Mitiq: A software package for error mitigation on noisy quantum computers

Ryan LaRose,[1,2] Andrea Mari,[1] Sarah Kaiser,[1] Peter J. Karalekas,[1,*] Andre A. Alves,[3] Piotr Czarnik,[4] Mohamed El Mandouh,[5] Max H. Gordon,[6] Yousef Hindy,[7] Aaron Robertson,[8] Purva Thakre,[9] Nathan Shammah,[1] and William J. Zeng[1,7,10]

[1] *Unitary Fund*
[2] *Michigan State University, East Lansing, MI*
[3] *Hamburg University of Applied Sciences, Hamburg, Germany*
[4] *Theoretical Division, Los Alamos National Laboratory, Los Alamos, NM 87545, USA*
[5] *Institute for Quantum Computing, University of Waterloo, Waterloo, ON, N2L 3G1, Canada*
[6] *Instituto de Física Teórica, UAM/CSIC, Universidad Autónoma de Madrid, Madrid, Spain*
[7] *Stanford University, Palo Alto, CA*
[8] *Independent researcher*
[9] *Southern Illinois University, Carbondale, IL*
[10] *Goldman, Sachs & Co, New York, NY*

(Dated: August 13, 2021)

We introduce Mitiq, a Python package for error mitigation on noisy quantum computers. Error mitigation techniques can reduce the impact of noise on near-term quantum computers with minimal overhead in quantum resources by relying on a mixture of quantum sampling and classical post-processing techniques. Mitiq is an extensible toolkit of different error mitigation methods, including zero-noise extrapolation, probabilistic error cancellation, and Clifford data regression. The library is designed to be compatible with generic backends and interfaces with different quantum software frameworks. We describe Mitiq using code snippets to demonstrate usage and discuss features and contribution guidelines. We present several examples demonstrating error mitigation on IBM and Rigetti superconducting quantum processors as well as on noisy simulators.

I. INTRODUCTION

Methods to counteract noise are critical for realizing practical quantum computation. While fault-tolerant quantum computers that use error-correcting codes are an ideal goal, they require physical resources beyond current experimental capabilities. It is therefore interesting and important to develop other methods for dealing with noise on near-term quantum computers.

In recent years, several methods, collectively referred to as quantum error mitigation methods [1], have been proposed and developed for this task. Among them are zero-noise extrapolation [2, 3], probabilistic error cancellation [2, 4], Clifford data regression [5, 6], dynamical decoupling [7–9], randomized compiling [10], and subspace expansion [11]. Several error mitigation methods have also been tested experimentally [12–18]. To aid research, improve reproducibility, and move towards practical applications, it is important to have a unified framework for implementing error mitigation techniques on multiple quantum back-ends.

To these ends, we introduce Mitiq: a software package for error mitigation on noisy quantum computers. Mitiq is an open-source Python library that interfaces with multiple quantum programming front-ends to implement error mitigation techniques on various real and simulated quantum processors. Mitiq supports Cirq [19], Qiskit [20], pyQuil [21], and Braket [22] circuit types and any back-ends, real or simulated, that can execute them. The library is extensible in that new front-ends and back-ends can be easily supported as they become available. Mitiq currently implements zero-noise extrapolation (ZNE), probabilistic error cancellation (PEC), and Clifford data regression (CDR), and its modular design allows support for additional techniques, as shown in Figure 1. Error mitigation methods can be applied in a few additional lines of code but the library is still flexible enough for advanced usage.

In Sec. II, we show how to get started with Mitiq and illustrate its main usage. We then show experimental and numerical examples in Sec. III that demonstrate how error mitigation with Mitiq improves the performance of noisy quantum computations. In Sec. IV, we describe in detail the zero-noise extrapolation module. In Sec. V, we give an overview of the probabilistic error cancellation module. In Sec. VI, we present the Clifford data regression module. We discuss further software details and library information in Sec. VII including future development, contribution guidelines, and planned maintenance and support. Finally, in Sec. VIII we discuss the relationship between Mitiq and other techniques for dealing with errors in quantum computers.

II. GETTING STARTED WITH MITIQ

A. Requirements and installation

Mitiq is a Python library that can be installed on Mac, Windows, and Linux operating systems via PyPI by ex-

* Current address: AWS Center for Quantum Computing, Pasadena, CA 91125, USA

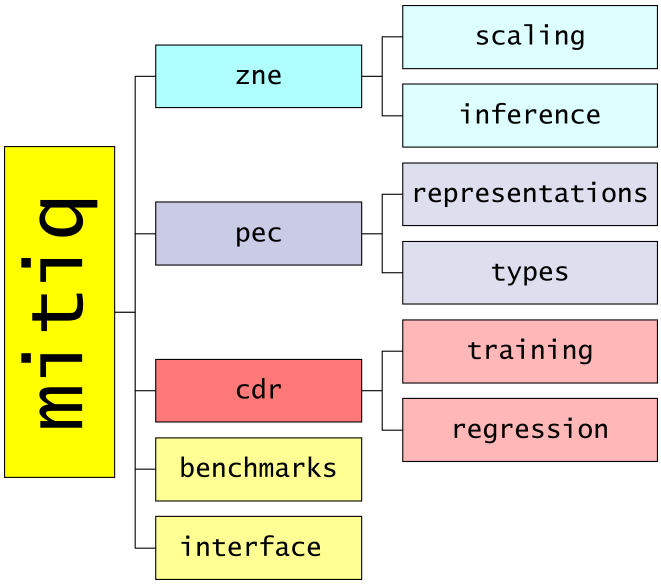

FIG. 1. The structure of Mitiq modules. Different error mitigation techniques are organized in different modules, including zero noise extrapolation (`zne`), probabilistic error cancellation (`pec`), and Clifford data regression (`cdr`). Other modules are dedicated to auxiliary tasks such as interfacing with different quantum software libraries (`interface`) and benchmarking error mitigation strategies (`benchmarks`).

Software Framework	Circuit Type
Cirq	`cirq.Circuit`
Qiskit	`qiskit.QuantumCircuit`
PyQuil	`pyquil.Program`
Braket	`braket.circuits.Circuit`

TABLE I. The quantum software frameworks compatible with Mitiq. Since Mitiq interacts with circuits but is not directly responsible for their execution, supporting a new circuit type requires only to define a few conversion functions. Therefore we expect the list in this table to grow in the future.

ecuting the instruction below at a command line.

```
pip install mitiq
```

Codeblock 1. Installing Mitiq through PyPI.

To test installation, one can run the following.

```
import mitiq
mitiq.about()
```

Codeblock 2. Testing installation & viewing package versions.

This code prints information about the Mitiq version and the versions of installed packages.

```
Mitiq: A Python toolkit for implementing error
    mitigation on quantum computers
=================================================
Authored by: Mitiq team, 2020 & later (https://
    github.com/unitaryfund/mitiq)

Mitiq Version:  0.9.3

Core Dependencies
-----------------
Cirq Version:   0.10.0
NumPy Version:  1.20.1
SciPy Version:  1.4.1

Optional Dependencies
---------------------
PyQuil Version: 2.28.0
Qiskit Version: 0.24.0
Braket Version: 1.5.16

Python Version: 3.7.7
Platform Info:  Linux (x86_64)
```

Codeblock 3. Example output of Codeblock 2.

In this example output, we see several packages. The core requirements of Mitiq are Cirq (used to internally represent and manipulate quantum circuits), NumPy (used for general numerical procedures), and SciPy [23] (used for curve fitting). The remaining packages (pyQuil, Qiskit, Braket) are optional quantum software packages which can interface with Mitiq. Although Mitiq's internal quantum circuit representation is a Cirq `Circuit`, any supported quantum circuit types can be used with Mitiq. The current supported circuit types are summarized in Table I. A Mitiq `QPROGRAM` is the union of all supported circuit representations which are installed with Mitiq. For example, if Qiskit is the only optional package installed, the `QPROGRAM` type will be the union of a Cirq `Circuit` and a Qiskit `QuantumCircuit`. If pyQuil is also installed, `QPROGRAM` will also include the pyQuil `Program` type.

The source code for Mitiq is hosted on GitHub at

https://github.com/unitaryfund/mitiq

and is distributed with an open-source software license: GNU GPL v. 3.0.

More details about the software, packaging information, and guidelines for contributing to Mitiq are included in Sec. VII.

B. Main usage

To implement error mitigation techniques in Mitiq, we assume that the user has a function which inputs a quantum circuit and returns the expectation value of an observable. Mitiq uses this function as an abstract interface of a generic noisy backend and we refer to it as an *executor* because it *executes* a quantum circuit. The signature of this function should be as follows:

```
def executor(circuit: mitiq.QPROGRAM) -> float:
```

Codeblock 4. Signature of an executor function which is used by Mitiq to perform quantum error mitigation.

Mitiq treats the `executor` as a black box to mitigate the expectation value of the observable returned by this function. The user is responsible for defining the body of the `executor`, which generally involves:

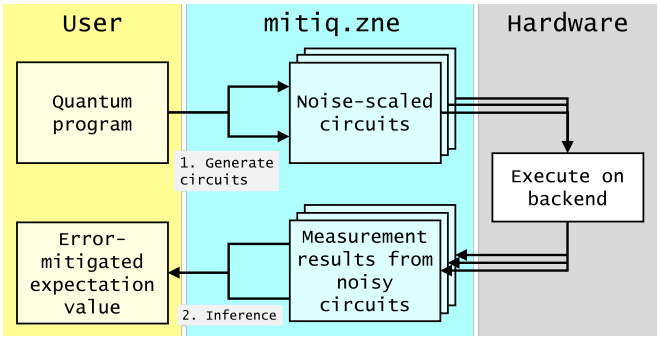

FIG. 2. Overview of the zero-noise extrapolation workflow in Mitiq. An input quantum program is converted into a set of noise-scaled circuits defined by a noise scaling method and a set of noise scale factors. These auxiliary circuits are executed on the back-end according to a user-defined executor function (see Appendix A for examples) producing set of noise-scaled expectation values. A classical inference technique is used to fit a model to these noise-scaled expectation values. Once the best-fit model is established, the zero-noise limit is returned to give an error-mitigated expectation value.

1. Running the `circuit` on a real or simulated QPU.
2. Post-processing to compute an expectation value.
3. Returning the expectation value as a floating-point number.

Example `executor` functions are shown in Appendix A. Since Mitiq treats the `executor` as a black box, circuits can be run on any quantum processor available to the user. For example, we present benchmarks run on IBM and Rigetti quantum processors as well as on noisy simulators in Sec. III.

Once the executor is defined, implementing a standard error mitigation technique such as zero-noise extrapolation (ZNE) needs only a single line of code:

```
from mitiq.zne import execute_with_zne

zne_value = execute_with_zne(circuit, executor)
```

Codeblock 5. Using Mitiq to perform zero-noise extrapolation. The `circuit` is a supported quantum program type, and the `executor` is a function which executes the circuit and returns an expectation value.

The `execute_with_zne` function uses the `executor` to evaluate the input `circuit` at different noise levels, extrapolates back to the zero-noise limit and then returns this value as an estimate of the noiseless observable. Figure 2 shows a high-level workflow.

As described in Sec. IV, there are multiple techniques to scale the noise in a quantum circuit and infer (extrapolate back to) the zero-noise limit. The default noise scaling method used by `execute_with_zne` is random local unitary folding [13] (see Sec. IV A), and the default inference technique is Richardson extrapolation (see Sec. IV B). Different techniques can be specified as arguments to `execute_with_zne` as follows.

```
zne_value = execute_with_zne(
    circuit,
    executor,
    scale_noise=<noise scaling method>,
    factory=<inference method>,
)
```

Codeblock 6. Providing arguments to `execute_with_zne` to use different noise scaling methods and inference techniques.

In addition to zero-noise extrapolation, one might be interested in applying a different error mitigation technique. For example, probabilistic error cancellation (PEC) [2, 4] is a method which promises to reduce the noise of a quantum computer with the only additional resource requirement being a higher sampling overhead.

Assuming the user has defined an `executor` as described above, PEC can be applied as follows:

```
from mitiq.pec import execute_with_pec

pec_value = execute_with_pec(
    circuit,
    executor,
    representations=<quasi-probability
    representations of ideal circuit gates>
)
```

Codeblock 7. Using Mitiq to perform probabilistic error cancellation. The `circuit` is a supported quantum program type, the `executor` is a function which executes the circuit and returns an expectation value and the `representations` argument contains information about the quasi-probability representations of the ideal gates in terms of the hardware noisy gates.

The `execute_with_pec` function internally samples from a quasi-probability representation of the input `circuit` that depends on the input `representations` of individual gates (see Sec. V B for more details on gate representations). The user-defined `executor` is used to run the sampled circuits. Eventually, `execute_with_pec` combines the results and returns an unbiased estimate of the ideal observable. As schematically represented in Figure 6, the workflow is very similar to the previous case of ZNE (shown in Figure 2) but, in this case, the noisy circuits are sampled probabilistically and executed at the base noise level of the underlying hardware (noise scaling is not used).

The code examples shown in Codeblocks 4-7 demonstrate the main usage of Mitiq. Alternatives to the `execute_with_zne` and `execute_with_pec` functions are described in Sec. VII A — these alternatives implement the same methods but offer different ways to call them which may be more convenient, depending on context.

In the following section, we show results of benchmarks using Mitiq on IBM and Rigetti quantum processors as well as noisy simulators. We then explain the structure of the library in more detail.

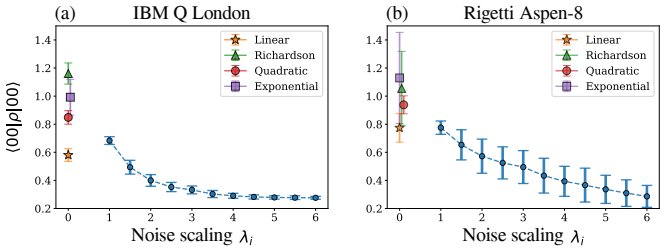

FIG. 3. Zero-noise extrapolation on two-qubit randomized benchmarking circuits run on (a) the IBMQ "London" quantum processor and (b) the Rigetti Aspen-8 quantum processor. Results are obtained from 50 randomized benchmarking circuits which contain, on average, 97 single-qubit gates and 17 two-qubit gates for (a) and 19 single-qubit gates and 7 two-qubit gates for (b). Noise is increased via random local unitary folding (see Sec. IV A), and markers show zero-noise values obtained by different extrapolation techniques (see Sec. IV B). (Note that some markers are staggered for visualization but all are extrapolated to the zero-noise limit.) In this example, the true zero-noise value is $\langle 00|\rho|00\rangle = 1$. For (b), qubits 32 and 33 are used on the Aspen-8 processor, while for (a) the same two qubits are not necessarily used for each run. For Richardson extrapolation, we used only three data points (first, middle, and last) to do the fitting.

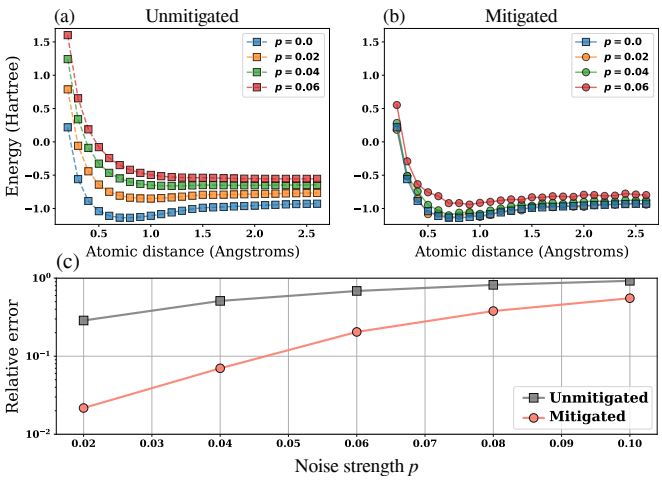

FIG. 4. Unmitigated (a) and mitigated (b) energy surfaces of H_2. The mitigated energy surfaces use zero-noise extrapolation with random local unitary folding (see Sec. IV A) and second-order polynomial inference (see Sec. IV B). Panel (c) quantifies the relative error of potential energy surfaces as the L_2 distance $||E_0(r) - E_p(r)||_2/||E_0(r)||_2$ for different (simulated) depolarizing noise strengths p.

III. BENCHMARKS WITH MITIQ

A. Randomized benchmarking circuits

Figure 3 shows the effect of zero-noise extrapolation on two-qubit randomized benchmarking circuits run on both IBM and Rigetti quantum computers. The blue curve shows the expectation value $\langle 00|\rho|00\rangle$ (which should be 1 for a noiseless circuit where $\rho = |00\rangle\langle 00|$) at different noise levels, and markers show mitigated observable values obtained from different inference techniques. Error bars show the standard deviation over fifty independent runs.

In Fig. 3(a), the expectation value decays, on average, exponentially as noise is increased by random local unitary folding described in Sec. IV A. (Note that such exponential decay is expected if a depolarizing noise model is assumed.) Accordingly, exponential inference provides a zero-noise value closest to the true noiseless value. In Fig. 3(b), the exponential decay is less pronounced and quadratic extrapolation provides the best zero-noise estimate in this case.

Depending on the noise model as well as base noise level, different inference techniques can provide better zero-noise estimates. We discuss inference techniques more in Sec. IV B and the limitations of zero-noise extrapolation more in Sec. VIII A.

B. Potential energy surface of H_2

We now consider a canonical example of computing the potential energy surface of molecular Hydrogen using the variational quantum eigensolver. We follow Ref. [24] and use the minimal STO-6G basis and Bravyi-Kitaev transformation to write the Hamiltonian for H_2 as

$$H = g_0 I + g_1 Z_0 + g_2 Z_1 + g_3 Z_0 Z_1 + g_4 X_0 X_1 + g_5 Y_0 Y_1. \quad (1)$$

Here, g_i are numerical coefficients which depend on the atomic separation and I, X, Y, and Z are Pauli operators.

Figure 4(a) shows unmitigated energy surfaces at three different noise levels while Fig. 4(b) shows the mitigated energy surfaces. To compute the mitigated curves, we use zero-noise extrapolation with random local unitary folding (see Sec. IV A) and second-order polynomial inference (see Sec. IV B). As can be seen, the mitigated curves overlap with the true noiseless curve much more closely than the unmitigated curves. The error is quantified in Fig. 4(c).

C. Probabilistic error cancellation example

We finally consider a toy example where Mitiq is used to apply probabilistic error cancellation. Consider the simple two-qubit circuit $\mathcal{U} = CNOT_{1,2} \circ X_1 \circ H_2$ [1] (where H_2 is the Hadamard gate applied on the second qubit)

[1] In this notation, the chronological order of the gates is from right to left, i.e., $CNOT$ is the last gate of $\mathcal{U}$.

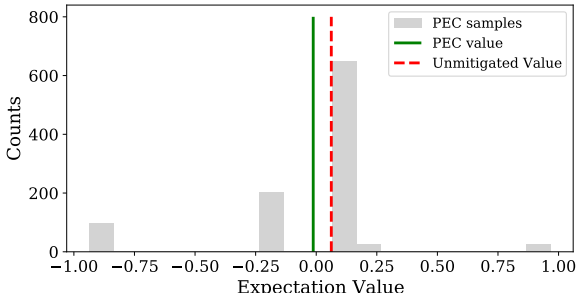

FIG. 5. Expectation value estimated with PEC (green line) and the corresponding histogram of raw PEC samples (gray bars). For comparison, also the unmitigated expectation value is shown (red dashed line). The exact ideal result is zero. For the numerics, we used a density matrix simulation and therefore shot noise is absent.

and assume that we want to measure the expectation value of $\mathcal{O} = |00\rangle\langle 00|$, whose exact theoretical value is zero. We also assume that each gate of the (simulated) backend is followed by local (single-qubit) depolarizing noise with error probability $p = 0.1$. Because of such noise, the unmitigated expectation value is nonzero (0.0622). However, after using Mitiq to implement PEC, one can improve the estimate by almost an order of magnitude (0.0071). The results are reported in Fig. 5, where the histogram of the raw PEC samples is also visible.

IV. ZERO-NOISE EXTRAPOLATION MODULE

We now describe the Mitiq library in more detail. The module structure is shown in Fig. 1 and includes a module to interface with supported quantum programming frameworks, several modules associated to different error mitigation techniques, and a module for benchmarking such techniques.

In this section we focus on the zero-noise extrapolation module `mitiq.zne`, while other error mitigation modules are considered in the next sections.

Zero-noise extrapolation was first introduced in [2, 3] and works by intentionally increasing (scaling) the noise of a quantum computation to then extrapolate back to the zero-noise limit. More specifically, let ρ be a state prepared by a quantum circuit and $E^\dagger = E$ be an observable. We wish to estimate $\mathrm{Tr}[\rho E] \equiv \langle E \rangle$ as though we had an ideal (noiseless) quantum computer, but there is a base noise level γ_0 which prevents us from doing so. For example, γ_0 could be the strength of a depolarizing channel in the circuit. The idea of zero-noise extrapolation is to compute

$$\langle E(\gamma_i) \rangle = \langle E(\lambda_i \gamma_0) \rangle \tag{2}$$

where (real) coefficients $\lambda_i \geq 1$ scale the base noise γ_0 of the quantum computer. After this, a curve is fit to the data collected via Eq. (2) which is then extrapolated to the zero-noise limit. This produces an estimate of the noiseless expectation value $\langle E \rangle$.

To implement zero-noise extrapolation, we thus need two subroutines:

1. A means of scaling the noise $\gamma_i = \lambda_i \gamma_0$ for different scale factors λ_i, and

2. A means of fitting a curve to the noisy expectation values and extrapolating to the zero-noise limit.

In the remainder of this section, we describe how these subroutines are implemented in Mitiq, showing several methods for both noise scaling as well as fitting/extrapolation, which we also refer to as inference.

A. Noise scaling

In one of the first formulations of zero-noise extrapolation [2], noise is scaled in superconducting processors by implementing pulses at lower amplitudes for longer time intervals. Considering that most quantum programming languages support gate-model circuits and not pulse-level access, it can be convenient to scale noise in a manner which acts on unitary gates instead of underlying pulses. For this reason, Mitiq implements *unitary folding*, introduced in [13], as a noise scaling method.

1. *Unitary Folding*

Unitary folding works by mapping gates (or groups of gates) G to

$$G \mapsto GG^\dagger G. \tag{3}$$

This leaves the ideal effect of the circuit invariant but increases its depth. If G is a gate of the circuit, we refer to the process as *local folding*. If G is the entire circuit, we call it *global folding*.

In Mitiq, folding functions input a circuit and a *scale factor* — i.e., a number to increase the depth of the circuit by. (In Eq. (2), each coefficient λ_i is a scale factor.) The minimum scale factor is one (which corresponds to folding no gates), a scale factor of three corresponds to folding all gates, and scale factors beyond three fold some or all gates more than once.

For local folding, there is a degree of freedom for which gates to fold first. This order in which gates are folded can affect how the noise is scaled and thus the overall effectiveness of zero-noise extrapolation. Because of this, Mitiq defines several local folding functions in `mitiq.zne.scaling`, including:

1. `fold_gates_from_left`

2. `fold_gates_from_right`

3. `fold_gates_at_random`

We explain how these functions work with the following example. We first define a circuit, here in Cirq, which for simplicity creates a Bell state.

```
import cirq

qreg = cirq.LineQubit.range(2)
circ = cirq.Circuit(
    cirq.ops.H.on(qreg[0]),
    cirq.ops.CNOT.on(qreg[0], qreg[1])
)
print("Original circuit:", circ, sep="\n")
# Original circuit:
# 0: ---H---@---
#           |
# 1: -------X---
```

Codeblock 8. Defining a Bell state circuit in Cirq to be folded.

We can now use a local folding function, e.g. `fold_gates_from_left`, to fold this circuit.

```
from mitiq.zne import scaling

folded = scaling.fold_gates_from_left(
    circ, scale_factor=2.
)
print("Folded circuit:", folded, sep="\n")
# Folded circuit:
# 0: ---H---H---H---@---
#                   |
# 1: ---------------X---
```

Codeblock 9. Local folding from left on a Cirq circuit.

We see that the first Hadamard gate H has been transformed as $H \mapsto HH^\dagger H$, to scale the depth of the circuit by a factor of two.

In Mitiq, folding functions do not modify the input circuit. Because of this, we can input the same circuit to `fold_gates_from_right` to see the effect of this method.

```
folded = scaling.fold_gates_from_right(
    circ, scale_factor=2.
)
print("Folded circuit:", folded, sep="\n")
# Folded circuit:
# 0: ---H---@---@---@---
#           |   |   |
# 1: -------X---X---X---
```

Codeblock 10. Local folding from right on a Cirq circuit. The `scaling` module is imported in Codeblock 9.

Here, we see that the second (CNOT) gate is folded instead of the first (Hadamard) gate, as expected when we start folding from the right (or end) of the circuit instead of the left (or start) of the circuit.

The previous functions fold gates according to the following rules:

1. If the scale factor is an odd integer $1+2n$, all gates are folded n times.

2. A generic real scale factor can always be written as $\lambda = 1 + 2(n + \delta)$, where n is an integer and $\delta < 1$. In this case, all gates are folded n times and, moreover, a subset of gates is folded one more time to better approximate the scale factor. The choice of this subset of gates can be random (in `fold_gates_at_random`) or deterministic (in `fold_gates_from_left` and `fold_gates_from_right`).

We emphasize that, although these examples used a Cirq `Circuit`, circuits can be defined in any supported quantum programming language and the interface is the same as above. In addition to Cirq, Mitiq supports other quantum libraries as listed in Table I. By default, all folding functions return a circuit with the same type as the input circuit.

In the previous examples, each folded gate counts equally in the folded circuit depth. However, this may not be a reasonable assumption for realistic hardware as different gates have different noise levels. Because of this, each folding function in Mitiq supports "folding by fidelity." This works by passing an input dictionary of gate fidelities (either known or estimated) as an optional argument to a folding function. More details on folding by fidelity can be found in Mitiq's documentation.

Finally, we mention global folding. In contrast to local folding which folds subsets of gates, global folding folds the entire circuit until the input scale factor is reached. Below we show an example of global folding using the same Bell state circuit `circ` defined in Codeblock 8.

```
folded = scaling.fold_global(circ, scale_factor=3.)
print("Folded circuit:", folded, sep="\n")
# Folded circuit:
# 0: ---H---@---@---H---H---@---
#           |   |             |
# 1: -------X---X-------------X---
```

Codeblock 11. Global folding on a Bell state circuit.

Here, we see that the entire Bell state circuit has been folded once to reach the input scale factor of three. If the input scale factor is not reached by an integer number of global folds, `fold_global` will fold a group of gates from the end of the circuit such that the scale factor is reached.

2. *Parameter-noise scaling*

A gate is an abstract elementary operation which, however, is physically implemented as a continuous dynamical evolution. This evolution is generated by a suitable time-dependent control of an Hamiltonian that depends on the details of the hardware. Errors in the calibration of control pulses (e.g. pulse-area errors) or the classical noise affecting their implementation (e.g. electronic noise) can generate a dynamical channel which is different from the desired ideal gate.

In order to mitigate these type of errors, we need a practical way of scaling them. In principle this would require the detailed knowledge of the platform-dependent pulses and Hamiltonians, however, in Mitiq a simplified noise model is used instead. The simplified model is based on the fact that any unitary gate G can always be

expressed as $G = \exp(-iH)$, for some constant Hamiltonian $H = H^\dagger$ (which may be different from the physical one). Therefore, each unitary gate admits a natural parametrization with respect to a real exponent θ:

$$G(\theta) = \exp(-iH\theta) = G^\theta. \tag{4}$$

A multi-parameter version of Eq. (4) was considered in [13], but is currently not used in Mitiq. It is also worth to mention that gates are often directly defined in the parametric form of Eq. (4) as, for example, in the case of Pauli rotations.

In this setting, a noise model approximately modeling calibration and control errors can be expressed with respect to the classical parameter θ. We can assume that the actual gate is generated by a noisy parameter $\hat{\theta}$ that we can model as a random variable with mean θ and with some variance σ^2. Noise scaling can be achieved by artificially injecting additional classical noise:

$$\hat{\theta} \to \hat{\theta}^{(\lambda)} = \hat{\theta} + \hat{\delta} \tag{5}$$

where $\hat{\delta}$ is a random variable with zero mean and variance equal to $(1-\lambda)\sigma^2$, such that the resulting noise scaled parameter $\hat{\theta}^{(\lambda)}$ has mean θ and variance $\lambda\sigma^2$.

In practice, if σ^2 is known for each noisy gate, parameter scaling can be obtained by randomly over-rotating or under-rotating each gate according to the stochastic angles defined in Eq. (4). This noise scaling technique can be applied with Mitiq as shown in the next Codeblock.

```
from mitiq.zne.scaling import scale_parameters

scaled_circuit = scale_parameters(
    circuit=<the circuit to scale>,
    scale_factor=<the noise scale factor>,
    base_variance=<the base level of noise>,
)
```

Codeblock 12. Applying parameter-noise scaling to a quantum circuit. The same base level of noise (`base_variance`) is assumed for each gate of the circuit.

If the value of the base noise σ^2 is unknown, it needs to be estimated in order to apply this noise scaling method. The function `compute_parameter_variance` in the submodule `mitiq.zne.scaling` can be used for this task. Alternatively, the user may independently perform a custom estimation of σ^2 and only use Mitiq for the noise scaling step described in Codeblock 12.

3. *Using noise scaling methods in* `execute_with_zne`

As mentioned in Sec. II B, the default noise scaling method in `execute_with_zne` is `fold_gates_at_random`. Different methods can be used by passing an optional argument to `execute_with_zne`. For example, to use global folding, one can do the following.

```
from mitiq.zne import execute_with_zne
from mitiq.zne.scaling import fold_global

zne_value = execute_with_zne(
    circuit,
    executor,
    scale_noise=fold_global
)
```

Codeblock 13. Using zero-noise extrapolation with global folding by passing `fold_global` as an optional argument to `execute_with_zne`. The `circuit` and `executor` are as in Sec. II B.

Depending on the noise model of the quantum processor, using a different folding method may better scale the noise and lead to better results.

To end the discussion on noise scaling, we note that some scaling methods are deterministic while some are non-deterministic. In particular, global folding and local folding from left/right return the same folded circuit if the scale factor is the same, but `fold_gates_at_random` can return different circuits for the same scale factor. Because of this, the function `execute_with_zne` has another optional argument `num_to_average` which corresponds to the number of times to compute expectation values at the same scale factor. For example, if `num_to_average = 3`, the noise scaling method is called three times at each scale factor, and the expectation value at this scale factor is the average over the three runs. Such averaging can smooth out effects due to non-deterministic noise scaling and lead to better results in zero-noise extrapolation. Fig. 4(b) uses `fold_gates_at_random` with `num_to_average = 5`.

B. Classical inference: Factory objects

In Mitiq, a `Factory` object is a self-contained representation of a classical inference technique. In effect, it performs the "extrapolation" part of zero-noise extrapolation. This representation is hardware-agnostic and even quantum-agnostic since it only deals with classical data — namely, the input and output of a noisy computation. The main tasks of a factory are as follows:

1. Compute the expectation value by running an executor function at a given noise level, and record the result;
2. Determine the next noise level at which the expectation value should be computed;
3. Perform classical inference using the history of noise levels and expectation values to compute the zero-noise extrapolated value.

The structure of a `Factory` is designed to account for adaptive fitting techniques in which the next noise level depends on the history of previous noise levels and expectation values. In Mitiq, (adaptive) fitting techniques in zero-noise extrapolation are represented by specific factory objects. All built-in factories, summarized in Table II, can be imported from the `mitiq.zne.inference` module.

Class	Extrapolation Method
`LinearFactory`	Extrapolation with a linear fit.
`RichardsonFactory`	Richardson extrapolation.
`PolyFactory`	Extrapolation with a polynomial fit.
`ExpFactory`	Extrapolation with an exponential fit.
`PolyExpFactory`	Similar to `ExpFactory` but the exponent can be a non-linear polynomial.
`AdaExpFactory`	Similar to `ExpFactory` but the noise scale factors are adaptively chosen.

TABLE II. Factories that can be imported from `mitiq.zne.inference` to perform different extrapolation methods. More information is available in the Mitiq documentation and an analysis of the different extrapolation methods can be found in Ref. [13].

1. *Using factories in* `execute_with_zne` *to perform different extrapolation methods*

We now show examples of performing zero-noise extrapolation with fitting techniques defined by factories in Table II. As mentioned in Sec. II B, this is done by providing a factory as an optional argument to `execute_with_zne`. To instantiate a non-adaptive factory, we input the noise scale factors we want to compute the expectation values at, as shown below for the `LinearFactory`.

```
from mitiq.zne.inference import LinearFactory

linear_factory = LinearFactory(
    scale_factors=[1.0, 2.0, 3.0],
)
```

Codeblock 14. Initializing a factory object.

Here the `scale_factors` define the noise levels at which to compute expectation values during zero-noise extrapolation. This factory can now be used as an argument in `execute_with_zne` as follows. As in Sec. II B, the `circuit` is the quantum program which prepares a state of interest and the `executor` is a function which executes the circuit and returns the expectation value of an observable.

```
from mitiq.zne import execute_with_zne

zne_value = execute_with_zne(
    circuit,
    executor,
    factory=linear_factory
)
```

Codeblock 15. Using a factory object as an optional argument of `mitiq.zne.execute_with_zne`.

Instead of the default Richardson extrapolation at noise scale factors 1, 2 and 3, this call to `execute_with_zne` will perform linear extrapolation at the specified noise scale factors. As mentioned in Sec. IV A, different noise scaling methods can also be used with the optional argument `scale_noise`.

Most extrapolation techniques implemented in Mitiq are static (i.e., non-adaptive) and can be instantiated in a similar manner as the `LinearFactory`. For example, to use a second-order polynomial fit, we use a `PolyFactory` object as follows.

```
from mitiq.zne import execute_with_zne
from mitiq.zne.inference import PolyFactory

zne_value = execute_with_zne(
    circuit,
    executor,
    factory=PolyFactory(
        scale_factors=[1.0, 2.0, 3.0], order=2
    )
)
```

Codeblock 16. Instantiating a second-order `PolyFactory`.

Other static factories follow similar patterns but may have additional arguments in their constructors. For example, `ExpFactory` can take in a value for the horizontal asymptote of the exponential fit. For full details, see the Mitiq documentation.

Last, we show an example of an adaptive fitting technique defined by the `AdaExpFactory`. To use this method (introduced and described in Ref. [13]), we can do the following:

```
from mitiq.zne import execute_with_zne
from mitiq.zne.inference import AdaExpFactory

zne_value = execute_with_zne(
    circuit,
    executor,
    factory=AdaExpFactory(
        scale_factor=2.0, steps=5
    )
)
```

Codeblock 17. Using `execute_with_zne` with an adaptive fitting technique.

Instead of a list of scale factors, here we provide the initial scale factor and the rest are determined adaptively. The number of scale factors determined is equal to the argument `steps`. Additional arguments which can be passed into the `AdaExpFactory` are described in the Mitiq documentation.

2. *Using custom fitting techniques*

A custom fitting technique can be used in Mitiq by defining a new factory class which inherits from the abstract class `mitiq.zne.inference.Factory` (for general techniques) or `mitiq.zne.inference.BatchedFactory` (a subclass of `Factory` suitable for non-adaptive techniques). To get noise scale factors and expectation values, the methods `Factory.get_scale_factors()` and `Factory.get_expectation_values()` can be used.

Below, we define a static factory which performs a second-order polynomial fit and forces the expectation value to be in the interval $[-1, 1]$.

```
from mitiq.zne.inference import (
    BatchedFactory, PolyFactory,
)
import numpy as np
```

```python

class MyFactory(BatchedFactory):
    @staticmethod
    def extrapolate(
        scale_factors, exp_values, full_output,
    ):
        result = PolyFactory.extrapolate(
            scale_factors,
            exp_values,
            order=2,
            full_output=full_output,
    )
        if not full_output:
            return np.clip(result, -1, 1)
        if full_output:
            # In this case "result" is a tuple
            zne_limit = np.clip(results[0], -1, 1)
            return (zne_limit, *results[1:])
```

Codeblock 18. Defining a custom fitting technique by creating a new factory object.

This factory can now be used as an argument in `execute_with_zne` to use the custom fitting technique. Other fitting techniques can be defined in a similar manner as the code block above.

V. PROBABILISTIC ERROR CANCELLATION MODULE

Probabilistic error cancellation (PEC) [2, 4] is another error mitigation technique which is available in Mitiq. Its workflow is schematically represented in Figure 6: a set of auxiliary circuits are probabilistically sampled, executed on a noisy backend and, eventually, the noisy results are post-processed to infer an error-mitigated expectation value. In principle, this method can probabilistically remove the noise of a quantum computer without additional resources apart from a higher sampling overhead. More information about the advantages and the limitations of PEC is given in Sec VIII B.

A key step of PEC is to represent each ideal unitary gate $\mathcal{G}$ in a circuit as an average over a set of noisy gates which are physically implementable $\{\mathcal{O}_\alpha\}$, weighted by a real quasi-probability distribution $\eta(\alpha)$:

$$\mathcal{G} = \sum_\alpha \eta(\alpha)\mathcal{O}_\alpha, \tag{6}$$

where $\sum_\alpha \eta(\alpha) = 1$ (trace-preserving condition). The calligraphic operators $\mathcal{G}$ and $\{\mathcal{O}_\alpha\}$ should be considered as linear super-operators acting on density matrices and not on state vectors [2, 4]. If a representation like Eq. (6) is known for each ideal gate of a circuit, then any ideal expectation value can be estimated as a Monte Carlo average over different noisy circuits, each one sampled according to the quasi-probability distributions associated to the ideal gates [2, 4]. The real coefficients $\eta(\alpha)$ which appear in Eq. (6) can be negative for some values of α and, because of this negativity, the required number of Monte Carlo samples can be large [2, 4]. In principle, assuming a perfect tomographic knowledge of the noisy gates $\mathcal{O}_\alpha$, this method allows for a perfect cancellation of the hardware noise (for a sufficiently large number of samples).

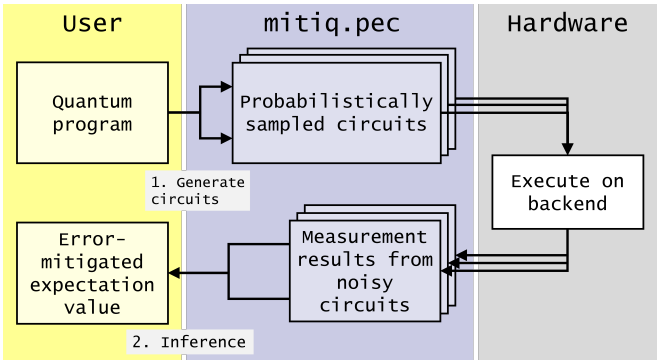

FIG. 6. Overview of the probabilistic error cancellation workflow in Mitiq. An input quantum program is converted into a set of auxiliary circuits which are probabilistically sampled according to the PEC technique described in the main text. These sampled circuits are executed on the back-end according to a user-defined executor function and produce a set of noisy results. The noisy expectation values are combined (with a suitable linear combination) to obtain an unbiased estimate of the ideal expectation value.

In the remainder of this section, we describe how one can define gate representations and how one can probabilistically sample from them using Mitiq.

A. Noisy Operations

The r.h.s. of Eq. (6) is a sum over noisy operations $\mathcal{O}_\alpha$. A noisy operation is an elementary gate (or a small sequence of gates) acting on specific qubits which can be physically implemented on hardware. To each noisy operation we can associate a (small) `QPROGRAM` describing the gates to be applied on the physical qubits. Moreover, from a quantum tomography analysis, one can associate to a noisy operation also a numerical matrix representing the completely-positive and trace-preserving channel induced by the operation. In Mitiq, this concept is captured by the `NoisyOperation` class, which can be initialized as follows:

```python
from mitiq.pec.types import NoisyOperation

noisy_operation = NoisyOperation(
    circuit=<sequence of operations as a QPROGRAM>,
    channel_matrix=<optional superoperator matrix>,
)
```

Codeblock 19. Initialization of a `NoisyOperation` from a sequence of operations and the (optional) super-operator matrix.

Once the set of all noisy operations $\{\mathcal{O}_\alpha\}$ has been defined, we can associate to each operation the corresponding quasi-probability $\eta(\alpha)$ via a simple Python dictionary:

```
basis_expansion = {
    <1st noisy operation>: <1st real coefficient>,
    <2nd noisy operation>: <2nd real coefficient>,
    ...
}
```

Codeblock 20. Defining a basis expansion as a Python dictionary which associates a real coefficient to each noisy operation.

B. `OperationRepresentation` Objects

The dictionary in the previous Codeblock 20 completely defines the linear combination in the r.h.s. of Eq. (6) but it contains no information about the l.h.s. of Eq. (6). This motivates the use of an `OperationRepresentation` class which can be used to store and manipulate all the information which is contained in Eq. (6).

```
from mitiq.pec.types import OperationRepresentation

operation_rep = OperationRepresentation(
    ideal=<ideal operation as a QPROGRAM>,
    basis_expansion=<basis expansion dictionary>,
)
```

Codeblock 21. Initializing an `OperationRepresentation` object. The first argument is the ideal operation that we want to express as a linear combination of noisy operations. The second argument is the associated `basis_expansion` which can be defined as shown in Codeblock 20.

Given a list of `OperationRepresentation` objects, associated to all the gates of a circuit of interest, the user can easily apply PEC via the function `execute_with_pec` as shown in Codeblock 7 of Sec II.

C. Sampling Functions

The function `execute_with_pec` internally performs the Monte Carlo sampling process which is necessary to estimate an expectation value with PEC. However, the user may be interested in manually sampling gates and circuits for a variety of reasons, *e.g.*, for research purposes, for intermediate manipulations, for efficiency optimizations, etc..

In particular, to sample an implementable `NoisyOperation` from the quasi-probability distribution of an ideal operation one can do as follows:

```
noisy_operation, sign, eta = operation_rep.sample()
```

Codeblock 22. Sampling an implementable `NoisyOperation` from the quasi-probability representation of an ideal operation. The quasi-probability representation is given by the `OperationRepresentation` object defined in Codeblock 21. In addition to the sampled `noisy_operation`, the method `sample()` returns the associated coefficient (`eta`) that appears in Eq. (6) and its sign (`sign`).

Typically, one is interested in sampling an entire implementable circuit from the quasi-probability representation of an ideal circuit. This can be easily achieved via the `sample_circuit` function, which internally performs repeated calls to the previous `sample_sequence` function:

```
from mitiq.pec.sampling import sample_circuit

sampled_circuit, sign, norm = sample_circuit(
  ideal_circuit=<ideal circuit as a QPROGRAM>,
  representations=<list of oper. representations>,
)
```

Codeblock 23. Sampling an implementable circuit from the quasi-probability representation of an `ideal_circuit`. Such quasi-probability distribution is implicitly deduced from the input list of `OperationRepresentations` objects associated to the gates of the input `ideal_circuit` .

VI. CLIFFORD DATA REGRESSION MODULE

In this section we present the `mitiq.cdr` module which implements two recent error mitigation approaches known as Clifford data regression (CDR) and variable noise Clifford data regression (vnCDR) [5, 6]. In both techniques, a trained regression model mapping noisy to exact expectation values is used to mitigate the effect of noise on some observable of interest. The model is trained using data produced by the execution of near-Clifford circuits performed on a noisy quantum computer and on a classical simulator.

A. Clifford data regression (CDR)

The Clifford data regression [5] technique uses near-Clifford quantum circuit data to learn a model approximating effects of the noise on an expectation value of an observable $\langle E \rangle = \mathrm{Tr}\rho E$ for a quantum state ρ given by a quantum circuit of interest. The learned model is used to mitigate the noisy expectation value $\langle E(\gamma_0) \rangle$ obtained with a quantum computer with the base noise level γ_0. The mitigated expectation value $\langle E \rangle^{\text{mitigated}}$ is obtained using the following procedure:

1. Construct the training circuits corresponding to states $\{\rho_i^{\text{train}}, i = 1, \ldots, n\}$ by replacing non-Clifford gates in the circuit of interest by Clifford gates.

2. For each training circuit ρ_i^{train} evaluate classically a noiseless expectation value of E, $y_i = \mathrm{Tr}\rho_i E$, and its noisy expectation value x_i using a quantum computer.

3. Fit exact and noisy expectation values of the training circuits $\{(x_i, y_i)\}$ with a linear model $y = ax+b$.

4. Use the fitted model to mitigate $\langle E(\gamma_0) \rangle$

$$\langle E \rangle^{\text{mitigated}} = a\langle E(\gamma_0) \rangle + b.$$

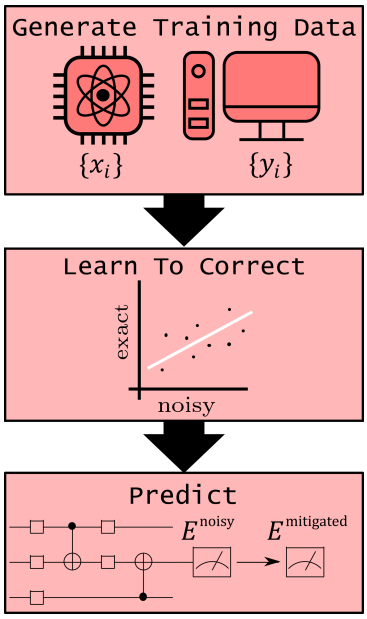

FIG. 7. Summary of the CDR mitigation method showing the steps realised to mitigate an observable of interest. First the training set composed of noisy $\{x_i\}$ and exact $\{y_i\}$ expectation values is generated using near-Clifford circuits which are classically simulable. This data is used to fit a linear ansatz which is then used to estimate the noise-free value for some observable of interest E. We can visualise vnCDR as adding another axis to the training data, along which noise is increased. Diagram modified from [5].

B. Variable noise Clifford data regression (vnCDR)

CDR can be generalized to enable learning the noise effects from near-Clifford training circuits simulated at different noise levels λ_l. This approach is called variable noise Clifford data regression [6] and can be used to learn a zero-noise extrapolation model for an observable E and a quantum circuit preparing the state ρ. The vnCDR procedure to obtain $\langle E\rangle^{\text{mitigated}}$ includes evaluation of the training circuits on a quantum computer at different noise rates $\lambda_l\gamma_0$ and fitting a extrapolation model:

1. Prepare the training circuits $\{\rho_i^{\text{train}}, i = 1,\ldots,n\}$ using Clifford substitutions, following the same procedure for CDR.

2. For each training circuit ρ_i^{train} evaluate classically a noiseless expectation value of E, $y_i = \text{Tr}\rho_i E$, and its noisy expectation values $x_{i,l}$ using a quantum computer with several noise rates $\lambda_l\gamma_0$, $\lambda_l \geq 1, l = 1,\ldots,m$.

3. Fit the expectation values of the training circuits with a linear ansatz $y = f(x_1, x_2, \ldots, x_m)$. Where

$$f(x_1, x_2, \ldots, x_m) = \sum_{l=1}^{m} x_l a_l + b \ . \tag{7}$$

4. Use the fitted ansatz to correct the noisy expectation values of E:

$$\langle E\rangle^{\text{mitigated}} = f(\langle E(\lambda_1\gamma_0)\rangle, \langle E(\lambda_2\gamma_0)\rangle, \ldots, \langle E(\lambda_m\gamma_0)\rangle) \ .$$

The default linear ansatz used within Mitiq includes a constant term. Recently this was shown to lead to better mitigated results on real quantum hardware [25].

C. Applying CDR and vnCDR with Mitiq

Clifford data regression is implemented in Mitiq according to the workflow schematically represented in Figure 8. This error mitigation technique can be applied with the following code Codeblock:

```
from mitiq.cdr import execute_with_cdr

cdr_values = execute_with_cdr(
    circuit,
    executor,
    simulator=<near-Clifford classical simulator>,
    observables=<the observables to mitigate>,
)
```

Codeblock 24. Applying CDR with Mitiq. The function `execute_with_cdr` can be used to mitigate errors the expectation values of the input `observables`. The input `executor` is a user-defined function for running the input circuit and the associated training circuits on a quantum backend. The input `simulator` is the ideal counterpart of the noisy `executor` and is necessary to obtain exact classical simulations of the (near-Clifford) training circuits.

Similarly, variable-noise Clifford data regression can be applied by specifying the optional list of noise scale factors in the function `execute_with_cdr`.

```
from mitiq.cdr import execute_with_cdr

cdr_values = execute_with_cdr(
    # The same arguments used for CDR:
    ...
    # Additional argument for applying vnCDR:
    scale_factors=<the noise scale factors>,
)
```

Codeblock 25. Applying vnCDR with Mitiq by calling `execute_with_cdr` and passing a list of noise `scale_factors`. Optionally, a noise scaling method can be specified via the argument `scale_noise`, whose default value is `fold_gates_at_random`.

One of the key features of both CDR and vnCDR is the construction of a set of classically simulable near-Clifford circuits. At the time of this writing, CDR implemented within Mitiq assumes that the input circuit is pre-compiled in the following gate set $\{R_Z, \sqrt{X}, \text{CNOT}\}$. This ensures that all the non-Clifford gates are contained in the R_Z gates. This is particularly suitable for IBM processors but may be less appropriate for other backends. Different gate sets may be supported in the future.

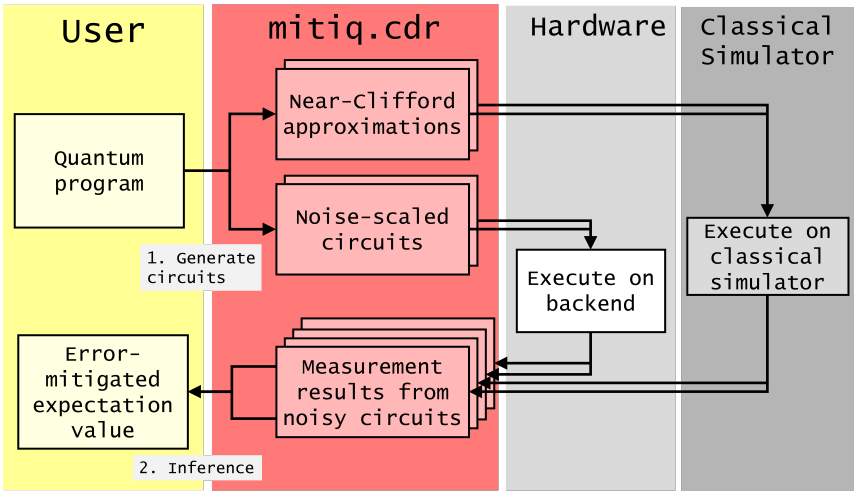

FIG. 8. Overview of the Clifford data regression workflow in Mitiq. An input quantum program is converted into a set of noise-scaled circuits and a set of training circuits (near-Clifford approximations in which the noise is also scaled). All the auxiliary circuits are executed on the real back-end while the near-Clifford training circuits are executed also on a classical simulator. The noisy and the exact (simulated) results are post-processed to infer the ideal expectation value of the original quantum program.

VII. ADDITIONAL LIBRARY INFORMATION

In this section we provide technical details and meta-information about the Mitiq library.

A. Alternative ways of using Mitiq

As we have already shown, errors affecting the estimation of expectation values can be reduced with appropriate functions returning the mitigated expectation value as a real number e.g. `execute_with_zne`, `execute_with_pec`. Here we show two alternative methods for applying the same error mitigation process. Depending on context, these alternative but equivalent methods may provide a simpler usage.

The first method is provided by the function `mitigate_executor` which inputs the same arguments as `execute_with_*` except the quantum circuit. This function returns a new executor which implements error mitigation when it is called with a quantum program, as shown below.

```
from mitiq.zne import mitigate_executor

mitigated_executor = mitigate_executor(
    executor,
    scale_noise=<noise scaling method>,
    factory=<inference method>
)

zne_value = mitigated_executor(circuit)
```

Codeblock 26. Modifying an `executor` with the function `mitigate_executor`. The new `mitigated_executor` performs zero-noise extrapolation when called on a quantum circuit.

The `mitigate_executor` function can also be imported from other modules in order to apply different techniques. For example, probabilistic error cancellation can be applied after importing `mitigate_executor` from `mitiq.pec`.

The second method is to directly decorate the `executor` function such that it automatically performs error mitigation when called. Also in this case, one should use the decorator corresponding to the desired error mitigation technique, e.g.: `zne_decorator`, `pec_decorator`, etc.

```
from mitiq import QPROGRAM
from mitiq.zne import zne_decorator

@zne_decorator(
    factory=<inference method>,
    scale_noise=<noise scaling method>
)
def executor(circuit: QPROGRAM) -> float:
    ...

zne_value = executor(circuit)
```

Codeblock 27. Decorating an `executor` with `zne_decorator` so that zero-noise extrapolation is implemented when the `executor` is called on a quantum program

In the above Codeblock, the `zne_decorator` takes the same optional arguments as `execute_with_zne`. If no optional arguments are used, the decorator should still be called with parentheses, e.g. `@zne_decorator()`.

Decorators (or `mitigate_executor`) could be used to easily stack multiple error mitigation techniques. For example, in the next Codeblock, a noisy executor is first mitigated with PEC and later with ZNE.

```
from mitiq import QPROGRAM
from mitiq.zne import zne_decorator
from mitiq.pec import pec_decorator
```

```

@zne_decorator(<zne arguments>)
@pec_decorator(<pec arguments>)
def executor(circuit: QPROGRAM) -> float:
    ...

mitigated_value = executor(circuit)
```

Codeblock 28. Multiple decorators can be used to combine different error mitigation methods

Whether there is any practical advantage in combining multiple techniques is still an open research question. Mitiq can be an appropriate toolkit for exploring this research direction.

B. Mitiq documentation

Mitiq's documentation is hosted online at https://mitiq.readthedocs.io and includes a User's Guide and an API glossary. The User's Guide contains more information on topics covered in this manuscript and additional information on topics not covered here; for example, more examples of executor functions and an advanced usage guide for factory objects. The API glossary is auto-generated from the docstrings (formatted comments to code objects) and contains information about public functions and classes defined in Mitiq.

C. Contribution guidelines

We welcome contributions to Mitiq from the larger community of quantum software developers. Contributions can come in the form of feedback about the library, feature requests, bug fixes, or pull requests. Feedback and feature requests can be done by opening an issue on the Mitiq GitHub repository. Bug fixes and other pull requests can be done by forking the Mitiq source code, making changes, then opening a pull request to the Mitiq GitHub repository. Pull requests are peer-reviewed by core developers to provide feedback and/or request changes. Contributors are expected to uphold Mitiq development practices including style guidelines and unit tests. More details can be found in the Contribution guidelines documentation.

VIII. DISCUSSION

Now that we have described error mitigation techniques in Mitiq and how to use them, we discuss limitations of these techniques as well as the relationship between zero-noise extrapolation, probabilistic error cancellation, and other strategies.

A. Limitations of zero-noise extrapolation

Zero-noise extrapolation [2, 3] is a useful error mitigation technique but it is not without limitations. First and foremost, the zero-noise estimate is extrapolated, meaning that performance guarantees are quite difficult in general. If a reasonable estimate of how increasing the noise affects the observable (e.g., the blue curves in Fig. 3) is known, then ZNE can produce good zero-noise estimates. This is the case for simple noise models such as depolarizing noise, but noise in actual quantum systems is more complicated and can produce different behavior than expected, e.g. Fig 3(b). In this case the performance of ZNE is tied to the performance of the underlying hardware. If expectation values do not produce a smooth curve as noise is increased, the zero-noise estimate may be poor and certain inference techniques may fail. In particular, one has to take into account that any initial error in the measured expectation values will propagate to the zero-noise extrapolation value. This fact can significantly amplify the final estimation uncertainty. In practice, this implies that the evaluation of a mitigated expectation value requires more measurement shots with respect to the unmitigated one.

Additionally, zero-noise extrapolation cannot increase the performance of arbitrary circuits. If the circuit is large enough such that the expectation of the observable is almost constant as noise is increased (e.g., if the state is maximally mixed), then extrapolation will of course not help the zero-noise estimate. The regime in which ZNE is applicable thus depends on the performance of the underlying hardware as well as the circuit. A detailed description of when zero-noise extrapolation is effective, and how effective it is, is the subject of ongoing research.

B. Limitations of probabilistic error cancellation

The limitations of probabilistic error cancellation [2, 4] are similar to those of other error mitigation methods: more circuit executions are necessary compared to the unmitigated case and the method it is not appropriate in the asymptotic regime of many gates or large noise. Compared to ZNE, PEC has the important advantage of producing an unbiased estimation. This means that, if the quasi-probability representations of all the gates are known with sufficiently large accuracy, in the limit of many samples, the PEC estimation converges to the ideal expectation value. Unfortunately, PEC has some practical disadvantages too. The number of samples grows exponentially with respect to the circuit size and to the amount of noise. Moreover, the full tomography of the noisy gates is typically necessary in order to build the quasi-probability representations for the ideal gates. One should also take into account that tomographic errors in the characterization of the hardware gates can propagate through the PEC process inducing a significant error in the final estimation.

C. Limitations of Clifford data regression

Clifford data regression [5, 6] has the promising advantage of being a self-tuning technique since the inference model is not assumed *a priori* but learned during the training phase. However, this technique presents some limitations as well. The training phase typically introduces a significant overhead (many training circuits must be executed with both quantum and classical hardware). Moreover, the training data is extracted from near-Clifford circuits which may have a different response to the hardware noise compared to the true circuit of interest. It is also worth noting that this technique requires an efficient classical simulator of near-Clifford circuits in addition to a quantum backend.

D. Overview of error mitigation techniques

Zero-noise extrapolation was first proposed in [2, 3] and first demonstrated experimentally in [12]. References [13, 26] have extended the noise scaling and extrapolation techniques. Additionally, these references and this paper show experimental demonstrations of zero-noise extrapolation and how it can improve the results of noisy quantum computations.

The purposeful randomization of gates is another approach to quantum error mitigation. Specific techniques include compiling the quantum circuit with twirling gates [10], expressing noiseless gates in a basis of noisy gates as in probabilistic error cancellation [2], and a hybrid proposal improving the scaling of the technique with circuit depth and other resources [4]. Such techniques have been investigated experimentally in trapped ions [18] and superconducting qubits [27] (implementing gate set tomography).

Subspace expansion refers to another set of error mitigation techniques. In Ref. [28], a hybrid quantum-classical hierarchy was introduced, while in Ref. [29], symmetry verification was introduced. It has been demonstrated with a stabilizer-like method [30], exploiting molecular symmetries [11], and with an experiment on a superconducting circuit device [31]. Other symmetry-based protocols have since been proposed [32–34]. Other error mitigation techniques include approximating error-correcting codes in quantum channels [35], and have been extended to improve quantum sensing [36], metrology [37], and reduce errors in analog quantum simulation [27].

E. Differences and relations to neighbouring fields

Quantum error mitigation is deeply connected to quantum error correction and quantum optimal control, two fields of study that also aim at reducing the impact of errors in quantum information processing in quantum computers. More generally, quantum error mitigation is informed of the general theory of open quantum systems. While these are fluid boundaries, it can be useful to point out some differences among these more established fields and the emerging niche of quantum error mitigation.

1. *Quantum error correction*

Quantum error correction creates logical qubits out of multiple error-prone physical qubits. After applying logical operations which correspond to the physical operations we want to perform in our circuit, ancilla qubits are measured to diagnose which (if any) errors occurred. Depending on the outcome of these "syndrome measurements," correction operations are performed to remove the errors (if any) that occurred. If the error rate lies below a certain threshold, errors can be actively removed. We can thus say that the goal of error correction is to detect and exactly correct errors, while the goal of error mitigation is to lessen the effect of errors.

The drawback of quantum error correction techniques is that they require a large overhead in terms of additional physical qubits needed to create logical qubits. Current quantum computing devices have been able to demonstrate some components of quantum error correction with a very small number of qubits [38, 39]. Indeed, some techniques for quantum error mitigation emerged as more practical quantum error correction solutions [40].

2. *Quantum optimal control*

Optimal control theory encompasses a versatile set of techniques that can be applied to many scenarios in quantum technology [41]. It is generally based on a feedback loop between an agent and a target system. A key difference between some quantum error mitigation techniques and quantum optimal control is that the former can be implemented in some instances with post-processing techniques, while the latter relies on an active feedback loop. An example of a specific application of optimal control to quantum dynamics that can be seen as a quantum error mitigation technique is dynamical decoupling [7–9]. This technique employs fast control pulses to effectively decouple a system from its environment, with techniques pioneered in the nuclear magnetic resonance community [42]. Quantum optimal control techniques are being integrated into quantum computing software as a means for noise characterization and error mitigation [43].

3. *Environment-induced error protection*

More in general, quantum computing devices can be studied in the framework of open quantum systems [44–46], that is, systems that exchange energy and informa-

tion with the surrounding environment in controlled and unwanted ways.

Since errors occur for several reasons in quantum computers, the microscopic description at the physical level can vary broadly, depending on the quantum computing platform that is used as well as the computing architecture, and error mitigation strategies can be employed with an awareness of this variability. For example, superconducting-circuit-based quantum computers have chips that are prone to cross-talk noise [47], while qubits encoded in trapped ions need to be shuttled with electromagnetic pulses, and solid-state artificial atoms, including quantum dots, are heavily affected by inhomogeneous broadening [48]. Considering the physical layer of the actual device [49, 50], as well as modeling and adapting the control pulses, can in practice result in more effective error mitigation strategies.

One approach to reduce the impact of noise and errors is to tailor a larger computational space to protect the system from exiting the computational basis. This approach has been particularly fruitful in the context of bosonic quantum codes [51–54].

Moreover, autonomous error correction approaches have been recently proposed and experimentally verified [55], which exploit the environment to induce error-robust processes. More in general, decoherence-free subspaces have been proposed within the study of Liouvillian dynamics [56–59].

IX. CONCLUSION

We have introduced a fully open-source library for quantum error mitigation on near-term quantum computers. Our library can interface with multiple quantum programming libraries — in particular Cirq, Qiskit, pyQuil, and Braket — and arbitrary quantum processors (real or simulated) available to the user. In this paper, we presented experimental and numerical examples demonstrating how error mitigation can enhance the results of a noisy quantum computation. We then discussed the library in detail, focusing on the specific modules of Mitiq associated to different error mitigation techniques: zero-noise extrapolation, probabilistic error cancellation and Clifford data regression. After mentioning additional software information including support and contribution guidelines, we discussed how the error mitigation techniques in our library relate to other error mitigation techniques as well as quantum error correction, quantum optimal control, and the theory of open quantum systems.

In future work, we plan to incorporate additional error mitigation techniques into the library and to expand the set of benchmarks to better understand when quantum error mitigation is beneficial. Work can also be done to improve the existing modules, for example by implementing different noise-scaling methods, inference techniques, or new error cancellation protocols. One candidate noise-scaling method is pulse stretching which will be possible when pulse-level access to quantum hardware becomes available through more cloud services [60]. A high-level road map for future development which includes more information on these ideas as well as other ideas can be found on the Mitiq Wiki.

ACKNOWLEDGEMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Accelerated Research in Quantum Computing under Award Number de-sc0020266 and by IBM under Sponsored Research Agreement No. W1975810. RL acknowledges support from a NASA Space Technology Graduate Research Opportunities Award. M.H.G is supported by "la Caixa" Foundation (ID100010434), Grant No. LCF/BQ/DI19/11730056. PC was supported by the Laboratory Directed Research and Development (LDRD) program of Los Alamos National Laboratory (LANL) under project numbers 20190659PRD4 and 20210116DR. We thank IBM and Rigetti for providing access to their quantum computers. The views expressed in this paper are those of the authors and do not reflect those of IBM or Rigetti.

[1] S. Endo, Z. Cai, S. C. Benjamin, and X. Yuan, *Hybrid quantum-classical algorithms and quantum error mitigation,* J. Phys. Soc. Japan **90**, 032001 (2021).

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

[3] Y. Li and S. C. Benjamin, *Efficient variational quantum simulator incorporating active error minimization,* Phys. Rev. X **7**, 021050 (2017).

[4] S. Endo, S. C. Benjamin, and Y. Li, *Practical quantum error mitigation for near-future applications,* Phys. Rev. X **8**, 031027 (2018).

[5] P. Czarnik, A. Arrasmith, P. J. Coles, and L. Cincio, *Error mitigation with Clifford quantum-circuit data,* arXiv:2005.10189 (2020).

[6] A. Lowe, M. H. Gordon, P. Czarnik, A. Arrasmith, P. J. Coles, and L. Cincio, *Unified approach to data-driven quantum error mitigation,* arXiv:2011.01157 (2020).

[7] L. F. Santos and L. Viola, *Dynamical control of qubit coherence: Random versus deterministic schemes,* Phys. Rev. A **72**, 062303 (2005).

[8] L. Viola and E. Knill, *Random decoupling schemes for quantum dynamical control and error suppression,* Phys. Rev. Lett. **94**, 060502 (2005).

[9] B. Pokharel, N. Anand, B. Fortman, and D. A. Lidar, *Demonstration of fidelity improvement using dynamical decoupling with superconducting qubits,* Phys. Rev. Lett. **121**, 220502 (2018).

[10] J. J. Wallman and J. Emerson, *Noise tailoring for scalable quantum computation via randomized compiling,* Phys. Rev. A **94**, 052325 (2016).

[11] J. R. McClean, Z. Jiang, N. C. Rubin, R. Babbush, and H. Neven, *Decoding quantum errors with subspace expansions,* Nature Commun. **11** (2020).

[12] 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).

[13] T. Giurgica-Tiron, Y. Hindy, R. LaRose, A. Mari, and W. J. Zeng, *Digital zero noise extrapolation for quantum error mitigation,* 2020 IEEE Int. Conf. Quantum Comp. Eng. (QCE) (2020).

[14] M. Urbanek, B. Nachman, and W. A. de Jong, *Error detection on quantum computers improving the accuracy of chemical calculations,* Phys. Rev. A **102** (2020).

[15] C. Vuillot, *Is error detection helpful on ibm 5q chips?* Quantum Inf. Comp. **18** (2018).

[16] G. A. Quantum *et al.*, *Hartree-Fock on a superconducting qubit quantum computer,* Science **369**, 1084 (2020).

[17] C. Song, J. Cui, H. Wang, J. Hao, H. Feng, and Y. Li, *Quantum computation with universal error mitigation on a superconducting quantum processor,* Science Adv. **5** (2019).

[18] S. Zhang, Y. Lu, K. Zhang, W. Chen, Y. Li, J.-N. Zhang, and K. Kim, *Error-mitigated quantum gates exceeding physical fidelities in a trapped-ion system,* Nature Communications **11**, 587 (2020).

[19] A. Ho and D. Bacon, *Announcing Cirq: An open-source framework for NISQ algorithms,* Google Blog (2018).

[20] H. Abraham *et al.*, Qiskit: An open-source framework for quantum computing, (2019).

[21] R. S. Smith, M. J. Curtis, and W. J. Zeng, A practical quantum instruction set architecture, (2016), arXiv:1608.03355 [quant-ph].

[22] Braket, https://github.com/aws/amazon-braket-sdk-python, (2021).

[23] P. Virtanen *et al.*, *SciPy 1.0: fundamental algorithms for scientific computing in Python,* Nature Meth. **17**, 261 (2020).

[24] P. J. J. O'Malley, R. Babbush, I. D. Kivlichan, J. Romero, J. R. McClean, R. Barends, J. Kelly, P. Roushan, A. Tranter, N. Ding, and et al., *Scalable quantum simulation of molecular energies,* Physical Review X **6** (2016), 10.1103/PhysRevX.6.031007, arXiv: 1512.06860.

[25] A. Sopena, M. H. Gordon, G. Sierra, and E. López, Simulating quench dynamics on a digital quantum computer with data-driven error mitigation, (2021), arXiv:2103.12680 [quant-ph].

[26] Z. Cai, *Multi-exponential error extrapolation and combining error mitigation techniques for nisq applications,* npj Quantum Inf. **7**, 80 (2021).

[27] J. Sun, X. Yuan, T. Tsunoda, V. Vedral, S. C. Benjamin, and S. Endo, *Mitigating realistic noise in practical noisy intermediate-scale quantum devices,* Phys. Rev. Applied **15**, 034026 (2021).

[28] J. R. McClean, M. E. Kimchi-Schwartz, J. Carter, and W. A. de Jong, *Hybrid quantum-classical hierarchy for mitigation of decoherence and determination of excited states,* Phys. Rev. A **95**, 042308 (2017).

[29] X. Bonet-Monroig, R. Sagastizabal, M. Singh, and T. E. O'Brien, *Low-cost error mitigation by symmetry verification,* Phys. Rev. A **98**, 062339 (2018).

[30] S. McArdle, X. Yuan, and S. Benjamin, *Error-mitigated digital quantum simulation,* Phys. Rev. Lett. **122**, 180501 (2019).

[31] R. Sagastizabal, X. Bonet-Monroig, M. Singh, M. A. Rol, C. C. Bultink, X. Fu, C. H. Price, V. P. Ostroukh, N. Muthusubramanian, A. Bruno, M. Beekman, N. Haider, T. E. O'Brien, and L. DiCarlo, *Experimental error mitigation via symmetry verification in a variational quantum eigensolver,* Phys. Rev. A **100**, 010302 (2019).

[32] B. Koczor, Exponential error suppression for near-term quantum devices, (2021), arXiv:2011.05942 [quant-ph].

[33] W. J. Huggins, S. McArdle, T. E. O'Brien, J. Lee, N. C. Rubin, S. Boixo, K. B. Whaley, R. Babbush, and J. R. McClean, Virtual distillation for quantum error mitigation, (2021), arXiv:2011.07064 [quant-ph].

[34] Z. Cai, Quantum error mitigation using symmetry expansion, (2021), arXiv:2101.03151 [quant-ph].

[35] C. Cafaro and P. van Loock, *Approximate quantum error correction for generalized amplitude-damping errors,* Phys. Rev. A **89**, 022316 (2014).

[36] M. Otten and S. K. Gray, *Recovering noise-free quantum observables,* Phys. Rev. A **99**, 012338 (2019).

[37] S. Zhou and L. Jiang, *Optimal approximate quantum error correction for quantum metrology,* Phys. Rev. Research **2**, 013235 (2020).

[38] M. Gong, X. Yuan, S. Wang, Y. Wu, Y. Zhao, C. Zha, S. Li, Z. Zhang, Q. Zhao, Y. Liu, and et al., *Experimental verification of five-qubit quantum error correction with superconducting qubits,* arXiv:1907.04507 [quant-ph] (2019).

[39] P. Schindler, J. T. Barreiro, T. Monz, V. Nebendahl, D. Nigg, M. Chwalla, M. Hennrich, and R. Blatt, *Experimental repetitive quantum error correction,* Science **332**, 1059 (2011).

[40] E. Knill, *Quantum computing with realistically noisy devices,* Nature **434**, 39 (2005).

[41] C. Brif, R. Chakrabarti, and H. Rabitz, *Control of quantum phenomena: past, present and future,* New J. Phys. **12**, 075008 (2010).

[42] L. Viola, E. Knill, and S. Lloyd, *Dynamical decoupling of open quantum systems,* Phys. Rev. Lett. **82**, 2417 (1999).

[43] H. Ball, M. Biercuk, A. Carvalho, J. Chen, M. R. Hush, L. A. de Castro, L. Li, P. J. Liebermann, H. Slatyer, C. Edmunds, V. Frey, C. Hempel, and A. Milne, *Software tools for quantum control: Improving quantum computer performance through noise and error suppression,* Quantum Sci. Tech. (2021).

[44] H. J. Carmichael, *Statistical Methods in Quantum Optics 1: Master Equations and Fokker-Planck Equations* (Springer-Verlag, 1999).

[45] H. Carmichael, *Statistical Methods in Quantum Optics 2: Non-Classical Fields* (Springer Berlin Heidelberg, 2007).

[46] H. Breuer and F. Petruccione, *The Theory of Open Quantum Systems* (OUP Oxford, 2007).

[47] P. Murali, D. C. Mckay, M. Martonosi, and A. Javadi-Abhari, *Software mitigation of crosstalk on noisy intermediate-scale quantum computers,* Proc. Twenty-Fifth Int. Conf. on Architect. Supp. for Progr. Lang. Op-

erat. Syst. (2020).
[48] I. Buluta, S. Ashhab, and F. Nori, *Natural and artificial atoms for quantum computation,* Rep. Progr. Phys. **74**, 104401 (2011).
[49] H. Silvério, S. Grijalva, C. Dalyac, L. Leclerc, P. J. Karalekas, N. Shammah, M. Beji, L.-P. Henry, and L. Henriet, Pulser: An open-source package for the design of pulse sequences in programmable neutral-atom arrays, (2021), arXiv:2104.15044 [quant-ph].
[50] B. Li, S. Ahmed, S. Saraogi, N. Lambert, F. Nori, A. Pitchford, and N. Shammah, Pulse-level noisy quantum circuits with QuTiP, (2021), arXiv:2105.09902 [quant-ph].
[51] D. Gottesman, A. Kitaev, and J. Preskill, *Encoding a qubit in an oscillator,* Phys. Rev. A **64**, 012310 (2001).
[52] M. Mirrahimi, Z. Leghtas, V. V. Albert, S. Touzard, R. J. Schoelkopf, L. Jiang, and M. H. Devoret, *Dynamically protected cat-qubits: a new paradigm for universal quantum computation,* New J. Phys. **16**, 045014 (2014).
[53] M. H. Michael, M. Silveri, R. T. Brierley, V. V. Albert, J. Salmilehto, L. Jiang, and S. M. Girvin, *New class of quantum error-correcting codes for a bosonic mode,* Phys. Rev. X **6**, 031006 (2016).
[54] V. V. Albert, J. P. Covey, and J. Preskill, *Robust encoding of a qubit in a molecule,* Physical Review X **10** (2020), 10.1103/physrevx.10.031050.
[55] J. M. Gertler, B. Baker, J. Li, S. Shirol, J. Koch, and C. Wang, *Protecting a bosonic qubit with autonomous quantum error correction,* Nature **590**, 243 (2021).
[56] D. A. Lidar, I. L. Chuang, and K. B. Whaley, *Decoherence-free subspaces for quantum computation,* Phys. Rev. Lett. **81**, 2594 (1998).
[57] E. Knill, R. Laflamme, and L. Viola, *Theory of quantum error correction for general noise,* Phys. Rev. Lett. **84**, 2525 (2000).
[58] A. F. Kockum, G. Johansson, and F. Nori, *Decoherence-free interaction between giant atoms in waveguide quantum electrodynamics,* Phys. Rev. Lett. **120**, 140404 (2018).
[59] S. Lieu, R. Belyansky, J. T. Young, R. Lundgren, V. V. Albert, and A. V. Gorshkov, *Symmetry breaking and error correction in open quantum systems,* Phys. Rev. Lett. **125**, 240405 (2020).
[60] T. A. Alexander, N. Kanazawa, D. J. Egger, L. Capelluto, C. J. Wood, A. Javadi-Abhari, and D. McKay, *Qiskit-Pulse: programming quantum computers through the cloud with pulses,* Quantum Sci. Tech. **5**, 044006 (2020).
[61] P. J. Karalekas, N. A. Tezak, E. C. Peterson, C. A. Ryan, M. P. da Silva, and R. S. Smith, *A quantum-classical cloud platform optimized for variational hybrid algorithms,* Quantum Sci. Tech. **5**, 024003 (2020).

Appendix A: Executor examples

For concreteness, in this appendix we include explicit examples of executor functions which were introduced in Sec. II B. As mentioned, an executor always accepts a quantum program, sometimes accepts other arguments, and returns an expectation value as a float.

1. Executors based on real hardware

Our first executor is the one used in creating Fig. 3(a). This executor runs a two-qubit circuit on an IBMQ quantum processor and returns the probability of the ground state.

```
import qiskit

provider = qiskit.IBMQ.load_account()

def executor(
    circuit: qiskit.QuantumCircuit,
    backend_name: str = "ibmq_santiago",
    shots: int = 1024
) -> float:
    # Execute the circuit
    job = qiskit.execute(
        experiments=circuit,
        backend=provider.get_backend(backend_name),
        optimization_level=0,
        shots=shots
    )

    # Get the measurement data
    counts = job.result().get_counts()

    # Return the observable
    return counts["00"] / shots
```

Codeblock 29. Defining an executor to run on IBMQ and return the probability of the ground state for a two-qubit circuit. Line 2 requires a valid IBMQ account with saved credentials. We assume that the input `circuit` contains terminal measurements on both qubits.

We also include the same executor function as above but this time running on Rigetti Aspen-8 and used in creating Fig. 3(b). Note that this executor requires additional steps compared to the same executor in Qiskit — namely the declaration of classical memory and the addition of measurement operations, as Rigetti QCS handles classical memory different than other platforms. Additionally, it is important to note the use of `basic_compile` from Mitiq which preserves folded gates when mapping to the native gate set of Aspen-8.

```
import pyquil
from mitiq.mitiq_pyquil.compiler import
    basic_compile

aspen8 = pyquil.get_qc("Aspen-8", as_qvm=False)

def executor(
    program: pyquil.Program,
    active_reset: bool = True,
    shots: int = 1024
) -> float:
    prog = Program()

    # Force qubits into the ground state
    if active_reset:
        prog += pyquil.gates.RESET()

    # Add the original program
    prog += program.copy()

    # Get list of qubits used in the program
```

```
    qubits = prog.get_qubits()

    # Add classical memory declaration
    ro = prog.declare("ro", "BIT", len(qubits))

    # Add measurement operations
    for idx, q in enumerate(qubits):
        prog += MEASURE(q, ro[idx])

    # Add number of shots
    prog.wrap_in_numshots_loop(shots)

    # Compile the program, keeping folded gates
    prog = basic_compile(prog)

    # Convert to an executable and run
    executable = aspen8.compiler.
     native_quil_to_executable(prog)
    results = aspen8.run(executable)

    # Return the observable
    all_zeros = [sum(b) == 0 for b in results]
    return sum(all_zeros) / shots
```

Codeblock 30. Defining an executor to run on Rigetti Aspen-8 and return the probability of the ground state. Line 3 requires a Rigetti Quantum Cloud Services (QCS) [61] account and reservation. We assume that the input `program` has no measurements, resets, or classical memory declarations.

In these examples, we see how the executor function abstracts away details about running on a back-end. This abstraction makes Mitiq compatible with multiple quantum processors using the same interface.

2. Executors based on a classical simulator

The executor function does not have to use a real quantum processor but instead can use a classical simulator.In this case, the executor is also responsible for adding noise to the circuit. The manner in which noise is added depends on the quantum programming library being used. We show below an example of an executor which adds depolarizing noise to a Cirq circuit and uses density matrix simulation. This executor inputs an arbitrary observable defined by a `cirq.PauliString` and returns its expectation value by sampling.

```
import cirq

dsim = cirq.DensityMatrixSimulator()

def executor(
    circ: Circuit,
    obs: cirq.PauliString,
    noise: float = 0.01,
    shots: int = 1024
) -> float:
    # Add depolarizing noise to the circuit
    noisy = circ.with_noise(
        cirq.depolarize(p=noise)
    )

    # Do the sampling
    psum = cirq.PauliSumCollector(
        noisy,
        obs,
        samples_per_term=shots
    )
    psum.collect(sampler=dsim)

    # Return the expectation value
    return psum.estimated_energy()
```

Codeblock 31. Cirq executor function based on a density matrix simulation with depolarizing noise and sampling. The observable is defined via `cirq.PauliString`.

Other noise models can be easily substituted into this executor by changing the channel in Line 13 from `cirq.depolarize` to a different channel, e.g. `cirq.amplitude_damp`. Executors using classical simulators in other quantum programming frameworks (e.g., Qiskit or pyQuil) can be defined in an analogous way, although each handles noise in different manners.

Finally, we note that executor functions provided to `execute_with_zne` must have only a single argument: the quantum program. The examples above include additional arguments, and it is often convenient to write executors this way. To make an executor with multiple arguments a function of one argument, we can use `functools.partial` as shown below.

```
from functools import partial

def executor(qprogram, arg1, arg2) -> float:
    ...

new_executor = partial(
    executor,
    arg1=arg1value,
    arg2=arg2value
)
```

Codeblock 32. Converting a multi-argument executor to a single-argument executor to use with `execute_with_zne`. The `functools` library is a built-in Python library.

The `new_executor` is now a function of a single argument (the quantum program) and can be used directly with `mitiq.zne.execute_with_zne` or `mitiq.pec.execute_with_pec`.