Amazon Braket PennyLane Plugin Documentation

Release 1.22.1.dev0

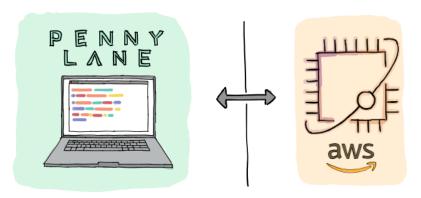
Amazon.com Inc.

CONTENTS

1	Devi	ces	3
2	Tuto		5
	2.1	Installation	
	2.2	Support	7
	2.3	The remote Braket device	7
	2.4	The local Braket device	9
	2.5	The local AHS device	10
	2.6	The remote AHS device	
	2.7	pennylane-braket	15
Рy	thon I	Module Index	181
In	dex		183

Release

1.22.1.dev0



The Amazon Braket Python SDK is an open source library that provides a framework to interact with quantum computing hardware devices and simulators through Amazon Braket.

PennyLane is a machine learning library for optimization and automatic differentiation of hybrid quantum-classical computations.

Once the Pennylane-Braket plugin is installed, the provided Braket devices can be accessed straight away in PennyLane, without the need to import any additional packages.

CONTENTS 1

2 CONTENTS

CHAPTER

ONE

DEVICES

This plugin provides four Braket devices for use with PennyLane - two supporting gate-based computations, and two supporting analog Hamiltonian simulation (AHS):

While the local device helps with small-scale simulations and rapid prototyping, the remote device allows you to run larger simulations or access quantum hardware via the Amazon Braket service.

4 Chapter 1. Devices

CHAPTER

TWO

TUTORIALS

To see the PennyLane-Braket plugin in action, you can use any of the qubit-based demos from the PennyLane documentation, for example the tutorial on qubit rotation, and simply replace 'default.qubit' with the 'braket.local. qubit' or the 'braket.aws.qubit' device:

```
dev = qml.device('braket.XXX.qubit', [...])
```

Tutorials that showcase the Braket devices can be found on the PennyLane website and the Amazon Braket examples GitHub repository.

2.1 Installation

Before you begin working with the Amazon Braket PennyLane Plugin, make sure that you installed or configured the following prerequisites:

- Download and install Python 3.9 or greater. If you are using Windows, choose the option *Add Python to environment variables* before you begin the installation.
- Make sure that your AWS account is onboarded to Amazon Braket, as per the instructions here.
- Download and install PennyLane:

```
pip install pennylane
```

You can then install the latest release of the PennyLane-Braket plugin as follows:

```
pip install amazon-braket-pennylane-plugin
```

You can also install the development version from source by cloning this repository and running a pip install command in the root directory of the repository:

```
git clone https://github.com/amazon-braket/amazon-braket-pennylane-plugin-python.git {\tt cd} amazon-braket-pennylane-plugin-python pip install .
```

You can check your currently installed version of amazon-braket-pennylane-plugin with pip show:

```
pip show amazon-braket-pennylane-plugin
```

or alternatively from within Python:

```
from braket import pennylane_plugin
pennylane_plugin.__version__
```

2.1.1 Tests

Make sure to install test dependencies first:

```
pip install -e "amazon-braket-pennylane-plugin-python[test]"
```

Unit tests

Run the unit tests using:

```
tox -e unit-tests
```

To run an individual test:

```
tox -e unit-tests -- -k 'your_test'
```

To run linters and unit tests:

tox

Integration tests

To run the integration tests, set the AWS_PROFILE as explained in the amazon-braket-sdk-python README:

```
export AWS_PROFILE=Your_Profile_Name
```

Running the integration tests creates an S3 bucket in the same account as the AWS_PROFILE with the following naming convention amazon-braket-pennylane-plugin-integ-tests-{account_id}.

Run the integration tests with:

```
tox -e integ-tests
```

To run an individual integration test:

```
tox -e integ-tests -- -k 'your_test'
```

2.1.2 Documentation

To build the HTML documentation, run:

```
tox -e docs
```

The documentation can then be found in the doc/build/documentation/html/ directory.

2.2 Support

- Source Code: https://github.com/amazon-braket/amazon-braket-pennylane-plugin-python
- Issue Tracker: https://github.com/amazon-braket/amazon-braket-pennylane-plugin-python/issues
- General Questions: https://quantumcomputing.stackexchange.com/questions/ask (add the tag amazon-braket)
- PennyLane Forum: https://discuss.pennylane.ai

If you are having issues, please let us know by posting the issue on our Github issue tracker, or by asking a question in the forum.

2.3 The remote Braket device

The remote qubit device of the PennyLane-Braket plugin runs gate-based quantum computations on Amazon Braket's remote service. The remote service provides access to hardware providers and a high-performance simulator backend.

A list of available quantum devices and their features can be found in the Amazon Braket Developer Guide.

2.3.1 Usage

After the Braket SDK and the plugin are installed, and once you sign up for Amazon Braket, you have access to the remote Braket device in PennyLane.

Instantiate an AWS device that communicates with the Braket service like this:

In this example, the string arn:aws:braket:::device/quantum-simulator/amazon/sv1 is the ARN that identifies the SV1 device. Other supported devices and their ARNs can be found in the Amazon Braket Developer Guide. Note that the plugin works with digital (qubit) gate-based devices only.

This device can then be used just like other devices for the definition and evaluation of QNodes within PennyLane.

For example:

```
@qml.qnode(remote_device)
def circuit(x, y, z):
    qml.RZ(z, wires=[0])
    qml.RY(y, wires=[0])
    qml.RX(x, wires=[0])
    qml.CNOT(wires=[0, 1])
    return qml.expval(qml.PauliZ(0)), var(qml.PauliZ(1))
```

When executed, the circuit performs the computation on the Amazon Braket service.

```
>>> circuit(0.2, 0.1, 0.3)
array([0.97517033, 0.04904283])
```

2.2. Support 7

2.3.2 Enabling the parallel execution of multiple circuits

Where supported by the backend of the Amazon Braket service, the remote device can be used to execute multiple quantum circuits in parallel. To unlock this feature, instantiate the device using the parallel=True argument:

```
>>> remote_device = qml.device('braket.aws.qubit', [... ,] parallel=True)
```

The details of the parallelization scheme depend on the PennyLane version you use, as well as your AWS account specifications.

For example, PennyLane 0.13.0 and higher supports the parallel execution of circuits created during the computation of gradients. Just by instantiating the remote device with the parallel=True option, this feature is automatically used and can lead to significant speedups of your optimization pipeline.

The maximum number of circuits that can be executed in parallel is specified by the max_parallel argument.

Make sure that this number is not larger than the maximum number of concurrent tasks allowed for your account on the backend you choose. See the Braket developer guide for more details.

The Braket remote device has the capability to retry failed circuit executions, up to 3 times per circuit by default. You can set a timeout by using the poll_timeout_seconds argument; the device will retry circuits that do not complete within the timeout. A timeout of 30 to 60 seconds is recommended for circuits with fewer than 25 qubits.

2.3.3 Device options

The default value of the shots argument is Shots.DEFAULT, resulting in the default number of shots specified by the remote device being used. For example, a simulator device may default to analytic mode while a QPU must pick a finite number of shots.

Setting shots=0 or shots=None will cause the device to run in analytic mode. If the device ARN points to a QPU, analytic mode is not available and an error will be raised.

2.3.4 Supported operations

The operations supported by this device vary based on the operations supported by the underlying Braket device. To check the device's supported operations, run

```
dev.operations
```

In addition to those provided by PennyLane, the PennyLane-Braket plugin provides the following framework-specific operations, which can be imported from braket.pennylane_plugin.ops:

```
braket.pennylane_plugin.CPhaseShift00(phi, wires)

braket.pennylane_plugin.CPhaseShift01(phi, wires)

braket.pennylane_plugin.CPhaseShift10(phi, wires)

braket.pennylane_plugin.CPhaseShift10(phi, wires)

braket.pennylane_plugin.PSWAP(phi, wires)

braket.pennylane_plugin.GPi(phi, wires)

braket.pennylane_plugin.GPi(phi, wires)

braket.pennylane_plugin.GPi2(phi, wires)

braket.pennylane_plugin.MS(phi_0, phi_1, wires)

braket.pennylane_plugin.MS(phi_0, phi_1, wires)

IonQ native GPi2 gate.

braket.pennylane_plugin.MS(phi_0, phi_1, wires)

IonQ native Mølmer-Sørenson gate.
```

2.3.5 Gradient computation on Braket with a QAOA Hamiltonian

Currently, PennyLane will compute grouping indices for QAOA Hamiltonians and use them to split the Hamiltonian into multiple expectation values. If you wish to use *SV1's adjoint differentiation capability<https://docs.aws.amazon.com/braket/latest/developerguide/hybrid.html>* when running QAOA from PennyLane, you will need reconstruct the cost Hamiltonian to remove the grouping indices from the cost Hamiltonian, like so:

```
cost_h, mixer_h = qml.qaoa.max_clique(g, constrained=False)
cost_h = qml.Hamiltonian(cost_h.coeffs, cost_h.ops)
```

2.4 The local Braket device

The local qubit device of the PennyLane-Braket plugin runs gate-based quantum computations on the local Braket SDK. This could be either utilizing the processors of your own PC, or those of a Braket notebook instance hosted on AWS.

This device is useful for small-scale simulations in which the time of sending a job to a remote service would add an unnecessary overhead. It can also be used for rapid prototyping before running a computation on a paid-for remote service.

2.4.1 Usage

After the Braket SDK and the plugin are installed you immediately have access to the local Braket device in PennyLane.

To instantiate the local Braket simulator, simply use:

You can define and evaluate quantum nodes with these devices just as you would with any other PennyLane device.

For example:

```
@qml.qnode(device_local)
def circuit(x, y, z):
    qml.RZ(z, wires=[0])
    qml.RY(y, wires=[0])
    qml.RX(x, wires=[0])
    qml.CNOT(wires=[0, 1])
    return qml.expval(qml.PauliZ(0)), var(qml.PauliZ(1))
```

When executed, the circuit will perform the computation on the local machine.

```
>>> circuit(0.2, 0.1, 0.3)
array([0.97517033, 0.04904283])
```

2.4.2 Device options

You can set shots to None (default) to get exact results instead of results calculated from samples.

2.4.3 Supported operations

The operations supported by this device vary based on the operations supported by the underlying Braket device. To check the device's supported operations, run

```
dev.operations
```

In addition to those provided by PennyLane, the PennyLane-Braket plugin provides the following framework-specific operations, which can be imported from braket.pennylane_plugin.ops:

<pre>braket.pennylane_plugin.CPhaseShift00(phi, wires)</pre>	Controlled phase shift gate phasing the $ 00\rangle$ state.
<pre>braket.pennylane_plugin.CPhaseShift01(phi, wires)</pre>	Controlled phase shift gate phasing the $ 01\rangle$ state.
<pre>braket.pennylane_plugin.CPhaseShift10(phi, wires)</pre>	Controlled phase shift gate phasing the $ 10\rangle$ state.
<pre>braket.pennylane_plugin.PSWAP(phi, wires)</pre>	Phase-SWAP gate.
<pre>braket.pennylane_plugin.GPi(phi, wires)</pre>	IonQ native GPi gate.
<pre>braket.pennylane_plugin.GPi2(phi, wires)</pre>	IonQ native GPi2 gate.
<pre>braket.pennylane_plugin.MS(phi_0, phi_1, wires)</pre>	IonQ native Mølmer-Sørenson gate.

2.5 The local AHS device

The local analog Hamiltonian simulation (AHS) device of the PennyLane-Braket plugin runs simulation on the local Braket SDK. This could be either utilizing the processors of your own PC, or those of a Braket notebook instance hosted on AWS.

This device is useful for small-scale simulations in which the time of sending a job to a remote service would add an unnecessary overhead. It can also be used for rapid prototyping before running a computation on a paid-for remote service.

10 Chapter 2. Tutorials

2.5.1 Usage

After the Braket SDK and the plugin are installed you immediately have access to the local Braket AHS simulator in PennyLane.

The local AHS device is not gate-based. Instead, it is compatible with the ParametrizedEvolution operator from pulse programming in PennyLane.

Note that pulse programming in PennyLane requires the module jax, which can be installed following the instructions [here](https://github.com/google/jax#installation).

To instantiate the local Braket simulator, simply use:

```
import pennylane as qml
device_local = qml.device("braket.local.ahs", wires=2)
```

This device can be used with a QNode within PennyLane. It accepts circuits with a single ParametrizedEvolution operator based on a ParametrizedHamiltonian compatible with the simulated hardware. More information about creating PennyLane operators for AHS can be found in the PennyLane docs.

Note: It is important to keep track of units when specifying electromagnetic pulses for hardware control. The frequency and amplitude provided in PennyLane for Rydberg atom systems are expected to be in units of MHz, time in microseconds, phase in radians, and distance in micrometers. All of these will be converted to SI units internally as needed for upload to the hardware, and frequency will be converted to angular frequency (multiplied by 2π).

When reading hardware specifications from the Braket backend, bear in mind that all units are SI and frequencies are in rad/s. This conversion is done when creating a pulse program for upload, and units in the PennyLane functions should follow the conventions specified in the PennyLane docs to ensure correct unit conversion. See rydberg_interaction and rydberg_drive in Pennylane for specification of expected input units, and examples for creating hardware-compatible ParametrizedEvolution operators in PennyLane.

Creating a register

The atom register defines where the atoms will be located, which determines the strength of the interaction between the atoms. Here we define coordinates for the atoms to be placed at (in micrometers), and create a constant interaction term for the Hamiltonian:

```
# number of coordinate pairs must match number of device wires
coordinates = [[0, 0], [0, 5]]

H_interaction = qml.pulse.rydberg_interaction(coordinates)
```

Creating a drive

We can create a drive with a global component and (positive) local detunings. If the local detunings are time-dependent, they must all have the same time-dependent envelope, but can have different, positive scaling factors.

```
from jax import numpy as jnp

# gaussian amplitude function (qml.pulse.rect enforces 0 at start and end for hardware)
def amp_fn(p, t):
    f = p[0] * jnp.exp(-(t - p[1])**2 / (2 * p[2]**2))
(continues on next page)
```

(continued from previous page)

```
return qml.pulse.rect(f, windows=[0.1, 1.7])(p, t)
# defining a linear detuning
def det_fn_global(p, t):
    return p * t
def det_fn_local(p, t):
    return p * t**2
# creating a global drive on all wires
H_global = qml.pulse.rydberg_drive(amplitude=amp_fn, phase=0, detuning=det_fn_global,__
\rightarrowwires=[0, 1])
# creating local drives
# note only local detuning is currently supported, so amplitude and phase are set to 0
H_local0 = qml.pulse.rydberg_drive(amplitude=0, phase=0, detuning = det_fn_local,_
\rightarrowwires=[0])
H_local1 = qml.pulse.rydberg_drive(amplitude=0, phase=0, detuning = det_fn_local,_
\rightarrowwires=[1])
# full hamiltonian
H = H_interaction + H_global + H_local0 + H_local1
```

Executing an AHS program

```
@qml.qnode(device_local)
def circuit(params):
    qml.evolve(H)(params, t=1.5)
    return qml.sample()

# amp_fn expects p to contain 3 parameters
amp_params = [2.5, 1, 0.3]
# global_det_fn expects p to be a single parameter
det_global_params = 0.2
# each of the local drives take a single parameter for p
# the detunings have the same shape, but vary by scaling factor p
local_params1 = 0.5
local_params2 = 1
```

When executed, the circuit will perform the computation on the local machine.

12 Chapter 2. Tutorials

2.6 The remote AHS device

The remote AHS device of the PennyLane-Braket plugin runs analog Hamiltonian simulation (AHS) on Amazon Braket's remote service. AHS is a quantum computing paradigm different from gate-based computing. AHS uses a well-controlled quantum system and tunes its parameters to mimic the dynamics of another quantum system, the one we aim to study.

The remote service provides access to running AHS on hardware. As AHS devices are not gate-based, they are not compatible with the standard PennyLane operators. Instead, they are compatible with pulse programming in PennyLane.

Note that pulse programming in PennyLane requires the module jax, which can be installed following the instructions [here](https://github.com/google/jax#installation).

More information about AHS and the capabilities of the hardware can be found in the Amazon Braket Developer Guide.

2.6.1 Usage

After the Braket SDK and the plugin are installed, and once you sign up for Amazon Braket, you have access to the remote AHS device in PennyLane.

Instantiate an AWS device that communicates with the hardware like this:

```
>>> import pennylane as qml
>>> device_arn = "arn:aws:braket:us-east-1::device/qpu/quera/Aquila"
>>> remote_device = qml.device("braket.aws.ahs", device_arn=device_arn, wires=3)
```

This device can be used with a QNode within PennyLane. It accepts circuits with a single ParametrizedEvolution operator based on a hardware-compatible ParametrizedHamiltonian. More information about creating PennyLane operators for AHS can be found in the PennyLane docs.

Note: It is important to keep track of units when specifying electromagnetic pulses for hardware control. The frequency and amplitude provided in PennyLane for Rydberg atom systems are expected to be in units of MHz, time in microseconds, phase in radians, and distance in micrometers. All of these will be converted to SI units internally as needed for upload to the hardware, and frequency will be converted to angular frequency (multiplied by 2π).

When reading hardware specifications from the Braket backend, bear in mind that all units are SI and frequencies are in rad/s. This conversion is done when creating a pulse program for upload, and units in the PennyLane functions should follow the conventions specified in the PennyLane docs to ensure correct unit conversion. See rydberg_interaction and rydberg_drive in Pennylane for specification of expected input units, and examples for creating hardware-compatible ParametrizedEvolution operators in PennyLane.

Creating a register

The atom register defines where the atoms will be located, and determines the strength of the interaction between the atoms. Here we define coordinates for the atoms to be placed at (in micrometers), and create a constant interaction term for the Hamiltonian:

```
# number of coordinate pairs must match number of device wires
coordinates = [[0, 0], [0, 5], [5, 0]]

H_interaction = qml.pulse.rydberg_interaction(coordinates)
```

Creating a global drive

Hardware currently only supports a single global drive pulse applied to all atoms in the register.

Here we define a global drive with time dependent amplitude and detuning, with phase set to 0.

```
from jax import numpy as jnp

# gaussian amplitude function (qml.pulse.rect enforces 0 at start and end for hardware)
def amp_fn(p, t):
    f = p[0] * jnp.exp(-(t - p[1])**2 / (2 * p[2]**2))
    return qml.pulse.rect(f, windows=[0.1, 1.7])(p, t)

# defining a linear detuning
def det_fn(p, t):
    return p * t

# creating a global drive on all wires
H_global = qml.pulse.rydberg_drive(amplitude=amp_fn, phase=0, detuning=det_fn, wires=[0, ...
    -1, 2])
```

Creating and executing the circuit

Once we have the terms describing the atomic interactions and the electromagnetic drive on the atoms, we can create and execute a circuit to run the pulse program on the hardware:

```
Qqml.qnode(remote_device)
def circuit(amp_params, det_params):
    qml.evolve(H_interaction + H_global)([amp_params, det_params], t=1.75)
    return qml.sample()
```

When executed, the circuit performs the computation on the hardware.

```
>>> amp_params = [2.5, 1, 0.3] # amp_fn expects p to contain 3 parameters
>>> det_params = 0.2 # det_fn expects p to be a single parameter
>>> circuit(amp_params, det_params)
array([0.97517033, 0.04904283])
```

2.6.2 Device options

14

The default value of the shots argument is Shots.DEFAULT, resulting in the default number of shots specified by the remote device being used. For example, a simulator device may default to analytic mode while a QPU must pick a finite number of shots.

This device is not compatible with analytic mode, so an error will be raised if shots=0 or shots=None.

2.6.3 Supported operations

The only operation supported for analog Hamiltonian simulation is a ParametrizedEvolution describing a hardware-compatible electromagnetic pulse.

2.7 pennylane-braket

This section contains the API documentation for the PennyLane-Braket plugin.

Warning: Unless you are a PennyLane plugin developer, you likely do not need to use these classes and functions directly.

See the *overview* page for more details using the available Braket devices with PennyLane.

2.7.1 Classes

AAMS(phi_0, phi_1, theta, wires)	IonQ native Arbitrary-Angle Mølmer-Sørenson gate.
<pre>BraketAwsAhsDevice(wires, device_arn[,])</pre>	Amazon Braket AHS device for hardware in PennyLane.
<pre>BraketAwsQubitDevice(wires, device_arn[,])</pre>	Amazon Braket AwsDevice qubit device for PennyLane.
<pre>BraketLocalAhsDevice(wires, *[, shots])</pre>	Amazon Braket LocalSimulator AHS device for Penny-
	Lane.
<pre>BraketLocalQubitDevice(wires[, backend, shots])</pre>	Amazon Braket LocalSimulator qubit device for Penny-
	Lane.
CPhaseShift00(phi, wires)	Controlled phase shift gate phasing the $ 00\rangle$ state.
CPhaseShift01(phi, wires)	Controlled phase shift gate phasing the $ 01\rangle$ state.
CPhaseShift10(phi, wires)	Controlled phase shift gate phasing the $ 10\rangle$ state.
GPi(phi, wires)	IonQ native GPi gate.
GPi2(phi, wires)	IonQ native GPi2 gate.
MS(phi_0, phi_1, wires)	IonQ native Mølmer-Sørenson gate.
PSWAP(phi, wires)	Phase-SWAP gate.

AAMS

class AAMS(phi_0, phi_1, theta, wires)

Bases: Operation

IonQ native Arbitrary-Angle Mølmer-Sørenson gate.

$$\mathsf{MS}(\phi_0,\phi_1,\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & 0 & 0 & -ie^{-i(\phi_0+\phi_1)}\sin\frac{\theta}{2} \\ 0 & \cos\frac{\theta}{2} & -ie^{-i(\phi_0-\phi_1)}\sin\frac{\theta}{2} & 0 \\ 0 & -ie^{i(\phi_0-\phi_1)}\sin\frac{\theta}{2} & \cos\frac{\theta}{2} & 0 \\ -ie^{i(\phi_0+\phi_1)}\sin\frac{\theta}{2} & 0 & 0 & \cos\frac{\theta}{2} \end{bmatrix}.$$

Details:

• Number of wires: 2

• Number of parameters: 2

Parameters

- **phi_0** (*float*) the first phase angle
- **phi_1** (*float*) the second phase angle
- **theta** (*float*) the entangling angle
- wires (int) the subsystem the gate acts on
- **id** (*str or None*) String representing the operation (optional)

arithmetic_depth	Arithmetic depth of the operator.
basis	The basis of an operation, or for controlled gates, of
Dasis	
1 . 1 .	the target operation.
batch_size	Batch size of the operator if it is used with broad-
	casted parameters.
control_wires	Control wires of the operator.
<pre>grad_method</pre>	
grad_recipe	Gradient recipe for the parameter-shift method.
has_adjoint	
has_decomposition	
•	
has_diagonalizing_gates	
<u>3</u> _5	
has_generator	
has matrix	
140_114 C1 11	
hash	Integer hash that uniquely represents the operator.
hyperparameters	Dictionary of non-trainable variables that this opera-
nyperparameters	tion depends on.
id	•
	Custom string to label a specific operator instance.
is_hermitian	This property determines if an operator is hermitian.
name	String for the name of the operator.
ndim_params	Number of dimensions per trainable parameter of the
	operator.
num_params	
num_wires	Number of wires the operator acts on.
parameter_frequencies	Returns the frequencies for each operator parame-
	ter with respect to an expectation value of the form
	$\langle \psi U(\mathbf{p})^{\dagger} \hat{OU}(\mathbf{p}) \psi \rangle$.
parameters	Trainable parameters that the operator depends on.
wires	Wires that the operator acts on.
	1

arithmetic_depth

Arithmetic depth of the operator.

basis = None

The basis of an operation, or for controlled gates, of the target operation. If not None, should take a value of "X", "Y", or "Z".

For example, X and CNOT have basis = "X", whereas ControlledPhaseShift and RZ have basis = "Z".

16 Chapter 2. Tutorials

Type

str or None

batch_size

Batch size of the operator if it is used with broadcasted parameters.

The batch_size is determined based on ndim_params and the provided parameters for the operator. If (some of) the latter have an additional dimension, and this dimension has the same size for all parameters, its size is the batch size of the operator. If no parameter has an additional dimension, the batch size is None.

Returns

Size of the parameter broadcasting dimension if present, else None.

Return type

int or None

control_wires

Control wires of the operator.

For operations that are not controlled, this is an empty Wires object of length 0.

Returns

The control wires of the operation.

Return type

Wires

grad_method = 'F'

grad_recipe = None

Gradient recipe for the parameter-shift method.

This is a tuple with one nested list per operation parameter. For parameter ϕ_k , the nested list contains elements of the form $[c_i, a_i, s_i]$ where i is the index of the term, resulting in a gradient recipe of

$$\frac{\partial}{\partial \phi_k} f = \sum_i c_i f(a_i \phi_k + s_i).$$

If None, the default gradient recipe containing the two terms $[c_0, a_0, s_0] = [1/2, 1, \pi/2]$ and $[c_1, a_1, s_1] = [-1/2, 1, -\pi/2]$ is assumed for every parameter.

Type

tuple(Union(list[list[float]], None)) or None

has_adjoint = True

has_decomposition = False

has_diagonalizing_gates = False

has_generator = False

has_matrix = True

hash

Integer hash that uniquely represents the operator.

Type

int

hyperparameters

Dictionary of non-trainable variables that this operation depends on.

```
Type dict
```

id

Custom string to label a specific operator instance.

is hermitian

This property determines if an operator is hermitian.

name

String for the name of the operator.

ndim_params

Number of dimensions per trainable parameter of the operator.

By default, this property returns the numbers of dimensions of the parameters used for the operator creation. If the parameter sizes for an operator subclass are fixed, this property can be overwritten to return the fixed value.

Returns

Number of dimensions for each trainable parameter.

Return type

tuple

 $num_params = 3$

$num_wires = 2$

Number of wires the operator acts on.

parameter_frequencies

Returns the frequencies for each operator parameter with respect to an expectation value of the form $\langle \psi | U(\mathbf{p})^{\dagger} \hat{O} U(\mathbf{p}) | \psi \rangle$.

These frequencies encode the behaviour of the operator $U(\mathbf{p})$ on the value of the expectation value as the parameters are modified. For more details, please see the pennylane. fourier module.

Returns

Tuple of frequencies for each parameter. Note that only non-negative frequency values are returned.

Return type

list[tuple[int or float]]

Example

```
>>> op = qml.CRot(0.4, 0.1, 0.3, wires=[0, 1])
>>> op.parameter_frequencies
[(0.5, 1), (0.5, 1), (0.5, 1)]
```

For operators that define a generator, the parameter frequencies are directly related to the eigenvalues of the generator:

```
>>> op = qml.ControlledPhaseShift(0.1, wires=[0, 1])
>>> op.parameter_frequencies
[(1,)]
```

(continues on next page)

(continued from previous page)

```
>>> gen = qml.generator(op, format="observable")
>>> gen_eigvals = qml.eigvals(gen)
>>> qml.gradients.eigvals_to_frequencies(tuple(gen_eigvals))
(1.0,)
```

For more details on this relationship, see eigvals_to_frequencies().

parameters

Trainable parameters that the operator depends on.

wires

Wires that the operator acts on.

Returns

wires

Return type

Wires

adjoint()	Create an operation that is the adjoint of this one.
<pre>compute_decomposition(*params[, wires])</pre>	Representation of the operator as a product of other operators (static method).
<pre>compute_diagonalizing_gates(*params, wires,)</pre>	Sequence of gates that diagonalize the operator in the computational basis (static method).
<pre>compute_eigvals(*params, **hyperparams)</pre>	Eigenvalues of the operator in the computational basis (static method).
<pre>compute_matrix(phi_0, phi_1, theta)</pre>	Representation of the operator as a canonical matrix in the computational basis (static method).
<pre>compute_sparse_matrix(*params, **hyper- params)</pre>	Representation of the operator as a sparse matrix in the computational basis (static method).
<pre>decomposition()</pre>	Representation of the operator as a product of other operators.
<pre>diagonalizing_gates()</pre>	Sequence of gates that diagonalize the operator in the computational basis.
eigvals()	Eigenvalues of the operator in the computational basis.
expand()	Returns a tape that contains the decomposition of the operator.
<pre>generator()</pre>	Generator of an operator that is in single-parameter- form.
<pre>label([decimals, base_label, cache])</pre>	A customizable string representation of the operator.
<pre>map_wires(wire_map)</pre>	Returns a copy of the current operator with its wires changed according to the given wire map.
<pre>matrix([wire_order])</pre>	Representation of the operator as a matrix in the computational basis.
pow(z)	A list of new operators equal to this one raised to the given power.
queue([context])	Append the operator to the Operator queue.
simplify()	Reduce the depth of nested operators to the minimum.
<pre>single_qubit_rot_angles()</pre>	The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.
<pre>sparse_matrix([wire_order])</pre>	Representation of the operator as a sparse matrix in the computational basis.
terms()	Representation of the operator as a linear combination of other operators.
validate_subspace(subspace)	Validate the subspace for qutrit operations.

adjoint()

Create an operation that is the adjoint of this one.

Adjointed operations are the conjugated and transposed version of the original operation. Adjointed ops are equivalent to the inverted operation for unitary gates.

Returns

The adjointed operation.

static compute_decomposition(*params, wires=None, **hyperparameters)

Representation of the operator as a product of other operators (static method).

$$O = O_1 O_2 \dots O_n$$
.

Note: Operations making up the decomposition should be queued within the compute_decomposition

method.

See also:

decomposition().

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- wires (Iterable[Any], Wires) wires that the operator acts on
- **hyperparams (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

decomposition of the operator

Return type

list[Operator]

static compute_diagonalizing_gates(*params, wires, **hyperparams)

Sequence of gates that diagonalize the operator in the computational basis (static method).

Given the eigendecomposition $O=U\Sigma U^\dagger$ where Σ is a diagonal matrix containing the eigenvalues, the sequence of diagonalizing gates implements the unitary U^\dagger .

The diagonalizing gates rotate the state into the eigenbasis of the operator.

See also:

diagonalizing_gates().

Parameters

- params (list) trainable parameters of the operator, as stored in the parameters attribute
- wires (Iterable[Any], Wires) wires that the operator acts on
- \bullet hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

list of diagonalizing gates

Return type

list[.Operator]

static compute_eigvals(*params, **hyperparams)

Eigenvalues of the operator in the computational basis (static method).

If diagonalizing_gates are specified and implement a unitary U^{\dagger} , the operator can be reconstructed as

$$O = U\Sigma U^{\dagger}$$
.

where Σ is the diagonal matrix containing the eigenvalues.

Otherwise, no particular order for the eigenvalues is guaranteed.

See also:

Operator.eigvals() and qml.eigvals()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- ****hyperparams** (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

eigenvalues

Return type

tensor_like

static compute_matrix(phi_0, phi_1, theta)

Representation of the operator as a canonical matrix in the computational basis (static method).

The canonical matrix is the textbook matrix representation that does not consider wires. Implicitly, this assumes that the wires of the operator correspond to the global wire order.

See also:

Operator.matrix() and qml.matrix()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- **hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

matrix representation

Return type

tensor_like

static compute_sparse_matrix(*params, **hyperparams)

Representation of the operator as a sparse matrix in the computational basis (static method).

The canonical matrix is the textbook matrix representation that does not consider wires. Implicitly, this assumes that the wires of the operator correspond to the global wire order.

See also:

sparse_matrix()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- ****hyperparams** (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

sparse matrix representation

Return type

scipy.sparse._csr.csr_matrix

decomposition()

Representation of the operator as a product of other operators.

$$O = O_1 O_2 \dots O_n$$

A DecompositionUndefinedError is raised if no representation by decomposition is defined.

See also:

compute_decomposition().

Returns

decomposition of the operator

Return type

list[Operator]

diagonalizing_gates()

Sequence of gates that diagonalize the operator in the computational basis.

Given the eigendecomposition $O = U\Sigma U^{\dagger}$ where Σ is a diagonal matrix containing the eigenvalues, the sequence of diagonalizing gates implements the unitary U^{\dagger} .

The diagonalizing gates rotate the state into the eigenbasis of the operator.

A DiagGatesUndefinedError is raised if no representation by decomposition is defined.

See also:

compute_diagonalizing_gates().

Returns

a list of operators

Return type

list[.Operator] or None

eigvals()

Eigenvalues of the operator in the computational basis.

If diagonalizing_gates are specified and implement a unitary U^{\dagger} , the operator can be reconstructed as

$$O = U\Sigma U^{\dagger}$$
.

where Σ is the diagonal matrix containing the eigenvalues.

Otherwise, no particular order for the eigenvalues is guaranteed.

Note: When eigenvalues are not explicitly defined, they are computed automatically from the matrix representation. Currently, this computation is *not* differentiable.

A EigvalsUndefinedError is raised if the eigenvalues have not been defined and cannot be inferred from the matrix representation.

See also:

compute_eigvals()

Returns

eigenvalues

Return type

tensor_like

expand()

Returns a tape that contains the decomposition of the operator.

Returns

quantum tape

Return type

.QuantumTape

generator()

Generator of an operator that is in single-parameter-form.

For example, for operator

$$U(\phi) = e^{i\phi(0.5Y + Z \otimes X)}$$

we get the generator

```
>>> U.generator()
(0.5) [Y0]
+ (1.0) [Z0 X1]
```

The generator may also be provided in the form of a dense or sparse Hamiltonian (using Hermitian and SparseHamiltonian respectively).

The default value to return is None, indicating that the operation has no defined generator.

label(decimals=None, base_label=None, cache=None)

A customizable string representation of the operator.

Parameters

- **decimals=None** (*int*) If None, no parameters are included. Else, specifies how to round the parameters.
- $base_label=None$ (str) overwrite the non-parameter component of the label
- **cache=None** (*dict*) dictionary that carries information between label calls in the same drawing

Returns

label to use in drawings

Return type

str

Example:

```
>>> op = qml.RX(1.23456, wires=0)
>>> op.label()
"RX"
>>> op.label(decimals=2)
"RX\n(1.23)"
>>> op.label(base_label="my_label")
```

(continues on next page)

(continued from previous page)

```
"my_label"
>>> op.label(decimals=2, base_label="my_label")
"my_label\n(1.23)"
```

If the operation has a matrix-valued parameter and a cache dictionary is provided, unique matrices will be cached in the 'matrices' key list. The label will contain the index of the matrix in the 'matrices' list.

```
>>> op2 = qml.QubitUnitary(np.eye(2), wires=0)
>>> cache = {'matrices': []}
>>> op2.label(cache=cache)
'U(M0)'
>>> cache['matrices']
[tensor([[1., 0.],
[0., 1.]], requires_grad=True)]
>>> op3 = qml.QubitUnitary(np.eye(4), wires=(0,1))
>>> op3.label(cache=cache)
'U(M1)'
>>> cache['matrices']
[tensor([[1., 0.],
        [0., 1.]], requires_grad=True),
tensor([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]], requires_grad=True)]
```

map_wires(wire_map: dict)

Returns a copy of the current operator with its wires changed according to the given wire map.

Parameters

wire_map (dict) – dictionary containing the old wires as keys and the new wires as values

Returns

new operator

Return type

.Operator

```
matrix(wire_order=None)
```

Representation of the operator as a matrix in the computational basis.

If wire_order is provided, the numerical representation considers the position of the operator's wires in the global wire order. Otherwise, the wire order defaults to the operator's wires.

If the matrix depends on trainable parameters, the result will be cast in the same autodifferentiation framework as the parameters.

A MatrixUndefinedError is raised if the matrix representation has not been defined.

See also:

```
compute_matrix()
```

Parameters

wire_order (Iterable) – global wire order, must contain all wire labels from the operator's wires

Returns

matrix representation

Return type

tensor_like

$pow(z) \rightarrow List[Operator]$

A list of new operators equal to this one raised to the given power.

Parameters

z (*float*) – exponent for the operator

Returns

list[Operator]

queue(context=<class 'pennylane.queuing.QueuingManager'>)

Append the operator to the Operator queue.

$$simplify() \rightarrow Operator$$

Reduce the depth of nested operators to the minimum.

Returns

simplified operator

Return type

.Operator

single_qubit_rot_angles()

The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.

Returns

A list of values $[\phi, \theta, \omega]$ such that $RZ(\omega)RY(\theta)RZ(\phi)$ is equivalent to the original operation.

Return type

tuple[float, float, float]

sparse_matrix(wire_order=None)

Representation of the operator as a sparse matrix in the computational basis.

If wire_order is provided, the numerical representation considers the position of the operator's wires in the global wire order. Otherwise, the wire order defaults to the operator's wires.

A SparseMatrixUndefinedError is raised if the sparse matrix representation has not been defined.

See also:

compute_sparse_matrix()

Parameters

wire_order (Iterable) – global wire order, must contain all wire labels from the operator's wires

Returns

sparse matrix representation

Return type

scipy.sparse._csr.csr_matrix

terms()

Representation of the operator as a linear combination of other operators.

$$O = \sum_{i} c_i O_i$$

A TermsUndefinedError is raised if no representation by terms is defined.

26 Chapter 2. Tutorials

Returns

list of coefficients c_i and list of operations O_i

Return type

tuple[list[tensor_like or float], list[.Operation]]

static validate_subspace(subspace)

Validate the subspace for qutrit operations.

This method determines whether a given subspace for qutrit operations is defined correctly or not. If not, a *ValueError* is thrown.

Parameters

subspace (tuple[int]) - Subspace to check for correctness

BraketAwsAhsDevice

class BraketAwsAhsDevice(wires: int | Iterable, device_arn: str, s3_destination_folder: S3DestinationFolder | None = None, *, poll_timeout_seconds: float = 432000, poll_interval_seconds: float = 1, shots: int | Shots = Shots.DEFAULT, aws_session: AwsSession | None = None)

Bases: BraketAhsDevice

Amazon Braket AHS device for hardware in PennyLane.

More information about AHS and the capabilities of the hardware can be found in the Amazon Braket Developer Guide.

Parameters

- wires (int or Iterable[int, str]) Number of subsystems represented by the device, or iterable that contains unique labels for the subsystems as numbers (i.e., [-1, 0, 2]) or strings (['ancilla', 'q1', 'q2']).
- **device_arn** (*str*) The ARN identifying the AwsDevice to be used to run circuits; The corresponding AwsDevice must support analog Hamiltonian simulation. You can get device ARNs from the Amazon Braket console or from the Amazon Braket Developer Guide.
- **s3_destination_folder** (AwsSession.S3DestinationFolder) Name of the S3 bucket and folder, specified as a tuple.
- poll_timeout_seconds (float) Total time in seconds to wait for results before timing out
- poll_interval_seconds (float) The polling interval for results in seconds.
- **shots** (*int or Shots.DEFAULT*) Number of executions to run to aquire measurements. Default: Shots.DEFAULT
- aws_session (Optional [AwsSession]) An AwsSession object created to manage interactions with AWS services, to be supplied if extra control is desired. Default: None

Note: It is important to keep track of units when specifying electromagnetic pulses for hardware control. The frequency and amplitude provided in PennyLane for Rydberg atom systems are expected to be in units of MHz, time in microseconds, phase in radians, and distance in micrometers. All of these will be converted to SI units internally as needed for upload to the hardware, and frequency will be converted to angular frequency (multiplied by 2π).

When reading hardware specifications from the Braket backend, bear in mind that all units are SI and frequencies are in rad/s. This conversion is done when creating a pulse program for upload, and units in the PennyLane functions should follow the conventions specified in the PennyLane docs to ensure correct unit conversion. See

rydberg_interaction and rydberg_drive in Pennylane for specification of expected input units, and examples for creating hardware compatible ParametrizedEvolution operators in PennyLane.

ahs_program	
analytic	Whether shots is None or not.
author	
circuit_hash	The hash of the circuit upon the last execution.
hardware_capabilities	Dictionary of hardware capabilities for the hardware device
measurement_map	Mapping used to override the logic of measurement processes.
name	
num_executions	Number of times this device is executed by the eval- uation of QNodes running on this device
obs_queue	The observables to be measured and returned.
observables	
op_queue	The operation queue to be applied.
operations	
parameters	Mapping from free parameter index to the list of Operations in the device queue that depend on it.
pennylane_requires	
register	Register a virtual subclass of an ABC.
result	
settings	Dictionary of constants set by the hardware.
short_name	
shot_vector	Returns the shot vector, a sparse representation of the shot sequence used by the device when evaluating QNodes.
shots	Number of circuit evaluations/random samples used to estimate expectation values of observables
state	Returns the state vector of the circuit prior to measurement.
stopping_condition	Returns the stopping condition for the device.
task	
version	
wire_map	Ordered dictionary that defines the map from user- provided wire labels to the wire labels used on this device
wires	All wires that can be addressed on this device

ahs_program

analytic

Whether shots is None or not. Kept for backwards compatability.

```
author = 'Xanadu Inc.'
```

circuit_hash

The hash of the circuit upon the last execution.

This can be used by devices in *apply()* for parametric compilation.

hardware_capabilities

Dictionary of hardware capabilities for the hardware device

```
measurement_map = {}
```

Mapping used to override the logic of measurement processes. The dictionary maps a measurement class to a string containing the name of a device's method that overrides the measurement process. The method defined by the device should have the following arguments:

- measurement (MeasurementProcess): measurement to override
- shot_range (tuple[int]): 2-tuple of integers specifying the range of samples to use. If not specified, all samples are used.
- bin_size (int): Divides the shot range into bins of size bin_size, and returns the measurement statistic separately over each bin. If not provided, the entire shot range is treated as a single bin.

Note: When overriding the logic of a MeasurementTransform, the method defined by the device should only have a single argument:

• tape: quantum tape to transform

Example:

Let's create device that inherits from DefaultQubit and overrides the logic of the *qml.sample* measurement. To do so we will need to update the measurement_map dictionary:

```
class NewDevice(DefaultQubit):
    def __init__(self, wires, shots):
        super().__init__(wires=wires, shots=shots)
        self.measurement_map[SampleMP] = "sample_measurement"

def sample_measurement(self, measurement, shot_range=None, bin_size=None):
    return 2
```

```
>>> dev = NewDevice(wires=2, shots=1000)
>>> @qml.qnode(dev)
... def circuit():
... return qml.sample()
>>> circuit()
tensor(2, requires_grad=True)
```

name = 'Braket Device for AHS in PennyLane'

num_executions

Number of times this device is executed by the evaluation of QNodes running on this device

```
Returns
             number of executions
         Return type
             int
obs_queue
     The observables to be measured and returned.
     Note that this property can only be accessed within the execution context of execute().
         Raises
             ValueError – if outside of the execution context
         Returns
             list[~.operation.Observable]
observables = { 'Hadamard', 'Hermitian', 'Identity', 'PauliX', 'PauliY', 'PauliZ',
'Prod', 'Projector', 'Sprod', 'Sum'}
op_queue
     The operation queue to be applied.
     Note that this property can only be accessed within the execution context of execute().
         Raises
             ValueError – if outside of the execution context
         Returns
             list[~.operation.Operation]
operations = {'ParametrizedEvolution'}
parameters
     Mapping from free parameter index to the list of Operations in the device queue that depend on it.
     Note that this property can only be accessed within the execution context of execute().
         Raises
             ValueError – if outside of the execution context
         Returns
             the mapping
         Return type
             dict[int->list[ParameterDependency]]
pennylane_requires = '>=0.30.0'
register
result
settinas
     Dictionary of constants set by the hardware.
     Used to enable initializing hardware-consistent Hamiltonians by saving all the values that would need to
```

30 Chapter 2. Tutorials

be passed, i.e.:

```
>>> dev_remote = qml.device('braket.aws.ahs', wires=3)
>>> dev_pl = qml.device('default.qubit', wires=3)
>>> settings = dev_remote.settings
>>> H_int = qml.pulse.rydberg_interaction(coordinates, **settings)
```

By passing the settings from the remote device to rydberg_interaction, an H_int Hamiltonian term is created using the constants specific to the hardware. This is relevant for simulating the hardware in PennyLane on the default.qubit device.

short_name = 'braket.aws.ahs'

shot_vector

Returns the shot vector, a sparse representation of the shot sequence used by the device when evaluating QNodes.

Example

The sparse representation of the shot sequence is returned, where tuples indicate the number of times a shot integer is repeated.

Type

list[ShotCopies]

shots

Number of circuit evaluations/random samples used to estimate expectation values of observables

state

Returns the state vector of the circuit prior to measurement.

Note: Only state vector simulators support this property. Please see the plugin documentation for more details.

stopping_condition

Returns the stopping condition for the device. The returned function accepts a queuable object (including a PennyLane operation and observable) and returns True if supported by the device.

Type

.BooleanFn

task

version = '0.32.0'

wire_map

Ordered dictionary that defines the map from user-provided wire labels to the wire labels used on this device

wires

All wires that can be addressed on this device

access_state([wires])	Check that the device has access to an internal state
	and return it if available.
<pre>active_wires(operators)</pre>	Returns the wires acted on by a set of operators.
<pre>adjoint_jacobian(tape[, starting_state,])</pre>	Implements the adjoint method outlined in Jones and
	Gacon to differentiate an input tape.
<pre>analytic_probability([wires])</pre>	Return the (marginal) probability of each computa-
2 12 2	tional basis state from the last run of the device.
apply(operations, **kwargs)	Convert the pulse operation to an AHS program and
	run on the connected device
batch_execute(circuits)	Execute a batch of quantum circuits on the device.
batch_transform(circuit)	Apply a differentiable batch transform for preprocess-
	ing a circuit prior to execution.
capabilities()	Get the capabilities of this device class.
check_validity(queue, observables)	Checks whether the operations and observables in
encen_variates (queue, observables)	queue are all supported by the device.
classical_shadow(obs, circuit)	Returns the measured bits and recipes in the classical
crassical_shadow(oos, chedit)	shadow protocol.
<pre>create_ahs_program(evolution)</pre>	Create AHS program for upload to hardware from a
create_ans_program(evolution)	ParametrizedEvolution
custom_expand(fn)	Register a custom expansion function for the device.
default_expand_fn(circuit[, max_expansion])	Method for expanding or decomposing an input cir-
deraute_expand_in(circuit[, max_expansion])	cuit.
define_wire_map(wires)	Create the map from user-provided wire labels to the
30	wire labels used by the device.
density_matrix(wires)	Returns the reduced density matrix over the given
	wires.
<pre>estimate_probability([wires, shot_range,])</pre>	Return the estimated probability of each computa-
5) ([···································	tional basis state using the generated samples.
execute(circuit, **kwargs)	It executes a queue of quantum operations on the de-
	vice and then measure the given observables.
<pre>execute_and_gradients(circuits[, method])</pre>	Execute a batch of quantum circuits on the device,
(and return both the results and the gradients.
<pre>execution_context()</pre>	The device execution context used during calls to
()	execute().
<pre>expand_fn(circuit[, max_expansion])</pre>	Method for expanding or decomposing an input cir-
companies (encure, mani_onpanies sin)	cuit.
<pre>expval(observable[, shot_range, bin_size])</pre>	Returns the expectation value of observable on spec-
	ified wires.
<pre>generate_basis_states(num_wires[, dtype])</pre>	Generates basis states in binary representation ac-
	cording to the number of wires specified.
<pre>generate_samples()</pre>	Returns the computational basis samples measured
	for all wires.
<pre>gradients(circuits[, method])</pre>	Return the gradients of a batch of quantum circuits on
,	the device.

continues on next page

32 Chapter 2. Tutorials

Table 1 – continued from previous page

	Thom provided page
<pre>map_wires(wires)</pre>	Map the wire labels of wires using this device's wire
	map.
<pre>marginal_prob(prob[, wires])</pre>	Return the marginal probability of the computational basis states by summing the probabilities on the non-specified wires.
<pre>mutual_info(wires0, wires1, log_base)</pre>	Returns the mutual information prior to measurement:
order_wires(subset_wires)	Given some subset of device wires return a Wires object with the same wires; sorted according to the device wire map.
<pre>post_apply()</pre>	Called during <i>execute()</i> after the individual operations have been executed.
<pre>post_measure()</pre>	Called during <i>execute()</i> after the individual observables have been measured.
<pre>pre_apply()</pre>	Called during <i>execute()</i> before the individual operations are executed.
<pre>pre_measure()</pre>	Called during <i>execute()</i> before the individual observables are measured.
<pre>probability([wires, shot_range, bin_size])</pre>	Return either the analytic probability or estimated probability of each computational basis state.
reset()	Reset the backend state.
<pre>sample(observable[, shot_range, bin_size,])</pre>	Return samples of an observable.
sample_basis_states(number_of_states,)	Sample from the computational basis states based on
	the state probability.
shadow_expval(obs, circuit)	Compute expectation values using classical shadows in a differentiable manner.
<pre>shot_vec_statistics(circuit)</pre>	Process measurement results from circuit execution using a device with a shot vector and return statistics.
<pre>states_to_binary(samples, num_wires[, dtype])</pre>	Convert basis states from base 10 to binary representation.
<pre>statistics(circuit[, shot_range, bin_size])</pre>	Process measurement results from circuit execution and return statistics.
<pre>supports_observable(observable)</pre>	Checks if an observable is supported by this device. Raises a ValueError,
<pre>supports_operation(operation)</pre>	Checks if an operation is supported by this device.
<pre>var(observable[, shot_range, bin_size])</pre>	Returns the variance of observable on specified wires.
vn_entropy(wires, log_base)	Returns the Von Neumann entropy prior to measurement.

access_state(wires=None)

Check that the device has access to an internal state and return it if available.

Parameters

wires (Wires) – wires of the reduced system

Raises

QuantumFunctionError – if the device is not capable of returning the state

Returns

the state or the density matrix of the device

Return type

array or tensor

static active_wires(operators)

Returns the wires acted on by a set of operators.

Parameters

operators (list[Operation]) – operators for which we are gathering the active wires

Returns

wires activated by the specified operators

Return type

Wires

adjoint_jacobian(tape: QuantumTape, starting_state=None, use_device_state=False)

Implements the adjoint method outlined in Jones and Gacon to differentiate an input tape.

After a forward pass, the circuit is reversed by iteratively applying adjoint gates to scan backwards through the circuit.

Note: The adjoint differentiation method has the following restrictions:

- As it requires knowledge of the statevector, only statevector simulator devices can be used.
- Only expectation values are supported as measurements.
- Does not work for parametrized observables like Hamiltonian or Hermitian.

Parameters

tape (. QuantumTape) – circuit that the function takes the gradient of

Keyword Arguments

- **starting_state** (*tensor_like*) post-forward pass state to start execution with. It should be complex-valued. Takes precedence over use_device_state.
- **use_device_state** (*bool*) use current device state to initialize. A forward pass of the same circuit should be the last thing the device has executed. If a starting_state is provided, that takes precedence.

Returns

the derivative of the tape with respect to trainable parameters. Dimensions are (len(observables), len(trainable_params)).

Return type

array or tuple[array]

Raises

34

QuantumFunctionError – if the input tape has measurements that are not expectation values or contains a multi-parameter operation aside from Rot

analytic_probability(wires=None)

Return the (marginal) probability of each computational basis state from the last run of the device.

PennyLane uses the convention $|q_0, q_1, \dots, q_{N-1}\rangle$ where q_0 is the most significant bit.

If no wires are specified, then all the basis states representable by the device are considered and no marginalization takes place.

Note: marginal_prob() may be used as a utility method to calculate the marginal probability distribution.

Parameters

wires (Iterable[Number, str], Number, str, Wires) — wires to return marginal probabilities for. Wires not provided are traced out of the system.

Returns

list of the probabilities

Return type

array[float]

apply(operations: list[ParametrizedEvolution], **kwargs)

Convert the pulse operation to an AHS program and run on the connected device

Parameters

 $\begin{tabular}{ll} \textbf{operations} & (\textit{list[ParametrizedEvolution]}) & - & a & list & containing & a & single \\ \textbf{ParametrizedEvolution operator} & & & & \\ \end{tabular}$

batch_execute(circuits)

Execute a batch of quantum circuits on the device.

The circuits are represented by tapes, and they are executed one-by-one using the device's execute method. The results are collected in a list.

For plugin developers: This function should be overwritten if the device can efficiently run multiple circuits on a backend, for example using parallel and/or asynchronous executions.

Parameters

circuits (list[QuantumTape]) - circuits to execute on the device

Returns

list of measured value(s)

Return type

list[array[float]]

batch_transform(circuit: QuantumTape)

Apply a differentiable batch transform for preprocessing a circuit prior to execution. This method is called directly by the QNode, and should be overwritten if the device requires a transform that generates multiple circuits prior to execution.

By default, this method contains logic for generating multiple circuits, one per term, of a circuit that terminates in expval(H), if the underlying device does not support Hamiltonian expectation values, or if the device requires finite shots.

Warning: This method will be tracked by autodifferentiation libraries, such as Autograd, JAX, Tensor-Flow, and Torch. Please make sure to use qml.math for autodiff-agnostic tensor processing if required.

Parameters

circuit (.QuantumTape) - the circuit to preprocess

Returns

Returns a tuple containing the sequence of circuits to be executed, and a post-processing function to be applied to the list of evaluated circuit results.

Return type

tuple[Sequence[.QuantumTape], callable]

classmethod capabilities()

Get the capabilities of this device class.

Inheriting classes that change or add capabilities must override this method, for example via

```
@classmethod
def capabilities(cls):
    capabilities = super().capabilities().copy()
    capabilities.update(
        supports_a_new_capability=True,
    )
    return capabilities
```

Returns

results

Return type

dict[str->*]

check_validity(queue, observables)

Checks whether the operations and observables in queue are all supported by the device.

Parameters

- **queue** (*Iterable[Operation]*) quantum operation objects which are intended to be applied on the device
- **observables** (*Iterable[Observable]*) observables which are intended to be evaluated on the device

Raises

 ${f Exception}$ — if there are operations in the queue or observables that the device does not support

classical_shadow(obs, circuit)

Returns the measured bits and recipes in the classical shadow protocol.

The protocol is described in detail in the classical shadows paper. This measurement process returns the randomized Pauli measurements (the recipes) that are performed for each qubit and snapshot as an integer:

- · 0 for Pauli X.
- 1 for Pauli Y, and
- 2 for Pauli Z.

It also returns the measurement results (the bits); 0 if the 1 eigenvalue is sampled, and 1 if the -1 eigenvalue is sampled.

The device shots are used to specify the number of snapshots. If T is the number of shots and n is the number of qubits, then both the measured bits and the Pauli measurements have shape (T, n).

This implementation is device-agnostic and works by executing single-shot tapes containing randomized Pauli observables. Devices should override this if they can offer cleaner or faster implementations.

See also:

classical_shadow()

Parameters

- **obs** (ClassicalShadowMP) The classical shadow measurement process
- circuit (QuantumTape) The quantum tape that is being executed

Returns

A tensor with shape (2, T, n), where the first row represents the measured bits and the second represents the recipes used.

Return type

tensor_like[int]

create_ahs_program(evolution: ParametrizedEvolution)

Create AHS program for upload to hardware from a ParametrizedEvolution

Parameters

evolution (*ParametrizedEvolution*) – the PennyLane operator describing the pulse to be converted into an AnalogHamiltonianSimulation program

Returns

a program containing the register and drive

information for running an AHS task on simulation or hardware

Return type

AnalogHamiltonianSimulation

custom_expand(fn)

Register a custom expansion function for the device.

Example

```
dev = qml.device("default.qubit", wires=2)

@dev.custom_expand
def my_expansion_function(self, tape, max_expansion=10):
    ...
    # can optionally call the default device expansion
    tape = self.default_expand_fn(tape, max_expansion=max_expansion)
    return tape
```

The custom device expansion function must have arguments self (the device object), tape (the input circuit to transform and execute), and max_expansion (the number of times the circuit should be expanded).

The default default_expand_fn() method of the original device may be called. It is highly recommended to call this before returning, to ensure that the expanded circuit is supported on the device.

default_expand_fn(circuit, max expansion=10)

Method for expanding or decomposing an input circuit. This method should be overwritten if custom expansion logic is required.

By default, this method expands the tape if:

- state preparation operations are called mid-circuit,
- nested tapes are present,
- any operations are not supported on the device, or
- multiple observables are measured on the same wire.

Parameters

- **circuit** (.QuantumTape) the circuit to expand.
- max_expansion (int) The number of times the circuit should be expanded. Expansion occurs when an operation or measurement is not supported, and results in a gate decomposition. If any operations in the decomposition remain unsupported by the device, another expansion occurs.

Returns

The expanded/decomposed circuit, such that the device will natively support all operations.

Return type

.QuantumTape

define_wire_map(wires)

Create the map from user-provided wire labels to the wire labels used by the device.

The default wire map maps the user wire labels to wire labels that are consecutive integers.

However, by overwriting this function, devices can specify their preferred, non-consecutive and/or non-integer wire labels.

Parameters

wires (Wires) – user-provided wires for this device

Returns

dictionary specifying the wire map

Return type

OrderedDict

Example

density_matrix(wires)

Returns the reduced density matrix over the given wires.

Parameters

wires (Wires) – wires of the reduced system

Returns

complex array of shape (2 ** len(wires), 2 ** len(wires)) representing the reduced density matrix of the state prior to measurement.

Return type

array[complex]

estimate_probability(wires=None, shot_range=None, bin_size=None)

Return the estimated probability of each computational basis state using the generated samples.

Parameters

- wires (Iterable[Number, str], Number, str, Wires) wires to calculate marginal probabilities for. Wires not provided are traced out of the system.
- **shot_range** (tuple[int]) 2-tuple of integers specifying the range of samples to use. If not specified, all samples are used.

• **bin_size** (*int*) – Divides the shot range into bins of size bin_size, and returns the measurement statistic separately over each bin. If not provided, the entire shot range is treated as a single bin.

Returns

list of the probabilities

Return type

array[float]

execute(circuit, **kwargs)

It executes a queue of quantum operations on the device and then measure the given observables.

For plugin developers: instead of overwriting this, consider implementing a suitable subset of

- apply()
- generate_samples()
- probability()

Additional keyword arguments may be passed to this method that can be utilised by apply(). An example would be passing the QNode hash that can be used later for parametric compilation.

Parameters

circuit (QuantumTape) – circuit to execute on the device

Raises

QuantumFunctionError – if the value of return_type is not supported

Returns

measured value(s)

Return type

array[float]

execute_and_gradients(circuits, method='jacobian', **kwargs)

Execute a batch of quantum circuits on the device, and return both the results and the gradients.

The circuits are represented by tapes, and they are executed one-by-one using the device's execute method. The results and the corresponding Jacobians are collected in a list.

For plugin developers: This method should be overwritten if the device can efficiently run multiple circuits on a backend, for example using parallel and/or asynchronous executions, and return both the results and the Jacobians.

Parameters

- circuits (list[.tape.QuantumTape]) circuits to execute on the device
- **method** (str) the device method to call to compute the Jacobian of a single circuit
- **kwargs keyword argument to pass when calling method

Returns

Tuple containing list of measured value(s) and list of Jacobians. Returned Jacobians should be of shape (output_shape, num_params).

Return type

tuple[list[array[float]], list[array[float]]]

execution_context()

The device execution context used during calls to execute().

You can overwrite this function to return a context manager in case your quantum library requires that; all operations and method calls (including apply() and expval()) are then evaluated within the context of this context manager (see the source of execute()) for more details).

```
expand_fn(circuit, max_expansion=10)
```

Method for expanding or decomposing an input circuit. Can be the default or a custom expansion method, see Device.default_expand_fn() and Device.custom_expand() for more details.

Parameters

- **circuit** (.QuantumTape) the circuit to expand.
- max_expansion (int) The number of times the circuit should be expanded. Expansion occurs when an operation or measurement is not supported, and results in a gate decomposition. If any operations in the decomposition remain unsupported by the device, another expansion occurs.

Returns

The expanded/decomposed circuit, such that the device will natively support all operations.

Return type

.QuantumTape

expval(observable, shot_range=None, bin_size=None)

Returns the expectation value of observable on specified wires.

Note: all arguments accept _lists_, which indicate a tensor product of observables.

Parameters

- **observable** (str or list[str]) name of the observable(s)
- wires (Wires) wires the observable(s) are to be measured on
- par (tuple or list[tuple]]) parameters for the observable(s)

Returns

expectation value $A = \psi A \psi$

Return type

float

static generate_basis_states(num wires, dtype=<class 'numpy.uint32'>)

Generates basis states in binary representation according to the number of wires specified.

The states_to_binary method creates basis states faster (for larger systems at times over x25 times faster) than the approach using itertools.product, at the expense of using slightly more memory.

Due to the large size of the integer arrays for more than 32 bits, memory allocation errors may arise in the states_to_binary method. Hence we constraint the dtype of the array to represent unsigned integers on 32 bits. Due to this constraint, an overflow occurs for 32 or more wires, therefore this approach is used only for fewer wires.

For smaller number of wires speed is comparable to the next approach (using itertools.product), hence we resort to that one for testing purposes.

Parameters

- num_wires (int) the number wires
- dtype=np.uint32 (type) the data type of the arrays to use

Returns

the sampled basis states

Return type

array[int]

generate_samples()

Returns the computational basis samples measured for all wires.

Returns

array of samples in the shape (dev.shots, dev.num_wires)

Return type

array[complex]

gradients(circuits, method='jacobian', **kwargs)

Return the gradients of a batch of quantum circuits on the device.

The gradient method method is called sequentially for each circuit, and the corresponding Jacobians are collected in a list.

For plugin developers: This method should be overwritten if the device can efficiently compute the gradient of multiple circuits on a backend, for example using parallel and/or asynchronous executions.

Parameters

- circuits (list[.tape.QuantumTape]) circuits to execute on the device
- **method** (*str*) the device method to call to compute the Jacobian of a single circuit
- **kwargs keyword argument to pass when calling method

Returns

List of Jacobians. Returned Jacobians should be of shape (output_shape, num_params).

Return type

list[array[float]]

map_wires(wires)

Map the wire labels of wires using this device's wire map.

Parameters

wires (Wires) – wires whose labels we want to map to the device's internal labelling scheme

Returns

wires with new labels

Return type

Wires

marginal_prob(prob, wires=None)

Return the marginal probability of the computational basis states by summing the probabilities on the non-specified wires.

If no wires are specified, then all the basis states representable by the device are considered and no marginalization takes place.

Note: If the provided wires are not in the order as they appear on the device, the returned marginal probabilities take this permutation into account.

For example, if the addressable wires on this device are Wires([0, 1, 2]) and this function gets passed wires=[2, 0], then the returned marginal probability vector will take this 'reversal' of the two wires into

account:

$$\mathbb{P}^{(2,0)} = [|00\rangle, |10\rangle, |01\rangle, |11\rangle]$$

Parameters

- prob The probabilities to return the marginal probabilities for
- wires (Iterable[Number, str], Number, str, Wires) wires to return marginal probabilities for. Wires not provided are traced out of the system.

Returns

array of the resulting marginal probabilities.

Return type

array[float]

mutual_info(wires0, wires1, log_base)

Returns the mutual information prior to measurement:

$$I(A, B) = S(\rho^A) + S(\rho^B) - S(\rho^{AB})$$

where S is the von Neumann entropy.

Parameters

- wires (Wires) wires of the first subsystem
- wires1 (Wires) wires of the second subsystem
- **log_base** (*float*) base to use in the logarithm

Returns

the mutual information

Return type

float

order_wires(subset_wires)

Given some subset of device wires return a Wires object with the same wires; sorted according to the device wire map.

Parameters

subset_wires (*Wires*) – The subset of device wires (in any order).

Raises

ValueError – Could not find some or all subset wires subset_wires in device wires device_wires.

Returns

a new Wires object containing the re-ordered wires set

Return type

ordered_wires (Wires)

post_apply()

Called during execute() after the individual operations have been executed.

post_measure()

Called during *execute()* after the individual observables have been measured.

pre_apply()

Called during *execute()* before the individual operations are executed.

pre_measure()

Called during *execute()* before the individual observables are measured.

```
probability(wires=None, shot_range=None, bin_size=None)
```

Return either the analytic probability or estimated probability of each computational basis state.

Devices that require a finite number of shots always return the estimated probability.

Parameters

wires (Iterable[Number, str], Number, str, Wires) — wires to return marginal probabilities for. Wires not provided are traced out of the system.

Returns

list of the probabilities

Return type

array[float]

reset()

Reset the backend state.

After the reset, the backend should be as if it was just constructed. Most importantly the quantum state is reset to its initial value.

sample(observable, shot_range=None, bin_size=None, counts=False)

Return samples of an observable.

Parameters

- **observable** (*Observable*) the observable to sample
- **shot_range** (tuple[int]) 2-tuple of integers specifying the range of samples to use. If not specified, all samples are used.
- bin_size (int) Divides the shot range into bins of size bin_size, and returns the measurement statistic separately over each bin. If not provided, the entire shot range is treated as a single bin.
- counts (bool) whether counts (True) or raw samples (False) should be returned

Raises

EigvalsUndefinedError – if no information is available about the eigenvalues of the observable

Returns

samples in an array of dimension (shots,) or counts

Return type

Union[array[float], dict, list[dict]]

sample_basis_states(number_of_states, state_probability)

Sample from the computational basis states based on the state probability.

This is an auxiliary method to the generate_samples method.

Parameters

- **number_of_states** (*int*) the number of basis states to sample from
- **state_probability** (array[float]) the computational basis probability vector

Returns

the sampled basis states

Return type

array[int]

shadow_expval(obs, circuit)

Compute expectation values using classical shadows in a differentiable manner.

Please refer to shadow_expval() for detailed documentation.

Parameters

- obs (ClassicalShadowMP) The classical shadow expectation value measurement process
- **circuit** (*QuantumTape*) The quantum tape that is being executed

Returns

expectation value estimate.

Return type

float

shot_vec_statistics(circuit: QuantumTape)

Process measurement results from circuit execution using a device with a shot vector and return statistics.

This is an auxiliary method of execute and uses statistics.

When using shot vectors, measurement results for each item of the shot vector are contained in a tuple.

Parameters

circuit (QuantumTape) – circuit to execute on the device

Raises

QuantumFunctionError – if the value of return_type is not supported

Returns

stastics for each shot item from the shot vector

Return type

tuple

static states_to_binary(samples, num_wires, dtype=<class 'numpy.int64'>)

Convert basis states from base 10 to binary representation.

This is an auxiliary method to the generate_samples method.

Parameters

- samples (array[int]) samples of basis states in base 10 representation
- **num_wires** (*int*) the number of qubits
- **dtype** (*type*) Type of the internal integer array to be used. Can be important to specify for large systems for memory allocation purposes.

Returns

basis states in binary representation

Return type

array[int]

statistics(*circuit: QuantumTape*, *shot_range=None*, *bin_size=None*)

Process measurement results from circuit execution and return statistics.

This includes returning expectation values, variance, samples, probabilities, states, and density matrices.

Parameters

- **circuit** (*QuantumTape*) the quantum tape currently being executed
- **shot_range** (tuple[int]) 2-tuple of integers specifying the range of samples to use. If not specified, all samples are used.
- **bin_size** (*int*) Divides the shot range into bins of size bin_size, and returns the measurement statistic separately over each bin. If not provided, the entire shot range is treated as a single bin.

Raises

QuantumFunctionError – if the value of return_type is not supported

Returns

the corresponding statistics

Return type

Union[float, List[float]]

supports_observable(observable)

Checks if an observable is supported by this device. Raises a ValueError,

if not a subclass or string of an Observable was passed.

Parameters

```
observable (type or str) – observable to be checked
```

Raises

ValueError – if *observable* is not a Observable class or string

Returns

True iff supplied observable is supported

Return type

bool

supports_operation(operation)

Checks if an operation is supported by this device.

Parameters

```
operation (type or str) – operation to be checked
```

Raises

ValueError – if *operation* is not a Operation class or string

Returns

True if supplied operation is supported

Return type

bool

var(observable, shot_range=None, bin_size=None)

Returns the variance of observable on specified wires.

Note: all arguments support _lists_, which indicate a tensor product of observables.

Parameters

- **observable** (str or list[str]) name of the observable(s)
- wires (Wires) wires the observable(s) is to be measured on
- par (tuple or list[tuple]]) parameters for the observable(s)

Raises

NotImplementedError – if the device does not support variance computation

Returns

variance
$$var(A) = \psi A^2 \psi - \psi A \psi^2$$

Return type

float

vn_entropy(wires, log_base)

Returns the Von Neumann entropy prior to measurement.

$$S(\rho) = -\text{Tr}(\rho \log(\rho))$$

Parameters

- wires (Wires) Wires of the considered subsystem.
- **log_base** (*float*) Base for the logarithm, default is None the natural logarithm is used in this case.

Returns

returns the Von Neumann entropy

Return type

float

BraketAwsQubitDevice

class BraketAwsQubitDevice(wires: int | Iterable, device_arn: str, s3_destination_folder: S3DestinationFolder

| None = None, *, shots: int | None | Shots = Shots.DEFAULT,

poll_timeout_seconds: float = 432000, poll_interval_seconds: float = 1,

aws_session: AwsSession | None = None, parallel: bool = False, max_parallel:

int | None = None, max_connections: int = 100, max_retries: int = 3,

**run_kwargs)

Bases: BraketQubitDevice

Amazon Braket AwsDevice qubit device for PennyLane.

Parameters

- wires (int or Iterable[Number, str]]) Number of subsystems represented by the device, or iterable that contains unique labels for the subsystems as numbers (i.e., [-1, 0, 2]) or strings (['ancilla', 'q1', 'q2']).
- device_arn (str) The ARN identifying the AwsDevice to be used to run circuits; The
 corresponding AwsDevice must support quantum circuits via OpenQASM. You can get device ARNs using AwsDevice.get_devices, from the Amazon Braket console or from the
 Amazon Braket Developer Guide.
- **s3_destination_folder** (AwsSession.S3DestinationFolder) Name of the S3 bucket and folder, specified as a tuple.
- **poll_timeout_seconds** (*float*) Total time in seconds to wait for results before timing out.

- poll_interval_seconds (float) The polling interval for results in seconds.
- **shots** (*int*, *None or Shots.DEFAULT*) Number of circuit evaluations or random samples included, to estimate expectation values of observables. If set to Shots.DEFAULT, uses the default number of shots specified by the remote device. If shots is set to 0 or None, the device runs in analytic mode (calculations will be exact). Analytic mode is not available on QPU and hence an error will be raised. Default: Shots.DEFAULT
- aws_session(Optional[AwsSession]) An AwsSession object created to manage interactions with AWS services, to be supplied if extra control is desired. Default: None Default: False
- max_parallel (int, optional) Maximum number of tasks to run on AWS in parallel. Batch creation will fail if this value is greater than the maximum allowed concurrent tasks on the device. If unspecified, uses defaults defined in AwsDevice. Ignored if parallel=False.
- max_connections (int) The maximum number of connections in the Boto3 connection pool. Also the maximum number of thread pool workers for the batch. Ignored if parallel=False.
- max_retries (int) The maximum number of retries to use for batch execution. When executing tasks in parallel, failed tasks will be retried up to max_retries times. Ignored if parallel=False.
- **verbatim** (*bool*) Whether to verbatim mode for the device. Note that verbatim mode only supports the native gate set of the device. Default False.
- **run_kwargs Variable length keyword arguments for braket.devices.Device.run().

analytic	Whether shots is None or not.
author	Whether shots is I tolle of hot
circuit	The last circuit run on this device.
circuit_hash	The hash of the circuit upon the last execution.
measurement_map	Mapping used to override the logic of measurement processes.
name	
num_executions	Number of times this device is executed by the eval- uation of QNodes running on this device
obs_queue	The observables to be measured and returned.
observables	<pre>set() -> new empty set object set(iterable) -> new set object</pre>
op_queue	The operation queue to be applied.
operations	The set of names of PennyLane operations that the device supports.
parallel	
parameters	Mapping from free parameter index to the list of Operations in the device queue that depend on it.
pennylane_requires	
short_name	
shot_vector	Returns the shot vector, a sparse representation of the shot sequence used by the device when evaluating QNodes.
shots	Number of circuit evaluations/random samples used to estimate expectation values of observables
state	Returns the state vector of the circuit prior to measurement.
stopping_condition	Returns the stopping condition for the device.
task	The task corresponding to the last run circuit.
use_grouping	
version	
wire_map	Ordered dictionary that defines the map from user- provided wire labels to the wire labels used on this device
wires	All wires that can be addressed on this device

analytic

Whether shots is None or not. Kept for backwards compatability.

author = 'Amazon Web Services'

circuit

The last circuit run on this device.

Type

Circuit

48 Chapter 2. Tutorials

circuit_hash

The hash of the circuit upon the last execution.

This can be used by devices in *apply()* for parametric compilation.

```
measurement_map = {}
```

Mapping used to override the logic of measurement processes. The dictionary maps a measurement class to a string containing the name of a device's method that overrides the measurement process. The method defined by the device should have the following arguments:

- measurement (MeasurementProcess): measurement to override
- shot_range (tuple[int]): 2-tuple of integers specifying the range of samples to use. If not specified, all samples are used.
- bin_size (int): Divides the shot range into bins of size bin_size, and returns the measurement statistic separately over each bin. If not provided, the entire shot range is treated as a single bin.

Note: When overriding the logic of a MeasurementTransform, the method defined by the device should only have a single argument:

• tape: quantum tape to transform

Example:

Let's create device that inherits from DefaultQubit and overrides the logic of the *qml.sample* measurement. To do so we will need to update the measurement_map dictionary:

```
class NewDevice(DefaultQubit):
    def __init__(self, wires, shots):
        super().__init__(wires=wires, shots=shots)
        self.measurement_map[SampleMP] = "sample_measurement"

def sample_measurement(self, measurement, shot_range=None, bin_size=None):
    return 2
```

```
>>> dev = NewDevice(wires=2, shots=1000)
>>> @qml.qnode(dev)
... def circuit():
... return qml.sample()
>>> circuit()
tensor(2, requires_grad=True)
```

name = 'Braket AwsDevice for PennyLane'

num_executions

Number of times this device is executed by the evaluation of QNodes running on this device

Returns

number of executions

Return type

int

obs_queue

The observables to be measured and returned.

Note that this property can only be accessed within the execution context of execute().

Raises

ValueError – if outside of the execution context

Returns

list[~.operation.Observable]

observables

op_queue

The operation queue to be applied.

Note that this property can only be accessed within the execution context of execute().

Raises

ValueError – if outside of the execution context

Returns

list[~.operation.Operation]

operations

The set of names of PennyLane operations that the device supports.

Type

frozenset[str]

parallel

parameters

Mapping from free parameter index to the list of Operations in the device queue that depend on it.

Note that this property can only be accessed within the execution context of execute().

Raises

ValueError – if outside of the execution context

Returns

the mapping

Return type

dict[int->list[ParameterDependency]]

```
pennylane_requires = '>=0.30.0'
```

```
short_name = 'braket.aws.qubit'
```

shot_vector

Returns the shot vector, a sparse representation of the shot sequence used by the device when evaluating QNodes.

Example

```
>>> dev = qml.device("default.qubit", wires=2, shots=[3, 1, 2, 2, 2, 2, 6, 1, 1, 5, 12, 10, 10])
>>> dev.shots
57
>>> dev.shot_vector
```

(continues on next page)

(continued from previous page)

```
[ShotCopies(3 shots x 1),
  ShotCopies(1 shots x 1),
  ShotCopies(2 shots x 4),
  ShotCopies(6 shots x 1),
  ShotCopies(1 shots x 2),
  ShotCopies(5 shots x 1),
  ShotCopies(12 shots x 1),
  ShotCopies(10 shots x 2)]
```

The sparse representation of the shot sequence is returned, where tuples indicate the number of times a shot integer is repeated.

Type

list[ShotCopies]

shots

Number of circuit evaluations/random samples used to estimate expectation values of observables

state

Returns the state vector of the circuit prior to measurement.

Note: Only state vector simulators support this property. Please see the plugin documentation for more details.

stopping_condition

Returns the stopping condition for the device. The returned function accepts a queuable object (including a PennyLane operation and observable) and returns True if supported by the device.

Type

.BooleanFn

task

The task corresponding to the last run circuit.

Type

QuantumTask

use_grouping

```
version = '1.22.1.dev0'
```

wire_map

Ordered dictionary that defines the map from user-provided wire labels to the wire labels used on this device

wires

All wires that can be addressed on this device

access_state([wires])	Check that the device has access to an internal state and return it if available.
<pre>active_wires(operators)</pre>	Returns the wires acted on by a set of operators.
adjoint_jacobian(tape[, starting_state,])	Implements the adjoint method outlined in Jones and Gacon to differentiate an input tape.

continues on next page

Table 2 – continued from previous page

<pre>analytic_probability([wires])</pre>	Return the (marginal) probability of each computa-
	tional basis state from the last run of the device.
<pre>apply(operations[, rotations,])</pre>	Instantiate Braket Circuit object.
batch_execute(circuits, **run_kwargs)	Execute a batch of quantum circuits on the device.
batch_transform(circuit)	Apply a differentiable batch transform for preprocess-
	ing a circuit prior to execution.
capabilities()	Add support for AG on sv1
<pre>check_validity(queue, observables)</pre>	Checks whether the operations and observables in queue are all supported by the device.
classical_shadow(obs, circuit)	Returns the measured bits and recipes in the classical shadow protocol.
custom_expand(fn)	Register a custom expansion function for the device.
<pre>default_expand_fn(circuit[, max_expansion])</pre>	Method for expanding or decomposing an input circuit.
<pre>define_wire_map(wires)</pre>	Create the map from user-provided wire labels to the wire labels used by the device.
<pre>density_matrix(wires)</pre>	Returns the reduced density matrix over the given wires.
<pre>estimate_probability([wires, shot_range,])</pre>	Return the estimated probability of each computational basis state using the generated samples.
<pre>execute(circuit[, compute_gradient])</pre>	It executes a queue of quantum operations on the device and then measure the given observables.
<pre>execute_and_gradients(circuits, **kwargs)</pre>	Execute a list of circuits and calculate their gradients.
execution_context()	The device execution context used during calls to execute().
<pre>expand_fn(circuit[, max_expansion])</pre>	Method for expanding or decomposing an input circuit.
<pre>expval(observable[, shot_range, bin_size])</pre>	Returns the expectation value of observable on specified wires.
<pre>generate_basis_states(num_wires[, dtype])</pre>	Generates basis states in binary representation according to the number of wires specified.
<pre>generate_samples()</pre>	Returns the computational basis samples generated for all wires.
<pre>gradients(circuits[, method])</pre>	Return the gradients of a batch of quantum circuits on the device.
<pre>map_wires(wires)</pre>	Map the wire labels of wires using this device's wire map.
<pre>marginal_prob(prob[, wires])</pre>	Return the marginal probability of the computational basis states by summing the probabilities on the non-specified wires.
<pre>mutual_info(wires0, wires1, log_base)</pre>	Returns the mutual information prior to measurement:
order_wires(subset_wires)	Given some subset of device wires return a Wires object with the same wires; sorted according to the device wire map.
<pre>post_apply()</pre>	Called during <i>execute()</i> after the individual operations have been executed.
<pre>post_measure()</pre>	Called during <i>execute()</i> after the individual observables have been measured.
<pre>pre_apply()</pre>	Called during <i>execute()</i> before the individual operations are executed.

continues on next page

Table 2 – continued from previous page

<pre>pre_measure()</pre>	Called during <i>execute()</i> before the individual observables are measured.
<pre>probability([wires, shot_range, bin_size])</pre>	Return either the analytic probability or estimated probability of each computational basis state.
reset()	Reset the backend state.
<pre>sample(observable[, shot_range, bin_size,])</pre>	Return samples of an observable.
<pre>sample_basis_states(number_of_states,)</pre>	Sample from the computational basis states based on the state probability.
shadow_expval(obs, circuit)	Compute expectation values using classical shadows in a differentiable manner.
<pre>shot_vec_statistics(circuit)</pre>	Process measurement results from circuit execution using a device with a shot vector and return statistics.
<pre>states_to_binary(samples, num_wires[, dtype])</pre>	Convert basis states from base 10 to binary representation.
<pre>statistics(braket_result, observables)</pre>	Processes measurement results from a Braket task result and returns statistics.
<pre>supports_observable(observable)</pre>	Checks if an observable is supported by this device. Raises a ValueError,
<pre>supports_operation(operation)</pre>	Checks if an operation is supported by this device.
<pre>var(observable[, shot_range, bin_size])</pre>	Returns the variance of observable on specified wires.
vn_entropy(wires, log_base)	Returns the Von Neumann entropy prior to measurement.

access_state(wires=None)

Check that the device has access to an internal state and return it if available.

Parameters

wires (Wires) – wires of the reduced system

Raises

QuantumFunctionError – if the device is not capable of returning the state

Returns

the state or the density matrix of the device

Return type

array or tensor

static active_wires(operators)

Returns the wires acted on by a set of operators.

Parameters

operators (list[Operation]) – operators for which we are gathering the active wires

Returns

wires activated by the specified operators

Return type

Wires

adjoint_jacobian(tape: QuantumTape, starting_state=None, use_device_state=False)

Implements the adjoint method outlined in Jones and Gacon to differentiate an input tape.

After a forward pass, the circuit is reversed by iteratively applying adjoint gates to scan backwards through the circuit.

Note: The adjoint differentiation method has the following restrictions:

- · As it requires knowledge of the statevector, only statevector simulator devices can be used.
- Only expectation values are supported as measurements.
- Does not work for parametrized observables like Hamiltonian or Hermitian.

Parameters

tape (. QuantumTape) – circuit that the function takes the gradient of

Keyword Arguments

- **starting_state** (*tensor_like*) post-forward pass state to start execution with. It should be complex-valued. Takes precedence over use_device_state.
- use_device_state (bool) use current device state to initialize. A forward pass of the same circuit should be the last thing the device has executed. If a starting_state is provided, that takes precedence.

Returns

the derivative of the tape with respect to trainable parameters. Dimensions are (len(observables), len(trainable_params)).

Return type

array or tuple[array]

Raises

QuantumFunctionError – if the input tape has measurements that are not expectation values or contains a multi-parameter operation aside from Rot

analytic_probability(wires=None)

Return the (marginal) probability of each computational basis state from the last run of the device.

PennyLane uses the convention $|q_0, q_1, \dots, q_{N-1}\rangle$ where q_0 is the most significant bit.

If no wires are specified, then all the basis states representable by the device are considered and no marginalization takes place.

Note: marginal_prob() may be used as a utility method to calculate the marginal probability distribution.

Parameters

wires (Iterable[Number, str], Number, str, Wires) — wires to return marginal probabilities for. Wires not provided are traced out of the system.

Returns

list of the probabilities

Return type

array[float]

```
apply(operations: Sequence[Operation], rotations: Sequence[Operation] | None = None, use_unique_params: bool = False, *, trainable_indices: frozenset[int] | None = None, **run_kwargs) \rightarrow Circuit
```

Instantiate Braket Circuit object.

54 Chapter 2. Tutorials

batch_execute(circuits, **run_kwargs)

Execute a batch of quantum circuits on the device.

The circuits are represented by tapes, and they are executed one-by-one using the device's execute method. The results are collected in a list.

For plugin developers: This function should be overwritten if the device can efficiently run multiple circuits on a backend, for example using parallel and/or asynchronous executions.

Parameters

circuits (list[QuantumTape]) - circuits to execute on the device

Returns

list of measured value(s)

Return type

list[array[float]]

batch_transform(circuit: QuantumTape)

Apply a differentiable batch transform for preprocessing a circuit prior to execution. This method is called directly by the QNode, and should be overwritten if the device requires a transform that generates multiple circuits prior to execution.

By default, this method contains logic for generating multiple circuits, one per term, of a circuit that terminates in expval(H), if the underlying device does not support Hamiltonian expectation values, or if the device requires finite shots.

Warning: This method will be tracked by autodifferentiation libraries, such as Autograd, JAX, Tensor-Flow, and Torch. Please make sure to use qml.math for autodiff-agnostic tensor processing if required.

Parameters

circuit (.QuantumTape) - the circuit to preprocess

Returns

Returns a tuple containing the sequence of circuits to be executed, and a post-processing function to be applied to the list of evaluated circuit results.

Return type

tuple[Sequence[.QuantumTape], callable]

capabilities()

Add support for AG on sv1

check_validity(queue, observables)

Checks whether the operations and observables in queue are all supported by the device.

Parameters

- **queue** (*Iterable[Operation]*) quantum operation objects which are intended to be applied on the device
- **observables** (*Iterable[Observable]*) observables which are intended to be evaluated on the device

Raises

DeviceError – if there are operations in the queue or observables that the device does not support

classical_shadow(obs, circuit)

Returns the measured bits and recipes in the classical shadow protocol.

The protocol is described in detail in the classical shadows paper. This measurement process returns the randomized Pauli measurements (the recipes) that are performed for each qubit and snapshot as an integer:

- 0 for Pauli X,
- · 1 for Pauli Y, and
- 2 for Pauli Z.

It also returns the measurement results (the bits); 0 if the 1 eigenvalue is sampled, and 1 if the -1 eigenvalue is sampled.

The device shots are used to specify the number of snapshots. If T is the number of shots and n is the number of qubits, then both the measured bits and the Pauli measurements have shape (T, n).

This implementation is device-agnostic and works by executing single-shot tapes containing randomized Pauli observables. Devices should override this if they can offer cleaner or faster implementations.

See also:

classical_shadow()

Parameters

- **obs** (*ClassicalShadowMP*) The classical shadow measurement process
- **circuit** (*QuantumTape*) The quantum tape that is being executed

Returns

A tensor with shape (2, T, n), where the first row represents the measured bits and the second represents the recipes used.

Return type

tensor_like[int]

custom_expand(fn)

Register a custom expansion function for the device.

Example

```
dev = qml.device("default.qubit", wires=2)

@dev.custom_expand
def my_expansion_function(self, tape, max_expansion=10):
    ...
    # can optionally call the default device expansion
    tape = self.default_expand_fn(tape, max_expansion=max_expansion)
    return tape
```

The custom device expansion function must have arguments self (the device object), tape (the input circuit to transform and execute), and max_expansion (the number of times the circuit should be expanded).

The default *default_expand_fn()* method of the original device may be called. It is highly recommended to call this before returning, to ensure that the expanded circuit is supported on the device.

default_expand_fn(circuit, max_expansion=10)

Method for expanding or decomposing an input circuit. This method should be overwritten if custom expansion logic is required.

By default, this method expands the tape if:

- state preparation operations are called mid-circuit,
- · nested tapes are present,
- · any operations are not supported on the device, or
- multiple observables are measured on the same wire.

Parameters

- **circuit** (. QuantumTape) the circuit to expand.
- max_expansion (int) The number of times the circuit should be expanded. Expansion
 occurs when an operation or measurement is not supported, and results in a gate decomposition. If any operations in the decomposition remain unsupported by the device, another
 expansion occurs.

Returns

The expanded/decomposed circuit, such that the device will natively support all operations.

Return type

.QuantumTape

define_wire_map(wires)

Create the map from user-provided wire labels to the wire labels used by the device.

The default wire map maps the user wire labels to wire labels that are consecutive integers.

However, by overwriting this function, devices can specify their preferred, non-consecutive and/or non-integer wire labels.

Parameters

wires (Wires) - user-provided wires for this device

Returns

dictionary specifying the wire map

Return type

OrderedDict

Example

density_matrix(wires)

Returns the reduced density matrix over the given wires.

Parameters

wires (Wires) – wires of the reduced system

Doturna

complex array of shape (2 ** len(wires), 2 ** len(wires)) representing the reduced density matrix of the state prior to measurement.

Return type

array[complex]

estimate_probability(wires=None, shot_range=None, bin_size=None)

Return the estimated probability of each computational basis state using the generated samples.

Parameters

- wires (Iterable[Number, str], Number, str, Wires) wires to calculate marginal probabilities for. Wires not provided are traced out of the system.
- **shot_range** (tuple[int]) 2-tuple of integers specifying the range of samples to use. If not specified, all samples are used.
- bin_size (int) Divides the shot range into bins of size bin_size, and returns the measurement statistic separately over each bin. If not provided, the entire shot range is treated as a single bin.

Returns

list of the probabilities

Return type

array[float]

execute($circuit: QuantumTape, compute_gradient=False, **run_kwargs) <math>\rightarrow$ ndarray

It executes a queue of quantum operations on the device and then measure the given observables.

For plugin developers: instead of overwriting this, consider implementing a suitable subset of

- apply()
- generate_samples()
- probability()

Additional keyword arguments may be passed to this method that can be utilised by apply(). An example would be passing the QNode hash that can be used later for parametric compilation.

Parameters

circuit (QuantumTape) - circuit to execute on the device

Raises

QuantumFunctionError – if the value of return_type is not supported

Returns

measured value(s)

Return type

array[float]

execute_and_gradients(circuits, **kwargs)

Execute a list of circuits and calculate their gradients. Returns a list of circuit results and a list of gradients/jacobians, one of each for each circuit in circuits.

The gradient is returned as a list of floats, 1 float for every instance of a trainable parameter in a gate in the circuit. Functions like qml.grad or qml.jacobian then use that format to generate a per-parameter format.

execution_context()

The device execution context used during calls to execute().

You can overwrite this function to return a context manager in case your quantum library requires that; all operations and method calls (including *apply()* and *expval()*) are then evaluated within the context of this context manager (see the source of *execute()* for more details).

expand_fn(*circuit*, *max expansion=10*)

Method for expanding or decomposing an input circuit. Can be the default or a custom expansion method, see Device.default_expand_fn() and Device.custom_expand() for more details.

Parameters

- **circuit** (.QuantumTape) the circuit to expand.
- max_expansion (int) The number of times the circuit should be expanded. Expansion occurs when an operation or measurement is not supported, and results in a gate decomposition. If any operations in the decomposition remain unsupported by the device, another expansion occurs.

Returns

The expanded/decomposed circuit, such that the device will natively support all operations.

Return type

.QuantumTape

expval(observable, shot_range=None, bin_size=None)

Returns the expectation value of observable on specified wires.

Note: all arguments accept _lists_, which indicate a tensor product of observables.

Parameters

- **observable** (*str or list[str]*) name of the observable(s)
- wires (Wires) wires the observable(s) are to be measured on
- par (tuple or list[tuple]]) parameters for the observable(s)

Returns

expectation value $A = \psi A \psi$

Return type

float

static generate_basis_states(num_wires, dtype=<class 'numpy.uint32'>)

Generates basis states in binary representation according to the number of wires specified.

The states_to_binary method creates basis states faster (for larger systems at times over x25 times faster) than the approach using itertools.product, at the expense of using slightly more memory.

Due to the large size of the integer arrays for more than 32 bits, memory allocation errors may arise in the states_to_binary method. Hence we constraint the dtype of the array to represent unsigned integers on 32 bits. Due to this constraint, an overflow occurs for 32 or more wires, therefore this approach is used only for fewer wires.

For smaller number of wires speed is comparable to the next approach (using itertools.product), hence we resort to that one for testing purposes.

Parameters

- **num_wires** (*int*) the number wires
- **dtype=np.uint32** (*type*) the data type of the arrays to use

Returns

the sampled basis states

Return type

array[int]

generate_samples()

Returns the computational basis samples generated for all wires.

Note that PennyLane uses the convention $|q_0, q_1, \dots, q_{N-1}\rangle$ where q_0 is the most significant bit.

Warning: This method should be overwritten on devices that generate their own computational basis samples, with the resulting computational basis samples stored as self._samples.

Returns

array of samples in the shape (dev.shots, dev.num_wires)

Return type

array[complex]

gradients(circuits, method='jacobian', **kwargs)

Return the gradients of a batch of quantum circuits on the device.

The gradient method method is called sequentially for each circuit, and the corresponding Jacobians are collected in a list.

For plugin developers: This method should be overwritten if the device can efficiently compute the gradient of multiple circuits on a backend, for example using parallel and/or asynchronous executions.

Parameters

- circuits (list[.tape.QuantumTape]) circuits to execute on the device
- **method** (*str*) the device method to call to compute the Jacobian of a single circuit
- **kwargs keyword argument to pass when calling method

Returns

List of Jacobians. Returned Jacobians should be of shape (output_shape, num_params).

Return type

list[array[float]]

map_wires(wires)

Map the wire labels of wires using this device's wire map.

Parameters

wires (Wires) – wires whose labels we want to map to the device's internal labelling scheme

Returns

wires with new labels

Return type

Wires

marginal_prob(prob, wires=None)

Return the marginal probability of the computational basis states by summing the probabilities on the non-specified wires.

If no wires are specified, then all the basis states representable by the device are considered and no marginalization takes place.

Note: If the provided wires are not in the order as they appear on the device, the returned marginal probabilities take this permutation into account.

For example, if the addressable wires on this device are Wires([0, 1, 2]) and this function gets passed wires=[2, 0], then the returned marginal probability vector will take this 'reversal' of the two wires into account:

$$\mathbb{P}^{(2,0)} = [|00\rangle, |10\rangle, |01\rangle, |11\rangle]$$

Parameters

- prob The probabilities to return the marginal probabilities for
- wires (Iterable[Number, str], Number, str, Wires) wires to return marginal probabilities for. Wires not provided are traced out of the system.

Returns

array of the resulting marginal probabilities.

Return type

array[float]

mutual_info(wires0, wires1, log_base)

Returns the mutual information prior to measurement:

$$I(A,B) = S(\rho^A) + S(\rho^B) - S(\rho^{AB})$$

where S is the von Neumann entropy.

Parameters

- wires (Wires) wires of the first subsystem
- wires1 (Wires) wires of the second subsystem
- log_base (float) base to use in the logarithm

Returns

the mutual information

Return type

float

order_wires(subset_wires)

Given some subset of device wires return a Wires object with the same wires; sorted according to the device wire map.

Parameters

subset_wires (*Wires*) – The subset of device wires (in any order).

Raises

ValueError – Could not find some or all subset wires subset_wires in device wires device_wires.

Returns

a new Wires object containing the re-ordered wires set

Return type

ordered_wires (Wires)

post_apply()

Called during *execute()* after the individual operations have been executed.

post_measure()

Called during execute() after the individual observables have been measured.

pre_apply()

Called during *execute()* before the individual operations are executed.

pre_measure()

Called during *execute()* before the individual observables are measured.

```
probability(wires=None, shot range=None, bin size=None)
```

Return either the analytic probability or estimated probability of each computational basis state.

Devices that require a finite number of shots always return the estimated probability.

Parameters

wires (Iterable[Number, str], Number, str, Wires) — wires to return marginal probabilities for. Wires not provided are traced out of the system.

Returns

list of the probabilities

Return type

array[float]

reset()

Reset the backend state.

After the reset, the backend should be as if it was just constructed. Most importantly the quantum state is reset to its initial value.

sample(observable, shot_range=None, bin_size=None, counts=False)

Return samples of an observable.

Parameters

- **observable** (*Observable*) the observable to sample
- **shot_range** (tuple[int]) 2-tuple of integers specifying the range of samples to use. If not specified, all samples are used.
- bin_size (int) Divides the shot range into bins of size bin_size, and returns the measurement statistic separately over each bin. If not provided, the entire shot range is treated as a single bin.
- counts (bool) whether counts (True) or raw samples (False) should be returned

Raises

 $\textbf{EigvalsUndefinedError} - if \ no \ information \ is \ available \ about \ the \ eigenvalues \ of \ the \ observable$

Returns

samples in an array of dimension (shots,) or counts

Return type

Union[array[float], dict, list[dict]]

sample_basis_states(number_of_states, state_probability)

Sample from the computational basis states based on the state probability.

This is an auxiliary method to the generate_samples method.

Parameters

62

- **number_of_states** (*int*) the number of basis states to sample from
- **state_probability** (array[float]) the computational basis probability vector

Returns

the sampled basis states

Return type

array[int]

shadow_expval(obs, circuit)

Compute expectation values using classical shadows in a differentiable manner.

Please refer to shadow_expval() for detailed documentation.

Parameters

- obs (ClassicalShadowMP) The classical shadow expectation value measurement process
- circuit (QuantumTape) The quantum tape that is being executed

Returns

expectation value estimate.

Return type

float

shot_vec_statistics(circuit: QuantumTape)

Process measurement results from circuit execution using a device with a shot vector and return statistics.

This is an auxiliary method of execute and uses statistics.

When using shot vectors, measurement results for each item of the shot vector are contained in a tuple.

Parameters

circuit (QuantumTape) - circuit to execute on the device

Raises

QuantumFunctionError – if the value of return_type is not supported

Returns

stastics for each shot item from the shot vector

Return type

tuple

static states_to_binary(samples, num_wires, dtype=<class 'numpy.int64'>)

Convert basis states from base 10 to binary representation.

This is an auxiliary method to the generate_samples method.

Parameters

- **samples** (array[int]) samples of basis states in base 10 representation
- $num_wires(int)$ the number of qubits
- **dtype** (*type*) Type of the internal integer array to be used. Can be important to specify for large systems for memory allocation purposes.

Returns

basis states in binary representation

Return type

array[int]

 $\textbf{statistics}(braket_result:\ GateModelQuantumTaskResult,\ observables:\ Sequence[Observable]) \rightarrow \\list[float]$

Processes measurement results from a Braket task result and returns statistics.

Parameters

- braket_result (GateModelQuantumTaskResult) the Braket task result
- **observables** (*list[Observable]*) the observables to be measured

Raises

QuantumFunctionError – if the value of return_type is not supported

Returns

the corresponding statistics

Return type

list[float]

supports_observable(observable)

Checks if an observable is supported by this device. Raises a ValueError,

if not a subclass or string of an Observable was passed.

Parameters

observable (*type or str*) – observable to be checked

Raises

ValueError – if *observable* is not a Observable class or string

Returns

True iff supplied observable is supported

Return type

bool

supports_operation(operation)

Checks if an operation is supported by this device.

Parameters

```
operation (type or str) – operation to be checked
```

Raises

ValueError – if *operation* is not a Operation class or string

Returns

True if supplied operation is supported

Return type

bool

var(observable, shot_range=None, bin_size=None)

Returns the variance of observable on specified wires.

Note: all arguments support _lists_, which indicate a tensor product of observables.

Parameters

- **observable** (str or list[str]) name of the observable(s)
- wires (Wires) wires the observable(s) is to be measured on
- par (tuple or list[tuple]]) parameters for the observable(s)

Raises

NotImplementedError – if the device does not support variance computation

Returns

variance
$$var(A) = \psi A^2 \psi - \psi A \psi^2$$

Return type

float

vn_entropy(wires, log base)

Returns the Von Neumann entropy prior to measurement.

$$S(\rho) = -\text{Tr}(\rho \log(\rho))$$

Parameters

- wires (Wires) Wires of the considered subsystem.
- **log_base** (*float*) Base for the logarithm, default is None the natural logarithm is used in this case.

Returns

returns the Von Neumann entropy

Return type

float

BraketLocalAhsDevice

class BraketLocalAhsDevice(wires: int | Iterable, *, shots: int | Shots = Shots.DEFAULT)

Bases: BraketAhsDevice

Amazon Braket LocalSimulator AHS device for PennyLane.

Runs programs on Braket's local AHS simulator. Can be used to emulate the BraketAwsAhsDevice.

Parameters

- wires (int or Iterable[int, str]) Number of subsystems represented by the device, or iterable that contains unique labels for the subsystems as numbers (i.e., [-1, 0, 2]) or strings (['ancilla', 'q1', 'q2']).
- **shots** (*int or Shots.DEFAULT*) Number of executions to run to aquire measurements. Default: Shots.DEFAULT

Note: It is important to keep track of units when specifying electromagnetic pulses for hardware control. The frequency and amplitude provided in PennyLane for Rydberg atom systems are expected to be in units of MHz, time in microseconds, phase in radians, and distance in micrometers. All of these will be converted to SI units internally as needed for upload to the hardware, and frequency will be converted to angular frequency (multiplied by 2π).

When reading hardware specifications from the Braket backend, bear in mind that all units are SI and frequencies are in rad/s. This conversion is done when creating a pulse program for upload, and units in the PennyLane functions should follow the conventions specified in the PennyLane docs to ensure correct unit conversion. See rydberg_interaction and rydberg_drive in Pennylane for specification of expected input units, and examples for creating hardware compatible ParametrizedEvolution operators in PennyLane.

ahs_program	
analytic	Whether shots is None or not.
author	
circuit_hash	The hash of the circuit upon the last execution.
measurement_map	Mapping used to override the logic of measurement
·	processes.
name	•
num_executions	Number of times this device is executed by the eval-
	uation of QNodes running on this device
obs_queue	The observables to be measured and returned.
observables	The observacion to be incusared and returned.
ODDEL VADLED	
op_queue	The operation queue to be applied.
operations	1 1
parameters	Mapping from free parameter index to the list of
F-11-11-11-11-11-11-11-11-11-11-11-11-11	Operations in the device queue that depend on it.
pennylane_requires	operations in the device queue that depend on its
perilly 14110_10441100	
register	Register a virtual subclass of an ABC.
result	
settings	Dictionary of constants set by the hardware.
short_name	
shot_vector	Returns the shot vector, a sparse representation of the
	shot sequence used by the device when evaluating
	QNodes.
shots	Number of circuit evaluations/random samples used
3110 C3	to estimate expectation values of observables
state	Returns the state vector of the circuit prior to mea-
State	
stonning condition	Surement. Paturns the stopping condition for the device
stopping_condition	Returns the stopping condition for the device.
task	
uancian	
version	
wire_map	Ordered dictionary that defines the map from user-
wit c_map	•
	provided wire labels to the wire labels used on this device
wires	All wires that can be addressed on this device
MITCS	All whee that can be addressed on this device

ahs_program

analytic

Whether shots is None or not. Kept for backwards compatability.

author = 'Xanadu Inc.'

66 Chapter 2. Tutorials

circuit_hash

The hash of the circuit upon the last execution.

This can be used by devices in *apply()* for parametric compilation.

```
measurement_map = {}
```

Mapping used to override the logic of measurement processes. The dictionary maps a measurement class to a string containing the name of a device's method that overrides the measurement process. The method defined by the device should have the following arguments:

- measurement (MeasurementProcess): measurement to override
- shot_range (tuple[int]): 2-tuple of integers specifying the range of samples to use. If not specified, all samples are used.
- bin_size (int): Divides the shot range into bins of size bin_size, and returns the measurement statistic separately over each bin. If not provided, the entire shot range is treated as a single bin.

Note: When overriding the logic of a MeasurementTransform, the method defined by the device should only have a single argument:

• tape: quantum tape to transform

Example:

Let's create device that inherits from DefaultQubit and overrides the logic of the *qml.sample* measurement. To do so we will need to update the measurement_map dictionary:

```
class NewDevice(DefaultQubit):
    def __init__(self, wires, shots):
        super().__init__(wires=wires, shots=shots)
        self.measurement_map[SampleMP] = "sample_measurement"

def sample_measurement(self, measurement, shot_range=None, bin_size=None):
    return 2
```

```
>>> dev = NewDevice(wires=2, shots=1000)
>>> @qml.qnode(dev)
... def circuit():
... return qml.sample()
>>> circuit()
tensor(2, requires_grad=True)
```

name = 'Braket LocalSimulator for AHS in PennyLane'

num_executions

Number of times this device is executed by the evaluation of QNodes running on this device

Returns

number of executions

Return type

int

obs_queue The observables to be measured and returned. Note that this property can only be accessed within the execution context of execute(). Raises **ValueError** – if outside of the execution context Returns list[~.operation.Observable] observables = {'Hadamard', 'Hermitian', 'Identity', 'PauliX', 'PauliY', 'PauliZ', 'Prod', 'Projector', 'Sprod', 'Sum'} op_queue The operation queue to be applied. Note that this property can only be accessed within the execution context of execute(). Raises **ValueError** – if outside of the execution context **Returns** list[~.operation.Operation] operations = {'ParametrizedEvolution'} parameters Mapping from free parameter index to the list of Operations in the device queue that depend on it. Note that this property can only be accessed within the execution context of execute(). Raises **ValueError** – if outside of the execution context Returns the mapping Return type dict[int->list[ParameterDependency]] pennylane_requires = '>=0.30.0'

register

result

settinas

Dictionary of constants set by the hardware.

Used to enable initializing hardware-consistent Hamiltonians by saving all the values that would need to be passed, i.e.:

```
>>> dev_remote = qml.device('braket.aws.ahs', wires=3)
>>> dev_pl = qml.device('default.qubit', wires=3)
>>> settings = dev_remote.settings
>>> H_int = qml.pulse.rydberg.rydberg_interaction(coordinates, **settings)
```

By passing the settings from the remote device to rydberg_interaction, an H_int Hamiltonian term is created using the constants specific to the hardware. This is relevant for simulating the remote device in PennyLane on the default.qubit device.

short_name = 'braket.local.ahs'

shot_vector

Returns the shot vector, a sparse representation of the shot sequence used by the device when evaluating QNodes.

Example

The sparse representation of the shot sequence is returned, where tuples indicate the number of times a shot integer is repeated.

Type

list[ShotCopies]

shots

Number of circuit evaluations/random samples used to estimate expectation values of observables

state

Returns the state vector of the circuit prior to measurement.

Note: Only state vector simulators support this property. Please see the plugin documentation for more details.

stopping_condition

Returns the stopping condition for the device. The returned function accepts a queuable object (including a PennyLane operation and observable) and returns True if supported by the device.

Type

.BooleanFn

task

```
version = '0.32.0'
```

wire_map

Ordered dictionary that defines the map from user-provided wire labels to the wire labels used on this device

wires

All wires that can be addressed on this device

access_state([wires])	Check that the device has access to an internal state
	and return it if available.
active_wires(operators)	Returns the wires acted on by a set of operators.
<pre>adjoint_jacobian(tape[, starting_state,])</pre>	Implements the adjoint method outlined in Jones and Gacon to differentiate an input tape.
<pre>analytic_probability([wires])</pre>	Return the (marginal) probability of each computational basis state from the last run of the device.
apply(operations, **kwargs)	Convert the pulse operation to an AHS program and run on the connected device
batch_execute(circuits)	Execute a batch of quantum circuits on the device.
batch_transform(circuit)	Apply a differentiable batch transform for preprocessing a circuit prior to execution.
capabilities()	Get the capabilities of this device class.
<pre>check_validity(queue, observables)</pre>	Checks whether the operations and observables in queue are all supported by the device.
classical_shadow(obs, circuit)	Returns the measured bits and recipes in the classical shadow protocol.
<pre>create_ahs_program(evolution)</pre>	Create AHS program for upload to hardware from a ParametrizedEvolution
custom_expand(fn)	Register a custom expansion function for the device.
<pre>default_expand_fn(circuit[, max_expansion])</pre>	Method for expanding or decomposing an input circuit.
<pre>define_wire_map(wires)</pre>	Create the map from user-provided wire labels to the wire labels used by the device.
<pre>density_matrix(wires)</pre>	Returns the reduced density matrix over the given wires.
<pre>estimate_probability([wires, shot_range,])</pre>	Return the estimated probability of each computational basis state using the generated samples.
execute(circuit, **kwargs)	It executes a queue of quantum operations on the device and then measure the given observables.
<pre>execute_and_gradients(circuits[, method])</pre>	Execute a batch of quantum circuits on the device, and return both the results and the gradients.
<pre>execution_context()</pre>	The device execution context used during calls to execute().
<pre>expand_fn(circuit[, max_expansion])</pre>	Method for expanding or decomposing an input circuit.
<pre>expval(observable[, shot_range, bin_size])</pre>	Returns the expectation value of observable on specified wires.
<pre>generate_basis_states(num_wires[, dtype])</pre>	Generates basis states in binary representation according to the number of wires specified.
<pre>generate_samples()</pre>	Returns the computational basis samples measured for all wires.
<pre>gradients(circuits[, method])</pre>	Return the gradients of a batch of quantum circuits on the device.
<pre>map_wires(wires)</pre>	Map the wire labels of wires using this device's wire map.
<pre>marginal_prob(prob[, wires])</pre>	Return the marginal probability of the computational basis states by summing the probabilities on the non-specified wires.
<pre>mutual_info(wires0, wires1, log_base)</pre>	Returns the mutual information prior to measurement:
	continues on next page

continues on next page

70 Chapter 2. Tutorials

Table 3 – continued from previous page

order_wires(subset_wires)	Given some subset of device wires return a Wires object with the same wires; sorted according to the device wire map.
post_apply()	Called during <i>execute()</i> after the individual operations have been executed.
<pre>post_measure()</pre>	Called during <i>execute()</i> after the individual observables have been measured.
<pre>pre_apply()</pre>	Called during <i>execute()</i> before the individual operations are executed.
<pre>pre_measure()</pre>	Called during <i>execute()</i> before the individual observables are measured.
<pre>probability([wires, shot_range, bin_size])</pre>	Return either the analytic probability or estimated probability of each computational basis state.
reset()	Reset the backend state.
<pre>sample(observable[, shot_range, bin_size,])</pre>	Return samples of an observable.
<pre>sample_basis_states(number_of_states,)</pre>	Sample from the computational basis states based on the state probability.
shadow_expval(obs, circuit)	Compute expectation values using classical shadows in a differentiable manner.
<pre>shot_vec_statistics(circuit)</pre>	Process measurement results from circuit execution using a device with a shot vector and return statistics.
<pre>states_to_binary(samples, num_wires[, dtype])</pre>	Convert basis states from base 10 to binary representation.
<pre>statistics(circuit[, shot_range, bin_size])</pre>	Process measurement results from circuit execution and return statistics.
<pre>supports_observable(observable)</pre>	Checks if an observable is supported by this device. Raises a ValueError,
<pre>supports_operation(operation)</pre>	Checks if an operation is supported by this device.
<pre>var(observable[, shot_range, bin_size])</pre>	Returns the variance of observable on specified wires.
vn_entropy(wires, log_base)	Returns the Von Neumann entropy prior to measurement.

access_state(wires=None)

Check that the device has access to an internal state and return it if available.

Parameters

wires (Wires) – wires of the reduced system

Raises

QuantumFunctionError – if the device is not capable of returning the state

Returns

the state or the density matrix of the device

Return type

array or tensor

static active_wires(operators)

Returns the wires acted on by a set of operators.

Parameters

operators (list[Operation]) – operators for which we are gathering the active wires

Returns

wires activated by the specified operators

Return type

Wires

adjoint_jacobian(tape: QuantumTape, starting_state=None, use_device_state=False)

Implements the adjoint method outlined in Jones and Gacon to differentiate an input tape.

After a forward pass, the circuit is reversed by iteratively applying adjoint gates to scan backwards through the circuit.

Note: The adjoint differentiation method has the following restrictions:

- As it requires knowledge of the statevector, only statevector simulator devices can be used.
- Only expectation values are supported as measurements.
- Does not work for parametrized observables like Hamiltonian or Hermitian.

Parameters

tape (.QuantumTape) - circuit that the function takes the gradient of

Keyword Arguments

- **starting_state** (*tensor_like*) post-forward pass state to start execution with. It should be complex-valued. Takes precedence over use_device_state.
- **use_device_state** (*bool*) use current device state to initialize. A forward pass of the same circuit should be the last thing the device has executed. If a **starting_state** is provided, that takes precedence.

Returns

the derivative of the tape with respect to trainable parameters. Dimensions are (len(observables), len(trainable_params)).

Return type

array or tuple[array]

Raises

QuantumFunctionError – if the input tape has measurements that are not expectation values or contains a multi-parameter operation aside from Rot

analytic_probability(wires=None)

Return the (marginal) probability of each computational basis state from the last run of the device.

PennyLane uses the convention $|q_0, q_1, \dots, q_{N-1}\rangle$ where q_0 is the most significant bit.

If no wires are specified, then all the basis states representable by the device are considered and no marginalization takes place.

Note: marginal_prob() may be used as a utility method to calculate the marginal probability distribution.

Parameters

wires (Iterable[Number, str], Number, str, Wires) — wires to return marginal probabilities for. Wires not provided are traced out of the system.

Returns

list of the probabilities

Return type

array[float]

apply(operations: list[ParametrizedEvolution], **kwargs)

Convert the pulse operation to an AHS program and run on the connected device

Parameters

 $\begin{tabular}{ll} \textbf{operations} & (\textit{list[ParametrizedEvolution]}) & - & a & list & containing & a & single \\ \textbf{ParametrizedEvolution operator} & & & & \\ \end{tabular}$

batch_execute(circuits)

Execute a batch of quantum circuits on the device.

The circuits are represented by tapes, and they are executed one-by-one using the device's execute method. The results are collected in a list.

For plugin developers: This function should be overwritten if the device can efficiently run multiple circuits on a backend, for example using parallel and/or asynchronous executions.

Parameters

circuits (list[QuantumTape]) - circuits to execute on the device

Returns

list of measured value(s)

Return type

list[array[float]]

batch_transform(circuit: QuantumTape)

Apply a differentiable batch transform for preprocessing a circuit prior to execution. This method is called directly by the QNode, and should be overwritten if the device requires a transform that generates multiple circuits prior to execution.

By default, this method contains logic for generating multiple circuits, one per term, of a circuit that terminates in expval(H), if the underlying device does not support Hamiltonian expectation values, or if the device requires finite shots.

Warning: This method will be tracked by autodifferentiation libraries, such as Autograd, JAX, Tensor-Flow, and Torch. Please make sure to use qml.math for autodiff-agnostic tensor processing if required.

Parameters

circuit (.QuantumTape) - the circuit to preprocess

Returns

Returns a tuple containing the sequence of circuits to be executed, and a post-processing function to be applied to the list of evaluated circuit results.

Return type

tuple[Sequence[.QuantumTape], callable]

classmethod capabilities()

Get the capabilities of this device class.

Inheriting classes that change or add capabilities must override this method, for example via

```
@classmethod
def capabilities(cls):
    capabilities = super().capabilities().copy()
    capabilities.update(
        supports_a_new_capability=True,
    )
    return capabilities
```

Returns

results

Return type

dict[str->*]

check_validity(queue, observables)

Checks whether the operations and observables in queue are all supported by the device.

Parameters

- **queue** (*Iterable[Operation]*) quantum operation objects which are intended to be applied on the device
- observables (Iterable[Observable]) observables which are intended to be evaluated on the device

Raises

Exception – if there are operations in the queue or observables that the device does not support

classical_shadow(obs, circuit)

Returns the measured bits and recipes in the classical shadow protocol.

The protocol is described in detail in the classical shadows paper. This measurement process returns the randomized Pauli measurements (the recipes) that are performed for each qubit and snapshot as an integer:

- 0 for Pauli X,
- · 1 for Pauli Y, and
- 2 for Pauli Z.

It also returns the measurement results (the bits); 0 if the 1 eigenvalue is sampled, and 1 if the -1 eigenvalue is sampled.

The device shots are used to specify the number of snapshots. If T is the number of shots and n is the number of qubits, then both the measured bits and the Pauli measurements have shape (T, n).

This implementation is device-agnostic and works by executing single-shot tapes containing randomized Pauli observables. Devices should override this if they can offer cleaner or faster implementations.

See also:

classical_shadow()

Parameters

- **obs** (*ClassicalShadowMP*) The classical shadow measurement process
- circuit (QuantumTape) The quantum tape that is being executed

Returns

A tensor with shape (2, T, n), where the first row represents the measured bits and the second represents the recipes used.

Return type

tensor_like[int]

create_ahs_program(evolution: ParametrizedEvolution)

Create AHS program for upload to hardware from a ParametrizedEvolution

Parameters

evolution (*ParametrizedEvolution*) – the PennyLane operator describing the pulse to be converted into an AnalogHamiltonianSimulation program

Returns

a program containing the register and drive

information for running an AHS task on simulation or hardware

Return type

AnalogHamiltonianSimulation

custom_expand(fn)

Register a custom expansion function for the device.

Example

```
dev = qml.device("default.qubit", wires=2)

@dev.custom_expand
def my_expansion_function(self, tape, max_expansion=10):
    ...
    # can optionally call the default device expansion
    tape = self.default_expand_fn(tape, max_expansion=max_expansion)
    return tape
```

The custom device expansion function must have arguments self (the device object), tape (the input circuit to transform and execute), and max_expansion (the number of times the circuit should be expanded).

The default *default_expand_fn()* method of the original device may be called. It is highly recommended to call this before returning, to ensure that the expanded circuit is supported on the device.

default_expand_fn(circuit, max expansion=10)

Method for expanding or decomposing an input circuit. This method should be overwritten if custom expansion logic is required.

By default, this method expands the tape if:

- state preparation operations are called mid-circuit,
- nested tapes are present,
- · any operations are not supported on the device, or
- multiple observables are measured on the same wire.

Parameters

• **circuit** (.QuantumTape) – the circuit to expand.

• max_expansion (int) – The number of times the circuit should be expanded. Expansion occurs when an operation or measurement is not supported, and results in a gate decomposition. If any operations in the decomposition remain unsupported by the device, another expansion occurs.

Returns

The expanded/decomposed circuit, such that the device will natively support all operations.

Return type

.QuantumTape

define_wire_map(wires)

Create the map from user-provided wire labels to the wire labels used by the device.

The default wire map maps the user wire labels to wire labels that are consecutive integers.

However, by overwriting this function, devices can specify their preferred, non-consecutive and/or non-integer wire labels.

Parameters

wires (Wires) – user-provided wires for this device

Returns

dictionary specifying the wire map

Return type

OrderedDict

Example

density_matrix(wires)

Returns the reduced density matrix over the given wires.

Parameters

wires (Wires) – wires of the reduced system

Returns

complex array of shape (2 ** len(wires), 2 ** len(wires)) representing the reduced density matrix of the state prior to measurement.

Return type

array[complex]

estimate_probability(wires=None, shot_range=None, bin_size=None)

Return the estimated probability of each computational basis state using the generated samples.

Parameters

- wires (Iterable[Number, str], Number, str, Wires) wires to calculate marginal probabilities for. Wires not provided are traced out of the system.
- **shot_range** (tuple[int]) 2-tuple of integers specifying the range of samples to use. If not specified, all samples are used.
- bin_size (int) Divides the shot range into bins of size bin_size, and returns the measurement statistic separately over each bin. If not provided, the entire shot range is treated as a single bin.

Returns

list of the probabilities

Return type

array[float]

execute(circuit, **kwargs)

It executes a queue of quantum operations on the device and then measure the given observables.

For plugin developers: instead of overwriting this, consider implementing a suitable subset of

- apply()
- generate_samples()
- probability()

Additional keyword arguments may be passed to this method that can be utilised by apply(). An example would be passing the QNode hash that can be used later for parametric compilation.

Parameters

circuit (QuantumTape) - circuit to execute on the device

Raises

QuantumFunctionError – if the value of return_type is not supported

Returns

measured value(s)

Return type

array[float]

execute_and_gradients(circuits, method='jacobian', **kwargs)

Execute a batch of quantum circuits on the device, and return both the results and the gradients.

The circuits are represented by tapes, and they are executed one-by-one using the device's execute method. The results and the corresponding Jacobians are collected in a list.

For plugin developers: This method should be overwritten if the device can efficiently run multiple circuits on a backend, for example using parallel and/or asynchronous executions, and return both the results and the Jacobians.

Parameters

- **circuits** (list[.tape.QuantumTape]) circuits to execute on the device
- method (str) the device method to call to compute the Jacobian of a single circuit
- **kwargs keyword argument to pass when calling method

Returns

Tuple containing list of measured value(s) and list of Jacobians. Returned Jacobians should be of shape (output_shape, num_params).

Return type

tuple[list[array[float]], list[array[float]]]

execution_context()

The device execution context used during calls to execute().

You can overwrite this function to return a context manager in case your quantum library requires that; all operations and method calls (including *apply()* and *expval()*) are then evaluated within the context of this context manager (see the source of *execute()* for more details).

```
expand_fn(circuit, max expansion=10)
```

Method for expanding or decomposing an input circuit. Can be the default or a custom expansion method, see Device.default_expand_fn() and Device.custom_expand() for more details.

Parameters

- **circuit** (.QuantumTape) the circuit to expand.
- max_expansion (int) The number of times the circuit should be expanded. Expansion
 occurs when an operation or measurement is not supported, and results in a gate decomposition. If any operations in the decomposition remain unsupported by the device, another
 expansion occurs.

Returns

The expanded/decomposed circuit, such that the device will natively support all operations.

Return type

.QuantumTape

```
expval(observable, shot_range=None, bin_size=None)
```

Returns the expectation value of observable on specified wires.

Note: all arguments accept _lists_, which indicate a tensor product of observables.

Parameters

- **observable** (*str or list[str]*) name of the observable(s)
- wires (Wires) wires the observable(s) are to be measured on
- par (tuple or list[tuple]]) parameters for the observable(s)

Returns

expectation value $A = \psi A \psi$

Return type

float

static generate_basis_states(num_wires, dtype=<class 'numpy.uint32'>)

Generates basis states in binary representation according to the number of wires specified.

The states_to_binary method creates basis states faster (for larger systems at times over x25 times faster) than the approach using itertools.product, at the expense of using slightly more memory.

Due to the large size of the integer arrays for more than 32 bits, memory allocation errors may arise in the states_to_binary method. Hence we constraint the dtype of the array to represent unsigned integers on 32 bits. Due to this constraint, an overflow occurs for 32 or more wires, therefore this approach is used only for fewer wires.

For smaller number of wires speed is comparable to the next approach (using itertools.product), hence we resort to that one for testing purposes.

Parameters

- **num_wires** (*int*) the number wires
- **dtype=np.uint32** (*type*) the data type of the arrays to use

Returns

the sampled basis states

Return type

array[int]

generate_samples()

Returns the computational basis samples measured for all wires.

Returns

array of samples in the shape (dev.shots, dev.num_wires)

Return type

array[complex]

gradients(circuits, method='jacobian', **kwargs)

Return the gradients of a batch of quantum circuits on the device.

The gradient method method is called sequentially for each circuit, and the corresponding Jacobians are collected in a list.

For plugin developers: This method should be overwritten if the device can efficiently compute the gradient of multiple circuits on a backend, for example using parallel and/or asynchronous executions.

Parameters

- **circuits** (list[.tape.QuantumTape]) circuits to execute on the device
- **method** (str) the device method to call to compute the Jacobian of a single circuit
- **kwargs keyword argument to pass when calling method

Returns

List of Jacobians. Returned Jacobians should be of shape (output_shape, num_params).

Return type

list[array[float]]

map_wires(wires)

Map the wire labels of wires using this device's wire map.

Parameters

wires (Wires) – wires whose labels we want to map to the device's internal labelling scheme

Returns

wires with new labels

Return type

Wires

marginal_prob(prob, wires=None)

Return the marginal probability of the computational basis states by summing the probabilities on the non-specified wires.

If no wires are specified, then all the basis states representable by the device are considered and no marginalization takes place.

Note: If the provided wires are not in the order as they appear on the device, the returned marginal probabilities take this permutation into account.

For example, if the addressable wires on this device are Wires([0, 1, 2]) and this function gets passed wires=[2, 0], then the returned marginal probability vector will take this 'reversal' of the two wires into account:

$$\mathbb{P}^{(2,0)} = [|00\rangle, |10\rangle, |01\rangle, |11\rangle]$$

Parameters

- prob The probabilities to return the marginal probabilities for
- wires (Iterable[Number, str], Number, str, Wires) wires to return marginal probabilities for. Wires not provided are traced out of the system.

Returns

array of the resulting marginal probabilities.

Return type

array[float]

mutual_info(wires0, wires1, log_base)

Returns the mutual information prior to measurement:

$$I(A,B) = S(\rho^A) + S(\rho^B) - S(\rho^{AB})$$

where S is the von Neumann entropy.

Parameters

- wires0 (Wires) wires of the first subsystem
- wires1 (Wires) wires of the second subsystem
- log_base (float) base to use in the logarithm

Returns

the mutual information

Return type

float

order_wires(subset_wires)

Given some subset of device wires return a Wires object with the same wires; sorted according to the device wire map.

Parameters

subset_wires (*Wires*) – The subset of device wires (in any order).

Raises

ValueError – Could not find some or all subset wires subset_wires in device wires device_wires.

Returns

a new Wires object containing the re-ordered wires set

Return type

ordered_wires (Wires)

post_apply()

Called during <code>execute()</code> after the individual operations have been executed.

post_measure()

Called during *execute()* after the individual observables have been measured.

pre_apply()

Called during execute() before the individual operations are executed.

pre_measure()

80

Called during execute() before the individual observables are measured.

probability(wires=None, shot_range=None, bin_size=None)

Return either the analytic probability or estimated probability of each computational basis state.

Devices that require a finite number of shots always return the estimated probability.

Parameters

wires (Iterable[Number, str], Number, str, Wires) — wires to return marginal probabilities for. Wires not provided are traced out of the system.

Returns

list of the probabilities

Return type

array[float]

reset()

Reset the backend state.

After the reset, the backend should be as if it was just constructed. Most importantly the quantum state is reset to its initial value.

sample(observable, shot_range=None, bin_size=None, counts=False)

Return samples of an observable.

Parameters

- **observable** (*Observable*) the observable to sample
- **shot_range** (tuple[int]) 2-tuple of integers specifying the range of samples to use. If not specified, all samples are used.
- bin_size (int) Divides the shot range into bins of size bin_size, and returns the measurement statistic separately over each bin. If not provided, the entire shot range is treated as a single bin.
- counts (bool) whether counts (True) or raw samples (False) should be returned

Raises

EigvalsUndefinedError – if no information is available about the eigenvalues of the observable

Returns

samples in an array of dimension (shots,) or counts

Return type

Union[array[float], dict, list[dict]]

sample_basis_states(number_of_states, state_probability)

Sample from the computational basis states based on the state probability.

This is an auxiliary method to the generate_samples method.

Parameters

- number_of_states (int) the number of basis states to sample from
- **state_probability** (array[float]) the computational basis probability vector

Returns

the sampled basis states

Return type

array[int]

shadow_expval(obs, circuit)

Compute expectation values using classical shadows in a differentiable manner.

Please refer to shadow_expval() for detailed documentation.

Parameters

- obs (ClassicalShadowMP) The classical shadow expectation value measurement process
- circuit (QuantumTape) The quantum tape that is being executed

Returns

expectation value estimate.

Return type

float

shot_vec_statistics(circuit: QuantumTape)

Process measurement results from circuit execution using a device with a shot vector and return statistics.

This is an auxiliary method of execute and uses statistics.

When using shot vectors, measurement results for each item of the shot vector are contained in a tuple.

Parameters

circuit (QuantumTape) - circuit to execute on the device

Raises

QuantumFunctionError – if the value of return_type is not supported

Returns

stastics for each shot item from the shot vector

Return type

tuple

static states_to_binary(samples, num_wires, dtype=<class 'numpy.int64'>)

Convert basis states from base 10 to binary representation.

This is an auxiliary method to the generate_samples method.

Parameters

- **samples** (array[int]) samples of basis states in base 10 representation
- **num_wires** (*int*) the number of qubits
- **dtype** (*type*) Type of the internal integer array to be used. Can be important to specify for large systems for memory allocation purposes.

Returns

basis states in binary representation

Return type

array[int]

statistics(circuit: QuantumTape, shot_range=None, bin_size=None)

Process measurement results from circuit execution and return statistics.

This includes returning expectation values, variance, samples, probabilities, states, and density matrices.

Parameters

82

• circuit (QuantumTape) – the quantum tape currently being executed

- **shot_range** (tuple[int]) 2-tuple of integers specifying the range of samples to use. If not specified, all samples are used.
- **bin_size** (*int*) Divides the shot range into bins of size bin_size, and returns the measurement statistic separately over each bin. If not provided, the entire shot range is treated as a single bin.

Raises

QuantumFunctionError – if the value of return_type is not supported

Returns

the corresponding statistics

Return type

Union[float, List[float]]

supports_observable(observable)

Checks if an observable is supported by this device. Raises a ValueError,

if not a subclass or string of an Observable was passed.

Parameters

observable (*type or str*) – observable to be checked

Raises

ValueError – if *observable* is not a Observable class or string

Returns

True iff supplied observable is supported

Return type

bool

supports_operation(operation)

Checks if an operation is supported by this device.

Parameters

operation (type or str) – operation to be checked

Raises

ValueError – if *operation* is not a Operation class or string

Returns

True if supplied operation is supported

Return type

bool

var(observable, shot_range=None, bin_size=None)

Returns the variance of observable on specified wires.

Note: all arguments support _lists_, which indicate a tensor product of observables.

Parameters

- **observable** (*str or list[str]*) name of the observable(s)
- wires (Wires) wires the observable(s) is to be measured on
- par (tuple or list[tuple]]) parameters for the observable(s)

Raises

NotImplementedError – if the device does not support variance computation

Returns

variance
$$var(A) = \psi A^2 \psi - \psi A \psi^2$$

Return type

float

vn_entropy(wires, log_base)

Returns the Von Neumann entropy prior to measurement.

$$S(\rho) = -\text{Tr}(\rho \log(\rho))$$

Parameters

- wires (Wires) Wires of the considered subsystem.
- **log_base** (*float*) Base for the logarithm, default is None the natural logarithm is used in this case.

Returns

returns the Von Neumann entropy

Return type

float

BraketLocalQubitDevice

class BraketLocalQubitDevice(wires: int | Iterable, backend: str | BraketSimulator = 'default', *, shots: int | None = None, **run_kwargs)

Bases: BraketQubitDevice

Amazon Braket LocalSimulator qubit device for PennyLane.

Parameters

- wires (int or Iterable[Number, str]]) Number of subsystems represented by the device, or iterable that contains unique labels for the subsystems as numbers (i.e., [-1, 0, 2]) or strings (['ancilla', 'q1', 'q2']).
- backend (Union[str, BraketSimulator]) The name of the simulator backend or the actual simulator instance to use for simulation. Defaults to the default simulator backend name.
- **shots** (*int or None*) Number of circuit evaluations or random samples included, to estimate expectation values of observables. If this value is set to None or 0, then the device runs in analytic mode (calculations will be exact). Default: None
- **run_kwargs Variable length keyword arguments for braket.devices.Device.run().

analytic	Whether shots is None or not.
author	
circuit	The last circuit run on this device.
circuit_hash	The hash of the circuit upon the last execution.
measurement_map	Mapping used to override the logic of measurement processes.
name	
num_executions	Number of times this device is executed by the eval- uation of QNodes running on this device
obs_queue	The observables to be measured and returned.
observables	<pre>set() -> new empty set object set(iterable) -> new set object</pre>
op_queue	The operation queue to be applied.
operations	The set of names of PennyLane operations that the device supports.
parameters	Mapping from free parameter index to the list of Operations in the device queue that depend on it.
pennylane_requires	
short_name	
shot_vector	Returns the shot vector, a sparse representation of the shot sequence used by the device when evaluating QNodes.
shots	Number of circuit evaluations/random samples used to estimate expectation values of observables
state	Returns the state vector of the circuit prior to measurement.
stopping_condition	Returns the stopping condition for the device.
task	The task corresponding to the last run circuit.
version	
wire_map	Ordered dictionary that defines the map from user- provided wire labels to the wire labels used on this device
wires	All wires that can be addressed on this device

analytic

Whether shots is None or not. Kept for backwards compatability.

author = 'Amazon Web Services'

circuit

The last circuit run on this device.

Type Circuit

circuit_hash

The hash of the circuit upon the last execution.

This can be used by devices in apply() for parametric compilation.

measurement_map = {}

Mapping used to override the logic of measurement processes. The dictionary maps a measurement class to a string containing the name of a device's method that overrides the measurement process. The method defined by the device should have the following arguments:

- measurement (MeasurementProcess): measurement to override
- shot_range (tuple[int]): 2-tuple of integers specifying the range of samples to use. If not specified, all samples are used.
- bin_size (int): Divides the shot range into bins of size bin_size, and returns the measurement statistic separately over each bin. If not provided, the entire shot range is treated as a single bin.

Note: When overriding the logic of a MeasurementTransform, the method defined by the device should only have a single argument:

• tape: quantum tape to transform

Example:

Let's create device that inherits from DefaultQubit and overrides the logic of the *qml.sample* measurement. To do so we will need to update the measurement_map dictionary:

```
class NewDevice(DefaultQubit):
    def __init__(self, wires, shots):
        super().__init__(wires=wires, shots=shots)
        self.measurement_map[SampleMP] = "sample_measurement"

def sample_measurement(self, measurement, shot_range=None, bin_size=None):
        return 2
```

```
>>> dev = NewDevice(wires=2, shots=1000)
>>> @qml.qnode(dev)
... def circuit():
... return qml.sample()
>>> circuit()
tensor(2, requires_grad=True)
```

name = 'Braket LocalSimulator for PennyLane'

num executions

Number of times this device is executed by the evaluation of QNodes running on this device

Returns

number of executions

Return type

int

obs_queue

The observables to be measured and returned.

Note that this property can only be accessed within the execution context of execute().

Raises

ValueError - if outside of the execution context

Returns

list[~.operation.Observable]

observables

op_queue

The operation queue to be applied.

Note that this property can only be accessed within the execution context of execute().

Raises

ValueError – if outside of the execution context

Returns

list[~.operation.Operation]

operations

The set of names of PennyLane operations that the device supports.

Type

frozenset[str]

parameters

Mapping from free parameter index to the list of Operations in the device queue that depend on it.

Note that this property can only be accessed within the execution context of execute().

Raises

ValueError – if outside of the execution context

Returns

the mapping

Return type

dict[int->list[ParameterDependency]]

```
pennylane_requires = '>=0.30.0'
```

```
short_name = 'braket.local.qubit'
```

shot_vector

Returns the shot vector, a sparse representation of the shot sequence used by the device when evaluating QNodes.

Example

The sparse representation of the shot sequence is returned, where tuples indicate the number of times a shot integer is repeated.

Type

list[ShotCopies]

shots

Number of circuit evaluations/random samples used to estimate expectation values of observables

state

Returns the state vector of the circuit prior to measurement.

Note: Only state vector simulators support this property. Please see the plugin documentation for more details.

stopping_condition

Returns the stopping condition for the device. The returned function accepts a queuable object (including a PennyLane operation and observable) and returns True if supported by the device.

Type

.BooleanFn

task

The task corresponding to the last run circuit.

Type

QuantumTask

version = '1.22.1.dev0'

wire_map

Ordered dictionary that defines the map from user-provided wire labels to the wire labels used on this device

wires

All wires that can be addressed on this device

access_state([wires])	Check that the device has access to an internal state and return it if available.
<pre>active_wires(operators)</pre>	Returns the wires acted on by a set of operators.
<pre>adjoint_jacobian(tape[, starting_state,])</pre>	Implements the adjoint method outlined in Jones and Gacon to differentiate an input tape.
<pre>analytic_probability([wires])</pre>	Return the (marginal) probability of each computational basis state from the last run of the device.
<pre>apply(operations[, rotations,])</pre>	Instantiate Braket Circuit object.
batch_execute(circuits)	Execute a batch of quantum circuits on the device.
<pre>batch_transform(circuit)</pre>	Apply a differentiable batch transform for preprocessing a circuit prior to execution.
<pre>capabilities()</pre>	Get the capabilities of this device class.
<pre>check_validity(queue, observables)</pre>	Checks whether the operations and observables in queue are all supported by the device.
classical_shadow(obs, circuit)	Returns the measured bits and recipes in the classical shadow protocol.
custom_expand(fn)	Register a custom expansion function for the device.

continues on next page

Table 4 – continued from previous page

Table 4 – continued from previous page	
<pre>default_expand_fn(circuit[, max_expansion])</pre>	Method for expanding or decomposing an input circuit.
<pre>define_wire_map(wires)</pre>	Create the map from user-provided wire labels to the wire labels used by the device.
<pre>density_matrix(wires)</pre>	Returns the reduced density matrix over the given wires.
<pre>estimate_probability([wires, shot_range,])</pre>	Return the estimated probability of each computational basis state using the generated samples.
<pre>execute(circuit[, compute_gradient])</pre>	It executes a queue of quantum operations on the device and then measure the given observables.
<pre>execute_and_gradients(circuits[, method])</pre>	Execute a batch of quantum circuits on the device, and return both the results and the gradients.
<pre>execution_context()</pre>	The device execution context used during calls to execute().
<pre>expand_fn(circuit[, max_expansion])</pre>	Method for expanding or decomposing an input circuit.
<pre>expval(observable[, shot_range, bin_size])</pre>	Returns the expectation value of observable on specified wires.
<pre>generate_basis_states(num_wires[, dtype])</pre>	Generates basis states in binary representation according to the number of wires specified.
<pre>generate_samples()</pre>	Returns the computational basis samples generated for all wires.
<pre>gradients(circuits[, method])</pre>	Return the gradients of a batch of quantum circuits on the device.
map_wires(wires)	Map the wire labels of wires using this device's wire map.
marginal_prob(prob[, wires])	Return the marginal probability of the computational basis states by summing the probabilities on the non-specified wires.
<pre>mutual_info(wires0, wires1, log_base)</pre>	Returns the mutual information prior to measurement:
order_wires(subset_wires)	Given some subset of device wires return a Wires object with the same wires; sorted according to the device wire map.
<pre>post_apply()</pre>	Called during <i>execute()</i> after the individual operations have been executed.
<pre>post_measure()</pre>	Called during <i>execute()</i> after the individual observables have been measured.
<pre>pre_apply()</pre>	Called during <i>execute()</i> before the individual operations are executed.
<pre>pre_measure()</pre>	Called during <i>execute()</i> before the individual observables are measured.
<pre>probability([wires, shot_range, bin_size])</pre>	Return either the analytic probability or estimated probability of each computational basis state.
<pre>reset() sample(observable[, shot_range, bin_size,])</pre>	Reset the backend state. Return samples of an observable.
sample_basis_states(number_of_states,)	Sample from the computational basis states based on the state probability.
shadow_expval(obs, circuit)	Compute expectation values using classical shadows in a differentiable manner.
<pre>shot_vec_statistics(circuit)</pre>	Process measurement results from circuit execution using a device with a shot vector and return statistics.
	continues on next page

continues on next page

Table 4 – continued from previous page

<pre>states_to_binary(samples, num_wires[, dtype])</pre>	Convert basis states from base 10 to binary representation.
statistics(braket_result, observables)	Processes measurement results from a Braket task result and returns statistics.
<pre>supports_observable(observable)</pre>	Checks if an observable is supported by this device. Raises a ValueError,
<pre>supports_operation(operation)</pre>	Checks if an operation is supported by this device.
<pre>var(observable[, shot_range, bin_size])</pre>	Returns the variance of observable on specified wires.
<pre>vn_entropy(wires, log_base)</pre>	Returns the Von Neumann entropy prior to measurement.

access_state(wires=None)

Check that the device has access to an internal state and return it if available.

Parameters

wires (Wires) – wires of the reduced system

Raises

QuantumFunctionError – if the device is not capable of returning the state

Returns

the state or the density matrix of the device

Return type

array or tensor

static active_wires(operators)

Returns the wires acted on by a set of operators.

Parameters

operators (list[Operation]) – operators for which we are gathering the active wires

Returns

wires activated by the specified operators

Return type

Wires

adjoint_jacobian(tape: QuantumTape, starting_state=None, use_device_state=False)

Implements the adjoint method outlined in Jones and Gacon to differentiate an input tape.

After a forward pass, the circuit is reversed by iteratively applying adjoint gates to scan backwards through the circuit.

Note: The adjoint differentiation method has the following restrictions:

- As it requires knowledge of the statevector, only statevector simulator devices can be used.
- Only expectation values are supported as measurements.
- Does not work for parametrized observables like Hamiltonian or Hermitian.

Parameters

tape (.QuantumTape) – circuit that the function takes the gradient of

Keyword Arguments

- **starting_state** (*tensor_like*) post-forward pass state to start execution with. It should be complex-valued. Takes precedence over use_device_state.
- use_device_state (bool) use current device state to initialize. A forward pass of the same circuit should be the last thing the device has executed. If a starting_state is provided, that takes precedence.

Returns

the derivative of the tape with respect to trainable parameters. Dimensions are (len(observables), len(trainable_params)).

Return type

array or tuple[array]

Raises

QuantumFunctionError – if the input tape has measurements that are not expectation values or contains a multi-parameter operation aside from Rot

analytic_probability(wires=None)

Return the (marginal) probability of each computational basis state from the last run of the device.

PennyLane uses the convention $|q_0, q_1, \dots, q_{N-1}\rangle$ where q_0 is the most significant bit.

If no wires are specified, then all the basis states representable by the device are considered and no marginalization takes place.

Note: marginal_prob() may be used as a utility method to calculate the marginal probability distribution.

Parameters

wires (Iterable[Number, str], Number, str, Wires) — wires to return marginal probabilities for. Wires not provided are traced out of the system.

Returns

list of the probabilities

Return type

array[float]

```
apply(operations: Sequence[Operation], rotations: Sequence[Operation] | None = None, use_unique_params: bool = False, *, trainable_indices: frozenset[int] | None = None, **run_kwargs) \rightarrow Circuit
```

Instantiate Braket Circuit object.

batch_execute(circuits)

Execute a batch of quantum circuits on the device.

The circuits are represented by tapes, and they are executed one-by-one using the device's execute method. The results are collected in a list.

For plugin developers: This function should be overwritten if the device can efficiently run multiple circuits on a backend, for example using parallel and/or asynchronous executions.

Parameters

circuits (list [QuantumTape]) – circuits to execute on the device

Returns

list of measured value(s)

Return type

list[array[float]]

batch_transform(circuit: QuantumTape)

Apply a differentiable batch transform for preprocessing a circuit prior to execution. This method is called directly by the QNode, and should be overwritten if the device requires a transform that generates multiple circuits prior to execution.

By default, this method contains logic for generating multiple circuits, one per term, of a circuit that terminates in expval(H), if the underlying device does not support Hamiltonian expectation values, or if the device requires finite shots.

Warning: This method will be tracked by autodifferentiation libraries, such as Autograd, JAX, Tensor-Flow, and Torch. Please make sure to use qml.math for autodiff-agnostic tensor processing if required.

Parameters

```
circuit (.QuantumTape) - the circuit to preprocess
```

Returns

Returns a tuple containing the sequence of circuits to be executed, and a post-processing function to be applied to the list of evaluated circuit results.

Return type

tuple[Sequence[.QuantumTape], callable]

classmethod capabilities()

Get the capabilities of this device class.

Inheriting classes that change or add capabilities must override this method, for example via

```
@classmethod
def capabilities(cls):
    capabilities = super().capabilities().copy()
    capabilities.update(
        supports_a_new_capability=True,
    )
    return capabilities
```

Returns

results

Return type

dict[str->*]

check_validity(queue, observables)

Checks whether the operations and observables in queue are all supported by the device.

Parameters

- **queue** (*Iterable[Operation]*) quantum operation objects which are intended to be applied on the device
- **observables** (*Iterable[Observable]*) observables which are intended to be evaluated on the device

Raises

DeviceError – if there are operations in the queue or observables that the device does not support

classical_shadow(obs, circuit)

Returns the measured bits and recipes in the classical shadow protocol.

The protocol is described in detail in the classical shadows paper. This measurement process returns the randomized Pauli measurements (the recipes) that are performed for each qubit and snapshot as an integer:

- 0 for Pauli X.
- · 1 for Pauli Y, and
- 2 for Pauli Z.

It also returns the measurement results (the bits); 0 if the 1 eigenvalue is sampled, and 1 if the -1 eigenvalue is sampled.

The device shots are used to specify the number of snapshots. If T is the number of shots and n is the number of qubits, then both the measured bits and the Pauli measurements have shape (T, n).

This implementation is device-agnostic and works by executing single-shot tapes containing randomized Pauli observables. Devices should override this if they can offer cleaner or faster implementations.

See also:

classical_shadow()

Parameters

- **obs** (ClassicalShadowMP) The classical shadow measurement process
- **circuit** (*QuantumTape*) The quantum tape that is being executed

Returns

A tensor with shape (2, T, n), where the first row represents the measured bits and the second represents the recipes used.

Return type

tensor_like[int]

custom_expand(fn)

Register a custom expansion function for the device.

Example

```
dev = qml.device("default.qubit", wires=2)

@dev.custom_expand
def my_expansion_function(self, tape, max_expansion=10):
    ...
    # can optionally call the default device expansion
    tape = self.default_expand_fn(tape, max_expansion=max_expansion)
    return tape
```

The custom device expansion function must have arguments self (the device object), tape (the input circuit to transform and execute), and max_expansion (the number of times the circuit should be expanded).

The default <code>default_expand_fn()</code> method of the original device may be called. It is highly recommended to call this before returning, to ensure that the expanded circuit is supported on the device.

default_expand_fn(*circuit*, *max_expansion=10*)

Method for expanding or decomposing an input circuit. This method should be overwritten if custom expansion logic is required.

By default, this method expands the tape if:

- state preparation operations are called mid-circuit,
- · nested tapes are present,
- any operations are not supported on the device, or
- multiple observables are measured on the same wire.

Parameters

- **circuit** (.QuantumTape) the circuit to expand.
- max_expansion (int) The number of times the circuit should be expanded. Expansion
 occurs when an operation or measurement is not supported, and results in a gate decomposition. If any operations in the decomposition remain unsupported by the device, another
 expansion occurs.

Returns

The expanded/decomposed circuit, such that the device will natively support all operations.

Return type

.QuantumTape

define_wire_map(wires)

Create the map from user-provided wire labels to the wire labels used by the device.

The default wire map maps the user wire labels to wire labels that are consecutive integers.

However, by overwriting this function, devices can specify their preferred, non-consecutive and/or non-integer wire labels.

Parameters

wires (Wires) – user-provided wires for this device

Returns

dictionary specifying the wire map

Return type

OrderedDict

Example

```
>>> dev = device('my.device', wires=['b', 'a'])
>>> dev.wire_map()
OrderedDict( [(<Wires = ['a']>, <Wires = [0]>), (<Wires = ['b']>, <Wires = [1]>

--)])
```

density_matrix(wires)

Returns the reduced density matrix over the given wires.

Parameters

wires (Wires) – wires of the reduced system

Returns

complex array of shape (2 ** len(wires), 2 ** len(wires)) representing the reduced density matrix of the state prior to measurement.

Return type

array[complex]

estimate_probability(wires=None, shot_range=None, bin_size=None)

Return the estimated probability of each computational basis state using the generated samples.

Parameters

- wires (Iterable[Number, str], Number, str, Wires) wires to calculate marginal probabilities for. Wires not provided are traced out of the system.
- **shot_range** (tuple[int]) 2-tuple of integers specifying the range of samples to use. If not specified, all samples are used.
- **bin_size** (*int*) Divides the shot range into bins of size bin_size, and returns the measurement statistic separately over each bin. If not provided, the entire shot range is treated as a single bin.

Returns

list of the probabilities

Return type

array[float]

 $execute(circuit: QuantumTape, compute_gradient=False, **run_kwargs) \rightarrow ndarray$

It executes a queue of quantum operations on the device and then measure the given observables.

For plugin developers: instead of overwriting this, consider implementing a suitable subset of

- apply()
- generate_samples()
- probability()

Additional keyword arguments may be passed to this method that can be utilised by apply(). An example would be passing the QNode hash that can be used later for parametric compilation.

Parameters

circuit (QuantumTape) - circuit to execute on the device

Raises

QuantumFunctionError – if the value of return_type is not supported

Returns

measured value(s)

Return type

array[float]

execute_and_gradients(circuits, method='jacobian', **kwargs)

Execute a batch of quantum circuits on the device, and return both the results and the gradients.

The circuits are represented by tapes, and they are executed one-by-one using the device's execute method. The results and the corresponding Jacobians are collected in a list.

For plugin developers: This method should be overwritten if the device can efficiently run multiple circuits on a backend, for example using parallel and/or asynchronous executions, and return both the results and the Jacobians.

Parameters

- circuits (list[.tape.QuantumTape]) circuits to execute on the device
- **method** (str) the device method to call to compute the Jacobian of a single circuit

• **kwargs – keyword argument to pass when calling method

Returns

Tuple containing list of measured value(s) and list of Jacobians. Returned Jacobians should be of shape (output_shape, num_params).

Return type

tuple[list[array[float]], list[array[float]]]

execution_context()

The device execution context used during calls to execute().

You can overwrite this function to return a context manager in case your quantum library requires that; all operations and method calls (including *apply()* and *expval()*) are then evaluated within the context of this context manager (see the source of *execute()* for more details).

expand_fn(circuit, max_expansion=10)

Method for expanding or decomposing an input circuit. Can be the default or a custom expansion method, see Device.default_expand_fn() and Device.custom_expand() for more details.

Parameters

- **circuit** (.QuantumTape) the circuit to expand.
- max_expansion (int) The number of times the circuit should be expanded. Expansion occurs when an operation or measurement is not supported, and results in a gate decomposition. If any operations in the decomposition remain unsupported by the device, another expansion occurs.

Returns

The expanded/decomposed circuit, such that the device will natively support all operations.

Return type

.QuantumTape

```
expval(observable, shot_range=None, bin_size=None)
```

Returns the expectation value of observable on specified wires.

Note: all arguments accept _lists_, which indicate a tensor product of observables.

Parameters

- **observable** (*str or list[str]*) name of the observable(s)
- wires (Wires) wires the observable(s) are to be measured on
- par (tuple or list[tuple]]) parameters for the observable(s)

Returns

expectation value $A = \psi A \psi$

Return type

float

static generate_basis_states(num_wires, dtype=<class 'numpy.uint32'>)

Generates basis states in binary representation according to the number of wires specified.

The states_to_binary method creates basis states faster (for larger systems at times over x25 times faster) than the approach using itertools.product, at the expense of using slightly more memory.

Due to the large size of the integer arrays for more than 32 bits, memory allocation errors may arise in the states_to_binary method. Hence we constraint the dtype of the array to represent unsigned integers on 32 bits. Due to this constraint, an overflow occurs for 32 or more wires, therefore this approach is used only for fewer wires.

For smaller number of wires speed is comparable to the next approach (using itertools.product), hence we resort to that one for testing purposes.

Parameters

- num_wires (int) the number wires
- **dtype=np.uint32** (*type*) the data type of the arrays to use

Returns

the sampled basis states

Return type

array[int]

generate_samples()

Returns the computational basis samples generated for all wires.

Note that PennyLane uses the convention $|q_0, q_1, \dots, q_{N-1}\rangle$ where q_0 is the most significant bit.

Warning: This method should be overwritten on devices that generate their own computational basis samples, with the resulting computational basis samples stored as self._samples.

Returns

array of samples in the shape (dev.shots, dev.num_wires)

Return type

array[complex]

```
gradients(circuits, method='jacobian', **kwargs)
```

Return the gradients of a batch of quantum circuits on the device.

The gradient method method is called sequentially for each circuit, and the corresponding Jacobians are collected in a list.

For plugin developers: This method should be overwritten if the device can efficiently compute the gradient of multiple circuits on a backend, for example using parallel and/or asynchronous executions.

Parameters

- **circuits** (list[.tape.QuantumTape]) circuits to execute on the device
- method(str) the device method to call to compute the Jacobian of a single circuit
- **kwargs keyword argument to pass when calling method

Returns

List of Jacobians. Returned Jacobians should be of shape (output_shape, num_params).

Return type

list[array[float]]

map_wires(wires)

Map the wire labels of wires using this device's wire map.

Parameters

wires (Wires) – wires whose labels we want to map to the device's internal labelling scheme

Returns

wires with new labels

Return type

Wires

marginal_prob(prob, wires=None)

Return the marginal probability of the computational basis states by summing the probabilities on the non-specified wires.

If no wires are specified, then all the basis states representable by the device are considered and no marginalization takes place.

Note: If the provided wires are not in the order as they appear on the device, the returned marginal probabilities take this permutation into account.

For example, if the addressable wires on this device are Wires([0, 1, 2]) and this function gets passed wires=[2, 0], then the returned marginal probability vector will take this 'reversal' of the two wires into account:

$$\mathbb{P}^{(2,0)} = [|00\rangle, |10\rangle, |01\rangle, |11\rangle]$$

Parameters

- prob The probabilities to return the marginal probabilities for
- wires (Iterable[Number, str], Number, str, Wires) wires to return marginal probabilities for. Wires not provided are traced out of the system.

Returns

array of the resulting marginal probabilities.

Return type

array[float]

mutual_info(wires0, wires1, log_base)

Returns the mutual information prior to measurement:

$$I(A,B) = S(\rho^A) + S(\rho^B) - S(\rho^{AB})$$

where S is the von Neumann entropy.

Parameters

- wires0 (Wires) wires of the first subsystem
- wires1 (Wires) wires of the second subsystem
- log_base (float) base to use in the logarithm

Returns

the mutual information

Return type

float

order_wires(subset_wires)

Given some subset of device wires return a Wires object with the same wires; sorted according to the device wire map.

Parameters

subset_wires (*Wires*) – The subset of device wires (in any order).

Raises

ValueError – Could not find some or all subset wires subset_wires in device wires device_wires.

Returns

a new Wires object containing the re-ordered wires set

Return type

ordered_wires (Wires)

post_apply()

Called during execute() after the individual operations have been executed.

post_measure()

Called during *execute()* after the individual observables have been measured.

pre_apply()

Called during *execute()* before the individual operations are executed.

pre_measure()

Called during *execute()* before the individual observables are measured.

```
probability(wires=None, shot_range=None, bin_size=None)
```

Return either the analytic probability or estimated probability of each computational basis state.

Devices that require a finite number of shots always return the estimated probability.

Parameters

wires (Iterable[Number, str], Number, str, Wires) — wires to return marginal probabilities for. Wires not provided are traced out of the system.

Returns

list of the probabilities

Return type

array[float]

reset()

Reset the backend state.

After the reset, the backend should be as if it was just constructed. Most importantly the quantum state is reset to its initial value.

```
{\tt sample}(observable, shot\_range=None, bin\_size=None, counts=False)
```

Return samples of an observable.

Parameters

- **observable** (*Observable*) the observable to sample
- **shot_range** (tuple[int]) 2-tuple of integers specifying the range of samples to use. If not specified, all samples are used.
- **bin_size** (*int*) Divides the shot range into bins of size bin_size, and returns the measurement statistic separately over each bin. If not provided, the entire shot range is treated as a single bin.
- counts (bool) whether counts (True) or raw samples (False) should be returned

Raises

EigvalsUndefinedError – if no information is available about the eigenvalues of the observable

Returns

samples in an array of dimension (shots,) or counts

Return type

Union[array[float], dict, list[dict]]

sample_basis_states(number_of_states, state_probability)

Sample from the computational basis states based on the state probability.

This is an auxiliary method to the generate samples method.

Parameters

- **number_of_states** (*int*) the number of basis states to sample from
- **state_probability** (array[float]) the computational basis probability vector

Returns

the sampled basis states

Return type

array[int]

shadow_expval(obs, circuit)

Compute expectation values using classical shadows in a differentiable manner.

Please refer to shadow_expval() for detailed documentation.

Parameters

- obs (ClassicalShadowMP) The classical shadow expectation value measurement process
- circuit (QuantumTape) The quantum tape that is being executed

Returns

expectation value estimate.

Return type

float

shot_vec_statistics(circuit: QuantumTape)

Process measurement results from circuit execution using a device with a shot vector and return statistics.

This is an auxiliary method of execute and uses statistics.

When using shot vectors, measurement results for each item of the shot vector are contained in a tuple.

Parameters

circuit (QuantumTape) - circuit to execute on the device

Raises

QuantumFunctionError – if the value of return_type is not supported

Returns

stastics for each shot item from the shot vector

Return type

tuple

static states_to_binary(samples, num_wires, dtype=<class 'numpy.int64'>)

Convert basis states from base 10 to binary representation.

This is an auxiliary method to the generate_samples method.

Parameters

- samples (array[int]) samples of basis states in base 10 representation
- **num_wires** (int) the number of qubits
- **dtype** (*type*) Type of the internal integer array to be used. Can be important to specify for large systems for memory allocation purposes.

Returns

basis states in binary representation

Return type

array[int]

 $\textbf{statistics}(braket_result:\ GateModelQuantumTaskResult,\ observables:\ Sequence[Observable]) \rightarrow \\list[float]$

Processes measurement results from a Braket task result and returns statistics.

Parameters

- braket_result (GateModelQuantumTaskResult) the Braket task result
- **observables** (*list[Observable]*) the observables to be measured

Raises

QuantumFunctionError – if the value of return_type is not supported

Returns

the corresponding statistics

Return type

list[float]

supports_observable(observable)

Checks if an observable is supported by this device. Raises a ValueError,

if not a subclass or string of an Observable was passed.

Parameters

```
observable (type or str) – observable to be checked
```

Raises

ValueError – if *observable* is not a Observable class or string

Returns

True iff supplied observable is supported

Return type

bool

supports_operation(operation)

Checks if an operation is supported by this device.

Parameters

```
operation (type or str) – operation to be checked
```

Raises

ValueError – if *operation* is not a Operation class or string

Returns

True if supplied operation is supported

Return type

bool

var(observable, shot_range=None, bin_size=None)

Returns the variance of observable on specified wires.

Note: all arguments support _lists_, which indicate a tensor product of observables.

Parameters

- **observable** (*str or list[str]*) name of the observable(s)
- wires (Wires) wires the observable(s) is to be measured on
- par (tuple or list[tuple]]) parameters for the observable(s)

Raises

NotImplementedError – if the device does not support variance computation

Returns

variance
$$var(A) = \psi A^2 \psi - \psi A \psi^2$$

Return type

float

vn_entropy(wires, log_base)

Returns the Von Neumann entropy prior to measurement.

$$S(\rho) = -\text{Tr}(\rho \log(\rho))$$

Parameters

- wires (Wires) Wires of the considered subsystem.
- **log_base** (*float*) Base for the logarithm, default is None the natural logarithm is used in this case.

Returns

returns the Von Neumann entropy

Return type

float

CPhaseShift00

class CPhaseShift00(phi, wires)

Bases: Operation

Controlled phase shift gate phasing the $|00\rangle$ state.

$$\mathtt{CPhaseShift00}(\phi) = \begin{bmatrix} e^{i\phi} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Details:

- Number of wires: 2
- Number of parameters: 1
- Gradient recipe:

$$\frac{d}{d\phi} \texttt{CPhaseShift00}(\phi) = \frac{1}{2} \left[\texttt{CPhaseShift00}(\phi + \pi/2) - \texttt{CPhaseShift00}(\phi - \pi/2) \right]$$

Parameters

- **phi** (*float*) the controlled phase angle
- wires (int) the subsystem the gate acts on
- **id** (*str*, *optional*) String representing the operation. Default: None

arithmetic_depth	Arithmetic depth of the operator.
basis	The basis of an operation, or for controlled gates, of the target operation.
batch_size	Batch size of the operator if it is used with broad- casted parameters.
control_wires	Control wires of the operator.
grad_method	
grad_recipe	Gradient recipe for the parameter-shift method.
has_adjoint	
has_decomposition	
has_diagonalizing_gates	
has_generator	
has_matrix	
hash	Integer hash that uniquely represents the operator.
hyperparameters	Dictionary of non-trainable variables that this operation depends on.
id	Custom string to label a specific operator instance.
is_hermitian	This property determines if an operator is hermitian.
name	String for the name of the operator.
ndim_params	Number of dimensions per trainable parameter of the operator.
num_params	
num_wires	Number of wires the operator acts on.
parameter_frequencies	
parameters	Trainable parameters that the operator depends on.
wires	Wires that the operator acts on.

arithmetic_depth

Arithmetic depth of the operator.

basis = None

The basis of an operation, or for controlled gates, of the target operation. If not None, should take a value of "X", "Y", or "Z".

For example, X and CNOT have basis = "X", whereas ControlledPhaseShift and RZ have basis = "Z".

Type

str or None

batch_size

Batch size of the operator if it is used with broadcasted parameters.

The batch_size is determined based on ndim_params and the provided parameters for the operator. If (some of) the latter have an additional dimension, and this dimension has the same size for all parameters, its size is the batch size of the operator. If no parameter has an additional dimension, the batch size is None.

Returns

Size of the parameter broadcasting dimension if present, else None.

Return type

int or None

control_wires

Control wires of the operator.

For operations that are not controlled, this is an empty Wires object of length 0.

Returns

The control wires of the operation.

Return type

Wires

grad_method = 'A'

Gradient recipe for the parameter-shift method.

This is a tuple with one nested list per operation parameter. For parameter ϕ_k , the nested list contains elements of the form $[c_i, a_i, s_i]$ where i is the index of the term, resulting in a gradient recipe of

$$\frac{\partial}{\partial \phi_k} f = \sum_i c_i f(a_i \phi_k + s_i).$$

If None, the default gradient recipe containing the two terms $[c_0, a_0, s_0] = [1/2, 1, \pi/2]$ and $[c_1, a_1, s_1] = [-1/2, 1, -\pi/2]$ is assumed for every parameter.

Type

tuple(Union(list[list[float]], None)) or None

has_adjoint = True

has_decomposition = True

has_diagonalizing_gates = False

has_generator = True

has_matrix = True

hash

Integer hash that uniquely represents the operator.

Type

int

hyperparameters

Dictionary of non-trainable variables that this operation depends on.

Type

dict

id

Custom string to label a specific operator instance.

is_hermitian

This property determines if an operator is hermitian.

name

String for the name of the operator.

ndim_params

Number of dimensions per trainable parameter of the operator.

By default, this property returns the numbers of dimensions of the parameters used for the operator creation. If the parameter sizes for an operator subclass are fixed, this property can be overwritten to return the fixed value.

Returns

Number of dimensions for each trainable parameter.

Return type

tuple

num_params = 1

$num_wires = 2$

Number of wires the operator acts on.

```
parameter_frequencies = [(1,)]
```

parameters

Trainable parameters that the operator depends on.

wires

Wires that the operator acts on.

Returns

wires

Return type

Wires

adjoint()	Create an operation that is the adjoint of this one.
<pre>compute_decomposition(phi, wires)</pre>	Representation of the operator as a product of other operators (static method).
<pre>compute_diagonalizing_gates(*params, wires,)</pre>	Sequence of gates that diagonalize the operator in the computational basis (static method).
<pre>compute_eigvals(*params, **hyperparams)</pre>	Eigenvalues of the operator in the computational basis (static method).
<pre>compute_matrix(phi)</pre>	Representation of the operator as a canonical matrix in the computational basis (static method).
<pre>compute_sparse_matrix(*params, **hyper- params)</pre>	Representation of the operator as a sparse matrix in the computational basis (static method).
decomposition()	Representation of the operator as a product of other operators.
<pre>diagonalizing_gates()</pre>	Sequence of gates that diagonalize the operator in the computational basis.
eigvals()	Eigenvalues of the operator in the computational basis.
expand()	Returns a tape that contains the decomposition of the operator.
generator()	Generator of an operator that is in single-parameter- form.
<pre>label([decimals, base_label, cache])</pre>	A customizable string representation of the operator.
<pre>map_wires(wire_map)</pre>	Returns a copy of the current operator with its wires changed according to the given wire map.
<pre>matrix([wire_order])</pre>	Representation of the operator as a matrix in the computational basis.
pow(z)	A list of new operators equal to this one raised to the given power.
queue([context])	Append the operator to the Operator queue.
simplify()	Reduce the depth of nested operators to the minimum.
<pre>single_qubit_rot_angles()</pre>	The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.
<pre>sparse_matrix([wire_order])</pre>	Representation of the operator as a sparse matrix in the computational basis.
terms()	Representation of the operator as a linear combination of other operators.
validate_subspace(subspace)	Validate the subspace for qutrit operations.

adjoint()

Create an operation that is the adjoint of this one.

Adjointed operations are the conjugated and transposed version of the original operation. Adjointed ops are equivalent to the inverted operation for unitary gates.

Returns

The adjointed operation.

static compute_decomposition(phi, wires)

Representation of the operator as a product of other operators (static method).

$$O = O_1 O_2 \dots O_n$$
.

Note: Operations making up the decomposition should be queued within the compute_decomposition

106 Chapter 2. Tutorials

method.

See also:

decomposition().

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- wires (Iterable[Any], Wires) wires that the operator acts on
- **hyperparams (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

decomposition of the operator

Return type

list[Operator]

static compute_diagonalizing_gates(*params, wires, **hyperparams)

Sequence of gates that diagonalize the operator in the computational basis (static method).

Given the eigendecomposition $O=U\Sigma U^\dagger$ where Σ is a diagonal matrix containing the eigenvalues, the sequence of diagonalizing gates implements the unitary U^\dagger .

The diagonalizing gates rotate the state into the eigenbasis of the operator.

See also:

diagonalizing_gates().

Parameters

- params (list) trainable parameters of the operator, as stored in the parameters attribute
- wires (Iterable[Any], Wires) wires that the operator acts on
- **hyperparams** (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

list of diagonalizing gates

Return type

list[.Operator]

static compute_eigvals(*params, **hyperparams)

Eigenvalues of the operator in the computational basis (static method).

If diagonalizing_gates are specified and implement a unitary U^{\dagger} , the operator can be reconstructed as

$$O = U\Sigma U^{\dagger}$$
.

where Σ is the diagonal matrix containing the eigenvalues.

Otherwise, no particular order for the eigenvalues is guaranteed.

See also:

```
Operator.eigvals() and qml.eigvals()
```

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- ****hyperparams** (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

eigenvalues

Return type

tensor like

static compute_matrix(phi)

Representation of the operator as a canonical matrix in the computational basis (static method).

The canonical matrix is the textbook matrix representation that does not consider wires. Implicitly, this assumes that the wires of the operator correspond to the global wire order.

See also:

Operator.matrix() and qml.matrix()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- **hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

matrix representation

Return type

tensor_like

static compute_sparse_matrix(*params, **hyperparams)

Representation of the operator as a sparse matrix in the computational basis (static method).

The canonical matrix is the textbook matrix representation that does not consider wires. Implicitly, this assumes that the wires of the operator correspond to the global wire order.

See also:

sparse_matrix()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- **hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

sparse matrix representation

Return type

scipy.sparse._csr.csr_matrix

decomposition()

Representation of the operator as a product of other operators.

$$O = O_1 O_2 \dots O_n$$

A DecompositionUndefinedError is raised if no representation by decomposition is defined.

See also:

compute_decomposition().

Returns

decomposition of the operator

Return type

list[Operator]

diagonalizing_gates()

Sequence of gates that diagonalize the operator in the computational basis.

Given the eigendecomposition $O=U\Sigma U^{\dagger}$ where Σ is a diagonal matrix containing the eigenvalues, the sequence of diagonalizing gates implements the unitary U^{\dagger} .

The diagonalizing gates rotate the state into the eigenbasis of the operator.

A DiagGatesUndefinedError is raised if no representation by decomposition is defined.

See also:

compute_diagonalizing_gates().

Returns

a list of operators

Return type

list[.Operator] or None

eigvals()

Eigenvalues of the operator in the computational basis.

If diagonalizing_gates are specified and implement a unitary U^{\dagger} , the operator can be reconstructed as

$$O = U\Sigma U^{\dagger}$$
.

where Σ is the diagonal matrix containing the eigenvalues.

Otherwise, no particular order for the eigenvalues is guaranteed.

Note: When eigenvalues are not explicitly defined, they are computed automatically from the matrix representation. Currently, this computation is *not* differentiable.

A EigvalsUndefinedError is raised if the eigenvalues have not been defined and cannot be inferred from the matrix representation.

See also:

compute_eigvals()

eigenvalues

Return type

tensor_like

expand()

Returns a tape that contains the decomposition of the operator.

Returns

quantum tape

Return type

.QuantumTape

generator()

Generator of an operator that is in single-parameter-form.

For example, for operator

$$U(\phi) = e^{i\phi(0.5Y + Z \otimes X)}$$

we get the generator

```
>>> U.generator()
(0.5) [Y0]
+ (1.0) [Z0 X1]
```

The generator may also be provided in the form of a dense or sparse Hamiltonian (using Hermitian and SparseHamiltonian respectively).

The default value to return is None, indicating that the operation has no defined generator.

label(decimals=None, base_label=None, cache=None)

A customizable string representation of the operator.

Parameters

- **decimals=None** (*int*) If None, no parameters are included. Else, specifies how to round the parameters.
- $base_label=None$ (str) overwrite the non-parameter component of the label
- **cache=None** (*dict*) dictionary that carries information between label calls in the same drawing

Returns

label to use in drawings

Return type

str

Example:

```
>>> op = qml.RX(1.23456, wires=0)
>>> op.label()
"RX"
>>> op.label(decimals=2)
"RX\n(1.23)"
>>> op.label(base_label="my_label")
```

(continues on next page)

(continued from previous page)

```
"my_label"
>>> op.label(decimals=2, base_label="my_label")
"my_label\n(1.23)"
```

If the operation has a matrix-valued parameter and a cache dictionary is provided, unique matrices will be cached in the 'matrices' key list. The label will contain the index of the matrix in the 'matrices' list.

```
>>> op2 = qml.QubitUnitary(np.eye(2), wires=0)
>>> cache = {'matrices': []}
>>> op2.label(cache=cache)
'U(M0)'
>>> cache['matrices']
[tensor([[1., 0.],
[0., 1.]], requires_grad=True)]
>>> op3 = qml.QubitUnitary(np.eye(4), wires=(0,1))
>>> op3.label(cache=cache)
'U(M1)'
>>> cache['matrices']
[tensor([[1., 0.],
        [0., 1.]], requires_grad=True),
tensor([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]], requires_grad=True)]
```

map_wires(wire_map: dict)

Returns a copy of the current operator with its wires changed according to the given wire map.

Parameters

wire_map (dict) - dictionary containing the old wires as keys and the new wires as values

Returns

new operator

Return type

.Operator

```
matrix(wire_order=None)
```

Representation of the operator as a matrix in the computational basis.

If wire_order is provided, the numerical representation considers the position of the operator's wires in the global wire order. Otherwise, the wire order defaults to the operator's wires.

If the matrix depends on trainable parameters, the result will be cast in the same autodifferentiation framework as the parameters.

A MatrixUndefinedError is raised if the matrix representation has not been defined.

See also:

compute_matrix()

Parameters

wire_order (*Iterable*) – global wire order, must contain all wire labels from the operator's wires

Returns

matrix representation

Return type

tensor like

$pow(z) \rightarrow List[Operator]$

A list of new operators equal to this one raised to the given power.

Parameters

z (*float*) – exponent for the operator

Returns

list[Operator]

queue(context=<class 'pennylane.queuing.QueuingManager'>)

Append the operator to the Operator queue.

$$simplify() \rightarrow Operator$$

Reduce the depth of nested operators to the minimum.

Returns

simplified operator

Return type

.Operator

single_qubit_rot_angles()

The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.

Returns

A list of values $[\phi, \theta, \omega]$ such that $RZ(\omega)RY(\theta)RZ(\phi)$ is equivalent to the original operation.

Return type

tuple[float, float, float]

sparse_matrix(wire_order=None)

Representation of the operator as a sparse matrix in the computational basis.

If wire_order is provided, the numerical representation considers the position of the operator's wires in the global wire order. Otherwise, the wire order defaults to the operator's wires.

A SparseMatrixUndefinedError is raised if the sparse matrix representation has not been defined.

See also:

compute_sparse_matrix()

Parameters

wire_order (*Iterable*) – global wire order, must contain all wire labels from the operator's wires

Returns

sparse matrix representation

Return type

scipy.sparse._csr.csr_matrix

terms()

Representation of the operator as a linear combination of other operators.

$$O = \sum_{i} c_i O_i$$

A TermsUndefinedError is raised if no representation by terms is defined.

112 Chapter 2. Tutorials

list of coefficients c_i and list of operations O_i

Return type

tuple[list[tensor_like or float], list[.Operation]]

static validate_subspace(subspace)

Validate the subspace for qutrit operations.

This method determines whether a given subspace for qutrit operations is defined correctly or not. If not, a *ValueError* is thrown.

Parameters

subspace (tuple[int]) - Subspace to check for correctness

CPhaseShift01

class CPhaseShift01(phi, wires)

Bases: Operation

Controlled phase shift gate phasing the $|01\rangle$ state.

$$\mathtt{CPhaseShiftO1}(\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{i\phi} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Details:

- Number of wires: 2
- Number of parameters: 1
- Gradient recipe:

$$\frac{d}{d\phi} \texttt{CPhaseShiftO1}(\phi) = \frac{1}{2} \left[\texttt{CPhaseShiftO1}(\phi + \pi/2) - \texttt{CPhaseShiftO1}(\phi - \pi/2) \right]$$

Parameters

- **phi** (*float*) the controlled phase angle
- wires (int) the subsystem the gate acts on
- **id** (str or None) String representing the operation (optional)

arithmetic_depth	Arithmetic depth of the operator.
basis	The basis of an operation, or for controlled gates, of
	the target operation.
batch_size	Batch size of the operator if it is used with broad-
	casted parameters.
control_wires	Control wires of the operator.
<pre>grad_method</pre>	·
·	
grad_recipe	Gradient recipe for the parameter-shift method.
has_adjoint	
has_decomposition	
has_diagonalizing_gates	
has_generator	
has_matrix	
hash	Integer hash that uniquely represents the operator.
hyperparameters	Dictionary of non-trainable variables that this opera-
	tion depends on.
id	Custom string to label a specific operator instance.
is_hermitian	This property determines if an operator is hermitian.
name	String for the name of the operator.
ndim_params	Number of dimensions per trainable parameter of the
	operator.
num_params	•
num_wires	Number of wires the operator acts on.
parameter_frequencies	
parameters	Trainable parameters that the operator depends on.
wires	Wires that the operator acts on.
	*

arithmetic_depth

Arithmetic depth of the operator.

basis = None

The basis of an operation, or for controlled gates, of the target operation. If not None, should take a value of "X", "Y", or "Z".

For example, X and CNOT have basis = "X", whereas ControlledPhaseShift and RZ have basis = "Z".

Type

str or None

batch_size

Batch size of the operator if it is used with broadcasted parameters.

The batch_size is determined based on ndim_params and the provided parameters for the operator. If (some of) the latter have an additional dimension, and this dimension has the same size for all parameters, its size is the batch size of the operator. If no parameter has an additional dimension, the batch size is None.

114 Chapter 2. Tutorials

Size of the parameter broadcasting dimension if present, else None.

Return type

int or None

control_wires

Control wires of the operator.

For operations that are not controlled, this is an empty Wires object of length 0.

Returns

The control wires of the operation.

Return type

Wires

grad_method = 'A'

grad_recipe = None

Gradient recipe for the parameter-shift method.

This is a tuple with one nested list per operation parameter. For parameter ϕ_k , the nested list contains elements of the form $[c_i, a_i, s_i]$ where i is the index of the term, resulting in a gradient recipe of

$$\frac{\partial}{\partial \phi_k} f = \sum_i c_i f(a_i \phi_k + s_i).$$

If None, the default gradient recipe containing the two terms $[c_0, a_0, s_0] = [1/2, 1, \pi/2]$ and $[c_1, a_1, s_1] = [-1/2, 1, -\pi/2]$ is assumed for every parameter.

Type

tuple(Union(list[list[float]], None)) or None

has_adjoint = True

has_decomposition = True

has_diagonalizing_gates = False

has_generator = True

has matrix = True

hash

Integer hash that uniquely represents the operator.

Type

int

hyperparameters

Dictionary of non-trainable variables that this operation depends on.

Type

dict

id

Custom string to label a specific operator instance.

is_hermitian

This property determines if an operator is hermitian.

name

String for the name of the operator.

ndim_params

Number of dimensions per trainable parameter of the operator.

By default, this property returns the numbers of dimensions of the parameters used for the operator creation. If the parameter sizes for an operator subclass are fixed, this property can be overwritten to return the fixed value.

Returns

Number of dimensions for each trainable parameter.

Return type

tuple

```
num_params = 1
```

$num_wires = 2$

Number of wires the operator acts on.

```
parameter_frequencies = [(1,)]
```

parameters

Trainable parameters that the operator depends on.

wires

Wires that the operator acts on.

Returns

wires

Return type

Wires

adjoint()	Create an operation that is the adjoint of this one.
<pre>compute_decomposition(phi, wires)</pre>	Representation of the operator as a product of other operators (static method).
<pre>compute_diagonalizing_gates(*params, wires,)</pre>	Sequence of gates that diagonalize the operator in the computational basis (static method).
<pre>compute_eigvals(*params, **hyperparams)</pre>	Eigenvalues of the operator in the computational basis (static method).
<pre>compute_matrix(phi)</pre>	Representation of the operator as a canonical matrix in the computational basis (static method).
<pre>compute_sparse_matrix(*params, **hyper- params)</pre>	Representation of the operator as a sparse matrix in the computational basis (static method).
decomposition()	Representation of the operator as a product of other operators.
<pre>diagonalizing_gates()</pre>	Sequence of gates that diagonalize the operator in the computational basis.
eigvals()	Eigenvalues of the operator in the computational basis.
expand()	Returns a tape that contains the decomposition of the operator.
generator()	Generator of an operator that is in single-parameter- form.
<pre>label([decimals, base_label, cache])</pre>	A customizable string representation of the operator.
<pre>map_wires(wire_map)</pre>	Returns a copy of the current operator with its wires changed according to the given wire map.
<pre>matrix([wire_order])</pre>	Representation of the operator as a matrix in the computational basis.
pow(z)	A list of new operators equal to this one raised to the given power.
queue([context])	Append the operator to the Operator queue.
simplify()	Reduce the depth of nested operators to the minimum.
<pre>single_qubit_rot_angles()</pre>	The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.
<pre>sparse_matrix([wire_order])</pre>	Representation of the operator as a sparse matrix in the computational basis.
terms()	Representation of the operator as a linear combination of other operators.
validate_subspace(subspace)	Validate the subspace for qutrit operations.

adjoint()

Create an operation that is the adjoint of this one.

Adjointed operations are the conjugated and transposed version of the original operation. Adjointed ops are equivalent to the inverted operation for unitary gates.

Returns

The adjointed operation.

static compute_decomposition(phi, wires)

Representation of the operator as a product of other operators (static method).

$$O = O_1 O_2 \dots O_n$$
.

Note: Operations making up the decomposition should be queued within the compute_decomposition

method.

See also:

decomposition().

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- wires (Iterable[Any], Wires) wires that the operator acts on
- **hyperparams (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

decomposition of the operator

Return type

list[Operator]

static compute_diagonalizing_gates(*params, wires, **hyperparams)

Sequence of gates that diagonalize the operator in the computational basis (static method).

Given the eigendecomposition $O=U\Sigma U^\dagger$ where Σ is a diagonal matrix containing the eigenvalues, the sequence of diagonalizing gates implements the unitary U^\dagger .

The diagonalizing gates rotate the state into the eigenbasis of the operator.

See also:

diagonalizing_gates().

Parameters

- params (list) trainable parameters of the operator, as stored in the parameters attribute
- wires (Iterable[Any], Wires) wires that the operator acts on
- \bullet hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

list of diagonalizing gates

Return type

list[.Operator]

static compute_eigvals(*params, **hyperparams)

Eigenvalues of the operator in the computational basis (static method).

If $diagonalizing_gates$ are specified and implement a unitary U^{\dagger} , the operator can be reconstructed as

$$O = U\Sigma U^{\dagger}$$
.

where Σ is the diagonal matrix containing the eigenvalues.

Otherwise, no particular order for the eigenvalues is guaranteed.

See also:

Operator.eigvals() and qml.eigvals()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- ****hyperparams** (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

eigenvalues

Return type

tensor like

static compute_matrix(phi)

Representation of the operator as a canonical matrix in the computational basis (static method).

The canonical matrix is the textbook matrix representation that does not consider wires. Implicitly, this assumes that the wires of the operator correspond to the global wire order.

See also:

Operator.matrix() and qml.matrix()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- **hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

matrix representation

Return type

tensor_like

static compute_sparse_matrix(*params, **hyperparams)

Representation of the operator as a sparse matrix in the computational basis (static method).

The canonical matrix is the textbook matrix representation that does not consider wires. Implicitly, this assumes that the wires of the operator correspond to the global wire order.

See also:

sparse_matrix()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- **hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

sparse matrix representation

Return type

scipy.sparse._csr.csr_matrix

decomposition()

Representation of the operator as a product of other operators.

$$O = O_1 O_2 \dots O_n$$

A DecompositionUndefinedError is raised if no representation by decomposition is defined.

See also:

compute_decomposition().

Returns

decomposition of the operator

Return type

list[Operator]

diagonalizing_gates()

Sequence of gates that diagonalize the operator in the computational basis.

Given the eigendecomposition $O = U\Sigma U^{\dagger}$ where Σ is a diagonal matrix containing the eigenvalues, the sequence of diagonalizing gates implements the unitary U^{\dagger} .

The diagonalizing gates rotate the state into the eigenbasis of the operator.

A DiagGatesUndefinedError is raised if no representation by decomposition is defined.

See also:

compute_diagonalizing_gates().

Returns

a list of operators

Return type

list[.Operator] or None

eigvals()

Eigenvalues of the operator in the computational basis.

If diagonalizing_gates are specified and implement a unitary U^{\dagger} , the operator can be reconstructed as

$$O = U\Sigma U^{\dagger}$$
.

where Σ is the diagonal matrix containing the eigenvalues.

Otherwise, no particular order for the eigenvalues is guaranteed.

Note: When eigenvalues are not explicitly defined, they are computed automatically from the matrix representation. Currently, this computation is *not* differentiable.

A EigvalsUndefinedError is raised if the eigenvalues have not been defined and cannot be inferred from the matrix representation.

See also:

compute_eigvals()

eigenvalues

Return type

tensor_like

expand()

Returns a tape that contains the decomposition of the operator.

Returns

quantum tape

Return type

.QuantumTape

generator()

Generator of an operator that is in single-parameter-form.

For example, for operator

$$U(\phi) = e^{i\phi(0.5Y + Z \otimes X)}$$

we get the generator

```
>>> U.generator()
  (0.5) [Y0]
+ (1.0) [Z0 X1]
```

The generator may also be provided in the form of a dense or sparse Hamiltonian (using Hermitian and SparseHamiltonian respectively).

The default value to return is None, indicating that the operation has no defined generator.

label(decimals=None, base_label=None, cache=None)

A customizable string representation of the operator.

Parameters

- **decimals=None** (*int*) If None, no parameters are included. Else, specifies how to round the parameters.
- $base_label=None$ (str) overwrite the non-parameter component of the label
- **cache=None** (*dict*) dictionary that carries information between label calls in the same drawing

Returns

label to use in drawings

Return type

str

Example:

```
>>> op = qml.RX(1.23456, wires=0)
>>> op.label()
"RX"
>>> op.label(decimals=2)
"RX\n(1.23)"
>>> op.label(base_label="my_label")
```

(continues on next page)

(continued from previous page)

```
"my_label"
>>> op.label(decimals=2, base_label="my_label")
"my_label\n(1.23)"
```

If the operation has a matrix-valued parameter and a cache dictionary is provided, unique matrices will be cached in the 'matrices' key list. The label will contain the index of the matrix in the 'matrices' list.

```
>>> op2 = qml.QubitUnitary(np.eye(2), wires=0)
>>> cache = {'matrices': []}
>>> op2.label(cache=cache)
'U(M0)'
>>> cache['matrices']
[tensor([[1., 0.],
[0., 1.]], requires_grad=True)]
>>> op3 = qml.QubitUnitary(np.eye(4), wires=(0,1))
>>> op3.label(cache=cache)
'U(M1)'
>>> cache['matrices']
[tensor([[1., 0.],
        [0., 1.]], requires_grad=True),
tensor([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]], requires_grad=True)]
```

map_wires(wire_map: dict)

Returns a copy of the current operator with its wires changed according to the given wire map.

Parameters

wire_map (dict) - dictionary containing the old wires as keys and the new wires as values

Returns

new operator

Return type

.Operator

```
matrix(wire_order=None)
```

Representation of the operator as a matrix in the computational basis.

If wire_order is provided, the numerical representation considers the position of the operator's wires in the global wire order. Otherwise, the wire order defaults to the operator's wires.

If the matrix depends on trainable parameters, the result will be cast in the same autodifferentiation framework as the parameters.

A MatrixUndefinedError is raised if the matrix representation has not been defined.

See also:

```
compute_matrix()
```

Parameters

wire_order (*Iterable*) – global wire order, must contain all wire labels from the operator's wires

Returns

matrix representation

Return type

tensor_like

$pow(z) \rightarrow List[Operator]$

A list of new operators equal to this one raised to the given power.

Parameters

z (*float*) – exponent for the operator

Returns

list[Operator]

queue(context=<class 'pennylane.queuing.QueuingManager'>)

Append the operator to the Operator queue.

$$simplify() \rightarrow Operator$$

Reduce the depth of nested operators to the minimum.

Returns

simplified operator

Return type

.Operator

single_qubit_rot_angles()

The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.

Returns

A list of values $[\phi, \theta, \omega]$ such that $RZ(\omega)RY(\theta)RZ(\phi)$ is equivalent to the original operation.

Return type

tuple[float, float, float]

sparse_matrix(wire_order=None)

Representation of the operator as a sparse matrix in the computational basis.

If wire_order is provided, the numerical representation considers the position of the operator's wires in the global wire order. Otherwise, the wire order defaults to the operator's wires.

A SparseMatrixUndefinedError is raised if the sparse matrix representation has not been defined.

See also:

compute_sparse_matrix()

Parameters

wire_order (*Iterable*) – global wire order, must contain all wire labels from the operator's wires

Returns

sparse matrix representation

Return type

scipy.sparse._csr.csr_matrix

terms()

Representation of the operator as a linear combination of other operators.

$$O = \sum_{i} c_i O_i$$

A TermsUndefinedError is raised if no representation by terms is defined.

list of coefficients c_i and list of operations O_i

Return type

tuple[list[tensor_like or float], list[.Operation]]

static validate_subspace(subspace)

Validate the subspace for qutrit operations.

This method determines whether a given subspace for qutrit operations is defined correctly or not. If not, a *ValueError* is thrown.

Parameters

subspace (tuple[int]) - Subspace to check for correctness

CPhaseShift10

class CPhaseShift10(phi, wires)

Bases: Operation

Controlled phase shift gate phasing the $|10\rangle$ state.

$$\mbox{CPhaseShift10}(\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & e^{i\phi} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Details:

- Number of wires: 2
- Number of parameters: 1
- · Gradient recipe:

$$\frac{d}{d\phi} \texttt{CPhaseShift10}(\phi) = \frac{1}{2} \left[\texttt{CPhaseShift10}(\phi + \pi/2) - \texttt{CPhaseShift10}(\phi - \pi/2) \right]$$

Parameters

- **phi** (*float*) the controlled phase angle
- wires (int) the subsystem the gate acts on
- **id** (str or None) String representing the operation (optional)

arithmetic_depth	Arithmetic depth of the operator.
basis	The basis of an operation, or for controlled gates, of
	the target operation.
batch_size	Batch size of the operator if it is used with broad-
	casted parameters.
control_wires	Control wires of the operator.
grad_method	
grad_recipe	Gradient recipe for the parameter-shift method.
has_adjoint	
has_decomposition	
has_diagonalizing_gates	
has_generator	
has_matrix	
hash	Integer hash that uniquely represents the operator.
hyperparameters	Dictionary of non-trainable variables that this operation depends on.
id	Custom string to label a specific operator instance.
is_hermitian	This property determines if an operator is hermitian.
name	String for the name of the operator.
ndim_params	Number of dimensions per trainable parameter of the operator.
num_params	
num_wires	Number of wires the operator acts on.
parameter_frequencies	-
parameters	Trainable parameters that the operator depends on.
wires	Wires that the operator acts on.

arithmetic_depth

Arithmetic depth of the operator.

basis = None

The basis of an operation, or for controlled gates, of the target operation. If not None, should take a value of "X", "Y", or "Z".

For example, X and CNOT have basis = "X", whereas ControlledPhaseShift and RZ have basis = "Z".

Type

str or None

batch_size

Batch size of the operator if it is used with broadcasted parameters.

The batch_size is determined based on ndim_params and the provided parameters for the operator. If (some of) the latter have an additional dimension, and this dimension has the same size for all parameters, its size is the batch size of the operator. If no parameter has an additional dimension, the batch size is None.

Size of the parameter broadcasting dimension if present, else None.

Return type

int or None

control_wires

Control wires of the operator.

For operations that are not controlled, this is an empty Wires object of length 0.

Returns

The control wires of the operation.

Return type

Wires

grad_method = 'A'

grad_recipe = None

Gradient recipe for the parameter-shift method.

This is a tuple with one nested list per operation parameter. For parameter ϕ_k , the nested list contains elements of the form $[c_i, a_i, s_i]$ where i is the index of the term, resulting in a gradient recipe of

$$\frac{\partial}{\partial \phi_k} f = \sum_i c_i f(a_i \phi_k + s_i).$$

If None, the default gradient recipe containing the two terms $[c_0, a_0, s_0] = [1/2, 1, \pi/2]$ and $[c_1, a_1, s_1] = [-1/2, 1, -\pi/2]$ is assumed for every parameter.

Type

tuple(Union(list[list[float]], None)) or None

has_adjoint = True

has_decomposition = True

has_diagonalizing_gates = False

has_generator = True

has_matrix = True

hash

Integer hash that uniquely represents the operator.

Type

int

hyperparameters

Dictionary of non-trainable variables that this operation depends on.

Type

dict

id

Custom string to label a specific operator instance.

is_hermitian

This property determines if an operator is hermitian.

name

String for the name of the operator.

ndim_params

Number of dimensions per trainable parameter of the operator.

By default, this property returns the numbers of dimensions of the parameters used for the operator creation. If the parameter sizes for an operator subclass are fixed, this property can be overwritten to return the fixed value.

Returns

Number of dimensions for each trainable parameter.

Return type

tuple

```
num_params = 1
```

$num_wires = 2$

Number of wires the operator acts on.

```
parameter_frequencies = [(1,)]
```

parameters

Trainable parameters that the operator depends on.

wires

Wires that the operator acts on.

Returns

wires

Return type

Wires

adjoint()	Create an operation that is the adjoint of this one.
compute_decomposition(phi, wires)	Representation of the operator as a product of other
compact_accomposition(pm, whos)	operators (static method).
<pre>compute_diagonalizing_gates(*params, wires,</pre>	Sequence of gates that diagonalize the operator in the
)	computational basis (static method).
<pre>compute_eigvals(*params, **hyperparams)</pre>	Eigenvalues of the operator in the computational ba-
	sis (static method).
<pre>compute_matrix(phi)</pre>	Representation of the operator as a canonical matrix
	in the computational basis (static method).
<pre>compute_sparse_matrix(*params, **hyper-</pre>	Representation of the operator as a sparse matrix in
params)	the computational basis (static method).
<pre>decomposition()</pre>	Representation of the operator as a product of other operators.
<pre>diagonalizing_gates()</pre>	Sequence of gates that diagonalize the operator in the computational basis.
eigvals()	Eigenvalues of the operator in the computational ba-
ergvars()	sis.
expand()	Returns a tape that contains the decomposition of the
	operator.
generator()	Generator of an operator that is in single-parameter-
	form.
<pre>label([decimals, base_label, cache])</pre>	A customizable string representation of the operator.
<pre>map_wires(wire_map)</pre>	Returns a copy of the current operator with its wires changed according to the given wire map.
<pre>matrix([wire_order])</pre>	Representation of the operator as a matrix in the computational basis.
pow(z)	A list of new operators equal to this one raised to the
	given power.
queue([context])	Append the operator to the Operator queue.
simplify()	Reduce the depth of nested operators to the minimum.
<pre>single_qubit_rot_angles()</pre>	The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.
<pre>sparse_matrix([wire_order])</pre>	Representation of the operator as a sparse matrix in
	the computational basis.
terms()	Representation of the operator as a linear combina-
	tion of other operators.
validate_subspace(subspace)	Validate the subspace for qutrit operations.

adjoint()

Create an operation that is the adjoint of this one.

Adjointed operations are the conjugated and transposed version of the original operation. Adjointed ops are equivalent to the inverted operation for unitary gates.

Returns

The adjointed operation.

static compute_decomposition(phi, wires)

Representation of the operator as a product of other operators (static method).

$$O = O_1 O_2 \dots O_n$$
.

Note: Operations making up the decomposition should be queued within the compute_decomposition

128 Chapter 2. Tutorials

method.

See also:

decomposition().

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- wires (Iterable[Any], Wires) wires that the operator acts on
- ****hyperparams** (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

decomposition of the operator

Return type

list[Operator]

static compute_diagonalizing_gates(*params, wires, **hyperparams)

Sequence of gates that diagonalize the operator in the computational basis (static method).

Given the eigendecomposition $O=U\Sigma U^\dagger$ where Σ is a diagonal matrix containing the eigenvalues, the sequence of diagonalizing gates implements the unitary U^\dagger .

The diagonalizing gates rotate the state into the eigenbasis of the operator.

See also:

diagonalizing_gates().

Parameters

- params (list) trainable parameters of the operator, as stored in the parameters attribute
- wires (Iterable[Any], Wires) wires that the operator acts on
- **hyperparams** (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

list of diagonalizing gates

Return type

list[.Operator]

static compute_eigvals(*params, **hyperparams)

Eigenvalues of the operator in the computational basis (static method).

If diagonalizing_gates are specified and implement a unitary U^{\dagger} , the operator can be reconstructed as

$$O = U\Sigma U^{\dagger}$$
.

where Σ is the diagonal matrix containing the eigenvalues.

Otherwise, no particular order for the eigenvalues is guaranteed.

See also:

Operator.eigvals() and qml.eigvals()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- ****hyperparams** (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

eigenvalues

Return type

tensor like

static compute_matrix(phi)

Representation of the operator as a canonical matrix in the computational basis (static method).

The canonical matrix is the textbook matrix representation that does not consider wires. Implicitly, this assumes that the wires of the operator correspond to the global wire order.

See also:

Operator.matrix() and qml.matrix()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- **hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

matrix representation

Return type

tensor_like

static compute_sparse_matrix(*params, **hyperparams)

Representation of the operator as a sparse matrix in the computational basis (static method).

The canonical matrix is the textbook matrix representation that does not consider wires. Implicitly, this assumes that the wires of the operator correspond to the global wire order.

See also:

sparse_matrix()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- **hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

sparse matrix representation

Return type

scipy.sparse._csr.csr_matrix

decomposition()

Representation of the operator as a product of other operators.

$$O = O_1 O_2 \dots O_n$$

A DecompositionUndefinedError is raised if no representation by decomposition is defined.

See also:

compute_decomposition().

Returns

decomposition of the operator

Return type

list[Operator]

diagonalizing_gates()

Sequence of gates that diagonalize the operator in the computational basis.

Given the eigendecomposition $O = U\Sigma U^{\dagger}$ where Σ is a diagonal matrix containing the eigenvalues, the sequence of diagonalizing gates implements the unitary U^{\dagger} .

The diagonalizing gates rotate the state into the eigenbasis of the operator.

A DiagGatesUndefinedError is raised if no representation by decomposition is defined.

See also:

compute_diagonalizing_gates().

Returns

a list of operators

Return type

list[.Operator] or None

eigvals()

Eigenvalues of the operator in the computational basis.

If diagonalizing_gates are specified and implement a unitary U^{\dagger} , the operator can be reconstructed as

$$O = U\Sigma U^{\dagger}$$
.

where Σ is the diagonal matrix containing the eigenvalues.

Otherwise, no particular order for the eigenvalues is guaranteed.

Note: When eigenvalues are not explicitly defined, they are computed automatically from the matrix representation. Currently, this computation is *not* differentiable.

A EigvalsUndefinedError is raised if the eigenvalues have not been defined and cannot be inferred from the matrix representation.

See also:

compute_eigvals()

eigenvalues

Return type

tensor_like

expand()

Returns a tape that contains the decomposition of the operator.

Returns

quantum tape

Return type

.QuantumTape

generator()

Generator of an operator that is in single-parameter-form.

For example, for operator

$$U(\phi) = e^{i\phi(0.5Y + Z \otimes X)}$$

we get the generator

```
>>> U.generator()
(0.5) [Y0]
+ (1.0) [Z0 X1]
```

The generator may also be provided in the form of a dense or sparse Hamiltonian (using Hermitian and SparseHamiltonian respectively).

The default value to return is None, indicating that the operation has no defined generator.

label(decimals=None, base_label=None, cache=None)

A customizable string representation of the operator.

Parameters

- **decimals=None** (*int*) If None, no parameters are included. Else, specifies how to round the parameters.
- $base_label=None$ (str) overwrite the non-parameter component of the label
- \bullet **cache=None** (dict) dictionary that carries information between label calls in the same drawing

Returns

label to use in drawings

Return type

str

Example:

```
>>> op = qml.RX(1.23456, wires=0)
>>> op.label()
"RX"
>>> op.label(decimals=2)
"RX\n(1.23)"
>>> op.label(base_label="my_label")
```

(continues on next page)

(continued from previous page)

```
"my_label"
>>> op.label(decimals=2, base_label="my_label")
"my_label\n(1.23)"
```

If the operation has a matrix-valued parameter and a cache dictionary is provided, unique matrices will be cached in the 'matrices' key list. The label will contain the index of the matrix in the 'matrices' list.

```
>>> op2 = qml.QubitUnitary(np.eye(2), wires=0)
>>> cache = {'matrices': []}
>>> op2.label(cache=cache)
'U(M0)'
>>> cache['matrices']
[tensor([[1., 0.],
[0., 1.]], requires_grad=True)]
>>> op3 = qml.QubitUnitary(np.eye(4), wires=(0,1))
>>> op3.label(cache=cache)
'U(M1)'
>>> cache['matrices']
[tensor([[1., 0.],
        [0., 1.]], requires_grad=True),
tensor([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]], requires_grad=True)]
```

map_wires(wire_map: dict)

Returns a copy of the current operator with its wires changed according to the given wire map.

Parameters

wire_map (dict) - dictionary containing the old wires as keys and the new wires as values

Returns

new operator

Return type

.Operator

```
matrix(wire_order=None)
```

Representation of the operator as a matrix in the computational basis.

If wire_order is provided, the numerical representation considers the position of the operator's wires in the global wire order. Otherwise, the wire order defaults to the operator's wires.

If the matrix depends on trainable parameters, the result will be cast in the same autodifferentiation framework as the parameters.

A MatrixUndefinedError is raised if the matrix representation has not been defined.

See also:

compute_matrix()

Parameters

wire_order (*Iterable*) – global wire order, must contain all wire labels from the operator's wires

Returns

matrix representation

Return type

tensor like

$pow(z) \rightarrow List[Operator]$

A list of new operators equal to this one raised to the given power.

Parameters

z (*float*) – exponent for the operator

Returns

list[Operator]

queue(context=<class 'pennylane.queuing.QueuingManager'>)

Append the operator to the Operator queue.

$$simplify() \rightarrow Operator$$

Reduce the depth of nested operators to the minimum.

Returns

simplified operator

Return type

.Operator

single_qubit_rot_angles()

The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.

Returns

A list of values $[\phi, \theta, \omega]$ such that $RZ(\omega)RY(\theta)RZ(\phi)$ is equivalent to the original operation.

Return type

tuple[float, float, float]

sparse_matrix(wire_order=None)

Representation of the operator as a sparse matrix in the computational basis.

If wire_order is provided, the numerical representation considers the position of the operator's wires in the global wire order. Otherwise, the wire order defaults to the operator's wires.

A SparseMatrixUndefinedError is raised if the sparse matrix representation has not been defined.

See also:

compute_sparse_matrix()

Parameters

wire_order (*Iterable*) – global wire order, must contain all wire labels from the operator's wires

Returns

sparse matrix representation

Return type

scipy.sparse._csr.csr_matrix

terms()

Representation of the operator as a linear combination of other operators.

$$O = \sum_{i} c_i O_i$$

A TermsUndefinedError is raised if no representation by terms is defined.

134 Chapter 2. Tutorials

list of coefficients c_i and list of operations O_i

Return type

tuple[list[tensor_like or float], list[.Operation]]

static validate_subspace(subspace)

Validate the subspace for qutrit operations.

This method determines whether a given subspace for qutrit operations is defined correctly or not. If not, a *ValueError* is thrown.

Parameters

subspace (tuple[int]) - Subspace to check for correctness

GPi

class GPi(phi, wires)

Bases: Operation

IonQ native GPi gate.

$$\mathtt{GPi}(\phi) = egin{bmatrix} 0 & e^{-i\phi} \ e^{i\phi} & 0 \end{bmatrix}.$$

Details:

- Number of wires: 1
- Number of parameters: 1

Parameters

- **phi** (*float*) the phase angle
- wires (int) the subsystem the gate acts on
- **id** (*str or None*) String representing the operation (optional)

$\begin{array}{cccccccccccccccccccccccccccccccccccc$		
the target operation. Batch_size	arithmetic_depth	Arithmetic depth of the operator.
$\begin{array}{c} batch_size & Batch_size \ \ \ \ \ \ \ \ \ \ \ \ \ $	basis	The basis of an operation, or for controlled gates, of
$ \begin{array}{c} {\rm casted\ parameters.} \\ {\rm control_wires} \\ {\rm grad_method} \\ \\ {\rm grad_recipe} \\ {\rm has_adjoint} \\ \\ {\rm has_decomposition} \\ \\ {\rm has_deadjoint} \\ \\ {\rm has_deadjoint$		the target operation.
control_wiresControl wires of the operator. $grad_method$ Gradient recipe for the parameter-shift method. $grad_recipe$ Gradient recipe for the parameter-shift method. $has_adjoint$ $has_decomposition$ $has_diagonalizing_gates$ $has_generator$ has_matrix Integer hash that uniquely represents the operator. $hyperparameters$ Dictionary of non-trainable variables that this operation depends on. id Custom string to label a specific operator instance. $is_hermitian$ This property determines if an operator is hermitian. $name$ String for the name of the operator. $ndim_params$ Number of dimensions per trainable parameter of the operator. num_params Number of wires the operator acts on. $parameter_frequencies$ Returns the frequencies for each operator parameter with respect to an expectation value of the form $(\psi U(\mathbf{p})^{\dagger}OU(\mathbf{p}) \psi)$. $parameters$ Trainable parameters that the operator depends on.	batch_size	Batch size of the operator if it is used with broad-
control_wiresControl wires of the operator. $grad_method$ Gradient recipe for the parameter-shift method. $grad_recipe$ Gradient recipe for the parameter-shift method. $has_adjoint$ $has_decomposition$ $has_diagonalizing_gates$ $has_generator$ has_matrix Integer hash that uniquely represents the operator. $hyperparameters$ Dictionary of non-trainable variables that this operation depends on. id Custom string to label a specific operator instance. $is_hermitian$ This property determines if an operator is hermitian. $name$ String for the name of the operator. $ndim_params$ Number of dimensions per trainable parameter of the operator. num_params Number of wires the operator acts on. $parameter_frequencies$ Returns the frequencies for each operator parameter with respect to an expectation value of the form $(\psi U(\mathbf{p})^{\dagger}OU(\mathbf{p}) \psi)$. $parameters$ Trainable parameters that the operator depends on.		casted parameters.
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	control wires	
$\begin{array}{cccccccccccccccccccccccccccccccccccc$		T
$\begin{array}{llllllllllllllllllllllllllllllllllll$	92 44	
$\begin{array}{llllllllllllllllllllllllllllllllllll$	grad recipe	Gradient recipe for the parameter-shift method
$\begin{array}{lll} has_decomposition \\ has_diagonalizing_gates \\ has_generator \\ hash & Integer hash that uniquely represents the operator. \\ hyperparameters & Dictionary of non-trainable variables that this operation depends on. \\ id & Custom string to label a specific operator instance. \\ is_hermitian & This property determines if an operator is hermitian. \\ name & String for the name of the operator. \\ ndim_params & Number of dimensions per trainable parameter of the operator. \\ num_params & Number of wires the operator acts on. \\ parameter_frequencies & Returns the frequencies for each operator parameter with respect to an expectation value of the form \langle \psi U(\mathbf{p})^{\dagger} \hat{O}U(\mathbf{p}) \psi \rangle. \\ parameters & Trainable parameters that the operator depends on. \\ \end{array}$		Gradient recipe for the parameter shift method.
$\begin{array}{lll} has_diagonalizing_gates \\ has_generator \\ \\ has_matrix \\ \\ hash & Integer hash that uniquely represents the operator. \\ hyperparameters & Dictionary of non-trainable variables that this operation depends on. \\ id & Custom string to label a specific operator instance. \\ is_hermitian & This property determines if an operator is hermitian. \\ name & String for the name of the operator. \\ ndim_params & Number of dimensions per trainable parameter of the operator. \\ num_params & Number of wires the operator acts on. \\ parameter_frequencies & Returns the frequencies for each operator parameter with respect to an expectation value of the form \langle \psi U(\mathbf{p})^\dagger \hat{O}U(\mathbf{p}) \psi \rangle. \\ parameters & Trainable parameters that the operator depends on. \\ \hline \end{array}$	nas_aujoint	
$\begin{array}{lll} has_diagonalizing_gates \\ has_generator \\ \\ has_matrix \\ \\ hash & Integer hash that uniquely represents the operator. \\ hyperparameters & Dictionary of non-trainable variables that this operation depends on. \\ id & Custom string to label a specific operator instance. \\ is_hermitian & This property determines if an operator is hermitian. \\ name & String for the name of the operator. \\ ndim_params & Number of dimensions per trainable parameter of the operator. \\ num_params & Number of wires the operator acts on. \\ parameter_frequencies & Returns the frequencies for each operator parameter with respect to an expectation value of the form \langle \psi U(\mathbf{p})^\dagger \hat{O}U(\mathbf{p}) \psi \rangle. \\ parameters & Trainable parameters that the operator depends on. \\ \hline \end{array}$	has desampesition	
$\begin{array}{lll} has_generator \\ has_matrix \\ hash & Integer hash that uniquely represents the operator. \\ hyperparameters & Dictionary of non-trainable variables that this operation depends on. \\ id & Custom string to label a specific operator instance. \\ is_hermitian & This property determines if an operator is hermitian. \\ name & String for the name of the operator. \\ ndim_params & Number of dimensions per trainable parameter of the operator. \\ num_params & Number of wires the operator acts on. \\ parameter_frequencies & Returns the frequencies for each operator parameter with respect to an expectation value of the form \langle \psi U(\mathbf{p})^\dagger \hat{O}U(\mathbf{p}) \psi \rangle. \\ parameters & Trainable parameters that the operator depends on. \\ \hline \end{array}$	nas_decomposition	
$\begin{array}{lll} has_generator \\ has_matrix \\ hash & Integer hash that uniquely represents the operator. \\ hyperparameters & Dictionary of non-trainable variables that this operation depends on. \\ id & Custom string to label a specific operator instance. \\ is_hermitian & This property determines if an operator is hermitian. \\ name & String for the name of the operator. \\ ndim_params & Number of dimensions per trainable parameter of the operator. \\ num_params & Number of wires the operator acts on. \\ parameter_frequencies & Returns the frequencies for each operator parameter with respect to an expectation value of the form \langle \psi U(\mathbf{p})^\dagger \hat{O}U(\mathbf{p}) \psi \rangle. \\ parameters & Trainable parameters that the operator depends on. \\ \hline \end{array}$	1 11 71 1	
$\begin{array}{lll} has & & & & & \\ hash & & & & \\ hyperparameters & & & \\ Dictionary of non-trainable variables that this operation depends on. \\ id & & & \\ Custom string to label a specific operator instance. \\ is_hermitian & & \\ This property determines if an operator is hermitian. \\ name & & \\ String for the name of the operator. \\ ndim_params & & \\ Number of dimensions per trainable parameter of the operator. \\ num_params & & \\ num_wires & & \\ num_wires & & \\ parameter_frequencies & & \\ Returns the frequencies for each operator parameter with respect to an expectation value of the form \langle \psi U(\mathbf{p})^{\dagger} \hat{O}U(\mathbf{p}) \psi \rangle. \\ parameters & & \\ Trainable parameters that the operator depends on. \\ \end{array}$	has_diagonalizing_gates	
$\begin{array}{lll} has & & & & & \\ hash & & & & \\ hyperparameters & & & \\ Dictionary of non-trainable variables that this operation depends on. \\ id & & & \\ Custom string to label a specific operator instance. \\ is_hermitian & & \\ This property determines if an operator is hermitian. \\ name & & \\ String for the name of the operator. \\ ndim_params & & \\ Number of dimensions per trainable parameter of the operator. \\ num_params & & \\ num_wires & & \\ num_wires & & \\ parameter_frequencies & & \\ Returns the frequencies for each operator parameter with respect to an expectation value of the form \langle \psi U(\mathbf{p})^{\dagger} \hat{O}U(\mathbf{p}) \psi \rangle. \\ parameters & & \\ Trainable parameters that the operator depends on. \\ \end{array}$		
$\begin{array}{c} hash \\ hyperparameters \\ \hline \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ $	has_generator	
$\begin{array}{c} hash \\ hyperparameters \\ \hline \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ $		
$\begin{array}{ll} \textit{hyperparameters} & \text{Dictionary of non-trainable variables that this operation depends on.} \\ \textit{id} & \text{Custom string to label a specific operator instance.} \\ \textit{is_hermitian} & \text{This property determines if an operator is hermitian.} \\ \textit{name} & \text{String for the name of the operator.} \\ \textit{ndim_params} & \text{Number of dimensions per trainable parameter of the operator.} \\ \textit{num_params} & \text{Number of wires the operator acts on.} \\ \textit{parameter_frequencies} & \text{Returns the frequencies for each operator parameter with respect to an expectation value of the form} \\ & \langle \psi U(\mathbf{p})^\dagger \hat{O} U(\mathbf{p}) \psi \rangle. \\ \textit{parameters} & \text{Trainable parameters that the operator depends on.} \\ \end{aligned}$	has_matrix	
$\begin{array}{ll} \textit{hyperparameters} & \text{Dictionary of non-trainable variables that this operation depends on.} \\ \textit{id} & \text{Custom string to label a specific operator instance.} \\ \textit{is_hermitian} & \text{This property determines if an operator is hermitian.} \\ \textit{name} & \text{String for the name of the operator.} \\ \textit{ndim_params} & \text{Number of dimensions per trainable parameter of the operator.} \\ \textit{num_params} & \text{Number of wires the operator acts on.} \\ \textit{parameter_frequencies} & \text{Returns the frequencies for each operator parameter with respect to an expectation value of the form} \\ & \langle \psi U(\mathbf{p})^\dagger \hat{O} U(\mathbf{p}) \psi \rangle. \\ \textit{parameters} & \text{Trainable parameters that the operator depends on.} \\ \end{aligned}$		
tion depends on. id Custom string to label a specific operator instance. is_hermitian This property determines if an operator is hermitian. name String for the name of the operator. ndim_params Number of dimensions per trainable parameter of the operator. num_params num_wires Number of wires the operator acts on. parameter_frequencies Returns the frequencies for each operator parameter with respect to an expectation value of the form $\langle \psi U(\mathbf{p})^{\dagger} \hat{O} U(\mathbf{p}) \psi \rangle$. parameters Trainable parameters that the operator depends on.	hash	Integer hash that uniquely represents the operator.
$ \begin{array}{cccccccccccccccccccccccccccccccccccc$	hyperparameters	Dictionary of non-trainable variables that this opera-
$ \begin{array}{cccccccccccccccccccccccccccccccccccc$		tion depends on.
$\begin{array}{ll} \textit{is_hermitian} & \text{This property determines if an operator is hermitian.} \\ \textit{name} & \text{String for the name of the operator.} \\ \textit{ndim_params} & \text{Number of dimensions per trainable parameter of the operator.} \\ \textit{num_params} & \text{Number of wires the operator acts on.} \\ \textit{parameter_frequencies} & \text{Returns the frequencies for each operator parameter with respect to an expectation value of the form} \\ & \langle \psi U(\mathbf{p})^{\dagger} \hat{O} U(\mathbf{p}) \psi \rangle. \\ \textit{parameters} & \text{Trainable parameters that the operator depends on.} \\ \end{array}$	id	
name String for the name of the operator.	is hermitian	
$\begin{array}{c} \textit{ndim_params} & \textit{Number of dimensions per trainable parameter of the} \\ \textit{operator.} \\ \\ \textit{num_params} \\ \\ \\ \textit{num_wires} & \textit{Number of wires the operator acts on.} \\ \textit{parameter_frequencies} & \textit{Returns the frequencies for each operator parameter with respect to an expectation value of the form} \\ & \langle \psi U(\mathbf{p})^{\dagger} \hat{O} U(\mathbf{p}) \psi \rangle. \\ \\ \textit{parameters} & \textit{Trainable parameters that the operator depends on.} \\ \end{array}$		1 1 7
$\begin{array}{c} \textit{num_params} \\ \\ \hline \textit{num_wires} \\ \\ \textit{parameter_frequencies} \\ \\ \textit{parameters} \\ \\ \\ \textit{parameters} \\ \\ \\ \textit{parameters} \\ \\ \\ \\ \textit{parameters} \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\$		
$\begin{array}{ll} \textit{num_params} \\ \\ \textit{num_wires} \\ \textit{parameter_frequencies} \\ \\ \textit{parameter_frequencies} \\ \\ \textit{vil}(\mathbf{p})^{\dagger} \hat{O}U(\mathbf{p}) \psi \rangle. \\ \\ \textit{parameters} \\ \\ \\ \textit{parameters} \\ \end{array}$ Number of wires the operator acts on. Returns the frequencies for each operator parameter with respect to an expectation value of the form $\langle \psi U(\mathbf{p})^{\dagger} \hat{O}U(\mathbf{p}) \psi \rangle. \\ \\ \textit{parameters} \\ \\ \end{aligned}$	Traini_par ans	
$\begin{array}{ll} \textit{num_wires} & \text{Number of wires the operator acts on.} \\ \textit{parameter_frequencies} & \text{Returns the frequencies for each operator parameter with respect to an expectation value of the form} \\ & \langle \psi U(\mathbf{p})^\dagger \hat{O} U(\mathbf{p}) \psi \rangle. \\ \textit{parameters} & \text{Trainable parameters that the operator depends on.} \\ \end{array}$	ทุเพ การกา	operator.
$\begin{array}{ccc} \textit{parameter_frequencies} & \text{Returns the frequencies for each operator parameter with respect to an expectation value of the form} \\ & & \langle \psi U(\mathbf{p})^\dagger \hat{O} U(\mathbf{p}) \psi \rangle. \\ \textit{parameters} & \text{Trainable parameters that the operator depends on.} \end{array}$	rum_params	
$\begin{array}{ccc} \textit{parameter_frequencies} & \text{Returns the frequencies for each operator parameter with respect to an expectation value of the form} \\ & & \langle \psi U(\mathbf{p})^\dagger \hat{O} U(\mathbf{p}) \psi \rangle. \\ \textit{parameters} & \text{Trainable parameters that the operator depends on.} \end{array}$	num tri roc	Number of wires the operator sets on
$\begin{array}{c} \text{ter with respect to an expectation value of the form} \\ \langle \psi U(\mathbf{p})^\dagger \hat{O} U(\mathbf{p}) \psi \rangle. \\ \\ \textit{parameters} \end{array}$ Trainable parameters that the operator depends on.		
$\langle \psi U(\mathbf{p})^{\dagger}\hat{O}U(\mathbf{p}) \psi\rangle.$ parameters Trainable parameters that the operator depends on.	parameter_irequencies	
parameters Trainable parameters that the operator depends on.		
	•	•
Wires that the operator acts on.	wires	Wires that the operator acts on.

arithmetic_depth

Arithmetic depth of the operator.

basis = None

The basis of an operation, or for controlled gates, of the target operation. If not None, should take a value of "X", "Y", or "Z".

For example, X and CNOT have basis = "X", whereas ControlledPhaseShift and RZ have basis = "Z".

Type

str or None

batch_size

Batch size of the operator if it is used with broadcasted parameters.

The batch_size is determined based on ndim_params and the provided parameters for the operator. If (some of) the latter have an additional dimension, and this dimension has the same size for all parameters, its size is the batch size of the operator. If no parameter has an additional dimension, the batch size is None.

136 Chapter 2. Tutorials

Size of the parameter broadcasting dimension if present, else None.

Return type

int or None

control_wires

Control wires of the operator.

For operations that are not controlled, this is an empty Wires object of length 0.

Returns

The control wires of the operation.

Return type

Wires

grad_method = 'F'

grad_recipe = None

Gradient recipe for the parameter-shift method.

This is a tuple with one nested list per operation parameter. For parameter ϕ_k , the nested list contains elements of the form $[c_i, a_i, s_i]$ where i is the index of the term, resulting in a gradient recipe of

$$\frac{\partial}{\partial \phi_k} f = \sum_i c_i f(a_i \phi_k + s_i).$$

If None, the default gradient recipe containing the two terms $[c_0, a_0, s_0] = [1/2, 1, \pi/2]$ and $[c_1, a_1, s_1] = [-1/2, 1, -\pi/2]$ is assumed for every parameter.

Type

tuple(Union(list[list[float]], None)) or None

has_adjoint = True

has_decomposition = False

has_diagonalizing_gates = False

has_generator = False

has matrix = True

hash

Integer hash that uniquely represents the operator.

Type

int

hyperparameters

Dictionary of non-trainable variables that this operation depends on.

Type

dict

id

Custom string to label a specific operator instance.

is_hermitian

This property determines if an operator is hermitian.

name

String for the name of the operator.

ndim_params

Number of dimensions per trainable parameter of the operator.

By default, this property returns the numbers of dimensions of the parameters used for the operator creation. If the parameter sizes for an operator subclass are fixed, this property can be overwritten to return the fixed value.

Returns

Number of dimensions for each trainable parameter.

Return type

tuple

```
num_params = 1
```

$num_wires = 1$

Number of wires the operator acts on.

parameter_frequencies

Returns the frequencies for each operator parameter with respect to an expectation value of the form $\langle \psi | U(\mathbf{p})^{\dagger} \hat{O} U(\mathbf{p}) | \psi \rangle$.

These frequencies encode the behaviour of the operator $U(\mathbf{p})$ on the value of the expectation value as the parameters are modified. For more details, please see the pennylane. fourier module.

Returns

Tuple of frequencies for each parameter. Note that only non-negative frequency values are returned.

Return type

list[tuple[int or float]]

Example

```
>>> op = qml.CRot(0.4, 0.1, 0.3, wires=[0, 1])
>>> op.parameter_frequencies
[(0.5, 1), (0.5, 1), (0.5, 1)]
```

For operators that define a generator, the parameter frequencies are directly related to the eigenvalues of the generator:

```
>>> op = qml.ControlledPhaseShift(0.1, wires=[0, 1])
>>> op.parameter_frequencies
[(1,)]
>>> gen = qml.generator(op, format="observable")
>>> gen_eigvals = qml.eigvals(gen)
>>> qml.gradients.eigvals_to_frequencies(tuple(gen_eigvals))
(1.0,)
```

For more details on this relationship, see eigvals_to_frequencies().

parameters

Trainable parameters that the operator depends on.

wires

Wires that the operator acts on.

wires

Return type

Wires

adjoint()	Create an operation that is the adjoint of this one.
<pre>compute_decomposition(*params[, wires])</pre>	Representation of the operator as a product of other operators (static method).
<pre>compute_diagonalizing_gates(*params, wires,)</pre>	Sequence of gates that diagonalize the operator in the computational basis (static method).
<pre>compute_eigvals(*params, **hyperparams)</pre>	Eigenvalues of the operator in the computational basis (static method).
<pre>compute_matrix(phi)</pre>	Representation of the operator as a canonical matrix in the computational basis (static method).
<pre>compute_sparse_matrix(*params, **hyper- params)</pre>	Representation of the operator as a sparse matrix in the computational basis (static method).
<pre>decomposition()</pre>	Representation of the operator as a product of other operators.
<pre>diagonalizing_gates()</pre>	Sequence of gates that diagonalize the operator in the computational basis.
eigvals()	Eigenvalues of the operator in the computational basis.
expand()	Returns a tape that contains the decomposition of the operator.
generator()	Generator of an operator that is in single-parameter- form.
<pre>label([decimals, base_label, cache])</pre>	A customizable string representation of the operator.
<pre>map_wires(wire_map)</pre>	Returns a copy of the current operator with its wires changed according to the given wire map.
<pre>matrix([wire_order])</pre>	Representation of the operator as a matrix in the computational basis.
pow(z)	A list of new operators equal to this one raised to the given power.
queue([context])	Append the operator to the Operator queue.
simplify()	Reduce the depth of nested operators to the minimum.
<pre>single_qubit_rot_angles()</pre>	The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.
<pre>sparse_matrix([wire_order])</pre>	Representation of the operator as a sparse matrix in the computational basis.
terms()	Representation of the operator as a linear combination of other operators.
validate_subspace(subspace)	Validate the subspace for qutrit operations.

adjoint()

Create an operation that is the adjoint of this one.

Adjointed operations are the conjugated and transposed version of the original operation. Adjointed ops are equivalent to the inverted operation for unitary gates.

Returns

The adjointed operation.

static compute_decomposition(*params, wires=None, **hyperparameters)

Representation of the operator as a product of other operators (static method).

$$O = O_1 O_2 \dots O_n$$
.

Note: Operations making up the decomposition should be queued within the compute_decomposition method.

See also:

decomposition().

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- wires (Iterable[Any], Wires) wires that the operator acts on
- **hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

decomposition of the operator

Return type

list[Operator]

static compute_diagonalizing_gates(*params, wires, **hyperparams)

Sequence of gates that diagonalize the operator in the computational basis (static method).

Given the eigendecomposition $O=U\Sigma U^\dagger$ where Σ is a diagonal matrix containing the eigenvalues, the sequence of diagonalizing gates implements the unitary U^\dagger .

The diagonalizing gates rotate the state into the eigenbasis of the operator.

See also:

diagonalizing_gates().

Parameters

- ullet params (list) trainable parameters of the operator, as stored in the parameters attribute
- wires (Iterable[Any], Wires) wires that the operator acts on
- \bullet hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

list of diagonalizing gates

Return type

list[.Operator]

static compute_eigvals(*params, **hyperparams)

Eigenvalues of the operator in the computational basis (static method).

If diagonalizing_gates are specified and implement a unitary U^{\dagger} , the operator can be reconstructed as

$$O = U\Sigma U^{\dagger}$$
.

where Σ is the diagonal matrix containing the eigenvalues.

Otherwise, no particular order for the eigenvalues is guaranteed.

See also:

Operator.eigvals() and qml.eigvals()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- **hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

eigenvalues

Return type

tensor_like

static compute_matrix(phi)

Representation of the operator as a canonical matrix in the computational basis (static method).

The canonical matrix is the textbook matrix representation that does not consider wires. Implicitly, this assumes that the wires of the operator correspond to the global wire order.

See also:

Operator.matrix() and qml.matrix()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- ****hyperparams** (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

matrix representation

Return type

tensor_like

static compute_sparse_matrix(*params, **hyperparams)

Representation of the operator as a sparse matrix in the computational basis (static method).

The canonical matrix is the textbook matrix representation that does not consider wires. Implicitly, this assumes that the wires of the operator correspond to the global wire order.

See also:

sparse_matrix()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- ****hyperparams** (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

sparse matrix representation

Return type

scipy.sparse._csr.csr_matrix

decomposition()

Representation of the operator as a product of other operators.

$$O = O_1 O_2 \dots O_n$$

A DecompositionUndefinedError is raised if no representation by decomposition is defined.

See also:

compute_decomposition().

Returns

decomposition of the operator

Return type

list[Operator]

diagonalizing_gates()

Sequence of gates that diagonalize the operator in the computational basis.

Given the eigendecomposition $O=U\Sigma U^{\dagger}$ where Σ is a diagonal matrix containing the eigenvalues, the sequence of diagonalizing gates implements the unitary U^{\dagger} .

The diagonalizing gates rotate the state into the eigenbasis of the operator.

A DiagGatesUndefinedError is raised if no representation by decomposition is defined.

See also:

compute_diagonalizing_gates().

Returns

a list of operators

Return type

list[.Operator] or None

eigvals()

Eigenvalues of the operator in the computational basis.

If diagonalizing_gates are specified and implement a unitary U^{\dagger} , the operator can be reconstructed as

$$O = U\Sigma U^{\dagger}$$
,

where Σ is the diagonal matrix containing the eigenvalues.

Otherwise, no particular order for the eigenvalues is guaranteed.

142 Chapter 2. Tutorials

Note: When eigenvalues are not explicitly defined, they are computed automatically from the matrix representation. Currently, this computation is *not* differentiable.

A EigvalsUndefinedError is raised if the eigenvalues have not been defined and cannot be inferred from the matrix representation.

See also:

compute_eigvals()

Returns

eigenvalues

Return type

tensor_like

expand()

Returns a tape that contains the decomposition of the operator.

Returns

quantum tape

Return type

.QuantumTape

generator()

Generator of an operator that is in single-parameter-form.

For example, for operator

$$U(\phi) = e^{i\phi(0.5Y + Z \otimes X)}$$

we get the generator

```
>>> U.generator()
    (0.5) [Y0]
+ (1.0) [Z0 X1]
```

The generator may also be provided in the form of a dense or sparse Hamiltonian (using Hermitian and SparseHamiltonian respectively).

The default value to return is None, indicating that the operation has no defined generator.

label(decimals=None, base_label=None, cache=None)

A customizable string representation of the operator.

Parameters

- **decimals=None** (*int*) If None, no parameters are included. Else, specifies how to round the parameters.
- base_label=None (str) overwrite the non-parameter component of the label
- cache=None (dict) dictionary that carries information between label calls in the same drawing

Returns

label to use in drawings

Return type str

Example:

```
>>> op = qml.RX(1.23456, wires=0)
>>> op.label()
"RX"
>>> op.label(decimals=2)
"RX\n(1.23)"
>>> op.label(base_label="my_label")
"my_label"
>>> op.label(decimals=2, base_label="my_label")
"my_label\n(1.23)"
```

If the operation has a matrix-valued parameter and a cache dictionary is provided, unique matrices will be cached in the 'matrices' key list. The label will contain the index of the matrix in the 'matrices' list.

```
>>> op2 = qml.QubitUnitary(np.eye(2), wires=0)
>>> cache = {'matrices': []}
>>> op2.label(cache=cache)
'U(M0)'
>>> cache['matrices']
[tensor([[1., 0.],
[0., 1.]], requires_grad=True)]
>>> op3 = qml.QubitUnitary(np.eye(4), wires=(0,1))
>>> op3.label(cache=cache)
'U(M1)'
>>> cache['matrices']
[tensor([[1., 0.],
        [0., 1.]], requires_grad=True),
tensor([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]], requires_grad=True)]
```

map_wires(wire_map: dict)

Returns a copy of the current operator with its wires changed according to the given wire map.

Parameters

wire_map (dict) - dictionary containing the old wires as keys and the new wires as values

Returns

new operator

Return type

.Operator

matrix(wire_order=None)

Representation of the operator as a matrix in the computational basis.

If wire_order is provided, the numerical representation considers the position of the operator's wires in the global wire order. Otherwise, the wire order defaults to the operator's wires.

If the matrix depends on trainable parameters, the result will be cast in the same autodifferentiation framework as the parameters.

A MatrixUndefinedError is raised if the matrix representation has not been defined.

See also:

compute_matrix()

Parameters

wire_order (*Iterable*) – global wire order, must contain all wire labels from the operator's wires

Returns

matrix representation

Return type

tensor_like

$pow(z) \rightarrow List[Operator]$

A list of new operators equal to this one raised to the given power.

Parameters

z (*float*) – exponent for the operator

Returns

list[Operator]

queue(context=<class 'pennylane.queuing.QueuingManager'>)

Append the operator to the Operator queue.

$simplify() \rightarrow Operator$

Reduce the depth of nested operators to the minimum.

Returns

simplified operator

Return type

.Operator

single_qubit_rot_angles()

The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.

Returns

A list of values $[\phi, \theta, \omega]$ such that $RZ(\omega)RY(\theta)RZ(\phi)$ is equivalent to the original operation.

Return type

tuple[float, float, float]

sparse_matrix(wire_order=None)

Representation of the operator as a sparse matrix in the computational basis.

If wire_order is provided, the numerical representation considers the position of the operator's wires in the global wire order. Otherwise, the wire order defaults to the operator's wires.

 $A \ Sparse {\tt Matrix} {\tt UndefinedError} \ is \ raised \ if \ the \ sparse \ matrix \ representation \ has \ not \ been \ defined.$

See also:

compute_sparse_matrix()

Parameters

wire_order (*Iterable*) – global wire order, must contain all wire labels from the operator's wires

Returns

sparse matrix representation

Return type

scipy.sparse._csr.csr_matrix

terms()

Representation of the operator as a linear combination of other operators.

$$O = \sum_{i} c_i O_i$$

A TermsUndefinedError is raised if no representation by terms is defined.

Returns

list of coefficients c_i and list of operations O_i

Return type

tuple[list[tensor_like or float], list[.Operation]]

static validate_subspace(subspace)

Validate the subspace for qutrit operations.

This method determines whether a given subspace for qutrit operations is defined correctly or not. If not, a *ValueError* is thrown.

Parameters

 $\textbf{subspace} \; (\textit{tuple[int]}) - Subspace \; to \; check \; for \; correctness$

GPi2

class GPi2(phi, wires)

Bases: Operation

IonQ native GPi2 gate.

$$\mathrm{GPi2}(\phi) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -ie^{-i\phi} \\ -ie^{i\phi} & 1 \end{bmatrix}.$$

Details:

- Number of wires: 1
- Number of parameters: 1

Parameters

- **phi** (*float*) the phase angle
- wires (int) the subsystem the gate acts on
- **id** (*str or None*) String representing the operation (optional)

arithmetic_depth	Auithmatic danth of the angustan
basis	Arithmetic depth of the operator.
Dasis	The basis of an operation, or for controlled gates, of
	the target operation.
batch_size	Batch size of the operator if it is used with broad-
	casted parameters.
control_wires	Control wires of the operator.
<pre>grad_method</pre>	
grad_recipe	Gradient recipe for the parameter-shift method.
has_adjoint	
has_decomposition	
- •	
has_diagonalizing_gates	
<u>-</u> g_g	
has_generator	
nas_generator	
has matrix	
IIa3_IIIa CI IX	
hash	Integer hash that uniquely represents the operator.
	Dictionary of non-trainable variables that this opera-
hyperparameters	tion depends on.
2.4	•
id	Custom string to label a specific operator instance.
is_hermitian	This property determines if an operator is hermitian.
name	String for the name of the operator.
ndim_params	Number of dimensions per trainable parameter of the
	operator.
num_params	
num_wires	Number of wires the operator acts on.
parameter_frequencies	Returns the frequencies for each operator parame-
	ter with respect to an expectation value of the form
	$\langle \psi U(\mathbf{p})^{\dagger} \hat{O} U(\mathbf{p}) \psi \rangle$.
parameters	Trainable parameters that the operator depends on.
wires	Wires that the operator acts on.
	· · r · · · · · · · · · · · · ·

arithmetic_depth

Arithmetic depth of the operator.

basis = None

The basis of an operation, or for controlled gates, of the target operation. If not None, should take a value of "X", "Y", or "Z".

For example, X and CNOT have basis = "X", whereas ControlledPhaseShift and RZ have basis = "Z".

Type

str or None

batch_size

Batch size of the operator if it is used with broadcasted parameters.

The batch_size is determined based on ndim_params and the provided parameters for the operator. If (some of) the latter have an additional dimension, and this dimension has the same size for all parameters, its size is the batch size of the operator. If no parameter has an additional dimension, the batch size is None.

Returns

Size of the parameter broadcasting dimension if present, else None.

Return type

int or None

control_wires

Control wires of the operator.

For operations that are not controlled, this is an empty Wires object of length 0.

Returns

The control wires of the operation.

Return type

Wires

grad_method = 'F'

grad_recipe = None

Gradient recipe for the parameter-shift method.

This is a tuple with one nested list per operation parameter. For parameter ϕ_k , the nested list contains elements of the form $[c_i, a_i, s_i]$ where i is the index of the term, resulting in a gradient recipe of

$$\frac{\partial}{\partial \phi_k} f = \sum_i c_i f(a_i \phi_k + s_i).$$

If None, the default gradient recipe containing the two terms $[c_0, a_0, s_0] = [1/2, 1, \pi/2]$ and $[c_1, a_1, s_1] = [-1/2, 1, -\pi/2]$ is assumed for every parameter.

Type

tuple(Union(list[list[float]], None)) or None

has_adjoint = True

has_decomposition = False

has_diagonalizing_gates = False

has_generator = False

has matrix = True

hash

Integer hash that uniquely represents the operator.

Type

int

hyperparameters

Dictionary of non-trainable variables that this operation depends on.

Type

dict

id

Custom string to label a specific operator instance.

is_hermitian

This property determines if an operator is hermitian.

name

String for the name of the operator.

ndim_params

Number of dimensions per trainable parameter of the operator.

By default, this property returns the numbers of dimensions of the parameters used for the operator creation. If the parameter sizes for an operator subclass are fixed, this property can be overwritten to return the fixed value.

Returns

Number of dimensions for each trainable parameter.

Return type

tuple

```
num_params = 1
```

num_wires = 1

Number of wires the operator acts on.

parameter_frequencies

Returns the frequencies for each operator parameter with respect to an expectation value of the form $\langle \psi | U(\mathbf{p})^{\dagger} \hat{O} U(\mathbf{p}) | \psi \rangle$.

These frequencies encode the behaviour of the operator $U(\mathbf{p})$ on the value of the expectation value as the parameters are modified. For more details, please see the pennylane. fourier module.

Returns

Tuple of frequencies for each parameter. Note that only non-negative frequency values are returned.

Return type

list[tuple[int or float]]

Example

```
>>> op = qml.CRot(0.4, 0.1, 0.3, wires=[0, 1])
>>> op.parameter_frequencies
[(0.5, 1), (0.5, 1), (0.5, 1)]
```

For operators that define a generator, the parameter frequencies are directly related to the eigenvalues of the generator:

```
>>> op = qml.ControlledPhaseShift(0.1, wires=[0, 1])
>>> op.parameter_frequencies
[(1,)]
>>> gen = qml.generator(op, format="observable")
>>> gen_eigvals = qml.eigvals(gen)
>>> qml.gradients.eigvals_to_frequencies(tuple(gen_eigvals))
(1.0,)
```

For more details on this relationship, see eigvals_to_frequencies().

parameters

Trainable parameters that the operator depends on.

wires

Wires that the operator acts on.

Returns

wires

Return type

Wires

adjoint()	Create an operation that is the adjoint of this one.
<pre>compute_decomposition(*params[, wires])</pre>	Representation of the operator as a product of other operators (static method).
<pre>compute_diagonalizing_gates(*params, wires,)</pre>	Sequence of gates that diagonalize the operator in the computational basis (static method).
<pre>compute_eigvals(*params, **hyperparams)</pre>	Eigenvalues of the operator in the computational basis (static method).
<pre>compute_matrix(phi)</pre>	Representation of the operator as a canonical matrix in the computational basis (static method).
<pre>compute_sparse_matrix(*params, **hyper- params)</pre>	Representation of the operator as a sparse matrix in the computational basis (static method).
decomposition()	Representation of the operator as a product of other operators.
diagonalizing_gates()	Sequence of gates that diagonalize the operator in the computational basis.
eigvals()	Eigenvalues of the operator in the computational basis.
expand()	Returns a tape that contains the decomposition of the operator.
generator()	Generator of an operator that is in single-parameter- form.
<pre>label([decimals, base_label, cache])</pre>	A customizable string representation of the operator.
<pre>map_wires(wire_map)</pre>	Returns a copy of the current operator with its wires changed according to the given wire map.
<pre>matrix([wire_order])</pre>	Representation of the operator as a matrix in the computational basis.
pow(z)	A list of new operators equal to this one raised to the given power.
queue([context])	Append the operator to the Operator queue.
simplify()	Reduce the depth of nested operators to the minimum.
<pre>single_qubit_rot_angles()</pre>	The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.
<pre>sparse_matrix([wire_order])</pre>	Representation of the operator as a sparse matrix in the computational basis.
terms()	Representation of the operator as a linear combination of other operators.
<pre>validate_subspace(subspace)</pre>	Validate the subspace for qutrit operations.

adjoint()

Create an operation that is the adjoint of this one.

Adjointed operations are the conjugated and transposed version of the original operation. Adjointed ops are equivalent to the inverted operation for unitary gates.

Returns

The adjointed operation.

static compute_decomposition(*params, wires=None, **hyperparameters)

150 Chapter 2. Tutorials

Representation of the operator as a product of other operators (static method).

$$O = O_1 O_2 \dots O_n$$
.

Note: Operations making up the decomposition should be queued within the compute_decomposition method.

See also:

decomposition().

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- wires (Iterable[Any], Wires) wires that the operator acts on
- **hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

decomposition of the operator

Return type

list[Operator]

static compute_diagonalizing_gates(*params, wires, **hyperparams)

Sequence of gates that diagonalize the operator in the computational basis (static method).

Given the eigendecomposition $O=U\Sigma U^\dagger$ where Σ is a diagonal matrix containing the eigenvalues, the sequence of diagonalizing gates implements the unitary U^\dagger .

The diagonalizing gates rotate the state into the eigenbasis of the operator.

See also:

diagonalizing_gates().

Parameters

- ullet params (list) trainable parameters of the operator, as stored in the parameters attribute
- wires (Iterable[Any], Wires) wires that the operator acts on
- **hyperparams** (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

list of diagonalizing gates

Return type

list[.Operator]

static compute_eigvals(*params, **hyperparams)

Eigenvalues of the operator in the computational basis (static method).

If diagonalizing_gates are specified and implement a unitary U^{\dagger} , the operator can be reconstructed as

$$O = U\Sigma U^{\dagger},$$

where Σ is the diagonal matrix containing the eigenvalues.

Otherwise, no particular order for the eigenvalues is guaranteed.

See also:

```
Operator.eigvals() and qml.eigvals()
```

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- **hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

eigenvalues

Return type

tensor_like

static compute_matrix(phi)

Representation of the operator as a canonical matrix in the computational basis (static method).

The canonical matrix is the textbook matrix representation that does not consider wires. Implicitly, this assumes that the wires of the operator correspond to the global wire order.

See also:

Operator.matrix() and qml.matrix()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- ****hyperparams** (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

matrix representation

Return type

tensor_like

static compute_sparse_matrix(*params, **hyperparams)

Representation of the operator as a sparse matrix in the computational basis (static method).

The canonical matrix is the textbook matrix representation that does not consider wires. Implicitly, this assumes that the wires of the operator correspond to the global wire order.

See also:

```
sparse_matrix()
```

152 Chapter 2. Tutorials

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- **hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

sparse matrix representation

Return type

scipy.sparse._csr.csr_matrix

decomposition()

Representation of the operator as a product of other operators.

$$O = O_1 O_2 \dots O_n$$

A DecompositionUndefinedError is raised if no representation by decomposition is defined.

See also:

compute_decomposition().

Returns

decomposition of the operator

Return type

list[Operator]

diagonalizing_gates()

Sequence of gates that diagonalize the operator in the computational basis.

Given the eigendecomposition $O=U\Sigma U^\dagger$ where Σ is a diagonal matrix containing the eigenvalues, the sequence of diagonalizing gates implements the unitary U^\dagger .

The diagonalizing gates rotate the state into the eigenbasis of the operator.

A DiagGatesUndefinedError is raised if no representation by decomposition is defined.

See also:

compute_diagonalizing_gates().

Returns

a list of operators

Return type

list[.Operator] or None

eigvals()

Eigenvalues of the operator in the computational basis.

If diagonalizing_gates are specified and implement a unitary U^{\dagger} , the operator can be reconstructed as

$$O = U\Sigma U^{\dagger}$$
.

where Σ is the diagonal matrix containing the eigenvalues.

Otherwise, no particular order for the eigenvalues is guaranteed.

Note: When eigenvalues are not explicitly defined, they are computed automatically from the matrix representation. Currently, this computation is *not* differentiable.

A EigvalsUndefinedError is raised if the eigenvalues have not been defined and cannot be inferred from the matrix representation.

See also:

compute_eigvals()

Returns

eigenvalues

Return type

tensor_like

expand()

Returns a tape that contains the decomposition of the operator.

Returns

quantum tape

Return type

.QuantumTape

generator()

Generator of an operator that is in single-parameter-form.

For example, for operator

$$U(\phi) = e^{i\phi(0.5Y + Z \otimes X)}$$

we get the generator

```
>>> U.generator()
    (0.5) [Y0]
+ (1.0) [Z0 X1]
```

The generator may also be provided in the form of a dense or sparse Hamiltonian (using Hermitian and SparseHamiltonian respectively).

The default value to return is None, indicating that the operation has no defined generator.

label(decimals=None, base_label=None, cache=None)

A customizable string representation of the operator.

Parameters

- **decimals=None** (*int*) If None, no parameters are included. Else, specifies how to round the parameters.
- base_label=None (str) overwrite the non-parameter component of the label
- cache=None (dict) dictionary that carries information between label calls in the same drawing

Returns

label to use in drawings

Return type

str

Example:

```
>>> op = qml.RX(1.23456, wires=0)
>>> op.label()
"RX"
>>> op.label(decimals=2)
"RX\n(1.23)"
>>> op.label(base_label="my_label")
"my_label"
>>> op.label(decimals=2, base_label="my_label")
"my_label\n(1.23)"
```

If the operation has a matrix-valued parameter and a cache dictionary is provided, unique matrices will be cached in the 'matrices' key list. The label will contain the index of the matrix in the 'matrices' list.

```
>>> op2 = qml.QubitUnitary(np.eye(2), wires=0)
>>> cache = {'matrices': []}
>>> op2.label(cache=cache)
'U(M0)'
>>> cache['matrices']
[tensor([[1., 0.],
[0., 1.]], requires_grad=True)]
>>> op3 = qml.QubitUnitary(np.eye(4), wires=(0,1))
>>> op3.label(cache=cache)
'U(M1)'
>>> cache['matrices']
[tensor([[1., 0.],
        [0., 1.]], requires_grad=True),
tensor([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]], requires_grad=True)]
```

map_wires(wire_map: dict)

Returns a copy of the current operator with its wires changed according to the given wire map.

Parameters

wire_map (dict) - dictionary containing the old wires as keys and the new wires as values

Returns

new operator

Return type

.Operator

matrix(wire_order=None)

Representation of the operator as a matrix in the computational basis.

If wire_order is provided, the numerical representation considers the position of the operator's wires in the global wire order. Otherwise, the wire order defaults to the operator's wires.

If the matrix depends on trainable parameters, the result will be cast in the same autodifferentiation framework as the parameters.

A MatrixUndefinedError is raised if the matrix representation has not been defined.

See also:

compute_matrix()

Parameters

wire_order (Iterable) – global wire order, must contain all wire labels from the operator's wires

Returns

matrix representation

Return type

tensor_like

$pow(z) \rightarrow List[Operator]$

A list of new operators equal to this one raised to the given power.

Parameters

z (*float*) – exponent for the operator

Returns

list[Operator]

queue(context=<class 'pennylane.queuing.QueuingManager'>)

Append the operator to the Operator queue.

$simplify() \rightarrow Operator$

Reduce the depth of nested operators to the minimum.

Returns

simplified operator

Return type

.Operator

single_qubit_rot_angles()

The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.

Returns

A list of values $[\phi, \theta, \omega]$ such that $RZ(\omega)RY(\theta)RZ(\phi)$ is equivalent to the original operation.

Return type

tuple[float, float, float]

sparse_matrix(wire_order=None)

Representation of the operator as a sparse matrix in the computational basis.

If wire_order is provided, the numerical representation considers the position of the operator's wires in the global wire order. Otherwise, the wire order defaults to the operator's wires.

A SparseMatrixUndefinedError is raised if the sparse matrix representation has not been defined.

See also:

```
compute_sparse_matrix()
```

Parameters

wire_order (*Iterable*) – global wire order, must contain all wire labels from the operator's wires

Returns

sparse matrix representation

156 Chapter 2. Tutorials

Return type

scipy.sparse._csr.csr_matrix

terms()

Representation of the operator as a linear combination of other operators.

$$O = \sum_{i} c_i O_i$$

A TermsUndefinedError is raised if no representation by terms is defined.

Returns

list of coefficients c_i and list of operations O_i

Return type

tuple[list[tensor_like or float], list[.Operation]]

static validate_subspace(subspace)

Validate the subspace for qutrit operations.

This method determines whether a given subspace for qutrit operations is defined correctly or not. If not, a *ValueError* is thrown.

Parameters

subspace (tuple[int]) - Subspace to check for correctness

MS

class MS(phi_0, phi_1, wires)

Bases: Operation

IonQ native Mølmer-Sørenson gate.

$$\mathrm{MS}(\phi_0,\phi_1) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 0 & -ie^{-i(\phi_0+\phi_1)} \\ 0 & 1 & -ie^{-i(\phi_0-\phi_1)} & 0 \\ 0 & -ie^{i(\phi_0-\phi_1)} & 1 & 0 \\ -ie^{i(\phi_0+\phi_1)} & 0 & 0 & 1 \end{bmatrix}.$$

Details:

- Number of wires: 2
- Number of parameters: 2

Parameters

- phi_0 (float) the first phase angle
- **phi_1** (*float*) the second phase angle
- wires (int) the subsystem the gate acts on
- **id** (*str or None*) String representing the operation (optional)

arithmetic_depth basis		
the target operation. Batch_size	arithmetic_depth	
$ \begin{array}{c} {\rm casted\ parameters.} \\ {\rm control_wires} \\ {\rm grad_method} \\ \\ {\rm grad_recipe} \\ {\rm has_adjoint} \\ \\ {\rm has_decomposition} \\ \\ {\rm has_deadjoint} \\ \\ {\rm has_deadjoint$	basis	
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	batch_size	
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	control_wires	Control wires of the operator.
$\begin{array}{lll} has_adjoint \\ has_decomposition \\ has_diagonalizing_gates \\ has_generator \\ hash & Integer hash that uniquely represents the operator. \\ hyperparameters & Dictionary of non-trainable variables that this operation depends on. \\ id & Custom string to label a specific operator instance. \\ is_hermitian & This property determines if an operator is hermitian. \\ name & String for the name of the operator. \\ ndim_params & Number of dimensions per trainable parameter of the operator. \\ num_params & Number of wires the operator acts on. \\ parameter_frequencies & Returns the frequencies for each operator parameter with respect to an expectation value of the form \langle \psi U(\mathbf{p})^{\dagger} \hat{O}U(\mathbf{p}) \psi \rangle. parameters & Trainable parameters that the operator depends on.$	<pre>grad_method</pre>	
$\begin{array}{lll} has_decomposition \\ has_diagonalizing_gates \\ has_generator \\ has_matrix \\ hash & Integer hash that uniquely represents the operator. \\ hyperparameters & Dictionary of non-trainable variables that this operation depends on. \\ id & Custom string to label a specific operator instance. \\ is_hermitian & This property determines if an operator is hermitian. \\ name & String for the name of the operator. \\ ndim_params & Number of dimensions per trainable parameter of the operator. \\ num_params & Number of wires the operator acts on. \\ parameter_frequencies & Returns the frequencies for each operator parameter with respect to an expectation value of the form \langle \psi U(\mathbf{p})^{\dagger} \hat{O}U(\mathbf{p}) \psi \rangle. \\ parameters & Trainable parameters that the operator depends on. \\ \end{array}$		Gradient recipe for the parameter-shift method.
$\begin{array}{lll} has_diagonalizing_gates \\ has_generator \\ \\ has_matrix \\ \\ hash & Integer hash that uniquely represents the operator. \\ hyperparameters & Dictionary of non-trainable variables that this operation depends on. \\ id & Custom string to label a specific operator instance. \\ is_hermitian & This property determines if an operator is hermitian. \\ name & String for the name of the operator. \\ ndim_params & Number of dimensions per trainable parameter of the operator. \\ num_params & Number of wires the operator acts on. \\ parameter_frequencies & Returns the frequencies for each operator parameter with respect to an expectation value of the form \langle \psi U(\mathbf{p})^\dagger \hat{O}U(\mathbf{p}) \psi \rangle. \\ parameters & Trainable parameters that the operator depends on. \\ \hline \end{array}$	has_adjoint	
$\begin{array}{lll} has_generator \\ has_matrix \\ hash & Integer hash that uniquely represents the operator. \\ hyperparameters & Dictionary of non-trainable variables that this operation depends on. \\ id & Custom string to label a specific operator instance. \\ is_hermitian & This property determines if an operator is hermitian. \\ name & String for the name of the operator. \\ ndim_params & Number of dimensions per trainable parameter of the operator. \\ num_params & Number of wires the operator acts on. \\ parameter_frequencies & Returns the frequencies for each operator parameter with respect to an expectation value of the form \langle \psi U(\mathbf{p})^\dagger \hat{O}U(\mathbf{p}) \psi \rangle. \\ parameters & Trainable parameters that the operator depends on. \\ \hline \end{array}$	has_decomposition	
$\begin{array}{lll} has \\ hash \\ hyperparameters \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\$	has_diagonalizing_gates	
hashInteger hash that uniquely represents the operator.hyperparametersDictionary of non-trainable variables that this opera- tion depends on.idCustom string to label a specific operator instance. $is_hermitian$ This property determines if an operator is hermitian.nameString for the name of the operator. $ndim_params$ Number of dimensions per trainable parameter of the operator. num_params Number of wires the operator acts on. $parameter_frequencies$ Returns the frequencies for each operator parameter with respect to an expectation value of the form $\langle \psi U(\mathbf{p})^{\dagger} \hat{O}U(\mathbf{p}) \psi \rangle$. $parameters$ Trainable parameters that the operator depends on.	has_generator	
$\begin{array}{ll} \textit{hyperparameters} & \text{Dictionary of non-trainable variables that this operation depends on.} \\ \textit{id} & \text{Custom string to label a specific operator instance.} \\ \textit{is_hermitian} & \text{This property determines if an operator is hermitian.} \\ \textit{name} & \text{String for the name of the operator.} \\ \textit{ndim_params} & \text{Number of dimensions per trainable parameter of the operator.} \\ \textit{num_params} & \text{Number of wires the operator acts on.} \\ \textit{parameter_frequencies} & \text{Returns the frequencies for each operator parameter with respect to an expectation value of the form} \\ & \langle \psi U(\mathbf{p})^\dagger \hat{O} U(\mathbf{p}) \psi \rangle. \\ \textit{parameters} & \text{Trainable parameters that the operator depends on.} \\ \end{aligned}$	has_matrix	
tion depends on. id Custom string to label a specific operator instance. is_hermitian This property determines if an operator is hermitian. name String for the name of the operator. ndim_params Number of dimensions per trainable parameter of the operator. num_params num_wires Number of wires the operator acts on. parameter_frequencies Returns the frequencies for each operator parameter with respect to an expectation value of the form $\langle \psi U(\mathbf{p})^{\dagger} \hat{O} U(\mathbf{p}) \psi \rangle$. parameters Trainable parameters that the operator depends on.	hash	Integer hash that uniquely represents the operator.
$\begin{array}{ll} \textit{is_hermitian} & \text{This property determines if an operator is hermitian.} \\ \textit{name} & \text{String for the name of the operator.} \\ \textit{ndim_params} & \text{Number of dimensions per trainable parameter of the operator.} \\ \textit{num_params} & \text{Number of wires the operator acts on.} \\ \textit{parameter_frequencies} & \text{Returns the frequencies for each operator parameter with respect to an expectation value of the form} \\ & \langle \psi U(\mathbf{p})^\dagger \hat{O} U(\mathbf{p}) \psi \rangle. \\ \textit{parameters} & \text{Trainable parameters that the operator depends on.} \\ \end{array}$	hyperparameters	•
name String for the name of the operator.	id	
$\begin{array}{c} \textit{ndim_params} & \textit{Number of dimensions per trainable parameter of the} \\ \textit{operator.} \\ \\ \textit{num_params} \\ \\ \\ \textit{num_wires} & \textit{Number of wires the operator acts on.} \\ \textit{parameter_frequencies} & \textit{Returns the frequencies for each operator parameter with respect to an expectation value of the form} \\ & \langle \psi U(\mathbf{p})^{\dagger} \hat{O} U(\mathbf{p}) \psi \rangle. \\ \\ \textit{parameters} & \textit{Trainable parameters that the operator depends on.} \\ \end{array}$	is_hermitian	
$\begin{array}{c} \text{operator.} \\ \text{num_params} \\ \\ \\ \hline $		
$\begin{array}{ll} \textit{num_wires} & \text{Number of wires the operator acts on.} \\ \textit{parameter_frequencies} & \text{Returns the frequencies for each operator parameter with respect to an expectation value of the form} \\ & \langle \psi U(\mathbf{p})^\dagger \hat{O} U(\mathbf{p}) \psi \rangle. \\ \textit{parameters} & \text{Trainable parameters that the operator depends on.} \\ \end{array}$	ndim_params	<u> </u>
$\begin{array}{ccc} \textit{parameter_frequencies} & \text{Returns the frequencies for each operator parameter with respect to an expectation value of the form} \\ & & \langle \psi U(\mathbf{p})^\dagger \hat{O} U(\mathbf{p}) \psi \rangle. \\ \textit{parameters} & \text{Trainable parameters that the operator depends on.} \end{array}$	num_params	
$\begin{array}{c} \text{ter with respect to an expectation value of the form} \\ \langle \psi U(\mathbf{p})^\dagger \hat{O} U(\mathbf{p}) \psi \rangle. \\ \\ \textit{parameters} \end{array}$	num_wires	
	parameter_frequencies	ter with respect to an expectation value of the form
wires Wires that the operator acts on.	parameters	
	wires	Wires that the operator acts on.

arithmetic_depth

Arithmetic depth of the operator.

basis = None

The basis of an operation, or for controlled gates, of the target operation. If not None, should take a value of "X", "Y", or "Z".

For example, X and CNOT have basis = "X", whereas ControlledPhaseShift and RZ have basis = "Z".

Type

str or None

batch_size

Batch size of the operator if it is used with broadcasted parameters.

The batch_size is determined based on ndim_params and the provided parameters for the operator. If (some of) the latter have an additional dimension, and this dimension has the same size for all parameters, its size is the batch size of the operator. If no parameter has an additional dimension, the batch size is None.

158 Chapter 2. Tutorials

Returns

Size of the parameter broadcasting dimension if present, else None.

Return type

int or None

control_wires

Control wires of the operator.

For operations that are not controlled, this is an empty Wires object of length 0.

Returns

The control wires of the operation.

Return type

Wires

grad_method = 'F'

grad_recipe = None

Gradient recipe for the parameter-shift method.

This is a tuple with one nested list per operation parameter. For parameter ϕ_k , the nested list contains elements of the form $[c_i, a_i, s_i]$ where i is the index of the term, resulting in a gradient recipe of

$$\frac{\partial}{\partial \phi_k} f = \sum_i c_i f(a_i \phi_k + s_i).$$

If None, the default gradient recipe containing the two terms $[c_0, a_0, s_0] = [1/2, 1, \pi/2]$ and $[c_1, a_1, s_1] = [-1/2, 1, -\pi/2]$ is assumed for every parameter.

Type

tuple(Union(list[list[float]], None)) or None

has_adjoint = True

has_decomposition = False

has_diagonalizing_gates = False

has_generator = False

has matrix = True

hash

Integer hash that uniquely represents the operator.

Type

int

hyperparameters

Dictionary of non-trainable variables that this operation depends on.

Type

dict

id

Custom string to label a specific operator instance.

is_hermitian

This property determines if an operator is hermitian.

name

String for the name of the operator.

ndim_params

Number of dimensions per trainable parameter of the operator.

By default, this property returns the numbers of dimensions of the parameters used for the operator creation. If the parameter sizes for an operator subclass are fixed, this property can be overwritten to return the fixed value.

Returns

Number of dimensions for each trainable parameter.

Return type

tuple

```
num_params = 2
```

$num_wires = 2$

Number of wires the operator acts on.

parameter_frequencies

Returns the frequencies for each operator parameter with respect to an expectation value of the form $\langle \psi | U(\mathbf{p})^{\dagger} \hat{O} U(\mathbf{p}) | \psi \rangle$.

These frequencies encode the behaviour of the operator $U(\mathbf{p})$ on the value of the expectation value as the parameters are modified. For more details, please see the pennylane. fourier module.

Returns

Tuple of frequencies for each parameter. Note that only non-negative frequency values are returned.

Return type

list[tuple[int or float]]

Example

```
>>> op = qml.CRot(0.4, 0.1, 0.3, wires=[0, 1])
>>> op.parameter_frequencies
[(0.5, 1), (0.5, 1), (0.5, 1)]
```

For operators that define a generator, the parameter frequencies are directly related to the eigenvalues of the generator:

```
>>> op = qml.ControlledPhaseShift(0.1, wires=[0, 1])
>>> op.parameter_frequencies
[(1,)]
>>> gen = qml.generator(op, format="observable")
>>> gen_eigvals = qml.eigvals(gen)
>>> qml.gradients.eigvals_to_frequencies(tuple(gen_eigvals))
(1.0,)
```

For more details on this relationship, see eigvals_to_frequencies().

parameters

Trainable parameters that the operator depends on.

wires

Wires that the operator acts on.

Returns

wires

Return type

Wires

adjoint()	Create an operation that is the adjoint of this one.
<pre>compute_decomposition(*params[, wires])</pre>	Representation of the operator as a product of other operators (static method).
<pre>compute_diagonalizing_gates(*params, wires,)</pre>	Sequence of gates that diagonalize the operator in the computational basis (static method).
<pre>compute_eigvals(*params, **hyperparams)</pre>	Eigenvalues of the operator in the computational basis (static method).
<pre>compute_matrix(phi_0, phi_1)</pre>	Representation of the operator as a canonical matrix in the computational basis (static method).
<pre>compute_sparse_matrix(*params, **hyper- params)</pre>	Representation of the operator as a sparse matrix in the computational basis (static method).
<pre>decomposition()</pre>	Representation of the operator as a product of other operators.
<pre>diagonalizing_gates()</pre>	Sequence of gates that diagonalize the operator in the computational basis.
eigvals()	Eigenvalues of the operator in the computational basis.
expand()	Returns a tape that contains the decomposition of the operator.
generator()	Generator of an operator that is in single-parameter- form.
<pre>label([decimals, base_label, cache])</pre>	A customizable string representation of the operator.
<pre>map_wires(wire_map)</pre>	Returns a copy of the current operator with its wires changed according to the given wire map.
<pre>matrix([wire_order])</pre>	Representation of the operator as a matrix in the computational basis.
pow(z)	A list of new operators equal to this one raised to the given power.
queue([context])	Append the operator to the Operator queue.
<pre>simplify()</pre>	Reduce the depth of nested operators to the minimum.
single_qubit_rot_angles()	The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.
<pre>sparse_matrix([wire_order])</pre>	Representation of the operator as a sparse matrix in the computational basis.
terms()	Representation of the operator as a linear combination of other operators.
validate_subspace(subspace)	Validate the subspace for qutrit operations.

adjoint()

Create an operation that is the adjoint of this one.

Adjointed operations are the conjugated and transposed version of the original operation. Adjointed ops are equivalent to the inverted operation for unitary gates.

Returns

The adjointed operation.

static compute_decomposition(*params, wires=None, **hyperparameters)

Representation of the operator as a product of other operators (static method).

$$O = O_1 O_2 \dots O_n$$
.

Note: Operations making up the decomposition should be queued within the compute_decomposition method.

See also:

decomposition().

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- wires (Iterable[Any], Wires) wires that the operator acts on
- **hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

decomposition of the operator

Return type

list[Operator]

static compute_diagonalizing_gates(*params, wires, **hyperparams)

Sequence of gates that diagonalize the operator in the computational basis (static method).

Given the eigendecomposition $O=U\Sigma U^{\dagger}$ where Σ is a diagonal matrix containing the eigenvalues, the sequence of diagonalizing gates implements the unitary U^{\dagger} .

The diagonalizing gates rotate the state into the eigenbasis of the operator.

See also:

diagonalizing_gates().

Parameters

- ullet params (list) trainable parameters of the operator, as stored in the parameters attribute
- wires (Iterable[Any], Wires) wires that the operator acts on
- **hyperparams** (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

list of diagonalizing gates

Return type

list[.Operator]

static compute_eigvals(*params, **hyperparams)

Eigenvalues of the operator in the computational basis (static method).

If diagonalizing_gates are specified and implement a unitary U^{\dagger} , the operator can be reconstructed as

$$O = U\Sigma U^{\dagger}$$
.

where Σ is the diagonal matrix containing the eigenvalues.

Otherwise, no particular order for the eigenvalues is guaranteed.

See also:

Operator.eigvals() and qml.eigvals()

Parameters

- ullet *params (list) trainable parameters of the operator, as stored in the parameters attribute
- **hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

eigenvalues

Return type

tensor_like

static compute_matrix(phi_0, phi_1)

Representation of the operator as a canonical matrix in the computational basis (static method).

The canonical matrix is the textbook matrix representation that does not consider wires. Implicitly, this assumes that the wires of the operator correspond to the global wire order.

See also:

Operator.matrix() and qml.matrix()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- ****hyperparams** (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

matrix representation

Return type

tensor_like

static compute_sparse_matrix(*params, **hyperparams)

Representation of the operator as a sparse matrix in the computational basis (static method).

The canonical matrix is the textbook matrix representation that does not consider wires. Implicitly, this assumes that the wires of the operator correspond to the global wire order.

See also:

sparse_matrix()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- ****hyperparams** (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

sparse matrix representation

Return type

scipy.sparse._csr.csr_matrix

decomposition()

Representation of the operator as a product of other operators.

$$O = O_1 O_2 \dots O_n$$

A DecompositionUndefinedError is raised if no representation by decomposition is defined.

See also:

compute_decomposition().

Returns

decomposition of the operator

Return type

list[Operator]

diagonalizing_gates()

Sequence of gates that diagonalize the operator in the computational basis.

Given the eigendecomposition $O=U\Sigma U^\dagger$ where Σ is a diagonal matrix containing the eigenvalues, the sequence of diagonalizing gates implements the unitary U^\dagger .

The diagonalizing gates rotate the state into the eigenbasis of the operator.

A DiagGatesUndefinedError is raised if no representation by decomposition is defined.

See also:

compute_diagonalizing_gates().

Returns

a list of operators

Return type

list[.Operator] or None

eigvals()

164

Eigenvalues of the operator in the computational basis.

If diagonalizing_gates are specified and implement a unitary U^{\dagger} , the operator can be reconstructed as

$$O = U\Sigma U^{\dagger}$$
,

where Σ is the diagonal matrix containing the eigenvalues.

Otherwise, no particular order for the eigenvalues is guaranteed.

Chapter 2. Tutorials

Note: When eigenvalues are not explicitly defined, they are computed automatically from the matrix representation. Currently, this computation is *not* differentiable.

A EigvalsUndefinedError is raised if the eigenvalues have not been defined and cannot be inferred from the matrix representation.

See also:

compute_eigvals()

Returns

eigenvalues

Return type

tensor_like

expand()

Returns a tape that contains the decomposition of the operator.

Returns

quantum tape

Return type

.QuantumTape

generator()

Generator of an operator that is in single-parameter-form.

For example, for operator

$$U(\phi) = e^{i\phi(0.5Y + Z \otimes X)}$$

we get the generator

```
>>> U.generator()
  (0.5) [Y0]
+ (1.0) [Z0 X1]
```

The generator may also be provided in the form of a dense or sparse Hamiltonian (using Hermitian and SparseHamiltonian respectively).

The default value to return is None, indicating that the operation has no defined generator.

label(decimals=None, base_label=None, cache=None)

A customizable string representation of the operator.

Parameters

- **decimals=None** (*int*) If None, no parameters are included. Else, specifies how to round the parameters.
- base_label=None (str) overwrite the non-parameter component of the label
- **cache=None** (*dict*) dictionary that carries information between label calls in the same drawing

Returns

label to use in drawings

Return type str

Example:

```
>>> op = qml.RX(1.23456, wires=0)
>>> op.label()
"RX"
>>> op.label(decimals=2)
"RX\n(1.23)"
>>> op.label(base_label="my_label")
"my_label"
>>> op.label(decimals=2, base_label="my_label")
"my_label\n(1.23)"
```

If the operation has a matrix-valued parameter and a cache dictionary is provided, unique matrices will be cached in the 'matrices' key list. The label will contain the index of the matrix in the 'matrices' list.

```
>>> op2 = qml.QubitUnitary(np.eye(2), wires=0)
>>> cache = {'matrices': []}
>>> op2.label(cache=cache)
'U(M0)'
>>> cache['matrices']
[tensor([[1., 0.],
[0., 1.]], requires_grad=True)]
>>> op3 = qml.QubitUnitary(np.eye(4), wires=(0,1))
>>> op3.label(cache=cache)
'U(M1)'
>>> cache['matrices']
[tensor([[1., 0.],
        [0., 1.]], requires_grad=True),
tensor([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]], requires_grad=True)]
```

map_wires(wire_map: dict)

Returns a copy of the current operator with its wires changed according to the given wire map.

Parameters

wire_map (dict) - dictionary containing the old wires as keys and the new wires as values

Returns

new operator

Return type

.Operator

matrix(wire_order=None)

166

Representation of the operator as a matrix in the computational basis.

If wire_order is provided, the numerical representation considers the position of the operator's wires in the global wire order. Otherwise, the wire order defaults to the operator's wires.

If the matrix depends on trainable parameters, the result will be cast in the same autodifferentiation framework as the parameters.

A MatrixUndefinedError is raised if the matrix representation has not been defined.

See also:

compute_matrix()

Parameters

wire_order (*Iterable*) – global wire order, must contain all wire labels from the operator's wires

Returns

matrix representation

Return type

tensor_like

$pow(z) \rightarrow List[Operator]$

A list of new operators equal to this one raised to the given power.

Parameters

z (*float*) – exponent for the operator

Returns

list[Operator]

queue(context=<class 'pennylane.queuing.QueuingManager'>)

Append the operator to the Operator queue.

$simplify() \rightarrow Operator$

Reduce the depth of nested operators to the minimum.

Returns

simplified operator

Return type

.Operator

single_qubit_rot_angles()

The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.

Returns

A list of values $[\phi, \theta, \omega]$ such that $RZ(\omega)RY(\theta)RZ(\phi)$ is equivalent to the original operation.

Return type

tuple[float, float, float]

sparse_matrix(wire_order=None)

Representation of the operator as a sparse matrix in the computational basis.

If wire_order is provided, the numerical representation considers the position of the operator's wires in the global wire order. Otherwise, the wire order defaults to the operator's wires.

A SparseMatrixUndefinedError is raised if the sparse matrix representation has not been defined.

See also:

compute_sparse_matrix()

Parameters

wire_order (*Iterable*) – global wire order, must contain all wire labels from the operator's wires

Returns

sparse matrix representation

Amazon Braket PennyLane Plugin Documentation, Release 1.22.1.dev0

Return type

scipy.sparse._csr.csr_matrix

terms()

Representation of the operator as a linear combination of other operators.

$$O = \sum_{i} c_i O_i$$

A TermsUndefinedError is raised if no representation by terms is defined.

Returns

list of coefficients c_i and list of operations O_i

Return type

tuple[list[tensor_like or float], list[.Operation]]

static validate_subspace(subspace)

Validate the subspace for qutrit operations.

This method determines whether a given subspace for qutrit operations is defined correctly or not. If not, a *ValueError* is thrown.

Parameters

subspace (tuple[int]) - Subspace to check for correctness

PSWAP

class PSWAP(phi, wires)

Bases: Operation

Phase-SWAP gate.

$$\mathrm{PSWAP}(\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & e^{i\phi} & 0 \\ 0 & e^{i\phi} & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Details:

• Number of wires: 2

• Number of parameters: 1

• Gradient recipe:

$$\frac{d}{d\phi} \mathtt{PSWAP}(\phi) = \frac{1}{2} \left[\mathtt{PSWAP}(\phi + \pi/2) - \mathtt{PSWAP}(\phi - \pi/2) \right]$$

Parameters

- **phi** (float) the phase angle
- wires (int) the subsystem the gate acts on
- **id** (*str or None*) String representing the operation (optional)

arithmetic_depth	Arithmetic depth of the operator.
basis	The basis of an operation, or for controlled gates, of
	the target operation.
batch_size	Batch size of the operator if it is used with broad-
	casted parameters.
control_wires	Control wires of the operator.
grad_method	1 · · · · · · · · · · · · · · · · · · ·
92 442	
grad_recipe	Gradient recipe for the parameter-shift method.
has_adjoint	Gradient recipe for the parameter-shift method.
nas_aujoint	
han dansumanitinu	
has_decomposition	
has_diagonalizing_gates	
has_generator	
has_matrix	
hash	Integer hash that uniquely represents the operator.
hyperparameters	Dictionary of non-trainable variables that this opera-
7	tion depends on.
id	Custom string to label a specific operator instance.
is hermitian	This property determines if an operator is hermitian.
name	String for the name of the operator.
	Number of dimensions per trainable parameter of the
ndim_params	
	operator.
num_params	
num_wires	Number of wires the operator acts on.
parameter_frequencies	Returns the frequencies for each operator parame-
	ter with respect to an expectation value of the form
	$\langle \psi U(\mathbf{p})^{\dagger} \hat{O} U(\mathbf{p}) \psi \rangle.$
parameters	Trainable parameters that the operator depends on.
wires	Wires that the operator acts on.
	1

arithmetic_depth

Arithmetic depth of the operator.

basis = None

The basis of an operation, or for controlled gates, of the target operation. If not None, should take a value of "X", "Y", or "Z".

For example, X and CNOT have basis = "X", whereas ControlledPhaseShift and RZ have basis = "Z".

Type

str or None

batch_size

Batch size of the operator if it is used with broadcasted parameters.

The batch_size is determined based on ndim_params and the provided parameters for the operator. If (some of) the latter have an additional dimension, and this dimension has the same size for all parameters, its size is the batch size of the operator. If no parameter has an additional dimension, the batch size is None.

Returns

Size of the parameter broadcasting dimension if present, else None.

Return type

int or None

control_wires

Control wires of the operator.

For operations that are not controlled, this is an empty Wires object of length 0.

Returns

The control wires of the operation.

Return type

Wires

grad_method = 'A'

Gradient recipe for the parameter-shift method.

This is a tuple with one nested list per operation parameter. For parameter ϕ_k , the nested list contains elements of the form $[c_i, a_i, s_i]$ where i is the index of the term, resulting in a gradient recipe of

$$\frac{\partial}{\partial \phi_k} f = \sum_i c_i f(a_i \phi_k + s_i).$$

If None, the default gradient recipe containing the two terms $[c_0, a_0, s_0] = [1/2, 1, \pi/2]$ and $[c_1, a_1, s_1] = [-1/2, 1, -\pi/2]$ is assumed for every parameter.

Type

tuple(Union(list[list[float]], None)) or None

has_adjoint = True

has_decomposition = True

has_diagonalizing_gates = False

has_generator = False

has matrix = True

hash

Integer hash that uniquely represents the operator.

Type

int

hyperparameters

Dictionary of non-trainable variables that this operation depends on.

Type

dict

id

Custom string to label a specific operator instance.

is_hermitian

This property determines if an operator is hermitian.

name

String for the name of the operator.

ndim_params

Number of dimensions per trainable parameter of the operator.

By default, this property returns the numbers of dimensions of the parameters used for the operator creation. If the parameter sizes for an operator subclass are fixed, this property can be overwritten to return the fixed value.

Returns

Number of dimensions for each trainable parameter.

Return type

tuple

```
num_params = 1
```

$num_wires = 2$

Number of wires the operator acts on.

parameter_frequencies

Returns the frequencies for each operator parameter with respect to an expectation value of the form $\langle \psi | U(\mathbf{p})^{\dagger} \hat{O} U(\mathbf{p}) | \psi \rangle$.

These frequencies encode the behaviour of the operator $U(\mathbf{p})$ on the value of the expectation value as the parameters are modified. For more details, please see the pennylane. fourier module.

Returns

Tuple of frequencies for each parameter. Note that only non-negative frequency values are returned.

Return type

list[tuple[int or float]]

Example

```
>>> op = qml.CRot(0.4, 0.1, 0.3, wires=[0, 1])
>>> op.parameter_frequencies
[(0.5, 1), (0.5, 1), (0.5, 1)]
```

For operators that define a generator, the parameter frequencies are directly related to the eigenvalues of the generator:

```
>>> op = qml.ControlledPhaseShift(0.1, wires=[0, 1])
>>> op.parameter_frequencies
[(1,)]
>>> gen = qml.generator(op, format="observable")
>>> gen_eigvals = qml.eigvals(gen)
>>> qml.gradients.eigvals_to_frequencies(tuple(gen_eigvals))
(1.0,)
```

For more details on this relationship, see eigvals_to_frequencies().

parameters

Trainable parameters that the operator depends on.

wires

Wires that the operator acts on.

Returns

wires

Return type

Wires

adjoint()	Create an operation that is the adjoint of this one.
<pre>compute_decomposition(phi, wires)</pre>	Representation of the operator as a product of other operators (static method).
<pre>compute_diagonalizing_gates(*params, wires</pre>	
)	computational basis (static method).
<pre>compute_eigvals(*params, **hyperparams)</pre>	Eigenvalues of the operator in the computational basis (static method).
<pre>compute_matrix(phi)</pre>	Representation of the operator as a canonical matrix in the computational basis (static method).
<pre>compute_sparse_matrix(*params, **hyper params)</pre>	- Representation of the operator as a sparse matrix in the computational basis (static method).
decomposition()	Representation of the operator as a product of other operators.
diagonalizing_gates()	Sequence of gates that diagonalize the operator in the computational basis.
eigvals()	Eigenvalues of the operator in the computational basis.
expand()	Returns a tape that contains the decomposition of the operator.
generator()	Generator of an operator that is in single-parameter- form.
<pre>label([decimals, base_label, cache])</pre>	A customizable string representation of the operator.
<pre>map_wires(wire_map)</pre>	Returns a copy of the current operator with its wires changed according to the given wire map.
<pre>matrix([wire_order])</pre>	Representation of the operator as a matrix in the computational basis.
pow(z)	A list of new operators equal to this one raised to the given power.
queue([context])	Append the operator to the Operator queue.
simplify()	Reduce the depth of nested operators to the minimum.
<pre>single_qubit_rot_angles()</pre>	The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.
<pre>sparse_matrix([wire_order])</pre>	Representation of the operator as a sparse matrix in the computational basis.
terms()	Representation of the operator as a linear combination of other operators.
<pre>validate_subspace(subspace)</pre>	Validate the subspace for qutrit operations.

adjoint()

Create an operation that is the adjoint of this one.

Adjointed operations are the conjugated and transposed version of the original operation. Adjointed ops are equivalent to the inverted operation for unitary gates.

Returns

The adjointed operation.

static compute_decomposition(phi, wires)

172 Chapter 2. Tutorials

Representation of the operator as a product of other operators (static method).

$$O = O_1 O_2 \dots O_n$$
.

Note: Operations making up the decomposition should be queued within the compute_decomposition method.

See also:

decomposition().

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- wires (Iterable[Any], Wires) wires that the operator acts on
- **hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

decomposition of the operator

Return type

list[Operator]

static compute_diagonalizing_gates(*params, wires, **hyperparams)

Sequence of gates that diagonalize the operator in the computational basis (static method).

Given the eigendecomposition $O=U\Sigma U^\dagger$ where Σ is a diagonal matrix containing the eigenvalues, the sequence of diagonalizing gates implements the unitary U^\dagger .

The diagonalizing gates rotate the state into the eigenbasis of the operator.

See also:

diagonalizing_gates().

Parameters

- ullet params (list) trainable parameters of the operator, as stored in the parameters attribute
- wires (Iterable[Any], Wires) wires that the operator acts on
- **hyperparams** (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

list of diagonalizing gates

Return type

list[.Operator]

static compute_eigvals(*params, **hyperparams)

Eigenvalues of the operator in the computational basis (static method).

If diagonalizing_gates are specified and implement a unitary U^{\dagger} , the operator can be reconstructed as

$$O = U\Sigma U^{\dagger},$$

where Σ is the diagonal matrix containing the eigenvalues.

Otherwise, no particular order for the eigenvalues is guaranteed.

See also:

Operator.eigvals() and qml.eigvals()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- **hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

eigenvalues

Return type

tensor_like

static compute_matrix(phi)

Representation of the operator as a canonical matrix in the computational basis (static method).

The canonical matrix is the textbook matrix representation that does not consider wires. Implicitly, this assumes that the wires of the operator correspond to the global wire order.

See also:

Operator.matrix() and qml.matrix()

Parameters

- *params (list) trainable parameters of the operator, as stored in the parameters attribute
- ****hyperparams** (*dict*) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

matrix representation

Return type

tensor_like

static compute_sparse_matrix(*params, **hyperparams)

Representation of the operator as a sparse matrix in the computational basis (static method).

The canonical matrix is the textbook matrix representation that does not consider wires. Implicitly, this assumes that the wires of the operator correspond to the global wire order.

See also:

sparse_matrix()

Parameters

- ullet *params (list) trainable parameters of the operator, as stored in the parameters attribute
- **hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns

sparse matrix representation

Return type

scipy.sparse._csr.csr_matrix

decomposition()

Representation of the operator as a product of other operators.

$$O = O_1 O_2 \dots O_n$$

A DecompositionUndefinedError is raised if no representation by decomposition is defined.

See also:

compute_decomposition().

Returns

decomposition of the operator

Return type

list[Operator]

diagonalizing_gates()

Sequence of gates that diagonalize the operator in the computational basis.

Given the eigendecomposition $O=U\Sigma U^\dagger$ where Σ is a diagonal matrix containing the eigenvalues, the sequence of diagonalizing gates implements the unitary U^\dagger .

The diagonalizing gates rotate the state into the eigenbasis of the operator.

A DiagGatesUndefinedError is raised if no representation by decomposition is defined.

See also:

compute_diagonalizing_gates().

Returns

a list of operators

Return type

list[.Operator] or None

eigvals()

Eigenvalues of the operator in the computational basis.

If diagonalizing_gates are specified and implement a unitary U^{\dagger} , the operator can be reconstructed as

$$O = U\Sigma U^{\dagger}$$
.

where Σ is the diagonal matrix containing the eigenvalues.

Otherwise, no particular order for the eigenvalues is guaranteed.

Note: When eigenvalues are not explicitly defined, they are computed automatically from the matrix representation. Currently, this computation is *not* differentiable.

A EigvalsUndefinedError is raised if the eigenvalues have not been defined and cannot be inferred from the matrix representation.

See also:

compute_eigvals()

Returns

eigenvalues

Return type

tensor_like

expand()

Returns a tape that contains the decomposition of the operator.

Returns

quantum tape

Return type

.QuantumTape

generator()

Generator of an operator that is in single-parameter-form.

For example, for operator

$$U(\phi) = e^{i\phi(0.5Y + Z \otimes X)}$$

we get the generator

```
>>> U.generator()
  (0.5) [Y0]
+ (1.0) [Z0 X1]
```

The generator may also be provided in the form of a dense or sparse Hamiltonian (using Hermitian and SparseHamiltonian respectively).

The default value to return is None, indicating that the operation has no defined generator.

label(decimals=None, base_label=None, cache=None)

A customizable string representation of the operator.

Parameters

- **decimals=None** (*int*) If None, no parameters are included. Else, specifies how to round the parameters.
- base_label=None (str) overwrite the non-parameter component of the label
- **cache=None** (*dict*) dictionary that carries information between label calls in the same drawing

Returns

label to use in drawings

Return type

str

Example:

```
>>> op = qml.RX(1.23456, wires=0)
>>> op.label()
"RX"
>>> op.label(decimals=2)
"RX\n(1.23)"
>>> op.label(base_label="my_label")
"my_label"
>>> op.label(decimals=2, base_label="my_label")
"my_label\n(1.23)"
```

If the operation has a matrix-valued parameter and a cache dictionary is provided, unique matrices will be cached in the 'matrices' key list. The label will contain the index of the matrix in the 'matrices' list.

```
>>> op2 = qml.QubitUnitary(np.eye(2), wires=0)
>>> cache = {'matrices': []}
>>> op2.label(cache=cache)
'U(M0)'
>>> cache['matrices']
[tensor([[1., 0.],
[0., 1.]], requires_grad=True)]
>>> op3 = qml.QubitUnitary(np.eye(4), wires=(0,1))
>>> op3.label(cache=cache)
'U(M1)'
>>> cache['matrices']
[tensor([[1., 0.],
        [0., 1.]], requires_grad=True),
tensor([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]], requires_grad=True)]
```

map_wires(wire_map: dict)

Returns a copy of the current operator with its wires changed according to the given wire map.

Parameters

wire_map (dict) - dictionary containing the old wires as keys and the new wires as values

Returns

new operator

Return type

.Operator

matrix(wire_order=None)

Representation of the operator as a matrix in the computational basis.

If wire_order is provided, the numerical representation considers the position of the operator's wires in the global wire order. Otherwise, the wire order defaults to the operator's wires.

If the matrix depends on trainable parameters, the result will be cast in the same autodifferentiation framework as the parameters.

A MatrixUndefinedError is raised if the matrix representation has not been defined.

See also:

compute_matrix()

Parameters

wire_order (*Iterable*) – global wire order, must contain all wire labels from the operator's wires

Returns

matrix representation

Return type

tensor_like

$pow(z) \rightarrow List[Operator]$

A list of new operators equal to this one raised to the given power.

Parameters

z (float) – exponent for the operator

Returns

list[Operator]

queue(context=<class 'pennylane.queuing.QueuingManager'>)

Append the operator to the Operator queue.

simplify() \rightarrow Operator

Reduce the depth of nested operators to the minimum.

Returns

simplified operator

Return type

.Operator

single_qubit_rot_angles()

The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.

Returns

A list of values $[\phi, \theta, \omega]$ such that $RZ(\omega)RY(\theta)RZ(\phi)$ is equivalent to the original operation.

Return type

tuple[float, float, float]

sparse_matrix(wire_order=None)

Representation of the operator as a sparse matrix in the computational basis.

If wire_order is provided, the numerical representation considers the position of the operator's wires in the global wire order. Otherwise, the wire order defaults to the operator's wires.

A SparseMatrixUndefinedError is raised if the sparse matrix representation has not been defined.

See also:

compute_sparse_matrix()

Parameters

wire_order (*Iterable*) – global wire order, must contain all wire labels from the operator's wires

Returns

sparse matrix representation

178 Chapter 2. Tutorials

Return type

scipy.sparse._csr.csr_matrix

terms()

Representation of the operator as a linear combination of other operators.

$$O = \sum_{i} c_i O_i$$

A TermsUndefinedError is raised if no representation by terms is defined.

Returns

list of coefficients c_i and list of operations O_i

Return type

tuple[list[tensor_like or float], list[.Operation]]

static validate_subspace(subspace)

Validate the subspace for qutrit operations.

This method determines whether a given subspace for qutrit operations is defined correctly or not. If not, a *ValueError* is thrown.

Parameters

subspace (tuple[int]) - Subspace to check for correctness

2.7.2 Class Inheritance Diagram

180 Chapter 2. Tutorials

PYTHON MODULE INDEX

b

 ${\tt braket.pennylane_plugin}, 15$

182 Python Module Index

INDEX

A	<pre>analytic_probability() (BraketLocalQubitDevice</pre>		
AAMS (class in braket.pennylane_plugin), 15	method), 91		
access_state() (BraketAwsAhsDevice method), 33	<pre>apply() (BraketAwsAhsDevice method), 35</pre>		
access_state() (BraketAwsQubitDevice method), 53	<pre>apply() (BraketAwsQubitDevice method), 54</pre>		
access_state() (BraketLocalAhsDevice method), 71	<pre>apply() (BraketLocalAhsDevice method), 73</pre>		
access_state() (BraketLocalQubitDevice method), 90	<pre>apply() (BraketLocalQubitDevice method), 91</pre>		
<pre>active_wires() (BraketAwsAhsDevice static method),</pre>	arithmetic_depth (AAMS attribute), 16		
33	arithmetic_depth (CPhaseShift00 attribute), 103		
<pre>active_wires() (BraketAwsQubitDevice static</pre>	<pre>arithmetic_depth (CPhaseShift01 attribute), 114</pre>		
method), 53	<pre>arithmetic_depth (CPhaseShift10 attribute), 125</pre>		
active_wires() (BraketLocalAhsDevice static	<pre>arithmetic_depth (GPi attribute), 136</pre>		
method), 71	<pre>arithmetic_depth (GPi2 attribute), 147</pre>		
active_wires() (BraketLocalQubitDevice static	<pre>arithmetic_depth (MS attribute), 158</pre>		
method), 90	arithmetic_depth (PSWAP attribute), 169		
adjoint() (AAMS method), 20	author (BraketAwsAhsDevice attribute), 29		
adjoint() (CPhaseShift00 method), 106	author (BraketAwsQubitDevice attribute), 48		
adjoint() (CPhaseShift01 method), 117	author (BraketLocalAhsDevice attribute), 66		
adjoint() (CPhaseShift10 method), 128	author (BraketLocalQubitDevice attribute), 85		
adjoint() (GPi method), 139	D.		
adjoint() (GPi2 method), 150	В		
adjoint() (MS method), 161	basis (AAMS attribute), 16		
adjoint() (PSWAP method), 172	basis (CPhaseShift00 attribute), 103		
<pre>adjoint_jacobian() (BraketAwsAhsDevice method),</pre>	basis (CPhaseShift01 attribute), 114		
34	basis (CPhaseShift10 attribute), 125		
<pre>adjoint_jacobian() (BraketAwsQubitDevice method),</pre>	basis (GPi attribute), 136		
53	basis (<i>GPi2 attribute</i>), 147		
<pre>adjoint_jacobian() (BraketLocalAhsDevice method),</pre>	basis (MS attribute), 158		
72	basis (PSWAP attribute), 169		
<pre>adjoint_jacobian() (BraketLocalQubitDevice</pre>	<pre>batch_execute() (BraketAwsAhsDevice method), 35</pre>		
method), 90	<pre>batch_execute() (BraketAwsQubitDevice method), 54</pre>		
ahs_program (BraketAwsAhsDevice attribute), 28	<pre>batch_execute() (BraketLocalAhsDevice method), 73</pre>		
ahs_program (BraketLocalAhsDevice attribute), 66	<pre>batch_execute() (BraketLocalQubitDevice method),</pre>		
analytic (BraketAwsAhsDevice attribute), 28	91		
analytic (BraketAwsQubitDevice attribute), 48	batch_size (AAMS attribute), 17		
analytic (BraketLocalAhsDevice attribute), 66	batch_size (CPhaseShift00 attribute), 104		
<pre>analytic (BraketLocalQubitDevice attribute), 85</pre>	<pre>batch_size (CPhaseShift01 attribute), 114</pre>		
<pre>analytic_probability() (BraketAwsAhsDevice</pre>	batch_size (CPhaseShift10 attribute), 125		
method), 34	batch_size (GPi attribute), 136		
<pre>analytic_probability() (BraketAwsQubitDevice</pre>	batch_size (GPi2 attribute), 147		
method), 54	batch_size (MS attribute), 158		
<pre>analytic_probability() (BraketLocalAhsDevice</pre>			
method), 72	<pre>batch_transform() (BraketAwsAhsDevice method), 35</pre>		

<pre>batch_transform() (BraketAwsQubitDevice method),</pre>	<pre>compute_decomposition() (GPi2 static method), 150</pre>			
55	compute_decomposition() (MS static method), 161			
<pre>batch_transform() (BraketLocalAhsDevice method),</pre>	compute_decomposition() (PSWAP static method), 172			
batch_transform() (BraketLocalQubitDevice	compute_diagonalizing_gates() (AAMS static			
method), 92	method), 21			
braket.pennylane_plugin module,15	<pre>compute_diagonalizing_gates() (CPhaseShift00 static method), 107</pre>			
BraketAwsAhsDevice (class in	compute_diagonalizing_gates() (CPhaseShift01			
braket.pennylane_plugin), 27	static method), 118			
BraketAwsQubitDevice (class in	<pre>compute_diagonalizing_gates() (CPhaseShift10</pre>			
braket.pennylane_plugin), 46	static method), 129			
BraketLocalAhsDevice (class in	<pre>compute_diagonalizing_gates() (GPi static</pre>			
braket.pennylane_plugin), 65	method), 140			
BraketLocalQubitDevice (class in	<pre>compute_diagonalizing_gates() (GPi2 static</pre>			
braket.pennylane_plugin), 84	method), 151			
	<pre>compute_diagonalizing_gates() (MS static</pre>			
C	method), 162			
capabilities() (BraketAwsAhsDevice class method),	<pre>compute_diagonalizing_gates() (PSWAP static method), 173</pre>			
capabilities() (BraketAwsQubitDevice method), 55	<pre>compute_eigvals() (AAMS static method), 21</pre>			
capabilities() (BraketLocalAhsDevice class method),	<pre>compute_eigvals() (CPhaseShift00 static method),</pre>			
73	107			
capabilities() (BraketLocalQubitDevice class	<pre>compute_eigvals() (CPhaseShift01 static method),</pre>			
method), 92	118			
check_validity() (BraketAwsAhsDevice method), 36	<pre>compute_eigvals() (CPhaseShift10 static method),</pre>			
<pre>check_validity() (BraketAwsQubitDevice method),</pre>	129			
55	<pre>compute_eigvals() (GPi static method), 140</pre>			
<pre>check_validity() (BraketLocalAhsDevice method),</pre>	<pre>compute_eigvals() (GPi2 static method), 151</pre>			
74	<pre>compute_eigvals() (MS static method), 162</pre>			
<pre>check_validity() (BraketLocalQubitDevice method),</pre>	<pre>compute_eigvals() (PSWAP static method), 173</pre>			
92	<pre>compute_matrix() (AAMS static method), 22</pre>			
circuit (BraketAwsQubitDevice attribute), 48	compute_matrix() (CPhaseShift00 static method), 108			
circuit (BraketLocalQubitDevice attribute), 85	<pre>compute_matrix() (CPhaseShift01 static method), 119</pre>			
circuit_hash (BraketAwsAhsDevice attribute), 29	compute_matrix() (CPhaseShift10 static method), 130			
circuit_hash (BraketAwsQubitDevice attribute), 48	compute_matrix() (GPi static method), 141			
circuit_hash (BraketLocalAhsDevice attribute), 66	compute_matrix() (GPi2 static method), 152			
circuit_hash (BraketLocalQubitDevice attribute), 85	compute_matrix() (MS static method), 163			
<pre>classical_shadow() (BraketAwsAhsDevice method),</pre>	compute_matrix() (PSWAP static method), 174			
36	compute_sparse_matrix() (AAMS static method), 22			
<pre>classical_shadow() (BraketAwsQubitDevice method),</pre>	compute_sparse_matrix() (CPhaseShift00 static			
55	method), 108			
classical_shadow() (BraketLocalAhsDevice method),	compute_sparse_matrix() (CPhaseShift01 static			
74	method), 119			
classical_shadow() (BraketLocalQubitDevice method), 93	compute_sparse_matrix() (CPhaseShift10 static method), 130			
compute_decomposition() (AAMS static method), 20	<pre>compute_sparse_matrix() (GPi static method), 141</pre>			
compute_decomposition() (CPhaseShift00 static	<pre>compute_sparse_matrix() (GPi2 static method), 152</pre>			
method), 106	<pre>compute_sparse_matrix() (MS static method), 163</pre>			
compute_decomposition() (CPhaseShift01 static	<pre>compute_sparse_matrix() (PSWAP static method).</pre>			
method), 117	174			
compute_decomposition() (CPhaseShift10 static	control_wires (AAMS attribute), 17			
method), 128	control_wires (CPhaseShift00 attribute), 104			
compute decomposition()(GPi static method), 139	<pre>control_wires (CPhaseShift01 attribute), 115</pre>			

control_wires (<i>CPhaseShift10 attribute</i>), 126 control_wires (<i>GPi attribute</i>), 137	<pre>diagonalizing_gates() (GPi2 method), 153 diagonalizing_gates() (MS method), 164</pre>		
control_wires (GPi2 attribute), 148	<pre>diagonalizing_gates() (PSWAP method), 175</pre>		
control_wires (MS attribute), 159	_		
control_wires (<i>PSWAP attribute</i>), 170	E		
CPhaseShift00 (class in braket.pennylane_plugin), 102	eigvals() (AAMS method), 23		
CPhaseShift01 (class in braket.pennylane_plugin), 113	3 () (),		
CPhaseShift10 (class in braket.pennylane_plugin), 124	eigvals() (CPhaseShift01 method), 120		
create_ahs_program() (BraketAwsAhsDevice	eigvals() (CPhaseShift10 method), 131		
method), 37	eigvals() (CPriasesniji io meinoa), 131 eigvals() (GPi method), 142		
create_ahs_program() (BraketLocalAhsDevice	eigvals() (GPi2 method), 153		
method), 75	eigvals() (MS method), 164		
custom_expand() (BraketAwsAhsDevice method), 37	=		
custom_expand() (BraketAwsQubitDevice method), 56	eigvals() (PSWAP method), 175		
custom_expand() (BraketLocalAhsDevice method), 75	estimate_probability() (BraketAwsAhsDevice		
custom_expand() (BraketLocalQubitDevice method),	method), 38		
93	estimate_probability() (BraketAwsQubitDevice		
93	method), 57		
D	estimate_probability() (BraketLocalAhsDevice method), 76		
decomposition() (AAMS method), 23 decomposition() (CPhaseShift00 method), 109	estimate_probability() (BraketLocalQubitDevice method), 95		
decomposition() (CPhaseShift01 method), 120	execute() (BraketAwsAhsDevice method), 39		
decomposition() (CPhaseShift10 method), 131	execute() (BraketAwsQubitDevice method), 58		
decomposition() (GPi method), 142	execute() (BraketLocalAhsDevice method), 77		
decomposition() (GPi2 method), 153	execute() (BraketLocalQubitDevice method), 95		
decomposition() (MS method), 164	execute_and_gradients() (BraketAwsAhsDevice		
decomposition() (PSWAP method), 175	method), 39		
<pre>default_expand_fn() (BraketAwsAhsDevice method),</pre>	execute_and_gradients() (BraketAwsQubitDevice		
37	method), 58		
<pre>default_expand_fn()</pre>	<pre>execute_and_gradients() (BraketLocalAhsDevice method), 77</pre>		
default_expand_fn() (BraketLocalAhsDevice	<pre>execute_and_gradients() (BraketLocalQubitDevice</pre>		
method), 75	method), 95		
<pre>default_expand_fn()</pre>	execution_context() (BraketAwsAhsDevice method), 39		
define_wire_map() (BraketAwsAhsDevice method), 38			
<pre>define_wire_map() (BraketAwsQubitDevice method),</pre>	method), 58		
57 define_wire_map() (BraketLocalAhsDevice method),	execution_context() (BraketLocalAhsDevice method), 77		
76	<pre>execution_context() (BraketLocalQubitDevice</pre>		
define_wire_map() (BraketLocalQubitDevice	method), 96		
method), 94	expand() (AAMS method), 24		
<pre>density_matrix() (BraketAwsAhsDevice method), 38</pre>	expand() (CPhaseShift00 method), 110		
<pre>density_matrix() (BraketAwsQubitDevice method),</pre>			
57	expand() (CPhaseShift10 method), 132		
<pre>density_matrix() (BraketLocalAhsDevice method),</pre>	expand() (GPi method), 143		
76	expand() (GPi2 method), 154		
<pre>density_matrix() (BraketLocalQubitDevice method),</pre>			
94	expand() (MS method), 176 expand() (PSWAP method), 176		
diagonalizing_gates() (AAMS method), 23	expand_fn() (BraketAwsAhsDevice method), 40		
diagonalizing_gates() (<i>CPhaseShift00 method</i>), 109	expand_fn() (BraketAwsQubitDevice method), 58		
diagonalizing_gates() (CPhaseShift01 method), 120	- · · · · · · · · · · · · · · · · · · ·		
diagonalizing_gates() (CPhaseShift10 method), 131	· · · · · · · · · · · · · · · · · · ·		
diagonalizing_gates() (<i>GPi method</i>), 142	expand_fn() (BraketLocalQubitDevice method), 96		
aragonarrzing_gacco() (Or i memou), 172	expval() (BraketAwsAhsDevice method), 40		

expval() (BraketAwsQubitDevice method), 59	attribute), 29			
expval() (BraketLocalAhsDevice method), 78	has_adjoint (AAMS attribute), 17			
expval() (BraketLocalQubitDevice method), 96	has_adjoint (CPhaseShift00 attribute), 104			
	has_adjoint (CPhaseShift01 attribute), 115			
G	has_adjoint (CPhaseShift10 attribute), 126			
<pre>generate_basis_states() (BraketAwsAhsDevice</pre>	has_adjoint (GPi attribute), 137			
static method), 40	has_adjoint (GPi2 attribute), 148			
<pre>generate_basis_states() (BraketAwsQubitDevice</pre>	has_adjoint (MS attribute), 159			
static method), 59	has_adjoint (PSWAP attribute), 170			
<pre>generate_basis_states() (BraketLocalAhsDevice</pre>	has_decomposition (AAMS attribute), 17			
static method), 78	has_decomposition (CPhaseShift00 attribute), 104			
<pre>generate_basis_states() (BraketLocalQubitDevice</pre>	has_decomposition (CPhaseShift01 attribute), 115			
static method), 96	has_decomposition (CPhaseShift10 attribute), 126			
<pre>generate_samples() (BraketAwsAhsDevice method),</pre>	has_decomposition (GPi attribute), 137			
41	has_decomposition (GPi2 attribute), 148			
<pre>generate_samples() (BraketAwsQubitDevice method),</pre>				
59	has_decomposition (<i>PSWAP attribute</i>), 170			
<pre>generate_samples() (BraketLocalAhsDevice method),</pre>	has_diagonalizing_gates (AAMS attribute), 17			
78	has_diagonalizing_gates (CPhaseShift00 attribute),			
<pre>generate_samples() (BraketLocalQubitDevice</pre>	104			
method), 97	has_diagonalizing_gates (CPhaseShift01 attribute), 115			
generator() (AAMS method), 24				
generator() (CPhaseShift00 method), 110	has_diagonalizing_gates (<i>CPhaseShift10 attribute</i>), 126			
generator() (CPhaseShift01 method), 121	has_diagonalizing_gates (<i>GPi attribute</i>), 137			
generator() (CPhaseShift10 method), 132	has_diagonalizing_gates (GF1 attribute), 137 has_diagonalizing_gates (GPi2 attribute), 148			
generator() (GPi method), 143	has_diagonalizing_gates (<i>MS attribute</i>), 159			
generator() (GPi2 method), 154	has_diagonalizing_gates (NS dirrodie), 170			
generator() (<i>MS method</i>), 165 generator() (<i>PSWAP method</i>), 176	has_generator (AAMS attribute), 17			
GPi (class in braket.pennylane_plugin), 135	has_generator (<i>CPhaseShift00 attribute</i>), 104			
GPi2 (class in braket.pennylane_plugin), 133 GPi2 (class in braket.pennylane_plugin), 146	has_generator (<i>CPhaseShift01 attribute</i>), 115			
grad_method (AAMS attribute), 17	has_generator (<i>CPhaseShift10 attribute</i>), 126			
grad_method (<i>PhaseShift00 attribute</i>), 104	has_generator (<i>GPi attribute</i>), 137			
grad_method (<i>CPhaseShift01 attribute</i>), 115	has_generator (<i>GPi2 attribute</i>), 148			
grad_method (<i>CPhaseShift10 attribute</i>), 126	has_generator (MS attribute), 159			
grad_method (<i>GPi attribute</i>), 137	has_generator (<i>PSWAP attribute</i>), 170			
grad_method (GPi2 attribute), 148	has_matrix (AAMS attribute), 17			
grad_method (MS attribute), 159	has_matrix (CPhaseShift00 attribute), 104			
grad_method (<i>PSWAP attribute</i>), 170	has_matrix (CPhaseShift01 attribute), 115			
grad_recipe (AAMS attribute), 17	has_matrix (CPhaseShift10 attribute), 126			
grad_recipe (CPhaseShift00 attribute), 104	has_matrix (GPi attribute), 137			
grad_recipe (CPhaseShift01 attribute), 115	has_matrix (GPi2 attribute), 148			
grad_recipe (CPhaseShift10 attribute), 126	has_matrix (MS attribute), 159			
grad_recipe (GPi attribute), 137	has_matrix (PSWAP attribute), 170			
grad_recipe (GPi2 attribute), 148	hash (AAMS attribute), 17			
grad_recipe (MS attribute), 159	hash (CPhaseShift00 attribute), 104			
grad_recipe (<i>PSWAP attribute</i>), 170	hash (CPhaseShift01 attribute), 115			
<pre>gradients() (BraketAwsAhsDevice method), 41</pre>	hash (CPhaseShift10 attribute), 126			
<pre>gradients() (BraketAwsQubitDevice method), 60</pre>	hash (GPi attribute), 137			
<pre>gradients() (BraketLocalAhsDevice method), 79</pre>	hash (GPi2 attribute), 148			
<pre>gradients() (BraketLocalQubitDevice method), 97</pre>	hash (MS attribute), 159			
	hash (PSWAP attribute), 170			
H	hyperparameters (AAMS attribute), 17			
hardware canabilities (<i>BraketAwsAhsDevice</i>	hyperparameters (CPhaseShift00 attribute), 104			

hyperparameters (<i>CPhaseShift01 attribute</i>), 115	matrix() (CPhaseShift00 method), 111		
hyperparameters (CPhaseShift10 attribute), 126	<pre>matrix() (CPhaseShiftO1 method), 122</pre>		
hyperparameters (GPi attribute), 137	<pre>matrix() (CPhaseShift10 method), 133</pre>		
hyperparameters (GPi2 attribute), 148	matrix() (GPi method), 144		
hyperparameters (MS attribute), 159	matrix() (GPi2 method), 155		
hyperparameters (PSWAP attribute), 170	<pre>matrix() (MS method), 166</pre>		
1	matrix() (PSWAP method), 177		
I	measurement_map (BraketAwsAhsDevice attribute), 29		
id (AAMS attribute), 18	measurement_map (BraketAwsQubitDevice attribute),		
id (CPhaseShift00 attribute), 105	49		
id (CPhaseShift01 attribute), 115	measurement_map (BraketLocalAhsDevice attribute), 67		
id (CPhaseShift10 attribute), 126	<pre>measurement_map (BraketLocalQubitDevice attribute),</pre>		
id (GPi attribute), 137	85		
id (GPi2 attribute), 148	module		
id (MS attribute), 159	braket.pennylane_plugin,15		
id (PSWAP attribute), 170	MS (class in braket.pennylane_plugin), 157		
is_hermitian (AAMS attribute), 18	mutual_info() (BraketAwsAhsDevice method), 42		
is_hermitian (CPhaseShift00 attribute), 105	mutual_info() (BraketAwsQubitDevice method), 61		
is_hermitian (CPhaseShift01 attribute), 115	mutual_info() (BraketLocalAhsDevice method), 80		
is_hermitian (CPhaseShift10 attribute), 126	mutual_info() (BraketLocalQubitDevice method), 98		
is_hermitian (GPi attribute), 137	macaal_inio() (BrancillocarguonBerrice memoa), >0		
	N		
is_hermitian (GPi2 attribute), 148			
is_hermitian (MS attribute), 159	name (AAMS attribute), 18		
is_hermitian (PSWAP attribute), 170	name (BraketAwsAhsDevice attribute), 29		
L	name (BraketAwsQubitDevice attribute), 49		
	name (BraketLocalAhsDevice attribute), 67		
label() (AAMS method), 24	name (BraketLocalQubitDevice attribute), 86		
label() (CPhaseShift00 method), 110	name (CPhaseShift00 attribute), 105		
label() (CPhaseShift01 method), 121	name (CPhaseShift01 attribute), 115		
label() (CPhaseShift10 method), 132	name (CPhaseShift10 attribute), 126		
label() (<i>GPi method</i>), 143	name (GPi attribute), 137		
label() (<i>GPi2 method</i>), 154	name (GPi2 attribute), 148		
label() (MS method), 165	name (MS attribute), 159		
label() (PSWAP method), 176	name (PSWAP attribute), 170		
	ndim_params (AAMS attribute), 18		
M	ndim_params (CPhaseShift00 attribute), 105		
<pre>map_wires() (AAMS method), 25</pre>	ndim_params (CPhaseShift01 attribute), 116		
map_wires() (BraketAwsAhsDevice method), 41	ndim_params (CPhaseShift10 attribute), 127		
map_wires() (BraketAwsQubitDevice method), 60	ndim_params (GPi attribute), 138		
map_wires() (BraketLocalAhsDevice method), 79	ndim_params (GPi2 attribute), 149		
map_wires() (BraketLocalQubitDevice method), 97	ndim_params (MS attribute), 160		
	ndim_params (PSWAP attribute), 171		
map_wires() (CPhaseShift00 method), 111	num_executions (BraketAwsAhsDevice attribute), 29		
map_wires() (CPhaseShift01 method), 122	num_executions (BraketAwsQubitDevice attribute), 49		
map_wires() (CPhaseShift10 method), 133	num_executions (BraketLocalAhsDevice attribute), 47		
map_wires() (GPi method), 144	num_executions (BraketLocalQubitDevice attribute), 07		
map_wires() (GPi2 method), 155	86		
map_wires() (MS method), 166			
<pre>map_wires() (PSWAP method), 177</pre>	num_params (AAMS attribute), 18		
marginal_prob() (BraketAwsAhsDevice method), 41	num_params (CPhaseShift00 attribute), 105		
marginal_prob() (BraketAwsQubitDevice method), 60	num_params (CPhaseShift01 attribute), 116		
<pre>marginal_prob() (BraketLocalAhsDevice method), 79</pre>	num_params (CPhaseShift10 attribute), 127		
$marginal_prob()$ (BraketLocalQubitDevice method),	num_params (GPi attribute), 138		
98	num_params (GPi2 attribute), 149		
<pre>matrix() (AAMS method), 25</pre>	num_params (MS attribute), 160		

num_params (<i>PSWAP attribute</i>), 171	parameters (GPi2 attribute), 149			
num_wires (AAMS attribute), 18	parameters (MS attribute), 160			
num_wires (CPhaseShift00 attribute), 105	parameters (PSWAP attribute), 171			
num_wires (CPhaseShift01 attribute), 116	<pre>pennylane_requires (BraketAwsAhsDevice attribute),</pre>			
num_wires (CPhaseShift10 attribute), 127	30			
num_wires (<i>GPi attribute</i>), 138	pennylane_requires (BraketAwsQubitDevice at-			
num_wires (GPi2 attribute), 149	tribute), 50			
num_wires (MS attribute), 160	pennylane_requires (BraketLocalAhsDevice at-			
num_wires (<i>PSWAP attribute</i>), 171	tribute), 68			
0	<pre>pennylane_requires (BraketLocalQubitDevice at- tribute), 87</pre>			
obs_queue (BraketAwsAhsDevice attribute), 30	<pre>post_apply() (BraketAwsAhsDevice method), 42</pre>			
obs_queue (BraketAwsQubitDevice attribute), 49	<pre>post_apply() (BraketAwsQubitDevice method), 61</pre>			
obs_queue (BraketLocalAhsDevice attribute), 67	<pre>post_apply() (BraketLocalAhsDevice method), 80</pre>			
obs_queue (BraketLocalQubitDevice attribute), 86	post_apply() (BraketLocalQubitDevice method), 99			
observables (BraketAwsAhsDevice attribute), 30	<pre>post_measure() (BraketAwsAhsDevice method), 42</pre>			
observables (BraketAwsQubitDevice attribute), 50	<pre>post_measure() (BraketAwsQubitDevice method), 61</pre>			
observables (BraketLocalAhsDevice attribute), 68	<pre>post_measure() (BraketLocalAhsDevice method), 80</pre>			
observables (BraketLocalQubitDevice attribute), 87	<pre>post_measure() (BraketLocalQubitDevice method), 99</pre>			
op_queue (BraketAwsAhsDevice attribute), 30	pow() (AAMS method), 26			
op_queue (BraketAwsQubitDevice attribute), 50	pow() (CPhaseShift00 method), 112			
op_queue (BraketLocalAhsDevice attribute), 68	pow() (CPhaseShift01 method), 123			
op_queue (BraketLocalQubitDevice attribute), 87	pow() (CPhaseShift10 method), 134			
operations (BraketAwsAhsDevice attribute), 30	pow() (GPi method), 145			
operations (BraketAwsQubitDevice attribute), 50	pow() (<i>GPi2 method</i>), 156			
operations (BraketLocalAhsDevice attribute), 68	pow() (MS method), 167			
operations (BraketLocalQubitDevice attribute), 87	pow() (PSWAP method), 178			
order_wires() (BraketAwsAhsDevice method), 42	<pre>pre_apply() (BraketAwsAhsDevice method), 42</pre>			
order_wires() (BraketAwsQubitDevice method), 61	<pre>pre_apply() (BraketAwsQubitDevice method), 62</pre>			
order_wires() (BraketLocalAhsDevice method), 80	<pre>pre_apply() (BraketLocalAhsDevice method), 80</pre>			
order_wires() (BraketLocalQubitDevice method), 98	pre_apply() (BraketLocalQubitDevice method), 99			
oraci_mires () (Brancillocarguous erice memoa), 50	<pre>pre_measure() (BraketAwsAhsDevice method), 43</pre>			
P	pre_measure() (BraketAwsQubitDevice method), 62			
	pre_measure() (BraketLocalAhsDevice method), 80			
parallel (BraketAwsQubitDevice attribute), 50	pre_measure() (BraketLocalQubitDevice method), 99			
parameter_frequencies (AAMS attribute), 18	probability() (BraketAwsAhsDevice method), 43			
parameter_frequencies (CPhaseShift00 attribute),	probability() (BraketAwsQubitDevice method), 62			
105	<pre>probability() (BraketLocalAhsDevice method), 80</pre>			
parameter_frequencies (CPhaseShift01 attribute),	<pre>probability() (BraketLocalQubitDevice method), 99</pre>			
116	PSWAP (class in braket.pennylane_plugin), 168			
parameter_frequencies (CPhaseShift10 attribute),				
127	Q			
parameter_frequencies (GPi attribute), 138	queue() (AAMS method), 26			
parameter_frequencies (GPi2 attribute), 149	queue() (CPhaseShift00 method), 112			
parameter_frequencies (MS attribute), 160	queue() (CPhaseShift01 method), 123			
parameter_frequencies (PSWAP attribute), 171	queue() (CPhaseShift10 method), 134			
parameters (AAMS attribute), 19	queue() (GPi method), 145			
parameters (BraketAwsAhsDevice attribute), 30	queue() (GPi2 method), 156			
parameters (BraketAwsQubitDevice attribute), 50	queue() (MS method), 150 queue() (MS method), 167			
parameters (BraketLocalAhsDevice attribute), 68	queue() (PSWAP method), 178			
parameters (BraketLocalQubitDevice attribute), 87	queue() (15mm memou), 170			
parameters (CPhaseShift00 attribute), 105	R			
parameters (CPhaseShift01 attribute), 116				
parameters (CPhaseShift10 attribute), 127	register (BraketAwsAhsDevice attribute), 30			
parameters (<i>GPi attribute</i>), 138	register (BraketLocalAhsDevice attribute), 68			

reset() (BraketAwsAhsDevice ma	ethod), 43	<pre>simplify() (PSWAP method), 178</pre>				
<pre>reset() (BraketAwsQubitDevice)</pre>	method), 62	single_qubit_rot_an	les() (AAMS method), 26			
reset() (BraketLocalAhsDevice)	method), 81	single_qubit_rot_an	gles()	(CPhaseShi	ift00	
reset() (BraketLocalQubitDevice		method), 112	(1)			
result (BraketAwsAhsDevice attr				(CPhaseShi	ift01	
result (BraketLocalAhsDevice at		method), 123	5	(.,	
0		single_qubit_rot_an	gles()	(CPhaseShi	ift10	
S		method), 134				
<pre>sample() (BraketAwsAhsDevice r</pre>	nethod), 43	<pre>single_qubit_rot_angles() (GPi method), 145</pre>				
sample() (BraketAwsQubitDevice method), 62		<pre>single_qubit_rot_angles() (GPi2 method), 156</pre>				
<pre>sample() (BraketLocalAhsDevice</pre>	e method), 81	<pre>single_qubit_rot_angles() (MS method), 167</pre>				
<pre>sample() (BraketLocalQubitDevi</pre>	ice method), 99	<pre>single_qubit_rot_angles() (PSWAP method), 178</pre>			178	
<pre>sample_basis_states()</pre>	(BraketAwsAhsDevice	<pre>sparse_matrix() (AA)</pre>	MS method), 26			
method), 43		<pre>sparse_matrix() (CP)</pre>	haseShift00 meth	nod), 112		
	BraketAwsQubitDevice	<pre>sparse_matrix() (CP)</pre>	haseShift01 meth	nod), 123		
method), 62	~	sparse_matrix() (CPhaseShift10 method), 134				
	BraketLocalAhsDevice	<pre>sparse_matrix() (GP)</pre>	i method), 145			
method), 81		<pre>sparse_matrix() (GP)</pre>	<i>i</i> 2 <i>method</i>), 156			
**	raketLocalQubitDevice	sparse_matrix() (MS method), 167				
method), 100		sparse_matrix() (PSWAP method), 178				
settings (BraketAwsAhsDevice a	attribute) 30	state (BraketAwsAhsDevice attribute), 31				
settings (BraketLocalAhsDevice		state (BraketAwsQubitDevice attribute), 51				
shadow_expval() (BraketAwsAh		state (BraketLocalAhsI				
shadow_expval() (BraketAwsQu						
- " , ~ ~	**	states_to_binary()	(BraketAwsAh		tatic	
shadow_expval() (BraketLocalA		method), 44	(Branca III sa III	servee s.	iciric	
shadow_expval() (BraketLocal 100	QubitDevice methoa),	states_to_binary()	(BraketAwsQub	oitDevice s	tatic	
		<i>method</i>), 63	(Brancii Iws Quo	ibevice si	icirc	
short_name (BraketAwsAhsDevice		states_to_binary()	(BraketLocalAl	hsDevice s	tatic	
short_name (BraketAwsQubitDevice attribute), 50		method), 82	(DrukeiLocaiAr	isDevice si	шис	
short_name (BraketLocalAhsDevice attribute), 68			(Prakati analOu	hitDaviaa a	tatio	
short_name (BraketLocalQubitDe		states_to_binary()	(Бтакеньосануи	dudevice si	шис	
<pre>shot_vec_statistics()</pre>	(BraketAwsAhsDevice	method), 100	A I D	-41 1\ AA		
method), 44		statistics() (BraketA				
	BraketAwsQubitDevice	statistics() (BraketA				
method), 63		statistics() (BraketL				
· · · · · · · · · · · · · · · · · · ·	BraketLocalAhsDevice	statistics() (BraketL				
method), 82		stopping_condition ((BraketAwsAhsD	evice attribi	ute),	
<pre>shot_vec_statistics() (Bit)</pre>	raketLocalQubitDevice	31				
method), 100		stopping_condition	(BraketAwsQu	ıbitDevice	at-	
<pre>shot_vector (BraketAwsAhsDevi</pre>	ice attribute), 31	tribute), 51				
<pre>shot_vector (BraketAwsQubitDe</pre>	evice attribute), 50	stopping_condition	(BraketLocalA	AhsDevice	at-	
<pre>shot_vector (BraketLocalAhsDe</pre>	evice attribute), 69	tribute), 69				
<pre>shot_vector (BraketLocalQubit)</pre>	Device attribute), 87	stopping_condition	(BraketLocalQ	ubitDevice	at-	
shots (BraketAwsAhsDevice attrib	bute), 31	tribute), 88				
shots (BraketAwsQubitDevice att	ribute), 51	supports_observable	() (Brake	etAwsAhsDe	evice	
shots (BraketLocalAhsDevice att	ribute), 69	method), 45				
shots (BraketLocalQubitDevice a	uttribute), 88	supports_observable	() (Braket	AwsQubitDe	evice	
<pre>simplify() (AAMS method), 26</pre>		method), 64				
simplify() (CPhaseShift00 meth	nod), 112	supports_observable	() (Braket	LocalAhsDe	evice	
simplify() (CPhaseShift01 meth		method), 83				
simplify() (CPhaseShift10 meth		supports_observable	() (BraketLa	ocalQubitDe	evice	
simplify() (GPi method), 145	// -	method), 101				
simplify() (GPi2 method), 156		supports_operation() (Brake	etAwsAhsDe	evice	
simplify() (MS method) 167		method), 45				

```
supports_operation()
                              (BraketAwsOubitDevice wire_map (BraketLocalOubitDevice attribute), 88
                                                      wires (AAMS attribute), 19
        method), 64
supports_operation()
                              (BraketLocalAhsDevice
                                                      wires (BraketAwsAhsDevice attribute), 32
                                                      wires (BraketAwsQubitDevice attribute), 51
         method), 83
supports_operation()
                            (BraketLocalQubitDevice
                                                      wires (BraketLocalAhsDevice attribute), 69
        method), 101
                                                      wires (BraketLocalQubitDevice attribute), 88
                                                      wires (CPhaseShift00 attribute), 105
Т
                                                      wires (CPhaseShift01 attribute), 116
                                                      wires (CPhaseShift10 attribute), 127
task (BraketAwsAhsDevice attribute), 31
                                                      wires (GPi attribute), 138
task (BraketAwsQubitDevice attribute), 51
                                                      wires (GPi2 attribute), 149
task (BraketLocalAhsDevice attribute), 69
                                                      wires (MS attribute), 160
task (BraketLocalQubitDevice attribute), 88
                                                      wires (PSWAP attribute), 171
terms() (AAMS method), 26
terms() (CPhaseShift00 method), 112
terms() (CPhaseShift01 method), 123
terms() (CPhaseShift10 method), 134
terms() (GPi method), 146
terms() (GPi2 method), 157
terms() (MS method), 168
terms() (PSWAP method), 179
U
use_grouping (BraketAwsQubitDevice attribute), 51
V
validate_subspace() (AAMS static method), 27
validate_subspace() (CPhaseShift00 static method),
         113
validate_subspace() (CPhaseShift01 static method),
         124
validate_subspace() (CPhaseShift10 static method),
validate_subspace() (GPi static method), 146
validate_subspace() (GPi2 static method), 157
validate_subspace() (MS static method), 168
validate_subspace() (PSWAP static method), 179
var() (BraketAwsAhsDevice method), 45
var() (BraketAwsQubitDevice method), 64
var() (BraketLocalAhsDevice method), 83
var() (BraketLocalQubitDevice method), 102
version (BraketAwsAhsDevice attribute), 31
version (BraketAwsQubitDevice attribute), 51
version (BraketLocalAhsDevice attribute), 69
version (BraketLocalQubitDevice attribute), 88
vn_entropy() (BraketAwsAhsDevice method), 46
vn_entropy() (BraketAwsQubitDevice method), 65
vn_entropy() (BraketLocalAhsDevice method), 84
vn_entropy() (BraketLocalQubitDevice method), 102
W
wire_map (BraketAwsAhsDevice attribute), 32
wire_map (BraketAwsQubitDevice attribute), 51
```

wire_map (BraketLocalAhsDevice attribute), 69