mitiq

Release 0.1.0

Tech Team @ Unitary Fund

Mar 22, 2020

Contents:

CHAPTER 1

Change Log

1.1 Version 0.1.0 (Date)

- **Initial release.**

CHAPTER 2

Users Guide

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
```

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 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 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.

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 algotrithm assuming an exponential ansatz y(x) = a + b * exp(-c * x), with c > 0.

If the asymptotic value (y(x->inf) = a) is known, a linear fit with respect to z(x) := log[sing(b) (y(x) - a)] is used. Otherwise, a non-linear fit of y(x) is perfomed.

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 algotrithm 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 algotrithm 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->inf) = 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 perfomed.

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-->inf)=-inf, such that y(x-->inf)-->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-->inf)=-inf, such that y(x-->inf)-->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(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 particular cases of PolyExpFactory, e.g., ExpFactory.

Parameters

- **instack** -- x data values.
- **outstack** -- y data values.
- **asymptote** -- y(x->inf).
- **order** -- extrapolation order.
- **eps** -- epsilon to regularize log(sign (instack - asymptote)) when the argument is to close to zero or negative.

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

Factory object implementing a zero-noise extrapolation algotrithm 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.quil.Program], 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 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"), difined 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 -> G G^dag G to a random subset of gates in the input circuit.

The folded circuit has a number of gates approximately equal to 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 -> G G^dag G to a subset of gates of the input circuit, starting with gates at the left (beginning) of the circuit.

The folded circuit has a number of gates approximately equal to 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 -> G G^dag G to a subset of gates of the input circuit, starting with gates at the right (end) of the circuit.

The folded circuit has a number of gates approximately equal to 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 -> M_i M_i^dag 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 stretch*len(circuit). The stretch factor can be any real number >= 1.

3.4 Matrices

3.5 Qiskit Utils

`mitiq.qiskit.qiskit_utils.`**`random_identity_circuit`**(*depth=None*)

Returns a single-qubit identity circuit based on Pauli gates.

CHAPTER 4

Mitiq Documentation

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

4.1 Requirements

The documentation is generated with Sphinx.

```
pip install -U sphinx
```

We are also including an `.md` file, so this tool is necessary

```
pip install --upgrade recommonmark
```

4.1.1 Check your Sphinx installation

To check that Sphinx is installed you can run

```
sphinx-build --version
```

4.2 How to Update the Documentation

4.2.1 Work in an environment

- Create a conda environment for the documentation

```
conda create -n mitiqenv
conda activate mitiqenv
```

4.2.2 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)

```
(mitiqenv) git checkout -b mitiq00X
```

4.2.3 The configuration file

- Since the documentation is already created, you need not to generate a configuration file from scratch. If you had to generate the documentation from scratch, the first step would have involved creating the `conf.py` file. This can be generated with a wizard from bash

```
(mitiqenv) sphinx-quickstart
```

which then asks some questions. Meta-data and specifications are accounted for in the `conf.py` file.

4.2.4 Build the documentation locally

- To build the documentation, from `bash`, move to the `docs` folder and run

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

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

4.2.5 Create the html

- To create the html structure,

```
make html
```

4.2.6 Create the pdf

- To create the latex files and output a pdf,

```
make latexpdf
```

4.2.7 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,

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

4.2.8 Update the guide with a tree of restructured text 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
```

4.2.9 If you want to include in the guide markdown files

- Information to the guide can also be added from markdown (`.md`) files. This requires `recommonmark` (`pip install --upgrade recommonmark`) and we added to the `conf.py` file

```
extensions = ['recommonmark']
```

4.2.10 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.

4.2.11 Save the pdf file in the `docs/pdf` folder

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.

4.3 Additional information

Here are some notes on how to build docs.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index