Tutorial

Pauli Propagation: A Computational Framework for Simulating Quantum Systems

Manuel S. Rudolph[1,2,*] Tyson Jones,[1,2] Yanting Teng[1,2] Armando Angrisani[1,2] and Zoë Holmes[1,2,3]

[1]*Institute of Physics, Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland*

[2]*Centre for Quantum Science and Engineering, Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland*

[3]*Algorithmiq Ltd. Kanavakatu 3 C, FI-00160 Helsinki, Finland*

(Received 13 January 2026; revised 16 April 2026; published 11 August 2026)

Classical methods to simulate quantum systems are not only a key element of the physicist's toolkit for studying many-body models but are also increasingly important for verifying and challenging upcoming quantum computers. Pauli propagation has recently emerged as a promising new family of classical algorithms for simulating digital quantum systems. Here we provide a comprehensive account of Pauli propagation, tracing its algorithmic structure from its bit-level implementation and formulation as a tree-search problem, all the way to its high-level user applications for simulating quantum circuits and dynamics. Utilizing these observations, we present `PauliPropagation.jl`, a `Julia` software package that can perform rapid Pauli propagation simulation straight out-of-the-box and can be used more generally as a building block for novel simulation algorithms.

DOI: 10.1103/6vd7-l9bn

I. INTRODUCTION

What makes a quantum system hard to simulate classically? This is a question that we cannot answer in this work. However, it is clear that not all quantum systems are intractable to simulate, and not all are simulable by the same computational frameworks. Tasks that are out of range of one approach are not precluded from study by another, perhaps bespoke, method. A ubiquitous and widely applicable framework for simulating quantum systems is the use of *tensor network methods* [1–5]. Such methods have been proven over again to be powerful and adaptable, and are still evolving, for example, for continually simulating quantum computations [6–8]. Yet despite their broad domains and popularity, other frameworks such as *neural quantum states* [9] can more effectively tackle some specific tasks of interest [10]. Indeed no single method is optimal in all regimes, and many challenging problems require drastically different approaches.

Pauli propagation is a relative newcomer to the corpus of classical simulators and yet is already competitive with other state-of-the-art methods for certain tasks [11–14]. At their core, Pauli propagation methods approximate the evolution of a quantum operator (typically, an observable in the Heisenberg picture) via a truncated Pauli path integral. While first shown to be powerful in noisy [13,15] and random [14] settings, the method has also been applied more broadly [16,17]. The approach looks particularly promising for quickly obtaining rough approximations of expectation values for quantum circuits on arbitrary topologies, and may find use in variational scenarios [14,18]. It is also well-suited for interfacing with quantum hardware, enabling the hybridization of classical and quantum computational power [17,19,20]. We refer to Fig. 1 for a schematic overview of the Pauli propagation framework.

Readers may be familiar with alternative names for Pauli propagation algorithms that reflect particular implementations, approximations, or goals. To the best of our knowledge, Ref. [21] first introduced the term "Pauli propagation," contrasting it with stabilizer simulation, and highlighting both Schrödinger and Heisenberg picture approaches. More recent implementations include *observable back-propagation on Pauli paths* (OBPPP) [13], the *low-weight efficient simulation algorithm* (LOWESA) [12,15], the *exclusive-or represented quantum algebra* (ORQA) [22,23], and *Clifford perturbation theory* (CPT) [24], which owe their names to specific implementation details and truncation strategies. One more general name is *sparse Pauli dynamics* (SPD) [11,16], which also falls under the framework of Pauli propagation with the specific goal of simulating real-time quantum dynamics. Very recently, a breakaway from the Pauli basis was proposed that works in the Majorana basis [25–28], but which shares most algorithmic features with Pauli propagation.

Most work so far on Pauli propagation has focused either on providing theoretical guarantees [14,15,29–31]

*Contact author: manuel.rudolph@epfl.ch

Published by the American Physical Society under the terms of the Creative Commons Attribution 4.0 International license. Further distribution of this work must maintain attribution to the author(s) and the published article's title, journal citation, and DOI.

or specific case studies [11,12,16,32]. Here we will take a step back and provide an end-to-end account of the algorithmic considerations underlying Pauli propagation methods. We start on the high-level with a discussion of the *mathematical foundations* of the approach and its applications. Next we proceed to the mid-level, where we discuss *algorithmic nuances* such as search and truncation strategies as well as methods to estimate simulation errors. Then we dive under the hood and discuss the *implementation details*; the nitty gritty of how to get your money's worth from your CPU (or GPU). Finally, we swoop back to the bigger picture, looking out to where Pauli propagation might find its niche and avenues for future developments.

To facilitate research *on* and *with* this framework, we provide an open-source library `PauliPropagation.jl` [33]. This high-level `Julia` package can be used straight out of the box to simulate large-scale quantum circuits. Through careful consideration of `Julia`'s multiple dispatching facilities, the package remains adaptable and highly extensible. We hope that the package will prove useful to users simulating quantum systems, developing quantum algorithms, finding novel applications of Pauli propagation, and devising new propagation methods.

II. A BIRD'S-EYE VIEW

A. Defining Pauli propagation

Propagation methods in their essential form compute the evolution of an object A under an operation $\mathcal{E}$ expressed as a sequence of smaller, *individually tractable* operations $\mathcal{E}_l, l = 1 \cdots m$,

$$\mathcal{E}[A] = \mathcal{E}_m[\cdots[\mathcal{E}_1[A]]]. \tag{1}$$

Often, but not always, we then compute the overlap between $\mathcal{E}[A]$ and some other object B to produce an output scalar,

$$\langle B, \mathcal{E}[A]\rangle \in \mathbb{C}. \tag{2}$$

In a typical statevector simulation, the object A would usually be an initial state, $\mathcal{E}$ a series of gates and channels, and B an observable operator of which the output expectation value is evaluated via a trace. By contrast, in "*Pauli*" propagation, this is generally reversed and A is an observable expressed in the Pauli basis and evolved by the adjoint circuit $\mathcal{E}$, where the $\mathcal{E}_l$ are maps between Pauli strings. The object B is typically the initial quantum state, and the Hilbert-Schmidt inner product is the trace $\mathrm{Tr}[B \cdot \mathcal{E}[A]]$. More concretely, the following (nonexhaustive) list of properties can in principle be computed using the primitive of Pauli propagation:

The evolution of observables: We can compute the Pauli basis decomposition of an observable O evolved under some channel $\mathcal{E}$, i.e., $\mathcal{E}[A] \equiv \mathcal{E}(O) = \sum_P c_P P$. Typically, we will work in the Heisenberg picture and so for unitary evolutions we compute $\mathcal{E}[O] \equiv U^\dagger O U$, or for a noise channel $\mathcal{N}$ we would compute the action of its adjoint $\mathcal{N}^\dagger[O]$. Evolution of *states* is also possible, i.e., we can also take $A = \rho$ and compute $U\rho U^\dagger$, but this is only efficient when the initial state is sparse in the Pauli basis, e.g., in the case of high-temperature states [34]. An extreme case is the maximally mixed state, which is representable by a single Pauli string, the all-identity $I/2^n$.

State expectation values: It is possible to compute $\mathrm{Tr}[\mathcal{E}[\rho]O]$ where ρ is a quantum state, O is an observable and $\mathcal{E}$ is a quantum channel as this is equivalent to applying the inverse of the channel on the observable. For example, in the case of unitary evolution, the evolved object will typically be the observable O and the operation $U^\dagger O U$ describes the unitary Heisenberg evolution of the system. We then compute the overlap of the propagated observable $U^\dagger O U$ with the state ρ, which can be very hard in general, but is trivial for many common states like the all-zero state or other stabilizer states.

Correlation functions in time: We can compute correlation functions like $\mathrm{Tr}[O_1 U O_2 U^\dagger]$ using the same approach as above, except the final overlap is taken with another (or the same) observable rather than a state ρ.

Variational optimization and machine learning: The code in `PauliPropagation.jl` is designed to be automatically differentiable by use of common `Julia` libraries. This unlocks variational optimization, for example, for training quantum machine learning models, which has been used to simulate quantum convolutional neural networks at large scale [32].

Hybridization with other classical/quantum tools: Simulations of a single quantum circuit can be hybridized between different types of hardware (quantum and classical) or simulation methods (classical and classical). For example, $\mathrm{Tr}[\rho U_2^\dagger U_1^\dagger O U_1 U_2] = \mathrm{Tr}[\mathcal{E}_2(\rho)\mathcal{E}_1(O)]$, where $\mathcal{E}_1(\cdots) = U_1^\dagger(\cdots)U_1$ and $\mathcal{E}_2(\cdots) = U_2(\cdots)U_2^\dagger$ denote the unitary evolution in the Schrödinger and Heisenberg picture, respectively. If $\mathcal{E}_2(O) = \sum_P c_P P$ is the half-backpropagated observable, the full expectation can be evaluated as $\sum_P c_P \mathrm{Tr}[\mathcal{E}_1(\rho)P]$. In this manner, one can trade shorter depth circuits for either simulation with the additional demands of computing the overlap of the two *half-evolved* objects $\mathcal{E}_2(\rho)$ and $\mathcal{E}_1(O)$.

For concreteness, we will focus on detailing the Pauli propagation framework for computing the evolution of observables in the Heisenberg picture. However, the reader should bare in mind that the framework can be applied more generally.

The "Pauli" propagation propagates *Pauli strings*. Namely, tensor products of n single qubit Pauli matrices I, X, Y, Z. These form a basis for all $2^n \times 2^n$ matrices. We write an object O in this Pauli basis as

$$O = \sum_{i=1}^{N_O} c_i P_i, \tag{3}$$

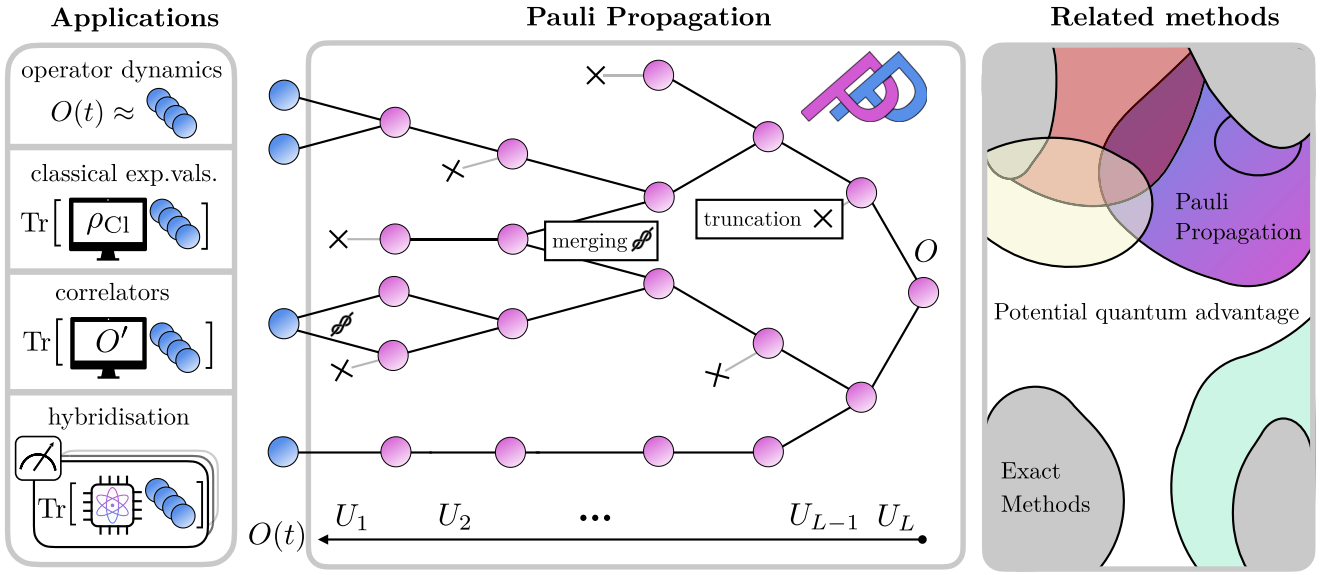


FIG. 1. Pauli propagation: applications and related methods. Quantum systems can be tackled via various classical simulations methods, each for different applications and systems of interest. *Pauli propagation* is a recent computational framework for computing the evolution of Pauli strings under quantum dynamics or arbitrary quantum circuits. Often but not necessarily, Pauli propagation involves Heisenberg-picture backpropagation of an observable. This evolved observable can be utilized for applications ranging from transparently studying operator dynamics, expectation values with states encoded on classical or quantum hardware, or operator correlation functions. Its placement in the classical simulation toolbox with other, more established methods is not fully known, but within few years, this approach has already shown promise. Pauli propagation is most closely related to stabilizer simulation and its extensions for general gates, but it is fundamentally different from, e.g., tensor network approaches. Our library `PauliPropagation.jl` provides state-of-the-art Pauli propagation techniques, and can be used for large-scale quantum simulation or as a building block in larger algorithmic frameworks.

where c_i are coefficients for the Pauli strings P_i, and N_O is the number of non-zero-coefficient Pauli strings in the Pauli decomposition of O.

The action of an operation $\mathcal{E}_l$ onto a Pauli string P can generally be written as

$$\mathcal{E}_l[P] = \sum_{j=1}^{N_{\mathcal{E}_l,P}} c_j^{\mathcal{E}_l} P_j, \tag{4}$$

where $c_j^{\mathcal{E}_l}$ are the evolved coefficients for the Pauli strings P_j, and $N_{\mathcal{E}_l,P}$ is the number of non-zero coefficient Pauli strings in the sum created from acting $\mathcal{E}_l$ onto Pauli P. We also call this number the *branching factor*, and we will say $\mathcal{E}_l$ is one-branching if it maps one Pauli string to one Pauli string (i.e., it does not branch), two-branching if it creates two Pauli strings from one, etc. In Sec. II B we will give examples of how an operator's action can be written in the Pauli basis for some commonly encountered operators. In Theory Box 1 we further discuss how the action of an operator on Pauli strings can be conveniently captured via the *Pauli transfer matrix* (PTM) formalism.

Algorithmically speaking, if the action in Eq. (4) can be defined for each $\mathcal{E}_l$, i.e., if it is known (or can be efficiently computed) which Pauli strings arise and with which coefficients, then Pauli propagation can be used to estimate $\mathcal{E}[O]$ via the sequential application of the operations to all arising Pauli strings. Concretely, we can write

$$\mathcal{E}[O] = \mathcal{E}_1[\cdots[\mathcal{E}_m[O]]] \tag{8}$$

$$= \mathcal{E}_1\left[\cdots\left[\mathcal{E}_m\left[\sum_{i=1}^{N_O} c_i P_i\right]\right]\right] \tag{9}$$

$$= \sum_{\alpha=1}^{N_{\mathcal{E},O}} c_\alpha^{\mathcal{E}} P_\alpha, \tag{10}$$

where the final line captures the fully evolved observable in the Pauli basis. One might wonder whether writing down this action is as hard as solving the entire simulation task. Fortunately, the answer is often "No." Typical gates are *local* acting only on k sites, for some small k, which means that the k-local gate can map one Pauli string only to at most 4^k Pauli strings. The mapping can be defined on those active sites, with all other Paulis in the Pauli string remaining unaffected.

We can then use the propagated operator in Eq. (10) to compute properties of interest. For example, to estimate observable expectation values one simply computes the

Theory Box 1: Pauli transfer matrices

In the Pauli transfer matrix (PTM) formalism [35] an n-qubit operator can be represented as a 4^n-dimensional vector, where the i'th entry is the coefficient of the corresponding *normalized* Pauli string $\mathsf{P}_i = 2^{-\frac{n}{2}} P_i$. That is, an operator O is represented as the vector $|O\rangle\rangle_i = \mathrm{Tr}[O\mathsf{P}_i]$, such that the Hilbert-Schmidt inner product can be expressed as $\mathrm{Tr}[A^\dagger B] = \langle\langle A|B\rangle\rangle$. Any operation $\mathcal{E}$ is then represented in the PTM formalism as $\boldsymbol{\mathcal{E}}$ where

$$[\boldsymbol{\mathcal{E}}]_{ij} = \mathrm{Tr}[\mathsf{P}_i \mathcal{E}(\mathsf{P}_j)]. \tag{5}$$

Equivalently, the PTM element $[\boldsymbol{\mathcal{E}}]_{ij}$ is (up to normalization) the coefficient c_i in Eq. (4) for the operation $\mathcal{E}$ applied to the Pauli string P_j. Thus the PTM framework formalizes the transformations an operation applies in the Pauli basis.

The expectation value of a state ρ evolved under an operation $\mathcal{E}$ with an operator O can be written as

$$\mathrm{Tr}[O\mathcal{E}[\rho]] = \langle\langle O|\boldsymbol{\mathcal{E}}|\rho\rangle\rangle. \tag{6}$$

Equivalently, we can consider the dual map $\mathcal{E}^\dagger$ evolving the operator O, such that $\mathrm{Tr}[\mathcal{E}^\dagger[O]\rho] = \mathrm{Tr}[O\mathcal{E}[\rho]]$:

$$\mathrm{Tr}[\mathcal{E}^\dagger[O]\rho] = \langle\langle \rho|\boldsymbol{\mathcal{E}}^\dagger|O\rangle\rangle. \tag{7}$$

The matrix $\boldsymbol{\mathcal{E}}^\dagger$ is the *transpose* of the matrix $\boldsymbol{\mathcal{E}}$, i.e. $[\boldsymbol{\mathcal{E}}^\dagger]_{ij} = [\boldsymbol{\mathcal{E}}]_{ji}$.

Software Box 1: Basics

The `PauliPropagation.jl` package can be used to compute or approximate objects represented and evolved in the Pauli basis, as well as to estimate expectation values with, e.g, quantum states. A circuit is a vector of gates, and the parameters of the parameterized gates (below `thetas`) are typically vectors of numbers. Importantly, both are defined in the order that they would act in the Schrödinger picture. This should be intuitive for users who are used to simulating quantum states. Our high-level type for Pauli strings is `PauliString`, which can be defined via its non-identity components on `nq` qubits. Here is a simple example code:

```
nq = 3
circuit = [
  PauliRotation(:X, 1),
  PauliRotation(:X, 2),
  CliffordGate(:CNOT, [2, 3])
]
thetas = [0.5, -1.2]
pstr = PauliString(nq, [:Z, :Z], [1, 3])
psum = propagate(circuit, pstr, thetas)
```

The output `psum` is a `PauliSum` object, our container for the propagating Pauli strings. As hinted at above, the circuit definition is in the Schrödinger picture, but `propagate()` applies the gates with its parameters in the **reverse order**, i.e., the Heisenberg picture. Finally, an expectation value with the all-zero initial state can be computed via `overlapwithzero(psum)`.
Note, all code examples in this manuscript are for the current version of the package (v0.7), and future changes may be introduced.

overlap between each of the Pauli strings in Eq. (10) with the initial state of interest. That is,

$$\mathrm{Tr}[\rho\mathcal{E}[O]] = \sum_{\alpha=1}^{N_{\mathcal{E},O}} c_\alpha^{\mathcal{E}} \mathrm{Tr}[\rho P_\alpha]. \tag{11}$$

In practice, estimating expectation values (i.e., inner products) with states is often drastically simpler than full state or operator evolutions. In Sec. III A, we outline several truncation strategies that can be employed to approximate the computation of Eq. (11) while not always faithfully approximating the object in Eq. (10).

B. Common gates and operations

In this section we detail the action of commonly encountered gates and noise channels in the Pauli basis by specifying the form Eq. (4) takes in each case.

Clifford Gates: These gates form a group defined by their action on Pauli strings. Each Clifford gate C transforms one Pauli string to another Pauli string, potentially itself, and potentially with a change in sign. That is, we have

$$C[P] = s_{C,P} \cdot P', \tag{12}$$

where $s_{C,P} \in \{\pm 1\}$ is a sign depending on the definition of C and its action on P, and P' is a Pauli string. This corresponds to conjugation by the corresponding unitary U_C of the Clifford gate in the computational basis, i.e., $U_C^\dagger P U_C$. Due to their one-branching (i.e., non-branching) nature, any circuit consisting of solely Clifford gates can be simulated in a time scaling polynomially in the number of gates.

Here are some examples of the H and CNOT Clifford gates acting upon Pauli strings in the Heisenberg picture

$$\begin{aligned}
\mathrm{H}[I] &= 1 \cdot I \\
\mathrm{H}[X] &= 1 \cdot Z \\
\mathrm{H}[Y] &= -1 \cdot Y \\
\mathrm{H}[Z] &= 1 \cdot X \\
\mathrm{CNOT}[ZX] &= 1 \cdot ZX \\
\mathrm{CNOT}[IZ] &= 1 \cdot ZZ \\
\mathrm{CNOT}[YX] &= 1 \cdot YI \\
\mathrm{CNOT}[XZ] &= -1 \cdot YY
\end{aligned}$$

Note, we here assume that the CNOT is controlled on the first (leftmost) qubit and we do not explicitly write the tensor product between the Pauli operators. That is, $ZX \equiv Z \otimes X \equiv Z_1 X_2$.

A few observations from these examples. The Hadamard H gate is its own inverse (applying it twice returns the original Pauli string) and transforms Z to X and vice versa. Even though the CNOT gate induces entanglement in states, it does not directly increase computational complexity. However, because it can increase (or decrease) the number of non-identity Paulis, it may make *subsequent* gate applications branch more.

1. Pauli rotations

Gates of the form $R_G(\theta) = e^{-i\theta G/2}$, where the generator G is an n-qubit Pauli string and θ is the angle of the rotation, are ubiquitous in quantum computing. They appear in many canonical circuits, including the Trotterization of Hamiltonian simulation. A Pauli rotation is always either one-branching (i.e., non-branching) or two-branching, irrespective of how many qubits the gate acts non-trivially upon. In particular, in the Heisenberg picture we have

$$R_G(\theta)[P] \coloneqq e^{i\theta G/2} P e^{-i\theta G/2} \tag{13}$$

such that for $\theta \in \mathbb{R}$

$$R_G(\theta)[P] = \begin{cases} P & \text{if } [P,G]=0 \\ \cos(\theta)P + \sin(\theta)P' & \text{else} \end{cases} \tag{14}$$

where $P' = i[G,P]/2$ is a new Pauli string.

Given the two-branching action of Pauli rotations, a circuit composed of m Pauli rotations can generate up to 2^m Pauli strings. However, in practice, substantially fewer terms will typically need to be computed for two reasons. First, a Pauli rotation causes branching only when the generator does not commute with the Pauli string operated on (as is for *half* of all strings) and commutations tend to dominate if O consists of *local* Pauli strings, i.e., ones that have few non-identity Paulis. Secondly, duplicate copies of the same Pauli string can be generated during propagation which can then be *merged* to reduce memory costs (see Sec. III B for more details). It is hard to overstate how much these two effects simplify the computation compared to the worst-case scaling. For an example of this effect see Theory Box 2.

Theory Box 2: Example of the importance of merging and commutativity

Consider the practical example of a two-qubit circuit $U = RX1(0.3)RX2(\pi/3)RZZ(-0.8)$ where the rightmost unitary is applied first to a hypothetical state in the Schrödinger picture. We take our final observable to be the two-qubit Pauli string $O = ZI$. Working in the Heisenberg picture, we first obtain

$$RX1(0.3)[O] = 0.96 \cdot ZI + 0.30 \cdot YI, \tag{15}$$

$$= O_3, \tag{16}$$

with coefficients shortened for display. Then the next rotation

$$RX2(\pi/3)[O_3] = 0.96 \cdot ZI + 0.30 \cdot YI, \tag{17}$$

$$= O_2 \tag{18}$$

leaves the Pauli sum unaffected because the $RX2$ gate commutes with both Pauli strings that carry the identity Pauli on the second site. And finally,

$$RZZ(-0.8)[O_2] = 0.96 \cdot ZI + 0.21 \cdot YI - 0.21 \cdot XZ, \tag{19}$$

$$= O_1, \tag{20}$$

where the ZI Pauli string commutes with the RZZ gate, $0.21 = 0.30 \cdot \cos(-0.8)$, and $0.21 = 0.30 \cdot \sin(-0.8)$. This example resulting in three Pauli strings in O_1 highlights that the expected $2^m = 8$ scaling is not representative of the true computational complexity of simulating circuits with m Pauli rotations.

T gates: T gates are special cases of Pauli-Z rotations with $\theta = \pi/4$. Thus it follows from Eq. (14) that if there is an X or Y at the location in the Pauli string on which T gates act then the operation is two-branching (and the new coefficients have equal magnitude), otherwise the gate leaves the Pauli string unchanged. Famously the worst-case simulation of a circuit composed only of Clifford and T gates is known to be exponential in the number of T gates, but not in the number of Clifford gates.

Pauli noise: Pauli noise is an umbrella term for *unital* quantum channels that are diagonal in the Pauli basis. This encompasses both homogeneous and inhomogeneous dephasing and depolarizing channels. Such noise multiplies the coefficients of some or all non-identity Paulis by a damping factor that is smaller than one. This does not change the Pauli strings themselves and so a Pauli noise

channel is non-branching. It is often sufficient to consider 1- and 2-local Pauli channels to capture passive and active (following two-qubit operations) error processes in quantum devices [31,36]. For example, the one-qubit inhomogeneous depolarizing channel, parameterized by the individual probabilities p_i of X, Y and Z errors upon the decohering qubit, modifies a density matrix ρ as

$$\mathcal{E}(\rho) = p_I\rho + p_x X\rho X + p_y Y\rho Y + p_z Z\rho Z, \tag{21}$$

where $p_I = 1-(p_x+p_y+p_z)$. Its corresponding PTM, agnostic to whether working in the Schrödinger or Heisenberg pictures, maps Pauli strings to

$$\mathcal{E}[I] = 1\cdot I, \qquad \mathcal{E}[Y] = (1-2p_x-2p_z)\cdot Y,$$
$$\mathcal{E}[X] = (1-2p_y-2p_z)\cdot X, \qquad \mathcal{E}[Z] = (1-2p_x-2p_y)\cdot Z.$$

Two-qubit depolarizing noise with uniform probability p scales all strings by $1-16p/15$, *except* for II which is unchanged. One (or two)-qubit dephasing (driving the qubits towards the z-axis of the Bloch sphere) scales all strings containing an X or Y by $1-2p$ (or $1-4p/3$).

Amplitude damping and arbitrary local noise: Moving beyond Pauli noise, any single-qubit channel $\mathcal{N}$ can be written in the Schrödinger picture as [37–39]

$$\mathcal{N}(\cdot) = U\mathcal{N}'(V(\cdot)V^\dagger)U^\dagger, \tag{22}$$

where U, V are arbitrary single-qubit unitaries and the action of $\mathcal{N}'$ on the single-qubit Pauli matrices is given by

$$\mathcal{N}'(I) = I + t_X X + t_Y Y + t_Z Z, \tag{23}$$

$$\mathcal{N}'(X) = D_X X, \tag{24}$$

$$\mathcal{N}'(Y) = D_Y Y, \tag{25}$$

$$\mathcal{N}'(Z) = D_Z Z, \tag{26}$$

where $\mathbf{D} = (D_X, D_Y, D_Z) \in [-1,1]^3$ and $\mathbf{t} = (t_X, t_Y, t_Z) \in [-1,1]^3$ are two vectors. If $\mathbf{t} = 0$ such that $\mathcal{N}(I) = I$, then the noise channel is said to be *unital* and reduces to Pauli noise (up to a change of basis). Otherwise, the channel is non-unital, i.e. $\mathcal{N}(I) \neq I$. A common example of non-unital noise is the amplitude-damping channel, characterized by $\mathbf{D} = (\sqrt{1-\gamma}, \sqrt{1-\gamma}, 1-\gamma)$ and $\mathbf{t} = (0,0,\gamma)$ for some $\gamma \in (0,1]$. Repeatedly applying an amplitude damping channel pushes a quantum state to the computational zero state.

In most applications of Pauli propagation, we are concerned with the dual representation $\mathcal{E}^\dagger$ of a channel $\mathcal{E}$, since we typically operate in the Heisenberg picture. Notably, $\mathcal{E}^\dagger$ is always unital, i.e., $\mathcal{E}^\dagger(I) = I$, but if $\mathcal{E}$ is non-unital, then $\mathcal{E}^\dagger$ is not trace-preserving. Thus, in the Heisenberg picture, noise channels do not affect or branch on I, but they may branch on X, Y, Z Paulis.

Controlled gates and measurements: Pure states like $|0\rangle\langle 0|^{\otimes n} = (I+Z)^{\otimes n}/2^n$ contain exponentially many terms in the Pauli basis. This means the introduction of control qubits to few-branching unitaries (or even most Cliffords) can make them prohibitively many-branching. As such, Pauli propagation techniques are typically ill-suited for the simulation of many-controlled unitaries. Despite this, *projectors* can often be efficiently implemented, even when global and with exponentially large PTMs. This is because inner products with stabilizer states like the all-zero state are efficiently computable. For example, the one-qubit zero projector $\Pi = |0\rangle\langle 0|$ is either two-branching, else *removes* a Pauli string.

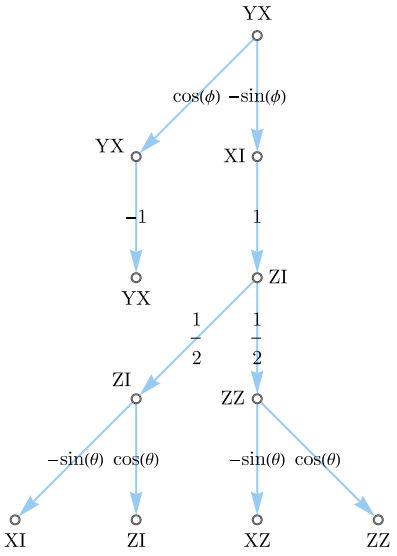
$$\Pi[I] = \Pi[Z] = (I+Z)/2, \qquad \Pi[X] = \Pi[Y] = 0. \tag{27}$$

The n-qubit zero projector factorizes into a product of n one-qubit projectors and is ergo principally 2^n-branching, but only when the targeted substring contains no X or Y. And strings differing only by $I \leftrightarrow Z$ in the targeted substring (of which there are $\leq 2^n$) are mapped to the *same* 2^n-term sum. Mid-circuit *measurements* which normalize the projector are also possible to efficiently simulate, though with dissimilar logic, namely, via ancilla qubits and controlled unitary operations [40].

The `QuESTlink` Mathematica library [41] is useful for quickly obtaining and visualizing Pauli transfer matrices of symbolic operations. For example, `DrawPauliTransferEval` can visualize the Heisenberg evolution of the string YXunder the circuit $CR_X(2\phi)\cdot(\mathrm{H}\otimes I)\cdot(R_Y(\theta)\otimes|0\rangle\langle 0|)$ as follows:

Software Box 2: Topologies and circuits

Multi-qubit gates can be applied between any pair (or more) of qubits, which is an advantage over more topology-tailored simulation methods. How qubits are connected does not directly determine the difficulty of the simulation, but it will likely affect how Pauli strings evolve and consequently how much they branch throughout the circuit. `PauliPropagation.jl` provides a suite of pre-defined entangling topologies in 1D and 2D, for example `bricklayertopology()` and `rectangletopology()` which can be passed to common circuit builders, for example, `tfitrottercircuit()` for simulating the dynamics of the transverse-field Ising model. Topologies are vectors of tuples like `topology = [(1, 2), (2, 3), (1, 3)]`, indicating which sites are connected via multi-qubit gates, and circuits are vectors of gates.



C. Pauli propagation surrogates

The word "surrogate" has been used within the fields of machine learning and quantum computing and has various meanings. What we refer to as a *Pauli propagation surrogate* is a compiled computational model for expectation values in the form of Eq. (11) that trades an initial overhead in time and especially memory against significantly faster re-evaluation of expectation values for different circuit parameterizations. One of the few Pauli propagation surrogates that we are aware of is known as LOWESA (low-weight efficient simulation algorithm). LOWESA was proposed in Ref. [15] for the classical simulation of noisy variational quantum algorithms, but the later works in Refs. [12,17,25,32] fully embraced the surrogation aspect of the initial proposal.

We would like to draw attention to two features in a simulation that can give rise to powerful surrogate models in the spirit of this work. First, if there is a lot of high-level logic in the algorithm, e.g., many if-else statements potentially with truncations. When this logic dictates the flow of the computation and only depends on the structure of the problem not its parameters, then the problem can be compiled once and subsequently evaluated deterministically without this logic. This is reminiscent of how software libraries for *automatic differentiation* work. A computational graph for the derivative of a function is built once and can then be evaluated again and again during common gradient descent routines. The graph does not change because it is fixed from the time that the function is defined, independent of the input parameters.

A second feature of quantum expectation value simulations, that can further drastically reduce re-evaluation time, is the fact that many Pauli terms annihilate when overlapped with an initial state (or operator). Consider the decomposition of an observable expectation function in the Pauli basis,

Software Box 3: Custom gates

The Pauli propagation framework can of course be applied to a broader class of operations than the ones listed in Sec. II B. There are a number of ways in which a new gate/operation can be defined in `PauliPropagation.jl`. The simplest is via the Pauli transfer matrix formalism (see Theory Box 1) and our `TransferMapGate` type. Alternatively, gates can be defined by directly specifying their action on Pauli strings in a custom `apply()` function, which may be more efficient but more complicated to implement. Depending on the gate type, it may be more efficient to manually control the branching pattern and implement more specialized memory management by overloading the `applytoall!()` function. We explain all approaches in our example Jupyter notebooks.

$$\langle O \rangle = \mathrm{Tr}[\rho U^\dagger(\boldsymbol{\theta}) O U(\boldsymbol{\theta})] \tag{28}$$

$$= \sum_\alpha c_\alpha(\boldsymbol{\theta}) \mathrm{Tr}[\rho P_\alpha] \tag{29}$$

where $U^\dagger(\boldsymbol{\theta}) O U(\boldsymbol{\theta}) = \sum_\alpha c_\alpha(\boldsymbol{\theta}) P_\alpha$ is a unitarily evolved observable written in the Pauli basis and ρ is the initial state. Usually, the sum over Pauli strings with their coefficients can instead be replaced or approximated with a significantly smaller sum over fewer terms where $\mathrm{Tr}[\rho P_\alpha]$ is large. Pauli paths leading to small or zero contributions do not need to be revisited when changing the parameters $\boldsymbol{\theta}$. See Fig. 2 for a proof-of-principle demonstration of the potential speed-ups.

Pauli propagation surrogates can be used naturally in the context of classically simulating variational quantum algorithms. Namely, one could optimize the parameterized quantum circuit fully classically on the approximate surrogate landscape. In some cases, one may need to be careful that the truncations used do not artificially change the depth or the positions of minima. For example, if aiming to find a ground state of a given Hamiltonian, a truncated Pauli propagation surrogate does not strictly simulate a normalized quantum state and thus could lead to energies that are lower than classically possible for a genuine quantum state. However, in other contexts, for example when classically simulating a quantum convolutional neural network for solving phase classification problems, as in Ref. [32], this is not an issue. The surrogate quantum convolutional neural network can simply act as a new quantum-inspired classical model for the task at hand.

Beyond variational problems, the prospects of Pauli propagation surrogates in the context of dynamical simulations are particularly intriguing. For example, when simulating the dynamics of a Hamiltonian $H(\boldsymbol{\lambda})$ the parameters of that system, $\boldsymbol{\lambda}$, appear as parameters in the gates of the circuit. A Pauli propagation surrogate of

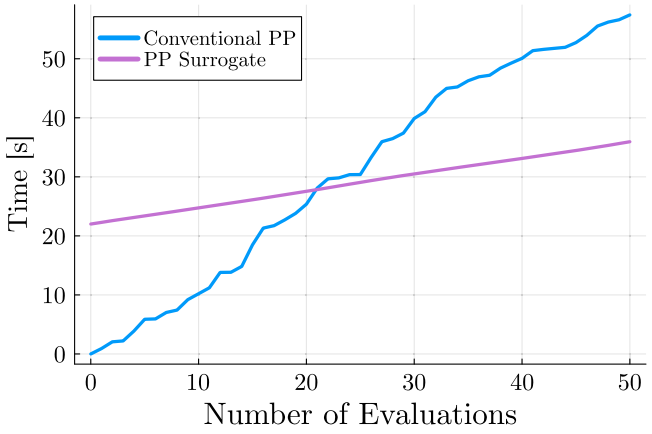


FIG. 2. Potential advantage of a Pauli propagation surrogate. A Pauli propagation surrogate invests time and especially memory in an initial surrogation stage, then repeated evaluations can be significantly faster than running conventional Pauli propagation over and over for different parameters. This is demonstrated on a 20-qubit example with 10 layers and random correlated parameters within each brickwork layer. We estimate the operator correlation value of Pauli Z in the middle of the qubit chain, which represents one of the strongest sparsifications of the surrogate relative to conventional Pauli propagation. The truncation parameters are chosen such that both models achieve approximately 0.005–0.010 average absolute error compared to statevector simulation, and both simulations use a single CPU thread for a fair comparison. We emphasize that this setup is chosen to demonstrate the potential advantages of surrogates, but our current surrogate implementation may not scale to the most interesting cases.

this circuit therefore has the potential to simulate a family of Hamiltonians. Similarly, if the initial state preparation circuit is included in the circuit description, one could include this circuit in the surrogate and explore the evolution of the system for a whole family of different initial states. We expect these capabilities to be most useful in the context of averaging over the effect of disorder or for meta-learning optimal Hamiltonians or initial states for various tasks.

We end by stressing that the potential of Pauli propagation surrogates has not yet been fully realized. Writing efficient code that builds a computational graph on the fly and compiles a function that can be evaluated very quickly is not trivial. `PauliPropagation.jl` currently contains a surrogate version close to the one used in Refs. [17,32], but it does not leverage sophisticated code compilation techniques that could drastically accelerate evaluation. As of now, all non-surrogate code in `PauliPropagation.jl` is differentiable for learning and optimization tasks, and currently is likely the better choice for almost all concrete applications.

D. Prior work and related methods

Pauli path integrals are a relatively recent theoretical tool for quantum information which gained traction after the seminal work of Ref. [29], providing polynomial-time classical algorithms for sampling from the output distribution of certain noisy circuits. Pauli-path based techniques also play a central role in spoofing the linear cross-entropy benchmark used in quantum supremacy experiments [42,43]. There have further been numerous papers providing guarantees for noisy [15,29–31,44,45] and noiseless [14,17] circuits as well as efforts to understand such methods through Fourier analysis [15,46,47]. However, in this section we focus on the even more recent developments in practical implementations of algorithms related to Pauli propagation.

Pauli propagation is related to so-called *stabilizer simulation* [48] for simulating Clifford circuits and the *extended stabilizer simulation* [49] for Clifford and few non-Clifford circuits. Those frameworks are concerned with how states can be compactly represented in the Pauli basis, and how that efficient representation can be used to estimate expectation values or sample. If an n-qubit quantum state is a Pauli stabilizer state [48,50], i.e. if the state can be fully characterized by n Pauli strings, how those Pauli strings are updated under the action of common gates is in the same spirit (and sometimes exactly the

Software Box 4: Works using our code

Very early versions of our Pauli propagation surrogate have been used to simulate the IBM "utility" experiment on 127 qubits [12]. More refined versions of this code were used to demonstrate that quantum convolutional neural networks (QCNNs) may be more broadly classically simulable than previously thought [32]. The current version of the Pauli propagation surrogate in `PauliPropagation.jl` was demonstrated in Ref. [17] in conjunction with theoretical guarantees for hybrid simulation schemes involving quantum computers. Modern versions of our more flexible non-surrogate code have been used in Ref. [14], demonstrating that Pauli propagation with Pauli weight truncation leads to efficient average-case simulation of circuits that were previously believed to be intractable. In Ref. [31] it was used to demonstrate efficient large-scale simulation of quantum circuits with arbitrary local noise. Reference [54] compresses 2D quantum dynamics circuits up to 30×30 grid size via automatic differentiation. Reference [55] demonstrates how spatial symmetries in quantum systems can be leveraged to reduce memory via symmetry merging strategies now available in our library. Reference [34] prepares thermal states via imaginary time evolution—a feature which has since been added to our library. Finally, `MajoranaPropagation.jl` [56] is built on top of `PauliPropagation.jl` to flexibly simulate fermionic systems in the related Majorana basis.

same) as in Pauli propagation. One software package for stabilizer simulation is STIM [51], which is vastly faster for simulating purely Clifford circuits, but does not support non-Clifford gates. For circuits consisting of Clifford gates and Pauli rotations (including T gates), for example, the resulting state will be a weighted sum of stabilizer states, each of them represented via n Pauli stabilizers. Due to the flexibility in our code design, it is possible to implement extended stabilizer simulation in the Schrödinger picture without redefining the propagation—only the action of gates on the so-called stabilizer *tableaus* [48]. More concrete implementations can be found via, e.g., Refs. [52,53].

One open-source package embracing Heisenberg evolution in the Pauli basis is `PauliStrings.jl`. Its focus is on simulating real-time quantum dynamics by solving the von Neumann equation directly via calculation of the nested commutator of the evolving observable $O(t)$ with the Hamiltonian H. The time derivative at time t is estimated like

$$\frac{\partial O(t)}{\partial t} = i[H, O(t)], \tag{30}$$

which can then be plugged into a partial differential equations solver of choice, for example, 4th-order Runge-Kutta. For continuous-time dynamics, this approach could potentially be compared to `PauliPropagation.jl` simulating a finely Trotterized quantum circuit, but a fair comparison would be fiddly given the different focuses of the two packages (e.g., real-time dynamics versus circuit simulation/Floquet dynamics). Alternatively, `PauliStrings.jl` can be used for Lanczos or Krylov methods. For more information we refer to the accompanying manuscript in Ref. [57]. Under the broad umbrella of Lanczos Pauli basis simulation, a number of themes have been explored that have been, or could be, applied to vanilla Pauli propagation. These include: surrogation [58], weight truncation for random circuits [59], extrapolation methods [60], generalizations beyond the Pauli basis [61] and limitations due to dynamical phase transitions [62].

Mirroring the progress in the field of Pauli propagation in general, the works in Refs. [11,16,24] demonstrate large-scale simulations with various truncation strategies and design choices. The source code for the corresponding publications is partially available therein, and the results can be replicated out-of-the-box via `PauliPropagation.jl`.

One tensor network approach from the physics community that mirrors some motivating principles of Pauli propagation is known as *dissipation-assisted operator evolution* (DAOE) [63]. It uses matrix product states (MPS) in the Pauli basis with the specific goal of estimating expectation values after dynamical evolution. DAOE shares several conceptual similarities with Pauli propagation. First, in the language of Pauli propagation, DAOE can be viewed as encoding the Pauli coefficients of a Heisenberg evolved observable in a tensor network (and thereby allowing access to a wide repertoire of tensor network tools and compression). Second, it employs a strategy similar to Pauli weight truncation [14] by imposing exponentially decaying coefficients for higher-weight Pauli operators. DAOE has been successfully employed to extract transport quantities from highly damped Paulis by extrapolating to the zero-damping limit, thus showing that the fields of tensor networks and Pauli propagation may continually and jointly progress for targeted applications.

There are, however, both conceptual and practical differences between Pauli propagation and DAOE-or rather, tensor networks in general. One difference is that Pauli propagation is limited precisely by the number of Pauli strings N, whereas an MPS bond dimension χ is only (very loosely) upper-bounded by it. At the same time, the memory and runtime of Pauli propagation are $\mathcal{O}(N)$, while the memory of MPS is $\mathcal{O}(\chi^2)$ and the runtime is $\mathcal{O}(\chi^3)$. In practice, the bond dimension χ is determined by how spatially factorizable the underlying Pauli sum is into the tensor network graph, and almost always we find that significant compression is possible, i.e., that $\chi \ll N$. An extreme example is the uniform superposition of all $N = 4^n$ Pauli strings over n sites, which is a propagation algorithm's nightmare, but it remains $\chi = 1$ as an MPS. Removing one Pauli string from that sum actually *increases* the bond dimension to two. Similarly, the MPS bond dimension in DAOE used for encoding operator entanglement can grow rapidly when Pauli strings are fully truncated or with the size of the window of undamped Pauli strings. The other extreme is a Pauli sum consisting of a polynomial number of random Pauli strings. Its factorization likely remains full rank, which results in a polynomial memory and runtime disadvantage over Pauli propagation. That being said, when simulating physical systems, the required computational resources for both methods tend to correlate quite strongly [64], making the comparison a case-by-case study.

Above we have detailed some classical simulation algorithms that are most closely related to Pauli propagation. However, a complete survey would be impossible. Let us end this section by simply highlighting that tensor network methods for evolving operators have substantially outgrown their origins to a highly flexible approach. In particular, state of the art tensor network methods can work in the Heisenberg picture [65,66] and tackle beyond 1D topologies [11,67]. The relative merits of Pauli propagation compared to tensor networks, and other approximate simulation methods, remains an open question.

III. MID LEVEL

A. Truncation strategies

Truncations are targeted approximations in the simulation of quantum systems that aim to achieve drastic

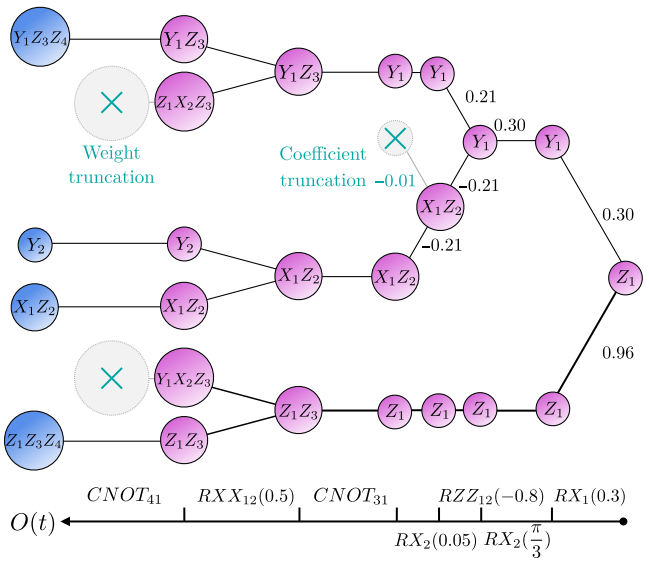


FIG. 3. Truncations during Pauli propagation. A tree graph illustrates the evolution of Pauli strings (nodes) and their associated coefficients (edges) under a unitary circuit, applied in sequence from right to left along the bottom axis. The shade of the edges indicates the magnitude of the coefficient size and the size of the node the weight of the corresponding Pauli string. The first three gates are illustrated in Theory Box 2. In the worst case, the leaves of the tree will grow exponentially with the number of gates, leading to an intractable tree. However, for many simulation tasks, this growth can be controlled using systematic truncation strategies described in Sec. III A. Two types of truncations are depicted: weight truncation, which discards Pauli strings with weight above a chosen threshold (e.g., weight 4), and coefficient truncation, which prunes strings with small coefficients (e.g., −0.01). Custom truncations are also supported by `PauliPropagation.jl`.

Software Box 5: Truncations

`PauliPropagation.jl` provides most truncations mentioned in Sec. III A by default. The most important keyword arguments to the `propagate()` function will be the minimum coefficient truncation threshold `min_abs_coeff`, and the weight truncation threshold `max_weight`. Custom truncation functions can be passed via the `customtruncfunc` keyword argument. The only truncation active by default is `min_abs_coeff=1e-10`, leaving typical calculations practically exact. We recommend approaching simulations *from fast to faithful*, i.e., starting with truncation values that clearly truncate too much and slowly making them more faithful. This is because the hardness of simulating quantum systems/circuits can vary drastically, and one will want to observe *convergence* similar to Fig. 6. For typical expectation value calculations, we might start off with `min_abs_coeff=1e-3` and `max_weight=5`. Often, such simulations will finish in (milli)seconds but provide imprecise results. This will be highly case-dependent.

reductions in computational resources while maintaining high accuracy. The degree of required accuracy strongly depends on each task. Some tasks require computations to be precise up to an exponentially small scale, while others only need to be visually close at the unit scale.

Looking at Eq. (11) there are currently two main families of truncations: 1) ones that truncate small (or likely small) coefficients c_α and 2) ones that truncate Paulis P_α where $\mathrm{Tr}[\rho P_\alpha]$ is small (or likely small). Expectation values can be significantly easier to simulate than full operators as the second truncation scheme only works in the latter case. In Fig. 3, we illustrate these two truncation strategies, using an example of a sequence of unitary gates evolving an initially local Pauli string. The resulting operator contains fewer Pauli strings when truncations are applied, compared to the full evolution.

Small coefficient truncation. The most straightforward truncation strategy is to discard Pauli strings whose current absolute coefficients fall below some threshold. While it is intuitive that truncating the smallest coefficients should result in the lowest approximation errors, and it practically often does, those errors may still accumulate to result in large errors. This is because complex quantum dynamics may generate a considerable number of new Pauli strings—exponential in the number of non-Clifford gates—each with a small coefficient. As a result, approximating the evolved observable with few Pauli terms can become extremely challenging, if not impossible.

This truncation strategy is tightly connected to the concept of magic (see box 6 for more details). Nevertheless, truncating small absolute coefficients is the current go-to truncation strategy for Pauli propagation, without which practical simulations are unimaginable. It has, for example, been used in Refs. [11,16] under the name of *sparse Pauli dynamics* (SPD), and is currently the only truncation in `PauliPropagation.jl` that is active by default. We note that some works report the coefficient truncation value *before* merging, while we currently truncate *after* merging. Truncating after merging is safer for general gates (our focus), and it is somewhat more accurate and resource-intensive. This means that, when using truncation thresholds reported in other works, `PauliPropagation.jl` simulation may be more accurate as a consequence of maintaining more Pauli strings (and, correspondingly, potentially slower). Thus slightly modified truncations may yield the most comparable results.

Pauli weight truncation. This truncation scheme drops Pauli strings with large weight, i.e., many non-identity Paulis, which are often likely to have a small overlap with the initial state. An intuitive example is that of the all-zero initial state $\rho = |0\rangle\langle 0|$, which can be written as a uniform sum of all Pauli strings consisting of I and Z Paulis. Thus,

if we consider a Pauli string P with W non-identity Paulis and assume that those Paulis are drawn randomly from the set $\{X, Y, Z\}$, then the "probability" that $\mathrm{Tr}[\rho P] = 0$ is $1 - (1/3)^W$ because a Pauli string containing *any* X or Y Pauli is orthogonal to $|0\rangle\langle 0|$. This argument in fact holds not just for the all-zero initial state but also for all stabilizer states and indeed random quantum states [14].

Reference [14] goes further and proves that this truncation strategy yields small average-case simulation errors for *locally-scrambling* circuits, i.e., for circuits that are locally least random for any observable and initial state. However, this provable guarantee is on average and has high error for specific domains in parameter space. Quantum circuits describing real-time quantum dynamics, for example, do not satisfy the theoretical assumptions and practically lead to larger errors. One may expect that this is due to Pauli strings P with large weight having larger overlaps with the initial state, i.e., larger $\mathrm{Tr}[\rho P]$ than in unstructured simulations, but this does not seem to be the only reason. While it has not yet been rigorously explored in the context of quantum circuit simulation, there exists indication that structured circuits with conserved quantities exhibit systematic operator *backflow* from high to low weight [63]. Thus, truncating high-weight Pauli strings may result in this backflow being interrupted, leading to strongly accumulating errors over time.

A variant of weight truncation whereby Z terms are not counted for Pauli weight (i.e., only Pauli strings with large numbers of X or Y terms are truncated) was employed in Ref. [16] for quantum dynamics where the initial state is the all-zero state. While this heuristic appears sensible, it is unclear whether it provides practical improvements over Pauli weight truncation, which is less accurate per truncation threshold but substantially cheaper.

There are a number of truncation strategies that can naturally be applied in the context of Pauli propagation surrogates. In the context of fixed angle simulations these truncation schemes are not strictly necessary or advantageous but in certain cases can be used to provide average case theoretical guarantees in some regimes. We detail these below.

Frequency truncation. The *frequency* of a path is the number of times a Pauli path picks up either a cosine or a sine coefficient—i.e., how often it branches at Pauli rotations—in a circuit consisting of Clifford gates and Pauli rotations. Truncating based on this quantity is effectively truncating according to the *average* coefficient over the entire parameter range. Consider that $\frac{1}{2\pi}\int_0^{2\pi}\sin^2(\theta)d\theta = \frac{1}{2\pi}\int_0^{2\pi}\cos^2(\theta)d\theta = \frac{1}{2}$. It becomes clear that if a path has picked up ℓ cosine or sine coefficients, then the average coefficient will be $(\frac{1}{2})^\ell$. This has been employed for the Pauli propagation surrogate algorithm LOWESA for simulating the so-called IBM quantum utility experiment [12]. We further note that in noisy quantum circuits and particular ansatz structures, frequency truncation provides provable average-case polynomial runtimes for inversely polynomially small errors [15,68].

Sine truncation. A more targeted version of frequency truncation, that we sometimes refer to as *small-angle truncation*, is the *sine truncation* dropping paths if the number of sine factors picked up in a circuit consisting of Clifford gates and Pauli rotations is above some threshold. This is the truncation at the heart of the so-called *Clifford perturbation theory* (CPT) approach [24] because this truncation can be understood as an expansion of the evolved observable in the power of sines. If the angles parametrizing Pauli rotations are small, this truncation tends to truncate the smallest coefficients. In fact, Ref. [17] proves that sine truncation can be applied to faithfully simulate small-angle regions—so-called *near-Clifford patches*—in polynomial time.

Path-weight truncation. What is commonly referred to as path weight is the *cumulative* Pauli weight of a Pauli as it propagates. Where precisely in the circuit the path weight is updated by adding the current Pauli weight is arbitrary and could be at every gate or every layer. The practical benefits of this truncation strategy are so far largely unexplored. However, it has been used to great success in formalizing the classical simulability of noisy quantum circuits [29–31]. In those theoretical results, path weight is a tight proxy for the magnitude of Pauli path coefficients due to the damping nature of common noise models. However, it is not clear whether this truncation scheme has advantages over direct small coefficient truncation in noisy circuits or Pauli weight truncation for noise-free circuits.

B. Pauli propagation is a tree search problem

We now face the question of how to compute all the contributing terms to Eq. (10) or at least the most important ones. Due to the branching nature of the simulation problem, this problem can be framed as a *tree search*. This is a widely-studied topic in computer science, with two general search approaches being the *depth-first search* (DFS) and *breadth-first search* (BFS). In our case, DFS amounts to simulating in a *path by path* fashion, and BFS as in a *gate by gate* fashion. This is sketched in Fig. 4.

The main advantage of DFS is its low memory usage. Whenever a Pauli string branches into more than one Pauli string, one always follows through with the simulation of one of those Pauli paths until the end of the circuit. The other is saved (e.g., via a *stack* data structure or recursion) until it is their turn to be propagated. At the end of the circuit, one computes the overlap of the final Pauli string with the initial state/observable and its contribution can be added to a running sum. Then the simulation can proceed to simulate the next path without storing in memory anything about the previous one. Thus DFS always has polynomial memory (likely linear) requirements in the

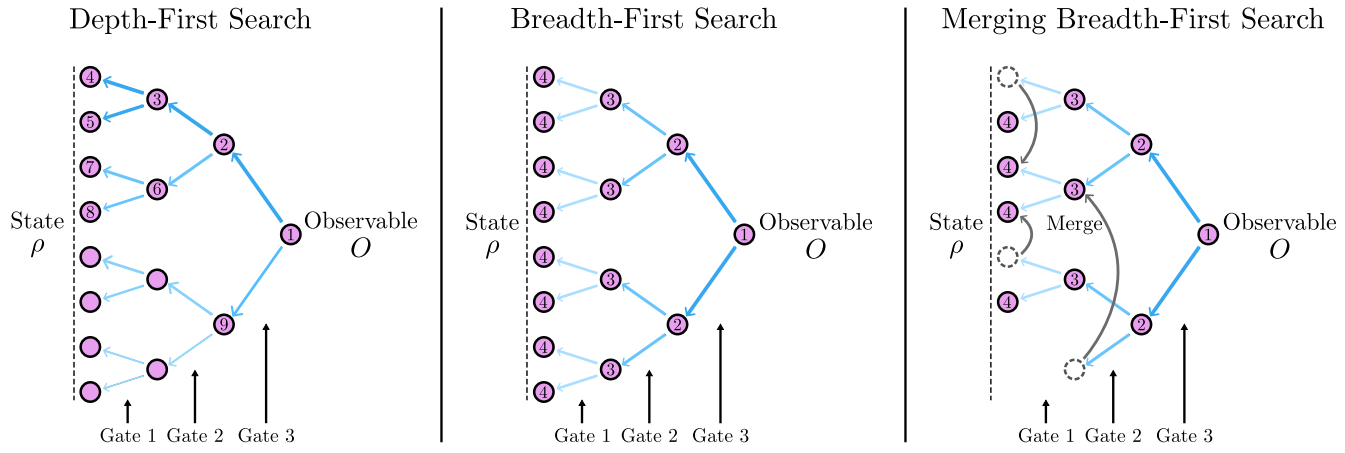


FIG. 4. Traversing the Pauli propagation tree. Repeated application of a gate to an observable represented as a sum of Pauli strings can cause repeated *branching* of Pauli strings into other Pauli strings with new coefficients. The exact branching pattern depends on the type of gate, but here we explore two-branching gates (e.g., Pauli rotations) for simplicity. The nodes in the trees represent Pauli strings at different positions in the circuit, and the strength of the arrows and the numbers on the nodes denote the order in which the algorithm visits the nodes. *Depth-first search* (DFS) is a strategy that fully propagates a Pauli string through the entire circuit before returning to propagating the next. This is very light on memory. *Breadth-first search* (BFS) instead evolves Pauli strings gate-by-gate, consequently being heavier in memory. However, knowledge about all Pauli strings after each gate allows for *merging* of identical Pauli strings. This *merging BFS* approach is asymptotically faster than DFS or BFS, and its memory requirements are in between the other two.

number of gates and qubits, which is exemplified by the credibility of the claims to have simulated the so-called "IBM quantum utility experiment" on a Commodore 64 computer with 64 kB of RAM [69]. The approach can also be efficiently parallelized as was demonstrated in Ref. [12] simulating that same quantum experiment. The major disadvantage of DFS is its runtime which is substantially larger than BFS due to the impossibility of merging paths.

BFS computes the Pauli sum at each point in the circuit. Commonly this causes an exponential increase in the number of stored Pauli strings with the number of branching gates, and thus a strictly higher peak memory requirement than DFS. However, BFS allows for the *merging* of paths where the current Pauli strings are identical by simply adding their coefficients together and then continues propagating them as one. This not only reduces the self-inflicted memory requirements relative to naive BFS, but crucially also the runtime relative to DFS. This is because numerous Pauli paths have collapsed into one, with every subsequent gate application effectively being parallelized among all merged paths. We call this approach *merging-BFS*.

Merging-BFS trades a likely exponential memory disadvantage for likely strong polynomial speed-up over DFS. Considering the common case of m two-branching Pauli rotations, the runtime measured by the number of paths can be up to 2^m. In practice, as discussed in Sec. II B, the exponent will not be exactly m, given that gates can commute with Pauli strings, especially for local Pauli strings consisting of mostly identity Paulis. Even so, the number of unique Pauli strings that merging-BFS would compute is upper-bounded by that same number, but for non-pathological cases, the rate of unique Pauli strings will be drastically lower than the number of paths. This is exemplified in Fig. 5. In the limit of deep circuits, merging-BFS is limited by the full Pauli space, i.e., 4^n unique Paulis, whereas the number of paths $\mathcal{O}(2^m)$ can keep growing. Thus, even simulating circuits with a few qubits, e.g., $n \lessapprox 12$, could run eternally using DFS if there are enough non-Clifford gates. For a schematic representation of the tree traversal algorithms we point to Fig. 4.

In the `PauliPropagation.jl` package we currently optimize for runtime over memory and focus on a BFS strategy. Indeed, a purely DFS strategy will require impossibly demanding runtimes except for simple case studies where truncations can be tailored to only explore relatively few contributing paths [12]. Eventually, we may need to develop hybrid DFS and (merging-)BFS approaches, potentially involving randomized techniques such as *Monte Carlo* sampling of paths. While currently not explored, such approaches may become necessary when memory becomes the limiting factor. Merging-BFS Pauli propagation can fill 1 TB of memory in a few hours using a single CPU thread when simulating physical problems of interest. Future versions of this code may be parallelizable over plenty CPU threads to the point where typical computational cluster nodes can run out of memory in little time. Given this context, departing from purely merging-BFS for hard simulations seems to be a certainty, but how to combine the best of all worlds is currently not clear.

C. Error estimates

When pushing the boundaries of classical simulation, there remains the question of how accurate the results are.

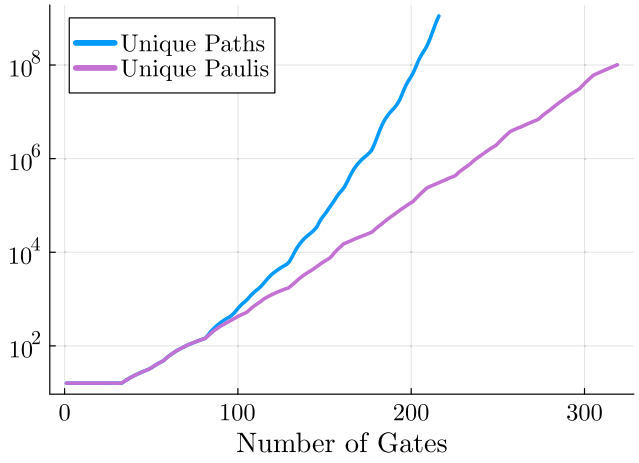


FIG. 5. The importance of merging Pauli paths. We consider a 16-qubit quantum circuit consisting of layers of RX, RZ and RZZ Pauli rotations on a periodic bricklayer topology and compare the number of Pauli paths to the number of unique Pauli strings in the simulation. This can be understood as the comparison between DFS or BFS, and the merging-BFS approach. The initial observable is the sum of single-qubit Z Pauli strings, which initially gives rise to operator lightcone effects. After the initial stages, the number of paths scales with a stronger exponential in the number of Pauli rotations, which translates into a relative slowdown of pure DFS/BFS approaches without merging. No truncations are employed here.

It is often not clear how much total error has been incurred via systematic approximations and truncations, and how *converged* the simulations are in the relevant truncation parameters. This is a challenge for all large-scale classical simulation methods, including Pauli propagation and tensor networks, as well as non-error-corrected quantum devices.

Two approaches for estimating errors in quantum circuit simulations are: (i) using the known error per gate to upper-bound the total error via the triangle inequality, and (ii) monitoring convergence by observing that quantities of interest stabilize as truncation thresholds are relaxed. The first approach of bounding errors per gate is, for example, used widely in tensor network simulations. For instance, in time-dependent variational principle (TDVP) algorithms, a gate is absorbed into the state, followed by a singular value decomposition (SVD). Small singular values are discarded to truncate the bond dimension, and the sum of the squared discarded singular values directly quantifies the infidelity of the gate application. This infidelity can then be used to bound deviations in expectation values. A similar strategy can be applied to Pauli propagation, which we discuss next.

Error per gate. During unitary evolutions in Pauli propagation, the two-norm of the observable $O = \sum_\alpha c_\alpha P$—i.e. $2^{-n}\mathrm{Tr}[O^2] = \sum_\alpha c_\alpha^2$—is conserved and any loss in that can be directly measured. Unfortunately, however, it is the ℓ_1-norm of the coefficients c_α that directly bounds the additive error on expectation values, as discussed in Theory Box 3.

While it is encouraging that Pauli propagation appears to perform significantly better currently known error estimates, it also means that having provable trust in Pauli propagation results is difficult to obtain. Remarkably, although non-trivial worst-case error bounds have been established for restricted circuit classes that generate low

Theory Box 3: Bounding the error per truncation

Assume we want to approximate $\mathrm{Tr}[O\mathcal{C}(\rho_0)] = \mathrm{Tr}[\mathcal{C}^\dagger(O)\rho_0]$, where $\mathcal{C} = \mathcal{C}_L \circ \mathcal{C}_{L-1} \circ \cdots \circ \mathcal{C}_1$ is an L-layered quantum circuit. Using Pauli propagation, we can approximate the observable $\mathcal{C}^\dagger(O)$ by iteratively back-propagating and truncating O layer by layer. Given $O = O_L$, we start by computing the observable $\mathcal{C}_L^\dagger(O_L) = \sum_\alpha c_\alpha P_\alpha$ and we truncate some of its Pauli terms, obtaining the observable O_{L-1}.

We can upper bound the truncation error via the triangle inequality

$$|\mathrm{Tr}[(O_{L-1} - \mathcal{C}_L^\dagger(O_L))\rho_{L-1}]| \tag{31}$$

$$\leq \sum_{\substack{\text{truncated}\\ \text{Paulis}}} |c_\alpha| := \Delta_L, \tag{32}$$

where $\rho_{L-1} = \mathcal{C}_{L-1} \circ \cdots \circ \mathcal{C}_1(\rho_0)$. Iterating the same argument over L layers and applying again the triangle inequality, we find that the total error is upper bounded by

$$\Delta := \Delta_L + \Delta_{L-1} + \cdots + \Delta_1, \tag{33}$$

where Δ_j is the ℓ_1-norm discarded during the $(L-j+1)$-th truncation. We stress that this is a *worst-case* bound, as we didn't make any assumption on the initial state ρ_0 or the circuit layers $\mathcal{C}_j$.

Under certain assumptions, this bound can be further tightened: if we want to approximate a two-point correlation function $2^{-n}\mathrm{Tr}[O\mathcal{C}(P_0)]$, the error per truncation can be bounded by the ℓ_2 norm

$$\frac{1}{2^n}|\mathrm{Tr}[(O_{L-1} - \mathcal{C}_L^\dagger(O_L))P_{L-1}]| \tag{34}$$

$$\leq \sqrt{\sum_{\substack{\text{truncated}\\ \text{Paulis}}} c_\alpha^2}, \tag{35}$$

where $P_{L-1} = \mathcal{C}_{L-1} \circ \cdots \circ \mathcal{C}_1(P_0)$ and we applied Cauchy-Schwarz's inequality. A similar bound based on the ℓ_2-norm also holds *on average* when the initial state ρ_0 is a random computational basis state [30].

Theory Box 4: Worst-case vs average-case errors

We provide two simple toy examples showing that worst-case and average-case errors can be substantially different.

Weight truncation. Let $O = Z^{\otimes n} + Z \otimes I^{\otimes(n-1)}$. Approximating O via the *weight-truncation* rule with some cutoff $k \geqslant 2$, we obtain $O' = Z \otimes I^{\otimes(n-1)}$. The truncation error is one if we take a computational basis state as input, but it is exponentially small on average over tensor products of *random* single-qubit stabilizer states

$$|\mathrm{Tr}[(O - O')|0^n\rangle\langle 0^n|]| = 1, \tag{36}$$

$$\mathbb{E}_{|\psi\rangle\sim \mathrm{Stab}_1^{\otimes n}} |\mathrm{Tr}[(O - O')|\psi\rangle\langle\psi|]| = 3^{-n}, \tag{37}$$

where $\mathrm{Stab}_1 = \{|0\rangle, |1\rangle, |+\rangle, |-\rangle, |+\rangle_y, |-\rangle_y\}$.

Small-coefficient truncation. Let $O = Z_1 + \frac{1}{n-1}\sum_{i=2}^{n} Z_i$, where Z_i denotes the Pauli string acting as Z on the i-th qubit and as identity elsewhere. Approximating O via the *small-coefficient truncation* rule with some cutoff $\tau > \frac{1}{n-1}$, we obtain $O' = Z_1 = Z \otimes I^{\otimes(n-1)}$. The truncation error is one if we take the computational zero state as input, but it scales inverse-polynomially if instead we consider *random* computational basis states

$$|\mathrm{Tr}[(O - O')|0^n\rangle\langle 0^n|]| = 1, \tag{38}$$

$$\mathbb{E}_{\boldsymbol{x}\sim\{0,1\}^n} \mathrm{Tr}[(O - O')|\boldsymbol{x}\rangle\langle\boldsymbol{x}|]^2 \tag{39}$$

$$\leqslant \mathrm{Tr}\left[(O - O')^2 \left(\frac{\mathbb{I}}{2^n}\right)\right] = \frac{1}{n-1}. \tag{40}$$

These examples highlight the gap between worst-case and average-case error bounds. While both offer guidance for choosing a simulation strategy, they must be interpreted with care and in context.

Theory Box 5: Monte Carlo error estimate

Here we convey the general mathematical intuition behind the Monte Carlo error estimate, which is described in detail in App. H of Ref. [14]. Given a circuit $U(\boldsymbol{\theta})$ parameterized by $\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_m)$, an observable O and a state ρ, we aim at approximating the expectation value $\langle O(\boldsymbol{\theta})\rangle := \mathrm{Tr}[OU(\boldsymbol{\theta})\rho U(\boldsymbol{\theta})^\dagger]$, which can be expressed as a sum over Pauli paths as follows:

$$\langle O(\boldsymbol{\theta})\rangle = \sum_{\text{Pauli paths}\,\gamma} \Phi_\gamma(\boldsymbol{\theta})\mathrm{Tr}[P_\gamma\rho]. \tag{41}$$

$\Phi(\gamma)$ are the coefficients associated to the paths γ and P_γ is the last Pauli string resulting from the path γ. These paths can be understood as branches of the propagation tree without merging. When approximating the true expectation value via $\langle O^{(\mathcal{S})}(\boldsymbol{\theta})\rangle$ with a truncated set of paths $\mathcal{S}$, we can efficiently estimate the mean square error (MSE) $\mathbb{E}_{\boldsymbol{\theta}}(\langle O(\boldsymbol{\theta})\rangle - \langle O^{(\mathcal{S})}(\boldsymbol{\theta})\rangle)^2$ on average over the parameters $\boldsymbol{\theta}$. Assuming that the angles θ_i are *independently* sampled, e.g., $\boldsymbol{\theta}$ is sampled uniformly from $(-\pi, \pi)^m$, the MSE can be rewritten in a strikingly simple form

$$\mathrm{MSE} = \sum_{\text{Pauli paths}\,\gamma} \underbrace{\mathbb{E}_{\boldsymbol{\theta}}\Phi_\gamma^2(\boldsymbol{\theta})}_{\Pr[\gamma]} \underbrace{\mathbf{1}_{\gamma\notin\mathcal{S}}\mathrm{Tr}[P_\gamma\rho]^2}_{g(\gamma)} \tag{42}$$

$$= \mathbb{E}_\gamma g(\gamma), \tag{43}$$

where $\mathbf{1}_{\gamma\notin\mathcal{S}}$ denotes the indicator function which equals to 0 if γ is in $\mathcal{S}$ and 1 otherwise. This characterization of the MSE ensures that it can be estimated with high probability and arbitrarily small error ϵ by sampling $\mathcal{O}(\epsilon^{-2})$ Pauli paths with probability $\Pr[\gamma] := \mathbb{E}_{\boldsymbol{\theta}}\Phi_\gamma^2(\boldsymbol{\theta})$ and averaging the associated values of $g(\gamma)$. In practice, the average error may be smaller due to the merging of Pauli paths, but also a lot larger for particular choices of $\boldsymbol{\theta}$, i.e., this value is not representative of the worst-case error.

magic or operate in high-noise regimes [17,21,45], a series of works has shown that average-case analyses can capture a broader ensemble of circuits under both noiseless and noisy conditions [14,29–31]. However, it is crucial to interpret these error guarantees in context. As we illustrate with a simple toy example in Theory Box 4, worst-case and average-case errors can be within a multiplicative factor which is exponential in system size.

Monte Carlo error estimates. One rather remarkable capability of Pauli path techniques is to provide efficient protocols for estimating errors on average over an ensemble of circuit parameterizations. Consider the case of Pauli rotations multiplying sine and cosine coefficients onto Pauli paths. We remember that $1 \leqslant |\sin(\theta)| + |\cos(\theta)| \leqslant \sqrt{2}$ but $\sin(\theta)^2 + \cos(\theta)^2 = 1$. This implies that while expectation values cannot in general be estimated efficiently with Monte Carlo sampling of Pauli paths due to the exponentially increasing one-norm of the coefficients, second order quantities such as variances and mean square errors can be efficiently estimated, as discussed in more detail in Box 5. However, it is important to stress that these error estimates are, as mentioned above, *on average* over an ensemble of circuit parameters under the condition that Pauli rotations are *independently parameterized*. While this can then be used to demonstrate efficient simulability on average over certain families of circuits, for any particular set of parameters, especially those not randomly

drawn from a distribution, the Monte Carlo estimates are likely to drastically underestimate the error.

Apparent convergence. A more concrete approach for estimating errors in approximate simulations is testing for convergence in the truncation values. It is reasonable to assume that results, e.g., expectation values, stabilize on any fixed scale when more and more computational resources are spent. This is strictly true for unbiased estimators, but it is still a working heuristic in the domain of tensor networks where truncation via bond dimension biases calculations. Many state-of-the-art results are attained by *extrapolating* from finite bond dimension simulations to the infinite bond dimension regime (e.g., Ref. [70], but it is common practice), yet there is no theoretical guarantee that this should work as well as it does. Extrapolating in Pauli propagation truncations appears more challenging in the Pauli picture where we commonly see oscillating expectation values rather than monotonically converging ones.

What remains is to visually observe convergence, as exemplified in Fig. 6. Choosing lower and lower coefficient truncation cutoffs, requiring additional computational resources, we see the expectation values in the real-time dynamics of a 6×6 system under the tilted-field Ising Hamiltonian coincide for longer and longer times. That being said, it is not *a priori* clear what a *good* accuracy is. The general features of the dynamics on this timescale can arguably be recovered with a threshold in the range 2^{-14}–2^{-16}, which for us took between 35 s and 5 min with off-the-shelf `PauliPropagation.jl` code and a single CPU thread. These truncations produced between 1.2 million and 9.3 million unique Pauli strings. The precise expectation values between $t = 1.0$ and $t = 1.5$ seemingly require a cutoff of 2^{-18} or below, which took 26 min with a single CPU thread and produced just under 50 million Pauli strings.

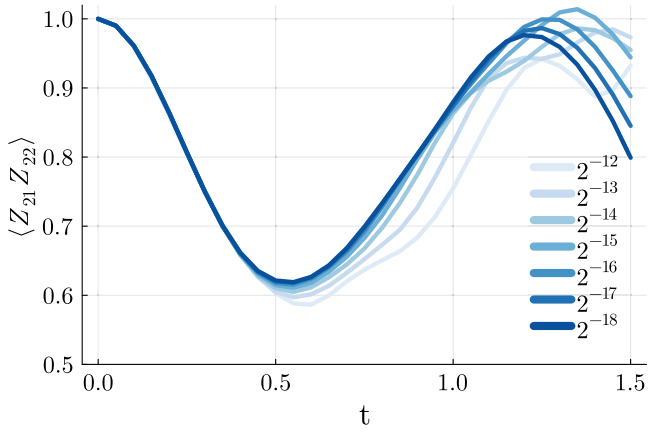


FIG. 6. Testing convergence in Pauli propagation. This is a 6×6 qubit system evolving under the so-called tilted-field Ising Hamiltonian consisting of Z, X and ZZ terms. We choose a time step of $dt = 0.05$ and simulate up to 30 Trotter layers. The different lines are different choices of the minimum absolute coefficient threshold. An additional Pauli weight truncation of nine is imposed, which we have not witnessed visually affecting the results on this time scale. We observe apparent convergence in the sense that the different simulations agree until longer and longer times, which increases trust in certain features in time.

IV. UNDER THE HOOD

This section considers the implementation nuances of Pauli propagation such as choices of data structures and high-performance optimizations. We will both motivate `PauliPropagation.jl`'s implementation choices and propose further generalizations relevant to the framework.

The goal of a high-level library like `PauliPropagation.jl` is to abstract many of the technical complications away, freeing users from worrying

Glossary Box 1: CS jargon

Explaining the internal engine of Pauli propagation requires using language from the computer science (CS) community. This box clarifies several such terms used below.

Amortized complexity averages the computational complexity of many *sequential* operations, as an alternative to the often overly pessimistic worst-case complexity. This is useful for capturing the practical performance of growing data structures, like when appending elements in a list (amoritzed $\mathcal{O}(1)$ time), for which the worst-case complexity corresponds to a non-representative, rare event ($\mathcal{O}(N)$ resizing).

Branching, in the context of a program's control flow, occurs when executing conditional instructions like `if` statements. This is distinct from when we say an operator induces "branching" of a Pauli string.

Endianness indicates the ordering of bits in a binary number; whether the rightmost bit is the least (little-endian) or most (big-endian) significant. It also defines the ordering of qubits in a quantum register and in our notation.

Enumeration involves iterating over the elements within a well-ordered container. In our context, "enumerating a Pauli sum" refers to sequentially processing each constituent Pauli string.

Hot loops are control flow loops (like induced by `for`) with many iterations and which dominate runtime, creating a "hotspot" in performance analysis. Accelerating the body of such loops is typically a priority during optimization. Like many quantum simulations, Pauli propagation involves exponentially-many repetitions of a numerical subroutine, forming a hot loop.

Software Box 6: Pauli string example

As an example, the high-level object representing the Pauli string *XIZY* can be defined in `PauliPropagation.jl` via

```
PauliString(4, [:X, :Z, :Y], [1, 3, 4])
```

A Pauli sum object representing $0.5 \cdot XIZY - 1.3 \cdot IZII$ can be defined by

```
psum = PauliSum(4)
add!(psum, [:X, :Z, :Y], [1, 3, 4], 0.5)
add!(psum, :Z, 2, 1.3)
```

Convention Box 1: Bit endianness

We notate binary numbers with the least significant bit *on the right* (so called "little-endian"). Where relevant, we use a subscript suffix to notate when a number is a base-n numeral, for example

$$6_{10} = 110_2.$$

Note this is the reverse of the big-endian convention so far adopted for notating Pauli strings, whereby the *leftmost* qubit is the lowest index, e.g.

$$ZX \equiv Z \otimes X \equiv Z_1 X_2.$$

Note too that we index from ***one*** (not zero) for both Pauli strings *and* bits, such that the bit at index one of 110_2 is zero. This further aligns with `Julia`'s indexing convention which starts from one (not zero).

about low-level optimizations. This section will lift the veil to some of the challenges under the hood: the choice of suitable data structures, memory management, control flow, parallelization and compilation. Such considerations are essential in the development of high-performance simulators like `PauliPropagation.jl`.

A. Representing a Pauli string

This manuscript has so far taken for granted the ability to symbolically denote a Pauli string, such as $X_1 Y_3 Z_4$, and perform operations on them. We here describe how to perform this in a program, and the numerous considerations for memory and runtime efficiency.

Let us first summarize the relevant properties and demands of the Pauli string structure in the Pauli propagation framework.

i. *Limited alphabet:* A Pauli string has an "alphabet of *four*: *I*, *X*, *Y* and *Z*. Since n bits can enumerate 2^n symbols, a single Pauli can be optimally represented by two bits (a "dibit or "crumb). In contrast, an intuitive but wasteful representation as an ASCII character in a string like `"I"`, `"X"`, `"Y"`, `"Z"` would consume eight bits. We seek a representation which takes advantage of the limited alphabet to save memory.

ii. *Bounded length:* Our Pauli strings contain only as many operators as there are qubits in the simulated circuits. This empowers us to bound the number of represented operators in the string, for example to 128 qubits. Such a "compile-time" bound enables the use of a static structure (rather than some dynamically growing structure) which asymptotically improves some operations and invites significant compiler optimizations.

iii. *Primitive:* During Pauli propagation, we will instantiate extremely many Pauli strings, motivating the use of a primitive type (like an integer) over an object (like a string or a custom class). This shrinks memory costs and avoids the substantial runtime overheads of object construction and garbage collection. It also permits a Pauli string to fit into a single CPU register for faster processing, and may enable a clever compiler (like `Julia`'s) to auto-vectorize looped processing, such as when we process all strings within a large Pauli sum.

iv. *Fast operations:* Operations between Pauli strings, like products and commutators, must be *as fast as possible*—ideally in effectively $\mathcal{O}(1)$ time—since they are repeatedly performed in "hot loops", i.e., performance-critical loops over many fast operations. This necessitates the use of a type compatible with single-instruction-multiple-data (SIMD) processing, such that multiple Paulis in a string are processed simultaneously.

v. *Individually addressable:* The Paulis in our string must be *individually addressable* so that we can quickly obtain a Pauli at a particular index, ideally in $\mathcal{O}(1)$ time. This is necessary to implement an operation on a particular qubit or site of a string.

vi. *Quickly comparable:* Pauli strings must be *quickly comparable* to keep merging-BFS efficient. That is, we should be able to in $\mathcal{O}(1)$ time decide whether two strings are identical. In Sec. IV C, we will additionally motivate that our Pauli string type is quickly "hashable" for integration with containers like dictionaries, or can be *sorted* for containers based on arrays.

For these reasons, a natural choice of type for a Pauli string is one single *unsigned integer*; the representation of the string as a base-four numeral. In this picture, each bit pair denotes a Pauli operator, and half as many Paulis can be represented as the integer type has bits. For example,

$$\mathrm{ZYX} = 27_{10} = 011011_2 = \underbrace{00}_{\mathrm{I}} \cdots \underbrace{00}_{\mathrm{I}}\ \underbrace{01}_{\mathrm{X}}\ \underbrace{10}_{\mathrm{Y}}\ \underbrace{11}_{\mathrm{Z}}.$$

We choose to label the Pauli strings and bitstrings according to Convention Box 1, and denote the base of the numeral via gray subscripts for clarity. The reason there are seemingly unnecessary identity Paulis on the left is because, depending on the bit-length of the chosen integer type, bits in excess of $2n$ will remain zero. We ensure optimal efficiency by choosing an integer precision inferred from the number of qubits in the user's simulated circuit, without compromising the compile-time bounds. This is done with type templating: At the time of instantiation of a `PauliString` or `PauliSum` object, we check whether suitable built-in integer types exist and otherwise generate the next higher precision type. `Julia`'s multiple dispatch will just-in-time (JIT) recompile any propagation-related function which is specialized upon the type, to run as fast as with a built-in, hardcoded integer type. All propagated Pauli strings will use the same precision of integer, avoiding type instability.

The representation of Pauli strings as unsigned integers is adopted by other simulators such as `QuEST` [71], `Stim` [51], as well as the related `PauliStrings.jl` [57] and Ref. [16]. Some works maintain *two* unsigned integers which each represent or "mask" the location of X and Z operators in a Pauli string. The location of a $iY = -X \cdot Z$ operator is consequently indicated by a one-bit in the same location of both masks. This representation bears closer similarity to the design choices in the stabilizer formalism [48]. Both representations satisfy all our previous considerations and, as we will discuss in the next section, enable rapid processing of Pauli strings with bitwise operations. We focus on the single integer representation for the remainder of this paper since this is adopted by `PauliPropagation.jl`.

B. Processing a Pauli string

Encoding Pauli strings as binary integers is not only memory efficient, but permits us to process them using so-called *bitwise* operations. Such low-level operations, illustrated in under the Hood Box 1, correspond one-to-one with CPU instructions and are processed in a dedicated arithmetic logic unit (ALU). Requiring a single clock cycle, they are extremely efficient—the fastest among arithmetic instructions (such as the addition of two integers), which are again orders of magnitude faster than floating-point operations (such as the addition of two decimals). This is regardless of the number of represented Pauli operators in the string, provided that the integer type fits within a single CPU register. In this section, we highlight several operations upon Pauli strings that are implemented by `PauliPropagation.jl`. These are similarly spirited but distinct from the optimizations recently added to `PauliStrings.jl` which performs analogous schemes on two distinct masks.

Under the Hood Box 1: Bitwise operations

Bitwise operations are defined by their operation directly upon the bits of a number. Particularly relevant to processing binary-encoded Pauli strings are: AND (&), OR (|), XOR ($\oplus$), left-bitshift (<<) and right-bitshift (>>). For our purposes, their main utility is that they simultaneously operate upon every corresponding pair of bits of the operands. We can in-fact uniquely define the above bitwise operations by their actions upon operands a, b

$$a = 0011_2 = 3_{10} \qquad a \,\&\, b = 0001_2$$
$$b = 0101_2 = 5_{10} \qquad a \,|\, b = 0111_2$$
$$a \oplus b = 0110_2.$$

AND, OR and XOR are useful for "masking" ; i.e, retaining, setting to one, or flipping specific bits, respectively. To do so, one prepares another integer m (the mask) with one ("set") bits at the locations of the targeted bits. For example, the mask $m = 0\cdots01001_2$ targets the *first* and *fourth* bits. The integer $a \,\&\, m$ retains only the targeted bits of a, $a \,|\, m$ has additional ones at the target indices, and $a \oplus m$ flips all targeted bits in a.

Preparing a mask is done with the left-bitshift operation, $a{<<}n$, which simply left-shifts the set bits of a by the specified number of positions n. Example

$$0011_2{<<}1 = 0\mathbf{11}0_2$$
$$0011_2{<<}3 = \mathbf{1}000_2.$$

Note, the leftmost $nbits$ of the m in a's full binary representation are discarded. Otherwise

$$a{<<}n \equiv a \times 2^n \qquad \text{for } a < 2^{m-n}.$$

The mask $m = 0\cdots01101_2$ can be prepared via

$$m = (1_{10} << 0) \,|\, (1_{10}{<<}2) \,|\, (1_{10}{<<}3).$$

Preparing a Pauli string. The integer 0 with bits $00\cdots00\ 00_2$ represents the all-identity Pauli string. Preparing a string where the i-th Pauli is *not* the identity requires overwriting the bits at indices $2i$ and $2i-1$ with the decimal value $1, 2, 3$ (for X, Y, Z respectively). We do this by *bit-shifting* the decimal value, as described in under the Hood Box 1, by *twice* the index. For example,

$$Y_3 \equiv 00\,10\ 00\ 00_2 = 10_2{<<}4 = 2_{10}{<<}4.$$

In general, the Pauli string with a single non-identity Pauli $\sigma \in \{1,2,3\}$ on site i is computed by $\sigma_{10}<<2(i-1)$. Preparing a Pauli string with *multiple* non-identity Paulis involves *masking* their individual bitmasks using the bitwise OR operator, also described in under the Hood Box 1. For example

$$Z_1 Y_3 \equiv \texttt{00 10 00 11}_2 = (2_{10}<<4)\,|\,(3_{10}<<0).$$

Setting a Pauli. Modifying the i-th Pauli in a Pauli string instantiated with integer `P` requires overwriting the bits at indices $2i$ and $2i-1$. When $\texttt{P} = 0$, this can be done using the bitwise OR operator as above, but otherwise requires first *unsetting* the targeted bits to zero. This is achieved using the *bitwise negation* operator ($\sim$) which unconditionally flips all bits. Consider

$$m = 00\cdots\overbrace{11}^{2i,2i-1}\cdots 00 = 11_2<<(2(i-1)),$$

which masks the targeted bits. The integer $\sim m = 11\cdots00\cdots11$ masks *every other* than the targeted bits. As such $\texttt{P}\,\&\,(\sim m)$ retains all ones of `P` *except* at the targeted bits which are overwritten to zero; this is equivalent to overwriting the i-th operator with identity I. The desired Pauli ($\sigma \in \{0,1,2,3\}$) can then be set using bitwise OR as before,

$$\texttt{P}' = (\texttt{P}\,\&\,(\sim m))\,|\,(\sigma<<(2(i-1)).$$

Extracting a Pauli. Consider a Pauli string instantiated with an unsigned integer `P`. The i-th Pauli operator $\texttt{P}^{(i)} \in \{0,1,2,3\}$ (which encodes respective symbols I, X, Y, Z) is constituted by the two bits at indices $2i$ and $2i-1$. We can extract the Pauli by bit-shifting this pair to the lowest two positions, then *masking* them through the bitwise AND operation; see under the Hood Box. 1 for an elaboration. The individual Pauli is given by

$$\texttt{P}^{(i)} = (\texttt{P}>>2(i-1))\,\&\,11_2, \tag{44}$$

where `>>` is a *right*-bitshift (the opposite of the left shift).

Products of Paulis. Obtaining the product of two Pauli strings is a common subroutine of Pauli propagation, e.g., when applying Pauli rotations. We recall that the product of two single-qubit Paulis $P \in \{X, Y, Z\}$ is given by

$$P_\alpha P_\beta = \delta_{\alpha\beta} I + i\epsilon_{\alpha\beta\gamma} P_\gamma, \tag{45}$$

where $\epsilon_{\alpha\beta\gamma}$ is the Levi-Civita tensor. It follows that when two Pauli strings are encoded as unsigned integers `P1` and `P2`, then the encoded product `P3` (without prefactor) is simply given by bitwise XOR:

$$\texttt{P3} \propto \texttt{P1} \oplus \texttt{P2}. \tag{46}$$

For example, $XY \cdot ZI = -\mathrm{i}\,YY$ and $\texttt{0110}_2 \oplus \texttt{1100}_2 = \texttt{1010}_2$. However, the prefactor of `P3` is also needed by Pauli propagation to obtain the coefficients of the final Pauli sum and evaluate the ultimate inner-product. We currently evaluate a cached Levi-Civita tensor for each pair of bits in `P1` and `P2`, necessitating a loop over the sites affected by the Pauli product. Note that a constant-time bitwise evaluation of the sign of the *commutator* of two Pauli strings is known [57].

Indexing into look-up tables. A core subroutine of Pauli propagation is mapping a *subset* of $\mathcal{O}(1)$ sites in a Pauli string to one or more other substrings. This is beneficial, for example, when applying general PTMs representing circuit operations. While sometimes performed with hard-coded logic, such as when implementing a Pauli rotation's PT map as per Eq. (14), output Pauli substrings are often instead obtained using *look-up tables*. Consider a function f, e.g., here corresponding to a Clifford gate, that acts on k qubits and thus maps k-qubit substrings to another. That is, we have

$$f\colon \mathcal{P}^{(k)} \to \mathcal{P}^{(k)}. \tag{47}$$

When k is tractably small, we can pre-compute $f(\texttt{P})$ for all $\texttt{P} \in \{0,\ldots,2^k-1\}$, store the outputs in an array $\boldsymbol{f}$, and thus can replace future evaluations of f with the lookup $\boldsymbol{f}[\texttt{P}+1]$. That is, we utilize that our Pauli substrings are instantiated as unsigned integers to "directly address" $\boldsymbol{f}$, offsetting by $+1$ due to `Julia`'s indexing convention.

For example, consider the PTM of the Hadamard gate

$$\mathrm{PTM}_H = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}. \tag{48}$$

While the action of PTM_H on the Pauli string X can be computed via matrix-vector multiplication, e.g.,

$$\mathrm{PTM}_H \cdot (0,1,0,0)^T = (0,0,0,1)^T \equiv Z, \tag{49}$$

it is faster to access a pre-computed lookup array

$$\boldsymbol{f}_H \coloneqq [I, Z, -Y, X], \qquad \boldsymbol{f}_H[1+1] = Z, \tag{50}$$

where we use the fact that X is instantiated by the unsigned integer $\texttt{P} = 1$, and offset by $+1$ since indexing from one.

Checking commutation. It is critical to quickly determine whether a Pauli string commutes with another, given that this check is critical to whether Pauli rotations induce path branching within "hot loops" [see Eq. (14)]. Two strings commute only if an *even* number of their individual Paulis *anti*-commute, which we can quickly determine using $\mathcal{O}(1)$ bitwise operations.

Consider single-qubit Paulis a and b, respectively, instantiated by two-bit integers with bits $(\texttt{a}_2, \texttt{a}_1)$ and

$(\mathtt{b}_2, \mathtt{b}_1)$. Since single-qubit Paulis only commute when identical or one is the identity, it turns out that two single-qubit Paulis a and b *anti*-commute when $c = 1$, where

$$c \coloneqq (\mathtt{a}_1 \,\&\, \mathtt{b}_2) \oplus (\mathtt{b}_1 \,\&\, \mathtt{a}_2). \tag{51}$$

This can be understood since c is only ever zero (implying commutation) when the XOR operands are equal, occurring when $a = b$ or one of a or b is identity (both AND operands include a zero and so return zero, yielding $0 \oplus 0 = 0$).

The above quantity is for a single site, but we can compute it for every corresponding pair of Paulis among many-qubit Pauli strings `A` and `B` by use of specialized bit masks. We prepare (at compile-time) an alternating mask

$$m_R = 01 \cdots 0101_2, \tag{52}$$

such that $\mathtt{A} \,\&\, m_R$ isolates every second bit of `A`; retaining only the *right* bit of each individual Pauli within, interspaced by zero bits, i.e.,

$$\mathtt{A}_R \coloneqq \mathtt{A} \,\&\, m_R \equiv (\cdots, 0, \mathtt{A}_3, 0, \mathtt{A}_1).$$

The mask $m_L = 10 \cdots 1010_2 = m_R{<<}1$ similarly retains all *left* bits, i.e.,

$$\mathtt{A}_L \coloneqq \mathtt{A} \,\&\, m_L \equiv (\cdots, \mathtt{A}_4, 0, \mathtt{A}_2, 0).$$

We evaluate $\mathtt{B}_R$ and $\mathtt{B}_L$ similarly, then bit-shift them by one so that their non-zero bits of interest align with those of $\mathtt{A}_L$ and $\mathtt{A}_R$, respectively. Therefore, the expression

$$C \coloneqq (\mathtt{A}_R \,\&\, (\mathtt{B}_L{>>}1)) \oplus (\mathtt{B}_R \,\&\, (\mathtt{A}_L{>>}1))$$

simultaneously evaluates the left-hand-side of Eq. (51) for all Paulis within `A` and `B`. The resulting integer

$$C \equiv (\cdots, 0, c_3, 0, c_2, 0, c_1)$$

consists of bits $c_i = 1$ indicating that the i-th individual Paulis anti-commute. The strings `A` and `B` therefore commute when C contains an even number of one bits, which can ordinarily be computed in a *single* bespoke CPU instruction such as the inbuilt `count_ones()` in `Julia`.

Computing Pauli weight. Counting the number of non-identity Paulis in a Pauli string, necessary when performing Pauli weight truncation, is equivalent to counting the number of Pauli bit-pairs that are not $\mathtt{00}_2$. Consider a single-qubit Pauli `b` with bits $\mathtt{b} = (\mathtt{b}_L, \mathtt{b}_R)$. The expression $\mathtt{b}_L \,|\, \mathtt{b}_R$ is only zero when both bits are zero and so `b` is the identity. Consider now a many-qubit Pauli `B` whereby `B >> 1` aligns the *left* bit of the Pauli bit pairs with the *right* bits

$$\mathtt{B} = (\cdots, \mathtt{B}_3, \mathtt{B}_2, \mathtt{B}_1)$$
$$\mathtt{B}{>>}1 = (\cdots, \mathtt{B}_4, \mathtt{B}_3, \mathtt{B}_2).$$

As such, the bits of $\mathtt{B} \,|\, (\mathtt{B}{>>}1)$ indicate (with a set-bit) where neighboring (and overlapping) pairs of bits contain at least one set-bit. We finally isolate every second bit (one from each non-overlapping neighboring pair) using the alternating mask m_R in Eq. (52). Hence we are left with the bitwise expression

$$W = m_R \,\&\, (\mathtt{B} \,|\, (\mathtt{B}{>>}1)).$$

Computing XY weight. Counting the number of X or Y Paulis (excluding Z) in a Pauli string follows an almost identical approach to above, except that the bitwise OR operation is replaced with an XOR. This is because $X = \mathtt{01}_2$ and $Y = \mathtt{10}_2$ have different first and second bits, while $I = \mathtt{00}_2$ and $Z = \mathtt{11}_2$ have repeated bits. Counting, for example, YZ weight is even simpler, because one only needs to count how many bit pairs carry `1` on the left bit.

C. Representing a Pauli sum

The previous sections motivated our representation of individual Pauli strings as unsigned integers. But Pauli propagation evolves rapidly growing *weighted sums* of Pauli strings, necessitating the design of a *container* data structure to store a large ensemble of the above discussed types, each coupled with a coefficient. This section considers several choices of canonical data structures (introduced in UTH Box 2) as motivated by the below list of considerations, and the performance considerations introduced in UTH Box 3. We note that this list focuses on merging-BFS simulations, the main focus of `PauliPropagation.jl` currently, and may differ for other variants of Pauli propagation. Below, we take N to be the number of Pauli strings presently in the container, and we will often discuss *amortized* asymptotic scaling, which can be understood as the *worst-case averaged* over many (towards infinite) repetitions.

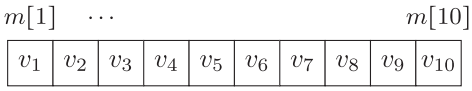
i. *Memory efficient*: Propagation sees the ensemble of evolving Pauli strings rapidly grow, requiring the simultaneous storage of an extreme number of Pauli strings—easily in the billions. Due to the small memory footprint of each string represented as an unsigned integer, a non-negligible memory overhead *per string* will multiplicatively exacerbate the memory costs. This encourages use of a *contiguous memory* data structure with a zero amortized overhead per element, such as an array or hash table.

ii. *Compact*: There are 4^n unique Pauli strings with n sites. When each is stored as an n-bit unsigned integer (ignoring storage of the coefficients), collectively storing all possible strings (like a full-state

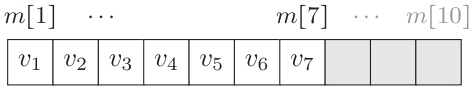
Under the Hood Box 2: Data structures

We here summarize several data structures relevant to Pauli propagation. Let $m[i]$ signify the i-th memory address and white and gray boxes indicate filled and empty memory slots, respectively.

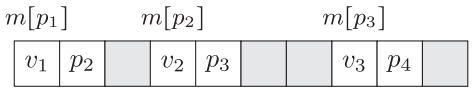
Arrays store a fixed number of directly addressable items contiguously in memory.



Lists permit appending items in amortized $\mathcal{O}(1)$, inserting items at specific locations in $\mathcal{O}(N)$, and can be implemented by an array with extra allocated memory. When exhausted, an entirely new array with additional capacity must be allocated and copied into, in $\mathcal{O}(N)$ time. In `Julia`, a 1D `Array` curiously follows this definition.

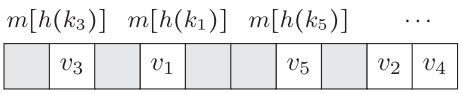


Linked lists can forego direct indexing by storing values alongside the memory address p of the next value. This allows $\mathcal{O}(1)$ insertion and deletion, but makes memory *non-contiguous*, sabotaging caching performance and parallel enumeration.



Sorted lists improve search from $\mathcal{O}(N)$ to $\mathcal{O}(\log(N))$. Insertion is $\mathcal{O}(N)$ or $\mathcal{O}(\log(N))$ when implemented with arrays or links, respectively.

Hash tables are commonly used to implement *dictionaries*. They store values of `(key,value)` pairs in a larger fixed-size array, at addresses informed by *hashing* the keys with a many-to-one function h. Assuming the "load factor" (number of values divided by array capacity) is kept small, insertion is amortized $\mathcal{O}(1)$. When the load factor grows too large, the array is resized and all items are re-inserted at cost $\mathcal{O}(N)$.



Under the Hood Box 3: HPC considerations

Different implementations of the same algorithm can have wildly different performances depending on the code's use of control flow and memory movement. Here we summarize some considerations relevant to Pauli propagation.

Branching of the control flow due to `if` statements is fine in most contexts but can damage performance when inside hot loops. This is because conditional instructions interfere with the CPU's use of instruction-level parallelism (ILP) which may resort to "branch prediction". When a CPU incorrectly guesses the condition outcome, it must undo speculatively executed instructions, interrupting execution.

Caching copies a subset of main memory into a smaller cache to speed up future accesses. A modern CPU has progressively larger caches as intermediaries to RAM, which are updated *in batch*; one "cache line at a time. The main memory only needs to be consulted when the CPU requests data not already in a cache (a "miss). Irregular access to data, like out-of-order indexing of an array, encourages expensive cache misses. HPC code accesses data in a predictable, contiguous manner.

Multithreading distributes a task between concurrent cores of a CPU(s). Efficient parallelization requires the task is non-trivial (to outweigh thread invocation costs), can be homogeneously divided between cores, and that the divided tasks are mostly independent.

simulator) would require $n\,4^n$ bits and quickly become prohibitive. For example, $n = 16$ qubits would require at least 8 GB to even represent the Pauli strings, not to speak of the 64 GB for their real-valued coefficients. Yet, we seek to push simulation far beyond this regime, to circuits of hundreds of qubits which require simultaneous storage of a tractable but unpredictable subset of all possible strings. As such, we require that our ensemble data structure is *compact*: that it can store a small fraction of all possible elements in a space without allocating memory for the full space. This eliminates "directly addressed" structures which map Pauli strings one-to-one with memory locations, like used in full-state simulation.

iii. *Dynamically sized*: It is typically hard to predict the total number of Pauli strings that will need to be propagated at the beginning of a simulation. As such, we cannot reliably pre-allocate our container, and must permit our ensemble to grow in tandem with our attempts to reduce costs using truncation. This requires that our container can be freely dynamically sized (like a linked list with $\mathcal{O}(1)$ insertion), else *infrequently* and *efficiently* re-allocated to a larger, static size when necessary (like an array with $\mathcal{O}(N)$ reallocation). It discourages use of non-linked structures where re-allocation requires re-insertion of

every element in non-constant time, such as trees implemented with static arrays.

iv. *Efficient search*: A core subroutine of Pauli propagation via merging-BFS is *searching* the container for a particular Pauli string, in order to merge with it. Since this is performed almost every time an operator is applied to a Pauli string, it is essential that this search is fast. Our use of unsigned integer primitives to represent Pauli strings enables us to hash or compare Paulis in $\mathcal{O}(1)$ time, enabling amortized $\mathcal{O}(1)$ and $\mathcal{O}(\log(N))$ searches inside hash tables and non-linked sorted lists, respectively.

v. *Efficient insertion, modification and deletion*: It is clear a rapidly growing ensemble should be stored in a container which supports efficient insertion of new elements. Yet our needs are greater. Pauli propagation sees the corpus of contained Pauli strings *change* frequently. As such, runtime is dominated by the costs of repeatedly enumerating and modifying the billions of contained strings. This might incentivize a data structure supporting in-place mutation, though this tends to be at odds with our other requirements. Thus, we might instead demand it is efficient to *remove* and *insert* Pauli strings, discouraging the use of non-linked sorted lists with $\mathcal{O}(N)$ penalties, and encouraging the use of hash tables or trees with amortized $\mathcal{O}(1)$ and $\mathcal{O}(\log(N))$ costs. Efficient removal is also crucially important for the truncation of Pauli strings.

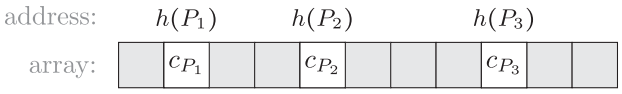
vi. *Efficient iteration*: Pauli propagation requires repeatedly iterating over or enumerating the Pauli sum to process every Pauli string it contains under the action of an operation. It is straightforward to enumerate in $\mathcal{O}(N)$ time using an array, linked-list, hash table or tree, though their runtimes can differ greatly. This is because only arrays store their items in contiguous memory (see UTH Box 2), supporting fast caching, parallel enumeration, and enabling vectorization. Hash tables, for example, are multiplicatively slower to enumerate than arrays, as informed by the ratio of filled slots called the *load factor*. Trees and linked structures can be slower still due to storing items at unpredictable memory locations.

vii. *Efficient union*: It is not straightforward to parallelize the processing of some of the mentioned data structures. We can sometimes work around this by partitioning our structure into multiple, smaller instances, process each independently and concurrently, before re-merging them. This is worthwhile when the processing time outweighs the partitioning and merging costs, and encourages use of a data structure which is efficiently *unioned*.

Combining all considerations, we have so far established the hash table and arrays to be very appropriate structures. They have different strength and weakness profiles, and one may be superior over the other depending on the quantum circuit, the task, or the computational hardware.

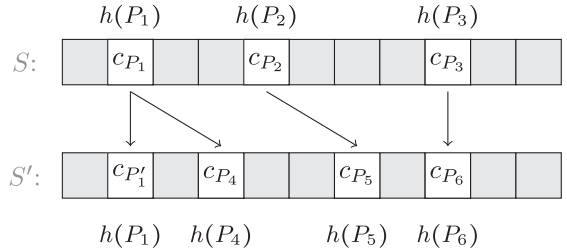
1. Hash tables

Hash tables are how native dictionaries in the `Julia` language are implemented. Our keys are Pauli strings which, since instantiated with unsigned integers, are hashable in $\mathcal{O}(1)$ time (The diligent reader may wonder how an unsigned integer with at least $2n$ bits can be hashed in $\mathcal{O}(1)$ time. Indeed the ultimate number of CPU clock cycles to perform a single bitwise instruction technically grows *linearly* with the bits in the operands, once the CPU register size has been exceeded. In practice the linear scaling is so small as to be insignificant, especially when considering the SIMD nature of CPU registers, hash-table implementations which cache hashes of keys, and sub-linear hash functions. In such cases, we can say the hash-table lookup time is effectively $\mathcal{O}(1)$ when amortized over many lookups.). The stored values are the associated coefficients of the strings in the represented Pauli sum. For example, sum $S = c_{P_1}P_1 + c_{P_2}P_2 + c_{P_3}P_3$ may be stored as:



where $h(P_i)$ denotes the hash function called upon the integer representation of Pauli string P_i. Beware: this sketch is a simplification. Each value is actually stored alongside its corresponding key (the associated Pauli string) as is necessary since the hash function is generally non-invertible and key-equality checks will be common.

Inserting, removing and searching for Pauli strings are all amortized $\mathcal{O}(1)$, permitting rapid modification of strings under the action of an operator. For example, S above may be modified to $S' = c_{P'_1}P_1 + c_{P_4}P_4 + c_{P_5}P_5 + c_{P_6}P_6$, resembling the following:



The next section elaborates on this process, which in fact makes use of *two* hash tables. Meanwhile, parallel processing is possible by using a *concurrent* hash table (making contentious actions "atomic"), or by division into separate, concurrently modified hash tables before a final, serial union.

There are a few drawbacks to choosing hash tables. First, the aforementioned operations are efficient when the load factor is small, i.e., when the number of empty hash table slots is low. Typically, a load factor of ≈ 0.5 is said to be desirable, necessitating that the internal array capacity is at least *twice* the number of Pauli strings contained within. Since a `Julia` hash table quadruples in capacity when the load factor exceeds 0.75, the internal array can be up to *five times* larger than necessary to store the strings contiguously.

Second, enumerating the hash table to process Pauli strings is suboptimal. This is because either the redundantly large internal array with empty entries is iterated over, or a separate list of keys is kept, requiring invoking superfluous re-hashing and control flow branching. These issues may be allayed by caching key hashes (sometimes called "tokens") or by hybridization with arrays; facilities offered by `Julia`'s `Dictionaries.jl` library.

And third, many implementations of hash maps are not *thread safe*, i.e., inserting elements is often sequential and not easily parallelizable. This can be solved via *locking* of hash map segments, or specialized lock-free data structures. Most general implementations in `Python` or `Julia`, however, remain serial. As such, there remain alternative choices of data structure to represent the Pauli sum which are optimal for different situations.

`PauliPropagation.jl` presently balances these considerations by instantiating its `PauliSum` type as a hash table in the form of `Julia`'s `Base.Dict` dictionary. This offers asymptotically favorable support for all circuit families, with simple and fast single-threaded defaults even for custom gates.

2. Arrays

Pauli sums can also be represented via two arrays, one array containing all Pauli strings (e.g., `Array{UInt32}` for $\leqslant 16$ qubits) and one array containing their coefficients (often `Array{Float64}`). When using a single thread, some elementary operations may be slower and others faster than with hash tables or dictionaries, but our design choices for arrays aim to maximize parallelization on CPU or GPU. This is particularly great news for running on HPC clusters (which we would like to use for their larger memory), because computations will be significantly sped up by their many CPU threads.

Arrays are highly advantageous for circuits consisting predominantly of non-branching gates such as Clifford operations or Pauli noise channels. This is typical, e.g., in quantum error correction circuits [72]. Because Cliffords map Pauli strings one-to-one, the number of Pauli strings is conserved, and no merging is necessary, thus relaxing our requirements for fast insertion, deletion and lookup. The contiguous data structure enables rapid enumeration and, more importantly, *in-place mutation*. A Pauli string can be modified by merely overwriting its array element, which is considerably faster than removal and re-insertion into a hash table despite the asymptotic equivalence. Modifying all Pauli strings also becomes *embarrassingly parallel*, and is straightforwardly accelerated with several CPU threads or GPUs.

One may wonder how branching gates and merging strings are efficient in this data structure. The key insight is that the arrays can be *sorted* by the numeric value of the Pauli string integer. Comparison-based sorting will take time $\mathcal{O}(N \log N)$, but there exist algorithms such as *Radix sort* that sort integers in time $\mathcal{O}(N)$ with non-negligible prefactors. Searching the sorted Pauli array to determine where to merge or insert new elements then takes time $\mathcal{O}(\log N)$. This is the approach taken in SPD [11,16]. The naive $\mathcal{O}(N)$ cost of insertion can be avoided by collecting the k new Pauli strings in a smaller, temporary array which is merged in-batch to the full ensemble array in $\mathcal{O}(N + k)$.

Another approach (indeed the one we take) is to *pre*-allocate and *over*-allocate all arrays. By sizing all arrays to an explicit capacity $M \geqslant N$, we amortize the $\mathcal{O}(N)$ resizing events and associated garbage collection over several gates, only kicking into action when a branching operation would result in over M (unmerged) Pauli strings.

To coordinate threads and perform massively parallel work, we leverage paradigms from the HPC and GPU communities, i.e., we write all subroutines as *kernels*. A kernel is a function that is defined to act on an array at index `i` without interfering with how it acts on index `i + 1`. This is necessary to allow all CPU and GPU threads to work independently of each other.

Our most frequent subroutine is commonly called *stream compaction* or simply a *filter* operation. Let us exemplify two important variants of this subroutine for truncating and merging a Pauli sum. We define `P` and `C` as the arrays carrying the Pauli strings and coefficients, and `P2` and `C2` their pre-allocated, auxiliary counterparts. We further allocate a boolean flag array `F` and an index array `I` of the same length. All loops and parallel operations on these arrays are written with the `AcceleratedKernels.jl` library that compiles all kernels to CPU or GPU, depending on which device the incoming arrays are allocated.

Truncate. Presume the function `iskept()` computes whether a term in the Pauli sum is kept, i.e., evaluates to one if kept and is zero otherwise. We then perform the following three steps:

1. Flag the elements that are not truncated.
```
for i = 1...N:
  F[i] = iskept(P[i], C[i])
```
2. Perform a parallel cumulative sum over `F` and write the results into `I`, `cumsum(F)` $\rightarrow$ `I`.
3. Write the untruncated elements.
```
for i = 1...N:
  if F[i] == true:
    P2[I[i]] = P[i]
    C2[I[i]] = C[i]
```

These steps will place the untruncated terms in `P2` and `C2`, where conveniently `I[N]` denotes the new number of valid terms, i.e., the new N. Any residual or "trash" terms in the valid region of the arrays from previous iterations are completely overwritten, as will be the remaining terms in `P` and `C` in the next iteration. It suffices to relabel `P2` and `C2` as the main arrays and proceed with the propagation.

Crucially, the pre-allocated flag array allows threads to independently check the truncation condition, and the index array carrying the cumulative sum allows threads to independently transfer un-truncated Pauli strings and coefficients into the correct slot of `P2` and `C2`, respectively. With a suitable implementation of a parallel cumulative sum, the entire procedure is *thread safe*.

Merge. We start by performing a sort permutation of `P` and `C` by the integer-valued entries of `P`. Importantly, this causes duplicate strings to be next to each other. Then perform the following three steps:

1. Flag the beginning of a new domain.
```
for i = 1...N:
  if i == 1:
    F[i] = true
  else:
    F[i] = (P[i] != P[i-1])
```
2. Perform a parallel cumulative sum over `F` and write the results into `I`, `cumsum(F)` → `I`.
3. Move the unique strings and accumulate their coefficients.
```
for i = 1...N:
  if F[i] == true:
  P2[I[i]] = P[i]
  C2[I[i]] = C[i]
  j = i + 1
  while (j ≤ N) and (P[j] == P[i]):
    C2[I[i]] += C[j]
    j += 1
```

Now `P2` and `C2` represent the merged Pauli sum. On GPUs, the inner `while` loop in step 3 is likely suboptimal if terms can be duplicated dozens or hundreds of times, but this can be solved by several sweeps with finite-sized loops over the inner index `j`.

Our library `PauliPropagation.jl` provides array-based capabilities starting version `0.7` using the `VectorPauliSum` type (given that `Vector` in `Julia` is an alias for a 1D `Array`). Figure 7 demonstrates the performance improvement we can attain for quantum circuits consisting exclusively of Pauli rotations on an Intel Xeon Platinum 8360Y CPU node. Circuits including Clifford gates would be sped up significantly more, yet we observe up to an order of magnitude improvement using ∼32 threads even for Pauli rotations. This corresponds to applying a gate to a physical observable (not a random Pauli sum) with 10^8 Pauli strings and pre-allocated memory in approximately 1 s, including merging and truncations. As is common in parallel computing, we observe strongly diminishing returns or even performance degradation when splitting the work over too many threads. Note also that local machines may have fewer and faster threads, in turn reducing the gap between single-threaded and multi-threaded propagation.

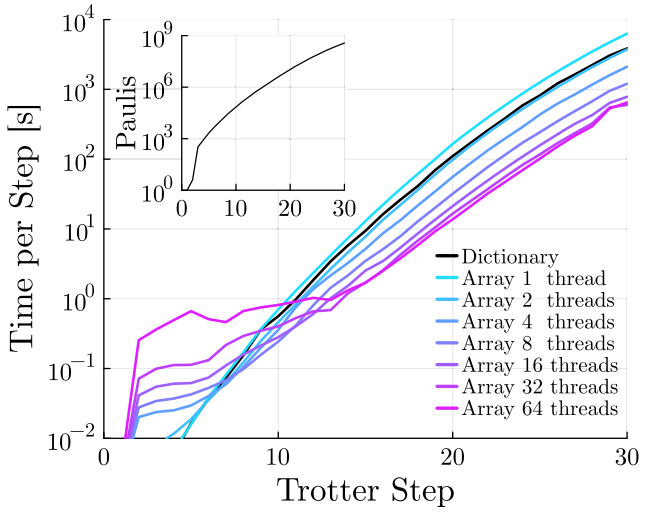


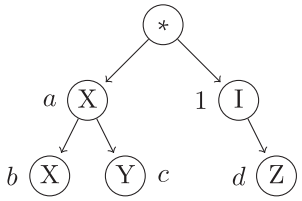
FIG. 7. Benchmarking multithreading capabilities in `PauliPropagation.jl`. We again consider the Trotterized dynamics of a 6×6 tilted-field Ising model with $dt = 0.05$, 132 Pauli rotations per step, the $Z_{21}Z_{22}$ observable, a coefficient cutoff of 2^{-20}, and no Pauli weight truncation. Multi-threaded array-based propagation here achieves up to an order of magnitude speedup over single-threaded dictionary-based propagation. We refer to Fig. 6 for the expectation values.

A downside of this array-based approach is that it is more challenging to write correct and efficient code. This is particularly true when targeting GPUs because their performance comes at the cost of additional constraints. For example, Pauli strings on many qubits (in our case >32) can no longer be represented by a single long integer due to GPU hardware architecture. Instead, they need to be split into chunks of `UInt32` or `UInt64`. Our GPU acceleration is still in an early stage, and future versions of `PauliPropagation.jl` will improve on it. However, when adhering to all architectural constraints, even custom extensions to our library compile to multi-threaded CPU or GPU code via `Julia`'s multiple dispatch and the `AcceleratedKernels.jl` library. Future engineering efforts by the community will also inform us whether more powerful parallelization approaches on CPU exist (consider, e.g., Ref. [23]), and how GPU acceleration can be fully leveraged.

3. Trees

Another interesting choice of data structure is a *tree* where each node stores an individual Pauli and a scalar, and the path from root to a leaf node encodes a Pauli string. A string's coefficient is the product of the scalars among its

constituent nodes. For example, the string $(ab)XX + (ac)XY + dIZ$ might be stored as



The tree has depth n (the number of qubits), and the i-th layer contains all Paulis of the i-th site. Strings are searched, inserted or removed in $\mathcal{O}(n)$ regardless of the number stored, and enumeration of all N leaf nodes remains $\mathcal{O}(N)$ though will be practically slower than enumerating an array. Modifying a node's Pauli or scalar simultaneously updates all weighted strings encoded by all descendant nodes. Such a structure (without scalars) has been used in algorithms to decompose dense matrices into Pauli strings [73], but could also be used in Pauli propagation to update many Pauli strings *in batch* through tree *transformations*. For example, a single-qubit Clifford gate upon the leftmost Pauli requires modifying only the first layer of nodes, avoiding processing all contained strings. Many-qubit and non-Clifford gates however require more complicated traversals and transformations, with performance dependent upon the structure of the tree and the distance of the affected site indices to its root. The use of trees was reported in Ref. [44] although only as a means of enumerating the strings, and was not used for accelerating the simulation of gates.

D. Implementing a gate

Recall from Theory Box 1 that the action of an n-qubit channel $\mathcal{E}$ upon a Pauli string can be captured by the operator's Pauli transfer matrix. For a generic k qubit channel, computing this $4^k \times 4^k$ matrix takes $\mathcal{O}(k16^k)$ time [74] and its subsequent operation upon a Pauli sum in any representation takes at least $\mathcal{O}(16^k)$. This is monstrously slow and memory inefficient. Such a method fails to leverage that for most canonical channels, the PTM is sparse, few-branching (i.e. contains few non-zero identities in any given column) and known at compile time. Here we will show how many gates and channels of interest to the quantum computing community offer optimized, specialized simulation.

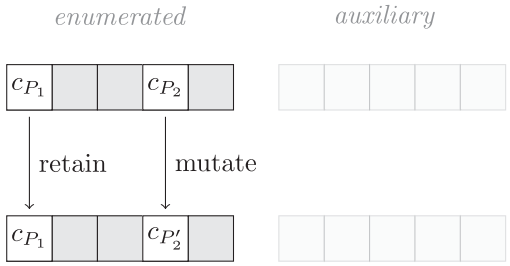
An essential motivation for the representations of `PauliString` and `PauliSum` described above was the speed to modify a Pauli string under the action of a circuit operator. We here describe precisely *how* a `PauliSum` is efficiently modified by the family of operators presented at a higher-level in Sec. II B. Throughout, we take `P` to be the unsigned integer instantiating the Pauli string P which is being operated upon, with corresponding coefficient $\mathtt{c}_\mathtt{P} \in \mathbb{R}$.

We note that to modify the stored Pauli strings we find it convenient to maintain *two* persistent Pauli sums. The specific usage of the auxiliary Pauli sum on which data structure was chosen to represent it, as well as on the circuit operator applied.

Pauli noise is implemented by a diagonal PTM irrespective of the number of decohering qubits. For example, single-qubit depolarizing noise on qubit i with error probability p scales Pauli strings with any non-identity Pauli on site i by a factor of $\lambda = 4p/3$, leaving the Pauli string itself unchanged. We can efficiently modify the coefficient of P without control flow branching via an expression like

$$\mathtt{c}_\mathtt{P} \to \mathtt{c}_\mathtt{P} \times (1 - \lambda \times (\mathtt{P}_i \neq 0)).$$

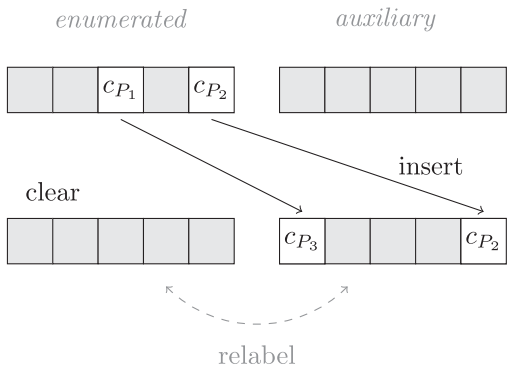
Similarly, dephasing noise updates the coefficient if $(\mathtt{P}_i \neq 0)\,\&\,(\mathtt{P}_i \neq 3)$, and many-qubit Pauli noise channels merely replace the conditional with slightly more complicated (but potentially $\mathcal{O}(1)$ bitwise) logic. The coefficients of the Pauli strings can be updated by directly mutating the enumerated Pauli sum, avoiding use of the auxiliary Pauli sum and associated memory-movement costs. All coefficients are independently modified in this manner, presenting an ideal scenario for parallelization. This can, for example, be sketched as follows (the auxiliary table is not used in this case of Pauli noise but is included in the sketch because it will be used for other gates).



Clifford gates are also non-branching but may modify P to a distinct Pauli string Q while updating only the *sign* of the coefficient. Namely,

$$\mathtt{P} \to \mathtt{Q}, \qquad \mathtt{c}_\mathtt{P} \to \mathtt{c}_\mathtt{Q} = \pm \mathtt{c}_\mathtt{P}.$$

The precise update only depends on the Paulis on the sites the Clifford gate acts on and is determined via lookup tables. Since a Clifford PTM is invertible, a set of unique Pauli strings are mapped to a set of unique output strings; no merging of strings occurs. This permits independent, embarrassingly parallel processing of the strings (if performed in-place, e.g., with arrays). For hash tables, we insert all new strings into the (initially empty) auxiliary hash table which is thereafter taken as the primary hash table.

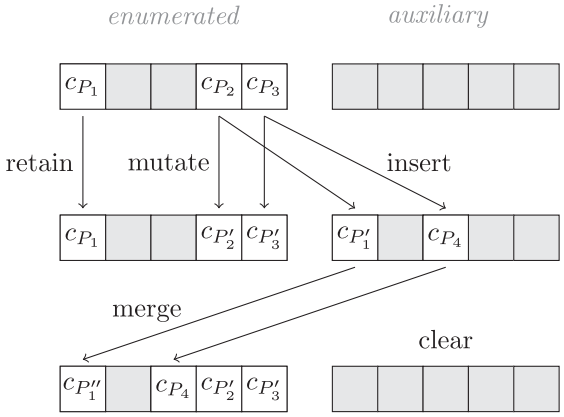


Pauli rotations of angle θ around the Pauli string G, $R_G(\theta) = e^{-i\theta G/2}$, are either one- or two-branching depending on whether P and G commute. Recall Eq. (14)

$$R_G(\theta)[P] = \begin{cases} P & \text{if } [P, G] = 0 \\ \cos(\theta)P + \sin(\theta)Q & \text{else} \end{cases}$$

where $Q = P \cdot G$, modulo the coefficient. Both Q and whether $[P, G] = 0$ can be found in $\mathcal{O}(1)$-many bitwise operations as described in Sec. IV B.

Since $R_G(\theta)[P]$ always includes P, we can modify its coefficient in place and decide where to place Q. For hash tables, we insert Q into the (initially empty) auxiliary hash table. Once enumeration is complete, we union the hash tables. This ensures the strings inserted into the auxiliary hash table are unique, relaxing the need to check for collisions and merging which otherwise introduces control flow branching. Our implementation employs further tricks to reduce memory movement and superfluous reallocating the hash tables.



For array data structures and our `VectorPauliSum` type, we adapt the stream compaction subroutine outlined in Sec. IV C. Inside the following pseudo-code, we again denote `P` and `C` as the main arrays carrying the Pauli strings and their coefficients, `P2` and `C2` their respective auxiliary arrays, `F` the boolean flag array, and `I` the index array. Define `paulirotationprod()` to compute the new Pauli string and its real-valued coefficient [see Eq. (14)]. We then apply a Pauli rotation generated by Pauli string `G` with parameter `theta` as follows:

1. Flag the elements that anticommute.
```
for i = 1...N:
  F[i] = !commutes(P[i], G).
```
2. Perform a parallel cumulative sum over `F` and write the results into `I`, `cumsum(F)` → `I`.
3. Ensure sufficient capacity.
```
if N + I[N] > length(P):
  determine new capacity, e.g., M = 2*(N + I[N])
  resize P, P2, C, C2, F, I to length M
```
4. Write new string and adapt coefficients.
```
for i = 1...N:
  if F[i] == true:
    Q, s = paulirotationprod(P[i], G)
    P[N+I[i]] = Q
    C[N+I[i]] = C[i] * s * sin(theta)
    C[i] = C[i] * cos(theta)
```

Step 3 ensures that all new terms can be written into the main arrays without additional memory movement involving the auxiliary arrays. Furthermore, the joint resize guarantees that all arrays are ready for subsequent processing, like merging or truncating.

Custom operations beyond those natively supported by `PauliPropagation.jl` can be efficiently implemented, provided their (potentially parameterized) action is known at compile-time. The package defines user-overloadable functions (like `apply()`) to specify the action of a custom operation upon a Pauli string. Such functions are automatically invoked during simulation, leveraging `Julia`'s multiple dispatch, yet abstract away the complications of merging Pauli strings and truncation. `PauliPropagation.jl` also permits overloading increasingly sophisticated functions (like `apply-toall!()`) in an "opt-in" fashion, to incorporate performance considerations like memory movement and type-stability. This offers a trade-off between performance and code complexity. Elegantly, these very functions are leveraged by the natively supported gates.

Arbitrary operations known only at runtime as few-qubit computational-basis matrices or Kraus operators can also, in principle, be efficiently implemented. If the gate is fixed, it can directly be turned into a `TransferMapGate`, which computes their PTMs via Eq. (5), converts it into a look-up map, and will then be sequentially applied to all Pauli strings. If instead the gate is parameterized, the unitary and its PT map can be computed once per gate (not once per Pauli string). Alternatively, one could precompute their PTMs by use of a symbolic algebra library. In that case, it is essential to symbolically simplify the PTM using prior knowledge, e.g., the realness of the parameters, since the density of the PTM can be crucial for efficiency. Precomputing the Pauli transfer map in this way is expensive, but may drastically improve the runtime of the propagation.

This technique is leveraged by `QuESTlink` functions such as `CalcPauliTransferMap` [41], and is a planned future facility of `PauliPropagation.jl`.

Further optimizations remain possible. For example, truncation of the propagated `PauliSum` via the strategies presented in Sec. III A presently involves periodically enumerating all contained Pauli strings and removing some from the hash table. This occurs after each gate, and enables simple, modular code with separated concerns. Alas, it is suboptimal; it invokes expensive re-enumeration of the full `PauliSum`, doubling memory-movement costs. It also sees the temporary storage of new Pauli strings which are immediately removed by truncation, incurring superfluous hashing and potentially hash table re-allocation costs. We could instead truncate earlier, moving logic into the `apply()` functions to avoid ever storing truncated strings. Such a scheme complicates the code and custom user truncations, and might only be worthwhile for hand-picked truncations and gate sets.

E. Calculating the overlap

The final step of Pauli propagation is to compute the *overlap* of the propagated Pauli sum with another object—like a state or observable—in order to compute an expectation value. Below, let S be the final n-qubit Pauli sum evolved from an initial observable P_{obs} under the circuit $\mathcal{E}^\dagger$ in the Heisenberg picture, i.e.

$$S = \mathcal{E}^\dagger(P_{\text{obs}}) = \sum_P c_P P. \tag{53}$$

Note that since *unnormalized* n-qubit Pauli strings do not form an ortho*normal* basis (consider $\text{Tr}[I^{\otimes n}] = 2^n$), the norm-squared of a Pauli string is 2^n in lieu of unity

$$\|P\|_2{}^2 = \langle P, P\rangle = \text{Tr}[PP] = 2^n, \tag{54}$$

having adopted the Hilbert-Schmidt inner product. As such, the coefficient of a string P in the Pauli sum S is obtained by scaling the inner-product of P and S.

$$c_P = \frac{1}{2^n}\langle P, S\rangle = \frac{1}{2^n}\text{Tr}[PS]. \tag{55}$$

The form of the output overlap depends on the goal of the simulation.

Correlation functions. A commonly studied quantity in quantum physics is the correlation between a given observable Q and its time evolved version $\mathcal{E}(Q)$. That is, the quantity

$$\frac{1}{2^n}\text{Tr}[Q\mathcal{E}(Q)] = c_Q, \tag{56}$$

where Q is a Pauli string. To compute such correlation functions it suffices to lookup the coefficient of Q, i.e., c_Q, in the data structure representing the Pauli sum $\mathcal{E}(Q)$. Under our use of a hash table, lookup is $\mathcal{O}(1)$ time and no summation is necessary. In `PauliPropagation.jl` code, this is performed as

```
c_Q = getcoeff(S, Q)
```

where `S` is a `PauliSum` object and `Q` is either the integer representation of Q or a `PauliString` object. This is equivalent to the generic approach

```
val = scalarproduct(S, Q)
```

State expectation values. Overlaps with quantum states ρ,

$$\langle P_{\text{obs}}\rangle = \text{Tr}[\mathcal{E}(\rho)P_{\text{obs}}] = \text{Tr}[\rho S] = \sum_P c_P \text{Tr}[\rho P], \tag{57}$$

where S is the propagated observable as defined in Eq. (53), can be evaluated in a variety of ways depending on the form of ρ.

Among the simplest is when ρ is a tensor product of single qubit *stabilizer states*, i.e., when it can be written as $\rho = \bigotimes_{i=1}^n (\frac{I+s_i\sigma_i}{2})$ where $\sigma_i \in (X, Y, Z)$ is a non-identity Pauli on site i and $s_i = \pm 1$ is a sign. For example, $\sigma_i = Z, s_i = +1, \forall i$ yields the all-zero state $|0\rangle\langle 0|$. Then, only if the Pauli on site i is I or σ_i does that term contribute, and c_P can be added to the running sum. In pseudo-code this may read as follows:

```
val = 0.0
for (c_P, P) in pairs(S):
  sign = 1
  for i in 1...n:
    if P_i == σ_i:
      sign *= s_i
    elseif P_i != I:
      sign = 0
  val += sign * c_P
```

In the end, `val` is the computed expectation value. Notice that only Pauli strings contribute with their coefficients when they contain I or σ_i on all sites i. If any Pauli is different, the Pauli string is *orthogonal* to the state ρ and does not contribute.

In the common case of the all-zero state only Pauli strings consisting of I and Z Paulis contribute which can be checked with $\mathcal{O}(1)$ bit operations. In `PauliPropagation.jl` this is implemented via the inbuilt function

```
val = overlapwithzero(S).
```

In many cases, this will be the overlap function of choice when simulating quantum circuits. Similarly, we

provide the `overlapwithplus()` for $\rho = |+\rangle\langle+|$ and `overlapwithcomputational()` for an arbitrary computational basis state $\rho = |x\rangle\langle x|$.

If ρ can be fully written down in the Pauli basis due to a low number of qubits or if it is a high-temperature state, then the overlap can be computed straightforwardly with one important caveat. Assume now that `rho` is a `PauliSum` object capturing the Pauli basis decomposition of ρ, then the function call

```
val = overlapwithpaulisum(rho, Q)
```

returns the correct expectation value. Note that this function calls the `scalarproduct()` function and multiplies the result by *an otherwise missing factor of* 2^n. This is because it is almost always more convenient for our code to function as if we worked with *normalized* Pauli operators for which $\mathrm{Tr}[P^2] = 1$ holds—something that only comes into play when explicitly representing quantum states.

More generally, for quantum states ρ that are neither stabilizer states nor efficiently representable in Pauli basis, the overlap can still be computed if the expectation values $\mathrm{Tr}[\rho P] = \langle P\rangle_\rho$ can be evaluated. This may be possible if the state is represented in another classical form, for example, as a tensor network, or if the state lives on a *quantum device*. In such cases, direct measurements or randomized protocols like classical shadows, can be used to estimate the Pauli expectation values $\langle P\rangle_\rho$, yielding

$$\mathrm{Tr}[\rho S] = \sum_P c_P \mathrm{Tr}[\rho P] = \sum_P c_P \langle P\rangle_\rho. \tag{58}$$

Then a `PauliSum` object `rhoP` storing the dictionary with pairs `P => ⟨P⟩_ρ` can handily be overlapped with the evolved Pauli sum via

```
val = scalarproduct(rhoP, Q).
```

In this manner, Pauli propagation proves to be a framework with surprising flexibility in how it can be used and combined with other approaches.

Theory Box 6: Magic and hardness of classical simulation in the Pauli basis

Pauli propagation methods are efficient when the Pauli spectrum of O is concentrated on few Pauli terms—a condition that is sufficient but not always necessary. A common way to formalize this notion is through the concept of *magic* [75]. As a preliminary step, we observe that the squared coefficients of O sum up to one, and therefore they can be interpreted as probability distribution, also known as *characteristic distribution*

$$\sum_\alpha c_\alpha^2 = 2^{-n}\|O\|_2^2 = 2^{-n}\|P\|_2^2 = 1. \tag{59}$$

A common metric of the "spreadness" of a distribution $p(x)$ is its *collision probability* $\sum_x p^2(x)$. In particular, the collision probability of the characteristic distribution of O is known as the *generalized Pauli purity*

$$P^{(2)}(O) \coloneqq \sum_\alpha c_\alpha^4. \tag{60}$$

A related quantity of interest is the *operator stabilizer entropy* (OSE) [75], defined as $\mathcal{M}^{(2)}(O) \coloneqq -\log\{P^{(2)}(O)\}$. We say that O exhibits "high magic" if $P^{(2)}(O)$ is exponentially small in system size. In such regimes, approximating O by truncating its small coefficients can be prohibitive, as the corresponding ℓ_1-norm is exponentially large

$$\sum_\alpha |c_\alpha| \geqslant \frac{(\sum_\alpha c_\alpha^2)^{\frac{3}{2}}}{(\sum_\alpha c_\alpha^4)^{\frac{1}{2}}} = \frac{1}{\sqrt{P^{(2)}(O)}} \geqslant \exp(n), \tag{61}$$

where the first step follows from Lemma 8 in Ref. [77]. As most quantities to be estimated using Pauli propagation scale non-exponentially, we see that the ℓ_1-norm cannot provide a non-trivial error estimates when the magic is high.

V. OUTLOOK

A. What is the niche of Pauli propagation?

1. Magic

Pauli propagation is most directly limited by the growth of the number of Pauli terms in the propagation tree. This is quantified by *magic* [75,76] as discussed in Theory Box 6. If the magic generated is low, then the number of Pauli strings with non-negligible coefficients will be small, and Pauli propagation equipped with exclusively coefficient truncation will be extraordinarily fast and precise at simulating not only expectation values but the full propagated object. On the other hand, if magic is high, then coefficients are more spread over a large number of Pauli strings, and for in the worst case (i.e., some initial states), one would have to keep propagating a large fraction of all Pauli strings for precise estimates. See Fig. 8 for an example of how the value of Pauli rotation angles can drastically change the difficulty of a simulation as measured by the number of Pauli strings above certain absolute coefficient thresholds. It remains an open question whether there exist physically interesting low magic but high entanglement circuits whereby Pauli propagation methods could prove very advantageous over popular tensor network methods, or to what extent Pauli propagation approaches can be successful in simulating high magic circuits.

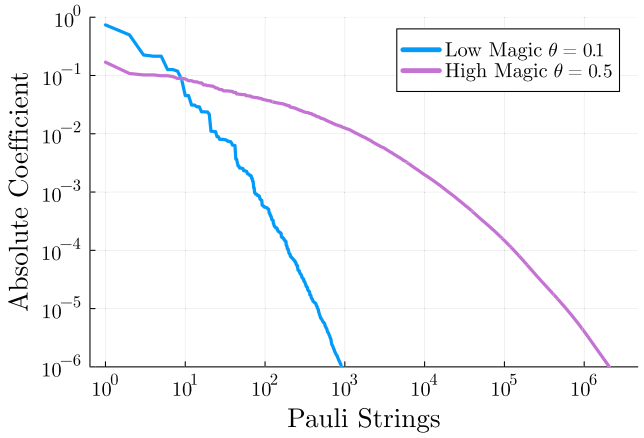


FIG. 8. Manifestation of magic in Pauli propagation simulation. We propagate a single-site Pauli in the middle of a 6×6 grid through a quantum circuit that corresponds to eight layers of the Trotterization of the tilted-field Ising Hamiltonian, i.e., eight repetitions of RX, RZ, and RZZ Pauli rotations. Importantly, all Pauli rotation angles are set to a single value θ, with $\theta = 0.1$ in one case and $\theta = 0.5$ in another. The latter induces higher magic, or in other words, creates Pauli strings with coefficients whose magnitude is more spread out. This typically results in harder simulation, because more Pauli strings need to be kept for the same accuracy. We note that the total number of Pauli strings would be the same in both cases if none were truncated, but for practical purposes, we employed a coefficient truncation of 10^{-7}.

2. Noise and open quantum systems

Several key factors contribute to non-unitary dynamics being potentially more promising applications for Pauli propagation. One quite naive argument is that, by working in the Pauli basis, we are already paying the cost of open quantum system simulation. This is exemplified by the significantly increased cost of exact density matrix simulation compared to pure state wavefunction simulation. More concretely, however, noise can *reduce* magic. Pauli noise strictly reduces magic, and even non-unital noise like amplitude damping tends to effectively reduce magic in unstructured circuits. Working with open quantum systems may also enable working directly with quantum states as mixed states can be sparse in the Pauli basis. An extreme example would be the maximally mixed state, which can be represented with a single Pauli string (the identity), suggesting that high-temperature physics might be amenable to efficient Pauli propagation simulation.

3. Expectation values in typical random or scrambling circuits

While magic can be a useful quantity for bounding worst-case errors and runtime, the difficulty of many simulations does not strictly correlate with magic. Namely, for many initial states, their overlap with most Pauli strings can be negligible. If it can be deduced which Pauli strings are most likely to contribute to expectation values during propagation, expectation values can be significantly more efficient than magic alone would indicate. This intuitive claim is formalized in Ref. [14] which shows that for typical quantum circuits, expectation values can be estimated to a reasonable approximation with polynomial resources. In other words, high magic implies worst-case hardness and is a necessary but not sufficient condition for average-case hardness. For one, this suggests that a wide range of parameterized quantum circuits used for variational algorithms can be simulated when randomly initialized [19,32]. It also implies that the scrambling behavior may lend certain chaotic quantum systems to be simulated using Pauli propagation methods [78–80].

That being said, in our experience, simulating nontrivial high-magic circuits beyond a certain precision remains hard—particularly in quantum dynamics. In current Pauli propagation approaches, there is no form of spatial factorization like in tensor networks that may enable working in exponential spaces with polynomial resources. Currently, the only way that exponential Pauli strings can contribute to a simulation is by explicitly propagating exponentially many Pauli strings. It thus remains to be seen to what extent the guarantees enjoyed by Pauli propagation for typical random circuits translate into practical benefits for physically motivated problems.

4. Local observables and operator lightcones

One feature that can make the Heisenberg simulation of quantum circuits substantially simpler than the Schrödinger evolution of states is the so-called *operator lightcone* or *entanglement lightcone*. It describes which operations in a circuit *can* contribute to certain observable expectation values at all (called the *hard* lightcone), or which operations *predominantly* contribute (called the *effective* lightcone). This effect is very visible for geometrically local observables and topologies, and low Pauli weight observables in general, where the number of active sites with non-identity components only grows slowly. Gates outside the hard lightcone could simply be removed from the circuit. This is often done in the Schrödinger picture but Pauli propagation natively skips over those gates when they commute with all current Pauli strings. However, some gates contribute only weakly based on their position in the circuit or because of their parameter. Suitable coefficient truncations will capture the most significant effect of such gates, which may thus not come at the full expected computational cost. An example demonstrating the drastic effects of Pauli weight of the initial observable can be viewed in Fig. 9.

5. Circuit topology

Pauli propagation in its current form is flexibly adaptable to any circuit topology, i.e., to which qubits are connected via multi-qubit gates. It is in principle not

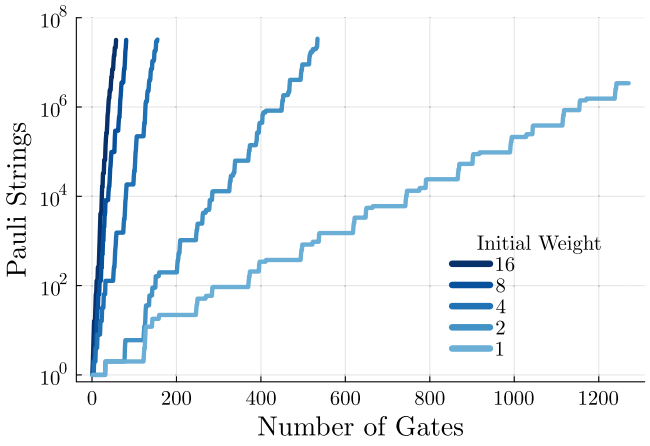


FIG. 9. Effect of operator lightcones. We demonstrate how the Pauli weight of the observable (sometimes called *bodyness*) affects the growth in number of Pauli strings in a simulation. The setup is a 32-qubit system with RX-RZ-RX rotations per qubit and RYY entanglers on a 1D periodic bricklayer ring. The non-identity Paulis are chosen at random and spaced out equally on the Pauli string, e.g., weight 16 is ZIYIYIZI…. If the non-identity Paulis were neighboring, the number of Pauli strings would still grow faster than with fewer Paulis, but likely not as fast as spaced-out Paulis.

strictly harder to simulate all-to-all gates, such as Pauli rotations with global generators. In practice, however, the topology strongly affects how quickly Pauli strings spread into the exponentially-sized space, with the operator lightcone, and consequently magic, increasing more quickly in highly connected systems. In contrast to tensor networks, current Pauli propagation approaches are not biased towards or against certain topologies, implying that in cases where tensor networks work well, they likely outperform Pauli propagation, but in cases where they do not, Pauli propagation may have a practical advantage. Given the impressive performance of MPS for simulating a wide range of 1D and quasi-1D systems, we do not expect Pauli propagation to yield a better time-accuracy trade-off there. However, in 2D and 3D systems, for example, as explored in Ref. [16], or in more exotic long-range connected systems, this may be possible. For a more detailed theoretical discussion of the role of topology for Pauli propagation see Theory Box 7.

Theory Box 7: Entanglement lightcone and geometric dimension

Consider a *local* Pauli operator P and a quantum circuit $U = U_L U_{L-1} \cdots U_1$ consisting of L layers. Each layer U_j consists of non-overlapping two-qubit gates arranged on a D-dimensional topology. A straightforward lightcone argument shows that the Heisenberg-evolved operator

$$U^\dagger P U$$

has support only on those qubits within a lightcone whose size scales as $\mathcal{O}(L^D)$. As a consequence, one could perform statevector simulation on these $\mathcal{O}(L^D)$ qubits and compute $\mathrm{Tr}[PU\rho U^\dagger]$ exactly with computational resources scaling as

$$\exp\left(\mathcal{O}(L^D)\right).$$

Pauli propagation and certain other methods operating in the Heisenberg picture will automatically adapt to such lightcones because gates outside the lightcone act trivially on identity Paulis. Truncations based on, for example, coefficient magnitude or Pauli weight additionally give rise to the concept of *effective* lightcones focusing on the most significant contributions to the expectation values. These dominant contributions often spread more slowly than the so-called *hard* lightcone containing all qubits where any non-identity Pauli can exist.

6. Operator dynamics

One concrete potential application for Pauli propagation methods is for studying operator spreading and operator hydrodynamics [81,82]. To learn about how operators spread in quantum systems, tensor networks have been a popular choice of computational framework [63]. While MPS has been applied to analyze coarse-grained behaviors such as operator spreading in one-dimensional systems, Pauli propagation captures the fine-grained evolution of individual Pauli strings at a microscopic level. In this context, Pauli propagation offers a direct algorithm for computing the dynamics of quantum circuits, particularly in near-Clifford, shallow-depth, or noisy settings. Its precision and flexibility in simulating individual Pauli string evolution make it particularly valuable in verifying and benchmarking current quantum simulations and experimental measurements of operator dynamics across various platforms, such as superconducting qubits [83,84], trapped ion systems [85], and Rydberg atom arrays [86].

Where tensor networks may, in some cases, allow to simulate for longer times or to higher precision, they need to be queried to retrieve information about individual Pauli strings. This is technically *efficient*, but is practically slower than a single look-up in the Pauli sum. More importantly, however, propagated Pauli sums transparently display which Pauli strings are present and with which coefficients. This could be imitated by sampling Pauli strings from a tensor network, but may be less practical.

7. Repeated evaluations

As discussed in Sec. II C, Pauli propagation surrogates aim to trade an initial computational overhead for a

potentially drastic speed-up for new parameters. There are a lot of nuances that make such an algorithm challenging to realize in practice, but one could imagine practitioners compiling a surrogate once, for example as part of a particular experimental setup, then the extremely fast re-evaluation is an integral part of the data processing pipeline. At this current stage, the surrogate implementation in `PauliPropagation.jl` is far from ideal (though significantly better than demonstrated in Ref. [12] and able to simulate quantum convolutional neural networks at scale until 1024 qubits [32]). But this is a possibility that appears truly unique for propagation methods.

8. Hybridization with quantum devices

Finally, and going hand-in-hand with the transparency of which Pauli strings are contained in an evolved observable, Pauli propagation can straightforwardly be combined with quantum devices within the simulation of any quantum circuit. Following the framework of classical-enhanced quantum computation from Ref. [19], a quantum device merely needs to perform Pauli measurements of the initial state or a partially evolved state to estimate the expectation values in Eq. (11). The hybrid framework is provably efficient in the same cases where Pauli propagation itself is provably efficient [17], but given the empirical success of Pauli propagation, it stands to reason that this framework can be pushed beyond classical and beyond current quantum limits with clever algorithm engineering. A proof-of-principle realization was recently presented in Ref. [20]. We believe such hybrid approaches are a promising avenue for eking advantage out of moderate-scale non-fault-tolerant quantum devices [87,88].

B. Avenues for further developments

Pauli propagation is a new classical simulation tool for quantum physics and quantum computing practitioners. Despite only recently entering the stage, Pauli propagation shows promise for becoming the computational framework of choice for some tasks. A certain dose of optimism is crucial to make use of up-and-coming technologies, but this is not to downplay the many ways in which Pauli propagation can be improved. Mirroring the structure of this manuscript, these range from high to low level.

Starting at the low level, the main challenges—perhaps unsurprisingly—are to increase the speed of computation and lift the absolute constraint imposed by finite memory. We already support multithreaded CPU simulation and limited GPU acceleration via the `VectorPauliSum` type. They heavily rely on sorting and writing back and forth between pre-allocated memory. While any performance increases are generally desirable, with our code available through `PauliPropagation.jl`, one likely runs out of memory before they run out of time. This is exacerbated by GPUs commonly having less memory than HPC CPU nodes. In other words, with current propagation approaches GPUs will be faster at running out of memory, unless applied to tasks that are more time- than memory-limited (e.g., learning and optimization tasks). Larger-scale quantum simulations may require the use of propagation methods beyond pure merging-BFS (see Sec. III B), for example, deterministic or randomized depth-first components. Both GPU-accelerated and non-memory-limited propagation strategies will in the future be supported via `PauliPropagation.jl`.

On the middle level, there are many questions surrounding how to best utilize prior knowledge about the circuits to be simulated. Pauli weight truncation, for example, can be seen as a truncation tailored to random quantum circuits [14] which is significantly less effective in structured circuits. A Majorana propagation method has recently been developed for the simulation of Fermionic systems [25–28]. This method is substantially more suited to simulating the long Pauli strings that arise naturally in Fermionic simulations. Symmetric merging strategies, whereby Pauli operators that behave identically due to symmetry properties of the circuit are merged to a single propagated string, can further reduce the memory cost of simulation [55,57].

Beyond manifesting physical understanding, there are many algorithmic and conceptual tricks to be played for performing Pauli propagation more efficiently. One such trick is what we call *meet-in-the-middle* simulation of expectation values, where the initial and final objects are propagated in the Schrödinger and Heisenberg pictures, respectively. This replaces one depth L simulation by two depth $L/2$ simulations, in turn halving the exponent of the computation, but with the additional overhead of calculating the inner product of the two half-evolved objects. In the case of spin correlation functions (see the enumeration in Sec. II A), both parts can be simulated via Pauli propagation with very efficient inner product computation, but in principle, any two computational methods (classical or quantum) can be combined [11,17,20]. Other conceptual innovations could, for example, exploit the unitarity of certain gates and iteratively compute or correct Pauli sum coefficients via backward and forward evolution, or find suitable extrapolations for expectation values via variable noise rates or truncation parameters. More exploratory would be to utilize Pauli propagation for sampling quantum states, which could be done via the sampling-to-expectation-value reduction described in Ref. [29], but it has, to our knowledge, never been realized at scale. Crucially, recent works appear to indicate that propagation methods can be very proficient at learning ground state preparation circuits or estimating the ground state energies of chemistry or condensed matter Hamiltonians [14,25,89–91]. Many more exciting innovations are possible and merely require practitioners to try them out.

Finally, on a high level, we need to better understand how Pauli propagation and similar methods can be useful

to practitioners. What are the applications where such methods stand above the rest? Perhaps it appears dissatisfying that our work cannot provide a definitive answer to this question, but we hope to have pointed out our current understanding of strengths and weaknesses. One point we would like to stress here is that current Pauli propagation codes are implemented on a very low computational level, using basic bit operations and simple data structures. This means that in its current form, Pauli propagation is exceptionally efficient in cases that it is well-equipped to handle, while suffering in other cases where more sophisticated algorithms (e.g. tensor networks) likely have the upper hand. That being said, it is possible that future variations of propagation methods exceed what we perceive as their current limitations, finding even broader application, and we hope that our package `PauliPropagation.jl` can play a role in these developments.

ACKNOWLEDGMENTS

The authors would like to acknowledge Tomislav Begušić, Joseph Tindall, Matteo D'Anna, Julia Gerecke, Julia Guignon, and Su Yeon Chang for helpful comments on this manuscript. The authors would like to acknowledge Patrick Rall, Nikita Astrakhantsev, Joseph Tindall, Lukas Devos, Marc Drudis, and Aaron Miller for insightful discussions. M. S. R. acknowledges funding from the 2024 Google PhD Fellowship and the Swiss National Science Foundation [Grants No200021-219329]. T. J. acknowledges support of the NCCR MARVEL, a National Centre of Competence in Research, funded by the Swiss National Science Foundation (Grants No. 205602). Y. T. acknowledges support from NCCR spin, a National Centre of Competence in Research, funded by the Swiss National Science Foundation (Grants No.565785). A. A. and Z. H. acknowledge support from the Sandoz Family Foundation-Monique de Meuron program for Academic Promotion.

DATA AVAILABILITY

All simulations in this manuscript were performed using the open-source library `PauliPropagation.jl` [33]. All data can be replicated on version `v0.7.3` using the descriptions provided in the text, and is available upon request.

APPENDIX: CONTROVERSIES

This work sits at the intersections of quantum information theory, quantum computing, high-performance classical computing, and software development.

In lieu of an appendix, this *controversies* section documents the process behind many of the work's decisions which compromise the conventions of the various fields.

1. Pauli normalization

The `PauliPropagation.jl` library effectively propagates observables in the *unnormalized* Pauli basis. That is, the coefficients of the terms in a Pauli sum $\mathcal{E}[P] = \sum_j c_j P_j$ are stored simply as c_j. We could have instead explicitly worked in a normalized Pauli basis $\mathsf{P}_i = 2^{-\frac{n}{2}} P_i$, as appeared in the definition of the PTM (Theory Box 1) or numerous theoretical works using the Pauli path formalism [14,29]. Had we done so, we might then have worked with Pauli sums of the form

$$\mathcal{E}[P] = \sum_j 2^{\frac{n}{2}} c_j \mathsf{P}_i, \tag{A1}$$

and so stored $2^{\frac{n}{2}} c_j$ instead of c_j. Choosing to work without normalization is akin to non-dimensionalization in continuous-space numerical modeling [92].

As a consequence, on viewing our Pauli strings as *vectors*, our propagated basis is *not* ortho*normal*, as mentioned in Sec. IV E. This sometimes necessitates diligence when evaluating quantities naturally expressed as Hilbert-Schmidt inner products. We saw, for example, that the `scalarproduct()` function computes the inner product between the *coefficients* of two Pauli sums (Or, at the risk of further confusing the reader, it can equivalently be viewed as computing the Hilbert Schmidt inner product between two Pauli sums of normalized Pauli strings.). However, to compute the Hilbert Schmidt inner product of a state and a Pauli sum, i.e., $\sum_j c_j \mathrm{Tr}[\rho P_j]$, we need to multiply `scalarproduct()` by 2^n to account for the fact that $\mathrm{Tr}[I] = 2^n$. This subtlety is hidden from a typical user who can work with the `overlapwithpaulisum(rho, Q)` function that already includes this 2^n factor.

2. Indexing and endianness

As noted in Convention Box 1, this manuscript has juggled multiple endian conventions. We notated binary numbers with little-endian (least significant bit on the right, indexing from right to left), prevalent among programmers. This matches the ubiquitous positional notation used to denote numerals in *any* base, including decimal.

$$\begin{aligned} 1101_2 &= 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0 \\ &= 13_{10} = 1\times 10^1 + 3\times 10^0 \end{aligned} \tag{A2}$$

We chose however to *index* bits and digits from *one* in lieu of the alternate, natural choice of *zero*. As such, the bit at index 1 of 1101_2 is 1. This matches the indexing convention of the `Julia` language within which `PauliPropagation.jl` is implemented, but clutters our bitwise logic (presented in Sec. IV B) with otherwise superfluous -1 index offsets.

In contrast, we chose to employ *big*-endian (indexing from left to right) more familiar to the QIT community

when notating Pauli strings, such that

$$ZX \equiv Z \otimes X \equiv Z_1 X_2. \tag{A3}$$

But consider that a Pauli string is a base-four numeral with digit symbols $\{I, X, Y, Z\} = \{0, 1, 2, 3\}$; an equivalence exploited by our instantiation of a string as an unsigned integer in Sec. IV A. We have thus adopted a distinct endianness for base-four numerals versus base-two and base-ten.

Further, consider that sites of a Pauli string correspond one-to-one with qubits in a circuit. This necessitates that our kets are also denoted in big-endian, i.e.

$$Z_1 X_2 |a, b\rangle \equiv (Z \otimes X)|a, b\rangle \equiv Z|a\rangle \otimes X|b\rangle \tag{A4}$$

where $|a\rangle$ has index one and $|b\rangle$ has index two. *Qubit* states are consequently also big-endian ($|01\rangle = |0\rangle_1 |1\rangle_2$), but this conflicts with the *bit* endianness in Eq. (A2). This tension is the unfortunate consequence of our attempts to use the most natural and common conventions of quantum information theorists and computer scientists, namely the conventions in Eq. (A3) and Eq. (A4), respectively.

3. Programming language

Pauli propagation simulation is an interesting case study. It is logic-heavy and yet when running at scale, it becomes memory-bandwidth-bound. Much of its logic is irreducible to numerical routines which can be dispatched to existing accelerated libraries; instead, the hot loops perform custom, nuanced processing. Since its algebra is highly scientific, so too must be its reader; it must be maintainable by the scientific community, and so be written in a language which is high-level and easy to use. Most importantly, to compete with quantum hardware providers the code needs to run *very fast*.

These attributes necessitate its implementation in a programming language which is both simple yet fast. Ideally, it should not delay the scientific programmer with long compilation times—yet it must make excellent use of compiler optimizations so that its high-level syntax does not compromise performance. It should further empower the use to extend the existing `PauliPropagation.jl` code base with project-specific adaptations that can be engineered to be equally performant. These considerations made us arrive at *Julia*. That being said, we are aware that `Python` is the language of choice for a large fraction of our readers. In fact, we encourage `Python` wrappers for `PauliPropagation.jl`, and invite the community to contribute!

[1] R. Orús, *A practical introduction to tensor networks: Matrix product states and projected entangled pair states*, Ann. Phys. (Amsterdam) **349**, 117 (2014).

[2] F. Verstraete, V. Murg, and J. I. Cirac, *Matrix product states, projected entangled pair states, and variational renormalization group methods for quantum spin systems*, Adv. Phys. **57**, 143 (2008).

[3] Y.-Y. Shi, L.-M. Duan, and G. Vidal, *Classical simulation of quantum many-body systems with a tree tensor network*, Phys. Rev. A **74**, 022320 (2006).

[4] F. Verstraete and J. I. Cirac, *Renormalization algorithms for quantum-many body systems in two and higher dimensions*, arXiv:cond-mat/0407066.

[5] J. Biamonte and V. Bergholm, *Tensor networks in a nutshell*, arXiv:1708.00006.

[6] J. Tindall, M. Fishman, E. M. Stoudenmire, and D. Sels, *Efficient tensor network simulation of IBM's Eagle kicked Ising experiment*, PRX Quantum **5**, 010308 (2024).

[7] J. Tindall, A. Mello, M. Fishman, M. Stoudenmire, and D. Sels, *Dynamics of disordered quantum systems with two- and three-dimensional tensor networks*.

[8] S. Patra, S. S. Jahromi, S. Singh, and R. Orus, *Efficient tensor network simulation of IBM's largest quantum processors*, arXiv:2309.15642.

[9] G. Carleo and M. Troyer, *Solving the quantum many-body problem with artificial neural networks*, Science **355**, 602 (2017).

[10] D. Wu, R. Rossi, F. Vicentini, N. Astrakhantsev, F. Becca, X. Cao, J. Carrasquilla, F. Ferrari, A. Georges, M. Hibat-Allah *et al.*, *Variational benchmarks for quantum many-body problems*.

[11] T. Begušić, J. Gray, and G. K.-L. Chan, *Fast and converged classical simulations of evidence for the utility of quantum computing before fault tolerance*, Sci. Adv. **10** (2024).

[12] M. S. Rudolph, E. Fontana, Z. Holmes, and L. Cincio, *Classical surrogate simulation of quantum systems with LOWESA*, arXiv:2308.09109.

[13] Y. Shao, F. Wei, S. Cheng, and Z. Liu, *Simulating noisy variational quantum algorithms: A polynomial approach*, Phys. Rev. Lett. **133**, 120603 (2024).

[14] A. Angrisani, A. Schmidhuber, M. S. Rudolph, M. Cerezo, Z. Holmes, and H.-Y. Huang, *Classically estimating observables of noiseless quantum circuits*, Phys. Rev. Lett. **135**, 170602 (2025).

[15] E. Fontana, M. S. Rudolph, R. Duncan, I. Rungger, and C. Cîrstoiu, *Classical simulations of noisy variational quantum circuits*, npj Quantum Inf. **11**, 1 (2025).

[16] T. Begušić and G. K.-L. Chan, *Real-time operator evolution in two and three dimensions via sparse Pauli dynamics*, PRX Quantum **6**, 020302 (2025).

[17] S. Lerch, R. Puig, M. Rudolph, A. Angrisani, T. Jones, M. Cerezo, S. Thanasilp, and Z. Holmes, *Efficient quantum-enhanced classical simulation for patches of quantum landscapes*, arXiv:2411.19896.

[18] P. Bermejo, B. Aizpurua, and R. Orús, *Improving gradient methods via coordinate transformations: Applications to quantum machine learning*, Phys. Rev. Res. **6**, 023069 (2024).

[19] M. Cerezo, M. Larocca, D. García-Martín, N. L. Diaz, P. Braccia, E. Fontana, M. S. Rudolph, P. Bermejo, A. Ijaz, S. Thanasilp *et al.*, *Does provable absence of barren plateaus imply classical simulability?*, Nat. Commun. **16**, 7907 (2025).

[20] B. Fuller, M. C. Tran, D. Lykov, C. Johnson, M. Rossmannek, K. X. Wei, A. He, Y. Kim, D. Vu, K. Sharma *et al.*, *Improved quantum computation using operator backpropagation*.

[21] P. Rall, D. Liang, J. Cook, and W. Kretschmer, *Simulation of qubit quantum circuits via Pauli propagation*, Phys. Rev. A **99**, 062337 (2019).

[22] L. Broers and L. Mathey, *Exclusive-or encoded algebraic structure for efficient quantum dynamics*.

[23] L. Broers, R.-Y. Sun, and S. Yunoki, *Scalable simulation of quantum many-body dynamics with or-represented quantum algebra*.

[24] T. Begušić, K. Hejazi, and G. K.-L. Chan, *Simulating quantum circuit expectation values by Clifford perturbation theory*, J. Chem. Phys. **162**, 154110 (2025).

[25] A. Miller, J. Favre, Z. Holmes, Ö. Salehi, R. Chakraborty, A. Nykänen, Z. Zimboras, A. Glos, and G. García-Pérez, *Simulation of fermionic circuits using Majorana propagation*, arXiv:2503.18939.

[26] M. D'Anna, J. Nys, and J. Carrasquilla, *Majorana string simulation of nonequilibrium dynamics in two-dimensional lattice fermion systems*, arXiv:2511.02809.

[27] F. Alam, J. L. Bosse, I. Čepaitė, A. Chapman, L. Clinton, M. Crichigno, E. Crosson, T. Cubitt, C. Derby, O. Dowinton *et al.*, *Fermionic dynamics on a trapped-ion quantum computer beyond exact classical simulation*, arXiv:2510.26300.

[28] F. Alam, J. L. Bosse, I. Čepaitė, A. Chapman, L. Clinton, M. Crichigno, E. Crosson, T. Cubitt, C. Derby, O. Dowinton *et al.*, *Programmable digital quantum simulation of 2D Fermi-Hubbard dynamics using 72 superconducting qubits*, arXiv:2510.26845.

[29] D. Aharonov, X. Gao, Z. Landau, Y. Liu, and U. Vazirani, *A polynomial-time classical algorithm for noisy random circuit sampling*, in *Proceedings of the 55th Annual ACM Symposium on Theory of Computing* (2023), Vol. 945.

[30] T. Schuster, C. Yin, X. Gao, and N. Y. Yao, *A polynomial-time classical algorithm for noisy quantum circuits*, Phys. Rev. X **15**, 041018 (2025).

[31] A. Angrisani, A. A. Mele, M. S. Rudolph, M. Cerezo, and Z. Holmes, *Simulating quantum circuits with arbitrary local noise using Pauli propagation*, PRX Quantum **7**, 020313 (2026).

[32] P. Bermejo, P. Braccia, M. S. Rudolph, Z. Holmes, L. Cincio, and M. Cerezo, *Quantum convolutional neural networks are effectively classically simulable*, PRX Quantum **7**, 020304 (2026).

[33] *PauliPropagation.jl*, GitHub Repository (2026), https://github.com/SparqleSim/PauliPropagation.jl.

[34] M. S. Rudolph, A. Angrisani, A. Wright, I. Sanderski, R. Puig, and Z. Holmes, *Thermal state simulation with Pauli and Majorana propagation*, arXiv:2602.04878.

[35] J. M. Chow, J. M. Gambetta, A. D. Córcoles, S. T. Merkel, J. A. Smolin, C. Rigetti, S. Poletto, G. A. Keefe, M. B. Rothwell, J. R. Rozen, M. B. Ketchen, and M. Steffen, *Universal quantum gate set approaching fault-tolerant thresholds with superconducting qubits*, Phys. Rev. Lett. **109**, 060501 (2012).

[36] J. Emerson, M. Silva, O. Moussa, C. Ryan, M. Laforest, J. Baugh, D. G. Cory, and R. Laflamme, *Symmetrized characterization of noisy quantum processes*, Science **317**, 1893 (2007).

[37] C. King and M. B. Ruskai, *Minimal entropy of states emerging from noisy quantum channels*, IEEE Trans. Inf. Theor. **47**, 192 (2001).

[38] M. Ben-Or, D. Gottesman, and A. Hassidim, *Quantum refrigerator*, arXiv:1301.1995.

[39] A. A. Mele, A. Angrisani, S. Ghosh, S. Khatri, J. Eisert, D. Stilck França, and Y. Quek, *Noise-induced shallow circuits and the absence of barren plateaus*, Nat. Phys. 1 (2026).

[40] A. Deshpande, M. Hinsche, S. Najafi, K. Sharma, R. Sweke, and C. Zoufal, *Dynamic parameterized quantum circuits: Expressive and barren-plateau free*, arXiv:2411.05760.

[41] T. Jones and S. Benjamin, *Questlink—mathematica embiggened by a hardware-optimised quantum emulator*, Quantum Sci. Technol. **5**, 034012 (2020).

[42] B. Barak, C.-N. Chou, and X. Gao, *Spoofing linear cross-entropy benchmarking in shallow quantum circuits*, arXiv:2005.02421.

[43] A. Tanggara, M. Gu, and K. Bharti, *Classically spoofing system linear cross entropy score benchmarking*, arXiv:2405.00789.

[44] Y. Shao, F. Wei, S. Cheng, and Z. Liu, *Simulating noisy variational quantum algorithms: A polynomial approach*, Phys. Rev. Lett. **133**, 120603 (2024).

[45] G. González-García, J. I. Cirac, and R. Trivedi, *Pauli path simulations of noisy quantum circuits beyond average case*, Quantum **9**, 1730 (2025).

[46] N. A. Nemkov, E. O. Kiktenko, and A. K. Fedorov, *Fourier expansion in variational quantum algorithms*, Phys. Rev. A **108**, 032406 (2023).

[47] C. Cirstoiu, *A Fourier analysis framework for approximate classical simulations of quantum circuits*, arXiv:2410.13856.

[48] S. Aaronson and D. Gottesman, *Improved simulation of stabilizer circuits*, Phys. Rev. A **70**, 052328 (2004).

[49] S. Bravyi and D. Gosset, *Improved classical simulation of quantum circuits dominated by Clifford gates*, Phys. Rev. Lett. **116**, 250501 (2016).

[50] D. Gottesman, *Stabilizer codes and quantum error correction*, Ph.D. thesis, California Institute of Technology, 1997, arXiv:quant-ph/9705052.

[51] C. Gidney, *Stim: A fast stabilizer circuit simulator*, Quantum **5**, 497 (2021).

[52] H. Pashayan, O. Reardon-Smith, K. Korzekwa, and S. D. Bartlett, *Fast estimation of outcome probabilities for quantum circuits*, PRX Quantum **3**, 020361 (2022).

[53] S. Bravyi, D. Browne, P. Calpin, E. Campbell, D. Gosset, and M. Howard, *Simulation of quantum circuits by low-rank stabilizer decompositions*, Quantum **3**, 181 (2019).

[54] M. D'Anna, Y. Zhang, R. Wiersema, M. S. Rudolph, and J. Carrasquilla, *Circuit compression for 2D quantum dynamics*, arXiv:2507.01883.

[55] Y. Teng, S. Y. Chang, M. S. Rudolph, and Z. Holmes, *Leveraging symmetry merging in Pauli propagation*, arXiv:2512.12094.

[56] *MajoranaPropagation.jl*, GitHub Repository (2025), https://github.com/SparqleSim/MajoranaPropagation.jl.

[57] N. Loizeau, J. C. Peacock, and D. Sels, *Quantum many-body simulations with paulistrings. jl*, SciPost Phys. Codebases **54**, r1.5 (2025).

[58] F. Uskov and O. Lychkovskiy, *Quantum dynamics in one and two dimensions via the recursion method*, Phys. Rev. B **109**, L140301 (2024).
[59] I. Ermakov, O. Lychkovskiy, and T. Byrnes, *Unified framework for efficiently computable quantum circuits*, arXiv:2401.08187.
[60] A. Teretenkov, F. Uskov, and O. Lychkovskiy, *Pseudomode expansion of many-body correlation functions*, Phys. Rev. B **111**, 174308 (2025).
[61] I. Ermakov, *Operator growth in many-body systems of higher spins*, arXiv:2504.07833.
[62] I. Shirokov, V. Khrushchev, F. Uskov, I. Dudinets, I. Ermakov, and O. Lychkovskiy, *Quench dynamics via recursion method and dynamical quantum phase transitions*, arXiv:2503.24362.
[63] T. Rakovszky, C. Von Keyserlingk, and F. Pollmann, *Dissipation-assisted operator evolution method for capturing hydrodynamic transport*, Phys. Rev. B **105**, 075131 (2022).10.1103/PhysRevB.105.075131
[64] N. Dowling, K. Modi, and G. A. White, *Bridging entanglement and magic resources through operator space*, arXiv:2501.18679.
[65] F. Verstraete, J. J. Garcia-Ripoll, and J. I. Cirac, *Matrix product density operators: Simulation of finite-temperature and dissipative systems*, Phys. Rev. Lett. **93**, 207204 (2004).
[66] M. J. Hartmann, J. Prior, S. R. Clark, and M. B. Plenio, *Density matrix renormalization group in the Heisenberg picture*, Phys. Rev. Lett. **102**, 057202 (2009).
[67] H.-J. Liao, K. Wang, Z.-S. Zhou, P. Zhang, and T. Xiang, *Simulation of IBM's kicked Ising experiment with projected entangled pair operator*, arXiv:2308.03082.
[68] V. Martinez, A. Angrisani, E. Pankovets, O. Fawzi, and D. Stilck França, *Efficient simulation of parametrized quantum circuits under nonunital noise through Pauli backpropagation*, Phys. Rev. Lett. **134**, 250602 (2025).
[69] *Anonymous, quantum disadvantage*, in *Sigbovik 2024* (Sigbovik, 2024), pp. 199–205.
[70] M. M. Rams, P. Czarnik, and L. Cincio, *Precise extrapolation of the correlation function asymptotics in uniform tensor network states with application to the Bose-Hubbard and XXZ models*, Phys. Rev. X **8**, 041033 (2018).
[71] T. Jones, A. Brown, I. Bush, and S. C. Benjamin, *Quest and high performance simulation of quantum computers*, Sci. Rep. **9**, 10736 (2019).
[72] E. Knill, *Quantum computing with realistically noisy devices*, Nature (London) **434**, 39 (2005).
[73] O. Koska, M. Baboulin, and A. Gazda, *A tree-approach Pauli decomposition algorithm with application to quantum computing*, in *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)* (Prometeus GmbH, 2024), pp. 1–11.
[74] L. Hantzko, L. Binkowski, and S. Gupta, *Pauli transfer matrices*, arXiv:2411.00526.
[75] N. Dowling, P. Kos, and X. Turkeshi, *Magic resources of the Heisenberg picture*, Phys. Rev. Lett. **135**, 050401 (2025).
[76] E. Chitambar and G. Gour, *Quantum resource theories*, Rev. Mod. Phys. **91**, 025001 (2019).
[77] A. Nietner, M. Ioannou, R. Sweke, R. Kueng, J. Eisert, M. Hinsche, and J. Haferkamp, *On the average-case complexity of learning output distributions of quantum circuits*, arXiv:2305.05765.
[78] D. A. Roberts and B. Yoshida, *Chaos and complexity by design*, J. High Energy Phys. 04 (2017) 121.
[79] R. Belyansky, P. Bienias, Y. A. Kharkov, A. V. Gorshkov, and B. Swingle, *Minimal model for fast scrambling*, Phys. Rev. Lett. **125**, 130601 (2020).
[80] M. R. Geller, A. Arrasmith, Z. Holmes, B. Yan, P. J. Coles, and A. Sornborger, *Quantum simulation of operator spreading in the chaotic Ising model*, Phys. Rev. E **105**, 035302 (2022).
[81] A. Nahum, J. Ruhman, S. Vijay, and J. Haah, *Quantum entanglement growth under random unitary dynamics*, Phys. Rev. X **7**, 031016 (2017).
[82] C. W. von Keyserlingk, T. Rakovszky, F. Pollmann, and S. L. Sondhi, *Operator hydrodynamics, OTOCs, and entanglement growth in systems without conservation laws*, Phys. Rev. X **8**, 021013 (2018).
[83] X. Mi, P. Roushan, C. Quintana, S. Mandra, J. Marshall, C. Neill, F. Arute, K. Arya, J. Atalaya, R. Babbush *et al.*, *Information scrambling in quantum circuits*, Science **374**, 1479 (2021).
[84] Y. Kim, A. Eddins, S. Anand, K. X. Wei, E. van den Berg, S. Rosenblatt, H. Nayfeh, Y. Wu, M. Zaletel, K. Temme, and A. Kandala, *Evidence for the utility of quantum computing before fault tolerance*, Nature (London) **618**, 500 (2023).
[85] M. K. Joshi, A. Elben, B. Vermersch, T. Brydges, C. Maier, P. Zoller, R. Blatt, and C. F. Roos, *Quantum information scrambling in a trapped-ion quantum simulator with tunable range interactions*, Phys. Rev. Lett. **124**, 240505 (2020).
[86] D. Bluvstein, A. Omran, H. Levine, A. Keesling, G. Semeghini, S. Ebadi, T. T. Wang, A. A. Michailidis, N. Maskara, W. W. Ho, S. Choi, M. Serbyn, M. Greiner, V. Vuletić, and M. D. Lukin, *Controlling quantum many-body dynamics in driven Rydberg atom arrays*, Science **371**, 1355 (2021).
[87] J. Preskill, *Quantum computing in the NISQ era and beyond*, Quantum **2**, 79 (2018).
[88] Z. Zimboras, B. Koczor, Z. Holmes, E.-M. Borrelli, A. Gilyén, H.-Y. Huang, Z. Cai, A. Acín, L. Aolita, L. Banchi *et al.*, *Myths around quantum computation before full fault tolerance: What no-go theorems rule out and what they don't*, arXiv:2501.05694.
[89] Z.-L. Li and S.-X. Zhang, *The dual role of low-weight Pauli propagation: A flawed simulator but a powerful initializer for variational quantum algorithms*, arXiv:2508.06358.
[90] C.-J. Lin, H. Gharibyan, and V. P. Su, *Utility-scale quantum state preparation: Classical training using Pauli path simulation*, arXiv:2510.02428.
[91] C. Shrikhande, A. Bachhar, A. R. Jimenez, and N. J. Mayhall, *Rapid ground state energy estimation with a sparse Pauli dynamics-enabled variational double bracket flow*, arXiv:2511.21651.
[92] P. J. Roache, *Verification and Validation in Computational Science and Engineering* (Hermosa Albuquerque, NM, 1998), Vol. 895.