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

Ryan LaRose,[1,2] Andrea Mari,[1] Peter J. Karalekas,[1] Nathan Shammah,[1] and William J. Zeng[1,3]

[1] *Unitary Fund*
[2] *Michigan State University, East Lansing, MI*
[3] *Goldman, Sachs & Co, New York, NY*

(Dated: September 10, 2020)

We introduce an open-source software package for error mitigation in quantum computation using zero-noise extrapolation. Error mitigation techniques improve computational performance (with respect to noise) with minimal overhead in quantum resources by relying on a mixture of quantum sampling and classical post-processing techniques. Our error mitigation package interfaces with multiple quantum computing software stacks, and we demonstrate improved performance on IBM and Rigetti superconducting quantum processors as well as noisy simulators. We describe the library using code snippets to demonstrate usage and discuss features and contribution guidelines.

arXiv:2009.04417v1 [quant-ph] 9 Sep 2020

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, have been proposed and developed for this task. Among them are zero-noise extrapolation [1, 2], probabilistic error cancellation [1, 3], dynamical decoupling [4–6], randomized compiling [7], and subspace expansion [8]. In contrast to quantum error correction which has yet to be fully demonstrated in experiments, error mitigation methods have been used experimentally in several papers [9–13]. 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 front-end quantum programming languages to implement error mitigation techniques on various real and simulated quantum processors. Mitiq supports Cirq [14], Qiskit [15], and pyQuil [16] programs and any programs or back-ends that interface with these packages [17]. 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 and is designed to be modular to support additional techniques. Error mitigation methods can be implemented in few additional lines of code while 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 benchmarking results in Sec. III that demonstrate how error mitigation with Mitiq improves the performance of noisy quantum computations. In Sec. IV, we introduce the library structure and describe in detail the zero-noise extrapolation module. We discuss further software details and library information in Sec. V including future development, contribution guidelines, and planned maintenance and support. Finally, in Sec. VI we discuss the relationship between Mitiq and additional quantum error mitigation techniques.

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 `pip` by executing the instruction below at a command line.

```
pip install mitiq
```

Codeblock 1. Installing Mitiq through PyPI.

To test installation, one can run the following.

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

Codeblock 2. Testing installation & viewing package versions.

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

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

Mitiq Version:  0.1.0

Cirq Version:   0.9.0.dev
NumPy Version:  1.18.5
SciPy Version:  1.4.1

PyQuil Version: 2.21.0
Qiskit Version: 0.15.1

Python Version: 3.6.8
Platform Info:  Linux (x86_64)
Install Path:   /path/to/mitiq/installation
```

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 [18] (used for curve fitting). The remaining packages (pyQuil and Qiskit) 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 Cirq `Circuits`, Qiskit `QuantumCircuits`, and pyQuil `Programs`. 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, the `QPROGRAM` type will also include a pyQuil `Program`.

The source code for Mitiq is hosted on GitHub at

https://github.com/unitaryfund/mitiq

and is distributed with a permissive 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. V.

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. This is the noisy function whose errors Mitiq will help mitigate. We refer to this function as an *executor* because it *executes* the quantum circuit. The signature of this function should be as follows:

```python
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:

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

Example `executor` functions are shown in Sec. IV C. Because Mitiq treats the `executor` as a black box, circuits can be run on any quantum processor available to the user. We include benchmarks run on IBM and Rigetti quantum processors as well as noisy simulators in Sec. III.

Once the executor is defined, implementing zero-noise extrapolation (ZNE) needs only a single line:

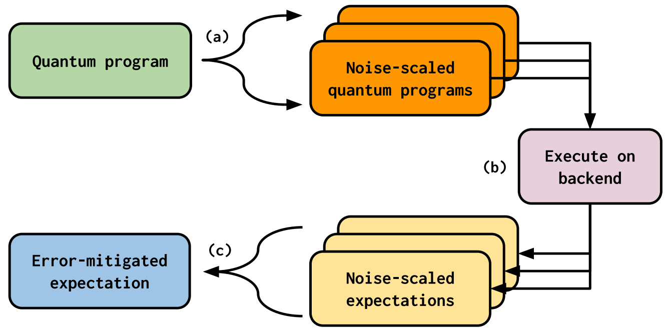

FIG. 1. Overview of the zero-noise extrapolation pipeline in Mitiq. (a) An input quantum program is converted into a set of noise-scaled programs defined by a noise scaling method and a set of scale factors. (b) These noise-scaled programs are executed on the back-end according to a user-defined executor function (see Sec. IV C for examples) and return a set of noise-scaled expectation values. (c) A classical inference technique is used to fit a curve to these noise-scaled expectation values. Once the best-fit curve is established, the zero-noise limit is returned to give an error-mitigated expectation value.

```python
from mitiq 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 1 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 [10] (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.

```python
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.

These code examples demonstrate the main usage of Mitiq. Alternatives to the `execute_with_zne` function are described in Sec. V A — these alternatives implement the same methods but offer different ways to call them which may be more convenient, depending on context. For many applications, ZNE can significantly improve the results of a noisy computation [10]. In the following section, we show results of benchmarks using Mitiq

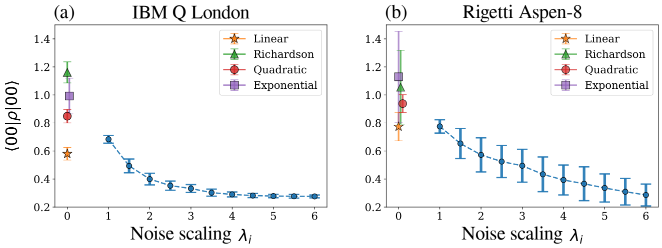

FIG. 2. Zero-noise extrapolation on two-qubit randomized benchmarking circuits run on (a) the IBMQ "London" quantum processor and (b) the Rigetti Aspen-8 quantum processor. Results are obtained from 50 randomized benchmarking circuits which contain, on average, 97 single-qubit gates and 17 two-qubit 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.

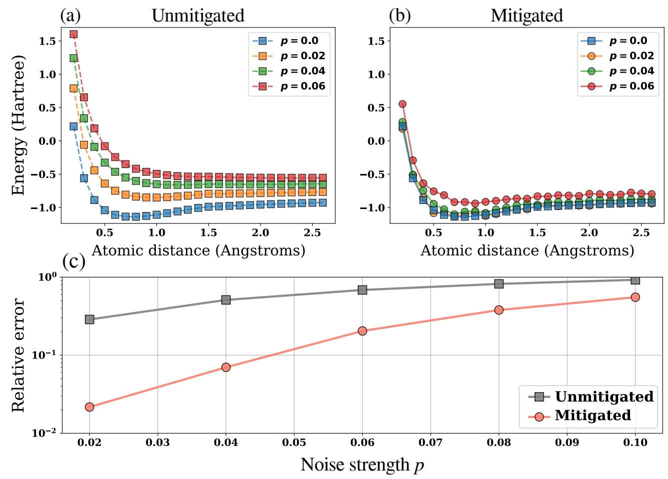

FIG. 3. Unmitigated (a) and mitigated (b) energy surfaces of H_2. The mitigated energy surfaces use zero-noise extrapolation with random local unitary folding (see Sec. IV A) and second-order polynomial inference (see Sec. IV B). Panel (c) quantifies the relative error of potential energy surfaces as the L_2 distance $||E_0(r) - E_p(r)||_2/||E_0(r)||_2$ for different (simulated) depolarizing noise strengths p.

on IBM and Rigetti quantum processors as well as noisy simulators. We then explain Mitiq's noise scaling methods and inference techniques in more detail.

III. BENCHMARKS WITH MITIQ

A. Randomized benchmarking

Figure 2 shows the results from error mitigation 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. 2(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. 2(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. VI 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. [19] 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 3(a) shows unmitigated energy surfaces at three different noise levels while Fig. 3(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. 3(c).

IV. LIBRARY STRUCTURE: ZERO-NOISE EXTRAPOLATION MODULE

We now describe the Mitiq library in more detail. An overview of its structure, shown in Fig. 4, includes two modules to interface with supported quantum programming languages (Qiskit and pyQuil) as well as a module for zero-noise extrapolation.

Zero-noise extrapolation was first introduced in [1, 2] and works by intentionally increasing (scaling) the noise of a quantum computation to then extrapolate back to

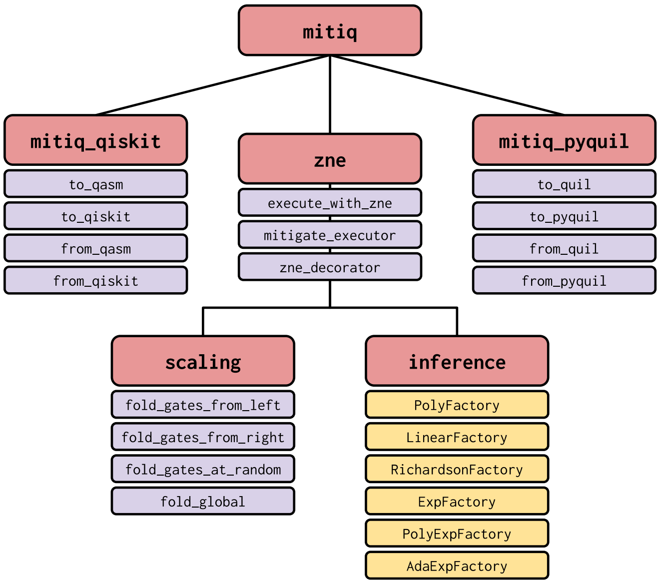

FIG. 4. Mitiq structure highlighting modules in red, functions in purple, and classes in orange. The modules `mitiq_qiskit` and `mitiq_pyquil` are used to interface with Qiskit and pyQuil, respectively, and the `zne` module contains functionality for implementing zero-noise extrapolation.

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 $\lambda_{i+1} > \lambda_i$ 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 the first formulation of zero-noise extrapolation [1], noise is scaled in superconducting processors by implementing pulses at lower amplitudes for longer time intervals. Because most quantum programming languages support gate-model circuits and not pulse-level access, it is 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 [10], as a noise scaling method.

1. *Unitary Folding*

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

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

This leaves the effect of the circuit invariant but increases its depth. If G is a subset of the gates in the circuit, we refer 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.

```python
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 7. 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 8. 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 9. Local folding from right on a Cirq circuit. The `scaling` module is imported in Codeblock 8.

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 `fold_gates_at_random` function folds gates according to the following rules:

1. Gates are selected at random and folded until the input scale factor is reached.
2. No gate is folded more than once for any scale factor less than or equal to three.

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 Qiskit [15] and pyQuil [16], and additional support will be added in the future. 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 7.

```
from mitiq.zne.scaling import fold_global

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

Codeblock 10. 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. *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 11. 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. 3(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 I, 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 I. 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. [10].

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 I. 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 12. 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 13. 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 14. 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. [10]), 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 15. 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 `BatchedFactory` (for static 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 $[-1, 1]$.

```
import numpy as np
from mitiq.zne.inference import BatchedFactory

class MyFactory(BatchedFactory):
    def reduce(self) -> float:
        # Get scale factors and exp values
        scale_factors = self.get_scale_factors()
        exp_vals = self.get_expectation_values()

        # Define the custom fit here!
        coeffs = np.polyfit(
            scale_factors, exp_vals, deg=2
        )
        zne_value = coeffs[-1]

        # Return the ZNE value
        return np.clip(zne_value, -1.0, 1.0)
```

Codeblock 16. 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.

C. Executor examples

For concreteness, we now 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 always returns an expectation value as a float.

Our first executor is the one used in creating Fig. 2(a). This executor runs a two-qubit circuit on an IBMQ quantum processor and returns the probability of the ground state.

```
1  import qiskit
2
3  provider = qiskit.IBMQ.load_account()
4
5  def executor(
6      circuit: qiskit.QuantumCircuit,
7      backend_name: str = "ibmq_london",
8      shots: int = 1024
9  ) -> float:
10     # Execute the circuit
11     job = qiskit.execute(
12         experiments=circuit,
13         backend=provider.get_backend(backend_name),
14         optimization_level=0,
15         shots=shots
16     )
17
18     # Get the measurement data
19     counts = job.result().get_counts()
20
21     # Return the observable
22     return counts["00"] / shots
```

Codeblock 17. 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. 2(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
```

```
24     ro = prog.declare("ro", "BIT", len(qubits))
25
26     # Add measurement operations
27     for idx, q in enumerate(qubits):
28         prog += MEASURE(q, ro[idx])
29
30     # Add number of shots
31     prog.wrap_in_numshots_loop(shots)
32
33     # Compile the program, keeping folded gates
34     prog = basic_compile(prog)
35
36     # Convert to an executable and run
37     executable = aspen8.compiler.
        native_quil_to_executable(prog)
38     results = aspen8.run(executable)
39
40     # Return the observable
41     all_zeros = [sum(b) == 0 for b in results]
42     return sum(all_zeros) / shots
```

Codeblock 18. 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) [17] 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.

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.

```
1  import cirq
2
3  dsim = cirq.DensityMatrixSimulator()
4
5  def executor(
6      circ: Circuit,
7      obs: cirq.PauliString,
8      noise: float = 0.01,
9      shots: int = 1024
10 ) -> float:
11     # Add depolarizing noise to the circuit
12     noisy = circ.with_noise(
13         cirq.depolarize(p=noise)
14     )
15
16     # Do the sampling
17     psum = cirq.PauliSumCollector(
18         noisy,
19         obs,
20         samples_per_term=shots
21     )
22     psum.collect(sampler=dsim)
23
24     # Return the expectation value
25     return psum.estimated_energy()
```

Codeblock 19. 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 languages (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 20. 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`.

V. ADDITIONAL LIBRARY INFORMATION

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

A. Alternative ways to use zero-noise extrapolation

Here we show two alternative methods for performing zero-noise extrapolation in Mitiq. Depending on context, these may provide simpler usage than the `execute_with_zne` function.

The first method is `mitigate_executor` which inputs the executor and the same optional arguments as `execute_with_zne` except the quantum program. This function returns a new executor which implements zero-noise extrapolation 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 21. Modifying an `executor` using the function `mitigate_executor`. The new `mitigated_executor` performs zero-noise extrapolation when called on a quantum program.

The second method is to decorate the `executor` with `mitiq.zne.zne_decorator` such that it automatically performs zero-noise extrapolation when called.

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

Again 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, i.e. `@zne_decorator()`.

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 self-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.

VI. 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 relation between zero-noise extrapolation and additional strategies.

A. Limitations of zero-noise extrapolation

Zero-noise extrapolation 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. 2) 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 2(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. Relation to other error mitigation techniques

Zero-noise extrapolation is one of several methods for quantum error mitigation. It was first proposed in [1, 2] and first demonstrated experimentally in [9]. References [10, 20] 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 [7], expressing noiseless gates in a basis of noisy gates as in probabilistic error cancellation [1], and a hybrid proposal improving the scaling of the technique with circuit depth and other resources [3]. Such techniques have been investigated experimentally in trapped ions [21] and superconducting qubits [22] (implementing gate set tomography).

Subspace expansion refers to another set of error mitigation techniques. In Ref. [23], a hybrid quantum-classical hierarchy was introduced, while in Ref. [24], symmetry verification was introduced. It has been demonstrated with a stabilizer-like method [25], exploiting molecular symmetries [8], and with an experiment on a superconducting circuit device [26]. Other error mitigation techniques include approximating error-correcting codes in quantum channels [27], and have been extended to improve quantum sensing [28], metrology [29], and reduce errors in analog quantum simulation [22].

C. Relation to quantum error correction

Quantum error mitigation is 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. While these are fluid boundaries, it can be useful to point out some differences among these two well-established fields and the emerging field of quantum error mitigation.

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 [30, 31]. Indeed, some techniques for quantum error mitigation emerged as more practical quantum error correction solutions [32].

D. Relation to quantum optimal control

Optimal control theory encompasses a versatile set of techniques that can be applied to many scenarios in quantum technology [33]. 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 [4–6]. This technique employs fast control pulses to effectively decouple a system from its environment, with techniques pioneered in the nuclear magnetic resonance community [34]. Quantum optimal control techniques are being integrated in the quantum computing software as a means for and noise characterization and error mitigation [35].

E. Relation to the theory of open quantum systems

More in general, quantum computing devices can be studied in the framework of open quantum systems [36–38], that is, systems that exchange energy and information with the surrounding environment. On one hand, the qubit-environment exchange can be controlled, and this feature is actually fundamental to extract information and process it. On the other hand, when this interaction is not controlled — and at the fundamental level it cannot be completely suppressed — noise eventually kicks in, thus introducing errors that are disruptive for the fidelity of the information-processing protocols.

Indeed, issues arise in quantum computation due to the fact that quantum computers are devices that are embedded in an environment and interact with it. This means that stored information can be corrupted or that desired programs are not necessarily faithfully executed during a computation.

Errors occur for several reasons in quantum computers, and 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. For example, superconducting-circuit-based quantum computers have chips that are prone to cross-talk noise [39], 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 [40].

VII. CONCLUSION

We have introduced a fully open-source library for quantum error mitigation using zero-noise extrapolation. Our library can interface with multiple quantum programming libraries — in particular Cirq, Qiskit, and pyQuil — and arbitrary quantum processors (real or simulated) available to the user. In this paper, we have demonstrated improved quantum computation using zero-noise extrapolation on two benchmarks. We then discussed the library in detail, demonstrating through code examples the noise scaling methods and inference techniques in the zero-noise extrapolation module. 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 zero-noise extrapolation module, for example by implementing different noise-scaling methods or inference techniques. 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 [41]. 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. 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] K. Temme, S. Bravyi, and J. M. Gambetta, *Error mitigation for short-depth quantum circuits,* Physical Review Letters **119** (2017), 10.1103/physrevlett.119.180509.

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

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

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

[5] L. Viola and E. Knill, *Random decoupling schemes for quantum dynamical control and error suppression,* Physical review letters **94**, 060502 (2005).

[6] B. Pokharel, N. Anand, B. Fortman, and D. A. Lidar, *Demonstration of fidelity improvement using dynamical decoupling with superconducting qubits,* Physical Review Letters **121**, 220502 (2018).

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

[8] J. R. McClean, Z. Jiang, N. C. Rubin, R. Babbush, and H. Neven, *Decoding quantum errors with subspace expansions,* Nature Communications **11** (2020), 10.1038/s41467-020-14341-w.

[9] 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).

[10] T. Giurgica-Tiron, Y. Hindy, R. LaRose, A. Mari, and W. J. Zeng, *Digital zero noise extrapolation for quantum error mitigation,* arXiv preprint arXiv:2005.10921 (2020).

[11] M. Urbanek, B. Nachman, and W. A. de Jong, *Quantum error detection improves accuracy of chemical calculations on a quantum computer,* arXiv:1910.00129 [physics, physics:quant-ph] (2019), arXiv: 1910.00129.

[12] C. Vuillot, *Is error detection helpful on ibm 5q chips?* Quantum Information and Computation **18**, 10.26421/QIC18.11-12, arXiv: 1705.08957.

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

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

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

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

[17] 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 Science and Technology **5**, 024003 (2020).

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

[19] 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.

[20] Z. Cai, *Multi-exponential error extrapolation and combining error mitigation techniques for nisq applications,* arXiv preprint arXiv:2007.01265 (2020).

[21] 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** (2020), 10.1038/s41467-020-14376-z.

[22] J. Sun, X. Yuan, T. Tsunoda, V. Vedral, S. C. Bejamin, and S. Endo, Practical quantum error mitigation for analog quantum simulation, (2020), arXiv:2001.04891 [quant-ph].

[23] 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).

[24] 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).

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

[26] 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).

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

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

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

[30] 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), arXiv: 1907.04507.

[31] 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**, 1059–1061 (2011).

[32] E. Knill, *Quantum computing with realistically noisy devices,* Nature **434**, 39–44 (2005).

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

[34] L. Viola, E. Knill, and S. Lloyd, *Dynamical decoupling of open quantum systems,* Physical Review Letters **82**, 2417–2421 (1999).

[35] H. Ball, M. J. Biercuk, A. Carvalho, R. Chakravorty, J. Chen, L. A. de Castro, S. Gore, D. Hover, M. Hush, P. J. Liebermann, *et al.*, *Software tools for quantum control: Improving quantum computer performance through noise and error suppression,* arXiv preprint arXiv:2001.04060 (2020).

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

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

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

[39] P. Murali, D. C. Mckay, M. Martonosi, and A. Javadi-Abhari, *Software mitigation of crosstalk on noisy intermediate-scale quantum computers,* Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (2020), 10.1145/3373376.3378477.

[40] I. Buluta, S. Ashhab, and F. Nori, *Natural and artificial atoms for quantum computation,* Reports on Progress in Physics **74**, 104401 (2011).

[41] 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. and Tech. (2020), 10.1088/2058-9565/aba404.