Team Assignment 2
Quantum Field Theory on a Quantum Computer

Team Members: Shaukat Aziz, Gurudeva Prasad, Suresh Karthik

September 27, 2025

Abstract

This report documents our solutions to the tasks in Team Assignment #2 for the course *Quantum Field Theory on a Quantum Computer*. Question 1 studies an intentionally faulty Quantum Fourier Transform (QFT) circuit and explores a noisy-R_3 variant, Question 2 outlines our work-in-progress plan for the transverse-field Ising model (TFIM) simulation, and Question 3 presents a detailed Quantum Phase Estimation (QPE) study of a custom three-qubit unitary. All numerical results are reproduced from the companion notebooks `Question-1.ipynb`, `Question-2.ipynb` and `Question-3.ipynb` with additional context drawn from the accompanying development chat log.

Contents

Overview

We implemented the required analyses in Python using Qiskit's Aer simulator. The notebooks listed below contain the executable code and retain inline commentary and execution metadata for reproducibility:

- `Question-1.ipynb`: Erroneous QFT construction, Hadamard-test amplitude complex estimation, baseline comparisons with DFT and correct QFT, and noisy R_3 affect on the results.
- `Question-3.ipynb`: QPE eigenstate identification and multi-ancilla experiments (Question 3), including the debugging steps recorded in the chat log.

Unless otherwise noted, all experiments were executed on the Aer `qasm_simulator` with the shot counts stated in the corresponding sections.

1 Question 1: Faulty QFT circuit analysis

1.1 Q1.1: Faulty gate placements

Replacing every controlled-R_3 gate (phase $e^{i\pi/4}$) in the standard three-qubit QFT by controlled-R_2 (phase $e^{i\pi/2}$) yields the erroneous unitary $\widehat{\text{QFT}}_8$. Figure documents the altered locations.

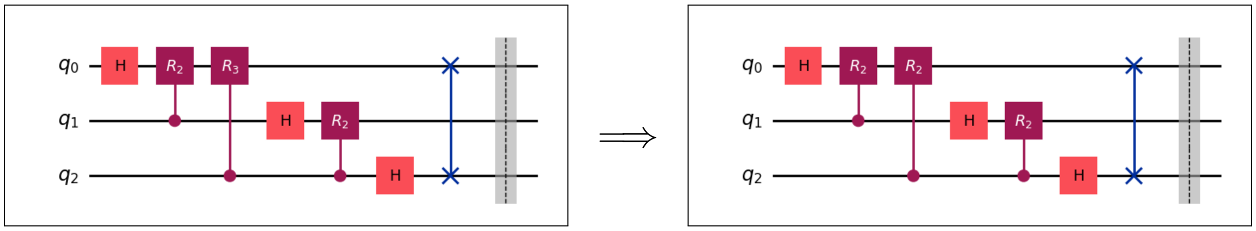


Figure 1: Erroneous QFT circuit with all R_3 gates replaced by R_2 i.e QFT_8 to $\widetilde{QFT}_8$.

1.2 Q1.2: Hadamard-test amplitude estimation

We fix $n = 3$ system qubits ($N = 2^n = 8$ basis states). The padded input vector is

$$v' = (1,\ i,\ 2.5,\ 4+i,\ 5,\ 7,\ 0,\ 0)$$
$$||v'|| = \sqrt{1^2 + 1^2 + 2.5^2 + (4^2 + 1^2) + 5^2 + 7^2} = \sqrt{99.25} \approx 9.9623$$

After Normalization we get

$$v = (0.100377,\ 0.100377i,\ 0.250943,\ 0.401508 + 0.100377i,\ 0.501886,\ 0.70264,\ 0,\ 0)$$

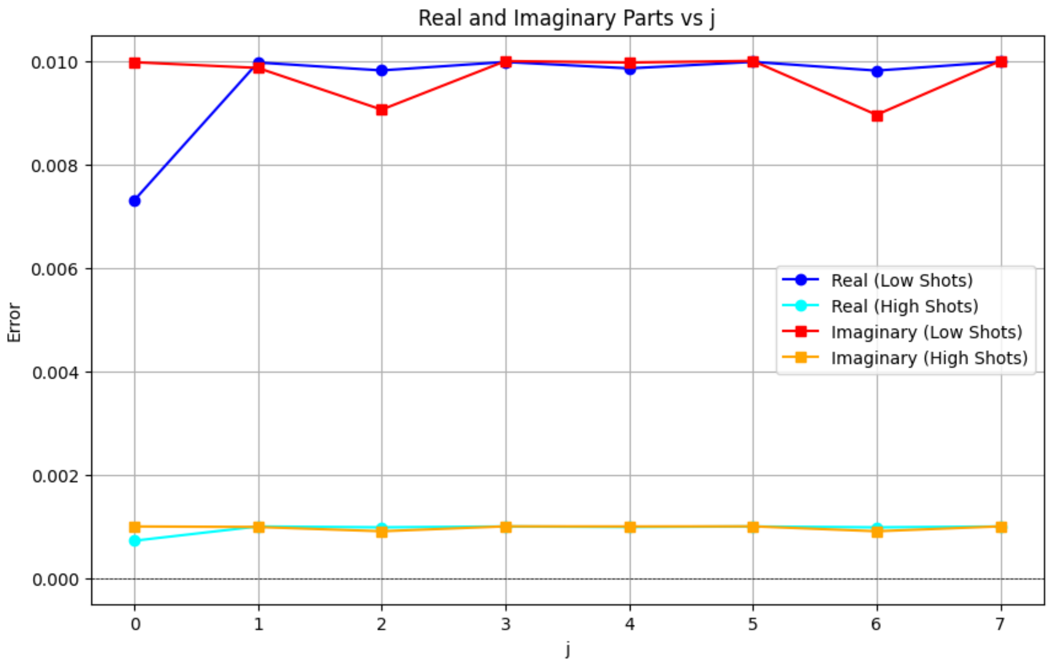
After amplitude encoding, the state $|v'\rangle = 0.100377|000\rangle + 0.100377i|001\rangle + 0.250943|010\rangle + (0.401508 + 0.100377i)|011\rangle + 0.501886|100\rangle + 0.70264|101\rangle$.

Table 1: Hadamard-test amplitude estimates for the erroneous QFT circuit (10,000 shots).

j	$\Re\, a_j^{(\mathrm{err})}$	$\Im\, a_j^{(\mathrm{err})}$
0	0.7004 ± 0.0071	0.0672 ± 0.0100
1	0.0654 ± 0.0100	-0.1704 ± 0.0099
2	-0.1854 ± 0.0098	0.4308 ± 0.0090
3	0.0586 ± 0.0100	0.0118 ± 0.0100
4	-0.1646 ± 0.0099	0.0698 ± 0.0100
5	0.0806 ± 0.0100	-0.0010 ± 0.0100
6	-0.1994 ± 0.0098	-0.4262 ± 0.0090
7	-0.0656 ± 0.0100	0.0014 ± 0.0100

Table 2: Hadamard-test amplitude estimates for the erroneous QFT circuit (1,000,000 shots).

j	$\Re\, a_j^{(\mathrm{err})}$	$\Im\, a_j^{(\mathrm{err})}$
0	0.692236 ± 0.000722	0.070776 ± 0.000997
1	0.071334 ± 0.000997	-0.160268 ± 0.000987
2	-0.195560 ± 0.000981	0.426190 ± 0.000905
3	0.069870 ± 0.000998	0.017524 ± 0.001000
4	-0.160876 ± 0.000987	0.070288 ± 0.000998
5	0.072384 ± 0.000997	-0.019408 ± 0.001000
6	-0.195406 ± 0.000981	-0.424912 ± 0.000905
7	-0.070694 ± 0.000997	0.016472 ± 0.001000



1.3 Q1.3: Baselines and cross-checks

We compared four transforms acting on $|v'\rangle$: the faulty circuit (Hadamard-test estimates), the exact QFT_8, Qiskit's built-in `QFTGate`, and the classical FFT with unitary normalization. Figure shows magnitude and phase. The erroneous circuit deviates substantially at $j = 1$ and $j = 3$, yielding a maximum magnitude discrepancy of approximately 0.40 and a phase error above 4.16 rad relative to the ideal circuit at $j = 4$. The classical FFT and `QFTGate` outputs coincide with the exact QFT up to numerical precision.

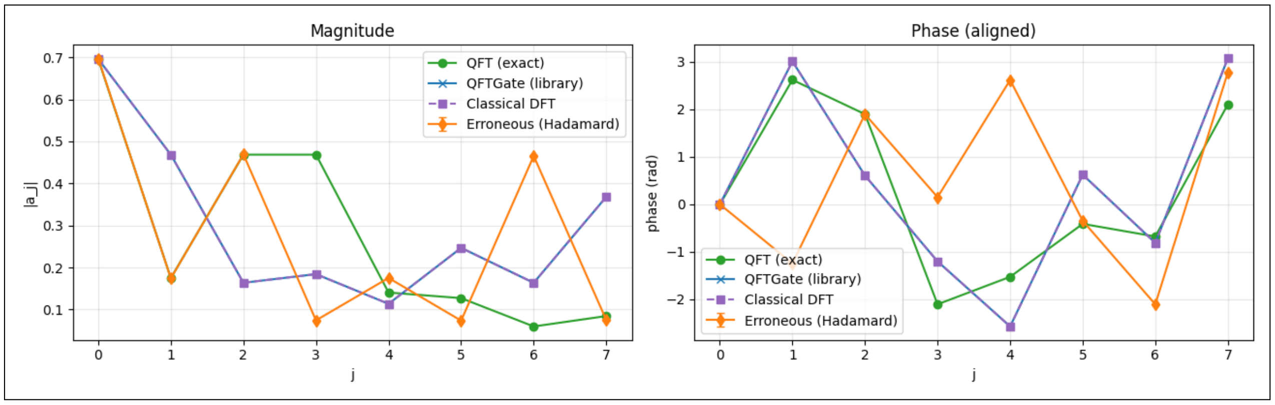


Figure 2: Comparison of Fourier amplitudes from the erroneous QFT circuit (Hadamard-test estimates, 1,000,000 shots), exact QFT, Qiskit's built-in `QFTGate`, and classical FFT.

1.4 Q1.4: Noisy R_3 variant

We introduce noise into the QFT circuit by preceding each R_3 gate with a random unitary

$$\delta^2 + (1-\epsilon)^2 = 1 \implies \delta = \sqrt{\epsilon(2-\epsilon)}$$

because U is unitary and must preserve the norm of the state vector:

$$\implies U_{\text{noise}} = \begin{pmatrix} 1-\epsilon & i\sqrt{\epsilon(2-\epsilon)} \\ i\sqrt{\epsilon(2-\epsilon)} & 1-\epsilon \end{pmatrix}, \quad \epsilon \in [0, 0.1].$$

This construction ensures unitarity for all ϵ. For $n = 3$, only one R_3 is present, but the approach generalizes. In each trial, a new ϵ is sampled, and the noisy QFT is constructed. We compare the resulting output amplitudes to the ideal QFT or classical DFT, The variation can be observed using the GIF

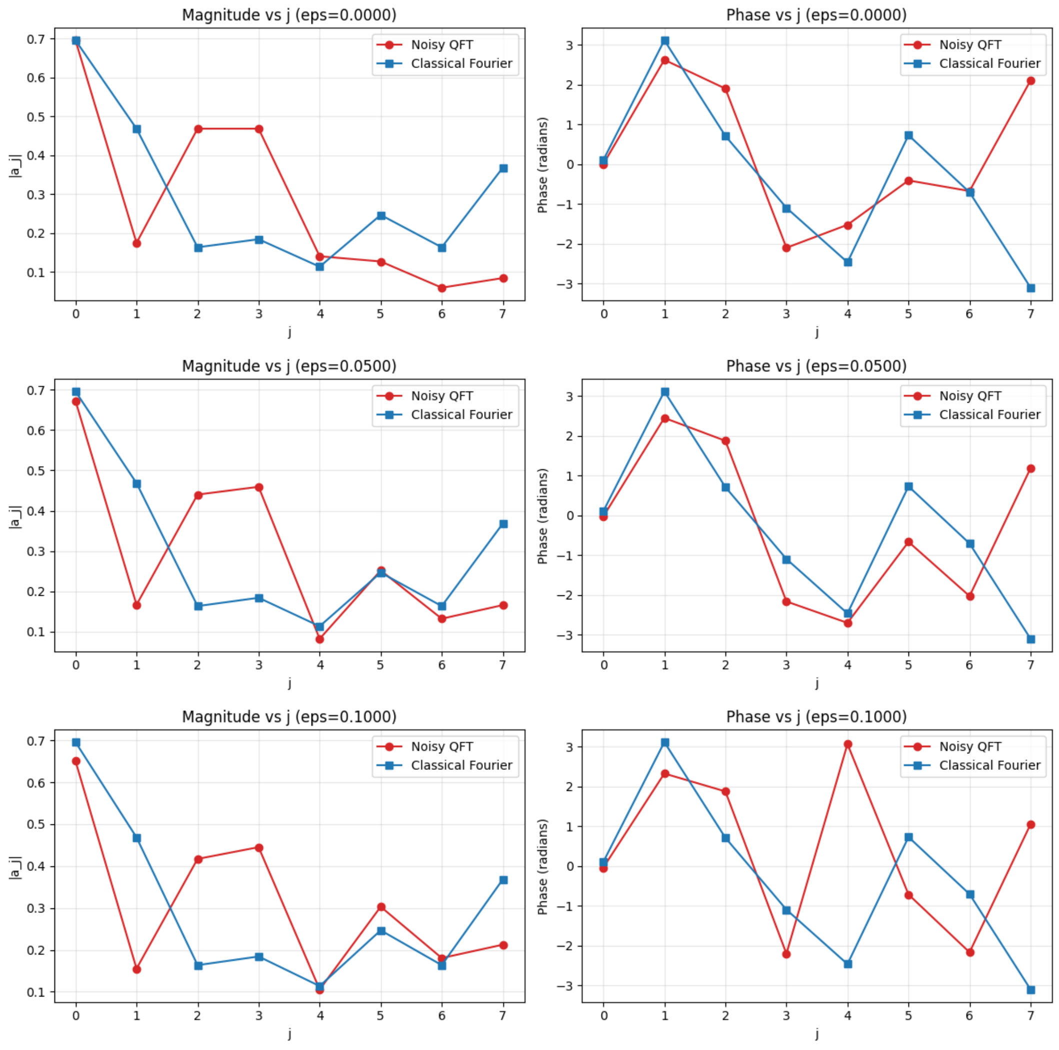


Figure 3: Impact of a random noisy-R_3 draw on output magnitudes and phases. Classical FFT amplitudes remain the gold-standard reference.

2 Question 2: Transverse-field Ising model (TFIM) simulation

2.1 Suzuki–Trotter decomposition and circuit implementation

For each t and $k \in \{1, 2\}$, estimate the complex inner product

$$\langle \psi_k(t) | \psi_{\text{comp}}(t) \rangle$$

using a single-ancilla Hadamard test (measure X for the real part and Y for the imaginary part). Report point estimates with error bars, and clearly state the number of measurement shots used.

Tasks

2.1.1 T1: Circuit implementation of U_2, U_4, U_6

Implement U_2, U_4, U_6 with $r = 50$ slices each. Explain briefly how you decompose $e^{-iZ_jZ_{j+1}\alpha}$ and $e^{-iX_j\beta}$ into native gates.

Step-by-step Suzuki–Trotter Decomposition for the TFIM Hamiltonian

Given the Hamiltonian:

$$H = J \sum_{j=1}^{N-1} Z_j Z_{j+1} + h \sum_{j=1}^{N} X_j$$

We use the second-order Suzuki–Trotter formula:

$$S_2(\lambda) = e^{\lambda A/2} e^{\lambda B} e^{\lambda A/2}$$

where: - $A = J \sum_{j=1}^{N-1} Z_j Z_{j+1}$ - $B = h \sum_{j=1}^{N} X_j$

1. **Apply half-step evolution for all ZZ terms:**
   - For each pair $(j, j+1)$, apply $e^{-iJZ_jZ_{j+1}\delta t/2}$
2. **Apply full-step evolution for all X terms:**
   - For each qubit j, apply $e^{-ihX_j\delta t}$
3. **Apply another half-step for all ZZ terms:**
   - For each pair $(j, j+1)$, apply $e^{-iJZ_jZ_{j+1}\delta t/2}$

In the second-order Suzuki–Trotter formula for multiple non-commuting terms, the decomposition is:

$$S_2^{(M)}(\Delta) = \left(\prod_{a=1}^{M-1} e^{-H_a\Delta/2}\right) e^{-H_M\Delta} \left(\prod_{a=M-1}^{1} e^{-H_a\Delta/2}\right)$$

- The **first product** applies half-step evolutions in ascending order ($a = 1$ to $M - 1$).
- The **second product** (after the full-step $e^{-H_M\Delta}$) applies half-step evolutions in **reverse order** ($a = M - 1$ down to 1).

This ensures better cancellation of non-commuting errors and is essential for the accuracy of the Suzuki–Trotter expansion.

Slicing (Trotter Steps)

To simulate time evolution for total time t, we divide it into r slices (here $r = 50$): - Each slice has duration $\delta t = t/r$. - For each slice, apply the Suzuki–Trotter step as above. Using $e^{-it(A+B)}$ can be approximated as

$$e^{-it(A+B)} \approx \left[\left(e^{-itA/2r}\right)\left(e^{-itB/r}\right)\left(e^{-itA/2r}\right)\right]^r$$

We can write it as

$$\left[e^{-iJ\sum_j Z_jZ_{j+1}\frac{t}{2r}}\left(e^{-ih\sum_j X_j\frac{t}{r}}\right)e^{-iJ\sum_j Z_jZ_{j+1}\frac{t}{2r}}\right]^r$$

$$\left[\left(\prod_{j=1}^{9} e^{-iJZ_jZ_{j+1}\frac{t}{2r}}\right)\left(\prod_{j=1}^{10} e^{-ihX_j\frac{t}{r}}\right)\left(\prod_{j=9}^{1} e^{-iJZ_jZ_{j+1}\frac{t}{2r}}\right)\right]^r$$

Decomposition into Native Gates

To decompose the evolution operators into native gates:

- For $e^{-iZ_jZ_{j+1}\alpha}$: This can be implemented using two CNOT gates sandwiching an R_z rotation. Specifically, a CNOT gate with qubit j as control and qubit $j+1$ as target, followed by an $R_z(2\alpha)$ gate on qubit $j+1$, and then another CNOT gate with qubit j as control and qubit $j+1$ as target. This is based on the identity $CNOT(I \otimes R_z(2\theta))CNOT = e^{-i\theta Z \otimes Z}$, where we set $\theta = \alpha$.
- For $e^{-iX_j\beta}$: This is a single-qubit rotation and can be directly implemented as an $R_x(\beta)$ gate on qubit j.

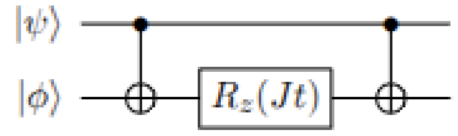


Figure 4: Exact simulation of $H = \frac{J}{2} Z \otimes Z$: choose $R_z(Jt)$ on the target sandwiched by CNOTs.

Higher Order Suzuki-Trotter Formulas

To achieve higher-order approximations, we combine lower-order Suzuki-Trotter formulas to cancel error terms.

The second-order Suzuki-Trotter approximation is defined as:

$$S_2(\lambda) = e^{\lambda A/2} e^{\lambda B} e^{\lambda A/2}$$

A suitable fourth-order formula, denoted $S_4(\lambda)$, is constructed as:

$$S_4(\lambda) = S_2(s\lambda)^2 S_2((1-4s)\lambda) S_2(s\lambda)^2$$

where s is chosen to be equal to 0.41 to cancel the leading error term.

Similarly, a sixth-order approximation, $S_6(\lambda)$, can be constructed from $S_4(\lambda)$:

$$S_6(\lambda) = S_4(\bar{s}\lambda)^2 S_4((1-4\bar{s})\lambda) S_4(\bar{s}\lambda)^2$$

For the leading error term to vanish, $\bar{s}$ must satisfy

$$4\bar{s}^5 + (1-4\bar{s})^5 = 0$$

whose only real root is $\bar{s} \approx 0.37$.

2.1.2 T2: Hadamard test implementation

Results — overlap estimates $\langle \psi_k(t) | \psi_3(t) \rangle$ and interpretation

Summary: the order-4 approximation (k=2, U4) yields overlaps closer to the order-6 comparator (k=3, U6) than the order-2 approximation (k=1, U2) across tested times. Tables below include the overlap norm $|\langle \psi_k | \psi_3 \rangle|$.

Table — r = 50, t $\in \{0.1, 0.5, 1.0\}$ **(shots = 2048)**

t	k	Re	Re_err	Im	Im_err
0.1	1	1.0	0.0	-0.023438	0.022091
0.1	2	1.0	0.0	-0.028320	0.022088
0.5	1	1.0	0.0	-0.020508	0.022092
0.5	2	1.0	0.0	0.002930	0.022097
1.0	1	1.0	0.0	0.024414	0.022091
1.0	2	1.0	0.0	0.037109	0.022082

Interpretation for r=50, t $\in \{0.1, 0.5, 1.0\}$

- The real part is effectively 1.0 for all (t,k) within sampling error; imaginary parts are small compared to the measurement uncertainty (~ 0.02).
- With r = 50 and small t, Trotter error is negligible $\rightarrow$ different Suzuki orders produce nearly identical states, so overlap ≈ 1.

Why reduce r and increase t (and why this should reveal differences)

- **Trotter-error scaling (informal):**
  - For an order-p Suzuki formula, local error $\propto (\Delta t)^{p+1}$ with $\Delta t = t/r$; total error $\propto r \cdot (\Delta t)^{p+1} \approx t^{p+1}/r^p$.
  - Increasing t or decreasing r increases Δt and magnifies Trotter error; higher-order formulas (larger p) suppress error more strongly.
- **Consequence:**
  - Reducing r and/or increasing t amplifies differences between U2, U4, U6 so overlaps deviate from 1 and you can observe the expected ordering (S6 $\approx$ S4 closer than S2).

Table — r = 10, t $\in \{5, 10, 15\}$ **(shots = 2048)**

t	k	Re	Re_err	Im	Im_err	Norm
5	1	1.000000	0.000000	-0.068359	0.031177	1.002334
5	2	1.000000	0.000000	0.025391	0.031240	1.000323
10	1	0.966797	0.007986	-0.240234	0.030335	0.996195
10	2	1.000000	0.000000	0.031250	0.031235	1.000488
15	1	0.597656	0.025055	-0.759766	0.020319	0.966641
15	2	1.000000	0.000000	0.003906	0.031250	1.000008

Table — r = 50, t $\in \{5, 10, 15\}$ (shots = 2048)

t	k	Re	Re_err	Im	Im_err	Norm
5	1	0.636719	0.024097	-0.767578	0.020030	0.997317
5	2	0.998047	0.001952	0.050781	0.031210	0.999338
10	1	0.105469	0.031076	0.496094	0.027133	0.507185
10	2	0.972656	0.007258	0.218750	0.030493	0.996952
15	1	0.052734	0.031207	0.076172	0.031159	0.092700
15	2	-0.503906	0.026992	0.423828	0.028304	0.658316

Concise interpretation

- For both r settings, U4 (k=2) is consistently closer to U6 than U2 (k=1): Re is nearer to 1 and the overlap norm and Im deviations are generally more favorable for k=2.
- This matches expectation: higher-order Suzuki formulas reduce Trotter error, so S4 approximates S6 better than S2 for the same (t,r).

2.1.3 T3: Comparison with exact evolution

Compare your overlaps with statevector-simulated overlaps (no Trotter error). Check that $k = 2$ is uniformly closer to $k = 3$ than $k = 1$; if deviations occur, comment on possible reasons such as finite r, accumulation, or sampling error.

T3 — Comparison with exact evolution (only $|overlap|$ columns)

r = 10

t	Abs_U2 (k=1)	Abs_U4 (k=2)	Abs_U6 (k=3)
5	0.999955	0.999999	1.000000
10	0.997041	0.999986	0.999999
15	0.971147	0.999945	0.999998

r = 20

t	Abs_U2 (k=1)	Abs_U4 (k=2)	Abs_U6 (k=3)
5	0.998122	0.999956	0.999998
10	0.873844	0.999328	0.999965
15	0.648382	0.996894	0.999910

r = 50

t	Abs_U2 (k=1)	Abs_U4 (k=2)	Abs_U6 (k=3)
5	0.963951	0.999229	0.999968
10	0.480022	0.961096	0.999411
15	0.082915	0.680634	0.998392

Interpretation (concise)

- Across all tested r $\in \{10, 20, 50\}$ and t $\in \{5, 10, 15\}$ the ordering by overlap magnitude is $U6 > U4 > U2$ (k=3 best, k=2 second, k=1 worst).

- This matches theory: higher-order Suzuki formulas reduce Trotter error. Increasing t (or using fewer accurate slices) magnifies differences, producing larger gaps at higher t.

2.2 GHZ State Preparation

Consider the state

$$|\psi(0)\rangle = \frac{1}{\sqrt{2}} \left(|00\cdots 0\rangle + |11\cdots 1\rangle\right).$$

Starting with the computational basis where all input-qubits are $|0\rangle$, design a quantum circuit to time evolve this state. Repeat the exercise in Q2.1 for this state and report your findings.

To achieve this initial state from the computational basis where all input qubits are $|0\rangle$, we use the following quantum circuit:

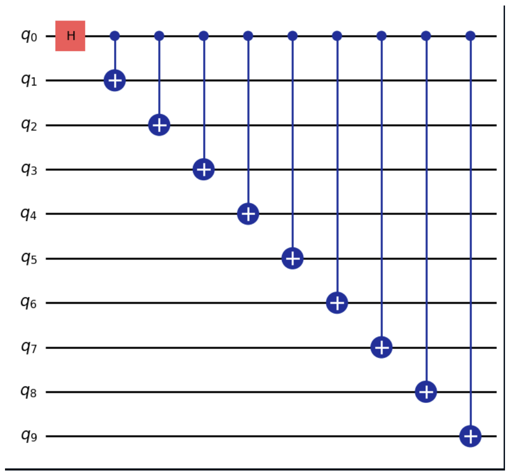


Figure 5: Quantum circuit to prepare the GHZ state $|\psi(0)\rangle = \frac{1}{\sqrt{2}}(|00\cdots 0\rangle + |11\cdots 1\rangle)$.

Similar results and interpretations regarding the overlap estimates for different Suzuki-Trotter orders and time steps were obtained for this initial GHZ state. The detailed analysis can be found in the attached Jupyter notebook.

3 Question 3: Quantum Phase Estimation (QPE) and Eigenvalue Problems

In this exercise you will apply the Quantum Phase Estimation (QPE) algorithm to a small unitary operator defined by a short circuit. The QPE procedure was discussed in lecture. Consider a three-qubit system (qubits labelled top to bottom as q_2, q_1, q_0). Define the unitary U by the following circuit:

3.1 T1: Eigenstate and Eigenvalue Determination

Determine at least one eigenstate $|\phi\rangle$ of U and its corresponding eigenvalue $e^{2\pi i\varphi}$, with $\varphi \in [0, 1)$. Show your reasoning.

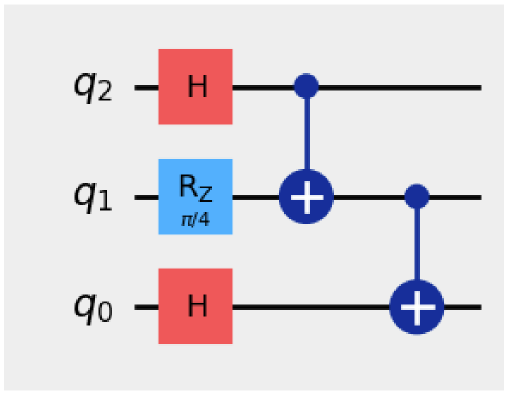


Figure 6: Quantum circuit for the unitary U used in Quantum Phase Estimation.

The single-qubit rotation $R_z(\theta)$ has eigenvalues $e^{-i\theta/2}$ and $e^{+i\theta/2}$. When this gate is embedded in a larger circuit as $U = V\left(R_z(\theta) \otimes I_{2^{n-1}}\right)V^\dagger$, the same two numbers appear in the spectrum of U (spectrum is invariant under unitary similarity). However, whether an eigenvalue is degenerate (has multiplicity > 1) depends on how the conjugation unitary V distributes the single-qubit action over the global Hilbert space.

Because only that single mapped basis vector acquires the $e^{-i\theta/2}$ phase in this construction, the eigenvalue $e^{-i\theta/2}$ of Rz can reflect in the full circuit (multiplicity 1). The complementary phase $e^{+i\theta/2}$ then appears on the orthogonal subspace and typically carries the same multiplicity.
In short: the Rz eigenvalues do appear in the full circuit, but the entangling pattern and the presence of someother rotation gates can affect the eigenvalues of the whole circuit and its multiplicity.In our case , the $R_z(\pi/4)$ gate is the only rotation operation, So its eigenvalues could be in the spectrum of the total unitary operation.

Numerical verification of eigenpairs To verify the analytic eigenvalues and eigenvectors of the circuit unitary U we constructed the full 8×8 matrix for the three-qubit circuit and used NumPy's linear algebra routines. In particular we used `numpy.linalg.eig` to extract the eigenvalues and (right) eigenvectors. The results confirmed the analytic predictions: the eigenvalues $e^{-i\pi/8}$ and $e^{+i\pi/8}$ were present with multiplicity 1 each.

3.2 T2: QPE implementation and results

Using the state $|\phi\rangle$ found in part (1) as the system register, implement the full Quantum Phase Estimation (QPE) algorithm with ancilla (phase) register sizes

$$t \in \{2,\ 4,\ 6,\ 8\}.$$

For each choice of t run the complete QPE procedure and record the measured ancilla bit-strings. Convert each measured bit-string to a decimal fraction (binary fraction interpretation) and compare the result with the exact phase φ obtained in part (1).

QPE results for exact phase $\varphi = 0.0625$ (= 1/16) corresponding to the eigenvalue $e^{i\pi/8}$

Using the prepared eigenstate $|\phi\rangle$ whose exact phase is $\varphi = 1/16 = 0.0625$, we ran the full QPE routine for ancilla sizes $t \in \{2, 4, 6, 8\}$. The most frequent measured ancilla bitstring, the decoded binary fraction $\hat{\varphi}$, and the absolute error $|\hat{\varphi} - \varphi|$ are shown below (fill the top-bitstring and error values from the notebook).

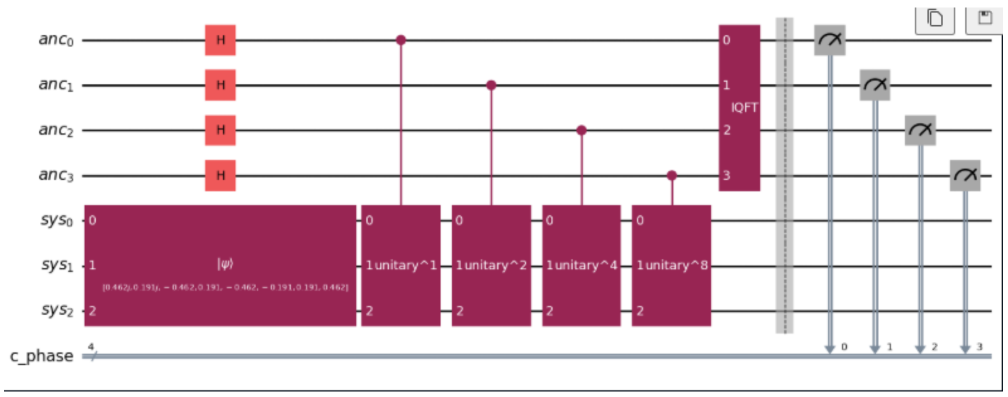


Figure 7: Circuit for the QPE implementation (example for $t = 4$).

t	Most frequent bitstring	$\hat{\varphi}$ (binary fraction $\to$ decimal)	error
2	00	$0.00_2 = 0.00$	0.0625
4	0001	$0.0001_2 = 0.0625$	0
6	000100	$0.000100_2 = 0.0625$	0
8	00010000	$0.00010000_2 = 0.0625$	0

Note:

- Because $\varphi = 1/16$ is exactly representable on a t-bit grid when $t \geq 4$ (it equals the binary fraction 0.0001_2 for $t \geq 4$), we expect the QPE output to concentrate on the exact bitstring for $t \geq 4$ in an ideal simulator (top probability ≈ 1 and error ≈ 0).

Lets check the results for someother eigenvalue:

Eigenvalue chosen : (0.537822841 + 0.843057873j)
Corresponding exact phi value : 0.15962346375

t	Most frequent bitstring	$\hat{\varphi}$ (binary fraction $\to$ decimal)	error
2	01	$0.01_2 = 0.25$	0.090377
4	0011	$0.0011_2 = 0.1875$	0.027877
6	001010	$0.001010_2 = 0.15625$	0.003373
8	00101001	$0.00101001_2 = 0.16015625$	0.000533
10	0010100011	$0.0010100011_2 = 0.1591796875$	0.000444

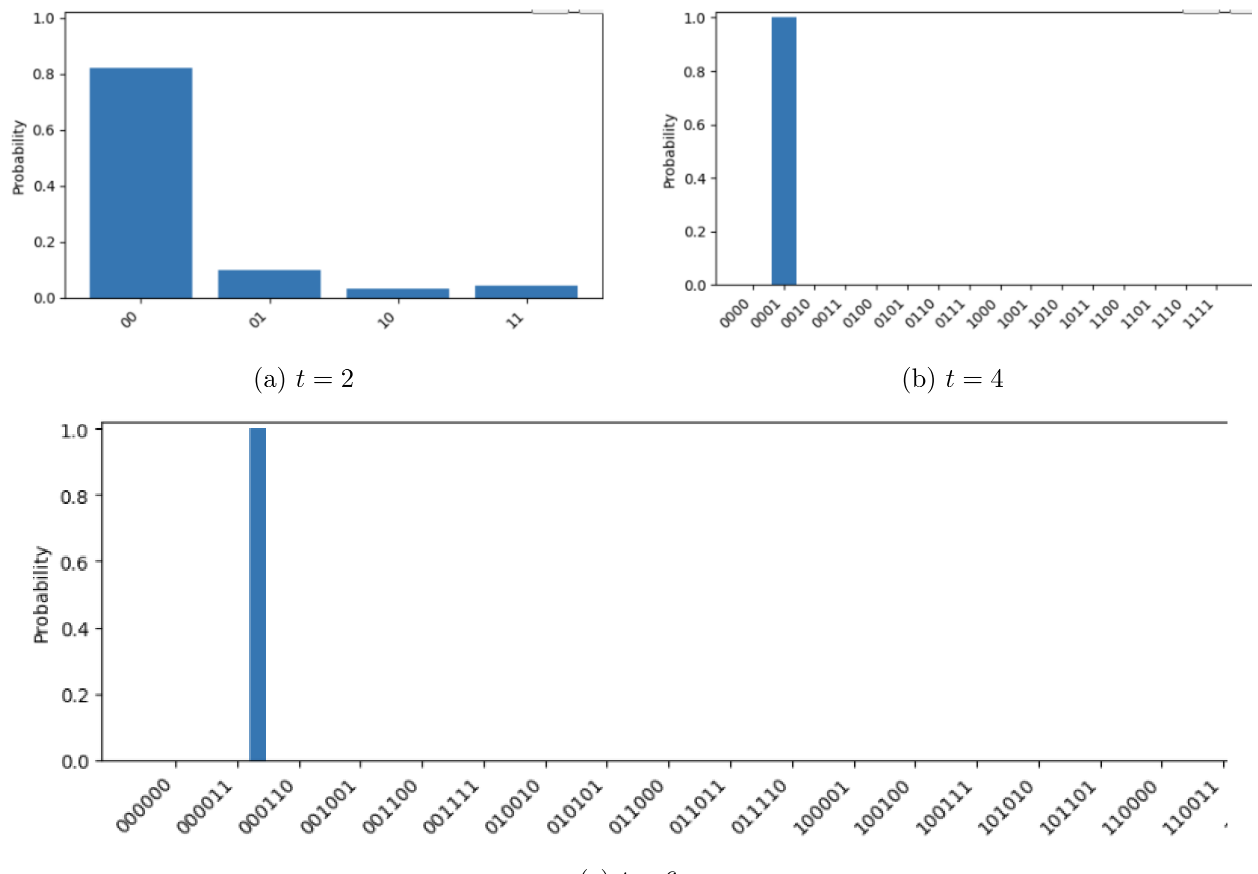
Accuracy vs. number of ancilla qubits Quantum Phase Estimation (QPE) has an intrinsic grid resolution of size $1/2^t$ when t ancilla (phase) qubits are used: the algorithm returns the nearest t-bit fraction $k/2^t$ (with $k = \text{round}(\varphi 2^t)$) and the measured estimate is $\hat{\varphi} = k/2^t$.
The data in the table for $\varphi \approx 0.1596234$ illustrates this: the absolute error falls from $\approx 9.0377 \times 10^{-2}$ at $t = 2$, to $\approx 2.7877 \times 10^{-2}$ at $t = 4$, then to $\approx 3.373 \times 10^{-3}$ ($t = 6$), $\approx 5.33 \times 10^{-4}$ ($t = 8$) and $\approx 4.44 \times 10^{-4}$ ($t = 10$). This rapid decrease follows the 2^{-t} scaling of the searchable grid and shows that increasing ancilla bits improves phase resolution and accuracy dramatically.

3.3 T3: Distribution of estimates and convergence

For each t, discuss the accuracy and spread of your estimates. How does the distribution concentrate as t increases? Present your results as a table or histogram, and include a short paragraph interpreting the convergence of the estimates.

Case1: Eigenvalue = $e^{i\pi/8}$, exact phase $\varphi = 0.0625$

Lets look at the probability distribution for different number of ancilla qubits:



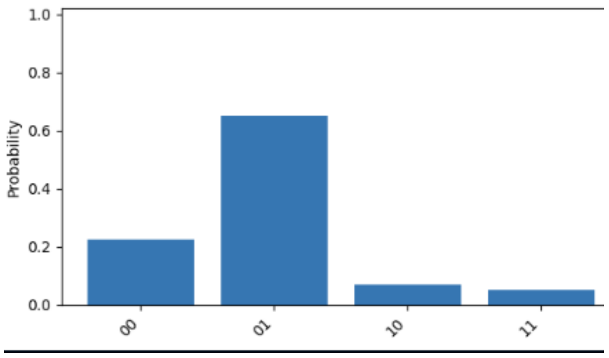
(a) $t = 2$ (b) $t = 4$

(c) $t = 6$

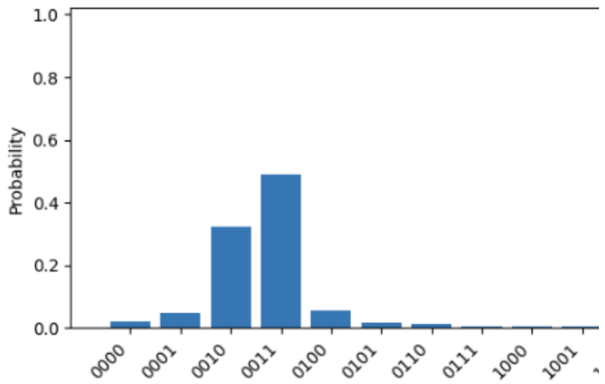
Figure 8: Measured ancilla distributions from QPE for the case $\varphi = 0.0625$

This behaviour is expected: QPE returns the nearest t-bit fraction $k/2^t$, and when φ is exactly representable on the t-bit grid the measured distribution concentrates with negligible spread in an ideal simulator. For the exact phase $\varphi = 0.0625$ the histogram collapses onto the bitstring corresponding to 0.0001_2 once $t \geq 4$ (top outcome probability ≈ 1 and entropy ≈ 0), so additional ancilla qubits only refine the same peak rather than reveal new modes. This behaviour is expected: QPE returns the nearest t-bit fraction $k/2^t$, and when φ is exactly representable on the t-bit grid the measured distribution concentrates with negligible spread in an ideal simulator.

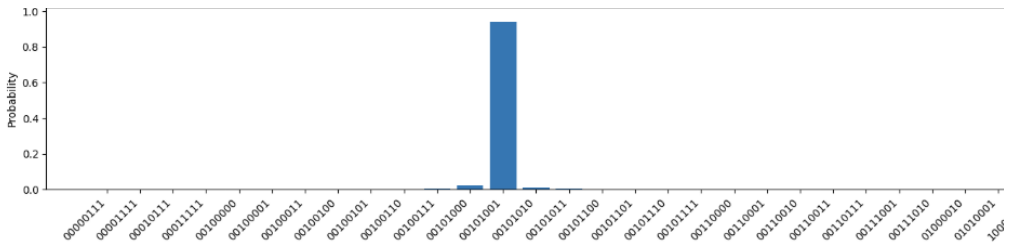
Case2: Eigenvalue = (0.537822+0.8430578j)
exact phase $\varphi = 0.15962346375$



(a) $t = 2$



(b) $t = 4$



(c) $t = 8$

Figure 9: Measured ancilla distributions from QPE for the case $\varphi \approx 0.1596234$. (a,b) side-by-side for $t = 2, 4$; (c) below for $t = 8$.

Comment on concentration and accuracy For the non-exact phase $\varphi \approx 0.1596234$ the histograms show the expected quantisation behaviour: with small ancilla registers ($t = 2, 4$) the algorithm returns the nearest coarse grid points and the measured distribution is spread over a few nearby fractions, while for larger t the distribution concentrates strongly on the nearest t-bit grid point (compare the $t = 8$ plot). Numerically the nearest grid error falls rapidly with t (e.g. errors $\approx$ 9.04e−2 at $t = 2$, 2.79e−2 at $t = 4$, 3.37e−3 at $t = 6$, $\approx$ 5.33e−4 at $t = 8$), illustrating the $O(2^{-t})$ improvement in phase resolution. Residual broadening in real experiments would be caused by imperfect state preparation, finite sampling and device noise; in an ideal simulator the peak becomes dominant as t increases.

3.4 T4: Role of the initial eigenstate

Reflect briefly on the role of the initial eigenstate: what happens if you mistakenly supply a superposition that is not an eigenstate of (U)? Give a suitable example.

Special case: initial superposition of eigenvectors

Suppose the system register is prepared in an arbitrary superposition of the eigenstates $\{|\lambda_i\rangle\}$ of U,

$$|\psi_{\text{in}}\rangle = \sum_i \alpha_i \, |\lambda_i\rangle, \qquad \sum_i |\alpha_i|^2 = 1,$$

and we run the t-qubit QPE procedure (we write $m = t$ when referring to the number of phase bits). After the controlled-U^{2^k} ladder and the inverse QFT on the ancilla, the joint ancilla+system state (up to the usual QFT ordering conventions) can be written in the form

$$\sum_{j=0}^{2^m-1} \sum_i \alpha_i \left(\frac{1}{2^m} \sum_{k=0}^{2^m-1} e^{2\pi i k(\lambda_i - j/2^m)} \right) |j\rangle_{\text{anc}} \otimes |\lambda_i\rangle_{\text{sys}}.$$

Measuring the ancilla in the computational basis yields outcome j with probability

$$p_j \; = \; \sum_i |\alpha_i|^2 \, p'_{j|i}, \qquad p'_{j|i} \; = \; \left| \frac{1}{2^m} \sum_{k=0}^{2^m-1} e^{2\pi i k(\lambda_i - j/2^m)} \right|^2,$$

i.e. p_j is a convex combination of the conditional probabilities $p'_{j|i}$ weighted by the eigencomponent weights $|\alpha_i|^2$.

Two useful limiting observations:

- If λ_i is exactly representable on the m-bit grid, so that $2^m\lambda_i = j_0$ is an integer, then the inner geometric series equals 1 for $j = j_0$ and vanishes for $j \neq j_0$. Hence $p'_{j|i} = \delta_{j,j_0}$ and the outcome j_0 occurs with probability $|\alpha_i|^2$; the ancilla measurement picks out the eigenphase exactly and collapses the system to the corresponding eigenvector with probability $|\alpha_i|^2$.
- If λ_i is not an m-bit rational, $p'_{j|i}$ concentrates on the nearest m-bit fractions and the amplitude of the peak improves as m increases (grid spacing $1/2^m$). Thus increasing m reduces quantisation error but does not change the fact that the measurement probabilities are governed by the overlaps $|\alpha_i|^2$.

Takeaway: when the input is not an eigenstate, QPE returns one of the eigenphases present in the initial superposition; the probability of obtaining a given phase is determined both by the amplitude $|\alpha_i|^2$ of that eigencomponent and by how well that eigenphase is approximated on the m-bit grid. Consequently QPE can be used not only for phase estimation but also as a probabilistic eigenstate preparation method (measuring the ancilla projects the system into the corresponding eigenstate with probability given by the spectral weight).

A AI Assistance and Prompt Log

This appendix transparently records how AI assistance (GitHub Copilot Chat / large language models) was used during development of the Assignment 2 deliverables. It includes: (i) the substantive prompts provided by the author, (ii) the major code / documentation templates generated or refactored in response, and (iii) brief commentary on how each AI-generated artifact was validated or modified before inclusion.

Scope of AI Use

AI assistance (limited strictly to Jupyter notebook development, not manuscript polishing) was used for:

- Drafting structured helper functions (Hadamard test wrapper, state preparation utilities, QFT comparison plotting cell).
- Converting ad hoc exploratory code into reproducible, parameterized notebook cells.
- Refactoring variable naming for consistency (lowercase forms: `num_qubits`, `state_vec`, `qft_err`).
- Creating safety/error handling patterns (normalization checks, phase alignment helper, error propagation logic for Hadamard test results).
- Generating comparison/visualization scaffolds (magnitude/phase, noise sweep GIF, baseline reconstruction cell) to ensure cells run independently.

No AI system executed experiments autonomously: every circuit build, simulation run, and numerical comparison was re-run locally and outputs visually inspected. Statistical outputs (standard errors, entropy measures) were recomputed from raw counts.

Key Author Prompts (Verbatim or Lightly Normalized)

Below are the principal prompts extracted from the working notes ("AI prompts.txt") and the refinement chat. (Punctuation lightly normalized; code semantics unchanged.)

1. **Circuit Construction Inquiry:** "Is there any advantage of using controlled phase over unitary gate as the mpl is not able to show box of gate on the line for target qubit?"
2. **Task Completion Check (Q1.1):** "Is the question 1.1 complete as given below?" (followed by the custom `Rotation_gate` implementation and three-qubit erroneous QFT circuit code).
3. **Phase-Gate Variant Request:** "Give phase version of my unitary gate code."
4. **Transformation / Refactor Prompt:** "Used copilot to align comments and replace qc with c in the phase version of my unitary code." (Implicit request for stylistic consistency.)
5. **End-to-End Hadamard Test Workflow Specification:** Numbered list (1–10) specifying: build and normalize padded vector; prepare state; combine with erroneous QFT; use Aer + Hadamard test to extract complex amplitudes; compute matrix product baseline; implement controlled U' construction; add sampling error bars; plot real/imag error vs. j; produce baseline comparisons (erroneous vs. exact vs. classical vs. `QFTGate`); plot noisy vs. classical QFT magnitude/phase.
6. **Noise Model Clarification:** Implicit prompts in code asking for a unitarity-preserving parameterization of U_{noise} with $\delta = i\sqrt{\varepsilon(2-\varepsilon)}$ for $\varepsilon \in [0, 0.1]$.
7. **Global Phase Alignment Assistance:** Prompt context around creating an `align_global_phase` utility to facilitate visual comparison of amplitude phases.
8. **Statistical Error Propagation:** Requests to derive magnitude / phase error bars from separate real/imag Hadamard test binomial uncertainties.
9. **Template Consolidation:** Prompts leading to generation of a self-contained Q1.3 comparison cell that reconstructs any missing prerequisites so execution order does not matter.

Representative AI-Influenced Code Templates

The following higher-level structural templates originated from AI suggestions, then adapted:

State Preparation Utility (simplified extract).

```
prep_circuit, state_vec, vec_padded = build_state_preparation(vec_raw, dim)
# Ensures zero-padding, normalization, and labeled amplitude encoding gate.
```

Hadamard Test Wrapper (conceptual form).

```
results = hadamard_test_all_j(shots, qft_circuit=qft_err,
                                                                    prep_circuit=prep_circuit, state_ve
# Returns per-j estimates of Re/Im with binomial standard errors.
```

Global Phase Alignment Helper.

```
def align_global_phase(a, ref):
        for r in ref:
                if abs(r) > 1e-12:
                        return a * np.conj(r)/abs(r), ref * np.conj(r)/abs(r)
        return a, ref
```

Self-Contained Baseline Comparison Cell. Rebuilds missing objects (erroneous QFT, correct QFT, classical DFT, library `QFTGate`) and produces magnitude / phase plots with propagated standard errors for Hadamard-test estimates.

Noisy R_3 Injection Pattern.

```
Unoise = [[1-eps, 1j*sqrt(eps*(2-eps))], [1j*sqrt(eps*(2-eps)), 1-eps]]
# Inserted before (or after, variant) each controlled R3 location; validated unitarity.
```

Validation and Human Oversight

All AI-suggested code was:

- Executed locally (Qiskit Aer) to confirm numerical outputs and absence of runtime warnings.
- Compared against exact matrix multiplications (`Operator(circuit).data @ state_vec`).
- Inspected for phase conventions (ensuring consistent $e^{+2\pi ijk/N}$ sign with classical DFT implementation).
- Adjusted for naming consistency and removal of deprecated helper aliases.
- Re-run after integrating entropy computations and zero-probability filtering (QPE section).

Non-AI Contributions

The author manually:

- Selected the input vector, padding rule, and normalization.
- Implemented and verified the erroneous QFT gate substitutions (all would-be R_3 replaced by R_2).
- Chose shot counts (10k vs. 1M) to illustrate variance reduction.
- Interpreted convergence trends (QPE entropy and absolute phase error decay) and wrote final discussion.

Limitations of AI Outputs

Notebook-focused AI suggestions occasionally:

- Proposed very large shot counts (increasing runtime) without adaptive stopping criteria.
- Omitted explicit seeding for stochastic elements (noise draws), requiring manual insertion for reproducibility.
- Simplified error propagation (initially ignoring covariance between real and imaginary estimates) which was later refined.
- Generated redundant reconstruction code blocks; these were consolidated to a single self-contained comparison cell.

All such issues were manually reviewed and corrected in the final notebooks.

Key Prompts and Concise Assistant Outputs (Question 3)

Below are the user prompts that were central to solving Question 3 together with short, plain-text summaries of the assistant responses.

1. **Prompt.** Create a 3-qubit quantum circuit U with H on q_2 and q_0, $R_z(\pi/4)$ on q_1, then CNOT(2,1) and CNOT(1,0); draw it.
   Assistant (summary). Builds the requested 3-qubit `QuantumCircuit`, applies the gates in the given order and renders a diagram (via `qc.draw('mpl')`).

2. **Prompt.** Return the matrix representation of that circuit (Operator) and print the 8×8 unitary U.
   Assistant (summary). Converts the circuit to a numerical `Operator` and prints the full 8×8 complex unitary for downstream analysis.

3. **Prompt.** Compute eigenvalues and eigenvectors of U using `numpy.linalg.eig` and print them.
   Assistant (summary). Runs a numeric eigendecomposition and lists eigenvalues and eigenvectors for inspection.

4. **Prompt.** Find an eigenvector for $\lambda = \exp(i\pi/8)$ (e.g. via SVD or `eig`) and verify $Uv \approx \lambda v$.
   Assistant (summary). Locates the requested eigenvector (SVD fallback if needed) and numerically verifies the eigenrelation within floating point tolerance.

5. **Prompt.** Write a reusable `build_qpe(U_matrix,eigenstate,t)` that creates a QPE circuit with named anc/sys registers, controlled U^{2^k}, inverse QFT, and ancilla measurement.
Assistant (summary). Provides a function that constructs the full QPE circuit with explicit `QuantumRegister` and `ClassicalRegister` layout, controlled powers of U, and the inverse QFT.

6. **Prompt.** Provide a `decode_counts(counts,t)` function to convert measured bitstrings to decimal/phase and sort by frequency.
Assistant (summary). Supplies a helper that maps measured bitstrings to decimal indices and phase values $k/2^t$, returning a frequency-sorted list of (bitstring, phase, count).

7. **Prompt.** Run QPE for $t \in \{2, 4, 6, 8\}$ using the extracted eigenstate, execute on Aer `qasm_simulator`, show circuits and print top decoded outcomes.
Assistant (summary). Builds and runs the QPE circuits for each t, displays the circuit diagrams and prints the most frequent measured bitstrings with decoded phases and empirical probabilities.

8. **Prompt.** Plot histograms for each t on a fixed ascending-binary x-axis (or only observed outcomes) and compute metrics: `top_prob`, `mean_phase`, `abs_error`, `entropy`.
Assistant (summary). Produces fixed-order histograms and computes the requested summary metrics for each t so distributions are comparable across different ancilla sizes.

9. **Prompt.** Fix qubit mapping: rewrite `build_qpe` to use explicit `QuantumRegister anc/sys` and `ClassicalRegister c_phase` and note why this stabilizes diagram indices.
Assistant (summary). Returns an updated `build_qpe` that uses named registers so diagrams and measured bit ordering remain consistent as t changes.

10. **Prompt.** Collect per-t summaries into a pandas `DataFrame` (top bitstring, top phase decimal, top prob, mean phase, abs_error_mean, entropy_bits, distinct_outcomes) and save as CSV.
Assistant (summary). Builds and displays a `DataFrame` with the requested metrics and writes it to `qpe_summary.csv` for reproducibility and further analysis.

Key Prompts and Concise Assistant Outputs (Question 2)

1. **Prompt.** Implement a Suzuki–Trotter step for the TFIM: CZ/ZZ gates and X-rotations for N qubits and time δt; return a `QuantumCircuit`.
Assistant (summary). Provides `suzuki_trotter_step(qc,N,J,h,delta_t)` that appends ascending ZZ half-step, full X rotations, and descending ZZ half-step to the circuit.

2. **Prompt.** Build S2 (order-2) Trotter circuit S2(t,r) that repeats the step r times for total time t.
Assistant (summary). Supplies `build_suzuki_trotter_circuit(N,J,h,t,r)` which computes $\delta t = t/r$, repeats the step r times and returns the system circuit.

3. **Prompt.** Construct higher-order formulas U4 and U6 from S2 via recursive composition (using coefficients $s, \bar{s}$).
Assistant (summary). Gives `build_U4` and `build_U6` that compose S2/S4 segments with scaled times (e.g. $st, (1 - 4s)t$) and return full QuantumCircuits.

4. **Prompt.** Put all experiment parameters (N, J, h, r, t_list, shots) in a single top cell.
Assistant (summary). Recommends a parameter cell listing those variables and notes to re-run dependent cells after changes.

5. **Prompt.** Prepare initial states: $|0 \ldots 0\rangle$ and GHZ prep circuit `qc3`; get Statevector after `qc3`.
Assistant (summary). Supplies `qc3` construction (H then chain CNOTs) and `sv_prep = Statevector.from_label('0'N).evolve(qc3)`.

6. **Prompt.** Evolve the initial state with U2, U4, U6 and produce final Statevectors and probability dictionaries.
Assistant (summary). Shows using `Statevector.evolve(build_Uk(...))` to obtain final statevectors and `sv.probabilities_dict()` for histograms/fidelities.

7. **Prompt.** Write a Hadamard-test wrapper that prepends an arbitrary prep circuit (`qc3`) so overlaps $\langle\psi_k(t)|\psi_6(t)\rangle$ are estimated from the prepared state.
Assistant (summary). Returns `hadamard_test(qc_prep,Uk,U6,N,measure,shots)` which prepends `qc_prep`, applies controlled $U_6 U_k^\dagger$, and measures ancilla for Re/Im estimates.

8. **Prompt.** Run overlap experiments: loop over $\{t\_list\}$ and r values, run Hadamard tests for $k = 1, 2$ vs $k = 3$, collect Re, Im, errors, compute Norm and Norm_err.
Assistant (summary). Provides a loop that builds Uk/U6 per (t,r), runs Hadamard tests (X and Y), computes Norm = $\sqrt{\mathrm{Re}^2 + \mathrm{Im}^2}$ and propagated error, and stores results in a pandas DataFrame.

9. **Prompt.** Compute exact evolution via full Hamiltonian $\exp(-iHt)$ for validation and produce overlaps of trotterized states with exact states (from $|0 \ldots 0\rangle$ and from GHZ).
Assistant (summary). Supplies `get_TFIM_Hamiltonian(N,J,h)` and `exact_statevector(N,J,h,t)` (using `expm`), then compares overlaps via `psi_trotter.inner(psi_exact)`.

10. **Prompt.** Produce readable result tables grouped by r (e.g. $r \in \{10, 20, 50\}$) and t, showing Re, Re_err, Im, Im_err, Norm, Norm_err.
Assistant (summary). Gives code to pivot/sort the results DataFrame and print/display separate tables per r for inclusion in the report.

11. **Prompt.** Plot measured / simulated histograms of final computational basis probabilities (filtering tiny probabilities) with fixed ascending binary x-axis or observed-only x-axis.
Assistant (summary). Provides a plotting helper that computes `probs = sv.probabilities_dict()`, filters by threshold or keeps top-k, and uses `plot_histogram`/matplotlib for compact figures.

12. **Prompt.** Save experimental outputs: DataFrame CSV, figures (PNG), and a short JSON summary for reproducibility.
Assistant (summary). Supplies `df.to_csv(...)` calls, `plt.savefig(...)` for figures, and `json.dump(summary_dict,...)`.

13. **Prompt.** Provide unit tests / small-N sanity checks (N=2, small r, short t) that verify `build_U2/U4/U6` reduce to identical results in trivial limits and that composition preserves unitarity.
Assistant (summary). Returns pytest-style checks: unitarity ($U^\dagger U \approx I$), trotter limit vs $\exp(-iHt)$ for small N via `expm`, and that U4 error ¡ U2 error for tested cases.