Creating a pipeline for switching between quantum error mitigation techniques

Andrei Oprescu, Raul de Brouwer, Andrei Mosescu, Antoni Spaanderman, Axel Koning and Jochem van der Vliet
(Dated: February 5, 2026)

As quantum computers continue to advance, noise and hardware imperfections remain a major obstacle to achieving reliable computational results on quantum devices, error mitigation techniques have emerged as a practical approach for reducing the impact of noise. This paper focuses on investigating the advantages and disadvantages of Zero noise extrapolation (ZNE), Probabilistic error cancellation (PEC) and Twirling readout error extinction (TREX), error mitigation methods. Afterwards a pipeline for switching between these methods, for where they are most suitable, is implemented. To assess each method's suitability, this paper measures ZNE's expectation value error over 6 different qubit counts and an increasing circuit depth, as well as PEC's expectation value error as the circuit depth is increased. Through these experiments, it was found that PEC, as long as the gate tomography of the quantum computer is available and accurate, can be used for shallow circuits with low depths. Moreover, ZNE is beneficial for slightly larger circuit depths and more qubits than PEC, as PEC's sampling overhead scales exponentially,while ZNE's cost scales linearly. Lastly TREX was found to be useful in low and high depth/qubit systems. With the information gathered, a plot with the method most suitable for different qubit-number and circuit-depth configurations was displayed. Ultimately, we composed a customizable python script using Qiskit that one can use for running any quantum circuit on a cloud quantum computer which will switch between PEC, ZNE and TREX based on their suitability.

I. INTRODUCTION

Near-term quantum devices operate in a regime where noise and errors strongly limit the outcome of computational measurements and results. Full quantum error correction remains physically demanding leading the way for quantum error mitigation to be developed. These techniques aim to reduce the impact of noise without requiring additional qubits or fault-tolerant hardware. Among the error mitigation approaches, the ones that stand out are Zero-Noise Extrapolation (ZNE), Probabilistic Error Cancellation (PEC) and Twirled Readout Error eXtinction (TREX).

ZNE works by introducing noise in the system in a specific way so that an expectation value can be derived of the state of the qubits. Which can be fitted using specific function curve-fits. Extrapolating this curve gives a noiseless expectation value. PEC works by disassembling a gate set into linear combinations of gates that do the same thing and testing which gate set causes the least noise. Finally TREX works by specifically targeting readout errors on noisy quantum systems. It does this by arbitrarily inserting random operators, which after twirling appears as a single bias factor. This bias factor can easily be determined from controlled calibration circuits, which can then be used to find the unbiased expectation value.

This paper will focus on simulating the error mitigation methods using Qiskit[1] backend, to find out what the advantages and disadvantages in different environments are. With this it can be decided in which situations which is better and an error mitigation pipeline which switches between these error mitigation techniques can be created.

II. ERROR MITIGATION METHODS

A. Zero noise extrapolation (ZNE)

The objective in ZNE is simple, find the average expectation value of some quantum observable A with respect to an evolved state $\rho_\lambda(T)$. Where T is the time that the evolved state is subject to noise. This noise is characterized by λ and using different extrapolation functions, the noiseless limit can be found [2].

The first step to ZNE is to intentionally scale noise. This can be done with different methods like Pulse-stretching, unitary folding or identity insertion scaling [3]. The second step involves extrapolating to the noiseless limit. This can be done by curve-fitting to the values measured at different noise levels, and then extrapolating a noiseless expectation value. This curve-fit can be made up of different functions depending on the system [2].

Techniques to intentionally increase noise level by increasing gate depth/complexity, can be done by either unitary folding or identity scaling [3]. In unitary folding, a mapping is done where a unitary version of the gate is added followed by another operation of the gate. For example $G \rightarrow GG^\dagger G$, in this mapping the 2 extra gates at the end cancel each other out because they form the identity matrix, but they still produce extra noise to the system. This method has a global and local variation, which slightly change the structure depending on what's useful. Figures of this can be seen below [4] [2] [5].

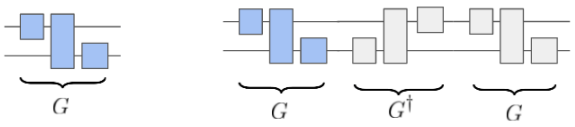


FIG. 1: Global unitary folding with general unitary gate G on the left and unitary folding on the right [3].

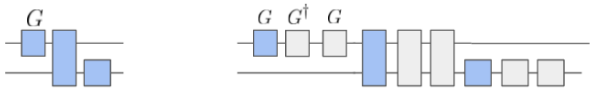


FIG. 2: Local unitary folding with general unitary gate G on the left and unitary folding on the right [3].

Identity insertion scaling works similarly. In this method a mapping is done where the identity matrix is inserted between every gate part of the original G operation, as seen in fig. 3. This works differently from unitary scaling as we are not using the matrix G to create extra noise, but rather a different identity matrix.

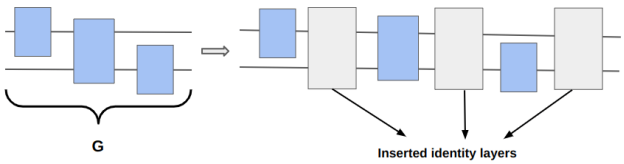


FIG. 3: Insertion scaling with general unitary gate G on the left and G with inserted identity matrices on the right [3].

Additionally, quantum systems with pulse level access can implement pulse-stretching. This way, noise is added by increasing the time over which pulses are implemented. This is shown in the following diagram.

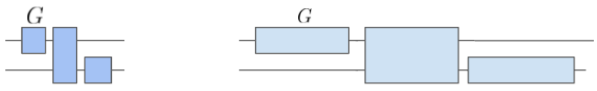


FIG. 4: Pulse stretching of general unitary gate G [3].

However as this method is difficult to achieve with the hardware provided by online sources, as IBM or Quantum Inspire, this paper will focus only on unitary folding or identity insertion scaling.

In the second step we extrapolate to the noiseless limit, with our objective being to find the expectation value of observable A. This is obtained from the final state $\rho_\lambda(T)$ as $E_K(\lambda) = tr(A\rho_\lambda(T))$. The noiseless limit is when $\lambda \to 0$, but to get this, first a range of expectation values $\langle E(\lambda)\rangle$ with $\lambda \geq 1$ are taken. Now we assume an extrapolation model using these expectation values, where f is the extrapolation model:

$$E(\lambda) \approx f(\lambda; p_1, p_2, ...p_m) \tag{1}$$

Then we fit the function to the optimal noise scaled values. And lastly evaluating the zero-noise limit $f(0; \tilde{p_1}, \tilde{p_2}, ...\tilde{p_m})$ [3][2][5][6].

The choice of the function f can range very wildly depending on the system. Chosen in this paper is a polynomial function of degree 2. Polynomial extrapolation is based on the following model of degree d:

$$E^d_{poly}(\lambda) = c_0 + c_1\lambda + ... + c_d\lambda^d \tag{2}$$

Here c_i are unknown constant real parameters. This function essentially corresponds to a Taylor series approximation justified in the weak noise regime. As opposed to Richardson extrapolation, this problem is in general only well defined if the number of data points m is at least equal to the number of free parameters $d + 1$. Another useful feature is that we can keep the extrapolation order small, but still keep a large number of data points m, which avoids an overfitting effect. However if we increase d by too much, then the bias is reduced, but the variance can increase so much that the mean squared error total is increased [6].

Richardson extrapolation is then also a particular case of polynomial extrapolation. The trick is to maximize degree d to the amount of data points m minus 1.

$$E^{m-1}_{Rich}(\lambda) = c_0 + c_1\lambda + ... + c_d\lambda^{m-1} \tag{3}$$

In this way the fitted polynomial can perfectly interpolate all the points and thus specifically cancel the first few noise terms based on asymptotic behavior. In this way in the ideal scenario where number of samples $N \to \infty$, the error with respect to the true expectation value is of the order $O(m)$. In reality however, there is a finite number of samples which makes it so the variance grows exponentially with m [6][2].

The last extrapolation method looked into is exponential based extrapolation.

$$E_{Exp}(\lambda) = a \pm e^{z_0 + z_1\lambda} = a + be^{-c\lambda} \tag{4}$$

In this case the set of a, b, c parameterize the same formula as the first with a, z_0, z_1. This extrapolation method assumes that the noise factor has some sort of exponential increase/decrease, which gives it an advantage in these specific cases. Another edge is that this extrapolation method is better than simple polynomial functions in moderate noise levels and there is less risk of overfitting in comparison to using higher order polynomials [7][6].

B. Probabilistic error cancellation (PEC)

Probabilistic error cancellation (PEC) [2] is a statistical error mitigation technique for quantum circuits. A benefit of this technique is that no extra qubits are needed for its implementation. The main goal of PEC is to recover an ideal, noiseless quantum circuit from a noisy quantum device. It works by replacing ideal gates G_i with a superposition of scaled noisy operationsre [2]:

$$G_i = \sum_\alpha \mu_{\alpha,i}\, O_{\alpha,i} \tag{5}$$

In the following, we show an example of a noisy X gate expressed as a linear combination of Pauli operators composed with the ideal gate:

$$\tilde{X} = (\mu_I I + \mu_X X + \mu_Y Y + \mu_Z Z) \circ X \tag{6}$$

The coefficients $\mu_{\alpha,i}$ form a normalized quasi-probability distribution, which means that individual $\mu_{\alpha,i}$ can be negative, but their sum is equal to 1 [8]:

$$\sum_\alpha \mu_{\alpha,i} = 1 \tag{7}$$

With this formulation, we can express the ideal expectation value of an observable A when a circuit U acts on an initialized state $\rho_0 = |0\cdots0\rangle\langle0\cdots0|$, but first we express U as a combination of G_i [2]:

$$U = \prod_i G_i = \prod_i \left(\sum_\alpha \mu_{\alpha,i}\, O_{\alpha,i} \right)$$

Expanding the product into a sum over all combinations of O_α, and using the same set of noisy operations for each G_i, we obtain:

$$U = \sum_{\alpha_1,\ldots,\alpha_n} \left(\prod_i \mu_{\alpha_i,i} \right) (O_{\alpha_1,1} \circ O_{\alpha_2,2} \circ \cdots \circ O_{\alpha_n,n}) \tag{8}$$

This expression can be compactly notated using vector notation:

$$U = \sum_{\boldsymbol{\alpha}} \mu_{\boldsymbol{\alpha}}\, O_{\boldsymbol{\alpha}}, \qquad \mu_{\boldsymbol{\alpha}} = \prod_i \mu_{\alpha_i,i}, \quad O_{\boldsymbol{\alpha}} = \circ_i O_{\alpha_i,i} \tag{9}$$

Using U, the expectation value of an observable A can then be written as:

$$\langle A \rangle = \sum_{\boldsymbol{\alpha}} \mu_{\boldsymbol{\alpha}}\, \mathrm{tr}\big[A\, O_{\boldsymbol{\alpha}}(\rho_0)\big] \tag{10}$$

A large benefit of the resulting expectation value is that it is bias free, meaning that it exactly reproduces the ideal noiseless expectation value in the limit of infinite sampling, at the cost of an increased statistical variance that scales with the quasi-probability overhead [2].

C. Twirling readout error extinction (TREX)

Twirling readout error extinction (TREX) is a method designed to reduce readout errors in quantum circuits. It improves the estimation of expectation values by randomly rotating the measurement basis prior to the measurement, after which some classical processing is done to undo this rotation to obtain the correct output. [9] An example of this is shown below, where the Z basis is used.

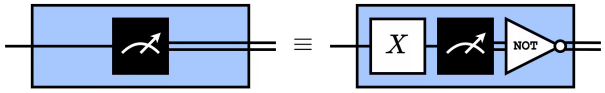
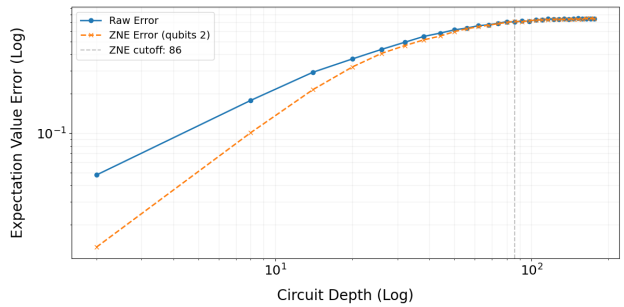


FIG. 5: Circuit that is used in TREX to mitigate measurement errors [10].

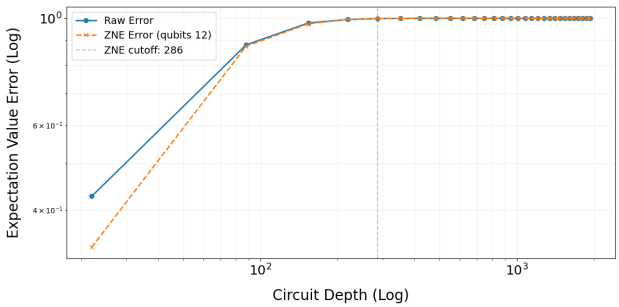
TREX is implemented by applying the Pauli X-gate on each qubit, with a 50% probability. If the Pauli X-gate was applied on the qubit, it is followed by a measurement and a classical NOT-gate. In an ideal system this will have no effect on the final outcome of the circuit, since each qubit flip is canceled by the bit flip and the measurement is considered to be ideal. If the measurement is non-ideal, it can show bias towards measuring a $|0\rangle$ or a $|1\rangle$. By randomly applying the flipping of the qubits, the detector measures both 1 and 0 equally often. This gives an estimate of the so called bias factor, which shows how much the detector is biased. In the final computation of your actual circuit you can compensate for this bias factor by simply dividing the average value by this bias factor.
The main advantage of TREX is that it is easy to implement since it does not require additional qubits and works on every circuit. This is why it is also used a lot and quite often is enabled by default on programmable quantum computers [11]. The main limitation TREX has is that it only removes potential bias in measurements, it does not correct random bit flips or other errors. That is why it is not very useful on its own, and why it is usually combined with other error correction methods such as ZNE or PEC.

III. RESULTS

In this section, we evaluate the performance of ZNE, PEC and TREX across a range of variables, such as



(a) The expectation value error of a 2-qubit circuit for increasing depths with a cutoff at 86.



(b) The expectation value error of a 12-qubit circuit for increasing depths with a cutoff at 286.

FIG. 6: The graphs of the unmitigated/ZNE-mitigated errors for two different circuits and their ZNE cutoff depths.

circuit depth and knowledge of the hardware. Our goal is to characterize the trade-off between sampling overhead and mitigation accuracy, ultimately displaying the logic behind our error mitigation pipeline.

A. Zero Noise Extrapolation Cutoff Evaluation

To assess the effectiveness of ZNE on a quantum circuit, we will use Mitiq, a python library dedicated to implementing and testing error mitigation techniques on a simulation of quantum hardware. For the implementation of the actual error mitigation switching pipeline, we will use Qiskit by IBM, as it provides a way to use quantum hardware provided by IBM, allowing us to run a circuit on a cloud quantum computer.

Deciding whether an error mitigation technique is suitable for a given circuit, we will evaluate the Expectation Value Error, as well as the sampling overhead.

ZNE's sampling overhead scales linearly with the depth of the circuit. Therefore, the main concern is it's ability to extrapolate the Expectation Value of the state of the qubits to a noiseless environment with an increase in circuit depth. At a specific circuit depth, ZNE becomes obsolete. Thus, we developed an algorithm in Mitiq to assess ZNE's effectiveness for different circuit depths and qubit number, which can be found in section VI B. The 1-qubit and 2-qubit gate errors were retrieved from the IBM-Torino cloud quantum computer to get an accurate simulation. For this experiment, the 1-qubit gate error was 0.009 and the 2-qubit gate error was 0.033.

From running this script on 2, 4, 6, 8, 10 and 12 qubits, we can get an insight into what the cutoff should be for when we should stop using ZNE and switch to another error mitigation method. The cutoff is chosen as the first circuit depth where the error of ZNE exceeds the unmitigated error. This point shows that the variance of ZNE becomes too large, and its effectiveness becomes negligible. Two plots containing ZNE's expectation value error and the unmitigated error across a range of depths on a 2-qubit and a 12-qubit circuit, can be found in fig. 6. Similar plots for circuits with different qubit numbers can be found in section VI A.

From the plots mentioned, we can observe that the depth cutoff point increases linearly as the number of qubits used increases and does not stay constant. This prompts us to examine the trend line of these increasing cutoff depths, and even extrapolate them to get a general formula for what the cutoff should be for an arbitrary number of qubits.

Therefore, after getting these depths, plotting them, and drawing a line of best fit, we get the plot from fig. 7.

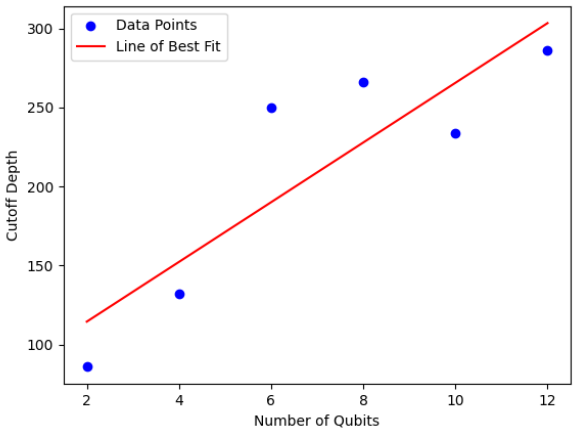
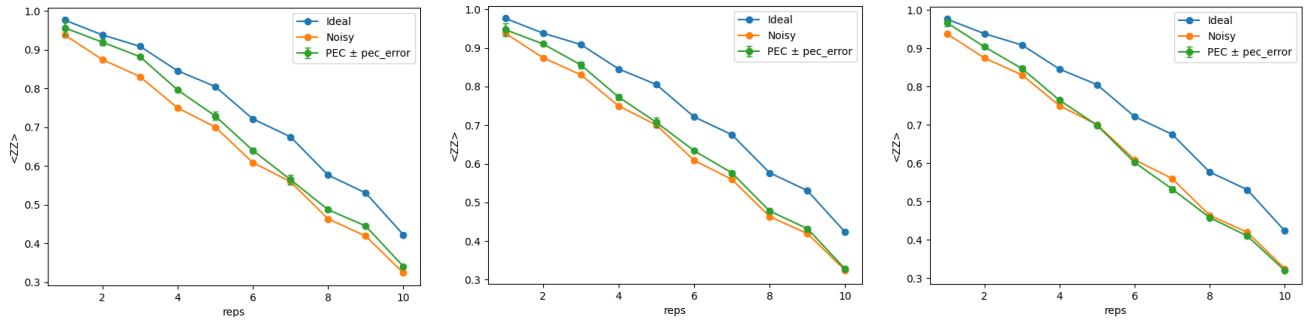


FIG. 7: Cutoff depth against number of qubits for ZNE including the trend line.

For circuits with qubit counts of 10 or less, the relationship appears to be linear. Therefore, when creating the error mitigation pipeline, we will take this into consideration, and use the linear relationship as the



(a) Expectation value against depth. The noise model PEC assumes the same as the back end's.

(b) Expectation value against depth. The noise model PEC assumes x1.5 the noise of the backend.

(c) Expectation value against depth. The noise model PEC assumes, doesn't include the phase damping.

FIG. 8: The ideal, unmitigated, and PEC expectation values under different simulation regimes.

marker for when we should stop using ZNE and start using only TREX.

B. Probabilistic Error Cancellation Cutoff Evaluation

We turn now to the evaluation of Probabilistic Error Cancellation (PEC) and how it compares to ZNE and no mitigation. We want to see how PEC works in practice, under different noise assumptions and how it's sampling overhead makes it use too many resources. Even if PEC is theoretically able to get unbiased expectation values, it's performance depends a lot on the accuracy of the noise model and sampling budget. To see how it's performance depends on the noise model assumed by PEC and the noise model of the backend, we used a simulation made with Qiskit Aer, for which we can control all the noise models.

In an ideal scenario where the noise model assumed by PEC exactly matches the noise model of the backend, PEC will reduce the expectation value error from the unmitigated results consistently, as seen in fig. 8a. At low depths, PEC will often perform better than ZNE, which is backed by the theoretical low bias property of PEC. When the circuit depth increases, the improvement becomes smaller. Although PEC still reduces the bias of the estimator, the accuracy is limited by the variance that increases with depth. For circuits with a larger depth PEC requires a large number of effective shots to get stable results. Even in an ideal scenario, we get as a result that PEC is impractical for deeper circuits because of it's overhead, but for small circuits PEC can outperform ZNE.

When using real hardware we don't usually know the exact noise model, so we also take a look at the scenarios when the assumed noise model of PEC does not match the back end's noise model. The first scenario is when the noise strength is overestimated by a factor of 1.5, and the second one when the true noise also includes a phase damping that PEC does not know about. We can see in fig. 8b that in the first case PEC still improves the error for lower depth circuits, but when the depth increases, the improvement decreases more rapidly than the matched-noise case. We can see that even small inaccuracies in the noise model can have a significant effect on the variance of PEC, especially at increased depths.

For the second scenario, we can see in fig. 8c that when a noise channel is missing from the assumed model, PEC performs a lot worse. There are still small improvements at low depths, but it very rapidly becomes worse than the unmitigated expectation values. In this regime, PEC does more bad then good, amplifying errors when the circuit is not trivial. From this we can clearly deduce that PEC is very sensitive to the correct noise model.

These plots were generated by the code provided in section VI C.

Now, we will reason about the cutoff point for PEC. Since PEC has low bias for its expectation value error, the main concern is the sampling overhead that comes with it. The sampling overhead is represented as a multiplier describing by how much the computation time will be increased.

The sampling overhead (γ) is the sum of the absolute values of the coefficients in its quasi-probability decomposition.

$$\gamma_i = \sum_{\alpha} |\eta_{i,\alpha}| \tag{11}$$

For an entire circuit consisting of D gates, the total overhead (γ_{total}) is the product of the overhead of all of the individual components.

$$\gamma_{total} = \prod_{i=1}^{D} \gamma_i \tag{12}$$

However, since γ_{total} is a product, if every layer has a similar average overhead $\bar{\gamma}$, the cost increases exponentially with the depth of the circuit d.

$$\gamma_{total} \approx (\bar{\gamma})^d \tag{13}$$

Since the number of shots N for attaining a specific precision ϵ depends on the following formula:

$$N_{mitigated} \approx \frac{\gamma_{total}^2}{\epsilon^2} \tag{14}$$

We see that the sampling overhead, as well as the number of shots required, scales exponentially with the depth of the circuit.

The value of the maximum PEC sampling overhead can be chosen arbitrarily. It depends on the user to decide how much overhead their circuit should have. A greater sampling overhead will yield more accurate results, but longer runtime. For this switching pipeline, we will choose a sampling overhead of 5.0, as it does not slow the program by a significant amount, but will allow our circuit to use PEC for large enough tasks for this error mitigation technique. Beyond this point, the mitigation does not have a practical advantage over the unmitigated or ZNE execution, since the PEC estimator is dominated by statistical fluctuations from the increased number of shots needed.

C. Error Mitigation Switching Pipeline

To show the use of ZNE, PEC, and TREX in different circuits, we will create multiple dummy circuits, with qubit numbers going from 1 to 30, and arbitrary depths from 1 to 30. For each arbitrary depth, we apply a CNOT gate between qubit $i-1$ and i for $i < n$ (n = arbitrary depth), from qubit 0 to qubit $n-1$. An example circuit can be found in fig. 9. The barriers are there only for visualization purposes. It is also important to note that the arbitrary depth is not a real representation of the depth of the circuit, and the real depth will have to be calculated after the circuit is prepared.

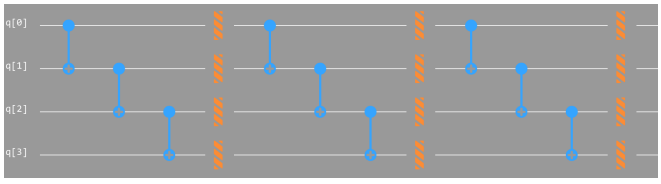


FIG. 9: Figure showing an example of a dummy circuit used for the pipeline with 4 qubits.

For each of these circuits, we will get their PEC sampling overhead and their depth. The logic behind the switching algorithm is as follows. If the PEC overhead is above a multiplier of 5, PEC will be used. This generally means that we will use PEC for smaller circuits, where the runtime is lower, such that a greater accuracy can be prioritized. Otherwise, if the circuit depth cutoff for the given number of qubits is below the one represented by the line of best fit found earlier, we will use ZNE. This ensures that ZNE is still useful for the error mitigation. Otherwise, only TREX will be used. It is also important to note that TREX will be used alongside both PEC and ZNE in their respective scenarios as it can be applied on top of them. For the circuits created, we will plot them on a color map which will display the error mitigation technique(s) used for each circuit.

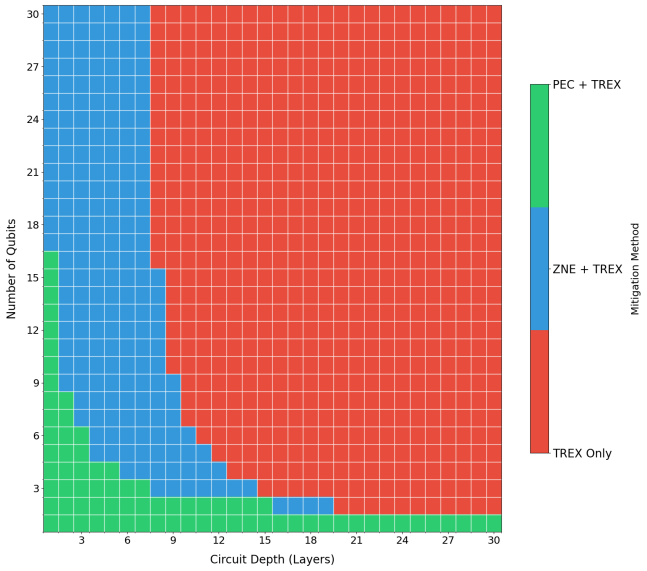


FIG. 10: The error mitigation methods used for different numbers of qubits and arbitrary circuit depths.

This color map can be seen in fig. 10, which was generated using the code provided in section VI D. Each square on the map represents a circuit with a specific number of qubits (y-axis) and a specific depth (x-axis), and their color is there to indicate which method is selected by the pipeline for that circuit. PEC is selected only for a small number of shallow circuits because of it's exponential overhead, being mostly used for 1 and 2 qubit circuits. On the other hand, ZNE is effective for greater depths than PEC as the number of qubits increases, remaining more favorable than TREX for shallow depths. Even so, the pipeline selects only TREX for larger depths and qubit counts, making it clear that TREX is the only viable mitigation technique with complex circuits.

A version of the code to be used for your own circuit is also provided in section VI E.

IV. DISCUSSION

A. Error mitigation scaling

Noise becomes exponentially worse with increasing circuit depth and increasing number of qubits. QEM techniques that are able to remove noise even when the circuit depth and number of qubits increases require exponential overhead, which is not practical for quantum computers in the future. To keep the promised exponential speedup of a future quantum computer we can not spend exponential time on QEM. Better QEM techniques are still being found by researchers, PEC might not be the best example now because better QEM techniques have been found. One of them we will briefly mention is Tensor-network Error Mitigation (TEM), which has a quadratically smaller measurement overhead compared to PEC [12]. An important thing to mention is that quadratically smaller overhead does not remove the exponential overhead.

Independent of the QEM technique, there is a fundamental limit. This has been proven by Takagi and Quek among others, using a theoretical model for QEM [13][14]. This fundamental limit arises from the exponential overhead required, which quickly becomes impractical as circuit depth and the number of qubits grow. Additionally, the exponential number of work needed to mitigate errors, negates (asymptotically) the exponential speedup. Research in quantum computing is driven by a possible exponential speedup on breaking some encryption mainly. QEM techniques that scale linearly in work required for some circuit depth and number of qubits are not able to remove all of the noise. TREX for example can only remove single qubit readout bias assuming this bias does not change between measurements.

This suggests that future quantum computer will not use QEM techniques requiring exponential overhead to completely remove bias, but might use a simple technique like TREX to remove single measurement bias. Future quantum devices will have to rely on error correction instead.

B. Error mitigation vs error correction

In the future, noise will probably not be removed using QEM, but its effect will have to be undone by Quantum Error Correction (QEC). The main difference between QEM and QEC is that QEM prevents noise from affecting the quantum state, either directly or in the long run by combining results of computations. QEC corrects errors after they have been made, like by using redundancy as seen in bit flip and phase flip codes.

C. Future research

Nonetheless, while QEC has not caught up, the relevance of error mitigation stays high. In future research it would therefore prove useful to improve on QEM data by simulating more by allowing for more shots. In addition, a more accurate correlation between qubit amount and error mitigation can be found for the mitigation techniques by allowing for more depth, simulation time and stronger computational power. Another variation can then also be to try different noise models instead of IBM-Torino, or using different (more advanced) functions for ZNE.

V. CONCLUSION

This paper studied the quantum error mitigation techniques ZNE, PEC and TREX to find the best trade-off between mitigation accuracy and sampling overhead. From this a pipeline was made that adapts the mitigation technique to circuit properties.

Our results show that ZNE is effective at moderate depths with an overhead that scales linearly. However this method breaks down beyond a cutoff depth that increases with qubit count, which can be estimated and extrapolated, enabling automated decisions. PEC showed the lowest bias when the noise model is accurate, but has an exponential sampling cost, making it not very practical past small, shallow circuits. Lastly, TREX has a low overhead and acts as a baseline mitigation that never becomes harmful. It cannot remove all noise, but remains robust at large depth/qubit count. The switching pipeline therefore uses PEC combined with TREX for low depth/qubit circuits. Then switches to ZNE + TREX, for circuits with higher than 2 qubits or large depth. Finally at the ZNE cutoff, only TREX stays viable.

Looking into the practical future of the QEM landscape we see that exponential noise growth is unavoidable as exponential overhead in strong QEM is fundamental. This theoretical limit overlaps with our findings and are reflected in the pipeline. However, while quantum error correction lacks the proper tools to reach full potential, QEM stays relevant. Therefore it would prove useful to expand on QEM research by making more simulations with more depth, simulation time and computational power and trying different noise models with different QEM techniques.

VI. APPENDIX

A. ZNE Cutoff Evaluation Graphs

Below are the graphs for the effectiveness of ZNE across different depths and qubit numbers

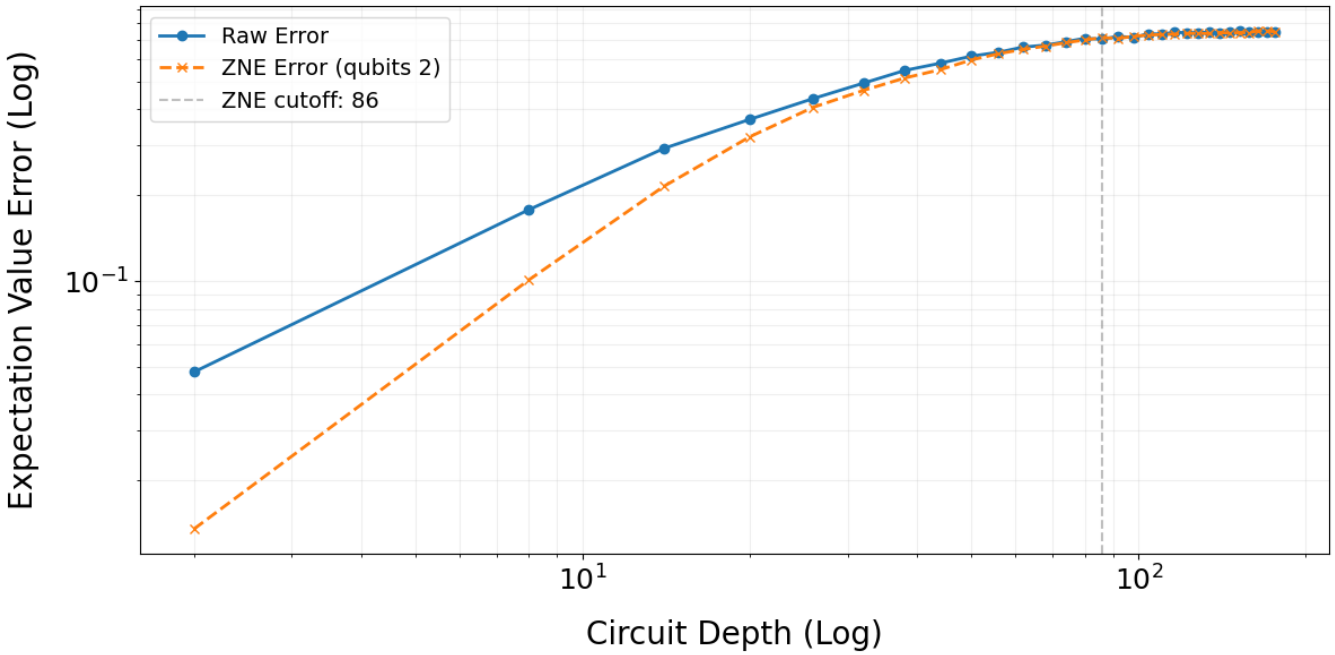


FIG. 11: The expectation value error of a 2-qubit circuit for increasing depths with a cutoff at 86.

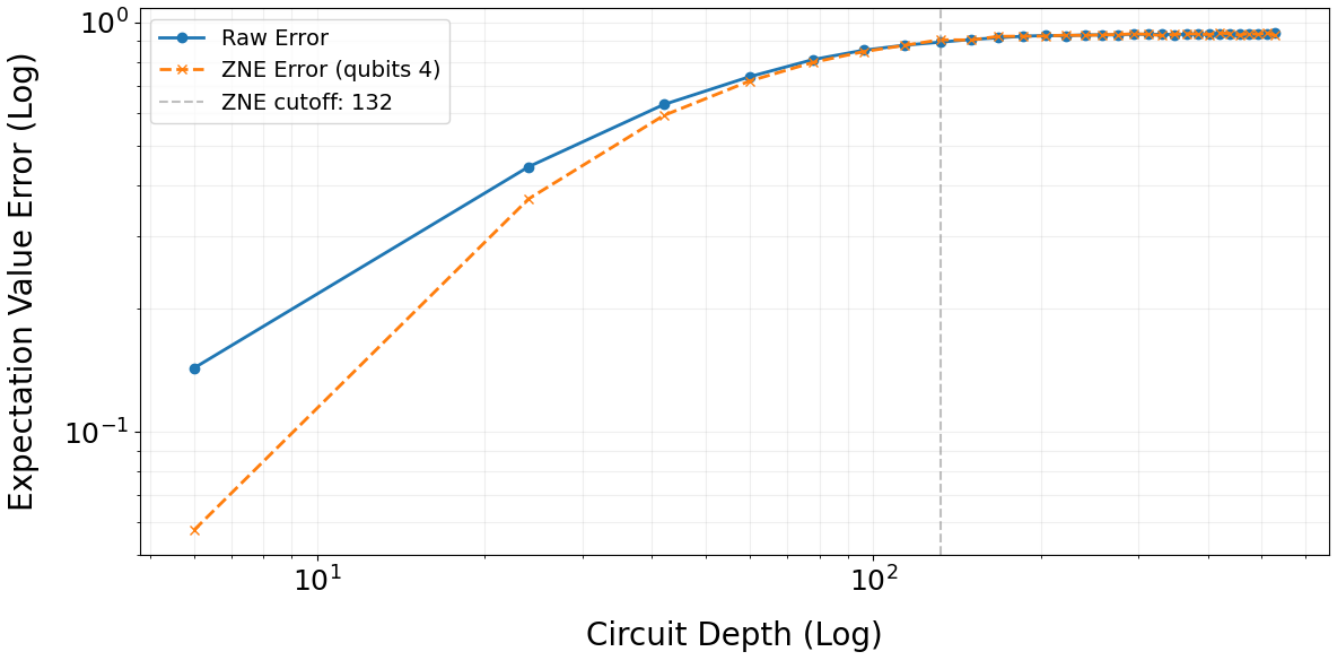


FIG. 12: The expectation value error of a 4-qubit circuit for increasing depths with a cutoff at 132.

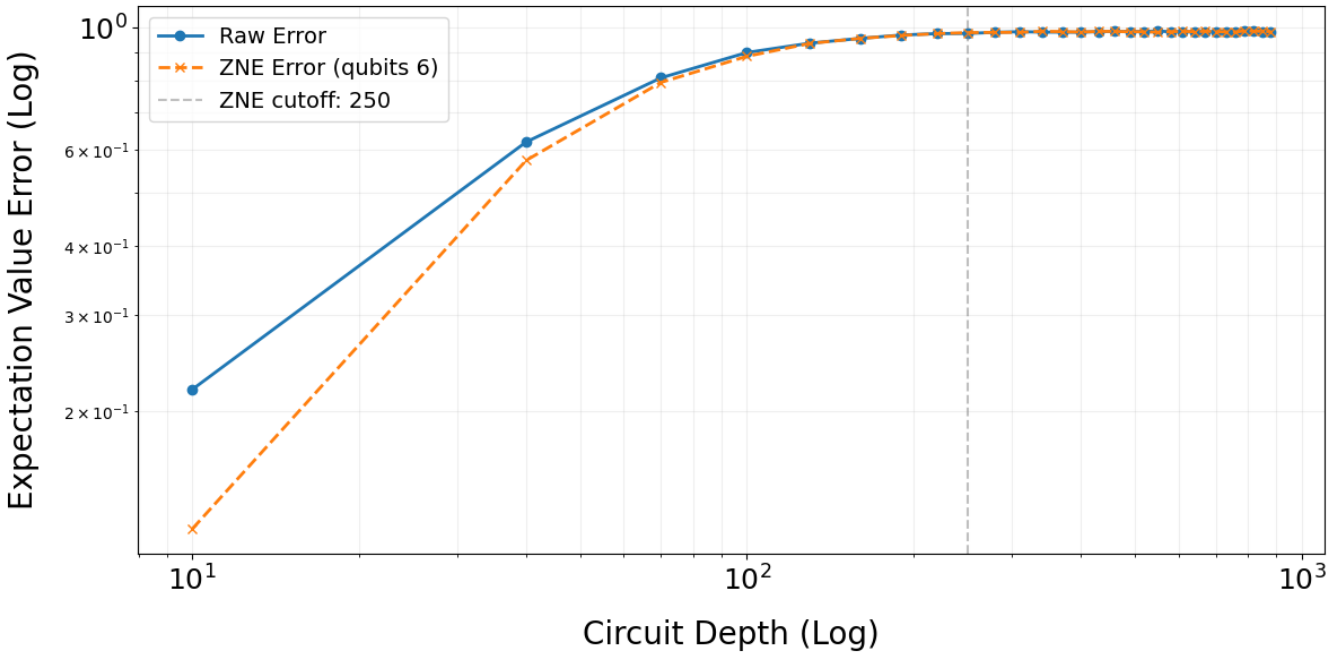


FIG. 13: The expectation value error of a 6-qubit circuit for increasing depths with a cutoff at 250.

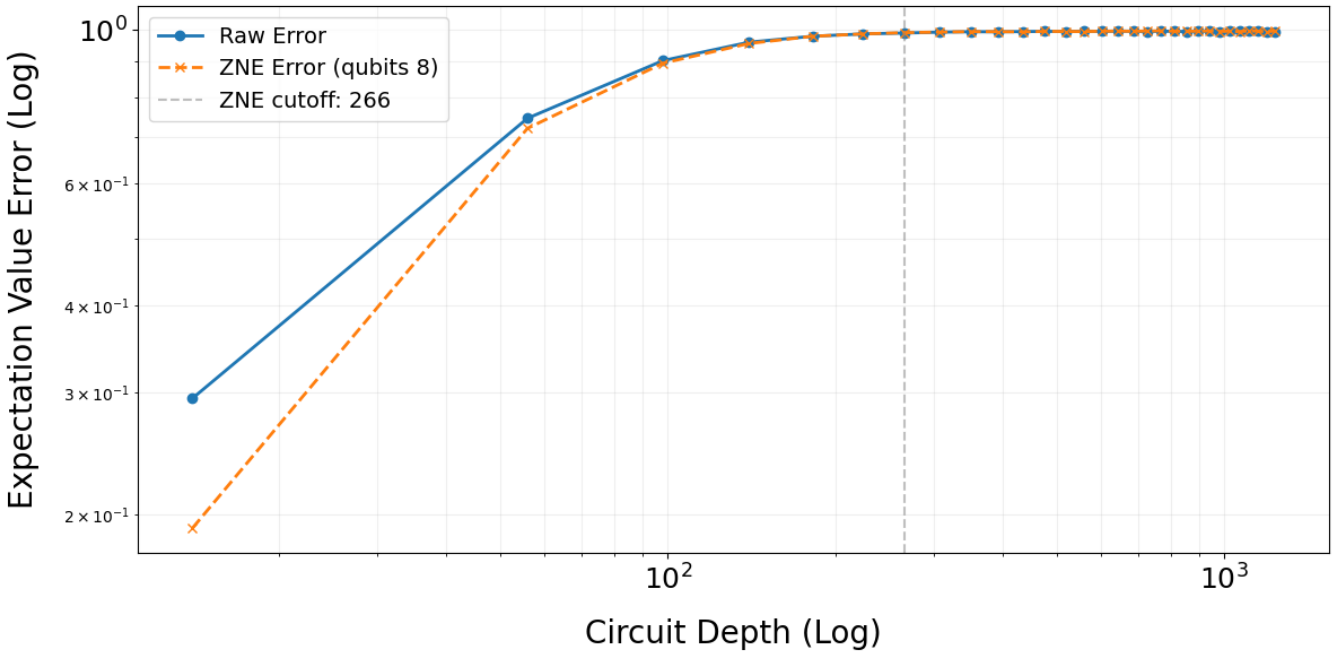


FIG. 14: The expectation value error of a 8-qubit circuit for increasing depths with a cutoff at 266.

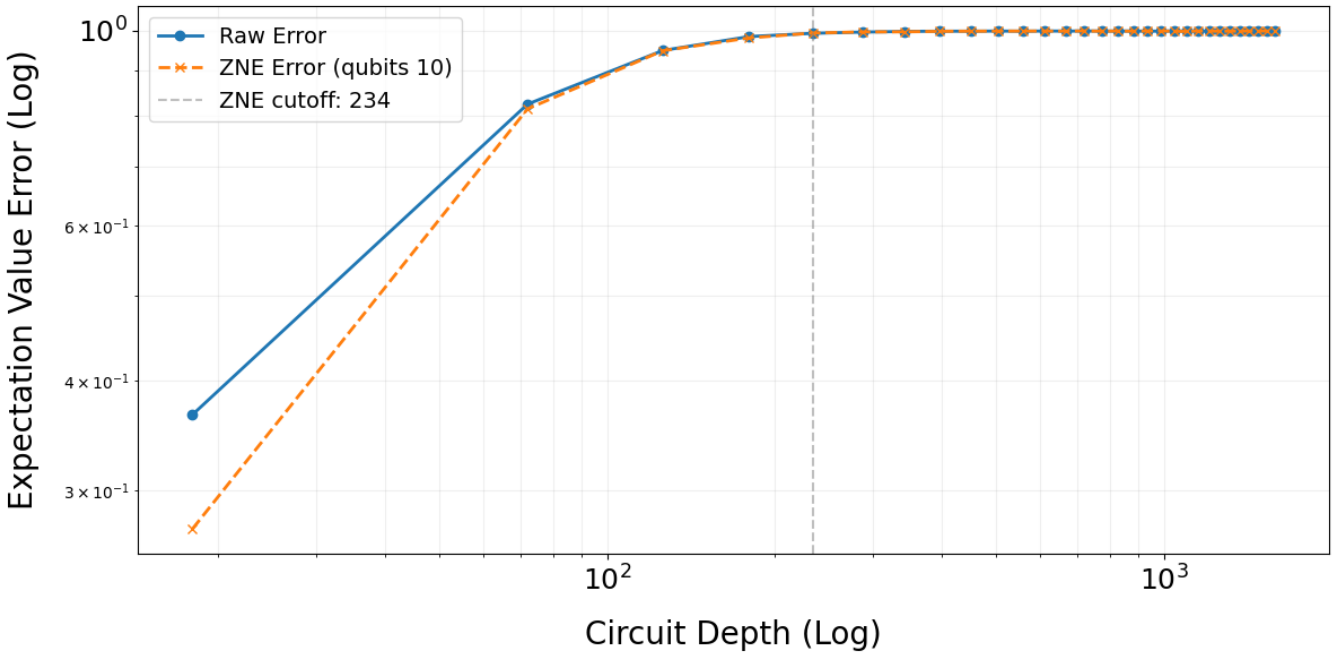


FIG. 15: The expectation value error of a 10-qubit circuit for increasing depths with a cutoff at 234.

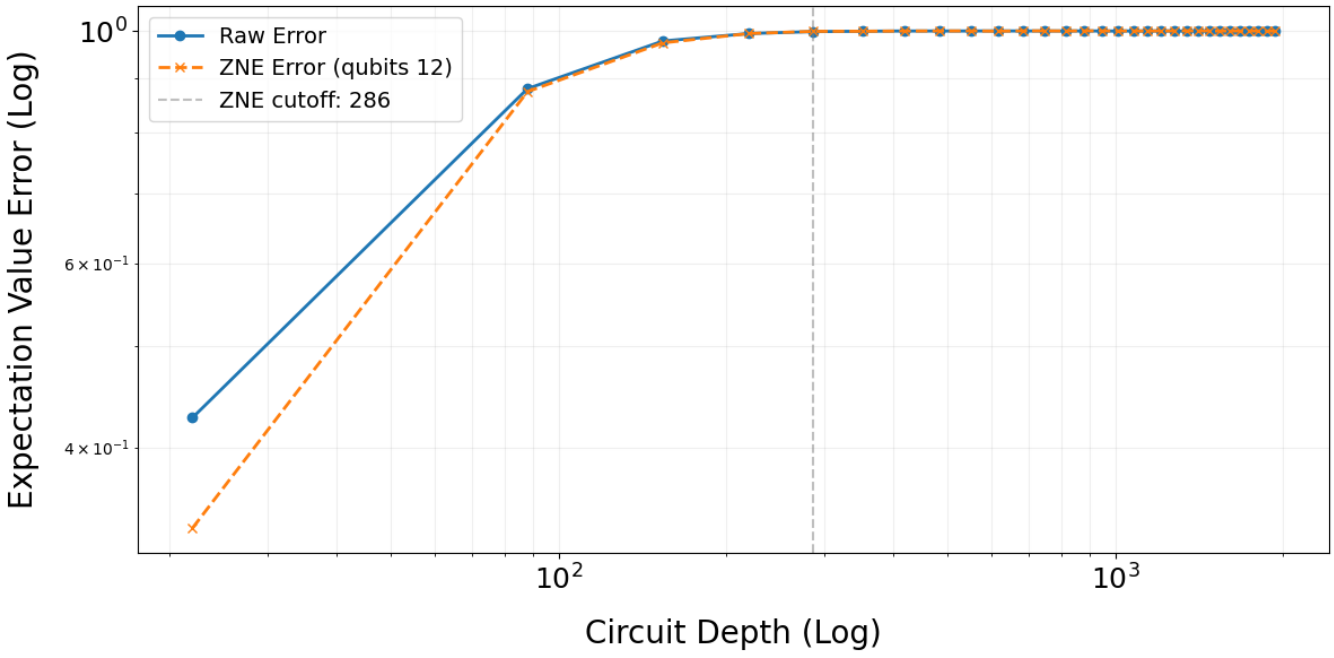


FIG. 16: The expectation value error of a 12-qubit circuit for increasing depths with a cutoff at 286.

B. ZNE Cutoff Depth Script

Below is the Python script used for determining the cutoff depth for ZNE.

Listing 1: Data Processing Script

```
import matplotlib.pyplot as plt
import numpy as np
from mitiq import zne
from mitiq.zne.inference import LinearFactory
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator
from qiskit_aer.noise import NoiseModel, depolarizing_error
from qiskit_ibm_runtime import QiskitRuntimeService
import json

service = QiskitRuntimeService()
backend = service.backend("ibm_torino")

# Get IBM hardware properties
props = backend.properties()

run_information = {}

# Extract average 1-qubit gate error
errors_1q = []
for q in range(backend.num_qubits):
    try:
        props = backend.target['sx'][(q,)]
        if props.error is not None:
            errors_1q.append(props.error)
    except KeyError:
        continue

avg_1q_error = np.mean(errors_1q)

# Extract average 2-qubit (CNOT) gate error
props = backend.properties()

# Extract 2-qubit errors by searching for gates with 2 qubits in their description
errors_2q = []

for gate in props.gates:
    # We only want gates that involve exactly two qubits
    if len(gate.qubits) == 2:
        # Pull the 'gate_error' parameter from the gate's parameter list
        for param in gate.parameters:
            if param.name == 'gate_error':
                errors_2q.append(param.value)


if errors_2q:
    avg_2q_error = np.mean(errors_2q)
    print(f"Found {len(errors_2q)} two-qubit gate calibrations.")
else:
    # If it's still empty use a default value
    print("Warning: Could not find 2-qubit gate errors. Falling back to 0.01.")
    avg_2q_error = 0.01

print(f"Real p_1q (Avg): {avg_1q_error:.5f}")
print(f"Real p_2q (Avg): {avg_2q_error:.5f}")
run_information["1q_error"] = avg_1q_error
run_information["2q_error"] = avg_2q_error

# Setup Noisy Simulator
noise_model = NoiseModel()
noise_model.add_all_qubit_quantum_error(depolarizing_error(avg_1q_error, 1), ['h'])
noise_model.add_all_qubit_quantum_error(depolarizing_error(avg_2q_error, 2), ['cx'])
backend = AerSimulator(noise_model=noise_model)
```

```
max_num_qubits = 12
num_qubits = 2

def executor(circuit: QuantumCircuit) -> float:
    meas_circ = circuit.copy()
    meas_circ.measure_all()

    job = backend.run(meas_circ, shots=10000)  # Higher shots for stability
    counts = job.result().get_counts()

    # Calculate expectation value of the |00> state
    return counts.get('0' * num_qubits, 0) / 10000


# Run Experiment
qubit_nrs = range(num_qubits, max_num_qubits + 1, 2)
depths = range(1, 91, 3)
unmitigated_errors = []
all_zne_errors = []
cutoff_depths = {}

for num_qubits in qubit_nrs:
    zne_errors = []
    unmitigated_errors = []
    actual_depths = []
    for d in depths:
        # A simple Bell-state mirror circuit
        print(f"Num qubits: {num_qubits}, d: {d}")
        c = QuantumCircuit(num_qubits)
        for _ in range(d):
            for k in range(num_qubits - 1):
                c.cx(k, k + 1)
            c.barrier()

        # Add the inverse to make the ideal result
        c.compose(c.inverse(), inplace=True)
        actual_depths.append(c.depth())

        # Unmitigated
        val_raw = executor(c)
        unmitigated_errors.append(abs(1.0 - val_raw))

        # ZNE mitigation (Linear)
        fac = LinearFactory(scale_factors=[1.0, 3.0, 5.0]) # Industry standard
        val_zne = zne.execute_with_zne(c, executor, factory=fac)

        zne_errors.append(abs(1.0 - val_zne))


    run_information[f"{num_qubits}_depth"] = actual_depths
    run_information[f"{num_qubits}_unmitigated"] = unmitigated_errors
    run_information[f"{num_qubits}_zne"] = zne_errors

    # Plotting the errors
    plt.figure(figsize=(12, 6))
    plt.plot(actual_depths, unmitigated_errors, 'o-', label='Raw Error', linewidth=2)
    plt.plot(actual_depths, zne_errors, 'x--', label=f'ZNE Error (qubits {num_qubits})',
        linewidth=2)

    legend_added = False
    for i in range(len(unmitigated_errors)):
        if unmitigated_errors[i] <= zne_errors[i]:
            print(f"{num_qubits} qubit cutoff: {actual_depths[i]}")
            cutoff_depths[num_qubits] = actual_depths[i]

            label_text = f'ZNE cutoff: {actual_depths[i]}' if not legend_added else ""
            plt.axvline(x=actual_depths[i], color='gray', linestyle='--', alpha=0.5,
                label=label_text)
            legend_added = True
            break
```

```
    plt.ylabel("Expectation Value Error", fontsize=20, labelpad=15)
    plt.xlabel("Circuit Depth", fontsize=20, labelpad=15)

    plt.tick_params(axis='both', which='major', labelsize=18)

    plt.legend(fontsize=14, loc='best')

    plt.tight_layout()
    plt.savefig(f"ZNE {num_qubits}q.png")
    plt.show()

print(unmitigated_errors)
run_information["cutoff_depths"] = cutoff_depths

x = np.array(list(cutoff_depths.keys()))
y = np.array(list(cutoff_depths.values()))

coefficients = np.polyfit(x, y, 1)
polynomial = np.poly1d(coefficients)

line_of_best_fit = polynomial(x)
run_information["polynomial"] = {}
run_information["polynomial"]["m"] = polynomial[0]
run_information["polynomial"]["c"] = polynomial[1]

with open("run_information.json", "w") as f:
    json.dump(run_information, f, indent=4)

print(f"Curve of best fit: {polynomial}")

plt.scatter(x, y, color='blue', label='Data Points')
plt.plot(x, line_of_best_fit, color='red', label='Line of Best Fit')

plt.xlabel('Number of Qubits')
plt.ylabel('Cutoff Depth')
plt.legend()
plt.savefig("Cutoff depth extrapolation.png")
plt.show()
```

C. PEC simulation script

Below is the Python script used for the experiments of PEC on a simulator.

Listing 2: PEC simulation script

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from qiskit.circuit.library import RealAmplitudes
from qiskit.quantum_info import Statevector, Operator, Pauli
from qiskit import transpile

from qiskit_aer import Aer
from qiskit_aer.noise import NoiseModel, depolarizing_error, phase_damping_error


from mitiq import pec, zne
from mitiq import Executor
from mitiq.pec.representations.depolarizing import
    represent_operations_in_circuit_with_local_depolarizing_noise
from mitiq.zne.scaling import fold_global
from mitiq.zne.inference import PolyFactory

SEED = 7
n_qubits = 2
SHOTS = 10000
```

```
depths = list(range(1, 4))
PEC_NUM_SAMPLES = 1000
P_TRUE = 0.01
MISMATCH = False
TRUE_NOISE_MODEL = False

# Function to execute noisy circuit and return expectation value
def execute_noisy_expectation_true(qc) -> float:
        meas = qc.copy()
        meas.measure_all()
        tqc = transpile(meas, aer_backend, optimization_level=0)
        job = aer_backend.run(tqc, shots=SHOTS)
        counts = job.result().get_counts()

        ev = 0
        for bitstr, c in counts.items():
                b = bitstr.replace(" ", "")[-2:]
                if b in ("00", "11"):
                        ev += c
                elif b in ("01", "10"):
                        ev -= c

        return float(ev / sum(counts.values()))

# Create the circuits
ZZ = Operator(Pauli("ZZ"))

circuits = []
for reps in depths:
        qc = RealAmplitudes(num_qubits=n_qubits, reps=reps, entanglement="full")
        params = np.full(qc.num_parameters, 0.1)
        qc = qc.assign_parameters(params, inplace=False)
        circuits.append(qc)

# Create the noise model of the backend and the backend
p_assumed = P_TRUE

# Mismatch scenario
if MISMATCH:
        p_assumed = P_TRUE * 1.5

nm = NoiseModel()
dep1 = depolarizing_error(P_TRUE, 1)
dep2 = depolarizing_error(P_TRUE, 2)

# True noise model with phase damping scenario
if TRUE_NOISE_MODEL:
        ph = phase_damping_error(2*P_TRUE)
        for g in ["rx", "ry", "rz", "x", "y", "z", "h", "sx", "id"]:
                nm.add_all_qubit_quantum_error
            (dep1.compose(ph), g)

        for g in ["cx", "cz", "ecr"]:
                nm.add_all_qubit_quantum_error
            (dep2, g)

# True noise model with depolarizing only scenario
else:
        for g in ["rx", "ry", "rz", "x", "y", "z", "h", "sx", "id"]:
                nm.add_all_qubit_quantum_error
            (dep1, g)

        for g in ["cx", "cz", "ecr"]:
                nm.add_all_qubit_quantum_error
            (dep2, g)

true_nm = nm
aer_backend = Aer.get_backend("aer_simulator")
aer_backend.set_options(noise_model=true_nm)
```

```
# Run the experiments
ideal_vals = []
noisy_vals = []
zne_vals = []
pec_vals = []
pec_errbars = []

noise_factors = [1, 3, 5]
factory = PolyFactory(scale_factors=noise_factors, order=2)

for reps, qc in zip(depths, circuits):
        print("The rep we are at:", reps)

        # Get the ideal values
        ideal=float(np.real(Statevector
        .from_instruction(qc)
        .expectation_value(ZZ)))
        ideal_vals.append(ideal)

        #Get the noisy values
        noisy = float(execute_noisy_expectation_true
        (qc.copy()))
        noisy_vals.append(noisy)

        # PEC representations
        reps_list =
        represent_operations_in_circuit_with
        _local_depolarizing_noise(qc, p_assumed)

        executor = Executor
        (execute_noisy_expectation_true)

        # Get ZNE values
        zne_val = zne.execute_with_zne(
                qc,
                executor,
                factory=factory,
                scale_noise=fold_global,
        )
        zne_vals.append(float(zne_val))

        # Get PEC values
        pec_value, pec_data = pec.execute_with_pec(
                qc,
                executor,
                observable=None,
                representations=reps_list,
                num_samples=PEC_NUM_SAMPLES,
                random_state=SEED + reps,
                full_output=True
        )
        pec_vals.append(float(pec_value))
        pec_errbars
        .append(float(pec_data["pec_error"]))

# Save the results
ideal_vals = np.array(ideal_vals)
noisy_vals = np.array(noisy_vals)
pec_vals = np.array(pec_vals)
pec_errbars = np.array(pec_errbars)
zne_vals = np.array(zne_vals)

noisy_abs_err = np.abs(noisy_vals - ideal_vals)
pec_abs_err = np.abs(pec_vals - ideal_vals)
zne_abs_err = np.abs(zne_vals - ideal_vals)



rows = []
for reps, ideal, noisy, pec_v, pec_e, zne_v in zip(depths, ideal_vals, noisy_vals, pec_vals,
```

```
    pec_errbars, zne_vals):
    rows.append({
        "reps": reps,
        "ideal_<ZZ>": ideal,
        "noisy_<ZZ>": noisy,
        "pec_<ZZ>": pec_v,
        "abs_err_noisy": abs(noisy - ideal),
        "abs_err_pec": abs(pec_v - ideal),
        "improvement": abs(noisy - ideal) - abs(pec_v - ideal),
        "pec_errorbar": pec_e,
        "zne_<ZZ>": zne_v,
        "abs_err_zne": abs(zne_v - ideal),
        "zne_improvement_over_noisy": abs(noisy - ideal) - abs(zne_v - ideal),
    })

df = pd.DataFrame(rows)
pd.set_option("display.max_columns", None)
print(df.to_string(index=False))

df.to_csv("pec_depth_table.csv", index=False)

# Plot the results
plt.figure()
plt.plot(depths, noisy_abs_err, marker="o", label="No mitigation")
plt.plot(depths, zne_abs_err,  marker="o", label="ZNE")
plt.plot(depths, pec_abs_err, marker="o", label="PEC")
plt.yscale("log")
plt.xlabel("Circuit depth")
plt.ylabel("Absolute error")
plt.title("PEC accuracy vs depth")
plt.legend()
plt.show()

plt.figure()
plt.plot(depths, ideal_vals, marker="o", label="Ideal")
plt.plot(depths, noisy_vals, marker="o", label="Noisy")
plt.plot(depths, zne_vals,  marker="o", label="ZNE")
plt.errorbar(depths, pec_vals, yerr=pec_errbars, marker="o", capsize=3, label="PEC ± pec_error")
plt.xlabel("reps")
plt.ylabel("<ZZ>")
plt.title("Expectation values vs depth")
plt.legend()
plt.show()
```

D. Quantum Error Mitigation Pipeline Color Map Demonstration

Below is the Python script used for creating the color map displaying the error mitigation methods used in different circuit configurations.

Listing 3: PEC simulation script

```
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.colors import ListedColormap
from qiskit import QuantumCircuit
from qiskit.quantum_info import SparsePauliOp
from qiskit_ibm_runtime import EstimatorOptions, EstimatorV2, QiskitRuntimeService
from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manager
import gc
from matplotlib.colors import ListedColormap
import matplotlib.ticker as mticker

# Create the ZNE cutoff polynomial
a = 10.83  # Slope
b = 99.2  # Intercept
ZNE_line_model = np.poly1d([a, b])
```

```
def get_average_2q_error(backend):
    """
    Safely retrieves the mean 2-qubit gate error from the backend target.
    """
    # Identify which entangling gate the backend actually uses
    # Most common are 'cx' or 'ecr'
    available_ops = backend.operation_names

    if 'cx' in available_ops:
        gate_name = 'cx'
    elif 'ecr' in available_ops:
        gate_name = 'ecr'
    elif 'cz' in available_ops:
        gate_name = 'cz'
    else:
        print("Warning: No standard 2-qubit gate found. Defaulting to 1% error.")
        return 0.01

    # Extract the properties from the target
    try:
        instruction_props = backend.target[gate_name]

        # Get errors for all qubit combinations where this gate is defined
        errors = [
            props.error for props in instruction_props.values()
            if props is not None and hasattr(props, 'error') and props.error is not None
        ]

        if not errors:
            return 0.01

        return sum(errors) / len(errors)

    except KeyError:
        return 0.01


def estimate_pec_overhead_dynamic(circuit, backend):
    avg_error = get_average_2q_error(backend)

    ops = circuit.count_ops()
    # Count all possible 2-qubit gates
    total_heavy_gates = ops.get('cx', 0) + ops.get('ecr', 0) + ops.get('cz', 0)

    # Gamma calculation based on actual hardware noise
    gamma_gate = 1 + (2 * avg_error)
    total_overhead = gamma_gate ** (2 * total_heavy_gates)

    return total_overhead, avg_error

def run_smart_mitigation_pipeline(circuit, backend, seed=42):
    pm = generate_preset_pass_manager(optimization_level=3, backend=backend)
    isa_circuit = pm.run(circuit)

    # Get dynamic hardware stats
    overhead_est, live_error_rate = estimate_pec_overhead_dynamic(isa_circuit, backend)
    depth = isa_circuit.depth()

    print(f"--- Hardware-Aware Analysis ---")
    print(f"Avg 2Q Error: {live_error_rate:.4f}")
    print(f"Estimated PEC Overhead: {overhead_est:.2f}x")

    ZNE_cutoff_depth = ZNE_line_model(circuit.num_qubits)

    options = EstimatorOptions()

    if overhead_est < 5:
        print("Selection: PEC (Explicitly enabled)")
        # In V2, we enable PEC like this:
        options.resilience.pec_mitigation = True
```

```
        options.resilience.pec.max_overhead = 100.0
        selection = 3
    elif depth < ZNE_cutoff_depth:
        print("Selection: Custom ZNE (Level 2)")
        options.resilience_level = 2
        options.resilience.zne_mitigation = True
        options.resilience.zne.extrapolator = "polynomial_degree_2"
        options.resilience.zne.noise_factors = [1, 3, 5]
        selection = 2
    else:
        print("Selection: TREX (Default Level 1)")
        options.resilience_level = 1
        selection = 1

    return pm, isa_circuit, options, selection


# --- EXECUTION ---

# Setup Service and Backend
try:
    service = QiskitRuntimeService()
    backend = service.backend("ibm_torino")
except:
    print("No IBM account found. Please ensure you have credentials saved.")
    backend = None

size_depth = 30
size_qubits = 30

algorithms_used = np.zeros((size_qubits, size_depth), dtype=int)

isa_depths = []

if backend:
    for i in range(1, size_qubits+1):
        for j in range(1, size_depth+1):
            # Create Dummy Circuit
            num_qubits = i
            depth_layers = j

            print("Qubits: ", num_qubits)
            print("Depth: ", depth_layers)
            qc = QuantumCircuit(num_qubits)

            # Create a dense circuit of entangling gates
            for _ in range(depth_layers):
                for k in range(num_qubits - 1):
                    qc.cx(k, k + 1)
                qc.barrier()  # Optional: helps visualize the layers

            # Define an observable matching the qubit count
            observable = SparsePauliOp("Z" * num_qubits)

            # Process Pipeline
            pm, isa_circuit, options, selection = run_smart_mitigation_pipeline(qc, backend)
            isa_observable = observable.apply_layout(isa_circuit.layout)
            isa_depths.append(isa_circuit.depth())

            algorithms_used[i-1, j-1] = selection

            # Run Job
            estimator = EstimatorV2(mode=backend, options=options)
            print(f"Submitting job to {backend.name}...")

            del qc
            del isa_circuit
            del pm
            gc.collect()
else:
```

```
    print("Backend not accessible.")

algorithms_used = np.array(algorithms_used)

fig, ax = plt.subplots(figsize=(16, 14))

# Custom Discrete Colormap
cmap = ListedColormap(['#e74c3c', '#3498db', '#2ecc71'])

# Create Heatmap
im = ax.imshow(algorithms_used,
               origin='lower',
               cmap=cmap,
               aspect='auto',
               extent=[0.5, size_depth + 0.5, 0.5, size_qubits + 0.5])

# Grid and Axis Formatting
ax.xaxis.set_major_locator(mticker.MaxNLocator(integer=True))
ax.yaxis.set_major_locator(mticker.MaxNLocator(integer=True))

# Set minor ticks for separators
ax.set_xticks(np.arange(0.5, size_depth + 1.5, 1), minor=True)
ax.set_yticks(np.arange(0.5, size_qubits + 1.5, 1), minor=True)

# Grid styling
ax.grid(visible=True, which='minor', color='white', linestyle='-', linewidth=1)
ax.grid(visible=False, which='major')

# Text Formatting
ax.set_xlabel("Circuit Depth (Layers)", fontsize=20, labelpad=15)
ax.set_ylabel("Number of Qubits", fontsize=20, labelpad=15)

# Increase size of the numbers on the axes
ax.tick_params(axis='both', which='major', labelsize=18)

# Creating the colorbar
cbar = plt.colorbar(im, ticks=[1, 2, 3], shrink=0.7)
cbar.ax.set_yticklabels(['TREX', 'ZNE', 'PEC'],
                        fontsize=20)
cbar.set_label('Mitigation Method', fontsize=18, labelpad=15)

plt.tight_layout()

plt.savefig("Error Mitigation Pipeline Colour Map.png")
plt.show()
```

E. Quantum Error Mitigation Pipeline Color Map Demonstration

Below is the Python script used for choosing the quantum error mitigation method for a given circuit.

Listing 4: PEC simulation script

```
import numpy as np
from qiskit import QuantumCircuit
from qiskit.quantum_info import SparsePauliOp
from qiskit_ibm_runtime import EstimatorOptions, EstimatorV2, QiskitRuntimeService
from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manager

# Create the ZNE cutoff polynomial
a = 10.83  # Slope
b = 99.2  # Intercept
ZNE_line_model = np.poly1d([a, b])


def get_average_2q_error(backend):
    """
    Safely retrieves the mean 2-qubit gate error from the backend target.
    """
```

```
    # Identify which entangling gate the backend actually uses
    # Most common are 'cx' or 'ecr'
    available_ops = backend.operation_names

    if 'cx' in available_ops:
        gate_name = 'cx'
    elif 'ecr' in available_ops:
        gate_name = 'ecr'
    elif 'cz' in available_ops:
        gate_name = 'cz'
    else:
        print("Warning: No standard 2-qubit gate found. Defaulting to 1% error.")
        return 0.01

    # Extract properties from target
    try:
        instruction_props = backend.target[gate_name]

        # Get errors for all qubit combinations where this gate is defined
        errors = [
            props.error for props in instruction_props.values()
            if props is not None and hasattr(props, 'error') and props.error is not None
        ]

        if not errors:
            return 0.01

        return sum(errors) / len(errors)

    except KeyError:
        return 0.01


def estimate_pec_overhead_dynamic(circuit, backend):
    avg_error = get_average_2q_error(backend)

    ops = circuit.count_ops()
    # Count all possible 2-qubit gates
    total_heavy_gates = ops.get('cx', 0) + ops.get('ecr', 0) + ops.get('cz', 0)

    # Gamma calculation based on actual hardware noise
    gamma_gate = 1 + (2 * avg_error)
    total_overhead = gamma_gate ** (2 * total_heavy_gates)

    return total_overhead, avg_error

def run_smart_mitigation_pipeline(circuit, backend, seed=42):
    pm = generate_preset_pass_manager(optimization_level=3, backend=backend)
    isa_circuit = pm.run(circuit)

    # Get dynamic hardware stats
    overhead_est, live_error_rate = estimate_pec_overhead_dynamic(isa_circuit, backend)
    depth = isa_circuit.depth()

    print(f"--- Hardware-Aware Analysis ---")
    print(f"Avg 2Q Error: {live_error_rate:.4f}")
    print(f"Estimated PEC Overhead: {overhead_est:.2f}x")

    ZNE_cutoff_depth = ZNE_line_model(circuit.num_qubits)

    options = EstimatorOptions()

    if overhead_est < 5:
        print("Selection: PEC (Explicitly enabled)")
        options.resilience.pec_mitigation = True
        options.resilience.pec.max_overhead = 100.0
    elif depth < ZNE_cutoff_depth:
        print("Selection: Custom ZNE (Level 2)")
        options.resilience_level = 2
        options.resilience.zne_mitigation = True
```

```
        options.resilience.zne.extrapolator = "polynomial_degree_2"
        options.resilience.zne.noise_factors = [1, 3, 5]
    else:
        print("Selection: TREX (Default Level 1)")
        options.resilience_level = 1

    return pm, isa_circuit, options


# --- EXECUTION ---

# Setup Service and Backend
try:
    service = QiskitRuntimeService()
    backend = service.backend("ibm_torino")
except:
    print("No IBM account found. Please ensure you have credentials saved.")
    backend = None

if backend:
    # Create Dummy Circuit
    # Dummy Circuit Start ----------------------------------------------------------------
    # Variables of the circuit to change
    num_qubits = 2
    depth_layers = 2

    print("Qubits: ", num_qubits)
    print("Depth: ", depth_layers)
    qc = QuantumCircuit(num_qubits)

    # Create a dense circuit of entangling gates
    for _ in range(depth_layers):
        for k in range(num_qubits - 1):
            qc.cx(k, k + 1)
        qc.barrier()  # Optional: helps visualize the layers

    # Define an observable matching the qubit count
    observable = SparsePauliOp("Z" * num_qubits)

    # Dummy Circuit End ------------------------------------------------------------------

    # Process Pipeline
    pm, isa_circuit, options = run_smart_mitigation_pipeline(qc, backend)
    isa_observable = observable.apply_layout(isa_circuit.layout)

    # Run Job
    estimator = EstimatorV2(mode=backend, options=options)
    print(f"Submitting job to {backend.name}...")

    # Run the circuit
    job = estimator.run([(isa_circuit, isa_observable)])
    result = job.result()

    # You can use the result form here on...
else:
    print("Backend not accessible.")
```


[1] A. Javadi-Abhari, M. Treinish, K. Krsulich, C. J. Wood, J. Lishman, J. Gacon, S. Martiel, P. D. Nation, L. S. Bishop, A. W. Cross, B. R. Johnson, and J. M. Gambetta, Quantum computing with Qiskit (2024), arXiv:2405.08810 [quant-ph].

[2] K. Temme, S. Bravyi, and J. M. Gambetta, Error mitigation for short-depth quantum circuits, Physical Review Letters **119**, 180509 (2017).

[3] Mitiq Project, What is the theory behind zne?, `https://mitiq.readthedocs.io/en/stable/guide/zne-5-theory.html` (2025), accessed: January 14, 2026.

[4] Y. Li and S. C. Benjamin, Efficient variational quantum simulator incorporating active error minimisation, Physical Review X **7**, 10.1103/PhysRevX.7.021050 (2017).

[5] A. Kandala, K. Temme, A. D. Córcoles, A. Mezzacapo, J. M. Chow, and J. M. Gambetta, Error mitigation extends the computational reach of a noisy quantum processor, Nature **567**, 491 (2019).
[6] T. Giurgica-Tiron, Y. Hindy, R. LaRose, A. Mari, and W. J. Zeng, Digital zero noise extrapolation for quantum error mitigation, in *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)* (IEEE, Denver, CO, USA, 2020) pp. 306–316.
[7] S. Endo, S. C. Benjamin, and Y. Li, Practical quantum error mitigation for near-future applications, Physical Review X **8**, 031027 (2018).
[8] H. Pashayan, J. J. Wallman, and S. D. Bartlett, Estimating outcome probabilities of quantum circuits using quasiprobabilities, Physical Review Letters **115**, 070501 (2015).
[9] E. van den Berg, Z. K. Minev, and K. Temme, Model-free readout-error mitigation for quantum expectation values, Physical Review A **105**, 032620 (2022).
[10] IBM Quantum, Error mitigation and suppression techniques, Online documentation (n.d.), retrieved January 19, 2026, from `https://quantum.cloud.ibm.com/docs/en/guides/error-mitigation-and-suppression-techniques`.
[11] IBM Quantum, Resilienceoptionsv2 — qiskit ibm runtime api reference, `https://quantum.cloud.ibm.com/docs/en/api/qiskit-ibm-runtime/options-resilience-options-v2` (2026), accessed January 18, 2026.
[12] S. Filippov, M. Leahy, M. A. Rossi, and G. García-Pérez, Scalable tensor-network error mitigation for near-term quantum computing, arXiv preprint arXiv:2307.11740 (2023).
[13] R. Takagi, S. Endo, S. Minagawa, and M. Gu, Fundamental limits of quantum error mitigation, npj Quantum Information **8**, 114 (2022).
[14] Y. Quek, D. Stilck França, S. Khatri, J. J. Meyer, and J. Eisert, Exponentially tighter bounds on limitations of quantum error mitigation, Nature Physics **20**, 1648 (2024).