Learning Quantum Error Models



William S. Moses Costin Iancu Bert de Jong

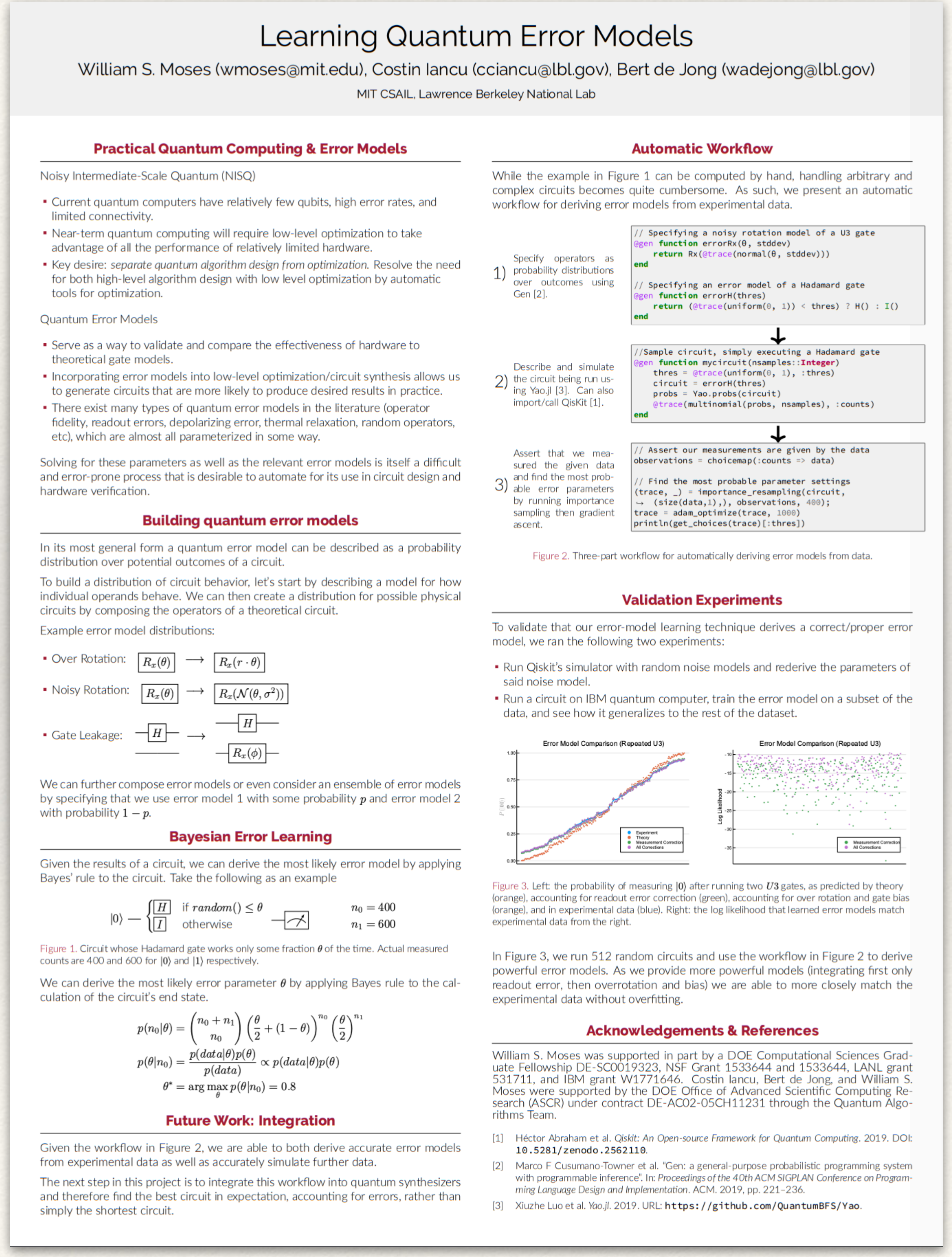
wmoses@mit.edu, {cciancu, wadejong}@lbl.gov

November 7, 2019

Learning Quantum Error Models

- Error models are crucial for characterizing hardware, synthesizing robust circuits, predicting the likelihood your computation will succeed, and much more.
- Deriving error models from circuits remains an error-prone and often manual process.
- We present a framework for automatically deriving error models from experimental data.
- Technique: Formulate error models as parameterized distributions over quantum operations, calculating the most likely model by automatic differentiation through the circuit

Come to the Poster



Learning Quantum Error Models

William S. Moses (wmoses@mit.edu), Costin Iancu (cciancu@lbl.gov), Bert de Jong (wadejong@lbl.gov)

MIT CSAIL, Lawrence Berkeley National Lab

Practical Quantum Computing & Error Models

Noisy Intermediate-Scale Quantum (NISQ)

- Current quantum computers have relatively few qubits, high error rates, and limited connectivity.
- Near-term quantum computing will require low-level optimization to take advantage of all the performance of relatively limited hardware.
- Key desire: *separate quantum algorithm design from optimization.* Resolve the need for both high-level algorithm design with low level optimization by automatic tools for optimization.

Quantum Error Models

- Serve as a way to validate and compare the effectiveness of hardware to theoretical gate models.
- Incorporating error models into low-level optimization/circuit synthesis allows us to generate circuits that are more likely to produce desired results in practice.
- There exist many types of quantum error models in the literature (operator fidelity, readout errors, depolarizing error, thermal relaxation, random operators, etc), which are almost all parameterized in some way.

Solving for these parameters as well as the relevant error models is itself a difficult and error-prone process that is desirable to automate for its use in circuit design and hardware verification.

Building quantum error models

In its most general form a quantum error model can be described as a probability distribution over potential outcomes of a circuit.

To build a distribution of circuit behavior, let's start by describing a model for how individual operands behave. We can then create a distribution for possible physical circuits by composing the operators of a theoretical circuit.

Example error model distributions:

- Over Rotation: $\boxed{R_x(\theta)} \longrightarrow \boxed{R_x(r \cdot \theta)}$
- Noisy Rotation: $\boxed{R_x(\theta)} \longrightarrow \boxed{R_x(\mathcal{N}(\theta, \sigma^2))}$
- Gate Leakage: $\boxed{H} \longrightarrow \boxed{H}, \boxed{R_x(\phi)}$

We can further compose error models or even consider an ensemble of error models by specifying that we use error model 1 with some probability p and error model 2 with probability $1-p$.

Bayesian Error Learning

Given the results of a circuit, we can derive the most likely error model by applying Bayes' rule to the circuit. Take the following as an example

$$|0\rangle - \begin{cases} \boxed{H} & \text{if } random() \leq \theta \\ \boxed{I} & \text{otherwise} \end{cases} \qquad n_0 = 400, \quad n_1 = 600$$

Figure 1. Circuit whose Hadamard gate works only some fraction θ of the time. Actual measured counts are 400 and 600 for $|0\rangle$ and $|1\rangle$ respectively.

We can derive the most likely error parameter θ by applying Bayes rule to the calculation of the circuit's end state.

$$p(n_0|\theta) = \binom{n_0+n_1}{n_0}\left(\frac{\theta}{2} + (1-\theta)\right)^{n_0}\left(\frac{\theta}{2}\right)^{n_1}$$

$$p(\theta|n_0) = \frac{p(data|\theta)p(\theta)}{p(data)} \propto p(data|\theta)p(\theta)$$

$$\theta^* = \arg\max_\theta p(\theta|n_0) = 0.8$$

Future Work: Integration

Given the workflow in Figure 2, we are able to both derive accurate error models from experimental data as well as accurately simulate further data.

The next step in this project is to integrate this workflow into quantum synthesizers and therefore find the best circuit in expectation, accounting for errors, rather than simply the shortest circuit.

Automatic Workflow

While the example in Figure 1 can be computed by hand, handling arbitrary and complex circuits becomes quite cumbersome. As such, we present an automatic workflow for deriving error models from experimental data.

1) Specify operators as probability distributions over outcomes using Gen [2].

```
// Specifying a noisy rotation model of a U3 gate
@gen function errorRx(θ, stddev)
    return Rx(@trace(normal(θ, stddev)))
end

// Specifying an error model of a Hadamard gate
@gen function errorH(thres)
    return (@trace(uniform(0, 1)) < thres) ? H() : I()
end
```

2) Describe and simulate the circuit being run using Yao.jl [3]. Can also import/call QisKit [1].

```
//Sample circuit, simply executing a Hadamard gate
@gen function mycircuit(nsamples::Integer)
    thres = @trace(uniform(0, 1), :thres)
    circuit = errorH(thres)
    probs = Yao.probs(circuit)
    @trace(multinomial(probs, nsamples), :counts)
end
```

3) Assert that we measured the given data and find the most probable error parameters by running importance sampling then gradient ascent.

```
// Assert our measurements are given by the data
observations = choicemap(:counts => data)

// Find the most probable parameter settings
(trace, _) = importance_resampling(circuit,
↪ (size(data,1),), observations, 400);
trace = adam_optimize(trace, 1000)
println(get_choices(trace)[:thres])
```

Figure 2. Three-part workflow for automatically deriving error models from data.

Validation Experiments

To validate that our error-model learning technique derives a correct/proper error model, we ran the following two experiments:

- Run Qiskit's simulator with random noise models and rederive the parameters of said noise model.
- Run a circuit on IBM quantum computer, train the error model on a subset of the data, and see how it generalizes to the rest of the dataset.

Figure 3. Left: the probability of measuring $|0\rangle$ after running two $U3$ gates, as predicted by theory (orange), accounting for readout error correction (green), accounting for over rotation and gate bias (orange), and in experimental data (blue). Right: the log likelihood that learned error models match experimental data from the right.

In Figure 3, we run 512 random circuits and use the workflow in Figure 2 to derive powerful error models. As we provide more powerful models (integrating first only readout error, then overrotation and bias) we are able to more closely match the experimental data without overfitting.

Acknowledgements & References

William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DE-SC0019323, NSF Grant 1533644 and 1533644, LANL grant 531711, and IBM grant W1771646. Costin Iancu, Bert de Jong, and William S. Moses were supported by the DOE Office of Advanced Scientific Computing Research (ASCR) under contract DE-AC02-05CH11231 through the Quantum Algorithms Team.

[1] Héctor Abraham et al. *Qiskit: An Open-source Framework for Quantum Computing*. 2019. DOI: 10.5281/zenodo.2562110.

[2] Marco F Cusumano-Towner et al. "Gen: a general-purpose probabilistic programming system with programmable inference". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 2019, pp. 221–236.

[3] Xiuzhe Luo et al. *Yao.jl*. 2019. URL: https://github.com/QuantumBFS/Yao.

Appendix

1) Specify operators as probability distributions over outcomes using Gen [2].

```
// Specifying a noisy rotation model of a U3 gate
@gen function errorRx(θ, stddev)
    return Rx(@trace(normal(θ, stddev)))
end

// Specifying an error model of a Hadamard gate
@gen function errorH(thres)
    return (@trace(uniform(0, 1)) < thres) ? H() : I()
end
```

↓

2) Describe and simulate the circuit being run using Yao.jl [3]. Can also import/call QisKit [1].

```
//Sample circuit, simply executing a Hadamard gate
@gen function mycircuit(nsamples::Integer)
    thres = @trace(uniform(0, 1), :thres)
    circuit = errorH(thres)
    probs = Yao.probs(circuit)
    @trace(multinomial(probs, nsamples), :counts)
end
```

↓

3) Assert that we measured the given data and find the most probable error parameters by running importance sampling then gradient ascent.

```
// Assert our measurements are given by the data
observations = choicemap(:counts => data)

// Find the most probable parameter settings
(trace, _) = importance_resampling(circuit,
↪  (size(data,1),), observations, 400);
trace = adam_optimize(trace, 1000)
println(get_choices(trace)[:thres])
```