
mitiq

Release 0.1.0

Tech Team @ Unitary Fund

Mar 21, 2020

Contents:

1	Change Log	1
1.1	Version 0.1.0 (Date)	1
2	Users Guide	3
2.1	Overview of mitiq	3
2.2	Zero Noise Extrapolation	3
3	API-doc	5
3.1	Factories	5
3.2	Zero Noise Extrapolation	7
3.3	Folding	9
3.4	Matrices	11
3.5	Qiskit Utils	11
4	Indices and tables	13
	Python Module Index	15
	Index	17

CHAPTER 1

Change Log

1.1 Version 0.1.0 (Date)

- Initial release.

2.1 Overview of mitiq

Welcome to *mitiq* Users Guide.

Mitiq is an open source toolkit for implementing error mitigation techniques on most current intermediate-scale quantum computers.

The library allows to postprocess results from quantum circuits with both analog and digital techniques, interfacing with a variety of quantum circuit libraries.

2.2 Zero Noise Extrapolation

2.2.1 Introduction

Zero noise extrapolation (ZNE) was introduced concurrently in Ref. [1] and [2]. With *mitiq.zne* module it is possible to extrapolate what the expected value would be without noise. This is done by first setting up one of the key objects in *mitiq*, which is a `mitiq.Factory` object.

2.2.2 Importing Quantum Circuits

mitiq allows one to flexibly import and export quantum circuits from other libraries. Here is an example:

```
>>> from mitiq import Factory
```


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 Factories

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

class mitiq.factories.**BatchedFactory** (*scalars: Iterable[float]*)

Abstract class of a non-adaptive Factory.

This is initialized with a given batch of scaling factors ("scalars"). The "self.next" method trivially iterates over the elements of "scalars" in a non-adaptive way. Convergence is achieved when all the corresponding 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.

is_converged () → bool

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

next () → float

Returns the next noise level to execute a circuit at.

class mitiq.factories.**ExpFactory** (*scalars: Iterable[float], asymptote: Optional[float] = None*)

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

If the asymptotic value ($y(x \rightarrow \infty) = a$) is known, a linear fit with respect to $z(x) := \log[\text{sing}(b) (y(x) - a)]$ is used. Otherwise, a non-linear fit of $y(x)$ is performed.

reduce () → float

Returns the zero-noise limit, assuming an exponential ansatz: $y(x) = a + b * \exp(-c * x)$, with $c > 0$.

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.

is_converged () → bool

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

next () → float

Returns the next noise level to execute a circuit at.

push (instack_val: float, outstack_val: float) → None

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.

reduce () → float

Returns the extrapolation to the zero-noise limit.

class mitiq.factories.**LinearFactory** (scalars: Iterable[float])

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

reduce () → float

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

class mitiq.factories.**PolyExpFactory** (scalars: Iterable[float], order: int, asymptote: Optional[float] = None)

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

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

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

If the asymptotic value ($y(x \rightarrow \infty) = a$) is known, a linear fit with respect to $z(x) := \log[s(y(x) - a)]$ is used. Otherwise, a non-linear fit of $y(x)$ is performed.

reduce () → float

Returns the zero-noise limit, assuming an exponential ansatz: $y(x) = a + s * \exp(z(x))$, where $z(x)$ is a polynomial of a given order. The parameter "s" is a sign variable which can be either 1 or -1, corresponding to decreasing and increasing exponentials, respectively. The parameter "s" is automatically deduced from the data. It is also assumed that $z(x \rightarrow \infty) = -\infty$, such that $y(x \rightarrow \infty) \rightarrow a$.

static static_reduce (instack: List[float], outstack: List[float], asymptote: Optional[float], order: int, eps: float = 1e-09) → float

Determines the zero-noise limit, assuming an exponential ansatz: $y(x) = a + s * \exp(z(x))$, where $z(x)$ is a polynomial of a given order.

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

It is also assumed that $z(x \rightarrow \infty) = -\infty$, such that $y(x \rightarrow \infty) \rightarrow a$.

If asymptote is None, the ansatz $y(x)$ is fitted with a non-linear optimization. Otherwise, a linear fit with respect to $z(x) := \log(\text{sign} * (y(x) - \text{asymptote}))$ is performed.

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

Parameters

- **instack** -- x data values.
- **outstack** -- y data values.
- **asymptote** -- $y(x \rightarrow \infty)$.
- **order** -- extrapolation order.
- **eps** -- epsilon to regularize $\log(\text{sign}(\text{instack} - \text{asymptote}))$ when the argument is too close to zero or negative.

class mitiq.factories.**PolyFactory** (*scalars: Iterable[float], order: int*)

Factory object implementing a zero-noise extrapolation algorithm based on a polynomial fit. Note: RichardsonFactory and LinearFactory are special cases of PolyFactory.

reduce () → float

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.

static static_reduce (*instack: List[float], outstack: List[float], order: int*) → float

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.

class mitiq.factories.**RichardsonFactory** (*scalars: Iterable[float]*)

Factory object implementing Richardson's extrapolation.

reduce () → float

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

3.2 Zero Noise Extrapolation

Zero-noise extrapolation tools.

```
mitiq.zne.execute_with_zne (qp: Union[qiskit.circuit.quantumcircuit.QuantumCircuit,
                                     pyquil.qil.Program], executor: Callable[[Union[qiskit.circuit.quantumcircuit.QuantumCircuit,
                                     pyquil.qil.Program]], float], fac: mitiq.factories.Factory = None, scale_noise:
                                     Callable[[Union[qiskit.circuit.quantumcircuit.QuantumCircuit,
                                     pyquil.qil.Program]], float], Union[qiskit.circuit.quantumcircuit.QuantumCircuit,
                                     pyquil.qil.Program]] = None) →
                                     Callable[[Union[qiskit.circuit.quantumcircuit.QuantumCircuit,
                                     pyquil.qil.Program]], float]
```

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

Parameters

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

```
mitiq.zne.mitigate_executor (executor: Callable[[Union[qiskit.circuit.quantumcircuit.QuantumCircuit,
pyquil.quil.Program]], float], fac: mitiq.factories.Factory = None, scale_noise:
Callable[[Union[qiskit.circuit.quantumcircuit.QuantumCircuit,
pyquil.quil.Program], float], Union[qiskit.circuit.quantumcircuit.QuantumCircuit,
pyquil.quil.Program]] = None) →
Callable[[Union[qiskit.circuit.quantumcircuit.QuantumCircuit,
pyquil.quil.Program]], float]
```

Takes as input a generic function ("executor"), defined by the user, which 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 zero-noise extrapolation at each call.

Parameters

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

```
mitiq.zne.qrun_factory (fac: mitiq.factories.Factory, qp: Union[qiskit.circuit.quantumcircuit.QuantumCircuit,
pyquil.quil.Program], executor: Callable[[Union[qiskit.circuit.quantumcircuit.QuantumCircuit,
pyquil.quil.Program]], float], scale_noise:
Callable[[Union[qiskit.circuit.quantumcircuit.QuantumCircuit,
pyquil.quil.Program], float], Union[qiskit.circuit.quantumcircuit.QuantumCircuit,
pyquil.quil.Program]]) → None
```

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

Parameters

- **fac** -- Factory object to run until convergence.
- **qp** -- Circuit to mitigate.
- **executor** -- Function which executes a circuit and returns an expectation value.
- **scale_noise** -- Function which scales the noise level of a quantum circuit.

```
mitiq.zne.run_factory (fac: mitiq.factories.Factory, noise_to_expval: Callable[[float], float],
max_iterations: int = 100) → None
```

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

Parameters

- **fac** -- Instance of Factory object to be run.
- **noise_to_expval** -- Function mapping noise scale values to expectation vales.
- **max_iterations** -- Maximum number of iterations (optional). Default value is 100.

```
mitiq.zne.zne_decorator (fac: mitiq.factories.Factory = None, scale_noise:
Callable[[Union[qiskit.circuit.quantumcircuit.QuantumCircuit,
pyquil.quil.Program], float], Union[qiskit.circuit.quantumcircuit.QuantumCircuit,
pyquil.quil.Program]] = None) → Callable[[Union[qiskit.circuit.quantumcircuit.QuantumCircuit,
pyquil.quil.Program]], float]
```

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

3.3 Folding

Functions to fold gates in Cirq circuits.

```
mitiq.folding_cirq.fold_gates(circuit:      cirq.circuits.circuit.Circuit,      moment_indices:
                               Iterable[int],      gate_indices:      List[Iterable[int]]) →
                               cirq.circuits.circuit.Circuit
```

Returns a new circuit with specified gates folded.

Parameters

- **circuit** -- Circuit to fold.
- **moment_indices** -- Indices of moments with gates to be folded.
- **gate_indices** -- Specifies which gates within each moment to fold.

Examples

(1) Folds the first three gates in moment two. >>> fold_gates(circuit, moment_indices=[1], gate_indices=[(0, 1, 2)])

(2) Folds gates with indices 1, 4, and 5 in moment 0, and gates with indices 0, 1, and 2 in moment 1.

```
>>> fold_gates(circuit, moment_indices=[0, 3], gate_indices=[(1, 4, 5), (0, 1, 2)])
```

```
mitiq.folding_cirq.fold_gates_at_random(circuit:      cirq.circuits.circuit.Circuit,      stretch:
                                         float,      seed:      Optional[int] = None) →
                                         cirq.circuits.circuit.Circuit
```

Returns a folded circuit by applying the map $G \rightarrow G G^{\dagger} G$ to a random subset of gates in the input circuit.

The folded circuit has a number of gates approximately equal to $\text{stretch} * n$ where n is the number of gates in the input circuit.

Parameters

- **circuit** -- Circuit to fold.
- **stretch** -- Factor to stretch the circuit by. Any real number in the interval [1, 3].
- **seed** -- [Optional] Integer seed for random number generator.

Note: Folding a single gate adds two gates to the circuit, hence the maximum stretch factor is 3.

```
mitiq.folding_cirq.fold_gates_from_left(circuit: cirq.circuits.circuit.Circuit, stretch: float)
                                         → cirq.circuits.circuit.Circuit
```

Returns a new folded circuit by applying the map $G \rightarrow G G^{\dagger} G$ to a subset of gates of the input circuit, starting with gates at the left (beginning) of the circuit.

The folded circuit has a number of gates approximately equal to $\text{stretch} * n$ where n is the number of gates in the input circuit.

Parameters

- **circuit** -- Circuit to fold.
- **stretch** -- Factor to stretch the circuit by. Any real number in the interval $[1, 3]$.

Note: Folding a single gate adds two gates to the circuit, hence the maximum stretch factor is 3.

```
mitiq.folding_cirq.fold_gates_from_right (circuit:      cirq.circuits.circuit.Circuit,  stretch:
                                         float) → cirq.circuits.circuit.Circuit
```

Returns a new folded circuit by applying the map $G \rightarrow G G^{\dagger} G$ to a subset of gates of the input circuit, starting with gates at the right (end) of the circuit.

The folded circuit has a number of gates approximately equal to $\text{stretch} * n$ where n is the number of gates in the input circuit.

Parameters

- **circuit** -- Circuit to fold.
- **stretch** -- Factor to stretch the circuit by. Any real number in the interval $[1, 3]$.

Note: Folding a single gate adds two gates to the circuit, hence the maximum stretch factor is 3.

```
mitiq.folding_cirq.fold_local (circuit:      cirq.circuits.circuit.Circuit,  stretch:      float,
                              fold_method:  Callable[[cirq.circuits.circuit.Circuit, float,
                              Tuple[Any]],      cirq.circuits.circuit.Circuit]  = <function
                              fold_gates_from_left>, fold_method_args:  Tuple[Any] = ())
                              → cirq.circuits.circuit.Circuit
```

Returns a folded circuit by folding gates according to the input fold method.

Parameters

- **circuit** -- Circuit to fold.
- **stretch** -- Factor to stretch the circuit by.
- **fold_method** -- Function which defines the method for folding gates. (e.g., Randomly selects gates to fold, folds gates starting from left of circuit, etc.)

Must have signature

```
def fold_method(circuit: Circuit, stretch: float, **kwargs): ...
```

and return a circuit.

- **fold_method_args** --

Any additional input arguments for the fold_method. The method is called with `fold_method(circuit, stretch, *fold_method_args)`.

Example

```
fold_method = fold_gates_at_random fold_method_args = (1,)
```

> Uses a seed of one for the fold_gates_at_random method.

```
mitiq.folding_cirq.fold_moments (circuit:      cirq.circuits.circuit.Circuit,      moment_indices:
                                List[int]) → cirq.circuits.circuit.Circuit
```

Returns a new circuit with moments folded by mapping

$M_i \rightarrow M_i M_i^{\dagger} M_i$

where M_i is a moment specified by an integer in moment_indices.

Parameters

- **circuit** -- Circuit to apply folding operation to.
- **moment_indices** -- List of integers that specify moments to fold.

```
mitiq.folding_cirq.unitary_folding (circuit:  cirq.circuits.circuit.Circuit, stretch: float) →
                                   cirq.circuits.circuit.Circuit
```

Applies global unitary folding and a final partial folding of the input circuit. Returns a circuit of depth approximately equal to $\text{stretch} \times \text{len}(\text{circuit})$. The stretch factor can be any real number ≥ 1 .

3.4 Matrices

3.5 Qiskit Utils

CHAPTER 4

Indices and tables

- `genindex`
- `modindex`
- `search`

m

- `mitiq`, [5](#)
- `mitiq.factories`, [5](#)
- `mitiq.folding_cirq`, [9](#)
- `mitiq.matrices`, [11](#)
- `mitiq.zne`, [7](#)

B

BatchedFactory (class in mitiq.factories), 5

E

execute_with_zne() (in module mitiq.zne), 7

ExpFactory (class in mitiq.factories), 5

F

Factory (class in mitiq.factories), 5

fold_gates() (in module mitiq.folding_cirq), 9

fold_gates_at_random() (in module mitiq.folding_cirq), 9

fold_gates_from_left() (in module mitiq.folding_cirq), 9

fold_gates_from_right() (in module mitiq.folding_cirq), 10

fold_local() (in module mitiq.folding_cirq), 10

fold_moments() (in module mitiq.folding_cirq), 11

I

is_converged() (mitiq.factories.BatchedFactory method), 5

is_converged() (mitiq.factories.Factory method), 6

L

LinearFactory (class in mitiq.factories), 6

M

mitigate_executor() (in module mitiq.zne), 7

mitiq (module), 5

mitiq.factories (module), 5

mitiq.folding_cirq (module), 9

mitiq.matrices (module), 11

mitiq.zne (module), 7

N

next() (mitiq.factories.BatchedFactory method), 5

next() (mitiq.factories.Factory method), 6

P

PolyExpFactory (class in mitiq.factories), 6

PolyFactory (class in mitiq.factories), 7

push() (mitiq.factories.Factory method), 6

Q

qrun_factory() (in module mitiq.zne), 8

R

reduce() (mitiq.factories.ExpFactory method), 5

reduce() (mitiq.factories.Factory method), 6

reduce() (mitiq.factories.LinearFactory method), 6

reduce() (mitiq.factories.PolyExpFactory method), 6

reduce() (mitiq.factories.PolyFactory method), 7

reduce() (mitiq.factories.RichardsonFactory method), 7

RichardsonFactory (class in mitiq.factories), 7

run_factory() (in module mitiq.zne), 8

S

static_reduce() (mitiq.factories.PolyExpFactory static method), 6

static_reduce() (mitiq.factories.PolyFactory static method), 7

U

unitary_folding() (in module mitiq.folding_cirq), 11

V

version() (in module mitiq), 5

Z

zne_decorator() (in module mitiq.zne), 8