SymPhase: Phase Symbolization for Fast Simulation of Stabilizer Circuits

Wang Fang[1,2], Mingsheng Ying[1,3]
[1]Institute of Software, Chinese Academy of Sciences, [2]University of Chinese Academy of Sciences, [3]Tsinghua University
{fangw,yingms}@ios.ac.cn

Abstract

This paper proposes an efficient stabilizer circuit simulation algorithm that only traverses the circuit forward once. We introduce phase symbolization into stabilizer generators, which allows possible Pauli faults in the circuit to be accumulated explicitly as symbolic expressions in the phases of stabilizer generators. This way, the measurement outcomes are also symbolic expressions, and we can sample them by substituting the symbolic variables with concrete values, without traversing the circuit repeatedly. We show how to integrate symbolic phases into the stabilizer tableau and maintain them efficiently using bit-vector encoding. A new data layout of the stabilizer tableau in memory is proposed, which improves the performance of our algorithm (and other stabilizer simulation algorithms based on the stabilizer tableau). We implement our algorithm and data layout in a Julia package named `SymPhase.jl`, and compare it with Stim, the state-of-the-art simulator, on several benchmarks. We show that `SymPhase.jl` has superior performance in terms of sampling time, which is crucial for generating a large number of samples for further analysis.

1 Introduction

With the rapid development of quantum hardware, designing and building large-scale fault-tolerant quantum computer architecture has become an urgent task [3, 9, 10]. It relies on quantum error correction (QEC) protocols, for which the implementation relies on stabilizer circuits [12]. Due to the complexity and unintuitive nature of quantum systems, it is essential to have efficient methods for simulating stabilizer circuits on classical computers, as this can help us design and test circuits and protocols before deploying them on quantum hardware, like classical EDA tools.

Fortunately, stabilizer circuits are a special class of quantum circuits that can be simulated in polynomial time on classical computers [13]. There are several efficient stabilizer circuit simulators available [2, 4, 11, 14], but they are still not sufficient for analyzing fault-tolerant gadgets. A typical example is that we need to repeatedly sample the faults that occur inside the circuit of a gadget and count the measurement outcomes of the circuit under these fault samples to evaluate the performance of the gadget. Existing simulators can generate a single sample very fast, but the number of samples can be in the millions when the circuit is large, making the simulation very slow. The state-of-the-art stabilizer circuit simulator, Stim, also mentioned that generating samples of QEC circuits remains the bottleneck in analysis [11].

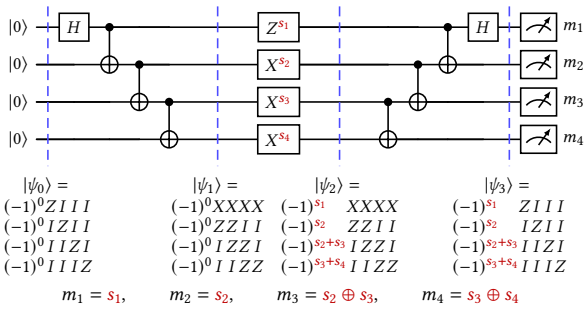


Figure 1: Overview of phase symbolization. Pauli faults in stabilizer circuits only affect the phases of stabilizer generators. As a result, possible Pauli faults can be accumulated explicitly in the phases with symbolic expressions, making measurement outcomes into symbolic expressions. With these symbolic expressions, we only need to substitute symbolic variables with concrete values to achieve sampling measurement outcomes, thus avoiding the cost of repeatedly traversing the circuit.

To address the difficulty of generating large samples of measurement outcomes, we propose a novel idea of phase symbolization for simulating stabilizer circuits. In standard stabilizer circuit simulations [2, 11, 12], where evolutions of quantum states are tracked with stabilizer generators (see the lists of Pauli strings in Fig. 1), we note that Pauli gates only affect the phase of stabilizer generators. For example, in Fig. 1, the gate Z^{s_1} with $s_1 \in \{0, 1\}$ only changes a phase $(-1)^0$ of $|\psi_1\rangle$ to the phase $(-1)^{s_1}$ of $|\psi_2\rangle$. As a result, possible Pauli faults in stabilizer circuits can be accumulated in the phase with symbolic variables as shown in Fig. 1, where $|\psi_1\rangle$ becomes $|\psi_2\rangle$ after passing through $Z^{s_1}, X^{s_2}, X^{s_3}$ and X^{s_4}. The introduction of this symbolization will not change the control flow of the standard stabilizer circuit simulation algorithm, thus we can easily extend existing algorithms with phase symbolization, but it will make the measurement outcomes into some symbolic expressions as these m_1, m_2, m_3, m_4 in Fig. 1. With these symbolic expressions, we can clearly see how the faults in the circuit affect the measurement outcomes, and we only need to substitute these symbolic variables with concrete values according to the fault model to achieve sampling measurement outcomes, thus avoiding the cost of repeatedly traversing the circuit.

Contribution and outline. After reviewing some background knowledge (§2), our major contributions are presented as follows:

- With the phase symbolization, an algorithm (Algorithm 1) for efficient sampling outcomes of stabilizer circuits that traverses the circuit only once is proposed (§3). Specifically, we describe how to integrate symbolic phases into the stabilizer tableau and maintain them efficiently through bit-vector encoding; and turn the sampling process into bit-matrix multiplication.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3655902

- For efficient implementation of our algorithm and also other stabilizer simulation algorithms based on stabilizer tableau, we propose a new data layout of the stabilizer tableau in the memory (§4), which has later been experimentally verified to have advantages over previous tools in some cases.
- We implement our algorithm and data layout in a Julia package named `SymPhase.jl` and evaluate its ability to surpass the state-of-the-art simulator, Stim, for sampling stabilizer circuits on several benchmarks (§5).

Related work. Stabilizer circuit simulation is a well-studied topic in quantum computing. A key method for simulating stabilizer circuits is the stabilizer tableau method proposed by [13] and improved by [2]. To speed up the sampling of stabilizer circuits with Pauli faults, a technique called Pauli frame was introduced by [17], which tracks the difference between the state with and without faults, and reduces the number of Pauli strings that need to be propagated for sampling an n-qubit circuit from n to 1. This method was also adopted by Stim, the state-of-the-art stabilizer simulator [11]. Recently, Delfosse and Paetznick [8] proposed a method that can extract the relationship between faults and measurement outcomes by traversing the circuit backward once, which greatly improves the sampling efficiency compared to previous work. Our work achieves the same result by traversing the circuit forward once. But the basic ideas of [8] and ours are fundamentally different. A comparison of the complexity of [8] with ours is presented in Table 1.

2 Background

This paper presupposes some basic knowledge of quantum computing, such as quantum bits (qubits) and quantum circuits. Readers who are unfamiliar with these concepts can refer to the textbook by Nielsen and Chuang [15, Chapter 2, 4].

2.1 Stabilizer Circuits

Pauli strings. There are four Pauli matrices:

$$I=\begin{pmatrix}1&0\\0&1\end{pmatrix},\quad X=\begin{pmatrix}0&1\\1&0\end{pmatrix},\quad Y=\begin{pmatrix}0&-i\\i&0\end{pmatrix},\quad Z=\begin{pmatrix}1&0\\0&-1\end{pmatrix}.$$

An n-qubit Pauli string is a tensor product of n Pauli matrices with a phase of ±1 or $\pm i$, e.g., $-XYZI = -X\otimes Y\otimes Z\otimes I$ is a 4-qubit Pauli string. We usually omit tensor product signs. To simplify the notation in dealing with multiple qubits, we also omit the I matrices in Pauli strings and use subscripts to indicate the qubits that the non-identity Pauli matrices act on. For example, $X_1Y_2Z_3$ means applying X to qubit 1, Y to qubit 2, Z to qubit 3 and I to the rest of qubits; when restricted to 4 qubits, $X_1Y_2Z_3$ is regarded as $XYZI$.

Stabilizer generators and stabilizer states. A state $|\psi\rangle$ is *stabilized* by a unitary U if $U|\psi\rangle = |\psi\rangle$, i.e., $|\psi\rangle$ is an eigenvector of U with eigenvalue 1. For example, the minus state $|-\rangle$ is stabilized by $-X$ and the bell state $|\beta_{00}\rangle = \frac{1}{\sqrt{2}}(|00\rangle+|11\rangle)$ is stabilized by XX. In this paper, we only consider states stabilized by Pauli strings. For an n-qubit state $|\psi\rangle$, let $\mathrm{Stab}(|\psi\rangle)$ denote the set of all n-qubit Pauli strings that stabilize $|\psi\rangle$. For any $P,Q\in\mathrm{Stab}(|\psi\rangle)$, we can easily check that $P\cdot Q, P\cdot Q^{-1}\in\mathrm{Stab}(|\psi\rangle)$, thus $\mathrm{Stab}(|\psi\rangle)$ is also a group and we call it the *stabilizer group* of $|\psi\rangle$. The independent generators, which are all Pauli strings, of the stabilizer group are called *stabilizer generators*.

An n-qubit state $|\psi\rangle$ is called a *stabilizer state* if $\mathrm{Stab}(|\psi\rangle)$ has n stabilizer generators. In this case, with global phase ignored, $|\psi\rangle$ is the only n-qubit state stabilized by $\mathrm{Stab}(|\psi\rangle)$. Therefore, there is a one-to-one correspondence between a stabilizer state $|\psi\rangle$ and its stabilizer group $\mathrm{Stab}(|\psi\rangle)$.

Clifford gates and stabilizer circuits. For a state $|\psi\rangle$ and a Pauli string P that stabilizes $|\psi\rangle$, a unitary U transforms $|\psi\rangle$ to $U|\psi\rangle$, which can be reflected by the transformation from P to $UPU^\dagger$ (conjugation by U) as $U|\psi\rangle$ is stabilized by $UPU^\dagger$. To ensure that $UPU^\dagger$ is still a Pauli string, we consider those unitaries U that conjugate Pauli strings to Pauli strings, i.e., for any Pauli string P, $UPU^\dagger$ is still a Pauli string. Such unitaries are called *Clifford gates* and can be constructed from the three gates [12]:

$$H=\frac{1}{\sqrt{2}}\begin{pmatrix}1&1\\1&-1\end{pmatrix},\quad S=\begin{pmatrix}1&0\\0&i\end{pmatrix},\quad CNOT=\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}$$

A *stabilizer circuit* is a quantum circuit that uses $H, S, CNOT$ gates (Clifford gates), computational measurements, and $|0\rangle^{\otimes n}$ as the initial state. The central idea of *stabilizer formalism* [15, Chapter 10.5] is to describe a state $|\psi\rangle$ by using its stabilizer group $\mathrm{Stab}(|\psi\rangle)$, which can be identified by stabilizer generators. For stabilizer circuits, the initial state $|0\rangle^{\otimes n}$ is a stabilizer state with n stabilizer generators $Z_1, Z_2, \ldots, Z_n$; the Clifford gates and computational measurements will turn it into states that also admit n stabilizer generators [15, Chapter 10.5]. This idea provides an efficient simulation of stabilizer circuits by tracking stabilizer generators, sometimes known as the *Gottesman-Knill theorem* [13].

2.2 Stabilizer Tableau Simulation

The most well-known approach to simulating n-qubit stabilizer circuits is to maintain an $n\times(2n+1)$ *stabilizer tableau* $(\boldsymbol{X}\mid\boldsymbol{Z}\mid\boldsymbol{R})$ that encodes n stabilizer generators $P_1,\ldots,P_n$ as follows.

$$(\boldsymbol{X}\mid\boldsymbol{Z}\mid\boldsymbol{R})=\left(\begin{array}{ccc|ccc|c}x_{11}&\cdots&x_{1n}&z_{11}&\cdots&z_{1n}&r_1\\\vdots&\ddots&\vdots&\vdots&\ddots&\vdots&\vdots\\x_{n1}&\cdots&x_{nn}&z_{n1}&\cdots&z_{nn}&r_n\end{array}\right)$$

The i-th row of the stabilizer tableau corresponds to a stabilizer generator P_i, where the bit-pairs $x_{ij}z_{ij} = 00, 10, 01, 11$ denote the j-th Pauli matrix on the j-th qubit: 00 means I, 10 means X, 01 means Z and 11 means Y; the bit $r_i = 0$ or 1 for positive or negative phase, respectively.

The updates corresponding to Clifford gates H, S, $CNOT$ require only $O(n)$ time. For example, an H gate on qubit a will set $r_i := r_i \oplus x_{ia}z_{ia}$ and swap x_{ia} with z_{ia} for all $i\in\{1,\ldots,n\}$, which matches the conjugation by H to Pauli matrices: $HXH^\dagger = Z, HZH^\dagger = X, HYH^\dagger = -Y$. However, the updates corresponding to computational basis measurements take $O(n^3)$ time in practice [2], which is in polynomial time but does not scale well enough.

The improved tableau algorithm. To improve the complexity of computational basis measurements in tableau simulation, Aaronson and Gottesman [2] (A-G) introduced *destabilizer generators* to stabilizer tableau as follows.

$$\left(\begin{array}{c|c|c}\bar{\boldsymbol{X}}&\bar{\boldsymbol{Z}}&\bar{\boldsymbol{R}}\\\hline\boldsymbol{X}&\boldsymbol{Z}&\boldsymbol{R}\end{array}\right)=\left(\begin{array}{ccc|ccc|c}\bar{x}_{11}&\cdots&\bar{x}_{1n}&\bar{z}_{11}&\cdots&\bar{z}_{1n}&\bar{r}_1\\\vdots&\ddots&\vdots&\vdots&\ddots&\vdots&\vdots\\\bar{x}_{n1}&\cdots&\bar{x}_{nn}&\bar{z}_{n1}&\cdots&\bar{z}_{nn}&\bar{r}_n\\\hline x_{11}&\cdots&x_{1n}&z_{11}&\cdots&z_{1n}&r_1\\\vdots&\ddots&\vdots&\vdots&\ddots&\vdots&\vdots\\x_{n1}&\cdots&x_{nn}&z_{n1}&\cdots&z_{nn}&r_n\end{array}\right)\tag{1}$$

The upper half of the tableau $\left(\bar{\boldsymbol{X}} \mid \bar{\boldsymbol{Z}} \mid \bar{\boldsymbol{R}}\right)$ represents n destabilizer generators $\bar{P}_1, \ldots, \bar{P}_n$ such that $\bar{P}_i$ anticommutes with P_i and commutes with P_j for $j \neq i$. With the help of destabilizer generators, the updates corresponding to computational basis measurements can be realized by a series of row operations (multiply two Pauli strings). Then the complexity of computational basis measurements is reduced to $O(n^2)$ time.

3 Phase Symbolization for Stabilizer Tableau

To speed up the simulation of stabilizer circuits, we introduce phase symbolization, which is based on the following key facts.

Fact 1. *The Pauli gates X, Y, Z only affect the phase part $\boldsymbol{R}, \bar{\boldsymbol{R}}$ of the stabilizer tableau. Specifically, for all $i \in \{1, \ldots, n\}$,*

- *X gate on qubit a: set $r_i := r_i \oplus z_{ia}, \bar{r}_i := \bar{r}_i \oplus \bar{z}_{ia}$;*
- *Y gate on qubit a: set $r_i := r_i \oplus x_{ia} \oplus z_{ia}, \bar{r}_i := \bar{r}_i \oplus \bar{x}_{ia} \oplus \bar{z}_{ia}$;*
- *Z gate on qubit a: set $r_i := r_i \oplus x_{ia}, \bar{r}_i := \bar{r}_i \oplus \bar{x}_{ia}$.*

Fact 2. *The control flow of A-G's algorithm is independent of the values of $\boldsymbol{R}, \bar{\boldsymbol{R}}$, i.e., all branches in this algorithm are determined by $\boldsymbol{X}, \boldsymbol{Z}, \bar{\boldsymbol{X}}, \bar{\boldsymbol{Z}}$ of the tableau. The values of $\boldsymbol{R}, \bar{\boldsymbol{R}}$ can only affect the outcomes of measurements.*

Combining Facts 1 and 2, whether a Pauli gate is applied to a qubit will be reflected in whether some rows of $\boldsymbol{R}$ and $\bar{\boldsymbol{R}}$ are flipped; hence, it will decide whether the later measurement outcomes are flipped. This phenomenon is also formalized into the *Pauli frame propagation* [17] in Stim [11]: To simulate a stabilizer circuit with Pauli faults, we first generate the noiseless measurement outcomes and then use the Pauli frame, a Pauli string that propagates on the circuit, to track the difference between the noiseless state and a sampled noisy state. This Pauli frame allows us to sample which measurements should be flipped by the noises. Since tracking the Pauli frame requires maintaining only one Pauli string, the subsequent sampling process takes $O(1)$ time per gate and measurement.

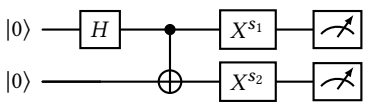
However, the Pauli frame propagation needs to go through the circuit for each sampling. In contrast, based on Facts 1 and 2, *we can identify which measurements in the circuit are affected by the preceding Pauli faults (and Pauli gates) and may need to be flipped.* Thus, we propose the symbolic phases to capture the flipping relationship.

3.1 Symbolic Phases

Instead of assigning specific values to the elements in $\boldsymbol{R}$ and $\bar{\boldsymbol{R}}$ of stabilizer tableau, we use symbolic expressions to represent them and call them symbolic phases. The stabilizer tableau becomes:

$$\left(\begin{array}{c|c|c} \bar{\boldsymbol{X}} & \bar{\boldsymbol{Z}} & \bar{\boldsymbol{R}} \\ \hline \boldsymbol{X} & \boldsymbol{Z} & \boldsymbol{R} \end{array}\right) = \left(\begin{array}{ccc|ccc|c} \bar{x}_{11} & \cdots & \bar{x}_{1n} & \bar{z}_{11} & \cdots & \bar{z}_{1n} & \bar{S}_1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \bar{x}_{n1} & \cdots & \bar{x}_{nn} & \bar{z}_{n1} & \cdots & \bar{z}_{nn} & \bar{S}_n \\ \hline x_{11} & \cdots & x_{1n} & z_{11} & \cdots & z_{1n} & S_1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{n1} & \cdots & x_{nn} & z_{n1} & \cdots & z_{nn} & S_n \end{array}\right) \tag{2}$$

where $S_i, \bar{S}_i$ are symbolic expressions over bit-symbols and bit-values with operator $\oplus$. For a better understanding, let us consider the following simple example circuit:



where s_1, s_2 are two bit-symbols indicating whether or not to apply the X gate. X^{s_1} and X^{s_2} characterize the possible behaviors of X-error on a single qubit. By A-G's algorithm, the stabilizer tableau for this circuit evolves as follows:

$$\left(\begin{array}{cc|cc|c} 1&0&0&0&0\\ 0&1&0&0&0\\ \hline 0&0&1&0&0\\ 0&0&0&1&0 \end{array}\right) \xrightarrow{H_1} \left(\begin{array}{cc|cc|c} 0&0&1&0&0\\ 0&1&0&0&0\\ \hline 1&0&0&0&0\\ 0&0&0&1&0 \end{array}\right) \xrightarrow{CNOT_{1,2}} \left(\begin{array}{cc|cc|c} 0&0&1&0&0\\ 0&1&0&0&0\\ \hline 1&1&0&0&0\\ 0&0&1&1&0 \end{array}\right) \xrightarrow{X_1^{s_1}} \left(\begin{array}{cc|cc|c} 0&0&1&0&s_1\\ 0&1&0&0&0\\ \hline 1&1&0&0&0\\ 0&0&1&1&s_1 \end{array}\right)$$

$$\xrightarrow{X_2^{s_2}} \left(\begin{array}{cc|cc|c} 0&0&1&0&s_1\\ 0&1&0&0&0\\ \hline 1&1&0&0&0\\ 0&0&1&1&s_1\oplus s_2 \end{array}\right) \xrightarrow{M_1} \left(\begin{array}{cc|cc|c} 1&1&0&0&0\\ 0&1&0&0&0\\ \hline 0&0&1&0&s_3\\ 0&0&1&1&s_1\oplus s_2 \end{array}\right) \xrightarrow{M_2} \left(\begin{array}{cc|cc|c} 1&1&0&0&0\\ 0&1&0&0&0\\ \hline 0&0&1&0&s_3\\ 0&0&1&1&s_1\oplus s_2 \end{array}\right)$$

$m_1 = s_3$ $\qquad$ $m_2 = s_1 \oplus s_2 \oplus s_3$

- The measurement on the first qubit has random outcomes; thus, we introduce a new bit-symbol s_3 to indicate outcome $m_1 = s_3$; this s_3 is kept in the tableau and used by future operations.
- After measuring the first qubit, we can find that the measurement on the second qubit is determined, and it results in an outcome $m_2 = s_1 \oplus s_2 \oplus s_3$

With the symbolic expressions $m_1 = s_1, m_2 = s_1 \oplus s_2 \oplus s_3$, we can sample concrete values of s_1, s_2, s_3 and substitute them in expressions to obtain samples of measurement outcomes. These symbols fall into two categories:

- Symbols induced by random measurements, e.g., the symbol s_3 above, are sampled to 0 (or 1) with probabilities 1/2 (or 1/2).
- Symbols induced by Pauli faults are sampled specifically according to the probability of the occurrence of Pauli strings in the Pauli faults. For example, a single-qubit X-error $\mathcal{E}(\rho) = (1-p)\rho + pX\rho X$ with parameter p corresponds to X^s and the bit-symbol s will be sampled to 0 and 1 with probabilities $1-p$ and p, respectively; a single-qubit depolarization $\mathcal{D}(\rho) = (1-p)\rho + \frac{p}{3}X\rho X + \frac{p}{3}Y\rho Y + \frac{p}{3}Z\rho Z$ corresponds to $X^{s_1}Z^{s_2}$ and the bit-symbols $s_1 s_2$ will be sampled to 00, 10, 01 and 11 with probabilities $1-p$, $p/3$, $p/3$ and $p/3$, respectively.

For general stabilizer circuits with Pauli faults, the introduction of symbolic phases will turn all the outcomes of measurements into symbolic expressions. Sampling the measurement outcomes becomes substituting the symbols according to probability and evaluating the symbolic expressions. This approach avoids the cost of repeatedly traversing the circuit like Pauli frame propagation.

3.2 Tableau Algorithm with Symbolic Phases

Now that we have introduced symbolic phases, let us see how to maintain them efficiently during the simulation process and how to speed up the sampling of stabilizer circuits with more details.

3.2.1 Representing symbolic expressions with bit-vectors. Since the symbolic expressions here only involve bit-symbols and operator $\oplus$, we can use bit-vectors to represent them. Considering that the circuit will introduce at most n_s symbols, we represent each bit-symbol $s_j, 1 \le j \le n_s$, with a bit-vector $\boldsymbol{s}_j$:

$$s_j \mapsto \boldsymbol{s}_j = \begin{pmatrix} \delta_{0,j} & \delta_{1,j} & \cdots & \delta_{n_s,j} \end{pmatrix} \in \mathbb{F}_2^{n_s+1},$$

where $\delta_{i,j} = 1$ if $i = j$ and $\delta_{i,j} = 0$ if $i \neq j$. In particular, we add a symbol s_0 to represent the constant 1. Then a symbolic expression $S = s_{j_1} \oplus s_{j_2} \oplus \cdots \oplus s_{j_k}, 0 \le j_1 \le \cdots \le j_k \le n_s$, is represented by a bit-vector $\boldsymbol{S}$:

$$S \mapsto \boldsymbol{S} = \boldsymbol{s}_{j_1} + \boldsymbol{s}_{j_2} + \cdots + \boldsymbol{s}_{j_k} \in \mathbb{F}_2^{n_s+1},$$

where + is the addition operator in $\mathbb{F}_2^{n_s+1}$, or it can be referred to the bitwise XOR. Therefore, the symbolic phases $\bar{S}_j, S_j$ in Eq. (2) are represented by bit-vectors $\boldsymbol{S}_j, \bar{\boldsymbol{S}}_j$:

$$\bar{S}_j \mapsto \bar{\boldsymbol{S}}_j = \begin{pmatrix} \bar{s}_{j,0} & \bar{s}_{j,1} & \cdots & \bar{s}_{j,n_s} \end{pmatrix} \in \mathbb{F}_2^{n_s+1},$$
$$S_j \mapsto \boldsymbol{S}_j = \begin{pmatrix} s_{j,0} & s_{j,1} & \cdots & s_{j,n_s} \end{pmatrix} \in \mathbb{F}_2^{n_s+1}.$$

And each measurement outcome m_k, which is also a symbolic expression, is represented by a bit-vector $\boldsymbol{m}_k$:

$$m_k \mapsto \boldsymbol{m}_k = \begin{pmatrix} m_{k,0} & m_{k,1} & \cdots & m_{k,n_s} \end{pmatrix} \in \mathbb{F}_2^{n_s+1}.$$

3.2.2 Extending A-G's algorithm to stabilizer tableau with symbolic phases. With the above representation, the stabilizer tableau with symbolic phases becomes a $2n \times (2n+n_s+1)$ bit-matrix:

$$\left(\begin{array}{ccc|ccc|ccc|c} \bar{x}_{11} & \cdots & \bar{x}_{1n} & \bar{z}_{11} & \cdots & \bar{z}_{1n} & \bar{s}_{1,0} & \bar{s}_{1,1} & \cdots & \bar{s}_{1,n_s} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \bar{x}_{n1} & \cdots & \bar{x}_{nn} & \bar{z}_{n1} & \cdots & \bar{z}_{nn} & \bar{s}_{n,0} & \bar{s}_{n,1} & \cdots & \bar{s}_{n,n_s} \\ \hline x_{11} & \cdots & x_{1n} & z_{11} & \cdots & z_{1n} & s_{1,0} & s_{1,1} & \cdots & s_{1,n_s} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nn} & z_{n1} & \cdots & z_{nn} & s_{n,0} & s_{n,1} & \cdots & s_{n,n_s} \end{array}\right) \tag{3}$$

The first $2n+1$ columns of Eq. (3) are the same as the original stabilizer tableau (see Eq. (1)) in A-G's algorithm [2]. We can extend A-G's algorithm to Eq. (3) as follows.

(**Init-C**) For Clifford gates, we update the first $2n+1$ columns of Eq. (3) as A-G's algorithm;

(**Init-P**) For Pauli faults, we first decompose them into some X^{s_j} and Z^{s_k}. Then, for X^{s_j} (Z^{s_k}), we treat the first $2n$ columns of Eq. (3) together with the j-th (k-th) column as a stabilizer tableau in A-G's algorithm and update it by X (Z) gate as A-G's algorithm.

(**Init-M**) For computational basis measurements, we update the first $2n+1$ columns of Eq. (3) as A-G's algorithm: when it comes to adding a phase $s_{j,0}$ to another phase $s_{k,0}$ for some j,k, we also add the remaining $s_{j,1},\ldots,s_{j,n_s}$ to $s_{k,1},\ldots,s_{k,n_s}$, respectively.

- If the measurement outcome is random, we fix it to 0 and apply an X^s at the measured qubit, where s is a bit-symbol with sampling probabilities of $1/2$ and $1/2$ for 0 and 1, respectively. Then, we record the bit-vector $\boldsymbol{s} \in \mathbb{F}_2^{n_s+1}$ for the symbol s as this measurement outcome.
- If the measurement outcome is determined, the measurement outcome output by A-G's algorithm is a summation over phases of some rows of stabilizer generators: $s_{j_1,0} \oplus s_{j_2,0} \oplus \cdots \oplus s_{j_k,0}$. Since we also track addition operations for the remaining n_s elements of each row, we record the bit-vector $\boldsymbol{S}_{j_1} + \boldsymbol{S}_{j_2} + \cdots + \boldsymbol{S}_{j_k} \in \mathbb{F}_2^{n_s+1}$ as this measurement outcome.

3.2.3 Sampling measurement outcomes as matrix multiplication. After traversing the circuit by using (**Init-C**), (**Init-P**) and (**Init-M**), we will get an array of bit-vectors $\boldsymbol{m}_1, \boldsymbol{m}_2, \ldots, \boldsymbol{m}_{n_m} \in \mathbb{F}_2^{n_2+1}$ representing the measurement outcomes. For all bit-symbols $s_1,\ldots,s_{n_s}$, we sample a bit-vector $\boldsymbol{b} = \begin{pmatrix} b_0 & b_1 & \cdots & b_{n_s} \end{pmatrix} \in \mathbb{F}_2^{n_s+1}$ as mentioned in Section 3.1, where $b_j, 1 \le j \le n_s$, is the sampled bit-value for the bit-symbol s_j and the first entry of $b_0 = 1$ is generated for the constant symbol s_0. Then, the sampled measurement outcome for $\boldsymbol{m}_j$ is $\boldsymbol{m}_j \boldsymbol{b}^\intercal = \sum_{k=0}^{n_s} m_{j,k} b_k \in \mathbb{F}_2$.

Further, if we want to generate n_{smp} samples of measurements outcomes for $\boldsymbol{m}_1, \boldsymbol{m}_2, \ldots, \boldsymbol{m}_{n_m}$, we can first generate n_{smp} bit-vectors $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{n_{\text{smp}}}$, then obtain samples of measurements outcomes as matrix multiplication:

$$\boldsymbol{M}_{\text{samples}} = \begin{pmatrix} \boldsymbol{m}_1^\intercal & \boldsymbol{m}_2^\intercal & \cdots & \boldsymbol{m}_{n_m}^\intercal \end{pmatrix}^\intercal \cdot \begin{pmatrix} \boldsymbol{b}_1^\intercal & \boldsymbol{b}_2^\intercal & \cdots & \boldsymbol{b}_{n_{\text{smp}}}^\intercal \end{pmatrix} \in \mathbb{F}_2^{n_m \times n_{\text{smp}}}, \tag{4}$$

where the j-th column of $\boldsymbol{M}_{\text{samples}}$ is the j-th sample of measurement outcomes.

3.2.4 Our algorithm. We present the Algorithm 1 based on previous discussions in this section. The algorithm takes a noisy stabilizer circuit C, a noise model $\mathbb{P}_C$ for C's Pauil noises, and an integer n_{smp} for the number of the measurement outcomes' samples as inputs.

Algorithm 1: Tableau Algorithm with Symbolic Phases.

Input: A noisy stabilizer circuit C, a noise model $\mathbb{P}_C$ for C's Pauil noises, an integer n_{smp}.
Output: n_{smp} samples of all the measurements in the circuit C.

1 $\boldsymbol{m}_1, \boldsymbol{m}_2, \ldots, \boldsymbol{m}_{n_m} \leftarrow$ `Initialization`(C);
2 $\boldsymbol{M}_{\text{samples}} \leftarrow$ `Sampling`$(n_{\text{smp}}, \mathbb{P}_C, \boldsymbol{m}_1, \boldsymbol{m}_2, \ldots, \boldsymbol{m}_{n_m})$;
3 **return** $\boldsymbol{M}_{\text{samples}}$;
1 **Procedure** `Initialization`(C)
2 Traverse circuit C by using (**Init-C**), (**Init-P**) and (**Init-M**) to obtain $\boldsymbol{m}_1, \boldsymbol{m}_2, \ldots, \boldsymbol{m}_{n_m}$;
3 **return** $\boldsymbol{m}_1, \boldsymbol{m}_2, \ldots, \boldsymbol{m}_{n_m}$;
1 **Procedure** `Sampling`$(n_{\text{sample}}, \mathbb{P}_C, \boldsymbol{m}_1, \boldsymbol{m}_2, \ldots, \boldsymbol{m}_{n_m})$
2 Sample $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{n_{\text{smp}}}$ from $\mathbb{P}_C$;
3 $\boldsymbol{M}_{\text{samples}} \leftarrow \begin{pmatrix} \boldsymbol{m}_1^\intercal & \boldsymbol{m}_2^\intercal & \cdots & \boldsymbol{m}_{n_m}^\intercal \end{pmatrix}^\intercal \cdot \begin{pmatrix} \boldsymbol{b}_1^\intercal & \boldsymbol{b}_2^\intercal & \cdots & \boldsymbol{b}_{\text{smp}}^\intercal \end{pmatrix}$;
4 **return** $\boldsymbol{M}_{\text{samples}}$;

Consider an n-qubit stabilizer circuit C contains n_g single-qubit and two-qubit gates, n_m computational basis measurements, and n_p single-qubit Pauli faults[1]. The cost of Algorithm 1 is divided into two parts:

- `Initialization`: (**Init-C**) has a cost of $O(n)$ for each gate, thus it takes $O(nn_g)$ time; (**Init-P**) has a cost of $O(n)$ for each single-qubit Pauli fault, thus it takes $O(nn_p)$ time; n_m measurements and n_p single-qubit Pauli faults introduce at most $n_m + n_p + 1$ bit-symbols, then the number of columns in Eq. (3) is at most $2n + n_m + n_p + 1$; (**Init-M**) has a cost of $O(n(n+n_m+n+p))$ for each measurement, thus it takes $O(nn_m(n+n_m+n_p))$ time. The total cost of `Initialization` is $O(nn_g + nn_p + nn_m(n+n_m+n_p))$. Since the cost of A-G's algorithm (without Pauli faults) is $O(nn_g+n^2n_m)$, we write the cost of `Initialization` as $O(nn_g+n^2n_m) + O(nn_m(n_m+n_p))$.
- `Sampling`: We only consider the cost of the line 3 in it[2]. The number of bit-symbols is at most n_m+n_p+1, then the cost is lower than the cost of multiplying a bit-matrix of size $n_m \times (n_m+n_p+1)$ by another bit-matrix of size $(n_m+n_p+1) \times n_{\text{smp}}$, which is $O(n_{\text{smp}} n_m (n_m+n_p))$.

We compare the complexity of Algorithm 1 with the algorithm (Pauli frame propagation [17]) used by Stim [11] and the recent ABC sim. algorithm [8] in Table 1. They differ as follows:

- Compared to Stim's, ABC sim. and our Algorithm 1 incur extra costs of $O(n_m(n_g+n_p))$ and $O(nn_m(n_m+n_p))$, respectively, for the `Initialization`. However, *the overhead of Algorithm 1 does*

[1] All Pauli faults can be decomposed into single-qubit Pauli faults.
[2] We do not take account the cost of sampling $\boldsymbol{b}_j$ because this cost is related to the noise model and will be *the same for different algorithms.*

not depend on the number of gates (n_g), while ABC sim.'s contains n_g. Thus, our Algorithm 1 is favorable when n_g is large.

- For `Sampling`, both ABC sim. and Algorithm 1 do not depend on n_g, thus ABC sim. and Algorithm 1 are improvements over Stim [11, 17]. However, ABC sim. and Algorithm 1 have an additional multiplication factor $O(n_m + n_p)$ resulting from matrix multiplication; for the case of sparse circuits, i.e., each measurement outcome is related to a small number of Pauli noises, $(\boldsymbol{m}_1^\intercal\ \boldsymbol{m}_2^\intercal\ \cdots\ \boldsymbol{m}_{nm}^\intercal)^\intercal$ is a column-sparse matrix, then the cost is reduced to $O(n_{\text{smp}} n_m)$.

Table 1: Complexity comparison of various algorithms for simulating stabilizer circuits. Our Algorithm 1 is advantageous when the circuits have a large number of quantum gates (n_g).

Algorithm	`Initialization`	`Sampling` ‡
Stim's [11]	$O(nn_g + n^2 n_m)$	$O(n_{\text{smp}}(n_g + n_m + n_p))$
ABC sim. [8]	$O(nn_g + n^2 n_m)$ ¶ $+ O(n_m(n_g + n_p))$	$O(n_{\text{smp}} n_m (n_m + n_p))$ *
Algorithm 1	$O(nn_g + n^2 n_m)$ $+ O(nn_m(n_m + n_p))$	$O(n_{\text{smp}} n_m (n_m + n_p))$ *

n: number of qubits, n_g: number of gates, n_m: number of measurements, n_p: number of single-qubit Pauli noises, n_{smp}: number of samples.
*: $O(n_{\text{smp}} n_m)$ for sparse circuits.
‡: The cost of sampling noises from $\mathbb{P}_C$ is not included because it is the same for all algorithms.
¶: ABC sim. obtains the flipping relationship between measurements and Pauli noises and does not obtain the measurement outcomes without noises. Thus, we should include this term for ABC sim.

4 Data Layout for Implementation

Although there are theoretically efficient algorithms (see Table 1), they still face some practical issues and challenges in implementing them for real applications. We next discuss the data layout of the stabilizer tableau for implementation.

In the implementation `chp.c` [1] accompanying with A-G's algorithm [2], the bits were packed into unsigned integers in memory as shown in Fig. 2a, where an `UInt-32` integer is interpreted as a 1×32 bit-matrix. Thus, the $n \times \lceil n/32 \rceil$ integer-matrix in Fig. 2a can be interpreted as an $n \times n$ bit-matrix. It offers a compact representation of the stabilizer tableau. When stored in *row-major order*, it also provides acceleration when we perform row operations for measurements because the rows are contiguous in memory. However, for quantum gates, which require column operations, the data layout in Fig. 2a is not friendly.

To balance the effects of data layout on row and column operations, Stim [11] interprets `UInt-64` integers as 8×8 bit-matrices and places them in column-major order as in Fig. 2b. This layout with column-major order is friendly to column operations so that quantum gates can be performed quickly. For measurements, especially a series of measurements, we can transpose it to row-major order temporarily and do a series of measurements before transposing it back for later quantum gates. Moreover, the contiguous memory in Stim enables the application of SIMD (*Single Instruction, Multiple Data*) operations, which can perform one instruction on multiple data elements (e.g., 256-bits/4×64-bits/8×32-bits) simultaneously.

Our data layout. Despite the high performance of Stim with its data layout (Fig. 2b), we observe that the transpose operation was time-consuming. Since the number of measurements in the circuits

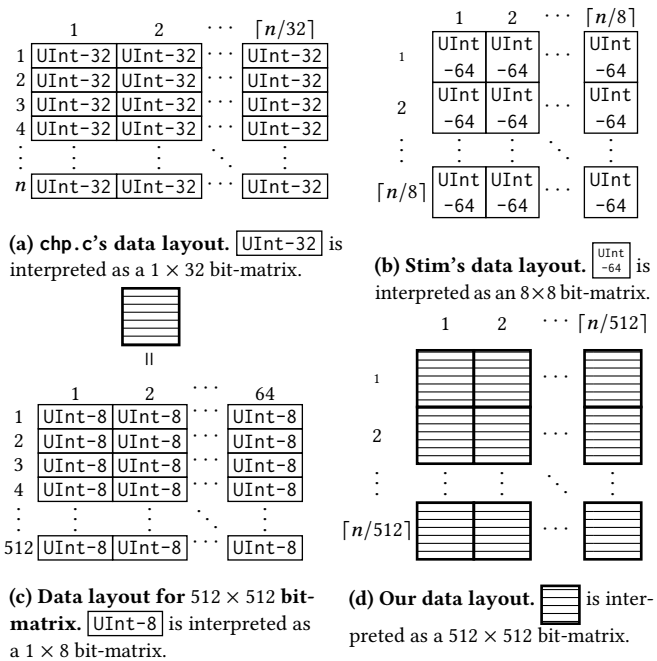


Figure 2: Data layout for stabilizer tableau.

is usually smaller than the number of gates, we adopt a new layout in Fig. 2d. Each shaded block contains an `UInt-8` matrix of size 512×64 in column-major order, which represents a 512×512 bit-matrix. This layout allows SIMD operations for doing column operations (gates).

For row operations (measurements), we only do local transpositions of shaded blocks (Fig. 2c). such local transpositions reduce the time required to transpose the entire bit-matrix. With local transpositions, Fig. 2c is in row-major order. For the entire bit-matrix, each row is not allocated continuously in memory but separated into groups of 512 bits. Although it prevents us from manipulating rows consecutively, the fixed length of 512 bits already provides sufficient speedup.

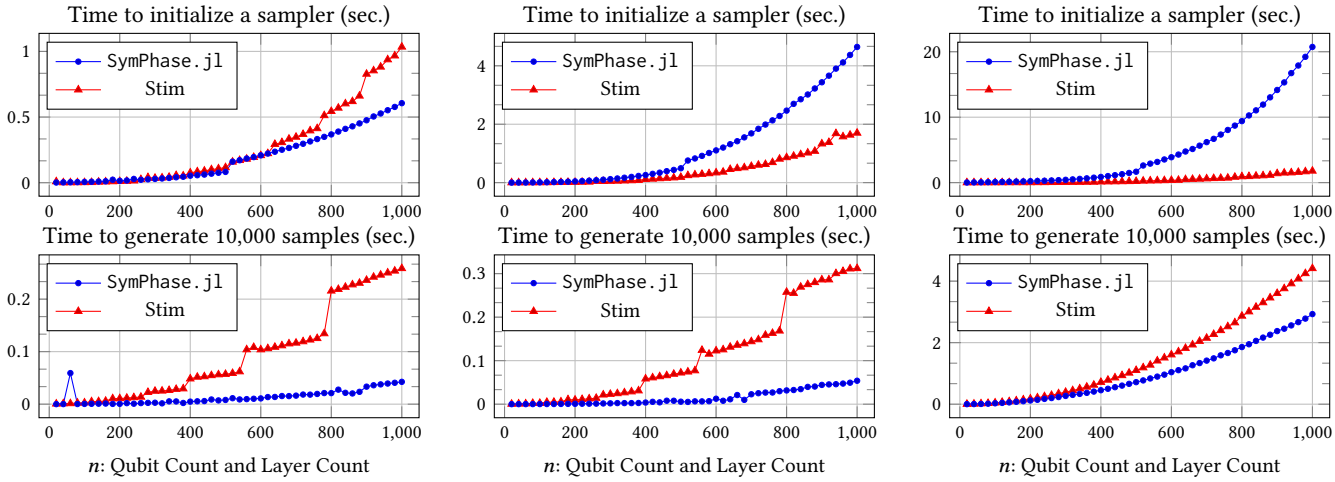
5 Evaluations

We have developed a Julia [5] package named `SymPhase.jl`[3] that implements Algorithm 1. `SymPhase.jl` uses the data layout shown in Fig. 2d for the `Initialization` and the sparse implementation of matrix multiplication for the `Sampling`. To demonstrate the efficiency of our Algorithm 1 in sampling results of stabilizer circuits and to evaluate the performance of `SymPhase.jl`, we chose to compare it with the state-of-the-art stabilizer simulator.

Baseline. The state-of-the-art stabilizer simulator known to us is Stim [11], which has not only surpassed popular simulators such as Qiskit's stabilizer method [16], Cirq's Clifford simulator [7], Aaronson and Gottesman [2]'s `chp.c` and GraphSim [4] in performance, but is also being actively developed[4].

Benchmark. We selected three classes of randomly generated circuits for the benchmark, which are variants of the benchmark used in Stim [11]. This way, we can avoid the influence of circuit

[3]See https://github.com/njuwfang/SymPhase.jl.
[4]See https://github.com/quantumlib/Stim.



(a) Each layer randomly selects 5 pairs (1 pair in Stim's benchmark) of qubits to apply *CNOT* gates.

(b) Each layer randomly selects $\lfloor \frac{n}{2} \rfloor$ pairs of qubits to apply *CNOT* gates.

(c) Same as Fig. 3b, but each layer additionally applies single-qubit depolarize noise to each qubit.

Figure 3: Performance results of sampling layered random interaction circuits. Each circuit is made up of n qubits with n layers. Each layer randomly applies an H, S and I gate to each qubit, then applies *CNOT* gates, then samples 5% of the qubits to measure in the computational basis. At the end of the circuit, each qubit is measured in the computational basis. The maximum size of the circuit in the experiment reaches 1,000 qubits, 1160,000 quantum gates, 2000,000 Pauli faults, and 51,000 measurements.

structures on the comparison results. For example, circuits for LDPC codes [6] are sparse, which gives us an advantage. The detailed descriptions are given in the captions of Figs. 3 and 3a to 3c.

Environment. All our experiments are carried out on a desktop with Intel(R) Core(TM) i7-9700 CPU@3.00GHz and 16G of RAM, running Ubuntu 22.04.2 LTS. The version of Stim is 1.12.0.

Result. The experimental results are shown in Figs. 3a to 3c. We report the time for Stim and `SymPhase.jl` to initialize a sampler (i.e., the time to analyze the input circuit and create a sampler for generating the measurement results) and the time for Stim's and `SymPhase.jl`'s samplers to generate 10,000 samples of measurement results. `SymPhase.jl` outperforms Stim in all benchmarks in terms of the sampling time, which validates the advantages of our algorithm (see Algorithm 1) and our package (`SymPhase.jl`) for sampling stabilizer circuits. On the other hand, our algorithm has an overhead for symbolic phases, which makes `SymPhase.jl` consume more time than Stim in initializing samplers. However, this overhead is one-time, and the performance of the sampler is crucial for generating a large number of samples for further analysis. Moreover, we also observe that in Fig. 3a, `SymPhase.jl` has a better initialization time than Stim, which indicates that our data layout has benefits in certain situations. This is worth further investigation.

6 Conclusion

We have presented phase symbolization for fast simulation of stabilizer circuits without traversing the circuit repeatedly. With a new layout of the stabilizer tableau, a package `SymPhase.jl` is implemented, which has been experimentally evaluated that it surpasses the existing state-of-the-art tool in sampling rate. Our techniques can provide a useful tool for simulating and analyzing stabilizer circuits, especially for fault-tolerant quantum computing.

Acknowledgments

We thank Kean Chen for insightful discussions and acknowledge the National Key R&D Program of China Grant No. 2023YFA1009403.

References

[1] Scott Aaronson. 2004. `chp.c`. https://www.scottaaronson.com/chp/chp.c.
[2] Scott Aaronson and Daniel Gottesman. 2004. Improved simulation of stabilizer circuits. *Phys. Rev. A* 70 (Nov 2004), 052328. Issue 5. https://doi.org/10.1103/PhysRevA.70.052328
[3] Google Quantum AI. 2023. Suppressing quantum errors by scaling a surface code logical qubit. *Nature* 614, 7949 (01 Feb 2023), 676–681. https://doi.org/10.1038/s41586-022-05434-1
[4] Simon Anders and Hans J. Briegel. 2006. Fast simulation of stabilizer circuits using a graph-state representation. *Phys. Rev. A* 73 (Feb 2006), 022334. Issue 2. https://doi.org/10.1103/PhysRevA.73.022334
[5] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671
[6] Nikolas P. Breuckmann and Jens Niklas Eberhardt. 2021. Quantum Low-Density Parity-Check Codes. *PRX Quantum* 2 (Oct 2021), 040101. Issue 4. https://doi.org/10.1103/PRXQuantum.2.040101
[7] Cirq Developers. 2018. Cirq. https://doi.org/10.5281/zenodo.4062499
[8] Nicolas Delfosse and Adam Paetznick. 2023. Simulation of noisy Clifford circuits without fault propagation. arXiv:2309.15345 [quant-ph]
[9] Qian Xu et al. 2023. Constant-Overhead Fault-Tolerant Quantum Computation with Reconfigurable Atom Arrays. arXiv:2308.08648 [quant-ph]
[10] Sergey Bravyi et al. 2023. High-threshold and low-overhead fault-tolerant quantum memory. arXiv:2308.07915 [quant-ph]
[11] Craig Gidney. 2021. Stim: a fast stabilizer circuit simulator. *Quantum* 5 (July 2021), 497. https://doi.org/10.22331/q-2021-07-06-497
[12] Daniel Gottesman. 1997. *Stabilizer Codes and Quantum Error Correction.* Ph. D. Dissertation. California Institute of Technology. arXiv:quant-ph/9705052 [quant-ph]
[13] Daniel Gottesman. 1998. The Heisenberg Representation of Quantum Computers. *arXiv e-prints* (July 1998). https://doi.org/10.48550/arXiv.quant-ph/9807006
[14] Stefan Krastanov. 2019. https://quantumsavory.github.io/QuantumClifford.jl/dev/ Accessed on 2023-10-31.
[15] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press. https://doi.org/10.1017/CBO9780511976667
[16] Qiskit Community. 2017. Qiskit: An Open-Source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2562110
[17] Patrick Rall, Daniel Liang, Jeremy Cook, and William Kretschmer. 2019. Simulation of qubit quantum circuits via Pauli propagation. *Phys. Rev. A* 99 (Jun 2019), 062337. Issue 6. https://doi.org/10.1103/PhysRevA.99.062337