fast_pauli

James E. T. Smith, Eugene Rublenko, Alex Lerner, Sebastien Roy,

Mar 12, 2025

CONTENTS

Welcome to `fast-pauli` from Qognitive, an open-source Python / C++ library for optimized operations on Pauli matrices and Pauli strings based on PauliComposer. In this guide, we'll introduce some of the important operations to help users get started as well as some conceptual background on Pauli matrices and Pauli strings.

If you want to follow along with this guide, please follow the installation instructions in *Introduction*. For more details on our programmatic interface, see the *Python API* or *C++ API* documentation.

CHAPTER

ONE

PAULI MATRICES

For a more in-depth overview of Pauli matrices, see here.

In math and physics, a Pauli matrix, named after the physicist Wolfgang Pauli, is any one of the special 2 x 2 complex matrices in the set (often denoted by the greek letter σ) :

$$\sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

All the Pauli matrices share the properties that they are:

1. Hermitian (equal to their own conjugate transpose) $\sigma_i = \sigma_i^\dagger$ for all $i \in \{x, y, z\}$
2. Involutory (they are their own inverse) $\sigma_i^2 = \sigma_0$ for all $i \in \{x, y, z\}$
3. Unitary (their inverse is equal to their conjugate transpose) $\sigma_i^{-1} = \sigma_i^\dagger$ for all $i \in \{x, y, z\}$

with the identity matrix σ_0 or the 2×2 Identity matrix I being the trivial case.

In `fast_pauli`, we represent pauli matrices using the `Pauli` class. For example, to represent the Pauli matrices, we can do:

```python
import fast_pauli as fp

pauli_0 = fp.Pauli('I')
pauli_x = fp.Pauli('X')
pauli_y = fp.Pauli('Y')
pauli_z = fp.Pauli('Z')

str(pauli_0)  # returns "I"
```

We can also multiply two `Pauli` objects together to get a `Pauli` object representing the matrix product of the two pauli matrices. The result includes a phase factor because the product of two Pauli matrices is not another Pauli matrix, but rather a Pauli matrix multiplied by either `i` or `-i` which we call a phase factor.

For example, we can compute the resulting `Pauli` object from multiplying σ_x and σ_y as follows:

```python
# phase = i, new_pauli = fp.Pauli('Z')
phase, new_pauli = pauli_x @ pauli_y
```

From here, we can also convert our `Pauli` object back to a dense numpy array if we'd like:

```python
pauli_x_np = pauli_x.to_tensor()
```

CHAPTER

TWO

PAULI STRINGS

Pauli strings are tensor-product combinations of Pauli matrices. For example, the following is a valid Pauli string:

$$\hat{\mathcal{P}} = \sigma_x \otimes \sigma_y \otimes \sigma_z$$

where $\otimes$ denotes the tensor or Kronecker product, and we can more simply denote by

$$\hat{\mathcal{P}} = XYZ$$

A Pauli string of length `N` is a tensor product of `N` Pauli matrices. We can represent a `N`-length Pauli String as a $2^N \times 2^N$ operator or matrix (in physics, an operator is represented by a matrix in a given basis), so `XYZ` is a 8×8 matrix. Since these operators can get large very quickly, `fast_pauli` represents Pauli strings sparsely as an array of `Pauli` objects.

For example, to construct the Pauli string `XYZ`, we can do:

```
P = fp.PauliString('XYZ')
```

Pauli Strings also support operations like addition, multiplication, and more. For example:

```
P1 = fp.PauliString('XYZ')
P2 = fp.PauliString('YZX')

# Get dim and n_qubits properties
# dim = 8, n_qubits = 3
P1.dim
P1.n_qubits

# Multiply two Pauli strings.
phase, new_string = P1 @ P2
```

We can also do more complicated things, like compute the action of a Pauli string $\hat{\mathcal{P}}$ on a vector $|\psi\rangle$, $\hat{\mathcal{P}}|\psi\rangle$, or compute the expectation value of a Pauli string with a state $\langle\psi|\hat{\mathcal{P}}|\psi\rangle$. As a side note, in this guide we will use state and vector interchangeably:

```
import numpy as np

# Apply P to a state
P = fp.PauliString('XY')
state = np.array([1, 0, 0, 1], dtype=complex)
new_state = P.apply(state)

# Compute the expected value of P with respect to a state or a batch of states
value = P.expectation_value(state)
```

(continues on next page)

(continued from previous page)

```
states = np.random.randn(4, 8) + 1j * np.random.randn(4, 8)
values = P.expectation_value(states)
```

We can also convert `PauliString` objects back to dense numpy arrays if we'd like, or extract their string representation:

```
P = fp.PauliString('XYZ')
P_np = P.to_tensor()

P_str = str(P) # Returns "XYZ"
```

For more details on the `PauliString` class, see the *Python API* or *C++ API* documentation.

CHAPTER

THREE

PAULI OPERATORS

The `PauliOp` class lets us represent operators that are linear combinations of Pauli strings with complex coefficients. For example, we can represent any arbitrary operator A as a sum of Pauli strings P_i with complex coefficients c_i:

$$A = \sum_i c_i P_i$$

In `fast_pauli`, we can construct `PauliOp` objects using the `PauliOp` constructor. For example, to construct the `PauliOp` object that represents the operator $A = 0.5 * XYZ + 0.5 * YYZ$, we can do:

```
coeffs = np.array([0.5, 0.5], dtype=complex)  # represent c_i in the sum above
pauli_strings = ['XYZ', 'YYZ']  # represent P_i in the sum above
A = fp.PauliOp(coeffs, pauli_strings)

# Get the number of qubits the operator acts on,
# dimension, number of pauli strings
# n_qubits = 3, dim = 8, n_pauli_strings = 2
A.n_qubits
A.dim
A.n_pauli_strings
```

Just like with `PauliString` objects, we can apply `PauliOp` objects to a set of vectors, or compute expectation values, as well as arithmetic operations. Just like with `PauliString` objects, we can also convert `PauliOp` objects back to dense numpy arrays if we'd like or get their string representation, in this case a list of strings:

```
coeffs = np.array([0.5, 0.5], dtype=complex)
pauli_strings = ['XYZ', 'YYZ']
A = fp.PauliOp(coeffs, pauli_strings)

# Adding two Pauli strings returns a PauliOp.
# The returned object is a PauliOp because
# the sum is a linear combination of Pauli strings
P1 = fp.PauliString('XYZ')
P2 = fp.PauliString('YZX')
O = P1 + P2

# PauliOp supports addition, subtraction, multiplication,
# scaling, as well as have PauliString objects
# as the second operand. All valid operations:
A1 = 0.5 * A
A2 = A + A1
A3 = A1 @ A2
```

(continues on next page)

(continued from previous page)

```
s = fp.PauliString('XYZ')
A4 = A1 + s

# Apply A to a single state / vector or set
states = np.random.rand(8, 10) + 1j * np.random.rand(8, 10)
new_states = A.apply(states)

# Compute the expectation value of A with respect to a state
values = A.expectation_value(states)

# Get dense matrix representation of A
A_dense = A.to_tensor()

# ['XYZ', 'YYZ']
A_str = A.pauli_strings_as_str
```

CHAPTER

FOUR

QISKIT INTEGRATION

fast_pauli also has integration with IBM's Qiskit SDK, allowing for easy interfacing with certain Qiskit objects. For example, we can convert between PauliOp objects and SparsePauliOp objects from Qiskit:

```
# Convert a fast_pauli PauliOp to a Qiskit SparsePauliOp object and back
O = fp.PauliOp([1], ['XYZ'])
qiskit_op = fp.to_qiskit(O)
fast_pauli_op = fp.from_qiskit(qiskit_op)

# Convert a fast_pauli PauliString to a Qiskit Pauli object
P = fp.PauliString('XYZ')
qiskit_pauli = fp.to_qiskit(P)
```

For more details on Qiskit conversions, see the *Python API* documentation.

CHAPTER

FIVE

BENCHMARKS

Here at Qognitive, we use Pauli Strings and Pauli Operators throughout our codebase. Some of our most critical performance bottlenecks involve applying these operators to a state or batch of states. These are only a few of the functions we've optimized in `fast-pauli`, check out our *Python API* and *C++ API*. All benchmark figures are interactive and we encourage you to explore them!

Below are several benchmarks comparing `fast-pauli` to `qiskit`. All benchmarks were run on a single machine with the following specifications:

CPU	13th Gen Intel(R) Core(TM) i9-13950HX
RAM	64GB
Threads	32
OS	Ubuntu 22.04.4 LTS
Architecture	x86_64
Compiler (for `fast-pauli`)	LLVM 18.1.8
Python	3.12.7

5.1 Pauli String Applied to a State

Starting simply, we benchmarked applying a single Pauli String ($\hat{\mathcal{P}}$) to a single state (ψ), which is equivalent to the following expression:

$$\hat{\mathcal{P}}\psi \tag{5.1}$$

We saw that the sparse representation of the Pauli String operator when applied to the state is significantly faster than the representation of the Pauli String operator used by Qiskit. For most operator sizes, we saw several orders of magnitude in performance improvement.

> **Note**
>
> All datapoints in our benchmarks have error bars indicating the standard deviation of the mean, but for most points the error bars are too small to see.

5.2 Pauli Operator Applied to a State

Next we benchmarked applying a Pauli Operator (a linear combination of Pauli Strings) to a single state:

$$\big(\sum_i c_i \hat{\mathcal{P}}_{\rangle}\big)\psi \tag{5.2}$$

Again, we saw significant performance improvements for the same reasons stated above and are often an order of magnitude faster than `qiskit`. $N_{\text{pauli strings}}$ is the number of Pauli Strings in the Pauli Operator, i.e. the number of terms in the linear combination shown in (5.2). Note that `fast-pauli` performs better relative to `qiskit` when the Pauli Operator is more sparse, i.e. when there are fewer Pauli Strings in the operator.

5.3 Expectation Values of a Pauli Operator

Finally, we benchmarked the expectation values of a Pauli Operator applied to a **batch** of states:

$$\psi_t \sum_i c_i \hat{\mathcal{P}}_\rangle \psi_t \tag{5.3}$$

In this benchmark, we chose a single number of Pauli Strings, $N_{\text{pauli strings}} = 1024$, and varied the number of qubits and states. Similar to the previous benchmarks, we saw significant performance improvements for `fast-pauli` compared to `qiskit`. In this benchmark, we tend to perform better when applying to a larger batch of states, but we point out that our advantage compared to `qiskit` narrows as the number of qubits increases. With that said, we're still more than 2x faster for these larger operators!

> **Note**
>
> The data point for `qiskit` with $N_{\text{qubits}} = 16$ and $N_{\text{states}} = 1000$ was not shown in the above plot because of OOM errors.

CHAPTER

SIX

PYTHON API

6.1 Pauli

class fast_pauli.**Pauli**(**args*, ***kwargs*)

A class for efficient representation of a 2×2 Pauli Matrix $\sigma_i \in \{I, X, Y, Z\}$

__getstate__

__init__

Overloaded function.

1. `__init__(self) -> None`

Default constructor to initialize with identity matrix.

2. `__init__(self, code: int) -> None`

Constructor given a numeric code.

Parameters

code (*int*) – Numerical label of type int for corresponding Pauli matrix $0 : I, 1 : X, 2 : Y, 3 : Z$

3. `__init__(self, symbol: str) -> None`

Constructor given Pauli matrix symbol.

Parameters

symbol (*str*) – Character label of type str corresponding to one of the Pauli Matrix symbols I, X, Y, Z

__matmul__

Returns matrix product of two Paulis as a tuple of phase and new Pauli object.

Parameters

rhs (Pauli) – Right hand side Pauli object

Returns

Phase and resulting Pauli object

Return type

tuple[complex, *Pauli*]

classmethod __new__(**args*, ***kwargs*)

__setstate__

`__str__`

Returns a string representation of Pauli matrix.

Returns
One of I, X, Y, Z, a single character string representing a Pauli Matrix

Return type
str

`clone`

Returns a copy of the Pauli object.

Returns
A copy of the Pauli object

Return type
Pauli

`to_tensor`

Returns a dense representation of Pauli object as a 2×2 matrix.

Returns
2D numpy array of complex numbers

Return type
np.ndarray

6.2 PauliString

class fast_pauli.**PauliString**(**args*, ***kwargs*)

A class representation of a Pauli String $\hat{\mathcal{P}}$ (i.e. a tensor product of Pauli matrices)

$$\hat{\mathcal{P}} = \bigotimes_i \sigma_i$$
$$\sigma_i \in \{I, X, Y, Z\}$$

`__add__`

Returns the sum of two Pauli strings in a form of PauliOp object.

Parameters
rhs (`PauliString`) – The other PauliString object to add

Returns
A linear combination of the PauliString objects as a PauliOp.

Return type
PauliOp

`__getstate__`

`__init__`

Overloaded function.

1. `__init__(self) -> None`

Default constructor to initialize with empty string.

2. `__init__(self, string: str) -> None`

Constructs a PauliString from a string and calculates the weight. This is often the most compact way to initialize a PauliString.

Parameters
`string` (*str*) – Pauli String representation. Each character should be one of I, X, Y, Z

3. `__init__(self, paulis: collections.abc.Sequence[fast_pauli._fast_pauli.Pauli]) -> None`

Constructs a PauliString from a list of Pauli objects and calculates the weight.

Parameters
`paulis` (*list[Pauli]*) – List of ordered Pauli objects

`__matmul__`

Returns matrix product of two pauli strings and their phase as a pair.

Parameters
`rhs` (`PauliString`) – Right hand side PauliString object

Returns
Phase and resulting PauliString object

Return type
tuple[complex, *PauliString*]

`classmethod __new__`(**args*, ***kwargs*)

`__setstate__`

`__str__`

Returns a string representation of PauliString object.

Returns
string representation of PauliString object

Return type
str

`__sub__`

Returns the difference of two Pauli strings in a form of PauliOp object.

Parameters
`rhs` (`PauliString`) – The other PauliString object to subtract

Returns
A linear combination of the PauliString objects as a PauliOp.

Return type
PauliOp

`apply`

Apply a Pauli string to a single dimensional state vector or a batch of states.

$$c\hat{\mathcal{P}}\psi_t$$

> **Note**
>
> For batch mode it applies the PauliString to each individual state separately. In this case, the input array is expected to have the shape of (n_dims, n_states) with states stored as columns.

Parameters

- **`states`** (*np.ndarray*) – The original state(s) represented as 1D (n_dims,) or 2D numpy array (n_dims, n_states) for batched calculation. Outer dimension must match the dimensionality of Pauli string.
- **`coeff`** (*complex*) – Scalar multiplication factor (c) to scale the PauliString before applying to states

Returns
New state(s) in a form of 1D (n_dims,) or 2D numpy array (n_dims, n_states) according to the shape of input states

Return type
np.ndarray

`clone`

Returns a copy of the PauliString object.

Returns
A copy of the PauliString object

Return type
PauliString

`property dim`

The dimension of PauliString 2^n, n - number of qubits

Type
int

`expectation_value`

Calculate expectation value(s) for a given single dimensional state vector or a batch of states.

$$\psi_t \hat{\mathcal{P}} \psi_t$$

> **Note**
>
> For batch mode it computes the expectation value for each individual state separately. In this case, the input array is expected to have the shape of (n_dims, n_states) with states stored as columns.

Parameters

- **`states`** (*np.ndarray*) – The original state(s) represented as 1D (n_dims,) or 2D numpy array (n_dims, n_states) for batched calculation. Outer dimension must match the dimensionality of Pauli string.
- **`coeff`** (*complex*) – Multiplication factor to scale the PauliString before calculating the expectation value

Returns
Expectation value(s) in the form of a 1D numpy array with a shape of (n_states,)

Return type
np.ndarray

`property n_qubits`

The number of qubits in PauliString (i.e. number of Pauli Matrices in tensor product)

Type
int

`to_tensor`

Returns a dense representation of PauliString.

Returns
2D numpy array of complex numbers

Return type
np.ndarray

`property weight`

The weight of PauliString (i.e. number of non-identity Pauli matrices in it)

Type
int

6.3 PauliOp

`class fast_pauli.PauliOp(*args, **kwargs)`

A class representation for a Pauli Operator A (i.e. a weighted sum of Pauli Strings)

$$A = \sum_j h_j \hat{\mathcal{P}}_j$$

$$\hat{\mathcal{P}} = \bigotimes_i \sigma_i \quad h_j \in \mathbb{C}$$

`__add__`

Overloaded function.

1. `__add__(self, arg: fast_pauli._fast_pauli.PauliOp, /) -> fast_pauli._fast_pauli.PauliOp`

Returns the sum of two Pauli Operators.

Parameters
rhs (`PauliOp`) – The other PauliOp object to add

Returns
New PauliOp instance holding the sum.

Return type
PauliOp

2. `__add__(self, arg: fast_pauli._fast_pauli.PauliString, /) -> fast_pauli._fast_pauli.PauliOp`

Returns the sum of Pauli Operator with Pauli String.

Parameters
: **rhs** (PauliString) – Right hand side PauliString object to add

Returns
: New PauliOp instance holding the sum.

Return type
: *PauliOp*

`__getstate__`

`__iadd__`

Overloaded function.

1. `__iadd__(self, arg: fast_pauli._fast_pauli.PauliOp, /) -> fast_pauli._fast_pauli.PauliOp`

Performs inplace addition with other Pauli Operator.

Parameters
: **other** (PauliOp) – Pauli operator object to add

Returns
: Current PauliOp instance after addition

Return type
: *PauliOp*

2. `__iadd__(self, arg: fast_pauli._fast_pauli.PauliString, /) -> fast_pauli._fast_pauli.PauliOp`

Performs inplace addition with Pauli String.

Parameters
: **other** (PauliString) – Pauli string object to add

Returns
: Current PauliOp instance after addition

Return type
: *PauliOp*

`__imul__`

Scale Pauli Operator inplace by a scalar value.

Parameters
: **other** (*complex or float*) – Scalar multiplier

Returns
: Current PauliOp instance after scaling

Return type
: *PauliOp*

`__init__`

Overloaded function.

1. `__init__(self) -> None`

Default constructor to initialize strings and coefficients with empty arrays.

2. `__init__(self, pauli_strings: collections.abc.Sequence[str]) -> None`

Construct a PauliOp from a list of strings and default corresponding coefficients to ones.

Parameters
pauli_strings (*List[str]*) – List of Pauli Strings as simple *str*. Each string should be composed of characters I, X, Y, Z and should have the same size

3. `__init__(self, arg: collections.abc.Sequence[fast_pauli._fast_pauli.PauliString], /) -> None`

Construct a PauliOp from a list of PauliString objects and default corresponding coefficients to ones.

Parameters
pauli_strings (*List[PauliString]*) – List of PauliString objects.

4. `__init__(self, coefficients: ndarray[dtype=complex128], pauli_strings: collections.abc.Sequence[fast_pauli._fast_pauli.PauliString]) -> None`

Construct a PauliOp from a list of PauliString objects and corresponding coefficients.

Parameters

- **coefficients** (*np.ndarray*) – Array of coefficients corresponding to Pauli strings.
- **pauli_strings** (*List[PauliString]*) – List of PauliString objects.

5. `__init__(self, coefficients: collections.abc.Sequence[complex], pauli_strings: collections.abc.Sequence[fast_pauli._fast_pauli.PauliString]) -> None`

Construct a PauliOp from a list of PauliString objects and corresponding coefficients.

Parameters

- **coefficients** (*List[complex]*) – List of coefficients corresponding to Pauli strings.
- **pauli_strings** (*List[PauliString]*) – List of PauliString objects.

6. `__init__(self, coefficients: collections.abc.Sequence[complex], pauli_strings: collections.abc.Sequence[str]) -> None`

Construct a PauliOp from a list of strings and corresponding coefficients.

Parameters

- **coefficients** (*np.ndarray*) – Array of coefficients corresponding to Pauli strings.
- **pauli_strings** (*List[str]*) – List of Pauli Strings as simple *str*. Each string should be composed of characters I, X, Y, Z and should have the same size

`__isub__`

Overloaded function.

1. `__isub__(self, arg: fast_pauli._fast_pauli.PauliOp, /) -> fast_pauli._fast_pauli.PauliOp`

Performs inplace subtraction with other Pauli Operator.

Parameters
other (*PauliOp*) – Pauli operator object to subtract

Returns
Current PauliOp instance after subtraction

Return type
PauliOp

2. `__isub__(self, arg: fast_pauli._fast_pauli.PauliString, /) -> fast_pauli._fast_pauli.PauliOp`

Performs inplace subtraction with Pauli String.

Parameters
other (`PauliString`) – Pauli string object to subtract

Returns
Current PauliOp instance after subtraction

Return type
PauliOp

`__matmul__`

Overloaded function.

1. `__matmul__(self, arg: fast_pauli._fast_pauli.PauliOp, /) -> fast_pauli._fast_pauli.PauliOp`

Efficient matrix multiplication of two Pauli Operators, leveraging their sparse structure.

Parameters
rhs (`PauliOp`) – Right hand side PauliOp object

Returns
New PauliOp instance containing the product

Return type
PauliOp

2. `__matmul__(self, arg: fast_pauli._fast_pauli.PauliString, /) -> fast_pauli._fast_pauli.PauliOp`

Efficient matrix multiplication of PauliOp with a PauliString on the right, leveraging their sparse structure.

Parameters
rhs (`PauliString`) – Right hand side PauliString object

Returns
New PauliOp instance containing the product

Return type
PauliOp

`__mul__`

Scale Pauli Operator by a scalar value.

Parameters
rhs (*`complex or float`*) – Right hand side scalar multiplier

Returns
New PauliOp instance containing the product

Return type
PauliOp

classmethod __new__(**args*, ***kwargs*)

__radd__

Returns the sum of Pauli Operators with Pauli String.

Parameters
lhs (PauliString) – Left hand side PauliString object to add

Returns
New PauliOp instance holding the sum.

Return type
PauliOp

__rmatmul__

Efficient matrix multiplication of PauliOp with a PauliString on the left, leveraging their sparse structure.

Parameters
rhs (PauliOp) – Left hand side PauliOp object

Returns
New PauliOp instance containing the product

Return type
PauliOp

__rmul__

Scale Pauli Operator by a scalar value.

Parameters
lhs (*complex or float*) – Left hand side scalar multiplier

Returns
New PauliOp instance containing the product

Return type
PauliOp

__rsub__

Returns the difference of Pauli Operators with Pauli String.

Parameters
lhs (PauliString) – Left hand side PauliString object to subtract

Returns
New PauliOp instance holding the difference.

Return type
PauliOp

__setstate__

__sub__

Overloaded function.

1. `__sub__(self, arg: fast_pauli._fast_pauli.PauliOp, /) -> fast_pauli._fast_pauli.PauliOp`

Returns the difference of two Pauli Operators.

Parameters
rhs (PauliOp) – The other PauliOp object to subtract

Returns
New PauliOp instance holding the difference.

Return type
PauliOp

2. `__sub__(self, arg: fast_pauli._fast_pauli.PauliString, /) -> fast_pauli._fast_pauli.PauliOp`

Returns the difference of Pauli Operator with Pauli String.

Parameters
`rhs` (`PauliString`) – Right hand side PauliString object to subtract

Returns
New PauliOp instance holding the difference.

Return type
PauliOp

`apply`

Apply a Pauli Operator to a single dimensional state vector or a batch of states.

$$(\sum_j h_j \hat{\mathcal{P}}_j)\psi_t$$

> **Note**
>
> For batch mode it applies the PauliOp to each individual state separately. In this case, the input array is expected to have the shape of (n_dims, n_states) with states stored as columns.

Parameters
`states` (*`np.ndarray`*) – The original state(s) represented as 1D (n_dims,) or 2D numpy array (n_dims, n_states) for batched calculation. Outer dimension must match the dimensionality of Pauli Operator.

Returns
New state(s) in a form of 1D (n_dims,) or 2D numpy array (n_dims, n_states) according to the shape of input states

Return type
np.ndarray

`clone`

Returns a copy of the PauliOp object.

Returns
A copy of the PauliOp object

Return type
PauliOp

`property coeffs`

The list of coefficients corresponding to Pauli strings

Type
List[complex]

`property dim`

The dimension of PauliStrings used to compose PauliOp 2^n, n - number of qubits

Type
int

`expectation_value`

Calculate expectation value(s) for a given single dimensional state vector or a batch of states.

$$\psi_t(\sum_j h_j \hat{\mathcal{P}}_j)\psi_t$$

> **Note**
>
> For batch mode it computes the expectation value for each individual state separately. In this case, the input array is expected to have the shape of (n_dims, n_states) with states stored as columns.

Parameters
states (*np.ndarray*) – The original state(s) represented as 1D (n_dims,) or 2D numpy array (n_dims, n_states) for batched calculation. Outer dimension must match the dimensionality of Pauli Operator.

Returns
Expectation value(s) in the form of a 1D numpy array with a shape of (n_states,)

Return type
np.ndarray

`extend`

Overloaded function.

1. `extend(self, other: fast_pauli._fast_pauli.PauliOp) -> None`

Add another PauliOp to the current one by extending the internal summation with new terms.

Parameters
other (PauliOp) – PauliOp object to extend the current one with

2. `extend(self, other: fast_pauli._fast_pauli.PauliString, multiplier: complex, dedupe: bool = True) -> None`

Add a Pauli String term with a corresponding coefficient to the summation inside PauliOp.

Parameters

- **other** (PauliString) – PauliString object to add to the summation
- **multiplier** (*complex*) – Coefficient to apply to the PauliString
- **dedupe** (*bool*) – Whether to deduplicate the set of PauliStrings

`property n_pauli_strings`

The number of PauliString terms in PauliOp

Type
int

property n_qubits

The number of qubits in PauliOp

Type
int

property pauli_strings

The list of PauliString objects corresponding to coefficients in PauliOp

Type
List[*PauliString*]

property pauli_strings_as_str

The list of PauliString representations corresponding to coefficients in PauliOp

Type
List[str]

scale

Overloaded function.

1. `scale(self, factor: complex) -> None`

Scale each individual term of Pauli Operator by a scalar value.

Parameters
factor (*complex or float*) – Scalar multiplier

2. `scale(self, factors: ndarray[dtype=complex128]) -> None`

Scale each individual term of Pauli Operator by a scalar value.

Parameters
factors (*np.ndarray*) – Array of factors to scale each term with. The length of the array should match the number of Pauli strings in PauliOp

to_tensor

Returns a dense representation of PauliOp.

Returns
2D numpy array of complex numbers with a shape of $2^n \times 2^n, n$ - number of qubits

Return type
np.ndarray

6.4 SummedPauliOp

class fast_pauli.**SummedPauliOp**(**args*, ***kwargs*)

__getstate__

__init__

Overloaded function.

2. `__init__(self, pauli_strings: collections.abc.Sequence[fast_pauli._fast_pauli.PauliString], coeffs: ndarray[dtype=complex128]) -> None`

Initialize SummedPauliOp from PauliStrings and coefficients.

Parameters

- **pauli_strings** (*List[PauliString]*) – List of PauliStrings to use in the SummedPauliOp (n_pauli_strings,)
- **coeffs** (*np.ndarray*) – Array of coefficients corresponding to the PauliStrings (n_pauli_strings, n_operators)

Returns
New SummedPauliOp instance

Return type
SummedPauliOp

3. `__init__(self, pauli_strings: collections.abc.Sequence[str], coeffs: ndarray[dtype=complex128]) -> None`

Initialize SummedPauliOp from PauliStrings and coefficients.

Parameters

- **pauli_strings** (*List[str]*) – List of PauliStrings to use in the SummedPauliOp (n_pauli_strings,)
- **coeffs** (*np.ndarray*) – Array of coefficients corresponding to the PauliStrings (n_pauli_strings, n_operators)

Returns
New SummedPauliOp instance

Return type
SummedPauliOp

classmethod __new__(**args*, ***kwargs*)

__setstate__

apply

Apply the SummedPauliOp to a batch of states.

$$(\sum_k \sum_i h_{ik} \hat{\mathcal{P}}_i) \psi_t$$

Parameters
states (*np.ndarray*) – The original state(s) represented as 2D numpy array (n_dims, n_states) for batched calculation.

Returns
New state(s) in a form of 2D numpy array (n_dims, n_states) according to the shape of input states

Return type
np.ndarray

apply_weighted

Apply the SummedPauliOp to a batch of states with corresponding weights.

$$(\sum_k x_{tk} \sum_i h_{ik} \hat{\mathcal{P}}_i) \psi_t$$

Parameters

- **states** (*np.ndarray*) – The original state(s) represented as 2D numpy array (n_dims, n_states) for batched calculation.

- **data** (*np.ndarray*) – The data to weight the operators corresponding to the states (n_operators, n_states)

Returns
New state(s) in a form of 2D numpy array (n_dims, n_states) according to the shape of input states

Return type
np.ndarray

clone

Returns a copy of the SummedPauliOp object.

Returns
A copy of the SummedPauliOp object

Return type
SummedPauliOp

property coeffs

Getter and setter for coefficients.

Returns
Array of coefficients corresponding with shape (n_operators, n_pauli_strings)

Return type
np.ndarray

property dim

Return the Hilbert space dimension of the SummedPauliOp.

Returns
Hilbert space dimension

Return type
int

expectation_value

Calculate expectation value(s) for a given batch of states.

$$\psi_t(\sum_k \sum_i h_{ik}\hat{\mathcal{P}}_i)\psi_t$$

Parameters
states (*np.ndarray*) – The state(s) represented as 2D numpy array (n_operators, n_states) for batched calculation.

Returns
Expectation value(s) in a form of 2D numpy array (n_operators, n_states) according to the shape of input states

Return type
np.ndarray

property n_operators

Return the number of Pauli operators in the SummedPauliOp.

Returns
Number of operators

Return type
int

property n_pauli_strings

Return the number of PauliStrings in the SummedPauliOp.

Returns
Number of PauliStrings

Return type
int

property pauli_strings

The list of PauliString objects corresponding to coefficients in SummedPauliOp

Type
List[*PauliString*]

property pauli_strings_as_str

The list of Pauli Strings representations corresponding to coefficients from SummedPauliOp

Type
List[str]

split

Returns all components of the SummedPauliOp expressed as a vector of PauliOps.

Returns
Components of the SummedPauliOp object

Return type
List[fp.PauliOp]

square

Square the SummedPauliOp.

Returns
New SummedPauliOp instance

Return type
SummedPauliOp

to_tensor

Returns a dense representation of SummedPauliOp.

Returns
3D numpy array of complex numbers with a shape of (n_operators, 2^n_qubits, 2^n_qubits)

Return type
np.ndarray

6.5 Helpers

fast_pauli.helpers.**calculate_pauli_strings**(*n_qubits: int*, *weight: int*) → list[*fast_pauli._fast_pauli.PauliString*]

Calculate all Pauli strings for a given weight.

Parameters

- **n_qubits** (*int*) – Number of qubits
- **weight** (*int*) – Weight of Pauli strings to return

Returns
List of PauliStrings

Return type
List[*PauliString*]

`fast_pauli.helpers.`**`calculate_pauli_strings_max_weight`**(*n_qubits: int*, *weight: int*) → list[*fast_pauli._fast_pauli.PauliString*]

Calculate all Pauli strings up to and including a given weight.

Parameters

- **`n_qubits`** (*`int`*) – Number of qubits
- **`weight`** (*`int`*) – Maximum weight of Pauli strings to return

Returns
List of PauliStrings

Return type
List[*PauliString*]

`fast_pauli.helpers.`**`pauli_string_sparse_repr`**(*paulis: collections.abc.Sequence[*fast_pauli._fast_pauli.Pauli*]*) → tuple[list[int], list[complex]]

Get a sparse representation of a list of Pauli strings.

Parameters
`paulis` (*`List[PauliString]`*) – List of PauliStrings

Returns
List of tuples representing the Pauli string in a sparse format

Return type
List[Tuple[int, int]]

`fast_pauli.helpers.`**`get_nontrivial_paulis`**(*weight: int*) → list[str]

Get all nontrivial Pauli strings up to a given weight.

Parameters
`weight` (*`int`*) – Maximum weight of Pauli strings to return

Returns
List of PauliStrings as strings

Return type
List[str]

CHAPTER

SEVEN

C++ API

7.1 Pauli

struct **Pauli**

A class for efficient representation of a 2x2 *Pauli* matrix $\sigma_i \in \{I, X, Y, Z\}$.

Public Functions

inline constexpr **Pauli**()

Default constructor, initializes to I.

template<class T> inline requires constexpr std::convertible_to< T, uint8_t > Pauli (T const code)

Constructor given a numeric code.

Template Parameters

T – Any type convertible to uint8_t

Parameters

code – 0: I, 1: X, 2: Y, 3: Z

Returns

requires

inline constexpr **Pauli**(char const symbol)

Constructor given *Pauli* matrix symbol.

Parameters

symbol – pauli matrix - I, X, Y, Z

Returns

template<std::floating_point **T**>
inline void **to_tensor**(std::mdspan<std::complex<*T*>, std::dextents<size_t, 2>> output) const

Returns the pauli matrix as a 2D vector of complex numbers.

Template Parameters

T – floating point type

Parameters

output – 2D mdspan to write the 2x2 pauli matrix

Friends

inline friend std::pair<std::complex<double>, *Pauli*> **operator***(*Pauli* const &lhs, *Pauli* const &rhs)

Returns the product of two pauli matrices and their phase as a pair.

Parameters

- **lhs** – left hand side pauli object
- **rhs** – right hand side pauli object

Returns

std::pair<std::complex<double>, *Pauli*> phase and resulting pauli matrix

7.2 PauliString

struct **PauliString**

A class representation of a *Pauli* string (i.e. a tensor product of 2x2 pauli matrices) $= \bigotimes_i \sigma_i$ where $\sigma_i \in \{I, X, Y, Z\}$.

Public Functions

PauliString() noexcept = default

Default constructor, initialize weight and empty vector for paulis.

inline **PauliString**(std::vector<*Pauli*> paulis)

Constructs a *PauliString* from a vector of pauli matrices and calculates the weight.

inline **PauliString**(std::span<fast_pauli::*Pauli*> paulis)

Constructs a *PauliString* from a span of pauli matrices and calculates the weight.

inline **PauliString**(std::string const &str)

Constructs a *PauliString* from a string and calculates the weight. This is often the most compact way to initialize a *PauliString*.

inline **PauliString**(char const *str)

Allows implicit conversion of string literals to PauliStrings. Ex: std::vector<PauliString> pauli_strings = {“IXYZ”, “IIIII”};.

inline size_t **n_qubits**() const noexcept

Return the number of qubits in the *PauliString*.

Returns

size_t

inline size_t **dim**() const noexcept

Return the dimension (2^n_qubits) of the *PauliString*.

> **Note**
>
> this returns 0 if the *PauliString* is empty.

Returns

size_t

template<std::floating_point **T**>

inline void **apply**(std::mdspan<std::complex<*T*>, std::dextents<size_t, 1>> new_states, std::mdspan<std::complex<*T*>, std::dextents<size_t, 1>> states, std::complex<*T*> const c = 1.0) const

Apply a pauli string (using the sparse representation) to a vector. This performs following matrix-vector multiplication $\hat{\mathcal{P}}\psi$.

Template Parameters

`T` – The floating point base to use for all the complex numbers

Parameters

- `new_states` – Output state
- `states` – The input vector to apply the *PauliString* to. Must be the
- `c` – Multiplication factor to apply to the *PauliString* same size as *PauliString.dim()*.

template<std::floating_point **T**, execution_policy **ExecutionPolicy**>
inline void **apply**(*ExecutionPolicy*&&, std::mdspan<std::complex<*T*>, std::dextents<size_t, 1>> new_states, std::mdspan<std::complex<*T*>, std::dextents<size_t, 1>> states, std::complex<*T*> const c = 1.0) const

Template Parameters

- `T` –
- `ExecutionPolicy` –

Parameters

- `new_states` –
- `states` –

template<std::floating_point **T**>
inline void **apply_batch**(std::mdspan<std::complex<*T*>, std::dextents<size_t, 2>> new_states_T, std::mdspan<std::complex<*T*>, std::dextents<size_t, 2>> const states_T, std::complex<*T*> const c) const

Apply the *PauliString* to a batch of states. This function takes a different shape of the states than the other apply functions. here all the states (new and old) are transposed so their shape is (n_dims x n_states). All the new_stats are overwritten, no need to initialize.

This performs following matrix-matrix multiplication $\hat{\mathcal{P}}\hat{\Psi}$ where matrix $\hat{\Psi}$ has ψ_t as columns

Template Parameters

`T` – The floating point base to use for all the complex numbers

Parameters

- `new_states_T` – The output states after applying the *PauliString* (n_dim x n_states)
- `states_T` – THe original states to apply the *PauliString* to (n_dim x n_states)
- `c` – Multiplication factor to apply to the *PauliString*

template<std::floating_point **T**, execution_policy **ExecutionPolicy**>
inline void **apply_batch**(*ExecutionPolicy*&&, std::mdspan<std::complex<*T*>, std::dextents<size_t, 2>> new_states_T, std::mdspan<std::complex<*T*>, std::dextents<size_t, 2>> const states_T, std::complex<*T*> const c) const

This performs following matrix-matrix multiplication $\hat{\mathcal{P}}\hat{\Psi}$ where matrix $\hat{\Psi}$ has ψ_t as columns

Template Parameters

- **T** – The floating point base to use for all the complex numbers
- **T** –
- **ExecutionPolicy** –

Parameters

- **new_states_T** – The output states after applying the *PauliString* (n_dim x n_states)
- **states_T** – THe original states to apply the *PauliString* to (n_dim x n_states)
- **c** – Multiplication factor to apply to the *PauliString*
- **new_states_T** –
- **states_T** –
- **c** –

template<std::floating_point **T**>
inline void **expectation_value**(std::mdspan<std::complex<*T*>, std::dextents<size_t, 1>> expectation_vals_out, std::mdspan<std::complex<*T*>, std::dextents<size_t, 2>> states, std::complex<*T*> const c = 1.0) const

Calculate expectation values for a given batch of states. This function takes in transposed states with (n_dims x n_states) shape.

It computes following inner product $\psi_t \hat{\mathcal{P}}_\rangle \psi_t$ for each state ψ_t from provided batch.

> **Note**
>
> The expectation values are added to corresponding coordinates in the expectation_vals_out vector.

Template Parameters

T – The floating point base to use for all the complex numbers

Parameters

- **expectation_vals_out** – accumulator for expectation values for each state in the batch
- **states** – THe original states to apply the *PauliString* to (n_dim x n_states)
- **c** – Multiplication factor to apply to the *PauliString*

template<std::floating_point **T**, execution_policy **ExecutionPolicy**>
inline void **expectation_value**(*ExecutionPolicy*&&, std::mdspan<std::complex<*T*>, std::dextents<size_t, 1>> expectation_vals_out, std::mdspan<std::complex<*T*>, std::dextents<size_t, 2>> states, std::complex<*T*> const c = 1.0) const

It computes following inner product $\psi_t \hat{\mathcal{P}}_\rangle \psi_t$ for each state ψ_t from provided batch.

> **Note**
>
> The expectation values are added to corresponding coordinates in the expectation_vals_out vector.

Template Parameters

- **T** – The floating point base to use for all the complex numbers
- **T** –

- **ExecutionPolicy** –

Parameters

- **expectation_vals_out** – accumulator for expectation values for each state in the batch
- **states** – THe original states to apply the *PauliString* to (n_dim x n_states)
- **c** – Multiplication factor to apply to the *PauliString*
- **expectation_vals_out** –
- **states** –
- **c** –

template<std::floating_point **T**>
inline void **to_tensor**(std::mdspan<std::complex<*T*>, std::dextents<size_t, 2>> output) const

Get the dense representation of the object as a 2D-array.

Template Parameters
T – The floating point base to use for all the complex numbers

Parameters
output – The output tensor to fill with the dense representation

Friends

inline friend std::pair<std::complex<double>, *PauliString*> **operator***(*PauliString* const &lhs, *PauliString* const &rhs)

Returns the result of matrix multiplication of two PauliStrings and their phase as a pair.

Parameters

- **lhs** – left hand side *PauliString*
- **rhs** – right hand side *PauliString*

Returns
std::pair<std::complex<double>, *PauliString*> phase and resulting *PauliString*

7.3 PauliOp

template<std::floating_point **T**, typename **H** = std::complex<*T*>>

struct **PauliOp**

A class representation for a *Pauli* Operator (i.e. a weighted sum of *Pauli* Strings) $\left(\sum_i h_i \hat{\mathcal{P}}_i\right)$ where $\hat{\mathcal{P}}_i$ are composed using $\sigma_i \in \{I, X, Y, Z\}$ and h_i are the coefficients.

Public Functions

PauliOp() = default

Default constructor, initialize empty vectors for paulis and coefficients.

inline **PauliOp**(std::vector<std::string> const &strings)

Construct a *PauliOp* from a vector of strings and default corresponding coeffs to ones.

Parameters
strings – vector of strings

inline **PauliOp**(std::vector<*PauliString*> strings)

Construct a *PauliOp* from a vector of PauliStrings and default corresponding coeffs to ones.

Parameters

strings – vector of *PauliString* objects

inline **PauliOp**(std::vector<*H*> coefficients, std::vector<*PauliString*> strings)

Construct a *PauliOp* from a vector of PauliStrings and corresponding coefficients.

Parameters

- **coefficients** – vector of coefficients
- **strings** – vector of *PauliString* objects

inline size_t **dim**() const

Return the dimension (2^n_qubits) of the PauliStrings used to compose *PauliOp*.

Returns

size_t

inline size_t **n_qubits**() const

Return the number of qubits in *PauliOp*.

Returns

size_t

inline size_t **n_pauli_strings**() const

Return the number of PauliStrings in *PauliOp*.

Returns

size_t

inline void **scale**(std::complex<*T*> factor)

Scale each individual term by a factor.

Parameters

factors – a factor to scale each term with

inline void **scale**(mdspan<std::complex<*T*>, std::dextents<size_t, 1>> factors)

Scale each individual term by a factor.

Parameters

factors – n_pauli_strings length array of factors to scale each term

inline *PauliOp*<*T*, *H*> **operator-**() const

Return the copy of *PauliOp* with the coefficients negated.

Returns

PauliOp<T, H>

inline void **extend**(*PauliString* pauli_str, std::complex<*T*> coeff, bool dedupe = true)

Add a *PauliString* term with appropriate coefficient to the summation inside *PauliOp*.

Parameters

- **pauli_str** – *PauliString* to add to the summation
- **coeff** – coefficient to apply to the the *PauliString*
- **dedupe** – whether to deduplicate provided *PauliString*

inline void **extend**(*PauliOp*<*T*, *H*> const &other_op)

Add another *PauliOp* to the current one by extending the internal summation with new terms.

> **Note**
>
> : for now it's very sloppy implementation just to have this functionality

Parameters

other_op – *PauliOp* to add to the current one

inline void **apply**(mdspan<std::complex<*T*>, std::dextents<size_t, 1>> state_out, mdspan<std::complex<*T*>, std::dextents<size_t, 1>> const state) const

Apply the *PauliOp* to a state.

This performs following matrix-matrix multiplication $\left(\sum_i h_i \hat{\mathcal{P}}_i\right)\hat{\psi}$

Parameters

- **state_out** – The output state after applying the *PauliOp*
- **states** – THe original state to apply the *PauliOp* to

template<execution_policy **ExecutionPolicy**>
inline void **apply**(*ExecutionPolicy* &&policy, mdspan<std::complex<*T*>, std::dextents<size_t, 1>> state_out, mdspan<std::complex<*T*>, std::dextents<size_t, 1>> const state) const

This performs following matrix-matrix multiplication $\left(\sum_i h_i \hat{\mathcal{P}}_i\right)\hat{\psi}$

Parameters

- **state_out** – The output state after applying the *PauliOp*
- **states** – THe original state to apply the *PauliOp* to
- **state_out** –
- **state** –

Template Parameters

ExecutionPolicy –

inline void **apply**(mdspan<std::complex<*T*>, std::dextents<size_t, 2>> new_states, mdspan<std::complex<*T*>, std::dextents<size_t, 2>> const states) const

Apply the *PauliOp* to a batch of states. Here all the states (new and old) are transposed so their shape is (n_dims x n_states). All the new_stats are overwritten, no need to initialize.

This performs following matrix-matrix multiplication $\left(\sum_i h_i \hat{\mathcal{P}}_i\right)\hat{\Psi}$ where matrix $\hat{\Psi}$ has ψ_t as columns

Parameters

- **new_states** – The output states after applying the *PauliOp* (n_dim x n_states)
- **states** – THe original states to apply the *PauliOp* to (n_dim x n_states)

template<execution_policy **ExecutionPolicy**>
inline void **apply**(*ExecutionPolicy*&&, mdspan<std::complex<*T*>, std::dextents<size_t, 2>> new_states, mdspan<std::complex<*T*>, std::dextents<size_t, 2>> const states) const

This performs following matrix-matrix multiplication $\left(\sum_i h_i \hat{\mathcal{P}}_i\right)\hat{\Psi}$ where matrix $\hat{\Psi}$ has ψ_t as columns

Parameters

- **new_states** – The output states after applying the *PauliOp* (n_dim x n_states)
- **states** – THe original states to apply the *PauliOp* to (n_dim x n_states)
- **new_states** –
- **states** –

Template Parameters
ExecutionPolicy –

inline void **expectation_value**(std::mdspan<std::complex<*T*>, std::dextents<size_t, 1>> expectation_vals_out, mdspan<std::complex<*T*>, std::dextents<size_t, 2>> states) const

Calculate the expectation value of the *PauliOp* on a batch of states.

It computes following inner product $\psi_t(\sum_i h_{ik}\hat{\mathcal{P}}_i)\psi_t$ for each state ψ_t from provided batch.

Parameters

- **expectation_vals_out** – expectation values for each state in the batch
- **states** – The states we want to use in our expectation value calculation (n_dim x n_states)

template<execution_policy **ExecutionPolicy**>
inline void **expectation_value**(*ExecutionPolicy*&&, std::mdspan<std::complex<*T*>, std::dextents<size_t, 1>> expectation_vals_out, mdspan<std::complex<*T*>, std::dextents<size_t, 2>> states) const

Template Parameters
ExecutionPolicy –

Parameters

- **expectation_vals_out** –
- **states** –

inline void **to_tensor**(std::mdspan<std::complex<*T*>, std::dextents<size_t, 2>> output) const

Get dense representation of *PauliOp* as a 2D-array.

Parameters
output – The output tensor to fill with the dense representation

Public Static Functions

static inline void **validate_pauli_strings**(std::vector<*PauliString*> const &pauli_strings)

Check that the dims of pauli strings are all the same.

Parameters
pauli_strings –

Friends

inline friend *PauliOp*<*T*, *H*> **operator***(*PauliOp*<*T*, *H*> const &pauli_op_left, *PauliString* const &pauli_str_right)

Matrix multiplication of *PauliOp* with a *PauliString* on the right.

Parameters

- **pauli_op_left** – left hand side *PauliOp*
- **pauli_str_right** – right hand side *PauliString*

Returns

PauliOp<T, H> new *PauliOp* instance containing the result of the multiplication

inline friend *PauliOp*<*T*, *H*> **operator***(*PauliString* const &pauli_str_left, *PauliOp*<*T*, *H*> const &pauli_op_right)

Matrix multiplication of *PauliOp* with a *PauliString* on the left.

Parameters

- **pauli_str_left** – left hand side *PauliString*
- **pauli_op_right** – right hand side *PauliOp*

Returns

PauliOp<T, H> new *PauliOp* instance containing the result of the multiplication

inline friend *PauliOp*<*T*, *H*> **operator***(*PauliOp*<*T*, *H*> const &lhs, *PauliOp*<*T*, *H*> const &rhs)

Matrix multiplication of two *Pauli* operators.

Parameters

- **lhs** – left hand side *PauliOp*
- **rhs** – right hand side *PauliOp*

Returns

PauliOp<T, H> new *PauliOp* instance containing the result of the multiplication

7.4 SummedPauliOp

template<std::floating_point **T**>

struct **SummedPauliOp**

A class representing a sum of *Pauli* operators $A = \sum_k A_k = \sum_{ik} h_{ik} \hat{\mathcal{P}}_i$. Where $\hat{\mathcal{P}}_i$ are *Pauli* strings and h_{ik} are complex-valued coefficients.

Public Functions

SummedPauliOp() noexcept = default

Default constructor.

inline **SummedPauliOp**(std::vector<*PauliString*> const &pauli_strings, std::vector<std::complex<*T*>> const &coeffs_raw)

Construct a new Summed *Pauli* Op object from a vector of PauliStrings and a blob of coefficients.

Parameters

- **pauli_strings** – The PauliStrings that define the set of PauliStrings used by all operators (n_pauli_strings)
- **coeffs_raw** – A vector of coefficients that define the weights of each *PauliString* in each operator. The coefficients here are a flattened version of h_{ik} in $A_k = \sum_i h_{ik} \hat{\mathcal{P}}_i$ (n_pauli_strings * n_operators,)

inline **SummedPauliOp**(std::vector<*PauliString*> const &pauli_strings, Tensor<2> const coeffs)

Construct a new Summed *Pauli* Op object from a vector of PauliStrings and an std::mdspan of coefficients.

Parameters

- **pauli_strings** – The PauliStrings that define the set of PauliStrings used by all operators (n_pauli_strings,)

- **coeffs** – A 2D std::mdspan of coefficients that define the weights of each *PauliString* in each operator. The coefficients here are h_{ik} in $A_k = \sum_i h_{ik} \hat{\mathcal{P}}_i$. (n_pauli_strings, n_operators)

inline **SummedPauliOp**(std::vector<std::string> const &pauli_strings, Tensor<2> const coeffs)

Construct a new Summed *Pauli* Op object from a vector of strings and a std::mdspan of coefficients.

Parameters

- **pauli_strings** – A vector of strings that define the set of PauliStrings used by all operators (n_pauli_strings)
- **coeffs** – A 2D std::mdspan of coefficients that define the weights of each *PauliString* in each operator. The coefficients here are h_{ik} in $A_k = \sum_i h_{ik} \hat{\mathcal{P}}_i$. (n_pauli_strings, n_operators)

inline size_t **dim**() const noexcept

Return the number of dimensions of the *SummedPauliOp*.

Returns
size_t

inline size_t **n_qubits**() const noexcept

Return the number of qubits in the *SummedPauliOp*.

Returns
size_t

inline size_t **n_operators**() const noexcept

Return the number of *Pauli* operators in the *SummedPauliOp*.

Returns
s

inline size_t **n_pauli_strings**() const noexcept

Return the number of PauliStrings in the *SummedPauliOp*.

Returns
size_t

inline fast_pauli::*SummedPauliOp*<*T*> **square**() const

Square the *SummedPauliOp*, mathematically $A_k \rightarrow A_k^2 = \sum_{a,b} T_{ab} A_a A_b$.

We use the sprarse structure of A_k operators and *PauliString* products to calculate the new coefficients.

Returns
SummedPauliOp<T>

inline void **apply**(Tensor<2> new_states, Tensor<2> states) const

Apply the *SummedPauliOp* to a set of states, mathematically $\big(\sum_k \sum_i h_{ik} \hat{\mathcal{P}}_i\big) \psi_t$.

Parameters

- **new_states** – The output states after applying the *SummedPauliOp* (n_dim, n_states)
- **states** – The input states to apply the *SummedPauliOp* to (n_dim, n_states)

template<execution_policy **ExecutionPolicy**>
inline void **apply**(*ExecutionPolicy*&&, Tensor<2> new_states, Tensor<2> states) const

Apply the *SummedPauliOp* to a set of states, mathematically $\big(\sum_k \sum_i h_{ik} \hat{\mathcal{P}}_i\big) \psi_t$.

Parameters

- **new_states** – The output states after applying the *SummedPauliOp* (n_dim, n_states)
- **states** – The input states to apply the *SummedPauliOp* to (n_dim, n_states)

Template Parameters
ExecutionPolicy –

template<std::floating_point **data_dtype**>
inline void **apply_weighted**(Tensor<2> new_states, Tensor<2> states, std::mdspan<*data_dtype*, std::dextents<size_t, 2>> data) const

Apply the *SummedPauliOp* to a set of weighted states.

Calculates $\big(\sum_k x_{tk} \sum_i h_{ik} \hat{\mathcal{P}}_i \big) \psi_t$

Template Parameters
data_dtype – The floating point type of the weights x_{kt} (n_operators, n_states)

Parameters

- **new_states** – The output states after applying the *SummedPauliOp* (n_dim, n_states)
- **states** – The input states to apply the *SummedPauliOp* to (n_dim, n_states)
- **data** – A 2D std::mdspan of the data x_{tk} that weights the operators in the expression above (n_operators, n_states)

template<execution_policy **ExecutionPolicy**, std::floating_point **data_dtype**>
inline void **apply_weighted**(*ExecutionPolicy*&&, Tensor<2> new_states, Tensor<2> states, std::mdspan<*data_dtype*, std::dextents<size_t, 2>> data) const

Template Parameters

- **ExecutionPolicy** – Execution policy for parallelization
- **data_dtype** – The floating point type of the weights x_{kt} (n_operators, n_states)

inline void **expectation_value**(Tensor<2> expectation_vals_out, Tensor<2> states) const

Calculate the expectation value of the *SummedPauliOp* on a batch of states. This function returns the expectation values of each operator for each input states, so the output tensor will have shape (n_operators x n_states).

$\psi_t \big(\sum_k \sum_i h_{ik} \hat{\mathcal{P}}_i \big) \psi_t$

Parameters

- **expectation_vals_out** – Output tensor for the expectation values (n_operators x n_states)
- **states** – The states used to calculate the expectation values (n_dim, n_states)

template<execution_policy **ExecutionPolicy**>
inline void **expectation_value**(*ExecutionPolicy*&&, Tensor<2> expectation_vals_out, Tensor<2> states) const

Calculate the expectation value of the *SummedPauliOp* on a batch of states. This function returns the expectation values of each operator for each input states, so the output tensor will have shape (n_operators x n_states).

$\psi_t \big(\sum_k \sum_i h_{ik} \hat{\mathcal{P}}_i \big) \psi_t$

Parameters

- **expectation_vals_out** – Output tensor for the expectation values (n_operators x n_states)

- **states** – The states used to calculate the expectation values (n_dim, n_states)

Template Parameters
ExecutionPolicy – Execution policy for parallelization

inline std::vector<*PauliOp*<*T*>> **split**() const

Return the components of the *SummedPauliOp* as a vector of PauliOps.

Returns
std::vector<PauliOp<T>>

inline void **to_tensor**(Tensor<3> A_k_out) const

Get the dense representation of the *SummedPauliOp* as a 3D tensor.

Parameters
A_k_out – The output tensor to fill with the dense representation

7.5 Helpers

std::vector<std::string> fast_pauli::**get_nontrivial_paulis**(size_t const weight)

Get the nontrivial sets of pauli matrices given a weight.

Parameters
weight – The *Pauli* weight to get the nontrivial paulis for.

Returns
std::vector<std::string>

std::vector<*PauliString*> fast_pauli::**calculate_pauli_strings**(size_t const n_qubits, size_t const weight)

Calculate all possible PauliStrings for a given number of qubits and weight and return them in lexicographical order.

Parameters

- **n_qubits** – The number of qubits.
- **weight** – The *Pauli* weight.

Returns
std::vector<PauliString>

std::vector<*PauliString*> fast_pauli::**calculate_pauli_strings_max_weight**(size_t n_qubits, size_t weight)

Calculate all possible PauliStrings for a given number of qubits and all weights less than or equal to a given weight.

Parameters

- **n_qubits** – The number of qubits.
- **weight** – The *Pauli* weight.

Returns
std::vector<PauliString>

template<std::floating_point **T**>
std::tuple<std::vector<size_t>, std::vector<std::complex<*T*>>> fast_pauli::**get_sparse_repr**(std::vector<*Pauli*> const &paulis)

Get the sparse representation of the pauli string matrix.

PauliStrings are always sparse and have only single non-zero element per row. It's N non-zero elements for NxN matrix where N is 2^n_qubits. Therefore k and m will always have N elements.

See Algorithm 1 in https://arxiv.org/pdf/2301.00560.pdf for details about the algorithm.

Template Parameters

T – The floating point type (i.e. std::complex<T> for the values of the *PauliString* matrix)

Parameters

- **k** – The column index of the matrix
- **m** – The values of the matrix

CHAPTER

EIGHT

INTRODUCTION

Welcome to `fast-pauli` from Qognitive, an open-source Python / C++ library for optimized operations on Pauli matrices and Pauli strings, inspired by PauliComposer paper. `fast-pauli` aims to provide a fast and efficient alternative to existing libraries for working with Pauli matrices and strings, with a focus on performance and usability. For example, `fast-pauli` provides optimized functions to apply Pauli strings and operators to a batch of states rather than just a single state vector. See our *Getting Started* guide for an introduction to some of the core functionality in `fast-pauli` and our *Benchmarks* for more details about how `fast-pauli` can speed up certain functions compared to Qiskit.

CHAPTER

NINE

INSTALLATION

In order to get started, we'll need to install the package and its dependencies.

9.1 Requirements

- CMake >= 3.25
- Ninja >= 1.11
- C++ compiler with OpenMP and C++20 support (LLVM recommended)
- Python >= 3.10
- scikit-build-core (ONLY for building from source with custom configuration)

In the following subsections, we describe several options for installing `fast_pauli`.

9.2 Install the Latest Release

```
pip install fast_pauli
```

9.3 Build from Source (Linux)

```
git clone git@github.com:qognitive/fast-pauli.git
cd fast_pauli
python -m pip install -e ".[dev]"
```

9.4 Build from Source (MacOS)

```
git clone git@github.com:qognitive/fast-pauli.git
cd fast_pauli
python -m pip install --upgrade pip
python -m pip install scikit-build-core
brew install llvm
pip install -e . -C cmake.args="-DCMAKE_CXX_COMPILER=$(brew --prefix llvm)/bin/clang++;-
↪DCMAKE_CXX_FLAGS='-stdlib=libc++ -fexperimental-library'"
```

9.5 Build from Source (Custom Config)

```
git clone git@github.com:qognitive/fast-pauli.git
cd fast-pauli
python -m pip install --upgrade pip
python -m pip install scikit-build-core
python -m pip install --no-build-isolation -ve ".[dev]" -C cmake.args="-DCMAKE_CXX_
↪COMPILER=<compiler> + <other cmake flags>"
```

9.6 Verify / Test Build

```
pytest -v tests/fast_pauli # + other pytest flags
```

INDEX

Symbols

A

C

D

E

F

G

N

P

S

T

W