

---

# **Amazon Braket PennyLane Plugin Documentation**

***Release 1.24.3.dev0***

**Amazon.com Inc.**

**2024-01-29**



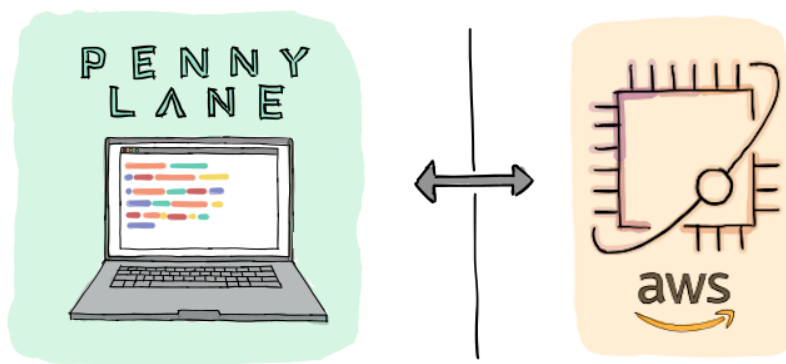
# CONTENTS

<b>1</b>	<b>Devices</b>	<b>3</b>
<b>2</b>	<b>Tutorials</b>	<b>5</b>
2.1	Installation . . . . .	5
2.2	Support . . . . .	7
2.3	The remote Braket device . . . . .	7
2.4	The local Braket device . . . . .	10
2.5	The local AHS device . . . . .	11
2.6	The remote AHS device . . . . .	13
2.7	pennylane-braket . . . . .	15
	<b>Python Module Index</b>	<b>185</b>
	<b>Index</b>	<b>187</b>



**Release**

1.24.3.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.



## **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.





## 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
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:

```
>>> import pennylane as qml
>>> s3 = ("my-bucket", "my-prefix")
>>> remote_device = qml.device("braket.aws.qubit", device_arn="arn:aws:braket:::device/
↳ quantum-simulator/amazon/sv1", s3_destination_folder=s3, wires=2)
```

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])
```

### 2.3.2 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.

```
>>> remote_device = qml.device('braket.aws.qubit', [...], parallel=True, max_
↳parallel=20)
```

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`:

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

## 2.3.5 Pulse Programming

The PennyLane-Braket plugin provides pulse-level control for the OQC Lucy QPU through PennyLane's `ParametrizedEvolution` operation. Compatible pulse Hamiltonians can be defined using the `qml.pulse.transmon_drive` function and used to create `ParametrizedEvolution`'s using `qml.evolve`:

```
duration = 15
def amp(p, t):
    return qml.pulse.pwc(duration)(p, t)

dev = qml.device("braket.aws.qubit", wires=8, device_arn="arn:aws:braket:eu-west-2::device/qpu/oqc/Lucy")

drive = qml.pulse.transmon.transmon_drive(amplitude=amp, phase=0, freq=4.8, wires=[0])

@qml.qnode(dev)
def circuit(params, t):
    qml.evolve(drive)(params, t)
    return qml.expval(qml.PauliZ(wires=0))
```

Note that the `freq` argument of `qml.pulse.transmon_drive` is specified in GHz, and for hardware upload the amplitude will be interpreted as an output power for control hardware in volts. The phase must be specified in radians.

The pulse settings for the device can be obtained using the `pulse_settings` property. These settings can be used to describe the transmon interaction Hamiltonian using `qml.pulse.transmon_interaction`:

```
dev = qml.device("braket.aws.qubit", wires=8, device_arn="arn:aws:braket:eu-west-2::device/qpu/oqc/Lucy")
pulse_settings = dev.pulse_settings
couplings = [0.01]*len(connections)
H = qml.pulse.transmon_interaction(**pulse_settings, coupling=couplings)
```

By passing `pulse_settings` from the remote device to `qml.pulse.transmon_interaction`, an  $H$  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.

Note that the user must supply coupling coefficients, as these are not available from the hardware backend. On the order of 10 MHz (0.01 GHz) is in a realistic range.

### 2.3.6 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](#) when running QAOA from PennyLane, you will need to 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:

```
import pennylane as qml
device_local = qml.device("braket.local.qubit", wires=2) # local state vector simulator
# device_local = qml.device("braket.local.qubit", backend="default", wires=2) # local
# state vector simulator
# device_local = qml.device("braket.local.qubit", backend="braket_sv", wires=2) # local
# state vector simulator
# device_local = qml.device("braket.local.qubit", backend="braket_dm", wires=2) # local
# state vector simulator
```

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`:

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

### 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 [\[here\]\(https://github.com/google/jax#installation\)](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  $2\pi$ ).

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))
    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,
    ↪ wires=[0, 1])

# creating local drives
# note only local detuning is currently supported, so amplitude and phase are set to 0
```

(continues on next page)



(continued from previous page)

```

H_local0 = qml.pulse.rydberg_drive(amplitude=0, phase=0, detuning = det_fn_local,
↪wires=[0])
H_local1 = qml.pulse.rydberg_drive(amplitude=0, phase=0, detuning = det_fn_local,
↪wires=[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.

```

>>> circuit([amp_params, det_global_params, local_params1, local_params2])
array([[0, 0],
       [0, 0],
       [0, 0],
       ...,
       [1, 0],
       [1, 0],
       [1, 0]])

```

## 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 [\[here\]\(https://github.com/google/jax#installation\)](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  $2\pi$ ).

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
```

(continues on next page)

(continued from previous page)

```
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:

```
@qml.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

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

<code>AAMS(phi_0, phi_1, theta, wires)</code>	IonQ native Arbitrary-Angle Mølmer-Sørensen gate.
<code>BraketAwsAhsDevice(wires, device_arn[, ...])</code>	Amazon Braket AHS device for hardware in PennyLane.
<code>BraketAwsQubitDevice(wires, device_arn[, ...])</code>	Amazon Braket AwsDevice qubit device for PennyLane.
<code>BraketLocalAhsDevice(wires, *[, shots])</code>	Amazon Braket LocalSimulator AHS device for PennyLane.
<code>BraketLocalQubitDevice(wires[, backend, shots])</code>	Amazon Braket LocalSimulator qubit device for PennyLane.
<code>CPhaseShift00(phi, wires)</code>	Controlled phase shift gate phasing the $ 00\rangle$ state.
<code>CPhaseShift01(phi, wires)</code>	Controlled phase shift gate phasing the $ 01\rangle$ state.
<code>CPhaseShift10(phi, wires)</code>	Controlled phase shift gate phasing the $ 10\rangle$ state.
<code>GPI(phi, wires)</code>	IonQ native GPI gate.
<code>GPI2(phi, wires)</code>	IonQ native GPI2 gate.
<code>MS(phi_0, phi_1, wires)</code>	IonQ native Mølmer-Sørensen gate.
<code>PSWAP(phi, wires)</code>	Phase-SWAP gate.

### AAMS

**class** `AAMS(phi_0, phi_1, theta, wires)`

Bases: `Operation`

IonQ native Arbitrary-Angle Mølmer-Sørensen gate.

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

#### 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)

<code>arithmetic_depth</code>	Arithmetic depth of the operator.
<code>basis</code>	The basis of an operation, or for controlled gates, of the target operation.
<code>batch_size</code>	Batch size of the operator if it is used with broadcasted parameters.
<code>control_wires</code>	Control wires of the operator.
<code>grad_method</code>	
<code>grad_recipe</code>	Gradient recipe for the parameter-shift method.
<code>has_adjoint</code>	
<code>has_decomposition</code>	
<code>has_diagonalizing_gates</code>	
<code>has_generator</code>	
<code>has_matrix</code>	
<code>hash</code>	Integer hash that uniquely represents the operator.
<code>hyperparameters</code>	Dictionary of non-trainable variables that this operation depends on.
<code>id</code>	Custom string to label a specific operator instance.
<code>is_hermitian</code>	This property determines if an operator is hermitian.
<code>name</code>	String for the name of the operator.
<code>ndim_params</code>	Number of dimensions per trainable parameter of the operator.
<code>num_params</code>	
<code>num_wires</code>	Number of wires the operator acts on.
<code>parameter_frequencies</code>	Returns the frequencies for each operator parameter with respect to an expectation value of the form $\langle \psi   U(\mathbf{p})^\dagger \hat{O} U(\mathbf{p})   \psi \rangle$ .
<code>parameters</code>	Trainable parameters that the operator depends on.
<code>pauli_rep</code>	A <code>PauliSentence</code> representation of the Operator, or <code>None</code> if it doesn't have one.
<code>wires</code>	Wires that the operator acts on.

**arithmetic\_depth**

Arithmetic depth of the operator.

**basis**

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  $\phi_k$ , 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,)]
>>> 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.

**pauli\_rep**

A PauliSentence representation of the Operator, or None if it doesn't have one.

**wires**

Wires that the operator acts on.

**Returns**

wires

**Return type**

Wires

<code>adjoint()</code>	Create an operation that is the adjoint of this one.
<code>compute_decomposition(*params[, wires])</code>	Representation of the operator as a product of other operators (static method).
<code>compute_diagonalizing_gates(*params, wires, ...)</code>	Sequence of gates that diagonalize the operator in the computational basis (static method).
<code>compute_eigvals(*params, **hyperparams)</code>	Eigenvalues of the operator in the computational basis (static method).
<code>compute_matrix(phi_0, phi_1, theta)</code>	Representation of the operator as a canonical matrix in the computational basis (static method).
<code>compute_sparse_matrix(*params, **hyperparams)</code>	Representation of the operator as a sparse matrix in the computational basis (static method).
<code>decomposition()</code>	Representation of the operator as a product of other operators.
<code>diagonalizing_gates()</code>	Sequence of gates that diagonalize the operator in the computational basis.
<code>eigvals()</code>	Eigenvalues of the operator in the computational basis.
<code>expand()</code>	Returns a tape that contains the decomposition of the operator.
<code>generator()</code>	Generator of an operator that is in single-parameter-form.
<code>label([decimals, base_label, cache])</code>	A customizable string representation of the operator.
<code>map_wires(wire_map)</code>	Returns a copy of the current operator with its wires changed according to the given wire map.
<code>matrix([wire_order])</code>	Representation of the operator as a matrix in the computational basis.
<code>pow(z)</code>	A list of new operators equal to this one raised to the given power.
<code>queue([context])</code>	Append the operator to the Operator queue.
<code>simplify()</code>	Reduce the depth of nested operators to the minimum.
<code>single_qubit_rot_angles()</code>	The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.
<code>sparse_matrix([wire_order])</code>	Representation of the operator as a sparse matrix in the computational basis.
<code>terms()</code>	Representation of the operator as a linear combination of other operators.
<code>validate_subspace(subspace)</code>	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  $\Sigma$  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  $\Sigma$  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  $\Sigma$  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  $\Sigma$  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(base_label="my_label")
"my_label"
>>> op = qml.RX(1.23456, wires=0, id="test_data")
>>> op.label()
"RX("test_data")"
>>> op.label(decimals=2)
"RX\n(1.23,"test_data")"
>>> op.label(base_label="my_label")
"my_label("test_data")"
>>> op.label(decimals=2, base_label="my_label")
"my_label\n(1.23,"test_data")"
```

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*) → 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.queueing.QueueingManager'>*)

Append the operator to the Operator queue.

**simplify**() → 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.

**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 acquire 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  $2\pi$ ).

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.

---



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

```
class NewDevice(DefaultQubitLegacy):
    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****settings**

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 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**

```
>>> dev = qml.device("default.qubit.legacy", wires=2, shots=[3, 1, 2, 2, 2, 2, 6, 1, 1, 5, 12, 10, 10])
>>> dev.shots
57
>>> dev.shot_vector
[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 queueable object (including a PennyLane operation and observable) and returns True if supported by the device.

**Type**

`.BooleanFn`

**task**

**version** = '0.34.0'

**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

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

continues on next page

Table 1 – continued from previous page

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

**operations** (list[ParametrizedEvolution]) – a list containing a single ParametrizedEvolution operator

**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, TensorFlow, 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-&gt;\*]

**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.legacy", 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**

```
>>> 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*, *\*\*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 [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**

statistics 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  $\text{var}(A) = \psi A^2 \psi - \psi A \psi^2$

**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()`.

<i>analytic</i>	Whether shots is None or not.
<i>author</i>	
<i>circuit</i>	The last circuit run on this device.
<i>circuit_hash</i>	The hash of the circuit upon the last execution.
<i>measurement_map</i>	Mapping used to override the logic of measurement processes.
<i>name</i>	
<i>num_executions</i>	Number of times this device is executed by the evaluation of QNodes running on this device
<i>obs_queue</i>	The observables to be measured and returned.
<i>observables</i>	set() -> new empty set object set(iterable) -> new set object
<i>op_queue</i>	The operation queue to be applied.
<i>operations</i>	The set of names of PennyLane operations that the device supports.
<i>parallel</i>	
<i>parameters</i>	Mapping from free parameter index to the list of Operations in the device queue that depend on it.
<i>pennylane_requires</i>	
<i>pulse_settings</i>	Dictionary of constants set by the hardware (qubit resonant frequencies, inter-qubit connection graph, wires and anharmonicities).
<i>short_name</i>	
<i>shot_vector</i>	Returns the shot vector, a sparse representation of the shot sequence used by the device when evaluating QNodes.
<i>shots</i>	Number of circuit evaluations/random samples used to estimate expectation values of observables
<i>state</i>	Returns the state vector of the circuit prior to measurement.
<i>stopping_condition</i>	Returns the stopping condition for the device.
<i>task</i>	The task corresponding to the last run circuit.
<i>use_grouping</i>	
<i>version</i>	
<i>wire_map</i>	Ordered dictionary that defines the map from user-provided wire labels to the wire labels used on this device
<i>wires</i>	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 a device that inherits from `DefaultQubitLegacy` and overrides the logic of the `qml.sample` measurement. To do so we will need to update the `measurement_map` dictionary:

```
class NewDevice(DefaultQubitLegacy):
    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'

**pulse\_settings**

Dictionary of constants set by the hardware (qubit resonant frequencies, inter-qubit connection graph, wires and anharmonicities).

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

```
>>> dev_remote = qml.device('braket.aws.qubit',
>>>                           wires=8,
>>>                           arn='arn:aws:braket:eu-west-2::device/qpu/oqc/Lucy
→ ')
>>> pulse_settings = dev_remote.pulse_settings
>>> H_int = qml.pulse.transmon_interaction(**pulse_settings, coupling=0.02)
```

By passing the `pulse_settings` from the remote device to `transmon_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.

Note that the user must supply coupling coefficients, as these are not available from the hardware backend.

**`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.legacy", wires=2, shots=[3, 1, 2, 2, 2, 2, 6, 1, 1, 5, 12, 10, 10])
>>> dev.shots
57
>>> dev.shot_vector
[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 queueable 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** = '1.24.3.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

<code>access_state([wires])</code>	Check that the device has access to an internal state and return it if available.
<code>active_wires(operators)</code>	Returns the wires acted on by a set of operators.
<code>adjoint_jacobian(tape[, starting_state, ...])</code>	Implements the adjoint method outlined in <a href="#">Jones and Gacon</a> to differentiate an input tape.
<code>analytic_probability([wires])</code>	Return the (marginal) probability of each computational basis state from the last run of the device.
<code>apply(operations[, rotations, ...])</code>	Instantiate Braket Circuit object.
<code>batch_execute(circuits, **run_kwargs)</code>	Execute a batch of quantum circuits on the device.
<code>batch_transform(circuit)</code>	Apply a differentiable batch transform for preprocessing a circuit prior to execution.
<code>capabilities()</code>	Add support for AG on sv1
<code>check_validity(queue, observables)</code>	Check validity of pulse operations before running the standard <code>check_validity</code> function
<code>classical_shadow(obs, circuit)</code>	Returns the measured bits and recipes in the classical shadow protocol.
<code>custom_expand(fn)</code>	Register a custom expansion function for the device.
<code>default_expand_fn(circuit[, max_expansion])</code>	Method for expanding or decomposing an input circuit.
<code>define_wire_map(wires)</code>	Create the map from user-provided wire labels to the wire labels used by the device.
<code>density_matrix(wires)</code>	Returns the reduced density matrix over the given wires.
<code>estimate_probability([wires, shot_range, ...])</code>	Return the estimated probability of each computational basis state using the generated samples.
<code>execute(circuit[, compute_gradient])</code>	It executes a queue of quantum operations on the device and then measure the given observables.
<code>execute_and_gradients(circuits, **kwargs)</code>	Execute a list of circuits and calculate their gradients.
<code>execution_context()</code>	The device execution context used during calls to <code>execute()</code> .
<code>expand_fn(circuit[, max_expansion])</code>	Method for expanding or decomposing an input circuit.
<code>expval(observable[, shot_range, bin_size])</code>	Returns the expectation value of observable on specified wires.
<code>generate_basis_states(num_wires[, dtype])</code>	Generates basis states in binary representation according to the number of wires specified.
<code>generate_samples()</code>	Returns the computational basis samples generated for all wires.
<code>gradients(circuits[, method])</code>	Return the gradients of a batch of quantum circuits on the device.
<code>map_wires(wires)</code>	Map the wire labels of wires using this device's wire map.

continues on next page

Table 2 – continued from previous page

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

Instantiate Braket Circuit object.

**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*)

Check validity of pulse operations before running the standard check\_validity function

Checks whether the operations and observables in queue are all supported by the device. Runs the standard check\_validity function for a PennyLane device, and an additional check to validate any pulse-operations in the form of a ParametrizedEvolution operation.

**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
- **RuntimeError** – if there are ParametrizedEvolution operations in the queue that are not supported because of invalid pulse parameters

**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.legacy", 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**

```
>>> 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) → 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**

- **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.

**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**

statistics 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**(*braket\_result: GateModelQuantumTaskResult*, *measurements: Sequence[MeasurementProcess]*) → *list[float]*

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

**Parameters**

- **braket\_result** (*GateModelQuantumTaskResult*) – the Braket task result
- **measurements** (*Sequence[MeasurementProcess]*) – the list of measurements

**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  $\text{var}(A) = \psi A^2 \psi - \psi A \psi^2$

**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 acquire 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  $2\pi$ ).

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.

---

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

**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 a device that inherits from `DefaultQubitLegacy` and overrides the logic of the `qml.sample` measurement. To do so we will need to update the `measurement_map` dictionary:

```
class NewDevice(DefaultQubitLegacy):
    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****settings**

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**

```
>>> dev = qml.device("default.qubit.legacy", wires=2, shots=[3, 1, 2, 2, 2, 2, 6, 1, 1, 5, 12, 10, 10])
>>> dev.shots
57
>>> dev.shot_vector
[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 queueable object (including a PennyLane operation and observable) and returns True if supported by the device.

**Type**

`.BooleanFn`

**task**

**version** = '0.34.0'

**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

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

continues on next page

Table 3 – continued from previous page

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

**operations** (list[ParametrizedEvolution]) – a list containing a single ParametrizedEvolution operator

**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, TensorFlow, 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.legacy", 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**

```
>>> 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*, *\*\*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 [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**

statistics 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  $\text{var}(A) = \psi A^2 \psi - \psi A \psi^2$

**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()`.

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

```
class NewDevice(DefaultQubitLegacy):
    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-&gt;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**

```
>>> dev = qml.device("default.qubit.legacy", wires=2, shots=[3, 1, 2, 2, 2, 2,
↳6, 1, 1, 5, 12, 10, 10])
>>> dev.shots
57
>>> dev.shot_vector
[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 queueable 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** = '1.24.3.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

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

continues on next page

Table 4 – continued from previous page

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

continues on next page

Table 4 – continued from previous page

<code>states_to_binary(samples, num_wires[, dtype])</code>	Convert basis states from base 10 to binary representation.
<code>statistics(braket_result, measurements)</code>	Processes measurement results from a Braket task result and returns statistics.
<code>supports_observable(observable)</code>	Checks if an observable is supported by this device. Raises a <code>ValueError</code> ,
<code>supports_operation(operation)</code>	Checks if an operation is supported by this device.
<code>var(observable[, shot_range, bin_size])</code>	Returns the variance of observable on specified wires.
<code>vn_entropy(wires, log_base)</code>	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*)  
→ 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, TensorFlow, 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-&gt;\*]

**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.legacy", 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**

```
>>> 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) → 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.

**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**

statistics 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**(*braket\_result: GateModelQuantumTaskResult, measurements: Sequence[MeasurementProcess]*) → `list[float]`

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

**Parameters**

- **braket\_result** (*GateModelQuantumTaskResult*) – the Braket task result
- **measurements** (*Sequence[MeasurementProcess]*) – the list of measurements

**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  $\text{var}(A) = \psi A^2 \psi - \psi A \psi^2$ **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.

$$\text{CPhaseShift00}(\phi) = \begin{bmatrix} e^{i\phi} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

**Details:**

- Number of wires: 2
- Number of parameters: 1
- Gradient recipe:

$$\frac{d}{d\phi} \text{CPhaseShift00}(\phi) = \frac{1}{2} [\text{CPhaseShift00}(\phi + \pi/2) - \text{CPhaseShift00}(\phi - \pi/2)]$$

#### Parameters

- **phi** (*float*) – the controlled phase angle
- **wires** (*int*) – the subsystem the gate acts on
- **id** (*str, optional*) – String representing the operation. Default: None

<i>arithmetic_depth</i>	Arithmetic depth of the operator.
<i>basis</i>	The basis of an operation, or for controlled gates, of the target operation.
<i>batch_size</i>	Batch size of the operator if it is used with broadcasted parameters.
<i>control_wires</i>	Control wires of the operator.
<i>grad_method</i>	
<i>grad_recipe</i>	Gradient recipe for the parameter-shift method.
<i>has_adjoint</i>	
<i>has_decomposition</i>	
<i>has_diagonalizing_gates</i>	
<i>has_generator</i>	
<i>has_matrix</i>	
<i>hash</i>	Integer hash that uniquely represents the operator.
<i>hyperparameters</i>	Dictionary of non-trainable variables that this operation depends on.
<i>id</i>	Custom string to label a specific operator instance.
<i>is_hermitian</i>	This property determines if an operator is hermitian.
<i>name</i>	String for the name of the operator.
<i>ndim_params</i>	Number of dimensions per trainable parameter of the operator.
<i>num_params</i>	
<i>num_wires</i>	Number of wires the operator acts on.
<i>parameter_frequencies</i>	
<i>parameters</i>	Trainable parameters that the operator depends on.
<i>pauli_rep</i>	A <code>PauliSentence</code> representation of the Operator, or <code>None</code> if it doesn't have one.
<i>wires</i>	Wires that the operator acts on.

#### **arithmetic\_depth**

Arithmetic depth of the operator.

#### **basis**

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'****grad\_recipe = None**

Gradient recipe for the parameter-shift method.

This is a tuple with one nested list per operation parameter. For parameter  $\phi_k$ , 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.

**pauli\_rep**

A PauliSentence representation of the Operator, or None if it doesn't have one.

**wires**

Wires that the operator acts on.

**Returns**

wires

**Return type**

Wires

<code>adjoint()</code>	Create an operation that is the adjoint of this one.
<code>compute_decomposition(phi, wires)</code>	Representation of the operator as a product of other operators (static method).
<code>compute_diagonalizing_gates(*params, wires, ...)</code>	Sequence of gates that diagonalize the operator in the computational basis (static method).
<code>compute_eigvals(*params, **hyperparams)</code>	Eigenvalues of the operator in the computational basis (static method).
<code>compute_matrix(phi)</code>	Representation of the operator as a canonical matrix in the computational basis (static method).
<code>compute_sparse_matrix(*params, **hyperparams)</code>	Representation of the operator as a sparse matrix in the computational basis (static method).
<code>decomposition()</code>	Representation of the operator as a product of other operators.
<code>diagonalizing_gates()</code>	Sequence of gates that diagonalize the operator in the computational basis.
<code>eigvals()</code>	Eigenvalues of the operator in the computational basis.
<code>expand()</code>	Returns a tape that contains the decomposition of the operator.
<code>generator()</code>	Generator of an operator that is in single-parameter-form.
<code>label([decimals, base_label, cache])</code>	A customizable string representation of the operator.
<code>map_wires(wire_map)</code>	Returns a copy of the current operator with its wires changed according to the given wire map.
<code>matrix([wire_order])</code>	Representation of the operator as a matrix in the computational basis.
<code>pow(z)</code>	A list of new operators equal to this one raised to the given power.
<code>queue([context])</code>	Append the operator to the Operator queue.
<code>simplify()</code>	Reduce the depth of nested operators to the minimum.
<code>single_qubit_rot_angles()</code>	The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.
<code>sparse_matrix([wire_order])</code>	Representation of the operator as a sparse matrix in the computational basis.
<code>terms()</code>	Representation of the operator as a linear combination of other operators.
<code>validate_subspace(subspace)</code>	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  $\Sigma$  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  $\Sigma$  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  $\Sigma$  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  $\Sigma$  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(base_label="my_label")
"my_label"
>>> op = qml.RX(1.23456, wires=0, id="test_data")
```

(continues on next page)

(continued from previous page)

```

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

```

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*) → 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.queueing.QueueingManager'>*)

Append the operator to the Operator queue.

**simplify**() → 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.

**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

**CPhaseShift01**

**class CPhaseShift01(phi, wires)**

Bases: Operation

Controlled phase shift gate phasing the  $|01\rangle$  state.

$$\text{CPhaseShift01}(\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{i\phi} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

**Details:**

- Number of wires: 2
- Number of parameters: 1
- Gradient recipe:

$$\frac{d}{d\phi} \text{CPhaseShift01}(\phi) = \frac{1}{2} [\text{CPhaseShift01}(\phi + \pi/2) - \text{CPhaseShift01}(\phi - \pi/2)]$$

**Parameters**

- **phi** (float) – the controlled phase angle
- **wires** (int) – the subsystem the gate acts on
- **id** (str or None) – String representing the operation (optional)

<code>arithmetic_depth</code>	Arithmetic depth of the operator.
<code>basis</code>	The basis of an operation, or for controlled gates, of the target operation.
<code>batch_size</code>	Batch size of the operator if it is used with broadcasted parameters.
<code>control_wires</code>	Control wires of the operator.
<code>grad_method</code>	
<code>grad_recipe</code>	Gradient recipe for the parameter-shift method.
<code>has_adjoint</code>	
<code>has_decomposition</code>	
<code>has_diagonalizing_gates</code>	
<code>has_generator</code>	
<code>has_matrix</code>	
<code>hash</code>	Integer hash that uniquely represents the operator.
<code>hyperparameters</code>	Dictionary of non-trainable variables that this operation depends on.
<code>id</code>	Custom string to label a specific operator instance.
<code>is_hermitian</code>	This property determines if an operator is hermitian.
<code>name</code>	String for the name of the operator.
<code>ndim_params</code>	Number of dimensions per trainable parameter of the operator.
<code>num_params</code>	
<code>num_wires</code>	Number of wires the operator acts on.
<code>parameter_frequencies</code>	
<code>parameters</code>	Trainable parameters that the operator depends on.
<code>pauli_rep</code>	A <code>PauliSentence</code> representation of the Operator, or <code>None</code> if it doesn't have one.
<code>wires</code>	Wires that the operator acts on.

**arithmetic\_depth**

Arithmetic depth of the operator.

**basis**

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'****grad\_recipe = None**

Gradient recipe for the parameter-shift method.

This is a tuple with one nested list per operation parameter. For parameter  $\phi_k$ , 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.

**pauli\_rep**

A PauliSentence representation of the Operator, or None if it doesn't have one.

**wires**

Wires that the operator acts on.

**Returns**

wires

**Return type**

Wires

<code>adjoint()</code>	Create an operation that is the adjoint of this one.
<code>compute_decomposition(phi, wires)</code>	Representation of the operator as a product of other operators (static method).
<code>compute_diagonalizing_gates(*params, wires, ...)</code>	Sequence of gates that diagonalize the operator in the computational basis (static method).
<code>compute_eigvals(*params, **hyperparams)</code>	Eigenvalues of the operator in the computational basis (static method).
<code>compute_matrix(phi)</code>	Representation of the operator as a canonical matrix in the computational basis (static method).
<code>compute_sparse_matrix(*params, **hyperparams)</code>	Representation of the operator as a sparse matrix in the computational basis (static method).
<code>decomposition()</code>	Representation of the operator as a product of other operators.
<code>diagonalizing_gates()</code>	Sequence of gates that diagonalize the operator in the computational basis.
<code>eigvals()</code>	Eigenvalues of the operator in the computational basis.
<code>expand()</code>	Returns a tape that contains the decomposition of the operator.
<code>generator()</code>	Generator of an operator that is in single-parameter-form.
<code>label([decimals, base_label, cache])</code>	A customizable string representation of the operator.
<code>map_wires(wire_map)</code>	Returns a copy of the current operator with its wires changed according to the given wire map.
<code>matrix([wire_order])</code>	Representation of the operator as a matrix in the computational basis.
<code>pow(z)</code>	A list of new operators equal to this one raised to the given power.
<code>queue([context])</code>	Append the operator to the Operator queue.
<code>simplify()</code>	Reduce the depth of nested operators to the minimum.
<code>single_qubit_rot_angles()</code>	The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.
<code>sparse_matrix([wire_order])</code>	Representation of the operator as a sparse matrix in the computational basis.
<code>terms()</code>	Representation of the operator as a linear combination of other operators.
<code>validate_subspace(subspace)</code>	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  $\Sigma$  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  $\Sigma$  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  $\Sigma$  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  $\Sigma$  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(base_label="my_label")
"my_label"
>>> op = qml.RX(1.23456, wires=0, id="test_data")
```

(continues on next page)

(continued from previous page)

```

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

```

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*) → 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.queueing.QueueingManager'>)

Append the operator to the Operator queue.

**simplify**() → 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.

**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

**CPhaseShift10**

**class CPhaseShift10(phi, wires)**

Bases: Operation

Controlled phase shift gate phasing the  $|10\rangle$  state.

$$\text{CPhaseShift10}(\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & e^{i\phi} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

**Details:**

- Number of wires: 2
- Number of parameters: 1
- Gradient recipe:

$$\frac{d}{d\phi} \text{CPhaseShift10}(\phi) = \frac{1}{2} [\text{CPhaseShift10}(\phi + \pi/2) - \text{CPhaseShift10}(\phi - \pi/2)]$$

**Parameters**

- **phi** (float) – the controlled phase angle
- **wires** (int) – the subsystem the gate acts on
- **id** (str or None) – String representing the operation (optional)

<code>arithmetic_depth</code>	Arithmetic depth of the operator.
<code>basis</code>	The basis of an operation, or for controlled gates, of the target operation.
<code>batch_size</code>	Batch size of the operator if it is used with broadcasted parameters.
<code>control_wires</code>	Control wires of the operator.
<code>grad_method</code>	
<code>grad_recipe</code>	Gradient recipe for the parameter-shift method.
<code>has_adjoint</code>	
<code>has_decomposition</code>	
<code>has_diagonalizing_gates</code>	
<code>has_generator</code>	
<code>has_matrix</code>	
<code>hash</code>	Integer hash that uniquely represents the operator.
<code>hyperparameters</code>	Dictionary of non-trainable variables that this operation depends on.
<code>id</code>	Custom string to label a specific operator instance.
<code>is_hermitian</code>	This property determines if an operator is hermitian.
<code>name</code>	String for the name of the operator.
<code>ndim_params</code>	Number of dimensions per trainable parameter of the operator.
<code>num_params</code>	
<code>num_wires</code>	Number of wires the operator acts on.
<code>parameter_frequencies</code>	
<code>parameters</code>	Trainable parameters that the operator depends on.
<code>pauli_rep</code>	A <code>PauliSentence</code> representation of the Operator, or <code>None</code> if it doesn't have one.
<code>wires</code>	Wires that the operator acts on.

**arithmetic\_depth**

Arithmetic depth of the operator.

**basis**

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'****grad\_recipe = None**

Gradient recipe for the parameter-shift method.

This is a tuple with one nested list per operation parameter. For parameter  $\phi_k$ , 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.

**pauli\_rep**

A PauliSentence representation of the Operator, or None if it doesn't have one.

**wires**

Wires that the operator acts on.

**Returns**

wires

**Return type**

Wires

<code>adjoint()</code>	Create an operation that is the adjoint of this one.
<code>compute_decomposition(phi, wires)</code>	Representation of the operator as a product of other operators (static method).
<code>compute_diagonalizing_gates(*params, wires, ...)</code>	Sequence of gates that diagonalize the operator in the computational basis (static method).
<code>compute_eigvals(*params, **hyperparams)</code>	Eigenvalues of the operator in the computational basis (static method).
<code>compute_matrix(phi)</code>	Representation of the operator as a canonical matrix in the computational basis (static method).
<code>compute_sparse_matrix(*params, **hyperparams)</code>	Representation of the operator as a sparse matrix in the computational basis (static method).
<code>decomposition()</code>	Representation of the operator as a product of other operators.
<code>diagonalizing_gates()</code>	Sequence of gates that diagonalize the operator in the computational basis.
<code>eigvals()</code>	Eigenvalues of the operator in the computational basis.
<code>expand()</code>	Returns a tape that contains the decomposition of the operator.
<code>generator()</code>	Generator of an operator that is in single-parameter-form.
<code>label([decimals, base_label, cache])</code>	A customizable string representation of the operator.
<code>map_wires(wire_map)</code>	Returns a copy of the current operator with its wires changed according to the given wire map.
<code>matrix([wire_order])</code>	Representation of the operator as a matrix in the computational basis.
<code>pow(z)</code>	A list of new operators equal to this one raised to the given power.
<code>queue([context])</code>	Append the operator to the Operator queue.
<code>simplify()</code>	Reduce the depth of nested operators to the minimum.
<code>single_qubit_rot_angles()</code>	The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.
<code>sparse_matrix([wire_order])</code>	Representation of the operator as a sparse matrix in the computational basis.
<code>terms()</code>	Representation of the operator as a linear combination of other operators.
<code>validate_subspace(subspace)</code>	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  $\Sigma$  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  $\Sigma$  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  $\Sigma$  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  $\Sigma$  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(base_label="my_label")
"my_label"
>>> op = qml.RX(1.23456, wires=0, id="test_data")
```

(continues on next page)

(continued from previous page)

```

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

```

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*) → 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.queueing.QueueingManager'>*)

Append the operator to the Operator queue.

**simplify**() → 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.

**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

## GPI

**class GPI(phi, wires)**

Bases: `Operation`

IonQ native GPI gate.

$$\text{GPI}(\phi) = \begin{bmatrix} 0 & e^{-i\phi} \\ e^{i\phi} & 0 \end{bmatrix}.$$

**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)

<code>arithmetic_depth</code>	Arithmetic depth of the operator.
<code>basis</code>	The basis of an operation, or for controlled gates, of the target operation.
<code>batch_size</code>	Batch size of the operator if it is used with broadcasted parameters.
<code>control_wires</code>	Control wires of the operator.
<code>grad_method</code>	
<code>grad_recipe</code>	Gradient recipe for the parameter-shift method.
<code>has_adjoint</code>	
<code>has_decomposition</code>	
<code>has_diagonalizing_gates</code>	
<code>has_generator</code>	
<code>has_matrix</code>	
<code>hash</code>	Integer hash that uniquely represents the operator.
<code>hyperparameters</code>	Dictionary of non-trainable variables that this operation depends on.
<code>id</code>	Custom string to label a specific operator instance.
<code>is_hermitian</code>	This property determines if an operator is hermitian.
<code>name</code>	String for the name of the operator.
<code>ndim_params</code>	Number of dimensions per trainable parameter of the operator.
<code>num_params</code>	
<code>num_wires</code>	Number of wires the operator acts on.
<code>parameter_frequencies</code>	Returns the frequencies for each operator parameter with respect to an expectation value of the form $\langle \psi   U(\mathbf{p})^\dagger \hat{O} U(\mathbf{p})   \psi \rangle$ .
<code>parameters</code>	Trainable parameters that the operator depends on.
<code>pauli_rep</code>	A <code>PauliSentence</code> representation of the Operator, or <code>None</code> if it doesn't have one.
<code>wires</code>	Wires that the operator acts on.

**arithmetic\_depth**

Arithmetic depth of the operator.

**basis**

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  $\phi_k$ , 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.

**pauli\_rep**

A PauliSentence representation of the Operator, or None if it doesn't have one.

**wires**

Wires that the operator acts on.

**Returns**

wires

**Return type**

Wires

<code>adjoint()</code>	Create an operation that is the adjoint of this one.
<code>compute_decomposition(*params[, wires])</code>	Representation of the operator as a product of other operators (static method).
<code>compute_diagonalizing_gates(*params, wires, ...)</code>	Sequence of gates that diagonalize the operator in the computational basis (static method).
<code>compute_eigvals(*params, **hyperparams)</code>	Eigenvalues of the operator in the computational basis (static method).
<code>compute_matrix(phi)</code>	Representation of the operator as a canonical matrix in the computational basis (static method).
<code>compute_sparse_matrix(*params, **hyperparams)</code>	Representation of the operator as a sparse matrix in the computational basis (static method).
<code>decomposition()</code>	Representation of the operator as a product of other operators.
<code>diagonalizing_gates()</code>	Sequence of gates that diagonalize the operator in the computational basis.
<code>eigvals()</code>	Eigenvalues of the operator in the computational basis.
<code>expand()</code>	Returns a tape that contains the decomposition of the operator.
<code>generator()</code>	Generator of an operator that is in single-parameter-form.
<code>label([decimals, base_label, cache])</code>	A customizable string representation of the operator.
<code>map_wires(wire_map)</code>	Returns a copy of the current operator with its wires changed according to the given wire map.
<code>matrix([wire_order])</code>	Representation of the operator as a matrix in the computational basis.
<code>pow(z)</code>	A list of new operators equal to this one raised to the given power.
<code>queue([context])</code>	Append the operator to the Operator queue.
<code>simplify()</code>	Reduce the depth of nested operators to the minimum.
<code>single_qubit_rot_angles()</code>	The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.
<code>sparse_matrix([wire_order])</code>	Representation of the operator as a sparse matrix in the computational basis.
<code>terms()</code>	Representation of the operator as a linear combination of other operators.
<code>validate_subspace(subspace)</code>	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  $\Sigma$  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  $\Sigma$  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  $\Sigma$  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  $\Sigma$  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(base_label="my_label")
"my_label"
>>> op = qml.RX(1.23456, wires=0, id="test_data")
>>> op.label()
"RX("test_data")"
>>> op.label(decimals=2)
"RX\n(1.23,"test_data")"
>>> op.label(base_label="my_label")
"my_label("test_data")"
>>> op.label(decimals=2, base_label="my_label")
"my_label\n(1.23,"test_data")"
```

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*) → 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.queueing.QueueingManager'>*)

Append the operator to the Operator queue.

**simplify**() → 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_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

## GPi2

**class GPi2**(*phi, wires*)

Bases: `Operation`

IonQ native GPi2 gate.

$$\text{GPi2}(\phi) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -ie^{-i\phi} \\ -ie^{i\phi} & 1 \end{bmatrix}.$$

**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)

<code>arithmetic_depth</code>	Arithmetic depth of the operator.
<code>basis</code>	The basis of an operation, or for controlled gates, of the target operation.
<code>batch_size</code>	Batch size of the operator if it is used with broadcasted parameters.
<code>control_wires</code>	Control wires of the operator.
<code>grad_method</code>	
<code>grad_recipe</code>	Gradient recipe for the parameter-shift method.
<code>has_adjoint</code>	
<code>has_decomposition</code>	
<code>has_diagonalizing_gates</code>	
<code>has_generator</code>	
<code>has_matrix</code>	
<code>hash</code>	Integer hash that uniquely represents the operator.
<code>hyperparameters</code>	Dictionary of non-trainable variables that this operation depends on.
<code>id</code>	Custom string to label a specific operator instance.
<code>is_hermitian</code>	This property determines if an operator is hermitian.
<code>name</code>	String for the name of the operator.
<code>ndim_params</code>	Number of dimensions per trainable parameter of the operator.
<code>num_params</code>	
<code>num_wires</code>	Number of wires the operator acts on.
<code>parameter_frequencies</code>	Returns the frequencies for each operator parameter with respect to an expectation value of the form $\langle \psi   U(\mathbf{p})^\dagger \hat{O} U(\mathbf{p})   \psi \rangle$ .
<code>parameters</code>	Trainable parameters that the operator depends on.
<code>pauli_rep</code>	A <code>PauliSentence</code> representation of the Operator, or <code>None</code> if it doesn't have one.
<code>wires</code>	Wires that the operator acts on.

**arithmetic\_depth**

Arithmetic depth of the operator.

**basis**

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  $\phi_k$ , 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,)]
```

(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.

**pauli\_rep**

A PauliSentence representation of the Operator, or None if it doesn't have one.

**wires**

Wires that the operator acts on.

**Returns**

wires

**Return type**

Wires

<code>adjoint()</code>	Create an operation that is the adjoint of this one.
<code>compute_decomposition(*params[, wires])</code>	Representation of the operator as a product of other operators (static method).
<code>compute_diagonalizing_gates(*params, wires, ...)</code>	Sequence of gates that diagonalize the operator in the computational basis (static method).
<code>compute_eigvals(*params, **hyperparams)</code>	Eigenvalues of the operator in the computational basis (static method).
<code>compute_matrix(phi)</code>	Representation of the operator as a canonical matrix in the computational basis (static method).
<code>compute_sparse_matrix(*params, **hyperparams)</code>	Representation of the operator as a sparse matrix in the computational basis (static method).
<code>decomposition()</code>	Representation of the operator as a product of other operators.
<code>diagonalizing_gates()</code>	Sequence of gates that diagonalize the operator in the computational basis.
<code>eigvals()</code>	Eigenvalues of the operator in the computational basis.
<code>expand()</code>	Returns a tape that contains the decomposition of the operator.
<code>generator()</code>	Generator of an operator that is in single-parameter-form.
<code>label([decimals, base_label, cache])</code>	A customizable string representation of the operator.
<code>map_wires(wire_map)</code>	Returns a copy of the current operator with its wires changed according to the given wire map.
<code>matrix([wire_order])</code>	Representation of the operator as a matrix in the computational basis.
<code>pow(z)</code>	A list of new operators equal to this one raised to the given power.
<code>queue([context])</code>	Append the operator to the Operator queue.
<code>simplify()</code>	Reduce the depth of nested operators to the minimum.
<code>single_qubit_rot_angles()</code>	The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.
<code>sparse_matrix([wire_order])</code>	Representation of the operator as a sparse matrix in the computational basis.
<code>terms()</code>	Representation of the operator as a linear combination of other operators.
<code>validate_subspace(subspace)</code>	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  $\Sigma$  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  $\Sigma$  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  $\Sigma$  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  $\Sigma$  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(base_label="my_label")
"my_label"
>>> op = qml.RX(1.23456, wires=0, id="test_data")
```

(continues on next page)

(continued from previous page)

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

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*) → 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.queueing.QueueingManager'>)

Append the operator to the Operator queue.

**simplify**() → 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.

**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-Sørensen gate.

$$\text{MS}(\phi_0, \phi_1) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 0 & -ie^{-i(\phi_0+\phi_1)} \\ 0 & 1 & -ie^{-i(\phi_0-\phi_1)} & 0 \\ 0 & -ie^{i(\phi_0-\phi_1)} & 1 & 0 \\ -ie^{i(\phi_0+\phi_1)} & 0 & 0 & 1 \end{bmatrix}.$$

**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)

<code>arithmetic_depth</code>	Arithmetic depth of the operator.
<code>basis</code>	The basis of an operation, or for controlled gates, of the target operation.
<code>batch_size</code>	Batch size of the operator if it is used with broadcasted parameters.
<code>control_wires</code>	Control wires of the operator.
<code>grad_method</code>	
<code>grad_recipe</code>	Gradient recipe for the parameter-shift method.
<code>has_adjoint</code>	
<code>has_decomposition</code>	
<code>has_diagonalizing_gates</code>	
<code>has_generator</code>	
<code>has_matrix</code>	
<code>hash</code>	Integer hash that uniquely represents the operator.
<code>hyperparameters</code>	Dictionary of non-trainable variables that this operation depends on.
<code>id</code>	Custom string to label a specific operator instance.
<code>is_hermitian</code>	This property determines if an operator is hermitian.
<code>name</code>	String for the name of the operator.
<code>ndim_params</code>	Number of dimensions per trainable parameter of the operator.
<code>num_params</code>	
<code>num_wires</code>	Number of wires the operator acts on.
<code>parameter_frequencies</code>	Returns the frequencies for each operator parameter with respect to an expectation value of the form $\langle \psi   U(\mathbf{p})^\dagger \hat{O} U(\mathbf{p})   \psi \rangle$ .
<code>parameters</code>	Trainable parameters that the operator depends on.
<code>pauli_rep</code>	A <code>PauliSentence</code> representation of the Operator, or <code>None</code> if it doesn't have one.
<code>wires</code>	Wires that the operator acts on.

**arithmetic\_depth**

Arithmetic depth of the operator.

**basis**

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  $\phi_k$ , 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.

**pauli\_rep**

A PauliSentence representation of the Operator, or None if it doesn't have one.

**wires**

Wires that the operator acts on.

**Returns**

wires

**Return type**

Wires

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

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

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*) → 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.queueing.QueueingManager'>*)

Append the operator to the Operator queue.

**simplify**() → 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_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.

$$\text{PSWAP}(\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & e^{i\phi} & 0 \\ 0 & e^{i\phi} & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

**Details:**

- Number of wires: 2

- Number of parameters: 1
- Gradient recipe:

$$\frac{d}{d\phi} \text{PSWAP}(\phi) = \frac{1}{2} [\text{PSWAP}(\phi + \pi/2) - \text{PSWAP}(\phi - \pi/2)]$$

#### Parameters

- **phi** (*float*) – the phase angle
- **wires** (*int*) – the subsystem the gate acts on
- **id** (*str or None*) – String representing the operation (optional)

<code>arithmetic_depth</code>	Arithmetic depth of the operator.
<code>basis</code>	The basis of an operation, or for controlled gates, of the target operation.
<code>batch_size</code>	Batch size of the operator if it is used with broadcasted parameters.
<code>control_wires</code>	Control wires of the operator.
<code>grad_method</code>	
<code>grad_recipe</code>	Gradient recipe for the parameter-shift method.
<code>has_adjoint</code>	
<code>has_decomposition</code>	
<code>has_diagonalizing_gates</code>	
<code>has_generator</code>	
<code>has_matrix</code>	
<code>hash</code>	Integer hash that uniquely represents the operator.
<code>hyperparameters</code>	Dictionary of non-trainable variables that this operation depends on.
<code>id</code>	Custom string to label a specific operator instance.
<code>is_hermitian</code>	This property determines if an operator is hermitian.
<code>name</code>	String for the name of the operator.
<code>ndim_params</code>	Number of dimensions per trainable parameter of the operator.
<code>num_params</code>	
<code>num_wires</code>	Number of wires the operator acts on.
<code>parameter_frequencies</code>	Returns the frequencies for each operator parameter with respect to an expectation value of the form $\langle \psi   U(\mathbf{p})^\dagger \hat{O} U(\mathbf{p})   \psi \rangle$ .
<code>parameters</code>	Trainable parameters that the operator depends on.
<code>pauli_rep</code>	A <code>PauliSentence</code> representation of the Operator, or <code>None</code> if it doesn't have one.
<code>wires</code>	Wires that the operator acts on.

#### `arithmetic_depth`

Arithmetic depth of the operator.



**basis**

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'****grad\_recipe = ([[0.5, 1, 1.5707963267948966], [-0.5, 1, -1.5707963267948966]],)**

Gradient recipe for the parameter-shift method.

This is a tuple with one nested list per operation parameter. For parameter  $\phi_k$ , 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.

**pauli\_rep**

A `PauliSentence` representation of the Operator, or `None` if it doesn't have one.

**wires**

Wires that the operator acts on.

**Returns**

wires

**Return type**

Wires

<code>adjoint()</code>	Create an operation that is the adjoint of this one.
<code>compute_decomposition(phi, wires)</code>	Representation of the operator as a product of other operators (static method).
<code>compute_diagonalizing_gates(*params, wires, ...)</code>	Sequence of gates that diagonalize the operator in the computational basis (static method).
<code>compute_eigvals(*params, **hyperparams)</code>	Eigenvalues of the operator in the computational basis (static method).
<code>compute_matrix(phi)</code>	Representation of the operator as a canonical matrix in the computational basis (static method).
<code>compute_sparse_matrix(*params, **hyperparams)</code>	Representation of the operator as a sparse matrix in the computational basis (static method).
<code>decomposition()</code>	Representation of the operator as a product of other operators.
<code>diagonalizing_gates()</code>	Sequence of gates that diagonalize the operator in the computational basis.
<code>eigvals()</code>	Eigenvalues of the operator in the computational basis.
<code>expand()</code>	Returns a tape that contains the decomposition of the operator.
<code>generator()</code>	Generator of an operator that is in single-parameter-form.
<code>label([decimals, base_label, cache])</code>	A customizable string representation of the operator.
<code>map_wires(wire_map)</code>	Returns a copy of the current operator with its wires changed according to the given wire map.
<code>matrix([wire_order])</code>	Representation of the operator as a matrix in the computational basis.
<code>pow(z)</code>	A list of new operators equal to this one raised to the given power.
<code>queue([context])</code>	Append the operator to the Operator queue.
<code>simplify()</code>	Reduce the depth of nested operators to the minimum.
<code>single_qubit_rot_angles()</code>	The parameters required to implement a single-qubit gate as an equivalent Rot gate, up to a global phase.
<code>sparse_matrix([wire_order])</code>	Representation of the operator as a sparse matrix in the computational basis.
<code>terms()</code>	Representation of the operator as a linear combination of other operators.
<code>validate_subspace(subspace)</code>	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  $\Sigma$  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  $\Sigma$  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  $\Sigma$  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  $\Sigma$  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(base_label="my_label")
"my_label"
>>> op = qml.RX(1.23456, wires=0, id="test_data")
```

(continues on next page)



(continued from previous page)

```

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

```

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*) → 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.queueing.QueueingManager'>*)

Append the operator to the Operator queue.

**simplify**() → 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.

**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



## PYTHON MODULE INDEX

### b

`braket.pennylane_plugin`, [16](#)



## A

AAMS (class in *braket.pennylane\_plugin*), 16  
 access\_state() (*BraketAwsAhsDevice* method), 34  
 access\_state() (*BraketAwsQubitDevice* method), 54  
 access\_state() (*BraketLocalAhsDevice* method), 73  
 access\_state() (*BraketLocalQubitDevice* method), 92  
 active\_wires() (*BraketAwsAhsDevice* static method), 34  
 active\_wires() (*BraketAwsQubitDevice* static method), 54  
 active\_wires() (*BraketLocalAhsDevice* static method), 73  
 active\_wires() (*BraketLocalQubitDevice* static method), 92  
 adjoint() (AAMS method), 20  
 adjoint() (*CPhaseShift00* method), 108  
 adjoint() (*CPhaseShift01* method), 119  
 adjoint() (*CPhaseShift10* method), 130  
 adjoint() (*GPI* method), 141  
 adjoint() (*GPI2* method), 153  
 adjoint() (*MS* method), 164  
 adjoint() (*PSWAP* method), 176  
 adjoint\_jacobian() (*BraketAwsAhsDevice* method), 35  
 adjoint\_jacobian() (*BraketAwsQubitDevice* method), 55  
 adjoint\_jacobian() (*BraketLocalAhsDevice* method), 74  
 adjoint\_jacobian() (*BraketLocalQubitDevice* method), 92  
 ahs\_program (*BraketAwsAhsDevice* attribute), 29  
 ahs\_program (*BraketLocalAhsDevice* attribute), 68  
 analytic (*BraketAwsAhsDevice* attribute), 29  
 analytic (*BraketAwsQubitDevice* attribute), 49  
 analytic (*BraketLocalAhsDevice* attribute), 68  
 analytic (*BraketLocalQubitDevice* attribute), 87  
 analytic\_probability() (*BraketAwsAhsDevice* method), 35  
 analytic\_probability() (*BraketAwsQubitDevice* method), 55  
 analytic\_probability() (*BraketLocalAhsDevice* method), 74

analytic\_probability() (*BraketLocalQubitDevice* method), 93  
 apply() (*BraketAwsAhsDevice* method), 36  
 apply() (*BraketAwsQubitDevice* method), 56  
 apply() (*BraketLocalAhsDevice* method), 75  
 apply() (*BraketLocalQubitDevice* method), 93  
 arithmetic\_depth (AAMS attribute), 17  
 arithmetic\_depth (*CPhaseShift00* attribute), 105  
 arithmetic\_depth (*CPhaseShift01* attribute), 116  
 arithmetic\_depth (*CPhaseShift10* attribute), 127  
 arithmetic\_depth (*GPI* attribute), 138  
 arithmetic\_depth (*GPI2* attribute), 149  
 arithmetic\_depth (*MS* attribute), 161  
 arithmetic\_depth (*PSWAP* attribute), 172  
 author (*BraketAwsAhsDevice* attribute), 29  
 author (*BraketAwsQubitDevice* attribute), 49  
 author (*BraketLocalAhsDevice* attribute), 68  
 author (*BraketLocalQubitDevice* attribute), 87

## B

basis (AAMS attribute), 17  
 basis (*CPhaseShift00* attribute), 105  
 basis (*CPhaseShift01* attribute), 116  
 basis (*CPhaseShift10* attribute), 127  
 basis (*GPI* attribute), 138  
 basis (*GPI2* attribute), 149  
 basis (*MS* attribute), 161  
 basis (*PSWAP* attribute), 172  
 batch\_execute() (*BraketAwsAhsDevice* method), 36  
 batch\_execute() (*BraketAwsQubitDevice* method), 56  
 batch\_execute() (*BraketLocalAhsDevice* method), 75  
 batch\_execute() (*BraketLocalQubitDevice* method), 93  
 batch\_size (AAMS attribute), 17  
 batch\_size (*CPhaseShift00* attribute), 106  
 batch\_size (*CPhaseShift01* attribute), 116  
 batch\_size (*CPhaseShift10* attribute), 127  
 batch\_size (*GPI* attribute), 138  
 batch\_size (*GPI2* attribute), 150  
 batch\_size (*MS* attribute), 161  
 batch\_size (*PSWAP* attribute), 173  
 batch\_transform() (*BraketAwsAhsDevice* method), 36

- batch\_transform() (*BraketAwsQubitDevice* method), 56
- batch\_transform() (*BraketLocalAhsDevice* method), 75
- batch\_transform() (*BraketLocalQubitDevice* method), 94
- braket.pennylane\_plugin module, 15
- BraketAwsAhsDevice (class *in* *braket.pennylane\_plugin*), 27
- BraketAwsQubitDevice (class *in* *braket.pennylane\_plugin*), 47
- BraketLocalAhsDevice (class *in* *braket.pennylane\_plugin*), 67
- BraketLocalQubitDevice (class *in* *braket.pennylane\_plugin*), 86
- ## C
- capabilities() (*BraketAwsAhsDevice* class method), 36
- capabilities() (*BraketAwsQubitDevice* method), 56
- capabilities() (*BraketLocalAhsDevice* class method), 75
- capabilities() (*BraketLocalQubitDevice* class method), 94
- check\_validity() (*BraketAwsAhsDevice* method), 37
- check\_validity() (*BraketAwsQubitDevice* method), 56
- check\_validity() (*BraketLocalAhsDevice* method), 76
- check\_validity() (*BraketLocalQubitDevice* method), 94
- circuit (*BraketAwsQubitDevice* attribute), 49
- circuit (*BraketLocalQubitDevice* attribute), 87
- circuit\_hash (*BraketAwsAhsDevice* attribute), 30
- circuit\_hash (*BraketAwsQubitDevice* attribute), 50
- circuit\_hash (*BraketLocalAhsDevice* attribute), 68
- circuit\_hash (*BraketLocalQubitDevice* attribute), 87
- classical\_shadow() (*BraketAwsAhsDevice* method), 37
- classical\_shadow() (*BraketAwsQubitDevice* method), 57
- classical\_shadow() (*BraketLocalAhsDevice* method), 76
- classical\_shadow() (*BraketLocalQubitDevice* method), 95
- compute\_decomposition() (AAMS static method), 21
- compute\_decomposition() (*CPhaseShift00* static method), 108
- compute\_decomposition() (*CPhaseShift01* static method), 119
- compute\_decomposition() (*CPhaseShift10* static method), 130
- compute\_decomposition() (*GPi* static method), 142
- compute\_decomposition() (*GPi2* static method), 153
- compute\_decomposition() (*MS* static method), 165
- compute\_decomposition() (*PSWAP* static method), 176
- compute\_diagonalizing\_gates() (AAMS static method), 21
- compute\_diagonalizing\_gates() (*CPhaseShift00* static method), 109
- compute\_diagonalizing\_gates() (*CPhaseShift01* static method), 120
- compute\_diagonalizing\_gates() (*CPhaseShift10* static method), 131
- compute\_diagonalizing\_gates() (*GPi* static method), 142
- compute\_diagonalizing\_gates() (*GPi2* static method), 154
- compute\_diagonalizing\_gates() (*MS* static method), 165
- compute\_diagonalizing\_gates() (*PSWAP* static method), 177
- compute\_eigvals() (AAMS static method), 22
- compute\_eigvals() (*CPhaseShift00* static method), 109
- compute\_eigvals() (*CPhaseShift01* static method), 120
- compute\_eigvals() (*CPhaseShift10* static method), 131
- compute\_eigvals() (*GPi* static method), 143
- compute\_eigvals() (*GPi2* static method), 154
- compute\_eigvals() (*MS* static method), 166
- compute\_eigvals() (*PSWAP* static method), 177
- compute\_matrix() (AAMS static method), 22
- compute\_matrix() (*CPhaseShift00* static method), 110
- compute\_matrix() (*CPhaseShift01* static method), 121
- compute\_matrix() (*CPhaseShift10* static method), 132
- compute\_matrix() (*GPi* static method), 143
- compute\_matrix() (*GPi2* static method), 155
- compute\_matrix() (*MS* static method), 166
- compute\_matrix() (*PSWAP* static method), 178
- compute\_sparse\_matrix() (AAMS static method), 22
- compute\_sparse\_matrix() (*CPhaseShift00* static method), 110
- compute\_sparse\_matrix() (*CPhaseShift01* static method), 121
- compute\_sparse\_matrix() (*CPhaseShift10* static method), 132
- compute\_sparse\_matrix() (*GPi* static method), 143
- compute\_sparse\_matrix() (*GPi2* static method), 155
- compute\_sparse\_matrix() (*MS* static method), 166
- compute\_sparse\_matrix() (*PSWAP* static method), 178
- control\_wires (AAMS attribute), 18
- control\_wires (*CPhaseShift00* attribute), 106
- control\_wires (*CPhaseShift01* attribute), 117



control\_wires (*CPhaseShift10 attribute*), 128  
 control\_wires (*GPI attribute*), 139  
 control\_wires (*GPI2 attribute*), 150  
 control\_wires (*MS attribute*), 162  
 control\_wires (*PSWAP attribute*), 173  
 CPhaseShift00 (*class in braket.pennylane\_plugin*), 104  
 CPhaseShift01 (*class in braket.pennylane\_plugin*), 115  
 CPhaseShift10 (*class in braket.pennylane\_plugin*), 126  
 create\_ahs\_program() (*BraketAwsAhsDevice method*), 38  
 create\_ahs\_program() (*BraketLocalAhsDevice method*), 77  
 custom\_expand() (*BraketAwsAhsDevice method*), 38  
 custom\_expand() (*BraketAwsQubitDevice method*), 57  
 custom\_expand() (*BraketLocalAhsDevice method*), 77  
 custom\_expand() (*BraketLocalQubitDevice method*), 95

## D

decomposition() (*AAMS method*), 23  
 decomposition() (*CPhaseShift00 method*), 111  
 decomposition() (*CPhaseShift01 method*), 122  
 decomposition() (*CPhaseShift10 method*), 133  
 decomposition() (*GPI method*), 144  
 decomposition() (*GPI2 method*), 156  
 decomposition() (*MS method*), 167  
 decomposition() (*PSWAP method*), 179  
 default\_expand\_fn() (*BraketAwsAhsDevice method*), 38  
 default\_expand\_fn() (*BraketAwsQubitDevice method*), 58  
 default\_expand\_fn() (*BraketLocalAhsDevice method*), 77  
 default\_expand\_fn() (*BraketLocalQubitDevice method*), 95  
 define\_wire\_map() (*BraketAwsAhsDevice method*), 39  
 define\_wire\_map() (*BraketAwsQubitDevice method*), 58  
 define\_wire\_map() (*BraketLocalAhsDevice method*), 78  
 define\_wire\_map() (*BraketLocalQubitDevice method*), 96  
 density\_matrix() (*BraketAwsAhsDevice method*), 39  
 density\_matrix() (*BraketAwsQubitDevice method*), 59  
 density\_matrix() (*BraketLocalAhsDevice method*), 78  
 density\_matrix() (*BraketLocalQubitDevice method*), 96  
 diagonalizing\_gates() (*AAMS method*), 23  
 diagonalizing\_gates() (*CPhaseShift00 method*), 111  
 diagonalizing\_gates() (*CPhaseShift01 method*), 122  
 diagonalizing\_gates() (*CPhaseShift10 method*), 133  
 diagonalizing\_gates() (*GPI method*), 144

diagonalizing\_gates() (*GPI2 method*), 156  
 diagonalizing\_gates() (*MS method*), 167  
 diagonalizing\_gates() (*PSWAP method*), 179

## E

eigvals() (*AAMS method*), 23  
 eigvals() (*CPhaseShift00 method*), 111  
 eigvals() (*CPhaseShift01 method*), 122  
 eigvals() (*CPhaseShift10 method*), 133  
 eigvals() (*GPI method*), 144  
 eigvals() (*GPI2 method*), 156  
 eigvals() (*MS method*), 167  
 eigvals() (*PSWAP method*), 179  
 estimate\_probability() (*BraketAwsAhsDevice method*), 39  
 estimate\_probability() (*BraketAwsQubitDevice method*), 59  
 estimate\_probability() (*BraketLocalAhsDevice method*), 78  
 estimate\_probability() (*BraketLocalQubitDevice method*), 97  
 execute() (*BraketAwsAhsDevice method*), 40  
 execute() (*BraketAwsQubitDevice method*), 59  
 execute() (*BraketLocalAhsDevice method*), 79  
 execute() (*BraketLocalQubitDevice method*), 97  
 execute\_and\_gradients() (*BraketAwsAhsDevice method*), 40  
 execute\_and\_gradients() (*BraketAwsQubitDevice method*), 60  
 execute\_and\_gradients() (*BraketLocalAhsDevice method*), 79  
 execute\_and\_gradients() (*BraketLocalQubitDevice method*), 97  
 execution\_context() (*BraketAwsAhsDevice method*), 40  
 execution\_context() (*BraketAwsQubitDevice method*), 60  
 execution\_context() (*BraketLocalAhsDevice method*), 79  
 execution\_context() (*BraketLocalQubitDevice method*), 98  
 expand() (*AAMS method*), 24  
 expand() (*CPhaseShift00 method*), 112  
 expand() (*CPhaseShift01 method*), 123  
 expand() (*CPhaseShift10 method*), 134  
 expand() (*GPI method*), 145  
 expand() (*GPI2 method*), 157  
 expand() (*MS method*), 168  
 expand() (*PSWAP method*), 180  
 expand\_fn() (*BraketAwsAhsDevice method*), 40  
 expand\_fn() (*BraketAwsQubitDevice method*), 60  
 expand\_fn() (*BraketLocalAhsDevice method*), 79  
 expand\_fn() (*BraketLocalQubitDevice method*), 98  
 expval() (*BraketAwsAhsDevice method*), 41

`expval()` (*BraketAwsQubitDevice method*), 60  
`expval()` (*BraketLocalAhsDevice method*), 80  
`expval()` (*BraketLocalQubitDevice method*), 98

## G

`generate_basis_states()` (*BraketAwsAhsDevice static method*), 41  
`generate_basis_states()` (*BraketAwsQubitDevice static method*), 60  
`generate_basis_states()` (*BraketLocalAhsDevice static method*), 80  
`generate_basis_states()` (*BraketLocalQubitDevice static method*), 98  
`generate_samples()` (*BraketAwsAhsDevice method*), 41  
`generate_samples()` (*BraketAwsQubitDevice method*), 61  
`generate_samples()` (*BraketLocalAhsDevice method*), 80  
`generate_samples()` (*BraketLocalQubitDevice method*), 99  
`generator()` (AAMS method), 24  
`generator()` (CPhaseShift00 method), 112  
`generator()` (CPhaseShift01 method), 123  
`generator()` (CPhaseShift10 method), 134  
`generator()` (GPi method), 145  
`generator()` (GPi2 method), 157  
`generator()` (MS method), 168  
`generator()` (PSWAP method), 180  
GPi (*class in braket.pennylane\_plugin*), 137  
GPi2 (*class in braket.pennylane\_plugin*), 148  
`grad_method` (AAMS attribute), 18  
`grad_method` (CPhaseShift00 attribute), 106  
`grad_method` (CPhaseShift01 attribute), 117  
`grad_method` (CPhaseShift10 attribute), 128  
`grad_method` (GPi attribute), 139  
`grad_method` (GPi2 attribute), 150  
`grad_method` (MS attribute), 162  
`grad_method` (PSWAP attribute), 173  
`grad_recipe` (AAMS attribute), 18  
`grad_recipe` (CPhaseShift00 attribute), 106  
`grad_recipe` (CPhaseShift01 attribute), 117  
`grad_recipe` (CPhaseShift10 attribute), 128  
`grad_recipe` (GPi attribute), 139  
`grad_recipe` (GPi2 attribute), 150  
`grad_recipe` (MS attribute), 162  
`grad_recipe` (PSWAP attribute), 173  
`gradients()` (*BraketAwsAhsDevice method*), 42  
`gradients()` (*BraketAwsQubitDevice method*), 61  
`gradients()` (*BraketLocalAhsDevice method*), 81  
`gradients()` (*BraketLocalQubitDevice method*), 99

## H

`hardware_capabilities` (*BraketAwsAhsDevice*

*attribute*), 30  
`has_adjoint` (AAMS attribute), 18  
`has_adjoint` (CPhaseShift00 attribute), 106  
`has_adjoint` (CPhaseShift01 attribute), 117  
`has_adjoint` (CPhaseShift10 attribute), 128  
`has_adjoint` (GPi attribute), 139  
`has_adjoint` (GPi2 attribute), 150  
`has_adjoint` (MS attribute), 162  
`has_adjoint` (PSWAP attribute), 173  
`has_decomposition` (AAMS attribute), 18  
`has_decomposition` (CPhaseShift00 attribute), 106  
`has_decomposition` (CPhaseShift01 attribute), 117  
`has_decomposition` (CPhaseShift10 attribute), 128  
`has_decomposition` (GPi attribute), 139  
`has_decomposition` (GPi2 attribute), 150  
`has_decomposition` (MS attribute), 162  
`has_decomposition` (PSWAP attribute), 173  
`has_diagonalizing_gates` (AAMS attribute), 18  
`has_diagonalizing_gates` (CPhaseShift00 attribute), 106  
`has_diagonalizing_gates` (CPhaseShift01 attribute), 117  
`has_diagonalizing_gates` (CPhaseShift10 attribute), 128  
`has_diagonalizing_gates` (GPi attribute), 139  
`has_diagonalizing_gates` (GPi2 attribute), 150  
`has_diagonalizing_gates` (MS attribute), 162  
`has_diagonalizing_gates` (PSWAP attribute), 173  
`has_generator` (AAMS attribute), 18  
`has_generator` (CPhaseShift00 attribute), 106  
`has_generator` (CPhaseShift01 attribute), 117  
`has_generator` (CPhaseShift10 attribute), 128  
`has_generator` (GPi attribute), 139  
`has_generator` (GPi2 attribute), 150  
`has_generator` (MS attribute), 162  
`has_generator` (PSWAP attribute), 173  
`has_matrix` (AAMS attribute), 18  
`has_matrix` (CPhaseShift00 attribute), 106  
`has_matrix` (CPhaseShift01 attribute), 117  
`has_matrix` (CPhaseShift10 attribute), 128  
`has_matrix` (GPi attribute), 139  
`has_matrix` (GPi2 attribute), 150  
`has_matrix` (MS attribute), 162  
`has_matrix` (PSWAP attribute), 173  
`hash` (AAMS attribute), 18  
`hash` (CPhaseShift00 attribute), 106  
`hash` (CPhaseShift01 attribute), 117  
`hash` (CPhaseShift10 attribute), 128  
`hash` (GPi attribute), 139  
`hash` (GPi2 attribute), 150  
`hash` (MS attribute), 162  
`hash` (PSWAP attribute), 173  
`hyperparameters` (AAMS attribute), 18  
`hyperparameters` (CPhaseShift00 attribute), 106

hyperparameters (*CPhaseShift01 attribute*), 117  
 hyperparameters (*CPhaseShift10 attribute*), 128  
 hyperparameters (*GPI attribute*), 139  
 hyperparameters (*GPI2 attribute*), 150  
 hyperparameters (*MS attribute*), 162  
 hyperparameters (*PSWAP attribute*), 174

## I

id (*AAMS attribute*), 18  
 id (*CPhaseShift00 attribute*), 107  
 id (*CPhaseShift01 attribute*), 117  
 id (*CPhaseShift10 attribute*), 128  
 id (*GPI attribute*), 139  
 id (*GPI2 attribute*), 151  
 id (*MS attribute*), 162  
 id (*PSWAP attribute*), 174  
 is\_hermitian (*AAMS attribute*), 19  
 is\_hermitian (*CPhaseShift00 attribute*), 107  
 is\_hermitian (*CPhaseShift01 attribute*), 117  
 is\_hermitian (*CPhaseShift10 attribute*), 128  
 is\_hermitian (*GPI attribute*), 140  
 is\_hermitian (*GPI2 attribute*), 151  
 is\_hermitian (*MS attribute*), 163  
 is\_hermitian (*PSWAP attribute*), 174

## L

label() (*AAMS method*), 24  
 label() (*CPhaseShift00 method*), 112  
 label() (*CPhaseShift01 method*), 123  
 label() (*CPhaseShift10 method*), 134  
 label() (*GPI method*), 145  
 label() (*GPI2 method*), 157  
 label() (*MS method*), 168  
 label() (*PSWAP method*), 180

## M

map\_wires() (*AAMS method*), 25  
 map\_wires() (*BraketAwsAhsDevice method*), 42  
 map\_wires() (*BraketAwsQubitDevice method*), 61  
 map\_wires() (*BraketLocalAhsDevice method*), 81  
 map\_wires() (*BraketLocalQubitDevice method*), 99  
 map\_wires() (*CPhaseShift00 method*), 113  
 map\_wires() (*CPhaseShift01 method*), 124  
 map\_wires() (*CPhaseShift10 method*), 135  
 map\_wires() (*GPI method*), 146  
 map\_wires() (*GPI2 method*), 158  
 map\_wires() (*MS method*), 169  
 map\_wires() (*PSWAP method*), 181  
 marginal\_prob() (*BraketAwsAhsDevice method*), 42  
 marginal\_prob() (*BraketAwsQubitDevice method*), 62  
 marginal\_prob() (*BraketLocalAhsDevice method*), 81  
 marginal\_prob() (*BraketLocalQubitDevice method*), 100  
 matrix() (*AAMS method*), 26

matrix() (*CPhaseShift00 method*), 113  
 matrix() (*CPhaseShift01 method*), 124  
 matrix() (*CPhaseShift10 method*), 135  
 matrix() (*GPI method*), 147  
 matrix() (*GPI2 method*), 158  
 matrix() (*MS method*), 170  
 matrix() (*PSWAP method*), 181  
 measurement\_map (*BraketAwsAhsDevice attribute*), 30  
 measurement\_map (*BraketAwsQubitDevice attribute*), 50  
 measurement\_map (*BraketLocalAhsDevice attribute*), 69  
 measurement\_map (*BraketLocalQubitDevice attribute*), 87  
 module  
   braket.pennylane\_plugin, 15  
 MS (*class in braket.pennylane\_plugin*), 160  
 mutual\_info() (*BraketAwsAhsDevice method*), 43  
 mutual\_info() (*BraketAwsQubitDevice method*), 62  
 mutual\_info() (*BraketLocalAhsDevice method*), 82  
 mutual\_info() (*BraketLocalQubitDevice method*), 100

## N

name (*AAMS attribute*), 19  
 name (*BraketAwsAhsDevice attribute*), 30  
 name (*BraketAwsQubitDevice attribute*), 50  
 name (*BraketLocalAhsDevice attribute*), 69  
 name (*BraketLocalQubitDevice attribute*), 88  
 name (*CPhaseShift00 attribute*), 107  
 name (*CPhaseShift01 attribute*), 118  
 name (*CPhaseShift10 attribute*), 129  
 name (*GPI attribute*), 140  
 name (*GPI2 attribute*), 151  
 name (*MS attribute*), 163  
 name (*PSWAP attribute*), 174  
 ndim\_params (*AAMS attribute*), 19  
 ndim\_params (*CPhaseShift00 attribute*), 107  
 ndim\_params (*CPhaseShift01 attribute*), 118  
 ndim\_params (*CPhaseShift10 attribute*), 129  
 ndim\_params (*GPI attribute*), 140  
 ndim\_params (*GPI2 attribute*), 151  
 ndim\_params (*MS attribute*), 163  
 ndim\_params (*PSWAP attribute*), 174  
 num\_executions (*BraketAwsAhsDevice attribute*), 30  
 num\_executions (*BraketAwsQubitDevice attribute*), 50  
 num\_executions (*BraketLocalAhsDevice attribute*), 69  
 num\_executions (*BraketLocalQubitDevice attribute*), 88  
 num\_params (*AAMS attribute*), 19  
 num\_params (*CPhaseShift00 attribute*), 107  
 num\_params (*CPhaseShift01 attribute*), 118  
 num\_params (*CPhaseShift10 attribute*), 129  
 num\_params (*GPI attribute*), 140  
 num\_params (*GPI2 attribute*), 151  
 num\_params (*MS attribute*), 163



num\_params (*PSWAP attribute*), 174

num\_wires (*AAMS attribute*), 19

num\_wires (*CPhaseShift00 attribute*), 107

num\_wires (*CPhaseShift01 attribute*), 118

num\_wires (*CPhaseShift10 attribute*), 129

num\_wires (*GPI attribute*), 140

num\_wires (*GPI2 attribute*), 151

num\_wires (*MS attribute*), 163

num\_wires (*PSWAP attribute*), 174

## O

obs\_queue (*BraketAwsAhsDevice attribute*), 30

obs\_queue (*BraketAwsQubitDevice attribute*), 50

obs\_queue (*BraketLocalAhsDevice attribute*), 69

obs\_queue (*BraketLocalQubitDevice attribute*), 88

observables (*BraketAwsAhsDevice attribute*), 31

observables (*BraketAwsQubitDevice attribute*), 51

observables (*BraketLocalAhsDevice attribute*), 70

observables (*BraketLocalQubitDevice attribute*), 89

op\_queue (*BraketAwsAhsDevice attribute*), 31

op\_queue (*BraketAwsQubitDevice attribute*), 51

op\_queue (*BraketLocalAhsDevice attribute*), 70

op\_queue (*BraketLocalQubitDevice attribute*), 89

operations (*BraketAwsAhsDevice attribute*), 31

operations (*BraketAwsQubitDevice attribute*), 51

operations (*BraketLocalAhsDevice attribute*), 70

operations (*BraketLocalQubitDevice attribute*), 89

order\_wires() (*BraketAwsAhsDevice method*), 43

order\_wires() (*BraketAwsQubitDevice method*), 62

order\_wires() (*BraketLocalAhsDevice method*), 82

order\_wires() (*BraketLocalQubitDevice method*), 100

## P

parallel (*BraketAwsQubitDevice attribute*), 51

parameter\_frequencies (*AAMS attribute*), 19

parameter\_frequencies (*CPhaseShift00 attribute*), 107

parameter\_frequencies (*CPhaseShift01 attribute*), 118

parameter\_frequencies (*CPhaseShift10 attribute*), 129

parameter\_frequencies (*GPI attribute*), 140

parameter\_frequencies (*GPI2 attribute*), 151

parameter\_frequencies (*MS attribute*), 163

parameter\_frequencies (*PSWAP attribute*), 174

parameters (*AAMS attribute*), 19

parameters (*BraketAwsAhsDevice attribute*), 31

parameters (*BraketAwsQubitDevice attribute*), 51

parameters (*BraketLocalAhsDevice attribute*), 70

parameters (*BraketLocalQubitDevice attribute*), 89

parameters (*CPhaseShift00 attribute*), 107

parameters (*CPhaseShift01 attribute*), 118

parameters (*CPhaseShift10 attribute*), 129

parameters (*GPI attribute*), 140

parameters (*GPI2 attribute*), 152

parameters (*MS attribute*), 163

parameters (*PSWAP attribute*), 175

pauli\_rep (*AAMS attribute*), 20

pauli\_rep (*CPhaseShift00 attribute*), 107

pauli\_rep (*CPhaseShift01 attribute*), 118

pauli\_rep (*CPhaseShift10 attribute*), 129

pauli\_rep (*GPI attribute*), 141

pauli\_rep (*GPI2 attribute*), 152

pauli\_rep (*MS attribute*), 164

pauli\_rep (*PSWAP attribute*), 175

pennylane\_requires (*BraketAwsAhsDevice attribute*), 31

pennylane\_requires (*BraketAwsQubitDevice attribute*), 51

pennylane\_requires (*BraketLocalAhsDevice attribute*), 70

pennylane\_requires (*BraketLocalQubitDevice attribute*), 89

post\_apply() (*BraketAwsAhsDevice method*), 43

post\_apply() (*BraketAwsQubitDevice method*), 63

post\_apply() (*BraketLocalAhsDevice method*), 82

post\_apply() (*BraketLocalQubitDevice method*), 101

post\_measure() (*BraketAwsAhsDevice method*), 43

post\_measure() (*BraketAwsQubitDevice method*), 63

post\_measure() (*BraketLocalAhsDevice method*), 82

post\_measure() (*BraketLocalQubitDevice method*), 101

pow() (*AAMS method*), 26

pow() (*CPhaseShift00 method*), 114

pow() (*CPhaseShift01 method*), 125

pow() (*CPhaseShift10 method*), 136

pow() (*GPI method*), 147

pow() (*GPI2 method*), 159

pow() (*MS method*), 170

pow() (*PSWAP method*), 182

pre\_apply() (*BraketAwsAhsDevice method*), 43

pre\_apply() (*BraketAwsQubitDevice method*), 63

pre\_apply() (*BraketLocalAhsDevice method*), 82

pre\_apply() (*BraketLocalQubitDevice method*), 101

pre\_measure() (*BraketAwsAhsDevice method*), 43

pre\_measure() (*BraketAwsQubitDevice method*), 63

pre\_measure() (*BraketLocalAhsDevice method*), 82

pre\_measure() (*BraketLocalQubitDevice method*), 101

probability() (*BraketAwsAhsDevice method*), 43

probability() (*BraketAwsQubitDevice method*), 63

probability() (*BraketLocalAhsDevice method*), 82

probability() (*BraketLocalQubitDevice method*), 101

PSWAP (*class in braket.pennylane\_plugin*), 171

pulse\_settings (*BraketAwsQubitDevice attribute*), 51

## Q

queue() (*AAMS method*), 26

queue() (*CPhaseShift00 method*), 114

queue() (*CPhaseShift01 method*), 125  
 queue() (*CPhaseShift10 method*), 136  
 queue() (*GPI method*), 147  
 queue() (*GPI2 method*), 159  
 queue() (*MS method*), 170  
 queue() (*PSWAP method*), 182

## R

register (*BraketAwsAhsDevice attribute*), 31  
 register (*BraketLocalAhsDevice attribute*), 70  
 reset() (*BraketAwsAhsDevice method*), 44  
 reset() (*BraketAwsQubitDevice method*), 63  
 reset() (*BraketLocalAhsDevice method*), 83  
 reset() (*BraketLocalQubitDevice method*), 101  
 result (*BraketAwsAhsDevice attribute*), 31  
 result (*BraketLocalAhsDevice attribute*), 70

## S

sample() (*BraketAwsAhsDevice method*), 44  
 sample() (*BraketAwsQubitDevice method*), 63  
 sample() (*BraketLocalAhsDevice method*), 83  
 sample() (*BraketLocalQubitDevice method*), 101  
 sample\_basis\_states() (*BraketAwsAhsDevice method*), 44  
 sample\_basis\_states() (*BraketAwsQubitDevice method*), 64  
 sample\_basis\_states() (*BraketLocalAhsDevice method*), 83  
 sample\_basis\_states() (*BraketLocalQubitDevice method*), 102  
 settings (*BraketAwsAhsDevice attribute*), 31  
 settings (*BraketLocalAhsDevice attribute*), 70  
 shadow\_expval() (*BraketAwsAhsDevice method*), 44  
 shadow\_expval() (*BraketAwsQubitDevice method*), 64  
 shadow\_expval() (*BraketLocalAhsDevice method*), 83  
 shadow\_expval() (*BraketLocalQubitDevice method*), 102  
 short\_name (*BraketAwsAhsDevice attribute*), 31  
 short\_name (*BraketAwsQubitDevice attribute*), 52  
 short\_name (*BraketLocalAhsDevice attribute*), 70  
 short\_name (*BraketLocalQubitDevice attribute*), 89  
 shot\_vec\_statistics() (*BraketAwsAhsDevice method*), 45  
 shot\_vec\_statistics() (*BraketAwsQubitDevice method*), 64  
 shot\_vec\_statistics() (*BraketLocalAhsDevice method*), 84  
 shot\_vec\_statistics() (*BraketLocalQubitDevice method*), 102  
 shot\_vector (*BraketAwsAhsDevice attribute*), 32  
 shot\_vector (*BraketAwsQubitDevice attribute*), 52  
 shot\_vector (*BraketLocalAhsDevice attribute*), 71  
 shot\_vector (*BraketLocalQubitDevice attribute*), 89  
 shots (*BraketAwsAhsDevice attribute*), 32

shots (*BraketAwsQubitDevice attribute*), 52  
 shots (*BraketLocalAhsDevice attribute*), 71  
 shots (*BraketLocalQubitDevice attribute*), 90  
 simplify() (*AAMS method*), 26  
 simplify() (*CPhaseShift00 method*), 114  
 simplify() (*CPhaseShift01 method*), 125  
 simplify() (*CPhaseShift10 method*), 136  
 simplify() (*GPI method*), 147  
 simplify() (*GPI2 method*), 159  
 simplify() (*MS method*), 170  
 simplify() (*PSWAP method*), 182  
 single\_qubit\_rot\_angles() (*AAMS method*), 26  
 single\_qubit\_rot\_angles() (*CPhaseShift00 method*), 114  
 single\_qubit\_rot\_angles() (*CPhaseShift01 method*), 125  
 single\_qubit\_rot\_angles() (*CPhaseShift10 method*), 136  
 single\_qubit\_rot\_angles() (*GPI method*), 147  
 single\_qubit\_rot\_angles() (*GPI2 method*), 159  
 single\_qubit\_rot\_angles() (*MS method*), 170  
 single\_qubit\_rot\_angles() (*PSWAP method*), 182  
 sparse\_matrix() (*AAMS method*), 26  
 sparse\_matrix() (*CPhaseShift00 method*), 114  
 sparse\_matrix() (*CPhaseShift01 method*), 125  
 sparse\_matrix() (*CPhaseShift10 method*), 136  
 sparse\_matrix() (*GPI method*), 147  
 sparse\_matrix() (*GPI2 method*), 159  
 sparse\_matrix() (*MS method*), 170  
 sparse\_matrix() (*PSWAP method*), 182  
 state (*BraketAwsAhsDevice attribute*), 32  
 state (*BraketAwsQubitDevice attribute*), 52  
 state (*BraketLocalAhsDevice attribute*), 71  
 state (*BraketLocalQubitDevice attribute*), 90  
 states\_to\_binary() (*BraketAwsAhsDevice static method*), 45  
 states\_to\_binary() (*BraketAwsQubitDevice static method*), 65  
 states\_to\_binary() (*BraketLocalAhsDevice static method*), 84  
 states\_to\_binary() (*BraketLocalQubitDevice static method*), 102  
 statistics() (*BraketAwsAhsDevice method*), 45  
 statistics() (*BraketAwsQubitDevice method*), 65  
 statistics() (*BraketLocalAhsDevice method*), 84  
 statistics() (*BraketLocalQubitDevice method*), 103  
 stopping\_condition (*BraketAwsAhsDevice attribute*), 32  
 stopping\_condition (*BraketAwsQubitDevice attribute*), 52  
 stopping\_condition (*BraketLocalAhsDevice attribute*), 71  
 stopping\_condition (*BraketLocalQubitDevice attribute*), 90

`supports_observable()` (*BraketAwsAhsDevice method*), 46  
`supports_observable()` (*BraketAwsQubitDevice method*), 65  
`supports_observable()` (*BraketLocalAhsDevice method*), 85  
`supports_observable()` (*BraketLocalQubitDevice method*), 103  
`supports_operation()` (*BraketAwsAhsDevice method*), 46  
`supports_operation()` (*BraketAwsQubitDevice method*), 65  
`supports_operation()` (*BraketLocalAhsDevice method*), 85  
`supports_operation()` (*BraketLocalQubitDevice method*), 103

## T

`task` (*BraketAwsAhsDevice attribute*), 32  
`task` (*BraketAwsQubitDevice attribute*), 52  
`task` (*BraketLocalAhsDevice attribute*), 71  
`task` (*BraketLocalQubitDevice attribute*), 90  
`terms()` (*AAMS method*), 27  
`terms()` (*CPhaseShift00 method*), 114  
`terms()` (*CPhaseShift01 method*), 125  
`terms()` (*CPhaseShift10 method*), 136  
`terms()` (*GPI method*), 148  
`terms()` (*GPI2 method*), 159  
`terms()` (*MS method*), 171  
`terms()` (*PSWAP method*), 182

## U

`use_grouping` (*BraketAwsQubitDevice attribute*), 52

## V

`validate_subspace()` (*AAMS static method*), 27  
`validate_subspace()` (*CPhaseShift00 static method*), 115  
`validate_subspace()` (*CPhaseShift01 static method*), 126  
`validate_subspace()` (*CPhaseShift10 static method*), 137  
`validate_subspace()` (*GPI static method*), 148  
`validate_subspace()` (*GPI2 static method*), 160  
`validate_subspace()` (*MS static method*), 171  
`validate_subspace()` (*PSWAP static method*), 183  
`var()` (*BraketAwsAhsDevice method*), 46  
`var()` (*BraketAwsQubitDevice method*), 66  
`var()` (*BraketLocalAhsDevice method*), 85  
`var()` (*BraketLocalQubitDevice method*), 104  
`version` (*BraketAwsAhsDevice attribute*), 32  
`version` (*BraketAwsQubitDevice attribute*), 53  
`version` (*BraketLocalAhsDevice attribute*), 71  
`version` (*BraketLocalQubitDevice attribute*), 90

`vn_entropy()` (*BraketAwsAhsDevice method*), 47  
`vn_entropy()` (*BraketAwsQubitDevice method*), 66  
`vn_entropy()` (*BraketLocalAhsDevice method*), 86  
`vn_entropy()` (*BraketLocalQubitDevice method*), 104

## W

`wire_map` (*BraketAwsAhsDevice attribute*), 32  
`wire_map` (*BraketAwsQubitDevice attribute*), 53  
`wire_map` (*BraketLocalAhsDevice attribute*), 71  
`wire_map` (*BraketLocalQubitDevice attribute*), 90  
`wires` (*AAMS attribute*), 20  
`wires` (*BraketAwsAhsDevice attribute*), 32  
`wires` (*BraketAwsQubitDevice attribute*), 53  
`wires` (*BraketLocalAhsDevice attribute*), 71  
`wires` (*BraketLocalQubitDevice attribute*), 90  
`wires` (*CPhaseShift00 attribute*), 107  
`wires` (*CPhaseShift01 attribute*), 118  
`wires` (*CPhaseShift10 attribute*), 129  
`wires` (*GPI attribute*), 141  
`wires` (*GPI2 attribute*), 152  
`wires` (*MS attribute*), 164  
`wires` (*PSWAP attribute*), 175