SANDIA REPORT
SAND2024-06114
Printed May 2024

Sandia National Laboratories

Mid-circuit Measurement & Branching in QSCOUT: A Ping-Pong Teleportation Exemplar Program

Kenneth M. Rudinger, Antonio E. Russo, Brandon P. Ruzic, Christopher G. Yale, Susan M. Clark, Andrew J. Landahl

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

ABSTRACT

This document is intended to help users program the new mid-circuit measurement (MCM) and classical branching capabilities of the Quantum Scientific Computing Open User Testbed (QSCOUT). Here, we present and explain an exemplar "ping-pong teleportation" program that makes repeated MCM and branching calls. The program is written in Jaqal, the quantum assembly language used by QSCOUT. This document is intended to accompany a companion Jupyter notebook `Exemplar_one_bit_teleportation_pingpong.ipynb`. The Jupyter code uses the python metaprogramming utility JaqalPaq (available at `gitlab.com/jaqal`) to emulate how the exemplar program would behave on the QSCOUT platform.

Acknowledgments

This paper benefited from helpful conversations from many people. We thank (in alphabetical order) the following members of Infleqtion's Superstaq team: Pranav Gokhale, Stephanie Lee, Victory Omole, Rich Rines, and Bharath Thotakura. We also thank (also in alphabetical order) the following QSCOUT team members: Ashlyn Burch, Joshua Goldberg, Daniel Lobser, Benjamin Morrison, and Jay Van Der Wall.

This material is supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research Quantum Testbed Program.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

CONTENTS

This page intentionally left blank.

LIST OF FIGURES

This page intentionally left blank.

LIST OF TABLES

This page intentionally left blank.

1. PING-PONG TELEPORTATION

Two new features of the QSCOUT platform are the ability to perform *mid-circuit measurements* (MCMs) and *branching* operations (conditioned on the results of MCMs). Together, these features open up a host of new adaptive quantum algorithms ripe for testbed exploration. To showcase these two new capabilities, we have developed the **Ping-Pong Teleportation** exemplar. At its core, this exemplar leverages the idea of *quantum state teleportation*, depicted in Fig. 1-1.

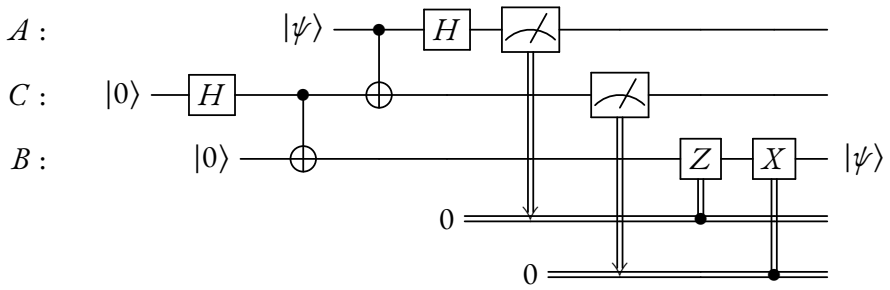

Figure 1-1.: Quantum state teleportation. Prior to Alice's arrival with her qubit state $|\psi\rangle$, Bob and Charlie prepare a shared Bell state. Alice then performs a two-bit Bell measurement of hers and Charlie's qubits. After a two-bit Pauli correction (sent from Alice to Bob), Bob's qubit now holds the state $|\psi\rangle$.

In the standard textbook form [17] of quantum state teleportation [3], three qubits are used, as depicted in Fig. 1-1. Notably, a **two**-qubit entangled state is required to move the quantum state from one physical carrier (Alice's) to another (Bob's). However, if Alice and Bob can perform a two-qubit gate directly between their qubits (such as with a controlled-NOT (*CNOT*) gate), then a **one**-qubit state (held by Bob) suffices. Zhou *et al.* call this protocol *one-bit teleportation* [25] when the two-qubit gate available is the *CNOT* gate, up to local unitary operations. (If a *SWAP* gate had been used instead, no measurement or classical communication would be required at all, obviating any notion of "teleportation," in which quantum information is transmitted over a combination of quantum and classical channels.) Many versions of this protocol can be constructed; here, we focus on what Zhou *et al.* call Z-teleportation and X-teleportation. These names underline the fact that one applies an adaptively chosen Z or X Pauli gate after the MCM to complete the process. These protocols are depicted in Fig. 1-2.

Both versions of one-qubit teleportation utilize a mid-circuit measurement and a branching correcting gate. This fact makes them excellent primitive circuits to showcase these new features of QSCOUT.

(a) Z-teleportation. **(b)** X-teleportation.

Figure 1-2.: Pauli Z and X versions of one-qubit teleportation.

After a one-qubit teleportation from Alice, Bob has Alice's (unknown) quantum state. This suggests one can implement a "ping-pong" teleportation protocol, in which Bob re-teleports the state back to Alice, and then Alice back to Bob, and so on. Importantly, it is necessary for each new recipient to have re-prepared his or her qubit into the state $|0\rangle$ (or some other known pure quantum state, which can be unitarily arrived at from $|0\rangle$) before the next step. Fortunately, the QSCOUT implementation of MCMs enforces a re-preparation of measured qubits after each MCM, in this case, each into the $|0\rangle$ state. An example of the ping-pong teleportation circuit with four Z-type teleportations is depicted in Fig. 1-3.

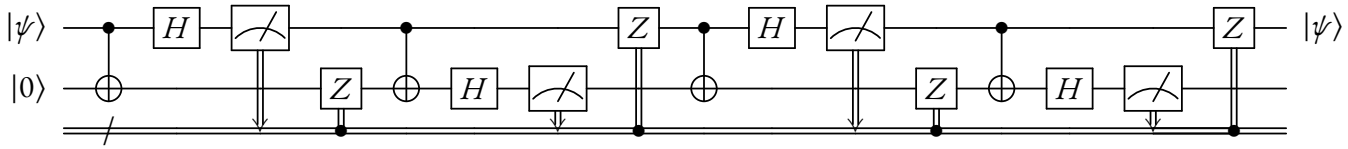

Figure 1-3.: Quantum circuit for "ping-pong" teleportation. Four rounds of teleportation (two complete trips back and forth) are depicted. Each qubit subjected to a mid-circuit measurement is immediately reset to $|0\rangle$ following the measurement.

2. PING-PONG TELEPORTATION EXEMPLAR EXPLORATIONS

We actually construct not one, but many variations of the Ping-Pong Teleportation (PPT) exemplar in the accompanying Jupyter notebook `Exemplar_one_bit_teleportation_pingpong.ipynb`. We do this to compare and contrast them, as a matter of curiosity. Our exemplars vary in the following five different dimensions, to allow the accompanying explorations:

Input state. What starting qubit state should we aim to teleport? How does PPT exemplar behavior change as a function of the input state?

Number of teleportation rounds. How many teleportation rounds should we consider? How does PPT exemplar behavior change as a function of the number of rounds of teleportation?

Teleportation basis. Which type of one-qubit teleportation should we use for each "ping-pong volley?" X-type or Z-type? How does PPT exemplar behavior change as a function of these choices?

Compilation / gate synthesis. How do we compile the PPT exemplar circuit from the standard gate basis of *CNOT*, Hadamard, *etc.* to a circuit over gates that are native to the QSCOUT platform? How does PPT exemplar behavior change as a function of the final circuit to which it is compiled?

Hardware noise model. Which noise model should we use to emulate the PPT exemplar? How does the exemplar perform relative to these different models?

To assess the PPT exemplar as a function of these variables, we need a fixed method of assessment to compare them fairly. This forces us to make two essential choices:

Success measure. How do we measure how well the PPT exemplar has succeeded, in the presence of a hardware noise model? Choices here range from full process tomography to state-dependent fidelity estimation of the output state.

Number of circuit samples. How many times should we re-run the full PPT exemplar circuit to build good statistics for estimating its performance? Each instance has many stochastic intermediate MCM outcomes, so repeated sampling will be necessary in an experimental study. In the QSCOUT emulator, however, sampling is not necessary because all conditional probabilities are tracked analytically.

In the following sections, we discuss the PPT exemplar variables we considered and our assessment methodology choices in greater detail. We start with a discussion of our assessment methodology choices, because it sets the framework for our PPT exemplar explorations.

2.1. Assessment methodologies chosen

2.1.1. Success measure

Given a fixed PPT exemplar circuit and hardware noise model, there are a variety of methods one could employ to assess the performance of that circuit. One could go so far as to perform full process tomography of all of the circuit's components, including mid-circuit measurements. Performing process tomography on mid-circuit measurements is a challenging subject and an area of active research [20, 14, 18]. Because our aim with the PPT exemplar is not to explore cutting-edge tomography methods, we choose our success measurement to be more basic. We choose it to be how well the final single-qubit state ρ of the PPT exemplar matches the single-qubit input state $|\psi\rangle$. This makes our success measure input-state-dependent, but it obviates the need for process tomography to compute it.

Even having made the choice to assess in this way, there are many choices for how to compare the agreement between the input state and the output state. Two common easy-to-compute choices are the trace distance and state fidelity [17]. We choose to use the state fidelity measure, which, for the states $|\psi\rangle$ and ρ, is defined as follows:

$$F = \langle\psi|\rho|\psi\rangle. \tag{2.1}$$

In theory, a straightforward way of estimating F is to perform a projective (von Neumann) measurement of ρ onto $|\psi\rangle\langle\psi|$ (*i.e.*, to measure ρ in the $|\psi\rangle$ basis, with outcomes $|\psi\rangle$ and $|\psi^{\perp}\rangle$). By repeating this measurement many times, over many random instances of the PPT exemplar circuit, one can construct an estimate of the average fidelity of the PPT exemplar circuit for the input state $|\psi\rangle$ by assigning the diagonal entries of ρ in the $|\psi\rangle$ basis to the frequencies observed of $|\psi\rangle$ and $|\psi^{\perp}\rangle$. In practice, however, it may or may not be easy to measure in the $|\psi\rangle$ basis. For this reason, we will restrict our attention to PPT exemplar circuit input states corresponding to bases in which we can measure easily in the QSCOUT platform. We describe this in greater detail in Sec. 2.2.1.[1]

2.1.2. Number of circuit samples

In an experimental exploration of the PPT exemplar over its varying parameters, the quantum circuit for each instance (*i.e.*, choice of state to teleport, number of rounds of teleportation, *etc.*) should be performed many times in order to accurately estimate the output state fidelity for that given instance. As discussed in Sec. 2.1.1, an estimator $\hat{F}$ for the fidelity F can be constructed from frequency counts of how often the final state matches the input state, N_{succ}. Expressed as a

[1] As a technical aside, this approach is subject to state preparation and measurement (SPAM) errors arising from imperfect implementations of these operations. There are more sophisticated tomography protocols that obviate these errors, but we do not consider them, for the sake of simplicity [15, 16].

ratio over the total number of trials N, we call this the *observed success probability* f of the protocol:

$$f = \frac{N_{succ}}{N}. \tag{2.2}$$

In order to determine how many circuit repetitions are necessary to achieve a desired confidence level for our estimate of F, we note that each trial that determines success or failure of the PPT exemplar is a Bernoulli process. In other words, it is as though each trial is equivalent to a coin flip whose (unknown) probability of coming up heads is F, and our object is to estimate this probability. By running many trials and counting the number of successes, it is further as though we are drawing N_{succ} from the binomial distribution $B(N, F)$ induced by this Bernoulli process. By standard results in probability theory, the expected value of N_{succ} is $N_{succ}F$ with variance $N_{succ}F(1-F)$.

For the binomial distribution, it is well known that using f to estimate F is quite good, as it is the maximum likelihood estimator for F that is uniformly unbiased with minimum variance [12]. That said, when estimating F with rare events (*e.g.*, if $f = 0$ or $f = 1$), this estimator will lead to the estimates that might be unrealistic (*e.g.*, $\hat{F} = 0$ or $\hat{F} = 1$). For example, if one observes $f = 1$ using experimental data, it seems unwise to conclude that the PPT circuit is absolutely perfect; it is more likely that the fidelity of the PPT circuit is just very high. If $f = 1$ is experimentally observed, then alternative estimators can be used, such as the "zero-occurrence" Bayes estimator, $\hat{F} = (N+1)/(N+2)$ (which itself may be seen as an instance of additive smoothing [6], in which a single "pseudo-count" is added to both the success and failure outcomes). For the purposes of our exemplar explorations, we stick with the standard maximum likelihood estimator, Eq. (2.2), for simplicity, understanding that our conclusions may be impaired by this choice.

Although the binomial distribution converges to the normal distribution in the limit of large N by the central limit theorem, the distribution of its mean is surprisingly non-normal even for very large values of N, in practice [4]. For this reason, a variety of methods for estimating confidence intervals exist, including the Wald method [4], the Agresti-Coull method [1], the Wilson method [24], and the Clopper-Pearson method [7]. While the Clopper-Pearson method is often favored because it is the most conservative, it is one of the more computationally demanding to calculate [22]. To avoid this, we favor the Wald method, whose simplicity leads to its recommendation by many textbooks, even though it is the most biased of the aforementioned methods [2].

In the Wald method, the estimator $\hat{F}$ for F is given by

$$f \pm z\sqrt{\frac{f(1-f)}{N}}, \tag{2.3}$$

where z is the $1 - \frac{\alpha}{2}$ quantile of a standard normal distribution for an α-sized confidence level [4]. (*E.g.*, for an $\alpha = 95\%$ confidence level, $z = 1.96$.) Emperically, the Wald method has been found to be valid when the observed events are not more rare than 2% (or more common than 98%) of the total number of trials and when the total number of trials is at least a thousand [23, 4]. Because we expect to be in this regime experimentally, the Wald method is appropriate choice for this task.

From Eq. (2.3), the confidence interval for the estimate of the PPT circuit fidelity shrinks as $\sim 1/\sqrt{N}$ for N circuit repetitions. The total number of trials one can run in a practial amount of time therefore places limits on how small one can shrink the confidence intervals for this fidelity estimate.

2.2. Exemplar variations considered

2.2.1. Input state

Given the discussion in Sec. 2.1.1, we wish to only teleport states that are both easy to prepare and easy to measure projectively. Therefore, we limit our attention to input states that coincide with the six antipodal single-qubit Pauli eigenstates. Table 2-1 enumerates these states, along with QSCOUT gates one can use to prepare/measure them by succeeding/preceding a preparation/measurement of them in the Z basis with these gates.

State	"Prepare" gate (gate after $\|0\rangle$)	"Unprepare" gate (gate before M_Z)
$\|0\rangle$	`I_Sx` (I)	`I_Sx` (I)
$\|1\rangle$	`Px` (X)	`Px` (X)
$\|+\rangle$	`Sy` (S_y)	`Syd` ($S_y^\dagger$)
$\|-\rangle$	`Syd` ($S_y^\dagger$)	`Sy` (S_y)
$\|y+\rangle$	`Sxd` ($S_x^\dagger$)	`Sx` (S_x)
$\|y-\rangle$	`Sx` (S_x)	`Sxd` ($S_x^\dagger$)

Table 2-1.: Input states considered. The six single-qubit Pauli eigenstates we consider as input states, along with the native QSCOUT gates used to prepare those states (the "Prepare gate" column) and the native QSCOUT gates used to measure final states in (the "Unprepare gate" column). An idle whose duration is that of an S_x gate is used for $|0\rangle$ (`I_Sx`), so that the overall depth of the exemplar circuit is unaffected by the input state choice.

Notice that the gates listed in the table are not the only choices possible. For example, it is well known that a Hadamard gate will prepare a $|+\rangle$ state from a $|0\rangle$ state, via the relation $|+\rangle = H|0\rangle$. However, the Hadamard gate is not a native QSCOUT gate, so we have chosen the gate S_y instead, because it is also true that $|+\rangle = S_y|0\rangle$. For a full listing of the gates that are native to the QSCOUT platform, please see the Jaqal langauge specification [11].

2.2.2. Number of teleportation rounds

In the PPT exemplar, we consider ping-pong circuits that vary from one to five rounds. Our choice for this range is informed from two directions. Firstly, while the fidelity of operations in the QSCOUT platform is high, in its current status, we expect that PPT sequences much longer than five will yield final states with very low fidelities. Secondly, our emulation of the QSCOUT platform subjected to noise for our most demanding noise models becomes slower exponentially with the number of intermediate MCMs, because we track all conditional probabilities analytically. Beyond five rounds, this tracking becomes too slow to be considered a "fast emulation," at least in qualitative terms. We expect advances in both the QSCOUT hardware and emulation software will extend the reasonable range beyond $N_{\text{rounds}} = 5$ in the future, but for the purposes of this exemplar study, this range is sufficient.

Generally speaking, we expect that the output state fidelity will decay exponentially with the number of teleportation rounds, with a scaling factor that depends on other details of the circuit (input state, compilation, *etc.*). In this way, the behavior is reminiscent of the decay rates observed in randomized benchmarking experiments [19]. However, the PPT exemplar circuit has added complexities with its internal adaptive behaviors that prevent one from inferring gate fidelities directly from its observed decay rate.[2]

2.2.3. Teleportation basis

As noted in Sec. 1, there are many versions of the one-bit teleportation circuit one can construct, including the X and Z varieties of them. For simplicity, we will explore only two scenarios: a) All teleportation rounds utilize Z-teleportation, as depicted in Fig. 1-2(a), and b) All teleportation rounds utilize X-teleportation, as depicted in Fig. 1-2(b).

2.2.4. Compilation

We consider three different compilations of the PPT exemplar, which we call *Maslov*, *by-hand*, and *Superstaq*. In the Maslov compilation, we replace each Hadamard and *CNOT* gate in the circuits in Fig. 1-2 with the circuits described by Maslov in Ref. [13], which utilize only gates that are available to the QSCOUT platform.

In the by-hand compliation, we further optimize the Maslov-compiled circuits by considering the PPT exemplar globally and applying the following transformations in an *ad hoc* way:

a) Dropping consecutive unitary gates that are inverses of each other.

b) Greedily commuting X rotations through XX (Mølmer-Sørensen) rotations to compress long sequences of single-qubit rotations into local (one-qubit) unitary gates.

c) Compiling all local (one-qubit) unitary gates into Z–X–Z Euler rotations, so that no single-qubit unitary requires more than three gates to implement.

[2] Indeed, as we will see in Sec. 4, we are not even guaranteed to observe an exponential decay in output state fidelity.

d) Dropping all local Z rotations that immediately follow preparations or immediately precede measurements in the Z basis.

In the Superstaq compilation [5, 9], we instead submit the Maslov-compiled circuits into the Superstaq compiler (version 0.3.21), which attempts to be noise- aware. Because of this, the Superstaq-compiled circuit may appear more complex than the by-hand circuit, but it may be the case that it performs better in a noisy environment.

The compiled circuits for single-round X and Z one-bit teleportation circuits are depicted in Fig. 2-1 as examples. In a multi-round PPT exemplar, the by-hand and Superstaq compilations may receive further optimizations.

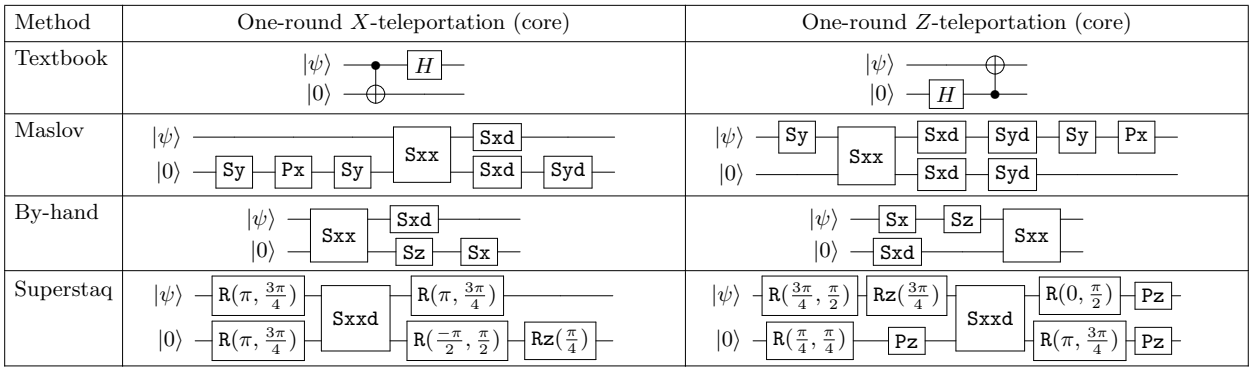

Figure 2-1.: Circuit diagrams for various compilations of the "core" portion of the X and Z teleportation circuits. The mid-circuit measurement and branching Pauli are omitted. In all instances, the state to be teleported is $|\psi\rangle$ and starts on the top qubit. The top row shows the standard "textbook" presentation of the circuits, while the subsequent three rows show the circuits compiled into native QSCOUT gates using different compilation schemes. For labeling the native QSCOUT gates, we follow the convention specified in Ref. [11]: `R`(φ, θ) denotes a counter-clockwise rotation by θ around the axis in the equatorial plane of the Bloch sphere that is an angle φ away from the x axis (measured counter-clockwise); `Sx`, `Sy`, and `Sz` denote counter-clockwise rotations by $\pi/2$ about the x, y, and z axes of the Bloch sphere; the versions of these with a trailing "`d`" denote clockwise rotations about the same axes by $\pi/2$; `Px`, `Py`, and `Pz` denote counter-clockwise rotations by π about the x, y, and z axes; `Sxx` is the Mølmer-Sørensen gate $\exp(-i\pi(X \otimes X)/4)$, and `Sxxd` is its inverse $\exp(i\pi(X \otimes X)/4)$.

2.2.5. Hardware noise models

We consider three hardware noise models, which we call *error-free*, *SNL-Toy2*, and *IonSim*. In the error-free noise model, all coherent gates, preparations, and measurements suffer no noise. This is the hardware noise model we wish we had!

The SNL-Toy2 noise model has a name indicating it is the second "toy" noise model we developed at Sandia National Laboratories for the purposes of testing concepts, and therefore does not correspond to any physically motivated noise model afflicting QSCOUT hardware. In this model,

all measurements and preparations are modeled as error-free. Furthermore, all single-qubit Z rotations are also modeled as error-free, because they occur virtually by frame updates in the QSCOUT hardware. All other single-qubit rotations about an arbitrarily chosen axis are modeled as acting ideally followed by a depolarizing channel [17] with depolarizing error probability

$$p_\theta^{(1)} = \frac{2|\theta|}{\pi} \times 10^{-3}, \tag{2.4}$$

where θ is the magnitude of the rotation, measured in radians. Furthermore, all two-qubit Mølmer-Sørensen gates $MS(\varphi, \theta) = \exp(-i\theta/2(\cos(\varphi)X + \sin(\varphi)Y)^{\otimes 2})$ are modeled as acting ideally followed by a two-qubit depolarizing channel (having 15 nontrival Pauli operations) with depolarizing error probability

$$p_\theta^{(2)} = \frac{2|\theta|}{\pi} \times 10^{-2}, \tag{2.5}$$

where again, θ is measured in radians. In all three compilations considered in Sec. 2.2.4, only the values of $\theta = \pm\pi/2$ (fully entangling gates) are used (along with $\varphi = 0$), so all two-qubit gates considered have depolarizing probability 10^{-2}.

Finally, the IonSim noise model is based on a detailed physical modeling of QSCOUT hardware, using Sandia's internal software package called `IonSim`. The details of this model are proprietary. However, to capture its predictions for the QSCOUT platform, we report the entanglement infidelity and half-diamond distance [21] that it predicts, along with what the error-free and SNL-Toy2 noise models predict, for $\pi/2$ rotations on one and two-qubits, in Table 2-2. Note that, as defined and discussed in Ref. [8] and elsewhere, the entanglement infidelity of a gate captures the gate's average error rate, while the half-diamond distance captures its worst-case error rate. (Both metrics are computed with respect to all possible input states and measurements.)[3]

[3]For a gate with purely incoherent error, its half-diamond distance and entanglement infidelity are equal, while for a gate with purely coherent errors, its half-diamond distance is quadratically larger than its entanglement infidelity [21]. Examining both quantities can indicate the relative strengths of incoherent and coherent error present in a gate.

Noise	Gate class	Entanglement Infidelity	Half-Diamond Distance
Error-free	$\mathtt{R}(\varphi, \pi/2)$	0	0
	$\mathtt{MS}(\varphi, \pm\pi/2)$	0	0
SNL-Toy2	$\mathtt{R}(\varphi, \pi/2)$	$7.5 \cdot 10^{-4}$	$7.5 \cdot 10^{-4}$
	$\mathtt{MS}(\varphi, \pm\pi/2)$	$9.375 \cdot 10^{-3}$	$9.375 \cdot 10^{-3}$
IonSim	$\mathtt{R}(\varphi, \pi/2)$	$6.54 \cdot 10^{-5}$	$8.08 \cdot 10^{-3}$
	$\mathtt{MS}(\varphi, \pm\pi/2)$	$1.82 \cdot 10^{-2}$	$9.16 \cdot 10^{-2}$

Table 2-2.: Error metrics for $\pm\pi/2$ gates from the three noise models we consider. The `MS` gate classs corresponds to what are labeled as `Sxx` and `Sxxd` in the circuit diagrams of Fig. 2-1. The `R` gates in both those diagrams and this table are the same. Note that the entanglement infidelity and half-diamond distance are identical for the SNL-Toy2 noise model, because it only contains stochastic errors. The inclusion of non-trivial coherent errors in the IonSim error model is reflected in the discrepancy between its entanglement infidelity and half-diamond distance.

3. MID-CIRCUIT MEASUREMENTS AND BRANCHING IN JAQAL AND JAQALPAQ

3.1. Realization in Jaqal

We use Jaqal (Just another quantum assembly language) [11] to specify the circuits in this document. To handle the new MCM and branching capabilities of QSCOUT, we have added two new Jaqal instructions, `measure_reprepare` amd `branch`, that are not in the current language specification, but soon will be. To mitigate this, we review these new instructions here.

Mid-circuit measurements are effected by the new Jaqal gate-like instruction `measure_reprepare`. This instruction acts on a variable-length list of arguments that correspond to qubit register addresses (or their aliases). At a low level in the QSCOUT hardware, the qubits not in those registers are shuttled away to a different location so that they are not disturbed by a subsequent destructive measurement of the indicated qubits. Following the destructive measurement, those qubits are re-prepared into the $|0\rangle$ state and the other qubits are returned to their original locations.

Branching on mid-circuit measurement outcomes is effected by the new Jaqal instruction `branch`, whose syntax is reminiscent of the `switch` statement from the classical programming language C, in which a variable is used to select different cases. For an m-bit mid-circuit measurement, the branch statement considers 2^m different cases, labeled by the strings representing them in binary. In each case, subsequent Jaqal code may be executed. Jaqal's `branch` statement lacks additional sophistications that the `switch` statement in C has, however, such as `break`, `default`, or fallthrough constructs [10].

While one could implement branching at a metaprogramming level above Jaqal, such as in JaqalPaq, we have opted to include this explicit `branch` instruction in Jaqal because we expect there will be use cases in which a faster turnaround on measured results is desired. A metaprogramming implementation would be slowed down by communication latency between the measurement outcomes and the classical computer that processes them.

To convey how these new instructions are used in practice, the code snippet below uses examples of both.

```
register q[5]
prepare_all
measure_reprepare q[3] q[1]
branch {
 '00': {
    // some gates if both qubits are measured to have outcome 0
  }
 '01': {
    // some gates if only q[3] is measured to have outcome 0
  }
 '10': {
    // some gates if only q[1] is measured to have outcome 0
  }
 '11': {
    // some gates if both qubits are measured to have outcome 1
  }
}
measure_all
```

Listing 3.1: A two-qubit mid-circuit measurement with a subsequent branch statement.

The outcome is specified with the leftmost qubit in the arguments of `measure_reprepare` corresponding to the leftmost binary digit (bit).

3.2. Realization in JaqalPaq

Using the JaqalPaq python package, Jaqal programs can be run directly on the QSCOUT hardware or emulated on a classical computer. In the accompanying Jupyter notebook, JaqalPaq is used to create and emulate Jaqal programs implementing the exemplar explorations described in Sec. 2.2. Although `measure_reprepare` and `branch` have not been added to the Jaqal language specification as of yet, the emulation of these instructions has already been added to JaqalPaq.

Before describing JaqalPaq's application programming interface (API) to the new `measure_reprepare` and `branch` instructions, let us briefly review JaqalPaq's API for interacting with the results of the execution of a Jaqal program generally. We describe the API as it exists now, but we expect that it will evolve and grow as we interact with more QSCOUT users and make changes in response to feedback from their experiences.

Consider the simple Jaqal circuit in Listing 3.2 (which does not make use of the new instructions):

```
1  from qscout.v1.std usepulses *
2  register q[2]
3  prepare_all // subcircuit 0
4  Sx q[0]
5  measure_all
6
7  prepare_all // subcircuit 1
8  Px q[0]
9  Sx q[1]
10 measure_all
```

Listing 3.2: Simple two-qubit circuit with multiple subcircuits

This is a complete Jaqal program that one might run on hardware or in emulation. We use the terms "Jaqal program" and "Jaqal circuit" interchangeably. The first line controls the QSCOUT gates that are available to the program, as well as their implementations. On hardware, this refers to the pulse and timing information, while in the emulator, this refers to the mathematical models of the gates, such as by coherent unitary matrices or incoherent quantum process matrices. For example, for an emulation using the SNLToy2 noise model described in Sec. 2.2.5, one would use `qscout.v1.std.noisy` instead. The next line uses the `register` instruction to define the number of qubits required for the Jaqal circuit, as well as how the program will refer to them (in this case, using the letter `q`). Next, two so-called "subcircuits" are defined. In a valid Jaqal program, each `prepare_all` is followed by a matching `measure_all`; those lines and the lines between them are called a "subcircuit". In this Jaqal circuit, there are two subcircuits: one that only applies a gate to qubit 0; and one that applies gates to both qubits. We will call the first one subcircuit 0, and the second one subcircuit 1.

When a Jaqal program is executed in hardware or in emulation, the system iterates over the subcircuits that are to be run (which are, by default, all subcircuits, in the order that they are expressed in the Jaqal circuit). When a subcircuit is run, the qubits are initialized by the `prepare_all` gate, all the Jaqal gates and mid-circuit measurements are run in the expressed order, and then a final measurement is performed by the `measure_all` gate. For each subcircuit, this process is repeated an integer `num_repeats` times which is by default 100, before moving on to the next subcircuit. For example, in Listing 3.2, the gates of subcircuit 0 are run 100 times, and then the gates of subcircuit 1 are run 100 times. The gross statistics of each subcircuit (and concomitant 100 repetitions) are collected and reported, and can be examined. Information about individual repetions done within a subcircuit (shot-by-shot information about, *e.g.*, each of the 100 readouts) are not available by default.

In JaqalPaq, the `run_jaqal_string` command is one of the ways a Jaqal program can be executed with fine-grained control on *how* it is executed, altering both the subcircuits to be run, their order, and the number of repetitions. The results of `run_jaqal_string`, as well as some aggregated statistical information, and possibly some information predicted by a noise model if run in an emulation mode, are stored in an `ExecutionResults` object in JaqalPaq.

The data in an `ExecutionResults` object can be accessed in multiple ways. One way is through a "circuit index view," which indexes the run groupings of subcircuits in some way. For example,

one can access the counts (or frequencies) of observed bit patterns organized by the time-ordering of the subcircuit executions that generated them.

To trace through some of the ways one might want to access information from an `ExecutionResults` object, let us walk through some short examples.

The first example, represented in Listing 3.3, executes the Jaqal program in Listing 3.2:

```
1  >>> exe_res = run_jaqal_string(jaqal_text,
2  ...                            overrides=dict(__repeats__=100))
3  >>> civ = exe_res.by_time[0]
4  >>> civ.normalized_counts
5  Tree{00: 0.53, 10: 0.47}
6  >>> ci.normalized_counts.by_int_dense
7  array([0.53, 0.47, 0.  , 0.  ])
8  >>> civ.normalized_counts['00']
9  0.53
10 >>> civ.normalized_counts['01']
11 0.0
12 >>> civ = exe_res.by_time[1]
13 >>> civ.normalized_counts
14 Tree{10: 0.61, 11: 0.39}
15 >>> civ.simulated_probabilities
16 Tree{10: 0.5000000000000001, 11: 0.4999999999999999}
```

Listing 3.3: A simple JaqalPaq processing of execution results from Listing 3.2.

In the first instruction on line 1, `jaqal_text` is a text string representing the Jaqal program in Listing 3.2. The call to `run_jaqal_string` causes the aforementioned Jaqal program to be executed, either on hardware or in emulation. The `overrides` keyword argument takes a dictionary object whose keys are `let` parameters, and the values specify what those `let` parameters are overridden to be (instead of what is specified in the Jaqal code). However, if a keyword begins and ends with two underscores, it is called a double-underscore or "dunder" keyword, and is handled directly by the hardware (or software simulator).[1] The only two keywords whose behavior we specify are those of `__repeats__` (which specifies the value of `num_repeats`) and `__index__` (see the discussion of Listing 3.4); all others should be assumed to have undefined behavior. The results are stored in `exe_res`, which is an `ExecutionResults` object.

The second instruction on line 3 extracts the outputs of all of the chronologically first set of subcircuit repetitions (indicated by the `.by_time` attribute with initial time index `[0]`). For this program, this corresponds to the repetitions of subcircuit 0. It stores the results in a `CircuitIndexView` object whose name is `civ`. Right now, it might seem that time ordering of

[1] The double-underscore naming convention, somewhat standard in python, is used to avoid clashes with other parameters in Jaqal programs that might also be overridden in the same call. For this reason, it is discouraged to have `let` parameters names that begin and end with two underscores.

the subcircuits and the ordering of the subcircuits in the Jaqal program are necessarily the same, but later we will describe an execution mode in which this is not the case.

As an example of accessing data from `civ`, we show in lines 3–9 that the `.normalized_counts` attribute stores a dictionary-like representation of the gross statistics of the measurement outcomes recorded at the `measure_all` stage of this subcircuit. This attribute doesn't utilize the time-ordering of the data in any way, and is also accessible by other kinds of circuit index views (not presented in this example). The dictionary itself has dictionary keys that are only the *observed* binary strings in the subcircuit, and for each one, the corresponding dictionary entry is the (floating-point) ratio of the number of times that outcome was observed to the total number of repetitions of that subcircuit, `num_repeats`. In this example, the second bit observed is always a `0`, because no gate is ever executed on the second qubit in subcircuit 0. In line 10, we see that, even though the outcome `01` is never observed, and is therefore not a key in the dictionary, the `CircuitIndexView` object is clever enough to report the floating point number `0.0` for its observed frequency.

In line 12, we overwrite `civ` with a similar circuit index view for the second subcircuit. In this subcircuit, the first bit is always observed to have the value 1.

In line 15, we show that, because `run_jaqal_string` was run in emulation mode (its default behavior) and not in hardware mode (not directly available to external users), the circuit index view `civ` has information above and beyond what is experimentally accessible. For example, the exact probabilities predicted by quantum mechanics for the given noise model (the default error-free unitary noise model, in this case) have been computed for the observed outcomes. The small deviation from the exact value of `0.5` is only because of numerical precision limitations in the emulator. The observed frequencies for only `num_repeats` repetitions is not `0.5` for both bit values, but if the number of repetitions was increased, both observed frequencies should converge to `0.5`.

Although this example used the unitary (noise-free) emulator, one can pass additional arguments to the `run_jaqal_string` command to utilize the SNLToy2 and IonSim noise models described in Sec. 2.2.5.

In our second example, in Listing 3.4, we show that the subcircuit executions of the same Jaqal program in Listing 3.2 can be collated as desired. In this Listing, the `__index__` override is also used, which is an array indicating the time ordering of subcircuit executions. The `[1, 1, 0]` argument indicates that the Jaqal program is to be executed so that it runs a number of repetitions of subcircuit 1, then subcircuit 1, and then subcircuit 0 again.

```
>>> exe_res = run_jaqal_string(
...  jaqal_text,
...  overrides=dict(__repeats__=100, __index__=[1,1,0]))
>>> exe_res.by_time[0].normalized_counts
Tree{10: 0.53, 11: 0.47}
>>> exe_res.by_time[1].normalized_counts
Tree{10: 0.45, 11: 0.55}
>>> exe_res.by_time[2].normalized_counts
Tree{00: 0.46, 10: 0.54}
```

Listing 3.4: A more complex, unusually collated, JaqalPaq processing of execution results from Listing 3.2.

Now let us see how `measure_reprepare` and `branch` instructions broaden the ways data can be extracted from an `ExecutionResults` object. Consider the Jaqal program in Listing 3.5:

```
register q[2]
prepare_all
Sx q[0]
measure_reprepare q[0]
branch {
 '0' : {}
 '1' : {Px q[1]}
}
measure_all
```

Listing 3.5: Simple Jaqal program containing a one-qubit mid-circuit measurement and subsequent classical feed-forward.

This Jaqal program contains only a single subcircuit, even though it utilizes the new instructions. It first measures the first qubit in the register with the `measure_reprepare` instruction, and then it measures both qubits with the `measure_all` instruction. In the absence of noise, the circuit should have the behavior that the first bit has a 50% chance of being observed to have the value `0`. Because of the repreparation part of the instruction, the first bit is reset to $|0\rangle$, regardless of what was observed. No matter what value this bit is observed to have, we expect the subsequent measurement of the two qubits will yield either `00` or `11`, with the first bit the same as the one-qubit MCM outcome. In Listing 3.6, we explore a way that this Jaqal program might be executed in (noise-free) emulation:

```
1  >>> exe_res = run_jaqal_string(jaqal_text)
2  >>> civ = exe_res.by_time[0]
3  >>> civ.normalized_counts
4  Tree{0: 0.56, 1: 0.44}
5  >>> civ.normalized_counts['0']
6  0.56
7  >>> civ.normalized_counts['0', :]
8  Tree{00: 0.56}
9  >>> civ.normalized_counts['1', :]
10 Tree{01: 0.44}
11 >>> civ.conditional_normalized_counts['0', :]
12 Tree{00: 1.0}
13 >>> civ.conditional_normalized_counts['1', :]
14 Tree{01: 1.0}
15 >>> civ.conditional_normalized_counts['1', '01']
16 1.0
```

Listing 3.6: A simple JaqalPaq processing of execution results from Listing 3.5.

In line 3, we see that the first bit is has an observed frequency of 0.56, which is reasonably close to the theoretical value of 0.5 when an infinite number of repetitions is used. To access the *conditional* probability of the values that the second qubit measurement will have, conditioned on the first qubit having the observed value `0`, our API uses a Python slicing-like construction in line 5 with the instruction `civ.normalized_counts['0', :]`. In line 7, we see that the `normalized_counts` attribute, as before, is a dictionary-like object.

As before, the keys of this dictionary are the measurement outcomes observed in the hardware (or simulation). If the measurement is a final measurement, the leftmost bit corresponds to qubit 0; if it is a mid-circuit measurement, the leftmost bit corresponds to the leftmost qubit passed to `measure_reprepare`. In this case, since 0 was measured at the mid-circuit measurement, no additional gates are performed in the `branch` statement (line 6 of Listing 3.5), and therefore we expect the final measurement to be of two qubits in the $+Z$ basis 100% of the time (in the ideal, noiseless case). Again, as before, zero-count events may not be listed as dictionary keys, as seen on line 8. Furthermore, notice that the values in this dictionary are normalized to the total number of repetitions. The sum of all values in the dictionary is equal to the parent key at the higher level, i.e., the real number `normalized_counts[s]` is always `sum(normalized_counts[s,:])`. For example, consider the situation in which the mid-circuit measurement of qubit 0 is finds it in the $-Z$ state, which occurs 44% of the time. Consequentially, a `Px` gate is applied to qubit 1 (line 7 of Listing 3.5). It follows, then, that we expect qubit 1 to be in the $-Z$ state, and qubit 0 to be in the $+Z$ state, at the final measurement. If a 1 readout occurs at the mid-circuit measurement, then, we expect to always get 01 readout at the final measurement. This is captured by `ci.normalized_counts['1',:]`, as on line 9, which gives information about the fraction of the total runs: 44% of the runs have a final readout of 01 following a mid-circuit readout of 1. If, instead, you would like to get the conditional, rather than absolute frequency, use `conditional_normalized_counts`, as on lines 9 and 10.

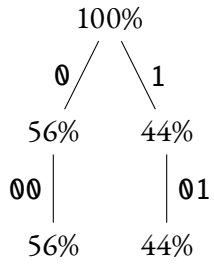

Figure 3-1.: Tree structure of an `ExecutionResults` object. Each node represents a (possibly partial) history of measurement readouts, and is labeled with the cumulative probability of finding that readout. Histories are connected by individual readouts, represented by edges and labeled by the readout. The root node has unit probability, and is connected by 0 and 1 edges, each representing one of the two possible mid-circuit measurements of qubit 0. Those nodes are then connected by edges representing the final measurement of both qubits.

4. RESULTS

In the Jupyter notebook `Exemplar_one_bit_teleportation_pingpong.ipynb`, we used JaqalPaq to emulate choices for each of the parameters listed below for the PPT exemplar:

- Pauli-basis input state eigenstate preparation (6 choices).
- Number of teleportation rounds (5 choices).
- Teleportation basis (2 choices).
- Compilation to QSCOUT gates (3 choices).
- Hardware noise model (2 nontrivial choices).

This yields 360 unique emulations (not counting the trivial error-free noise model variations). An exhaustive description of these results is presented in `Exemplar_one_bit_teleportation_pingpong.ipynb`. To get a flavor of what was obtained there, we present results for some of the combinations here.

In Fig. 4-1, we depict 60 of these emulations as a collection of plots. Each plot captures the output state fidelity as a function of the number of teleportation rounds, with plots for different compilation methods overlayed, plots for different noise models placed one above the other, and plots for different Z-basis input state preparations placed beside each other. All plots use the fixed choice of using all X-basis teleportations. The only choices not listed here are those for X and Y input state preparations, and those for Z-basis teleportations. No error bars are depicted in the plots because the emulator computes the fidelities exactly rather than by using statistical sampling, which would be needed in an actual experiment.

4.1. Reflections on results

First, we see that performance does not perfectly correlate to circuit depth. For example, the shallowest compilation (by-hand) often performs the best, but that is not the case for the `IonSim` noise model with a $|z\ 0\rangle$ input. Similarly, the deepest compilation (Maslov) is the worst-performing compilation in only one of the four cases considered (intriguingly, the $|z\ 0\rangle$ case again). Second, while the performances of the SNL-Toy2 cases for the two input preparations considered ($|z\ 0\rangle$ and $|z\ 1\rangle$) are essentially the same (and only offset quantitatively by a small amount, likely due to an additional `Px` gate at the beginning and end of the $|z\ 1\rangle$ circuit, the performances of the IonSim cases for these two input preparations differ dramatically. This is almost certainly caused by various real-world asymmetries captured by the IonSim error model (which includes a large quantity of coherent error, as indicated by the difference between infidelities and diamond distances). Nevertheless, it is still striking to see how different the predictions are when using IonSim error model to emulate the performance of two circuits that are identical except for the input state preparations.

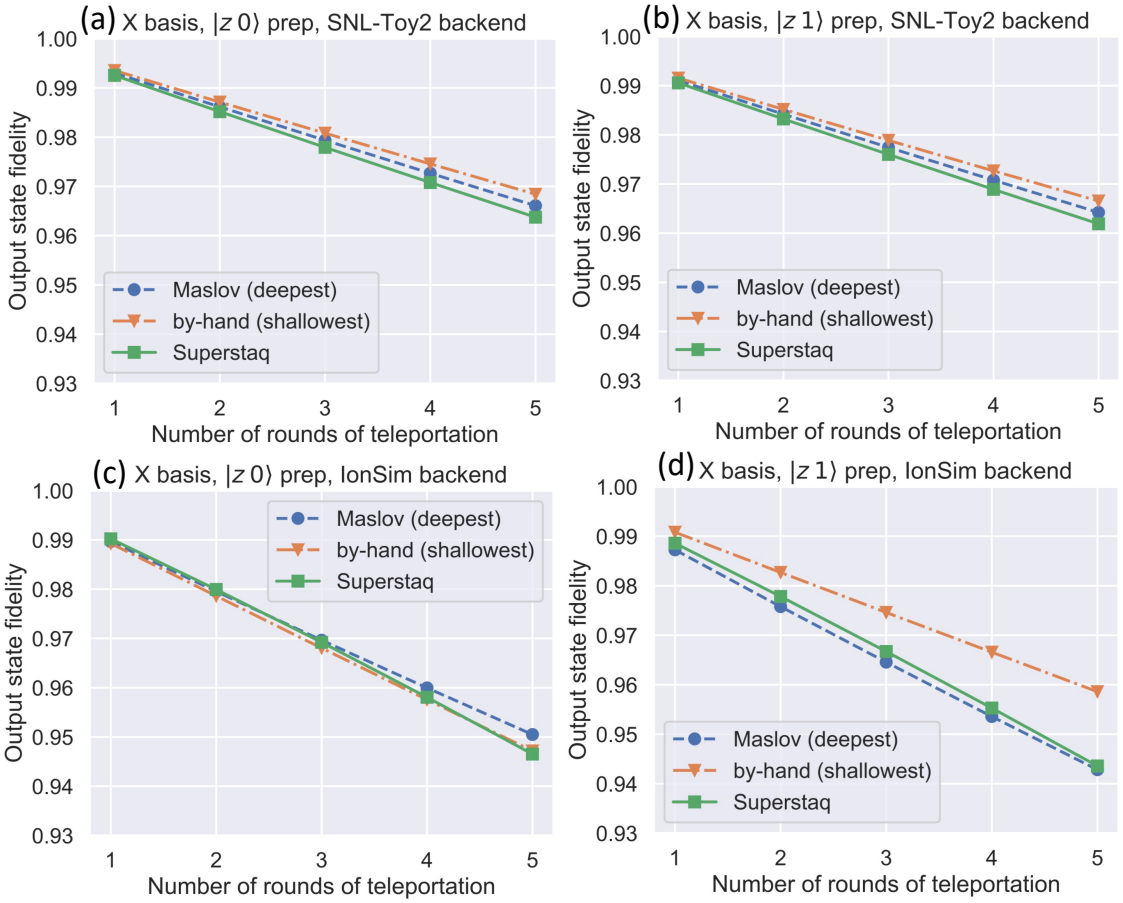

Figure 4-1.: Output state fidelity versus number of rounds of X teleportation of Z-basis input state preparations. All variations on options for input-state direction, compilation method, and hardware noise model ("backend") are depicted.

BIBLIOGRAPHY

[1] Alan Agresti and Brent A. Coull. Approximate is better than "exact" for interval estimation of binomial proportions. *The American Statistician*, 52(2):119–126, 1998.

[2] Per Gösta Andersson. The Wald confidence interval for a binomial p as an illuminating "bad" example. *The American Statistician*, 77(4):443–448, 2023.

[3] Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K. Wootters. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Phys. Rev. Lett.*, 70:1895, 1993.

[4] Lawrence D Brown, T Tony Cai, and Anirban DasGupta. Interval estimation for a binomial proportion. *Statistical science*, 16(2):101–133, 2001.

[5] Colin Campbell, Frederic T. Chong, Denny Dahl, Paige Frederick, Palash Goiporia, Pranav Gokhale, Benjamin Hall, Salahedeen Issa, Eric Jones, Stephanie Lee, Andrew Litteken, Victory Omole, David Owusu-Antwi, Michael A. Perlin, Rich Rines, Kaitlin N. Smith, Noah Goss, Akel Hashim, Ravi Naik, Ed Younis, Daniel Lobser, Christopher G. Yale, Benchen Huang, and Ji Liu. Superstaq: Deep optimization of quantum programs. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 01, pages 1020–1032, 2023.

[6] Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. In *34th Annual Meeting of the Association for Computational Linguistics*, pages 310–318, Santa Cruz, California, USA, June 1996. Association for Computational Linguistics.

[7] C. J. Clopper and E. S. Pearson. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4):404–413, 1934.

[8] Jeongwan Haah, Robin Kothari, Ryan O'Donnell, and Ewin Tang. Query-optimal estimation of unitary channels in diamond distance. *arXiv preprint*, 2023.

[9] Infleqtion. Python qiskit-superstaq package, version 0.3.21, 2023.

[10] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Pearson, 2nd edition, 1988.

[11] Andrew J. Landahl, Daniel S. Lobser, Benjamin C. A. Morrison, Kenneth M. Rudinger, Antonio E. Russo, Jay W. Van Der Wall, and Peter Maunz. Jaqal, the quantum assembly language for QSCOUT, 2020.

[12] E. L. Lehmann and Henry Scheffé. *Completeness, Similar Regions, and Unbiased Estimation-Part I*, pages 233–268. Springer US, Boston, MA, 2012.

[13] Dmitri Maslov. Basic circuit compilation techniques for an ion-trap quantum machine. *New Journal of Physics*, 19(2):023035, 2017.

[14] Seth Merkel, Petar Jurcevic, Luke Govia, and David McKay. Characterization of mid-circuit measurement on multi-qubit devices. In *APS March Meeting Abstracts*, volume 2022, pages K35–006, 2022.

[15] Seth T. Merkel, Jay M. Gambetta, John A. Smolin, Stefano Poletto, Antonio D. Córcoles, Blake R. Johnson, Colm A. Ryan, and Matthias Steffen. Self-consistent quantum process tomography. *Phys. Rev. A*, 87:062119, Jun 2013.

[16] Erik Nielsen, John King Gamble, Kenneth Rudinger, Travis Scholten, Kevin Young, and Robin Blume-Kohout. Gate set tomography. *Quantum*, 5:557, 2021.

[17] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.

[18] Luciano Pereira, Juan José García-Ripoll, and T Ramos. Parallel tomography of quantum non-demolition measurements in multi-qubit devices. *npj Quantum Information*, 9(1):22, 2023.

[19] Timothy Proctor, Kenneth Rudinger, Kevin Young, Mohan Sarovar, and Robin Blume-Kohout. What randomized benchmarking actually measures. *Phys. Rev. Lett.*, 119:130502, Sep 2017.

[20] Kenneth Rudinger, Guilhem J Ribeill, Luke CG Govia, Matthew Ware, Erik Nielsen, Kevin Young, Thomas A Ohki, Robin Blume-Kohout, and Timothy Proctor. Characterizing midcircuit measurements on a superconducting qubit using gate set tomography. *Physical Review Applied*, 17(1):014014, 2022.

[21] Yuval R Sanders, Joel J Wallman, and Barry C Sanders. Bounding quantum gate error rate based on reported average fidelity. *New Journal of Physics*, 18(1):012002, dec 2015.

[22] Måns Thulin. The cost of using exact confidence intervals for a binomial proportion. *Electronic Journal of Statistics*, 8(1):817 – 840, 2014.

[23] Sean Wallis. Binomial confidence intervals and contingency tests: mathematical fundamentals and the evaluation of alternative methods. *Journal of Quantitative Linguistics*, 20(3):178–208, 2013.

[24] Edwin B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927.

[25] Xinlan Zhou, Debbie W. Leung, and Isaac L. Chuang. Methodology for quantum logic gate construction. *Phys. Rev. A*, 62:052316, Oct. 2000.

DISTRIBUTION

Email—Internal

Name	Org.	Sandia Email Address
Susan Clark	5225	sclark@sandia.gov
Chris DeRose	5225	cderose@sandia.gov
Andrew Landahl	1425	alandahl@sandia.gov
Kenneth Rudinger	1425	kmrudin@sandia.gov
Antonio Russo	1425	arusso@sandia.gov
Brandon Ruzic	1425	bruzic@sandia.gov
Christopher Yale	5225	cgyale@sandia.gov
Technical Library	1911	sanddocs@sandia.gov

Email—External

Name	Company Email Address	Company Name
Kalyan Perumalla	Kalyan.Perumalla@science.doe.gov	Department of Energy

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.