Universität Augsburg
Fakultät für Angewandte Informatik

PauliEngine: High-Performant Symbolic Arithmetic for Quantum Operations

Leon Müller , Adelina Bärligea, Alexander Knapp,

Jakob S. Kottmann

23.02.2026

Topics

1 Preliminaries

2 Data structure

3 Benchmarks

4 Initial Applications

5 Conclusion

Preliminaries

Pauli matrices and Pauli strings

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

Pauli strings:

$$P_{\mathbf{k}} = \bigotimes_{i=1}^{N} \sigma_{k_i}$$

Weighted Pauli strings:

$(c, P) \quad c \in \mathbb{C}$

Preliminaries

Hermitian and unitary operators

Hermitian operators:

$$H = \sum_{k=1}^{M} c_k P_k$$

- Sum of weighted Pauli strings
- Hermitian if c_k is real for all k
- Anti-Hermitian if c_k is imaginary for all k
- every N-qubit observable and every generator of a quantum evolution can be written as such a combination

Unitary operators:

- N-qubit quantum operations are represented by a unitary 2^N x 2^N matrix
- Where G is a Hermitian operator

$$U = e^{iG}$$

Preliminaries

Example - Multiplication

Pauli matrices have nice properties:

$$[\sigma_j, \sigma_k] = 2\, i\, \varepsilon_{jkl}\, \sigma_l$$

$$\{\sigma_j, \sigma_k\} = 2\, \delta_{jk}\, I$$

H_1 = X Z + X Y

H_2 = X Y + Z Z

$H_1 * H_2$ = I X + Y I + I I + Y X

Data structure

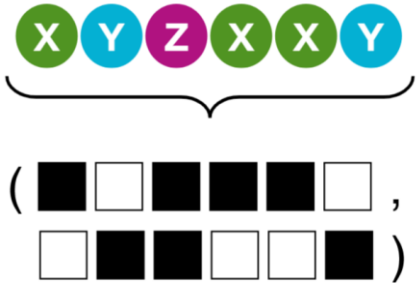
Binary Simplectic Representation

Pauli string encoded into two bit strings called X and Y

$$\boldsymbol{P} = \bigotimes_{i=1}^{n} \sigma_i \rightarrow (\boldsymbol{x}, \boldsymbol{y}) = (x_1 \dots x_n, y_1 \dots y_n)$$

$$\text{with } x_i = \begin{cases} 1 & \text{if } \sigma_i \in \{\mathbf{X}, \mathbf{Z}\} \\ 0 & \text{else} \end{cases}$$

$$\text{and } y_i = \begin{cases} 1 & \text{if } \sigma_i \in \{\mathbf{Y}, \mathbf{Z}\} \\ 0 & \text{else} \end{cases}$$



Data structure

Binary Simplectic Representation

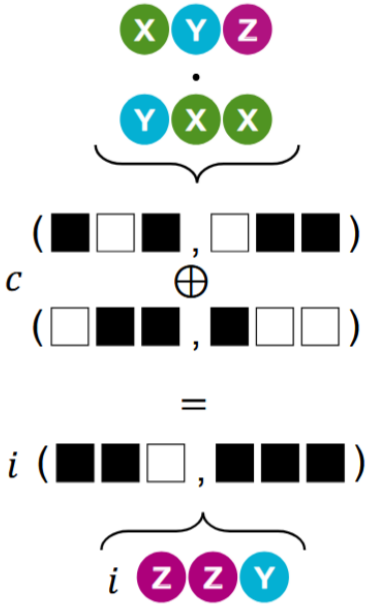
Multiplication of two Pauli strings can now be achieved by a simple XOR-operation on their respective bitstrings

$$P = (x, y)\ ,\quad P' = (x', y')$$

$$P'' = (x'', y'') = P \times P' \qquad \text{where} \quad x'' = x \oplus x' \text{ and } y'' = y \oplus y'$$

This reproduces the multiplication table of single-qubit Pauli matrices

$*$	I	X	Y	Z
1	1	X	Y	Z
X	X	1	iZ	$-iY$
Y	Y	$-iZ$	I	iX
Z	Z	iY	$-iX$	1



Data structure

Binary Simplectic Representation

This does not account for the complex phase that gets introduced when multiplying Pauli matrices

The complex Factors can be recovered with the following formulas:

$$F_+ = (\neg \mathbf{x} \wedge \mathbf{x}' \wedge \mathbf{y} \wedge \mathbf{y}') \oplus (\mathbf{x} \wedge \neg \mathbf{x}' \wedge \neg \mathbf{y} \wedge \mathbf{y}') \oplus (\mathbf{x} \wedge \mathbf{x}' \wedge \mathbf{y} \wedge \neg \mathbf{y}')$$

$$F_- = (\neg \mathbf{x} \wedge \mathbf{x}' \wedge \mathbf{y} \wedge \neg \mathbf{y}') \oplus (\mathbf{x} \wedge \neg \mathbf{x}' \wedge \mathbf{y} \wedge \mathbf{y}') \oplus (\mathbf{x} \wedge \mathbf{x}' \wedge \neg \mathbf{y} \wedge \mathbf{y}')$$

Where F_+ counts the occurrences of the +i factor and F_- counts the occurrences of the −i factor

The total phase factor of the multiplication is then:

$$c = i^{(|F_+| - |F_-|) \mod 4}$$

where $|F\pm|$ denotes the Hamming weight (the number of set bits)

Data structure

Fast commutators

Let $\tau = |F_+| - |F_-|$

if τ even: $[\boldsymbol{P}, \boldsymbol{P}'] = 0$

if τ odd: $[\boldsymbol{P}, \boldsymbol{P}'] = 2\boldsymbol{P}\boldsymbol{P}'$

Only one multiplication is required instead of two

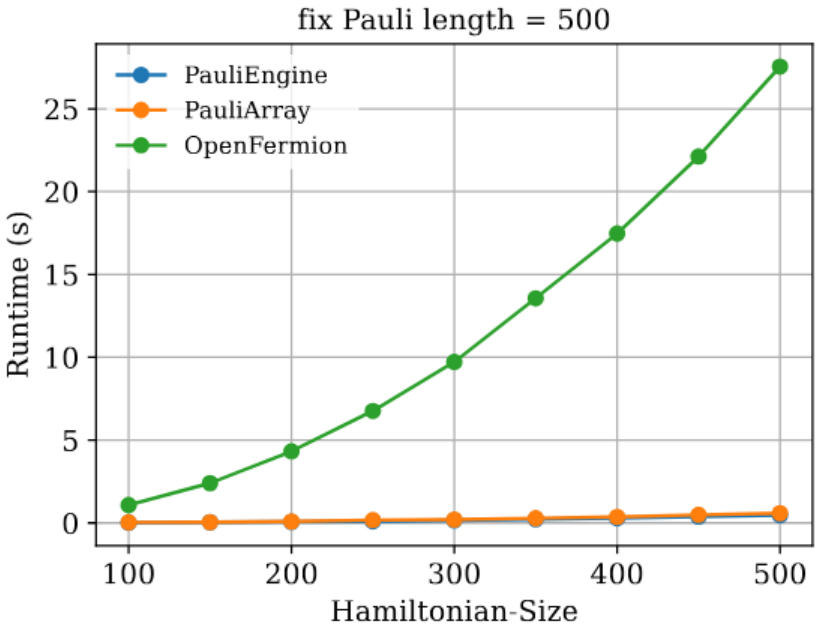
Data structure

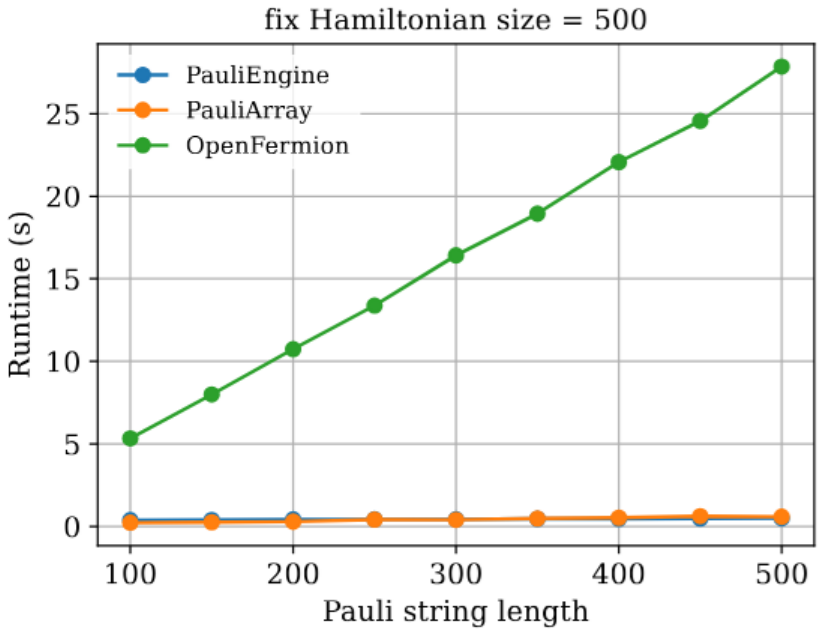
- On top of fast computations, PauliEngine also supports symbolic coefficients for Pauli strings
- This enables :
  - Symbolic differentiation
  - parametrized operators
  - delayed substitution of parameter values

Benchmarks

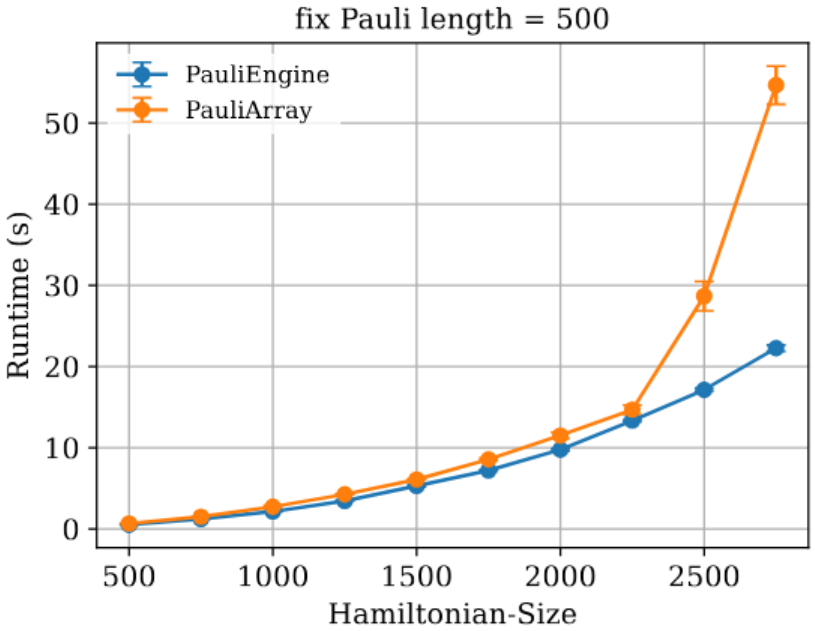
- The runtime of PauliEngine is evaluated through benchmarks comparing it to other frameworks.
- The evaluated operation in the benchmark is the multiplication of two operators
- For each benchmark instance, random Hamiltonians are generated by sampling Pauli strings of a fixed length with uniformly random local Paulis (including the identity)
- Two complementary benchmarks are considered
  - scaling with Hamiltonian size (i.e., the number of Pauli terms), while keeping the Pauli string length fixed at 500.
  - Scaling with Paul string length, while keeping the Hamiltonian size fixed at 500 terms

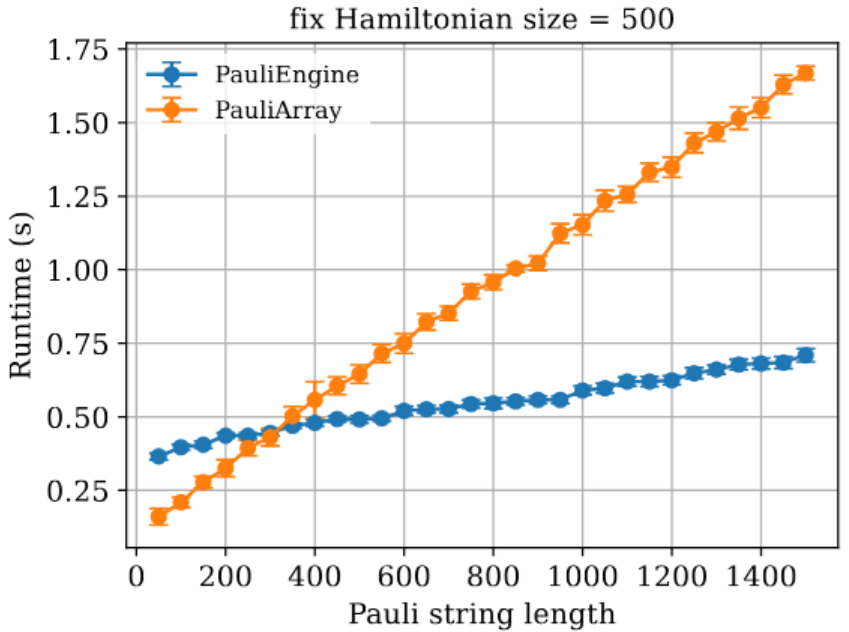
Benchmarks


fix Pauli length = 500
PauliEngine
PauliArray
OpenFermion
Runtime (s)
0
5
10
15
20
25
100
200
300
400
500
Hamiltonian-Size


fix Hamiltonian size = 500
PauliEngine
PauliArray
OpenFermion
Runtime (s)
0
5
10
15
20
25
100
200
300
400
500
Pauli string length

Benchmarks


fix Pauli length = 500
PauliEngine
PauliArray
Runtime (s)
0
10
20
30
40
50
500
1000
1500
2000
2500
Hamiltonian-Size


fix Hamiltonian size = 500
PauliEngine
PauliArray
Runtime (s)
0.25
0.50
0.75
1.00
1.25
1.50
1.75
0
200
400
600
800
1000
1200
1400
Pauli string length

Initial Applications

Fast Lie-Algebraic Classical Simulation

Dynamical Lie Algebras have become the main structural tool to explain the trainability vs expressivity trade-off in variational quantum algorithms and can be used to simulate some of them efficiently:

Given:

- Input state ρ_{in}
- Observable $O = \sum_\alpha (\boldsymbol{\omega})_\alpha h_\alpha$
- Circuit $U(\boldsymbol{\theta}) = \prod_{l=1}^{L} e^{-i\theta_l H_l}$

Get:

Expectation Value

$\langle O(\boldsymbol{\theta}) \rangle \equiv \mathrm{Tr}\left[O U(\boldsymbol{\theta}) \rho_{\text{in}} U^\dagger(\boldsymbol{\theta}) \right]$

Preprocessing

0. Extract set of generators

$\rightarrow \mathcal{G} = \{H_1, \dots, H_L\}$

1. Compute Lie closure
for the dynamical Lie algebra (DLA)

$\rightarrow \mathfrak{g} = \{h_1, \dots h_d\} = \mathrm{span}\langle \mathcal{G} \rangle_{\text{Lie}} \subseteq \mathfrak{su}(2^n)$

with $d = \dim(\mathfrak{g}) = \mathcal{O}(\mathrm{poly}(n))$

2. Compute structure constants
for the adjoint representation

$\rightarrow (\bar{h}_\gamma)_{\alpha\beta} = \frac{\langle h_\gamma, [h_\alpha, h_\beta] \rangle}{\langle h_\gamma, h_\gamma \rangle}$

with $\langle h_\alpha, h_\beta \rangle := tr[h_\alpha^\dagger h_\beta]$

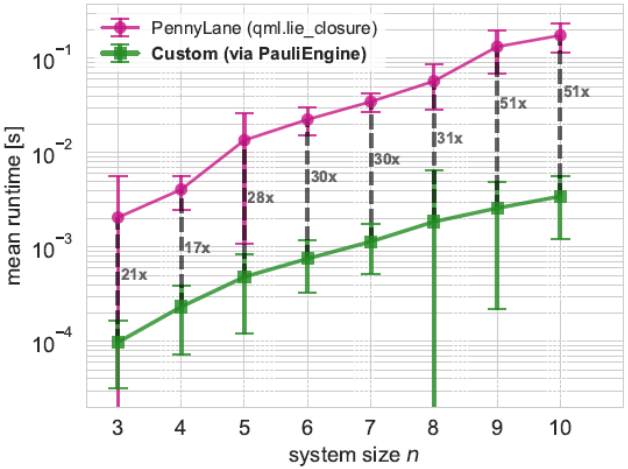
Initial Applications

Fast Lie-Algebraic Classical Simulation

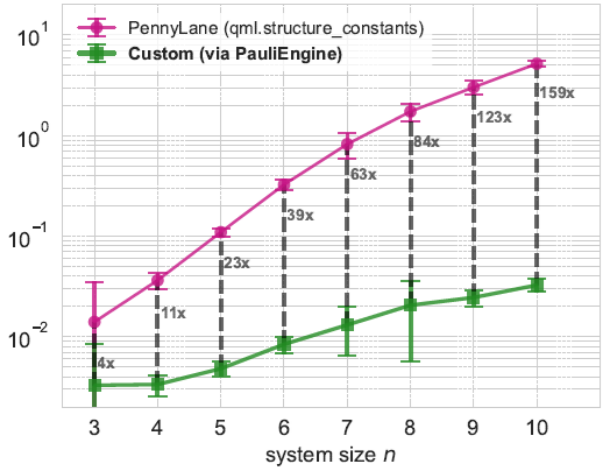
Consider a **Transverse-Field Ising problem** on an open chain:

- The generators $\mathcal{G} = \{Z_i Z_{i+1}\}_{i=1}^{n-1} \cup \{X_i\}_{i=1}^{n}$ form a **set of Pauli strings**.
- We know that $\mathfrak{g} \cong \mathfrak{so}(2n)$ **grows only polynomially** with $d = \mathcal{O}(n^2)$.

We compute…



…the **Lie closure**



…the **structure constants**

Conclusion

- Employs optimized bitwise operations for
  - Fast multiplication
  - Efficient commutator evaluation
- Provides memory-efficient computations for large-scale operator manipulation
- Supports both numerical and symbolic coefficients
- Offers a lightweight Python interface
- Suitable as a building block for quantum software frameworks

Example code

[4]:
```
import PauliEngine as pe
# Example : Z on qubit 0 with coefficient 1.0
p1 = pe.PauliString (1.0, {0: "Z"})

# Example : X on qubit 1 with coefficient "a" ( symbolic )
p2 = pe.PauliString ("a", {1: "X"})

# Example : OpenFermion style
p3 = pe.PauliString ((1.0, [("X", 0), ("Y", 2) ]))
```

[5]:
```
# Multiplication
p5 = p2 * p1
# fast commutators
c = p1.commutator(p2)
```

[8]:
```
# create PauliString objects
p1 = pe.PauliString (1.0 , {0: "Z"}) # not parametrized
p2 = pe.PauliString ("a", {1: "X"})  # parametrized
# assemble operator
H = pe.QubitHamiltonian([p1 , p2 ])
```

[9]:
```
# Differentiate
dH = H.diff ("a")
# Substitute
H2 = H.subs ({"a": 2.0})
```

Thank you!

PauliEngine can be accessed on Github:

Coming soon – Installable via pip