Politecnico di Torino

Politecnico di Torino

Master of science-level of the Bologna process in Quantum Engineering

A.a. 2025/2026

Graduation Session: March/April 2026

Partially fault-tolerant QAOA for max-cut and portfolio optimization

Supervisors:
Prof. Riccardo ADAMI
Dr. Emanuele DRI
Dr. Giacomo VITALI

Candidate:
Marco PENNAZIO

Abstract

This thesis addresses the critical challenge of executing variational quantum algorithms on noisy intermediate-scale quantum (NISQ) devices by developing and validating a custom error detection architecture. This architecture is based on the [**4**,**2**,**2**] encoding scheme, that in our case is specifically tailored for the Quantum Approximate Optimization Algorithm (QAOA) and applied to the Portfolio Optimization problem.

The methodology first employed the max-cut problem (4 node ring graph) as a pilot case. The encoded circuit successfully reproduced the known optimal bit strings (0101 and 1010), confirming the correct implementation of the custom [**4**,**2**,**2**] encoding and decoding logic. This required implementing a dedicated library of redefined quantum gates and a decoding function to remap the sampled physical states back into the original logical code space.

For the core portfolio optimization problem, we implemented the necessary Hamiltonian structure using QAOA with three layers, avoiding excessive circuit depth. A specialized post-selection filter was designed to enhance solution quality, particularly where the native QAOA proved inefficient for this complex problem. Comparative analyses were conducted on both 4 asset (plotted against a 10000 Monte Carlo possible portfolios, generated in a Markowitz Efficient Frontier) and 3 asset portfolios (visualized using a star plot).

The study compared the encoded and unencoded QAOA performance across noiseless, simulated noisy, and real quantum hardware (the `ibm_torino` device). Results confirmed that despite the greater circuit depth of the encoded system, which introduced sampling discrepancies, the core solutions remained consistent with theoretical predictions. The efficacy of the error detection was quantified via post-selection rates: approximately 12.66% for the simulated noisy model and 12.57% for execution on the real `ibm_torino` hardware.

These findings validate the successful design and implementation of the [**4**,**2**,**2**] architecture for the use case under analysis, showing the potential of applying error detection to non-trivial optimization problems and of partially fault-tolerant applications on near-term quantum processors.

Acknowledgements

I would like to extend my particular gratitude to my co-supervisors, Emanuele Dri and Giacomo Vitali, and the entire team at Fondazione LINKS. Thank you for welcoming me, for your constant guidance, and for providing the necessary technical support throughout this project. You cultivated an inclusive and encouraging environment where I felt accepted and free to express myself.

A special thank you goes to Matteo, for working alongside me on this project and for being a reliable point of reference whenever I needed advice.

To my main supervisor, Professor Riccardo Adami, I offer my sincere thanks for being an excellent educator who taught me the beauty and enduring value of culture and research. He has been a true inspiration, always available, and bravely taking on a new challenge outside his immediate field of expertise, proving himself to be a truly wonderful person. My deepest and most sincere thanks go to everyone who has been close to me during these past five years, especially my family—those who watched me grow and who made sacrifices to allow me to study and secure a future better than they could have ever desired for themselves.

To my parents, Barbara and Daniele: thank you for your unwavering belief in me and for teaching me to always believe in what I do, to overcome obstacles, and to always dream big.

To my little sister Sofia, for always being nearby and for being one of the greatest joys in my life.

To my grandparents, Ida, Ettore, Franca, Giovanni, and my great-grandparents, Domenico, Teresina, Maria and Cesare, for raising me and for being role models, making my childhood an especially happy period.

To my uncles, Federico and Andrea, for gifting me beautiful moments of fun and lightheartedness, which helped ease the burden of more difficult times.

A special thank you to my fiancée, Giulia, for teaching me the beauty of sharing time and deep affection with a person who accepts you completely and in the simplicity of who you are.

I also thank everyone at the Supernova for being a constant presence over these six years, and a necessary part of my daily life. Thank you for helping me truly understand the beauty of making music together and fulfilling my need to express

myself freely and creatively in the fields of art, songwriting, and music.

Finally, thank you to all my friends who made my university years beautiful and unforgettable, sharing moments of joy and pleasantness. You made this entire experience unique and memorable, and I will carry it with me always.

Table of Contents

List of Figures VII

Glossary XI

1 Introduction 1

1.1 Quantum Computing . . . 1

1.1.1 Introduction: the dawn of a new computational era . . . 1

1.1.2 The quantum revolution: a paradigm shift . . . 1

1.1.3 The rise of classical computing: from war efforts to data processing . . . 2

1.1.4 The emergence of quantum computing . . . 2

1.2 Quantum circuits with variational approaches . . . 4

1.2.1 Variational Quantum Algorithms (VQA) . . . 5

1.3 Quantum Approximate Optimization Algorithm (QAOA) . . . 6

1.3.1 Introduction to QAOA . . . 6

1.3.2 Hamiltonian construction and computational basis mapping 7

1.3.3 Algebraic (matrix) form of the Schrödinger equation . . . 7

1.3.4 Superposition implementation . . . 8

1.3.5 Ansatz design . . . 8

1.3.6 Optimizer . . . 12

1.3.7 Overall QAOA structure . . . 12

1.4 Fault-tolerant circuit implementation and quantum error detection . 14

1.4.1 Types of decoherence . . . 14

1.4.2 Fault-tolerant quantum computing (FTQC) . . . 15

1.4.3 Quantum Error Detection (QED) . . . 16

1.5 Problems and applications . . . 19

1.5.1 Max-Cut . . . 19

1.5.2 Portfolio Optimization . . . 22

1.6 Tools adopted . . . 25

1.6.1 Qiskit . . . 26

1.6.2 CUDA-Q . . . 26

2 Methodology 27
2.1 Research objective and context 27
2.1.1 Final objective: Portfolio Optimization 27
2.1.2 Quantum hardware constraints and encoding 27
2.2 Algorithms development 28
2.2.1 Initial test case: Max-Cut problem 28
2.2.2 Portfolio Optimization: encoded and unencoded analysis . . 37
2.2.3 In-depth analysis of the quantum hardware ibm_torino . . . 56

3 Results 60
3.1 Results from Max-Cut 60
3.2 Results from unencoded implementation using Qiskit's built-in methods 60
3.3 Results using `QAOAAnsatz()` and custom post-selection 68
3.3.1 Expected impact on solution quality 71
3.3.2 Results from the unencoded noiseless circuit implementation with QAOAAnsatz() and post-selection filtering 72
3.4 Results from the encoded noiseless circuit implementation 78
3.5 Noisy simulation results 83
3.6 Hardware execution results 87

4 Conclusions and future works 95
4.1 Max cut problem: validation of encoding architecture and error detection process 95
4.2 Comparative performance of encoded vs. unencoded Portfolio Optimization 95
4.2.1 Initial unencoded analysis and theoretical congruence 96
4.2.2 Necessity of manual Ansatz and post-selection filter 96
4.2.3 Algorithm efficiency validation through under-sampling . . . 97
4.2.4 Implementation of the encoded Ansatz and dimensionality reduction 97
4.2.5 Analysis of noiseless and noisy results 97
4.2.6 Post-selection metrics and final deduction 98
4.3 Future works 98

Bibliography 100

List of Figures

1.1 The number of different permutations possible within a bitstring increases exponentially with the increasing number of bits in the system. [3] . . . 3

1.2 The classical bit admits only the discrete states '0' and '1', whereas the qubit achieves a 'superposition state' of the two logical ones. [8] 4

1.3 The application of a sequence of Hadamard gates is instrumental in generating a perfect superposition across all computational basis states, thus enabling quantum parallelism [22] . . . 9

1.4 This figure depicts the outcome of Trotterization, demonstrating the discretization of a continuous system, paralleling the quantization process of analog signals into discrete levels [24, p. 57]. . . . 10

1.5 This figure illustrates the components of a Cost Hamiltonian: its linear term (left, Figure a), which models on-site behavior, and its quadratic term (right, Figure b), representing interaction between elements [22]. For optimization, the initial parameters are randomly set to $\frac{\pi}{2}$. . . . 11

1.6 This figure illustrates the Mixer Unitary for a system composed of four qubits [22]. Also here the parameters to optimize are randomly set to $\frac{\pi}{2}$. . . . 12

1.7 The complete architecture of the Quantum Approximate Optimization Algorithm (QAOA), integrating its key components [26, p. 3]. . . 13

1.8 Mathematical formulation of a parameterized single-qubit rotation gate (left) and the bit-flip error channel (right), illustrating unitary dynamics versus stochastic noise processes. . . . 14

1.9 Mathematical representation of the parameterized $R_z(\theta)$ gate (left) and the phase-flip decoherence process (right), showing the exponential suppression of off-diagonal terms in the density matrix. . . . 15

1.10 Diagram illustrating the implementation of a parity check for Quantum Error Detection. [37, p. 197] . . . 16

1.11 Circuit diagram illustrating error detection apparatus, as implemented by Infleqtion [29]. 17
1.12 Schematic representation of a fault-tolerant QAOA system incorporating syndrome measurement layers, similar to approaches developed by institutions like JP Morgan [38]. 18
1.13 Table of various [4,2,2] fault-tolerant gate implementations by Infleqtion [29], illustrating states useful for circuit construction. 20
1.14 Example of an adjacency matrix. 21
1.15 The 4-node ring graph (cycle C_4) with nodes labeled from 0 to 3, starting at the top-left corner and increasing clockwise. 21

2.1 The adjacency matrix **A** for the 10-node test graph 31
2.2 Topology of the **ibm_torino** processor (Heron r1) visualizing the **readout error** for each qubit. The values displayed represent the probability of measurement error for each physical qubit, indicating the spatial variation of readout fidelity across the chip. 58
2.3 Distribution of the **readout error** values per qubit. This histogram quantitatively illustrates the correlation between each physical qubit (ID) and its measured readout error rate within the topology of the real quantum device. 59

3.1 Results from the unencoded Qiskit implementation (100 iterations, budget = 2000 and 10000 shots). 68
3.2 Portfolio optimization results using `QAOAAnsatz()` and custom post-selection (4 assets, 100 iterations) for 10000 shots. 70
3.3 Plot illustrating the Markowitz efficient frontier generated for the portfolio optimization problem. The specific optimization parameters are: 100 iterations, 3 QAOA layers, a budget constraint of Budget = 2000, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$ 71
3.4 Results of the portfolio optimization for the 3 asset case, illustrating the performance of the `QAOAAnsatz()` combined with a custom post-selection filter across optimization depths equal to three ($p = 3$ layers). The results are shown for $1, 3, 5, 20$, and 100 iterations. . . . 78
3.5 Results of the portfolio optimization for the 3 asset case, illustrating the performance of the encoded [4,2,2] Ansatz combined with a custom post-selection filter across optimization depths equal to three ($p = 3$ layers). The results are shown for $3, 5$, and 100 iterations. . . 82

3.6 Results of the portfolio optimization for the 3 asset case, illustrating the performance of the `QAOAAnsatz()` combined with a custom post-selection filter across optimization depths equal to three ($p = 3$ layers) in a noisy simulated environment. The results are shown for 3 iterations. 84
3.7 Results of the portfolio optimization for the 3 asset case, illustrating the performance of the encoded [4,2,2] Ansatz combined with a custom post-selection filter across optimization depths equal to three ($p = 3$ layers) in a noisy simulated environment. The results are shown for 3 iterations. 86
3.8 Results of the portfolio optimization for the 3 asset case, illustrating the performance of the `QAOAAnsatz()` combined with a custom post-selection filter across optimization depths equal to three ($p = 3$ layers) on "**ibm_torino**" real hardware. The results are shown for 3 and 20 iterations. 89
3.9 Results of the portfolio optimization for the 3 asset case, illustrating the performance of the encoded [4,2,2] Ansatz combined with a custom post-selection filter across optimization depths equal to three ($p = 3$ layers) on "**ibm_torino**" real hardware. The results are shown for 3 iterations. 93

Glossary

VQA

Variational Quantum Algorithm

VQE

Variational Quantum Eigensolver

QAOA

Quantum Approximate Optimization Algorithm

QEC

Quantum Error Correction

QED

Quantum Error Detection

QUBO

Quadratic Unconstrained Binary Optimization

Chapter 1

Introduction

1.1 From the birth of quantum mechanics to the modern quantum computing

1.1.1 Introduction: the dawn of a new computational era

The past century has witnessed an extraordinary surge in scientific and technological advancements, particularly in fundamental research and its practical applications. This progress has led to the development of systems that transcend the inherent limitations imposed by classical physics.

1.1.2 The quantum revolution: a paradigm shift

A pivotal breakthrough in this paradigm occurred in 1901 when the German physicist Max Planck made a groundbreaking discovery: the emission spectrum of a "black body" [1] exhibited discrete peaks in its possible emission frequencies. This contradicted the prevailing theories of classical physics, which had, until then, comprehensively described our observable world.

From this moment onward, a multitude of scientists embarked on the study and analysis of these novel phenomena. Their aim was to describe a world invisible to the naked eye, yet one that occasionally manifests itself with macroscopic effects on our reality. This endeavor gave rise to what would eventually become one of the most enigmatic disciplines: "Quantum Physics". Its principles underpin various modern technologies, including superconductors, atomic bombs, radiotherapy, and Light Emitting Diodes (LEDs).

1.1.3 The rise of classical computing: from war efforts to data processing

While quantum mechanics was advancing its fundamental discoveries, the advent of world war II necessitated the creation of rapid and efficient new computational systems. The urgent need to decrypt secret Nazi messages spurred the development of the first such machine, "Enigma" [2], conceived by the mathematician Alan Turing in England. This innovation provided a crucial contribution to the victory of the United Kingdom in the conflict.

Following 1945, nations globally began to recognize the imperative of developing and implementing similar computational systems. These machines were essential to overcome the then insurmountable physical limitations imposed by the vast volume of data humans could process. This marked the genesis of the era of computational systems, more commonly known as "Computers".

These systems operate on a language understood by these "electronic brains", essentially electrical signals that can manifest as either the presence or absence of a signal. This binary nature opened up a vast field of application for Boolean Mathematics, specifically the "binary system", which associates the absence of a signal with a "logical 0" and its presence with a "logical 1". The conversion of a physical signal into a numerical representation, as in this case, enables complex calculations to be performed directly by a machine, achieving significantly faster processing times and relatively high accuracy.

The fundamental unit of these binary systems is termed a "Bit", capable of assuming either of the two aforementioned values. Combining multiple bits exponentially increases the number of possible combinations (as shown in Figure 1.1). Each combination is representable as a sequence of bits, known as a "string".

In classical computing applications, a single string can be evaluated at a time, albeit with varying speeds, but this process is fundamentally constrained by a sequential regime imposed by technological limitations. Currently, significant levels of parallelization can be achieved through the classical High-Performance Computing (HPC) systems utilizing "Clusters" of computers. These clusters offer substantial computational power with high efficiency, though they involve a complex hardware structure and non-negligible maintenance costs [4].

1.1.4 The emergence of quantum computing

Consequently, with the widespread implementation of classical computing machines and the multiple experimental validations of Quantum Mechanics' theories, the 1990s saw the birth of "Quantum Computing". This emerging scientific discipline integrates the computational principles developed thus far with the fundamental principles of atomic physics.

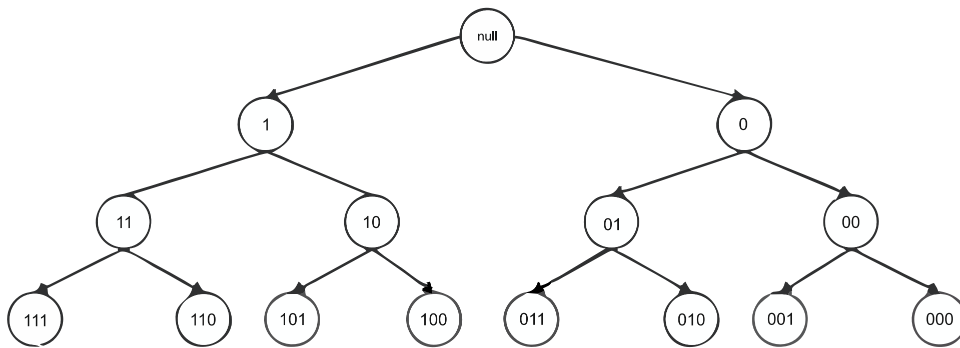


Figure 1.1: The number of different permutations possible within a bitstring increases exponentially with the increasing number of bits in the system. [3]

The foundational principle of quantum computing stems from the discrete nature of atoms (which can only exist in specific configurations known as "States") and the probabilistic interpretation of quantum mechanics [5][6]. This dictates that, under certain conditions, a quantum system can exist in a perfectly balanced "superposition of states". This concept is termed the "superposition principle" and it is graphically represented in Figure 1.2.

Therefore, drawing parallels with the Boolean logic of a bit, this superposition of states can be described as a probability-weighted sum of the states associated with the "logical 0" and "logical 1" of our system. This superimposed state is defined as a "qubit".

In practical terms, this allows for the creation of a qubit string that inherently contains all possible configurations of all classical strings, enabling their simultaneous evaluation.

In computational applications, control operations on bits and qubits are performed using "logic gates" in the classical domain and "quantum gates" in the quantum domain. The latter, unlike their classical counterparts, are physical impulses applied to the atomic system [7].

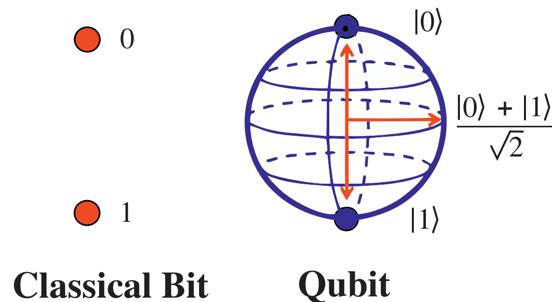


Figure 1.2: The classical bit admits only the discrete states ’0’ and ’1’, whereas the qubit achieves a ’superposition state’ of the two logical ones. [8]

From this principle emerges the objective of creating a "quantum computer" with virtually instantaneous computational efficiency, all at the cost of a single hardware device.

Over the years, numerous research centers and companies have engaged in the construction and development of these devices, leading to the recent advancement and small-scale commercialization of quantum computing systems for academic and corporate research purposes [9].

However, the inherent difficulty in controlling structures at the atomic scale, largely due to decoherence effects caused by "noisy" environments, currently impedes the full utilization and subsequent scalability of quantum-oriented computational systems. Among these challenges, this document will specifically focus on overcoming the limitations of random errors through the use of "fault-tolerant" devices.

1.2 Quantum circuits with variational approaches

In the field of computational physics, one of the fundamental pillars for the development of engineering algorithms is the search for optimal points (maximum and minimum), which falls within the domain of "optimizers". In the realm of

classical algorithms, numerous methods are designed to search for the best possible solutions for a given type of problem as comprehensively and efficiently as possible.

With the introduction of quantum computers and, consequently, quantum-based High-Performance Computing [10, 11], there has been an effort to implement algorithms for solving more or less complex problems for which one or more optimal or pseudo-optimal solutions are sought.

The difficulty in maintaining "coherent" and thus time-constant states for our physical qubits implies a tendency to introduce errors into our system, thereby limiting the possibility of scaling our systems in terms of the number of qubits and circuit depth. These limitations are not negligible during the implementation phase of our algorithms on real circuits. Therefore, the decision was made to adopt "near-term" computing approaches that feature a quantum execution part and a classical part, the latter serving, for instance, to optimize the search for our optimal parameters in circuits with variational approaches.

1.2.1 Variational Quantum Algorithms (VQA)

As previously indicated, Variational Quantum Algorithms (VQA) represent a class of "near-term" algorithms [12, 13] based on the construction of a parameterized quantum circuit and a classical optimization component for its parameters.

The fundamental building blocks that enable the definition and implementation of these particular types of systems are: the construction of an Ansatz (a parameterized circuit) [14, 15], the quantum execution of the parameterized circuit, the classical optimization of the parameters after each execution, and the iteration of this optimization process for an arbitrary number of times based on the needs of the problem.

This class of algorithms demonstrates considerable flexibility in the computational field, offering an effective solution that is comparable to the limitations imposed by our problem and its complexity.

All the points previously discussed regarding the functional blocks (Ansatz construction, etc.) will be further elaborated in the following section dedicated to a specific type of variational algorithm that will be implemented and used for the purposes of this work. Before delving theoretically and notionally into QAOA, we will clarify what this particular type of variational algorithm is and why it is useful and most suitable for our purposes.

1.3 Quantum Approximate Optimization Algorithm (QAOA)

The quest for increasingly optimal solutions to complex problems in applied physics and engineering has driven the research and creation of algorithms designed to meet this need in the simplest and most efficient manner possible.

Over the years, various algorithms have been tested and implemented. Among the most efficient in the realm of computational sciences applied to quantum contexts is the Quantum Approximate Optimization Algorithm (QAOA). This system is based on identifying a minimum or maximum value that is as close as possible to the global optimum, accounting for a small tolerance inherent to the system's complexity. For this reason, it is classified as an "approximate" algorithm.

1.3.1 Introduction to QAOA

The Quantum Approximate Optimization Algorithm (QAOA) [16, 17] is a particular type of variational algorithm created specifically to solve combinatorial optimization problems. It features a functional structure identical to the one presented previously but with a more rigid and synthetic scheme, as it must act directly on combinatorial structures, which are simpler and more manageable than generic ones.

Advantages of QAOA over other algorithms

In recent years, efforts have been made to create algorithms that could offer advantages and compete computationally with classical ones. To date, QAOA has demonstrated excellent performance in these terms, establishing itself as one of the most prominent and effective algorithms computationally, also from the perspective of system scalability potential.

Relative to the current error-prone computational capacity provided by quantum computers, QAOA has shown a significant advantage in solving problems such as Max-Cut for 3-regular graphs on a sufficiently large number of nodes [18] to the point of competing with, or even surpassing, the performance of classical algorithms employed so far.

Furthermore, from this very evidence, it has been deduced and tested that the use of techniques such as warm-starting QAOA with help from Goemans-Williamson [19] and the use of "amplitude-amplification" [20] have led quantum circuits to obtain optimal results by significantly reducing circuit depth and the number of gates used, and thus the possibility of a deterioration of computational capabilities due to decoherence effects, which will be discussed and explained later. The algorithm's solution process involves four distinct phases, which are reported in the immediately following subsections.

1.3.2 Hamiltonian construction and computational basis mapping

Every physical system, particularly a quantum one, can be represented by a mathematical object called a Hamiltonian. This Hamiltonian describes the system's structure and its energetic evolution, especially when dynamic systems are considered.

The most renowned form of the Hamiltonian is found within the Schrödinger equation, which describes the dynamics of any quantum system in nature, representing its evolution and establishing the foundation for studying its internal configuration.

1.3.3 Algebraic (matrix) form of the Schrödinger equation

While the Schrödinger equation is fundamentally based on differential operators acting on wave functions, in contexts such as quantum computation, where states are often represented by vectors in finite-dimensional spaces, a matrix formulation is commonly employed. This approach is especially useful when dealing with systems that have a discrete and finite number of states, like qubits.

In a finite-dimensional quantum system, the state of the system at time t can be represented by a state vector (or column vector), and observables (such as the Hamiltonian) by matrices.

Consider a system residing in an N-dimensional Hilbert space. The state of the system at time t is given by a state vector $|\Psi(t)\rangle$. The time-dependent Schrödinger equation in this algebraic form is represented as in the equation (1.1).

$$i\hbar\frac{d}{dt}|\Psi(t)\rangle = \hat{H}|\Psi(t)\rangle \tag{1.1}$$

Here:

- $|\Psi(t)\rangle$: Represents the state vector, an $N \times 1$ column vector. This vector encapsulates the superposition of basis states.
- $\hat{H}$: Represents the Hamiltonian matrix, an $N \times N$ Hermitian matrix that corresponds to the system's total energy operator.

Case of a single qubit

For a single qubit, which exists in a two-dimensional Hilbert space ($N = 2$), the generic state at time t is given by Equation (1.2).

$$|\Psi(t)\rangle = \alpha(t)|0\rangle + \beta(t)|1\rangle = \begin{pmatrix} \alpha(t) \\ \beta(t) \end{pmatrix} \tag{1.2}$$

Where $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ are the computational basis states. $\alpha(t)$ and $\beta(t)$ are complex probability amplitudes that depend on time.

The Hamiltonian matrix $\hat{H}$ for a single qubit is a 2×2 Hermitian matrix. This algebraic formulation is highly powerful as it allows for the modeling of discrete quantum system evolution and interactions through matrix operations, which form the bedrock of quantum simulators and quantum computers.

In our context, we will deal with Hamiltonians of quadratic systems. These systems exhibit a linear contribution related to the parameters of the individual components and quadratic contributions arising from the multiple interactions between pairs of elements within the overall system (e.g., Max-Cut graphs, discussed in the subsequent section). They may also include penalty terms when referring to constrained physical systems.

Therefore, our initial step for implementing the algorithm is to construct the Hamiltonian based on the morphology of the system under study. Subsequently, it is converted into a computational basis, enabling its processing by our quantum computing apparatus. From this conversion, we can determine the number of qubits required for implementing the corresponding circuit of our algorithm.

1.3.4 Superposition implementation

Once our quantum circuit is initialized, to simultaneously evaluate all possible configurations of our system in accordance with the probabilistic logic of quantum mechanics, we must prepare all qubits in a "superposition state". This is achieved by applying a Hadamard quantum gate [21] to each qubit, bringing it to the desired condition (as can be seen in Figure 1.3).

Furthermore, we randomly initialize a set of default parameters to be applied to our system that we will call "thetas" (as represented in Equation 1.3).

$$\theta_i \quad \text{for } i = 1, \ldots, N_{\text{qubits}} \tag{1.3}$$

These parameters will subsequently be optimized to contribute to the optimal or pseudo-optimal solution of our problem.

1.3.5 Ansatz design

Our ultimate goal is to evaluate an energetic minimum within our system. This is achieved by calculating the expectation value of our Hamiltonian's eigenvalues, through the application of two eigenstates of our system, expressed using bra-ket notation, as presented in Equation (1.4).

$$E = \frac{\langle \psi(\theta) | H | \psi(\theta) \rangle}{\langle \psi(\theta) | \psi(\theta) \rangle}, \quad \forall \theta. \tag{1.4}$$

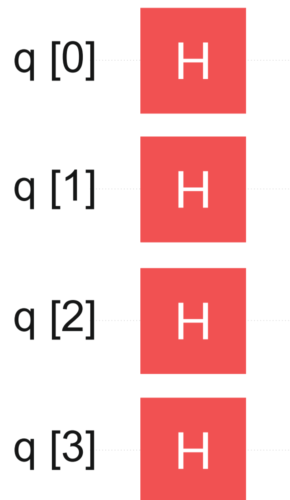


Figure 1.3: The application of a sequence of Hadamard gates is instrumental in generating a perfect superposition across all computational basis states, thus enabling quantum parallelism [22]

These unitary quantum states, which symbolize the evolution of our system, can be translated into our circuit via quantum gates. Each applied quantum gate influences the circuit's depth, meaning an increase in depth will consequently increase the algorithm's execution time.

We can reasonably assume that during the application of each individual gate, the value of our Hamiltonian remains constant over time. This assumption defines the process of "Trotterization" [23, p. 9], which transposes a Hamiltonian, initially defined on a computational basis, into the discrete regime. This effectively creates a step-wise evolution of our energy profile, analogous to the process of quantizing an analog signal for conversion into its digital representation, as depicted in Figure 1.4.

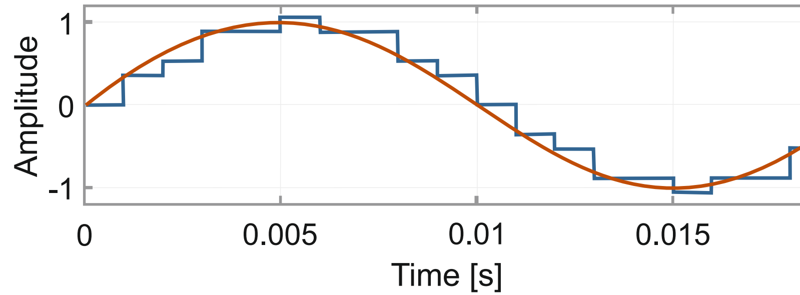


Figure 1.4: This figure depicts the outcome of Trotterization, demonstrating the discretization of a continuous system, paralleling the quantization process of analog signals into discrete levels [24, p. 57].

At this juncture, we consider the two fundamental blocks of our algorithm that will hardware mimic the unitary evolution of our state: the Mixer Unitary and the Cost Hamiltonian.

Cost Hamiltonian

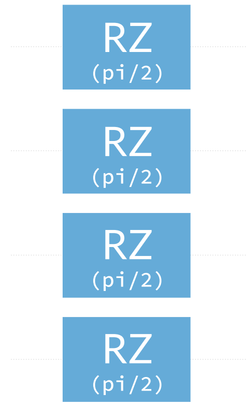
The role of this circuit block is to define, at the gate level, the interactions occurring within our system. This is typically accomplished through controlled $R_z(\theta_i)$ gates, shown in Figure 1.5, which are controllable by the parameter (θ_i) that must be optimized subsequently. This optimization facilitates either a constructive or destructive interference, or more precisely, an interaction between the two states of the interacting qubits (as as can be seen in Figure 1.5b).

Evidently, our interactions must represent the perfect hardware transposition of our Hamiltonian interaction and linear terms, as shown in Figure 1.5a, in the computational basis.

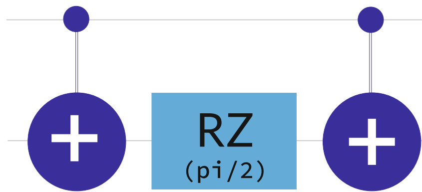
Mixer Unitary

While the Cost Hamiltonian purpose is to allow interaction between the qubits pairs without altering their current states, the Mixer Unitary role is precisely to vary the current states of the qubits across all possible configurations, due to the application of a $R_x(\theta_i)$ gates, on each qubits of our circuit (as depicted in Figure 1.6).

The sequential application of these two blocks, repeated for an arbitrary number of times (defined as the "number of layers") and alternated with the optimization process, facilitates the generation of our unitary evolutionary states, known as



(a) Example of a linear term in the Cost Hamiltonian.



(b) Example of a quadratic term in the Cost Hamiltonian.

Figure 1.5: This figure illustrates the components of a Cost Hamiltonian: its linear term (left, Figure a), which models on-site behavior, and its quadratic term (right, Figure b), representing interaction between elements [22]. For optimization, the initial parameters are randomly set to $\frac{\pi}{2}$.

Figure 1.6: This figure illustrates the Mixer Unitary for a system composed of four qubits [22]. Also here the parameters to optimize are randomly set to $\frac{\pi}{2}$.

"Ansatz", within our system. These Ansatz states enable the definition of our discretized energy profile.

1.3.6 Optimizer

To identify an optimal or pseudo-optimal value for our problem, a classical optimization algorithm is required. This algorithm aims to find an optimal point (a minimum or maximum, depending on the problem defined). Over the years, various types of optimization algorithms have been studied [23, p. 23]. However, for our specific purposes, the gradient-free optimizer named COBYLA was selected, as it proves optimal for our Max-Cut and Portfolio Optimization objectives, which will be discussed later [23, p. 51].

Through the cyclic application of this algorithm after each layer (i.e., after each Mixer Unitary and Cost Hamiltonian), we achieve optimization of our optimal energy profile values under study via "successive approximations".

To derive the measurements pertinent to our system, by projecting the ensemble of states permitted by the superposition principle according to the theories of Von Neumann [25] through a measurement on the qubits of our system, we can obtain the optimal statistical profile of the states and, consequently, their associated energetic eigenvalues.

1.3.7 Overall QAOA structure

In summary, by integrating the concepts previously discussed, we can now present the complete QAOA structure in Figure 1.7.

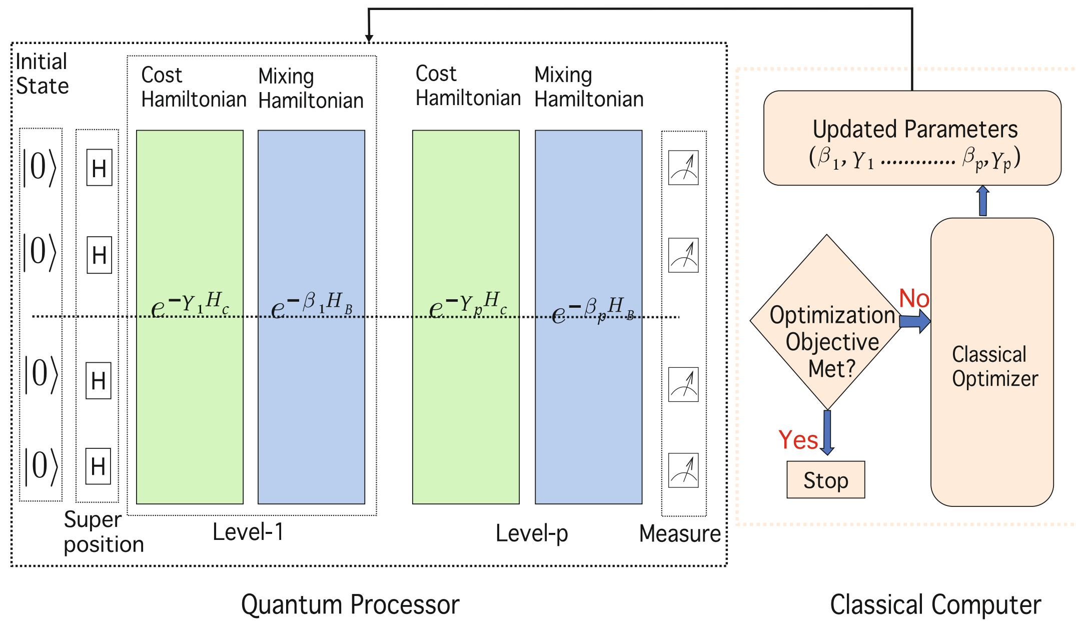


Figure 1.7: The complete architecture of the Quantum Approximate Optimization Algorithm (QAOA), integrating its key components [26, p. 3].

1.4 Fault-tolerant circuit implementation and quantum error detection

Working with atomic-scale systems inherently leads to an exponential increase in their sensitivity to noise. This characteristic is particularly pronounced in quantum computers and, consequently, in all their derived applications.

As briefly discussed in previous sections (or specifically in the subsection 1.1.4), one of the most significant challenges for the control and application of quantum computing systems is the phenomenon of "decoherence". Decoherence causes the intended prepared and processed quantum state to degrade as the dimensions and depth of the quantum circuit increase [27].

Decoherence arises from varying intensities of environmental interactions between our system and its external surroundings. The effects of decoherence can be broadly categorized into "energetic" (or relaxation) and "dephasing" types.

1.4.1 Types of decoherence

Energetic decoherence (bit-flip errors)

Energetic decoherence leads the system to transition from its desired initialization or control state to an undesired one. This occurs due to an energetic "relaxation" or "excitation" that tends to drive the atom towards its naturally more stable energetic conformation. In the context of quantum computing, for instance, this can cause the $|1\rangle$ state to flip to $|0\rangle$ or vice versa, depending on the specific circumstances and the type of environmental stimulus experienced by the system. This phenomenon invariably introduces errors in the final result and is therefore referred to as a "bit-flip"; we can observe a representation of the error model into a quantum channel in Figure 1.8b. In the computational domain, a bit-flip error can be modeled, for the sake of simplicity and for didactic purposes, by an R_x-gate (as can be seen in Figures 1.8a).

$$R_x(\theta) = e^{-i\frac{\theta}{2}\sigma_x} = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) \\ -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix}$$

(a) Matrix representation of the parameterized $R_x(\theta)$ gate.

$$\mathcal{E}(\rho) = (1-p)\rho + p\,X\rho X$$

(b) Kraus representation of the bit-flip quantum channel, with error probability p.

Figure 1.8: Mathematical formulation of a parameterized single-qubit rotation gate (left) and the bit-flip error channel (right), illustrating unitary dynamics versus stochastic noise processes.

Dephasing decoherence (phase-flip errors)

The second type of decoherence, dephasing, impacts the phase of the qubits in our system, leading to spurious values. This deviation from the previously designed and precisely calibrated control logic for a specific purpose generates an undesirable interference phenomenon, favoring states that are also undesired for the system's solution. Consequently, this leads to a degradation in the statistical and qualitative fidelity of the results (as can be seen in Figure 1.9b).

Therefore, in the computational domain, these occurrences are termed "phase-flip" errors. They can be modeled, for didactic purposes, by applying $R_z(\theta)$ gates, as shown in Figure 1.9a, which directly act on the phase of the noise-affected qubits within our system.

$$R_z(\theta) = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix}$$

(a) Matrix representation of the parameterized $R_z(\theta)$ gate.

$$\rho(t) = \begin{bmatrix} \rho_{00}(t=0) & \rho_{01}(t=0)e^{-\Gamma t} \\ \rho_{10}(t=0)e^{-\Gamma t} & \rho_{11}(t=0) \end{bmatrix}$$

(b) Density matrix evolution under phase decoherence, where Γ denotes the dephasing rate.

Figure 1.9: Mathematical representation of the parameterized $R_z(\theta)$ gate (left) and the phase-flip decoherence process (right), showing the exponential suppression of off-diagonal terms in the density matrix.

1.4.2 Fault-tolerant quantum computing (FTQC)

In the years following the discovery of these experimental evidences, numerous scientists and engineers addressed the challenge of creating "Fault-Tolerant" systems [28]. These systems are designed with the capability to be "immune" to errors that could potentially occur within the physical-logical system.

To implement such systems for the purposes outlined in subsequent chapters of this thesis, we will refer to fault-tolerant systems developed by Infleqtion. These systems are designed for creating computational structures for Hamiltonian ground states, particularly relevant in "Computational Chemistry" [29]. However, the underlying principles are broadly applicable to any other research and development area based on these macro-topics.

1.4.3 Quantum Error Detection (QED)

Another fundamental aspect for the future implementation of truly "fault-tolerant" systems is the application of "error detection" subroutines [30, 31, 32, 33, 34]. These encompass the set of processes designed to reveal potential errors that might occur within our quantum architectures.

This field, specifically applied to quantum computing, is known as "Quantum Error Detection". The discipline began to formalize in the mid 1990s with the theoretical contributions of Shore's algorithms [35, 36]. Shore's approach involves both redundancy and encoding of qubits within the system to identify which qubit is affected by the so-called "syndrome", an error that has corrupted the value of a single qubit. However, this method proves to be extremely resource-intensive in terms of the number of qubits required, making it impractical for current quantum computing systems.

Parity check method

One of the simplest and most effective methods for Quantum Error Detection is the "Parity Check" [37, Chapter 5]. This method involves computing the sum modulo two of the values of two qubits in a given state. If the result is $|0\rangle$, it can be deduced that the two qubits possess the same logical value. Conversely, if the sum yields $|1\rangle$, it implies that their logical values are different. We can observe the circuit implementation in Figure 1.10.

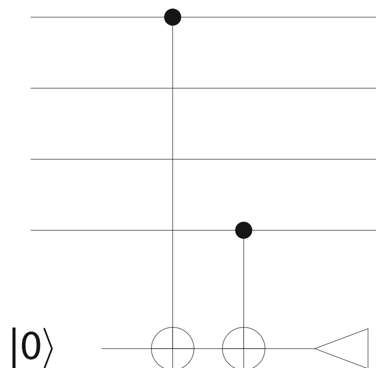


Figure 1.10: Diagram illustrating the implementation of a parity check for Quantum Error Detection. [37, p. 197]

This system proves particularly effective when analyzing two qubits that are intended to undergo the same form of control or initialization. Furthermore, the structure of the sum qubit, referred to as the "ancilla", can be manipulated to

achieve consistency even for values that, due to design and functionality, must inherently differ. Nevertheless, various phenomena can occur atypically and randomly, potentially rendering the error imperceptible to this system.

Implementations of error detection for bit-flip and phase-flip

As previously discussed, the Parity Check can be applied to two arbitrary qubits in our system. This involves applying control gates to our, so called, "ancilla" qubits, which are used to store the sum result for error detection. The measurement is then performed on this ancilla, thereby avoiding the "destruction" of the main qubit values that would occur from direct measurement.

In Figure 1.11, we demonstrate an implementation for a system developed by Infleqtion.

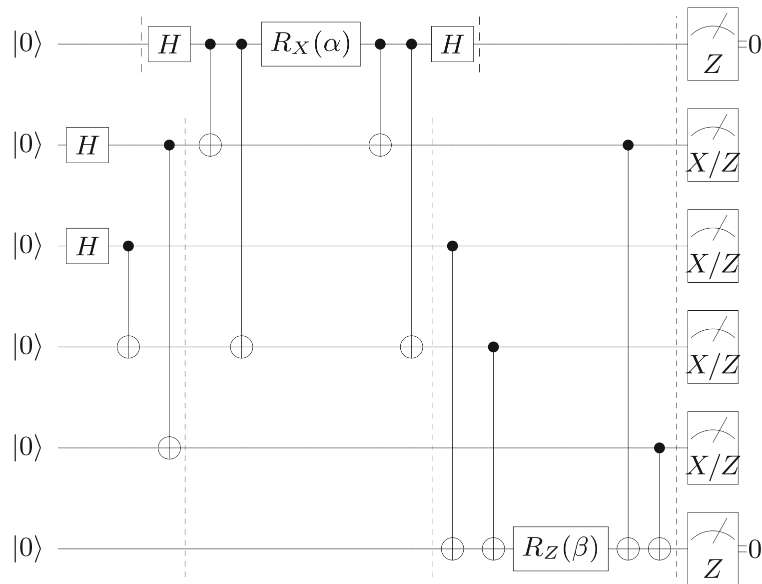


Figure 1.11: Circuit diagram illustrating error detection apparatus, as implemented by Infleqtion [29].

As depicted in Figure 1.11, a "syndrome measurement and error detection" layer can be effectively inserted, interspersed with the process layers that are more functional to the execution of our quantum algorithm.

We then illustrate, in Figure 1.12, a system that leverages a fault-tolerant approach for QAOA, interspersing its computational layers with syndrome measurement layers. This method aims to either confirm or discard the results obtained from the algorithm's execution in noisy computational environments.

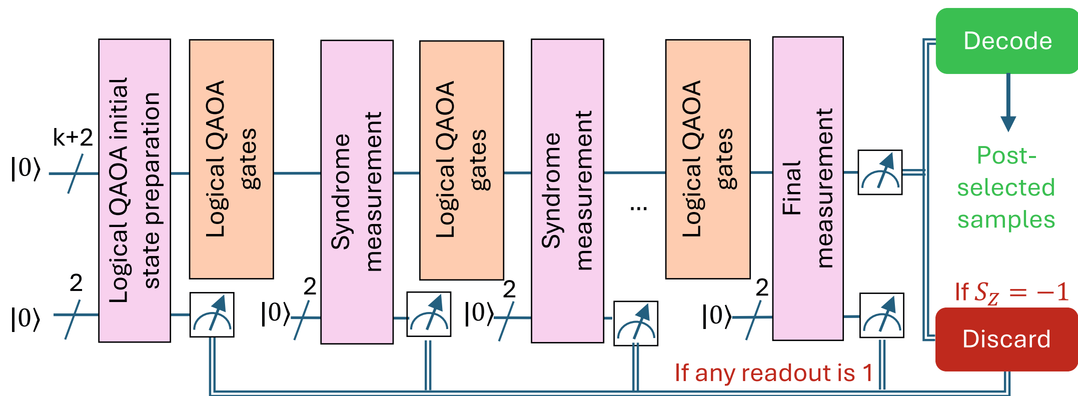


Figure 1.12: Schematic representation of a fault-tolerant QAOA system incorporating syndrome measurement layers, similar to approaches developed by institutions like JP Morgan [38].

These systems, implemented to render a quantum computing system as immune as possible to the occurrence of errors, are generally referred to as "stabilizers". Their

detailed description can be found in Daniel Gottesman's seminal book, "Surviving as a Quantum Computer in a Classical World" [37].

The [4,2,2] encoding method

The encoding method for fault-tolerant gates and circuits is designated as [4,2,2]. This method involves dividing the quantum circuit into blocks of four physical qubits. Conventionally, within each block, the logical value is mapped onto the two upper qubits, while the two lower qubits are employed as "control" for redundancy, thereby enabling the detection of potential errors. This specific method will be referred to later in this chapter as a "parity check" [39, 40].

The table shown in Figure 1.13 presents various [4,2,2] implementations, as treated by Infleqtion, illustrating the states that are instrumental in constructing the circuits for the demonstrations presented in this work.

1.5 Problems and applications

In this section, we will discuss two application cases of our QAOA algorithms, with the goal of optimizing two well-known problems in the computational world: Max-Cut [41, 42] and Portfolio Optimization [43, 44].

1.5.1 Max-Cut

This type of problem aims to find the "cut", or the set of nodes, that is optimal (with the minimum or maximum cost depending on the requirement) relative to a topology of points, called "nodes", which are connected in different ways depending on the type of structure considered.

Each connection can have a specific weight based on the parameters of the case being dealt with. On a geometric and topological level, these structures, formed by nodes and connections, are called "graphs" and are common objects in computational mathematics.

The graph must be represented in a numerical form that is compatible with our computing systems. Among the many possible representations, the most common is the "adjacency matrix", in which the graph is defined as an $N \times N$ matrix, where N represents the number of nodes present, arranged in progressive order to serve as indicators for the two ends of the connections, and where the elements inside the matrix represent the values associated with the various connections between the nodes.

Operation	Unencoded	FT Encoded	Compiled
PREP_00 (q0, q1)			
PREP_0+ (q0, q1)			
PREP_BELL (q0, q1)			
X(q0)			
Z(q0)			
X(q1)			
Z(q1)			
CZ(q0, q1)			
CX(q0, q1)			
CX(q1, q0)			
HH(q0, q1)			
MEAS (q0, q1)			

Figure 1.13: Table of various [4,2,2] fault-tolerant gate implementations by Infleqtion [29], illustrating states useful for circuit construction.

Figure 1.14: Example of an adjacency matrix.

Our problem

For our case study, a four-point ring graph was chosen, as indicated in the NVIDIA test example [45] and depicted in Figure 1.15.

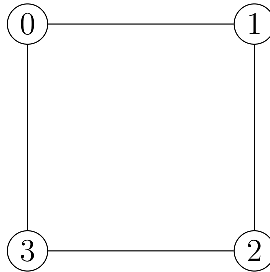


Figure 1.15: The 4-node ring graph (cycle C_4) with nodes labeled from 0 to 3, starting at the top-left corner and increasing clockwise.

Now, given the topology of this object, we must design a Hamiltonian that represents it so that it can be inserted into the parameter optimization algorithm that will define the QAOA's Ansatz, in order to return the best possible cut.

How to construct the Hamiltonian

For our problem, we know that the qubits of our logical system will be equal to the nodes of our graph. Therefore, our Hamiltonian will be purely an interaction Hamiltonian, and thus, a sum of the interactions between all possible pairs of consecutive logical qubits with a cyclic structure, by means of tensor products between Z gates on the qubits involved, with uniform weights on the connections.

Thus, we can computationally describe our Hamiltonian for Max-Cut as shown in Equation (1.5).

$$H_C = \sum_{\langle i,j \rangle \in E} (Z_i \otimes Z_j) \tag{1.5}$$

How to construct the Ansatz for QAOA

In the case of QAOA, the Ansatz must perfectly mimic the previously defined Hamiltonian at the circuit level. Therefore, the mix-unitary will simply be the application of parameterized Rx gates on all the logical qubits of our circuit, while the Cost Hamiltonian will be composed of parameterized control-Rz gates between all possible pairs of adjacent qubits in a cyclic structure.

In the following chapters, we will analyze the practical implementation of all these concepts, also in the case encoded with the method [4,2,2].

1.5.2 Portfolio Optimization

Portfolio Optimization is our second and final application discussed in this work.

Problem description and objective

In the financial field, Portfolio Optimization is one of the most widely discussed problems for which modern computational techniques can be used to solve this far from trivial problem.

The ultimate goal is to create an optimization algorithm that allows us to analyze the historical evolution of prices for given assets and to evaluate how many shares to buy and sell (in our case, we will only deal with optimization problems for buying [46]) with the ultimate goal of having a portfolio that maximizes the economic "return" and minimizes the "volatility" of a given asset structure.

This will be verified by comparing the data we obtain with those verified by the Markowitz efficient frontier [47].

Definition of the Hamiltonian in the continuous domain and mapping to the Ising computational structure

The fundamental parameters that will be essential for defining a Hamiltonian, and thus the energy profile described by our problem, are: the expected return, the covariance matrix, and the vector of prices for each asset.

Expected return (μ) [Equation (1.6)] This parameter describes the average gain we can expect due to the negative fluctuation of prices for a given asset i, calculated over a given discrete time interval from 0 to T.

$$\mu_i = \frac{1}{T-1} \sum_{t=1}^{T} r_i^t \tag{1.6}$$

Covariance matrix (Σ)[Equation (1.7)] This is the covariance matrix of the asset prices that defines how the various assets and their prices are correlated with each other, and it will be a useful parameter for defining volatility.

$$\sigma_{i,j} = E[(r_i - \mu_i)(r_j - \mu_j)] = \frac{1}{T-1} \sum_{t=1}^{T} ((r_i^t - \mu_i)(r_j^t - \mu_j)) \tag{1.7}$$

Vector of prices (P)[Equation (1.8)] This object is simply the vector of today's prices (at time T) for the assets considered.

$$P_i = p_i^T \tag{1.8}$$

Once these parameters are obtained, we can proceed with the definition and construction of our problem [46].

Definition of the mathematical problem

The problem can be defined as follows. We start by defining the cost function in the continuous domain as shown in Equation (1.9).

$$\mathcal{L}(x) : \mu^T x - q x^T \Sigma x \tag{1.9}$$

Now we impose the constraint that the sum of the purchased actions is equal to the imposed budget, and we aim to maximize our cost function in a way that increases returns while minimizing volatility (as we can observe in the system of Equation (1.10)).

$$\begin{cases} \max \mathcal{L}(x) : \max(\mu^T x - q x^T \Sigma x) \\ s.t. \sum_{i=1}^{N} x_i = 1 \end{cases} \tag{1.10}$$

Now we reformulate the problem in the discrete domain so that the shares are integers, which corresponds to a more realistic case, and we normalize the three previously seen parameters with respect to the budget, as depicted in the group of equations in Equation (1.11) .

$$P' = \frac{P}{B}, \quad \mu' = P' \circ \mu, \qquad \Sigma' = (P' \circ \Sigma)^T \circ P' \tag{1.11}$$

The problem is then reformulated as in the system of Equation (1.12).

$$\begin{cases} \max \mathcal{L}(n) : \max(\mu' n - q n' \Sigma' n) \\ s.t. P' n = 1 \end{cases} \tag{1.12}$$

Quantum formulation

To map our problem onto a quantum circuit to perform optimization, we must first calculate the number of bits needed to represent our problem. Subsequently, we must build an encoding matrix that will allow us to move between the integer domain and the binary domain required by our machine.

Let's calculate the number of qubits needed to encode our problem. First, we calculate the maximum integer number of shares to represent the prices for each asset, as indicated by Equation (1.13).

$$n_i^{max} = \text{Int}\left(\frac{B}{P_i}\right) \tag{1.13}$$

Next, we want to represent the elements of this vector of integers ($n_{\max}$) as the number of qubits necessary for the encoding of this algebraic object, with the following Equation (1.14).

$$d_i = \text{Int}(\log_2 n_i^{\max}) \tag{1.14}$$

Now we know the number of qubits needed to encode our problem, which will be equal to the sum of the maximum qubits required to encode the normalized price of our assets, so that it is computationally processable. We proceed to create a matrix of dimensions $N \times M$, where N will be the number of assets and M will be the total sum of the qubits required by our system for each asset. This will serve as a map to lead us from the computational domain (binary system) to the integer domain for a more understandable and direct visualization of the shares to buy for each asset in order to obtain our optimal portfolio, maximizing returns and minimizing volatility. The matrix is constructed as shown in Equation (1.15).

$$C = \begin{pmatrix} 2^0 & \dots & 2^{d_1} & 0 & \dots & 0 & \dots & 0 & \dots & 0 \\ 0 & \dots & 0 & 2^0 & \dots & 2^{d_2} & \dots & 0 & \dots & 0 \\ \vdots & & \vdots & \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & 0 & 0 & \dots & 0 & \dots & 2^0 & \dots & 2^{d_N} \end{pmatrix} \tag{1.15}$$

At this point, we can convert our previously normalized parameters: expectation values, covariance matrix, and prices, by mapping them to a "Binary Quadratic Optimization Problem" model as depicted in the group of equations in Equation (1.16).

$$\mathbf{n} = \mathbf{C}\mathbf{b}, \quad \mu'' = \mathbf{C}^T\mu', \quad \Sigma'' = \mathbf{C}^T\Sigma'\mathbf{C}, \quad \text{e} \quad P'' = \mathbf{C}^T P' \tag{1.16}$$

The problem is then reformulated, as shown in Equation (1.17), with the new terms:

$$\max_b \mathcal{L}(b) : \max_b(\mu''^T b - q b^T \Sigma'' b) \tag{1.17}$$

At this point, it is possible for us to trace back to a "QUBO" (Quadratic Unconstrained Binary Optimization) model by integrating the budget constraint into our cost function, also integrating a penalty term λ in order to keep my optimization function as much as possible within the imposed limits.

$$\max_b \mathcal{L}(b) : \max_b(\mu''^T b - q b^T \Sigma'' b - \lambda(P''^T b - 1)^2) \tag{1.18}$$

At this point, our QUBO formulation, shown by Equation (1.18), is still a maximization problem and is not compatible with a Hamiltonian form in computational terms applicable to our QAOA optimization system; therefore, a final step is required.

In the final analysis, recent studies have shown the strong compatibility between QUBO models and Ising Hamiltonian structures [48]. The Ising Hamiltonian structure is defined in terms of linear and quadratic coefficients, therefore describing an interaction between two objects; which is perfectly comparable to our Portfolio Optimization system, in which a covariance relationship is defined between the price fluctuations in all possible assets.

The bits of our QUBO model must be converted into spin elements in the way shown by Equation (1.19).

$$b_i \rightarrow \frac{1 + s_i}{2} \tag{1.19}$$

Furthermore, it is necessary to convert and rearrange the coefficients of the QUBO model in terms compatible with our Ising system; this can be done, for example, through tools available from Python libraries as we will see in the following chapters.

At this point, it is possible to obtain the Ising Hamiltonian corresponding to our QUBO model in the form depicted by Equation (1.20), in terms of spin components:

$$\min_s \mathcal{Z}(s) : \min_s \left(\sum_i h_i s_i + \sum_{i,j} J_{ij} s_i s_j \right) \tag{1.20}$$

As a final step, we proceed to translate this Hamiltonian into terms of Z gates, in order to map the Ising Hamiltonian into computational terms (as we can observe in Equation 1.21).

$$H = \sum_i h_i Z_i + \sum_{i,j} J_{ij} Z_i \otimes Z_j + offset \tag{1.21}$$

At this point, we have constructed our Hamiltonian in a suitable manner to design our Ansatz in order to calculate our optimal portfolio, using QAOA.

1.6 Tools adopted

For the purposes of this work, we have utilized two fundamental tools for the development and creation of our algorithms: Qiskit and CUDAQ.

1.6.1 Qiskit

Qiskit [49] is an open-source framework developed by IBM for writing quantum circuits for both simulations and execution on real IBM quantum hardware, to which it is possible to connect remotely.

The logic follows a CPU-centric paradigm with a certain "serialization" of commands and actions imparted to the system. This is due to the specific architecture defined by IBM, which does not allow for significant computational speedups. Qiskit's lack of performance in terms of speedup is balanced by a simpler and more immediate programming approach. It presents a "qubit-agnostic" approach, making it compatible with the different technologies offered by its developer. This system is now mature, although it is in continuous evolution.

1.6.2 CUDA-Q

CUDA-Q [50] is also a tool conceived and developed for the simulation of quantum circuits with high computational performance. This is thanks to its GPU-centric architecture, which allows for a significant speedup compared to the CPU-centric paradigm, with potential scalability in the number of qubits usable during simulations and executions.

This particular integrated library was designed to be optimally interfaced with NVIDIA hardware. Due to its system's inherently parallel logic, CUDA-Q presents a higher complexity in terms of programmability compared to Qiskit. This is because GPUs require memory contiguity to ensure optimal performance, thus requiring linear indexing and the flattening of N-dimensional data structures into one-dimensional ones.

Both libraries are programmable in Python. CUDA-Q also offers the possibility of being programmed in C++ and its compiler (nvq++) is optimized to efficiently execute hybrid code on various hardware architectures.

In summary, Qiskit is an ideal choice for those who want a complete framework oriented towards learning and research, with immediate access to IBM's hardware. CUDA-Q is the best choice for developers and researchers who need high-performance quantum simulations, leveraging NVIDIA hardware to accelerate hybrid workflows.

For our purposes, we will predominantly use Qiskit, while we will relegate CUDA-Q only for the development and initial testing of our first Max-Cut algorithms that will be discussed in the following chapters.

Chapter 2

Methodology

2.1 Research objective and context

Given the theoretical premises established in the preceding chapter, we now possess the necessary background to comprehend the methodology and experimental development of our research project.

2.1.1 Final objective: Portfolio Optimization

Our ultimate goal is to perform **Portfolio Optimization** using the Quantum Approximate Optimization Algorithm (QAOA) (refer to Section 1.3 in Chapter 1 for the algorithm's introduction).

The algorithm will take into consideration an arbitrary number of assets, their corresponding daily price changes over a one-year time frame, and a set of parameters quantifying the allocated budget and the user's risk tolerance. The output will be the optimal portfolio, defined in terms of the integer number of shares to purchase, with the aim of maximizing return while minimizing volatility. The resulting distribution of solutions will be visualized on the Markowitz efficient frontier to allow for visual validation of the algorithm's optimized results.

The entire work will encompass two separate simulations: a **noiseless** simulation and a **noisy** simulation, in order to quantitatively compare the results obtained under different operational configurations.

2.1.2 Quantum hardware constraints and encoding

Our objective is to design the algorithm to be processable on real quantum hardware, which currently implies a limitation on the number of qubits.

Given the current state-of-the-art of quantum computers, an encoding procedure is necessary. This will be achieved using the [**4,2,2**] **code** developed by Infleqtion

(as discussed in Subsection 1.4.3 of Chapter 1), to make the process as **fault-tolerant** as possible. This approach will incorporate an error detection apparatus (note that since this work is aimed at a neutral atom machine, which currently does not support error correction, we will limit ourselves to the error detection process).

2.2 Algorithms development

To properly encode the Portfolio Optimization algorithm, we must first construct encoding models based on well-known logical structures to verify the correct functioning of our encoding and decoding processes, should the latter be required.

2.2.1 Initial test case: Max-Cut problem

Our well-known test case is the Max-Cut problem for a 4-node ring graph, a standard structure provided by NVIDIA within the CUDA-Q framework, for which the two optimal solutions are clearly defined. These known solutions are essential for guiding the debugging process of our encoded algorithm. If our computed results deviate from the expected logical results, it would indicate an error within our control flow, compromising the fidelity of the final outcomes.

PROGRAM 0: CUDA-Q verification of the Max-Cut structure

We will analyze a case study available online from NVIDIA, which implements a well-structured QAOA algorithm for the Max-Cut of a 4-node ring structure, where the solutions are known [45]. The program is developed in CUDA-Q.

In this initial program, we first tested the algorithm to verify the correctness of the stated results. Subsequently, we proceeded with a generalization for arbitrary graphs, which will serve as a starting point for future encoding processes, once the scalability of quantum computers permits volumes comparable to those indicated here.

Generalized QAOA algorithm for an arbitrary graph (NVIDIA adaptation)

1. **Define necessary libraries:**
   We begin by importing and defining the necessary libraries required for the proper execution of our code. First, we import `cudaq`, which serves as our foundational library for writing and defining the quantum program. Subsequently, we import `spin` from `cudaq`, which is responsible for defining the Hamiltonian through the interactions between the various nodes of the graph. Finally, we

import `numpy` and `math`, which are essential for handling the numerical and mathematical structures within our algorithm.

2. **Define the graph connection function (`define_graph_connections`):**
   This function processes an adjacency matrix from the `graph_matrix.txt` file describing a graph and translates it into a list-of-lists data structure. This structure is essential for constructing the Interaction Hamiltonian and the QAOA Ansatz. To properly and completely structure our function, we follow a straightforward and efficient logic tailored for processing Kernel-GPU structures that leverage memory contiguity to optimize data processing.

   Initially, we open the file containing the adjacency matrix that represents our graph and read all its rows, storing them in a variable. Subsequently, the matrix is converted into an equivalent form with a structure that is comprehensible to our software. The number of qubits is then defined as equal to the number of elements in a row of the matrix, which corresponds to the number of nodes in the graph.

   Next, we define two initially empty lists that will hold the graph interactions in matrix form and in a linearized format compatible with the kernel logic.

   We then iterate over the elements of the adjacency matrix to check whether any of them are equal to "1.0", which, in our case, corresponds to the interaction weight. If such elements are found, it indicates that a connection exists between the nodes identified by the corresponding indices. These connections are then numerically appended to our data structures, faithfully reconstructing the interaction structures needed to define our Cost Hamiltonian. The function in the end returns a list of interactions for the QAOA kernel, containing the indices for the control and target qubits involved in the interaction, a list of interactions with their indices and weights/values (useful for Hamiltonian construction), and the number of qubits (i.e., the number of nodes) for our system. Note that we discard all "self-interactions" where a qubit interacts with itself.

3. **Define the Hamiltonian construction function (`construct_graph_hamiltonian`):**
   Given the list of connections, as defined by the model in the previous chapter (refer to Section 1.5.1 in Chapter 1), we multiply each interaction weight by the tensor product of the Z quantum gates (interaction gates) applied to the qubits of the interaction pair. We sum all contributions and return the Hamiltonian necessary for evaluating the expectation value to be minimized by our algorithm.

4. **Define fundamental QAOA circuit parameters:**
   We set the number of layers to $p = 2$ and define the number of parameters

(angles) that our program must return, which is $2p = 4$, as we must implement a Cost Hamiltonian and a Mixer Hamiltonian, each with its own angle, for every layer.

5. **Linearization of connection structure for GPUs:**
   Since we are working with CUDA-Q and, consequently, with algorithms that will be processed on GPUs to boost performance and leverage memory contiguity, the linearization of multi-dimensional structures like matrices is necessary. In our case, we linearize our list of connections for the kernel as indicated below. Elements in even positions (including position 0) serve as the control qubits, while those in odd positions serve as the target qubits.

6. **Define the kernel as the QAOA Ansatz (`kernel_qaoa_general_graph`):**
   We define the kernel function, which takes the number of qubits, the number of layers, the list of angles, and our linearized interaction vector as arguments. This function returns the circuit that serves as the QAOA Ansatz, including:

   - An initial superposition layer using **Hadamard gates** to allow for statistical equipartition of solutions.
   - The **Cost Hamiltonian** translated into parameterized interactions (control-R_z gates) based on the angles, applied to the involved qubits according to the interaction list.
   - The **Mixer Layer**, which applies a parameterized R_x gate to every qubit to explore the entire space of possible combinations.

   We apply the Cost Hamiltonian and the Mixer for the specified number of layers (p), perform measurements, and retrieve the angles to optimize the process using a classical optimizer (we will use COBYLA).

7. **Define initial parameters and the optimizer:**
   We define random seeds and set up the classical optimizer that will be used in our program. Specifically, as previously announced, we employ `COBYLA`. It is worth noting that other optimizers are available, including gradient-based ones, which we will not implement or discuss in this thesis.

   Subsequently, we initialize the angles to be assigned to our parameters, which will be optimized as the number of layers in our circuit increases.

8. **Define the objective function, optimization, and circuit sampling:**
   We now define the objective function by transforming our linearized vector into a Python list and returning the **expectation value** of the Hamiltonian with respect to the states defined by the Ansatz. Using this result, we

optimize the parameters with COBYLA. At the end of the circuit, we impose the measurement by sampling our results and visualizing their statistical distribution. If desired, the circuit can be printed at the end of the algorithm to provide a graphical visualization of the developed hardware implementation.

Optional random graph generation: Should one wish to generate a random adjacency matrix (as can be seen in Figure 2.1) starting from an arbitrary graph of N nodes, it is possible, using the `graph_analysis.py` program and the `networkx` library, to construct such a graph. The program outputs an adjacency matrix to a file (`graph_matrix.txt`) and a plot of the graph. This allows the matrix to be fed into our main program to perform Max-Cut via QAOA for a generic graph. Program correctness was previously verified by applying the algorithm to a small adjacency matrix corresponding to the standard 4-qubit Max-Cut graph instance documented by NVIDIA.

$$\mathbf{A} = \begin{pmatrix} 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 1.0 & 1.0 \\ 0.0 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 1.0 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 0.0 & 0.0 & 1.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 1.0 & 0.0 & 1.0 & 0.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 0.0 & 0.0 & 1.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 0.0 & 1.0 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

Figure 2.1: The adjacency matrix **A** for the 10-node test graph

PROGRAM 1: Qiskit implementation of encoded QAOA, Max-Cut on the 4-node ring graph with [4,2,2] encoding:

Following the verification of the CUDA-Q Max-Cut algorithm on the 4-node ring graph benchmark (based on the NVIDIA case study), we proceed to implement the encoded version using the [**4**,**2**,**2**] module developed by Infleqtion. This is performed on the identical graph model, as detailed in Chapter 1 Section 1.4.3, to confirm that the selected encoding method yields results consistent with the unencoded approach.

Implementation context: Due to the collaborative nature of this research project across multiple sections, this specific implementation utilizes Qiskit libraries. The CUDA-Q implementation is delegated to other team members, allowing for a complementary integration and comparative analysis, particularly in preparation

for future execution on actual quantum hardware. This hardware-specific phase, however, falls outside the scope of this thesis, for the study case of the Max-Cut, which focuses exclusively on the logical and functional aspects of the quantum codes under simulation. In contrast to the Max-Cut case, the Portfolio Optimization algorithms will also be executed on physical quantum hardware. This step is crucial to evaluate their performance and the validity of the results under real-world conditions.

Development of the encoded [4,2,2] QAOA for the 4-node Max-Cut case: The development of the encoded QAOA for the NVIDIA 4-node test graph is structured as follows:

1. **Library and utility integration:**
   We integrate the necessary libraries, including a custom file named `"funzioni_logiche_qiskit"`. This file contains the predefined encoded quantum gates and the post-processing function crucial for program execution.

   The file's structure and contents will be presented at the conclusion of this subsection to provide an immediate and functional overview of the full system implementation. Here, it is crucial to focus on importing all the libraries necessary to define our circuit via software. These include `QuantumRegister` components combined together to form a single, comprehensive `QuantumCircuit` that implements our algorithm. Additionally, we import mathematical tools useful for working in binary spaces, such as `itertools`, which will be used to generate all possible bitstrings producible by our code, and `numpy`, to generate data structures that are interpretable by our software.

2. **Parameter definition:**
   We define the fundamental parameters for the quantum circuit construction: the number of logical qubits, a reduced number of ancillas to minimize hardware requirements, the number of QAOA layers, the resulting number of optimization angles, the number of encoded blocks, the block size, and the nature of the parameters ($\vec{\theta}$). The ancilla qubits are specifically dedicated to the detection of bit-flip and phase-flip errors following the encoding stage. While the original implementation requires an additional ancilla for each encoded block, we optimized the qubit count for our simulation by employing only two ancillas for all blocks, incorporating a reset operation upon syndrome detection.

   This program is intended as a control check on the proposed encoding scheme's efficacy, rather than an optimal simulation for a specific study case.

3. **QAOA encoded Ansatz construction:**
   We define the quantum circuit that serves as the QAOA Ansatz.

a) **Function declaration:** We declare the function, passing the necessary parameters for the algorithm's execution, that are: the number of logical qubits under consideration, the number of circuit layers to be employed, the list of parameters to be optimized, the block size for encoding (expressed in terms of physical qubits per block), the number of ancillary qubits utilized, and the total number of encoding blocks, which is set to $\lceil N_{\text{log}}/2 \rceil$, where N_{log} denotes the number of logical qubits. This choice arises from the constraint that each encoding block accommodates at most two logical qubits.

b) **Circuit register constitution:**
The quantum circuit is constructed from several registers: the logical qubit register (`qvector`), system ancillas (`ancillas`), syndrome preparation ancillas (`syndrome_prep`), and a classical register (`syn_class`) for recording ancilla measurements. All registers are incorporated into the complete quantum circuit object (`qc`), which is returned by the function.

c) **State encoding and error detection:**
All $|0\rangle$ states are encoded using the method suggested by Infleqtion. For each block, an ancilla verifies the operation's integrity. If this ancilla measures $|1\rangle$ (detected via Qiskit's `if_test()` method), it is immediately reset. This reset mechanism is purely illustrative for error detection purposes and is sufficient for our current scope.

d) **QAOA evolution and ancilla checks:**
The Cost and Mixer Hamiltonians are implemented using the encoded functions from `funzioni_logiche_qiskit`. The process is iterated over each layer to form the full Ansatz. At the end of each layer, dedicated ancillas are analyzed and reset if a $|1\rangle$ state is measured.

e) **Circuit verification:**
The circuit could be printed for verification and returned by the function.

4. **Observable Hamiltonian definition:**
We define, in Listing 2.1, the Hamiltonian required for evaluating the system's expected energy value, which serves as the objective function for optimization. The entire Hamiltonian must be mapped from the logical to the physical space using the `logical_z_spin()` function for each logical Z interaction gate. The interaction weights remain invariant.

```
# Define the Cost Hamiltonian, mapping it to the physical
    qubits.
cost_hamiltonian = 0.5 * (logical_z_spin(0) @ logical_z_spin
    (1))\
     + 0.5 * (logical_z_spin(1) @ logical_z_spin(2)) \
     + 0.5 * (logical_z_spin(2) @ logical_z_spin(3)) \
     + 0.5 * (logical_z_spin(3) @ logical_z_spin(0))
```

Listing 2.1: Physical Observable Hamiltonian

5. **Circuit generation and measurement:** The circuit is generated using the previously defined function, and measurement gates are appended (as can be seen in Listing 2.2).

```
circuit=qaoa_algorithm_ansatz(logical_qubit, layer_count,
    tehtas, block_size, n_ancillas)
circuit.measure_all()
circuit.parameters
```

Listing 2.2: Circuit Generation and Measurement

6. **Backend selection and transpilation:**
   We select the simulation backend and impose transpilation with optimization level 3, yielding a physical circuit candidate for optimization. This process translates the quantum circuit from a *technology-agnostic* representation, completely independent of hardware constraints, into a form compatible with the physical limitations and control capabilities of a selected quantum hardware platform. For simulations not tied to specific hardware, we leverage one of the numerous simulators available within the Qiskit ecosystem; in our case, we employ the `AerSimulator()` class provided by the `qiskit-aer` library.

7. **Initial parameters and cost function:**
   We choose the initial parameters and define the `cost_func_estimator` function. This function computes the energy expectation value for the encoded system, ensuring the observable is correctly mapped from virtual to physical qubits.

8. **Optimization process:**
   The optimization process employs the COBYLA algorithm to find the optimal parameters. These values are then assigned to the physical, encoded circuit. With regard to the minimization process in our program, we decided to adopt the function `minimize` available in the `scipy` library, in order to determine the accessible energetic minimum corresponding to the degree of approximation of the energetic Hamiltonian defined by our system.

9. **Sampling and result analysis:**
   The results are sampled and stored in dictionaries, mapping binary and integer configurations to their recurrence counts, sorted by decreasing probability.

   We must reverse the bitstring order due to Qiskit's Little Endian convention and filter the results to include only those where all ancilla measurements are $|0\rangle$. While this filter is simplified here due to the reduced, reset-based ancilla system, it is indispensable in a complete fault-tolerant architecture, particularly in noisy environments.

10. **Distribution visualization and decoding:**
    We generate and display the histogram of the result distribution.

    Finally, the custom postprocessing function, `post_processing_decoding`, remaps the measured physical configurations back to their logical counterparts. This function returns the two most probable Max-Cut logical configurations and the dictionary containing the logical-to-physical map, allowing for a final performance evaluation against the unencoded NVIDIA result.

LIBRARY: encoded quantum gates and post-processing function:

As previously introduced, we now detail the structure of the dedicated library file, `funzioni_logiche_qiskit`. This file is essential for providing the [**4**,**2**,**2**] encoded quantum gates necessary to correctly implement the QAOA Ansatz and the corresponding logical **Z** operator transformation for the Cost Hamiltonian. A comprehensive reference file covering all FT quantum gates discussed by Infleqtion, implemented in CUDA-Q, is available in the associated GitHub repository [51].

The internal structure and functions of our Qiskit library are detailed as follows.

1. **Library imports:**
   We import the necessary Python and Qiskit libraries required for the implementation of the encoded logic. The first library to be imported is `qiskit`, which enables the use of all the tools necessary to construct our quantum circuit. Subsequently, we import `product` from the `itertools` library, in order to generate all possible combinations obtainable within our system. This will be useful for creating a dictionary that maps logical combinations to their physical counterparts. Finally, we import the Pauli operators, which will be instrumental in remapping the Cost Hamiltonian from the logical to the physical representation.

2. **Implementation of encoded $\mathbf{R_x}$ gate:**
   The fault-tolerant $\mathbf{R_x}$ gate, essential for the Mixer Hamiltonian, is implemented according to the specifications of the [**4**,**2**,**2**] code, this part is omitted by the paper of Infleqtion, but we have reconstruct its based on the studies of

our research team [51] and its implemented in Listing 2.3.

```
def logical_rx(logical_target : int, theta: float, qvec:
    qiskit.QuantumRegister, anc: qiskit.QuantumRegister, qc:
    qiskit.QuantumCircuit  ):


    block_id = logical_target // 2
    pos_inside_block = logical_target % 2

    offset = block_id * 4

    qc.barrier()
    if(pos_inside_block == 1):

        qc.h(anc)
        qc.cx(anc, qvec[offset])
        qc.cx(anc, qvec[offset + 2])
        qc.rx(theta, anc)
        qc.cx(anc, qvec[offset])
        qc.cx(anc, qvec[offset + 2])
        qc.h(anc)

    else:

        ## swap
        qc.swap(qvec[offset + 1], qvec[offset + 2])

        ## rx
        qc.h(anc)
        qc.cx(anc, qvec[offset])
        qc.cx(anc, qvec[offset + 2])
        qc.rx(theta, anc)
        qc.cx(anc, qvec[offset])
        qc.cx(anc, qvec[offset + 2])
        qc.h(anc)
        ## swap
        qc.barrier()
        qc.swap(qvec[offset + 1], qvec[offset + 2])
        ##
    qc.barrier()
```

Listing 2.3: Encoded $\mathbf{R_x}$ Gate Implementation

3. **Implementation of encoded $\mathbf{R_{zz}}$ gate:**
   The fault-tolerant $\mathbf{R_{zz}}$ gate is implemented for the Cost Hamiltonian, adhering to the [**4**,**2**,**2**] encoding scheme presented by Infleqtion [29].

4. **Post-processing decoding function:**
   This crucial function handles the post-processing of measurement results. It takes the measured physical bitstring combinations (filtered for syndrome-free states) and decodes them, creating a dictionary that maps the physical states to their corresponding logical Max-Cut configurations. This function constitutes the core of the decoding process within our system and has been specifically designed to map the encoded physical states to their logical counterparts, thereby enabling a comparison between unencoded and encoded systems. The entire procedure is carried out through a "dictionary" logic.

   As a first step, we generate all possible combinations available in our logical circuit, thereby constructing the complete code space available for our noiseless system and defining the keys of our dictionary. Subsequently, each key is associated with its corresponding logical decoding, thus establishing a unique and well-defined mapping between the encoded and unencoded code spaces.

   Once this system has been constructed, we translate into the logical space all the strings sampled from our circuit and return the two most probable and non-identical strings produced by the sampling. At this stage, we can provide the list of optimal solutions identified.

5. **Implementation of encoded Z operator mapping:**
   This function is responsible for transforming the logical **Z** operators of the unencoded Hamiltonian into their physical [**4**,**2**,**2**] encoded representation, required for the evaluation of the energy expectation value. The underlying logic consists in taking the presented list containing the logical interactions together with their indices. By providing these indices to our function, it will generate a new list of interactions perfectly mapped onto the encoded system, thereby ensuring that the evaluation phase of our circuit remains consistent with the physical–mathematical model upon which our energetic assessments are performed.

The resulting library file provides a clear, correct, and modular structure for implementing the useful functions for the logical flow of our overall quantum algorithm.

2.2.2 Portfolio Optimization: encoded and unencoded analysis

Having validated the functional correctness of our logical models and encoding systems (as demonstrated in the previous section), we proceed to implement the quantum models related to our primary use case: Portfolio Optimization (as outlined in Chapter 1 Section 1.5.2). Our objective here is to confirm the consistency of

the output specifically, the optimal portfolios derived from the optimization when comparing the encoded and unencoded quantum algorithms.

To correctly implement the optimization model via QAOA, the overall workflow is divided into three synergistically collaborating sub-programmes:

1. [**extract_data_for_processing_PO**]: Data Extraction and Structure Generation. Extracts necessary financial data and constructs the fundamental data structures required for defining the QAOA Hamiltonian model.

2. [**PO_construct_Hamiltonian**]: Hamiltonian Model Construction. Defines the Hamiltonian model based on the extracted data structures, as articulated in Subsection 1.5.2 of Chapter 1.

3. [**PO_unencoded_QAOA**]: QAOA Implementation and Analysis. Execute both encoded and unencoded QAOA and perform the resulting portfolio analysis.

We now analyze the first two programs, which serve as essential utility libraries for the subsequent implementation of the two QAOA versions.

[extract_data_for_processing_PO]: Financial data extraction

This program is responsible for processing a `.csv` dataset named `"full_assets_data"`, which contains the daily price variations for four assets over one year. The program extracts the following key financial parameters necessary for the Hamiltonian model:

- **Mean return (μ):** The average return, weighted over the considered time horizon, calculated as the sum of the daily price variations relative to the previous day (see Chapter 1, Subsection 1.5.2).

- **Covariance and variance matrix (Σ):** The volatility matrix.

- **Current price vector:** The current prices of the assets, required for calculating the final investment outlay of the optimal portfolio.

Programme workflow:

1. **Dataset visualization:** The partial input dataset is displayed in Listing 2.4.

```
Date,AAPL,IBM,NFLX,TSLA
2011-12-23,12.108014106750488,105.93802642822266,...
2011-12-27,12.20407772064209,106.05270385742188,...
2011-12-28,12.087297439575195,105.50223541259766,...
2011-12-29,12.161747932434082,106.75801086425781,...
...

```

Listing 2.4: Sample Dataset (`full_assets_data.csv`)

2. **Library imports:**
   Essential Python numerical libraries are imported. We import the `pandas` library to manage our dataset of annual financial data in a proper, concise, and comprehensive manner, which will serve as the foundation for the subsequent construction of our mathematical model, instrumental for the implementation of our Ansatz for our QAOA (Quantum Approximate Optimization Algorithm).

   The `numpy` library will be used to construct and process our data structures, such as numerical arrays and matrices, along with all consequential algebraic operations between them.

3. **Function definition:**
   Functions for calculating Σ, μ, and the current price array are defined, adhering to the models presented in the introduction.

   The mean economic return, μ_i, is calculated by considering the sum of daily price variations, Δp_i^t, normalized by the total number of days, T, considered in the financial evaluation period. Mathematically, the daily return R_t for an asset is typically formulated as in Equation (2.1).

$$r_i^t = \frac{p_i^t - p_i^{t-1}}{p_i^{t-1}} \tag{2.1}$$

   And the mean return, μ, for the period is defined as in Equation (2.2).

$$\mu_i = E[r_i] = \frac{1}{T} \sum_{t=1}^{T} r_i^t \tag{2.2}$$

   This value is instrumental in assessing the average numerical trend of prices, indicating whether they tend to increase or decrease. When seeking the optimal combination, in terms of initial outlay, we aim to evaluate the most cost-effective assets available, among other factors.

The Covariance Matrix, Σ, evaluates the average correlation between every asset in our portfolio. This correlation is quantified by calculating the deviation of the daily return, R_t, from the calculated mean return, μ, for each pair of assets (i, j) (as we can observe in Equation 2.3 and 2.4). The covariance between two assets, i and j, is computed as the scalar product of these deviations, weighted by the number of days considered minus one $(T-1)$:

$$\sigma_i^2 = E[(r_i - \mu_i)^2] = \frac{1}{T-1} \sum_{t=1}^{T} (r_i^t - \mu_i)^2, \tag{2.3}$$

$$\sigma_{ij} = E[(r_i - \mu_i)(r_j - \mu_j)] = \frac{1}{T-1} \sum_{t=1}^{T} ((r_i^t - \mu_i)(r_j^t - \mu_j)). \tag{2.4}$$

This metric provides a numerical and conceptual measure of volatility and thus quantifies the investment risk associated with each asset. Our primary objective in portfolio selection is to identify a solution that minimizes this factor, thus prioritizing investments that are as secure as possible.

Finally, the last function simply extracts the current prices, $P_i = p_i^T$, possessed by each asset. These values are essential for calculating the total investment, or initial investment capital, required for the optimal solution selected.

4. **Main program body:**
The main program logic gets the user-defined budget, risk attitude, number of selected assets and data reading intervals. It then utilizes the defined functions and parameters to extract and return the processed data required for the Hamiltonian construction in the subsequent program.

Our data extraction logic is deliberately simplistic and eschews the use of pre-existing, dedicated functions within the Python ecosystem. This choice was made to adhere as closely as possible to the entire structure proposed by the mathematical model, thereby enabling a concurrent verification of the model itself.

The critical step in extracting the data for our structure is the systematic exclusion of both the first row and the first column. This exclusion is necessary because these components contain, respectively, the names of the assets and the dates corresponding to the days covered by our financial analysis.

It is also important to note that, according to our implemented logic, only the last N assets from the total pool of available assets within our portfolio will be considered. Here, N corresponds to the number of assets arbitrarily selected by the user for analysis. This selection criterion focuses the subsequent mathematical treatment exclusively on the last N assets based on the original data arrangement.

[PO_construct_Hamiltonian]: Hamiltonian model construction

This program represents the core of our modeling implementation, defining the optimization problem in a format that is processable by QAOA. All steps outlined here reference the model detailed in Chapter 1, Subsection 1.5.2.

Program workflow:

1. **Library imports:**
   Essential Python calculation libraries for data processing are imported. We import the `math` library specifically to facilitate the correct construction of the mathematical model. This model involves various operations on one-dimensional and two-dimensional objects, such as calculating a square root, with the ultimate goal of constructing the associated quadratic model.

   The `QuadraticProgram` class is then utilized to define the quadratic model associated with our problem simply and directly, once the fundamental mathematical objects necessary for the structure's construction have been extracted.

   Finally, the `numpy` library is employed to define the basic data objects, which contain the data extracted from our files, and to perform the associated operations in a straightforward manner.

2. **Function definition:**
   We define our core function that exclusively processes the data objects extracted by our dedicated data extraction routine. This approach is adopted to avoid redundant re-extraction and reprocessing of data directly from the original data frame.

   This defined function is intentionally the most straightforward and immediate implementation for achieving our main goal: the construction of the quadratic mathematical model, which subsequently allows us to define the problem Hamiltonian for the Quantum Approximate Optimization Algorithm (QAOA).

3. **Data extraction call:**
   The first program is called to extract the necessary data for processing and building a computationally compatible model (as shown in Listing 2.5).

```
1     total_budget, mu, sigma_matrix, actual_prices_array,
    num_assets, num_time_steps,buy_assets_dictionary=
    extract_datas_from_input('data_and_metadata_assets/
    full_assets_data.csv', total_budget, num_components)
2
```

Listing 2.5: Data Extraction Call

4. **Data normalization:**
   The extracted data is normalized with respect to the user-defined budget. To correctly construct our model, it is essential to normalize the obtained data with respect to the allocated budget. This crucial step ensures that we subsequently deal with integer quantities of shares to be included in our portfolio.

   This normalization procedure is necessary to guarantee that the resulting financial optimization model adheres as closely as possible to a real-world context, given that, in practice, the shares included in a portfolio cannot be fractional.

5. **Qubit count calculation:**
   The required number of qubits needed to encode the problem is calculated as the total budget dived to the actual price for each assets, than we proceed to sum up all the single values rounded up and we obtain the maximum number of qubits required for our problem.

6. **Encoding matrix construction:**
   An encoding matrix is constructed to map the computational space (binary strings) to the integer space, which is essential for correctly reading the final portfolio. The construction of this specific matrix is initiated by considering the number of qubits required to encode each asset. For every asset, we create a matrix row whose length corresponds to the total number of qubits required by our system, the rows are initially void (all elements are zeros).

   We begin filling the first N positions of the first row with the first N powers of 2, starting from 2^0, where N denotes the number of qubits assigned to that specific asset. In the subsequent rows, we repeat the procedure, starting the placement of the powers of 2 from the element immediately following the last position considered in the preceding row.

   This structure fundamentally allows for a unique mapping from the binary space to the integer space. Consequently, once our optimization is successfully completed in the binary space, we can reuse this matrix to map the result back to the integer space, thereby clearly identifying the optimal number of shares for each asset within our portfolio.

7. **QUBO coefficient generation:**
   The coefficients for the QUBO (Quadratic Unconstrained Binary Optimization) problem are generated using the encoding matrix applied to the normalized μ, normalized Σ, and the normalized current price vector. This maps the problem into the binary domain.

- A Penalty Factor (λ) (denoted `<lambda_penality_term>`) is set to incorporate the budget constraint into the objective function, thereby formally rendering the problem "Unconstrained".
- A Risk Parameter (q) (denoted `<risk_avversion_parameter_q>`) is introduced to model the user's risk appetite. A high parameter value implies a more risk-averse investor, while the opposite is true for low values.

8. **Quadratic problem definition:**
   The quadratic problem is defined in the computational basis to maximize returns and minimize volatility. We utilize the `.minimize()` method because the objective function is structured with negative signs for returns and positive signs for volatility and the penalty factor. Conversely, if the objective were to invert the sign of our objective function, we would be required to use the `.maximize()` function.

   These processes are fundamental within the construction of our Quadratic Program, in order to achieve mathematically correct optimization. Any alternative procedure would be incongruent with the established goals of our problem and would, therefore, lead to erroneous and non-optimal solutions for the purposes of our system.

9. **Ising Hamiltonian extraction:**
   The '.to_ising()' method is used to extract the Ising Hamiltonian as a `SparsePauliOp` object. The linear coefficient vector and the quadratic interaction matrix are manually reconstructed from this object, which is necessary for implementing the encoded [**4**,**2**,**2**] QAOA Ansatz. To achieve this specific purpose, we first extract the indices of the coefficients from our Hamiltonian structure where either a linear or quadratic interaction is applied.

   Subsequently, we proceed to insert these indices into an algebraic structure, ensuring the numerical values of each interaction are placed in the correct corresponding position. These extracted values are then utilized to construct our Ansatz and execute our optimization algorithm.

10. **Data return:** All calculated data and parameters necessary for the subsequent QAOA programme are returned in Listing 2.6.

```
1 return qp, ising_Hamiltonian, num_qubits, C,
     lambda_penality_term, total_budget, J, h, constant_offset
2
```

Listing 2.6: Model Data Return

[PO_unencoded_QAOA]: Unencoded QAOA Implementation and Analysis

This section details the construction of the final notebook that utilizes the calculated data and the Hamiltonian model to implement the unencoded QAOA and extract the optimization results for analysis.

Programme workflow:

1. **Library installation and version check:** All necessary libraries are installed and verified for mutual version compatibility (as we can observe in Listing 2.7).

```
1 %pip install qiskit==1.4.2 qiskit-aer==0.15.1 qiskit-algorithms==0.3.0 qiskit-ibm-runtime==0.30.0 qiskit_finance==0.4.1 pylatexenc==2.10 numpy==2.0 pandas==2.2.2 matplotlib==3.9.2 seaborn==0.13.2 mplfinance==0.12.10b0 pyqubo==1.5.0 --quiet
2
3
```

Listing 2.7: Library Installation and Environment Setup

2. **Model library imports:**
   The previously created libraries for Hamiltonian model extraction and related data processing are imported. We import the necessary functions from the custom-built file libraries. This import is specifically performed to utilize the function that processes the input data and furnishes the necessary structures required for both the construction of the Hamiltonian and, following that, the Hamiltonian itself

3. **Data extraction:**
   The necessary data is extracted using the functions contained in the imported libraries. The first function, responsible for processing the input financial data, requires the extraction of the following core parameters from the dataframe:

   - The **total budget** (B) allocated for the portfolio.
   - The **Mean return vector** (μ), representing the expected return for each asset.
   - The **Covariance matrix** (Σ), which quantifies asset volatility and correlation.
   - The **actual prices array** (P), detailing the current market price for each asset.

- The **number of time periods** (T) used for the financial evaluation.
- The output dictionary, containing the initially selected assets, denoted as `buy_assets_dictionary`.

The second function, designated for the construction and formulation of the problem Hamiltonian, must extract and provide the following objects, which define the optimization problem in the quantum framework:

- The mathematical **Quadratic Program** (`qp`).
- The **problem Hamiltonian** ($\mathcal{H}$).
- The total **number of qubits** (N_{qubits}) required for the encoding scheme.
- The **encoding matrix** (C), which maps the binary space to the integer share quantities.
- The **lambda penalty term** (λ), used to enforce the budget constraint.
- The coefficient matrices for the Ising Hamiltonian: the **quadratic coupling terms** (J) and the **linear field terms** (h).
- The **offset value**, representing a constant energy shift in the Hamiltonian.

4. **Algorithm Library Imports:**
   All other libraries required for the QAOA algorithm execution are imported:

   - The QAOA algorithm directly from `Qiskit`.
   - The desired Sampler to execute the quantum circuit.
   - A standard classical optimizer (e.g., COBYLA or SLSQP) for the iterative variational process.
   - The `numpy` and `pandas` libraries for data handling and mathematical operations.
   - The `MinimumEigenOptimizer` to interface the classical optimization framework with the quantum solver.

5. **QAOA parameter definition:**
   Parameters essential for the QAOA execution (e.g., number of layers, initial angles) are defined. Choose a good set of parameters influence the performances of our algorithm also in terms of optimal solution.

6. **Optimization setup:** The desired number of optimization cycles (iterations) is defined, along with utility lists for storing data required for post-processing and visualization. The initial number of iterations for the classical optimizer is set to 100 for the preliminary checking phase. Subsequently, this count will be reduced to ensure compatibility with noisy simulations, particularly in terms

of required computational time. This reduction is critical for scenarios where the simulation is conducted not on actual quantum hardware, but utilizing a high-fidelity noise model extracted from a real device.

The fundamental lists required for storing the results are defined and initialized as empty arrays. These lists are crucial for tracking key metrics across multiple optimization runs, specifically:

- Expected returns (μ)
- Measured volatilities (σ)
- Final optimal portfolios
- Total budgets invested

7. **QAOA execution:**
   The QAOA is executed by instantiating it directly from Qiskit's classes. The `MinimumEigenOptimizer()` class is used as the optimization framework, and the `.solve()` method is applied to find the optimal bitstring combination corresponding to the quadratic problem (as returned by the Hamiltonian library). It is crucial to note that within `Qiskit`, the `QAOA` algorithm, unlike the `VQE` algorithm, does not permit the definition and direct assignment of a custom Ansatz model.

   While the internal `.ansatz` attribute, which is present for the generic `VQE` class, is initially set to `"None"` (thereby allowing us to assign our custom Ansatz to `Qiskit`'s `QAOA` without raising an execution error), this assignment will not be functionally implemented. Instead, the custom model will be bypassed, with the algorithm defaulting to the use of the built-in `QAOAAnsatz()` provided by `Qiskit`.

8. **Result decoding:**
   The optimal angles are extracted (though not critical for this unencoded implementation) and the optimal bitstring derived from the optimization results is decoded using the encoding matrix. This yields the optimal integer portfolio calculated for each iteration. At this juncture, the optimal combination of integer shares within the selected portfolio, along with the corresponding budget required, can be displayed on the terminal.

   This step serves a crucial verification purpose: to confirm that the entire mathematical model, which was provided as an argument to the `.solve()` function, has been formulated correctly. Successful output thus confirms that the proposed optimization process was executed accurately.

9. **Portfolio processing:**
   The optimal portfolio results are processed by calculating the annualized

return, volatility, and required budget for each found portfolio, excluding those that exceed a predefined threshold.

10. **Visualization:** The processed data structures are visualized within the Markowitz Efficiency Frontier. The objective is to position the found optimal portfolios in the upper part of the scatter plot, close to the frontier. The final positions are contingent upon the parameters set in the model.

Once the optimal solutions are confirmed to be within the set budget threshold and positioned favorably toward the efficiency frontier, we can conclude that the unencoded program is fully functional and proceed to the encoded portion of our use case.

Portfolio Optimization with encoded QAOA

Having validated our optimal solutions using the QAOA algorithm implemented in Qiskit, we now aim to replicate the same optimization logic using the [4,2,2] quantum error-correcting code. For demonstrative and functional purposes, we approximate the number of ancilla qubits by using only two ancillas throughout the entire circuit.

Our primary objective is to implement an encoded QAOA workflow that yields optimal or near-optimal results, comparable to those obtained via Qiskit's native, unencoded solvers (e.g., `.solve()`). To achieve this, two key components are required:

1. A parameterized Ansatz that accurately represents the Ising Hamiltonian derived from the portfolio optimization objective function.

2. A post-processing decoding logic that maps the physical measurement outcomes back to valid logical solutions interpretable within the original mathematical model.

1. Ansatz construction To construct the encoded Ansatz, we implement the Ising Hamiltonian using quantum gates compatible with the [4,2,2] encoding scheme, following the gate decompositions proposed by Infleqtion [29]. Specifically, this involves encoded versions of R_z, controlled-R_z, and R_x gates. While R_x and controlled-R_z gates were already implemented in our earlier Max-Cut experiments, the encoded R_z gate required independent verification. We mathematically confirmed the equivalence between the encoded R_z implementation (as described in Infleqtion's documentation [29]), Figure 5, and Qiskit's native unencoded R_z gate; the verification process could be found in the Git Hub of our research team [51] and can be observed in Listing 2.8. This validation ensures the correctness of the encoded gate, which we now integrate into our Ansatz.

```python
def logical_rz(logical_target : int, theta: float, qvec:
    QuantumRegister, anc: QuantumRegister, qc: QuantumCircuit  ):

    block_id = logical_target // 2
    pos_inside_block = logical_target % 2

    offset = block_id * 4

    qc.barrier()
    if(pos_inside_block == 1):


        qc.cx(qvec[offset], anc)
        qc.cx(qvec[offset + 1], anc)
        qc.rz(theta, anc)
        qc.cx(qvec[offset], anc)
        qc.cx(qvec[offset + 1], anc)


    else:

        ## rz

        qc.cx(qvec[offset], anc)
        qc.cx(qvec[offset + 2], anc)
        qc.rz(theta, anc)
        qc.cx(qvec[offset], anc)
        qc.cx(qvec[offset + 2], anc)

        ##
    qc.barrier()
```

Listing 2.8: Implementation of the encoded R_z gate

2. Decoding and post-selection logic The encoded circuit yields measurement results over physical qubits. Our goal is to decode these into logical bitstrings that can be directly evaluated against the original Quadratic Unconstrained Binary Optimization (QUBO) model, enabling fair comparison with unencoded results.

For this prototyping phase, we assume the mathematical correctness of both the Ising Hamiltonian and the underlying optimization model. Therefore, to isolate and validate only the decoding and post-selection components, we temporarily employ Qiskit's built-in `QAOAAnsatz()` to generate the Ansatz. This allows us to focus exclusively on the post-processing pipeline. The verification procedure proceeds as follows:

a. Construct a parameterized Ansatz automatically from the Ising Hamiltonian using `QAOAAnsatz()`, with three QAOA layers (matching the unencoded case), and append measurement operations. Define the simulation backend using `AerSimulator()` and create a pass manager for circuit transpilation.

b. Print the total number of qubits in the circuit and the ordering of its free parameters to ensure consistency.

c. Initialize algorithmic parameters (e.g., 100 optimization iterations) and run Qiskit's QAOA solver to obtain optimal variational parameters. This phase of the process is fundamental to ensure the correctness of the parameter optimization for the current encoded model. This assurance is derived from the prior verification conducted in the preceding paragraph on the unencoded version of the problem, utilizing the default, automatic `QAOA` implementation provided by `Qiskit` as a reliable benchmark.

d. Reorder the optimized parameters to match the circuit's internal parameter ordering (as printed in step b), assign them to the Ansatz, and instantiate the optimized circuit. This procedure is paramount for achieving the optimal configuration of our custom Ansatz during the measurement phase. An erroneous parametrization order within the quantum circuit would inevitably lead to flawed measurement statistics, consequently resulting in suboptimal outcomes concerning the sampling process and the post-selection of results that satisfy the imposed constraints.

e. execute the transpiled circuit on the chosen backend, and sample the measurement outcomes for subsequent decoding. It is critical that the number of measurement "shots" used for each sampling run remains significantly below the total number of expressible combinations in our quantum circuit.

   This assumption is fundamental because we have elected to employ a post-processing filter on the results, which selects the final output based on the bit string that yields the minimum objective function value. It is therefore evident that if all possible bit strings were sampled, the final result would depend not on the efficacy of the optimization algorithm, but rather solely on the post-selection routine.

   In our specific case, for the full four-asset circuit utilized during verification, we adopted a shot count of one thousand ($N_{\text{shots}} = 1000$). This configuration ensures that we sample, at most, approximately one sixty-fifth (1/65) of all possible solutions.

f. Iterate over all sampled bitstrings, converting each into a list of integers for further processing. Once the bit strings have been extracted from our counts

dictionary, we proceed to process and transform them into a format that is easily analyzable and decodable. We chose to manage this process using `numpy.array` objects to facilitate all subsequent algebraic operations.

After conversion, individual strings are analyzed and discarded if the ancilla qubits exhibit a measured value of '1'. Such a result signifies the occurrence of an error within the quantum circuit during computation.

Subsequently, we decode the remaining qubits in the string that contain the useful encoded information. For this purpose, an external function is specifically called at the end of the procedure. This function maps the states from the encoded binary code space to the decoded solution space, setting a "flag" if the proposed solution falls outside the admissible code space constraints.

Based on the return value of this decoding flag, which, in our case, must be different from `None` to be considered valid, we proceed with the core elaboration of the post-processing filter to finally extract the bit string that corresponds to our optimal solution.

g. Evaluate the energy (i.e., objective function value) of each processed bitstring using the original QUBO model. Store results in a dictionary mapping bitstrings to their corresponding energies, the ancillary flag and its count. Should the selected bit string possess a key already present in the dictionary, a situation that occurs because multiple physically sampled strings can map to the same logical solution string, we proceed by adding only the elements that are not yet present in the dictionary. Conversely, for entries that are already present, we update the existing record by aggregating the relevant data (as we can see in Listing 2.9).

```
1    if str_decoded_bitstring not in possible_solutions:
2        possible_solutions[str_decoded_bitstring]=(qp.
   objective.evaluate(decoded_bitstring), anc_flag, count)
3    else:
4        obj_cost, anc_flag, old_count = possible_solutions[
   str_decoded_bitstring]
5        new_count = old_count + count
6        possible_solutions[str_decoded_bitstring] = (obj_cost,
    anc_flag, new_count)
7
```

Listing 2.9: Energy evaluation

h. Sort the dictionary by ascending energy values to prioritize low-energy (i.e., near-optimal) candidates (as shown in Listing 2.10).

```
1         sorted_dict = sorted(possible_solutions.items(), key=
    lambda item: item[1], reverse=False)
2
```

Listing 2.10: Sorting by energy

i. Iterate through the sorted dictionary and identify the first feasible solution that satisfies all problem constraints (e.g., budget, cardinality). Feasibility is assessed according to the original mathematical model.

j. Convert the selected feasible bitstring into an integer portfolio vector, append it to the list of optimal portfolios, and compute the associated budget allocation. By displaying these values on the terminal, we can immediately perform a preliminary check to ensure that our program is respecting the imposed budget constraints within the defined tolerance limits. An unjustifiably high outlay would signify a malfunction within our optimization and post-selection processes, indicating that the constraints were not correctly enforced. Finally, we proceed to calculate the `post selection rate` to ascertain the proportion of error-free counts relative to the total number of sampled counts. We anticipate that in the noise-free case, the post-selection rate will be equal to 100%, whereas in the noisy case, this rate will fall significantly below this threshold.

k. Reuse and adapt the post-processing routines from the unencoded case to compute annualized return and volatility for each optimal portfolio (point 8, of the file `PO_unencoded_QAOA`), then plot the results on the Markowitz efficient frontier (point 9, of the file `PO_unencoded_QAOA`).

l. If the resulting portfolios lie near the upper-efficient frontier, as observed in the unencoded case, we conclude that our decoding and post-processing logic has been functionally validated. This consideration is particularly relevant for scenarios involving a large number of assets in the portfolio, though it is not strictly mandatory for smaller cases, such as the one investigated in this study.

Encoded noiseless QAOA implementation Building on the validation above, we now implement the full encoded QAOA workflow under noiseless conditions:

1. Install all required Python libraries that you need, that are the same as before in our case.

2. Import necessary functions and reduce the problem size to the last three assets with a total budget of 500, resulting in a 6-logical-qubit instance. Similarly as

in the previous case, we must import the functions that yield the fundamental data structures processed from the input financial dataframe. Furthermore, we import the function which, taking these derived structures as input, processes them to furnish the Hamiltonian and all auxiliary tools necessary for realizing our optimization process, including the bidirectional mapping between the encoded and unencoded solution spaces.

3. Define problem-specific parameters (e.g., risk aversion, expected returns, covariance matrix). The parameters of the optimization problem are defined by extracting the necessary data structures and mathematical objects required for constructing the Ansatz of the encoded circuit. As in the previous instances, this is achieved by utilizing the functions contained within the two custom-developed file libraries.

   Crucially, the extracted objects are analogous to those retrieved for the unencoded case, ensuring consistency across both formulations of the optimization problem.

4. Import all useful modules, define the parameters and local variables to construct the encoded Ansatz. In particular, we require specific libraries such as `QuantumRegister` and `QuantumCircuit`. These components are essential for constructing our customized circuit-Ansatz, enabling its correct execution with the optimal parameters that will be determined and optimized by the Qiskit QAOA process. We shall utilize the `Parameter` function, available within the `qiskit.circuit` library, to formally define the parametrized angles $(\vec{\gamma}, \vec{\beta})$ that are subject to optimization by the classical solver.

5. Implement the encoded R_z gate and construct a custom encoded QAOA Ansatz that exactly mirrors the structure of the Ising Hamiltonian. Since the Ansatz must perfectly emulate the energy profile defined by our problem Hamiltonian ($\mathcal{H}$), its structure must incorporate contributions derived from both linear and quadratic interactions.

   The Ansatz will feature a term for the linear interactions, where the parametrized angles are multiplied by a factor of $2h_i$, and a term for the quadratic interactions, where a separate parametrized angle is multiplied by a factor of $2J_{ij}$.

   The constant offset term may be omitted for the purpose of our optimization, as it only introduces a uniform shift across the entire energy profile without altering the relative energies of the individual solutions to be selected.

6. Instantiate the full QAOA circuit using the custom Ansatz, append measurements, define the `AerSimulator` backend, and configure a pass manager for transpilation, than transpile the circuit on the selected backend.

7. Print the number of physical qubits and parameter ordering of the encoded circuit for verification. It must be noted that, as previously indicated, a mismatch in the parameter ordering within the quantum circuit during the assignment of the optimized values would lead to an erroneous execution of the circuit and, consequently, to inaccurate optimization results.

8. Reuse the optimization and parameter assignment workflow from the prototyping phase (steps c. and d. above) to obtain and apply optimal angles to the encoded Ansatz. In particular, the primary hyperparameters that must be defined are the number of iterations and the number of measurement shots (N_{shots}) for the classical optimization process. Furthermore, it is necessary to initialize the empty lists that will serve as data structures to store the results yielded by the optimization procedure.

9. Transpile and execute the encoded circuit, then extract the sampled measurement counts. It is imperative that the number of measurement counts (N_{shots}) utilized remains significantly lower than the total number of expressible bit strings that the quantum circuit can generate. This constitutes a mandatory assumption designed to ensure the correct and reliable validation function of our post-selection filter. A sufficiently high number of measurement shots implies that we have potentially explored the entirety of the solution space. If this occurs, the correct theoretical solution will almost certainly be present within the sampling results. This scenario creates a validation ambiguity: we could be employing an efficient optimization algorithm without explicitly recognizing its efficacy, as the final result might be determined by exhaustive sampling rather than algorithmic superiority. Conversely, if we explore only a small portion of the solution space through under-sampling, we are then fundamentally reliant on having an inherently efficient and correctly designed algorithm to locate the optimal solution.

10. The sampled results are first converted from bit strings into a processable integer array format. Subsequently, we must evaluate whether the ancilla qubits are all measured in the zero state ($|0\rangle$). If this condition is not met (i.e., if any ancilla is measured at '1'), the corresponding solution is afflicted by an error and must be discarded immediately. Conversely, if all ancillas are verified to be in the zero state, the solution is deemed valid and passed to the decoding process, which is delegated to an appropriate external function.

11. For each block of four physical qubits, which corresponds to one logical qubit under the specific [4,2,2] encoding scheme, decoding is performed using an appropriate external function specifically designed for this logic.

    In principle, we opted for a "dictionary" logic, where the physical (encoded) bit strings are confronted with a predefined dictionary. This structure utilizes

all possible physical strings as the dictionary keys, with the corresponding logical decoded string serving as the value.

The function handles out-of-code-space detection as follows: if a sampled string falls outside the defined code space, the function returns the flag `None`. Conversely, if the string is found within the dictionary, the function returns the decoded logical string.

12. Each reconstructed logical bit string must be evaluated using the QUBO model (or the objective function of the Quadratic Program). The calculated results are then stored within a dictionary structure, which simultaneously records the calculated objective function value, the ancilla flag status (for error indication), and the cumulative shot count for that specific logical solution. The logical decoded bit string may be associated with multiple physical bit strings sampled from the quantum circuit. To avoid the problem of redundancy in the result set, we implemented a strategy where: if the logical key already exists in the dictionary, we update the count value associated with that solution by aggregating the new sample; conversely, if the key is novel, a new element is inserted into the dictionary. This aggregation process ensures that statistics are accurately collected for each unique logical solution. This dictionary is subsequently sorted in ascending order based on the calculated energy value.

13. Iterate over the sorted logical solutions, check feasibility against problem constraints, and collect all feasible optimal portfolios.

14. For each feasible portfolio, compute budget allocation (point 8, of the file `PO_unencoded_QAOA`), annualized return, and volatility, then plot the points on the Markowitz efficient frontier (point 9, of the file `PO_unencoded_QAOA`).

15. Finally, compare these results with those obtained in the prototyping phase (where `QAOAAnsatz()` was used). If the efficient frontiers align closely, this confirms the correctness of both the encoded Ansatz and the decoding/post-selection pipeline, thereby validating the entire encoded noiseless QAOA implementation.

Encoded noisy QAOA In the context of quantum circuits exposed to noisy environments, two main approaches can be adopted. The first involves selecting a `FakeBackend` from those available in Qiskit, extracting its associated noise model, and executing the circuit using this model after the parameter training phase, which remains noiseless. The second method consists of running the same circuit directly on real quantum hardware provided by IBM, allowing for a more realistic evaluation of the program performance.

In the case where the noise model is extracted from a `FakeBackend` and applied to the execution of the optimized circuit, only two minor code modification is required: transpilation and execution (as we can see in Listing 2.11 and 2.12). It is sufficient to select a fake backend capable of supporting the dimensions of our quantum circuit; in this study, we selected `FakeTorino()` for this purpose. Subsequently, the simulation is executed directly on this noise model.

It must be noted that executing simulations on a local system, such as a personal computer, necessitates high computational times for each individual iteration. This constraint imposes significant limits on the number of feasible iterations when adopting this simulation approach.

Definition of the noisy simulation and pass manager:

```
1  circuit=encoded_422_Portfolio_optimization_qaoa_algorithm_ansatz(
       logical_qubit, layer_count, thetas)
2  circuit.measure_all()
3  circuit.parameters
4
5  from qiskit_ibm_runtime.fake_provider import FakeTorino
6  from qiskit.transpiler.preset_passmanagers import
       generate_preset_pass_manager
7
8  fake_device = FakeTorino()
9
10 pm = generate_preset_pass_manager(optimization_level=3,
11                                   backend=fake_device)
12 candidate_circuit = pm.run(circuit)
```

Listing 2.11: Noisy simulation and pass manager setup

Definition of the noisy transpilation and sampling:

```
1      job = fake_device.run(optimal_circuit, shots=shots)
2      result_noisy = job.result()
3      counts_noisy = result_noisy.get_counts()
```

Listing 2.12: Noisy transpilation and sampling

To run the program on real quantum hardware, only the initial backend definition needs to be slightly modified (as we can observe in Listing 2.13). This involves inserting IBM credentials to enable connection to the desired backend:

```
1  QiskitRuntimeService.save_account(token="YOUR_API_TOKEN", instance
       ="YOUR_INSTANCE", overwrite=True)
2  service = QiskitRuntimeService()
3  real_device = service.backends(name="ibm_torino")
```

Listing 2.13: Execution on real IBM hardware

2.2.3 In-depth analysis of the quantum hardware ibm_torino

We provide a detailed technical analysis of the real quantum hardware utilized for the execution of the noisy simulations: the IBM_Torino system.

As reported in Table 2.1, the device is based on the "Heron r1" processor architecture, which features 133 physical qubits. This high qubit count is more than adequate for hosting the dimensionality of our encoded optimization problem.

The following section reports the technical specifications of the quantum hardware employed.

Table 2.1: Configuration parameters and noise metrics of the ibm_torino backend (Heron r1).

Metric	**Value**	**Notes**
Number of Qubits	133	
Status	Online	
Region	Washington DC (us-east)	
Processor Type	Heron r1	
Error and Coherence Metrics		
2Q Error (median)	2.84×10^{-3}	
2Q Error (layered)	1.11×10^{-2}	
2Q Error (best)	1.25×10^{-3}	
CZ Error (median)	2.837×10^{-3}	
SX Error (median)	3.125×10^{-4}	
Readout Error (median)	3.09×10^{-2}	
T1 (median)	181.59	μs
T2 (median)	126.1	μs
Operational Capacity		
Basis Gates	$cz, id, rx, rz, rzz, sx, x$	
CLOPS	220	K (Thousands)
QPU Version	1.0.105	

The backend parameters for **ibm_torino** (Processor Type: **Heron r1**) provide a quantitative measure of the processor's capacity and fidelity.

- **Qubits (133):** Indicates the total number of physical qubits available on the processor. A larger quantity allows for the encoding of more complex problems.
- **Basis gates:** The minimal set of universal quantum operations natively supported by the processor ($cz, id, rx, rz, rzz, sx, x$). All quantum circuits are

transpiled into this gate set.

- **2Q error (median, layered, best):** The median error probability for two-qubit operations (**2Q** gates), typically CZ or CNOT. The median value serves as the standard measure. The *layered error* accounts for the cumulative error across one layer of the circuit.

- **CZ error (median) and SX error (median):** The median error probabilities for the fundamental two-qubit (**CZ**) and single-qubit (**SX**, Square root of X) gates used as basis gates.

- **Readout Error (median):** The probability of erroneously measuring the state of a qubit (e.g., measuring $|1\rangle$ when the actual state was $|0\rangle$).

- **T1 (median):** The longitudinal or population **relaxation time**. It measures the average time required for an excited qubit to return to its ground state ($|1\rangle \to |0\rangle$). A higher value indicates greater stability.

- **T2 (median):** The transversal **decoherence time**. It measures the average time required for the quantum state (phase) to decay due to environmental interactions. $T_2 \leq 2T_1$; a higher T_2 indicates greater coherence.

- **CLOPS (Circuit layer operations per second):** A benchmark metric that indicates the processor's execution speed, quantified in thousands of quantum circuit layers per second.

We now proceed to visualize the topology of our quantum device in Figure 2.2. Within the image, one can distinguish zones of varying lightness and darkness; these color variations describe the gradient of the `read_out_error` across the qubits of our machine. Specifically, a darker color indicates a lower error rate, while a lighter color signifies a proportionally higher error rate at those particular qubit locations, or at the interconnecting links, within the system's topology.

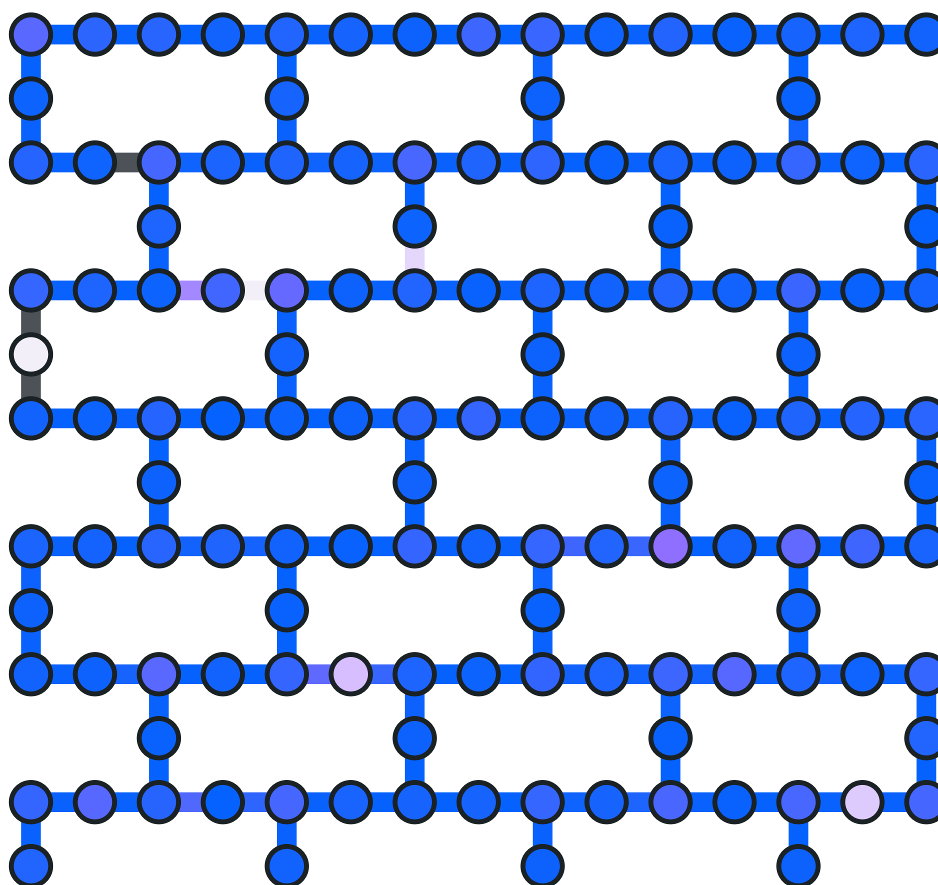


Figure 2.2: Topology of the **ibm_torino** processor (Heron r1) visualizing the **readout error** for each qubit. The values displayed represent the probability of measurement error for each physical qubit, indicating the spatial variation of readout fidelity across the chip.

We can also achieve a more direct and immediate visualization of the readout error values for each qubit within the device's topology by utilizing a dedicated histogram (that can be observed in Figure 2.3) provided by the manufacturer (IBM). This representation correlates the individual qubit identifier with its measured error rate, offering a quantitative complement to the spatial map analysis previously discussed.

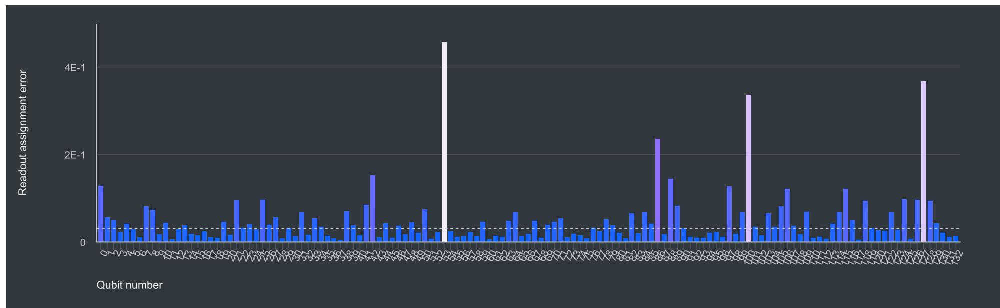


Figure 2.3: Distribution of the **readout error** values per qubit. This histogram quantitatively illustrates the correlation between each physical qubit (ID) and its measured readout error rate within the topology of the real quantum device.

For illustrative purposes, and to demonstrate the diverse plotting capabilities provided by the IBM Quantum Experience interface, we elected to present only the graphs pertinent to the readout error. Should the reader require an in-depth analysis of additional hardware parameters, these may be readily accessed and explored by selecting the available options within the IBM user interface, authenticated via personal credentials.

Chapter 3

Results

Given the development processes of our algorithms and the associated methodology, we now proceed to analyze the results obtained for Max Cut and Portfolio Optimization.

3.1 Results from Max-Cut

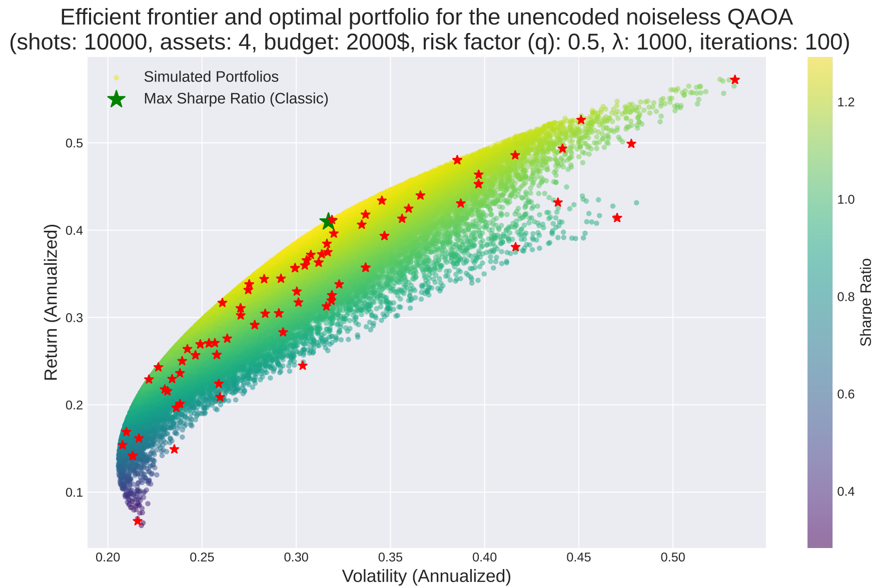
As previously referenced, the Max-Cut problem served a purely demonstrative function, acting as a pilot case where the known optimal solutions [50] are well-established. The primary objective was to take this initially unencoded circuit and apply our [4,2,2] encoding logic. This allowed us to sample the results and subsequently decode them through our implemented logic, thereby demonstrating the correctness of our error detection algorithm. This was validated by successfully retrieving the identical solutions established by NVIDIA for the 4 node ring connected graph 2.2.1.

The resulting encoded circuit yielded results perfectly equivalent to those of the unencoded version, specifically the bit strings $'\mathbf{0101}'$ and $'\mathbf{1010}'$, which is consistent with the established solutions for the case under analysis.

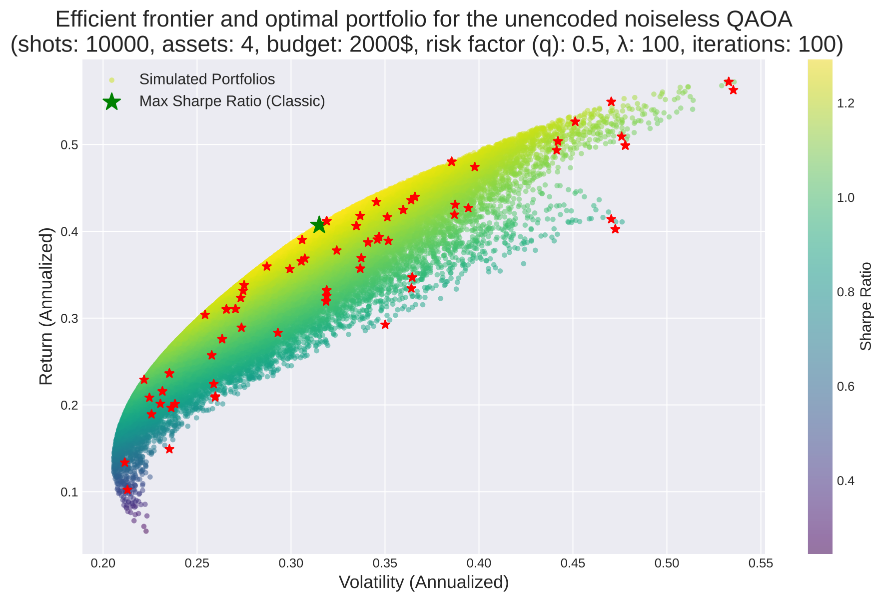
3.2 Results from unencoded implementation using Qiskit's built-in methods

We first examine the outputs of the unencoded implementation, which relies entirely on Qiskit's native methods (e.g., `.solve()`), over 100 iterations. In this setup, we vary the penalty term λ and the risk aversion parameter q, while keeping the budget fixed at 2000. This analysis provides a clear understanding of how the optimal portfolio allocations shift across the Markowitz efficient frontier as these

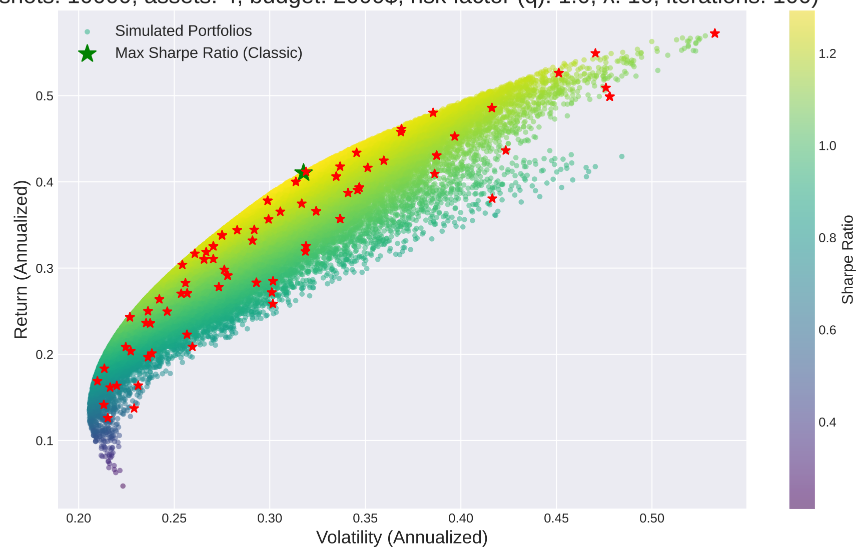
parameters change. These results will serve as a benchmark for validating the outcomes obtained from our custom post-processing pipeline using `QAOAAnsatz()` and the encoded implementation. The following figures (Figure 3.1a, 3.1b, 3.1c, 3.1d, 3.1e, 3.1f, 3.1g, 3.1h) present the results of the unencoded implementation executed with Qiskit's built-in solvers.



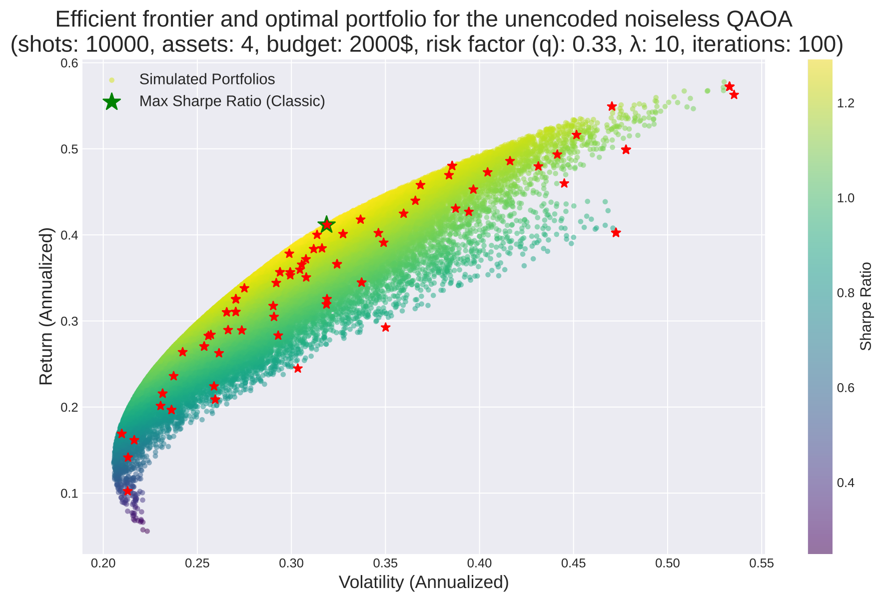
(a) Plot illustrating the Markowitz efficient frontier generated under specific optimization parameters: 100 iterations, 3 QAOA layers, a budget constraint of Budget $= 2000$, a penalty coefficient of $\lambda = 1000$, and a risk-aversion parameter of $q = 0.5$.



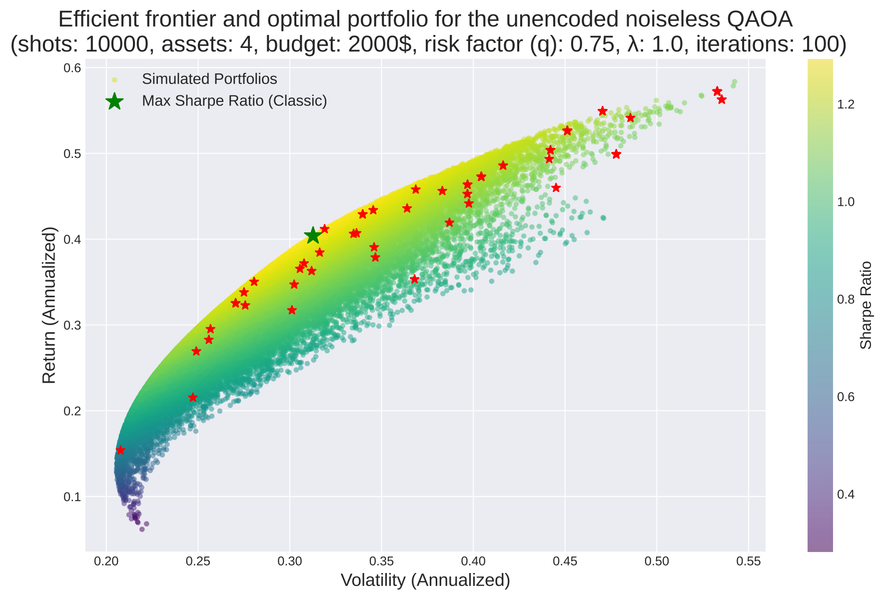
(b) Plot illustrating the Markowitz efficient frontier generated under specific optimization parameters: 100 iterations, 3 QAOA layers, a budget constraint of Budget = 2000, a penalty coefficient of $\lambda = 100$, and a risk-aversion parameter of $q = 0.5$.



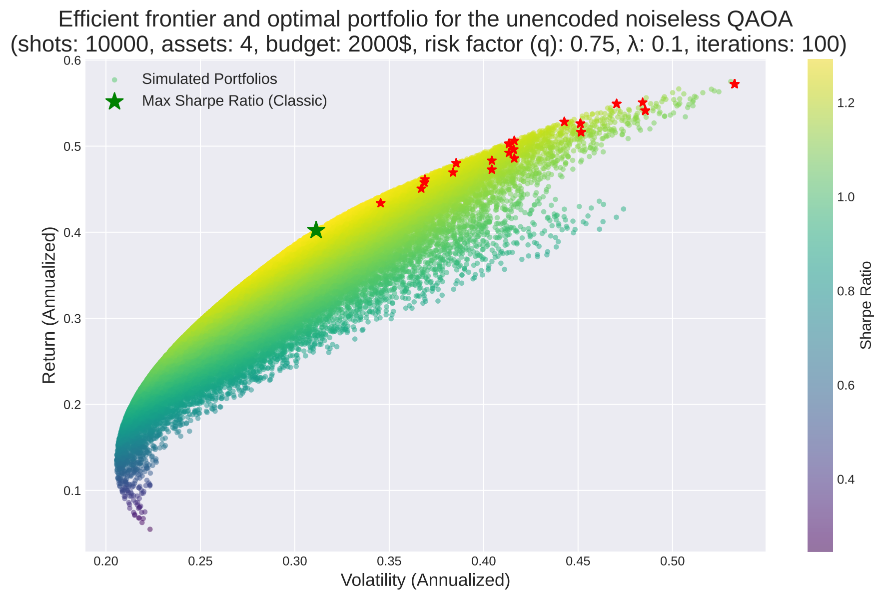
(c) Plot illustrating the Markowitz efficient frontier generated under specific optimization parameters: 100 iterations, 3 QAOA layers, a budget constraint of Budget $= 2000$, a penalty coefficient of $\lambda = 10$, and a risk-aversion parameter of $q = 1$.



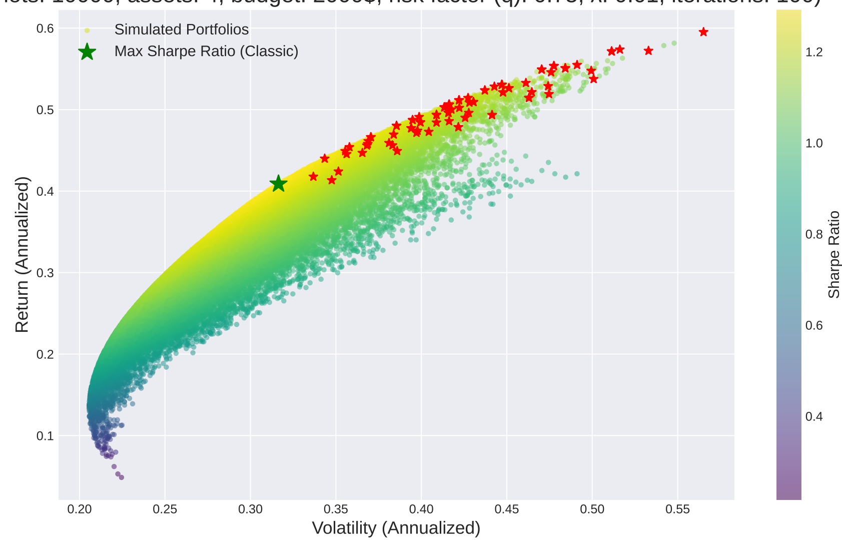
(d) Plot illustrating the Markowitz efficient frontier generated under specific optimization parameters: 100 iterations, 3 QAOA layers, a budget constraint of Budget = 2000, a penalty coefficient of $\lambda = 10$, and a risk-aversion parameter of $q = 0.33$.



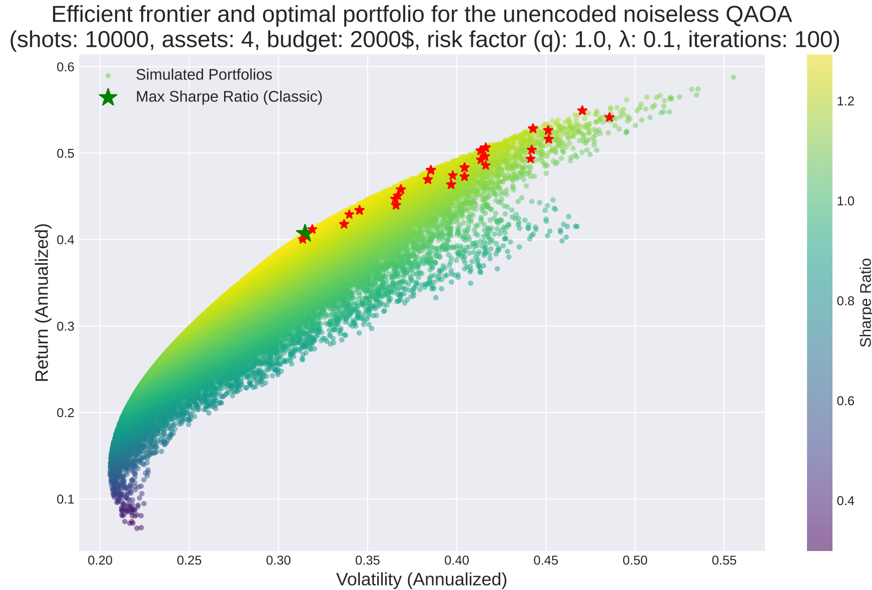
(e) Plot illustrating the Markowitz efficient frontier generated under specific optimization parameters: 100 iterations, 3 QAOA layers, a budget constraint of Budget $= 2000$, a penalty coefficient of $\lambda = 1$, and a risk-aversion parameter of $q = 0.75$.



(f) Plot illustrating the Markowitz efficient frontier generated under specific optimization parameters: 100 iterations, 3 QAOA layers, a budget constraint of Budget $= 2000$, a penalty coefficient of $\lambda = 0.1$, and a risk-aversion parameter of $q = 0.75$.



(g) Plot illustrating the Markowitz efficient frontier generated under specific optimization parameters: 100 iterations, 3 QAOA layers, a budget constraint of Budget $= 2000$, a penalty coefficient of $\lambda = 0.01$, and a risk-aversion parameter of $q = 0.75$.



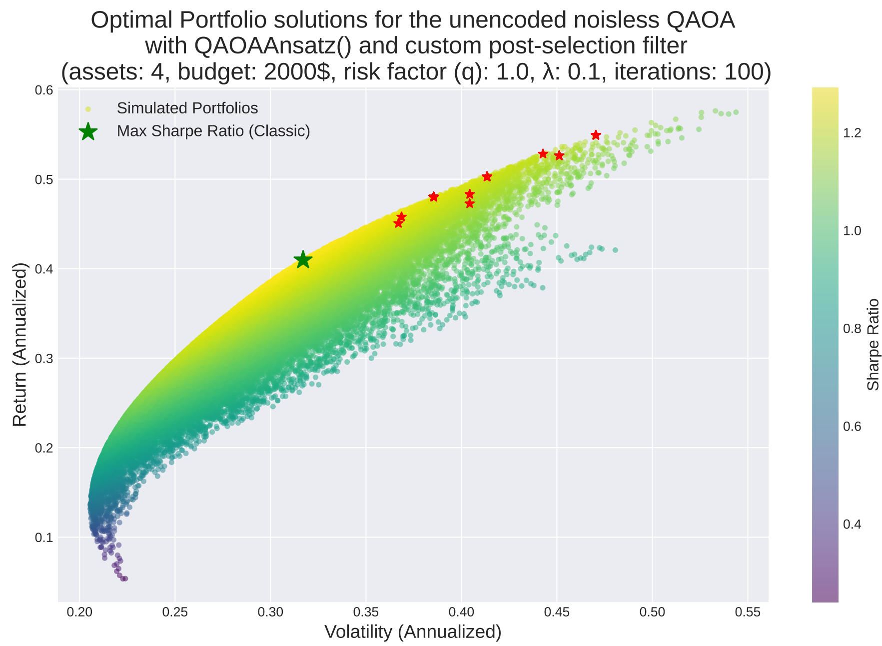
(h) Plot illustrating the Markowitz efficient frontier generated under specific optimization parameters: 100 iterations, 3 QAOA layers, a budget constraint of Budget = 2000, a penalty coefficient of $\lambda = 0.1$, and a risk-aversion parameter of $q = 1$.

Figure 3.1: Results from the unencoded Qiskit implementation (100 iterations, budget = 2000 and 10000 shots).

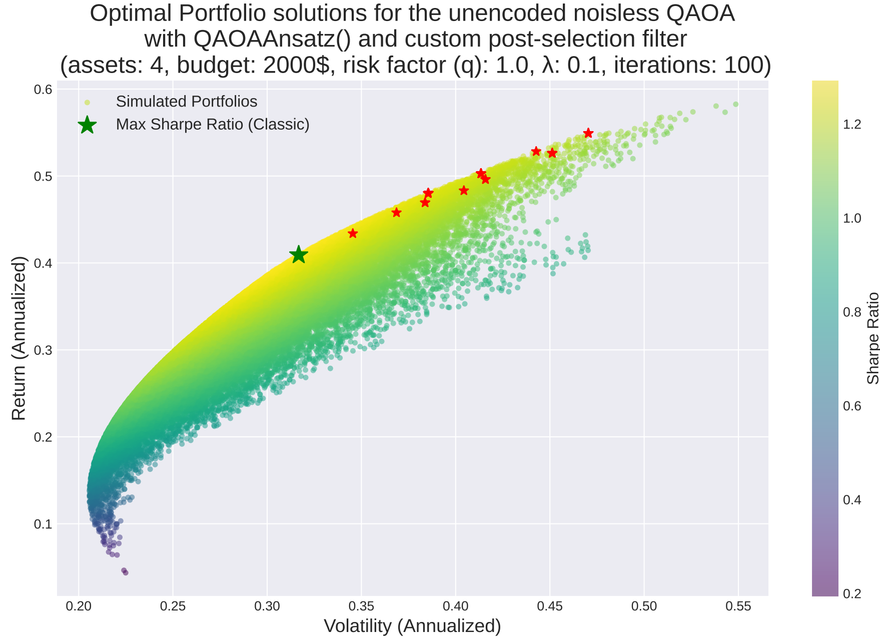
In general, we observe that a higher risk tolerance (i.e., lower q) leads to optimal portfolios clustered in the upper region of the Markowitz efficient frontier, characterized by higher expected returns but also greater volatility (as can be seen in Figure 3.1f, 3.1g, 3.1h). Conversely, lower risk tolerance shifts the optimal points toward the lower-left portion of the frontier, reflecting more conservative allocations (as shown in Figure 3.1a, 3.1b, 3.1d). Furthermore, varying the penalty term λ influences how closely the obtained solutions adhere to the efficient frontier, our ultimate validation target for the full dataset.

3.3 Results using `QAOAAnsatz()` and custom post-selection

We now analyze the outcomes of our custom algorithm, where the quantum circuit is explicitly constructed using `QAOAAnsatz()`, and results are filtered via our handcrafted post-selection logic, rather than relying on Qiskit's automated solvers. The following figures (Figure 3.2a and 3.2b) show the output for the full dataset (4 assets), with 100 iterations, budget = 2000, $\lambda = 0.1$, $q = 1$ and 10000 shots.



(a) First simulation of my system for the full dataset (4 assets), with 100 iterations, budget = 2000, $\lambda = 0.1$, $q = 1$ and 10000 shots



(b) Second simulation of my system for the full dataset (4 assets), with 100 iterations, budget = 2000, $\lambda = 0.1$, $q = 1$ and 10000 shots

Figure 3.2: Portfolio optimization results using `QAOAAnsatz()` and custom post-selection (4 assets, 100 iterations) for 10000 shots.

In both plots of Figure 3.2a and 3.2b, the optimal solutions cluster in the high-return region of the efficient frontier, consistent with the behavior observed in the unencoded Qiskit implementation. Notably, our post-selection method yields significantly lower dispersion and a tighter fit to the efficient frontier, confirming the effectiveness of our filtering strategy. To provide empirical validation that our system does not oversample a condition where the number of sampled bit strings is so high as to guarantee inclusion of the mathematically optimal solution, we implemented a decisive reduction in the number of measurement shots from $N_{\text{shots}} = 10000$ to $N_{\text{shots}} = 1000$ (as depicted in Figure 3.3). This new configuration ensures that we sample at most $1/65$ of the total potentially generable bit strings, which number 2^{16} (or 65536) in our code space. This system serves to verify that our algorithm yields results consistent with the optimization model and constraints, without the post-selection filter artificially validating the algorithm's actual efficiency.

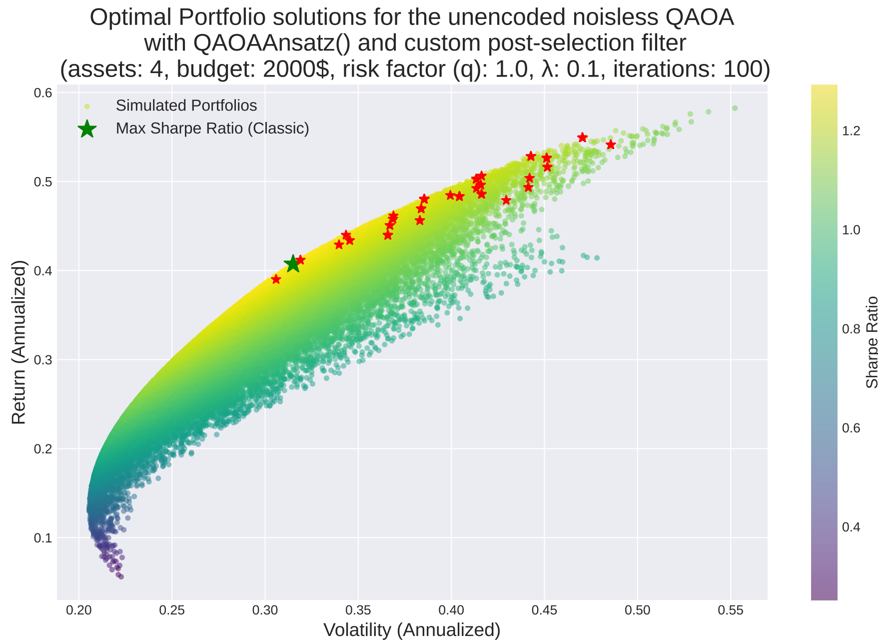


Figure 3.3: Plot illustrating the Markowitz efficient frontier generated for the portfolio optimization problem. The specific optimization parameters are: 100 iterations, 3 QAOA layers, a budget constraint of Budget = 2000, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$.

3.3.1 Expected impact on solution quality

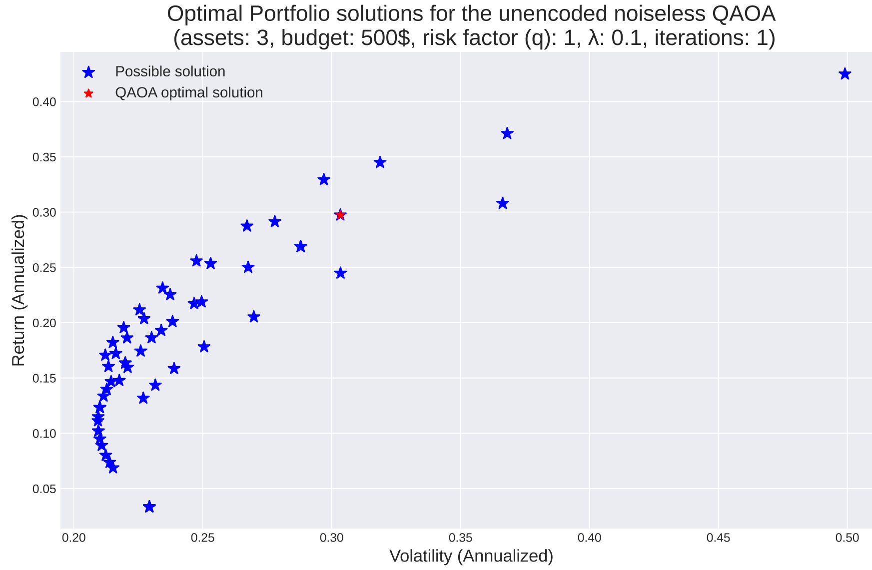
By decreasing the number of samples, we anticipate that the resulting solution set will be less thigh in the immediate vicinity of the efficient frontier. Specifically, we expect the output to be slightly more dispersed and less localized compared to the previous, higher-shot scenario. Having validated our post-selection pipeline, we now evaluate its performance on a reduced dataset comprising only the last three assets of the original set. This reduction serves two purposes: (i) it enables a direct comparison with the encoded Ansatz implementation, and (ii) it approximately halves the number of physical qubits required during the encoding phase, thereby improving resource efficiency. The necessity for this modified approach arises because our current compilation system proves inadequate for synthesizing quantum circuits of the size and complexity required by our 4-asset system, analogous to the issue encountered in the previously discussed case.

Given that the reduced 3-asset case contains significantly fewer solutions, precisely $2^6 = 64$, a plot generated against a large cluster of portfolios obtained via Monte Carlo methods (as previously done) would encompass a solution space considerably larger than the one admissible by our quantum system. Therefore, to

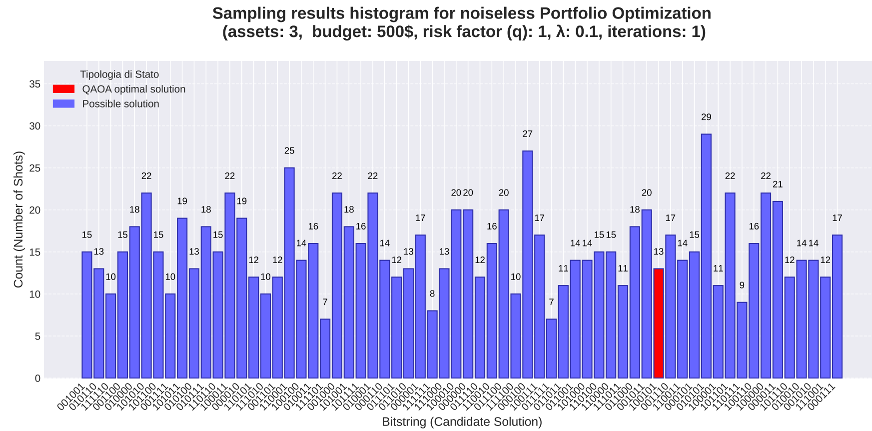
improve clarity and intelligibility, we decided to forego the Monte Carlo method. Instead, we present a plot focusing exclusively on the admissible solutions defined on the Markowitz efficient frontier, distinctly highlighting the specific solutions obtained by our quantum algorithm with a differentiated color. Consistent with the preceding rationale regarding sampling constraints and algorithm validation, all subsequent results presented herein have been evaluated using a fixed number of measurement shots, $N_{\text{shots}} = 1000$.

3.3.2 Results from the unencoded noiseless circuit implementation with QAOAAnsatz() and post-selection filtering

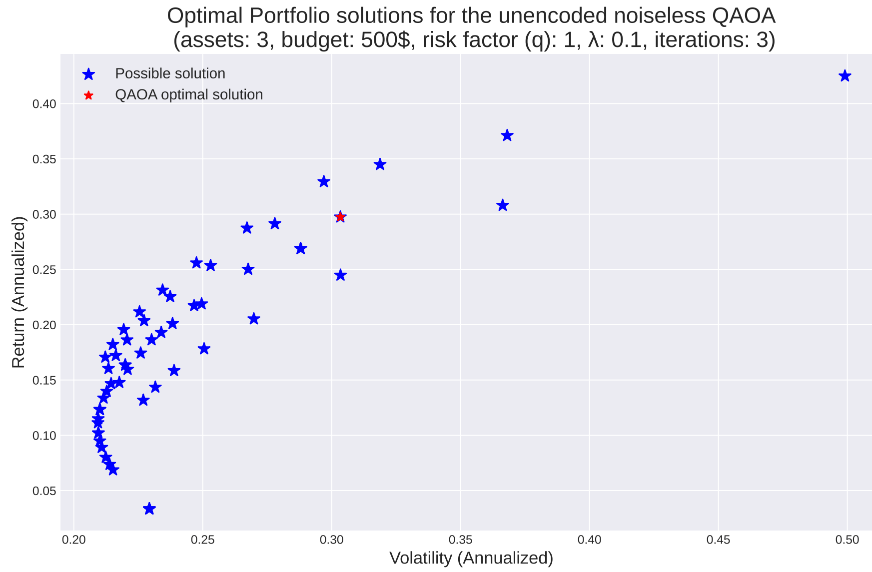
We now proceed to analyze the solutions favored as optimal by our system for each simulation conducted under a noiseless scenario. For each identified optimal solution, we will visualize the corresponding "star plot" and the correlated histogram for the distribution of the possible solutions to verify the combined trade-off between the expected return and the volatility presented by the selected portfolio composition. We will highlight in red the optimal solution finding using the logic of our algorithm to differentiate from other solutions. The following results (shown in Figure 3.4a, 3.4b, 3.4c, 3.4d, 3.4e, 3.4f, 3.4g, 3.4h, 3.4i, 3.4j) correspond to the 3-asset case, with budget = 500, $\lambda = 0.1$, $q = 1$, for different number of iterations.



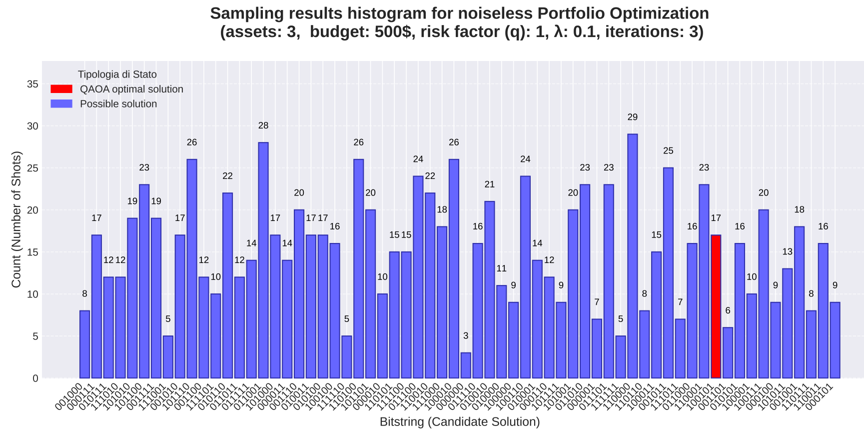
(a) Plot illustrating the graph of stars generated for the portfolio optimization problem. The specific optimization parameters are: 1 iteration, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



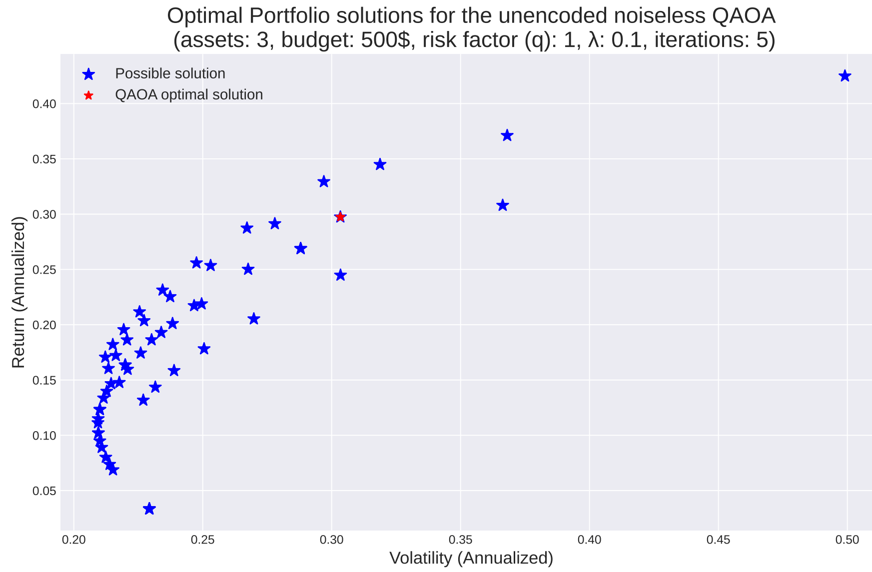
(b) Plot illustrating the histogram generated for the portfolio optimization problem. The specific optimization parameters are: 1 iteration, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



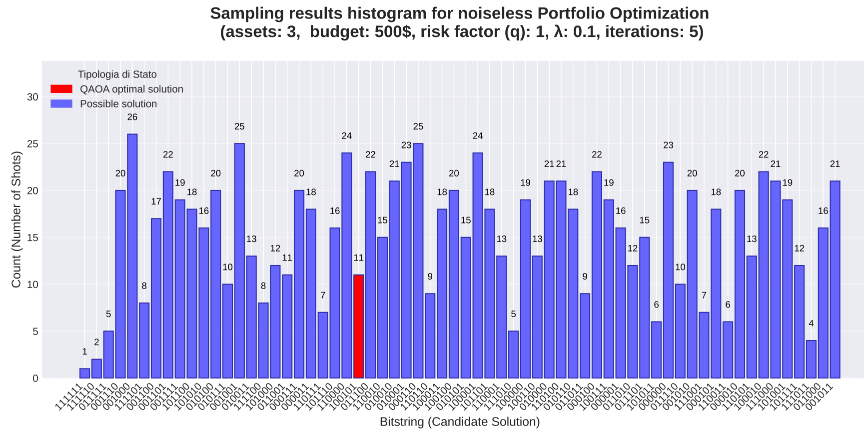
(c) Plot illustrating the graph of stars generated for the portfolio optimization problem. The specific optimization parameters are: 3 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



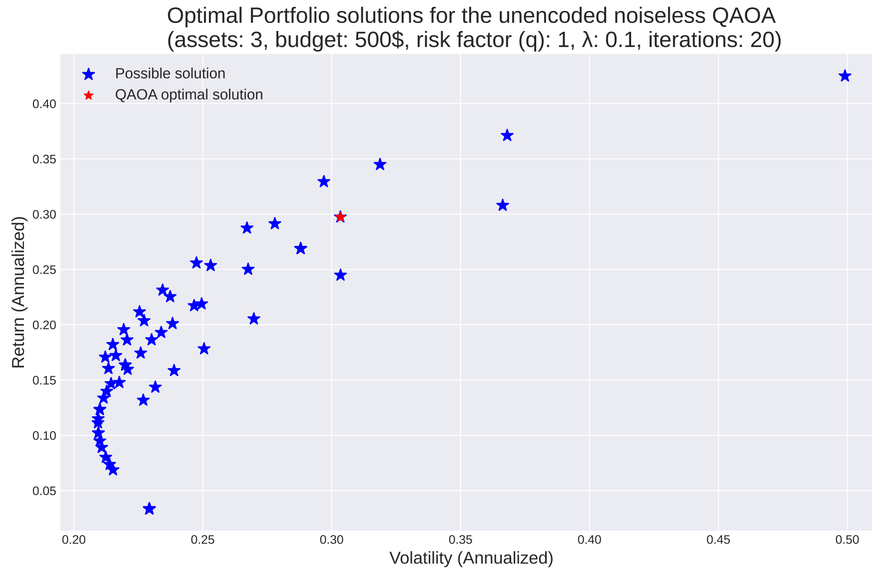
(d) Plot illustrating the histogram generated for the portfolio optimization problem. The specific optimization parameters are: 3 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



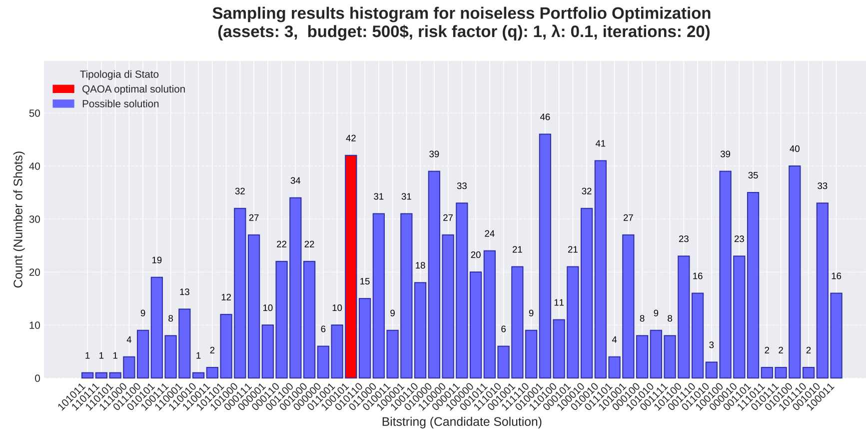
(e) Plot illustrating the graph of stars generated for the portfolio optimization problem. The specific optimization parameters are: 5 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



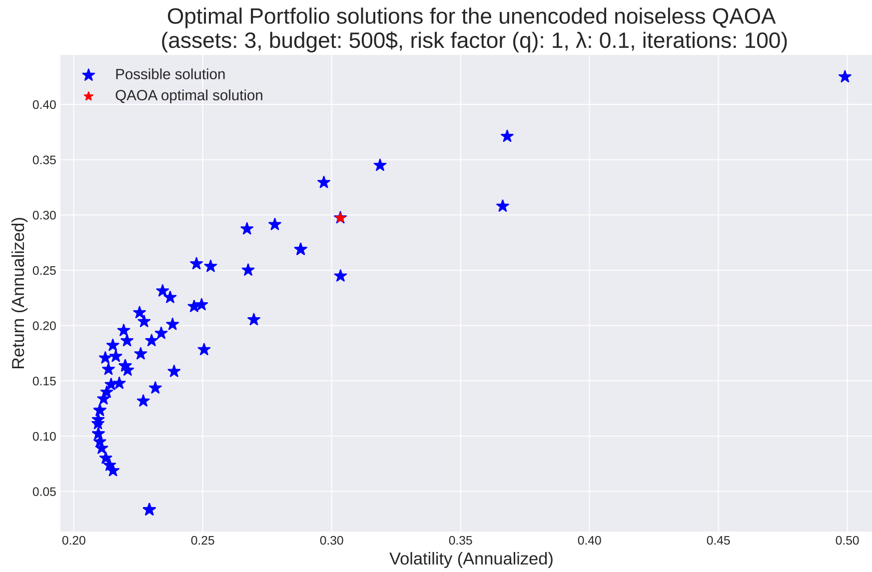
(f) Plot illustrating the histogram generated for the portfolio optimization problem. The specific optimization parameters are: 5 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



(g) Plot illustrating the graph of stars generated for the portfolio optimization problem. The specific optimization parameters are: 20 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



(h) Plot illustrating the histogram generated for the portfolio optimization problem. The specific optimization parameters are: 20 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



(i) Plot illustrating the graph of stars generated for the portfolio optimization problem. The specific optimization parameters are: 100 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$

Sampling results histogram for noiseless Portfolio Optimization
(assets: 3, budget: 500$, risk factor (q): 1, λ: 0.1, iterations: 100)

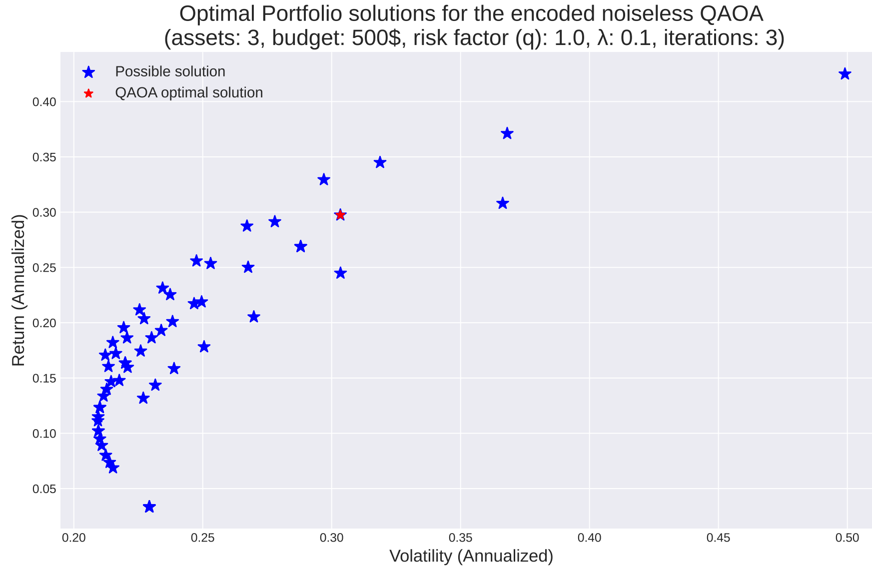
(j) Plot illustrating the histogram generated for the portfolio optimization problem. The specific optimization parameters are: 100 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$

Figure 3.4: Results of the portfolio optimization for the 3 asset case, illustrating the performance of the `QAOAAnsatz()` combined with a custom post-selection filter across optimization depths equal to three ($p = 3$ layers). The results are shown for $1, 3, 5, 20$, and 100 iterations.

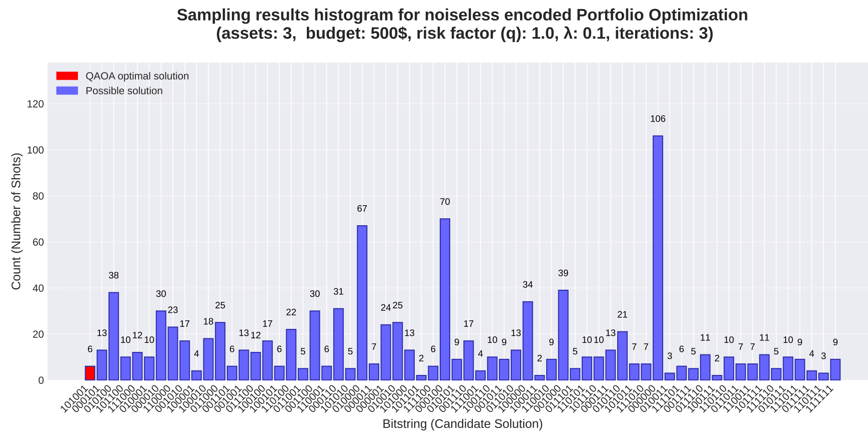
As illustrated by the plots in Figure 3.4, it is evident that the proposed algorithm is capable of correctly identifying the mathematically optimal solution (as shown in the star plots in Figure 3.4a, 3.4c, 3.4e, 3.4g, 3.4i). However, regarding the sampled optimal solution, a clear convergence is not observed. This leads to a discrepancy between the analytical optimum and the statistically sampled results, as further demonstrated by the histograms provided in Figure 3.4b, 3.4d, 3.4f, 3.4h, 3.4j.

3.4 Results from the encoded noiseless circuit implementation

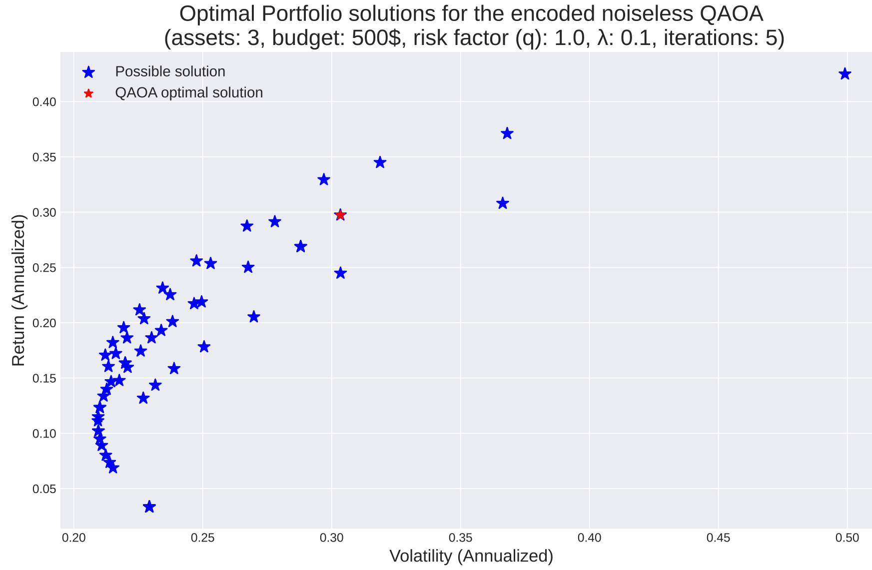
We now test our fully handcrafted encoded circuit combined with the custom post-selection logic. We expect similar outcomes to the `QAOAAnsatz()` based approach when applied to the reduced 3-asset dataset under noiseless simulation. The following figure (Figure 3.5) shows the results after different number of iteration, with budget $= 500$, $\lambda = 0.1$, and $q = 1$. We have chosen to reduce the number of experiments due to the cost in terms of time required for the execution of the encoded circuit.



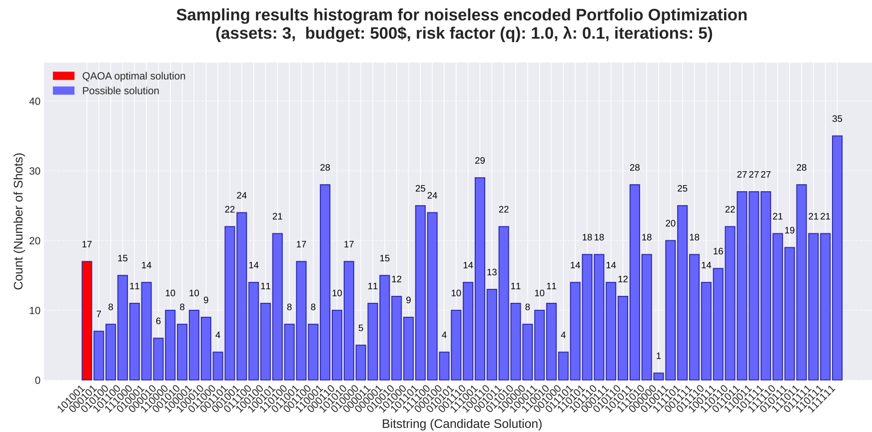
(a) Plot illustrating the graph of stars generated from the encoded [4,2,2] circuit, for the portfolio optimization problem. The specific optimization parameters are: 3 iterations, 3 QAOA layers, a budget constraint of Budget = 500, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



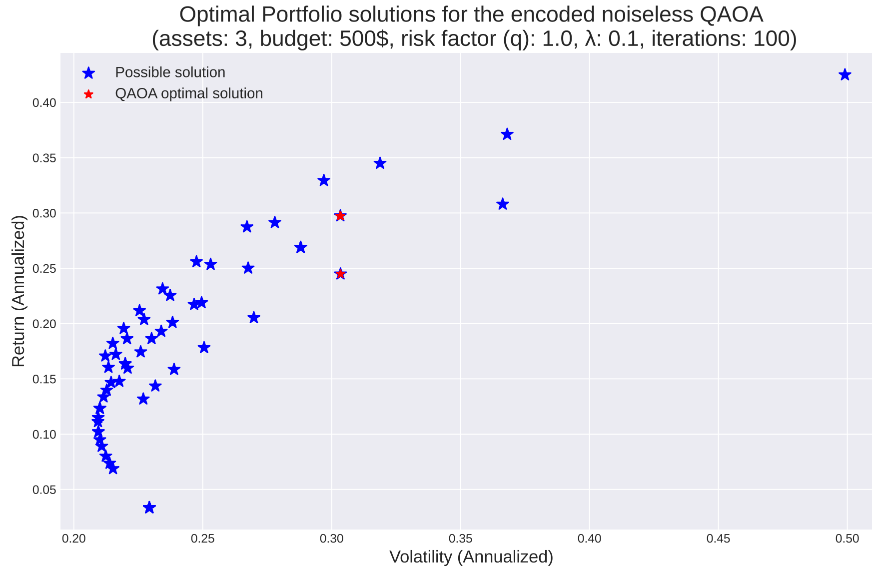
(b) Plot illustrating the histogram generated for from the encoded [4,2,2] circuit, for the portfolio optimization problem. The specific optimization parameters are: 3 iterations, 3 QAOA layers, a budget constraint of Budget = 500, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



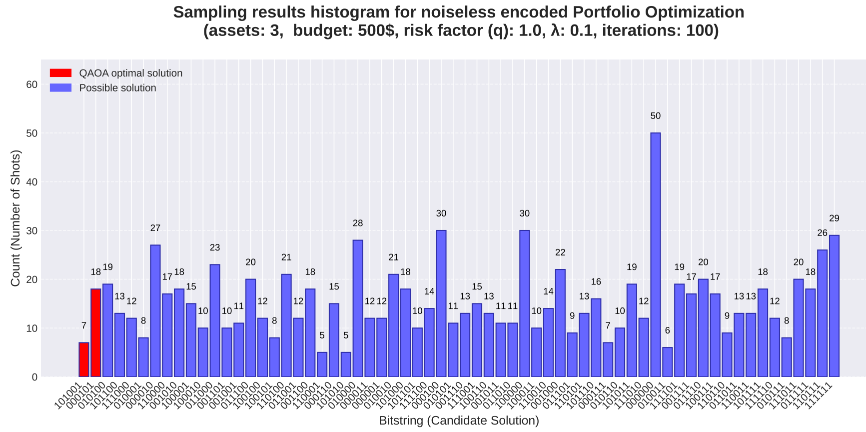
(c) Plot illustrating the graph of stars generated from the encoded [4,2,2] circuit, for the portfolio optimization problem. The specific optimization parameters are: 5 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



(d) Plot illustrating the histogram generated for from the encoded [4,2,2] circuit, for the portfolio optimization problem. The specific optimization parameters are: 5 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



(e) Plot illustrating the graph of stars generated from the encoded [4,2,2] circuit, for the portfolio optimization problem. The specific optimization parameters are: 100 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



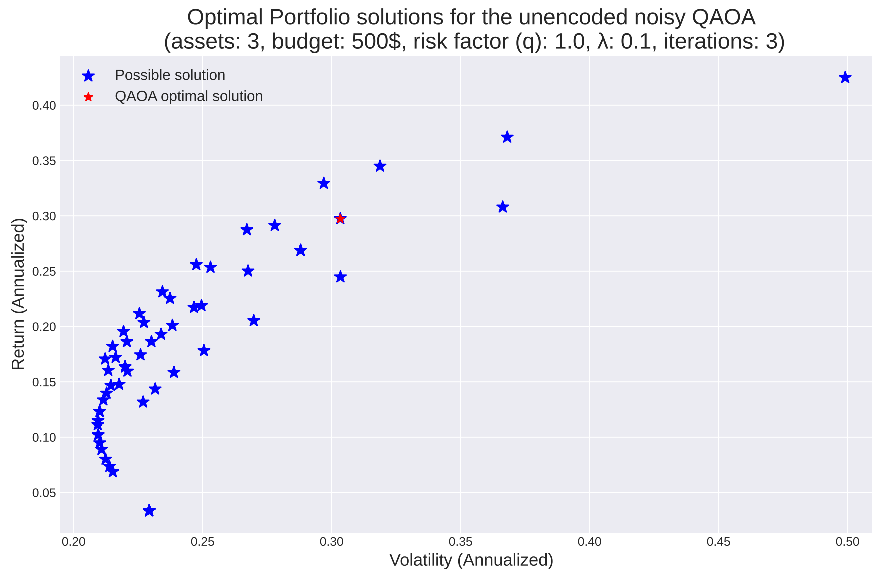
(f) Plot illustrating the histogram generated for from the encoded [4,2,2] circuit, for the portfolio optimization problem. The specific optimization parameters are: 100 iterations, 3 QAOA layers, a budget constraint of Budget = 500, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$

Figure 3.5: Results of the portfolio optimization for the 3 asset case, illustrating the performance of the encoded [4,2,2] Ansatz combined with a custom post-selection filter across optimization depths equal to three ($p = 3$ layers). The results are shown for $3, 5$, and 100 iterations.

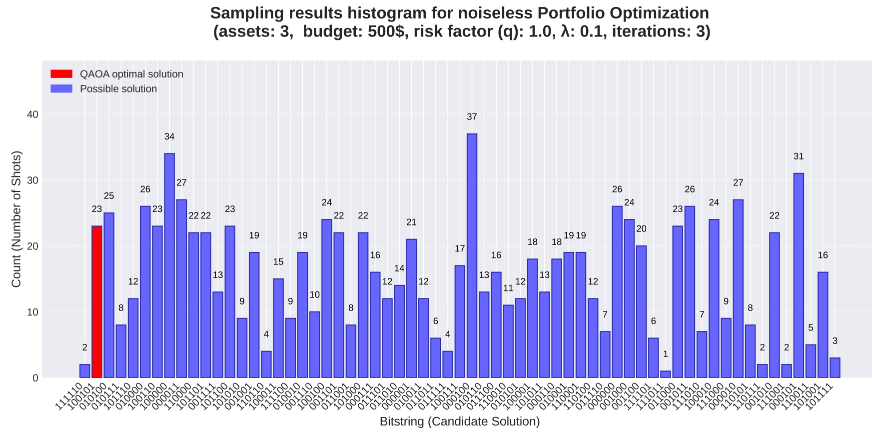
The plot of the "graph of stars" reproduces the same optimal portfolio distribution observed with the `QAOAAnsatz()` implementation for the experiments with 3 and 5 iterations (Figure 3.5a, 3.5b, 3.5c and 3.5d), and a very similar one for 100 iterations (Figure 3.5e and 3.5f), with a post-selection rate of 100%, confirming the correctness of our encoded circuit design.

3.5 Noisy simulation results

We now assess the robustness of our approach under realistic noise conditions. We simulate the circuit using a noise model extracted from the IBM Quantum fake backend `FakeTorino()`. Crucially, noise is applied only after parameter training has converged in a noiseless environment. The results below (shown in Figure 3.6 and 3.7) correspond to 3 iterations with an average post-selection success rate of approximately 12,37% for the encoded [4,2,2] circuit. It is important to note that we opted for a single simulation run for this specific type of experiment due to its high computational cost in terms of time, particularly when executed on a local system, as is the case in this study. Should one wish to replicate the experiment using a greater number of iterations, this remains feasible. We now proceed to present the results extracted from the respective noisy unencoded and noisy encoded simulations below.

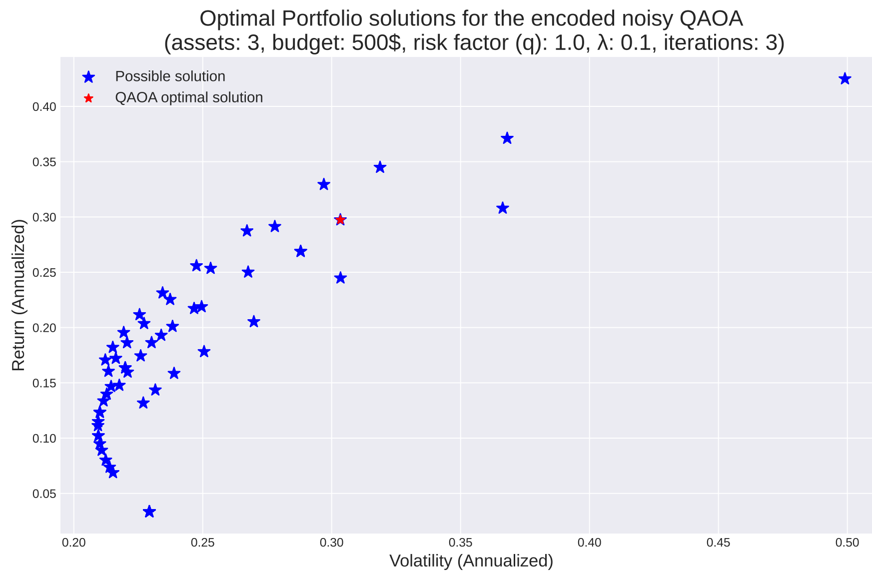


(a) Plot illustrating the graph of stars generated from the simulated noisy unencoded circuit, for the portfolio optimization problem. The specific optimization parameters are: 3 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$

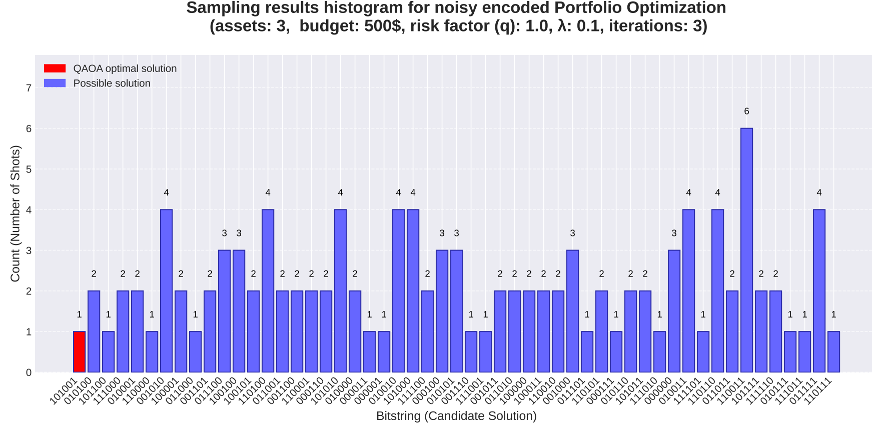


(b) Plot illustrating the histogram generated for from the simulated noisy unencoded circuit, for the portfolio optimization problem. The specific optimization parameters are: 3 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$

Figure 3.6: Results of the portfolio optimization for the 3 asset case, illustrating the performance of the `QAOAAnsatz()` combined with a custom post-selection filter across optimization depths equal to three ($p = 3$ layers) in a noisy simulated environment. The results are shown for 3 iterations.



(a) Plot illustrating the graph of stars generated from the simulated noisy encoded [4,2,2] circuit, for the portfolio optimization problem. The specific optimization parameters are: 3 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



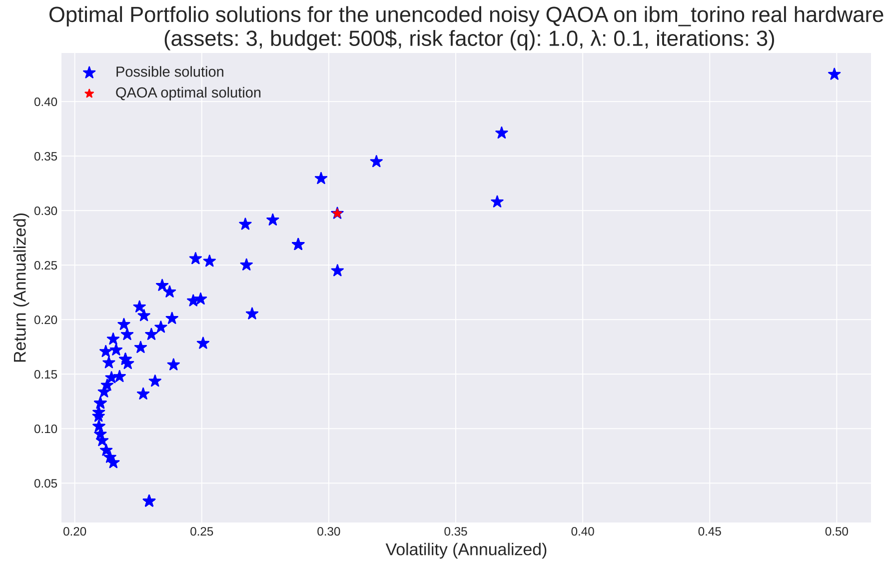
(b) Plot illustrating the histogram generated for from the simulated noisy encoded [4,2,2] circuit, for the portfolio optimization problem. The specific optimization parameters are: 3 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$

Figure 3.7: Results of the portfolio optimization for the 3 asset case, illustrating the performance of the encoded [4,2,2] Ansatz combined with a custom post-selection filter across optimization depths equal to three ($p = 3$ layers) in a noisy simulated environment. The results are shown for 3 iterations.

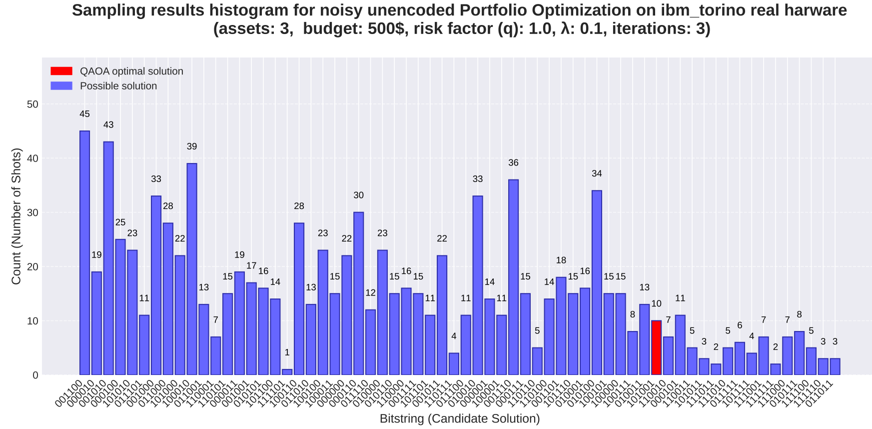
The results indicate that the "star plot" or "graph of stars" visualizations converge upon the same optimal solution (as can be seen in Figure 3.6a and 3.7a), yet they demonstrate a differing number of total sampled bit strings and statistical distribution of the latter (as shown in Figure 3.6b and 3.7b). This discrepancy in sampling frequency is likely attributable to the significant increase in the circuit depth required by the encoded system, which is considerably greater than that of the unencoded counterpart.

3.6 Hardware execution results

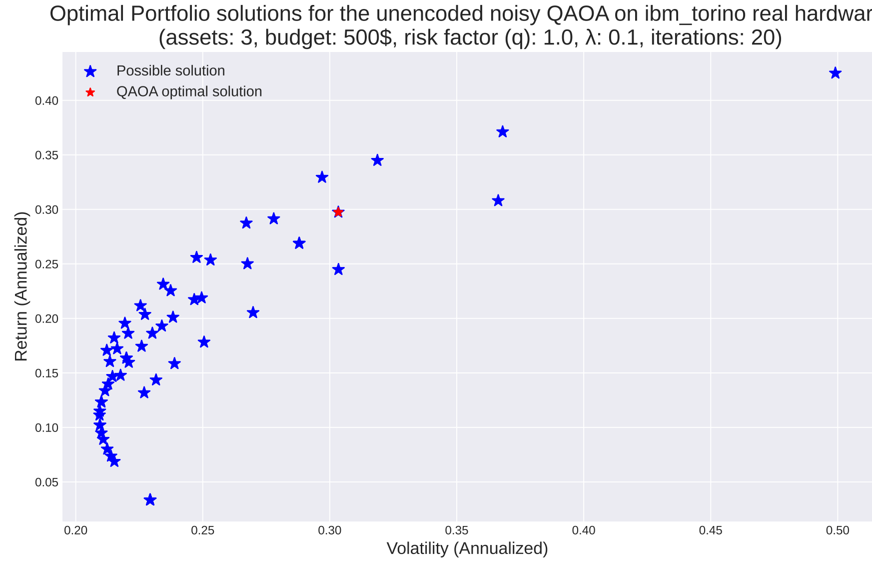
Finally, we execute the encoded circuit on real IBM Quantum hardware, we have choose "**ibm_torino**". This specific hardware was selected primarily because its topology allows for the successful encoding of our quantum circuit. Furthermore, it represented one of the four available options, constrained by the ten minutes of monthly access granted by our IBM Quantum account. All detailed technical specifications for this device were provided at the conclusion of the preceding Chapter 2.2.3. The subsequent sections present the results for the unencoded circuit executions (shown in Figure 3.8), followed by the results obtained from the encoded circuit (as can be seen in Figure 3.9), respectively, for different number of iterations.



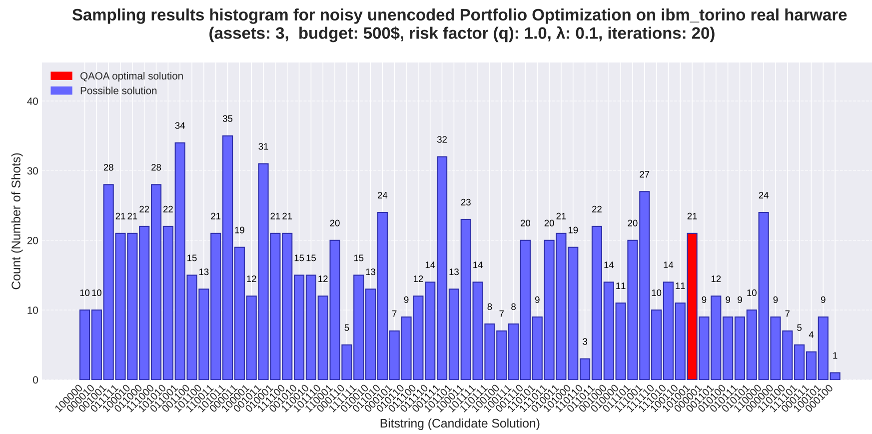
(a) Plot illustrating the graph of stars generated from the simulated noisy unencoded circuit, for the portfolio optimization problem on "**ibm_torino**" real hardware. The specific optimization parameters are: 3 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



(b) Plot illustrating the histogram generated for from the simulated noisy unencoded circuit, for the portfolio optimization problem on "**ibm_torino**" real hardware. The specific optimization parameters are: 3 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$

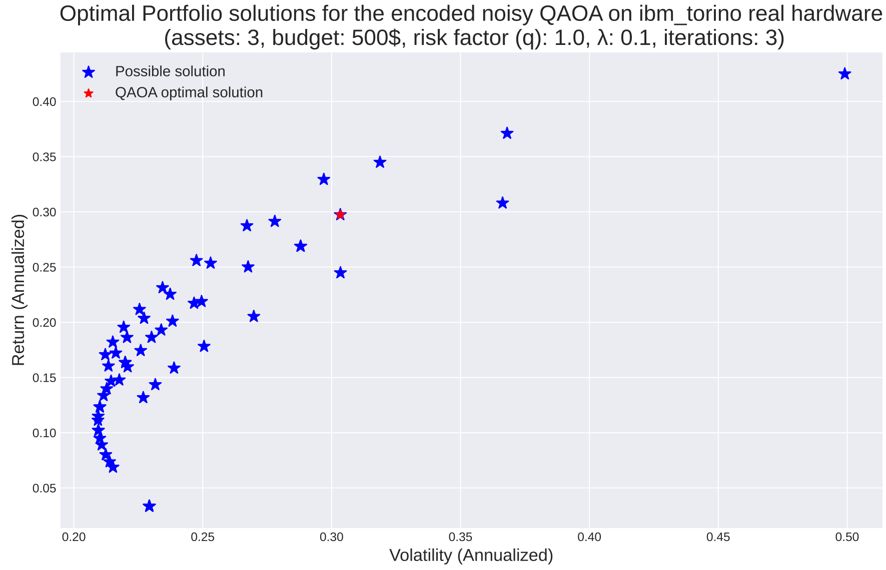


(c) Plot illustrating the graph of stars generated from the simulated noisy unencoded circuit on "**ibm_torino**" real hardware, for the portfolio optimization problem. The specific optimization parameters are: 20 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$

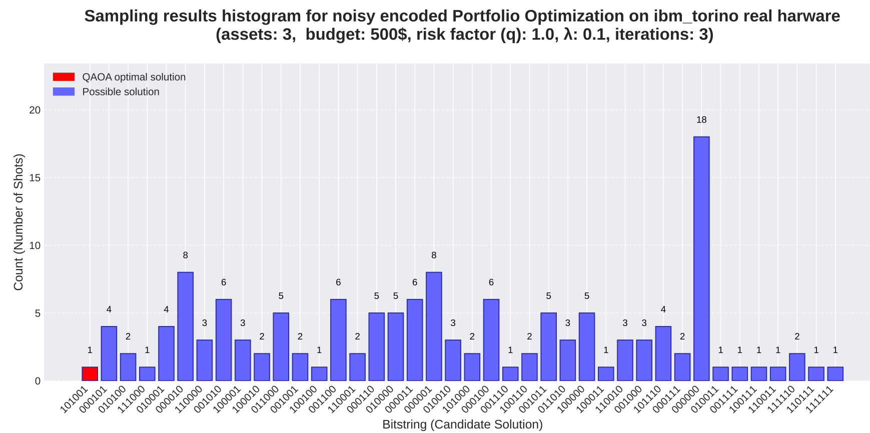


(d) Plot illustrating the histogram generated for from the simulated noisy unencoded circuit, for the portfolio optimization problem on "**ibm_torino**" real hardware. The specific optimization parameters are: 20 iterations, 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$

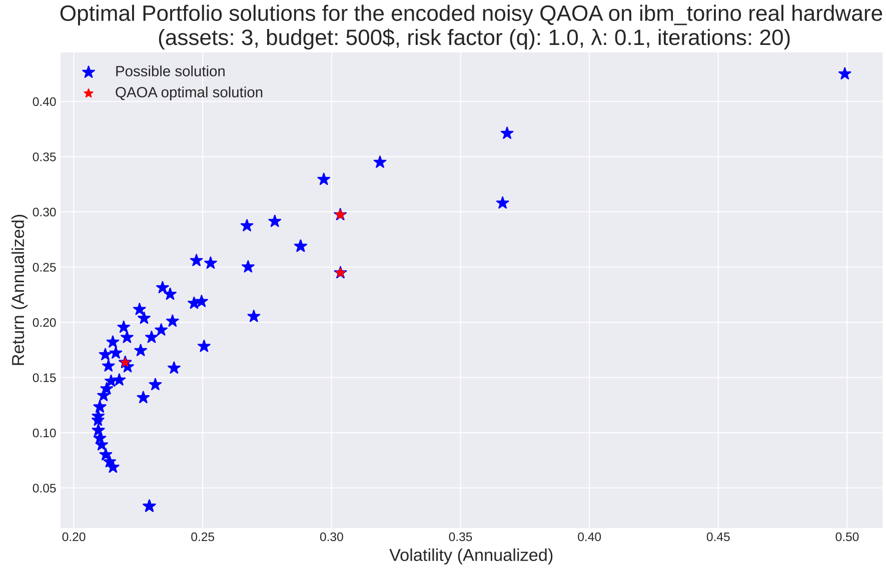
Figure 3.8: Results of the portfolio optimization for the 3 asset case, illustrating the performance of the `QAOAAnsatz()` combined with a custom post-selection filter across optimization depths equal to three ($p = 3$ layers) on "**ibm_torino**" real hardware. The results are shown for 3 and 20 iterations.



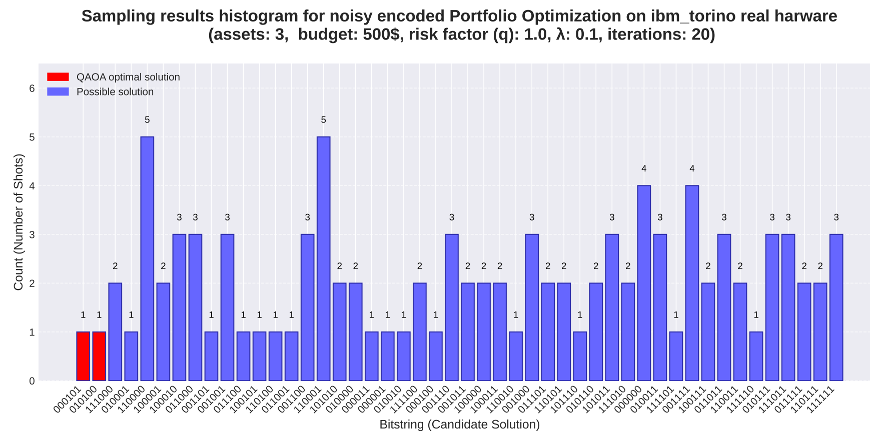
(a) Plot illustrating the graph of stars generated from the noisy encoded [4,2,2] circuit on "**ibm_torino**" real hardware, for the portfolio optimization problem. The specific optimization parameters are: 3 iterations, 3 QAOA layers, a budget constraint of Budget = 500, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



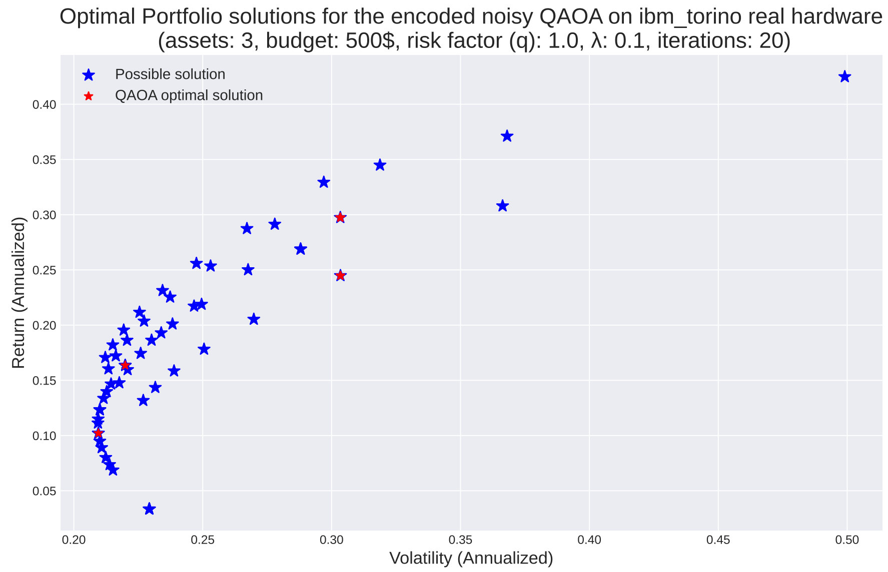
(b) Plot illustrating the histogram generated for from the noisy encoded [4,2,2] circuit on "**ibm_torino**" real hardware, for the portfolio optimization problem. The specific optimization parameters are: 3 iterations, 3 QAOA layers, a budget constraint of Budget = 500, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



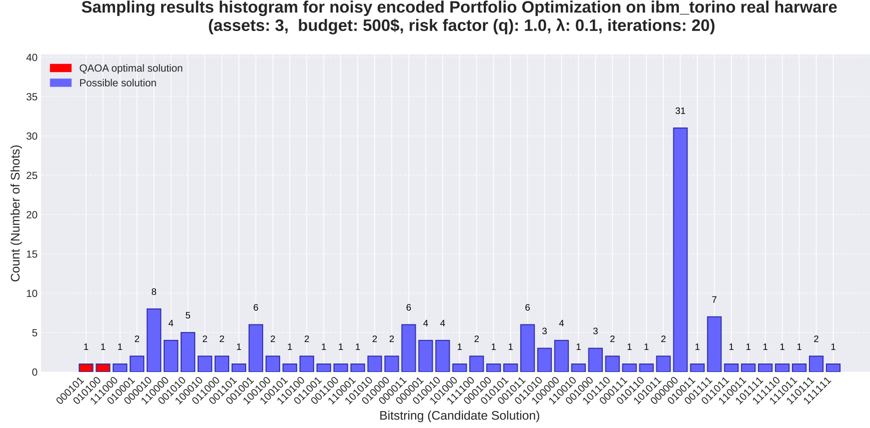
(c) Plot illustrating the graph of stars generated from the noisy encoded [4,2,2] circuit on "**ibm_torino**" real hardware, for the portfolio optimization problem. The specific optimization parameters are: 20 iterations (first execution), 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



(d) Plot illustrating the histogram generated for from the noisy encoded [4,2,2] circuit on "**ibm_torino**" real hardware, for the portfolio optimization problem. The specific optimization parameters are: 20 iterations (first execution), 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



(e) Plot illustrating the graph of stars generated from the noisy encoded [4,2,2] circuit on "**ibm_torino**" real hardware, for the portfolio optimization problem. The specific optimization parameters are: 20 iterations (second execution), 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and and $N_{\text{shots}} = 1000$



(f) Plot illustrating the histogram generated for from the noisy encoded [4,2,2] circuit on "**ibm_torino**" real hardware, for the portfolio optimization problem. The specific optimization parameters are: 20 iterations (second execution), 3 QAOA layers, a budget constraint of Budget $= 500$, a penalty coefficient of $\lambda = 0.1$, a risk-aversion parameter of $q = 1$ and $N_{\text{shots}} = 1000$

Figure 3.9: Results of the portfolio optimization for the 3 asset case, illustrating the performance of the encoded [4,2,2] Ansatz combined with a custom post-selection filter across optimization depths equal to three ($p = 3$ layers) on "**ibm_torino**" real hardware. The results are shown for 3 iterations.

In the results obtained from the noisy real hardware simulation for the encoded system with 20 iterations, the histogram displays only a limited number of highlighted columns relative to the total expected solutions observed in the star plot visualization (as depicted in Figure 3.9c, 3.9d, 3.9e and 3.9f). This observation is due to the fact that the histogram exclusively represents the results of the final iteration. Furthermore, as the simulation is under sampled for the previously established reasons, we do not sample all possible combinations within the code space. Consequently, although the star plot may indicate N solutions across the entire space generated by all iterations, the histogram will only highlight those solutions that are both part of the global solution space and coexist within the constrained subspace produced by the final iteration of our optimization cycle.

Even when executed on the real quantum hardware, the results obtained for the encoded and unencoded versions are observed to be approximately similar (as can be observed in Figure 3.8 and 3.9). The minor differences visible within the resulting plots are primarily attributable to the combination of the high circuit depth of the encoded system and a significantly low number of iterations, which prevents a thorough statistical analysis.

The limitation of the iteration count to a maximum of 20 was imposed by the

available execution time on the IBM Quantum platform under our current free access account. Nonetheless, nothing prevents the replication of this experiment with a significantly higher number of iterations, provided sufficient execution time is made available. Finally, to establish a comparative assessment between the noisy encoded simulation and the execution performed on the real hardware, we can extract the post-selection values from both scenarios. When evaluated over three iterations, consistent with the preceding analysis, this value is approximately 12.67%; conversely, the same metric for twenty iterations yields a result of approximately 12.57%.

Chapter 4

Conclusions and future works

4.1 Max cut problem: validation of encoding architecture and error detection process

The results obtained from the Max-Cut problem, particularly their perfect equivalence to those derived from the unencoded case with known solutions presented by NVIDIA, serve as compelling evidence that the entire encoding process was performed consistently and correctly. The same validation holds true for the subsequent decoding process.

This prior consideration indicates that the individual encoded gates were correctly designed, implemented, and subsequently applied to the circuit structure.

The absence of an aggressive post-selection logic, coupled with the effective selection of sampled strings afflicted by syndromes, is symptomatic of a well-structured algorithm capable of mapping and solving these types of problems, particularly concerning graphs of this specific dimension.

4.2 Comparative performance of encoded vs. unencoded Portfolio Optimization

The Portfolio Optimization problem constitutes our primary case study and is significantly more complex at a computational level compared to the Max-Cut problem.

4.2.1 Initial unencoded analysis and theoretical congruence

Initial results obtained from executing the unencoded circuit via the Qiskit `.solve()` method, achieved by varying parameters such as the budget, risk factor, and the penalty coefficient λ for the 4 asset problem, demonstrated theoretical congruence. The solutions consistently map along the Markowitz efficient frontier and within the cluster of generated portfolios (as can be seen in Section 3.2). Specifically, increasing the budget and the risk factor shifts the cluster of optimal star plots vertically towards the apex of the Markowitz plot. Conversely, varying the value of λ significantly influences the adherence of the found solutions to the efficient frontier, consequently reducing result dispersion.

4.2.2 Necessity of manual Ansatz and post-selection filter

To correctly implement our encoded Ansatz manually and derive results comparable to those generated by the Qiskit `.solve()` method, it was necessary to construct a custom test circuit featuring a dedicated Ansatz and a post-selection filter. This necessity arose because the base QAOA algorithm alone is insufficient to identify mathematically comparable solutions without the aid of a filter applied to the sampled strings. Specifically, the experimental results demonstrate and justify that the implementation of a post-selection filter is essential for our custom circuit. Without this component, the QAOA algorithm would select the bitstring with the highest count, which, however, does not correspond to the optimal solution. This discrepancy is primarily attributed to the inherent complexity of the Portfolio Optimization problem. In contrast, for the Max-Cut problem, the most frequent sample typically coincides with the optimal solution; consequently, a post-selection filter is not required. To establish a benchmark between the Ansatz and the filter, the only viable option was the utilization of the Qiskit `QAOAAnsatz()` function to generate a correctly implemented QAOA Ansatz, allowing us to focus subsequent efforts on validating the filter logic.

Filter validation through shot count increase

Our unencoded circuit, containing the Ansatz generated by `QAOAAnsatz()` and augmented with a custom-designed selection filter for the sampled results, was tested by increasing the number of shots. As the shot count increased to 10000 in our case, the selected results progressively adhered more closely to the efficient frontier curve, thereby converging toward the mathematically correct solution. The data extracted from these analyses (shown in Section 3.2) confirmed the correct implementation of our post-selection filter but offered no information regarding the intrinsic efficiency of the QAOA algorithm itself.

4.2.3 Algorithm efficiency validation through under-sampling

To specifically verify the intrinsic efficiency of the algorithm, we reduced the number of sampling shots to $N_{\text{shots}} = 1000$, placing the sampling well below the size of the total available code space ($2^{16} = 65536$). The results (depicted in Section 3.3) confirmed the functionality of our logic even under this constrained sampling regime, definitively validating the correctness of our post-selection methodology and positioning us to proceed with the implementation of the encoded Ansatz.

4.2.4 Implementation of the encoded Ansatz and dimensionality reduction

The initial attempt to implement the encoded Ansatz for the 4 asset system was hampered by the excessive circuit dimensions, which prevented successful synthesis by our compiler. Consequently, we reduced the problem size from four assets to three assets by imposing a lower budget constraint. This ensured the circuit dimensionality was suitable for successful synthesis. At this stage, we transitioned from representing the results on the Markowitz efficient frontier to utilizing a star plot visualization, better suited for systems of this dimension, complemented by a solution histogram.

4.2.5 Analysis of noiseless and noisy results

Noiseless case analysis

For the noiseless case, the plots (represented in Section 3.4) reveal that the star plot results are identical for low iteration counts. However, a slight deviation is observable in the case with 100 iterations. Regarding the solution distributions, the sampling of the correct solution is more pronounced in the unencoded system compared to the encoded system. These discrepancies are likely due to the significantly greater circuit depth and larger code space associated with our encoded system relative to its unencoded counterpart. Crucially, the correct solution is sampled by both systems, validating the core functionality of the encoded design.

Noisy and real hardware analysis

The same conclusions hold true for the noisy cases, encompassing both simulations based on an IBM noise model and executions performed on the real quantum hardware. In these scenarios, the inherent discrepancy in results is naturally amplified by the fact that our circuit operates within a noisy environment.

4.2.6 Post-selection metrics and final deduction

The histograms illustrating the distribution of sampled solutions reveal that our QAOA algorithm, particularly in the encoded case, frequently samples bitstrings that are inconsistent with the mathematically determined optima. This trend necessitates the design and implementation of a specific post-selection filter, as we done, to bridge the discrepancy between the analytical solutions and the statistically prevalent results. This discrepancy may be considered an inherent limitation of the present work, primarily arising from the technological constraints of currently available quantum hardware. However, substantial improvements are expected in future research as quantum processors continue to evolve and achieve higher performance standards.

From the post-selection metrics collected from the encoded circuits, the selection rates for syndrome free solutions in the noiseless systems (described in Section 3.4) were 100 %. In contrast, for the simulated noise (as can be observed in Section 3.5) and real hardware systems (shown in Section 3.6), the rates were approximately 12.66 % and 12.57 % for three and twenty iterations, respectively. These latter data strongly confirm the correct operation of our error detection system.

We can therefore deduce that our system has been correctly designed, validated, and implemented, rendering a non trivial optimization problem for QAOA, such as Portfolio Optimization, partially fault tolerant through our specifically designed and implemented error detection architecture.

4.3 Future works

Building upon the successful validation and implementation of the encoded quantum circuit, several avenues for future research emerge, primarily driven by the computational and hardware constraints encountered in the present study. We can outline potential research directions aimed at implementing increasingly general and scalable quantum systems over time.

Solution improvements and model fine-tuning

Given the observed discrepancies between the most probable bitstrings sampled by our algorithm and the mathematically optimal solutions, specifically regarding the encoded circuit, a primary objective for future research is to enhance the quality of these results. This can be achieved by refining the mathematical model, the training parameters and leveraging the continuous technological advancements in quantum computing hardware. Such efforts aim to bridge the existing gap between the statistically optimal solutions generated by our QAOA implementation and the analytical optima derived from the theoretical framework employed in this study.

Hardware improvements and fault-tolerance

Given the current limitations imposed by real quantum hardware, future work could focus on implementing systems with significantly higher machine coherence. By applying encoded algorithms, such as the one developed in this work, a robust level of noise intolerance could be achieved. Furthermore, the potential application of hardware-level error correction would render these systems nearly completely fault tolerant in practice.

Scalability and algorithmic efficiency assessment

Should the synthesis capacity allowed by Qiskit increase in the coming years, this experiment could be replicated on larger problem sizes. This would enable an effective evaluation of the efficiency of our quantum algorithm relative to its classical counterpart, allowing for a rigorous assessment of their respective computational efficiencies.

Short term optimization strategies

In the short term, recognizing the suboptimal efficiency of the standard QAOA in solving complex problems like Portfolio Optimization, there is scope to hybridize our algorithm with "warm-starting" techniques [52]. This approach could eliminate the need for the applied post-selection filter, thereby streamlining the computational process of result evaluation.

Another immediate application would involve utilizing a similar, but mathematically simpler, use case, such as the "Portfolio Selection" problem [53, 54, 55, 56, 57]. This would require a significantly smaller number of qubits, consequently allowing for the mapping of a greater number of assets within our system.

Alternative algorithmic approaches

A further immediate variant to streamline the post-selection process involves employing alternative variational algorithms, such as the Variational Quantum Eigensolver (VQE). VQE could be used to less rigidly mimic the Hamiltonian profile of our problem, thereby leading to a probabilistic convergence of solutions, similar to the approach successfully applied to the Max-Cut problem. Furthermore, the scope of this analysis could be extended to include non-variational frameworks [58, 59], most notably Quantum Annealing [60].

Bibliography

[1] Timothy H. Boyer. «Derivation of the Blackbody Radiation Spectrum without Quantum Assumptions». In: *Phys. Rev.* 182 (5 June 1969), pp. 1374–1383. DOI: 10.1103/PhysRev.182.1374. URL: https://link.aps.org/doi/10.1103/PhysRev.182.1374 (cit. on p. 1).

[2] Nigel P. Smart. «The Enigma Machine». In: *Cryptography Made Simple*. Cham: Springer International Publishing, 2016, pp. 133–161. ISBN: 978-3-319-21936-3. DOI: 10.1007/978-3-319-21936-3_8. URL: https://doi.org/10.1007/978-3-319-21936-3_8 (cit. on p. 2).

[3] Community Wiki (various contributors). *Finding the total number of 8-bit strings*. 2014. URL: https://math.stackexchange.com/questions/783120/finding-the-total-number-of-8-bit-strings (visited on 07/30/2025) (cit. on p. 3).

[4] Jianqi Lai, Hang Yu, Zhengyu Tian, and Hua Li. «Hybrid MPI and CUDA parallelization for CFD applications on multi-GPU HPC clusters». In: *Scientific programming* 2020.1 (2020), p. 8862123 (cit. on p. 2).

[5] Niels Bohr et al. *The quantum postulate and the recent development of atomic theory*. Vol. 3. Printed in Great Britain by R. & R. Clarke, Limited, 1928 (cit. on p. 3).

[6] Brigitte Falkenburg and Peter Mittelstaedt. «Probabilistic Interpretation of Quantum Mechanics». In: *Compendium of Quantum Physics*. Ed. by Daniel Greenberger, Klaus Hentschel, and Friedel Weinert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 485–491. ISBN: 978-3-540-70626-7. DOI: 10.1007/978-3-540-70626-7_148. URL: https://doi.org/10.1007/978-3-540-70626-7_148 (cit. on p. 3).

[7] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. 10th Anniversary Edition. Cambridge: Cambridge University Press, 2010. ISBN: 9781107002173 (cit. on p. 4).

[8] Poetry in Physics. *Illustration of a Qubit*. 2017. URL: https://poetryinphysics.wordpress.com/wp-content/uploads/2017/03/qubit.png (visited on 07/30/2025) (cit. on p. 4).

[9] S. Gayathri Devi, S. Manjula Gandhi, S. Chandia, and P. Boobalaragavan. «Exploring IBM Quantum Experience». In: *Quantum Computing: A Shift from Bits to Qubits*. Ed. by Rajiv Pandey, Nidhi Srivastava, Neeraj Kumar Singh, and Kanishka Tyagi. Singapore: Springer Nature Singapore, 2023, pp. 265–282. ISBN: 978-981-19-9530-9. DOI: 10.1007/978-981-19-9530-9_15. URL: https://doi.org/10.1007/978-981-19-9530-9_15 (cit. on p. 4).

[10] Travis S. Humble, Alexander McCaskey, Dmitry I. Lyakh, Meenambika Gowrishankar, Albert Frisch, and Thomas Monz. «Quantum Computers for High-Performance Computing». In: *IEEE Micro* 41.5 (2021), pp. 15–23. DOI: 10.1109/MM.2021.3099140 (cit. on p. 5).

[11] Nishant Saurabh, Shantenu Jha, and Andre Luckow. «A Conceptual Architecture for a Quantum-HPC Middleware». In: *2023 IEEE International Conference on Quantum Software (QSW)*. 2023, pp. 116–127. DOI: 10.1109/QSW59989.2023.00023 (cit. on p. 5).

[12] He-Liang Huang, Xiao-Yue Xu, Chu Guo, Guojing Tian, Shi-Jie Wei, Xiaoming Sun, Wan-Su Bao, and Gui-Lu Long. «Near-term quantum computing techniques: Variational quantum algorithms, error mitigation, circuit compilation, benchmarking and classical simulation». In: *Science China Physics, Mechanics & Astronomy* 66.5 (Apr. 2023). ISSN: 1869-1927. DOI: 10.1007/s11433-022-2057-y. URL: http://dx.doi.org/10.1007/s11433-022-2057-y (cit. on p. 5).

[13] Hsin-Yuan Huang, Kishor Bharti, and Patrick Rebentrost. *Near-term quantum algorithms for linear systems of equations*. 2019. arXiv: 1909.07344 [quant-ph]. URL: https://arxiv.org/abs/1909.07344 (cit. on p. 5).

[14] Y. Herasymenko and T.E. O'Brien. «A diagrammatic approach to variational quantum ansatz construction». In: *Quantum* 5 (Dec. 2021), p. 596. ISSN: 2521-327X. DOI: 10.22331/q-2021-12-02-596. URL: https://doi.org/10.22331/q-2021-12-02-596 (cit. on p. 5).

[15] Anbang Wu, Gushu Li, Yuke Wang, Boyuan Feng, Yufei Ding, and Yuan Xie. *Towards Efficient Ansatz Architecture for Variational Quantum Algorithms*. 2021. arXiv: 2111.13730 [quant-ph]. URL: https://arxiv.org/abs/2111.13730 (cit. on p. 5).

[16] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. *A Quantum Approximate Optimization Algorithm*. 2014. arXiv: 1411.4028 [quant-ph]. URL: https://arxiv.org/abs/1411.4028 (cit. on p. 6).

[17] Jaeho Choi and Joongheon Kim. «A Tutorial on Quantum Approximate Optimization Algorithm (QAOA): Fundamentals and Applications». In: *2019 International Conference on Information and Communication Technology Convergence (ICTC)*. 2019, pp. 138–142. DOI: 10.1109/ICTC46691.2019.8939749 (cit. on p. 6).

[18] Tongyang Li, Yuexin Su, Ziyi Yang, and Shengyu Zhang. «Quantum approximate optimization algorithms for maximum cut on low-girth graphs». In: *Phys. Rev. Res.* 7 (3 July 2025), p. 033014. DOI: 10.1103/jypq-v1fn. URL: https://link.aps.org/doi/10.1103/jypq-v1fn (cit. on p. 6).

[19] Brandon Augustino, Madelyn Cain, Edward Farhi, Swati Gupta, Sam Gutmann, Daniel Ranard, Eugene Tang, and Katherine Van Kirk. *Strategies for running the QAOA at hundreds of qubits*. 2024. arXiv: 2410.03015 [quant-ph]. URL: https://arxiv.org/abs/2410.03015 (cit. on p. 6).

[20] Sivaprasad Omanakuttan et al. *Threshold for Fault-tolerant Quantum Advantage with the Quantum Approximate Optimization Algorithm*. 2025. arXiv: 2504.01897 [quant-ph]. URL: https://arxiv.org/abs/2504.01897 (cit. on p. 6).

[21] Wikipedia contributors. *Quantum logic gate*. 2025. URL: https://en.wikipedia.org/wiki/Quantum_logic_gate (visited on 07/31/2025) (cit. on p. 8).

[22] IBM Quantum. *IBM Quantum Composer*. 2025. URL: https://quantum.cloud.ibm.com/composer (cit. on pp. 9, 11, 12).

[23] Kostas Blekos, Dean Brand, Andrea Ceschini, Chiao-Hui Chou, Rui-Hao Li, Komal Pandya, and Alessandro Summer. «A review on Quantum Approximate Optimization Algorithm and its variants». In: *Physics Reports* 1068 (June 2024), pp. 1–66. ISSN: 0370-1573. DOI: 10.1016/j.physrep.2024.03.002. URL: http://dx.doi.org/10.1016/j.physrep.2024.03.002 (cit. on pp. 9, 12).

[24] Luiz Benicio Degli Esposte Rosa. «AUTOMATIC CODE GENERATION FOR IIR EMBEDDED SYSTEMS IN FIXED-POINT ARITHMETIC». PhD thesis. Sept. 2017 (cit. on p. 10).

[25] Jeffrey Bub. «Von Neumann's Projection Postulate as a Probability Conditionalization Rule in Quantum Mechanics». In: *Journal of Philosophical Logic* 6.1 (1977), pp. 381–390. URL: http://www.jstor.org/stable/30227141 (cit. on p. 12).

[26] Subrata Das and Swaroop Ghosh. *TrojanNet: Detecting Trojans in Quantum Circuits using Machine Learning*. June 2023. DOI: 10.48550/arXiv.2306.16701 (cit. on p. 13).

[27] Maximilian Schlosshauer. «Quantum decoherence». In: *Physics Reports* 831 (2019). Quantum decoherence, pp. 1–57. ISSN: 0370-1573. DOI: https://doi.org/10.1016/j.physrep.2019.10.001. URL: https://www.sciencedirect.com/science/article/pii/S0370157319303084 (cit. on p. 14).

[28] P.W. Shor. «Fault-tolerant quantum computation». In: *Proceedings of 37th Conference on Foundations of Computer Science*. 1996, pp. 56–65. DOI: 10.1109/SFCS.1996.548464 (cit. on p. 15).

[29] Matt. J. Bedalov et al. *Fault-Tolerant Operation and Materials Science with Neutral Atom Logical Qubits*. 2024. arXiv: 2412.07670 [quant-ph]. URL: https://arxiv.org/abs/2412.07670 (cit. on pp. 15, 17, 20, 36, 47).

[30] Norbert M. Linke, Mauricio Gutierrez, Kevin A. Landsman, Caroline Figgatt, Shantanu Debnath, Kenneth R. Brown, and Christopher Monroe. «Fault-tolerant quantum error detection». In: *Science Advances* 3.10 (2017), e1701074. DOI: 10.1126/sciadv.1701074. eprint: https://www.science.org/doi/pdf/10.1126/sciadv.1701074. URL: https://www.science.org/doi/abs/10.1126/sciadv.1701074 (cit. on p. 16).

[31] Kento Tsubouchi, Yasunari Suzuki, Yuuki Tokunaga, Nobuyuki Yoshioka, and Suguru Endo. «Virtual quantum error detection». In: *Phys. Rev. A* 108 (4 Oct. 2023), p. 042426. DOI: 10.1103/PhysRevA.108.042426. URL: https://link.aps.org/doi/10.1103/PhysRevA.108.042426 (cit. on p. 16).

[32] Chris N Self, Marcello Benedetti, and David Amaro. «Protecting expressive circuits with a quantum error detection code». In: *Nature Physics* 20.2 (2024), pp. 219–224 (cit. on p. 16).

[33] Christian Kraglund Andersen, Ants Remm, Stefania Lazar, Sebastian Krinner, Nathan Lacroix, Graham J Norris, Mihai Gabureac, Christopher Eichler, and Andreas Wallraff. «Repeated quantum error detection in a surface code». In: *Nature Physics* 16.8 (2020), pp. 875–880 (cit. on p. 16).

[34] Antonio D Córcoles, Easwar Magesan, Srikanth J Srinivasan, Andrew W Cross, Matthias Steffen, Jay M Gambetta, and Jerry M Chow. «Demonstration of a quantum error detection code using a square lattice of four superconducting qubits». In: *Nature communications* 6.1 (2015), p. 6979 (cit. on p. 16).

[35] Peter W. Shor. «Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer». In: *SIAM Journal on Computing* 26.5 (Oct. 1997), pp. 1484–1509. ISSN: 1095-7111. DOI: 10.1137/s0097539795293172. URL: http://dx.doi.org/10.1137/S0097539795293172 (cit. on p. 16).

[36] Peter W. Shor. *Fault-tolerant quantum computation*. 1997. arXiv: quant-ph/9605011 [quant-ph]. URL: https://arxiv.org/abs/quant-ph/9605011 (cit. on p. 16).

[37] Daniel Gottesman. *Surviving as a Quantum Computer in a Classical World*. Cambridge University Press, 2024 (cit. on pp. 16, 19).

[38] Zichang He, David Amaro, Ruslan Shaydulin, and Marco Pistoia. «Performance of quantum approximate optimization with quantum error detection». In: *Communications Physics* 8.1 (May 2025). ISSN: 2399-3650. DOI: 10.1038/s42005-025-02136-8. URL: http://dx.doi.org/10.1038/s42005-025-02136-8 (cit. on p. 18).

[39] Nikolas P Breuckmann and Jens Niklas Eberhardt. «Quantum low-density parity-check codes». In: *PRX quantum* 2.4 (2021), p. 040101 (cit. on p. 19).

[40] Joschka Roffe, David R White, Simon Burton, and Earl Campbell. «Decoding across the quantum low-density parity-check code landscape». In: *Physical Review Research* 2.4 (2020), p. 043423 (cit. on p. 19).

[41] Hans L. Bodlaender and Klaus Jansen. «On the complexity of the maximum cut problem». In: *Nordic J. of Computing* 7.1 (Mar. 2000), pp. 14–31. ISSN: 1236-6064 (cit. on p. 19).

[42] Hans L Bodlaender and Klaus Jansen. «On the complexity of the maximum cut problem». In: *Nordic Journal of Computing* 7.1 (2000), pp. 14–31 (cit. on p. 19).

[43] Michael J Best. *Portfolio optimization*. CRC Press, 2010 (cit. on p. 19).

[44] Abhishek Gunjan and Siddhartha Bhattacharyya. «A brief review of portfolio optimization techniques». In: *Artificial Intelligence Review* 56.5 (2023), pp. 3847–3886 (cit. on p. 19).

[45] NVIDIA. *QAOA Example*. https://nvidia.github.io/cuda-quantum/0.8.0/using/examples/qaoa.html. 2024 (cit. on pp. 21, 28).

[46] Giuseppe Buonaiuto, Francesco Gargiulo, Giuseppe De Pietro, Massimo Esposito, and Marco Pota. «Best practices for portfolio optimization by quantum computing, experimented on real quantum devices». In: *Scientific Reports* 13.1 (Nov. 2023), p. 19434. DOI: 10.1038/s41598-023-45392-w. URL: https://doi.org/10.1038/s41598-023-45392-w (cit. on pp. 22, 23).

[47] Wikipedia. *Efficient frontier*. https://en.wikipedia.org/wiki/Efficient_frontier (cit. on p. 22).

[48] Andrew Lucas. «Ising formulations of many NP problems». In: *Frontiers in Physics* 2 (2014). ISSN: 2296-424X. DOI: 10.3389/fphy.2014.00005. URL: http://dx.doi.org/10.3389/fphy.2014.00005 (cit. on p. 25).

[49] IBM. *Qiskit Guides*. `https://quantum.cloud.ibm.com/docs/en/guides` (cit. on p. 26).

[50] NVIDIA. *CUDA Quantum*. `https://nvidia.github.io/cuda-quantum/latest/index.html` (cit. on pp. 26, 60).

[51] Matteo Child et al. *logical qaoa implementation*. URL: `To%20be%20disclosed` (cit. on pp. 35, 36, 47).

[52] Daniel J Egger, Jakub Mareček, and Stefan Woerner. «Warm-starting quantum optimization». In: *Quantum* 5 (2021), p. 479 (cit. on p. 99).

[53] Bin Li and Steven CH Hoi. «Online portfolio selection: A survey». In: *ACM Computing Surveys (CSUR)* 46.3 (2014), pp. 1–36 (cit. on p. 99).

[54] Frank J Fabozzi, Harry M Markowitz, and Francis Gupta. «Portfolio selection». In: *Handbook of finance* 2 (2008), pp. 3–13 (cit. on p. 99).

[55] Norm Archer and Fereidoun Ghasemzadeh. «Project Portfolio Selection». In: *The Wiley Guide to Project, Program, and Portfolio Management* (2007), p. 94 (cit. on p. 99).

[56] Bernard J Kornfeld and Sami Kara. «Project portfolio selection in continuous improvement». In: *International Journal of Operations & Production Management* 31.10 (2011), pp. 1071–1088 (cit. on p. 99).

[57] Norm P Archer and Fereidoun Ghasemzadeh. «An integrated framework for project portfolio selection». In: *International Journal of Project Management* 17.4 (1999), pp. 207–216 (cit. on p. 99).

[58] Patrick Rebentrost and Seth Lloyd. «Quantum computational finance: quantum algorithm for portfolio optimization». In: *KI-Künstliche Intelligenz* 38.4 (2024), pp. 327–338 (cit. on p. 99).

[59] Abha Satyavan Naik, Esra Yeniaras, Gerhard Hellstern, Grishma Prasad, and Sanjay Kumar Lalta Prasad Vishwakarma. «From portfolio optimization to quantum blockchain and security: A systematic review of quantum computing in finance». In: *Financial Innovation* 11.1 (2025), pp. 1–67 (cit. on p. 99).

[60] Davide Venturelli and Alexei Kondratyev. «Reverse quantum annealing approach to portfolio optimization problems». In: *Quantum Machine Intelligence* 1.1 (2019), pp. 17–30 (cit. on p. 99).