Qubits In Practice: An Educational MATLAB Approach to Quantum Algorithms

Matteo Rossi
Department of Electronics, Information and Bioengineering
Politecnico di Milano
Milan, Italy
matteo26.rossi@mail.polimi.it

***Abstract*—This paper presents three interactive educational resources developed in MATLAB for introducing key quantum computing algorithms and concepts: Shor's algorithm, the Quantum Approximate Optimization Algorithm (QAOA), and quantum repetition codes for error correction. Each resource is implemented as a self-contained Live Script, combining theoretical exposition with executable code, visualization tools, and modular components for hands-on exploration. Shor's algorithm illustrates period finding and modular exponentiation in quantum factoring; QAOA is presented as a customizable variational method for solving QUBO problems, considering Multi-Knapsack and MaxCut; and repetition codes demonstrate bit flip error correction via syndrome extraction. The resources are designed to support learning by exposing trade-offs and quantum-classical workflows. They provide learners with opportunities to modify components, analyze performance, and build critical intuition about the strengths and limitations of quantum techniques. These materials aim to link abstract quantum theory and practical algorithm engineering, supporting both academic coursework and independent exploration of quantum programming.**

***Index Terms*—Shor's Algorithm, QAOA, Multi-Knapsack Problem, Repetition Codes, MATLAB, Quantum Computing**

I. Introduction

Quantum computing is an evolving computational paradigm that relies on quantum mechanical properties to process information in fundamentally different ways than classical computation. Unlike traditional bits that exist in definite states of 0 or 1, *qubits* can exist in superposition of both states simultaneously. Combined with *entanglement*, these phenomena enable quantum algorithms to achieve exponential speedup over classical computation [1].

MATLAB is a programming platform originally designed for numerical computing used by engineers and scientists. It provides an integrated development environment with visualization capabilities, extensive toolboxes for specialized applications, and Live Scripts that combine exectuable code with text and interactive elements.

Despite growing interest in quantum computing education, current resources suffer from a disconnection between theoretical understanding and practical implementation. Most educational materials either focus exclusively on mathematical formulations without programming references. Additionally, existing useful resources related to a subject are scattered around different places, and rarely offer interactive environments where students can experiment with parameters and observe their effects in real-time. The lack of modular, well-documented implementations further limits educators' ability to adapt materials for different learning contexts. Furthermore, existing MATLAB-based quantum computing resources are predominantly practical code examples without theoretical context, leaving students to reason on underlying theory on their own. The MATLAB Support Package for Quantum Computing documentation solely describes the API, creating a need for educational materials that combine the package's capabilities with theoretical aspects.

To overcome these limitations, this paper presents three MATLAB-based educational resources for quantum computing: Shor's algorithm, the Quantum Approximate Optimization Algorithm (QAOA), and quantum error correction using repetition codes. Each module combines theoretical foundations with hands-on implementation through interactive Live Scripts, enabling exploration of quantum circuits, simulations, and algorithmic trade-offs, while its modularity allows easy adaptation to different learning contexts. By making the code openly available, this work also helps lower the barriers to entry for those looking to explore quantum computing. All code related to this work is publicly available online. [1] [2] [3]

II. Background

Quantum computing operates on qubits, the fundamental unit of quantum information, mathematically represented as a superposition of basis states $|0\rangle$ and $|1\rangle$: $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. This superposition enables qubits to encode exponentially more information than classical bits.

Quantum gates manipulate qubits through unitary transformations. Single-qubit gates like Pauli-X, Pauli-Z, and Hadamard operate on individual qubits, while multi-qubit gates such as CNOT create entanglement between qubits, establishing correlations impossible in classical systems. These gates form a universal set, meaning any quantum computation can be composed by sequences of these elementary operations.

The quantum circuit model represents quantum algorithms as directed graphs where qubits flow through sequences of quantum gates. Circuits begin with state preparation, then ap-

[1] **Shor's Algorithm** - https://github.com/necst/qc-with-matlab/blob/main/Projects/Shor_Algorithm.mlx
[2] **QAOA** - https://github.com/necst/qc-with-matlab/tree/main/Projects/QAOA
[3] **Repetition Codes** - https://github.com/necst/qc-with-matlab/blob/main/Projects/QEC_Repetition_Codes.mlx

ply gates in temporal order, and conclude with measurements that collapse quantum states to classical outcomes.

The MATLAB Support Package for Quantum Computing provides tools for quantum algorithm development and simulation. Its API provides circuit creation functions, gate application methods and simulation capabilities, representing quantum states as complex vectors. Circuit visualization generates interactive diagrams showing gate sequences. Custom gate definitions enable implementation of specialized operations beyond the standard gate library. The package is fully integrated with all other MATLAB's packages, making computation and analysis accessible for users familiar with MATLAB's programming paradigm.

III. Tutorial on Shor's Algorithm

Shor's Algorithm [2] provides a solution to the problem of integer factorization. While its implementation on real hardware remains limited by current quantum resources, its pedagogical value is unmatched for introducing quantum modular arithmetic and circuit-based period finding. The tutorial aims to illustrate its theoretical foundations, analyzing the issues and technicalities that arise when creating its building blocks, even in a predetermined small-scale example. The script simulates modular exponentiation using pre-built circuit components given a fixed base ($a = 7$), and guides through the construction of the circuit for factoring a small number which is a product of two primes ($N = 15$).

A. Theoretical Overview and Problem Statement

In 1994, Peter Shor introduced an algorithm that provides a polynomial-time solution to the integer factorization problem on a quantum computer, offering an exponential speedup over the best known classical algorithms. The quantum advantage relies in the reduction of the problem of factoring to a mathematical subproblem—period finding—which quantum computers can solve efficiently using the Quantum Fourier Transform (QFT) and Quantum Phase Estimation (QPE).

Given a composite number N, Shor's algorithm chooses a random integer $a < N$ and analyzes the period behavior of the function $f(x) = a^x \ mod \ N$. Discovering the order r of this function–the smallest positive integer such that $a^r = 1 \ mod \ N$–is the key step. Once r is known, classical-post processing techniques can extract the nontrivial factors of N with high probability.

B. Quantum Subroutines: Phase Estimation and QFT

a) Modular Exponentiation as a Quantum Oracle: The algorithm begins by preparing the input state in superposition and applies a unitary operator encoding the modular exponentiation function $f(x) = a^x \ mod \ N$. This function maps:

$$|x\rangle \otimes |0\rangle \mapsto |x\rangle \otimes |a^x \bmod N\rangle \tag{1}$$

The resource includes a compiled version of this unitary, based on the chosen numbers and inspired by [3], which is suitable for our purpose.

b) Quantum Phase Estimation (QPE): : this procedure is used to extract the period r of the modular exponentiation operator. It works by estimating the eigenphase ϕ corresponding to an eigenvector $|\psi\rangle$ of a unitary operator U, such that:

$$U|\psi\rangle = e^{2\pi i\phi}|\psi\rangle \tag{2}$$

In this context, the operator U is the modular multiplication, and its eigenstates are periodic functions. QPE uses the Quantum Fourier Transform (QFT) to convert periodicity in amplitude into measurable phase information. With high probability, this procedure yields a rational approximation of $\phi = s/r$, from which the order r can be recovered classically via continued fractions.

C. MATLAB Live Script Implementation

a) Compiled Modular Exponentiation Circuit: Generic modular exponentiation requires extensive gate sequences unsuitable for simulation. The resource presents a compiled circuit for the case of $7^x \mod 15$, implemented via the function `controlled_compiled_amod15(pow, mapping)`. It uses a fixed pattern of controlled SWAP and X gates applied across qubits to perform the operation. This reduces the gate count from hundreds to dozens while preserving the algorithm's structure.

b) Inverse QFT with Explicit Endianness Handling: The $\text{QFT}^\dagger$ implementation explicitly manages qubit ordering through SWAP gate triplets rather than relying on conventions. This makes bit-ordering transparent and avoids measurement interpretation errors, common in these scenarios.

c) Classical Period Extraction: The continued fraction implementation uses MATLAB's `rat()` function with fixed tolerance to extract period candidates from normalized phase measurements. The algorithm processes measurement results from the upper register only, implementing the principle of implicit measurement for the computational register.

d) Circuit Integration and Simulation: The complete circuit combines 8 qubits (4 for phase estimation, 4 for modular arithmetic) with 2048 measurement samples. The simulation outputs probability distributions that demonstrate quantum interference patterns, with successful period detection occurring at states corresponding to multiples of $\frac{r}{2^n}$.

D. Learning Outcomes

By interacting with this resource, learners are expected to:

- Understand Shor's algorithm structure and its quantum-classical workflow for integer factorization.
- Trace the interaction between quantum subroutines including modular exponentiation and phase estimation.
- Analyze the role of quantum interference, entanglement, and measurement in achieving computational speedup.
- Execute all algorithm steps, interpreting intermediate results and connecting theory to observable outcomes.
- Evaluate resource requirements and performance trade-offs compared to classical factorization methods.
- Adapt the implementation for other small numbers N.

IV. QAOA: An Extendable MATLAB Class for Customizable Hybrid Optimization

The Quantum Approximate Optimization Algorithm (QAOA) [4], [5] represented a novel approach to combinatorial optimization, leveraging the synergy between quantum circuits and classical post-processing. Introduced by Farhi, Goldstone, and Gutmann in 2014 [6], it targets problems where classical methods face exponential complexity, such as resource allocation, logistics, and graph partitioning. Its pedagogical value lies in demystifying variational quantum algorithms, a cornerstone of near-term quantum computing, while offering hands-on experience in hybrid algorithm design, cost function modeling, and parameter tuning. This resource merges theory and practice through a modular MATLAB class architecture. Unlike black-box implementations, it exposes the inner workings of QAOA, empowering users to customize cost Hamiltonians for QUBO-encoded problems, benchmark results against different solvers, experiment with different mixer Hamiltonians and initialization strategies, and visualize the iterative quantum-classical feedback loop.

A. Theoretical Overview and Problem Statement

QAOA is designed to solve combinatorial optimization problems that can be represented through the Quadratic Unconstrained Binary Optimization (QUBO) [7] form:

$$\min_{x \in 0,1^n} x^T H x \tag{3}$$

where H encodes the problem's cost landscape.

Within this framework, two benchmark problems are considered:

- **0/1 Multi-Knapsack Problem:** selecting a subset of items to maximize total value without violating knapsack capacity constraints.
- **MaxCut:** partitioning the nodes of a graph to maximize the weight of the edges crossing the cut.

The QAOA algorithm proceeds by alternating between two parameterized unitaries: the problem unitary, derived from the cost Hamiltonian, and a mixing unitary. A classical optimizer adjusts the parameters $(\vec{\gamma}, \vec{\beta})$ of the quantum circuit to minimize the expected energy of the quantum state. This hybrid loop iteratively refines the parameters, combining quantum state preparation with classical numerical optimization.

B. MATLAB Live Script Implementation

1) Class Architecture and Design Principles: The QAOA resource employs a modular MATLAB class architecture where the core `QAOA.m` class encapsulates the quantum-classical hybrid loop while exposing key design choices including cost Hamiltonian, mixer construction, parameter initialization, and classical optimization strategy.

This architecture reflects three main principles:

- **Modularity**: Each QAOA's component—cost function, mixer layer, parameter vector, and optimizer—is implemented as an independent subroutine with clear API to facilitate extension, debugging and reusability.
- **Transparency**: All operations are explicitly defined and commented, avoiding black-box functions. Learners can trace the entire optimization process and understand each component's contribution.
- **Comparability**: Results can be directly benchmarked against classical solvers (MILP, Tabu Search) and MATLAB's native QAOA, reinforcing understanding by contrasting different optimization paradigms applied to the same QUBO problem.

Through this object-oriented design, students interact with the underlying algorithmic components, adjust hyperparameters, alter cost function penalties, and observe how these changes affect convergence and solution quality. In this way, the class is a toolbox which can be used through an hands-on approach.

2) Core Properties and Workflow: The QAOA class exposes a set of core properties that define the behavior of the quantum-classical optimization loop.

- `H`: Cost Hamiltonian matrix of the objective function $x^T H x$ representing the QUBO-encoded problem.
- `n_layers`: Number p of QAOA layers. Each layer consists of one application of both the cost and mixer unitaries. Increasing this parameter increases circuit depth and generally improves solution accuracy at the cost of runtime and optimization complexity.
- `init_type`: Parameter initialization strategy for $(\vec{\gamma}, \vec{\beta})$. The resource includes random initialization and adiabatic-inspired ramp initialization.
- `mixer`: Mixer Hamiltonian identifier. Supported options include standard transverse-field X and a more entangling XY mixer, each impacting the expressiveness of the quantum state.
- `optimizer`: Classical optimization routine handle `fminsearch`, `fmincon`, `surrogateopt`, to be selected based on the problem and the smoothness/noisiness of the landscape.
- `direction`: Objective direction (minimize/maximize) of the cost function. This allows the class to remain agnostic to how the QUBO is formulated.

The algorithm proceeds through the following stages:

- **QUBO Initialization**: The user defines a QUBO-formulated problem, which is passed to the class via the cost Hamiltonian H. The script provides examples that construct H from either a graph (for MaxCut) or a constraint matrix (for Multi-Knapsack).
- **Parameter Initialization**: Based on the specified `init_type`, the class generates an initial set of variational parameters $(\vec{\gamma}, \vec{\beta})$. This enables learners to observe how initialization affects convergence.
- **Circuit Construction**: The class builds the QAOA circuit alternating between the cost and mixer unitaries. The circuit can be fully visualized, with also the possibility of inspecting composite gates.
- **Expectation Evaluation**: The circuit is simulated using matrix multiplication, and the resulting quantum state

is measured virtually. The expectation value of the cost Hamiltonian with respect to this state is computed and returned to the classical optimizer.
- **Optimization**: The classical optimizer iteratively updates the parameters to optimize the expected cost. Each iteration triggers a new circuit construction and evaluation.
- **Sampling and Interpretation**: Once convergence is reached, the final quantum state is sampled to produce a bitstring solution, which is interpreted as a candidate solution to the original optimization problem and can be compared to known optima or classical solver outputs.

3) Solving QUBO Problems: Multi-Knapsack and MaxCut: The **Multi-Knapsack** problem involves allocating items, each characterized by a weight and a profit that may differ across knapsacks with distinct capacities. The aim is to maximize total profit by assigning each item to at most one knapsack without exceeding capacities. [8], [9].

A dedicated `KnapsackSolver` class automates QUBO translation through:

- **Formulation**: Converts item–knapsack assignments and constraints into binary variables, adding slack variables for linear encoding of capacity violations.
- **QUBO Construction**: Assembles the full cost Hamiltonian with: the objective value, single-assignment penalties (each item in at most 1 knapsack), and capacity penalties.
- **Solver Integration**: Supports multiple solving strategies including MILP (Exact), Tabu Search (Heuristic), Native QAOA (MATLAB), Custom QAOA (this resource).
- **Interpretability**: Offers methods to display, verify, and interpret bitstring solutions into readable item–knapsack assignments.

A test scenario (4 items, 2 knapsacks) was solved using all four approaches. While MILP provided the exact solution, both heuristic (Tabu) and hybrid (QAOA) approaches recovered the optimum after multiple runs. The custom QAOA implementation, in particular, highlights the trade-off between shot count, optimizer choice, and convergence robustness in variational methods. This hands-on comparison reveals the strengths and limitations of each paradigm.

TABLE I
SOLVER RESULTS ON MULTI-KNAPSACK PROBLEM (SCENARIO: 4 ITEMS, 2 KNAPSACKS)

Solver	Feasible	Optimal
MILP (Exact)	Yes	Yes
Tabu Search	Yes	Yes (After Retries)
Native QAOA (MATLAB)	Yes	Yes (Probabilistic)
Custom QAOA (This work)	Yes	Yes (After Parameters Tuning)

MaxCut serves as a minimal yet meaningful benchmark problem for QAOA. The script includes small 2-node and 4-node graph instances, illustrating:

- The translation from adjacency matrix to QUBO.
- Construction of the cost Hamiltonian from edge weights.
- Visualization of solution bitstrings and cut values.

Focusing on the 4-node graph problem instance, after running the circuit and simulating 1000 quantum measurements, the resulting bitstring distribution is shown in Figure 1. Two quantum states—`0110` and `1001`—emerge, each corresponding to a valid solution achieving the optimal cut value.

Bitstrings are interpreted as vertex assignments: each bit indicates the partition of the corresponding node. In this example, the bitstring `0110` corresponds to partitions {{2},{3}} and {{1}, {4}}. The resulting cut crosses the maximum number of edges, confirming its optimality (Figure 2).

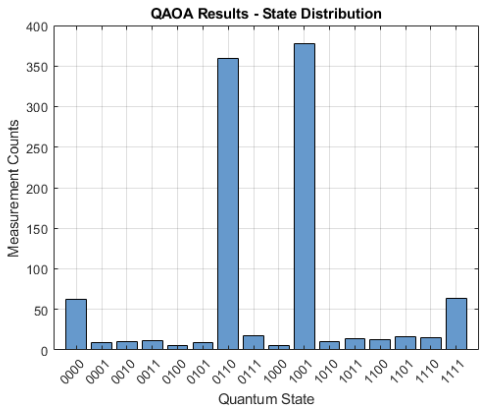

Fig. 1. QAOA measurement results for 4-node MaxCut. Bitstrings `0110` and `1001` dominate the distribution, indicating optimal cuts.

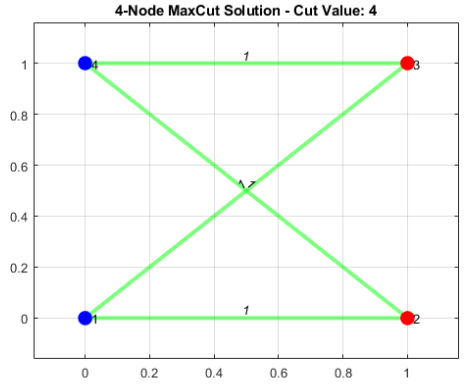

Fig. 2. Graph visualization of one optimal solution (bitstring `0110`). Cut edges are highlighted.

C. Learning Outcomes

By interacting with this resource, learners are expected to:

- Understand the structure and principles of QAOA.
- Convert combinatorial optimization problems into QUBO form suitable for QVA.
- Configure and run customizable QAOA simulations, adjusting mixer Hamiltonians, initialization, and optimizers.
- Analyze output distribution of bitstring measurements and interpret them in the original problem, verifying solution feasibility and quality.
- Compare QAOA performance with classical solvers, understanding trade-offs between exact, heuristic, and hybrid optimization techniques.

V. Tutorial on Repetition Codes in QEC

Quantum computing offers potential speedups for specific problems but faces a major obstacle: the fragility of quantum states. Qubits are vulnerable to decoherence, noise, and errors, which can corrupt information during computation [10].

Quantum Error Correction (QEC) mitigates this by encoding logical qubits redundantly across multiple physical qubits. Unlike classical repetition, quantum information cannot be cloned due to the No-Cloning Theorem, requiring more nuanced methods for error detection and correction. This resource explores one of the simplest QEC methods: the quantum repetition code. Inspired by its classical version, it encodes a logical qubit into multiple entangled physical qubits to detect and correct bit-flip errors via syndrome measurements, exposing error information without collapsing the quantum state. This tutorial introduces the core principles and practical implementation of quantum repetition codes for error correction. It makes learners gain insight into a full QEC workflow by encoding arbitrary quantum circuits with 3 or higher-distance repetition code to protect aganst bit flip errors. Inject, detect and correct bit-flip errors using ancilla-based syndrome measurements [11]. Quantify the resource overhead (qubits, gates) introduced by error correction from and hardware and cost perspective and evaluate correction effectiveness via fidelity and logical error rate simulations [12].

A. Theoretical Overview and Problem Statement

In classical computing, repetition codes provide a simple yet effective method for detecting and correcting errors by duplicating bits. A bit encoded as 0 might become 000, allowing recovery through majority voting if at most one bit is flipped. However, quantum computing introduces subtleties that prevent direct analogies—most notably, the No-Cloning Theorem, which forbids copying arbitrary quantum states. This makes the design of quantum error correction (QEC) fundamentally different and more complex.

To mitigate bit flip errors, the quantum repetition code encodes a single logical qubit into an entangled state of three physical qubits:

$$|\psi\rangle_L = \alpha\,|000\rangle + \beta\,|111\rangle \tag{4}$$

This encoded state resides in a two-dimensional codespace, embedded within the larger Hilbert space of three qubits. Errors displace the state outside this codespace, and QEC aims to detect and correct such displacements while preserving the encoded quantum information.

A direct measurement to detect an error would collapse the state, destroying its superposition. Instead, syndrome measurement is used: ancillary qubits measure the parity between adjacent physical qubits, yielding information about the error's location. For the 3-qubit code, two independent parity observables Z_1Z_2 and Z_2Z_3 are sufficient. The outcomes of these measurements form a 2-bit syndrome, which uniquely identifies whether an error occurred and on which qubit [13].

This approach works even in superposition because bit flip errors commute with parity observables. Thus, we extract error information without collapsing logical information.

TABLE II
Bit Flip Error Correction Process. Adapted from [14].

State	Syndrome	Correction	Explanation
$\alpha\,\|000\rangle + \beta\,\|111\rangle$	00	$I \otimes I \otimes I$	No Error
$\alpha\,\|100\rangle + \beta\,\|011\rangle$	10	$X \otimes I \otimes I$	Correcting Error on q_1
$\alpha\,\|010\rangle + \beta\,\|101\rangle$	11	$I \otimes X \otimes I$	Correcting Error on q_2
$\alpha\,\|001\rangle + \beta\,\|110\rangle$	01	$I \otimes I \otimes X$	Correcting Error on q_3

In general, for a code of distance d, up to $\lfloor \frac{d-1}{2} \rfloor$ bit-flip errors can be corrected using $d-1$ ancilla-based parity checks.

B. MATLAB Live Script Implementation

1) Workflow Overview and Circuit Analysis: The initial step analyzes the original (unprotected) circuit structure by counting qubits and gates by type. This baseline enables precise quantification of resource overhead introduced by this particular error correction.

2) Repetition Encoding and Transversal Gate Mapping: Once the original circuit has been analyzed, the next phase involves transforming it into a fault-tolerant version using a quantum repetition code of configurable distance d. Each logical qubit is encoded into d physical qubits through entanglement, spreading information across code blocks to enable detection and correction of single bit-flip errors.

The encoding process is implemented in two steps:

- `encodeSingleQubitRepetition` generates CNOT gates to entangle d physical qubits per logical qubit.
- `encodeAllLogicalQubits` applies this encoding to all qubits in the original circuit.

Following the encoding, the original logical gates must be remapped transversally to act correctly on the new encoded layout. Single-qubit gates (e.g., X, H, RZ) are applied in parallel across all d physical qubits of a code block, while multi-qubit gates (e.g., CNOT or Toffoli) are replicated across corresponding pairs of code blocks. The function `applyRepetitionCode` combines encoding and remapped gates into a complete error-protected circuit.

3) Syndrome Measurement, Error Injection and Correction: After encoding the circuit, we need a mechanism to detect and correct potential bit-flip errors without disrupting the logical quantum information. This is achieved through syndrome measurements, which extract error information using ancilla qubits while preserving the encoded state.

In the 3-qubit repetition code, two parity checks— Z_1Z_2 and Z_2Z_3 —are sufficient to detect any single-qubit bit flip error. The function `buildSyndromeBlock` constructs the CNOT-based measurement circuit and allocates the required ancilla qubits (one per parity check, i.e., $d-1$ per logical qubit). These ancillas are used to extract a syndrome, a binary vector that encodes which (if any) qubit has flipped.

To evaluate the system's correction capability, the script includes error injection using the `injectBitFlipError`

function. This allows any physical qubit to be manually flipped via an X gate, simulating the effect of a bit flip error.

The correction process involves:

- `measureSyndromeAndDecode`: simulates a measurement to extract the error syndrome;
- `applySyndromeCorrection`: interprets the syndrome to determine which physical qubit experienced an error and applies an X correction gate only to that qubit.

Each bit-flip error produces a unique syndrome, which can be reverted through a corrective action, demonstrating effective circuit-level error detecting and correction through modular, reusable components, that can be adapted to more complex circuits or longer repetition distances.

4) Fidelity Verification and Resource Overhead: To assess the effectiveness of the error correction scheme, the resource includes a fidelity analysis phase. Fidelity measures how close the corrected quantum state is to the ideal, error-free logical state. A fidelity close to 1 indicates successful correction.

The function `pureStateFidelity` computes it, and a loop systematically injects single-qubit bit-flip errors across each physical qubit in an encoded code block. After each injection, the correction pipeline is executed: syndrome extraction, corrective gate application, and state decoding. The fidelity is then computed against the reference state, confirming successful quantum information recovery and validating the repetition code's theoretical correction capacity at distance d.

The function `countGates` summarizes gate usage by type, and a final table contrasts the original and encoded circuit statistics. This allows learners to quantitatively observe the cost of achieving fault tolerance and to reflect on the practical trade-offs in quantum computing implementations.

5) Monte Carlo Logical Error Rate Simulation: While fidelity analysis confirms the code's ability to correct isolated errors, a Monte Carlo simulation evaluates performance under realistic probabilistic noise by estimating logical error rate versus physical bit-flip probability p.

In this simulation, the code performs repeated trials in which each physical qubit experiences a bit-flip with probability p. Each trial checks whether majority vote over d physical qubits yields correct logical values. If more than half of the qubits flip, the correction fails—resulting in a logical error.

Varying p from 0 to 0.5 across 10^5 trials per point, the constructed curve of logical error rate vs. physical error rate demonstrates a non-linear behavior:

- For small p, the logical error rate is significantly reduced compared to the unencoded case.
- As p approaches 0.5, the benefits of repetition vanish and error correction becomes ineffective, unless d grows.

This illustrates the error suppression threshold: the region where encoding improves reliability. It also allows to observe that QEC only works when the underlying physical qubit reliability is above a critical level.

The resulting plot (Figure 3) visually compares the Monte Carlo-estimated logical error rates against the uncorrected case, offering a statistical view of how error correction scales with hardware noise and parameter selection.

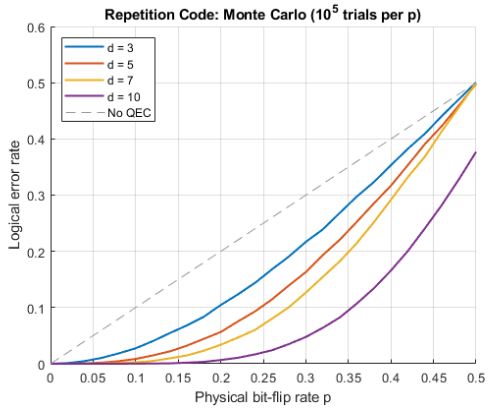

Fig. 3. Monte Carlo simulation of logical error rate versus physical bit-flip probability p for a repetition code of distance d.

C. Learning Outcomes

Through this resource, learners will:

- Understand how the quantum repetition code protects against single-qubit bit flip errors.
- Implement encoding, error injection, syndrome measurement, and correction in MATLAB.
- Simulate to assess fidelity and error-correction success.
- Quantify the resource overhead introduced by error correction techniques.
- Evaluate logical error rates via Monte Carlo simulation and interpret threshold behavior.

VI. Conclusion

This work presents three educational resources that guide learners through some of the foundations of quantum computing. Each one combines theoretical concepts and MATLAB code implementations, demostrating how also quantum algorithms translate from abstract concepts to executable code.

The algorithms treat different topics, but together offer an overview of what can be performed in this domain. Shor's algorithm reveals quantum computing's computational advantage through a number-theoretic problem. QAOA explains the hybrid quantum-classical paradigm that defines near-term quantum devices. Quantum error correction tackles the challenge of maintaining quantum coherence in realistic systems.

Implementing these algorithms exposes practical considerations often hidden in purely theoretical treatments. Learners encounter circuit optimization challenges, resource limitations, and performance trade-offs that develop critical intuition about when different quantum approaches are most effective. This hands-on experience reinforces that quantum computing extends beyond mathematical formalism into a domain of engineering decisions and computational constraints.

REFERENCES

[1] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, 2002.

[2] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proc. 35th Annu. Symp. Found. Comput. Sci.*, 1994, pp. 124–134. doi:10.1109/SFCS.1994.365700

[3] O. Gamel and D. F. V. James, "Simplified factoring algorithms for validating small-scale quantum information processing technologies", 2013, arXiv:1310.6446v2 [quant-ph].

[4] F. G. Gemeinhardt, A. Garmendía, M. Wimmer, B. Weder, and F. Leymann, "Quantum combinatorial optimization in the NISQ era: A systematic mapping study," *ACM Comput. Surv.*, vol. 56, no. 3, pp. 1–36, Oct. 2023, doi:10.1145/3620668.

[5] K. Blekos, D. Brand, A. Ceschini, C.-H. Chou, R.-H. Li, K. Pandya, and A. Summer, "A review on Quantum Approximate Optimization Algorithm and its variants," 2023, arXiv:2306.09198v2 [quant-ph].

[6] E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm," 2014, arXiv:1411.4028 [quant-ph].

[7] F. Glover, G. Kochenberger, and Y. Du, "A tutorial on formulating and using QUBO models," arXiv:1811.11538v6 [cs.DS], 2018. [Online].

[8] E. Guney, J. Ehrenthal, and T. Hanne, "Quantum approaches to the 0/1 multi-knapsack problem," 2025. [Note: Future publication, no further details available.] https://www.scitepress.org/Papers/2025/133877/133877.pdf

[9] A. Awasthi *et al.*, "Quantum computing techniques for multi-knapsack problems," in *Intelligent Computing – Proceedings of the Computing Conference 2023*, K. Arai, Ed., Lecture Notes in Networks and Systems, vol. 739, Springer, Cham, 2023, pp. 264–284. doi:10.1007/978-3-031-36440-7_17, arXiv:2301.05750v2 [quant-ph].

[10] P. W. Shor, "Scheme for reducing decoherence in quantum computer memory," *Phys. Rev. A*, vol. 52, no. 4, pp. R2493–R2496, 1995. doi:10.1103/PhysRevA.52.R2493

[11] D. Ristè, S. Poletto, M.-Z. Huang, A. Bruno, V. Vesterinen, O.-P. Saira, and L. DiCarlo, "Detecting bit-flip errors in a logical qubit using stabilizer measurements," arXiv preprint arXiv:1411.5542, 2014.

[12] Z. Chen, K. J. Satzinger, J. Atalaya, A. N. Korotkov, *et al.*, "Exponential suppression of bit or phase-flip errors with repetitive error correction," arXiv preprint arXiv:2102.06132 [quant-ph], 2021.

[13] A. Pesah, "Stabilizer formalism 1," [Online]. Available: https://arthurpesah.me/blog/2023-01-31-stabilizer-formalism-1/. Accessed: May 2025.

[14] IBM Quantum, "Foundations of Quantum Error Correction: Correcting Quantum Errors," [Online]. Available: https://learning.quantum.ibm.com/course/foundations-of-quantum-error-correction/correcting-quantum-errors. Accessed: May 2025.