000 repetitions per oracle.

This procedure ensures that both the intrinsic shot noise (internal variance) and the run-to-run fluctuations are faithfully included in the reported error bars.

The complete source code used for data analysis is available in Appendix 7.

As already explained, a negative value of noise reduction indicates that the encoded circuit attains a smaller distance from the ideal probability distribution compared to the bare circuit, thereby demonstrating that the fault-tolerant encoding effectively mitigates noise. Conversely, positive values correspond to cases where the logical encoding introduces an overhead that outweighs its protective benefits. By analyzing Fig. 4.4, a clear distinction emerges between the constant

and balanced oracles. For the balanced oracles $f(x) = 1 + x$ and $f(x) = x$, the encoded implementation consistently yields a strong and statistically significant noise reduction, with average improvements approaching -65%. This indicates that the additional structure of the encoded circuit successfully suppresses noise in more complex oracle configurations. In contrast, the constant oracles $f(x) = 0$ and $f(x) = 1$ show values fluctuating around zero, with large uncertainties that sometimes obscure any net benefit from the encoding. These results suggest that the protective effect of the [[4,2,2]] code becomes more evident in scenarios where the oracle induces nontrivial transformations, while its advantage is less pronounced for simpler mappings.

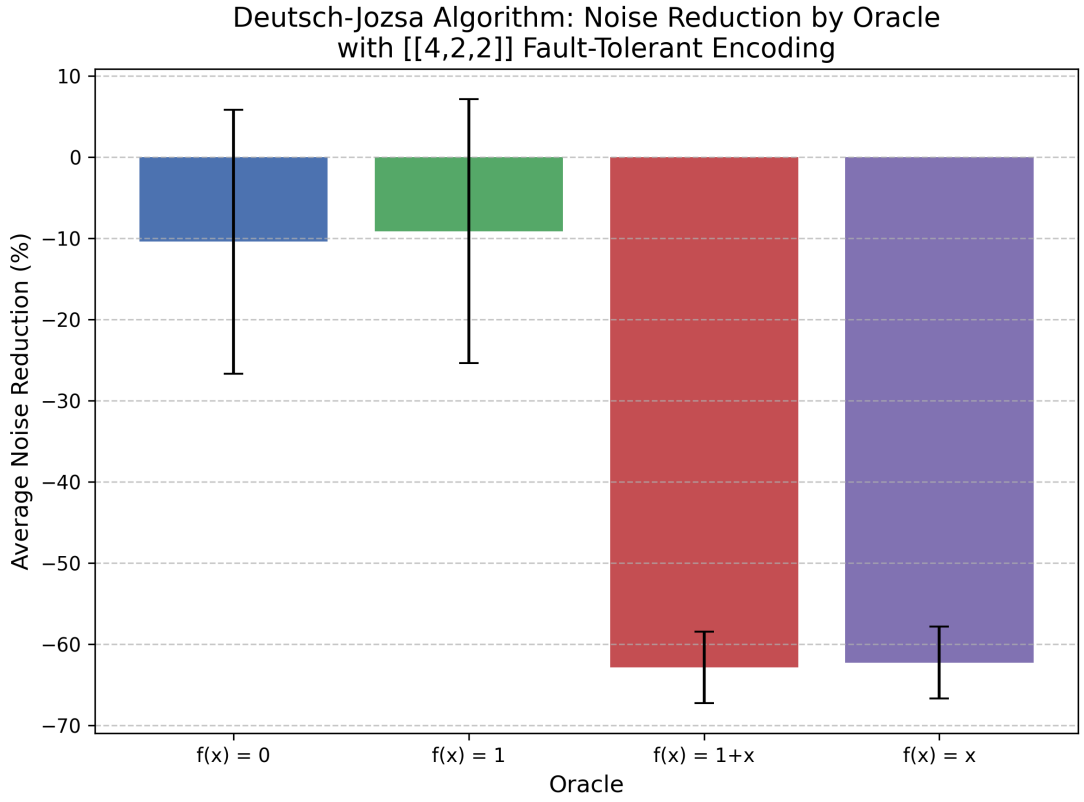


Figure 4.4: Fault-Tolerant Deutsch-Jozsa Algorithm: Average noise reduction for each oracle on the *IQMFakeAdonis* backend, computed over 1000 repetitions per oracle, with 8192 shots per run. The metric is defined as $(D_{\text{encoded}} - D_{\text{bare}})/D_{\text{bare}} \times 100$, with error bars denoting the combined uncertainty as described in the text. Negative values indicate that the encoded circuit achieves a lower error rate than the bare circuit, corresponding to a successful reduction of noise.

The average post-selection ratio from the experiments on the *IQMFakeAdonis* backend was analyzed. This metric provides insight into the rate of successful error

detection and is crucial for evaluating the practical performance of a fault-tolerant circuit. Fig. 4.5 shows the average post-selection ratio across 1000 independent repetitions for each oracle. Because the y-axis does not start at zero, the differences appear visually amplified, though they are in fact minimal.

Unlike the ideal noiseless simulation where the post-selection ratio was 1.00 (as shown in Fig. 4.3), the results from the noisy backend show a significant decrease. For the constant oracles ($f(x) = 0$ and $f(x) = 1$), the average post-selection ratio remains relatively high, around 0.80. This indicates that approximately 80% of the runs were considered successful, meaning the syndrome measurements correctly returned the "no-error" state.

However, a slight drop in the post-selection ratio is observed for the balanced oracles ($f(x) = 1+x$ and $f(x) = x$), where the ratio falls below 0.790. This difference is a direct consequence of the higher gate count and circuit depth associated with balanced oracles in this implementation. The increased number of gates leads to a higher probability of gate errors, such as bit flips or phase flips, which are detected by the syndrome measurements. When an error is detected, the run is considered a failure and is not post-selected, thus lowering the ratio.

The error bars in Fig. 4.4 correspond to the Standard Error of the Mean (SEM), calculated as $\sigma/\sqrt{N}$, where σ is the standard deviation across the runs and $N = 1000$.

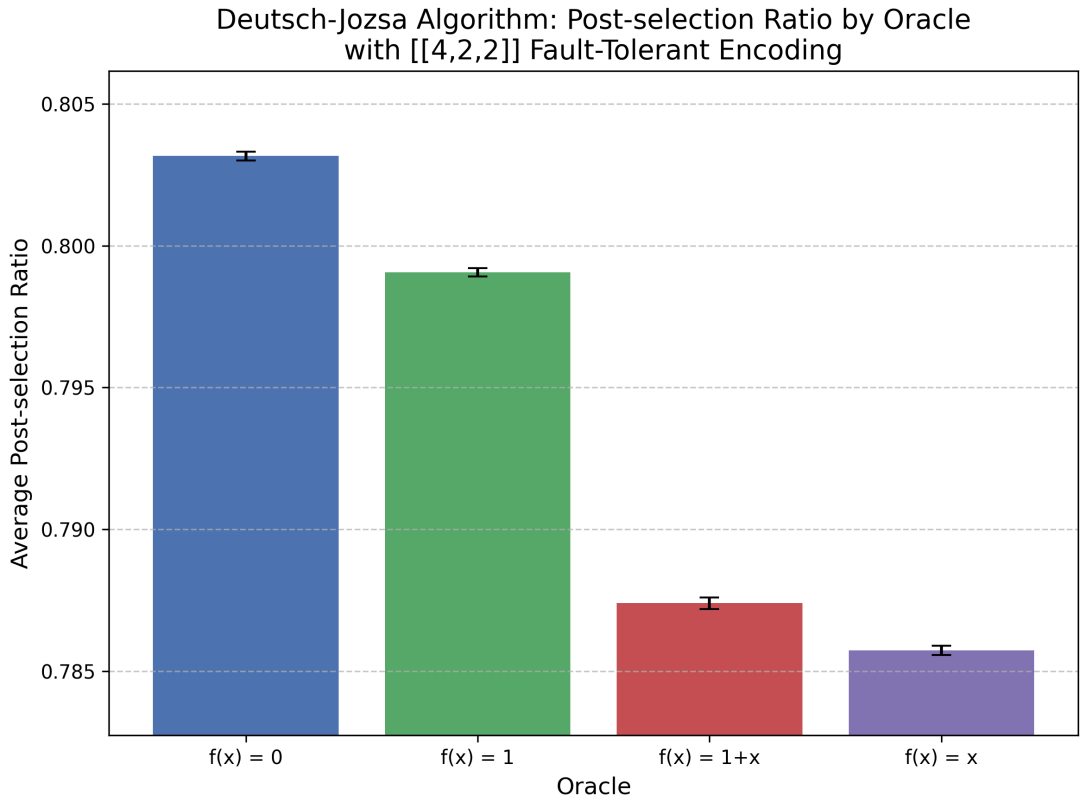


Figure 4.5: Fault-Tolerant Deutsch-Jozsa Algorithm: Average post-selection ratio for each oracle on the *IQMFakeAdonis* backend, computed across 1000 independent repetitions per oracle, with 8192 shots per run. The error bars represent the Standard Error of the Mean (SEM), calculated as $\sigma/\sqrt{N}$, where σ is the standard deviation across the repetitions and $N = 1000$. Note that the y-axis does not start from 0, which can visually exaggerate the differences among values; in reality, the observed variations are relatively small.

4.3.3 IQM Spark: Real Hardware Execution

The experiment was conducted with a high number of shots (50,000 shots per run, repeated 700 times per oracle) to reduce the impact of *shot noise*, whose magnitude scales inversely with the square root of the number of shots ($\propto 1/\sqrt{N_{shots}}$). By employing 50,000 shots, the statistical uncertainty in the estimated probabilities is significantly minimized, allowing for a more accurate and reliable determination of the average post-selection ratio and noise reduction.

Fig. 4.6 illustrates the average noise reduction for the Deutsch-Jozsa algorithm executed on real hardware. The data clearly shows that the constant oracles, $f(x) = 0$ and $f(x) = 1$, exhibit significant negative noise reduction values, with

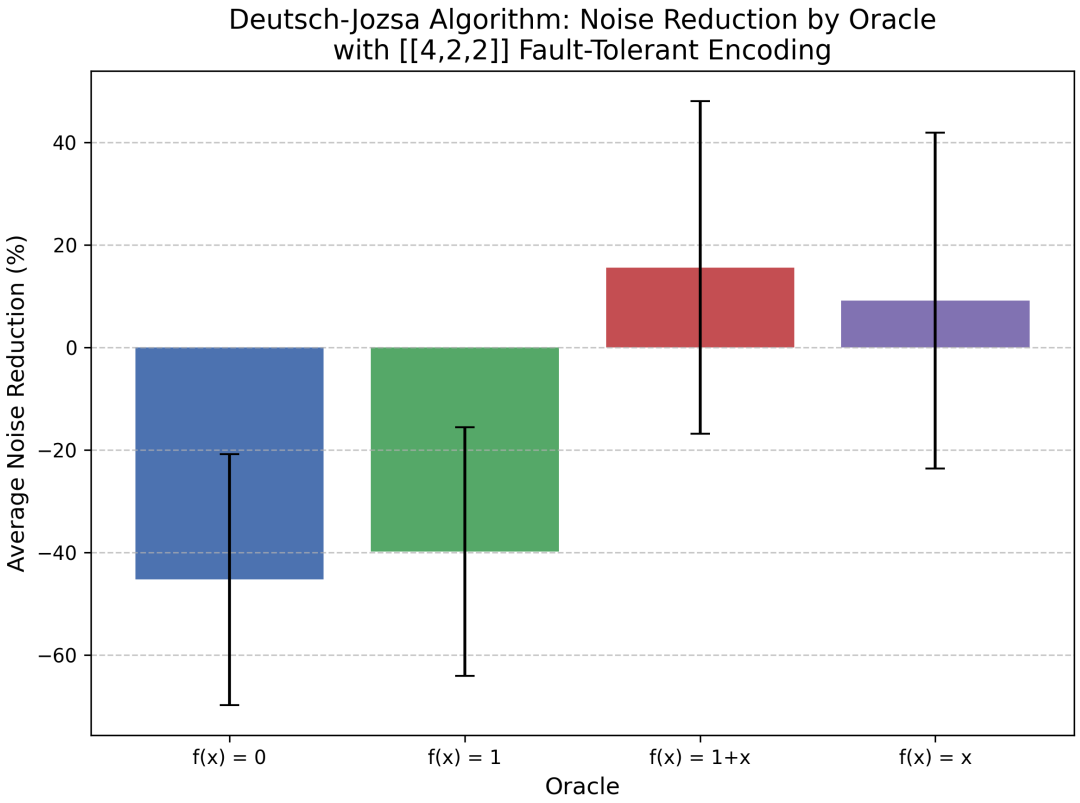


Figure 4.6: Fault-Tolerant Deutsch-Jozsa Algorithm: Average noise reduction for each oracle on the *IQM Spark* processor, computed over 700 repetitions per oracle, with 50,000 shots per run. The metric is defined as $(D_{\text{encoded}} - D_{\text{bare}})/D_{\text{bare}} \times 100$, with error bars denoting the combined uncertainty as described in the text. Negative values indicate that the encoded circuit achieves a lower error rate than the bare circuit, corresponding to a successful reduction of noise.

average noise reduction values of -45% and -40% respectively. This indicates that the encoding was highly effective in mitigating noise for these simpler circuit implementations. In contrast, the balanced oracles, $f(x) = 1 + x$ and $f(x) = x$, show positive average noise reduction values of $+17\%$ and $+10\%$ respectively. This suggests that the logical overhead associated with the more complex circuits for balanced functions outweighs the noise-mitigation benefits, leading to an overall increase in errors.

Overall, the analysis is severely constrained by the presence of large error bars. In several cases, the fluctuations exceed the absolute magnitude of the mean values, and for some oracles (e.g., $f(x) = 0$ and $f(x) = 1$) the error bars extend well beyond the corresponding averages. This indicates a statistical inconsistency: the mean values alone are not reliable indicators of the actual effect of the encoding, since

the variability across runs is of the same order or even larger than the estimated value. Consequently, although the mean values suggest that encoding may either improve or worsen performance depending on the oracle, the large uncertainties prevent any definitive conclusion.

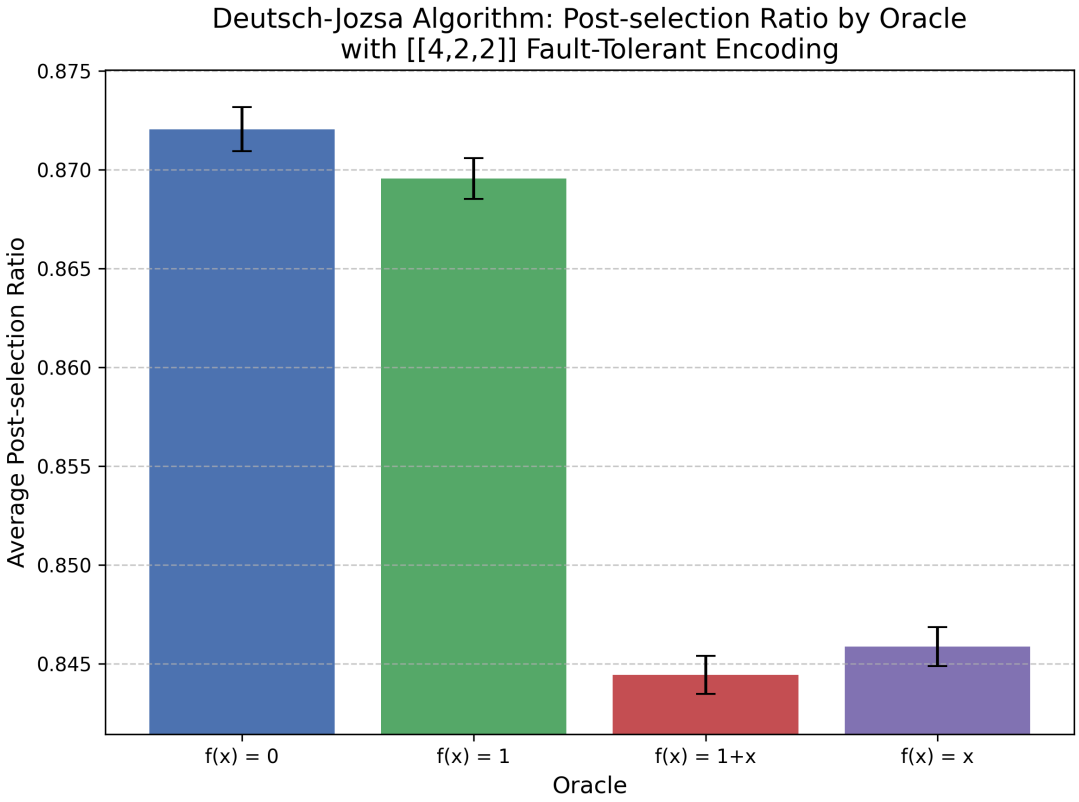


Figure 4.7: Fault-Tolerant Deutsch-Jozsa Algorithm: Average post-selection ratio for each oracle on the *IQM Spark* processor, computed across 700 independent repetitions per oracle, with 50,000 shots per run. The error bars represent the Standard Error of the Mean (SEM), calculated as $\sigma/\sqrt{N}$, where σ is the standard deviation across the repetitions and $N = 700$. Note that the y-axis does not start from 0, which can visually exaggerate the differences among values; in reality, the observed variations are relatively small.

Fig. 4.7 presents the average post-selection ratio. The y-axis does not begin at zero, which can overemphasize the differences; the actual variations are minimal. The plot shows that the constant oracles ($f(x) = 0$ and $f(x) = 1$) exhibit a significantly higher average post-selection ratio (approximately 0.872 and 0.869, respectively) compared to the balanced oracles ($f(x) = 1 + x$ and $f(x) = x$), which have ratios of approximately 0.844 and 0.846. This difference suggests that the

constant oracles result in a higher fidelity of the final state within the logical subspace, likely due to a simpler circuit implementation that introduces less noise and decoherence.

A direct comparison between the results obtained on the *IQMFakeAdonis* backend and those on the *IQM Spark* processor highlights some important discrepancies that can be attributed to the fundamental difference between a noise fake backend and a real hardware platform. On *IQMFakeAdonis*, the balanced oracles ($f(x) = 1 + x$ and $f(x) = x$) consistently demonstrated strong negative noise reduction, indicating that the encoding was effective in mitigating noise under these more complex scenarios. In contrast, the constant oracles ($f(x) = 0$ and $f(x) = 1$) showed values close to zero, with uncertainties large enough to obscure any conclusive improvement. This trend suggests that the noise model used in the fake backend captures gate errors in a way that makes the protective effect of the [[4,2,2]] code more evident for circuits with greater depth and gate count.

On the real *IQM Spark* hardware, however, the opposite behavior was observed: the constant oracles exhibited significant noise reduction, while the balanced ones often resulted in a worsening of the error rate. This inversion can be explained by the fact that real quantum devices are subject not only to gate noise, but also to additional sources of error such as crosstalk, leakage, decoherence, and time-correlated noise. The balanced oracles, which already involve deeper circuits and larger gate counts, accumulate these errors in a way that outweighs the benefits of the encoding, thereby leading to positive (worsened) noise reduction values. In contrast, the constant oracles benefit from their simpler structure, where the encoding provides a relatively greater protective effect without being overwhelmed by hardware-induced errors.

Another notable difference lies in the post-selection ratios. On *IQMFakeAdonis*, values typically ranged between 0.79 and 0.80, whereas on *IQM Spark* the average ratios were slightly higher, approaching 0.87 for the constant oracles and around 0.84 for the balanced ones. This discrepancy reflects the fact that in the fake backend, all errors prescribed by the noise model are faithfully registered as faults, whereas in real hardware, not all physical errors are detected or propagated into the logical subspace in the same way. Consequently, real devices may yield slightly inflated post-selection ratios, even though the overall logical performance can still be degraded. This subtle effect indicates that higher post-selection ratios do not necessarily translate into superior logical behavior, but instead reflect the complex interplay between error detection and the physical noise landscape.

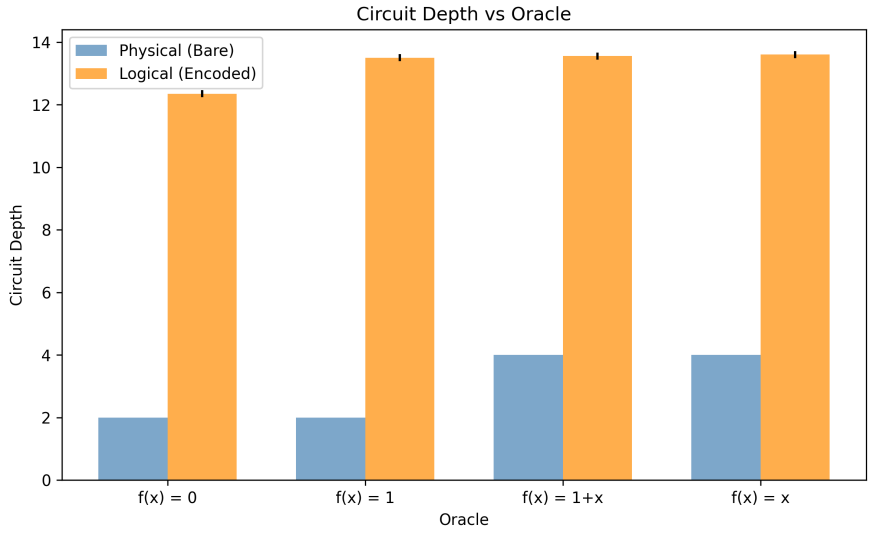
To further contextualize these observations, additional analyses were carried out focusing on *circuit depth*, *circuit size*, and the number of SWAP operations. The *circuit depth* quantifies the number of layers of gates executed sequentially, corresponding to the effective execution time and the degree of exposure to decoherence. The *circuit size*, on the other hand, counts the total number of operations applied, regardless of their parallelizability, and thus provides a measure of the accumulation of raw errors due to gate imperfections. While both metrics are related, they capture different aspects of circuit complexity: a circuit may have a

large size but relatively shallow depth if many gates can be executed in parallel, or conversely a modest size but high depth if gates must be performed sequentially. Including these analyses enables a more nuanced understanding of why certain oracle implementations exhibit better or worse performance across different platforms, as they reveal the structural differences in resource requirements that amplify or mitigate the impact of noise.

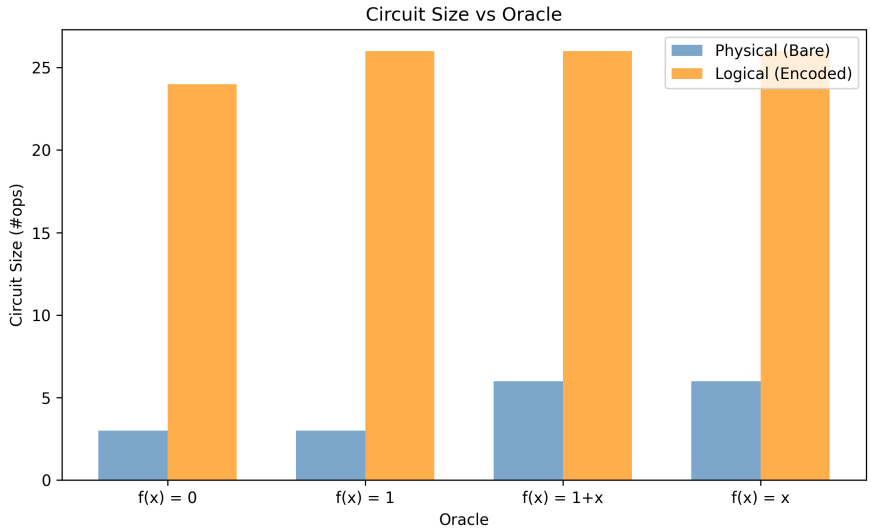
The data presented in Fig. 4.8 provides direct evidence of the resource overhead introduced by the fault-tolerant encoding. As shown in Fig. 4.8(a), the circuit depth for all logical (encoded) circuits is considerably larger than their unencoded counterparts. For constant oracles, the logical circuit depth is approximately 12, a substantial increase from the bare physical depth of around 2. The depth for balanced oracles exhibits a more pronounced increase, rising from a physical depth of approximately 4 to a logical depth of about 14. This notable increase in depth correlates with a prolonged circuit execution, thereby exacerbating the impact of decoherence and time-dependent noise on qubit state fidelity.

Similarly, the circuit size, as seen in Fig. 4.8(b), follows a comparable trend. The logical encoding introduces a significant overhead in the total number of gate operations. The bare circuits for constant oracles have a size of roughly 3, which expands to over 24 for their logical equivalents. For balanced oracles, the size increases from about 6 gates to over 25. This increase in the total number of operations leads to a higher accumulation of raw errors due to the imperfect fidelity of each gate, which further substantiates the performance disparities observed between the constant and balanced oracles in terms of noise reduction and post-selection ratio. The inherent complexity of the balanced oracles necessitates greater depth and size in both their bare and encoded forms, and this substantial logical overhead serves as a primary driver for their degraded performance relative to the simpler constant oracles.

Furthermore, the number of SWAP operations was evaluated for each oracle implementation and found to be zero in all cases. This absence of SWAP gates ensures that the observed trends are not influenced by qubit connectivity constraints.



(a) Circuit Depth vs Oracle



(b) Circuit Size vs Oracle

Figure 4.8: Additional circuit metrics for the fault-tolerant Deutsch-Jozsa algorithm on *IQMSpark* processor. (a) Circuit depth, representing the sequential layers of operations. (b) Circuit size, indicating the total number of applied gates. Together, these metrics provide complementary insights into the structural complexity of the circuits and their susceptibility to noise.

Chapter 5

Encoded VQE for AIM

5.1 Introduction

Studying the Anderson Impurity Model (AIM) with Variational Quantum Eigensolver (VQE) (see Section 1.1.3) in combination with logical encoding techniques, such as the $[\![4,2,2]\!]$ code, provides a meaningful opportunity to explore the practical benefits of quantum error detection in a physically relevant setting. While many fault-tolerant demonstrations are limited to abstract benchmarks or toy problems, applying encoded quantum algorithms to a well-known model from condensed matter physics offers deeper insight into how error detection affects both the expressibility and stability of variational circuits. The AIM involves nontrivial correlations that are challenging to capture classically, and studying its simulation under realistic noise with logical qubits allows to move beyond simple gate-level analysis toward more application-driven evaluations.

The Anderson Impurity Model (AIM) is a paradigmatic model for describing a small number of interacting electronic degrees of freedom (the *impurity*) embedded in a non-interacting environment (the *bath*). It plays a central role in the study of strongly correlated electron systems, including magnetic impurities in metals, quantum dots, and as a key component in the Dynamical Mean Field Theory (DMFT) [21].

The basic AIM consists of three parts: the impurity (or dot), which features a local Coulomb interaction; the bath, representing a continuum of non-interacting fermionic states; and a hybridization term that couples the impurity to the bath.

Historically introduced by Anderson to model magnetic impurities in metals [22], the AIM Hamiltonian in its canonical second-quantized form is given by:

$$H = \sum_{k,\sigma} \epsilon_k c^\dagger_{k\sigma} c_{k\sigma} + \sum_{\sigma} \epsilon_\sigma d^\dagger_\sigma d_\sigma + U d^\dagger_\uparrow d_\uparrow d^\dagger_\downarrow d_\downarrow + \sum_{k,\sigma} V_k (d^\dagger_\sigma c_{k\sigma} + c^\dagger_{k\sigma} d_\sigma), \tag{5.1}$$

where $c_{k\sigma}$ and d_σ denote the annihilation operators for conduction and impurity electrons, respectively; k is the wavevector index for the conduction electrons,

and σ is the spin. The parameters ϵ_k and ϵ_σ correspond to the energy levels of conduction and impurity electrons, U is the on-site Coulomb repulsion at the impurity, and V_k is the hybridization strength.

This model admits several physically distinct regimes depending on the relative position of the impurity energy level ϵ_d and the Fermi energy E_F:

- **Empty orbital regime:** $\epsilon_d \gg E_F$ or $\epsilon_d + U \gg E_F$, where the impurity site remains unoccupied.
- **Intermediate valence regime:** $\epsilon_d \approx E_F$ or $\epsilon_d + U \approx E_F$, where charge fluctuations are significant.
- **Local moment regime:** $\epsilon_d \ll E_F \ll \epsilon_d + U$, where the impurity hosts a magnetic moment.

In the local moment regime, the system exhibits the Kondo effect: at sufficiently low temperature, the impurity spin becomes screened by the conduction electrons, forming a non-magnetic singlet many-body state [23, 24].

In the experimental demonstration reported in [25], a simplified single-impurity Anderson model (SIAM) has been used as a prototype for simulating materials systems. The Hamiltonian of the SIAM in its fermionic form reads:

$$H_{\text{SIAM}} = h \sum_{\sigma=\uparrow,\downarrow} c^\dagger_{I\sigma} c_{I\sigma} + U\, c^\dagger_{I\uparrow} c_{I\uparrow} c^\dagger_{I\downarrow} c_{I\downarrow} + \epsilon \sum_{\sigma=\uparrow,\downarrow} c^\dagger_{B\sigma} c_{B\sigma} + V \sum_{\sigma=\uparrow,\downarrow} \left(c^\dagger_{I\sigma} c_{B\sigma} + \text{h.c.} \right) \quad (5.2)$$

where $c^\dagger_{i\sigma}$ ($c_{i\sigma}$) are fermionic creation (annihilation) operators for an electron with spin σ on site i, which can be either impurity ($i = I$) or bath ($i = B$). This Hamiltonian requires four qubits to be simulated on a quantum computer in its general form.

At half-filling, the model is simplified by taking the chemical potential $h = -U/2$ and the bath energy $\epsilon = 0$, where the system still retains important features such as the metal-insulator transition. To make the model suitable for quantum simulation, a mapping to qubit operators is necessary. Ordering the fermionic orbitals as $(I\uparrow, B\uparrow, I\downarrow, B\downarrow) \rightarrow (0,1,2,3)$, and applying the Bravyi-Kitaev transformation [26], the Hamiltonian becomes:

$$H_{\text{BK}} = \frac{U}{4}(Z_0 Z_2 - 1) + \frac{V}{2}(X_0 - X_0 Z_1 - Z_1 X_2 Z_3 + X_2). \quad (5.3)$$

Here, X_i and Z_i denote the standard Pauli operators acting on qubit i. The relevant subspace supporting the ground state at half-filling is spanned by the four fermionic basis states $\{|0101\rangle, |0110\rangle, |1001\rangle, |1010\rangle\}$, which map under the Bravyi-Kitaev transformation to $\{|0111\rangle, |0110\rangle, |0011\rangle, |0010\rangle\}$, all having the form $|0 z_2 1 z_0\rangle$. Therefore, the state of qubits 1 and 3 is fixed, and the Hamiltonian effectively reduces to a two-qubit model acting on qubits 0 and 2, defined by the Coulomb interaction U and hybridization strength V.

This leads to a reduced Hamiltonian of the form:

$$H(U,V) = \frac{U}{4}(Z_0Z_2 - 1) + V(X_0 + X_2), \tag{5.4}$$

which acts on the reduced basis $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$, where $|z_2z_0\rangle$ defines the logical qubit states. This qubit-reduction technique, although inspired by similar methods applied to molecular systems such as H_2 [27], is a novel application in the context of the SIAM.

Each term in Eq. 5.4 can be measured using the [[4,2,2]] code, as both Z and X basis measurements are available in the logical gateset.

5.2 Methods

My implementation is inspired by the methodology presented in [25], where the authors propose a restricted two-parameter Hamiltonian Varational Ansatz of the form:

$$U(\alpha,\beta) = e^{-i\beta Z_0Z_2/2} \cdot e^{-i\alpha X_0/2}, \tag{5.5}$$

acting on the entangled Bell state $|\phi^+\rangle = (|00\rangle + |11\rangle)/\sqrt{2}$. This ansatz is expressive enough to approximate the ground state of the SIAM Hamiltonian and can be trained using classical optimization.

Following the authors' implementation, the unencoded ansatz circuit begins with a Bell state preparation, followed by sequential application of $RX(\alpha)$ and $ZZ(\beta)$ interactions. The complete circuit is given in Fig. 5.1.

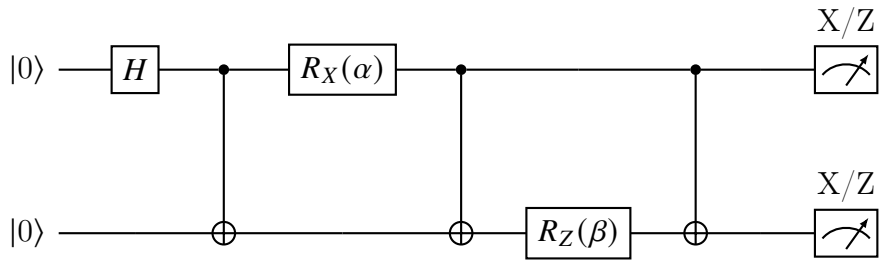


Figure 5.1: Physical (unencoded) ground state preparation circuit for Anderson Impurity Model, using a restricted Hamiltonian Variational Ansatz [25].

To increase resilience to noise, the authors have adopted the [[4,2,2]] error-detecting code, which encodes two logical qubits in four data qubits and uses ancilla-mediated gates for non-transversal operations.

The full encoded circuit is given in Fig. 5.2.

The encoded circuit is structured to accomplish these tasks:

- The **first segment** prepares fault-tolerantly the logical Bell state;

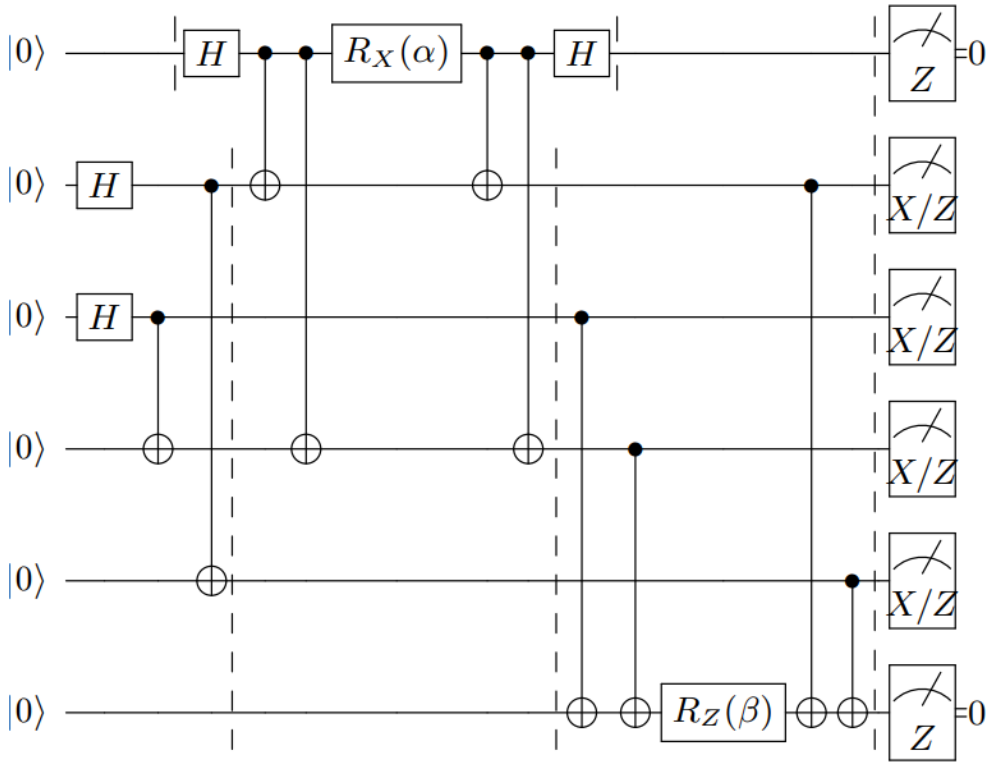


Figure 5.2: Logical (encoded) ground state preparation circuit for Anderson Impurity Model, using a restricted Hamiltonian Variational Ansatz. Though not fully fault tolerant, the use of ancillary flag qubits enables us to detect if a single-qubit error occurs nearly anywhere in the circuit [25].

- The **second segment** applies a mediated $RX(\alpha)$ rotation through the top ancilla;
- The **third segment** applies a mediated $ZZ(\beta)$ interaction, together with a parity-check-like structure acting as an implicit measurement of the $ZZZZ$ stabilizer;
- The **final segment** performs measurements: the ancillae are measured in the Z basis; the data qubits are measured in either the Z or X basis.

The post-selection step is implemented as follows:

- If the top ancilla measures $|1\rangle$ after $RX(\alpha)$, a phase error is suspected and the shot is discarded;

- If the bottom ancilla ancilla measures $|1\rangle$ after $ZZ(\beta)$, an X-type error on any data qubit (or the ancilla itself) is likely, and again the shot is discarded.

Although this is not a fully fault-tolerant scheme, it detects most single-qubit errors, with only a few remaining undetectable:

1. X errors immediately before/after the $RX(\alpha)$ gate on the ancilla, mimicking a shift in α;

2. Z errors near the $R_Z(\beta)$ gate on the ancilla, interpreted as a shift in β.

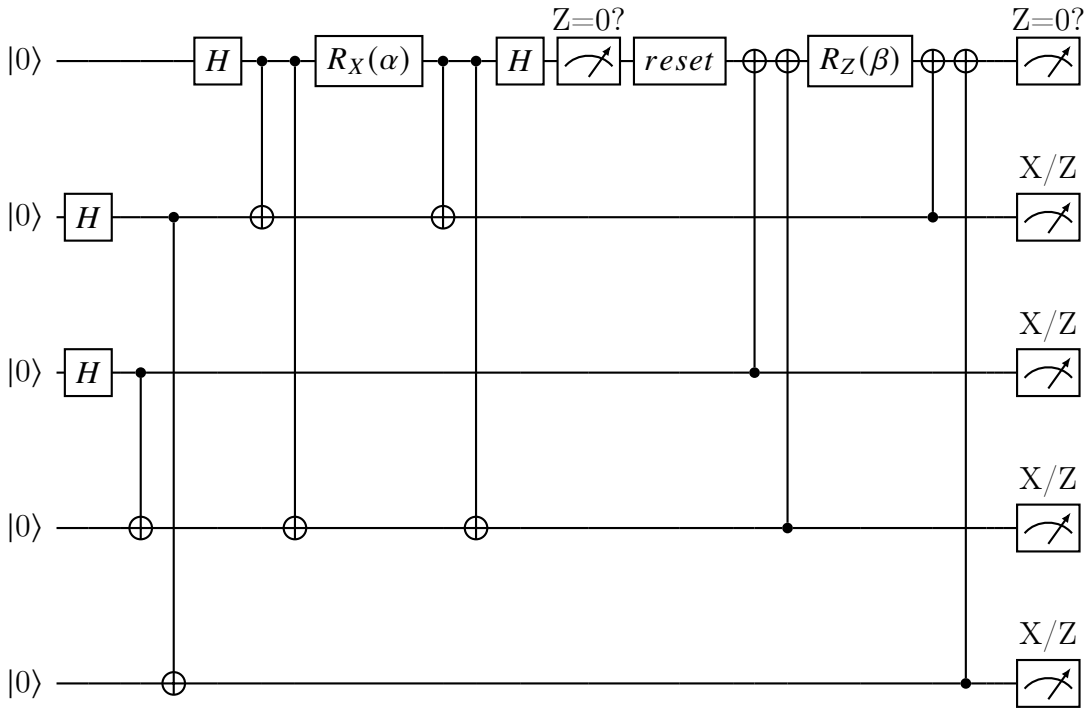


Figure 5.3: Modified ground state preparation encoded circuit for Anderson Impurity Model adapted for the IQM Spark processor. The design leverages mid-circuit measurement and qubit reset to reuse a single ancilla qubit, enabling implementation within the hardware's 5-qubit constraint.

The logic and structural design of the encoded ansatz illustrated in Figure 5.2 were faithfully preserved. However, in order to accommodate the practical constraints imposed by the available hardware (specifically, a quantum device limited to only 5 qubits), significant adaptations to the original circuit implementation were necessary. The most notable modification involves the reuse of a single ancilla qubit through the application of mid-circuit measurement followed by qubit reset operations.

Another important distinction lies in the software framework utilized for the circuit implementation. Whereas the original authors employed `CUDA-Q` for their code development, `Qiskit` was adopted in this case. This choice was primarily motivated by the need to ensure compatibility with the IQM Spark environment, which is the target platform for subsequent experimentation stages. Additionally, `Qiskit` offers robust community support and extensive documentation, facilitating reproducibility and future scalability of the work.

In light of these adaptations, the version of the encoded circuit tailored for execution on the IQM Spark processor is depicted in Fig. 5.3.

For the classical optimization subroutine, *COBYLA* (Constrained Optimization BY Linear Approximations) was employed, a gradient-free algorithm particularly suitable for problems with nonlinear constraints. COBYLA iteratively approximates the objective function and constraints using linear models constructed from function evaluations at nearby points. At each step, it solves a trust-region subproblem of the form:

$$\min_{\mathbf{x}\in\mathbb{R}^n} \hat{f}(\mathbf{x}) \quad \text{subject to} \quad \hat{c}_i(\mathbf{x}) \geq 0 \quad \text{for } i = 1, \ldots, m, \quad \|\mathbf{x} - \mathbf{x}_k\| \leq \rho_k, \tag{5.6}$$

where $\hat{f}(\mathbf{x})$ and $\hat{c}_i(\mathbf{x})$ are linear approximations of the objective and constraint functions around the current point $\mathbf{x}_k$, and ρ_k is the trust-region radius. The method does not rely on gradient information, making it especially useful when derivatives are unavailable or unreliable, as often occurs in quantum variational algorithms.

5.3 Results

The results of the not fully fault-tolerant implementation of the AIM via VQE are presented in this section.

5.3.1 AerSimulator Results: The Noiseless Baseline

Before analyzing the performance of AIM circuits under noisy conditions or on real hardware, it is essential to establish a noiseless baseline. To this end, Qiskit's *AerSimulator* was used to validate the correctness of the circuit logic and the post-processing pipeline in the absence of hardware noise.

Exactly as in the previous implementation of the Deutsch-Jozsa algorithm, it is worth stressing that analyzing encoded circuits in a noiseless scenario does not provide any physically relevant insight. In the absence of errors, the additional overhead introduced by encoding has no practical advantage, since there are no faults to detect or mitigate. The noiseless simulations are therefore only a consistency check: they confirm that the encoding correctly preserves the intended computation and that the decoding maps the four-qubit codewords back to the two logical qubits without introducing unwanted effects. In other words, this

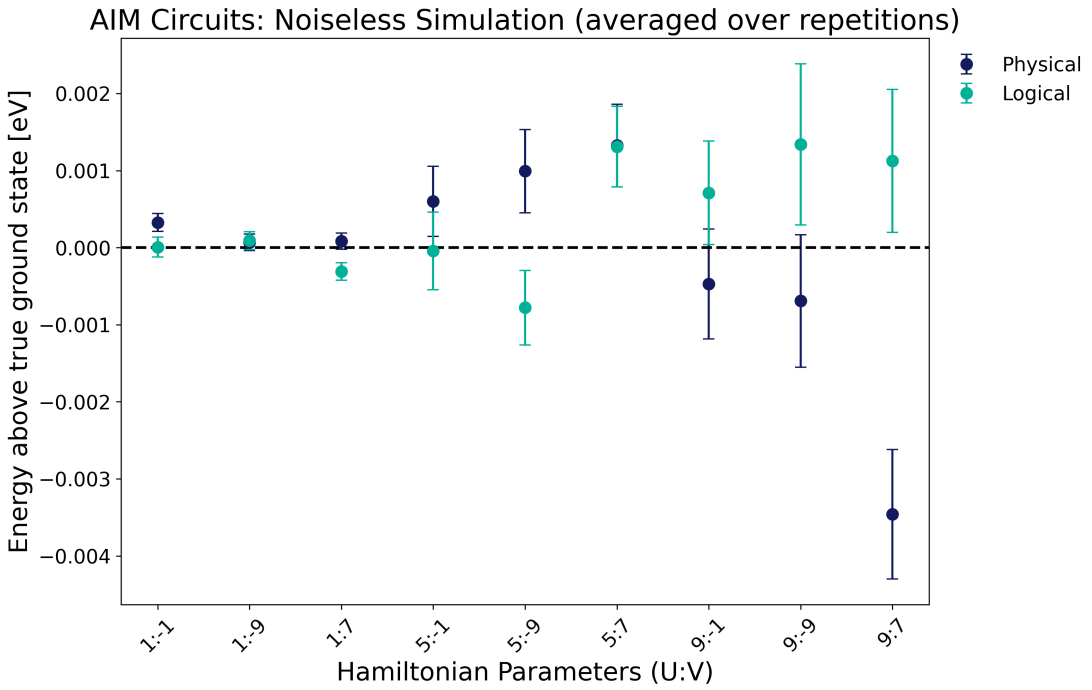


Figure 5.4: AIM Circuits: Noiseless simulation results averaged over 200 repetitions per parameter setting ($U : V$). Each circuit execution used 1024 shots. The black dashed line denotes the brute-force ground state energy. Error bars correspond to the Standard Error of the Mean (SEM), computed as the sample standard deviation divided by $\sqrt{200}$. Note that the y-axis scale has been zoomed, which visually exaggerates the differences between the physical and logical energies; in reality, these differences are extremely small, appearing only in the decimal places.

step ensures that any deviations observed in later experiments can be confidently attributed to physical noise sources, rather than structural issues in the code or encoding strategy.

Fig. 5.4 shows the average energy with respect to the exact ground state for both the *physical* and *logical* AIM circuits. Each data point represents the average over 200 independent repetitions per layer ($U : V$), with each circuit executed using 1024 shots. The error bars correspond to the Standard Error of the Mean (SEM), calculated by dividing the sample standard deviation by $\sqrt{200}$. It should be noted that the y-axis has been rescaled, which makes the gap between physical and logical energies appear larger than it actually is. In practice, these discrepancies are minimal and only become visible at the level of decimal precision.

The noiseless simulations confirm that both circuit families reproduce the ground state energy with high accuracy. The key observations from the figure are as follows:

- For small Hamiltonian parameters ($U : V$), both the physical and logical circuits yield energies very close to the exact solution, with negligible deviations (within 10^{-3} eV) and very small error bars. This demonstrates the stability of the circuits under these conditions.
- As the parameter values increase, both the energy deviation from the true ground state and the variance increase significantly. This effect is visible in the growing length of the error bars for both the physical and logical circuits. This increase in the error bars is expected. For larger values of $U : V$, the Hamiltonian becomes more complex, requiring the VQE algorithm to explore a more intricate energy landscape. This leads to a greater statistical variance in the measured energy values for a fixed number of shots (1024 in this case), as the circuit's final state becomes more sensitive to small changes in the variational parameters.

Overall, these noiseless simulations provide crucial validation of the methodology. Since no systematic deviations from the brute-force solution are observed, the implemented circuits and decoding logic are confirmed to be structurally correct. This provides a solid baseline for interpreting results under noisy conditions or on real quantum hardware.

5.3.2 IQMFakeAdonis Results: Performance on a Noisy Backend

The performance of the AIM circuits on the *IQMFakeAdonis* backend is illustrated in Fig. 5.5, where the energy deviations from the true ground state are presented for both the physical and logical circuit implementations. The data for each Hamiltonian parameter setting ($U : V$) are averaged over 200 independent repetitions, with each run consisting of 8192 shots. The error bars represent the Standard Error of the Mean (SEM), calculated as $\sigma/\sqrt{200}$, where σ is the standard deviation of the energy deviations across the repetitions.

The logical circuit, utilizing a partially fault-tolerant implementation with the [[4,2,2]] quantum error-detecting code, consistently achieves lower energy deviations compared to the bare physical circuit. This outcome aligns with the theoretical expectation that the logical encoding should mitigate the effects of noise present on the quantum hardware. A lower energy deviation indicates a closer approximation to the true ground state, signifying improved performance.

Specifically, the results demonstrate that for all tested parameter settings, the logical circuit outperforms the physical circuit. The magnitude of the performance gain varies depending on the Hamiltonian parameters, with some settings showing a more significant reduction in energy deviation than others. This empirical evidence validates the effectiveness of the [[4,2,2]] code in this context, highlighting the potential of quantum error-detecting codes to enhance the robustness of variational quantum algorithms like VQE in the presence of noise. This marks a crucial step toward achieving practical, fault-tolerant quantum computation.

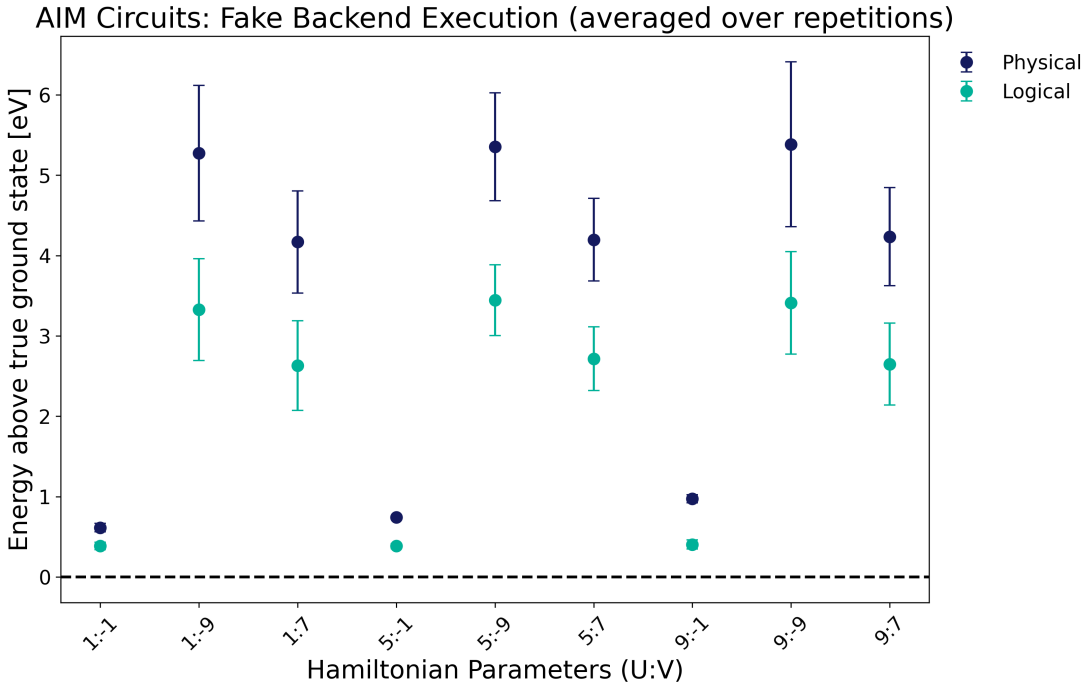


Figure 5.5: AIM Circuits: Noisy *IQMFakeAdonis* backend results averaged over 200 independent repetitions per parameter setting $(U : V)$. Each circuit execution used 8192 shots. The black dashed line denotes the brute-force ground state energy. Error bars correspond to the Standard Error of the Mean (SEM), computed as the sample standard deviation divided by $\sqrt{200}$.

5.3.3 IQM Spark: Real Hardware Execution

The performance of the AIM circuits on the real *IQM Spark* hardware is shown in Figure 5.6. For this set of experiments, 150 independent repetitions were run for each Hamiltonian parameter setting $(U : V)$, with a high shot count of 50,000 per run. This large number of shots was chosen to minimize statistical sampling noise (shot noise) and to obtain a more precise estimate of the energy expectation values, ensuring that the measurements reflect the system's behavior rather than stochastic fluctuations. The error bars represent the Standard Error of the Mean (SEM), calculated as $\sigma/\sqrt{150}$, where σ is the standard deviation of the measured energies across the 150 repetitions.

In stark contrast to the results from the simulated backend, the execution on real hardware shows a significant deviation from theoretical expectations. The logical circuit not only fails to outperform the bare physical circuit but exhibits a substantial degradation in performance. This is evidenced by the significantly higher average energies and larger data dispersion, as shown by the much wider

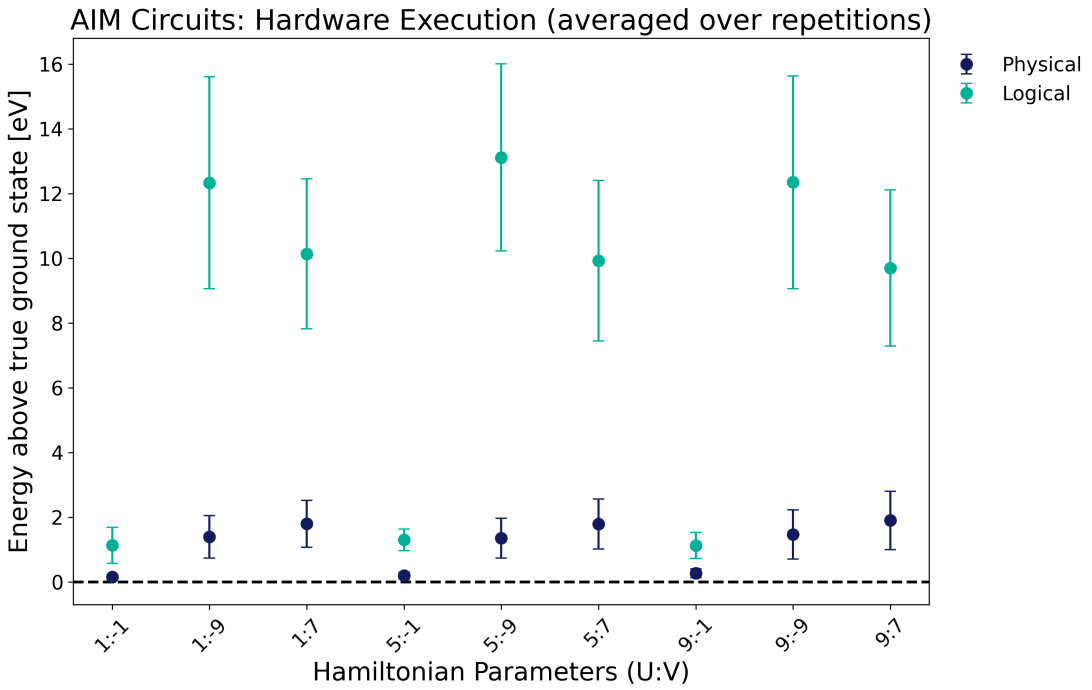


Figure 5.6: AIM Circuits: Real Hardware Execution averaged over 150 independent repetitions per parameter setting $(U : V)$. Each circuit execution used 50,000 shots. The black dashed line denotes the brute-force ground state energy. Error bars correspond to the Standard Error of the Mean (SEM), computed as the sample standard deviation divided by $\sqrt{150}$.

error bars for the logical circuit.

This failure can be attributed to several factors. Primarily, this implementation is not fully fault-tolerant. The [[4,2,2]] code is an error-detecting code, not a error-correcting code, and crucially, as noted in the methodology section, not all errors are detectable by this specific implementation. Specifically, single-qubit errors such as X errors immediately before or after the $RX(\alpha)$ gate on the ancilla, or Z errors near the $R_Z(\beta)$ gate on the ancilla, may remain undetected and propagate. Additionally, the experimental implementation of the logical circuit relies on mid-circuit measurements and qubit resets, which represent advanced operations that are not yet fully optimized or robust on current quantum hardware. These operations typically exhibit slower execution times and higher error rates compared to standard gate operations. In particular, the duration of a reset can approach or even exceed the coherence time of the qubits, potentially introducing decoherence before subsequent gates are applied. As a result, any imperfections in these intermediate measurements or reset operations can generate additional errors that propagate throughout the circuit, thereby compounding the challenge

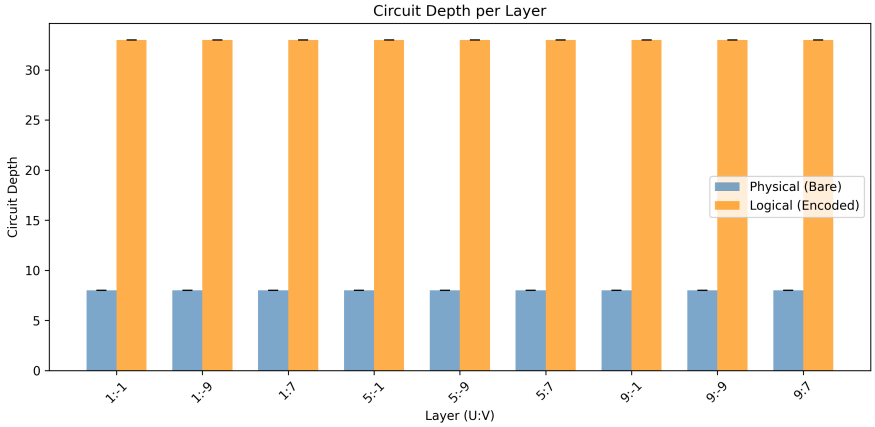
for the limited error-detecting capabilities of the employed code. This limitation underscores the current immaturity of hardware-level support for fault-tolerant protocols that rely on real-time mid-circuit control.

Ultimately, the overhead introduced by the logical encoding (in terms of increased circuit depth and qubit count) amplifies the effects of noise on the device to a greater extent than the code can mitigate, leading to the observed output.

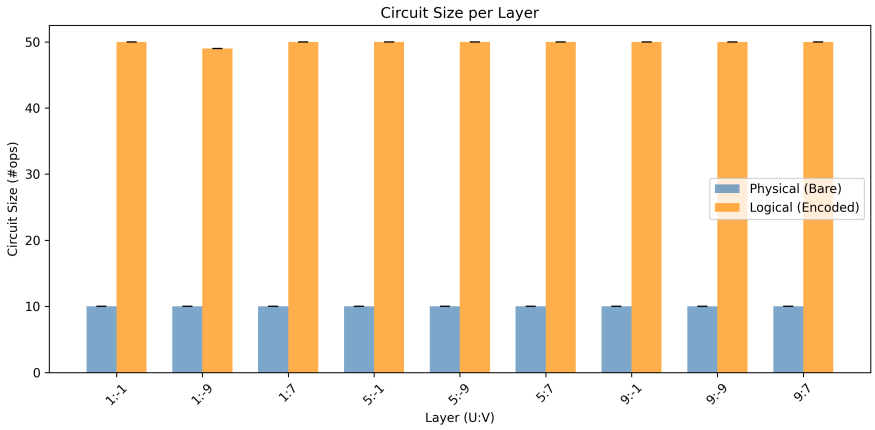
As already done for the previous implementation, in order to complement these observations, additional structural metrics of the AIM circuits were investigated, focusing on *circuit depth*, *circuit size*, and the number of SWAP operations. The *circuit depth* measures the number of sequential layers of gates and thus reflects the effective execution time of the circuit and its exposure to decoherence. The *circuit size*, in contrast, quantifies the total number of gates applied, regardless of their parallelizability, and therefore provides a measure of the cumulative error contribution from imperfect operations. Although related, these two quantities capture different aspects of circuit complexity: a circuit can exhibit a relatively small depth but large size if many gates can be executed in parallel, or conversely, a modest size but large depth if operations must be applied sequentially. Including these analyses allows for a more detailed understanding of why logical and physical circuits behave differently across noise models and real hardware, as they reveal how resource overheads in the encoded implementation exacerbate or mitigate the impact of noise.

The circuit metrics for the AIM via VQE implementation, as depicted in Fig. 5.7, highlight a consistent and substantial overhead introduced by the logical encoding across all circuit layers. As shown in Fig. 5.7(a), the circuit depth for the logical (encoded) circuits is consistently around 33, a significant increase compared to the physical (bare) circuit depth, which remains stable at approximately 8. This substantial increase in depth signifies a longer execution time, which directly correlates with a greater exposure to decoherence and other time-dependent noise processes, thereby degrading the circuit's fidelity. Similarly, in Fig. 5.7(b), the bare circuits have a consistent size of about 10 operations per layer, whereas the logical circuits exhibit a five-fold increase to around 50 operations per layer. This higher gate count directly contributes to the accumulation of raw errors due to the imperfect fidelity of each individual gate.

Finally, the absence of SWAP operations in all cases ensures that the observed behavior is not biased by qubit connectivity constraints.



(a) Circuit Depth vs Layer $(U : V)$



(b) Circuit Size vs Layer $(U : V)$

Figure 5.7: Additional circuit metrics for the AIM via VQE implementation on *IQM Spark* processor. (a) Circuit depth, representing the number of sequential layers of gates and exposure to decoherence. (b) Circuit size, corresponding to the total number of operations applied and the cumulative error contribution. These complementary metrics clarify the structural overhead of the logical encoding and its implications for performance under noise.

Chapter 6

QRC with repeated measurements

6.1 Introduction

Reservoir Computing (RC) is a machine learning framework designed for processing temporal data using the intrinsic dynamics of a fixed system, known as the *reservoir*. In classical RC, the reservoir is typically a recurrent neural network or a physical system whose internal parameters remain untrained. Instead, learning is performed only at the output layer via linear regression. The reservoir acts as a nonlinear temporal filter, transforming input sequences into high-dimensional representations that retain memory of past inputs. This structure enables efficient learning for tasks such as time-series prediction, classification, and control.

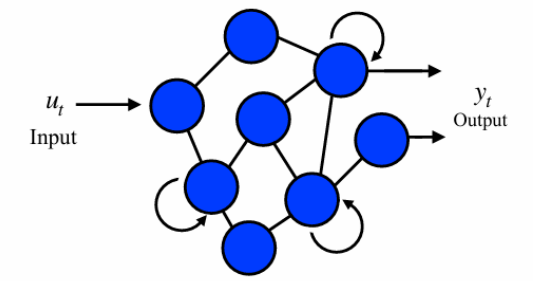


Figure 6.1: Typical reservoir system. The input passes through the intermediate (artificial or physical) layer and is linearly regressed at the output.The weights of the intermediate layer are fixed and are not used for learning [28].

Quantum Reservoir Computing (QRC) extends this paradigm to quantum systems, leveraging their complex dynamics, high-dimensional Hilbert spaces, and intrinsic nonlinearity. In QRC, input signals are encoded into quantum states, which

evolve under fixed unitary dynamics. Outputs are extracted through measurements, often on ancilla qubits. The term *reservoir* reflects the system's ability to store and process temporal information through quantum correlations and entanglement.

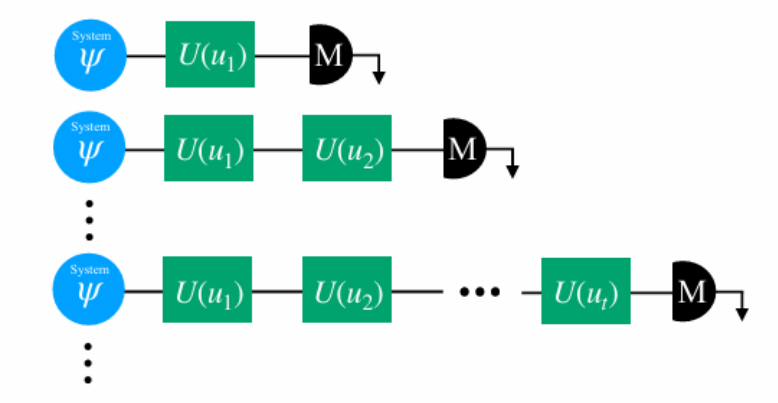


Figure 6.2: Conventional QRC model. To obtain the output signal at time t, the quantum circuit corresponding to the t-th repetition is prepared, and projective measurements are performed at the end of the circuit to estimate the expected values [28].

Running QRC experiments on real quantum hardware serves multiple purposes. It validates theoretical models under realistic noise and decoherence, and it benchmarks the dynamical capacity, memory depth, and nonlinearity of quantum systems without relying on fault-tolerant computation. Importantly, it broadens the scope of benchmarking beyond algorithms designed for error-corrected devices, offering insights into the practical utility of quantum dynamics for machine learning tasks. This contributes to a more inclusive and hardware-aware evaluation of quantum computational resources, especially in the context of hybrid quantum-classical systems and near-term applications.

6.2 Methods

To better understand the implementation proposed by Yasuda et al. [28], it is appropriate to first examine the underlying mathematical formalism.

The quantum reservoir computing (QRC) model under consideration consists of a system of n qubits and an equal number of ancilla qubits. The dynamics of the system are governed by an input-dependent unitary transformation, followed by projective measurements performed on the ancilla qubits at each timestep.

The evolution of the system at time t is described by:

$$\rho_t(m_t) = \frac{1}{p(m_t)} \mathrm{Tr}_a \left[M_{m_t} \hat{U}(u_t) \left(\rho_{t-1}^{(m_{t-1})} \otimes \sigma_a \right) \hat{U}^\dagger(u_t) M_{m_t}^\dagger \right], \tag{6.1}$$

where $\rho_t(m_t)$ denotes the system's density matrix conditioned on the measurement result m_t, and $\hat{U}(u_t)$ is a unitary operator parameterized by the input u_t. The ancilla is initialized in the state $\sigma_a = (|0\rangle\langle 0|)^{\otimes n}$.

The measurement operator associated with a specific outcome m_t is defined as:

$$M_{m_t} = \mathbb{I}_s \otimes \left(\bigotimes_{i=1}^{n} |m_{i,t}\rangle\langle m_{i,t}| \right), \tag{6.2}$$

where $m_{i,t} \in \{0,1\}$ indicates the measurement result of the i-th ancilla qubit, and $m_t = (m_{1,t}, \ldots, m_{n,t})$ is the resulting bitstring.

The probability of obtaining a given measurement result m_t is given by:

$$p(m_t) = \mathrm{Tr}\left[M_{m_t} \hat{U}(u_t) \left(\rho_{t-1}^{(m_{t-1})} \otimes \sigma_a \right) \hat{U}^\dagger(u_t) M_{m_t}^\dagger \right]. \tag{6.3}$$

The QRC dynamics are inherently stochastic, being conditioned on the ancilla measurement outcomes at each timestep. The output of the reservoir at time t is defined as a column vector of expectation values of the Pauli-Z operators acting on the ancilla qubits:

$$h(\rho_t) = \begin{bmatrix} \langle Z_{1,a} \rangle \\ \langle Z_{2,a} \rangle \\ \vdots \\ \langle Z_{n,a} \rangle \end{bmatrix}, \tag{6.4}$$

where $Z_{i,a}$ denotes the Pauli-Z operator acting on the i-th ancilla qubit, expressed as $Z_{i,a} = \mathbb{I} \otimes \cdots \otimes Z \otimes \cdots \otimes \mathbb{I}$.

The output $h(\rho_t)$ is estimated by repeating the quantum experiment N_s times and averaging over the corresponding ancilla measurement outcomes, denoted by the set $B_t = \{m_t^{(1)}, m_t^{(2)}, \ldots, m_t^{(N_s)}\}$. Each expectation value $\langle Z_{i,a} \rangle$ is approximated as:

$$\langle Z_{i,a} \rangle \approx \frac{1}{N_s} \sum_{m_k \in B_t} \left[\mathbb{I}_{B_{i,0}}(m_k) - \mathbb{I}_{B_{i,1}}(m_k) \right], \tag{6.5}$$

where $\mathbb{I}_A(\cdot)$ denotes the indicator function and $B_{i,l} = \{m \in B_t \mid m_i = l\}$.

Accordingly, the ensemble dynamics of the system may be described by the linear map:

$$\rho_t^{\mathrm{ens}} = \sum_{m'_{t-1} \in \{0,1\}^n} M'_{m'} \hat{U}(u_t) \rho_{t-1}^{\mathrm{ens}} \hat{U}^\dagger(u_t) M'^\dagger_{m'}, \tag{6.6}$$

with the operator M'_m defined as:

$$M'_m = \left(\mathbb{I}_s \otimes \prod_{i=1}^{n} X_i^{\mathbb{I}[m_{i,t-1}=1]} \right) M_m. \tag{6.7}$$

To construct a regression model, the reservoir outputs are collected over a set of timesteps and organized into the matrix:

$$X = \begin{bmatrix} h(\rho_{t_f}) & h(\rho_{t_f+1}) & \cdots & h(\rho_{t_l}) & \mathbf{1} \end{bmatrix}^T, \tag{6.8}$$

where t_f and t_l indicate the initial and final timesteps of the training phase, respectively, and $\mathbf{1}$ is a column vector of ones.

The predicted output is computed via:

$$y_{\text{pred}} = X w_{\text{out}}, \tag{6.9}$$

where w_{out} denotes the output weight vector. The optimal weight vector that minimizes the squared error $\|y_{\text{target}} - y_{\text{pred}}\|^2$ is obtained using the Moore-Penrose pseudo-inverse:

$$y_{\text{pred}} = X(X^T X)^{-1} X^T y_{\text{target}}. \tag{6.10}$$

The structure of the input-dependent unitary $\hat{U}(u_t)$ is constructed as a product of 4-qubit unitaries acting on disjoint subsets of the system and ancilla:

$$\hat{U}(u_t) = \prod_{i=0}^{n/4-1} \bar{U}_{4i,4i+1,4i+2,4i+3}, \tag{6.11}$$

where each $\bar{U}_{j,k,l,m}$ acts on system qubits j and k and ancilla qubits l and m, and is defined as:

$$\bar{U}_{j,k,l,m} = \text{CX}_{k,m} \cdot \text{CX}_{j,l} \cdot U_{j,k}(u_t), \tag{6.12}$$

$$U_{j,k}(u_t) = \text{CX}_{j,k} \cdot R_{X_j}(s u_t) \cdot R_{Z_k}(s u_t) \cdot \text{CX}_{j,k} \cdot R_{X_j}(s u_t), \tag{6.13}$$

where $s \in \mathbb{R}$ is a scaling factor and the rotation gates are defined as:

$$R_{X_j}(\theta) = \exp(-i\theta X/2), \qquad R_{Z_j}(\theta) = \exp(-i\theta Z/2). \tag{6.14}$$

The Nonlinear Autoregressive Moving Average (NARMA) model is a benchmark task commonly used to evaluate the nonlinear and memory capabilities of dynamical systems. Given an input time series $\{u_t\}$, the model generates a corresponding output time series $\{y_t\}$. The `NARMA10` task is defined by the recursive relation:

$$y_{t+1} = \alpha y_t + \beta y_t \sum_{i=0}^{n-1} y_{t-i} + \gamma u_{t-n+1} u_t + \delta, \tag{6.15}$$

where the parameters are set to $(\alpha, \beta, \gamma, \delta) = (0.3, 0.05, 1.5, 0.1)$, and the nonlinearity degree is determined by $n = 10$.

The input time series used for the task is given by:

$$u_t = \delta \sin\left(\frac{2\pi\bar{\alpha}t}{T}\right) \sin\left(\frac{2\pi\bar{\beta}t}{T}\right) \sin\left(\frac{2\pi\bar{\gamma}t}{T}\right) + 1, \tag{6.16}$$

The goal is to construct a quantum reservoir computer whose output sequence $\{\hat{y}_t\}$ closely approximates the target sequence $\{y_t\}$ generated by the `NARMA10` model.

To evaluate the performance of the QRC, two metrics are employed: the Normalized Mean Square Error (NMSE) and the Dynamic Time Warping (DTW). The NMSE is defined as:

$$\mathrm{NMSE} = \frac{1}{M_{\mathrm{eval}}} \sum_{t=1}^{M_{\mathrm{eval}}} (y_t - \hat{y}_t)^2, \tag{6.17}$$

where M_{eval} is the number of evaluation points.

In addition to NMSE, the DTW distance is used to quantify the similarity between two time series $S = \{s_i\}_{i=1}^{M}$ and $T = \{t_j\}_{j=1}^{N}$. It is recursively defined as:

$$\mathrm{DTW}(S,T) = f(M,N), \tag{6.18}$$

where

$$f(i,j) = |s_i - t_j| + \min \begin{cases} f(i,j-1), \\ f(i-1,j), \\ f(i-1,j-1), \end{cases} \tag{6.19}$$

$$f(0,0) = 0, \quad f(i,0) = f(0,j) = \infty. \tag{6.20}$$

For this implementation, two different ansatzes have been employed: the first corresponds to the repeated measurement quantum reservoir circuit introduced by the authors of the paper [28], while the second is the Multiscale Entanglement Renormalization Ansatz (MERA) presented in the ansatz catalog [29]. Both ansatzes were adapted to the specific task of encoding input sequences into a quantum reservoir system and extracting output features through measurements on ancilla qubits.

The first ansatz is the one proposed in the reference paper, where the design is centered on a repeated measurement scheme that allows the system to process temporal data efficiently. The model consists of a set of system qubits and an equal number of ancilla qubits. At each timestep, the input value is encoded into the system by single-qubit rotations, in particular through R_x and R_z gates applied to the system qubits, together with entangling CNOT gates that couple them. The crucial part is the interaction between system and ancilla: the system qubits are connected to the ancilla qubits via CNOT gates, and after each interaction the ancillas are measured and reset to $|0\rangle$ (see Fig. 6.3). This repeated projective measurement extracts stochastic output information without destroying the system dynamics, which is preserved across timesteps. The measurement results are collected from the ancilla qubits, mapped into ± 1 values depending on the outcome,

and then averaged over many shots to form the reservoir state.

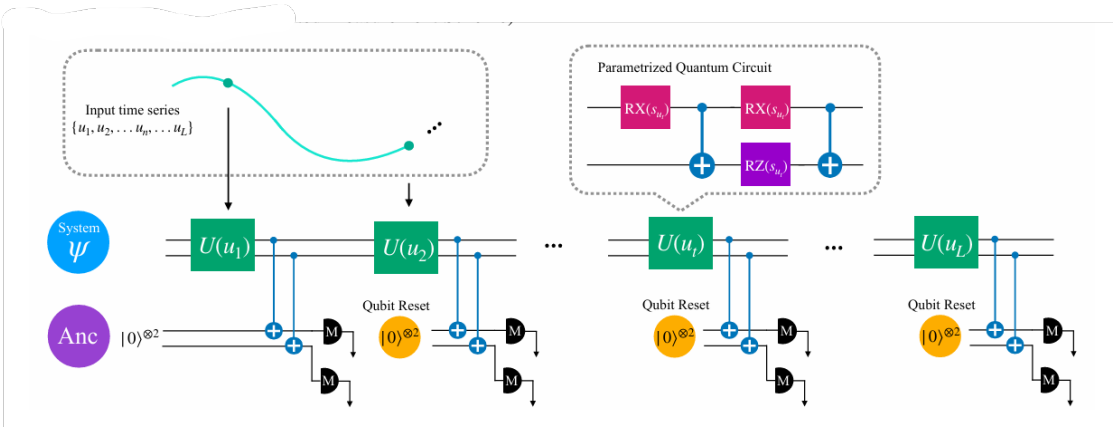


Figure 6.3: Subsystem Structure (Repeated Measurement Scheme) for the QRC model proposed in [28].

The advantage of this approach, compared to conventional quantum reservoir computing where the circuit is restarted at each timestep, is that the repeated measurement scheme enables producing the entire time series within a single run, reducing execution time and improving reproducibility of the dynamics. In the implementation, the circuit is built by initializing all qubits, encoding each input with scaled rotation angles, applying system-system and system-ancilla interactions, measuring and resetting the ancillas, and finally introducing barriers to clearly separate timesteps. A simplified version of the corresponding code is shown below.

```
def _build_full_circuit(self, input_sequence):
    for t, u in enumerate(input_sequence):
        angle = self.scale_input * u
        qc.rx(angle, sys_qr[0])
        qc.cx(sys_qr[0], sys_qr[1])
        qc.rx(angle, sys_qr[0])
        qc.rz(angle, sys_qr[1])
        qc.cx(sys_qr[0], sys_qr[1])
        qc.cx(sys_qr[0], anc_qr[0])
        qc.cx(sys_qr[1], anc_qr[1])
        qc.measure(anc_qr[0], cl_reg[2*t])
        qc.measure(anc_qr[1], cl_reg[2*t+1])
        qc.reset(anc_qr[0])
        qc.reset(anc_qr[1])
        qc.barrier()
```

For the complete implementation, including initialization and data collection, see Appendix 7.

The second ansatz is the MERA (Multiscale Entanglement Renormalization Ansatz), a tensor-network-inspired representation designed to efficiently encode quantum many-body states. Its key feature is the hierarchical structure: layers of unitary operations capture entanglement at different scales, from local to global, providing a coarse-grained description of the system. In the circuit implementation, this translates into repeated layers where entangling CNOT operations are combined

with parametric single-qubit unitaries, here realized as $U(\theta, \theta, \theta)$ gates applied both to system and ancilla qubits (see Fig. 6.4). The structure alternates between entangling gates and local unitaries, mimicking the multiscale renormalization process.

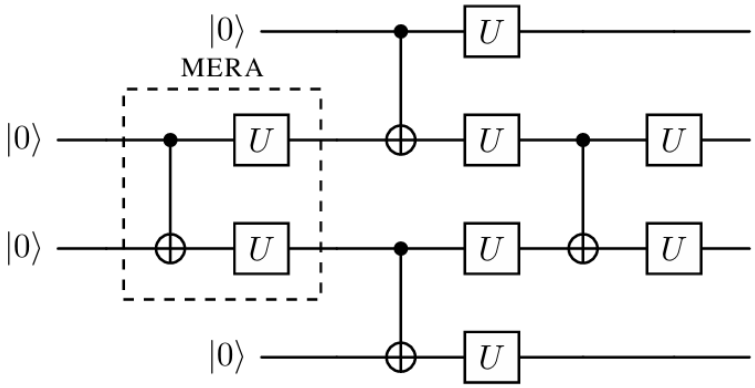


Figure 6.4: Representation of the Multiscale Entanglement Renormalization Ansatz (MERA) [29].

As in the first ansatz, ancillas are measured at each timestep and reset to allow iterative processing of the input sequence. The MERA circuit therefore preserves the hierarchical encoding of correlations while simultaneously producing outputs through ancilla measurements. This ansatz is motivated by its ability to efficiently represent states of lattice systems and compute local expectation values with controlled resources, and in practice it allows the reservoir to capture correlations across multiple scales of the input sequence. A code excerpt highlighting the central structure is reported in the following.

```
def _build_full_circuit(self, input_sequence):
    for t, u in enumerate(input_sequence):
        angle = self.scale_input * u
        qc.cx(sys_qr[0], sys_qr[1])
        for qb in sys_qr:
            qc.append(UGate(angle, angle, angle), [qb])
        qc.cx(anc_qr[0], sys_qr[0])
        for qb in anc_qr:
            qc.append(UGate(angle, angle, angle), [qb])
        qc.barrier()
        qc.measure(anc_qr[0], cl_reg[2*t])
        qc.measure(anc_qr[1], cl_reg[2*t+1])
        qc.reset(anc_qr[0])
        qc.reset(anc_qr[1])
        qc.barrier()
```

Again, the full implementation is provided in Appendix 7.

As a final remark, in order to map the quantum reservoir states to the target sequence, ridge regression combined with cross-validation was employed. A plain linear regression model minimizes the least-squares error, but this often leads to overfitting, especially when there are many features or strong correlations among

them. Ridge regression addresses this issue by introducing an ℓ_2 regularization term that penalizes large regression coefficients. The effect is a more stable model that controls variance while only slightly increasing bias, which generally makes predictions more reliable.

The strength of the regularization is controlled by the parameter α. Choosing the right α is crucial: if it is too small, the model behaves almost like ordinary least squares and may overfit; if it is too large, the model underfits because the coefficients are overly suppressed. In practice, instead of fixing α manually, I defined a set of candidate values to be explored:

```
alphas = np.logspace(-4, 2, 8)
```

This command generates 8 values between 10^{-4} and 10^{2}, equally spaced on a logarithmic scale. These values range from very weak regularization (close to plain linear regression) to very strong regularization (coefficients pushed towards zero). The idea is that the optimal balance is usually found in the intermediate values.

To automatically select the best α, I used a grid search with cross-validation:

```
ridge_cv = GridSearchCV(Ridge(), 'alpha': alphas, cv=5)
```

Here, the argument `cv` specifies the number of folds used in cross-validation. To make this concrete, with `cv=5` the training dataset is divided into five equal parts (folds). In the first round, the model is trained on four folds and validated on the remaining one. In the second round, another fold is used for validation while the remaining four are used for training. This process continues until every fold has served once as the validation set. The performance scores from all five rounds are then averaged to evaluate how well the model generalizes to unseen data.

Cross-validation therefore reduces the risk that the choice of hyperparameters depends on a particular split of the data. Instead, the model is tested across several different partitions, giving a more reliable estimate of its predictive performance. In this setting, grid search combines cross-validation with the exploration of different values of α, so that the final model corresponds to the best trade-off between bias and variance.

6.3 Results

The results of the QRC implementation are presented in this section.

The core of the implementation is based on the *repeated measurement scheme*, a novel approach designed to mitigate the effects of noise and environmental fluctuations. This methodology, as described in the reference paper [28], stands in direct contrast to the conventional *natural noise scheme*, which relies on the intrinsic dissipative dynamics of the physical hardware as a computational resource.

Given this foundational principle, running the QRC experiment on a noiseless simulator, such as the *AerSimulator*, would be counterproductive. A noiseless environment would remove the very mechanism on which this scheme is built,

making the simulation results meaningless for an experiment designed to operate with and exploit noise. The results that follow are therefore obtained both from the noisy simulated backend *IQMFakeAdonis* and from the real quantum computer *IQM Spark*.

Representative outputs are reported in Fig. 6.9 and Fig. 6.11 for the ansatz proposed in [28], and in Fig. 6.10 and Fig. 6.12 for the MERA, executed on both the *IQMFakeAdonis* backend and the *IQM Spark* processor. Each run used 8192 shots.

6.3.1 IQMFakeAdonis Results: Performance on a Noisy Backend

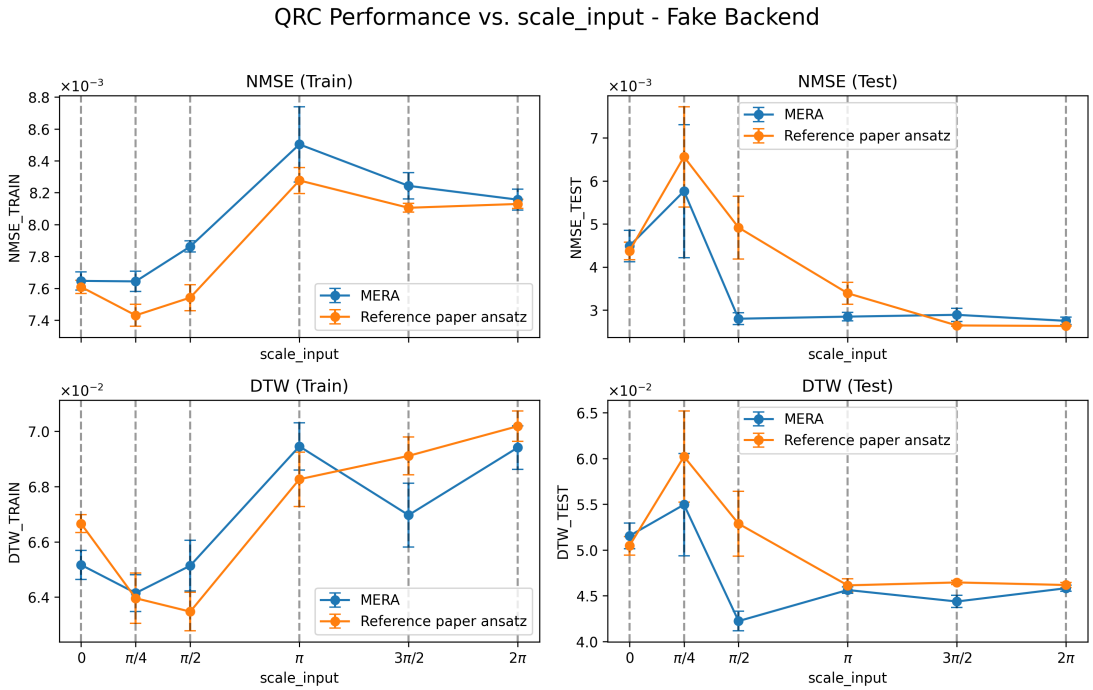


Figure 6.5: Quantum Reservoir Computing (QRC) performance metrics obtained using the *IQMFakeAdonis* backend. The plots report the Normalized Mean Squared Error (NMSE) and Dynamic Time Warping (DTW) distances for both training and testing phases, across varying values of the input scaling parameter. Two ansatzes are compared: the MERA (blue) and the ansatz of the reference paper (orange). Error bars represent the Standard Error of the Mean (SEM), computed as the standard deviation over 10 repetitions (with 8192 shots per run) divided by $\sqrt{10}$.

Fig. 6.5 illustrates the performance of QRC as a function of the input scaling parameter using the noisy backend *IQMFakeAdonis*, with `train_length=80`,

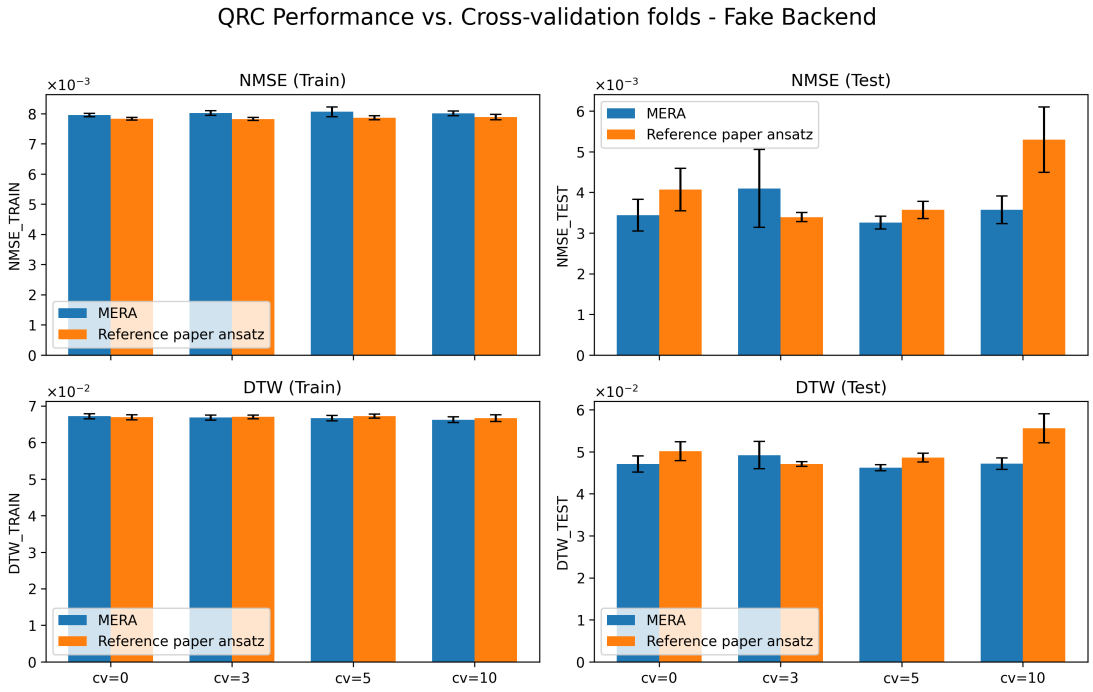


Figure 6.6: Performance metrics of Quantum Reservoir Computing (QRC) obtained on the noisy backend *IQMFakeAdonis*. The plots display the Normalized Mean Squared Error (NMSE) and the Dynamic Time Warping (DTW) distance for both training and testing phases, evaluated across different values of the input scaling parameter and the k-fold cross-validation parameter `cv`. A comparison is provided between the MERA (blue) and the ansatz of the reference paper (orange). Error bars represent the Standard Error of the Mean (SEM), computed over 10 repetitions (with 8192 shots per run) for each value of `cv`.

`test_length=20` and `washout=10`. For these fixed lengths of training, test, and washout, a total of 10 independent repetitions were performed with 8192 shots per run, for each value of $cv \in \{0,3,5,10\}$.

The NMSE (Train) curves indicate that both ansatzes achieve relatively low error across all scaling values ($\approx 8 \times 10^{-3}$), with only minor variations. The ansatz of the reference paper consistently attains slightly lower NMSE values compared to MERA, particularly at $\pi/4$, $\pi/2$ and π scaling, where the gap is most evident. Nevertheless, both models maintain stability with respect to error fluctuations, as confirmed by the narrow confidence intervals.

The NMSE (Test) curves reveal a more distinct behavior: for scaling values ranging from $\pi/4$ to π, the MERA outperforms the ansatz of the reference paper. At higher scaling ($3/2\pi$ to 2π), both ansatzes converge to very low NMSE values, with minimal difference between them. This suggests that the noisy backend favors

robust generalization at larger input scaling.

Regarding DTW (Train), the ansatz of the reference paper achieves an advantage at $\pi/2$ and π, while the MERA outperforms the ansatz of the reference paper at larger input scaling parameters.

For the DTW (Test), the MERA presents a better performance for lower scaling input parameters, especially at $\pi/2$.

Fig. 6.6 shows the dependency on the cross-validation parameter. The NMSE (Train) remains comparable across folds for both ansatzes, with the ansatz of the reference paper slightly outperforming the MERA in most cases. In the NMSE (Test) curves, a more visible difference emerges: the ansatz of the reference paper achieves consistently lower errors at $cv = 3$. Instead, at $cv = 10$, the MERA clearly outperforms the ansatz of the reference paper. The results are subject to higher fluctuations because the number of evaluation points is smaller for the test.

In DTW (Train), no ansatz demonstrates a net superior performance compared to the other. For DTW (Test), however, the trend is very similar to the one of the NMSE (Test): the ansatz of the reference paper slightly outperforms MERA for $cv = 3$, while at $cv = 10$ the gap in performance between the two ansatzes is more evident.

6.3.2 IQM Spark: Real Hardware Execution

Fig. 6.7 illustrates the QRC performance obtained on the real quantum processor *IQM Spark* as a function of the input scaling parameter, with `train_length=80`, `test_length=20` and `washout=10`. For these fixed lengths of training, test, and washout, a total of 10 independent repetitions were performed with 8192 shots per run, for each value of $cv \in \{0,3,5,10\}$.

The NMSE (Train) curves show a clear separation between the two ansatzes: the ansatz of the reference paper consistently achieves lower error across almost all scaling values, with a pronounced advantage at $\pi/2$, π and $3\pi/2$. MERA displays a larger error spread at π.

In terms of NMSE (Test), the ansatz of the reference paper again shows stronger generalization. As scaling increases, both ansatzes converge, with minimal difference observed at $3\pi/2$ and 2π.

The DTW (Train) plots reveal a consistent superiority of the ansatz of the reference paper across almost the entire scaling range. At $\pi/2$ and $3\pi/2$, the ansatz of the reference paper achieves a lower DTW with respect to the MERA, showing a more accurate temporal alignment. This trend is mirrored in the DTW (Test) curves: the ansatz of the reference paper achieves visibly lower DTW values across almost all scalings, with the strongest improvement at $3\pi/2$, where the reduction compared to MERA is most pronounced.

Fig. 6.8 presents the impact of the cross-validation parameter on hardware runs. For NMSE (Train), the ansatz of the reference paper shows more stable and consistently lower errors across folds, while MERA fluctuates more significantly,

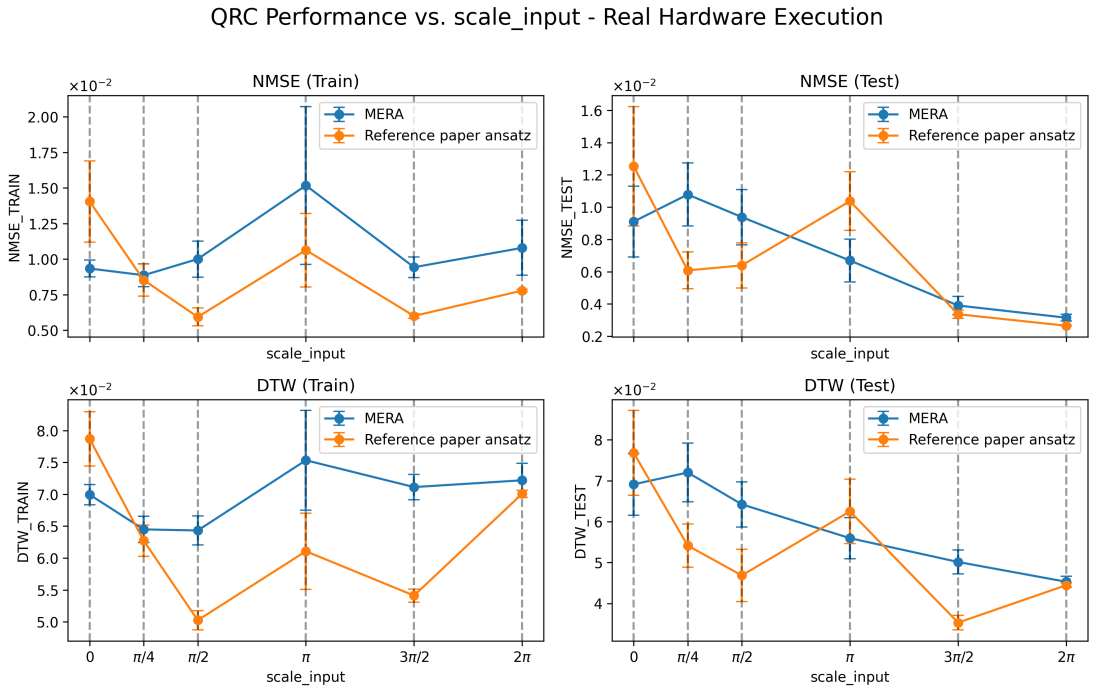


Figure 6.7: Quantum Reservoir Computing (QRC) performance metrics obtained using the *IQM Spark* quantum processor. The plots report the Normalized Mean Squared Error (NMSE) and Dynamic Time Warping (DTW) distances for both training and testing phases, across varying values of the input scaling parameter. Two ansatzes are compared: the MERA (blue) and the ansatz of the reference paper (orange). Error bars represent the Standard Error of the Mean (SEM), computed as the standard deviation over 10 repetitions (with 8192 shots per run) divided by $\sqrt{10}$.

particularly at $cv = 5$. Similarly, NMSE (Test) indicates that the ansatz of the reference paper maintains an advantage at higher folds.

For DTW (Train), the ansatz of the reference paper once again outperforms MERA in all folds, with a clear reduction of distance and lower variance. DTW (Test) confirms this trend: while MERA tends to maintain higher DTW values, the ansatz of the reference paper consistently achieves better alignment, especially at $cv = 5$ and $cv = 10$.

These results highlight that in real hardware execution, the ansatz of the reference paper systematically outperforms MERA across both NMSE and DTW metrics, suggesting superior resilience against hardware-induced noise and decoherence.

A direct comparison between the two execution environments reveals important differences. On the noisy backend *IQMFakeAdonis*, both ansatzes achieve relatively similar performance. Variances remain modest, reflecting the backend's noise model

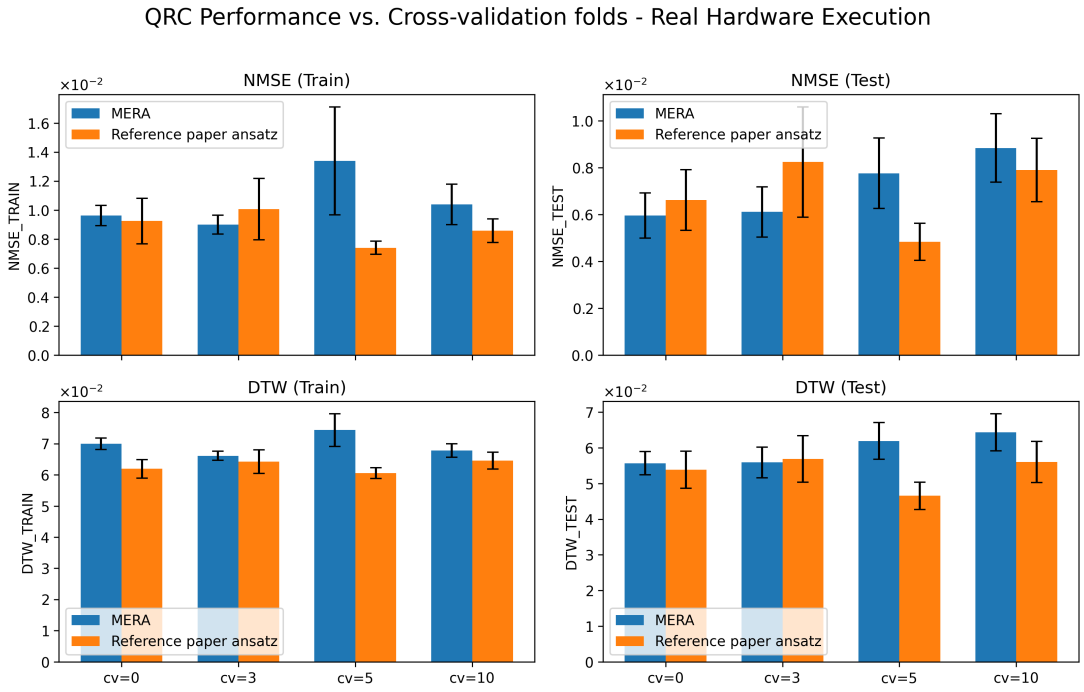


Figure 6.8: Performance metrics of Quantum Reservoir Computing (QRC) obtained on the *IQM Spark* quantum processor. The plots display the Normalized Mean Squared Error (NMSE) and the Dynamic Time Warping (DTW) distance for both training and testing phases, evaluated across different values of the input scaling parameter and the k-fold cross-validation parameter `cv`. A comparison is provided between the MERA (blue) and the ansatz of the reference paper (orange). Error bars represent the Standard Error of the Mean (SEM), computed over 10 repetitions (with 8192 shots per run) for each value of `cv`.

stability.

On the real hardware, however, the discrepancy between the ansatzes becomes much clearer. The ansatz of the reference paper consistently outperforms MERA across nearly all metrics. The hardware execution introduces stronger fluctuations and higher variances compared to the simulated backend, underscoring the impact of real noise sources such as decoherence, gate errors, and readout imperfections.

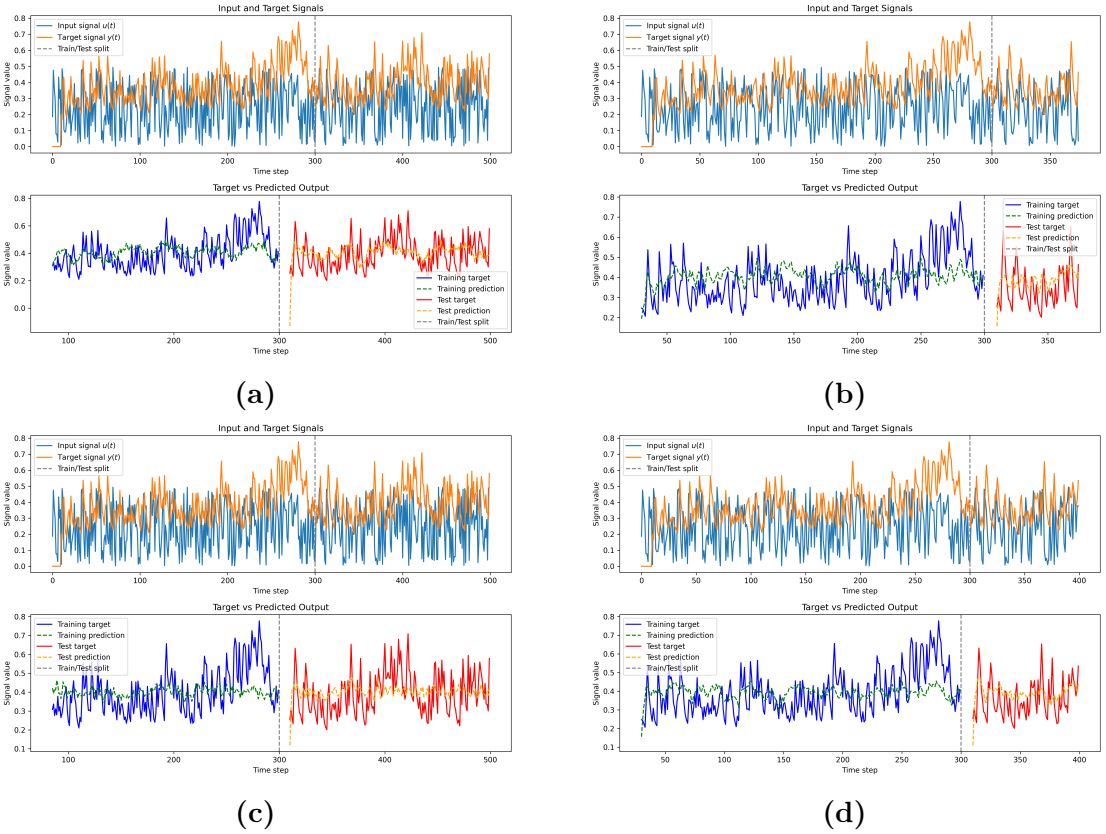


Figure 6.9: Representative outputs of the Quantum Reservoir Computing (QRC) implementation on the *IQMFakeAdonis* backend. The figure displays a selection of results from 10 independent repetitions, each utilizing the ansatz described in the reference paper. All runs were executed with a total of 8192 shots and a 5-fold cross-validation (`cv` = 5). The hyperparameter configurations for the time series data for the different subplots are as follows: (a) and (c) a training length of `train_len` = 300, a testing length of `test_len` = 200, and a washout period of `washout` = 50; (b) a training length of `train_len` = 300, a testing length of `test_len` = 120, and a washout period of `washout` = 20; (d) a training length of `train_len` = 300, a testing length of `test_len` = 100, and a washout period of `washout` = 25.

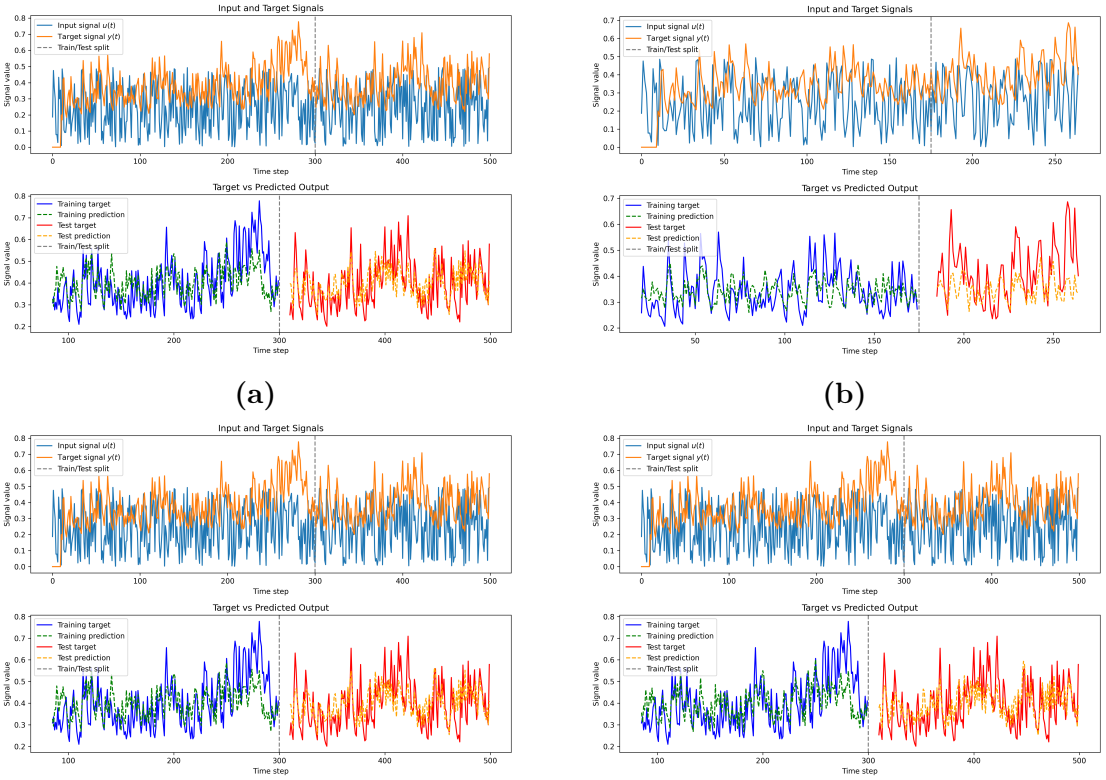


Figure 6.10: Representative outputs of the Quantum Reservoir Computing (QRC) implementation on the *IQMFakeAdonis* backend. The figure displays a selection of results from 10 independent repetitions, each utilizing the MERA. All runs were executed with a total of 8192 shots and a 5-fold cross-validation (`cv` = 5). The hyperparameter configurations for the time series data for the different subplots are as follows: (a), (c) and (d) a training length of `train_len` = 300, a testing length of `test_len` = 200, and a washout period of `washout` = 75; (b) a training length of `train_len` = 175, a testing length of `test_len` = 100, and a washout period of `washout` = 25.

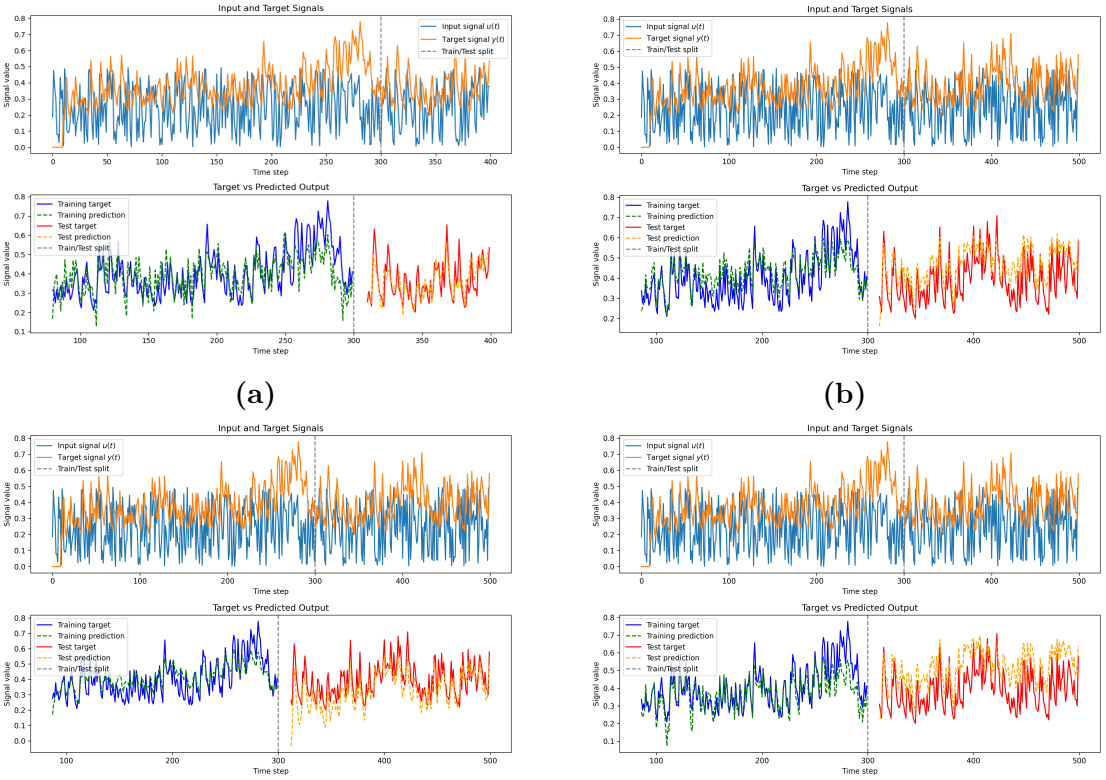


Figure 6.11: Representative outputs of the Quantum Reservoir Computing (QRC) implementation on the *IQM Spark* processor. The figure displays a selection of results from 10 independent repetitions, each utilizing the ansatz described in the reference paper. All runs were executed with a total of 8192 shots and a 5-fold cross-validation (`cv` = 5). The hyperparameter configurations for the time series data for the different subplots are as follows: (a) a training length of `train_len` = 300, a testing length of `test_len` = 100, and a washout period of `washout` = 50. (b), (c) and (d) a training length of `train_len` = 300, a testing length of `test_len` = 200, and a washout period of `washout` = 50.

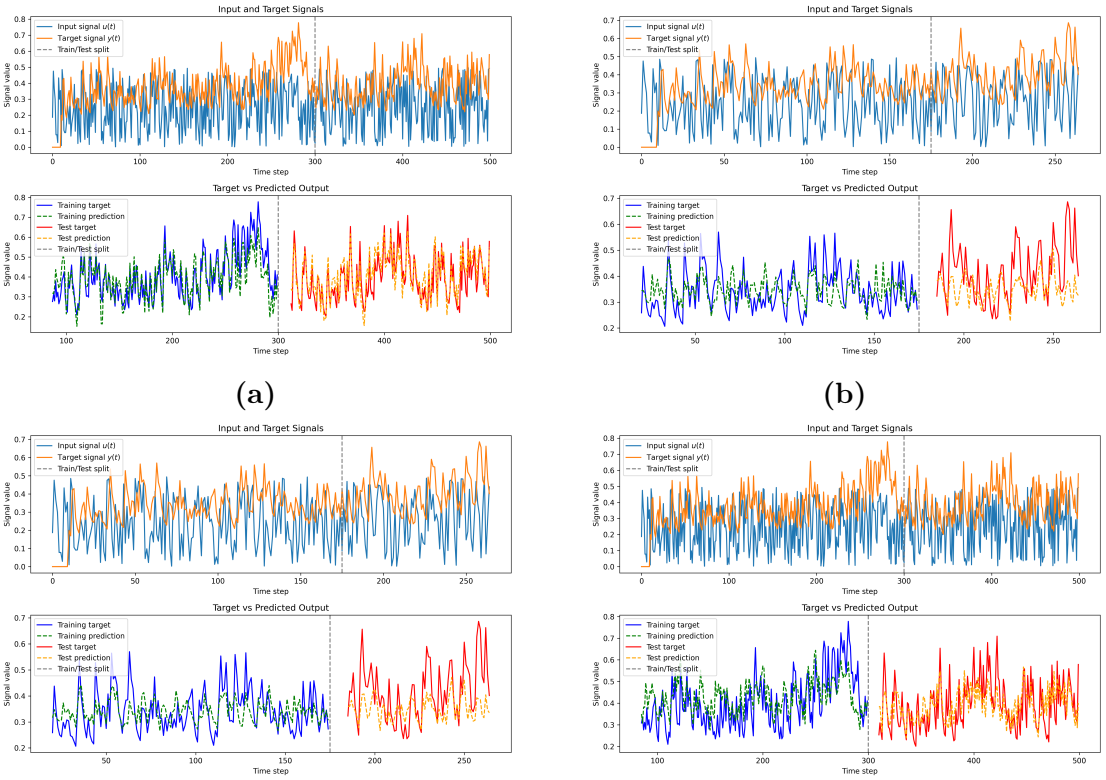


Figure 6.12: Representative outputs of the Quantum Reservoir Computing (QRC) implementation on the *IQM Spark* processor. The figure displays a selection of results from 10 independent repetitions, each utilizing the MERA. All runs were executed with a total of 8192 shots and a 5-fold cross-validation (`cv` = 5). The hyperparameter configurations for the time series data for the different subplots are as follows: (a) and (d) a training length of `train_len` = 300, a testing length of `test_len` = 200, and a washout period of `washout` = 50; (b) and (c) a training length of `train_len` = 175, a testing length of `test_len` = 100, and a washout period of `washout` = 25.

Chapter 7

Conclusions

This work has investigated three advanced implementations of quantum algorithms and models across noiseless simulations, a noisy simulated backend, and real quantum hardware. The first implementation concerned the fault-tolerant realization of the Deutsch-Jozsa algorithm using the [[4,2,2]] error-detecting code. In this setting, logical qubits were prepared and transversal operations performed without the use of ancillary qubits, allowing the detection and rejection of errors by means of post-selection. The second implementation addressed a variational quantum algorithm, namely the Variational Quantum Eigensolver (VQE), applied to the Anderson Impurity Model, with the aim of comparing bare and encoded circuits in reproducing ground state energies. The third implementation explored Quantum Reservoir Computing (QRC) for sequence prediction tasks, contrasting a simple reference ansatz from the literature with a more complex MERA-inspired structure. Taken together, these three case studies span fault-tolerant computation, variational algorithms, and quantum machine learning, offering a broad perspective on the challenges of near-term quantum devices.

In order to assess correctness, the first two implementations were first tested under noiseless simulations. It is important to emphasize that such results hold little physical significance: in an ideal environment, with no noise and perfect gate operations, error detection and encoding provide no tangible advantage, since no errors are present to be detected or corrected. The purpose of noiseless simulations was therefore not to demonstrate practical benefits, but to establish a rigorous baseline, ensuring that the encodings, logical state preparations, and circuit constructions were implemented correctly, and that no mistakes were introduced at the code or workflow level. This baseline was indispensable for interpreting the results obtained on noisy backends and hardware.

Subsequently, the three implementations were executed on *IQMFakeAdonis*, a noisy simulated backend. It must be stressed that this is a backend with a fixed and controlled noise model, chosen because it matches the gate set and coupling map of the *IQM Spark* processor. On this platform, the results highlighted the potential of logical encoding and advanced algorithms under controlled noise conditions.

In the Deutsch-Jozsa algorithm, balanced oracle functions particularly benefited from encoding, showing improved noise resilience. In the VQE experiments, logical circuits consistently reduced energy deviations compared to bare implementations, providing evidence of noise mitigation. In QRC, both ansatzes performed stably with only modest differences, confirming that logical encodings and advanced designs can operate reliably in a fixed-noise environment. These findings demonstrate that, under simplified assumptions of static noise, the advantages of encoding and advanced protocols are clearly observable.

Execution on the real *IQM Spark* processor, however, presented a more nuanced picture. Hardware noise is not fixed but varies with calibrations and environmental factors, leading to significant fluctuations in performance. In the Deutsch-Jozsa implementation, constant oracles benefited from encoding, but balanced oracles experienced a degradation due to the increased circuit depth and gate count associated with encoding. In the VQE experiments, logical circuits failed to surpass the bare ones: their larger depth, combined with the need for mid-circuit measurement and reset, led to broader error bars and deviations in the estimation of the ground state energy. In QRC, the simpler reference ansatz from the literature proved more resilient to hardware noise than the more complex MERA-inspired structure, underscoring that current hardware favors shallow, resource-efficient designs.

These observations make clear that while the device can successfully run simple circuits from the literature (appearing fully functional in those contexts) limitations emerge when more advanced and resource-intensive algorithms are attempted.

A central conclusion concerns the role of mid-circuit measurement and reset operations. These are indispensable for reusing ancillary qubits, enabling conditional operations, and implementing quantum error correction cycles. They constitute the very foundation of scalable fault-tolerance. On the present hardware, however, these operations are not fully optimized: their execution times approach, and in some cases exceed, qubit coherence times. This mismatch directly impacts performance, limiting the benefits of encoding and advanced algorithms. Thus, while theoretical frameworks for error detection and logical encoding are sound, their practical effectiveness is constrained by the current maturity of hardware operations.

In summary, this work has shown that noiseless and noisy simulated backends validate the theoretical soundness of the three implementations, while real hardware exposes bottlenecks related to decoherence, fluctuating noise, and particularly mid-circuit measurement and reset. The experiments confirm that the hardware operates reliably for simple benchmarks, but advanced protocols reveal subtler limitations, highlighting the distance still to be bridged between theory and practice in quantum fault-tolerance and learning models.

Future Improvements

Several paths for future research emerge from this study, aimed at improving the reliability and scalability of advanced quantum protocols:

- **Optimization of Mid-Circuit Measurement and Reset:** These operations are essential for ancilla reuse and constitute the backbone of quantum error correction. At present, these operations are not optimized and their execution times may approach or exceed coherence times, creating a bottleneck that limits the benefits of error-detecting codes. Enhancing both the speed and fidelity of these operations is indispensable for scalable fault-tolerance.
- **Development of Noise-Aware Simulated Backends:** *IQMFakeAdonis* employs a fixed noise model, which fails to reproduce the variability of real hardware such as *IQM Spark*. Future work should focus on constructing backends that incorporate historical calibration data and realistic noise fluctuations of the real quantum processor, providing a more faithful testbed for pre-hardware circuit validation and benchmarking.
- **Advanced Error-Mitigation and Circuit Optimization:** Improvements in transpilation, qubit mapping, and gate cancellation can substantially reduce circuit depth and gate count. Shallow and noise-resilient ansatzes are particularly important for variational and quantum machine learning algorithms, whose performance is highly sensitive to depth and error accumulation.
- **Hybrid Classical-Quantum Post-Processing:** Incorporating statistical corrections, filtering, and post-selection strategies can mitigate the impact of noise in experimental data, complementing hardware-level improvements and enhancing the fidelity of results.
- **Hardware-Level Enhancements:** Long-term progress requires improving gate fidelities, reducing crosstalk, and fully optimizing readout, reset, and mid-circuit measurements. These enhancements are particularly crucial for logical encodings, where the overhead amplifies the impact of hardware imperfections.

Code Availability

```python
from qiskit import QuantumCircuit, ClassicalRegister, transpile
import matplotlib as plt
import numpy as np
import pandas as pd

# --- Define initial bare state ---
def bare_initial_state():
    qc = QuantumCircuit(2)
    qc.x(0)
    qc.h(0)
    qc.x(1)
    qc.h(1)
    return qc

# --- Define initial encoded state ---
def encoded_initial_state():
    qc = QuantumCircuit(4)
    qc.h(1)
    qc.cx(1, 0)
    qc.h(3)
    qc.cx(3, 2)
    qc.barrier()
    qc.y(1)
    qc.y(3)
    return qc

# --- Define bare oracle ---
def bare_oracle(fx):
    qc = QuantumCircuit(2)
    if fx == "0":
        pass
    elif fx == "1":
        qc.x(0)
    elif fx == "x":
        qc.cx(1, 0)
    elif fx == "1+x":
        qc.x(0)
        qc.cx(1, 0)
    return qc

# --- Define encoded oracle ---
def encoded_oracle(fx):
    qc = QuantumCircuit(4)
    if fx == "0":
        pass
    elif fx == "1":
        qc.z(0)
        qc.z(1)
        qc.barrier()
    elif fx == "x":
        qc.s(0)
        qc.z(1)
        qc.z(2)
        qc.s(3)
        qc.barrier()
        qc.s(1)
        qc.s(2)
        qc.barrier()
    elif fx == "1+x":
        qc.z(0)
        qc.s(1)
        qc.z(2)
        qc.s(3)
        qc.barrier()
```

```
        qc.s(0)
        qc.s(2)
        qc.barrier()
    return qc

# --- Build full bare circuit ---
def bare_circuit(fx):
    qc = bare_initial_state()
    qc = qc.compose(bare_oracle(fx))
    qc.h(1)
    qc.barrier()
    full_qc = QuantumCircuit(2, 1)
    full_qc = full_qc.compose(qc)
    full_qc.measure(1, 0)
    return full_qc

# --- Build full encoded circuit ---
def encoded_circuit(fx):
    qc = encoded_initial_state()
    qc = qc.compose(encoded_oracle(fx))
    qc.h([0, 1, 2, 3])
    qc.barrier()
    full_qc = QuantumCircuit(4, 4)
    full_qc = full_qc.compose(qc)
    full_qc.measure(0, 0)
    full_qc.measure(1, 1)
    full_qc.measure(2, 2)
    full_qc.measure(3, 3)
    return full_qc

# --- Transpile bare circuit ---
def transpiled_circuit_bare(circuit, backend):
    mapping = {circuit.qubits[0]: 0, circuit.qubits[1]: 2}
    initial_layout = Layout(mapping)
    transpiled = transpile(
        circuit,
        backend=backend,
        coupling_map=iqm_coupling_map,
        optimization_level=0
    )
    print(transpiled)
    swap_count_optimized = transpiled.count_ops().get('swap', 0)
    print("Number of SWAP gates after optimization:", swap_count_optimized)
    layout_info = transpiled._layout
    if isinstance(layout_info, TranspileLayout):
        print(layout_info.final_layout)
    return transpiled

# --- Transpile encoded circuit ---
def transpiled_circuit_encoded(circuit, backend):
    mapping = {
        circuit.qubits[0]: 0,
        circuit.qubits[1]: 1,
        circuit.qubits[2]: 2,
        circuit.qubits[3]: 3
    }
    initial_layout = Layout(mapping)
    transpiled = transpile(
        circuit,
        backend=backend,
        coupling_map=iqm_coupling_map,
        optimization_level=0
    )
    print(transpiled)
    swap_count_optimized = transpiled.count_ops().get('swap', 0)
    print("Number of SWAP gates after optimization:", swap_count_optimized)
    layout_info = transpiled._layout
    if isinstance(layout_info, TranspileLayout):
```

```
        print(layout_info.final_layout)
    return transpiled

# --- Run simulation and get raw counts ---
def run_simulation(circuit, backend, shots):
    results = backend.run(circuit, shots=shots).result()
    counts = results.get_counts()
    print(f"Original Counts: {counts} \n")
    return counts

# --- Filter counts for valid encoded outcomes ---
def filter_encoded_counts(counts):
    return {k: v for k, v in counts.items() if k.count('1') % 2 == 0}

# --- Compute ratio of valid post-selected outcomes ---
def compute_postselection_ratio(counts):
    total = sum(counts.values())
    valid = sum(v for k, v in counts.items() if k.count('1') % 2 == 0)
    return valid / total if total > 0 else 0.0

# --- Statistical distance for bare circuit ---
def statistical_distance_bare(P0, P1, Q0, Q1):
    return 0.5 * (abs(P0 - Q0) + abs(P1 - Q1))

# --- Logical probabilities for encoded outcomes ---
def compute_logical_probabilities(filtered_counts):
    total_valid = sum(filtered_counts.values())
    R00 = (filtered_counts.get('0000', 0) + filtered_counts.get('1111', 0)) /
    total_valid
    R01 = (filtered_counts.get('1100', 0) + filtered_counts.get('0011', 0)) /
    total_valid
    R10 = (filtered_counts.get('1010', 0) + filtered_counts.get('0101', 0)) /
    total_valid
    R11 = (filtered_counts.get('0110', 0) + filtered_counts.get('1001', 0)) /
    total_valid
    return {"R00": R00, "R01": R01, "R10": R10, "R11": R11}

# --- Statistical distance for encoded circuit ---
def statistical_distance_encoded(P0, P1, logical_probs):
    R0 = logical_probs["R00"] + logical_probs["R01"]
    R1 = logical_probs["R10"] + logical_probs["R11"]
    return 0.5 * (abs(P0 - R0) + abs(P1 - R1))

# --- Noise reduction factor ---
def compute_noise_reduction(D_bare, D_enc):
    return (D_enc - D_bare) / D_bare if D_bare != 0 else float('nan')

# --- Statistical error for bare distance ---
def statistical_distance_bare_error(P0, P1, Q0, Q1, N):
    sigma_Q0 = np.sqrt(Q0 * (1 - Q0) / N) if N > 0 else 0
    sigma_Q1 = np.sqrt(Q1 * (1 - Q1) / N) if N > 0 else 0
    return 0.5 * np.sqrt(sigma_Q0**2 + sigma_Q1**2)

# --- Statistical error for encoded distance ---
def statistical_distance_encoded_error(P0, P1, logical_probs, counts_filtered,
    N_valid):
    if N_valid == 0:
        return 0.0
    terms = ['0000','1111','1100','0011','1010','0101','0110','1001']
    variances = {}
    for term in terms:
        freq = counts_filtered.get(term, 0)
        p = freq / N_valid
        variances[term] = p * (1 - p) / N_valid
    R0_variance = variances['0000'] + variances['1111'] + variances['1100'] +
    variances['0011']
    R1_variance = variances['1010'] + variances['0101'] + variances['0110'] +
    variances['1001']
```

```
    return 0.5 * np.sqrt(R0_variance + R1_variance)

# --- Error on noise reduction ---
def compute_noise_reduction_error(D_bare, D_enc, sigma_bare, sigma_enc):
    if D_bare == 0:
        return float('nan')
    term_bare = (D_enc / D_bare) * sigma_bare
    term_enc = sigma_enc
    return np.sqrt(term_enc**2 + term_bare**2) / abs(D_bare)

# --- Analyze bare and encoded circuits ---
def analyze_circuits(ideal_P0, ideal_P1, bare_circuit, encoded_circuit, backend,
    shots):
    transpiled_bare = transpiled_circuit_bare(bare_circuit, backend)
    transpiled_encoded = transpiled_circuit_encoded(encoded_circuit, backend)
    counts_bare = run_simulation(transpiled_bare, backend, shots)
    counts_encoded = run_simulation(transpiled_encoded, backend, shots)
    total_bare = sum(counts_bare.values())
    Q0 = counts_bare.get('0', 0) / total_bare if total_bare > 0 else 0
    Q1 = counts_bare.get('1', 0) / total_bare if total_bare > 0 else 0
    D_bare = statistical_distance_bare(ideal_P0, ideal_P1, Q0, Q1)
    sigma_bare = statistical_distance_bare_error(ideal_P0, ideal_P1, Q0, Q1,
    total_bare)
    postselection_ratio = compute_postselection_ratio(counts_encoded)
    filtered_counts = filter_encoded_counts(counts_encoded)
    logical_probs = compute_logical_probabilities(filtered_counts)
    D_enc = statistical_distance_encoded(ideal_P0, ideal_P1, logical_probs)
    N_valid = sum(v for k, v in counts_encoded.items() if k.count('1') % 2 == 0)
    sigma_enc = statistical_distance_encoded_error(ideal_P0, ideal_P1,
    logical_probs, counts_encoded, N_valid)
    noise_reduction = compute_noise_reduction(D_bare, D_enc)
    sigma_reduction = compute_noise_reduction_error(D_bare, D_enc, sigma_bare,
    sigma_enc)
    return {
        'postselection_ratio': postselection_ratio,
        'D_bare': D_bare,
        'sigma_bare': sigma_bare,
        'D_enc': D_enc,
        'sigma_enc': sigma_enc,
        'noise_reduction': noise_reduction,
        'sigma_reduction': sigma_reduction
    }

# --- Run full analysis and save results ---
def run_full_analysis(oracle_definitions, backend, shots=50000, repetitions=40,
    csv_filename="deutsch_422_results.csv"):
    all_rows = []
    for rep in range(1, repetitions + 1):
        print(f"\n### Repetition {rep} of {repetitions} ###")
        for oracle_name, P0, P1, bare_circ, encoded_circ in oracle_definitions:
            print(f"--- Oracle: {oracle_name} ---")
            res = analyze_circuits(P0, P1, bare_circ, encoded_circ, backend,
    shots)
            row = {
                "Repetition": rep,
                "Oracle": oracle_name,
                "D_bare": res['D_bare'],
                "sigma_bare": res['sigma_bare'],
                "D_encoded": res['D_enc'],
                "sigma_encoded": res['sigma_enc'],
                "D_diff": res['D_enc'] - res['D_bare'],
                "sigma_diff": np.sqrt(res['sigma_bare']**2 + res['sigma_enc']**2),
                "Postselection": res['postselection_ratio'],
                "Noise_Reduction_(%)": res['noise_reduction'] * 100,
                "sigma_Reduction_(%)": res['sigma_reduction'] * 100
            }
            all_rows.append(row)
    df_full = pd.DataFrame(all_rows)
```

```
    df_full.to_csv(csv_filename, index=False)
    print(f"\nAll results saved in: {csv_filename}")

# --- Define oracles and run experiment ---
oracle_definitions = [
    ("f(x) = 0",    0, 1, bare_circuit("0"),    encoded_circuit("0")),
    ("f(x) = x",    1, 0, bare_circuit("x"),    encoded_circuit("x")),
    ("f(x) = 1+x",  1, 0, bare_circuit("1+x"),  encoded_circuit("1+x")),
    ("f(x) = 1",    0, 1, bare_circuit("1"),    encoded_circuit("1"))
]

shots = 50000
repetitions = 1000
csv_filename = "deutsch_422_results.csv"

run_full_analysis(
    oracle_definitions,
    backend=noisy_simulator,
    shots=shots,
    repetitions=repetitions,
    csv_filename=csv_filename
)
```

Listing 1: Source code for the fault-tolerant implementation of the Deutsch-Jozsa Algorithm

```
from qiskit import QuantumCircuit
from qiskit.circuit import Parameter
from qiskit.quantum_info import SparsePauliOp
from qiskit_algorithms import VQE
from qiskit_algorithms.optimizers import COBYLA
from qiskit.primitives import Estimator
import numpy as np

# Definition of the parametric ansatz
def ansatz(n_qubits: int) -> tuple[QuantumCircuit, list[Parameter]]:
    # Two variational parameters
    theta_0 = Parameter('theta_0')
    theta_1 = Parameter('theta_1')

    qc = QuantumCircuit(n_qubits)
    qc.h(0)
    qc.cx(0, 1)

    qc.rx(theta_0, 0)
    qc.cx(0, 1)
    qc.rz(theta_1, 1)
    qc.cx(0, 1)

    return qc, [theta_0, theta_1]


def run_logical_vqe(qiskit_hamiltonian: SparsePauliOp) -> tuple[float,
    list[float]]:
    np.random.seed(42)

    # Initial angles for the optimizer
    init_angles = np.random.random(2) * 1e-1

    # Obtain Qiskit Ansatz
    num_qubits = qiskit_hamiltonian.num_qubits
    qc_ansatz, params = ansatz(num_qubits)

    # VQE solver setup
    vqe_solver = VQE(ansatz=qc_ansatz,
                     optimizer=COBYLA(maxiter=350, tol=1e-10),
```

```
                    initial_point=init_angles, estimator=Estimator())

    # Compute the minimum eigenvalue (energy) of the Hamiltonian
    result = vqe_solver.compute_minimum_eigenvalue(qiskit_hamiltonian)

    return result.eigenvalue.real, result.optimal_point.tolist()


from qiskit import QuantumCircuit
from qiskit.circuit import Parameter
from typing import List

def aim_physical_circuit(angles: List[float], basis: str, ignore_meas_id: bool =
    False) -> QuantumCircuit:
    qc = QuantumCircuit(2)

    # Bell state preparation
    qc.h(0)
    qc.cx(0, 1)

    # Rx gate on the first qubit
    qc.rx(angles[0], 0)

    # ZZ interaction
    qc.cx(0, 1)
    qc.rz(angles[1], 1)
    qc.cx(0, 1)

    if basis == "z_basis":
        if not ignore_meas_id:
            qc.barrier()  # equivalent to identity
        qc.measure_all()
    elif basis == "x_basis":
        qc.h([0, 1])
        if not ignore_meas_id:
            qc.barrier()
        qc.measure_all()
    else:
        raise ValueError(f"Unsupported basis provided: {basis}")

    return qc


from qiskit import QuantumCircuit
from qiskit.circuit import Parameter
from typing import List

def aim_logical_circuit(angles: List[float], basis: str, ignore_meas_id: bool =
    False) -> QuantumCircuit:
    qc = QuantumCircuit(5, 6)

    # Bell state preparation
    qc.h([1, 2])
    qc.cx(2, 3)
    qc.cx(1, 4)

    # Rx on the first qubit
    qc.h(0)
    qc.cx(0, 1)
    qc.cx(0, 3)
    qc.rx(angles[0], 0)
    qc.cx(0, 1)
    qc.cx(0, 3)
    qc.h(0)
    qc.measure(0, 0) # --> syndrome 1
    qc.reset(0)
    qc.barrier()

```

```
    # Rz on the second qubit
    qc.cx(2, 0)
    qc.cx(3, 0)
    qc.rz(angles[1], 0)
    qc.cx(1, 0)
    qc.cx(4, 0)
    qc.barrier()
    qc.measure(0, 1) # --> syndrome 2

    if basis == "z_basis":
        if not ignore_meas_id:
            qc.barrier()
        qc.measure([1, 2, 3, 4], [2, 3, 4, 5])
    elif basis == "x_basis":
        qc.h([1, 2, 3, 4])
        qc.swap(2, 3)
        if not ignore_meas_id:
            qc.barrier()
        qc.measure([1, 2, 3, 4], [2, 3, 4, 5])
    else:
        raise ValueError(f"Unsupported basis provided: {basis}")

    import matplotlib.pyplot as plt
    fig = qc.draw(output='mpl')
    fig.savefig('circuit.png', dpi=300, bbox_inches='tight')
    plt.close(fig)

    return qc


from qiskit import transpile
from qiskit_aer import AerSimulator

simulator = AerSimulator()

def generate_circuit_set(ignore_meas_id: bool = False) -> dict:
    u_vals = [1, 5, 9]
    v_vals = [-9, -1, 7]
    circuit_dict = {}

    for u in u_vals:
        for v in v_vals:
            # Build Hamiltonian in Qiskit
            hamiltonian = (
                0.25 * u * SparsePauliOp.from_list([("ZZ", 1.0)])
                - 0.25 * u * SparsePauliOp.from_list([("II", 1.0)])
                + v * SparsePauliOp.from_list([("XI", 1.0)])
                + v * SparsePauliOp.from_list([("IX", 1.0)])
            )

            # Run VQE to get optimal angles
            _, opt_params = run_logical_vqe(hamiltonian)
            angles = [float(angle) for angle in opt_params]
            print(f"Computed optimal angles={angles} for U={u}, V={v}")

            tmp_physical_dict = {}
            tmp_logical_dict = {}

            for basis in ("z_basis", "x_basis"):
                # Create circuits
                physical = aim_physical_circuit(angles, basis,
    ignore_meas_id=ignore_meas_id)
                logical = aim_logical_circuit(angles, basis,
    ignore_meas_id=ignore_meas_id)

                # Transpile both circuits
                transpiled_physical = transpile(
                    physical,
```

```
                    noisy_simulator,
                    optimization_level=0,
                    coupling_map=iqm_coupling_map
                )
                transpiled_logical = transpile(
                    logical,
                    noisy_simulator,
                    optimization_level=0,
                    coupling_map=iqm_coupling_map
                )

                tmp_physical_dict[basis] = transpiled_physical
                tmp_logical_dict[basis] = transpiled_logical

            circuit_dict[f"{u}:{v}"] = {
                "physical": tmp_physical_dict,
                "logical": tmp_logical_dict,
            }

    print("\nFinished building optimized circuits!")
    return circuit_dict

from typing import Mapping, Sequence
import numpy as np
from collections import defaultdict

# Helper to compute the number of qubits from counts
def _num_qubits(counts: Mapping[str, float]) -> int:
    for key in counts:
        if key.isdecimal():
            return len(key)
    return 0


def process_counts(counts: Mapping[str, float]) -> dict[str, float]:
    """
    - Reverse the bitstring.
    - Keep only bitstrings ending with '00' (null syndromes).
    - Extract the 4 data qubits (bits -3,-4,-5,-6).
    - Keep only those with even number of '1'.
    - Return filtered dictionary with the 4 remaining bits as keys.
    """
    new_data = defaultdict(float)

    for bitstring, count in counts.items():
        reversed_bitstring = bitstring[::-1]

        # Condition 1: flag bits must be '00'
        if not reversed_bitstring.startswith("00"):
            continue

        # Extract the 4 data qubits
        data_bits = reversed_bitstring[2:6]

        # Condition 2: even number of ones
        if data_bits.count("1") % 2 != 0:
            continue

        new_data[data_bits] += count

    return dict(new_data)


def decode(counts: Mapping[str, float]) -> dict[str, float]:
    """
    - Assume each key has 4 bits (from process_counts output).
    - Apply the [[4,2,2]] code map to decode into two logical bits.
    """
```

```
    physical_to_logical = {
        "0000": "00",
        "1111": "00",
        "0011": "01",
        "1100": "01",
        "0101": "10",
        "1010": "10",
        "0110": "11",
        "1001": "11",
    }

    logical_counts = defaultdict(float)

    for key, val in counts.items():
        logical_key = physical_to_logical.get(key)
        if logical_key is not None:
            logical_counts[logical_key] += val

    return dict(logical_counts)


# Expectation value for the X observable
def ev_x(counts: Mapping[str, float]) -> float:
    ev = 0.0
    for k, val in counts.items():
        ev += val * ((-1)**int(k[0]) + (-1)**int(k[1]))
    total = sum(counts.values())
    ev /= total
    return ev


# Expectation value for the XX observable
def ev_xx(counts: Mapping[str, float]) -> float:
    ev = 0.0
    for k, val in counts.items():
        ev += val * (-1) ** k.count("1")
    total = sum(counts.values())
    ev /= total
    return ev


# Expectation value for the ZZ observable
def ev_zz(counts: Mapping[str, float]) -> float:
    ev = 0.0
    for k, val in counts.items():
        ev += val * (-1) ** k.count("1")
    total = sum(counts.values())
    ev /= total
    return ev


def _aim_energies(
    counts_data: Mapping[tuple[int, int, str], dict[str, float]],
) -> tuple[dict[tuple[int, int], float], dict[tuple[int, int], float]]:
    evxs: dict[tuple[int, int], float] = {}
    evxxs: dict[tuple[int, int], float] = {}
    evzzs: dict[tuple[int, int], float] = {}
    totals: dict[tuple[int, int], float] = {}

    for key, counts in counts_data.items():
        h_params, basis = key
        key_a, key_b = h_params.split(":")
        u, v = int(key_a), int(key_b)
        if basis.startswith("x"):
            evxs[u, v] = ev_x(counts)
            evxxs[u, v] = ev_xx(counts)
        else:
            evzzs[u, v] = ev_zz(counts)
```

```
        totals.setdefault((u, v), 0)
        totals[u, v] += sum(counts.values())

    energies = {}
    uncertainties = {}
    for u, v in evxs.keys() & evzzs.keys():
        string_key = f"{u}:{v}"
        energies[string_key] = u * (evzzs[u, v] - 1) / 4 + v * evxs[u, v]

        uncertainty_xx = 2 * v**2 * (1 + evxxs[u, v]) - u * v * evxs[u, v] / 2
        uncertainty_zz = u**2 * (1 - evzzs[u, v]) / 2

        uncertainties[string_key] = np.sqrt(
            (uncertainty_zz + uncertainty_xx - energies[string_key] ** 2) /
    (totals[u, v] / 2)
        )

    return energies, uncertainties


def aim_logical_energies(data_ordering, counts_list):
    for i, counts in enumerate(counts_list):
        processed = process_counts(counts)
        decoded = decode(processed)
        if not decoded:
            print(f"[DEBUG] No decoded data at step {i}")
        else:
            print(f"[DEBUG] Decoded at step {i}: {decoded}")
    counts_data = {
        data_ordering[i]: decode(process_counts(counts))
        for i, counts in enumerate(counts_list)
    }
    return _aim_energies(counts_data)


def process_counts_unencoded(counts: Mapping[str, float]) -> dict[str, float]:
    """Extract the 2 bits from the measured bitstrings."""
    return {k[::-1]: v for k, v in counts.items()}


def aim_physical_energies(
    data_ordering: object, counts_list: Sequence[dict[str, float]]
) -> tuple[dict[tuple[int, int], float], dict[tuple[int, int], float]]:
    counts_data = {
        data_ordering[i]: process_counts_unencoded(counts)
        for i, counts in enumerate(counts_list)
    }
    return _aim_energies(counts_data)


from typing import Dict, Tuple, List, Any
import os

def _get_energy_diff(
    bf_energies: Dict[str, float],
    physical_energies: Dict[str, float],
    logical_energies: Dict[str, float],
) -> Tuple[List[float], List[float]]:
    physical_energy_diff = []
    logical_energy_diff = []

    # Data ordering following bf_energies keys
    for layer in bf_energies.keys():
        physical_sim_energy = physical_energies[layer]
        logical_sim_energy = logical_energies[layer]
        true_energy = bf_energies[layer]
        u, v = layer.split(":")
```

```
        print(f"Layer=({u}, {v}) has brute-force energy of: {true_energy}")
        print(f"Physical circuit of layer=({u}, {v}) got an energy of:
    {physical_sim_energy}")
        print(f"Logical circuit of layer=({u}, {v}) got an energy of:
    {logical_sim_energy}")
        print("-" * 72)

        if logical_sim_energy < physical_sim_energy:
            print("Logical circuit achieved the lower energy!")
        else:
            print("Physical circuit achieved the lower energy")
        print("-" * 72, "\n")

        physical_energy_diff.append(-1 * (true_energy - physical_sim_energy))
        logical_energy_diff.append(-1 * (true_energy - logical_sim_energy))

    return physical_energy_diff, logical_energy_diff


def submit_aim_circuits(
    circuit_dict: Dict[str, Any],
    *,
    folder_path: str = "future_aim_results",
    shots_count: int = 50000,
    run_async: bool = False,
) -> Dict[str, List[Dict[str, int]]] | None:
    if run_async:
        os.makedirs(folder_path, exist_ok=True)
    else:
        aim_results = {"physical": [], "logical": []}

    for layer in circuit_dict.keys():
        if run_async:
            print(f"Posting circuits associated with layer=('{layer}')")
        else:
            print(f"Running circuits associated with layer=('{layer}')")

        for basis in ("z_basis", "x_basis"):
            if run_async:
                u, v = layer.split(":")
                tmp_physical_results = noisy_simulator.run(
                    circuit_dict[layer]["physical"][basis],
    shots=shots_count).result()
                file =
    open(f"{folder_path}/physical_{basis}_job_u={u}_v={v}_result.txt", "w")
                file.write(str(tmp_physical_results.get_counts()))
                file.close()

                tmp_logical_results = noisy_simulator.run(
                    circuit_dict[layer]["logical"][basis],
    shots=shots_count).result()
                file =
    open(f"{folder_path}/logical_{basis}_job_u={u}_v={v}_result.txt", "w")
                file.write(str(tmp_logical_results.get_counts()))
                file.close()
            else:
                tmp_physical_results = noisy_simulator.run(
                    circuit_dict[layer]["physical"][basis],
                    shots=shots_count).result()
                tmp_logical_results = noisy_simulator.run(
                    circuit_dict[layer]["logical"][basis],
                    shots=shots_count).result()

                aim_results["physical"].append({k: v for k, v in
    tmp_physical_results.get_counts().items()})
                aim_results["logical"].append({k: v for k, v in
    tmp_logical_results.get_counts().items()})

```

```
    if not run_async:
        print("\nCompleted all circuit sampling!")
        return aim_results
    else:
        print("\nAll circuits submitted for async sampling!")


bf_energies = {
    "1:-9": -18.251736027394713,
    "1:-1": -2.265564437074638,
    "1:7": -14.252231964940428,
    "5:-9": -19.293350575766127,
    "5:-1": -3.608495283014149,
    "5:7": -15.305692796870582,
    "9:-9": -20.39007993367173,
    "9:-1": -5.260398644698076,
    "9:7": -16.429650912487233,
}

import numpy as np
import csv
import os

REPEAT_COUNT = 100
csv_file = "energy_comparison_IQMFakeAdonis.csv"

# Complete CSV header
fieldnames = [
    "Repetition Index",
    "Layer (u:v)",
    "Brute-Force Energy",
    "Physical Circuit Energy",
    "Logical Circuit Energy",
    "Logical Lower Energy"
]

# Create (or overwrite) the CSV file with header
with open(csv_file, mode='a', newline='') as csvfile:
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()

# Loop repetitions
for repeat_index in range(51, REPEAT_COUNT+1):
    print(f"\n[INFO] Starting repetition {repeat_index}/{REPEAT_COUNT}")

    sim_circuit_dict = generate_circuit_set()
    aim_sim_data = submit_aim_circuits(sim_circuit_dict)

    circuit_layers = sim_circuit_dict.keys()
    data_ordering = []
    for key in circuit_layers:
        for basis in ("z_basis", "x_basis"):
            data_ordering.append((key, basis))

    sim_physical_energies, _ = aim_physical_energies(data_ordering,
    aim_sim_data["physical"])
    sim_logical_energies, _ = aim_logical_energies(data_ordering,
    aim_sim_data["logical"])

    # Compute differences for log
    _get_energy_diff(bf_energies, sim_physical_energies, sim_logical_energies)

    # Save results into CSV
    with open(csv_file, mode='a', newline='') as csvfile:
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        for layer in sim_physical_energies.keys():
            physical_energy = sim_physical_energies.get(layer)
            logical_energy = sim_logical_energies.get(layer)
```

```
            brute_force_energy = bf_energies.get(layer)
            if physical_energy is not None and logical_energy is not None and
    brute_force_energy is not None:
                logical_lower = logical_energy < physical_energy
                writer.writerow({
                    "Repetition Index": repeat_index,
                    "Layer (u:v)": layer,
                    "Brute-Force Energy": brute_force_energy,
                    "Physical Circuit Energy": physical_energy,
                    "Logical Circuit Energy": logical_energy,
                    "Logical Lower Energy": logical_lower,
                })

print(f"\n CSV file '{csv_file}' created with all {REPEAT_COUNT} repetitions.")
```

Listing 2: Source code for the SIAM study via VQE and partial error-detection

```
import numpy as np
import matplotlib.pyplot as plt
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, transpile
from qiskit.circuit.library import UGate
from qiskit_aer import AerSimulator
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV

def compute_dtw(s, t):
    """Compute DTW distance between sequences s and t."""
    M, N = len(s), len(t)
    f = np.full((M + 1, N + 1), np.inf)
    f[0, 0] = 0
    for i in range(1, M + 1):
        for j in range(1, N + 1):
            cost = abs(s[i-1] - t[j-1])
            f[i, j] = cost + min(f[i - 1, j], f[i, j - 1], f[i - 1, j - 1])
    return f[M, N]

class QuantumReservoirComputer:
    def __init__(self, n_system, n_ancilla, scale_input,
                 interaction_theta=np.pi, num_shots=50000):
        if n_system != 2 or n_ancilla != 2:
            raise NotImplementedError("Only 2 system qubits + 2 ancilla
    supported")
        self.n_system = n_system
        self.n_ancilla = n_ancilla
        self.scale_input = scale_input
        self.interaction_theta = interaction_theta
        self.num_shots = num_shots
        self.simulator = noisy_simulator
        self.ridge_model = None

    def _build_full_circuit(self, input_sequence):
        """Build quantum circuit for the input sequence."""
        T = len(input_sequence)
        sys_qr = QuantumRegister(self.n_system, 'sys')
        anc_qr = QuantumRegister(self.n_ancilla, 'anc')
        cl_reg = ClassicalRegister(2 * T, 'c')
        qc = QuantumCircuit(sys_qr, anc_qr, cl_reg)
        for qb in list(sys_qr) + list(anc_qr):
            qc.initialize([1, 0], qb)
        for t, u in enumerate(input_sequence):
            angle = self.scale_input * u
            qc.rx(angle, sys_qr[0])
            qc.cx(sys_qr[0], sys_qr[1])
            qc.rx(angle, sys_qr[0])
            qc.rz(angle, sys_qr[1])
            qc.cx(sys_qr[0], sys_qr[1])
            qc.cx(sys_qr[0], anc_qr[0])
```

```
            qc.cx(sys_qr[1], anc_qr[1])
            qc.measure(anc_qr[0], cl_reg[2*t])
            qc.measure(anc_qr[1], cl_reg[2*t+1])
            qc.reset(anc_qr[0])
            qc.reset(anc_qr[1])
            qc.barrier()
        return qc

    def _run_reservoir(self, input_sequence):
        """Run the reservoir circuit and collect ancilla measurements."""
        T = len(input_sequence)
        z = np.zeros((T, self.n_ancilla, self.num_shots))
        qc = self._build_full_circuit(input_sequence)
        transpiled = transpile(qc, backend=noisy_simulator,
                               coupling_map=iqm_coupling_map,
                               optimization_level=0)
        job = self.simulator.run(transpiled, shots=self.num_shots, memory=True)
        memory = job.result().get_memory()
        for shot_idx, bstr in enumerate(memory):
            for t in range(T):
                b0 = int(bstr[-(2*t + 1)])
                b1 = int(bstr[-(2*t + 2)])
                z[t, 0, shot_idx] = 1 if b0 == 0 else -1
                z[t, 1, shot_idx] = 1 if b1 == 0 else -1
        reservoir_states = np.mean(z, axis=2)
        return reservoir_states

    def _add_memory(self, X, depth=10):
        """Embed memory into reservoir states (time-delay)."""
        T, N = X.shape
        return np.hstack([X[depth - d:T - d, :] for d in range(depth)])

    def train(self, input_sequence, target_sequence):
        """Train linear readout via Ridge regression with CV."""
        X0 = self._run_reservoir(input_sequence)
        X = self._add_memory(X0, depth=10)
        y = target_sequence[10:].reshape(-1)
        alphas = np.logspace(-4, 2, 8)
        ridge_cv = GridSearchCV(Ridge(), {'alpha': alphas}, cv=10)
        ridge_cv.fit(X, y)
        self.ridge_model = ridge_cv.best_estimator_

    def predict(self, input_sequence):
        """Predict output sequence given new inputs."""
        X0 = self._run_reservoir(input_sequence)
        X = self._add_memory(X0, depth=10)
        return self.ridge_model.predict(X)

def generate_narma10(length, seed=42):
    """Generate NARMA10 benchmark dataset."""
    np.random.seed(seed)
    u = np.random.uniform(0, 0.5, length)
    y = np.zeros(length)
    alpha, beta, gamma, delta = 0.3, 0.05, 1.5, 0.1
    for t in range(9, length -1):
        y[t+1] = (alpha * y[t]
                  + beta * y[t] * np.sum(y[t-9:t+1])
                  + gamma * u[t-9] * u[t] + delta)
    return u, y

# --- MAIN SCRIPT ---
train_len, test_len, washout = 300, 200, 75
total = train_len + test_len
u, y = generate_narma10(total)
u_train, y_train = u[washout:train_len], y[washout:train_len]
u_test, y_test = u[train_len:], y[train_len:]

best = {'nmse': np.inf}
```

```
scales = [np.pi, np.pi/4, np.pi/2, 2*np.pi, 3*np.pi]
results = []

for scale in scales:
    print(f"\n=== scale_input = {scale:.3f} ===")
    qrc = QuantumReservoirComputer(2, 2, scale_input=scale,
                                   interaction_theta=np.pi,
                                   num_shots=8192)
    qrc.train(u_train, y_train)
    pred_train = qrc.predict(u_train)
    pred_test = qrc.predict(u_test)

    # --- Evaluate performance ---
    y_train_eval = y_train[10:]
    M_eval_train = len(y_train_eval)
    nmse_tr = np.sum((pred_train - y_train_eval) ** 2) / M_eval_train
    dtw_tr = compute_dtw(pred_train, y_train_eval) / M_eval_train

    y_test_eval = y_test[10:]
    M_eval_test = len(y_test_eval)
    nmse_te = np.sum((pred_test - y_test_eval) ** 2) / M_eval_test
    dtw_te = compute_dtw(pred_test, y_test_eval) / M_eval_test

    print(f"M_eval (train) = {M_eval_train}, M_eval (test) = {M_eval_test}")
    print(f"NMSE  (train)  = {nmse_tr:.7e}")
    print(f"DTW   (train)  = {dtw_tr:.7e}")
    print(f"NMSE  (test)   = {nmse_te:.7e}")
    print(f"DTW   (test)   = {dtw_te:.7e}")

    results.append({
        'scale': scale,
        'nmse': nmse_te,
        'dtw': dtw_te,
        'nmse_train': nmse_tr,
        'dtw_train': dtw_tr,
        'model': qrc,
        'pred_test': pred_test,
        'pred_train': pred_train
    })

# --- Normalized score selection ---
nmse_vals = np.array([r['nmse'] for r in results])
dtw_vals = np.array([r['dtw'] for r in results])
nmse_z = (nmse_vals - nmse_vals.mean()) / nmse_vals.std()
dtw_z = (dtw_vals - dtw_vals.mean()) / dtw_vals.std()
scores = nmse_z + dtw_z

best_idx = np.argmin(scores)
best_result = results[best_idx]

print(f"\nBest scale_input = {best_result['scale']:.3f}, "
      f"NMSE = {best_result['nmse']:.7e}, DTW = {best_result['dtw']:.7e}")

# --- Plot results ---
pred_tr = best_result['model'].predict(u_train)
pred_te = best_result['model'].predict(u_test)
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 7))

ax1.plot(np.arange(total), u, label='Input signal $u(t)$')
ax1.plot(np.arange(total), y, label='Target signal $y(t)$')
ax1.axvline(train_len, color='gray', ls='--', label='Train/Test split')
ax1.set_title('Input and Target Signals')
ax1.set_xlabel('Time step')
ax1.set_ylabel('Signal value')
ax1.legend()

ax2.plot(np.arange(washout + 10, train_len), y_train[10:], 'b-', label='Training
    target')
```

```
ax2.plot(np.arange(washout + 10, train_len), pred_tr, 'g--', label='Training
    prediction')
ax2.plot(np.arange(train_len + 10, total), y_test[10:], 'r-', label='Test target')
ax2.plot(np.arange(train_len + 10, total), pred_te, color='orange',
    linestyle='--', label='Test prediction')
ax2.axvline(train_len, color='gray', ls='--', label='Train/Test split')
ax2.set_title('Target vs Predicted Output')
ax2.set_xlabel('Time step')
ax2.set_ylabel('Signal value')
ax2.legend()

plt.tight_layout()
plt.savefig("qrc_output3_IQMFakeAdonis.png", dpi=300)
plt.show()

import csv, os

# --- Save results as CSV ---
image_filename = "qrc_output3_IQMFakeAdonis.png"
csv_filename = os.path.splitext(image_filename)[0] + ".csv"

with open(csv_filename, mode='w', newline='') as csv_file:
    writer = csv.writer(csv_file)
    writer.writerow(["scale_input", "nmse_train", "dtw_train", "nmse_test",
    "dtw_test"])
    for r in results:
        writer.writerow([r['scale'], r['nmse_train'], r['dtw_train'], r['nmse'],
    r['dtw']])

print(f"\nResults saved in: {csv_filename}")
```

Listing 3: Source code for the Quantum Reservoir Computing experiment and NARMA10 benchmark

```
    import numpy as np
import matplotlib.pyplot as plt
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, transpile
from qiskit.circuit.library import UGate
from qiskit_aer import AerSimulator
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV
import csv
import os

# --- Dynamic Time Warping (DTW) function ---
def compute_dtw(s, t):
    M, N = len(s), len(t)
    f = np.full((M + 1, N + 1), np.inf)
    f[0, 0] = 0
    for i in range(1, M + 1):
        for j in range(1, N + 1):
            cost = abs(s[i-1] - t[j-1])
            f[i, j] = cost + min(f[i - 1, j], f[i, j - 1], f[i - 1, j - 1])
    return f[M, N]


class QuantumReservoirComputer:
    def __init__(self, n_system, n_ancilla, scale_input,
                 interaction_theta=np.pi, num_shots=50000):
        if n_system != 2 or n_ancilla != 2:
            raise NotImplementedError("Only 2 system qubits + 2 ancilla qubits
    are supported")
        self.n_system = n_system
        self.n_ancilla = n_ancilla
        self.scale_input = scale_input
        self.interaction_theta = interaction_theta
```

```
        self.num_shots = num_shots
        self.simulator = backend
        self.ridge_model = None

    # --- Build the full quantum circuit for a given input sequence ---
    def _build_full_circuit(self, input_sequence):
        T = len(input_sequence)
        sys_qr = QuantumRegister(self.n_system, 'sys')
        anc_qr = QuantumRegister(self.n_ancilla, 'anc')
        cl_reg = ClassicalRegister(2 * T, 'c')
        qc = QuantumCircuit(sys_qr, anc_qr, cl_reg)

        # Initialize all qubits in |0>
        for qb in list(sys_qr) + list(anc_qr):
            qc.initialize([1, 0], qb)

        # Time evolution with input encoding
        for t, u in enumerate(input_sequence):
            angle = self.scale_input * u

            qc.cx(sys_qr[0], sys_qr[1])
            for qb in sys_qr:
                qc.append(UGate(angle, angle, angle), [qb])
            qc.cx(anc_qr[0], sys_qr[0])
            for qb in anc_qr:
                qc.append(UGate(angle, angle, angle), [qb])
            for qb in sys_qr:
                qc.append(UGate(angle, angle, angle), [qb])
            qc.cx(sys_qr[0], sys_qr[1])
            for qb in sys_qr:
                qc.append(UGate(angle, angle, angle), [qb])

            # Mid-circuit measurement and reset of ancilla
            qc.barrier()
            qc.measure(anc_qr[0], cl_reg[2*t])
            qc.measure(anc_qr[1], cl_reg[2*t+1])
            qc.reset(anc_qr[0])
            qc.reset(anc_qr[1])
            qc.barrier()
        return qc

    # --- Execute the reservoir and collect the averaged output states ---
    def _run_reservoir(self, input_sequence):
        T = len(input_sequence)
        z = np.zeros((T, self.n_ancilla, self.num_shots))
        qc = self._build_full_circuit(input_sequence)
        transpiled = transpile(qc, backend=backend)
        job = self.simulator.run(transpiled, shots=self.num_shots, memory=True)
        memory = job.result().get_memory()
        for shot_idx, bstr in enumerate(memory):
            for t in range(T):
                b0 = int(bstr[-(2*t + 1)])
                b1 = int(bstr[-(2*t + 2)])
                z[t, 0, shot_idx] = 1 if b0 == 0 else -1
                z[t, 1, shot_idx] = 1 if b1 == 0 else -1
        reservoir_states = np.mean(z, axis=2)
        return reservoir_states

    # --- Add memory depth to the reservoir states ---
    def _add_memory(self, X, depth=10):
        T, N = X.shape
        return np.hstack([X[depth - d:T - d, :] for d in range(depth)])

    # --- Train using ridge regression with cross-validation ---
    def train(self, input_sequence, target_sequence):
        X0 = self._run_reservoir(input_sequence)
        X = self._add_memory(X0, depth=10)
        y = target_sequence[10:].reshape(-1)
```

```
        alphas = np.logspace(-4, 2, 8)
        ridge_cv = GridSearchCV(Ridge(), {'alpha': alphas}, cv=7)
        ridge_cv.fit(X, y)
        self.ridge_model = ridge_cv.best_estimator_

    # --- Predict outputs using the trained model ---
    def predict(self, input_sequence):
        X0 = self._run_reservoir(input_sequence)
        X = self._add_memory(X0, depth=10)
        return self.ridge_model.predict(X)


# --- Generate NARMA10 sequence ---
def generate_narma10(length, seed=42):
    np.random.seed(seed)
    u = np.random.uniform(0, 0.5, length)
    y = np.zeros(length)
    alpha, beta, gamma, delta = 0.3, 0.05, 1.5, 0.1
    for t in range(9, length - 1):
        y[t+1] = (alpha * y[t]
                  + beta * y[t] * np.sum(y[t-9:t+1])
                  + gamma * u[t-9] * u[t] + delta)
    return u, y


# --- MAIN ---
train_len, test_len, washout = 300, 200, 75
total = train_len + test_len
u, y = generate_narma10(total)
u_train, y_train = u[washout:train_len], y[washout:train_len]
u_test, y_test = u[train_len:], y[train_len:]

results = []
for scale in [np.pi, np.pi/4, np.pi/2, 2*np.pi, 3*np.pi, 4*np.pi, 5*np.pi]:
    print(f"\n=== scale_input = {scale:.3f} ===")
    qrc = QuantumReservoirComputer(2, 2, scale_input=scale,
                                   interaction_theta=np.pi,
                                   num_shots=8192)
    qrc.train(u_train, y_train)
    pred_train = qrc.predict(u_train)
    pred_test = qrc.predict(u_test)

    y_train_eval = y_train[10:]
    M_eval_train = len(y_train_eval)
    nmse_tr = np.sum((pred_train - y_train_eval) ** 2) / M_eval_train
    dtw_tr = (compute_dtw(pred_train, y_train_eval)) / M_eval_train

    # --- TEST ---
    y_test_eval = y_test[10:]
    M_eval_test = len(y_test_eval)
    nmse_te = np.sum((pred_test - y_test_eval) ** 2) / M_eval_test
    dtw_te = (compute_dtw(pred_test, y_test_eval)) / M_eval_test

    # --- PRINT RESULTS ---
    print(f"M_eval (train) = {M_eval_train}, M_eval (test) = {M_eval_test}")
    print(f"NMSE  (train)  = {nmse_tr:.7e}")
    print(f"DTW   (train)  = {dtw_tr:.7e}")
    print(f"NMSE  (test)   = {nmse_te:.7e}")
    print(f"DTW   (test)   = {dtw_te:.7e}")

    results.append({
        'scale': scale,
        'nmse': nmse_te,
        'dtw': dtw_te,
        'nmse_train': nmse_tr,
        'dtw_train': dtw_tr,
        'model': qrc,
        'pred_test': pred_test,
```

```
        'pred_train': pred_train
    })

# --- Normalization of NMSE and DTW for combined selection ---
nmse_vals = np.array([r['nmse'] for r in results])
dtw_vals = np.array([r['dtw'] for r in results])

nmse_z = (nmse_vals - nmse_vals.mean()) / nmse_vals.std()
dtw_z = (dtw_vals - dtw_vals.mean()) / dtw_vals.std()

scores = nmse_z + dtw_z
best_idx = np.argmin(scores)
best_result = results[best_idx]

print(f"\nBest scale_input = {best_result['scale']:.3f}, NMSE =
    {best_result['nmse']:.7e}, DTW = {best_result['dtw']:.7e}")

# --- Plot results ---
pred_tr = best_result['model'].predict(u_train)
pred_te = best_result['model'].predict(u_test)

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 7))

# Upper subplot: Input and Target
ax1.plot(np.arange(total), u, label='Input signal $u(t)$')
ax1.plot(np.arange(total), y, label='Target signal $y(t)$')
ax1.axvline(train_len, color='gray', ls='--', label='Train/Test split')
ax1.set_title('Input and Target Signals')
ax1.set_xlabel('Time step')
ax1.set_ylabel('Signal value')
ax1.legend()

# Lower subplot: Target vs Prediction
t_train = np.arange(washout + 10, train_len)
t_test = np.arange(train_len + 10, total)
ax2.plot(t_train, y_train[10:], 'b-', label='Training target')
ax2.plot(t_train, pred_tr, 'g--', label='Training prediction')
ax2.plot(t_test, y_test[10:], 'r-', label='Test target')
ax2.plot(t_test, pred_te, color='orange', linestyle='--', label='Test prediction')
ax2.axvline(train_len, color='gray', ls='--', label='Train/Test split')
ax2.set_title('Target vs Predicted Output')
ax2.set_xlabel('Time step')
ax2.set_ylabel('Signal value')
ax2.legend()

plt.tight_layout()
plt.savefig("qrc_output2_MERA_SPARK.png", dpi=300)
plt.show()

# --- Save results to CSV ---
image_filename = "qrc_output2_MERA_SPARK.png"
csv_filename = os.path.splitext(image_filename)[0] + ".csv"

with open(csv_filename, mode='w', newline='') as csv_file:
    writer = csv.writer(csv_file)
    writer.writerow(["scale_input", "nmse_train", "dtw_train", "nmse_test",
    "dtw_test"])
    for r in results:
        writer.writerow([r['scale'], r['nmse_train'], r['dtw_train'], r['nmse'],
    r['dtw']])

print(f"\nResults saved in: {csv_filename}")
```

Listing 4: Source code for the Quantum Reservoir Computing experiment and `NARMA10` benchmark with MERA

Bibliography

[1] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. 10th Anniversary Edition. Cambridge: Cambridge University Press, 2010. ISBN: 9781107002173 (cit. on pp. 3, 7–9, 13, 18, 38, 44).

[2] Giovanna Turvani. `Personal course materials`. Lecture slides from the course "Quantum Hardware Design and Optimization", Master's Degree in Quantum Engineering, Politecnico di Torino, Academic Year 2024/2025 (cit. on pp. 12, 17).

[3] Yaru Wang, Haodong Jiang, Hong Wang, and Qianheng Duan. «An improved quantum algorithm for the quantum learning with errors problem». In: *Quantum Information Processing* 21 (Apr. 2022). DOI: `10.1007/s11128-022-03507-8` (cit. on p. 15).

[4] Robert Loredo. *Learn Quantum Computing with Python and IBM Quantum Experience. A hands-on introduction to quantum computing and writing your own quantum programs with Python*. Packt Publishing Ltd., 2020. ISBN: 978-1-83898-100-6 (cit. on p. 15).

[5] Jules Tilly et al. «The Variational Quantum Eigensolver: a review of methods and best practices». In: *arXiv preprint arXiv:2111.05176* (2022) (cit. on pp. 18, 22).

[6] Nilanjana Datta. *Quantum Information and Computation: The Quantum Fourier Transform and Periodicities*. Lecture Notes, Part IIC Lent Term 2019–2020. 2020 (cit. on pp. 23, 24).

[7] Daniel Gottesman. «Stabilizer Codes and Quantum Error Correction». PhD thesis. California Institute of Technology, 1997. URL: `https://arxiv.org/pdf/quant-ph/9705052v1` (cit. on pp. 25, 46).

[8] Susan Loepp and William K. Wootters. *Protecting Information: From Classical Error Correction to Quantum Cryptography*. Cambridge University Press, 2006. ISBN: 9781139457668 (cit. on p. 27).

[9] Victor V. Albert, Philippe Faist, Alexander Barg, Daniel Gottesman, Leonid Pryadko, et al. *Error Correction Zoo*. 2024. URL: `https://errorcorrectionzoo.org/` (cit. on pp. 38, 41).

[10] Atsushi Hu, Joey Li, and Rebecca Shapiro. *Quantum Benchmarking on the [[4,2,2]] Code*. Tech. rep. Duke University, DOmath 2018, July 2018. URL: https://sites.math.duke.edu/DOmath/DOmath2018/hu-li-shapiro.pdf (cit. on p. 39).

[11] Arthur Pesah. *An Interactive Introduction to the Surface Code*. https://arthurpesah.me/blog/2023-05-13-surface-code/. May 2023 (cit. on p. 42).

[12] Meenambika Gowrishankar, Daniel Claudino, Jerimiah Wright, and Travis Humble. «Logical Error Rates for a [[4,2,2]]-encoded Variational Quantum Eigensolver Ansatz». In: *arXiv preprint arXiv:2405.03032* (2025). Version 2, submitted on 14 Jan 2025. URL: https://arxiv.org/pdf/2405.03032v2 (cit. on p. 47).

[13] Neil W. Ashcroft and N. David Mermin. *Solid State Physics*. Brooks/Cole, a part of Cengage Learning, 1976. ISBN: 9788131500521 (cit. on p. 48).

[14] Renato S. Gonnelli. Personal course materials. Lecture slides from the course "Fisica dello stato solido (Solid State Physics)", Bachelor's Degree in Physical Engineering, Politecnico di Torino, Academic Year 2022/2023 (cit. on pp. 48, 49).

[15] Erik Piatti. Personal course materials. Lecture slides from the course "Quantum Condensed Matter Physics", Master's Degree in Quantum Engineering, Politecnico di Torino, Academic Year 2023/2024 (cit. on pp. 48–50, 52).

[16] Daniele Torsello. Personal course materials. Lecture slides from the course "Quantum Devices", Master's Degree in Quantum Engineering, Politecnico di Torino, Academic Year 2023/2024 (cit. on pp. 48, 56, 57).

[17] Jami Rönkkö et al. «On-premises superconducting quantum computer for education and research». In: *EPJ Quantum Technology* (2024). URL: https://doi.org/10.1140/epjqt/s40507-024-00243-z (cit. on pp. 58, 60, 61, 63).

[18] Philip Krantz, Morten Kjaergaard, Fei Yan, Terry P. Orlando, Simon Gustavsson, and William D. Oliver. «A Quantum Engineer's Guide to Superconducting Qubits». In: *Applied Physics Reviews* (2019). Updated draft dated July 9, 2021. URL: https://arxiv.org/pdf/1904.06560 (cit. on p. 59).

[19] Daniel Gottesman. «Quantum fault tolerance in small experiments». In: *arXiv preprint arXiv:1610.03507* (2016). URL: https://doi.org/10.48550/arXiv.1610.03507 (cit. on p. 65).

[20] Divyanshu Singh and Shiroman Prakash. «Fault-Tolerant Implementation of the Deutsch-Josza Algorithm». In: *arXiv preprint arXiv:2412.04791* (December 6, 2024). URL: https://arxiv.org/pdf/2412.04791 (cit. on pp. 67–70).

[21] Michael Wolfgang Kinza. «Single Impurity Anderson Model and Dynamical Mean Field Theory: A Functional Renormalization Group Study». PhD thesis. RWTH Aachen University, December 17, 2013. URL: https://publications.rwth-aachen.de/record/229234/files/4979.pdf (cit. on p. 82).

[22] P. W. Anderson. «Localized magnetic states in metals». In: *Physical Review* 124.1 (1961), pp. 41–53. DOI: 10.1103/PhysRev.124.41 (cit. on p. 82).

[23] J. R. Schrieffer and P. A. Wolff. «Relation between the Anderson and Kondo Hamiltonians». In: *Physical Review* 149.2 (1966), pp. 491–492. DOI: 10.1103/PhysRev.149.491 (cit. on p. 83).

[24] A. C. Hewson. *The Kondo Problem to Heavy Fermions*. Cambridge University Press, 1993 (cit. on p. 83).

[25] Matt J. Bedalov, Matt Blakely, and Peter D. et al. Buttler. «Fault-Tolerant Operation and Materials Science with Neutral Atom Logical Qubits». In: *arXiv preprint arXiv:2412.07670* (December 10, 2024). URL: https://arxiv.org/pdf/2412.07670 (cit. on pp. 83–85).

[26] Sergey B. Bravyi and Alexei Yu. Kitaev. «Fermionic quantum computation». In: *Annals of Physics* 298.1 (2002), pp. 210–226 (cit. on p. 83).

[27] Peter J. J. O'Malley et al. «Scalable Quantum Simulation of Molecular Energies». In: *Physical Review X* 6.3 (2016), p. 031007. DOI: 10.1103/PhysRevX.6.031007 (cit. on p. 84).

[28] Toshiki Yasuda, Yudai Suzuki, Tomoyuki Kubota, Kohei Nakajima, Qi Gao, Wenlong Zhang, Satoshi Shimono, Hendra I. Nurdin, and Naoki Yamamoto. *Quantum reservoir computing with repeated measurements on superconducting devices*. arXiv preprint arXiv:2310.06706v1. October 10, 2023. arXiv: 2310.06706 [quant-ph]. URL: https://arxiv.org/abs/2310.06706v1 (cit. on pp. 94, 95, 98, 99, 101, 102).

[29] Xiaoyu Guo, Takahiro Muta, and Jianjun Zhao. «Quantum Circuit Ansatz: Patterns of Abstraction and Reuse of Quantum Algorithm Design». In: (Dec. 2024). arXiv:2405.05021. DOI: 10.48550/arXiv.2405.05021. URL: https://doi.org/10.48550/arXiv.2405.05021 (cit. on pp. 98, 100).