mitiq

Release 0.1.0

Tech Team @ Unitary Fund

Jun 04, 2020

Contents:

CHAPTER 1

Mitiq

A Python toolkit for implementing error mitigation on quantum computers.

1.1 Features

Mitiq performs error mitigation protocols on quantum circuits using zero-noise extrapolation.

1.2 Contents

```
mitiq/mitiq/
    | about
    | benchmarks    (package)
        |- maxcut
        |- tests    (package)
            |- test_maxcut
            |- test_random_circ
        |- random_circ
        |- utils
    | factories
    | folding
    | matrices
    | mitiq_pyquil   (package)
        |- pyquil_utils
        |- tests   (package)
            |- test_zne
    | mitiq_qiskit   (package)
        |- conversions
        |- qiskit_utils
        |- tests   (package)
            |- test_conversions
            |- test_zne
```

(continues on next page)

(continued from previous page)

```
    | tests    (package)
        |- test_factories
        |- test_folding
        |- test_matrices
        |- test_utils
        |- test_zne
    | utils
    | zne
```

1.3 Installation

To install locally use:

```
pip install -e .
```

To install for development use:

```
pip install -e .[development]
```

Note that this will install our testing environment that depends on `qiskit` and `pyquil`.

1.4 Use

A Getting Started tutorial can be found in the Documentation.

1.5 Documentation

`Mitiq` documentation is found under `mitiq/docs`. A pdf with the documentation updated to the latest release can be found here.

1.6 Development and Testing

Ensure that you have installed the development environment. Then you can run tests and build the docs with `./test_build.sh`.

1.7 Contributing

You can find information on contributing to `mitiq` code in the contributing guidelines.

To contribute to the documentation, read the instructions in the `mitiq/docs` folder.

1.8 Authors

Ryan LaRose, Andrea Mari, Nathan Shammah, and Will Zeng. An up-to-date list of authors can be found here

1.9 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.

Today's *mitiq* library is based around the zero-noise extrapolation technique. These references [1][2] 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* and *qiskit* inputs 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 work similarly as you will see in *Qiskit Mitigation*.

2.2.1 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 observable's measurements as as
        tuple (expectation value, variance).
    """
    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
```

The default implementation uses Richardson extrapolation to extrapolate the expectation value to the zero noise limit [1]. `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.factories import LinearFactory

fac = LinearFactory(scale_factors=[1.0, 2.0, 2.5])
linear = execute_with_zne(circ, noisy_simulation, fac=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*.

2.2.2 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.
```

(continues on next page)

(continued from previous page)

```
    """
    # 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
    )
    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(120):
    _ = circ.x(0)
_ = circ.measure(0, 0)

unmitigated = qs_noisy_simulation(circ)
mitigated = execute_with_zne(circ, qs_noisy_simulation)
exact = 1
# The mitigation should improve the result.
print(abs(exact - mitigated) < abs(exact - unmitigated))
```

```
True
```

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 Unitary Folding

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.

2.3.1 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.folding.fold_gates_from_left`
2. `mitiq.folding.fold_gates_from_right`
3. `mitiq.folding.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.folding 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.folding 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`.

2.3.2 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.folding 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")
```

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

```
Original circuit:
                ┌───┐
q0_0: |0>┤ H ├──■──
         └───┘┌─┴─┐
q0_1: |0>─────┤ X ├
              └───┘
```

We can now fold this circuit as follows.

> # Fold the circuit >>> folded = fold_gates_from_left(circ, scale_factor=2.) >>> # print("Folded circuit:", folded, sep="n")

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

```
Folded circuit:
                ┌───┐┌──────────┐┌────────┐┌───────────┐┌───┐
q_0: |0>  H ├ Ry(pi/4) ├ Rx(-pi) ├ Ry(-pi/4) ├ H ├───
         └───┘└──────────┘└────────┘└───────────┘└───┘
q_1: |0>──────────────────────────────────────────── 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`.

Note: Compared to the previous example which input a Cirq circuit, we see that this folded circuit has more gates. In particular, the inverse Hadamard gate is expressed differently (but equivalently) as a product of three rotations. This behavior occurs because circuits are first converted to `mitiq`'s internal representation (Cirq circuits), then folded, then converted back to the input circuit type. Because different circuits decompose gates differently, some gates (or their inverses) may be expressed differently (but equivalently) across different circuits.

2.3.3 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.folding 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`.

2.3.4 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.folding 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 Factory Objects

Factories are important elements of the `mitiq` library.

The abstract class `Factory` is a high-level representation of a generic error mitigation method. A factory is not just hardware-agnostic, it is even *quantum-agnostic*, in the sense that it only deals with classical data: the classical input and the classical output of a noisy computation.

Specific classes derived from `Factory`, like `LinearFactory`, `RichardsonFactory`, etc., represent different zero-noise extrapolation methods.

The main 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 (`self.instack`) and results (`self.outstack`), 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 `self.instack` and `self.outstack`.

The abstract class of a non-adaptive extrapolation method is `BatchedFactory`. The main feature of `BatchedFactory` is that all the noise scale factors are determined *a priori* by the initialization argument `scale_factors`. All non-adaptive methods are derived from `BatchedFactory`.

2.4.1 Example: basic usage of a factory

To make an example, let us assume that the result of our quantum computation is an expectation value which has a linear dependance on the noise. Since our aim is to understand the usage of a factory, instead of actually running quantum experiments, we simply simulate an effective classical model which returns the expectation value as a function of the noise scale factor.

```
def noise_to_expval(scale_factor: float) -> float:
    """A simple linear model for the expectation value."""
    ZERO_NOISE_LIMIT = 0.5
    NOISE_ERROR = 0.7
    return ZERO_NOISE_LIMIT + NOISE_ERROR * scale_factor
```

In this case the zero-noise limit is `0.5` and we would like to deduce it by evaluating the function only for values of `scale_factor` which are larger than or equal to 1.

Note: For implementing zero-noise extrapolation, it is not necessary to know the details of the noise model. It is also not necessary to control the absolute strength of the noise acting on the physical system. The only key assumption is that we can artificially scale the noise with respect to its normal level by a dimensionless `scale_factor`. A practical approach for scaling the noise is discussed in the *Unitary Folding* section.

In this example, we plan to measure the expectation value at 3 different noise scale factors: `SCALE_FACTORS = [1.0, 2.0, 3.0]`.

To get the zero-noise limit, we are going to use a `LinearFactory` object, run it until convergence (in this case until 3 expectation values are measured and saved) and eventually perform the zero-noise extrapolation.

```
from mitiq.factories import LinearFactory

# Some fixed noise scale factors
SCALE_FACTORS = [1.0, 2.0, 3.0]

# Instantiate a LinearFactory object
fac = LinearFactory(SCALE_FACTORS)

# Run the factory until convergence
while not fac.is_converged():
    # Get the next noise scale factor from the factory
    next_scale_factor = fac.next()
    # Evaluate the expectation value
    expval = noise_to_expval(next_scale_factor)
    # Save the noise scale factor and the result into the factory
    fac.push(next_scale_factor, expval)

# Evaluate the zero-noise extrapolation.
zn_limit = fac.reduce()
print(f"{zn_limit:.3}")
```

```
0.5
```

In the previous code block we used the main methods of a typical `Factory` object:

- **self.next** to get the next noise scale factor;
- **self.push** to save data into the factory;
- **self.is_converged** to know if enough data has been pushed;

- **self.reduce** to get the zero-noise extrapolation.

Since our idealized model `noise_to_expval` is linear and noiseless, the extrapolation will exactly match the true zero-noise limit `0.5`:

```
print(f"The zero-noise extrapolation is: {zn_limit:.3}")
```

```
The zero-noise extrapolation is: 0.5
```

Note: In a real scenario, the quantum expectation value can be determined only up to some statistical uncertainty (due to a finite number of measurement shots). This makes the zero-noise extrapolation less trivial. Moreover the expectation value could depend non-linearly on the noise level. In this case factories with higher extrapolation *order* (`PolyFactory`, `RichardsonFactory`, etc.) could be more appropriate.

The `Factory().iterate` method

Running a factory until convergence is a typical step of the zero-noise extrapolation workflow. For this reason, every factory can be run to convergence using an `iterate` method. The previous example can be simplified to the following equivalent code:

```
from mitiq.factories import LinearFactory

# Some fixed noise scale factors
SCALE_FACTORS = [1.0, 2.0, 3.0]
# Instantiate a LinearFactory object
fac = LinearFactory(SCALE_FACTORS)
# Run the factory until convergence
fac.iterate(noise_to_expval)
# Evaluate the zero-noise extrapolation.
zn_limit = fac.reduce()
print(f"The zero-noise extrapolation is: {zn_limit:.3}")
```

```
The zero-noise extrapolation is: 0.5
```

2.4.2 Built-in factories

All the built-in factories of `mitiq` can be found in the submodule `mitiq.factories`.

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

Continued on next page

Table 1 – continued from previous page

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

2.4.3 Defining a custom Factory

If necessary, the user can modify an existing extrapolation method by subclassing the corresponding factory.

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`, and `self.reduce`. Moreover `self.__init__` can also be overridden if necessary.

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

2.4.4 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.factories import BatchedFactory
import numpy as np

class MyFactory(BatchedFactory):
    """Factory object implementing a linear extrapolation taking
    into account that the expectation value must be within a given
    interval. If the zero-noise extrapolation 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:
```

(continues on next page)

(continued from previous page)

```
    """
    Fits a line to the data with a least squared method.
    Extrapolates and, if necessary, clips.

    Returns:
        The clipped extrapolation to the zero-noise limit.
    """
    # Fit a line and get the intercept
    _, intercept = np.polyfit(self.instack, self.outstack, 1)

    # Return the clipped zero-noise extrapolation.
    return np.clip(intercept, 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.

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.

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

2.5.1 What is quantum error mitigation

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.factories` 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 [1]. 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 [2]. 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.

Other error mitigation techniques

Other examples of error mitigation techniques include injecting noisy gates for randomized compiling and probabilistic error cancellation, 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 is quantum error mitigation 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 [3]. 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) [4][5][6][7], 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 [8].

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 [9].

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 [10]. 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 [4][5][6][7], 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 [11].

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 [12] and K. Temme *et al.*, *Phys. Rev. Lett.*, 2017 [1]
    - Experiment on superconducting circuit chip, A. Kandala *et al.*, *Nature*, 2019 [2]
- **On randomization methods:**
    - Randomized compiling with twirling gates, J. Wallman *et al.*, *Phys. Rev. A*, 2016 [13]
    - Porbabilistic error correction, K. Temme *et al.*, *Phys. Rev. Lett.*, 2017 [1]
    - Practical proposal, S. Endo *et al.*, *Phys. Rev. X*, 2018 [14]
    - Experiment on trapped ions, S. Zhang *et al.*, *Nature Comm.* 2020 [15]

- Experiment with gate set tomography on a supeconducting circuit device, J. Sun *et al.*, 2019 arXiv [16]

- **On subspace expansion:**
  - By hybrid quantum-classical hierarchy introduction, J. McClean *et al.*, *Phys. Rev. A*, 2017 [17]
  - By symmetry verification, X. Bonet-Monroig *et al.*, *Phys. Rev. A*, 2018 [18]
  - With a stabilizer-like method, S. McArdle *et al.*, *Phys. Rev. Lett.*, 2019, [19]
  - Exploiting molecular symmetries, J. McClean *et al.*, *Nat. Comm.*, 2020 [20]
  - Experiment on a superconducting circuit device, R. Sagastizabal *et al.*, *Phys. Rev. A*, 2019 [21]
- **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 [22]:
  - Extending the variational quantum eigensolver (VQE) to excited states, R. M. Parrish *et al.*, *Phys. Rev. Lett.*, 2017 [23]
  - Quantum imaginary time evolution, M. Motta *et al.*, *Nat. Phys.*, 2020 [24]
  - Error mitigation for analog quantum simulation, J. Sun *et al.*, 2020, arXiv [16]
- For an extensive introduction: S. Endo, *Hybrid quantum-classical algorithms and error mitigation*, PhD Thesis, 2019, Oxford University (Link).

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.
```

(continues on next page)

(continued from previous page)

```
    """
    # (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.factories.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.
↪folding.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.folding.fold_local(
        circuit, scale, method=mitiq.folding.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.factories.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).

CHAPTER 3

API-doc

This is the top level module from which functions and classes of Mitiq can be directly imported.

`mitiq.version()`
Returns the Mitiq version number.

3.1 About

Command line output of information on Mitiq and dependencies.

`mitiq.about.about()`
About box for Mitiq. Gives version numbers for Mitiq, NumPy, SciPy, Cirq, PyQuil, Qiskit.

3.2 Benchmarks

3.2.1 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]`) -- 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, int], 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]`) -- The noise scaling method for ZNE.
- **`factory`** (`Optional[Factory]`) -- The factory to use for ZNE.

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

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
```

Parameters **`verbose`** (`bool`) --

3.2.2 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=None*, *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** (`Optional[Callable[[Optional[Circuit], float], Optional[Circuit]]]`) -- 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.

3.2.3 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*`)`

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.

Return type `List[Circuit]`

Returns A list of randomized benchmarking circuits.

3.2.4 Utils

`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.

3.3 Factories

Contains all the main classes corresponding to different zero-noise extrapolation methods.

`class` `mitiq.factories.`**`AdaExpFactory`**`(`*steps*, *scale_factor=2*, *asymptote=None*, *avoid_log=False*`)`

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 (if known) of the expectation value. Default is None.

- **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.

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.

is_converged()

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

Return type bool

next()

Returns the next noise level to execute a circuit at.

Return type float

reduce()

Returns the zero-noise limit.

Return type float

class mitiq.factories.**BatchedFactory**(*scale_factors*)

Abstract class of a non-adaptive Factory.

This is initialized with a given batch of "scale_factors". The "self.next" method trivially iterates over the elements of "scale_factors" in a non-adaptive way. Convergence is achieved when all the correpsonding expectation values have been measured.

Specific (non-adaptive) zero-noise extrapolation algorithms can be derived from this class by overriding the "self.reduce" and (if necessary) the "__init__" method.

Parameters **scale_factors** (Iterable[float]) -- Iterable of noise scale factors at which expectation values should be measured.

Raises

- **ValueError** -- If the number of scale factors is less than 2.
- **IndexError** -- If an iteration step fails.

is_converged()

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

Return type bool

next()

Returns the next noise level to execute a circuit at.

Return type float

exception mitiq.factories.**ConvergenceWarning**

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

class mitiq.factories.**ExpFactory**(*scale_factors*, *asymptote=None*, *avoid_log=False*)

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 logarithmic transformation.

Parameters

- **scale_factors** (Iterable[float]) -- Iterable 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.

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.

reduce()

Returns the zero-noise limit

Return type float

exception mitiq.factories.**ExtrapolationError**

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

exception mitiq.factories.**ExtrapolationWarning**

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

class mitiq.factories.**Factory**

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, adaptive or non-adaptive, are derived from this class. A Factory object is not supposed to directly perform any quantum computation, only the classical results of quantum experiments are processed by it.

abstract is_converged()

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

Return type bool

iterate(*noise_to_expval*, *max_iterations=100*)

Runs a factory until convergence (or iterations reach "max_iterations").

Parameters

- **noise_to_expval** (Callable[[float], float]) -- Function mapping noise scale to expectation vales.
- **max_iterations** (int) -- Maximum number of iterations (optional). Default: 100.

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

Return type *Factory*

abstract next()

Returns the next noise level to execute a circuit at.

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** (`float`) --
- **outstack_val** (`float`) --

Return type `None`

abstract reduce ()

Returns the extrapolation to the zero-noise limit.

Return type `float`

reset ()

Resets the instack and outstack of the Factory to empty values.

Return type `None`

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

Runs the factory until convergence executing quantum circuits. Accepts different noise levels.

Parameters

- **qp** (`Optional[Circuit]`) -- Circuit to mitigate.
- **executor** (`Callable[[Optional[Circuit]], float]`) -- Function executing a circuit; returns an expectation value.
- **scale_noise** (`Callable[[Optional[Circuit], float], Optional[Circuit]]`) -- Function that scales the noise level of a quantum circuit.
- **max_iterations** (`int`) -- Maximum number of iterations (optional). Default: 100.

Return type *`Factory`*

class mitiq.factories.**LinearFactory** (*scale_factors*)

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

Parameters **scale_factors** (`Iterable[float]`) -- Iterable of noise scale factors at which expectation values should be measured.

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)
```

reduce ()

Determines, with a least squared method, the line of best fit associated to the data points. The intercept is returned.

Return type `float`

class `mitiq.factories.`**PolyExpFactory**(*scale_factors*, *order*, *asymptote=None*, *avoid_log=False*)

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** (`Iterable[float]`) -- Iterable 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]`) -- 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.

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.

reduce()

Returns the zero-noise limit.

Return type `float`

static static_reduce(*instack*, *outstack*, *asymptote*, *order*, *avoid_log=False*, *eps=1e-06*)

Determines the zero-noise limit, assuming an exponential ansatz: y(x) = a + sign * exp(z(x)), where z(x) is a polynomial.

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.

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.

This static method is equivalent to the "self.reduce" method of PolyExpFactory, but can be called also by other factories which are related to PolyExpFactory, e.g., ExpFactory, AdaExpFactory.

Parameters

- **instack** (`List[float]`) -- x data values.
- **outstack** (`List[float]`) -- y data values.
- **asymptote** (`Optional[float]`) -- y(x->inf).

- **order** (int) -- Extrapolation order (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 (instack - asymptote)) when the argument is to close to zero or negative.

Returns

Where "znl" is the zero-noise-limit and "params" are the optimal fitting parameters.

Return type (znl, params)

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.

class mitiq.factories.**PolyFactory**(*scale_factors*, *order*)

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

Parameters

- **scale_factors** (Iterable[float]) -- Iterable 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.

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.

reduce()

Determines with a least squared method, the polynomial of degree equal to "self.order" which optimally fits the input data. The zero-noise limit is returned.

Return type float

static static_reduce(*instack*, *outstack*, *order*)

Determines with a least squared method, the polynomial of degree equal to 'order' which optimally fits the input data. The zero-noise limit is returned.

This static method is equivalent to the "self.reduce" method of PolyFactory, but can be called also by other factories which are particular cases of PolyFactory, e.g., LinearFactory and RichardsonFactory.

Parameters

- **instack** (List[float]) -- The array of noise scale factors.
- **outstack** (List[float]) -- The array of expectation values.
- **order** (int) -- Extrapolation order (degree of the polynomial fit). It cannot exceed len(scale_factors) - 1.

Raises

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

Return type float

class mitiq.factories.**RichardsonFactory**(*scale_factors*)
Factory object implementing Richardson's extrapolation.

Parameters **scale_factors** (Iterable[float]) -- Iterable of noise scale factors at which expectation values should be measured.

Raises

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

reduce()
Returns the Richardson's extrapolation to the zero-noise limit.

Return type float

3.4 Folding

Functions for folding gates in valid mitiq circuits.

Public functions work for any circuit types supported by mitiq. Private functions work only for iternal mitiq circuit representations.

exception mitiq.folding.**UnsupportedCircuitError**

mitiq.folding.**convert_from_mitiq**(*circuit*, *conversion_type*)
Converts a mitiq circuit to a type specificed by the conversion type.

Parameters

- **circuit** (Circuit) -- Mitiq circuit to convert.
- **conversion_type** (str) -- String specifier for the converted circuit type.

Return type Optional[Circuit]

mitiq.folding.**convert_to_mitiq**(*circuit*)
Converts any valid input circuit to a mitiq circuit.

Parameters **circuit** (Optional[Circuit]) -- Any quantum circuit object supported by mitiq. See mitiq.SUPPORTED_PROGRAM_TYPES.

Raises ***UnsupportedCircuitError*** -- If the input circuit is not supported.

Returns Mitiq circuit equivalent to input circuit. input_circuit_type: Type of input circuit represented by a string.

Return type circuit

mitiq.folding.**converter**(*fold_method*)
Decorator for handling conversions.

Parameters **fold_method** (Callable) --

Return type Callable

mitiq.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

3.5 Matrices

`mitiq.matrices.npI = array([[1, 0], [0, 1]])`
Defines the identity matrix in SU(2) algebra as a (2,2) *np.array*.

`mitiq.matrices.npX = array([[0, 1], [1, 0]])`
Defines the sigma_x Pauli matrix in SU(2) algebra as a (2,2) *np.array*.

`mitiq.matrices.npY = array([[ 0.+0.j, -0.-1.j], [ 0.+1.j, 0.+0.j]])`
Defines the sigma_y Pauli matrix in SU(2) algebra as a (2,2) *np.array*.

`mitiq.matrices.npZ = array([[ 1, 0], [ 0, -1]])`
Defines the sigma_z Pauli matrix in SU(2) algebra as a (2,2) *np.array*.

3.6 Mitiq - PyQuil

3.6.1 PyQuil Utils

`mitiq.mitiq_pyquil.pyquil_utils.add_depolarizing_noise(pq, noise)`
Returns a quantum program with depolarizing channel noise.

Parameters

- **pq** (Program) -- Quantum program as Program.
- **noise** (float) -- Noise constant for depolarizing channel.

Returns Quantum program with added noise.

Return type pq

`mitiq.mitiq_pyquil.pyquil_utils.measure(circuit, qid)`
Returns a circuit adding a register for readout results.

Parameters

- **circuit** -- Quantum circuit as Program.
- **qid** -- position of the measurement in the circuit.

Returns Quantum program with added measurement.

`mitiq.mitiq_pyquil.pyquil_utils.random_identity_circuit(depth=None)`
Returns a single-qubit identity circuit based on Pauli gates.

`mitiq.mitiq_pyquil.pyquil_utils.run_program(pq, shots=500)`
Returns the expectation value of a circuit run several times.

Parameters

- **pq** (Program) -- Quantum circuit as Program.
- **shots** (int) -- (Default: 500) Number of shots the circuit is run.

Returns Expectation value.

Return type expval

`mitiq.mitiq_pyquil.pyquil_utils.`**`run_with_noise`**`(`*circuit*, *noise*, *shots*`)`
Returns the expectation value of a circuit run several times with noise.

Parameters

- **`circuit`** (`Program`) -- Quantum circuit as `Program`.
- **`noise`** (`float`) -- Noise constant for depolarizing channel.
- **`shots`** (`int`) -- Number of shots the circuit is run.

Returns Expectation value.

Return type expval

`mitiq.mitiq_pyquil.pyquil_utils.`**`scale_noise`**`(`*pq*, *param*`)`
Returns a circuit rescaled by the depolarizing noise parameter.

Parameters

- **`pq`** (`Program`) -- Quantum circuit as `Program`.
- **`param`** (`float`) -- noise scaling.

Return type `Program`

Returns Quantum program with added noise.

3.7 Mitiq - Qiskit

3.7.1 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.

3.7.2 Qiskit Utils

`mitiq.mitiq_qiskit.qiskit_utils.`**`measure`**(*circuit*, *qid*)

Apply the measure method on the first qubit of a quantum circuit given a classical register.

Parameters

- **`circuit`** -- Quantum circuit.
- **`qid`** -- classical register.

Returns circuit after the measurement.

Return type circuit

`mitiq.mitiq_qiskit.qiskit_utils.`**`random_identity_circuit`**(*num_cliffords=None*, *seed=None*)

Returns a single-qubit identity circuit.

Parameters

- **`num_cliffords`** (*`int`*) -- Number of cliffords used to generate the random circuit.
- **`seed`** (`Optional[int]`) -- Optional seed for random number generator.

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

Return type circuit

`mitiq.mitiq_qiskit.qiskit_utils.`**`run_program`**(*pq*, *shots=100*, *seed=None*)

Runs a single-qubit circuit for multiple shots and returns the expectation value of the ground state projector.

Parameters

- **`pq`** (`QuantumCircuit`) -- Quantum circuit.
- **`shots`** (`int`) -- Number of shots to run the circuit on the back-end.
- **`seed`** (`Optional[int]`) -- Optional seed for qiskit simulator.

Returns expected value.

Return type expval

`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

`mitiq.mitiq_qiskit.qiskit_utils.`**`scale_noise`**(*pq*, *param*)

Scales the noise in a quantum circuit of the factor *param*.

Parameters

- **pq** (QuantumCircuit) -- Quantum circuit.
- **noise** -- Noise constant going into *depolarizing_error*.
- **shots** -- Number of shots to run the circuit on the back-end.
- **param** (float) --

Returns quantum circuit as a qiskit.QuantumCircuit object.

Return type pq

3.8 Utils

Utility functions.

3.9 Zero Noise Extrapolation

Zero-noise extrapolation tools.

mitiq.zne.**execute_with_zne**(*qp*, *executor*, *fac=None*, *scale_noise=None*)

Takes as input a quantum circuit and returns the associated expectation value evaluated with error mitigation.

Parameters

- **qp** (Optional[Circuit]) -- Quantum circuit to execute with error mitigation.
- **executor** (Callable[[Optional[Circuit]], float]) -- Function executing a circuit and producing an expect. value (without error mitigation).
- **fac** (Optional[*Factory*]) -- Factory object determining the zero-noise extrapolation algorithm. If not specified, LinearFactory([1.0, 2.0]) will be used.
- **scale_noise** (Optional[Callable[[Optional[Circuit], float], Optional[Circuit]]]) -- Function for scaling the noise of a quantum circuit. If not specified, a default method will be used.

Return type float

mitiq.zne.**mitigate_executor**(*executor*, *fac=None*, *scale_noise=None*)

Returns an error-mitigated version of the input "executor". Takes as input a generic function ("executor"), defined by the user, that executes a circuit with an arbitrary backend and produces an expectation value.

Returns an error-mitigated version of the input "executor", having the same signature and automatically performing ZNE at each call.

Parameters

- **executor** (Callable[[Optional[Circuit]], float]) -- Function executing a circuit and returning an exp. value.
- **fac** (Optional[*Factory*]) -- Factory object determining the zero-noise extrapolation algorithm. If not specified, LinearFactory([1.0, 2.0]) is used.
- **scale_noise** (Optional[Callable[[Optional[Circuit], float], Optional[Circuit]]]) -- Function for scaling the noise of a quantum circuit. If not specified, a default method is used.

Return type Callable[[Optional[Circuit]], float] `mitiq.zne.`**`zne_decorator`**(*fac=None*, *scale_noise=None*)

Decorator which automatically adds error mitigation to any circuit-executor function defined by the user.

It is supposed to be applied to any function which executes a quantum circuit with an arbitrary backend and produces an expectation value.

Parameters

- **`fac`** (`Optional`[*`Factory`*]) -- Factory object determining the zero-noise extrapolation algorithm. If not specified, LinearFactory([1.0, 2.0]) will be used.
- **`scale_noise`** (`Optional`[`Callable`[[`Optional`[`Circuit`], `float`], `Optional`[`Circuit`]]]) -- Function for scaling the noise of a quantum circuit. If not specified, a default method will be used.

Return type `Callable`[[`Optional`[`Circuit`]], `float`]

CHAPTER 4

Contributing to mitiq

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

4.1 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.about()` to document your dependencies and working environment.

4.2 Opening a pull request

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

4.2.1 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.

4.2.2 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.

4.2.3 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
```

4.2.4 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 .[development]
```

This command will use `pip` to read the requirements contained in `requirements.txt` and `development_requirements.txt`

4.2.5 Adding tests

If you add new features to a function or class, it is strongly encouraged 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.

4.2.6 Updating the documentation

Follow the guidelines in the Contributing to docs instructions (look here on GitHub), which include guidelines about updating the API-doc list of modules and writing examples in the users guide.

4.2.7 Checking local tests

You can check that tests run with `pytest`. The `test_build.sh` file contains some bash commands to automate all tests. If you added new test packages, add them there too, so that they will be tested also in continuous integration. To test this run from root

```
(myenv) ./test_build.sh
```

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

```
./test_build.sh -docs
```

4.2.8 Code style

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

- Please get familiar with PEP 8 (code) and PEP 257 (docstrings) guidelines.
- You can use ``*black*` <https://github.com/psf/black>`_ code formatter to implement some PEP 8 and PEP 257 rules. For example, line length limit is 79 characters.
- Use annotations for type hints in the objects' signature.
- Write google-style docstrings.

4.2.9 Code of conduct

Mitiq development abides to the Contrutors' Covenant.

CHAPTER 5

Contributing to the Documentation

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

5.1 Requirements

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

```
pip -e install .[development]
```

as they are present in the `development_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`.

5.2 How to Update the Documentation

5.2.1 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.

5.2.2 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']
```

5.2.3 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
```

5.2.4 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.

5.2.5 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.

5.2.6 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.

5.2.7 Create the html

- To create the html structure,

```
make html
```

5.2.8 Create the pdf

- To create the latex files and output a pdf,

```
make latexpdf
```

5.3 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

```
./test_build.sh -docs
```

to obtain the same result, calling the `test_build.sh` file with bash script.

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.

5.4 How to Make a New Release of the Documentation

5.4.1 Work in an environment

- Create a conda environment for the documentation .. code-block:: bash

  conda create -n mitiqenv conda activate mitiqenv

5.4.2 Create a new branch

- Create a branch in `git` for the documentation with the release number up to minor (e.g., 0.0.2--->00X) .. code-block:: bash

  (mitiqenv) git checkout -b mitiq00X

5.4.3 Create the html and pdf file and save it in the `docs/pdf` folder

- To create the html structure .. code-block:: bash

  make html

  and for the pdf, .. code-block:: bash

  make latexpdf

  Since the `docs/build` folder is not kept track of, copy the pdf file with the documentation from `docs/build/latex` to the `docs/pdf` folder, naming it according to the release version with major and minor. Make a copy named `Mitiq-latest-release.pdf` in the same folder.

5.5 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.

CHAPTER 6

Change Log

6.1 Version 0.1.0 (Date)

- **Initial release.**

CHAPTER 7

References

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] Howard J. Carmichael. *Statistical Methods in Quantum Optics 1: Master Equations and Fokker-Planck Equations*. Springer-Verlag, (1999). ISBN 978-3-540-54882-9.

[5] H.J. Carmichael. *Statistical Methods in Quantum Optics 2: Non-Classical Fields*. Springer Berlin Heidelberg, (2007). ISBN 9783540713197.

[6] 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.

[7] H.P. Breuer and F. Petruccione. *The Theory of Open Quantum Systems*. OUP Oxford, (2007). ISBN 9780199213900.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

Python Module Index

m

Index

A

B

C

E

F

I

L

M

N