mitiq

Release 0.4.0

Tech Team @ Unitary Fund

Dec 06, 2020

Contents:

CHAPTER 1

Mitiq

Mitiq is a Python toolkit for implementing error mitigation techniques on quantum computers.

Current quantum computers are noisy due to interactions with the environment, imperfect gate applications, state preparation and measurement errors, etc. Error mitigation seeks to reduce these effects at the software level by compiling quantum programs in clever ways.

Want to know more? Check out our documentation.

1.1 Installation

Mitiq can be installed from PyPi via

```
pip install mitiq
```

To test installation, run

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

This prints out version information about core requirements and optional quantum software packages which Mitiq can interface with.

If you'd prefer to clone and install from source, our would like to develop Mitiq, check out the contribution guidelines for more information.

1.1.1 Supported quantum programming libraries

Mitiq can currently interface with

- Cirq >= 0.9.0,
- Qiskit >= 0.19.0, and
- pyQuil >= 2.18.0.

Cirq is a core requirement of Mitiq and is automatically installed. To use Mitiq with other quantum programming libraries, install the optional package(s) following the instructions linked above.

1.1.2 Supported quantum processors

Mitiq can be used on any quantum processor which can be accessed by supported quantum programming libraries and is available to the user.

1.2 Getting started

See the getting started guide in Mitiq's documentation for a complete walkthrough of how to use mitiq. For a quick preview, check out the following snippet for a simple example of Mitiq in action:

```
import numpy as np
from cirq import depolarize, Circuit, DensityMatrixSimulator, LineQubit, X
from mitiq import execute_with_zne

def noisy_simulation(circ: Circuit) -> float:
    """Simulates a circuit with depolarizing noise.

    Args:
        circ: The quantum program as a Cirq Circuit.

    Returns:
        The expectation value of the |0><0| observable.
    """
    circuit = circ.with_noise(depolarize(p=0.001))
    rho = DensityMatrixSimulator().simulate(circuit).final_density_matrix
    return np.real(np.trace(rho @ np.diag([1, 0])))

# simple circuit that should compose to the identity when noiseless
circ = Circuit(X(LineQubit(0)) for _ in range(80))

# run the circuit using a density matrix simulator with depolarizing noise
unmitigated = noisy_simulation(circ)
print(f"Error in simulation (w/o  mitigation): {1.0 - unmitigated:.{3}}")

# run again, but using mitiq's zero-noise extrapolation to mitigate errors
mitigated = execute_with_zne(circ, noisy_simulation)
print(f"Error in simulation (with mitigation): {1.0 - mitigated:.{3}}")
```

Sample output:

```
Error in simulation (w/o  mitigation): 0.0506
Error in simulation (with mitigation): 0.000519
```

1.3 Error mitigation techniques

Mitiq currently implements zero-noise extrapolation and is designed to support additional techniques.

1.4 Documentation

Mitiq's documentation is hosted at mitiq.readthedocs.io. A PDF version of the latest release can be found here.

1.5 Developer information

We welcome contributions to Mitiq including bug fixes, feature requests, etc. Please see the contribution guidelines for more details. To contribute to the documentation, please see these documentation guidelines.

1.6 Authors

An up-to-date list of authors can be found here.

1.7 Citation

If you use Mitiq in your research, please reference the Mitiq preprint as follows:

```
@misc{larose2020mitiq,
    title={Mitiq: A software package for error mitigation on noisy quantum computers},
    author={Ryan LaRose and Andrea Mari and Peter J. Karalekas
            and Nathan Shammah and William J. Zeng},
    year={2020},
    eprint={2009.04417},
    archivePrefix={arXiv},
    primaryClass={quant-ph}
}
```

1.8 License

GNU GPL v.3.0.

CHAPTER 2

Users Guide

2.1 Overview of mitiq

Welcome to the *mitiq* Users Guide.

2.1.1 What is mitiq for?

Today's quantum computers have a lot of noise. This is a problem for quantum programmers everywhere. Mitiq is an open source Python library currently under development by Unitary Fund. It helps solve this problem by compiling your programs to be more robust to noise.

Mitiq helps you do more quantum programming with less quantum compute.

The current Mitiq library is based around the zero-noise extrapolation technique. These references [2][3] give background on the technique. The implementation in mitiq is an optimized, extensible framework for zero-noise extrapolation. In the future other error-mitigating techniques will be added to Mitiq.

Mitiq is a framework agnostic library with a long term vision to be useful for quantum programmers using any quantum programming framework and any quantum backend. Today we support Cirq, Qiskit, and PyQuil frontends and backends.

Check out more in our getting started section.

2.2 Getting Started

Improving the performance of your quantum programs is only a few lines of code away.

This getting started shows examples using cirq cirq and qiskit. We'll first test mitiq by running against the noisy simulator built into cirq. The qiskit example works similarly as you will see in *Qiskit Mitigation*.

2.2.1 Multi-platform Framework

In mitiq, a "back-end" is a function that executes quantum programs. A "front-end" is a library/language that constructs quantum programs. mitiq lets you mix and match these. For example, you could write a quantum program in qiskit and then execute it using a cirq backend, or vice versa.

Back-ends are abstracted to functions called executors that always accept a quantum program, sometimes accept other arguments, and always return an expectation value as a float. You can see some examples of different executors for common packages *here* and in this getting started. If your quantum programming interface of choice can be used to make a Python function with this type, then it can be used with mitiq.

2.2.2 Error Mitigation with Zero-Noise Extrapolation

We define some functions that make it simpler to simulate noise in cirq. These don't have to do with mitiq directly.

```
import numpy as np
from cirq import Circuit, depolarize
from cirq import LineQubit, X, DensityMatrixSimulator

SIMULATOR = DensityMatrixSimulator()
# 0.1% depolarizing noise
NOISE = 0.001

def noisy_simulation(circ: Circuit) -> float:
    """ Simulates a circuit with depolarizing noise at level NOISE.
    Args:
        circ: The quantum program as a cirq object.

    Returns:
        The expectation value of the |0> state.
    """
    circuit = circ.with_noise(depolarize(p=NOISE))
    rho = SIMULATOR.simulate(circuit).final_density_matrix
    # define the computational basis observable
    obs = np.diag([1, 0])
    expectation = np.real(np.trace(rho @ obs))
    return expectation
```

Now we can look at our example. We'll test single qubit circuits with even numbers of X gates. As there are an even number of X gates, they should all evaluate to an expectation of 1 in the computational basis if there was no noise.

```
from cirq import Circuit, LineQubit, X

qbit = LineQubit(0)
circ = Circuit(X(qbit) for _ in range(80))
unmitigated = noisy_simulation(circ)
exact = 1
print(f"Error in simulation is {exact - unmitigated:.{3}}")
```

```
Error in simulation is 0.0506
```

This shows the impact the noise has had. Let's use `mitiq` to improve this performance.

```
from mitiq import execute_with_zne

mitigated = execute_with_zne(circ, noisy_simulation)
print(f"Error in simulation is {exact - mitigated:.{3}}")
```

```
Error in simulation is 0.000519
```

```
print(f"Mitigation provides a {(exact - unmitigated) / (exact - mitigated):.{3}}␣
↪factor of improvement.")
```

```
Mitigation provides a 97.6 factor of improvement.
```

You can also use `mitiq` to wrap your backend execution function into an error-mitigated version.

```
from mitiq import mitigate_executor

run_mitigated = mitigate_executor(noisy_simulation)
mitigated = run_mitigated(circ)
print(round(mitigated,5))
```

```
0.99948
```

Note: As shown here, `mitiq` wraps executor functions that have a specific type: they take quantum programs as input and return expectation values. However, one often has an execution function with other arguments such as the number of shots, the observable to measure, or the noise level of a noisy simulation. It is still easy to use these with mitiq by using partial function application. Here's a pseudo-code example:

```
from functools import partial

def shot_executor(qprogram, n_shots) -> float:
    ...
# we partially apply the n_shots argument to get a function that just
# takes a quantum program
mitigated = execute_with_zne(circ, partial(shot_executor, n_shots=100))
```

You can read more about functools partial application here.

The default implementation uses Richardson extrapolation to extrapolate the expectation value to the zero noise limit [2]. `Mitiq` comes equipped with other extrapolation methods as well. Different methods of extrapolation are packaged into `Factory` objects. It is easy to try different ones.

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

fac = LinearFactory(scale_factors=[1.0, 2.0, 2.5])
linear = execute_with_zne(circ, noisy_simulation, factory=fac)
print(f"Mitigated error with the linear method is {exact - linear:.{3}}")
```

```
Mitigated error with the linear method is 0.00638
```

You can read more about the `Factory` objects that are built into `mitiq` and how to create your own here.

Another key step in zero-noise extrapolation is to choose how your circuit is transformed to scale the noise. You can read more about the noise scaling methods built into `mitiq` and how to create your own here.

2.2.3 Qiskit Mitigation

`Mitiq` is designed to be agnostic to the stack that you are using. Thus for `qiskit` things work in the same manner as before. Since we are now using `qiskit`, we want to run the error mitigated programs on a qiskit backend. Let's define the new backend that accepts `qiskit` circuits. In this case it is a simulator, but you could also use a QPU.

```
import qiskit
from qiskit import QuantumCircuit

# Noise simulation packages
from qiskit.providers.aer.noise import NoiseModel
from qiskit.providers.aer.noise.errors.standard_errors import depolarizing_error

# 0.1% depolarizing noise
NOISE = 0.001

QISKIT_SIMULATOR = qiskit.Aer.get_backend("qasm_simulator")

def qs_noisy_simulation(circuit: QuantumCircuit, shots: int = 4096) -> float:
    """Runs the quantum circuit with a depolarizing channel noise model at
    level NOISE.

    Args:
        circuit (qiskit.QuantumCircuit): Ideal quantum circuit.
        shots (int): Number of shots to run the circuit
                     on the back-end.

    Returns:
        expval: expected values.
    """
    # initialize a qiskit noise model
    noise_model = NoiseModel()

    # we assume a depolarizing error for each
    # gate of the standard IBM basis
    noise_model.add_all_qubit_quantum_error(depolarizing_error(NOISE, 1), ["u1", "u2",
↪ "u3"])

    # execution of the experiment
    job = qiskit.execute(
        circuit,
        backend=QISKIT_SIMULATOR,
        basis_gates=["u1", "u2", "u3"],
        # we want all gates to be actually applied,
        # so we skip any circuit optimization
        optimization_level=0,
        noise_model=noise_model,
        shots=shots,
        seed_transpiler=1,
```

(continues on next page)

(continued from previous page)

```
        seed_simulator=1
    )
    results = job.result()
    counts = results.get_counts()
    expval = counts["0"] / shots
    return expval
```

We can then use this backend for our mitigation.

```
from qiskit import QuantumCircuit
from mitiq import execute_with_zne

circ = QuantumCircuit(1, 1)
for _ in range(100):
     _ = circ.x(0)
_ = circ.measure(0, 0)

exact = 1
unmitigated = qs_noisy_simulation(circ)
mitigated = execute_with_zne(circ, qs_noisy_simulation)

# The mitigation should improve the result.
assert abs(exact - mitigated) < abs(exact - unmitigated)
```

Note that we don't need to even redefine factories for different stacks. Once you have a `Factory` it can be used with different front and backends.

2.3 Back-end Plug-ins: Executor Examples

`Mitiq` uses `executor` functions to abstract different backends. `Executors` always accept a quantum program, sometimes accept other arguments, and always return an expectation value as a float. If your quantum programming interface of choice can be used to make a Python function with this type, then it can be used with mitiq.

These example executors as especially flexible as they accept an arbitrary observable. You can instead hardcode your choice of observable in any way you like. All that matters from `mitiq`'s perspective is that your executor accepts a quantum program and returns a float.

2.3.1 Cirq Executors

This section includes noisy and noiseless simulator executor examples using `cirq`.

Cirq: Wavefunction Simulation

This executor can be used for noiseless simulation. Note that this executor can be *wrapped using partial function application* to be used in `mitiq`.

```
import numpy as np
from cirq import Circuit

def wvf_sim(circ: Circuit, obs: np.ndarray) -> float:
    """Simulates noiseless wavefunction evolution and returns the
    expectation value of some observable.
```

(continues on next page)

(continued from previous page)

```
    Args:
        circ: The input Cirq circuit.
        obs: The observable to measure as a NumPy array.

    Returns:
        The expectation value of obs as a float.
    """
    final_wvf = circ.final_wavefunction()
    return np.real(final_wvf.conj().T @ obs @ final_wvf)
```

Cirq: Wavefunction Simulation with Sampling

We can add in functionality that takes into account some finite number of samples (aka shots). Here we will use `cirq`'s *PauliString* methods to construct our observable. You can read more about these methods in the `cirq` documentation here.

```
def wvf_sampling_sim(circ: Circuit, obs: cirq.PauliString, shots: int) -> float:
    """Simulates noiseless wavefunction evolution and returns the
    expectation value of a PauliString observable.

    Args:
        circ: The input Cirq circuit.
        obs: The observable to measure as a cirq.PauliString.
        shots: The number of measurements.

    Returns:
        The expectation value of obs as a float.
    """

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

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

Cirq: Density-matrix Simulation with Depolarizing Noise

This executor can be used for noisy depolarizing simulation.

```
import numpy as np
from cirq import Circuit, depolarize
from cirq import DensityMatrixSimulator

SIMULATOR = DensityMatrixSimulator()

def noisy_sim(circ: Circuit, obs: np.ndarray, noise: float) -> float:
    """Simulates a circuit with depolarizing noise at level noise.

    Args:
        circ: The input Cirq circuit.
        obs: The observable to measure as a NumPy array.
```

(continues on next page)

(continued from previous page)

```
        noise: The depolarizing noise as a float, i.e. 0.001 is 0.1% noise.

    Returns:
        The expectation value of obs as a float.
    """
    circuit = circ.with_noise(depolarize(p=noise))
    rho = SIMULATOR.simulate(circuit).final_density_matrix
    expectation = np.real(np.trace(rho @ obs))
    return expectation
```

Other noise models can be used by substituting the `depolarize` channel with any other channel available in `cirq`, for example `cirq.amplitude_damp`. More details can be found in the `cirq` noise documentation

Cirq: Density-matrix Simulation with Depolarizing Noise and Sampling

You can also include both noise models and finite sampling in your executor.

```
import numpy as np
from cirq import Circuit, depolarize
from cirq import DensityMatrixSimulator

SIMULATOR = DensityMatrixSimulator()

def noisy_sample_sim(circ: Circuit, obs: cirq.PauliString, noise: float, shots: int) -
↪> float:
    """Simulates a circuit with depolarizing noise at level noise.

    Args:
        circ: The input Cirq circuit.
        obs: The observable to measure as a NumPy array.
        noise: The depolarizing noise strength as a float, i.e. 0.001 is 0.1%.
        shots: The number of measurements.

    Returns:
        The expectation value of obs as a float.
    """
    # add the noise
    noisy = circ.with_noise(depolarize(p=noise))

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

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

2.3.2 PyQuil Executors

This section contains executors for use with pyQuil.

PyQuil: Quantum Cloud Services

This executor can be used to run on Quantum Cloud Services (QCS), the hardware platform provided by Rigetti Computing. Requires a QCS account and reservation on a quantum processor (QPU).

In addition, `mitiq_pyquil/executors.py` has a function `generate_qcs_executor` for easily generating a QCS executor of this form from a template.

Note that you will have to replace the string in `get_qc` with the name of an actual Rigetti QPU, and will need to have a QCS account and reservation, in order to run on real quantum hardware.

```
from pyquil import Program, get_qc
from pyquil.gates import MEASURE, RESET, X

from mitiq.mitiq_pyquil.compiler import basic_compile
from mitiq.mitiq_pyquil.pyquil_utils import ground_state_expectation

# replace with qpu = get_qc("Aspen-8") to run on the Aspen-8 QPU
qpu = get_qc("2q-pyqvm")

def executor(program: Program, shots: int = 1000) -> float:
    p = Program()

    # add reset
    p += RESET()

    # add main body program
    p += program.copy()

    # add memory declaration
    qubits = p.get_qubits()
    ro = p.declare("ro", "BIT", len(qubits))

    # add measurements
    for idx, q in enumerate(qubits):
        p += MEASURE(q, ro[idx])

    # add numshots
    p.wrap_in_numshots_loop(shots)

    # nativize the circuit
    p = basic_compile(p)

    # compile the circuit
    b = qpu.compiler.native_quil_to_executable(p)

    # run the circuit, collect bitstrings
    qpu.reset()
    results = qpu.run(b)

    # compute ground state expectation value
    return ground_state_expectation(results)
```

(continues on next page)

(continued from previous page)

```
# prepare state |11>
program = Program()
program += X(0)
program += X(1)

# should give 0.0 with a noiseless backend
executor(program)
```

2.3.3 Qiskit Executors

This section includes noisy and noiseless simulator executor examples using `qiskit`.

Qiskit: Wavefunction Simulation

This executor can be used for noiseless simulation. Note that this executor can be *wrapped using partial function application* to be used in `mitiq`.

```
import numpy as np
import qiskit
from qiskit import QuantumCircuit

wvf_simulator = qiskit.Aer.get_backend('statevector_simulator')

def qs_wvf_sim(circ: QuantumCircuit, obs: np.ndarray) -> float:
    """Simulates noiseless wavefunction evolution and returns the
    expectation value of some observable.

    Args:
        circ: The input Qiskit circuit.
        obs: The observable to measure as a NumPy array.

    Returns:
        The expectation value of obs as a float.
    """
    result = qiskit.execute(circ, wvf_simulator).result()
    final_wvf = result.get_statevector()
    return np.real(final_wvf.conj().T @ obs @ final_wvf)
```

Qiskit: Wavefunction Simulation with Sampling

The above executor can be modified to still perform exact wavefunction simulation, but to also include finite sampling of measurements. Note that this executor can be *wrapped using partial function application* to be used in `mitiq`.

Note that this executor implementation measures arbitrary observables by using a change of basis into the computational basis. More information about the math behind how this example is available here.

```
import copy

QISKIT_SIMULATOR = qiskit.Aer.get_backend("qasm_simulator")

def qs_wvf_sampling_sim(circ: QuantumCircuit, obs: np.ndarray, shots: int) -> float:
    """Simulates the evolution of the circuit and returns
```

(continues on next page)

(continued from previous page)

```
    the expectation value of the observable.

    Args:
        circ: The input Qiskit circuit.
        obs: The observable to measure as a NumPy array.
        shots: The number of measurements.

    Returns:
        The expectation value of obs as a float.

    """
    if len(circ.clbits) > 0:
        raise ValueError("This executor only works on programs with no classical bits.
↪")

    circ = copy.deepcopy(circ)
    # we need to modify the circuit to measure obs in its eigenbasis
    # we do this by appending a unitary operation
    eigvals, U = np.linalg.eigh(obs) # obtains a U s.t. obs = U diag(eigvals) U^dag
    circ.unitary(np.linalg.inv(U), qubits=range(circ.n_qubits))

    circ.measure_all()

    # execution of the experiment
    job = qiskit.execute(
        circ,
        backend=QISKIT_SIMULATOR,
        # we want all gates to be actually applied,
        # so we skip any circuit optimization
        optimization_level=0,
        shots=shots
    )
    results = job.result()
    counts = results.get_counts()
    expectation = 0
    # classical bits are included in bitstrings with a space
    # this is what breaks if you have them
    for bitstring, count in counts.items():
        expectation += eigvals[int(bitstring, 2)] * count / shots
    return expectation
```

Qiskit: Density-matrix Simulation with Depolarizing Noise

TODO

Qiskit: Density-matrix Simulation with Depolarizing Noise and Sampling

This executor can be used for noisy depolarizing simulation.

```
import qiskit
from qiskit import QuantumCircuit
import numpy as np
import copy

# Noise simulation packages
from qiskit.providers.aer.noise import NoiseModel
from qiskit.providers.aer.noise.errors.standard_errors import depolarizing_error

QISKIT_SIMULATOR = qiskit.Aer.get_backend("qasm_simulator")

def qs_noisy_sampling_sim(circ: QuantumCircuit, obs: np.ndarray, noise: float, shots:␣
↪int) -> float:
    """Simulates the evolution of the noisy circuit and returns
    the expectation value of the observable.

    Args:
        circ: The input Cirq circuit.
        obs: The observable to measure as a NumPy array.
        noise: The depolarizing noise strength as a float, i.e. 0.001 is 0.1%.
        shots: The number of measurements.

    Returns:
        The expectation value of obs as a float.
    """
    if len(circ.clbits) > 0:
        raise ValueError("This executor only works on programs with no classical bits.
↪")

    circ = copy.deepcopy(circ)
    # we need to modify the circuit to measure obs in its eigenbasis
    # we do this by appending a unitary operation
    eigvals, U = np.linalg.eigh(obs) # obtains a U s.t. obs = U diag(eigvals) U^dag
    circ.unitary(np.linalg.inv(U), qubits=range(circ.n_qubits))

    circ.measure_all()

    # initialize a qiskit noise model
    noise_model = NoiseModel()

    # we assume the same depolarizing error for each
    # gate of the standard IBM basis
    noise_model.add_all_qubit_quantum_error(depolarizing_error(noise, 1), ["u1", "u2",
↪ "u3"])
    noise_model.add_all_qubit_quantum_error(depolarizing_error(noise, 2), ["cx"])

    # execution of the experiment
    job = qiskit.execute(
        circ,
        backend=QISKIT_SIMULATOR,
        backend_options={'method':'density_matrix'},
        noise_model=noise_model,
        # we want all gates to be actually applied,
        # so we skip any circuit optimization
```

(continues on next page)

(continued from previous page)

```
        basis_gates=noise_model.basis_gates,
        optimization_level=0,
        shots=shots,
    )
    results = job.result()
    counts = results.get_counts()
    expectation = 0
    # classical bits are included in bitstrings with a space
    # this is what breaks if you have them
    for bitstring, count in counts.items():
        expectation += eigvals[int(bitstring, 2)] * count / shots
    return expectation
```

Other noise models can be defined using any functionality available in `qiskit`. More details can be found in the `qiskit` simulator documentation

Qiskit: Hardware

An example of an executor that runs on IBMQ hardware is given *here*.

2.3.4 TensorFlow Quantum Executors

This section provides an example of how to use TensorFlow Quantum as an executor. Note that at the time of this writing, TensorFlow Quantum is limited to

1. `Cirq` `Circuits` that use `Cirq` `GridQubit` instances
2. Unitary circuits only, so non-unitary errors need to use Monte Carlo simulations

Despite this latter limitation, there is a crossover point where Monte Carlo using Tensorflow evaluates faster than the exact density matrix simulation using `Cirq`.

Below is an example to use TensorFlow Quantum to simulate a bit-flip channel:

```
import numpy as np
import sympy
# tensorflow-quantum 0.4.0 is unavailable on Windows
try:
    import tensorflow as tf
    import tensorflow_quantum as tfq
    tfq_exists = True
except ImportError:
    tfq_exists = False
from cirq import Circuit


def stochastic_bit_flip_simulation(circ: Circuit, p: float, num_monte_carlo: int =
↪100) -> float:
    """
    Simulates a circuit with random bit flip (X(\pi)) errors
    Args:
        circ: The quantum program as a cirq object
        p: probability of an X(\pi) gate on each qubit after each gate in circ
        num_monte_carlo: number of random trajectories to average over
    Returns:
```

(continues on next page)

(continued from previous page)

```
        The expectation value of the 0 state as a float
    """
    nM = len(circ.moments)
    nQ = len(circ.all_qubits())

    # Create array of symbolic variables and reshape to natural circuit␣
↪parameterization
    h = sympy.symbols(''.join(['h_{0} '.format(i) for i in range(nM * nQ)]),␣
↪positive=True)
    h_array = np.asarray(h).reshape((nQ, nM))

    # Symbolicly add X gates to the input circuit
    noisy_circuit = Circuit()
    for i, moment in enumerate(circ.moments):
        noisy_circuit.append(moment)
        for j, q in enumerate(circ.all_qubits()):
            noisy_circuit.append(cirq.rx(h_array[j, i]).on(q))

    # rotations will be pi w/ prob p, 0 w/ prob 1-p
    vals = [np.reshape((np.random.rand(nQ, nM) < p) * np.pi, (1, nQ * nM)) for _ in␣
↪range(num_monte_carlo)]

    # needs to be a rank 2 tensor
    vals = np.squeeze(vals)
    if num_monte_carlo == 1:
        vals = [vals]

    # Instantiate tfq layer for computing state vector
    state = tfq.layers.State()

    # Execute monte carlo sim with symbolic values specified by vals
    out = state(noisy_circuit, symbol_names=h, symbol_values=vals).to_tensor()

    # Fancy way of computing and summing individual density operators, follwed by␣
↪averaging
    dm = tf.tensordot(tf.transpose(out), tf.math.conj(out), axes=[[1], [0]]).numpy() /
↪ num_monte_carlo

    # return measurement of 0 state
    return np.real(dm[0, 0])
```

2.4 Zero Noise Extrapolation

Zero noise extrapolation has two main components: noise scaling and then extrapolation.

2.4.1 Digital noise scaling: Unitary Folding

Zero noise extrapolation has two main components: noise scaling and then extrapolation. Unitary folding is a method for noise scaling that operates directly at the gate level. This makes it easy to use across platforms. It is especially appropriate when your underlying noise should scale with the depth and/or the number of gates in your quantum program. More details can be found in [27] where the unitary folding framework was introduced.

At the gate level, noise is amplified by mapping gates (or groups of gates) G to

$$G \mapsto GG^\dagger G.$$

This makes the circuit longer (adding more noise) while keeping its effect unchanged (because $G^\dagger = G^{-1}$ for unitary gates). We refer to this process as *unitary folding*. If G is a subset of the gates in a circuit, we call it *local folding*. If G is the entire circuit, we call it *global folding*.

In `mitiq`, folding functions input a circuit and a *scale factor* (or simply *scale*), i.e., a floating point value which corresponds to (approximately) how much the length of the circuit is scaled. The minimum scale factor is one (which corresponds to folding no gates). A scale factor of three corresponds to folding all gates locally. Scale factors beyond three begin to fold gates more than once.

Local folding methods

For local folding, there is a degree of freedom for which gates to fold first. The order in which gates are folded can have an important effect on how the noise is caled. As such, `mititq` defines several local folding methods.

We introduce three folding functions:

1. `mitiq.zne.scaling.fold_gates_from_left`
2. `mitiq.zne.scaling.fold_gates_from_right`
3. `mitiq.zne.scaling.fold_gates_at_random`

The `mitiq` function `fold_gates_from_left` will fold gates from the left (or start) of the circuit until the desired scale factor is reached.

```
>>> import cirq
>>> from mitiq.zne.scaling import fold_gates_from_left

# Get a circuit to fold
>>> 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───

# Fold the circuit
>>> folded = fold_gates_from_left(circ, scale_factor=2.)
>>> print("Folded circuit:", folded, sep="\n")
Folded circuit:
0: ───H───H───H───@───
                  │
1: ───────────────X───
```

In this example, we see that the folded circuit has the first (Hadamard) gate folded.

Note: mitiq folding functions do not modify the input circuit.

Because input circuits are not modified, we can reuse this circuit for the next example. In the following code, we use the fold_gates_from_right function on the same input circuit.

```
>>> from mitiq.zne.scaling import fold_gates_from_right

# Fold the circuit
>>> folded = fold_gates_from_right(circ, scale_factor=2.)
>>> print("Folded circuit:", folded, sep="\n")
Folded circuit:
0: ───H───@───@───@───
          │   │   │
1: ───────X───X───X───
```

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

Finally, we mention fold_gates_at_random which 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 <= 3.
3. "Virtual gates" (i.e., gates appearing from folding) are never folded.

All of these local folding methods can be called with any scale_factor >= 1.

Any supported circuits can be folded

Any program types supported by mitiq can be folded, and the interface for all folding functions is the same. In the following example, we fold a Qiskit circuit.

Note: This example assumes you have Qiskit installed. mitiq can interface with Qiskit, but Qiskit is not a core mitiq requirement and is not installed by default.

```
>>> import qiskit
>>> from mitiq.zne.scaling import fold_gates_from_left

# Get a circuit to fold
>>> qreg = qiskit.QuantumRegister(2)
>>> circ = qiskit.QuantumCircuit(qreg)
>>> _ = circ.h(qreg[0])
>>> _ = circ.cnot(qreg[0], qreg[1])
>>> print("Original circuit:", circ, sep="\n")
Original circuit:
         ┌───┐
q31_0: ┤ H ├──■──
       └───┘┌─┴─┐
q31_1: ─────┤ X ├
            └───┘
```

This code (when the print statement is uncommented) should display something like:

We can now fold this circuit as follows.

```
>>> folded = fold_gates_from_left(circ, scale_factor=2.)
>>> print("Folded circuit:", folded, sep="\n")
Folded circuit:
      ┌──────────┐
q_0: ─┤ H ├┤ H ├┤ H ├───
      └───┘└───┘└─────
q_1: ──────────────── X ├
                      └───
```

By default, the folded circuit has the same type as the input circuit. To return an internal `mitiq` representation of the folded circuit (a Cirq circuit), one can use the keyword argument `return_mitiq=True`.

Folding gates by fidelity

In local folding methods, gates can be folded according to custom fidelities by passing the keyword argument `fidelities` into a local folding method. This argument should be a dictionary where each key is a string which specifies the gate and the value of the key is the fidelity of that gate. An example is shown below where we set the fidelity of all single qubit gates to be 1.0, meaning that these gates introduce no errors in the computation.

```
from cirq import Circuit, LineQubit, ops
from mitiq.zne.scaling import fold_gates_at_random

qreg = LineQubit.range(3)
circ = Circuit(
    ops.H.on_each(*qreg),
    ops.CNOT.on(qreg[0], qreg[1]),
    ops.T.on(qreg[2]),
    ops.TOFFOLI.on(*qreg)
)
print(circ)
# 0: ───H───@───@───
#           │   │
# 1: ───H───X───@───
#               │
# 2: ───H───T───X───


folded = fold_gates_at_random(
    circ, scale_factor=3., fidelities={"single": 1.0,
                                       "CNOT": 0.99,
                                       "TOFFOLI": 0.95}
)
print(folded)
# 0: ───H───@───@───@───@───@───@───
#           │   │   │   │   │   │
# 1: ───H───X───X───X───@───@───@───
#                       │   │   │
# 2: ───H───T───────────X───X───X───
```

We can see that only the two-qubit gates and three-qubit gates have been folded in the folded circuit.

Specific gate keys override the global "single", "double", or "triple" options. For example, the dictionary `fidelities = {"single": 1.0, "H": 0.99}` sets all single qubit gates to fidelity one except the Hadamard gate.

A full list of string keys for gates can be found with `help(fold_method)` where `fold_method` is a valid local folding method. Fidelity values must be between zero and one.

Global folding

As mentioned, global folding methods fold the entire circuit instead of individual gates. An example using the same Cirq circuit above is shown below.

```
>>> import cirq
>>> from mitiq.zne.scaling import fold_global

# Get a circuit to fold
>>> 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───

# Fold the circuit
>>> folded = fold_global(circ, scale_factor=3.)
>>> print("Folded circuit:", folded, sep="\n")
Folded circuit:
0: ───H───@───@───H───H───@───
          │   │           │
1: ───────X───X───────────X───
```

Notice that this circuit is still logically equivalent to the input circuit, but the global folding strategy folds the entire circuit until the input scale factor is reached. As with local folding methods, global folding can be called with any `scale_factor >= 3`.

Custom folding methods

Custom folding methods can be defined and used with `mitiq` (e.g., with `mitiq.execute_with_zne`. The signature of this function must be as follows.

```
import cirq
from mitiq.zne.scaling import converter

@converter
def my_custom_folding_function(circuit: cirq.Circuit, scale_factor: float) -> cirq.
↪Circuit:
    # Insert custom folding method here
    return folded_circuit
```

Note: The `converter` decorator makes it so `my_custom_folding_function` can be used with any supported circuit type, not just Cirq circuits. The body of the `my_custom_folding_function` should assume the input circuit is a Cirq circuit, however.

This function can then be used with `mitiq.execute_with_zne` as an option to scale the noise:

```
# Variables circ and scale are a circuit to fold and a scale factor, respectively
zne = mitiq.execute_with_zne(circuit, executor, scale_noise=my_custom_folding_
↪function)
```

2.4.2 Classical fitting and extrapolation: Factory Objects

A *Factory* object is a self-contained representation of an error mitigation method.

This representation is not just hardware-agnostic, it is even *quantum-agnostic*, in the sense that it mainly deals with classical data: the classical input and the classical output of a noisy computation. Nonetheless, a factory can easily interact with a quantum system via its `self.run` method which is the only interface between the "classical world" of a factory and the "quantum world" of a circuit.

The typical tasks of a factory are:

1. Record the result of the computation executed at the chosen noise level;
2. Determine the noise scale factor at which the next computation should be run;
3. Given the history of noise scale factors and results, evaluate the associated zero-noise extrapolation.

The structure of the *Factory* class is adaptive by construction, since the choice of the next noise level can depend on the history of these values. Obviously, non-adaptive methods are supported too and they actually represent the most common choice. Non-adaptive factories are instances of *BatchedFactory* objects. Adaptive factories are instances of *AdaptiveFactory* objects.

Specific classes derived from the abstract class *Factory* represent different zero-noise extrapolation methods. All the built-in factories can be found in the module *mitiq.zne.inference* and are summarized in the following table.

mitiq.zne.inference.LinearFactory	Factory object implementing zero-noise extrapolation based on a linear fit.
mitiq.zne.inference.RichardsonFactory	Factory object implementing Richardson extrapolation.
mitiq.zne.inference.PolyFactory	Factory object implementing a zero-noise extrapolation algorithm based on a polynomial fit.
mitiq.zne.inference.ExpFactory	Factory object implementing a zero-noise extrapolation algorithm assuming an exponential ansatz y(x) = a + b * exp(-c * x), with c > 0.
mitiq.zne.inference.PolyExpFactory	Factory object implementing a zero-noise extrapolation algorithm assuming an (almost) exponential ansatz with a non linear exponent, i.e.:
mitiq.zne.inference.AdaExpFactory	Factory object implementing an adaptive zero-noise extrapolation algorithm assuming an exponential ansatz y(x) = a + b * exp(-c * x), with c > 0.

Once instantiated, a factory can be passed as an argument to the high-level functions contained in the module `mitiq.zne`. Alternatively, a factory can be directly used to implement a zero-noise extrapolation procedure in a fully self-contained way.

To clarify this aspect, we now perform the same zero-noise extrapolation with both methods.

Using a factory object with the `mitiq.zne` module

Let us consider an `executor` function which is similar to the one used in the *getting started* section.

```
import numpy as np
from cirq import Circuit, depolarize, DensityMatrixSimulator

# initialize a backend
SIMULATOR = DensityMatrixSimulator()
# 5% depolarizing noise
NOISE = 0.05

def executor(circ: Circuit) -> float:
    """Executes a circuit with depolarizing noise and
    returns the expectation value of the projector |0><0|."""
    circuit = circ.with_noise(depolarize(p=NOISE))
    rho = SIMULATOR.simulate(circuit).final_density_matrix
    obs = np.diag([1, 0])
    expectation = np.real(np.trace(rho @ obs))
    return expectation
```

Note: In this example we used *Cirq* but other quantum software platforms can be used, as shown in the *getting started* section.

We also define a simple quantum circuit whose ideal expectation value is by construction equal to `1.0`.

```
from cirq import LineQubit, X, H

qubit = LineQubit(0)
circuit = Circuit(X(qubit), H(qubit), H(qubit), X(qubit))
expval = executor(circuit)
exact = 1.0
print(f"The ideal result should be {exact}")
print(f"The real result is {expval:.4f}")
print(f"The abslute error is {abs(exact - expval):.4f}")
```

```
The ideal result should be 1.0
The real result is 0.8794
The abslute error is 0.1206
```

Now we are going to initialize three factory objects, each one encapsulating a different zero-noise extrapolation method.

```
from mitiq.zne.inference import LinearFactory, RichardsonFactory, PolyFactory

# method: scale noise by 1 and 2, then extrapolate linearly to the zero noise limit.
linear_fac = LinearFactory(scale_factors=[1.0, 2.0])

# method: scale noise by 1, 2 and 3, then evaluate the Richardson extrapolation.
richardson_fac = RichardsonFactory(scale_factors=[1.0, 2.0, 3.0])

# method: scale noise by 1, 2, 3, and 4, then extrapolate quadratically to the zero␣
↪noise limit.
poly_fac = PolyFactory(scale_factors=[1.0, 2.0, 3.0, 4.0], order=2)
```

The previous factory objects can be passed as arguments to the high-level functions in `mitiq.zne`. For example:

```python
from mitiq.zne.zne import execute_with_zne

zne_expval = execute_with_zne(circuit, executor, factory=linear_fac)
print(f"Error with linear_fac: {abs(exact - zne_expval):.4f}")

zne_expval = execute_with_zne(circuit, executor, factory=richardson_fac)
print(f"Error with richardson_fac: {abs(exact - zne_expval):.4f}")

zne_expval = execute_with_zne(circuit, executor, factory=poly_fac)
print(f"Error with poly_fac: {abs(exact - zne_expval):.4f}")
```

```
Error with linear_fac: 0.0291
Error with richardson_fac: 0.0070
Error with poly_fac: 0.0110
```

We can also specify the number of shots to use for each noise-scaled circuit.

```python
from mitiq.zne.inference import LinearFactory

# Specify the number of shots for each scale factor.
factory_with_shots = LinearFactory(scale_factors=[1.0, 2.0], shot_list=[100, 200])
```

In this case the factory will pass the number of shots from the *shot_list* to the *executor*. Accordingly, the *executor* should support a *shots* keyword argument, otherwise the shot values will go unused.

Using batched executors with :class:`.BatchedFactory`s

As mentioned, `BatchedFactory` objects are such that all circuits to execute can be precomputed. This is in contrast to `AdapativeFactory` objects in which the next circuit to execute depends on the result of the previous circuit execution.

If the quantum processor is costly to access (e.g., in a queue-based system), executing circuits sequentially can result in high runtimes for zero-noise extrapolation. To deal with this, all classical inference techniques which inherit from a `BatchedFactory` can use a "batched executor." In contrast to the previous `executor` which inputs a single circuit and outputs a single expectation value, a batched executor inputs a list of circuits and outputs a list of expectation values (one for each circuit).

To indicate that an executor is batched, one must provide a return annotation which is either a `numpy.ndarray`, `List[float]`, `Tuple[float]`, `Sequence[float]`, or `Iterable[float]`. For example:

```python
from typing import List
from mitiq import QPROGRAM

def batched_executor(circuits: List[QPROGRAM]) -> List[float]:
    pass
```

A :class::.*BatchedFactory* will detect from the return annotation if an executor is batched or not. If no annotation is provided, the executor is assumed to be sequential (i.e., not batched).

Directly using a factory for error mitigation

Zero-noise extrapolation can also be implemented by directly using the methods `self.run` and `self.reduce` of a *Factory* object.

The method `self.run` evaluates different expectation values at different noise levels until a sufficient amount of data is collected.

The method `self.reduce` instead returns the final zero-noise extrapolation which, in practice, corresponds to a statistical inference based on the measured data.

```
# we import one of the built-in noise scaling function
from mitiq.zne.scaling import fold_gates_at_random

linear_fac.run(circuit, executor, scale_noise=fold_gates_at_random)
zne_expval = linear_fac.reduce()
print(f"Error with linear_fac: {abs(exact - zne_expval):.4f}")

richardson_fac.run(circuit, executor, scale_noise=fold_gates_at_random)
zne_expval = richardson_fac.reduce()
print(f"Error with richardson_fac: {abs(exact - zne_expval):.4f}")

poly_fac.run(circuit, executor, scale_noise=fold_gates_at_random)
zne_expval = poly_fac.reduce()
print(f"Error with poly_fac: {abs(exact - zne_expval):.4f}")
```

```
Error with linear_fac: 0.0291
Error with richardson_fac: 0.0070
Error with poly_fac: 0.0110
```

Behind the scenes, a factory object collects different expectation values at different scale factors. After running a factory, this information can be accessed with appropriate *get* methods. For example:

```
scale_factors = poly_fac.get_scale_factors()
print("Scale factors:", scale_factors)
exp_values = poly_fac.get_expectation_values()
print("Expectation values:", np.round(exp_values, 2))
```

```
Scale factors: [1. 2. 3. 4.]
Expectation values: [0.88 0.79 0.72 0.67]
```

If the user has manually evaluated a list of expectation values associated to a list of scale factors, the simplest way to estimate the corresponding zero-noise limit is to directly call the static *extrapolate* method of the desired factory class (in this case initializing a factory object is unnecessary). For example:

```
zero_limit = PolyFactory.extrapolate(scale_factors, exp_values, order=2)
print(f"Error with PolyFactory.extrapolate method: {abs(exact - zero_limit):.4f}")
```

```
Error with PolyFactory.extrapolate method: 0.0110
```

Beyond the zero-noise limit, additional information about the fit (e.g., optimal parameters, errors, extrapolation curve, etc.) can be returned from *extrapolate* by specifying *full_output* = *True*.

There are also a number of methods to get additional information calculated by the factory class:

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

(continues on next page)

(continued from previous page)

```
fac = LinearFactory(scale_factors=[1.0, 2.0, 3.0])
_ = execute_with_zne(circuit, executor, factory=fac)
print(f"Zero-noise limit: {fac.get_zero_noise_limit():.4f}")
print(f"Fit error on zero-noise limit: {fac.get_zero_noise_limit_error():.4f}")
print(f"Covariance of fitted model parameters: {np.round(fac.get_parameters_
↪covariance(), 5)}")
print(f"Fitted model parameters: {np.round(fac.get_optimal_parameters(), 4)}")
print(f"Extrapolation curve evaluated at zero: {fac.get_extrapolation_curve()(0):.4f}
↪")
```

```
Zero-noise limit: 0.9562
Fit error on zero-noise limit: 0.0138
Covariance of fitted model parameters: [[ 4.0e-05 -8.0e-05]
 [-8.0e-05  1.9e-04]]
Fitted model parameters: [-0.0805  0.9562]
Extrapolation curve evaluated at zero: 0.9562
```

Advanced usage of a factory

Note: This section can be safely skipped by all the readers who are interested in a standard usage of `mitiq`. On the other hand, more experienced users and `mitiq` contributors may find this content useful to understand how a factory object actually works at a deeper level.

In this advanced section we present a *low-level usage* and a *very-low-level usage* of a factory. Again, for simplicity, we solve the same zero-noise extrapolation problem that we have just considered in the previous sections.

Eventually we will also discuss how the user can easily define a custom factory class.

Low-level usage: the `run_classical` method.

The `self.run` method takes as arguments a circuit and other "quantum" objects. On the other hand, the core computation performed by any factory corresponds to a some classical computation applied to the measurement results.

At a lower level, it is possible to clearly separate the quantum and the classical steps of a zero-noise extrapolation procedure. This can be done by defining a function which maps a noise scale factor to the corresponding expectation value.

```
def noise_to_expval(scale_factor: float) -> float:
   """Function returning an expectation value for a given scale_factor."""
   # apply noise scaling
   scaled_circuit = fold_gates_at_random(circuit, scale_factor)
   # return the corresponding expectation value
   return executor(scaled_circuit)
```

Note: The body of the previous function contains the execution of a quantum circuit. However, if we see it as a "black-box", it is just a classical function mapping real numbers to real numbers.

The function `noise_to_expval` encapsulate the "quantum part" of the problem. The "classical part" of the problem can be solved by passing `noise_to_expval` to the `self.run_classical` method of a factory. This method will repeatedly call `noise_to_expval` for different noise levels, so one can view `self.run_classical` as the classical counterpart of the quantum method `self.run`.

```
linear_fac.run_classical(noise_to_expval)
zne_expval = linear_fac.reduce()
print(f"Error with linear_fac: {abs(exact - zne_expval):.4f}")

richardson_fac.run_classical(noise_to_expval)
zne_expval = richardson_fac.reduce()
print(f"Error with richardson_fac: {abs(exact - zne_expval):.4f}")

poly_fac.run_classical(noise_to_expval)
zne_expval = poly_fac.reduce()
print(f"Error with poly_fac: {abs(exact - zne_expval):.4f}")
```

```
Error with linear_fac: 0.0291
Error with richardson_fac: 0.0070
Error with poly_fac: 0.0110
```

Note: With respect to `self.run`, the `self.run_classical` method is much more flexible and can be applied whenever the user is able to autonomously scale the noise level associated to an expectation value. Indeed, the function `noise_to_expval` can represent any experiment or any simulation in which noise can be artificially increased. The scenario is therefore not restricted to quantum circuits but can be easily extended to annealing devices or to gates which are controllable at a pulse level. In principle, one could even use the `self.run_classical` method to mitigate experiments which are unrelated to quantum computing.

Defining a custom factory

If necessary, the user can modify an existing extrapolation methods by subclassing one of the *built-in factories*.

Alternatively, a new adaptive extrapolation method can be derived from the abstract class *`Factory`*. In this case its core methods must be implemented: `self.next`, `self.push`, `self.is_converged`, `self.reduce`, etc. Typically, the `self.__init__` method must be overridden.

A new non-adaptive method can instead be derived from the abstract *`BatchedFactory`* class. In this case it is usually sufficient to override only the `self.__init__` and the `self.reduce` methods, which are responsible for the initialization and for the final zero-noise extrapolation, respectively.

Example: a simple custom factory

Assume that, from physical considerations, we know that the ideal expectation value (measured by some quantum circuit) must always be within two limits: `min_expval` and `max_expval`. For example, this is a typical situation whenever the measured observable has a bounded spectrum.

We can define a linear non-adaptive factory which takes into account this information and clips the result if it falls outside its physical domain.

```
from typing import Iterable
from mitiq.zne.inference import BatchedFactory, mitiq_polyfit
import numpy as np

class MyFactory(BatchedFactory):
    """Factory object implementing a linear extrapolation taking
```

(continues on next page)

(continued from previous page)

```
    into account that the expectation value must be within a given
    interval. If the zero-noise limit falls outside the
    interval, its value is clipped.
    """

    def __init__(
            self,
            scale_factors: Iterable[float],
            min_expval: float,
            max_expval: float,
        ) -> None:
        """
        Args:
            scale_factors: The noise scale factors at which
                           expectation values should be measured.
            min_expval: The lower bound for the expectation value.
            min_expval: The upper bound for the expectation value.
        """
        super(MyFactory, self).__init__(scale_factors)
        self.min_expval = min_expval
        self.max_expval = max_expval

    def reduce(self) -> float:
        """
        Fit a linear model and clip its zero-noise limit.

        Returns:
            The clipped extrapolation to the zero-noise limit.
        """
        # Fit a line and get the optimal parameters (slope, intercept)
        opt_params, _ = mitiq_polyfit(
            self.get_scale_factors(), self.get_expectation_values(), deg=1
      )

        # Return the clipped zero-noise extrapolation.
        return np.clip(opt_params[-1], self.min_expval, self.max_expval)
```

This custom factory can be used in exactly the same way as we have shown in the previous section. By simply replacing `LinearFactory` with `MyFactory` in all the previous code snippets, the new extrapolation method will be applied.

Regression tools in `mitiq.zne.inference`

In the body of the previous `MyFactory` example, we imported and used the *`mitiq_polyfit()`* function. This is simply a wrap of `numpy.polyfit()`, slightly adapted to the notion and to the error types of `mitiq`. This function can be used to fit a polynomial ansatz to the measured expectation values. This function performs a least squares minimization which is **linear** (with respect to the coefficients) and therefore admits an algebraic solution.

Similarly, from *`mitiq.zne.inference`* one can also import *`mitiq_curve_fit()`*, which is instead a wrap of `scipy.optimize.curve_fit()`. Differently from *`mitiq_polyfit()`*, *`mitiq_curve_fit()`* can be used with a generic (user-defined) ansatz. Since the fit is based on a numerical **non-linear** least squares minimization, this method may fail to converge or could be subject to numerical instabilities.

2.5 About Error Mitigation

This is intended as a primer on quantum error mitigation, providing a collection of up-to-date resources from the academic literature, as well as other external links framing this topic in the open-source software ecosystem. This recent review article [1] summarizes the theory behind many error-mitigating techniques.

- *What quantum error mitigation is*
- *Why quantum error mitigation is important*
- *Related fields*
- *External References*

2.5.1 What quantum error mitigation is

Quantum error mitigation refers to a series of modern techniques aimed at reducing (*mitigating*) the errors that occur in quantum computing algorithms. Unlike software bugs affecting code in usual computers, the errors which we attempt to reduce with mitigation are due to the hardware.

Quantum error mitigation techniques try to *reduce* the impact of noise in quantum computations. They generally do not completely remove it. Alternative nomenclature refers to error mitigation as (approximate) error suppression or approximate quantum error correction, but it is worth noting that it is different from error correction. Among the ideas that have been developed so far for quantum error mitigation, a leading candidate is zero-noise extrapolation.

Zero-noise extrapolation

The crucial idea behind zero-noise extrapolation is that, while some minimum strength of noise is unavoidable in the system, quantified by a quantity λ, it is still possible to *increase* it to a value $\lambda' = c\lambda$, with $c > 1$, so that it is then possible to extrapolate the zero-noise limit. This is done in practice by running a quantum circuit (simulation) and calculating a given expectation variable, $\langle X\rangle_\lambda$, then re-running the calculation (which is indeed a time evolution) for $\langle X\rangle_{\lambda'}$, and then extracting $\langle X\rangle_0$. The extraction for $\langle X\rangle_0$ can occur with several statistical fitting models, which can be linear or non-linear. These methods are contained in the `mitiq.zne.inference` and `mitiq.zne` modules.

In theory, one way zero-noise extrapolation can be simulated, also with `mitiq`, is by picking an underlying noise model, e.g., a memoryless bath such that the system dissipates with Lindblad dynamics. Likewise, zero-noise extrapolation can be applied also to non-Markovian noise models [2]. However, it is important to point out that zero-noise extrapolation is a very general method in which one is free to scale and extrapolate almost whatever parameter one wishes to, even if the underlying noise model is unknown.

In experiments, zero-noise extrapolation has been performed with pulse stretching [3]. In this way, a difference between the effective time that a gate is affected by decoherence during its execution on the hardware was introduced by controlling only the gate-defining pulses. The effective noise of a quantum circuit can be scaled also at a gate-level, i.e., without requiring a direct control of the physical hardware. For example this can be achieved with the unitary folding technique, a method which is present in the `mitiq` toolchain.

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 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. 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.

In Mitiq, this technique is implemented in the module `mitiq.zne`.

Probabilistic error cancellation

Probabilistic error cancellation uses a quasi-probability representation [2] to express an ideal (unitary) quantum channel as a linear combination of noisy operations. Given a set of noisy but implementable operations $\Omega = \{O_1, \dots, O_m\}$, an ideal unitary gate can be expressed as $\mathcal{G} = \sum_\alpha \eta_\alpha \mathcal{O}_\alpha = \gamma \sum_\alpha P(\alpha)\sigma(\alpha)\mathcal{O}_\alpha$, where η_α are real coefficients, $\gamma = \sum_\alpha |\eta_\alpha|$, $P(\alpha) = |\eta_\alpha|/\gamma$ is a probability distribution, and $\sigma(\alpha) = \mathrm{sign}(\eta_\alpha)$.

In this setting, we would like to estimate the ideal expectation value of some observable of interest $\langle X \rangle_{\text{ideal}}$, after the action of an ideal circuit given by a sequence of ideal quantum gates $\{\mathcal{G}_i\}_{i=1}^L$. This can be achieved by sampling for each ideal gate $\mathcal{G}_i$ a noisy operation $\mathcal{O}_\alpha$ with probability $P_i(\alpha)$. This random sampling will produce a noisy circuit (given by the sequence of sampled operations $\{\mathcal{O}_{\alpha_i}\}_{i=1}^L$) whose execution produces the final mixed state ρ_f. Then, by measuring the observable X, setting $\gamma_{\text{tot}} := \prod_i^L \gamma_i$ and $\sigma_{\text{tot}} = \prod_{i=1}^L \sigma_i(\alpha)$, one can obtain an unbiased estimate of the ideal expectation value as $\langle X \rangle_{\text{ideal}} = \mathbb{E}\left[\gamma_{\text{tot}}\sigma_{\text{tot}} X_{\text{noisy}}\right]$, where X_{noisy} is the experimental estimate of $\mathrm{tr}[\rho_f X]$ and $\mathbb{E}$ is the sample average over many repetitions of the previous procedure.

In Mitiq, this technique is implemented in the module `mitiq.pec`.

Limitations of probabilistic error cancellation

The number of samples required to estimate the ideal expectation value with error δ and probability $1-\epsilon$ scales as $\left(2\gamma_{\rm tot}^2/\delta^2\right)\log(2/\epsilon)$ [4]. Thus, the sampling overhead is determined by $\gamma_{\rm tot}$ which grows exponentially in the number of gates. It is then crucial to find a linear decomposition that minimizes $\gamma_{\rm tot}$. In addition, a full characterization of the noisy operations up to a good precision is required, which can be costly depending on the implementation.

Other error mitigation techniques

Other examples of error mitigation techniques include injecting noisy gates for randomized compiling or the use of subspace reductions and symmetries. A collection of references on this cutting-edge implementations can be found in the *Research articles* subsection.

2.5.2 Why quantum error mitigation is important

The noisy intermediate scale quantum computing (NISQ) era is characterized by short or medium-depth circuits in which noise affects state preparation, gate operations, and measurement [5]. Current short-depth quantum circuits are noisy, and at the same time it is not possible to implement quantum error correcting codes on them due to the needed qubit number and circuit depth required by these codes.

Error mitigation offers the prospects of writing more compact quantum circuits that can estimate observables with more precision, i.e. increase the performance of quantum computers. By implementing quantum optics tools (such as the modeling noise and open quantum systems) [6][7][8][9], standard as well as cutting-edge statistics and inference techniques, and tweaking them for the needs of the quantum computing community, `mitiq` aims at providing the most comprehensive toolchain for error mitigation.

2.5.3 Related fields

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.

It is fair to say that even the terminology of "quantum error mitigation" or "error mitigation" has only recently coalesced (from ~2015 onward), while even in the previous decade similar concepts or techniques were scattered across these and other fields. Suggestions for additional references are welcome.

Quantum error correction

Quantum error correction is different from quantum error mitigation, as it introduces a series of techniques that generally aim at completely *removing* the impact of errors on quantum computations. In particular, if errors occurs below a certain threshold, the robustness of the quantum computation can be preserved, and fault tolerance is reached.

The main issue of quantum error correction techniques are that generally they require a large overhead in terms of additional qubits on top of those required for the quantum computation. Current quantum computing devices have been able to demonstrate quantum error correction only with a very small number of qubits. What is now referred quantum error mitigation is generally a series of techniques that stemmed as more practical quantum error correction solutions [10].

Quantum optimal control

Optimal control theory is a very versatile set of techniques that can be applied for many scopes. It entails many fields, and it is generally based on a feedback loop between an agent and a target system. Optimal control is applied to several quantum technologies, including in the pulse shaping of gate design in quantum circuits calibration against noisy devices [11].

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 in dynamical decoupling [12]. This technique employs fast control pulses to effectively decouple a system from its environment, with techniques pioneered in the nuclear magnetic resonance community.

Open quantum systems

More in general, quantum computing devices can be studied in the framework of open quantum systems [6][7][8][9], that is, systems that exchange energy and information with the surrounding environment. On the 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, a series of issues arise when someone wants to perform a calculation on a quantum computer. This is 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, during calculations, the protocols are not faithful.

Errors occur for a series of 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, 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 [13].

2.5.4 External References

Here is a list of useful external resources on quantum error mitigation, including software tools that provide the possibility of studying quantum circuits.

Research articles

A list of research articles academic resources on error mitigation:

- **On zero-noise extrapolation:**
  - Theory, Y. Li and S. Benjamin, *Phys. Rev. X*, 2017 [14] and K. Temme *et al.*, *Phys. Rev. Lett.*, 2017 [2]
  - Experiment on superconducting circuit chip, A. Kandala *et al.*, *Nature*, 2019 [3]
- **On probabilistic error cancellation:**
  - Theory, Y. Li and S. Benjamin, *Phys. Rev. X*, 2017 [14] and K. Temme *et al.*, *Phys. Rev. Lett.*, 2017 [2]
  - Resource analysis for probabilistic error cancellation, Ryuji Takagi, arxiv, 2020 [4]
- **On randomization methods:**

  - Randomized compiling with twirling gates, J. Wallman *et al.*, *Phys. Rev. A*, 2016 [15]
  - Porbabilistic error correction, K. Temme *et al.*, *Phys. Rev. Lett.*, 2017 [2]
  - Practical proposal, S. Endo *et al.*, *Phys. Rev. X*, 2018 [16]
  - Experiment on trapped ions, S. Zhang *et al.*, *Nature Comm.* 2020 [17]
  - Experiment with gate set tomography on a supeconducting circuit device, J. Sun *et al.*, 2019 arXiv [18]
- **On subspace expansion:**
  - By hybrid quantum-classical hierarchy introduction, J. McClean *et al.*, *Phys. Rev. A*, 2017 [19]
  - By symmetry verification, X. Bonet-Monroig *et al.*, *Phys. Rev. A*, 2018 [20]
  - With a stabilizer-like method, S. McArdle *et al.*, *Phys. Rev. Lett.*, 2019, [21]
  - Exploiting molecular symmetries, J. McClean *et al.*, *Nat. Comm.*, 2020 [22]
  - Experiment on a superconducting circuit device, R. Sagastizabal *et al.*, *Phys. Rev. A*, 2019 [23]
- **On other error-mitigation techniques such as:**
  - Approximate error-correcting codes in the generalized amplitude-damping channels, C. Cafaro *et al.*, *Phys. Rev. A*, 2014 [24]:
  - Extending the variational quantum eigensolver (VQE) to excited states, R. M. Parrish *et al.*, *Phys. Rev. Lett.*, 2017 [25]
  - Quantum imaginary time evolution, M. Motta *et al.*, *Nat. Phys.*, 2020 [26]
  - Error mitigation for analog quantum simulation, J. Sun *et al.*, 2020, arXiv [18]
- For an extensive introduction: S. Endo, *Hybrid quantum-classical algorithms and error mitigation*, PhD Thesis, 2019, Oxford University (Link), or [1].

Software

Here is a (non-comprehensive) list of open-source software libraries related to quantum computing, noisy quantum dynamics and error mitigation:

- **IBM Q**'s Qiskit provides a stack for quantum computing simulation and execution on real devices from the cloud. In particular, `qiskit.Aer` contains the `NoiseModel` object, integrated with `mitiq` tools. Qiskit's OpenPulse provides pulse-level control of qubit operations in some of the superconducting circuit devices. `mitiq` is integrated with `qiskit`, in the *`qiskit_utils`* and *`conversions`* modules.
- **Goole AI Quantum**'s Cirq offers quantum simulation of quantum circuits. The `cirq.Circuit` object is integrated in `mitiq` algorithms as the default circuit.
- **Rigetti Computing**'s PyQuil is a library for quantum programming. Rigetti's stack offers the execution of quantum circuits on superconducting circuits devices from the cloud, as well as their simulation on a quantum virtual machine (QVM), integrated with `mitiq` tools in the *`pyquil_utils`* module.
- QuTiP, the quantum toolbox in Python, contains a quantum information processing module that allows to simulate quantum circuits, their implementation on devices, as well as the simulation of pulse-level control and time-dependent density matrix evolution with the `qutip.Qobj` object and the `Processor` object in the `qutip.qip` module.
- Krotov is a package implementing Krotov method for optimal control interfacing with QuTiP for noisy density-matrix quantum evolution.

- PyGSTi allows to characterize quantum circuits by implementing techniques such as gate set tomography (GST) and randomized benchmarking.

This is just a selection of open-source projects related to quantum error mitigation. A more comprehensinve collection of software on quantum computing can be found here and on Unitary Fund's list of supported projects.

2.6 Error mitigation on IBMQ backends

This tutorial shows an example of how to mitigate noise on IBMQ backends, broken down in the following steps.

- *Setup: Defining a circuit*
- *High-level usage*
- *Cirq frontend*
- *Lower-level usage*

2.6.1 Setup: Defining a circuit

First we import Qiskit and mitiq.

```
import qiskit
import mitiq
from mitiq.mitiq_qiskit.qiskit_utils import random_identity_circuit
```

For simplicity, we'll use a random single-qubit circuit with ten gates that compiles to the identity, defined below.

```
>>> circuit = random_identity_circuit(depth=10)
>>> print(circuit)
         ┌───┐┌───┐┌───┐┌───┐┌───┐┌───┐┌───┐┌───┐┌───┐┌───┐┌───┐
q_0: |0>┤ Y ├┤ Y ├┤ X ├┤ Z ├┤ Z ├┤ Z ├┤ Z ├┤ X ├┤ X ├┤ Z ├┤ Y ├
         └───┘└───┘└───┘└───┘└───┘└───┘└───┘└───┘└───┘└───┘└───┘
 c_0: 0
```

Currently this circuit has no measurements, but we will add a measurement below and use the probability of the ground state as our observable to mitigate.

2.6.2 High-level usage

To use `mitiq` with just a few lines of code, we simply need to define a function which inputs a circuit and outputs the expectation value to mitigate. This function will:

1. [Optionally] Add measurement(s) to the circuit.
2. Run the circuit.
3. Convert from raw measurement statistics (or a different output format) to an expectation value.

We define this function in the following code block. Because we are using IBMQ backends, we first load our account.

Note: The following code requires a valid IBMQ account. See https://quantum-computing.ibm.com/ for instructions.

```
provider = qiskit.IBMQ.load_account()

def armonk_executor(circuit: qiskit.QuantumCircuit, shots: int = 1024) -> float:
    """Returns the expectation value to be mitigated.

    Args:
        circuit: Circuit to run.
        shots: Number of times to execute the circuit to compute the expectation␣
↪value.
    """
    # (1) Add measurements to the circuit
    circuit.measure(circuit.qregs[0], circuit.cregs[0])

    # (2) Run the circuit
    job = qiskit.execute(
        experiments=circuit,
        # Change backend=provider.get_backend("ibmq_armonk") to run on hardware
        backend=provider.get_backend("ibmq_qasm_simulator"),
        optimization_level=0,  # Important!
        shots=shots
    )

    # (3) Convert from raw measurement counts to the expectation value
    counts = job.result().get_counts()
    if counts.get("0") is None:
        expectation_value = 0.
    else:
        expectation_value = counts.get("0") / shots
    return expectation_value
```

At this point, the circuit can be executed to return a mitigated expectation value by running `mitiq.execute_with_zne`, as follows.

```
mitigated = mitiq.execute_with_zne(circuit, armonk_executor)
```

As long as a circuit and a function for executing the circuit are defined, the `mitiq.execute_with_zne` function can be called as above to return zero-noise extrapolated expectation value(s).

Options

Different options for noise scaling and extrapolation can be passed into the `mitiq.execute_with_zne` function. By default, noise is scaled by locally folding gates at random, and the default extrapolation is Richardson.

To specify a different extrapolation technique, we can pass a different `Factory` object to `execute_with_zne`. The following code block shows an example of using linear extrapolation with five different (noise) scale factors.

```
linear_factory = mitiq.zne.inference.LinearFactory(scale_factors=[1.0, 1.5, 2.0, 2.5,␣
↪3.0])
mitigated = mitiq.execute_with_zne(circuit, armonk_executor, fac=linear_factory)
```

To specify a different noise scaling method, we can pass a different function for the argument `scale_noise`. This function should input a circuit and scale factor and return a circuit. The following code block shows an example of scaling noise by folding gates starting from the left (instead of at random, the default behavior for `mitiq.execute_with_zne`).

```
mitigated = mitiq.execute_with_zne(circuit, armonk_executor, scale_noise=mitiq.zne.
↪scaling.fold_gates_from_left)
```

Any different combination of noise scaling and extrapolation technique can be passed as arguments to `mitiq.execute_with_zne`.

Cirq frontend

It isn't necessary to use Qiskit frontends (circuits) to run on IBM backends. We can use conversions in `mitiq` to use any supported frontend with any supported backend. Below, we show how to run a Cirq circuit on an IBMQ backend.

First, we define the Cirq circuit.

```
import cirq

qbit = cirq.GridQubit(0, 0)
cirq_circuit = cirq.Circuit(cirq.ops.H.on(qbit)
```

Now, we simply add a line to our executor function which converts from a Cirq circuit to a Qiskit circuit.

```
from mitiq.mitiq_qiskit.conversions import to_qiskit

def cirq_armonk_executor(cirq_circuit: cirq.Circuit, shots: int = 1024) -> float:
    qiskit_circuit = to_qiskit(cirq_circuit)
    return armonk_executor(qiskit_circuit, shots)
```

After this, we can use `mitiq.execute_with_zne` in the same way as above.

```
mitigated = mitiq.execute_with_zne(cirq_circuit, cirq_armonk_executor)
```

As above, different noise scaling or extrapolation methods can be used.

2.6.3 Lower-level usage

Here, we give more detailed usage of the `mitiq` library which mimics what happens in the call to `mitiq.execute_with_zne` in the previous example. In addition to showing more of the `mitiq` library, this example explains the code in the previous section in more detail.

First, we define factors to scale the circuit length by and fold the circuit using the `fold_gates_at_random` local folding method.

```
depth = 10
circuit = random_identity_circuit(depth=depth)

scale_factors = [1., 1.5, 2., 2.5, 3.]
folded_circuits = [
        mitiq.zne.scaling.fold_local(
        circuit, scale, method=mitiq.zne.scaling.fold_gates_at_random
    ) for scale in scale_factors
]
```

We now add the observables we want to measure to the circuit. Here we use a single observable $\Pi_0 \equiv |0\rangle\langle 0|$ -- i.e., the probability of measuring the ground state -- but other observables can be used.

```
for folded_circuit in folded_circuits:
    folded_circuit.measure(folded_circuit.qregs[0], folded_circuit.cregs[0])
```

For a noiseless simulation, the expectation of this observable should be 1.0 because our circuit compiles to the identity. For noisy simulation, the value will be smaller than one. Because folding introduces more gates and thus more noise, the expectation value will decrease as the length (scale factor) of the folded circuits increase. By fitting this to a curve, we can extrapolate to the zero-noise limit and obtain a better estimate.

In the code block below, we setup our connection to IBMQ backends.

Note: The following code requires a valid IBMQ account. See https://quantum-computing.ibm.com/ for instructions.

```
provider = qiskit.IBMQ.load_account()
print("Available backends:", *provider.backends(), sep="\n")
```

Depending on your IBMQ account, this print statement will display different available backend names. Shown below is an example of executing the folded circuits using the IBMQ Armonk single qubit backend. Depending on what backends are available, you may wish to choose a different backend by changing the `backend_name` below.

```
shots = 8192
backend_name = "ibmq_armonk"

job = qiskit.execute(
   experiments=folded_circuits,
   # Change backend=provider.get_backend(backend_name) to run on hardware
   backend=provider.get_backend("ibmq_qasm_simulator"),
   optimization_level=0,  # Important!
   shots=shots
)
```

Note: We set the `optimization_level=0` to prevent any compilation by Qiskit transpilers.

Once the job has finished executing, we can convert the raw measurement statistics to observable values by running the following code block.

```
all_counts = [job.result().get_counts(i) for i in range(len(folded_circuits))]
expectation_values = [counts.get("0") / shots for counts in all_counts]
```

We can now see the unmitigated observable value by printing the first element of `expectation_values`. (This value corresponds to a circuit with scale factor one, i.e., the original circuit.)

```
>>> print("Unmitigated expectation value:", round(expectation_values[0], 3))
Unmitigated expectation value: 0.945
```

Now we can use the `reduce` method of `mitiq.Factory` objects to extrapolate to the zero-noise limit. Below we use a linear fit (order one polynomial fit) and print out the extrapolated zero-noise value.

```
>>> fac = mitiq.zne.inference.LinearFactory(scale_factors)
>>> fac.instack, fac.outstack = scale_factors, expectation_values
>>> zero_noise_value = fac.reduce()
>>> print(f"Extrapolated zero-noise value:", round(zero_noise_value, 3))
Extrapolated zero-noise value: 0.961
```

For this example, we indeed see that the extrapolated zero-noise value (0.961) is closer to the true value (1.0) than the unmitigated expectation value (0.945).

2.7 Mitigating a MaxCut Landscape with QAOA

This tutorial shows an example of mitigating the energy landscape for a two-qubit instance of MaxCut using the quantum alternating operator ansatz (QAOA). We first import the libraries we will use.

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

from cirq import Circuit, CNOT, DensityMatrixSimulator, H, LineQubit, depolarize, rz
from mitiq.zne import mitigate_executor

SIMULATOR = DensityMatrixSimulator()
```

We will use the density matrix simulator to compute the final density matrix of our noisy circuit, from which we then compute expectation values to mitigate.

2.7.1 Defining the noiseless circuit

We define a function below which returns a two-qubit QAOA circuit at a specified driver angle γ. The mixer angle β is set to $\pi/8$ and we will sweep over γ to compute an energy landscape.

```
def maxcut_qaoa_circuit(gamma: float) -> Circuit:
    """Returns two-qubit MaxCut QAOA circuit with beta = pi/8 and with the provided␣
↪gamma.

    Args:
        gamma: One of the two variational parameters (the other is fixed).

    Returns:
        A two-qubit MaxCut QAOA circuit with fixed beta and gamma.
    """
    q0, q1 = LineQubit.range(2)

    return Circuit(
        H.on_each(q0, q1),
        CNOT.on(q0, q1),
        rz(2 * gamma).on(q1),
        CNOT.on(q0, q1),
        H.on_each(q0, q1),
        rz(np.pi / 4).on_each(q0, q1),
        H.on_each(q0, q1)
    )
```

We can visualize the circuit for a particular γ as follows.

```
>>> circ = maxcut_qaoa_circuit(gamma=np.pi)
>>> print(circ)
0: ───H───@───────────────@───H───Rz(0.25)───H───
          │               │
1: ───H───X───Rz(2)───X───H───Rz(0.25)───H───
```

2.7.2 Defining the executor

To interface with `mitiq`, we now define an executor function which adds noise to the circuit and computes the expectation value of the usual QAOA observable $Z \otimes Z$, i.e., Pauli-Z on each qubit. The code block below first creates this observable, then sets a noise value, then defines the executor.

```
# Observable to measure
z = np.diag([1, -1])
zz = np.kron(z, z)

# Strength of noise channel
p = 0.05


def executor(circ: Circuit) -> float:
    """
    Simulates the execution of a circuit with depolarizing noise.

    Args:
        circ: The input circuit.

    Returns:
        The expectation value of the ZZ observable.
    """
    # Add depolarizing noise to the circuit
    circuit = circ.with_noise(depolarize(p))

    # Get the final density matrix of the circuit
    rho = SIMULATOR.simulate(circuit).final_density_matrix

    # Evaluate the ZZ expectation value
    expectation = np.real(np.trace(rho @ zz))
    return expectation
```

Note: The above code block uses depolarizing noise, but any channel in Cirq can be substituted in.

2.7.3 Computing the unmitigated landscape

We now compute the unmitigated energy landscape $\langle Z \otimes Z \rangle(\gamma)$ in the following code block.

```
gammas = np.linspace(-np.pi, np.pi, 50)
expectations = []

for gamma in gammas:
    circ = maxcut_qaoa_circuit(gamma)
    expectation = executor(circ)
    expectations.append(expectation)
```

The following code plots these values for visualization.

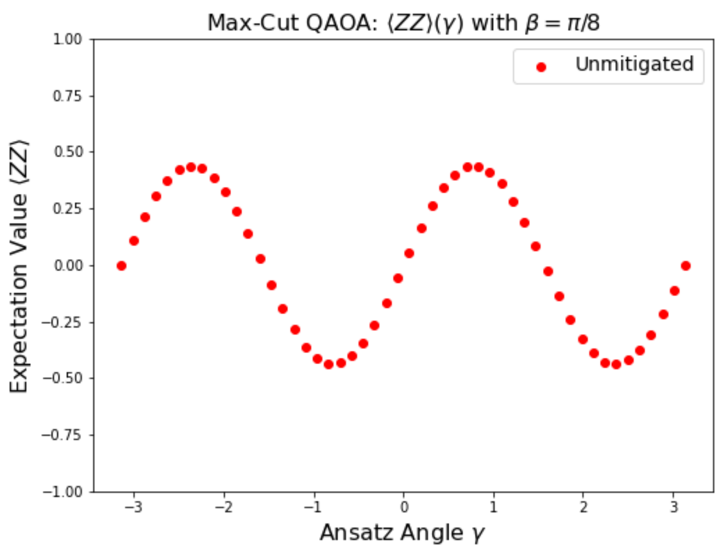
```
plt.figure(figsize=(8, 6))
plt.scatter(gammas, expectations, color="r", label="Unmitigated")
plt.title(rf"Max-Cut QAOA: $\langle ZZ \rangle (\gamma)$ with $\beta = \pi/8$",
↪fontsize=16)
```

(continues on next page)

(continued from previous page)

```
plt.xlabel(r"Ansatz Angle $\gamma$", fontsize=16)
plt.ylabel(r"Expectation Value $\langle ZZ \rangle$", fontsize=16)
plt.legend(fontsize=14)
plt.ylim(-1, 1);
```

The plot is shown below.



2.7.4 Computing the mitigated landscape

We now do the same task but use `mitiq` to mitigate the energy landscape.

We do so by first getting a mitigated executor as follows.

```
mitigated_executor = mitigate_executor(executor)
```

We then run the same code above to compute the energy landscape, but this time use the `mitigated_executor` instead of just the executor.

```
mitigated_expectations = []

for gamma in gammas:
    circ = maxcut_qaoa_circuit(gamma)
    mitigated_expectation = mitigated_executor(circ)
    mitigated_expectations.append(mitigated_expectation)
```

We can visualize the mitigated landscape alongside the unmitigated landscape with the following code for plotting.

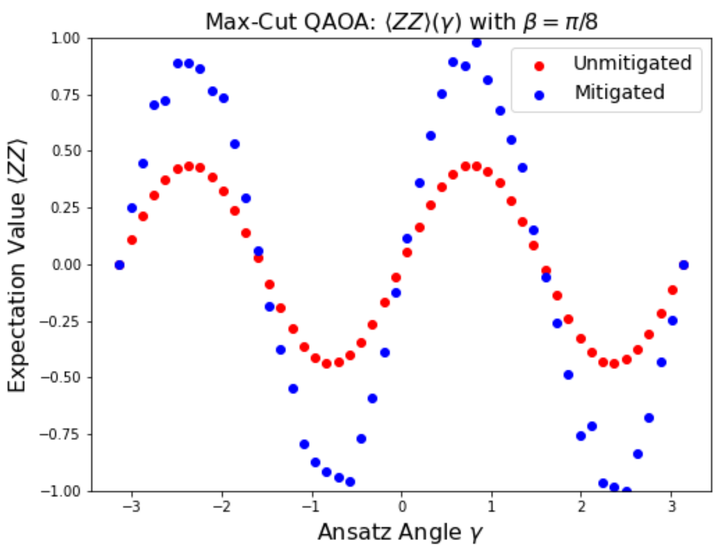
```
plt.figure(figsize=(8, 6))
plt.scatter(gammas, expectations, color="r", label="Unmitigated")
plt.scatter(gammas, mitigated_expectations, color="b", label="Mitigated")
plt.title(rf"Max-Cut QAOA: $\langle ZZ \rangle (\gamma)$ with $\beta = \pi/8$",␣
↪fontsize=16)
plt.xlabel(r"Ansatz Angle $\gamma$", fontsize=16)
plt.ylabel(r"Expectation Value $\langle ZZ \rangle$", fontsize=16)
```

(continues on next page)

(continued from previous page)

```
plt.legend(fontsize=14)
plt.ylim(-1, 1);
```

This cell produces a plot which looks like the following.



As we can see, the mitigated landscape is significantly closer to the noiseless landscape than the unmitigated curve.

Acknowledgements

The code for this documentation was written by Peter Karalekas.

2.8 Defining Hamiltonians as Linear Combinations of Pauli Strings

This tutorial shows an example of using Hamiltonians defined as `PauliSum` objects or similar objects in other supported frontends. The usage of these Hamiltonian-like objects does not change the interface with `mitiq`, but we show an example for users who prefer these constructions.

Specifically, in this tutorial we will mitigate the expectation value of the Hamiltonian

$$H := 1.5X_1Z_2 - 0.7Y_1$$

on two qubits.

2.8.1 Setup

First we import libraries for this example.

```
from functools import partial

import matplotlib.pyplot as plt
import numpy as np

import cirq
from mitiq import execute_with_zne
from mitiq.zne.inference import LinearFactory
```

2.8.2 Defining the Hamiltonian

Now we define the Hamiltonian as a `PauliSum` by defining the Pauli strings $X_1 Z_2$ and Y_1 then taking a linear combination of these to create the Hamiltonian above.

```
# Qubit register
qreg = cirq.LineQubit.range(2)

# Two Pauli operators
string1 = cirq.PauliString(cirq.ops.X.on(qreg[0]), cirq.ops.Z.on(qreg[1]))
string2 = cirq.PauliString(cirq.ops.Y.on(qreg[1]))

# Hamiltonian
ham = 1.5 * string1 - 0.7 * string2
```

By printing the Hamiltonian we see:

```
>>> print(ham)
1.500*X(0)*Z(1)-0.700*Y(1)
```

Note that we could have created the Hamiltonian by directly defining a `PauliSum` with the coefficients.

2.8.3 Using the Hamiltonian in the executor

To interface with `mitiq`, we define an `executor` function which maps an input (noisy) circuit to an expectation value. In the code block below, we show how to define this function and return the expectation of the Hamiltonian above.

```
def executor(
    circuit: cirq.Circuit,
    hamiltonian: cirq.PauliSum,
    noise_value: float
) -> float:
    """Runs the circuit and returns the expectation value of the Hamiltonian.

    Args:
        circuit: Defines the ansatz wavefunction.
        hamiltonian: Hamiltonian to compute the expectation value of w.r.t. the
↪ansatz wavefunction.
        noise_value: Probability of depolarizing noise.
    """
```

(continues on next page)

(continued from previous page)

```
    # Add noise
    noisy_circuit = circuit.with_noise(cirq.depolarize(noise_value))

    # Get the final density matrix
    dmat = cirq.DensityMatrixSimulator().simulate(noisy_circuit).final_density_matrix

    # Return the expectation value
    return hamiltonian.expectation_from_density_matrix(
        dmat,
        qubit_map={ham.qubits[i]: i for i in range(len(ham.qubits))}
    ).real
```

This executor inputs a Hamiltonian as well as a noise value, adds noise, then uses the `cirq.PauliSum.expectation_from_density_matrix()` method to return the expectation value.

The remaining interface is as usual with `mitiq`. For the sake of example, we show an application mitigating the expectation value of H with an example ansatz at different noise levels.

2.8.4 Example usage

Below we create an example ansatz parameterized by one angle γ.

```
def ansatz(gamma: float) -> cirq.Circuit:
    """Returns the ansatz circuit."""
    return cirq.Circuit(
        cirq.ops.ry(gamma).on(qreg[0]),
        cirq.ops.CNOT.on(*qreg),
        cirq.ops.rx(gamma / 2).on_each(qreg)
    )
```

For the angle $\gamma = \pi$, this ansatz has the following structure:

```
>>> print(ansatz(gamma=np.pi))
0: ───Ry(π)───@───Rx(0.5π)───
              │
1: ───────────X───Rx(0.5π)───
```

We now compute expectation values of H using the `executor` as follows.

```
pvals = np.linspace(0, 0.01, 20)
expvals = [executor(ansatz(gamma=np.pi), ham, p) for p in pvals]
```

We can compute mitigated expectation values at these same noise levels by running the following. Here, we use a *`LinearFactory`* and use the `partial` function to update the `executor` for each noise value. The latter point ensures `this_executor` has the correct signature (input circuit, output float) to use with *`execute_with_zne()`*.

```
fac = LinearFactory(scale_factors=list(range(1, 6)))
mitigated_expvals = []

for p in pvals:
    this_executor = partial(executor, hamiltonian=ham, noise_value=p)
    mitigated_expvals.append(
        execute_with_zne(ansatz(gamma=np.pi), this_executor, factory=fac)
    )
```

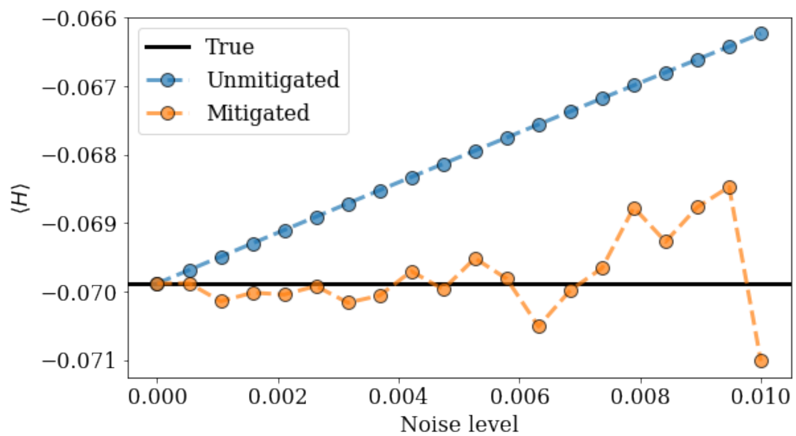
We can now visualize the effect that error mitigation has by running the following code for plotting.

```
plt.rcParams.update({"font.family": "serif", "font.size": 16})
plt.figure(figsize=(9, 5))

plt.axhline(y=expvals[0], lw=3., label="True", color="black")
plt.plot(pvals, expvals, "--o", lw=3, markersize=10, markeredgecolor="black", alpha=0.
↪7, label="Unmitigated")
plt.plot(pvals, mitigated_expvals, "--o", lw=3, markersize=10, markeredgecolor="black
↪", alpha=0.7, label="Mitigated")

plt.xlabel("Noise level")
plt.ylabel(r"$\langle H \rangle$")
plt.legend()
plt.show()
```

This produces a plot of expectation value (unmitigated and mitigated) $\langle H \rangle$ vs. noise strength p. We include the true (noiseless) expectation value on the plot for comparison.



As we can see, the mitigated expectation values are closer, on average, to the true expectation value.

2.8.5 Sampling

Finally, we note that $\langle H \rangle$ can be estimated by sampling using the `cirq.PauliSumCollector`. An example of a `sampling_executor` which uses this is shown below.

```
def sampling_executor(
    circuit: cirq.Circuit,
    hamiltonian: cirq.PauliSum,
    noise_value: float,
    nsamples: int = 10_000
) -> float:
    """Runs the circuit and returns the expectation value of the Hamiltonian.

    Args:
        circuit: Defines the ansatz wavefunction.
        hamiltonian: Hamiltonian to compute the expectation value of w.r.t. the␣
↪ansatz wavefunction.
        noise_value: Probability of depolarizing noise.
        nsamples: Number of samples to take per each term of the Hamiltonian.
    """
    # Add noise
```

(continues on next page)

(continued from previous page)

```
    noisy_circuit = circuit.with_noise(cirq.depolarize(noise_value))

    # Do the sampling
    psum = cirq.PauliSumCollector(circuit, ham, samples_per_term=nsamples)
    psum.collect(sampler=cirq.Simulator())

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

This executor can be used in the same way as the previously defined `executor` which used a density matrix simulation to evaluate $\langle H \rangle$.

CHAPTER 3

API-doc

3.1 Benchmarks

3.1.1 MaxCut

This module contains methods for benchmarking mitiq error extrapolation against a standard QAOA for MAXCUT.

`mitiq.benchmarks.maxcut.`**`make_maxcut`**(*graph*, *noise=0*, *scale_noise=None*, *factory=None*)

Makes an executor that evaluates the QAOA ansatz at a given beta and gamma parameters.

Parameters

- **graph** (`List[Tuple[int, int]]`) -- The MAXCUT graph as a list of edges with integer labelled nodes.
- **noise** (`float`) -- The level of depolarizing noise.
- **scale_noise** (`Optional[Callable[[Union[Circuit, Program, QuantumCircuit], float], Union[Circuit, Program, QuantumCircuit]]]`) -- The noise scaling method for ZNE.
- **factory** (`Optional[Factory]`) -- The factory to use for ZNE.

Return type `Tuple[Callable[[ndarray], float], Callable[[ndarray], Circuit], ndarray]`

Returns

(ansatz_eval, ansatz_maker, cost_obs) as a triple. Here

ansatz_eval: function that evalutes the maxcut ansatz on the noisy cirq backend.

ansatz_maker: function that returns an ansatz circuit. cost_obs: the cost observable as a dense matrix.

`mitiq.benchmarks.maxcut.`**`make_noisy_backend`**(*noise*, *obs*)

Helper function to match mitiq's backend type signature.

Parameters

- **noise** (float) -- The level of depolarizing noise.
- **obs** (ndarray) -- The observable that the backend should measure.

Return type Callable[[Circuit], float]

Returns A mitiq backend function.

mitiq.benchmarks.maxcut.**run_maxcut**(*graph*, *x0*, *noise=0*, *scale_noise=None*, *factory=None*, *verbose=True*)

Solves MAXCUT using QAOA on a cirq wavefunction simulator using a Nelder-Mead optimizer.

Parameters

- **graph** (List[Tuple[int, int]]) -- The MAXCUT graph as a list of edges with integer labelled nodes.
- **x0** (ndarray) -- The initial parameters for QAOA [betas, gammas]. The size of x0 determines the number of p steps.
- **noise** (float) -- The level of depolarizing noise.
- **scale_noise** (Optional[Callable[[Union[Circuit, Program, QuantumCircuit], float], Union[Circuit, Program, QuantumCircuit]]]) -- The noise scaling method for ZNE.
- **factory** (Optional[*Factory*]) -- The factory to use for ZNE.
- **verbose** (bool) -- An option to pass to minimize.

Return type Tuple[float, ndarray, List[float]]

Returns A triple of the minimum cost, the values of beta and gamma that obtained that cost, and a list of costs at each iteration step.

Example

Run MAXCUT with 2 steps such that betas = [1.0, 1.1] and gammas = [1.4, 0.7] on a graph with four edges and four nodes.

```
>>> graph = [(0, 1), (1, 2), (2, 3), (3, 0)]
>>> fun,x,traj = run_maxcut(graph, x0=[1.0, 1.1, 1.4, 0.7])
Optimization terminated successfully.
         Current function value: -4.000000
         Iterations: 108
         Function evaluations: 188
```

3.1.2 Random Circuits

Contains methods used for testing mitiq's performance on random circuits.

mitiq.benchmarks.random_circuits.**rand_circuit_zne**(*n_qubits*, *depth*, *trials*, *noise*, *fac=None*, *scale_noise=<function fold_gates_at_random>*, *op_density=0.99*, *silent=True*, *seed=None*)

Benchmarks a zero-noise extrapolation method and noise scaling executor by running on randomly sampled quantum circuits.

Parameters

- **n_qubits** (int) -- The number of qubits.
- **depth** (int) -- The depth in moments of the random circuits.
- **trials** (int) -- The number of random circuits to average over.
- **noise** (float) -- The noise level of the depolarizing channel for simulation.
- **fac** (Optional[*Factory*]) -- The Factory giving the extrapolation method.
- **scale_noise** (Callable[[Union[Circuit, Program, QuantumCircuit], float], Union[Circuit, Program, QuantumCircuit]]) -- The method for scaling noise, e.g. fold_gates_at_random
- **op_density** (float) -- The expected proportion of qubits that are acted on in any moment.
- **silent** (bool) -- If False will print out statements every tenth trial to track progress.
- **seed** (Optional[int]) -- Optional seed for random number generator.

Return type Tuple[ndarray, ndarray, ndarray]

Returns The triple (exacts, unmitigateds, mitigateds) where each is a list whose values are the expectations of that trial in noiseless, noisy, and error-mitigated runs respectively.

mitiq.benchmarks.random_circuits.**sample_projector**(*n_qubits*, *seed=None*)

Constructs a projector on a random computational basis state of n_qubits.

Parameters

- **n_qubits** (int) -- A number of qubits
- **seed** (Union[None, int, RandomState]) -- Optional seed for random number generator. It can be an integer or a numpy.random.RandomState object.

Return type ndarray

Returns A random computational basis projector on n_qubits. E.g., for two qubits this could be np.diag([0, 0, 0, 1]), corresponding to the projector on the |11> state.

3.1.3 Randomized Benchmarking

Contains methods used for testing mitiq's performance on randomized benchmarking circuits.

mitiq.benchmarks.randomized_benchmarking.**rb_circuits**(*n_qubits*, *num_cliffords*, *trials*, *qubit_labels=None*)

Generates a set of randomized benchmarking circuits, i.e. circuits that are equivalent to the identity.

Parameters

- **n_qubits** (int) -- The number of qubits. Can be either 1 or 2
- **num_cliffords** (List[int]) -- A list of numbers of Clifford group elements in the random circuits. This is proportional to the eventual depth per circuit.
- **trials** (int) -- The number of random circuits at each num_cfd.
- **qubit_labels** (Optional[List[int]]) --

Return type List[Circuit]

Returns A list of randomized benchmarking circuits.

3.1.4 Utils

Utility functions for benchmarking.

`mitiq.benchmarks.utils.`**`noisy_simulation`**(*circ*, *noise*, *obs*)

Simulates a circuit with depolarizing noise at level NOISE.

Parameters

- **circ** (Circuit) -- The quantum program as a cirq object.
- **noise** (float) -- The level of depolarizing noise.
- **obs** (ndarray) -- The observable that the backend should measure.

Return type float

Returns The observable's expectation value.

3.2 Mitiq - PyQuil

3.2.1 PyQuil Utils

pyQuil utility functions.

`mitiq.mitiq_pyquil.pyquil_utils.`**`generate_qcs_executor`**(*qc*, *expectation_fn*, *shots=1000*, *reset=True*, *debug=False*)

Generates an executor for QCS that ingests pyQuil programs.

Parameters

- **qc** (QuantumComputer) -- The QuantumComputer object to use as backend.
- **expectation_fn** (Callable[[ndarray], float]) -- Takes in bitstring results and produces a float.
- **shots** (int) -- Number of shots to take.
- **reset** (bool) -- Whether or not to enable active reset.
- **debug** (bool) -- If true, print the program after compilation.

Return type Callable[[Program], float]

Returns A customized executor function.

`mitiq.mitiq_pyquil.pyquil_utils.`**`ground_state_expectation`**(*results*)

Example expectation_fn. Computes the ground state expectation, also called survival probability.

Parameters **results** (ndarray) -- Array of bitstrings from running a quantum program.

Return type float

Returns A single expectation value computed from the results.

3.3 Mitiq - Qiskit

3.3.1 Conversions

Functions to convert between Mitiq's internal circuit representation and Qiskit's circuit representation.

`mitiq.mitiq_qiskit.conversions.`**`from_qasm`**(*qasm*)

Returns a Mitiq circuit equivalent to the input QASM string.

Parameters **`qasm`** (`str`) -- QASM string to convert to a Mitiq circuit.

Return type `Circuit`

Returns Mitiq circuit representation equivalent to the input QASM string.

`mitiq.mitiq_qiskit.conversions.`**`from_qiskit`**(*circuit*)

Returns a Mitiq circuit equivalent to the input Qiskit circuit.

Parameters **`circuit`** (`QuantumCircuit`) -- Qiskit circuit to convert to a Mitiq circuit.

Return type `Circuit`

Returns Mitiq circuit representation equivalent to the input Qiskit circuit.

`mitiq.mitiq_qiskit.conversions.`**`to_qasm`**(*circuit*)

Returns a QASM string representing the input Mitiq circuit.

Parameters **`circuit`** (`Circuit`) -- Mitiq circuit to convert to a QASM string.

Returns QASM string equivalent to the input Mitiq circuit.

Return type QASMType

`mitiq.mitiq_qiskit.conversions.`**`to_qiskit`**(*circuit*)

Returns a Qiskit circuit equivalent to the input Mitiq circuit.

Parameters **`circuit`** (`Circuit`) -- Mitiq circuit to convert to a Qiskit circuit.

Return type `QuantumCircuit`

Returns Qiskit.QuantumCircuit object equivalent to the input Mitiq circuit.

3.3.2 Qiskit Utils

Qiskit utility functions.

`mitiq.mitiq_qiskit.qiskit_utils.`**`random_one_qubit_identity_circuit`**(*num_cliffords*)

Returns a single-qubit identity circuit.

Parameters **`num_cliffords`** (*`int`*) -- Number of cliffords used to generate the circuit.

Returns Quantum circuit as a `qiskit.QuantumCircuit` object.

Return type circuit

`mitiq.mitiq_qiskit.qiskit_utils.`**`run_with_noise`**(*circuit*, *noise*, *shots*, *seed=None*)

Runs the quantum circuit with a depolarizing channel noise model.

Parameters

- **`circuit`** (`QuantumCircuit`) -- Ideal quantum circuit.
- **`noise`** (`float`) -- Noise constant going into *depolarizing_error*.
- **`shots`** (`int`) -- The Number of shots to run the circuit on the back-end.

- **seed** (Optional[int]) -- Optional seed for qiskit simulator.

Returns expected values.

Return type expval

3.4 Probabilistic Error Cancellation

3.4.1 Decomposition into Quasi-Probability Representation

mitiq.pec.decomposition.depolarizing.**depolarizing_decomposition**(*ideal_operation*, *noise_level*)

As described in [Temme2017], optimally decompose a single-qubit ideal_operation $\mathcal{U}_\beta$ into its quasi-probability representation (QPR), which is a linear combination of noisy implementable operations $\{\eta_\alpha \mathcal{O}_\alpha\}$.

This function assumes depolarizing noise is the only noise present. In particular, it assumes that the basis of implementable operations includes any desired ideal operation followed by a depolarizing channel (meaning that all $\mathcal{O}_\alpha = \mathcal{D} \circ \mathcal{U}$, where $\mathcal{D}(\rho) = (1-\epsilon)\rho + \epsilon I/(2^n)$). Given that assumption, it was proven in [Takagi2020] that this method gives an optimal decomposition for a given depolarizing noise_level (we can easily calculate ϵ from this noise_level value). For a single-qubit ideal_operation, the optimal decomposition is as follows:

$$\mathcal{U}_\beta = \eta_1 \mathcal{O}_1 + \eta_2 \mathcal{O}_2 + \eta_3 \mathcal{O}_3 + \eta_4 \mathcal{O}_4$$

$$\begin{aligned}
\eta_1 &= 1 + \frac{3}{4}\frac{\epsilon}{1-\epsilon}, & \mathcal{O}_1 &= \mathcal{D} \circ \mathcal{I} \circ \mathcal{U} \\
\eta_2 &= -\frac{1}{4}\frac{\epsilon}{1-\epsilon}, & \mathcal{O}_2 &= \mathcal{D} \circ \mathcal{X} \circ \mathcal{U} \\
\eta_3 &= -\frac{1}{4}\frac{\epsilon}{1-\epsilon}, & \mathcal{O}_3 &= \mathcal{D} \circ \mathcal{Y} \circ \mathcal{U} \\
\eta_4 &= -\frac{1}{4}\frac{\epsilon}{1-\epsilon}, & \mathcal{O}_4 &= \mathcal{D} \circ \mathcal{Z} \circ \mathcal{U}
\end{aligned}$$

Parameters

- **ideal_operation** (Operation) -- The input ideal operation (gate + qubit) to decompose.
- **noise_level** (float) -- The noise level (as a float) of the depolarizing channel.

Return type List[Tuple[float, List[Operation]]]

Returns The quasi-probability representation (QPR) of the ideal_operation, encoded as a list of tuples, where the first element in each tuple is a float coefficient, and the second element is a list of Cirq Operation objects to replace the ideal_operation with.

Note: In the description we say "noisy implementable operation" but we return lists of operations. This is a subtle point, relying on our assumption that we can implement *any* ideal_operation followed by a single depolarizing noise channel. When running on a simulator or QPU, this assumption breaks down for high noise_level values.

3.4.2 Probabilistic Error Cancellation (High-Level Tools)

High-level probabilistic error cancellation tools.

exception mitiq.pec.pec.**LargeSampleWarning**

Warning is raised when PEC sample size is greater than 10 ** 5

mitiq.pec.pec.**execute_with_pec**(*circuit*, *executor*, *decomposition_dict*, *precision=0.03*, *num_samples=None*, *random_state=None*, *full_output=False*)

Evaluates the expectation value associated to the input circuit using probabilistic error cancellation (PEC) [Temme2017] [Endo2018].

This function implements PEC by:

1. Sampling different implementable circuits from the quasi-probability representation of the input circuit;
2. Evaluating the noisy expectation values associated to the sampled circuits (through the "executor" function provided by the user);
3. Estimating the ideal expectation value from a suitable linear combination of the noisy ones.

Parameters

- **circuit** (Union[Circuit, Program, QuantumCircuit]) -- The input circuit to execute with error-mitigation.
- **executor** (Callable[[Union[Circuit, Program, QuantumCircuit]], float]) -- A function which executes a circuit and returns an expectation value.
- **decomposition_dict** (Dict[Operation, List[Tuple[float, List[Operation]]]]) -- The decomposition dictionary containing the quasi-probability representation of the ideal operations (those which are part of the input circuit).
- **num_samples** (Optional[int]) -- The number of noisy circuits to be sampled for PEC. If not given, this is deduced from the argument 'precision'.
- **precision** (float) -- The desired estimation precision (assuming the observable is bounded by 1). The number of samples is deduced according to the formula (one_norm / precision) ** 2, where 'one_norm' is related to the negativity of the quasi-probability representation [Temme2017]. If 'num_samples' is explicitly set by the user, 'precision' is ignored and has no effect.
- **random_state** (Union[None, int, RandomState]) -- Seed for sampling circuits.
- **full_output** (bool) -- If False only the average PEC value is returned. If True an estimate of the associated error is returned too.

Returns

The PEC estimate of the ideal expectation value associated to the input circuit.

pec_error: The estimated error between the mitigated 'pec_value' and the actual ideal expectation value. This is estimated as the ratio pec_std / sqrt(num_samples), where 'pec_std' is the standard deviation of the PEC samples, i.e., the square root of the mean squared deviation of the sampled values from 'pec_value'. This is returned only if 'full_output' is True.

Return type pec_value

3.4.3 Sampling from a Noisy Decomposition of an Ideal Operation

Tools for sampling from the noisy decomposition of ideal operations.

`mitiq.pec.sampling.`**`sample_circuit`**(*ideal_circuit*, *decomposition_dict*, *random_state=None*)

Samples an implementable circuit according from the PEC decomposition of the input ideal circuit. Moreover it also returns the "sign" and "norm" parameters which are necessary for the Monte Carlo estimation.

Parameters

- **ideal_circuit** (`Circuit`) -- The ideal circuit from which an implementable circuit is sampled.
- **decomposition_dict** (`Dict[Operation, List[Tuple[float, List[Operation]]]]`) -- The decomposition dictionary containing the quasi- probability representation of the ideal operations (those which are part of "ideal_circuit").
- **random_state** (`Union[None, int, RandomState]`) -- Seed for sampling.

Returns

The sampled implementable circuit. sign: The sign associated to sampled_circuit. norm: The one norm of the decomposition coefficients

(of the full circuit).

Return type imp_circuit

`mitiq.pec.sampling.`**`sample_sequence`**(*ideal_operation*, *decomposition_dict*, *random_state=None*)

Samples an implementable sequence from the PEC decomposition of the input ideal operation. Moreover it also returns the "sign" and "norm" parameters which are necessary for the Monte Carlo estimation.

Parameters

- **ideal_operation** (`Operation`) -- The ideal operation from which an implementable sequence is sampled.
- **decomposition_dict** (`Dict[Operation, List[Tuple[float, List[Operation]]]]`) -- The decomposition dictionary from which the decomposition of the input ideal_operation can be extracted.
- **random_state** (`Union[None, int, RandomState]`) -- Seed for sampling.

Returns

The sampled implementable sequence as list of one or more operations.

sign: The sign associated to sampled sequence. norm: The one norm of the decomposition coefficients.

Return type imp_seq

3.4.4 Probabilistic Error Cancellation (Utilities)

Utilities related to probabilistic error cancellation.

`mitiq.pec.utils.`**`get_coefficients`**(*ideal_operation*, *decomposition_dict*)

Extracts, from the input decomposition dictionary, the decomposition coefficients associated to the input ideal_operation.

Parameters

- **ideal_operation** (`Operation`) -- The input ideal operation.
- **decomposition_dict** (`Dict[Operation, List[Tuple[float, List[Operation]]]]`) -- The input decomposition dictionary.

Return type `List[float]`

Returns The decomposition coefficients of the input operation.

`mitiq.pec.utils.`**`get_imp_sequences`**(*ideal_operation*, *decomposition_dict*)

Extracts, from the input decomposition dictionary, the list of implementable sequences associated to the input ideal_operation.

Parameters

- **ideal_operation** (`Operation`) -- The input ideal operation.
- **decomposition_dict** (`Dict[Operation, List[Tuple[float, List[Operation]]]]`) -- The input decomposition dictionary.

Return type `List[List[Operation]]`

Returns The list of implementable sequences.

`mitiq.pec.utils.`**`get_one_norm`**(*ideal_operation*, *decomposition_dict*)

Extracts, from the input decomposition dictionary, the one-norm (i.e. the sum of absolute values) of the the decomposition coefficients associated to the input ideal_operation.

Parameters

- **ideal_operation** (`Operation`) -- The input ideal operation.
- **decomposition_dict** (`Dict[Operation, List[Tuple[float, List[Operation]]]]`) -- The input decomposition dictionary.

Return type `float`

Returns The one-norm of the decomposition coefficients.

`mitiq.pec.utils.`**`get_probabilities`**(*ideal_operation*, *decomposition_dict*)

Evaluates, from the input decomposition dictionary, the normalized probability distribution associated to the input ideal_operation.

Sampling implementable sequences with this distribution (taking into account the corresponding "sign") approximates the exact decomposition of the input ideal_operation.

Parameters

- **ideal_operation** (`Operation`) -- The input ideal operation.
- **decomposition_dict** (`Dict[Operation, List[Tuple[float, List[Operation]]]]`) -- The input decomposition dictionary.

Return type `List[float]`

Returns The probability distribution suitable for Monte Carlo sampling.

3.5 Utils

Utility functions.

3.6 Zero Noise Extrapolation

3.6.1 Zero Noise Extrapolation (High-Level Tools)

High-level zero-noise extrapolation tools.

`mitiq.zne.zne.`**`execute_with_zne`**(*qp*, *executor*, *factory=None*, *scale_noise=<function fold_gates_at_random>*, *num_to_average=1*)

Returns the zero-noise extrapolated expectation value that is computed by running the quantum program *qp* with the executor function.

Parameters

- **qp** (`Union[Circuit, Program, QuantumCircuit]`) -- Quantum program to execute with error mitigation.
- **executor** (`Callable[[Union[Circuit, Program, QuantumCircuit]], float]`) -- Executes a circuit and returns an expectation value.
- **factory** (`Optional[`*`Factory`*`]`) -- Factory object that determines the zero-noise extrapolation method.
- **scale_noise** (`Callable[[Union[Circuit, Program, QuantumCircuit], float], Union[Circuit, Program, QuantumCircuit]]`) -- Function for scaling the noise of a quantum circuit.
- **num_to_average** (`int`) -- Number of times expectation values are computed by the executor after each call to scale_noise, then averaged.

Return type `float`

`mitiq.zne.zne.`**`mitigate_executor`**(*executor*, *factory=None*, *scale_noise=<function fold_gates_at_random>*, *num_to_average=1*)

Returns an error-mitigated version of the input *executor*.

The input *executor* executes a circuit with an arbitrary backend and produces an expectation value (without any error mitigation). The returned executor executes the circuit with the same backend but uses zero-noise extrapolation to produce a mitigated expectation value.

Parameters

- **executor** (`Callable[[Union[Circuit, Program, QuantumCircuit]], float]`) -- Executes a circuit and returns an expectation value.
- **factory** (`Optional[`*`Factory`*`]`) -- Factory object determining the zero-noise extrapolation method.
- **scale_noise** (`Callable[[Union[Circuit, Program, QuantumCircuit], float], Union[Circuit, Program, QuantumCircuit]]`) -- Function for scaling the noise of a quantum circuit.
- **num_to_average** (`int`) -- Number of times expectation values are computed by the executor after each call to scale_noise, then averaged.

Return type `Callable[[Union[Circuit, Program, QuantumCircuit]], float]` `mitiq.zne.zne.`**`zne_decorator`**(*factory=None*, *scale_noise=<function fold_gates_at_random>*, *num_to_average=1*)

Decorator which adds error mitigation to an executor function, i.e., a function which executes a quantum circuit with an arbitrary backend and returns an expectation value.

Parameters

- **`factory`** (`Optional[Factory]`) -- Factory object determining the zero-noise extrapolation method.
- **`scale_noise`** (`Callable[[Union[Circuit, Program, QuantumCircuit], float], Union[Circuit, Program, QuantumCircuit]]`) -- Function for scaling the noise of a quantum circuit.
- **`num_to_average`** (`int`) -- Number of times expectation values are computed by the executor after each call to scale_noise, then averaged.

Return type `Callable[[Callable[[Union[Circuit, Program, QuantumCircuit]], float]], Callable[[Union[Circuit, Program, QuantumCircuit]], float]]`

3.6.2 Inference and Extrapolation: Factories

Classes corresponding to different zero-noise extrapolation methods.

`class` `mitiq.zne.inference.`**`AdaExpFactory`**(*steps*, *scale_factor=2.0*, *asymptote=None*, *avoid_log=False*, *max_scale_factor=6.0*)

Factory object implementing an adaptive zero-noise extrapolation algorithm assuming an exponential ansatz y(x) = a + b * exp(-c * x), with c > 0.

The noise scale factors are are chosen adaptively at each step, depending on the history of collected results.

If y(x->inf) is unknown, the ansatz y(x) is fitted with a non-linear optimization.

If y(x->inf) is given and avoid_log=False, the exponential model is mapped into a linear model by logarithmic transformation.

Parameters

- **`steps`** (`int`) -- The number of optimization steps. At least 3 are necessary.
- **`scale_factor`** (`float`) -- The second noise scale factor (the first is always 1.0). Further scale factors are adaptively determined.
- **`asymptote`** (`Optional[float]`) -- The infinite-noise limit y(x->inf) (optional argument).
- **`avoid_log`** (`bool`) -- If set to True, the exponential model is not linearized with a logarithm and a non-linear fit is applied even if asymptote is not None. The default value is False.
- **`max_scale_factor`** (`float`) -- Maximum noise scale factor. Default is 6.0.

Raises

- **`ValueError`** -- If data is not consistent with the extrapolation model.
- ***`ExtrapolationError`*** -- If the extrapolation fit fails.
- ***`ExtrapolationWarning`*** -- If the extrapolation fit is ill-conditioned.

`static extrapolate`(*scale_factors*, *exp_values*, *asymptote=None*, *avoid_log=False*, *eps=1e-06*, *full_output=False*)

Static method which evaluates the extrapolation to the zero-noise limit assuming an exponential ansatz y(x) = a + b * exp(-c * x), with c > 0.

If y(x->inf) is unknown, the ansatz y(x) is fitted with a non-linear optimization.

If y(x->inf) is given and avoid_log=False, the exponential model is mapped into a linear model by a logarithmic transformation.

Parameters

- **`scale_factors`** (`Sequence[float]`) -- The array of noise scale factors.
- **`exp_values`** (`Sequence[float]`) -- The array of expectation values.
- **`asymptote`** (`Optional[float]`) -- The infinite-noise limit y(x->inf) (optional argument).
- **`avoid_log`** (`bool`) -- If set to True, the exponential model is not linearized with a logarithm and a non-linear fit is applied even if asymptote is not None. The default value is False.
- **`eps`** (`float`) -- Epsilon to regularize log(sign(scale_factors - asymptote)) when the argument is to close to zero or negative.
- **`full_output`** (`bool`) -- If False (default), only the zero-noise limit is returned. If True, additional results are returned too.

Returns

The extrapolated zero-noise limit. If "full_output" is False (default value), only this parameter is returned.

zne_error: The standard deviation of the extrapolated zero-noise limit deduced from the covariance matrix "params_cov".

opt_params: The parameter array of the best fitting model. params_cov: The parameter covariance matrix of the best fitting

model.

zne_curve: The callable function which best fit the input data. It maps a real noise scale factor to a real expectation value. It is equal "zne_limit" when evaluated at zero.

Return type zne_limit

Raises

- **`ValueError`** -- If the arguments are not consistent with the extrapolation model.
- ***`ExtrapolationError`*** -- If the extrapolation fit fails.
- ***`ExtrapolationWarning`*** -- If the extrapolation fit is ill-conditioned.

Note: This method computes the zero-noise limit only from the information contained in the input arguments. To extrapolate from the internal data of an instantiated Factory object, the bound method ".reduce()" should be called instead.

`is_converged``()`

Returns True if all the needed expectation values have been computed, else False.

Return type `bool`

`next``()`

Returns a dictionary of parameters to execute a circuit at.

Return type `Dict[str, float]`

reduce()

Returns the zero-noise limit found by fitting an exponential model to the internal data stored in the factory.

Return type float

Returns The zero-noise limit.

class mitiq.zne.inference.**AdaptiveFactory**

Abstract class designed to adaptively produce a new noise scaling parameter based on a historical stack of previous noise scale parameters ("self._instack") and previously estimated expectation values ("self._outstack").

Specific zero-noise extrapolation algorithms which are adaptive are derived from this class.

abstract is_converged()

Returns True if all needed expectation values have been computed, else False.

Return type bool

abstract next()

Returns a dictionary of parameters to execute a circuit at.

Return type Dict[str, float]

abstract reduce()

Returns the extrapolation to the zero-noise limit.

Return type float

run(*qp*, *executor*, *scale_noise*, *num_to_average=1*, *max_iterations=100*)

Evaluates a sequence of expectation values by executing quantum circuits until enough data is collected (or iterations reach "max_iterations").

Parameters

- **qp** (Union[Circuit, Program, QuantumCircuit]) -- Circuit to mitigate.
- **executor** (Callable[..., float]) -- Function executing a circuit; returns an expectation value. If shot_list is not None, then "shot" must be an additional argument of the executor.
- **scale_noise** (Callable[[Union[Circuit, Program, QuantumCircuit], float], Union[Circuit, Program, QuantumCircuit]]) -- Function that scales the noise level of a quantum circuit.
- **num_to_average** (int) -- Number of times expectation values are computed by the executor after each call to scale_noise, then averaged.
- **max_iterations** (int) -- Maximum number of iterations (optional).

Return type *AdaptiveFactory*

run_classical(*scale_factor_to_expectation_value*, *max_iterations=100*)

Evaluates a sequence of expectation values until enough data is collected (or iterations reach "max_iterations").

Parameters

- **scale_factor_to_expectation_value** (Callable[..., float]) -- Function mapping a noise scale factor to an expectation value. If shot_list is not None, "shots" must be an argument of this function.
- **max_iterations** (int) -- Maximum number of iterations (optional). Default: 100.

Raises ***ConvergenceWarning*** -- If iteration loop stops before convergence.

Return type *AdaptiveFactory*

class mitiq.zne.inference.**BatchedFactory**(*scale_factors*, *shot_list=None*)

Abstract class of a non-adaptive Factory initialized with a pre-determined set of scale factors.

Specific (non-adaptive) extrapolation algorithms are derived from this class by defining the *reduce* method.

Parameters

- **scale_factors** (Sequence[float]) --
- **shot_list** (Optional[List[int]]) --

run(*qp*, *executor*, *scale_noise*, *num_to_average=1*)

Computes the expectation values at each scale factor and stores them in the factory. If the executor returns a single expectation value, the circuits are run sequentially. If the executor is batched and returns a list of expectation values (one for each circuit), then the circuits are sent to the backend as a single job. To detect if an executor is batched, it must be annotated with a return type that is one of the following:

- Iterable[float]
- List[float]
- Sequence[float]
- Tuple[float]
- numpy.ndarray

Parameters

- **qp** (Union[Circuit, Program, QuantumCircuit]) -- Quantum circuit to run.
- **executor** (Union[Callable[..., float], Callable[..., List[float]]]) -- A "single executor" (1) or a "batched executor" (2). (1) A function which inputs a single circuit and outputs a single expectation value of interest. (2) A function which inputs a list of circuits and outputs a list of expectation values (one for each circuit). A batched executor can also take an optional "kwargs_list" argument to set a list of keyword arguments (one for each circuit). This is necessary only if the factory is initialized using the optional "shot_list" parameter.
- **scale_noise** (Callable[[Union[Circuit, Program, QuantumCircuit], float], Union[Circuit, Program, QuantumCircuit]]) -- Noise scaling function.
- **num_to_average** (int) -- The number of circuits executed for each noise scale factor. This parameter can be used to increase the precision of the "executor" or to average the effect of a non-deterministic "scale_noise" function.

Return type *BatchedFactory*

run_classical(*scale_factor_to_expectation_value*)

Computes expectation values by calling the input function at each scale factor.

Parameters **scale_factor_to_expectation_value** (Callable[..., float]) -- Function mapping a noise scale factor to an expectation value. If shot_list is not None, "shots" must be an argument of this function.

Return type *BatchedFactory*

exception mitiq.zne.inference.**ConvergenceWarning**

Warning raised by *Factory* objects when their *run_classical* method fails to converge.

class mitiq.zne.inference.**ExpFactory**(*scale_factors*, *asymptote=None*, *avoid_log=False*, *shot_list=None*)

Factory object implementing a zero-noise extrapolation algorithm assuming an exponential ansatz y(x) = a + b * exp(-c * x), with c > 0.

If y(x->inf) is unknown, the ansatz y(x) is fitted with a non-linear optimization.

If y(x->inf) is given and avoid_log=False, the exponential model is mapped into a linear model by a logarithmic transformation.

Parameters

- **scale_factors** (Sequence[float]) -- Sequence of noise scale factors at which expectation values should be measured.
- **asymptote** (Optional[float]) -- Infinite-noise limit (optional argument).
- **avoid_log** (bool) -- If set to True, the exponential model is not linearized with a logarithm and a non-linear fit is applied even if asymptote is not None. The default value is False.
- **shot_list** (Optional[List[int]]) -- Optional sequence of integers corresponding to the number of samples taken for each expectation value. If this argument is explicitly passed to the factory, it must have the same length of scale_factors and the executor function must accept "shots" as a valid keyword argument.

Raises

- **ValueError** -- If data is not consistent with the extrapolation model.
- ***ExtrapolationError*** -- If the extrapolation fit fails.
- ***ExtrapolationWarning*** -- If the extrapolation fit is ill-conditioned.

static extrapolate(*scale_factors*, *exp_values*, *asymptote=None*, *avoid_log=False*, *eps=1e-06*, *full_output=False*)

Static method which evaluates the extrapolation to the zero-noise limit assuming an exponential ansatz y(x) = a + b * exp(-c * x), with c > 0.

If y(x->inf) is unknown, the ansatz y(x) is fitted with a non-linear optimization.

If y(x->inf) is given and avoid_log=False, the exponential model is mapped into a linear model by a logarithmic transformation.

Parameters

- **scale_factors** (Sequence[float]) -- The array of noise scale factors.
- **exp_values** (Sequence[float]) -- The array of expectation values.
- **asymptote** (Optional[float]) -- The infinite-noise limit y(x->inf) (optional argument).
- **avoid_log** (bool) -- If set to True, the exponential model is not linearized with a logarithm and a non-linear fit is applied even if asymptote is not None. The default value is False.
- **eps** (float) -- Epsilon to regularize log(sign(scale_factors - asymptote)) when the argument is to close to zero or negative.
- **full_output** (bool) -- If False (default), only the zero-noise limit is returned. If True, additional information about the extrapolated limit is returned too.

Returns

The extrapolated zero-noise limit. If "full_output" is False (default value), only this parameter is returned.

zne_error: The error associated to the extrapolated zero-noise limit deduced from the covariance matrix "params_cov".

opt_params: The parameter array of the best fitting model. params_cov: The parameter covariance matrix of the best fitting

model.

zne_curve: The callable function which best fit the input data. It maps a real noise scale factor to a real expectation value. It is equal "zne_limit" when evaluated at zero.

Return type zne_limit

Raises

- **ValueError** -- If the arguments are not consistent with the extrapolation model.
- ***ExtrapolationError*** -- If the extrapolation fit fails.
- ***ExtrapolationWarning*** -- If the extrapolation fit is ill-conditioned.

Note: This method computes the zero-noise limit only from the information contained in the input arguments. To extrapolate from the internal data of an instantiated Factory object, the bound method ".reduce()" should be called instead.

---

reduce()

Returns the zero-noise limit found by fitting an exponential model to the internal data stored in the factory.

Return type `float`

Returns The zero-noise limit.

exception mitiq.zne.inference.**ExtrapolationError**

Error raised by *Factory* objects when the extrapolation fit fails.

exception mitiq.zne.inference.**ExtrapolationWarning**

Warning raised by *Factory* objects when the extrapolation fit is ill-conditioned.

class mitiq.zne.inference.**Factory**

Abstract base class which performs the classical parts of zero-noise extrapolation. This minimally includes:

- scaling circuits,
- sending jobs to execute,
- collecting the results,
- fitting the collected data,
- Extrapolating to the zero-noise limit.

If all scale factors are set a priori, the jobs can be batched. This is handled by a BatchedFactory.

If the next scale factor depends on the previous history of results, jobs are run sequentially. This is handled by an AdaptiveFactory.

get_expectation_values()

Returns the expectation values computed by the factory.

Return type `ndarray`

get_extrapolation_curve()
Returns the extrapolation curve, i.e., a function which inputs a noise scale factor and outputs the associated expectation value. This function is the solution of the regression problem used to evaluate the zero-noise extrapolation.

Return type Callable[[float], float]

get_optimal_parameters()
Returns the optimal model parameters produced by the extrapolation fit.

Return type ndarray

get_parameters_covariance()
Returns the covariance matrix of the model parameters produced by the extrapolation fit.

Return type ndarray

get_scale_factors()
Returns the scale factors at which the factory has computed expectation values.

Return type ndarray

get_zero_noise_limit()
Returns the last evaluation of the zero-noise limit computed by the factory. To re-evaluate its value, the method 'reduce' should be called first.

Return type float

get_zero_noise_limit_error()
Returns the extrapolation error representing the uncertainty affecting the zero-noise limit. It is deduced by error propagation from the covariance matrix associated to the fit parameters.

Note: this quantity is only related to the ability of the model to fit the measured data. Therefore, it may underestimate the actual error existing between the zero-noise limit and the true ideal expectation value.

Return type float

push(*instack_val*, *outstack_val*)
Appends "instack_val" to "self._instack" and "outstack_val" to "self._outstack". Each time a new expectation value is computed this method should be used to update the internal state of the Factory.

Parameters

- **instack_val** (Dict[str, float]) --
- **outstack_val** (float) --

Return type *Factory*

abstract reduce()
Returns the extrapolation to the zero-noise limit.

Return type float

reset()
Resets the internal state of the Factory.

Return type *Factory*

abstract run(*qp*, *executor*, *scale_noise*, *num_to_average=1*)
Calls the executor function on noise-scaled quantum circuit and stores the results.

Parameters

- **qp** (Union[Circuit, Program, QuantumCircuit]) -- Quantum circuit to scale noise in.
- **executor** (Callable[..., float]) -- Function which inputs a (list of) quantum circuits and outputs a (list of) expectation values.
- **scale_noise** (Callable[[Union[Circuit, Program, QuantumCircuit], float], Union[Circuit, Program, QuantumCircuit]]) -- Function which inputs a quantum circuit and outputs a noise-scaled quantum circuit.
- **num_to_average** (int) -- Number of times the executor function is called on each noise-scaled quantum circuit.

Return type *Factory*

abstract run_classical(*scale_factor_to_expectation_value*)

Calls the function scale_factor_to_expectation_value at each scale factor of the factory, and stores the results.

Parameters **scale_factor_to_expectation_value** (Callable[..., float]) -- A function which inputs a scale factor and outputs an expectation value. This does not have to involve a quantum processor making this a "classical analogue" of the run method.

Return type *Factory*

class mitiq.zne.inference.**LinearFactory**(*scale_factors*, *shot_list=None*)

Factory object implementing zero-noise extrapolation based on a linear fit.

Parameters

- **scale_factors** (Sequence[float]) -- Sequence of noise scale factors at which expectation values should be measured.
- **shot_list** (Optional[List[int]]) -- Optional sequence of integers corresponding to the number of samples taken for each expectation value. If this argument is explicitly passed to the factory, it must have the same length of scale_factors and the executor function must accept "shots" as a valid keyword argument.

Raises

- **ValueError** -- If data is not consistent with the extrapolation model.
- ***ExtrapolationWarning*** -- If the extrapolation fit is ill-conditioned.

Example

```
>>> NOISE_LEVELS = [1.0, 2.0, 3.0]
>>> fac = LinearFactory(NOISE_LEVELS)
```

static extrapolate(*scale_factors*, *exp_values*, *full_output=False*)

Static method which evaluates the linear extrapolation to the zero-noise limit.

Parameters

- **scale_factors** (Sequence[float]) -- The array of noise scale factors.
- **exp_values** (Sequence[float]) -- The array of expectation values.
- **full_output** (bool) -- If False (default), only the zero-noise limit is returned. If True, additional results are returned too.

Returns

The extrapolated zero-noise limit. If "full_output" is False (default value), only this parameter is returned.

zne_error: The error associated to the extrapolated zero-noise limit deduced from the covariance matrix "params_cov".

opt_params: The parameter array of the best fitting model. params_cov: The parameter covariance matrix of the best fitting

model.

zne_curve: The callable function which best fit the input data. It maps a real noise scale factor to a real expectation value. It is equal "zne_limit" when evaluated at zero.

Return type zne_limit

Raises *ExtrapolationWarning* -- If the extrapolation fit is ill-conditioned.

Note: This method computes the zero-noise limit only from the information contained in the input arguments. To extrapolate from the internal data of an instantiated Factory object, the bound method ".reduce()" should be called instead.

---

reduce()

Returns the zero-noise limit found by fitting a line to the internal data stored in the factory.

Return type `float`

Returns The zero-noise limit.

class mitiq.zne.inference.**PolyExpFactory**(*scale_factors*, *order*, *asymptote=None*, *avoid_log=False*, *shot_list=None*)

Factory object implementing a zero-noise extrapolation algorithm assuming an (almost) exponential ansatz with a non linear exponent, i.e.:

y(x) = a + sign * exp(z(x)), where z(x) is a polynomial of a given order.

The parameter "sign" is a sign variable which can be either 1 or -1, corresponding to decreasing and increasing exponentials, respectively. The parameter "sign" is automatically deduced from the data.

If y(x->inf) is unknown, the ansatz y(x) is fitted with a non-linear optimization.

If y(x->inf) is given and avoid_log=False, the exponential model is mapped into a polynomial model by logarithmic transformation.

Parameters

- **scale_factors** (`Sequence[float]`) -- Sequence of noise scale factors at which expectation values should be measured.
- **order** (`int`) -- Extrapolation order (degree of the polynomial z(x)). It cannot exceed len(scale_factors) - 1. If asymptote is None, order cannot exceed len(scale_factors) - 2.
- **asymptote** (`Optional[float]`) -- The infinite-noise limit y(x->inf) (optional argument).
- **avoid_log** (`bool`) -- If set to True, the exponential model is not linearized with a logarithm and a non-linear fit is applied even if asymptote is not None. The default value is False.
- **shot_list** (`Optional[List[int]]`) -- Optional sequence of integers corresponding to the number of samples taken for each expectation value. If this argument is explicitly passed

to the factory, it must have the same length of scale_factors and the executor function must accept "shots" as a valid keyword argument.

Raises

- **ValueError** -- If data is not consistent with the extrapolation model.
- *ExtrapolationError* -- If the extrapolation fit fails.
- *ExtrapolationWarning* -- If the extrapolation fit is ill-conditioned.

static extrapolate(*scale_factors*, *exp_values*, *order*, *asymptote=None*, *avoid_log=False*, *eps=1e-06*, *full_output=False*)

Static method which evaluates the extrapolation to the zero-noise limit with an exponential ansatz (whose exponent is a polynomial of degree "order").

The exponential ansatz is y(x) = a + sign * exp(z(x)) where z(x) is a polynomial and "sign" is either +1 or -1 corresponding to decreasing and increasing exponentials, respectively. The parameter "sign" is automatically deduced from the data.

It is also assumed that z(x-->inf) = -inf, such that y(x-->inf) --> a.

If asymptote is None, the ansatz y(x) is fitted with a non-linear optimization.

If asymptote is given and avoid_log=False, a linear fit with respect to z(x) := log[sign * (y(x) - asymptote)] is performed.

Parameters

- **scale_factors** (Sequence[float]) -- The array of noise scale factors.
- **exp_values** (Sequence[float]) -- The array of expectation values.
- **asymptote** (Optional[float]) -- The infinite-noise limit y(x->inf) (optional argument).
- **order** (int) -- The degree of the polynomial z(x).
- **avoid_log** (bool) -- If set to True, the exponential model is not linearized with a logarithm and a non-linear fit is applied even if asymptote is not None. The default value is False.
- **eps** (float) -- Epsilon to regularize log(sign(scale_factors - asymptote)) when the argument is to close to zero or negative.
- **full_output** (bool) -- If False (default), only the zero-noise limit is returned. If True, additional information about the extrapolated limit is returned too.

Returns

The extrapolated zero-noise limit. If "full_output" is False (default value), only this parameter is returned.

zne_error: The error associated to the extrapolated zero-noise limit deduced from the covariance matrix "params_cov".

opt_params: The parameter array of the best fitting model. params_cov: The parameter covariance matrix of the best fitting

model.

zne_curve: The callable function which best fit the input data. It maps a real noise scale factor to a real expectation value. It is equal "zne_limit" when evaluated at zero.

Return type zne_limit

Raises

- **ValueError** -- If the arguments are not consistent with the extrapolation model.
- *ExtrapolationError* -- If the extrapolation fit fails.
- *ExtrapolationWarning* -- If the extrapolation fit is ill-conditioned.

Note: This method computes the zero-noise limit only from the information contained in the input arguments. To extrapolate from the internal data of an instantiated Factory object, the bound method ".reduce()" should be called instead.

---

reduce()

Returns the zero-noise limit found by fitting an the model to the internal data stored in the factory.

Return type `float`

Returns The zero-noise limit.

class `mitiq.zne.inference.`**PolyFactory**(*scale_factors*, *order*, *shot_list=None*)

Factory object implementing a zero-noise extrapolation algorithm based on a polynomial fit.

Parameters

- **scale_factors** (`Sequence[float]`) -- Sequence of noise scale factors at which expectation values should be measured.
- **order** (`int`) -- Extrapolation order (degree of the polynomial fit). It cannot exceed len(scale_factors) - 1.
- **shot_list** (`Optional[List[int]]`) -- Optional sequence of integers corresponding to the number of samples taken for each expectation value. If this argument is explicitly passed to the factory, it must have the same length of scale_factors and the executor function must accept "shots" as a valid keyword argument.

Raises

- **ValueError** -- If data is not consistent with the extrapolation model.
- *ExtrapolationWarning* -- If the extrapolation fit is ill-conditioned.

Note: RichardsonFactory and LinearFactory are special cases of PolyFactory.

---

static extrapolate(*scale_factors*, *exp_values*, *order*, *full_output=False*)

Static method which evaluates a polynomial extrapolation to the zero-noise limit.

Parameters

- **scale_factors** (`Sequence[float]`) -- The array of noise scale factors.
- **exp_values** (`Sequence[float]`) -- The array of expectation values.
- **order** (`int`) -- The extrapolation order (degree of the polynomial fit).
- **full_output** (`bool`) -- If False (default), only the zero-noise limit is returned. If True, additional information about the extrapolated limit is returned too.

Returns

The extrapolated zero-noise limit. If "full_output" is False (default value), only this parameter is returned.

zne_error: The error associated to the extrapolated zero-noise limit deduced from the covariance matrix "params_cov".

opt_params: The parameter array of the best fitting model. params_cov: The parameter covariance matrix of the best fitting

model.

zne_curve: The callable function which best fit the input data. It maps a real noise scale factor to a real expectation value. It is equal "zne_limit" when evaluated at zero.

Return type zne_limit

Raises *ExtrapolationWarning* -- If the extrapolation fit is ill-conditioned.

Note: This method computes the zero-noise limit only from the information contained in the input arguments. To extrapolate from the internal data of an instantiated Factory object, the bound method ".reduce()" should be called instead.

---

reduce()

Evaluates the zero-noise limit found by fitting a polynomial of degree *self.order* to the internal data stored in the factory.

Return type `float`

Returns The zero-noise limit.

class mitiq.zne.inference.**RichardsonFactory**(*scale_factors*, *shot_list=None*)

Factory object implementing Richardson extrapolation.

Parameters

- **scale_factors** (`Sequence[float]`) -- Sequence of noise scale factors at which expectation values should be measured.
- **shot_list** (`Optional[List[int]]`) -- Optional sequence of integers corresponding to the number of samples taken for each expectation value. If this argument is explicitly passed to the factory, it must have the same length of scale_factors and the executor function must accept "shots" as a valid keyword argument.

Raises

- **ValueError** -- If data is not consistent with the extrapolation model.
- *ExtrapolationWarning* -- If the extrapolation fit is ill-conditioned.

static extrapolate(*scale_factors*, *exp_values*, *full_output=False*)

Static method which evaluates the Richardson extrapolation to the zero-noise limit.

Parameters

- **scale_factors** (`Sequence[float]`) -- The array of noise scale factors.
- **exp_values** (`Sequence[float]`) -- The array of expectation values.
- **full_output** (`bool`) -- If False (default), only the zero-noise limit is returned. If True, additional results are returned too.

Returns

The extrapolated zero-noise limit. If "full_output" is False (default value), only this parameter is returned.

zne_error: The error associated to the extrapolated zero-noise limit deduced from the covariance matrix "params_cov".

opt_params: The parameter array of the best fitting model. params_cov: The parameter covariance matrix of the best fitting

model.

zne_curve: The callable function which best fit the input data. It maps a real noise scale factor to a real expectation value. It is equal "zne_limit" when evaluated at zero.

Return type zne_limit

Raises *ExtrapolationWarning* -- If the extrapolation fit is ill-conditioned.

Note: This method computes the zero-noise limit only from the information contained in the input arguments. To extrapolate from the internal data of an instantiated Factory object, the bound method ".reduce()" should be called instead.

reduce()

Evaluates the zero-noise limit found by applying Richardson extrapolation to the internal data stored in the factory.

Return type `float`

Returns The zero-noise limit.

`mitiq.zne.inference.`**mitiq_curve_fit**(*ansatz*, *scale_factors*, *exp_values*, *init_params=None*)

This is a wrapping of the *scipy.optimize.curve_fit* function with custom errors and warnings. It is used to make a non-linear fit.

Parameters

- **ansatz** (`Callable`[..., `float`]) -- The model function used for zero-noise extrapolation. The first argument is the noise scale variable, the remaining arguments are the parameters to fit.
- **scale_factors** (`Sequence`[`float`]) -- The array of noise scale factors.
- **exp_values** (`Sequence`[`float`]) -- The array of expectation values.
- **init_params** (`Optional`[`List`[`float`]]) -- Initial guess for the parameters. If None, the initial values are set to 1.

Returns

The array of optimal parameters. params_cov: The covariance matrix of the parameters.

If ill conditioned, params_cov may contain np.inf elements.

Return type opt_params

Raises

- *ExtrapolationError* -- If the extrapolation fit fails.
- *ExtrapolationWarning* -- If the extrapolation fit is ill-conditioned.

`mitiq.zne.inference.`**mitiq_polyfit**(*scale_factors*, *exp_values*, *deg*, *weights=None*)

This is a wrapping of the *numpy.polyfit* function with custom warnings. It is used to make a polynomial fit.

Parameters

- **scale_factors** (Sequence[float]) -- The array of noise scale factors.
- **exp_values** (Sequence[float]) -- The array of expectation values.
- **deg** (int) -- The degree of the polynomial fit.
- **weights** (Optional[Sequence[float]]) -- Optional array of weights for each sampled point. This is used to make a weighted least squares fit.

Returns

The array of optimal parameters. params_cov: The covariance matrix of the parameters.

If data is not enough to estimate the covariance matrix, params_cov is returned as None.

Return type opt_params

Raises *ExtrapolationWarning* -- If the extrapolation fit is ill-conditioned.

3.6.3 Noise Scaling: Unitary Folding

Functions for local and global unitary folding on supported circuits.

exception mitiq.zne.scaling.folding.**UnfoldableCircuitError**

exception mitiq.zne.scaling.folding.**UnfoldableGateError**

mitiq.zne.scaling.folding.**fold_gates_at_random**(*circuit*, *scale_factor*, *seed=None*, ***kwargs*)

Returns a folded circuit by applying the map G -> G G^dag G to a random subset of gates in the input circuit.

The folded circuit has a number of gates approximately equal to scale_factor * n where n is the number of gates in the input circuit.

Parameters

- **circuit** (Union[Circuit, Program, QuantumCircuit]) -- Circuit to fold.
- **scale_factor** (float) -- Factor to scale the circuit by. Any real number >= 1.
- **seed** (Optional[int]) -- [Optional] Integer seed for random number generator.
- **kwargs** (Any) --

Keyword Arguments

- **fidelities** (*Dict[str, float]*) -- Dictionary of gate fidelities. Each key is a string which specifies the gate and each value is the fidelity of that gate. When this argument is provided, folded gates contribute an amount proportional to their infidelity (1 - fidelity) to the total noise scaling. Fidelity values must be in the interval (0, 1]. Gates not specified have a default fidelity of 0.99**n where n is the number of qubits the gates act on.

  Supported gate keys are listed in the following table.

  "H" | Hadamard "X" | Pauli X "Y" | Pauli Y "Z" | Pauli Z "I" | Identity "CNOT" | CNOT "CZ" | CZ gate "TOFFOLI" | Toffoli gate "single" | All single qubit gates "double" | All two-qubit gates "triple" | All three-qubit gates

  Keys for specific gates override values set by "single", "double", and "triple".

  For example, *fidelities = {"single": 1.0, "H", 0.99}* sets all single-qubit gates except Hadamard to have fidelity one.

- **squash_moments** (*bool*) -- If True, all gates (including folded gates) are placed as early as possible in the circuit. If False, new moments are created for folded gates. This option only applies to QPROGRAM types which have a "moment" or "time" structure. Default is True.
- **return_mitiq** (*bool*) -- If True, returns a mitiq circuit instead of the input circuit type (if different). Default is False.

Returns The folded quantum circuit as a QPROGRAM.

Return type folded

`mitiq.zne.scaling.folding.`**`fold_gates_from_left`**(*circuit*, *scale_factor*, ***kwargs*)

Returns a new folded circuit by applying the map G -> G G^dag G to a subset of gates of the input circuit, starting with gates at the left (beginning) of the circuit.

The folded circuit has a number of gates approximately equal to scale_factor * n where n is the number of gates in the input circuit.

Parameters

- **circuit** (`Union[Circuit, Program, QuantumCircuit]`) -- Circuit to fold.
- **scale_factor** (`float`) -- Factor to scale the circuit by. Any real number >= 1.
- **kwargs** (`Any`) --

Keyword Arguments

- **fidelities** (*Dict[str, float]*) -- Dictionary of gate fidelities. Each key is a string which specifies the gate and each value is the fidelity of that gate. When this argument is provided, folded gates contribute an amount proportional to their infidelity (1 - fidelity) to the total noise scaling. Fidelity values must be in the interval (0, 1]. Gates not specified have a default fidelity of 0.99**n where n is the number of qubits the gates act on.

  Supported gate keys are listed in the following table.

  "H" | Hadamard "X" | Pauli X "Y" | Pauli Y "Z" | Pauli Z "I" | Identity "CNOT" | CNOT "CZ" | CZ gate "TOFFOLI" | Toffoli gate "single" | All single qubit gates "double" | All two-qubit gates "triple" | All three-qubit gates

  Keys for specific gates override values set by "single", "double", and "triple".

  For example, *fidelities = {"single": 1.0, "H", 0.99}* sets all single-qubit gates except Hadamard to have fidelity one.

- **squash_moments** (*bool*) -- If True, all gates (including folded gates) are placed as early as possible in the circuit. If False, new moments are created for folded gates. This option only applies to QPROGRAM types which have a "moment" or "time" structure. Default is True.
- **return_mitiq** (*bool*) -- If True, returns a mitiq circuit instead of the input circuit type (if different). Default is False.

Returns The folded quantum circuit as a QPROGRAM.

Return type folded

`mitiq.zne.scaling.folding.`**`fold_gates_from_right`**(*circuit*, *scale_factor*, ***kwargs*)

Returns a new folded circuit by applying the map G -> G G^dag G to a subset of gates of the input circuit, starting with gates at the right (end) of the circuit.

The folded circuit has a number of gates approximately equal to scale_factor * n where n is the number of gates in the input circuit.

Parameters

- **circuit** (Union[Circuit, Program, QuantumCircuit]) -- Circuit to fold.
- **scale_factor** (float) -- Factor to scale the circuit by. Any real number >= 1.
- **kwargs** (Any) --

Keyword Arguments

- **fidelities** (*Dict[str, float]*) -- Dictionary of gate fidelities. Each key is a string which specifies the gate and each value is the fidelity of that gate. When this argument is provided, folded gates contribute an amount proportional to their infidelity (1 - fidelity) to the total noise scaling. Fidelity values must be in the interval (0, 1]. Gates not specified have a default fidelity of 0.99**n where n is the number of qubits the gates act on.

  Supported gate keys are listed in the following table.

  > "H" | Hadamard "X" | Pauli X "Y" | Pauli Y "Z" | Pauli Z "I" | Identity "CNOT" | CNOT "CZ" | CZ gate "TOFFOLI" | Toffoli gate "single" | All single qubit gates "double" | All two-qubit gates "triple" | All three-qubit gates

  Keys for specific gates override values set by "single", "double", and "triple".

  For example, *fidelities = {"single": 1.0, "H", 0.99}* sets all single-qubit gates except Hadamard to have fidelity one.

- **squash_moments** (*bool*) -- If True, all gates (including folded gates) are placed as early as possible in the circuit. If False, new moments are created for folded gates. This option only applies to QPROGRAM types which have a "moment" or "time" structure. Default is True.
- **return_mitiq** (*bool*) -- If True, returns a mitiq circuit instead of the input circuit type (if different). Default is False.

Returns The folded quantum circuit as a QPROGRAM.

Return type folded

mitiq.zne.scaling.folding.**fold_global**(*circuit*, *scale_factor*, ***kwargs*)

Returns a new circuit obtained by folding the global unitary of the input circuit.

The returned folded circuit has a number of gates approximately equal to scale_factor * len(circuit).

Parameters

- **circuit** (Union[Circuit, Program, QuantumCircuit]) -- Circuit to fold.
- **scale_factor** (float) -- Factor to scale the circuit by.
- **kwargs** (Any) --

Keyword Arguments

- **squash_moments** (*bool*) -- If True, all gates (including folded gates) are placed as early as possible in the circuit. If False, new moments are created for folded gates. This option only applies to QPROGRAM types which have a "moment" or "time" structure. Default is True.
- **return_mitiq** (*bool*) -- If True, returns a mitiq circuit instead of the input circuit type (if different). Default is False.

Returns the folded quantum circuit as a QPROGRAM.

Return type folded

`mitiq.zne.scaling.folding.`**`squash_moments`**(*circuit*)

Returns a copy of the input circuit with all gates squashed into as few moments as possible.

Parameters **`circuit`** (`Circuit`) -- Circuit to squash moments of.

Return type `Circuit`

CHAPTER 4

Citing

If you are using Mitiq for your research, please cite it:

```
@misc{Mitiq,
  author={
  LaRose, Ryan and Mari, Andrea and Shammah, Nathan and Karalekas, Peter and Zeng,␣
↪Will},
  title = {Mitiq: A software package for error mitigation on near-term quantum␣
↪computers},
  howpublished = {\url{https://github.com/unitaryfund/mitiq}},
  year={2020}
}
```

You can download the bibtex file.

If you have developed new features for error mitigation, or found bugs in mitiq, please consider contributing your code.

CHAPTER 5

Contributing

5.1 Contributing to Mitiq

Contributions are welcome, and they are greatly appreciated, every little bit helps.

5.1.1 Opening an issue

You can begin contributing to `mitiq` code by raising an issue, reporting a bug or proposing a new feature request, using the labels to organize it. Please use `mitiq.about()` to document your dependencies and working environment.

5.1.2 Opening a pull request

You can open a pull request by pushing changes from a local branch, explaining the bug fix or new feature.

Version control with git

git is a language that helps keeping track of the changes made. Have a look at these guidelines for getting started with git workflow. Use short and explanatory comments to document the changes with frequent commits.

Forking the repository

You can fork mitiq from the github repository, so that your changes are applied with respect to the current master branch. Use the Fork button, and then use git from the command line to clone your fork of the repository locally on your machine.

```
(base) git clone https://github.com/your_github_username/mitiq.git
```

You can also use SSH instead of a HTTPS protocol.

Working in a virtual environment

It is best to set up a clean environment with anaconda, to keep track of all installed applications.

```
(base) conda create -n myenv python=3
```

accept the configuration ([y]) and switch to the environment

```
(base) conda activate myenv
(myenv) conda install pip
```

Once you will finish the modifications, you can deactivate the environment with

```
(myenv) conda deactivate myenv
```

Development install

In order to install all the libraries useful for contributing to the development of the library, from your local clone of the fork, run

```
(myenv) pip install -e .
(myenv) pip install -r requirements.txt
```

Adding tests

If you add new features to a function or class, it is required to add tests for such object. Mitiq uses a nested structure for packaging tests in directories named `tests` at the same level of each module.

Updating the documentation

Follow these instructions for contributing to the documentation which include guidelines about updating the API-doc list of modules and writing examples in the users guide.

Checking local tests

You can check that tests run with `pytest`. The Makefile contains some commands for running different collections of tests for the repository.

To run just the tests contained in `mitiq/tests` and `mitiq/benchmarks/tests` run

```
(myenv) make test
```

To run the tests for the pyQuil and Qiskit plugins (which of course require for pyQuil and Qiskit to be installed) run

```
(myenv) make test-pyquil
(myenv) make test-qiskit
```

NOTE: For the pyQuil tests to run, you will need to have QVM & quilc servers running in the background. The easiest way to do this is with Docker via

```
docker run --rm -idt -p 5000:5000 rigetti/qvm -S
docker run --rm -idt -p 5555:5555 rigetti/quilc -R
```

You can also check that all tests run also in the documentation examples and docstrings with

```
(myenv) make docs
```

If you add new `/tests` directories, you will need to update the `Makefile` so that they will be included as part of continuous integration.

Style Guidelines

Mitiq code is developed according the best practices of Python development.

- Please get familiar with PEP 8 (code) and PEP 257 (docstrings) guidelines.
- Use annotations for type hints in the objects' signature.
- Write google-style docstrings.

We use Black and `flake8` to automatically lint the code and enforce style requirements as part of the CI pipeline. You can run these style tests yourself locally by running `make check-style` (to check for violations of the `flake8` rules) and `make check-format` (to see if `black` would reformat the code) in the top-level directory of the repository. If you aren't presented with any errors, then that means your code is good enough for the linter (`flake8`) and formatter (`black`). If `make check-format` fails, it will present you with a diff, which you can resolve by running `make format`. Black is very opinionated, but saves a lot of time by removing the need for style nitpicks in PR review. We only deviate from its default behavior in one category: we choose to use a line length of 79 rather than the Black default of 88 (this is configured in the ``*pyproject.toml*` <pyproject.toml>`_ file).

Code of conduct

Mitiq development abides to the Contributors' Covenant.

5.2 Contributing to the Documentation

This is the Contributors guide for the documentation of Mitiq, the Python toolkit for implementing error mitigation on quantum computers.

5.2.1 Requirements

The documentation is generated with Sphinx. The necessary packages can be installed, from the root `mitiq` directory

```
pip install -e .
pip install -r requirements.txt
```

as they are present in the `requirements.txt` file. Otherwise, with

```
pip install -U sphinx m2r sphinxcontrib-bibtex pybtex sphinx-copybutton sphinx-
↪autodoc-typehints
```

`m2r` allows to include `.md` files, besides `.rst`, in the documentation. `sphinxcontrib-bibtex` allows to include citations in a `.bib` file and `pybtex` allows to customize how they are rendered, e.g., APS-style. `sphinx-copybutton` allows to easily copy-paste code snippets from examples. `sphinx-autodoc-typehints` allows to control how annotations are displayed in the API-doc part of the documentation, integrating with `sphinx-autodoc` and `sphinx-napoleon`.

You can check that Sphinx is installed with `sphinx-build --version`.

In addition, there are two requirements, `tensorflow` and `tensorflow-quantum`, which are used solely in `guide/guide-executors.rst`. They can be installed via:

```
pip install -r docs/requirements.txt
```

If they are not installed, the test that uses them will be skipped. We do this because `tensorflow-quantum` has incompatibility issues -- version `0.4.0` works on `py38` but not Windows, and version `0.3.1` works on Windows but not `py38`. Therefore, these two requirements cannot be installed on Windows. See gh-419 for more information.

5.2.2 How to Update the Documentation

The configuration file

- Since the documentation is already created, you need not to generate a configuration file from scratch (this is done with `sphinx-quickstart`). Meta-data, extentions and other custom specifications are accounted for in the `conf.py` file.

Add features in the conf.py file

- To add specific feature to the documentation, extensions can be include. For example to add classes and functions to the API doc, make sure that autodoc extension is enabled in the `conf.py` file, and for tests the `doctest` one,

```
extensions = ['sphinx.ext.autodoc','sphinx.ext.doctest']
```

Update the guide with a tree of restructured text files

You need not to modify the `docs/build` folder, as it is automatically generated. You will modify only the `docs/source` files.

The documentation is divided into a **guide**, whose content needs to be written from scratch, and an **API-doc** part, which can be partly automatically generated.

- To add information in the guide, it is possible to include new information as a restructured text (`.rst`) or markdown (`.md`) file.

The main file is `index.rst`. It includes a `guide.rst` and an `apidoc.rst` file, as well as other files. Like in LaTeX, each file can include other files. Make sure they are included in the table of contents

```
.. toctree::
   :maxdepth: 2
   :caption: Contents:

   changelog.rst
```

You can include markdown files in the guide

- Information to the guide can also be added from markdown (`.md`) files, since `m2r` (`pip install --upgrade m2r`) is installed and added to the `conf.py` file (`extensions = ['m2r']`). Just add the `.md` file to the toctree.

To include `.md` files outside of the documentation `source` directory, you can add in `source` an `.rst` file to the toctree that contains inside it the

`.. mdinclude:: ../file.md` command, where `file.md` is the one to be added.

Automatically add information to the API doc

- New modules, classes and functions can be added by listing them in the appropriate `.rst` file (such as `autodoc.rst` or a child), e.g.,

```
Factories
---------
.. automodule:: mitiq.factories
   :members:
```

will add all elements of the `mitiq.factories` module. One can hand-pick classes and functions to add, to comment them, as well as exclude them.

Build the documentation locally

- To build the documentation, from `bash`, move to the `docs` folder and run .. code-block:: bash

  sphinx-build -b html source build

  this generates the `docs/build` folder. This folder is not kept track of in the github repository, as `docs/build` is present in the `.gitignore` file.

The `html` and `latex` and `pdf` files will be automatically created in the `docs/build` folder.

Create the html

- To create the html structure,

```
make html
```

Create the pdf

- To create the latex files and output a pdf,

```
make latexpdf
```

5.2.3 How to Test the Documentation Examples

There are several ways to check that the documentation examples work. Currently, `mitiq` is testing them with the `doctest` extension of `sphinx`. This is set in the `conf.py` file and is executed with

```
make doctest
```

from the `mitiq/docs` directory. From the root directory `mitiq`, simply run

```
make docs
```

to obtain the same result.

These equivalent commands test the code examples in the guide and ".rst" files, as well as testing the docstrings, since these are imported with the `autodoc` extension.

When writing a new example, you can use different directives in the rst file to include code blocks. One of them is

```
.. code-block:: python

   1+1         # simple example
```

In order to make sure that the block is parsed with `make doctest`, use the `testcode` directive. This can be used in pair with `testoutput`, if something is printed, and, eventually `testsetup`, to import modules or set up variables in an invisible block. An example is:

```
.. testcode:: python

   1+1         # simple example
```

with no output and

```
.. testcode:: python

   print(1+1)        # explicitly print

.. testoutput:: python

   2        # match the print message
```

The use of `testsetup` allows blocks that do not render:

```
.. testsetup:: python

   import numpy as np  # this block is not rendered in the html or pdf

.. testcode:: python

   np.array(2)

.. testoutput:: python

   array(2)
```

There is also the `doctest` directive, which allows to include interactive Python blocks. These need to be given this way:

```
  .. doctest:: python

     >>> import numpy as np
     >>> print(np.array(2))
     array(2)

  Notice that no space is left between the last input and the output.

A way to test docstrings without installing sphinx is with `\ ``pytest`` +
``doctest`` <http://doc.pytest.org/en/latest/doctest.html>`_\ :
```

```
pytest --doctest-glob='*.rst'
```

or alternatively

```
pytest --doctest-modules
```

However, this only checks `doctest` blocks, and does not recognize `testcode` blocks. Moreover, it does not parse the `conf.py` file nor uses sphinx. A way to include testing of `testcode` and `testoutput` blocks is with the ``*pytest-sphinx*` <https://github.com/thisch/pytest-sphinx>`_ plugin. Once installed,

```
pip install pytest-sphinx
```

it will show up as a plugin, just like `pytest-coverage` and others, simply calling

```
pytest --doctest-glob='*.rst'
```

The `pytest-sphinx` plugin does not support `testsetup` directives.

In order to skip a test, if this is problematic, one can use the `SKIP` and `IGNORE` keywords, adding them as comments next to the relevant line or block:

```
>>> something_that_raises()  # doctest: +IGNORE
```

One can also use various `doctest` features by configuring them in the `docs/pytest.ini` file.

5.2.4 Additional information

Here are some notes on how to build docs.

Here is a cheat sheet for restructed text formatting, e.g. syntax for links etc.

5.3 Releasing a new version of Mitiq

Note: These instructions are aimed at the mantainers of the `mitiq` library.

When the time is ready for a new release, follow the checklist and instructions of this document to go through all the steps below:

- *Work in a siloed environment*

5.3.1 Work in a siloed environment

It is recommended that the release is performed in a new, clean virtual environment, which makes it easier to verify that everything is working as intended.

```
$ conda create -n mitiqenv
$ conda activate mitiqenv
```

5.3.2 Update the changelog

This task has two parts. One, make sure that `CHANGELOG.md` has an entry for each pull request (PR) since the last release (PRs). These entries should contain a short description of the PR, as well as the author username and PR number in the form (@username, gh-xxx). Two, the release author should add a "Summary" section with a couple sentences describing the latest release, and then update the title of the release section to include the release date and remove the "In Development" designation.

5.3.3 Bump version in VERSION.txt

When releasing a new version, one must of course update the `VERSION.txt` file which is the single source of truth for version information. We try to follow SemVer, so typically a release will involve changing the version `vX.Y.Z` to `vX.(Y+1).Z`, constituting a MINOR version increase.

5.3.4 Generate the HTML and PDF file for the docs

To create the HTML documentation, run the following from the top-level directory of the repository:

```
$ make docs
```

To create the PDF documentation, do the following:

```
$ make -C latexpdf
```

Finally, Since the `docs/build` folder is not version controlled, copy the newly created PDF file from `docs/build/latex` to `docs/pdf` folder as `mitiq.pdf` to overwrite the previous version.

5.3.5 Create a PR with the above changes

After the required changes to `VERSION.txt` and `CHANGELOG.md` have been made, and the PDF documentation has been generated and moved to the correct location, it is recommended that the release author make a PR to master with these changes (rather than pushing directly to master) just in case. After this PR has been merged, the release author can go to the next step.

5.3.6 Create a new tag

Tag the new commit to master (using `git tag`) with a tag that matches the number `VERSION.txt` (with a preceding "v", so `0.1.0` is `v0.1.0`) and push this tag to the Github repository.

5.3.7 Create a source & built distribution locally

From the top-level directory of the repository, run:

```
$ python setup.py sdist bdist_wheel
```

This will create a "source" distribution and a "built" distribution using `wheel`. This should create a `build/` and `sdist/` folder.

NOTE: You will need to have installed `wheel` to create the "built" distribution.

5.3.8 Release the new version on Github

Note: You need to have write access to the `mitiq` Github repository to make a new release.

Make a new release on Github here.

- Choose the tag you recently created, and add information on the release by pulling from `CHANGELOG.md` as in previous releases.
- Github will create compressed files with the repository. Upload the `mitiq.pdf` file and add the locally generated distribution tarball and wheel.

5.3.9 Release and test the new version on TestPyPI

Before uploading the package on PyPI, since that action cannot be undone, it is good practice to upload it on the test channel TestPyPI.

Note: You need to be a registered user on TestPyPI and a maintainer of the `mitiq` project in order to be able to upload the package.

Upload the package. In order to upload it, you need to have `twine`, which can be installed with `pip install twine`. Go to the `mitiq` directory, after having created the source distribution version `sdist`, and simply run:

```
$ twine upload --repository testpypi dist/*
```

You can then check at here that the library has been correctly uploaded.

In order to check that the distribution runs correctly, set up a new virtual environment and try to install the library. For example, for version v0.1.0 this is done via:

```
$ pip install -i https://test.pypi.org/simple/ --extra-index-url https://pypi.python.
↪org/simple/ mitiq==0.1.0
```

The --extra-index-url is necessary since otherwise TestPyPI would be looking for the required dependencies therein, but we want it to install them from the real PyPI channel.

5.3.10 Release the new version on PyPI

Note: You need to be a registered user on PyPI and a maintainer of the mitiq project in order to be able to upload the package.

If you already created the source distribution and wheels and tested it on TestPyPI, then you need to just run the following from the top-level directory of the mitiq repository:

```
$ twine upload dist/*
```

You will be prompted to insert your login credentials (username and password). You can then verify the upload here.

5.3.11 Update the changelog for development

Add a new section to the CHANGELOG.md to track changes in the following release, meaning that if vX.Y.Z was just released, then there should be a section for vX.(Y+1).Z that is marked "In Development".

5.3.12 Release the new docs on Read the Docs

Nothing has to be done here -- if all the above steps have been completed, ReadTheDocs will automatically build new latest and stable versions of the documentation.

5.4 Contributor Covenant Code of Conduct

5.4.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

5.4.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

5.4.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

5.4.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

5.4.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at info@unitary.fund. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

5.4.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq

CHAPTER 6

Changelog

6.1 Version 0.5.0 (In Development)

6.1.1 All Changes

6.2 Version 0.4.0 (December 6th, 2020)

6.2.1 Summary

This release adds new getter methods for fit errors, extrapolation curves, etc. in ZNE factory objects as well as custom types for noisy operations, noisy bases, and decompositions in PEC. It also includes small updates and fixes to the documentation, seeding options for PEC sampling functions, and bug fixes for a few non-deterministic test failures.

6.2.2 All Changes

- Add reference to review paper in docs (@willzeng, gh-423).
- Add unitary folding API to RTD (@rmlarose, gh-429).
- Add theory subsection on PEC in docs (@elmandouh, gh-428).
- Fix small typo in documentation function name (@nathanshammah, gh-435).
- Seed Qiskit simulator to fix non-deterministic test failure (@rmlarose, gh-425).
- Fix formatting typo and include hyperlinks to documentation objects (@nathanshammah, gh-438).
- Remove error in docs testing without tensorflow (@nathanshammah, gh-439).
- Add seed to PEC functions (@rmlarose, gh-432).
- Consolidate functions to generate randomized benchmarking circuits in different platforms, and clean up pyquil utils (@rmlarose, gh-426).

- Add new get methods (for fit errors, extrapolation curve, etc.) to Factory objects (@crazy4pi314, @andreamari, gh-403).
- Update notebook version in requirements to resolve vulnerability found by security bot.(@nathanshammah, gh-445).
- Add brief description of noise and error mitigtation to readme (@rmlarose, gh-422).
- Fix broken links in documentation (@purva-thakre, gh-448).
- Link to stable RTD instead of latest RTD in readme (@rmlarose, gh-449).
- Add option to automatically deduce the number of samples in PEC (@andreamari, gh-451).
- Fix PEC sampling bug (@rmlarose, gh-453).
- Add types for PEC (@rmlarose, gh-408).
- Add warning for large samples in PEC (@sid1993, gh-459).
- Seed a PEC test to avoid non-deterministic failure (@andreamari, gh-460).
- Update contributing docs (@purva-thakre, gh-465).

6.3 Version 0.3.0 (October 30th, 2020)

6.3.1 Summary

Factories now support "batched" executors, meaning that when a backend allows for the batch execution of a collection of quantum circuits, factories can now leverage that functionality. In addition, the main focus of this release was implementing probabilistic error cancellation (PEC), which was introduced in Temme2017 as a method for quantum error mitigation. We completed a first draft of the major components in the PEC workflow, and in the next release plan to demonstrate the full end-to-end operation of the new technique.

6.3.2 All Changes

- Fix broken links on the website (@erkska, gh-400).
- Use cirq v0.9.0 instead of cirq-unstable (@karalekas, gh-402).
- Update mitiq.about() (@rmlarose, gh-399).
- Refresh the release process documentation (@karalekas, gh-392).
- Redesign factories, batch runs in BatchedFactory, fix Qiskit utils tests (@rmlarose, @andreamari, gh-381).
- Add note on batched executors to docs (@rmlarose, gh-405).
- Added Tensorflow Quantum executor to docs (@k-m-schultz, gh-348).
- Fix a collection of small build & docs issues (@karalekas, gh-410).
- Add optimal QPR decomposition for depolarizing noise (@karalekas, gh-371).
- Add PEC basic implementation assuming a decomposition dictionary is given (@andreamari, gh-373).
- Make tensorflow requirements optional for docs (@karalekas, gh-417).

Thanks to @erkska and @k-m-schultz for their contributions to this release!

6.4 Version 0.2.0 (October 4th, 2020)

6.4.1 Announcements

The preprint for Mitiq is live on the arXiv here!

6.4.2 Summary

This release centered on source code reorganization and documentation, as well as wrapping up some holdovers from the initial public release. In addition, the team began investigating probabilistic error cancellation (PEC), which will be the main focus of the following release.

6.4.3 All Changes

- Re-organize scaling code into its own module (@rmlarose, gh-338).
- Add BibTeX snippet for arXiv citation (@karalekas, gh-351).
- Fix broken links in PR template (@rmlarose, gh-359).
- Add limitations of ZNE section to docs (@rmlarose, gh-361).
- Add static extrapolate method to all factories (@andreamari, gh-352).
- Removes barriers in conversions from a Qiskit circuit (@rmlarose, gh-362).
- Add arXiv badge to readme header (@nathanshammah, gh-376).
- Add note on shot list in factory docs (@rmlarose, gh-375).
- Spruce up the README a bit (@karalekas, gh-383).
- Make mypy checking stricter (@karalekas, gh-380).
- Add pyQuil executor examples and benchmarking circuits (@karalekas, gh-339).

6.5 Version 0.1.0 (September 2nd, 2020)

6.5.1 Summary

This marks the first public release of Mitiq on a stable version.

6.5.2 All Changes

- Add static extrapolate method to all factories (@andreamari, gh-352).
- Add the `angle` module for parameter noise scaling (@yhindy, gh-288).
- Add to the documentation instructions for maintainers to make a new release (@nathanshammah, gh-332).
- Add basic compilation facilities, don't relabel qubits (@karalekas, gh-324).
- Update readme (@rmlarose, gh-330).
- Add mypy type checking to CI, resolve existing issues (@karalekas, gh-326).

- Add readthedocs badge to readme (@nathanshammah, gh-329).
- Add change log as markdown file (@nathanshammah, gh-328).
- Add documentation on mitigating the energy landscape for QAOA MaxCut on two qubits (@rmlarose, gh-241).
- Simplify inverse gates before conversion to QASM (@andreamari, gh-283).
- Restructure library with `zne/` subpackage, modules renaming (@nathanshammah, gh-298).
- [Bug Fix] Fix minor problems in executors documentation (@andreamari, gh-292).
- Add better link to docs and more detailed features (@andreamari, gh-306).
- [Bug Fix] Fix links and author list in README (@willzeng, gh-302).
- Add new sections and more explanatory titles to the documentation's guide (@nathanshammah, gh-285).
- Store optimal parameters after calling reduce in factories (@rmlarose, gh-318).
- Run CI on all commits in PRs to master and close #316 (@karalekas, gh-325).
- Add Unitary Fund logo to the documentation html and close #297 (@nathanshammah, gh-323).
- Add circuit conversion error + tests (@rmlarose, gh-321).
- Make test file names unique (@rmlarose, gh-319).
- Update package version from v. 0.1a2, released, to 0.10dev (@nathanshammah, gh-314).

6.6 Version 0.1a2 (August 17th, 2020)

- **Initial public release**: on Github and PyPI.

6.7 Version 0.1a1 (June 5th, 2020)

- **Initial release (internal).**

CHAPTER 7

References

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Bibliography

[Temme2017] : Kristan Temme, Sergey Bravyi, Jay M. Gambetta, "Error mitigation for short-depth quantum circuits," *Phys. Rev. Lett.* **119**, 180509 (2017), (https://arxiv.org/abs/1612.02058).

[Takagi2020] : Ryuji Takagi, "Optimal resource cost for error mitigation," (https://arxiv.org/abs/2006.12509).

[Temme2017] : Kristan Temme, Sergey Bravyi, Jay M. Gambetta, "Error mitigation for short-depth quantum circuits," *Phys. Rev. Lett.* **119**, 180509 (2017), (https://arxiv.org/abs/1612.02058).

[Endo2018] : Suguru Endo, Simon C. Benjamin, Ying Li, "Practical Quantum Error Mitigation for Near-Future Applications" *Phys. Rev. **X 8**, 031027 (2018), (https://arxiv.org/abs/1712.09271).

[Takagi2020] : Ryuji Takagi, "Optimal resource cost for error mitigation," (https://arxiv.org/abs/2006.12509).

[1] Suguru Endo, Zhenyu Cai, Simon C. Benjamin, and Xiao Yuan. Hybrid quantum-classical algorithms and quantum error mitigation. (2020). arXiv:2011.01382.

[2] Kristan Temme, Sergey Bravyi, and Jay M. Gambetta. Error mitigation for short-depth quantum circuits. *Physical Review Letters*, (2017). URL: http://dx.doi.org/10.1103/PhysRevLett.119.180509, doi:10.1103/physrevlett.119.180509.

[3] Abhinav Kandala, Kristan Temme, Antonio D. Córcoles, Antonio Mezzacapo, Jerry M. Chow, and Jay M. Gambetta. Error mitigation extends the computational reach of a noisy quantum processor. *Nature*, 567(7749):491–495, (2019). URL: https://doi.org/10.1038/s41586-019-1040-7, doi:10.1038/s41586-019-1040-7.

[4] Ryuji Takagi. Optimal resource cost for error mitigation. *arXiv preprint arXiv:2006.12509*, (2020).

[5] John Preskill. Quantum computing in the NISQ era and beyond. *Quantum*, 2:79, (2018). URL: http://dx.doi.org/10.22331/q-2018-08-06-79, doi:10.22331/q-2018-08-06-79.

[6] Howard J. Carmichael. *Statistical Methods in Quantum Optics 1: Master Equations and Fokker-Planck Equations*. Springer-Verlag, (1999). ISBN 978-3-540-54882-9.

[7] H.J. Carmichael. *Statistical Methods in Quantum Optics 2: Non-Classical Fields*. Springer Berlin Heidelberg, (2007). ISBN 9783540713197.

[8] C. Gardiner and P. Zoller. *Quantum Noise: A Handbook of Markovian and Non-Markovian Quantum Stochastic Methods with Applications to Quantum Optics*. Springer, (2004). ISBN 9783540223016.

[9] H.P. Breuer and F. Petruccione. *The Theory of Open Quantum Systems*. OUP Oxford, (2007). ISBN 9780199213900.

[10] E. Knill. Quantum computing with realistically noisy devices. *Nature*, 434(7029):39–44, (2005). URL: http://dx.doi.org/10.1038/nature03350, doi:10.1038/nature03350.

[11] Constantin Brif, Raj Chakrabarti, and Herschel Rabitz. Control of quantum phenomena: past, present and future. *New Journal of Physics*, 12(7):075008, (2010). URL: http://dx.doi.org/10.1088/1367-2630/12/7/075008, doi:10.1088/1367-2630/12/7/075008.

[12] Lorenza Viola, Emanuel Knill, and Seth Lloyd. Dynamical decoupling of open quantum systems. *Physical Review Letters*, 82(12):2417–2421, (1999). URL: http://dx.doi.org/10.1103/PhysRevLett.82.2417, doi:10.1103/physrevlett.82.2417.

[13] Iulia Buluta, Sahel Ashhab, and Franco Nori. Natural and artificial atoms for quantum computation. *Reports on Progress in Physics*, 74(10):104401, (2011). URL: http://dx.doi.org/10.1088/0034-4885/74/10/104401, doi:10.1088/0034-4885/74/10/104401.

[14] Ying Li and Simon C. Benjamin. Efficient variational quantum simulator incorporating active error minimization. *Phys. Rev. X*, 7:021050, (2017). URL: https://link.aps.org/doi/10.1103/PhysRevX.7.021050, doi:10.1103/PhysRevX.7.021050.

[15] Joel J. Wallman and Joseph Emerson. Noise tailoring for scalable quantum computation via randomized compiling. *Phys. Rev. A*, 94:052325, (2016). URL: https://link.aps.org/doi/10.1103/PhysRevA.94.052325, doi:10.1103/PhysRevA.94.052325.

[16] Suguru Endo, Simon C. Benjamin, and Ying Li. Practical quantum error mitigation for near-future applications. *Phys. Rev. X*, 8:031027, (2018). URL: https://link.aps.org/doi/10.1103/PhysRevX.8.031027, doi:10.1103/PhysRevX.8.031027.

[17] Shuaining Zhang, Yao Lu, Kuan Zhang, Wentao Chen, Ying Li, Jing-Ning Zhang, and Kihwan Kim. Error-mitigated quantum gates exceeding physical fidelities in a trapped-ion system. *Nature Communications*, (2020). URL: http://dx.doi.org/10.1038/s41467-020-14376-z, doi:10.1038/s41467-020-14376-z.

[18] Jinzhao Sun, Xiao Yuan, Takahiro Tsunoda, Vlatko Vedral, Simon C. Bejamin, and Suguru Endo. Practical quantum error mitigation for analog quantum simulation. (2020). arXiv:2001.04891.

[19] Jarrod R. McClean, Mollie E. Kimchi-Schwartz, Jonathan Carter, and Wibe A. de Jong. Hybrid quantum-classical hierarchy for mitigation of decoherence and determination of excited states. *Phys. Rev. A*, 95:042308, (2017). URL: https://link.aps.org/doi/10.1103/PhysRevA.95.042308, doi:10.1103/PhysRevA.95.042308.

[20] 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). URL: https://link.aps.org/doi/10.1103/PhysRevA.98.062339, doi:10.1103/PhysRevA.98.062339.

[21] Sam McArdle, Xiao Yuan, and Simon Benjamin. Error-mitigated digital quantum simulation. *Phys. Rev. Lett.*, 122:180501, (2019). URL: https://link.aps.org/doi/10.1103/PhysRevLett.122.180501, doi:10.1103/PhysRevLett.122.180501.

[22] Jarrod R. McClean, Zhang Jiang, Nicholas C. Rubin, Ryan Babbush, and Hartmut Neven. Decoding quantum errors with subspace expansions. *Nature Communications*, (2020). URL: http://dx.doi.org/10.1038/s41467-020-14341-w, doi:10.1038/s41467-020-14341-w.

[23] 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). URL: https://link.aps.org/doi/10.1103/PhysRevA.100.010302, doi:10.1103/PhysRevA.100.010302.

[24] Carlo Cafaro and Peter van Loock. Approximate quantum error correction for generalized amplitude-damping errors. *Phys. Rev. A*, 89:022316, (2014). URL: https://link.aps.org/doi/10.1103/PhysRevA.89.022316, doi:10.1103/PhysRevA.89.022316.

[25] Robert M. Parrish, Edward G. Hohenstein, Peter L. McMahon, and Todd J. Mart\'ınez. Quantum computation of electronic transitions using a variational quantum eigensolver. *Phys. Rev. Lett.*, 122:230401, (2019). URL: https://link.aps.org/doi/10.1103/PhysRevLett.122.230401, doi:10.1103/PhysRevLett.122.230401.

[26] Mario Motta, Chong Sun, Adrian T. K. Tan, Matthew J. O'Rourke, Erika Ye, Austin J. Minnich, Fernando G. S. L. Brandão, and Garnet Kin-Lic Chan. Publisher correction: determining eigenstates and thermal states on a quantum computer using quantum imaginary time evolution. *Nature Physics*, 16(2):231–231, (2020). URL: https://doi.org/10.1038/s41567-019-0756-5, doi:10.1038/s41567-019-0756-5.

[27] Tudor Giurgica-Tiron, Yousef Hindy, Ryan LaRose, Andrea Mari, and William J. Zeng. Digital zero noise extrapolation for quantum error mitigation. (2020). arXiv:2005.10921.

Python Module Index

m

Index

U

Z