A Software Simulator for Noisy Quantum Circuits

by

Utkarsh Azad

in

arXiv.org > quant-ph > arXiv:1908.05154 Search... Help | Advanced Search

Report No: IIIT/TR/2019/-1

Centre for Computational Natural Sciences and Bioinformatics
International Institute of Information Technology
Hyderabad - 500 032, INDIA
August 2019

A Software Simulator for Noisy Quantum Circuits

Himanshu Chaudhary,[1, *] Biplab Mahato,[1, †] Lakshya Priyadarshi,[2, ‡]
Naman Roshan,[3, §] Utkarsh,[4, ¶] and Apoorva D. Patel[5, **]

[1]*Undergraduate Physics, Indian Institute of Science, Bangalore 560012*
[2]*Undergraduate Computer Science, Institute of Engineering and Technology, Lucknow 226021*
[3]*Undergraduate Physics, Indian Institute of Technology, Kanpur 208016*
[4]*Undergraduate Computer Science, International Institute of Information Technology, Hyderabad 500032*
[5]*Centre for High Energy Physics, Indian Institute of Science, Bangalore 560012*
(Dated: August 15, 2019)

We have developed a software library that simulates noisy quantum logic circuits. We represent quantum states by their density matrices, and incorporate possible errors in initialisation, logic gates, memory and measurement using simple models. Our quantum simulator is implemented as a new backend on IBM's open-source Qiskit platform. In this document, we provide its description, and illustrate it with some simple examples.

I. MOTIVATION

The field of quantum technologies has made rapid strides in recent years, and is poised for significant breakthroughs in the coming years. Practical applications are expected to appear first in sensing and metrology, then in communications and simulations, then as feedback to foundations of quantum theory, and ultimately in computation. The essential features that contribute to these technologies are superposition, entanglement, squeezing and tunneling of quantum states. The theoretical foundation of the field is clear; laws of quantum mechanics are precisely known, and elementary hardware components work as predicted [1, 2]. The challenge is a large scale integration, say of 10 or more components.

Quantum systems are highly sensitive to disturbances from the environment; even necessary controls and observations perturb them. The available, and upcoming, quantum devices are noisy, and techniques to bring down the environmental error rate are being intensively pursued. At the same time, it is necessary to come up with error-resilient system designs, as well as techniques that validate and verify the results. This era of noisy intermediate scale quantum systems has been labeled NISQ [3]. Such systems are often special purpose platforms, with limited capabilities. They roughly span devices with 10-100 qubits, 10-1000 logic operations, limited interactions between qubits, and with no error correction since the fault-tolerance threshold is orders of magnitudes away.

Worldwide, many universities and companies have developed software quantum simulators for help in investigations of noisy quantum processors [4]. They are programs running on classical parallel computer platforms, and can model and benchmark 10-50 qubit systems. (It is not possible to classically simulate larger qubit systems due to exponential growth in the Hilbert space size.) For a user accessing a computer on the cloud and obtaining the output of a program, it makes little difference whether there is a genuine quantum processor at the other end or just a suitable software simulator. For this purpose, instead of using exact algebra, the simulator has to be designed to mimic a noisy quantum processor.

A quantum computation may suffer from many sources of error: due to imprecise initial state preparation, due to imperfect logic gate implementation, due to disturbances to the data in memory, and due to error-prone measurements. (It is safe to assume that the program instructions, which are classical, are essentially error-free.) So a realistic quantum simulator would have to include all of them with appropriate probability distributions. Additional features that can be included are restrictions on possible logic operations and connectivity between the components, which would imitate what may be the structure of a real quantum processor. With such improvisations, the simulation results would look close to what a noisy quantum processor would deliver, and one can test how well various algorithms work with imperfect quantum components. More importantly, one can vary the imperfections and the connectivity in the software simulator to figure out what design for the noisy quantum processor would produce the best results.

Quantum simulators serve an important educational purpose as well. They are portable, and can be easily distributed over existing computational facilities world wide. They are therefore an excellent way to attract students to the field of quantum technology, providing a platform to acquire the skills of 'programming' as well as 'designing' quantum processors. Programming a quantum processor is qualitatively different from the classical experience of computer programming, and so the opportunity and exposure provided by quantum simulators would be of vital importance in developing future expertise in the field. It is with such an aim that we have constructed a software library for simulating noisy quantum

*Electronic address: himanshuc@iisc.ac.in
†Electronic address: biplab@iisc.ac.in
‡Electronic address: lakshyapriyadarshi1@gmail.com
§Electronic address: namanr@iitk.ac.in
¶Electronic address: utkarsh.azad@research.iiit.ac.in
**Electronic address: adpatel@iisc.ac.in

logic circuits. We provide the details in what follows.

II. IMPLEMENTATION

In the standard formulation of quantum mechanics, states are vectors in a Hilbert space and evolve by unitary transformations, $|\psi\rangle \rightarrow U|\psi\rangle$. This evolution is deterministic, continuous and reversible. It is appropriate for describing the pure states of a closed quantum system, but is insufficient for describing the mixed states that result from interactions of an open quantum system with its environment.

The most general description of a quantum system is in terms of its density matrix ρ, which evolves according to a linear completely positive trace preserving map known as the superoperator [1, 2]. For generic mixed states, ρ is a Hermitian positive semi-definite matrix, with $Tr(\rho) = 1$ and $\rho^2 \preceq \rho$, while for pure states, $\rho = |\psi\rangle\langle\psi|$ and $\rho^2 = \rho$. The density matrix provides an ensemble description of the quantum system, and so is inherently probabilistic, in contrast to the state vector description that can describe individual experimental system evolution. Nonetheless, it allows determination of the expectation value of any physical observable, $\langle O \rangle = Tr(O\rho)$, which is defined as the average result over many experimental realisations.

In its discrete form, a superoperator can be specified by its Kraus operator-sum representation:

$$\rho \rightarrow \sum_{\mu} M_{\mu}\rho M_{\mu}^{\dagger} , \quad \sum_{\mu} M_{\mu}^{\dagger}M_{\mu} = I . \quad (1)$$

Unitary evolution, $\rho \rightarrow U\rho U^{\dagger}$, and orthogonal projective measurement, $\rho \rightarrow \sum_k P_k \rho P_k$, are special cases of this representation. (Note that the projection operators satisfy $P_k = P_k^{\dagger}$, $\sum_k P_k = I$ and $P_k P_l = P_k \delta_{kl}$.) Also, various environmental disturbances to the quantum system can be modeled by suitable choices of $\{M_{\mu}\}$.

In going from a description based on $|\psi\rangle$ to the one based on ρ, the degrees of freedom get squared. This property is fully consistent with the Schmidt decomposition, which implies that any correlation between the system and the environment can be specified by modeling the environment using a set of degrees of freedom as large as that for the system. The squaring of the degrees of freedom is the price to be paid for the flexibility to include all possible environmental effects on the quantum system, and it slows down the performance of our quantum simulator.

We consider computational problems whose algorithms have already been converted to discrete quantum logic circuits acting on a set of qubits. We also assume that all logic gate instructions can be executed with a fixed clock step. In this framework, the computational complexity of the program is specified by the number of qubits and the total number of clock steps. Since the quantum state deteriorates with time due to environmental disturbances, we reduce the total execution time by identifying non-overlapping logic operations at every clock step and then implementing them in parallel.

Our quantum simulator is an open-source software written in Python, which is added as a new backend to IBM's Qiskit platform [5]. That extends the existing Qiskit capability, while retaining the convenience (e.g. portability, documentation, graphical interface) of the Qiskit format. Being an open-source software platform, Qiskit is popular, and a variety of quantum algorithms have been implemented using it [6]. Its comparison with other quantum computing software packages, in terms of features and performance, is also available [7]. Our simulator, with a user guide, is available as a "derivative work" of Qiskit at https://github.com/indian-institute-of-science-qc/qiskit-aakash.

A. The Quantum State

We express the density matrix of an n-qubit quantum register in the orthogonal Pauli basis:

$$\rho = \sum_{i_1,i_2,\ldots,i_n} a_{i_1 i_2 \ldots i_n}(\sigma_{i_1} \otimes \sigma_{i_2} \otimes \ldots \otimes \sigma_{i_n}) . \quad (2)$$

Here $i_1, i_2, \ldots, i_n \in \{0, 1, 2, 3\}$, $\sigma_0 \equiv I$, and $a_{i_1 i_2 \ldots i_n}$ are 4^n real coefficients encoded as an array. The normalisation $Tr(\rho) = 1$ implies $a_{00\ldots0} = 2^{-n}$, and the constraint $Tr(\rho^2) \leq 1$ implies $\sum_{i_1,i_2,\ldots,i_n} a_{i_1 i_2 \ldots i_n}^2 \leq 2^{-n}$. We find this expression for the density matrix easier to work with, compared to expressing it as a $2^n \times 2^n$ complex matrix.

Quantum dynamics is linear in terms of ρ. Moreover, we consider problems where all operations—logic gates, errors and measurements—are local, i.e. act on only a few qubits. Then during a single operation, only a few subscripts of $a_{i_1 i_2 \ldots i_n}$ change while all the rest remain unaltered. Such operations are efficiently implemented in the software using linear algebra vector instructions, with explicit evaluation of Eq. (1). We list density matrix transformations for some commonly used logic gates in Appendix A.

We allow several options to initialise the density matrix: The all-zero state is $2^{-n}(I + \sigma_3)^{\otimes n}$, the uniform superposition state is $2^{-n}(I + \sigma_1)^{\otimes n}$, a state specified by a binary string of 0's and 1's is mapped to a matching tensor product of $\frac{1}{2}(1 + \sigma_3)$ and $\frac{1}{2}(1 - \sigma_3)$ factors, and a custom density matrix can be read from a file as the 4^n coefficients. We also use the overlap, $Tr(\rho_1\rho_2)$, as a convenient measure of closeness of two density matrices.

B. The Logic Gates

Generic unitary transformations acting on the density matrix belong to the group $SU(2^n)$, since ρ does not contain the overall unobservable phase that $|\psi\rangle$ has. It is well-known that any such transformation can be decomposed in to a sequence of one-qubit and two-qubit logic

gates. The optimal choice for these elementary gates depends on what operations are convenient to execute on the quantum hardware, but it is a small set in any case. We choose this select set to be the one-qubit rotations about the fixed Cartesian axes (i.e. x, y and z) and the two-qubit C-NOT gate, which is suitable for most hardware implementations.

We assume that the program to be executed is available as a time-ordered sequence of logic gate operations. If that is not so, then a compiler would be needed to convert instructions in a high-level language to a sequence of logic gate operations. We also assume that the C-NOT gate can be applied between any two qubits of a register. If there are restrictions on qubit connectivity, then again it would be the task of a compiler to express the C-NOT gate as a sequence of operations along an available qubit interaction route.

We follow the Qiskit convention in describing the logic gates. A single qubit rotation by angle θ about the axis $\hat{n}$ is $R_n(\theta) = e^{-i\hat{n}\cdot\vec{\sigma}\theta/2}$. Then using Euler decomposition, any single qubit rotation is decomposed as:

$$
\begin{aligned}
u3(\theta, \phi, \lambda) &= e^{-i(\phi+\lambda)/2} \begin{pmatrix} \cos\frac{\theta}{2} & -e^{i\lambda}\sin\frac{\theta}{2} \\ e^{i\phi}\sin\frac{\theta}{2} & e^{i(\phi+\lambda)}\cos\frac{\theta}{2} \end{pmatrix} \\
&= R_z(\phi)R_y(\theta)R_z(\lambda) .
\end{aligned} \tag{3}
$$

We also use the Qiskit preprocessor and transpiler to simplify the quantum logic circuit. First, the preprocessor converts several of the commonly used quantum logic gates to the select set of gates, e.g. the phase gates are expressed in terms of $u1(\theta) = R_z(\theta)$, the Hadamard gate becomes $H = i\,u3(\frac{\pi}{2}, 0, \pi)$, and the Toffoli gate ($C^2$-NOT) becomes a combination of C-NOT and phase gates. Then the transpiler optimises the quantum logic circuit, wherever possible, by collapsing adjacent gates and by cancelling gates using commutation rules.

C. Projective Measurements

Given the density matrix ρ, the expectation value of any Hermitian operator O is easily obtained by expressing it in the Pauli basis,

$$
O = \sum_{i_1, i_2, \ldots, i_n} b_{i_1 i_2 \ldots i_n} (\sigma_{i_1} \otimes \sigma_{i_2} \otimes \ldots \otimes \sigma_{i_n}) , \tag{4}
$$

and then evaluating the inner product,

$$
\langle O \rangle = Tr(O\rho) = 2^n \sum_{i_1, i_2, \ldots, i_n} a_{i_1 i_2 \ldots i_n} b_{i_1 i_2 \ldots i_n} . \tag{5}
$$

Furthermore, the reduced density matrix with the degrees of freedom of kth qubit summed over, $Tr_k(\rho)$, is specified by the 4^{n-1} coefficients $2a_{i_1 \ldots i_{k-1} 0 i_{k+1} \ldots i_n}$. This prescription can be repeated to reduce the density matrix over as many qubits as desired.

Quantum measurement modifies the quantum state as well; components orthogonal to the direction of measurement vanish up on a projective measurement. So when

k^{th} qubit is measured along direction $\hat{n}$, the coefficients $a_{\ldots i_k \ldots}$ are set to zero for $i_k \perp \hat{n}$, while those for $i_k \| \hat{n}$ and $i_k = 0$ remain unchanged.

With these ingredients, we have implemented several projective measurement options:

• *Expectation value of a Pauli operator string:* This is just $\langle \sigma_{i_1} \otimes \sigma_{i_2} \otimes \ldots \otimes \sigma_{i_n} \rangle = 2^n a_{i_1 i_2 \ldots i_n}$. The density matrix is updated for each k, by projecting the coefficients with subscripts $i_k \in \{1, 2, 3\}$ and leaving those with subscripts $i_k = 0$ unaltered.

• *Single qubit measurement:* When the kth qubit is measured along direction $\hat{n}$, the two possible results have probabilities $\frac{1}{2}\langle (I \pm \hat{n} \cdot \vec{\sigma})_k \rangle$. In terms of the coefficients $\{c_0, \vec{c}\} = a_{0 \ldots i_k \ldots 0}$, the two probabilities are $\frac{1}{2}(1 \pm 2^n \hat{n} \cdot \vec{c})$. The density matrix is updated by projecting coefficients with subscript i_k.

• *Ensemble measurement:* For simultaneous binary measurement of all the qubits in the computational basis, the probabilities of the 2^n possible results are:
$2^{-n}\langle \prod_k (I \pm \sigma_3)_k \rangle = \sum_{j_1, \ldots, j_n \in \{0,3\}} (\prod_k s_k) a_{j_1 \ldots j_n}$, with the sign $s_k = \delta_{j_k, 0} \pm \delta_{j_k, 3}$. This measurement is typically performed at the end of the computation.

• *Bell-basis measurement of a pair of qubits:* For this joint measurement of two qubits, the projection operators for the four orthonormal state vectors are given by:

$$
\frac{|00\rangle + |11\rangle}{\sqrt{2}} \rightarrow \frac{1}{4}(I \otimes I + \sigma_1 \otimes \sigma_1 - \sigma_2 \otimes \sigma_2 + \sigma_3 \otimes \sigma_3),
$$

$$
\frac{|00\rangle - |11\rangle}{\sqrt{2}} \rightarrow \frac{1}{4}(I \otimes I - \sigma_1 \otimes \sigma_1 + \sigma_2 \otimes \sigma_2 + \sigma_3 \otimes \sigma_3),
$$

$$
\frac{|01\rangle + |10\rangle}{\sqrt{2}} \rightarrow \frac{1}{4}(I \otimes I + \sigma_1 \otimes \sigma_1 + \sigma_2 \otimes \sigma_2 - \sigma_3 \otimes \sigma_3),
$$

$$
\frac{|01\rangle - |10\rangle}{\sqrt{2}} \rightarrow \frac{1}{4}(I \otimes I - \sigma_1 \otimes \sigma_1 - \sigma_2 \otimes \sigma_2 - \sigma_3 \otimes \sigma_3).
$$

So for the Bell-basis measurement of kth and l^{th} qubits, the probabilities of the four outcomes are determined by the coefficients with subscripts $i_k = i_l$ and all other subscripts set to zero. These four probabilities can be used to quantify entanglement between the two qubits. The post-measurement density matrix is obtained by setting the coefficients with $i_k \neq i_l$ to zero, while not changing those with $i_k = i_l$.

• *Qubit reset:* Although not a measurement, this instruction permits reuse of a qubit. The transformation to reset a quantum state to $|0\rangle$ is: $\rho \rightarrow P_0 \rho P_0 + \sigma_1 P_1 \rho P_1 \sigma_1$. When the kth qubit is reset, the coefficients with $i_k \in \{1, 2\}$ are made zero, and the coefficient with $i_k = 3$ is made equal to the one with $i_k = 0$.

• *Complete tomography:* The 4^n coefficients of the density matrix determine expectation values of all the physical operators that can be measured in principle, although only a set of mutually commuting operators can be measured in a single quantum experiment. In our classical simulation, we can store the full density matrix at any stage of a program, and use it later as the initial state of another program.

D. The Partitioned Logic Circuit

An open quantum system continuously deteriorates in time. To mitigate that, it is useful to reduce the total execution time of a quantum program as much as possible. Towards this end, we restructure the quantum circuit produced by the transpiler as follows.

The preprocessor decomposes the logic gates provided by the user to the select set $\{u1, u3, \text{C-NOT}\}$. This decomposition adds to the number of logic gates in the circuit, increasing its depth. As a countermeasure, we go through the sequence of operations on each qubit, and merge consecutive single-qubit rotations that we find in to a single one (e.g. $u3 * u3 \to u3$), using $SU(2)$ group composition rules.

Next we arrange the complete list of instructions in to a set of partitions, such that all operations in a single partition can be executed as parallel threads during a single clock step. To accomplish this, the clock step has to be longer than the execution times of individual operations (i.e. $u3$, C-NOT and various measurements). We partition the circuit by organising the list of instructions as a stack of sequential operations for every qubit, introducing barriers such that each qubit can have at most one operation in a partition, and combining non-overlapping qubit operations in to a single partition wherever possible. In particular, this procedure puts logic gate operations and measurement operations in separate partitions (note that a single qubit measurement may affect the whole quantum register in case of entangled quantum states). Thus a partition may have either multiple quantum logic gates operating on different qubits, or multiple single qubit measurements on distinct qubits. We let expectation value calculations, ensemble measurements and Bell-basis measurements form partitions on their own.

We provide details of the merging and the partitioning logic circuit operations in Appendices A and B, with simple illustrations.

III. THE NOISY EVOLUTION

The manipulations of circuit operations described in the previous section are carried out at the classical level; even when a quantum hardware is available, they would be implemented by a classical compiler. So we safely assume that they are error-free. It is the execution of the partitioned circuit on a quantum backend that is influenced by the environment. Assuming that the environment disturbs each qubit independently, we now present simple models that include the environmental noise in the simulator at various stages of the program execution.

A. Initialisation Error

The initial state of the program is often an equilibrium state. So we allow a fully-factorised thermal state as one of the initial state options:

$$\rho_{\text{th}} = \begin{pmatrix} p & 0 \\ 0 & 1-p \end{pmatrix}^{\otimes n} , \quad \frac{p}{1-p} = \exp\left(\frac{E_1 - E_0}{kT}\right) . \quad (6)$$

Here the parameter p is provided by the user.

B. Logic Gate Execution Error

The single qubit rotations in our select set have fixed rotation axes, and we assume that errors arise from inaccuracies in their rotation angles. Let α denote the inaccuracy in the angle, with the mean $\langle\!\langle \alpha \rangle\!\rangle = \overline{\alpha}$ and the fluctuations symmetric about $\overline{\alpha}$. Then the replacement $\theta \to \theta + \alpha$ in $R_n(\theta)$ modifies the density matrix transformation according to the substitutions:

$$\cos\theta \to r\cos(\theta + \overline{\alpha}) , \quad \sin\theta \to r\sin(\theta + \overline{\alpha}) , \quad (7)$$

where $\overline{\alpha}$ and $r = \langle\!\langle \cos(\alpha - \overline{\alpha}) \rangle\!\rangle$ are the parameters provided by the user. They may depend on the rotation axis (i.e. x, y or z).

To model the error in the C-NOT gate, we assume that C-NOT is implemented as a transition selective pulse that exchanges amplitudes of the two target qubit levels when the control qubit state is $|1\rangle$. Then the error is in the duration of the transition selective pulse, and alters only the second half of the unitary operator, $U_{cx} = |0\rangle\langle 0| \otimes I + |1\rangle\langle 1| \otimes \sigma_1$. It can be included in the same manner as the error in single qubit rotation angle (i.e. as a disturbance to the rotation operator σ_1). The corresponding two parameters, analogous to $\overline{\alpha}$ and r, are provided by the user.

C. Measurement Error

Projective measurements of quantum systems are not perfect in practice. We model a single qubit measurement error as depolarisation, which is equivalent to a bit-flip error in a binary measurement. Then when the kth qubit is measured along direction $\hat{n}$, the coefficients $a_{i_1 \ldots i_k \ldots i_n}$ in the post-measurement state are set to zero for $i_k \perp \hat{n}$, reduced by a multiplicative factor d_1 for $i_k \| \hat{n}$, and left unaffected for $i_k = 0$. Also, the probabilities of the two outcomes become $\frac{1}{2}(1 \pm 2^n d_1 \hat{n} \cdot \vec{c})$, in the notation of Section II.C. Here the parameter d_1 is provided by the user. In case of a measurement of a multi-qubit Pauli operator string, the above procedure is applied to every qubit whose measurement operator has $i_k \neq 0$.

In the case of a Bell-basis measurement, the post-measurement coefficients with $i_k \neq i_l$ are set to zero, those with $i_k = i_l \in \{1, 2, 3\}$ are reduced by a multiplicative factor d_2, and those with $i_k = i_l = 0$ are left the same. Also, the probabilities of the four outcomes are obtained by reducing the $i_k = i_l \in \{1, 2, 3\}$ contributions by the factor d_2 that is provided by the user.

D. Memory Errors

An open quantum system undergoes decoherence and decay, irrespective of whether it is being manipulated by some instruction or not. These effects cause maximum damage to a quantum signal, because they act on all the qubits all the time, while operational errors are confined to particular qubits at specific times. We assume that these memory errors are small during a clock step, and implement them by modifying the density matrix at the end of every clock step, in the spirit of the Trotter expansion. Such an implementation is actually the reason behind our partitioning of the quantum circuit.

Taking the σ_3 basis as the computational basis, the decoherence effect is to suppress the off-diagonal coefficients with $i_k \in \{1, 2\}$ for every qubit by a multiplicative factor f. It can be represented by the Kraus operators:

$$M_0 = \sqrt{\frac{1+f}{2}}\, I \ , \quad M_1 = \sqrt{\frac{1-f}{2}}\, \sigma_3 \ . \qquad (8)$$

In terms of the clock step Δt and the decoherence time T_2, the parameter $f = \exp(-\Delta t / T_2)$, and it is provided by the user.

We consider the decay of the quantum state towards the thermal state, ρ_{th}, defined in Section III.A. This evolution is represented by the Kraus operators:

$$M_0 = \sqrt{p}\begin{pmatrix} 1 & 0 \\ 0 & \sqrt{g} \end{pmatrix}, \ M_1 = \sqrt{p}\begin{pmatrix} 0 & \sqrt{1-g} \\ 0 & 0 \end{pmatrix}, \qquad (9)$$

$$M_2 = \sqrt{1-p}\begin{pmatrix} \sqrt{g} & 0 \\ 0 & 1 \end{pmatrix}, \ M_3 = \sqrt{1-p}\begin{pmatrix} 0 & 0 \\ \sqrt{1-g} & 0 \end{pmatrix}.$$

Its effect on every qubit is to suppress the off-diagonal coefficients with $i_k \in \{1, 2\}$ by $\sqrt{g}$, and change the diagonal coefficients according to:

$$a_{...3...} \rightarrow g\, a_{...3...} + (2p-1)(1-g)a_{...0...} \ . \qquad (10)$$

In terms of the clock step Δt and the relaxation time T_1, the parameter $g = \exp(-\Delta t / T_1)$, and it is provided by the user. (Note that our Kraus representation automatically ensures the physical constraint $T_2 \leq 2T_1$).

We execute both the decoherence and the decay operations at the end of every partition.

IV. TESTS AND EXAMPLES

We have tested our density matrix simulator against Qiskit's state vector version, using circuits of randomly generated quantum logic operations. Both give identical results, when all the errors are absent. Since our density matrix simulator works with 4^n coefficients, it is slower than the state vector simulator that works with 2^n coefficients. On the other hand, it produces the complete output probability distribution in one run, while the state vector simulators require multiple runs of the program for the same purpose. We can simulate circuits with 10
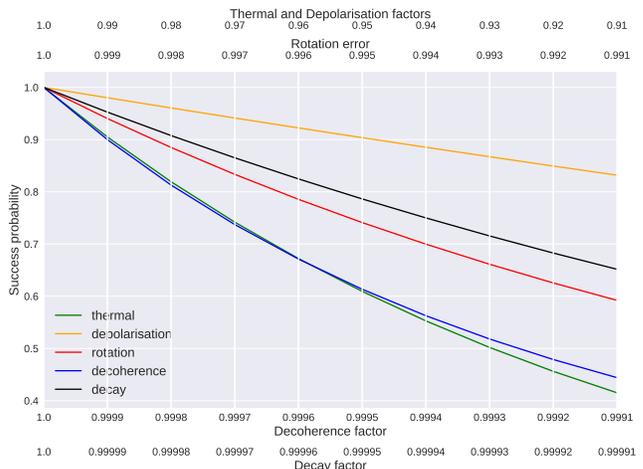

FIG. 1: Success probability of the binary addition program, 110+11=1001, as a function of different types of errors. For easy comparison, multiple parameters are plotted along the X-axis with different scales: thermal factor p (green), depolarisation factor d_1 (orange), rotation error parameter r (red), decoherence factor f (blue) and decay factor g (black). In all cases, the success probability is observed to decrease exponentially.

qubits and 100 operations in a few minutes on a laptop; it would be practical to handle larger quantum systems, say up to 15 qubits and 1000 operations, on more powerful dedicated computers.

The main achievement of our simulator is the ability to simulate noisy quantum systems, using simple error models. In such simulations, the final results are probability distributions over the possible outcomes of the algorithm, and their stability against variations of the error parameters can be explicitly checked. The distributions can be easily visualised using various types of plots, and we expect exponential deterioration of the quantum signal with increasing error rates.

As a straightforward example, we simulated the binary addition algorithm. That requires three quantum subregisters, two for the two numbers and one for the carry bit. We varied the error parameters one at a time, and observed the probability distributions of the final sum. (To interpret the results correctly, we needed to invert Qiskit's convention of the least significant bit first and the most significant bit last, to the standard numerical convention of the most significant bit first and the least significant bit last.) Our results for the probability of the correct answer, for the addition $110 + 11 = 1001$, are shown in Fig. 1. Although the success probability decreases exponentially in all the cases, we see a wide variation in sensitivity of the calculation to the different types of errors. Thermal (p) and depolarisation (d_1) factors act only at the ends of the program, and produce the smallest errors. Rotation angle fluctuations (r) in the logic gates give rise to intermediate size errors. Decoherence (f) and decay (g) factors that act throughout the
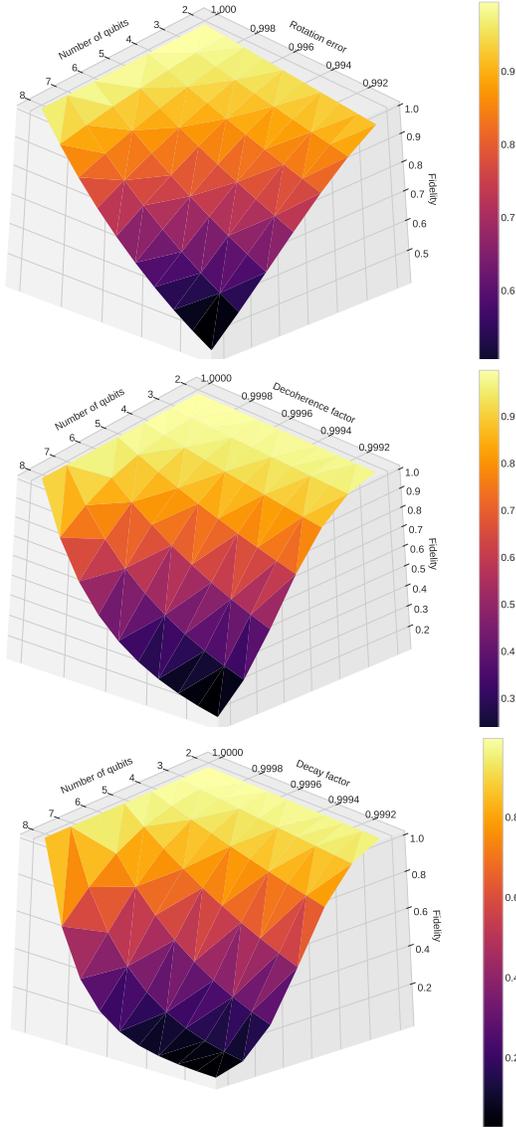
FIG. 2: Fidelity of the Quantum Fourier Transform program, for different number of qubits n and as a function of different error parameters: (Top) The rotation error parameter r, with the same value for R_x, R_y, R_z and C-NOT operations; (Middle) The decoherence parameter f; (Bottom) The decay parameter g. The fidelity decreases first quadratically and then exponentially, both as a function of n and the error parameters.

program cause the largest errors, with decay dominating over decoherence. This pattern, together with the actual error parameter values, gives us an estimate of how accurately we need to control various errors in quantum hardware in order to get meaningful results. We also observed that decreasing p, d_1 and g more or less kept the probability distribution centred around the correct answer, but decreasing f tended to make the distribution flat and decreasing g drove the distribution towards the all-zero state.

As a second example, we simulated the Quantum Fourier Transform algorithm for various number of qubits, again varying the error parameters one at a time. Our results for the final state fidelity, with reference to the exact result, are displayed in Fig. 2. We find that the fidelity deviates from 1 quadratically for very small errors, but subsequently drops exponentially, both as a function of the number of qubits and the error parameters. This is the expected behaviour, and the hierarchy of sensitivity to different errors is the same as in case of the addition algorithm.

Appendix A: Some Logic Gate Transformations for the Density Matrix

The single qubit density matrix is $\rho = a_0 I + \vec{a} \cdot \vec{\sigma}$, with $a_0 = \frac{1}{2}$. It is straightforward to apply commonly used one-qubit logic gates to it:

$$
\begin{aligned}
\sigma_1 \rho \sigma_1 &= a_0 I + a_1 \sigma_1 - a_2 \sigma_2 - a_3 \sigma_3 ~, \\
\sigma_2 \rho \sigma_2 &= a_0 I - a_1 \sigma_1 + a_2 \sigma_2 - a_3 \sigma_3 ~, \\
\sigma_3 \rho \sigma_3 &= a_0 I - a_1 \sigma_1 - a_2 \sigma_2 + a_3 \sigma_3 ~, \\
H \rho H &= a_0 I + a_3 \sigma_1 + a_2 \sigma_2 + a_1 \sigma_3 ~, \\
S \rho S^\dagger &= a_0 I - a_2 \sigma_1 + a_1 \sigma_2 + a_3 \sigma_3 ~, \\
S^\dagger \rho S &= a_0 I + a_2 \sigma_1 - a_1 \sigma_2 + a_3 \sigma_3 ~, \\
T \rho T^\dagger &= a_0 I + \frac{a_1 - a_2}{\sqrt{2}} \sigma_1 + \frac{a_1 + a_2}{\sqrt{2}} \sigma_2 + a_3 \sigma_3 ~, \\
T^\dagger \rho T &= a_0 I + \frac{a_1 + a_2}{\sqrt{2}} \sigma_1 - \frac{a_1 - a_2}{\sqrt{2}} \sigma_2 + a_3 \sigma_3 ~.
\end{aligned}
\tag{11}
$$

Rotation errors in these one-qubit transformations are incorporated by changing them from $R_n(\theta) \rho R_n^\dagger(\theta)$ to $R_n(\theta + \alpha) \rho R_n^\dagger(\theta + \alpha)$.

The two-qubit C-NOT transformation is $U_{cx} \rho U_{cx}^\dagger$, with $U_{cx} = |0\rangle\langle 0| \otimes I + |1\rangle\langle 1| \otimes \sigma_1$. Transition selective pulse error in the C-NOT gate is included by changing σ_1 to $R_x(\alpha)\sigma_1$ in U_{cx}.

Consecutive rotations of a single qubit can be merged in to a single one using $SU(2)$ group composition rules. We rewrite Qiskit's $u2(\phi, \lambda)$ logic gate as $u3(\frac{\pi}{2}, \phi, \lambda)$, which reduces merging possibilities to only four cases: $u1 * u1$, $u1 * u3$, $u3 * u1$ and $u3 * u3$. The first three are easily taken care of by adding the R_z rotation angles, e.g. $u1(\theta_1) \times u1(\theta_2) = u1(\theta_1 + \theta_2)$. To take care of the last one, we express:

$$
\begin{aligned}
& u3(\theta_1, \phi_1, \lambda_1) * u3(\theta_2, \phi_2, \lambda_2) \\
&= R_z(\phi_2) R_y(\theta_2) R_z(\lambda_2) R_z(\phi_1) R_y(\theta_1) R_z(\lambda_1) \\
&= R_z(\phi_2) R_y(\theta_2) R_z(\lambda_2 + \phi_1) R_y(\theta_1) R_z(\lambda_1) \\
&= R_z(\phi_2) R_z(\alpha) R_y(\beta) R_z(\gamma) R_z(\lambda_1) \\
&= R_z(\phi_2 + \alpha) R_y(\beta) R_z(\gamma + \lambda_1) \\
&= u3(\beta, \phi_2 + \alpha, \gamma + \lambda_1) ~.
\end{aligned}
\tag{12}
$$

Here the YZY Euler decomposition on the third line is converted to the ZYZ Euler decomposition on the fourth
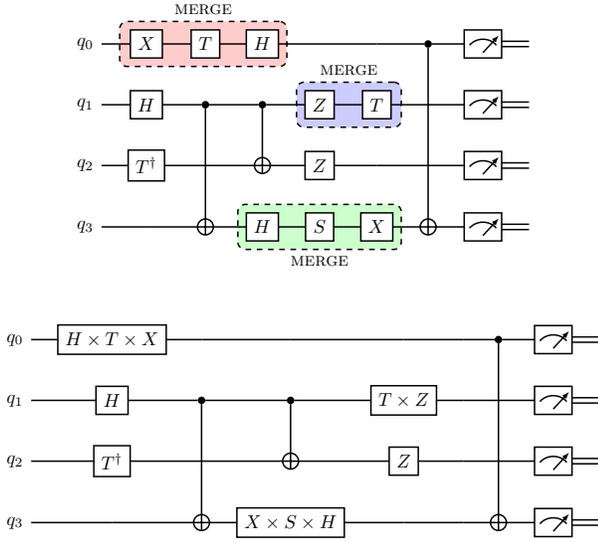
FIG. 3: (Top) A quantum logic circuit with commonly used gates. (Bottom) The same logic circuit after merging several one-qubit gates.

line. This conversion is conveniently performed by explicitly matching the product matrices and using the `arctan2` math-library function to extract the angles.

An illustration of how this merging can simplify a logic circuit is presented in Fig. 3. Note that the reversal of the operator order is due to the convention of the left-most gate acting first in a circuit and the rightmost factor acting first in a matrix operation.

Appendix B: Logic Circuit Rearrangement

To minimise decay and decoherence errors, we need to reduce the logic circuit depth as much as possible. For this purpose, we look for maximum parallelisation of the program provided as a time-ordered instruction set. We rearrange instructions in to a set of partitions, preserving their temporal order, such that all instructions in a given partition commute with each other and can be executed simultaneously while the partitions are executed in succession. Then each partition is assigned a clock step, and overall decay and decoherence errors depend on the total number of partitions.

We note that all our logic gates including their errors involve only one or two qubits, while any projective measurement operation may affect the density matrix globally. So the partitions fall in to two categories; they have either only unitary logic gates ($u1$, $u3$ or C-NOT) or only projective measurement operations. These two categories can have different clock step duration if required. To begin with, we therefore go through the whole instruction set and insert barriers between sets of consecutive logic gates and sets of consecutive measurement operations. We also separate out expectation value calculations, en-
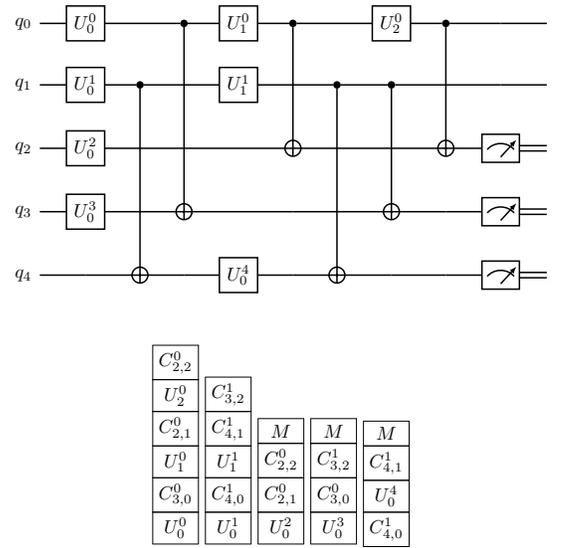
FIG. 4: (Top) A quantum logic circuit specified by sequential instructions. (Middle) The qubit operation stack generated from the logic circuit. (Bottom) The partitioned logic circuit constructed from the qubit stack.

semble measurements and Bell-basis measurements from the rest of the instructions by inserting barriers. The instructions between successive barriers are then inspected to check if their further partitioning is necessary.

We implement a simple partitioning procedure, which may not be optimal, but works well in practice. Our first step is to construct a qubit stack from the instruction set, which lists the temporal sequence of instructions that act on every qubit. A multi-qubit instruction (e.g. C-NOT) is listed in the column of each participating qubit. In case of a single qubit measurement or reset, we add a dummy instruction to the rest of the qubit columns as a barrier. An example of this construction is shown in Fig. 4, and the pseudocode of our algorithm is presented in Fig. 5.

Our next step is to sequentially inspect the bottom instruction for every qubit column in the stack, and pop it in to a new partition under certain conditions. In case of a logic gate partition, at most only one instruction from a column can go in to a partition, and a multi-qubit instruction can go in to the partition only if it is

---
Algorithm 1: Algorithm for constructing qubit stack from instruction set
---

Input : Instruction Set: *iSet*,
    Number of qubits: *numQubits*
Output: Qubit stacks: *stacks*,
    Stacks maximum depth: *depth*

1 **def** *qubitStacks (iSet, numQubits)*:
2   Initialize *numQ* empty stacks: $[\,[\,],[\,]\ldots[\,]\,] \leftarrow qubitStack$
3   **for** *instruction* **in** *iSet*:
4   **if** *not isMeasure(instruction)* **and** *not isReset(instruction)*:
5   **for** *qubit* **in** *instruction.qubits*:
6   qubitStack[qubit].append(instruction)
7   **elif** *isMeasure(instruction)*:
8   qubit ← *instruction.qubits*[0]
9   **if** *not isMeasureDummy(qubitStack[qubit][-1])*:
10   qubitStack[qubit].append(instruction)
11   **for** *qb* **in**
      set(range(numQ)).difference(set(instruction.qubits)):
12   qubitStack[qb].append(dummymeasureinstruction)
13   **else**:
14   qubitStack[qubit][-1] ← instruction
15   **elif** *isReset(instruction)*:
16   qubit ← *instruction.qubits*[0]
17   **if** *not isMeasureDummy(qubitStack[qubit][-1])*:
18   qubitStack[qubit].append(instruction)
19   **for** *qb* **in**
      set(range(numQ)).difference(set(instruction.qubits)):
20   qubitStack[qb].append(dummyresetinstruction)
21   **else**:
22   qubitStack[qubit][-1] ← instruction
23   depth ← max([len(stack) **for** stack **in** qubitStack])
24   **return** qubitStack, depth

FIG. 5: Pseudocode for the algorithm that constructs the qubit operation stack from the instruction set.

present at the bottom of columns of each participating qubit. In case of a measurement partition, dummy instructions are ignored, which allows simultaneous single qubit measurement or reset on distinct qubits to be in the same partition (we have called that partial measurement) while preventing successive measurements on the same qubit to do so. This process of creating new partitions is repeated until all the columns in the qubit stack become empty. An example of this procedure is shown in Fig. 4, and the pseudocode of our algorithm is presented in Fig. 6.

At the end, we point out that an efficient software rescheduling of the program instructions is desirable even when the algorithm is to be implemented on a quantum hardware.

Algorithm 2: Algorithm for partitioning the logic circuit

Input : Instruction Set: *iSet*,
    Number of qubits: *numQ*
Output: Partitioned instruction set: *piSet*,
    Number of partitions: *levels*

1 **def** *partitionInstructions (iSet, numQ)*:
2   iStack, depth ← qubitStacks(iSet, numQ)
3   Initialize empty partitions: $[\,[\,],[\,]\ldots[\,]\,] \leftarrow sequence$
4   Initialize level: level
5   **while** *iSet*:
6   **if** *level == len(sequence)*:
7   sequence.append([ ])
8   **for** *qubit* **in** *range(numQ)*:
9   **if** *iStack[qubit]*:
10   gate ← iStack[qubit][0]
11   **else**:
12   **continue**
13   **if** *isDummy(gate)*:
14   **continue**
15   **elif** *isSingle(gate)*:
16   sequence[level].append(gate)
17   iSet.remove(gate)
18   iStack[qubit].pop(0)
19   **elif** *isCX(gate)*:
20   firstQb, secondQb = gate.qubits
21   currGate ← iStack[firstQb][0]
22   buffGate ← iStack[secondQb][0]
23   **if** *currGate == buffGate*:
24   sequence[level].append(gate)
25   iSet.remove(gate)
26   iStack[firstQb].pop(0)
27   iStack[secondQb].pop(0)
28   **else**:
29   **continue**
30   **elif** *isMeasure(gate)*:
31   allDummy ← True
32   **for** *x* **in** *numQ*:
33   **if** *not isMeasure(iStack[x][0]* **and** *not*
34   *isMeasureDummy(iStack[x][0]))*:
35   allDummy ← False
36   **break**
37   **if** *allDummy*:
38   **for** *x* **in** *range(numQ)*:
39   instruction ← iStack[x][0]
40   **if** *isMeasure(instruction)*:
41   sequence[level].append(instruction)
42   iSet.remove(instruction)
43   iStack[x].pop(0)
44   **break**
45   **elif** *isReset(gate)*:
46   allDummy ← True
47   **for** *x* **in** *numQ*:
48   **if** *not isReset(iStack[x][0]* **and** *not*
49   *isResetDummy(iStack[x][0]))*:
50   allDummy ← False
51   **break**
52   **if** *allDummy*:
53   **for** *x* **in** *range(numQ)*:
54   instruction ← iStack[x][0]
55   **if** *isReset(instruction)*:
56   sequence[level].append(instruction)
57   iSet.remove(instruction)
58   iStack[x].pop(0)
59   **break**
60   **if** *not iSet*:
61   **break**
62   level ← level + 1
63   **return** sequence, level

FIG. 6: Pseudocode for the algorithm that partitions the qubit operation stack in to sequential levels.

[1] J. Preskill, Lecture Notes for the Course on Quantum Computation, http://www.theory.caltech.edu/people/preskill/ph219/

[2] M.A. Nielsen and I.L. Chuang, *Quantum Computation and Quantum Information*, (Cambridge University Press, 2000).

[3] J. Preskill, Quantum Computing in the NISQ Era and Beyond, Quantum 2 (2018) 79.

[4] See for instance, http://quantiki.org/wiki/list-qc-simulators

[5] See https://qiskit.org/ and https://github.com/Qiskit/qiskit-terra Qiskit has copyright under Apache License 2.0.

[6] See for instance, P.J. Coles et al., *Quantum Algorithm Implementations for Beginners*, arXiv:1804.03719.

[7] See for instance, R. LaRose, *Overview and Comparison of Gate Level Quantum Software Platforms*, arXiv:1807.02500.