# **PennyLane-Cirq Documentation**

Release 0.28.0

Xanadu Inc.

## **CONTENTS**

1	Devi	ces	3
2	Tuto		5
	2.1	Installation	5
	2.2	Support	6
	2.3	Simulator device	6
	2.4	Mixed Simulator Device	8
	2.5	The qsim device	9
	2.6	Pasqal Device	11
	2.7	pennylane-cirq	12
Рy	thon I	Module Index	107
In	dex		109

## Release 0.28.0

The PennyLane-Cirq plugin integrates the Cirq quantum computing framework with PennyLane's quantum machine learning capabilities.

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

Cirq is a software library for quantum computing.

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

CONTENTS 1

2 CONTENTS

	_
СНАРТЕЯ	R
ONE	Ε

## **DEVICES**

Currently, PennyLane-Cirq provides four Cirq devices for PennyLane:

4 Chapter 1. Devices

**CHAPTER** 

**TWO** 

## **TUTORIALS**

Check out these demos to see the PennyLane-Cirq plugin in action:

You can also try it out using any of the qubit based demos from the PennyLane documentation, for example the tutorial on qubit rotation. Simply replace 'default.qubit' with the 'cirq.simulator' device

```
dev = qml.device('cirq.simulator', wires=XXX)
```

## 2.1 Installation

This plugin requires Python version 3.8 or above, as well as PennyLane and Cirq. Installation of this plugin, as well as all dependencies, can be done using pip:

```
$ pip install pennylane-cirq
```

Alternatively, you can install PennyLane-Cirq from the source code by navigating to the top directory and running:

```
$ python setup.py install
```

## 2.1.1 Dependencies

PennyLane-Cirq requires the following libraries be installed:

• Python >= 3.8

as well as the following Python packages:

- PennyLane >= 0.17
- Cirq >= 0.10.0

To use the qsim and qsimh devices, the qsim-Cirq interface qsimcirq is required:

· qsimcirq

It can be installed using pip:

```
$ pip install qsimcirq
```

If you currently do not have Python 3 installed, we recommend Anaconda for Python 3, a distributed version of Python packaged for scientific computation.

## **2.1.2 Tests**

To test that the PennyLane-Cirq plugin is working correctly you can run

```
$ make test
```

in the source folder.

## 2.1.3 Documentation

To build the HTML documentation, go to the top-level directory and run:

```
$ make docs
```

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

## 2.2 Support

- Source Code: https://github.com/PennyLaneAI/pennylane-cirq
- Issue Tracker: https://github.com/PennyLaneAI/pennylane-cirq/issues
- 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 Simulator device

You can instantiate the device in PennyLane as follows:

```
import pennylane as qml
dev = qml.device('cirq.simulator', wires=2)
```

This device can then be used just like other devices for the definition and evaluation of QNodes within PennyLane. A simple quantum function that returns the expectation value of a measurement and depends on three classical input parameters would look like:

```
@qml.qnode(dev)
def circuit(x, y, z):
    qml.RZ(z, wires=[0])
    qml.RY(y, wires=[0])
    qml.RX(x, wires=[0])
```

(continues on next page)

(continued from previous page)

```
qml.CNOT(wires=[0, 1])
return qml.expval(qml.PauliZ(wires=1))
```

You can then execute the circuit like any other function to get the quantum mechanical expectation value.

```
circuit(0.2, 0.1, 0.3)
```

## 2.3.1 Device options

Cirq has different ways of defining qubits, e.g., LineQubit or GridQubit. The Cirq device therefore accepts an additional argument qubits=None that you can use to define your own qubits and give them to the device as a list.

```
import cirq

qubits = [
    cirq.GridQubit(0, 0),
    cirq.GridQubit(0, 1),
    cirq.GridQubit(1, 0),
    cirq.GridQubit(1, 1),
]

dev = qml.device("cirq.simulator", wires=4, qubits=qubits)
```

The wire of each qubit corresponds to its index in the qubit list. In the above example, the wire 2 corresponds to cirq.GridQubit(1, 0).

If no qubits are given, the plugin will create an array of LineQubit instances.

#### 2.3.2 Custom simulators

The simulator device can also be instantiated using an optional custom simulator object:

```
import pennylane as qml
import cirq

sim = cirq.Simulator()
dev = qml.device("cirq.simulator", wires=2, simulator=sim)
```

If the simulator argument is not provided, the device will by default create a cirq.Simulator simulator.

## 2.3.3 Supported operations

The cirq.simulator device supports all PennyLane operations and observables.

2.3. Simulator device 7

## 2.4 Mixed Simulator Device

You can instantiate the mixed-state simulator device in PennyLane as follows:

```
import pennylane as qml
dev = qml.device('cirq.mixedsimulator', wires=2)
```

This device can then be used just like other devices for the definition and evaluation of QNodes within PennyLane. Unlike the cirq.simulator backend, this device also supports several of Cirq's custom non-unitary channels, e.g., BitFlip or Depolarize.

```
from pennylane_cirq import ops

@qml.qnode(dev)
def circuit(x, p, q):
    qml.RX(x, wires=[0])
    ops.BitFlip(p, wires=[0])
    ops.Depolarize(q, wires=[1])
    qml.CNOT(wires=[0, 1])
    return qml.expval(qml.PauliZ(wires=1))
circuit(0.2, 0.1, 0.3)
```

This device stores the internal state of the quantum simulation as a density matrix. This has additional memory overhead compared to pure-state simulation, but allows for additional channels to be performed. The density matrix can be accessed after a circuit execution using dev.state.

## 2.4.1 Device options

Cirq has different ways of defining qubits, e.g., LineQubit or GridQubit. The Cirq device therefore accepts an additional argument qubits=None that you can use to define your own qubits and give them to the device as a list.

```
import cirq

qubits = [
    cirq.GridQubit(0, 0),
    cirq.GridQubit(0, 1),
    cirq.GridQubit(1, 0),
    cirq.GridQubit(1, 1),
]

dev = qml.device("cirq.mixedsimulator", wires=4, qubits=qubits)
```

The wire of each qubit corresponds to its index in the qubit list. In the above example, the wire 2 corresponds to cirq.GridQubit(1, 0).

If no qubits are given, the plugin will create an array of LineQubit instances.

## 2.4.2 Supported operations

The cirq.mixedsimulator device supports all PennyLane operations and observables.

It also supports the following non-unitary channels from Cirq (found in pennylane\_cirq.ops): BitFlip, PhaseFlip, PhaseDamp, AmplitudeDamp, and Depolarize.

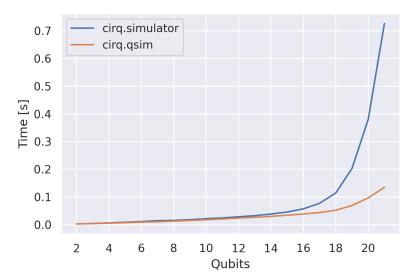
## 2.5 The qsim device

The qsim circuit simulator package provides two additional circuit simulators that can be used with PennyLane-Cirq:

- · qsim, a Schrödinger full state-vector simulator
- qsimh, a hybrid Schrödinger-Feynman simulator. This simulator cuts the qubit lattice into two parts; each part is
  individually simulated using qsim, with Feynman-style path summation used to return the final result. Compared
  to full state-vector simulation, qsimh reduces memory requirements, at the expense of an increased runtime.

For further details see the qsim website.

For a large number of qubits, these simulators are considerably faster than the Cirq simulator:



In order to use these devices, the qsimcirq package must first be installed:

```
pip install qsimcirq
```

You can instantiate the qsim device in PennyLane as follows:

```
import pennylane as qml
dev = qml.device('cirq.qsim', wires=2)
```

This device can then be used just like other devices for the evaluation of QNodes within PennyLane. A simple quantum function that returns the expectation value of a measurement and depends on three classical input parameters would look like:

```
@qml.qnode(dev)
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(wires=1))
```

You can then execute the circuit like any other function to get the quantum mechanical expectation value:

```
circuit(0.2, 0.1, 0.3)
```

The qsimh device can be loaded similarly, with the addition of the keyword argument qsimh\_options as described below.

## 2.5.1 Device options

qsim and qsimh use the same method of defining qubits as Cirq, e.g., LineQubit or GridQubit. As with the Cirq device, the qsim and qsimh devices therefore accept an additional argument qubits that can be used to define your own qubits and pass them to the device as a list.

```
import cirq

qubits = [
    cirq.GridQubit(0, 0),
    cirq.GridQubit(0, 1),
    cirq.GridQubit(1, 0),
    cirq.GridQubit(1, 1),
]

dev = qml.device("cirq.qsim", wires=4, qubits=qubits)
```

The wire of each qubit corresponds to its index in the qubit list. In the above example, the wire 2 corresponds to cirq.GridQubit(1, 0).

If no qubits are given, the plugin will create an array of LineQubit instances.

#### gsimh options

qsimh requires specific options to be set for the simulator. These can be passed by the positional argument qsimh\_options. See the qsimh usage documentation for further details.

```
import cirq

qubits = [
    cirq.GridQubit(0, 0),
    cirq.GridQubit(0, 1),
    cirq.GridQubit(1, 0),
    cirq.GridQubit(1, 1),
]

qsimh_options = {
```

(continues on next page)

(continued from previous page)

```
'k': [0],
'w': 0,
'p': 0,
'r': 2
}
dev = qml.device("cirq.qsimh", wires=4, qsimh_options=qsimh_options, qubits=qubits)
```

## 2.5.2 Supported operations

The cirq.qsim and cirq.qsimh devices support most PennyLane operations and observables, with the exceptions of inverse operations and QubitUnitary gates on 3 or more qubits.

For state preparation qsim relies on decomposing BasisState into a set of PauliX gates and QubitStateVector via Möttönen state preparation.

## 2.6 Pasqal Device

You can instantiate a simulator for Pasqal's neutral-atom devices in PennyLane as follows:

```
import pennylane as qml
dev = qml.device("cirq.pasqal", wires=2, control_radius=1.5)
```

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

The Pasqal device supports unique features of Pasqal's quantum computing hardware provided via Cirq, namely the ThreeDQubit and the notion of a control\_radius.

Note that if not specified, ThreeDGridQubits are automatically arranged in a linear arrangement along the first coordinate axis, separated by a distance of control\_radius / 2. That is, (0, 0, 0), (control\_radius/2, 0, 0), (control\_radius, 0, 0).

For more details about Pasqal devices, consult the Cirq docs.

## 2.7 pennylane-cirq

This section contains the API documentation for the PennyLane-Cirq 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 Cirq devices with PennyLane.

## 2.7.1 Plugin overview

## 2.7.2 Classes

AmplitudeDamp(*params[, wires, do_queue, id])	Cirq amplitude_damp operation.
BitFlip(*params[, wires, do_queue, id])	Cirq bit_flip operation.
Depolarize(*params[, wires, do_queue, id])	Cirq depolarize operation.
<pre>MixedStateSimulatorDevice(wires[, shots, qubits])</pre>	Cirq mixed-state simulator device for PennyLane.
PasqalDevice(wires, control_radius[, shots,])	Cirq Pasqal device for PennyLane.
PhaseDamp(*params[, wires, do_queue, id])	Cirq phase_damp operation.
PhaseFlip(*params[, wires, do_queue, id])	Cirq phase_flip operation.
SimulatorDevice(wires[, shots, qubits,])	Cirq simulator device for PennyLane.

## **AmplitudeDamp**

class AmplitudeDamp(\*params, wires=None, do\_queue=True, id=None)

Bases: pennylane.operation.Operation

Cirq amplitude\_damp operation.

See the Cirq docs for further details.

base_name	If inverse is requested, this is the name of the original
	operator to be inverted.
basis	The target operation for controlled gates.
control_wires	Control wires of the operator.
eigvals	Eigenvalues of an instantiated operator.
grad_method	
grad_recipe	Gradient recipe for the parameter-shift method.
has_matrix	
hash	Integer hash that uniquely represents the operator.
hyperparameters	Dictionary of non-trainable variables that this opera-
	tion depends on.
id	Custom string to label a specific operator instance.
inverse	Boolean determining if the inverse of the operation
	was requested.
	continues on next page

12 Chapter 2. Tutorials

Table 2 – continued from previous page

matrix	Matrix representation of an instantiated operator in
	the computational basis.
name	Name of the operator.
num_params	
num_wires	
par_domain	
parameter_frequencies	Returns the frequencies for each operator parame-
	ter with respect to an expectation value of the form
	$\langle \psi   U(\mathbf{p})^{\dagger} \hat{O} U(\mathbf{p})   \psi \rangle.$
parameters	Trainable parameters that the operator depends on.
single_qubit_rot_angles	The parameters required to implement a single-qubit
	gate as an equivalent Rot gate, up to a global phase.
wires	Wires that the operator acts on.

#### base\_name

If inverse is requested, this is the name of the original operator to be inverted.

#### basis = None

The target operation for controlled gates. 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

#### 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

## eigvals

Eigenvalues of an instantiated operator. Note that the eigenvalues are not guaranteed to be in any particular order.

**Warning:** The eigvals property is deprecated and will be removed in an upcoming release. Please use qml.eigvals instead.

## **Example:**

```
>>> U = qml.RZ(0.5, wires=1)
>>> U.eigvals
>>> array([0.96891242-0.24740396j, 0.96891242+0.24740396j])
```

**Returns** eigvals representation

**Return type** array

#### grad\_method = None

#### 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\_matrix = False

#### 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.

#### inverse

Boolean determining if the inverse of the operation was requested.

#### matrix

Matrix representation of an instantiated operator in the computational basis.

**Warning:** The matrix property is deprecated and will be removed in an upcoming release. Please use qml.matrix instead.

#### **Example:**

**Returns** matrix representation

Return type array

#### name

Name of the operator.

```
num_params = 1
num_wires = 1
par domain = 'R'
```

#### 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.

## 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]

#### wires

Wires that the operator acts on.

Returns wires

Return type Wires

<pre>adjoint([do_queue])</pre>	Create an operation that is the adjoint of this one.
<pre>compute_decomposition(*params[, wires])</pre>	Representation of the operator as a product of other
	operators (static method).
<pre>compute_diagonalizing_gates(*params, wires,</pre>	Sequence of gates that diagonalize the operator in the
)	computational basis (static method).
<pre>compute_eigvals(*params, **hyperparams)</pre>	Eigenvalues of the operator in the computational ba-
	sis (static method).
<pre>compute_matrix(*params, **hyperparams)</pre>	Representation of the operator as a canonical matrix
	in the computational basis (static method).
	continues on next page

Table 3 – continued from previous page

<pre>compute_sparse_matrix(*params, **hyper-</pre>	Representation of the operator as a sparse matrix in
params)	the computational basis (static method).
<pre>compute_terms(*params, **hyperparams)</pre>	Representation of the operator as a linear combina-
	tion of other operators (static method).
<pre>decomposition()</pre>	Representation of the operator as a product of other
	operators.
diagonalizing_gates()	Sequence of gates that diagonalize the operator in the
	computational basis.
expand()	Returns a tape that has recorded the decomposition
	of the operator.
generator()	Generator of an operator that is in single-parameter-
	form.
<pre>get_eigvals()</pre>	Eigenvalues of the operator in the computational ba-
	sis (static method).
<pre>get_matrix([wire_order])</pre>	Representation of the operator as a matrix in the com-
	putational basis.
<pre>get_parameter_shift(idx)</pre>	Multiplier and shift for the given parameter, based on
	its gradient recipe.
inv()	Inverts the operator.
label([decimals, base_label, cache])	A customizable string representation of the operator.
queue([context])	Append the operator to the Operator queue.
<pre>sparse_matrix([wire_order])</pre>	Representation of the operator as a sparse matrix in
	the computational basis.
terms()	Representation of the operator as a linear combina-
	tion of other operators.

## adjoint(do\_queue=False)

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.

**Parameters do\_queue** – Whether to add the adjointed gate to the context queue.

**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**

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

• **hyperparams** (*dict*) – non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

Returns 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.

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, 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:

get\_eigvals() and 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(\*params, \*\*hyperparams)

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:

get\_matrix() and 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** matrix representation

**Return type** scipy.sparse.coo.coo\_matrix

### static compute\_terms(\*params, \*\*hyperparams)

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

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

#### See also:

terms()

#### **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** list of coefficients and list of operations

**Return type** tuple[list[tensor\_like or float], list[Operation]]

#### 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.

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

#### expand()

Returns a tape that has recorded 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.

#### get\_eigvals()

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

If  $diagonalizing\_gates$  are specified and implement a unitary U, 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

#### get\_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

### get\_parameter\_shift(idx)

Multiplier and shift for the given parameter, based on its gradient recipe.

**Parameters** idx (int) – parameter index within the operation

**Returns** list of multiplier, coefficient, shift for each term in the gradient recipe

**Return type** list[[float, float, float]]

Note that the default value for shift is None, which is replaced by the default shift  $\pi/2$ .

#### inv()

Inverts the operator.

This method concatenates a string to the name of the operation, to indicate that the inverse will be used for computations.

Any subsequent call of this method will toggle between the original operation and the inverse of the operation.

**Returns** operation to be inverted

Return type Operator

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 caries information between label calls in the same drawing

**Returns** label to use in drawings

Return type str

#### **Example:**

```
>>> op = qml.RX(1.23456, wires=0)
>>> op.label()
"RX"
>>> op.label(decimals=2)
"RX\n(1.23)"
>>> op.label(base_label="my_label")
"my_label"
>>> op.label(decimals=2, base_label="my_label")
"my_label\n(1.23)"
>>> op.inv()
>>> op.label()
"RX1"
```

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)]
```

**queue**(context=<class 'pennylane.queuing.QueuingContext'>)
Append the operator to the Operator queue.

#### 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.

Note: The wire\_order argument is currently not implemented, and using it will raise an error.

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** matrix representation

Return type scipy.sparse.coo.coo\_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.

#### See also:

compute\_terms()

**Returns** list of coefficients  $c_i$  and list of operations  $O_i$ 

**Return type** tuple[list[tensor\_like or float], list[Operation]]

## **BitFlip**

class BitFlip(\*params, wires=None, do\_queue=True, id=None)

Bases: pennylane.operation.Operation

Cirq bit\_flip operation.

See the Cirq docs for further details.

base_name	If inverse is requested, this is the name of the original
	operator to be inverted.
basis	The target operation for controlled gates.
control_wires	Control wires of the operator.
eigvals	Eigenvalues of an instantiated operator.
grad_method	
grad_recipe	Gradient recipe for the parameter-shift method.
has_matrix	

continues on next page

Table 4 – continued from previous page

rabio i contant	idea irem previeue page
hash	Integer hash that uniquely represents the operator.
hyperparameters	Dictionary of non-trainable variables that this opera-
	tion depends on.
id	Custom string to label a specific operator instance.
inverse	Boolean determining if the inverse of the operation
	was requested.
matrix	Matrix representation of an instantiated operator in
	the computational basis.
name	Name of the operator.
num_params	
num_wires	
par_domain	
parameter_frequencies	Returns the frequencies for each operator parame-
	ter with respect to an expectation value of the form
	$\langle \psi   U(\mathbf{p})^{\dagger} \hat{O} U(\mathbf{p})   \psi \rangle$ .
parameters	Trainable parameters that the operator depends on.
single_qubit_rot_angles	The parameters required to implement a single-qubit
	gate as an equivalent Rot gate, up to a global phase.
wires	Wires that the operator acts on.

## base\_name

If inverse is requested, this is the name of the original operator to be inverted.

#### basis = None

The target operation for controlled gates. 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

## 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

## eigvals

Eigenvalues of an instantiated operator. Note that the eigenvalues are not guaranteed to be in any particular order.

**Warning:** The eigvals property is deprecated and will be removed in an upcoming release. Please use qml.eigvals instead.

## Example:

```
>>> U = qml.RZ(0.5, wires=1)
>>> U.eigvals
>>> array([0.96891242-0.24740396j, 0.96891242+0.24740396j])
```

**Returns** eigvals representation

**Return type** array

#### grad\_method = None

#### 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\_matrix = False

#### 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.

## inverse

Boolean determining if the inverse of the operation was requested.

#### matrix

Matrix representation of an instantiated operator in the computational basis.

**Warning:** The matrix property is deprecated and will be removed in an upcoming release. Please use qml.matrix instead.

## **Example:**

**Returns** matrix representation

Return type array

#### name

Name of the operator.

```
num_params = 1
num_wires = 1
par_domain = 'R'
```

### 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.

#### 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]

#### wires

Wires that the operator acts on.

Returns wires

Return type Wires

adjoint([do_queue])	Create an operation that is the adjoint of this one.
<pre>compute_decomposition(*params[, wires])</pre>	Representation of the operator as a product of other
	operators (static method).
	continues on next page

. .

Table 5 – continued from previous page

compute_eigvals(*params, **hyperparams)  Compute_matrix(*params, **hyperparams)  Compute_matrix(*params, **hyperparams)  Compute_sparse_matrix(*params, **hyperparams)  Compute_sparse_matrix(*params, **hyperparams)  Compute_sparse_matrix(*params, **hyperparams)  Compute_terms(*params, **hyperparams)  Compute_terms(*params, **hyperparams)  Compute_terms(*params, **hyperparams)  Representation of the operator as a sparse matrix in the computational basis (static method).  Representation of the operator as a linear combination of other operators (static method).  Representation of the operator as a product of other operators.  Representation of the operator as a product of other operators.  Representation of the operator as a product of other operators.  Returns a tape that has recorded the decomposition of the operator.  Returns a tape that has recorded the decomposition of the operator.  Returns a tape that has recorded the decomposition of the operator.  Returns a tape that has recorded the decomposition of the operator.  Returns a tape that has recorded the decomposition of the operator.  Representation of the operator in the computational basis (static method).  Representation of the operator as a matrix in the computational basis.  Representation of the operator as a matrix in the computational basis.  Representation of the operator as a matrix in the computational basis.  Representation of the operator as a sparse matrix in the operator.  A customizable string representation of the operator.  Append the operator to the Operator as a sparse matrix in the computational basis.  Representation of the operator as a sparse matrix in the computational basis.  Representation of the operator as a sparse matrix in the computational basis.	<pre>compute_diagonalizing_gates(*params, wires,</pre>	Sequence of gates that diagonalize the operator in the
sis (static method).  compute_matrix(*params, **hyperparams)  compute_sparse_matrix(*params, **hyperparams)  compute_sparse_matrix(*params, **hyperparams)  compute_terms(*params, **hyperparams)  compute_terms(*params, **hyperparams)  compute_terms(*params, **hyperparams)  compute_terms(*params, **hyperparams)  decomposition()  decomposition()  decomposition()  computational basis (static method).  Representation of the operator as a linear combination of other operators (static method).  decomposition()  Representation of the operator as a product of other operators.  Sequence of gates that diagonalize the operator in the computational basis.  Returns a tape that has recorded the decomposition of the operator.  generator()  Generator of an operator that is in single-parameter-form.  get_eigvals()  Eigenvalues of the operator in the computational basis (static method).  get_matrix([wire_order])  Representation of the operator as a matrix in the computational basis.  get_parameter_shift(idx)  Multiplier and shift for the given parameter, based on its gradient recipe.  inv()  Inverts the operator.  deueue([context])  A customizable string representation of the operator.  queue([context])  A customizable string representation of the operator.  Append the operator to the Operator queue.  sparse_matrix([wire_order])  Representation of the operator as a sparse matrix in the computational basis.  terms()  Representation of the operator as a linear combination of the ope	)	computational basis (static method).
compute_matrix(*params, **hyperparams)       Representation of the operator as a canonical matrix in the computational basis (static method).         compute_sparse_matrix(*params, **hyperparams)       Representation of the operator as a sparse matrix in the computational basis (static method).         compute_terms(*params, **hyperparams)       Representation of the operator as a linear combination of other operators (static method).         decomposition()       Representation of the operator as a product of other operators.         diagonalizing_gates()       Sequence of gates that diagonalize the operator in the computational basis.         expand()       Returns a tape that has recorded the decomposition of the operator.         generator()       Generator of an operator that is in single-parameter-form.         get_eigvals()       Eigenvalues of the operator in the computational basis (static method).         get_matrix([wire_order])       Representation of the operator as a matrix in the computational basis.         get_parameter_shift(idx)       Multiplier and shift for the given parameter, based on its gradient recipe.         inv()       Inverts the operator.         label([decimals, base_label, cache])       A customizable string representation of the operator.         queue([context])       Append the operator to the Operator queue.         sparse_matrix([wire_order])       Representation of the operator as a sparse matrix in the computational basis.	<pre>compute_eigvals(*params, **hyperparams)</pre>	Eigenvalues of the operator in the computational ba-
in the computational basis (static method).  compute_sparse_matrix(*params, **hyper- params)  compute_terms(*params, **hyperparams)  Representation of the operator as a sparse matrix in the computational basis (static method).  compute_terms(*params, **hyperparams)  Representation of the operator as a linear combination of other operators (static method).  decomposition()  Representation of the operator as a product of other operators.  diagonalizing_gates()  Sequence of gates that diagonalize the operator in the computational basis.  expand()  Returns a tape that has recorded the decomposition of the operator.  generator()  Generator of an operator that is in single-parameter-form.  get_eigvals()  Eigenvalues of the operator in the computational basis (static method).  get_matrix([wire_order])  Representation of the operator as a matrix in the computational basis.  get_parameter_shift(idx)  Multiplier and shift for the given parameter, based on its gradient recipe.  inv()  Inverts the operator.  label([decimals, base_label, cache])  A customizable string representation of the operator as a sparse matrix in the computational basis.  representation of the operator as a sparse matrix in the computational basis.  Representation of the operator as a sparse matrix in the computational basis.		sis (static method).
compute_sparse_matrix(*params, params)**hyper paramsRepresentation of the operator as a sparse matrix in the computational basis (static method).compute_terms(*params, **hyperparams)Representation of the operator as a linear combination of other operators (static method).decomposition()Representation of the operator as a product of other operators.diagonalizing_gates()Sequence of gates that diagonalize the operator in the computational basis.expand()Returns a tape that has recorded the decomposition of the operator.generator()Generator of an operator that is in single-parameterform.get_eigvals()Eigenvalues of the operator in the computational basis (static method).get_matrix([wire_order])Representation of the operator as a matrix in the computational basis.get_parameter_shift(idx)Multiplier and shift for the given parameter, based on its gradient recipe.inv()Inverts the operator.label([decimals, base_label, cache])A customizable string representation of the operator queue.sparse_matrix([wire_order])Representation of the operator as a sparse matrix in the computational basis.terms()Representation of the operator as a linear combina-	<pre>compute_matrix(*params, **hyperparams)</pre>	Representation of the operator as a canonical matrix
params) the computational basis (static method).  compute_terms(*params, **hyperparams) Representation of the operator as a linear combination of other operators (static method).  decomposition() Representation of the operator as a product of other operators.  diagonalizing_gates() Sequence of gates that diagonalize the operator in the computational basis.  expand() Returns a tape that has recorded the decomposition of the operator.  generator() Generator of an operator that is in single-parameter-form.  get_eigvals() Eigenvalues of the operator in the computational basis (static method).  get_matrix([wire_order]) Representation of the operator as a matrix in the computational basis.  get_parameter_shift(idx) Multiplier and shift for the given parameter, based on its gradient recipe.  inv() Inverts the operator.  label([decimals, base_label, cache]) A customizable string representation of the operator queue.  sparse_matrix([wire_order]) Representation of the operator queue.  sparse_matrix([wire_order]) Representation of the operator as a sparse matrix in the computational basis.  terms() Representation of the operator as a linear combina-		in the computational basis (static method).
compute_terms(*params, **hyperparams)Representation of the operator as a linear combination of other operators (static method).decomposition()Representation of the operator as a product of other operators.diagonalizing_gates()Sequence of gates that diagonalize the operator in the computational basis.expand()Returns a tape that has recorded the decomposition of the operator.generator()Generator of an operator that is in single-parameter-form.get_eigvals()Eigenvalues of the operator in the computational basis (static method).get_matrix([wire_order])Representation of the operator as a matrix in the computational basis.get_parameter_shift(idx)Multiplier and shift for the given parameter, based on its gradient recipe.inv()Inverts the operator.label([decimals, base_label, cache])A customizable string representation of the operator.queue([context])Append the operator to the Operator queue.sparse_matrix([wire_order])Representation of the operator as a sparse matrix in the computational basis.terms()Representation of the operator as a linear combina-	compute_sparse_matrix(*params, **hyper-	Representation of the operator as a sparse matrix in
tion of other operators (static method).  decomposition() Representation of the operator as a product of other operators.  diagonalizing_gates() Sequence of gates that diagonalize the operator in the computational basis.  expand() Returns a tape that has recorded the decomposition of the operator.  generator() Generator of an operator that is in single-parameter-form.  get_eigvals() Eigenvalues of the operator in the computational basis (static method).  get_matrix([wire_order]) Representation of the operator as a matrix in the computational basis.  get_parameter_shift(idx) Multiplier and shift for the given parameter, based on its gradient recipe.  inv() Inverts the operator.  label([decimals, base_label, cache]) A customizable string representation of the operator.  queue([context]) Representation of the operator as a sparse matrix in the computational basis.  terms() Representation of the operator as a linear combina-	params)	
decomposition()Representation of the operator as a product of other operators.diagonalizing_gates()Sequence of gates that diagonalize the operator in the computational basis.expand()Returns a tape that has recorded the decomposition of the operator.generator()Generator of an operator that is in single-parameter-form.get_eigvals()Eigenvalues of the operator in the computational basis (static method).get_matrix([wire_order])Representation of the operator as a matrix in the computational basis.get_parameter_shift(idx)Multiplier and shift for the given parameter, based on its gradient recipe.inv()Inverts the operator.label([decimals, base_label, cache])A customizable string representation of the operator.queue([context])Append the operator to the Operator queue.sparse_matrix([wire_order])Representation of the operator as a sparse matrix in the computational basis.terms()Representation of the operator as a linear combina-	<pre>compute_terms(*params, **hyperparams)</pre>	
operators.  diagonalizing_gates() Sequence of gates that diagonalize the operator in the computational basis.  expand() Returns a tape that has recorded the decomposition of the operator.  generator() Generator of an operator that is in single-parameter-form.  get_eigvals() Eigenvalues of the operator in the computational basis (static method).  get_matrix([wire_order]) Representation of the operator as a matrix in the computational basis.  get_parameter_shift(idx) Multiplier and shift for the given parameter, based on its gradient recipe.  inv() Inverts the operator.  label([decimals, base_label, cache]) A customizable string representation of the operator.  queue([context]) Append the operator to the Operator queue.  sparse_matrix([wire_order]) Representation of the operator as a linear combina-  terms() Representation of the operator as a linear combina-		tion of other operators (static method).
diagonalizing_gates()Sequence of gates that diagonalize the operator in the computational basis.expand()Returns a tape that has recorded the decomposition of the operator.generator()Generator of an operator that is in single-parameter-form.get_eigvals()Eigenvalues of the operator in the computational basis (static method).get_matrix([wire_order])Representation of the operator as a matrix in the computational basis.get_parameter_shift(idx)Multiplier and shift for the given parameter, based on its gradient recipe.inv()Inverts the operator.label([decimals, base_label, cache])A customizable string representation of the operator.queue([context])Append the operator to the Operator queue.sparse_matrix([wire_order])Representation of the operator as a sparse matrix in the computational basis.terms()Representation of the operator as a linear combina-	<pre>decomposition()</pre>	Representation of the operator as a product of other
computational basis.  Returns a tape that has recorded the decomposition of the operator.  Generator of an operator that is in single-parameter-form.  get_eigvals()  Generator of an operator that is in single-parameter-form.  get_eigvals()  Eigenvalues of the operator in the computational basis (static method).  get_matrix([wire_order])  Representation of the operator as a matrix in the computational basis.  get_parameter_shift(idx)  Multiplier and shift for the given parameter, based on its gradient recipe.  inv()  Inverts the operator.  label([decimals, base_label, cache])  A customizable string representation of the operator.  queue([context])  Append the operator to the Operator queue.  sparse_matrix([wire_order])  Representation of the operator as a sparse matrix in the computational basis.  terms()  Representation of the operator as a linear combina-		•
expand()Returns a tape that has recorded the decomposition of the operator.generator()Generator of an operator that is in single-parameter-form.get_eigvals()Eigenvalues of the operator in the computational basis (static method).get_matrix([wire_order])Representation of the operator as a matrix in the computational basis.get_parameter_shift(idx)Multiplier and shift for the given parameter, based on its gradient recipe.inv()Inverts the operator.label([decimals, base_label, cache])A customizable string representation of the operator.queue([context])Append the operator to the Operator queue.sparse_matrix([wire_order])Representation of the operator as a sparse matrix in the computational basis.terms()Representation of the operator as a linear combina-	<pre>diagonalizing_gates()</pre>	
generator()Generator of an operator that is in single-parameter-form.get_eigvals()Eigenvalues of the operator in the computational basis (static method).get_matrix([wire_order])Representation of the operator as a matrix in the computational basis.get_parameter_shift(idx)Multiplier and shift for the given parameter, based on its gradient recipe.inv()Inverts the operator.label([decimals, base_label, cache])A customizable string representation of the operator.queue([context])Append the operator to the Operator queue.sparse_matrix([wire_order])Representation of the operator as a sparse matrix in the computational basis.terms()Representation of the operator as a linear combina-		
generator()Generator of an operator that is in single-parameter-form.get_eigvals()Eigenvalues of the operator in the computational basis (static method).get_matrix([wire_order])Representation of the operator as a matrix in the computational basis.get_parameter_shift(idx)Multiplier and shift for the given parameter, based on its gradient recipe.inv()Inverts the operator.label([decimals, base_label, cache])A customizable string representation of the operator.queue([context])Append the operator to the Operator queue.sparse_matrix([wire_order])Representation of the operator as a sparse matrix in the computational basis.terms()Representation of the operator as a linear combina-	expand()	
form.  get_eigvals()  get_matrix([wire_order])  get_parameter_shift(idx)  inv()  linverts the operator.  label([decimals, base_label, cache])  sparse_matrix([wire_order])  sparse_matrix([wire_order])  sparse_matrix([wire_order])  Rejresentation of the operator as a matrix in the computational basis.  Multiplier and shift for the given parameter, based on its gradient recipe.  Inverts the operator.  A customizable string representation of the operator.  Append the operator to the Operator queue.  sparse_matrix([wire_order])  Representation of the operator as a sparse matrix in the computational basis.  terms()  Representation of the operator as a linear combina-		<u> </u>
get_eigvals()Eigenvalues of the operator in the computational basis (static method).get_matrix([wire_order])Representation of the operator as a matrix in the computational basis.get_parameter_shift(idx)Multiplier and shift for the given parameter, based on its gradient recipe.inv()Inverts the operator.label([decimals, base_label, cache])A customizable string representation of the operator.queue([context])Append the operator to the Operator queue.sparse_matrix([wire_order])Representation of the operator as a sparse matrix in the computational basis.terms()Representation of the operator as a linear combina-	generator()	
sis (static method).  get_matrix([wire_order])  Representation of the operator as a matrix in the computational basis.  get_parameter_shift(idx)  Multiplier and shift for the given parameter, based on its gradient recipe.  inv()  Inverts the operator.  label([decimals, base_label, cache])  A customizable string representation of the operator.  queue([context])  Append the operator to the Operator queue.  sparse_matrix([wire_order])  Representation of the operator as a sparse matrix in the computational basis.  terms()  Representation of the operator as a linear combina-		
get_matrix([wire_order])Representation of the operator as a matrix in the computational basis.get_parameter_shift(idx)Multiplier and shift for the given parameter, based on its gradient recipe.inv()Inverts the operator.label([decimals, base_label, cache])A customizable string representation of the operator.queue([context])Append the operator to the Operator queue.sparse_matrix([wire_order])Representation of the operator as a sparse matrix in the computational basis.terms()Representation of the operator as a linear combina-	<pre>get_eigvals()</pre>	
putational basis.  get_parameter_shift(idx)  Multiplier and shift for the given parameter, based on its gradient recipe.  inv()  Inverts the operator.  label([decimals, base_label, cache])  A customizable string representation of the operator.  queue([context])  Append the operator to the Operator queue.  sparse_matrix([wire_order])  Representation of the operator as a sparse matrix in the computational basis.  terms()  Representation of the operator as a linear combina-		
get_parameter_shift(idx)       Multiplier and shift for the given parameter, based on its gradient recipe.         inv()       Inverts the operator.         label([decimals, base_label, cache])       A customizable string representation of the operator.         queue([context])       Append the operator to the Operator queue.         sparse_matrix([wire_order])       Representation of the operator as a sparse matrix in the computational basis.         terms()       Representation of the operator as a linear combina-	<pre>get_matrix([wire_order])</pre>	
its gradient recipe.  inv()  Inverts the operator.  label([decimals, base_label, cache])  A customizable string representation of the operator.  queue([context])  Append the operator to the Operator queue.  sparse_matrix([wire_order])  Representation of the operator as a sparse matrix in the computational basis.  terms()  Representation of the operator as a linear combina-		•
inv()       Inverts the operator.         label([decimals, base_label, cache])       A customizable string representation of the operator.         queue([context])       Append the operator to the Operator queue.         sparse_matrix([wire_order])       Representation of the operator as a sparse matrix in the computational basis.         terms()       Representation of the operator as a linear combina-	<pre>get_parameter_shift(idx)</pre>	
label([decimals, base_label, cache])       A customizable string representation of the operator.         queue([context])       Append the operator to the Operator queue.         sparse_matrix([wire_order])       Representation of the operator as a sparse matrix in the computational basis.         terms()       Representation of the operator as a linear combina-		<u> </u>
queue([context])       Append the operator to the Operator queue.         sparse_matrix([wire_order])       Representation of the operator as a sparse matrix in the computational basis.         terms()       Representation of the operator as a linear combina-		
sparse_matrix([wire_order])Representation of the operator as a sparse matrix in the computational basis.terms()Representation of the operator as a linear combina-		
the computational basis.  terms() Representation of the operator as a linear combina-		
terms() Representation of the operator as a linear combina-	<pre>sparse_matrix([wire_order])</pre>	<u> </u>
tion of other operators.	terms()	
		tion of other operators.

#### adjoint(do\_queue=False)

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.

Parameters do\_queue - Whether to add the adjointed gate to the context queue.

**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.

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

#### See also:

diagonalizing\_gates().

#### **Parameters**

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

Returns list of diagonalizing gates

Return type list[Operator]

## static compute\_eigvals(\*params, \*\*hyperparams)

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

If diagonalizing\_gates are specified and implement a unitary U, 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:

get\_eigvals() and eigvals()

#### **Parameters**

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

**Returns** eigenvalues

#### **Return type** tensor\_like

#### static compute\_matrix(\*params, \*\*hyperparams)

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:

get\_matrix() and 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 matrix representation

Return type scipy.sparse.coo.coo\_matrix

## static compute\_terms(\*params, \*\*hyperparams)

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

$$O = \sum_{i} c_{i} O_{i}$$

### See also:

terms()

#### **Parameters**

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

**Returns** list of coefficients and list of operations

**Return type** tuple[list[tensor\_like or float], list[Operation]]

#### decomposition()

Representation of the operator as a product of other operators.

$$Q = Q_1 Q_2 \dots Q_n$$

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.

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

#### expand()

Returns a tape that has recorded 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.

#### get\_eigvals()

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

If diagonalizing\_gates are specified and implement a unitary U, 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

#### get\_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

## get\_parameter\_shift(idx)

Multiplier and shift for the given parameter, based on its gradient recipe.

**Parameters** idx (int) – parameter index within the operation

Returns list of multiplier, coefficient, shift for each term in the gradient recipe

**Return type** list[[float, float, float]]

Note that the default value for shift is None, which is replaced by the default shift  $\pi/2$ .

#### inv()

Inverts the operator.

This method concatenates a string to the name of the operation, to indicate that the inverse will be used for computations.

Any subsequent call of this method will toggle between the original operation and the inverse of the operation.

**Returns** operation to be inverted

### Return type Operator

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 caries information between label calls in the same drawing

**Returns** label to use in drawings

Return type str

#### **Example:**

```
>>> op = qml.RX(1.23456, wires=0)
>>> op.label()
"RX"
>>> op.label(decimals=2)
"RX\n(1.23)"
>>> op.label(base_label="my_label")
"my_label"
>>> op.label(decimals=2, base_label="my_label")
"my_label\n(1.23)"
>>> op.inv()
>>> op.label()
"RX1"
```

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.

(continues on next page)

(continued from previous page)

[0., 1., 0., 0.],
[0., 0., 1., 0.],
[0., 0., 0., 1.]], requires\_grad=True)]

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

Append the operator to the Operator queue.

## 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.

**Note:** The wire\_order argument is currently not implemented, and using it will raise an error.

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 matrix representation

Return type scipy.sparse.coo.coo\_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.

## See also:

compute\_terms()

**Returns** list of coefficients  $c_i$  and list of operations  $O_i$ 

**Return type** tuple[list[tensor\_like or float], list[Operation]]

## **Depolarize**

class Depolarize(\*params, wires=None, do\_queue=True, id=None)

Bases: pennylane.operation.Operation

Cirq depolarize operation.

See the Cirq docs for further details.

base_name	If inverse is requested, this is the name of the original
	operator to be inverted.
basis	The target operation for controlled gates.
	continues on next page

Table 6 – continued from previous page

control_wires	Control wires of the operator.
eigvals	Eigenvalues of an instantiated operator.
grad_method	Digentation of all installation operator.
9_ 40	
grad_recipe	Gradient recipe for the parameter-shift method.
has_matrix	
hash	Integer hash that uniquely represents the operator.
hyperparameters	Dictionary of non-trainable variables that this opera-
	tion depends on.
id	Custom string to label a specific operator instance.
inverse	Boolean determining if the inverse of the operation
	was requested.
matrix	Matrix representation of an instantiated operator in
	the computational basis.
name	Name of the operator.
num_params	
num_wires	
par_domain	
parameter_frequencies	Returns the frequencies for each operator parame-
	ter with respect to an expectation value of the form
	$\langle \psi   U(\mathbf{p})^\dagger \hat{OU}(\mathbf{p})   \psi \rangle$ .
parameters	Trainable parameters that the operator depends on.
single_qubit_rot_angles	The parameters required to implement a single-qubit
	gate as an equivalent Rot gate, up to a global phase.
wires	Wires that the operator acts on.

## base\_name

If inverse is requested, this is the name of the original operator to be inverted.

## basis = None

The target operation for controlled gates. target operation. If not None, should take a value of "X", "Y", or "7"

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

Type str 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

## eigvals

Eigenvalues of an instantiated operator. Note that the eigenvalues are not guaranteed to be in any particular order.

**Warning:** The eigvals property is deprecated and will be removed in an upcoming release. Please use qml.eigvals instead.

## **Example:**

```
>>> U = qml.RZ(0.5, wires=1)
>>> U.eigvals
>>> array([0.96891242-0.24740396j, 0.96891242+0.24740396j])
```

Returns eigvals representation

Return type array

grad\_method = None

#### 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\_matrix = False

#### 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.

#### inverse

Boolean determining if the inverse of the operation was requested.

## matrix

Matrix representation of an instantiated operator in the computational basis.

**Warning:** The matrix property is deprecated and will be removed in an upcoming release. Please use qml.matrix instead.

## **Example:**

**Returns** matrix representation

Return type array

#### name

Name of the operator.

```
num_params = 1
num_wires = 1
par_domain = 'R'
```

# 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.

# 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]

#### wires

Wires that the operator acts on.

Returns wires

Return type Wires

adjoint([do_queue])	Create an operation that is the adjoint of this one.
<pre>compute_decomposition(*params[, wires])</pre>	Representation of the operator as a product of other
	operators (static method).
<pre>compute_diagonalizing_gates(*params, wires,</pre>	Sequence of gates that diagonalize the operator in the
)	computational basis (static method).
<pre>compute_eigvals(*params, **hyperparams)</pre>	Eigenvalues of the operator in the computational ba-
	sis (static method).
<pre>compute_matrix(*params, **hyperparams)</pre>	Representation of the operator as a canonical matrix
	in the computational basis (static method).
compute_sparse_matrix(*params, **hyper-	Representation of the operator as a sparse matrix in
params)	the computational basis (static method).
compute_terms(*params, **hyperparams)	Representation of the operator as a linear combina-
	tion of other operators (static method).
<pre>decomposition()</pre>	Representation of the operator as a product of other
	operators.
diagonalizing_gates()	Sequence of gates that diagonalize the operator in the
	computational basis.
expand()	Returns a tape that has recorded the decomposition
	of the operator.
generator()	Generator of an operator that is in single-parameter-
	form.
<pre>get_eigvals()</pre>	Eigenvalues of the operator in the computational ba-
	sis (static method).
<pre>get_matrix([wire_order])</pre>	Representation of the operator as a matrix in the com-
	putational basis.
<pre>get_parameter_shift(idx)</pre>	Multiplier and shift for the given parameter, based on
	its gradient recipe.
inv()	Inverts the operator.
label([decimals, base_label, cache])	A customizable string representation of the operator.
queue([context])	Append the operator to the Operator queue.
sparse_matrix([wire_order])	Representation of the operator as a sparse matrix in
	the computational basis.
terms()	Representation of the operator as a linear combina-
	tion of other operators.

# adjoint(do\_queue=False)

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.

**Parameters do\_queue** – Whether to add the adjointed gate to the context queue.

**Returns** The adjointed operation.

static compute\_decomposition(\*params, wires=None, \*\*hyperparameters)

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

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

36 Chapter 2. Tutorials

**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.

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, 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:

get\_eigvals() and 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(\*params, \*\*hyperparams)

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:

get\_matrix() and 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
- $\bullet$  hyperparams (dict) non-trainable hyperparameters of the operator, as stored in the hyperparameters attribute

**Returns** matrix representation

Return type scipy.sparse.coo.coo\_matrix

#### static compute\_terms(\*params, \*\*hyperparams)

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

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

38 Chapter 2. Tutorials

#### See also:

terms()

#### **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** list of coefficients and list of operations

**Return type** tuple[list[tensor\_like or float], list[Operation]]

## 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.

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

# expand()

Returns a tape that has recorded 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.

## get\_eigvals()

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

If diagonalizing\_gates are specified and implement a unitary U, 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

```
get_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

# get\_parameter\_shift(idx)

Multiplier and shift for the given parameter, based on its gradient recipe.

**Parameters** idx (int) – parameter index within the operation

Returns list of multiplier, coefficient, shift for each term in the gradient recipe

**Return type** list[[float, float, float]]

Note that the default value for shift is None, which is replaced by the default shift  $\pi/2$ .

#### inv()

Inverts the operator.

This method concatenates a string to the name of the operation, to indicate that the inverse will be used for computations.

Any subsequent call of this method will toggle between the original operation and the inverse of the operation.

**Returns** operation to be inverted

#### Return type Operator

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 caries information between label calls in the same drawing

**Returns** label to use in drawings

Return type str

### **Example:**

```
>>> op = qml.RX(1.23456, wires=0)
>>> op.label()
"RX"
>>> op.label(decimals=2)
"RX\n(1.23)"
>>> op.label(base_label="my_label")
"my_label"
>>> op.label(decimals=2, base_label="my_label")
"my_label\n(1.23)"
>>> op.inv()
>>> op.label()
"RX1"
```

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.

(continues on next page)

(continued from previous page)

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

Append the operator to the Operator queue.

#### 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.

**Note:** The wire order argument is currently not implemented, and using it will raise an error.

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** matrix representation

**Return type** scipy.sparse.coo.coo\_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.

#### See also:

```
compute_terms()
```

**Returns** list of coefficients  $c_i$  and list of operations  $O_i$ 

**Return type** tuple[list[tensor\_like or float], list[Operation]]

# **MixedStateSimulatorDevice**

class MixedStateSimulatorDevice(wires, shots=None, qubits=None)

Bases: pennylane\_cirq.simulator\_device.SimulatorDevice

Cirq mixed-state simulator 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']).
- **shots** (*int*) Number of circuit evaluations/random samples used to estimate expectation values of observables. Shots need to >= 1. If None, expectation values are calculated analytically.
- **qubits** (*List[cirq.Qubit]*) A list of Cirq qubits that are used as wires. The wire number corresponds to the index in the list. By default, an array of cirq.LineQubit instances is created.

analytic	Whether shots is None or not.
author	
circuit_hash	The hash of the circuit upon the last execution.
name	
num_executions	Number of times this device is executed by the eval- uation of QNodes running on this device
obs_queue	The observables to be measured and returned.
observables	set() -> new empty set object set(iterable) -> new set object
op_queue	The operation queue to be applied.
operations	Get the supported set of operations.
parameters	Mapping from free parameter index to the list of
	Operations in the device queue that depend on it.
pennylane_requires	
short_name	
shot_vector	Returns the shot vector, a sparse representation of the shot sequence used by the device when evaluating ONodes.
shot_vector shots	shot sequence used by the device when evaluating QNodes.  Number of circuit evaluations/random samples used
	shot sequence used by the device when evaluating QNodes.
shots	shot sequence used by the device when evaluating QNodes.  Number of circuit evaluations/random samples used to estimate expectation values of observables  Returns the density matrix of the circuit prior to mea-
shots	shot sequence used by the device when evaluating QNodes.  Number of circuit evaluations/random samples used to estimate expectation values of observables  Returns the density matrix of the circuit prior to measurement.
shots state stopping_condition	shot sequence used by the device when evaluating QNodes.  Number of circuit evaluations/random samples used to estimate expectation values of observables  Returns the density matrix of the circuit prior to measurement.

#### 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.

## name = 'Cirq Mixed-State Simulator device 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

#### 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 = '>=0.17.0'
```

```
short_name = 'cirq.mixedsimulator'
```

# shot\_vector

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

#### Example

```
>>> dev = qml.device("default.qubit", wires=2, shots=[3, 1, 2, 2, 2, 2, 6, 1, 1, 

5, 12, 10, 10])
>>> dev.shots
57
```

(continues on next page)

(continued from previous page)

```
>>> dev.shot_vector
[ShotTuple(shots=3, copies=1),
   ShotTuple(shots=1, copies=1),
   ShotTuple(shots=2, copies=4),
   ShotTuple(shots=6, copies=1),
   ShotTuple(shots=1, copies=2),
   ShotTuple(shots=5, copies=1),
   ShotTuple(shots=12, copies=1),
   ShotTuple(shots=10, copies=2)]
```

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

**Type** list[ShotTuple[int, int]]

#### shots

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

#### state

Returns the density matrix of the circuit prior to measurement.

**Note:** The state includes possible basis rotations for non-diagonal observables. Note that this behaviour differs from PennyLane's default.qubit plugin.

## stopping\_condition

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

Type BooleanFn

version = '0.28.0'

#### wire\_map

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

## wires

All wires that can be addressed on this device

access_state([wires])	Check that the device has access to an internal state
	and return it if available.
active_wires(operators)	Returns the wires acted on by a set of operators.
<pre>adjoint_jacobian(tape[, starting_state,])</pre>	Implements the adjoint method outlined in Jones and
	Gacon to differentiate an input tape.
analytic_probability([wires])	Return the (marginal) probability of each computa-
	tional basis state from the last run of the device.
apply(operations, **kwargs)	Apply quantum operations, rotate the circuit into
	the measurement basis, and compile and execute the
	quantum circuit.
batch_execute(circuits)	Execute a batch of quantum circuits on the device.
batch_transform(circuit)	Apply a differentiable batch transform for prepro-
	cessing a circuit prior to execution.
capabilities()	Get the capabilities of this device class.
<pre>check_validity(queue, observables)</pre>	Checks whether the operations and observables in
	queue are all supported by the device.
	continues on next page

Table 9 – continued from previous page

	d from previous page
custom_expand(fn)	Register a custom expansion function for the device.
<pre>default_expand_fn(circuit[, max_expansion])</pre>	Method for expanding or decomposing an input circuit.
define_wire_map(wires)	Create the map from user-provided wire labels to the
define_wire_map(wires)	wire labels used by the device.
di	
density_matrix(wires)	Returns the reduced density matrix prior to measure-
	ment.
<pre>estimate_probability([wires, shot_range,])</pre>	Return the estimated probability of each computa-
	tional basis state using the generated samples.
execute(circuit, **kwargs)	Execute a queue of quantum operations on the device and then measure the given observables.
<pre>execute_and_gradients(circuits[, method])</pre>	Execute a batch of quantum circuits on the device,
	and return both the results and the gradients.
execution_context()	The device execution context used during calls to
	execute().
<pre>expand_fn(circuit[, max_expansion])</pre>	Method for expanding or decomposing an input cir-
- · · · · - · · · · · · · · · · · · · ·	cuit.
<pre>expval(observable[, shot_range, bin_size])</pre>	Returns the expectation value of observable on spec-
	ified wires.
<pre>generate_basis_states(num_wires[, dtype])</pre>	Generates basis states in binary representation ac-
5 · · · · · · · · · · · · · · · · · · ·	cording to the number of wires specified.
<pre>generate_samples()</pre>	Returns the computational basis samples generated
<i>y</i>	for all wires.
<pre>gradients(circuits[, method])</pre>	Return the gradients of a batch of quantum circuits
grade circle (circuitati, methodi)	on the device.
map_wires(wires)	Map the wire labels of wires using this device's wire
map_wii es(whes)	map.
marginal_prob(prob[, wires])	Return the marginal probability of the computational
marginar_pros(proof, whest)	basis states by summing the probabilities on the non-
	specified wires.
order_wires(subset_wires)	Given some subset of device wires return a Wires ob-
oraci_wires(sabset_wires)	ject with the same wires; sorted according to the de-
	vice wire map.
post_apply()	Called during execute() after the individual opera-
post_appiy()	tions have been executed.
post_measure()	Called during execute() after the individual ob-
post_measure()	servables have been measured.
pre_apply()	Called during execute() before the individual op-
prc_appry()	erations are executed.
<pre>pre_measure()</pre>	Called during <i>execute()</i> before the individual ob-
pre_measure()	servables are measured.
<pre>probability([wires, shot_range, bin_size])</pre>	Return either the analytic probability or estimated
probability ([wires, shot_range, oni_size])	probability of each computational basis state.
reset()	Reset the backend state.
	Return a sample of an observable.
<pre>sample(observable[, shot_range, bin_size]) sample_basis_states(number_of_states,)</pre>	
sample_vasis_states(minuel_oi_states,)	Sample from the computational basis states based on the state probability
states to himawy(somples num wines dt	the state probability.
<pre>states_to_binary(samples, num_wires[, dtype])</pre>	Convert basis states from base 10 to binary represen-
statistics(shear-last sheat are 12 at 12	tation.
statistics(observables[, shot_range, bin_size])	Process measurement results from circuit execution
	and return statistics.
	continues on next page

46 Chapter 2. Tutorials

Table 9 – continued from previous page

<pre>supports_observable(observable)</pre>	Checks if an observable is supported by this device.
	Raises a ValueError,
<pre>supports_operation(operation)</pre>	Checks if an operation is supported by this device.
to_paulistring(observable)	Convert an observable to a cirq.PauliString
<pre>var(observable[, shot_range, bin_size])</pre>	Returns the variance of observable on specified
	wires.

#### 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, 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 inverse (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 Hamiltonian observables.

**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

**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, \*\*kwargs)

Apply quantum operations, rotate the circuit into the measurement basis, and compile and execute the quantum circuit.

This method receives a list of quantum operations queued by the QNode, and should be responsible for:

- · Constructing the quantum program
- (Optional) Rotating the quantum circuit using the rotation operations provided. This diagonalizes the circuit so that arbitrary observables can be measured in the computational basis.
- Compile the circuit
- Execute the quantum circuit

Both arguments are provided as lists of PennyLane Operation instances. Useful properties include name, wires, and parameters, and inverse:

**Parameters operations** (list[Operation]) – operations to apply to the device

#### **Keyword Arguments**

- **rotations** (*list[Operation]*) operations that rotate the circuit pre-measurement into the eigenbasis of the observables.
- hash (int) the hash value of the circuit constructed by CircuitGraph.hash

#### 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[tapes.QuantumTape]) – circuits to execute on the device
```

**Returns** list of measured value(s)

**Return type** list[array[float]]

# batch\_transform(circuit)

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()

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_inverse_operations=False,
        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. Includes checks for inverse operations.

**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

#### custom\_expand(fn)

Register a custom expansion function for the device.

#### Example

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

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

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

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

#### default\_expand\_fn(circuit, max expansion=10)

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

By default, this method expands the tape if:

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

## **Parameters**

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

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

Return type QuantumTape

# define\_wire\_map(wires)

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

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

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

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

**Returns** dictionary specifying the wire map

Return type OrderedDict

## Example

## density\_matrix(wires)

Returns the reduced density matrix prior to measurement.

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

## 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)
```

Execute 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 the 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** (*CircuitGraph*) – 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 Device.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 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]

# 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)
```

Return a sample of an observable.

The number of samples is determined by the value of Device.shots, which can be directly modified.

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 sampling

**Returns** samples in an array of dimension (shots,)

**Return type** array[float]

# 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]

## 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(observables, 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**

- **observables** (*List[Observable]*) the observables to be measured
- **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 iff supplied operation is supported

Return type bool

#### to\_paulistring(observable)

Convert an observable to a cirq.PauliString

var(observable, shot\_range=None, bin\_size=None)

Returns the variance of observable on specified wires.

Note: all arguments support \_lists\_, which indicate a tensor product of observables.

#### **Parameters**

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

Raises NotImplementedError – if the device does not support variance computation

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

Return type float

# **PasqalDevice**

class PasqalDevice(wires, control\_radius, shots=None, qubits=None)

Bases: pennylane\_cirq.simulator\_device.SimulatorDevice

Cirq Pasqal device for PennyLane.

#### **Parameters**

- wires (int) the number of wires to initialize the device with
- **control\_radius** (*float*) The maximum distance between qubits for a controlled gate. Distance is measured in units of the ThreeDGridQubit indices.
- **shots** (*int*) Number of circuit evaluations/random samples used to estimate expectation values of observables. Shots need to be >= 1. If None, expectation values are calculated analytically.
- **qubits** (List[cirq.ThreeDGridQubit]) A list of Cirq ThreeDGridQubits that are used as wires. If not specified, the ThreeDGridQubits are put in a linear arrangement along the first coordinate axis, separated by a distance of control\_radius / 2. i.e., (0,0,0), (control\_radius/2,0,0), (control\_radius,0,0), etc.

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

# Table 10 – continued from previous page

wires

All wires that can be addressed on this device

# 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.

# name = 'Cirq Pasqal device 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

## 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 = '>=0.17.0'
```

# short\_name = 'cirq.pasqal' shot\_vector

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

# Example

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

**Type** list[ShotTuple[int, int]]

#### 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:** The state includes possible basis rotations for non-diagonal observables. Note that this behaviour differs from PennyLane's default.qubit plugin.

#### stopping\_condition

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

Type BooleanFn

version = '0.28.0'

#### wire\_map

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

#### wires

All wires that can be addressed on this device

access_state([wires])	Check that the device has access to an internal state
	and return it if available.
active_wires(operators)	Returns the wires acted on by a set of operators.
adjoint_jacobian(tape[, starting_state,])	Implements the adjoint method outlined in Jones and
	Gacon to differentiate an input tape.
analytic_probability([wires])	Return the (marginal) probability of each computa-
	tional basis state from the last run of the device.
apply(operations, **kwargs)	Apply quantum operations, rotate the circuit into
	the measurement basis, and compile and execute the
	quantum circuit.
batch_execute(circuits)	Execute a batch of quantum circuits on the device.
	continues on next page

Table 11 – continued from previous page

	d from previous page
batch_transform(circuit)	Apply a differentiable batch transform for prepro-
	cessing a circuit prior to execution.
capabilities()	Get the capabilities of this device class.
<pre>check_validity(queue, observables)</pre>	Checks whether the operations and observables in
	queue are all supported by the device.
custom_expand(fn)	Register a custom expansion function for the device.
<pre>default_expand_fn(circuit[, max_expansion])</pre>	Method for expanding or decomposing an input cir-
· · · · · · · · · · · · · · · · · · ·	cuit.
<pre>define_wire_map(wires)</pre>	Create the map from user-provided wire labels to the
	wire labels used by the device.
<pre>density_matrix(wires)</pre>	Returns the reduced density matrix prior to measure-
	ment.
<pre>estimate_probability([wires, shot_range,])</pre>	Return the estimated probability of each computa-
	tional basis state using the generated samples.
execute(circuit, **kwargs)	Execute a queue of quantum operations on the device
	and then measure the given observables.
<pre>execute_and_gradients(circuits[, method])</pre>	Execute a batch of quantum circuits on the device,
	and return both the results and the gradients.
execution_context()	The device execution context used during calls to
	execute().
<pre>expand_fn(circuit[, max_expansion])</pre>	Method for expanding or decomposing an input cir-
	cuit.
<pre>expval(observable[, shot_range, bin_size])</pre>	Returns the expectation value of observable on spec-
	ified wires.
<pre>generate_basis_states(num_wires[, dtype])</pre>	Generates basis states in binary representation ac-
	cording to the number of wires specified.
<pre>generate_samples()</pre>	Returns the computational basis samples generated
g-11-02 u t-0_5 ump 2-05 ()	for all wires.
<pre>gradients(circuits[, method])</pre>	Return the gradients of a batch of quantum circuits
gradients (encodist, memory)	on the device.
map_wires(wires)	Map the wire labels of wires using this device's wire
map_wii es(wiies)	map.
marginal_prob(prob[, wires])	Return the marginal probability of the computational
marginar_pros(proof, whest)	basis states by summing the probabilities on the non-
	specified wires.
order_wires(subset_wires)	Given some subset of device wires return a Wires ob-
or acr_wires(subset_wires)	ject with the same wires; sorted according to the de-
	vice wire map.
post_apply()	Called during execute() after the individual opera-
booc-abbra()	tions have been executed.
nost massira()	Called during execute() after the individual ob-
<pre>post_measure()</pre>	
nno anniu()	servables have been measured.
<pre>pre_apply()</pre>	Called during execute() before the individual op-
nno mozguno()	erations are executed.
<pre>pre_measure()</pre>	Called during <i>execute()</i> before the individual observables are measured.
and held to definite a had a second to the term	servables are measured.
<pre>probability([wires, shot_range, bin_size])</pre>	Return either the analytic probability or estimated
	probability of each computational basis state.
reset()	Reset the backend state.
<pre>sample(observable[, shot_range, bin_size])</pre>	Return a sample of an observable.
	continues on next page

continues on next page

Table 11 – continued from previous page

	1 1 5
<pre>sample_basis_states(number_of_states,)</pre>	Sample from the computational basis states based on
	the state probability.
<pre>states_to_binary(samples, num_wires[, dtype])</pre>	Convert basis states from base 10 to binary represen-
	tation.
<pre>statistics(observables[, shot_range, bin_size])</pre>	Process measurement results from circuit execution
	and return statistics.
<pre>supports_observable(observable)</pre>	Checks if an observable is supported by this device.
	Raises a ValueError,
<pre>supports_operation(operation)</pre>	Checks if an operation is supported by this device.
to_paulistring(observable)	Convert an observable to a cirq.PauliString
<pre>var(observable[, shot_range, bin_size])</pre>	Returns the variance of observable on specified
	wires.

#### 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, 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 inverse (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 Hamiltonian observables.

 $\begin{picture}(Colors Tape) - Circuit that the function takes the gradient of the colors of the c$ 

## **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

**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, \*\*kwargs)

Apply quantum operations, rotate the circuit into the measurement basis, and compile and execute the quantum circuit.

This method receives a list of quantum operations queued by the QNode, and should be responsible for:

- Constructing the quantum program
- (Optional) Rotating the quantum circuit using the rotation operations provided. This diagonalizes the circuit so that arbitrary observables can be measured in the computational basis.
- Compile the circuit
- Execute the quantum circuit

Both arguments are provided as lists of PennyLane Operation instances. Useful properties include name, wires, and parameters, and inverse:

**Parameters operations** (list[Operation]) – operations to apply to the device

#### **Keyword Arguments**

- **rotations** (*list[Operation]*) operations that rotate the circuit pre-measurement into the eigenbasis of the observables.
- hash (int) the hash value of the circuit constructed by CircuitGraph.hash

#### 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[tapes.QuantumTape]) – circuits to execute on the device

**Returns** list of measured value(s)

**Return type** list[array[float]]

#### batch\_transform(circuit)

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()

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_inverse_operations=False,
        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. Includes checks for inverse operations.

## **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

## custom\_expand(fn)

Register a custom expansion function for the device.

#### **Example**

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

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

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

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

#### default\_expand\_fn(circuit, max\_expansion=10)

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

By default, this method expands the tape if:

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

# **Parameters**

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

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

Return type QuantumTape

#### define\_wire\_map(wires)

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

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

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

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

Returns dictionary specifying the wire map

Return type OrderedDict

#### Example

## density\_matrix(wires)

Returns the reduced density matrix prior to measurement.

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

## 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)

Execute 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 the 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** (CircuitGraph) – 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 Device.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]

## 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)
```

Return a sample of an observable.

The number of samples is determined by the value of Device.shots, which can be directly modified.

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 sampling

Returns samples in an array of dimension (shots,)

**Return type** array[float]

## 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]

# 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(observables, 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**

- **observables** (*List[Observable]*) the observables to be measured
- **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 iff supplied operation is supported

Return type bool

#### to\_paulistring(observable)

Convert an observable to a cirq.PauliString

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**

70

- **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)

 $\textbf{Raises} \ \ \textbf{NotImplementedError} - if \ the \ device \ does \ not \ support \ variance \ computation$ 

Returns variance  $var(A) = \psi A^2 \psi - \psi A \psi^2$ Return type float

# **PhaseDamp**

class PhaseDamp(\*params, wires=None, do\_queue=True, id=None)

Bases: pennylane.operation.Operation

Cirq phase\_damp operation.

See the Cirq docs for further details.

base_name	If inverse is requested, this is the name of the original
	operator to be inverted.
basis	The target operation for controlled gates.
control_wires	Control wires of the operator.
eigvals	Eigenvalues of an instantiated operator.
grad_method	
grad_recipe	Gradient recipe for the parameter-shift method.
has_matrix	
hash	Integer hash that uniquely represents the operator.
hyperparameters	Dictionary of non-trainable variables that this opera-
	tion depends on.
id	Custom string to label a specific operator instance.
inverse	Boolean determining if the inverse of the operation
	was requested.
matrix	Matrix representation of an instantiated operator in
	the computational basis.
name	Name of the operator.
num_params	
num_wires	
par_domain	
parameter_frequencies	Returns the frequencies for each operator parame-
	ter with respect to an expectation value of the form $\langle \psi   U(\mathbf{p})^{\dagger} \hat{O} U(\mathbf{p})   \psi \rangle$ .
parameters	Trainable parameters that the operator depends on.
single_qubit_rot_angles	The parameters required to implement a single-qubit
	gate as an equivalent Rot gate, up to a global phase.
wires	Wires that the operator acts on.

# base\_name

If inverse is requested, this is the name of the original operator to be inverted.

# basis = None

The target operation for controlled gates. 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

#### 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

# eigvals

Eigenvalues of an instantiated operator. Note that the eigenvalues are not guaranteed to be in any particular order.

**Warning:** The eigvals property is deprecated and will be removed in an upcoming release. Please use qml.eigvals instead.

# **Example:**

```
>>> U = qml.RZ(0.5, wires=1)
>>> U.eigvals
>>> array([0.96891242-0.24740396j, 0.96891242+0.24740396j])
```

**Returns** eigvals representation

Return type array

grad\_method = None

## 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\_matrix = False

#### hash

Integer hash that uniquely represents the operator.

Type int

# hyperparameters

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

Type dict

id

72

Custom string to label a specific operator instance.

# inverse

Boolean determining if the inverse of the operation was requested.

#### matrix

Matrix representation of an instantiated operator in the computational basis.

**Warning:** The matrix property is deprecated and will be removed in an upcoming release. Please use qml.matrix instead.

## **Example:**

**Returns** matrix representation

Return type array

#### name

Name of the operator.

```
num_params = 1
num_wires = 1
par_domain = 'R'
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,)
```

73

For more details on this relationship, see eigvals\_to\_frequencies().

# parameters

Trainable parameters that the operator depends on.

# 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]

## wires

Wires that the operator acts on.

Returns wires

Return type Wires

adjoint([do_queue])	Create an operation that is the adjoint of this one.
<pre>compute_decomposition(*params[, wires])</pre>	Representation of the operator as a product of other
	operators (static method).
<pre>compute_diagonalizing_gates(*params, wires,</pre>	Sequence of gates that diagonalize the operator in the
)	computational basis (static method).
<pre>compute_eigvals(*params, **hyperparams)</pre>	Eigenvalues of the operator in the computational ba-
	sis (static method).
<pre>compute_matrix(*params, **hyperparams)</pre>	Representation of the operator as a canonical matrix
	in the computational basis (static method).
<pre>compute_sparse_matrix(*params, **hyper-</pre>	Representation of the operator as a sparse matrix in
params)	the computational basis (static method).
<pre>compute_terms(*params, **hyperparams)</pre>	Representation of the operator as a linear combina-
	tion of other operators (static method).
decomposition()	Representation of the operator as a product of other
	operators.
diagonalizing_gates()	Sequence of gates that diagonalize the operator in the
	computational basis.
expand()	Returns a tape that has recorded the decomposition
	of the operator.
generator()	Generator of an operator that is in single-parameter-
	form.
<pre>get_eigvals()</pre>	Eigenvalues of the operator in the computational ba-
	sis (static method).
<pre>get_matrix([wire_order])</pre>	Representation of the operator as a matrix in the com-
	putational basis.
<pre>get_parameter_shift(idx)</pre>	Multiplier and shift for the given parameter, based on
	its gradient recipe.
inv()	Inverts the operator.
label([decimals, base_label, cache])	A customizable string representation of the operator.
queue([context])	Append the operator to the Operator queue.
sparse_matrix([wire_order])	Representation of the operator as a sparse matrix in
	the computational basis.
terms()	Representation of the operator as a linear combina-

# adjoint(do\_queue=False)

Create an operation that is the adjoint of this one.

74 Chapter 2. Tutorials

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

**Parameters do\_queue** – Whether to add the adjointed gate to the context queue.

**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.

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, 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:

get\_eigvals() and eigvals()

#### **Parameters**

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

Returns eigenvalues

Return type tensor\_like

# static compute\_matrix(\*params, \*\*hyperparams)

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:

```
get_matrix() and 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:

76

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 matrix representation

Return type scipy.sparse.coo.coo\_matrix

#### static compute\_terms(\*params, \*\*hyperparams)

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

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

#### See also:

terms()

#### **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** list of coefficients and list of operations

**Return type** tuple[list[tensor\_like or float], list[Operation]]

## 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.

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

#### expand()

Returns a tape that has recorded 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.

#### get\_eigvals()

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

If  $diagonalizing\_gates$  are specified and implement a unitary U, 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

## get\_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

## get\_parameter\_shift(idx)

Multiplier and shift for the given parameter, based on its gradient recipe.

**Parameters** idx (int) – parameter index within the operation

**Returns** list of multiplier, coefficient, shift for each term in the gradient recipe

**Return type** list[[float, float, float]]

Note that the default value for shift is None, which is replaced by the default shift  $\pi/2$ .

## inv()

Inverts the operator.

This method concatenates a string to the name of the operation, to indicate that the inverse will be used for computations.

Any subsequent call of this method will toggle between the original operation and the inverse of the operation.

**Returns** operation to be inverted

Return type Operator

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

A customizable string representation of the operator.

#### **Parameters**

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

**Returns** label to use in drawings

Return type str

# **Example:**

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

(continues on next page)

(continued from previous page)

```
>>> op.inv()
>>> op.label()
"RX1"
```

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)]
```

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

Append the operator to the Operator queue.

```
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.

**Note:** The wire\_order argument is currently not implemented, and using it will raise an error.

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** matrix representation

Return type scipy.sparse.coo.coo\_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.

80 Chapter 2. Tutorials

# See also:

compute\_terms()

**Returns** list of coefficients  $c_i$  and list of operations  $O_i$ 

**Return type** tuple[list[tensor\_like or float], list[Operation]]

# **PhaseFlip**

class PhaseFlip(\*params, wires=None, do\_queue=True, id=None)

Bases: pennylane.operation.Operation

Cirq phase\_flip operation.

See the Cirq docs for further details.

base_name	If inverse is requested, this is the name of the original
	operator to be inverted.
basis	The target operation for controlled gates.
control_wires	Control wires of the operator.
eigvals	Eigenvalues of an instantiated operator.
grad_method	
grad_recipe	Gradient recipe for the parameter-shift method.
has_matrix	
hash	Integer hash that uniquely represents the operator.
hyperparameters	Dictionary of non-trainable variables that this opera-
	tion depends on.
id	Custom string to label a specific operator instance.
inverse	Boolean determining if the inverse of the operation
	was requested.
matrix	Matrix representation of an instantiated operator in
	the computational basis.
name	Name of the operator.
num_params	
num_wires	
par_domain	
parameter_frequencies	Returns the frequencies for each operator parame-
	ter with respect to an expectation value of the form
	$\langle \psi   U(\mathbf{p})^{\dagger} \hat{O} U(\mathbf{p})   \psi \rangle$ .
parameters	Trainable parameters that the operator depends on.
single_qubit_rot_angles	The parameters required to implement a single-qubit
	gate as an equivalent Rot gate, up to a global phase.
wires	Wires that the operator acts on.

## base\_name

If inverse is requested, this is the name of the original operator to be inverted.

basis = None

The target operation for controlled gates. 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

# 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

#### eigvals

Eigenvalues of an instantiated operator. Note that the eigenvalues are not guaranteed to be in any particular order

**Warning:** The eigvals property is deprecated and will be removed in an upcoming release. Please use qml.eigvals instead.

# **Example:**

```
>>> U = qml.RZ(0.5, wires=1)
>>> U.eigvals
>>> array([0.96891242-0.24740396j, 0.96891242+0.24740396j])
```

**Returns** eigvals representation

Return type array

grad\_method = None

# 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\_matrix = False

## hash

82

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.

#### inverse

Boolean determining if the inverse of the operation was requested.

#### matrix

Matrix representation of an instantiated operator in the computational basis.

**Warning:** The matrix property is deprecated and will be removed in an upcoming release. Please use qml.matrix instead.

# **Example:**

**Returns** matrix representation

Return type array

#### name

Name of the operator.

```
num_params = 1
num_wires = 1
par_domain = 'R'
```

# 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")
```

(continues on next page)

(continued from previous page)

```
>>> 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.

# 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]

#### wires

Wires that the operator acts on.

Returns wires

Return type Wires

<pre>adjoint([do_queue])</pre>	Create an operation that is the adjoint of this one.
<pre>compute_decomposition(*params[, wires])</pre>	Representation of the operator as a product of other
	operators (static method).
<pre>compute_diagonalizing_gates(*params, wires,</pre>	Sequence of gates that diagonalize the operator in the
)	computational basis (static method).
<pre>compute_eigvals(*params, **hyperparams)</pre>	Eigenvalues of the operator in the computational ba-
	sis (static method).
<pre>compute_matrix(*params, **hyperparams)</pre>	Representation of the operator as a canonical matrix
	in the computational basis (static method).
<pre>compute_sparse_matrix(*params, **hyper-</pre>	Representation of the operator as a sparse matrix in
params)	the computational basis (static method).
<pre>compute_terms(*params, **hyperparams)</pre>	Representation of the operator as a linear combina-
	tion of other operators (static method).
<pre>decomposition()</pre>	Representation of the operator as a product of other
	operators.
diagonalizing_gates()	Sequence of gates that diagonalize the operator in the
	computational basis.
expand()	Returns a tape that has recorded the decomposition
	of the operator.
<pre>generator()</pre>	Generator of an operator that is in single-parameter-
	form.
<pre>get_eigvals()</pre>	Eigenvalues of the operator in the computational ba-
	sis (static method).
<pre>get_matrix([wire_order])</pre>	Representation of the operator as a matrix in the com-
	putational basis.
<pre>get_parameter_shift(idx)</pre>	Multiplier and shift for the given parameter, based on
	its gradient recipe.
inv()	Inverts the operator.
label([decimals, base_label, cache])	A customizable string representation of the operator.
queue([context])	Append the operator to the Operator queue.
	continues on next page

continues on next page

Table 15 – continued from previous page

<pre>sparse_matrix([wire_order])</pre>	Representation of the operator as a sparse matrix in
	the computational basis.
terms()	Representation of the operator as a linear combina-
	tion of other operators.

# adjoint(do\_queue=False)

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.

**Parameters do\_queue** – Whether to add the adjointed gate to the context queue.

**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.

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, 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:

get\_eigvals() and 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(\*params, \*\*hyperparams)

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:

```
get_matrix() and matrix()
```

#### **Parameters**

- params (list) trainable parameters of the operator, as stored in the parameters attribute
- $\bullet$  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:

86

```
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 matrix representation

Return type scipy.sparse.coo.coo\_matrix

# static compute\_terms(\*params, \*\*hyperparams)

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

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

#### See also:

terms()

#### **Parameters**

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

**Returns** list of coefficients and list of operations

**Return type** tuple[list[tensor\_like or float], list[Operation]]

# 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.

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

#### expand()

Returns a tape that has recorded 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.

## get\_eigvals()

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

If  $diagonalizing\_gates$  are specified and implement a unitary U, 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

# get\_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

## get\_parameter\_shift(idx)

Multiplier and shift for the given parameter, based on its gradient recipe.

**Parameters** idx (int) – parameter index within the operation

Returns list of multiplier, coefficient, shift for each term in the gradient recipe

**Return type** list[[float, float, float]]

Note that the default value for shift is None, which is replaced by the default shift  $\pi/2$ .

#### inv()

Inverts the operator.

This method concatenates a string to the name of the operation, to indicate that the inverse will be used for computations.

Any subsequent call of this method will toggle between the original operation and the inverse of the operation.

Returns operation to be inverted

Return type Operator

**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 caries information between label calls in the same drawing

Returns label to use in drawings

Return type str

#### **Example:**

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

(continues on next page)

(continued from previous page)

```
>>> op.label(base_label="my_label")
"my_label"
>>> op.label(decimals=2, base_label="my_label")
"my_label\n(1.23)"
>>> op.inv()
>>> op.label()
"RX1"
```

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)]
```

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

Append the operator to the Operator queue.

```
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.

**Note:** The wire\_order argument is currently not implemented, and using it will raise an error.

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** matrix representation

**Return type** scipy.sparse.coo.coo\_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.

## See also:

compute\_terms()

**Returns** list of coefficients  $c_i$  and list of operations  $O_i$ 

**Return type** tuple[list[tensor\_like or float], list[Operation]]

## **SimulatorDevice**

class SimulatorDevice(wires, shots=None, qubits=None, simulator=None)

Bases: pennylane\_cirq.cirq\_device.CirqDevice

Cirq simulator 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']).
- **shots** (*int*) Number of circuit evaluations/random samples used to estimate expectation values of observables. Shots need to >= 1. If None, expectation values are calculated analytically.
- **qubits** (*List[cirq.Qubit]*) A list of Cirq qubits that are used as wires. The wire number corresponds to the index in the list. By default, an array of cirq.LineQubit instances is created.
- **simulator** (*Optional* [*cirq*. *Simulator*]) Optional custom simulator object to use. If None, the default cirq. Simulator() will be used instead.

analytic	Whether shots is None or not.
author	
circuit_hash	The hash of the circuit upon the last execution.
name	
num_executions	Number of times this device is executed by the eval-
	uation of QNodes running on this device
obs_queue	The observables to be measured and returned.
observables	set() -> new empty set object set(iterable) -> new set
	object
op_queue	The operation queue to be applied.
operations	Get the supported set of operations.
parameters	Mapping from free parameter index to the list of
	Operations in the device queue that depend on it.
pennylane_requires	
_	continues on next page

Table 16 – continued from previous page

short_name	
shot_vector	Returns the shot vector, a sparse representation of the shot sequence used by the device when evaluating
	QNodes.
shots	Number of circuit evaluations/random samples used to estimate expectation values of observables
state	Returns the state vector of the circuit prior to measurement.
stopping_condition	Returns the stopping condition for the device.
version	
wire_map	Ordered dictionary that defines the map from user- provided wire labels to the wire labels used on this device
wires	All wires that can be addressed on this device

# analytic

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

## author = 'Xanadu Inc'

# circuit\_hash

The hash of the circuit upon the last execution.

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

# name = 'Cirq Simulator device 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

# 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 = '>=0.17.0'
short_name = 'cirq.simulator'
```

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

# Example

shot\_vector

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

```
Type list[ShotTuple[int, int]]
```

## 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:** The state includes possible basis rotations for non-diagonal observables. Note that this behaviour differs from PennyLane's default.qubit plugin.

# stopping\_condition

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

```
Type BooleanFn
```

```
version = '0.28.0'
```

## wire\_map

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

93

#### wires

All wires that can be addressed on this device

access_state([wires])	Check that the device has access to an internal state
_	and return it if available.
active_wires(operators)	Returns the wires acted on by a set of operators.
adjoint_jacobian(tape[, starting_state,])	Implements the adjoint method outlined in Jones and
	Gacon to differentiate an input tape.
analytic_probability([wires])	Return the (marginal) probability of each computa-
	tional basis state from the last run of the device.
apply(operations, **kwargs)	Apply quantum operations, rotate the circuit into
	the measurement basis, and compile and execute the
	quantum circuit.
batch_execute(circuits)	Execute a batch of quantum circuits on the device.
batch_transform(circuit)	Apply a differentiable batch transform for prepro-
	cessing a circuit prior to execution.
capabilities()	Get the capabilities of this device class.
<pre>check_validity(queue, observables)</pre>	Checks whether the operations and observables in
	queue are all supported by the device.
custom_expand(fn)	Register a custom expansion function for the device.
<pre>default_expand_fn(circuit[, max_expansion])</pre>	Method for expanding or decomposing an input cir-
	cuit.
<pre>define_wire_map(wires)</pre>	Create the map from user-provided wire labels to the
	wire labels used by the device.
density_matrix(wires)	Returns the reduced density matrix prior to measure-
	ment.
<pre>estimate_probability([wires, shot_range,])</pre>	Return the estimated probability of each computa-
	tional basis state using the generated samples.
execute(circuit, **kwargs)	Execute a queue of quantum operations on the device
	and then measure the given observables.
<pre>execute_and_gradients(circuits[, method])</pre>	Execute a batch of quantum circuits on the device,
	and return both the results and the gradients.
execution_context()	The device execution context used during calls to
	execute().
<pre>expand_fn(circuit[, max_expansion])</pre>	Method for expanding or decomposing an input cir-
	cuit.
<pre>expval(observable[, shot_range, bin_size])</pre>	Returns the expectation value of observable on spec-
	ified wires.
<pre>generate_basis_states(num_wires[, dtype])</pre>	Generates basis states in binary representation ac-
	cording to the number of wires specified.
<pre>generate_samples()</pre>	Returns the computational basis samples generated
	for all wires.
<pre>gradients(circuits[, method])</pre>	Return the gradients of a batch of quantum circuits
	on the device.
<pre>map_wires(wires)</pre>	Map the wire labels of wires using this device's wire
	map.
<pre>marginal_prob(prob[, wires])</pre>	Return the marginal probability of the computational
	basis states by summing the probabiliites on the non-
	specified wires.
order_wires(subset_wires)	Given some subset of device wires return a Wires ob-
order_wires(subset_wires)	Given some subset of device wires return a Wires object with the same wires; sorted according to the de-
order_wires(subset_wires)	
<pre>order_wires(subset_wires)  post_apply()</pre>	ject with the same wires; sorted according to the de-

94 Chapter 2. Tutorials

Table 17 – continued from previous page

Table 17 Continue	a nom previous page
<pre>post_measure()</pre>	Called during execute() after the individual ob-
	servables have been measured.
pre_apply()	Called during execute() before the individual op-
	erations are executed.
<pre>pre_measure()</pre>	Called during <i>execute()</i> before the individual ob-
	servables are measured.
<pre>probability([wires, shot_range, bin_size])</pre>	Return either the analytic probability or estimated
	probability of each computational basis state.
reset()	Reset the backend state.
<pre>sample(observable[, shot_range, bin_size])</pre>	Return a sample of an observable.
<pre>sample_basis_states(number_of_states,)</pre>	Sample from the computational basis states based on
	the state probability.
states_to_binary(samples, num_wires[, dtype])	Convert basis states from base 10 to binary represen-
	tation.
<pre>statistics(observables[, shot_range, bin_size])</pre>	Process measurement results from circuit execution
	and return statistics.
<pre>supports_observable(observable)</pre>	Checks if an observable is supported by this device.
	Raises a ValueError,
supports_operation(operation)	Checks if an operation is supported by this device.
to_paulistring(observable)	Convert an observable to a cirq.PauliString
<pre>var(observable[, shot_range, bin_size])</pre>	Returns the variance of observable on specified
-	wires.

# 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, 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 inverse (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 Hamiltonian observables.

**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

**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, \*\*kwargs)

Apply quantum operations, rotate the circuit into the measurement basis, and compile and execute the quantum circuit.

This method receives a list of quantum operations queued by the QNode, and should be responsible for:

- Constructing the quantum program
- (Optional) Rotating the quantum circuit using the rotation operations provided. This diagonalizes the circuit so that arbitrary observables can be measured in the computational basis.
- · Compile the circuit
- Execute the quantum circuit

Both arguments are provided as lists of PennyLane Operation instances. Useful properties include name, wires, and parameters, and inverse:

(continues on next page)

(continued from previous page)

```
<Wires = [0]>
>>> op.parameters # returns a list of parameters
[0.2]
>>> op.inverse # check if the operation should be inverted
False
>>> op = qml.RX(0.2, wires=[0]).inv
>>> op.inverse
True
```

**Parameters operations** (list[Operation]) – operations to apply to the device

# **Keyword Arguments**

- **rotations** (*list[Operation]*) operations that rotate the circuit pre-measurement into the eigenbasis of the observables.
- hash (int) the hash value of the circuit constructed by CircuitGraph.hash

# 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[tapes.QuantumTape]) – circuits to execute on the device

**Returns** list of measured value(s)

**Return type** list[array[float]]

## batch\_transform(circuit)

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()

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_inverse_operations=False,
        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. Includes checks for inverse operations.

#### **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

#### custom\_expand(fn)

Register a custom expansion function for the device.

## Example

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

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

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

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

#### default\_expand\_fn(circuit, max expansion=10)

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

By default, this method expands the tape if:

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

#### **Parameters**

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

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

Return type QuantumTape

#### define\_wire\_map(wires)

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

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

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

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

Returns dictionary specifying the wire map

Return type OrderedDict

## Example

# density\_matrix(wires)

Returns the reduced density matrix prior to measurement.

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

```
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)

Execute 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 the 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** (*CircuitGraph*) – 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 Device.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 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]

# 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)
```

Return a sample of an observable.

The number of samples is determined by the value of Device.shots, which can be directly modified.

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 sampling

**Returns** samples in an array of dimension (shots,)

**Return type** array[float]

# 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]

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(observables, 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**

- **observables** (*List* [*Observable*]) the observables to be measured
- **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 iff supplied operation is supported

Return type bool

# to\_paulistring(observable)

Convert an observable to a cirq.PauliString

var(observable, shot\_range=None, bin\_size=None)

Returns the variance of observable on specified wires.

Note: all arguments support \_lists\_, which indicate a tensor product of observables.

## **Parameters**

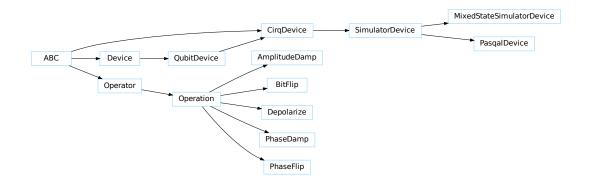
- **observable** (str or list[str]) name of the observable(s)
- wires (Wires) wires the observable(s) is to be measured on
- par (tuple or list[tuple]]) parameters for the observable(s)

Raises NotImplementedError – if the device does not support variance computation

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

Return type float

# 2.7.3 Class Inheritance Diagram



106 Chapter 2. Tutorials

# **PYTHON MODULE INDEX**

p
pennylane\_cirq, 12

108 Python Module Index

# **INDEX**

A	basis (Depolarize attribute), 33
<pre>access_state() (MixedStateSimulatorDevice method),</pre>	basis (PhaseDamp attribute), 71
47	basis (PhaseFlip attribute), 81
<pre>access_state() (PasqalDevice method), 61</pre>	batch_execute() (MixedStateSimulatorDevice
<pre>access_state() (SimulatorDevice method), 95</pre>	method), 48
<pre>active_wires() (MixedStateSimulatorDevice static</pre>	batch_execute() (PasqalDevice method), 63
method), 47	batch_execute() (SimulatorDevice method), 97
<pre>active_wires() (PasqalDevice static method), 61</pre>	batch_transform() (MixedStateSimulatorDevice
<pre>active_wires() (SimulatorDevice static method), 95</pre>	method), 49
<pre>adjoint() (AmplitudeDamp method), 16</pre>	batch_transform() (PasqalDevice method), 63
adjoint() (BitFlip method), 26	batch_transform() (SimulatorDevice method), 97
adjoint() (Depolarize method), 36	BitFlip (class in pennylane_cirq), 22
<pre>adjoint() (PhaseDamp method), 74</pre>	C
adjoint() (PhaseFlip method), 85	
adjoint_jacobian() (MixedStateSimulatorDevice	<pre>capabilities() (MixedStateSimulatorDevice method),</pre>
method), 47	49
adjoint_jacobian() (PasqalDevice method), 61	capabilities() (PasqalDevice method), 63
adjoint_jacobian() (SimulatorDevice method), 95	capabilities() (SimulatorDevice method), 97
AmplitudeDamp (class in pennylane_cirq), 12	check_validity() (MixedStateSimulatorDevice
analytic (MixedStateSimulatorDevice attribute), 43	method), 49
analytic (PasqalDevice attribute), 58	check_validity() (PasqalDevice method), 64
analytic (SimulatorDevice attribute), 92	check_validity() (SimulatorDevice method), 98
analytic_probability() (MixedStateSimulatorDe-	<pre>circuit_hash (MixedStateSimulatorDevice attribute),</pre>
vice method), 47	44
analytic_probability() (PasqalDevice method), 62	circuit_hash (PasqalDevice attribute), 58
<pre>analytic_probability() (SimulatorDevice method),</pre>	circuit_hash (SimulatorDevice attribute), 92
96	compute_decomposition() (AmplitudeDamp static
apply() (MixedStateSimulatorDevice method), 48	method), 16
apply() (PasqalDevice method), 62	compute_decomposition() (BitFlip static method), 26
apply() (Simulator Device method), 96	compute_decomposition() (Depolarize static
author (MixedStateSimulatorDevice attribute), 44	method), 36
author (PasqalDevice attribute), 58	compute_decomposition() (PhaseDamp static
author (SimulatorDevice attribute), 92	method), 75
В	<pre>compute_decomposition() (PhaseFlip static method),</pre>
base_name (AmplitudeDamp attribute), 13	<pre>compute_diagonalizing_gates() (AmplitudeDamp</pre>
base_name (BitFlip attribute), 23	static method), 17
base_name (Depolarize attribute), 33	compute_diagonalizing_gates() (BitFlip static
base_name (PhaseDamp attribute), 71	method), 27
base_name ( <i>PhaseFlip attribute</i> ), 81	<pre>compute_diagonalizing_gates() (Depolarize static</pre>
basis (AmplitudeDamp attribute), 13	method), 37
basis (BitFlip attribute), 23	

	Annation was said O
<pre>compute_diagonalizing_gates() (PhaseDamp static</pre>	density_matrix() (MixedStateSimulatorDevice method), 51
<pre>compute_diagonalizing_gates() (PhaseFlip static</pre>	<pre>density_matrix() (PasqalDevice method), 65 density_matrix() (SimulatorDevice method), 99</pre>
<pre>compute_eigvals() (AmplitudeDamp static method),</pre>	Depolarize (class in pennylane_cirq), 32 diagonalizing_gates() (AmplitudeDamp method),
compute_eigvals() (BitFlip static method), 27	19
<pre>compute_eigvals() (Depolarize static method), 37</pre>	<pre>diagonalizing_gates() (BitFlip method), 29</pre>
<pre>compute_eigvals() (PhaseDamp static method), 75</pre>	<pre>diagonalizing_gates() (Depolarize method), 39</pre>
<pre>compute_eigvals() (PhaseFlip static method), 86</pre>	<pre>diagonalizing_gates() (PhaseDamp method), 77</pre>
<pre>compute_matrix() (AmplitudeDamp static method), 17</pre>	<pre>diagonalizing_gates() (PhaseFlip method), 87</pre>
compute_matrix() (BitFlip static method), 28	_
compute_matrix() (Depolarize static method), 38	E
<pre>compute_matrix() (PhaseDamp static method), 76</pre>	eigvals (AmplitudeDamp attribute), 13
<pre>compute_matrix() (PhaseFlip static method), 86</pre>	eigvals (BitFlip attribute), 23
<pre>compute_sparse_matrix() (AmplitudeDamp static</pre>	eigvals (Depolarize attribute), 33
method), 18	eigvals (PhaseDamp attribute), 72
<pre>compute_sparse_matrix() (BitFlip static method), 28</pre>	eigvals (PhaseFlip attribute), 82
<pre>compute_sparse_matrix() (Depolarize static</pre>	<pre>estimate_probability() (MixedStateSimulatorDe- vice method), 51</pre>
compute_sparse_matrix() (PhaseDamp static	<pre>estimate_probability() (PasqalDevice method), 65</pre>
method), 76	estimate_probability() (SimulatorDevice method),
compute_sparse_matrix() (PhaseFlip static method), 86	99
compute_terms() (AmplitudeDamp static method), 18	execute() (MixedStateSimulatorDevice method), 51
compute_terms() (BitFlip static method), 28	execute() (PasqalDevice method), 65
compute_terms() (Depolarize static method), 38	execute() (SimulatorDevice method), 99
compute_terms() (PhaseDamp static method), 77	execute_and_gradients() (MixedStateSimulatorDe-
compute_terms() (PhaseFlip static method), 77  compute_terms() (PhaseFlip static method), 87	vice method), 51
control_wires (AmplitudeDamp attribute), 13	execute_and_gradients() (PasqalDevice method), 66
control_wires (http://wirebute), 23	execute_and_gradients() (SimulatorDevice
control_wires (Depolarize attribute), 33	method), 100 execution_context() (MixedStateSimulatorDevice
control_wires (PhaseDamp attribute), 72	
control_wires (PhaseFlip attribute), 82	method), 52
custom_expand() (MixedStateSimulatorDevice	execution_context() (PasqalDevice method), 66 execution_context() (SimulatorDevice method), 100
method), 50	expand() (AmplitudeDamp method), 19
custom_expand() (PasqalDevice method), 64	expand() (BitFlip method), 29
custom_expand() (SimulatorDevice method), 98	expand() (Depolarize method), 39
	expand() (PhaseDamp method), 77
D	expand() (PhaseFlip method), 88
decomposition() (AmplitudeDamp method), 18	expand_fn() (MixedStateSimulatorDevice method), 52
decomposition() (BitFlip method), 29	expand_fn() (PasqalDevice method), 66
decomposition() (Depolarize method), 39	expand_fn() (SimulatorDevice method), 100
decomposition() (PhaseDamp method), 77	expval() (MixedStateSimulatorDevice method), 52
decomposition() (PhaseFlip method), 87	expval() (PasqalDevice method), 66
<pre>default_expand_fn() (MixedStateSimulatorDevice</pre>	expval() (SimulatorDevice method), 101
default_expand_fn() (PasqalDevice method), 64	G
default_expand_fn() (SimulatorDevice method), 98	<pre>generate_basis_states() (MixedStateSimulatorDe-</pre>
define_wire_map() (MixedStateSimulatorDevice	vice static method), 52
method), 50	<pre>generate_basis_states() (PasqalDevice static</pre>
define_wire_map() (PasqalDevice method), 65	method), 67
define_wire_map() (SimulatorDevice method), 99	<pre>generate_basis_states() (SimulatorDevice static     method), 101</pre>

	Device hyperparameters (PhaseFlip attribute), 82
method), 53	· I
generate_samples() (PasqalDevice method), 67	101
<pre>generate_samples() (SimulatorDevice method), generator() (AmplitudeDamp method), 19</pre>	20 (imprimile 2 amp annie me), 1
generator() (AmplitudeDamp method), 19 generator() (BitFlip method), 29	id (BitFlip attribute), 24
generator() (Burup memoa), 29 generator() (Depolarize method), 39	id (Depolarize attribute), 34
generator() (PhaseDamp method), 78	id (PhaseDamp attribute), 72
generator() (PhaseFlip method), 88	id (PhaseFlip attribute), 83
get_eigvals() (AmplitudeDamp method), 19	inv() (AmplitudeDamp method), 20
get_eigvals() (AmplitudeDamp memod), 19 get_eigvals() (BitFlip method), 29	inv() (BitFlip method), 30
get_eigvals() (Depolarize method), 40	inv() (Depolarize method), 41
get_eigvals() (PhaseDamp method), 78	inv() (PhaseDamp method), 79
get_eigvals() (PhaseFlip method), 88	inv() (PhaseFlip method), 89
get_ergvars() (I maser up memoa), 66 get_matrix() (AmplitudeDamp method), 20	inverse (AmplitudeDamp attribute), 14
get_matrix() (ImplitudeDump method), 20 get_matrix() (BitFlip method), 30	inverse (BitFlip attribute), 24
get_matrix() (Depolarize method), 40	inverse (Depolarize attribute), 34
get_matrix() (PhaseDamp method), 78	inverse ( <i>PhaseDamp attribute</i> ), 72
get_matrix() (PhaseFlip method), 88	inverse (PhaseFlip attribute), 83
get_matir*() (**********************************	ethod),
20	<del>-</del>
<pre>get_parameter_shift() (BitFlip method), 30</pre>	label() (AmplitudeDamp method), 21
get_parameter_shift() (Depolarize method), 4	label() ( <i>BitFlip method</i> ), 31
get_parameter_shift() (PhaseDamp method),	70 Tabel() (Depotanze method), 41
get_parameter_shift() (PhaseFlip method), 89	Tabel() (FnaseDamp meinoa), 19
grad_method (AmplitudeDamp attribute), 13	label() ( <i>PhaseFlip method</i> ), 89
grad_method (BitFlip attribute), 24	N //
grad_method (Depolarize attribute), 34	M
grad_method ( <i>PhaseDamp attribute</i> ), 72	<pre>map_wires() (MixedStateSimulatorDevice method), 53</pre>
grad_method ( <i>PhaseFlip attribute</i> ), 82	<pre>map_wires() (PasqalDevice method), 67</pre>
grad_recipe (AmplitudeDamp attribute), 14	<pre>map_wires() (SimulatorDevice method), 102</pre>
grad_recipe (BitFlip attribute), 24	marginal_prob() (MixedStateSimulatorDevice
grad_recipe (Depolarize attribute), 34	method), 54
grad_recipe (PhaseDamp attribute), 72	<pre>marginal_prob() (PasqalDevice method), 68</pre>
grad_recipe (PhaseFlip attribute), 82	marginal_prob() (SimulatorDevice method), 102
gradients() (MixedStateSimulatorDevice method	d), 53 matrix (AmplitudeDamp attribute), 14
gradients() (PasqalDevice method), 67	matrix (BitFlip attribute), 24
gradients() (SimulatorDevice method), 101	matrix (Depolarize attribute), 34
	matrix (PhaseDamp attribute), 73
H	matrix (PhaseFlip attribute), 83
has_matrix (AmplitudeDamp attribute), 14	MixedStateSimulatorDevice (class in penny-
has_matrix (BitFlip attribute), 24	lane_cirq), 43
has_matrix (Depolarize attribute), 34	module
has_matrix (PhaseDamp attribute), 72	pennylane_cirq, 12
has_matrix (PhaseFlip attribute), 82	N1
hash (AmplitudeDamp attribute), 14	N
hash (BitFlip attribute), 24	name (AmplitudeDamp attribute), 14
hash (Depolarize attribute), 34	name (BitFlip attribute), 25
hash (PhaseDamp attribute), 72	name (Depolarize attribute), 35
hash ( <i>PhaseFlip attribute</i> ), 82	name (MixedStateSimulatorDevice attribute), 44
hyperparameters ( <i>AmplitudeDamp attribute</i> ), 14	name (PasqalDevice attribute), 58
hyperparameters ( <i>BitFlip attribute</i> ), 24	name (PhaseDamp attribute), 73
hyperparameters (Depolarize attribute), 34	name (PhaseFlip attribute), 83
hyperparameters (PhaseDamp attribute), 72	name (SimulatorDevice attribute), 92

num_executions (MixedStateSimulatorDevice at-	pennylane_cirq
tribute), 44	module, 12
num_executions (PasqalDevice attribute), 58	<pre>pennylane_requires (MixedStateSimulatorDevice at-</pre>
num_executions (SimulatorDevice attribute), 92	tribute), 44
num_params (AmplitudeDamp attribute), 14	pennylane_requires (PasqalDevice attribute), 58
num_params (BitFlip attribute), 25	pennylane_requires (SimulatorDevice attribute), 93
num_params (Depolarize attribute), 35	PhaseDamp (class in pennylane_cirq), 71
num_params (PhaseDamp attribute), 73	PhaseFlip (class in pennylane_cirq), 81
num_params ( <i>PhaseFlip attribute</i> ), 83	post_apply() (MixedStateSimulatorDevice method), 54
num_wires (AmplitudeDamp attribute), 14	post_apply() (PasqalDevice method), 68
num_wires (BitFlip attribute), 25	post_apply() (SimulatorDevice method), 103
num_wires (Depolarize attribute), 35	<pre>post_measure() (MixedStateSimulatorDevice method),</pre>
num_wires ( <i>PhaseDamp attribute</i> ), 73	54
num_wires ( <i>PhaseFlip attribute</i> ), 83	post_measure() (PasqalDevice method), 68
, or	post_measure() (SimulatorDevice method), 103
0	pre_apply() (MixedStateSimulatorDevice method), 54
oha guaya (MinadStataSignulatorDaniaa attributa) 44	pre_apply() (PasqalDevice method), 68
obs_queue (MixedStateSimulatorDevice attribute), 44	pre_apply() (SimulatorDevice method), 103
obs_queue (PasqalDevice attribute), 58	pre_measure() (MixedStateSimulatorDevice method),
obs_queue (SimulatorDevice attribute), 92	54
observables (MixedStateSimulatorDevice attribute), 44	pre_measure() (PasqalDevice method), 68
observables (PasqalDevice attribute), 58	pre_measure() (SimulatorDevice method), 103
observables (SimulatorDevice attribute), 92	probability() (MixedStateSimulatorDevice method),
op_queue (MixedStateSimulatorDevice attribute), 44	54
op_queue (PasqalDevice attribute), 58	probability() (PasqalDevice method), 68
op_queue (SimulatorDevice attribute), 92	probability() (SimulatorDevice method), 103
operations (MixedStateSimulatorDevice attribute), 44	probability () (Simulator Device method), 103
operations (PasqalDevice attribute), 58	Q
operations (Simulator Device attribute), 92	
order_wires() (MixedStateSimulatorDevice method),	queue() (AmplitudeDamp method), 21
54	queue() (BitFlip method), 32
order_wires() (PasqalDevice method), 68	queue() (Depolarize method), 42
order_wires() (SimulatorDevice method), 102	queue() (PhaseDamp method), 80
P	queue() (PhaseFlip method), 90
•	R
par_domain ( <i>AmplitudeDamp attribute</i> ), 14	
par_domain (BitFlip attribute), 25	reset() (MixedStateSimulatorDevice method), 55
par_domain (Depolarize attribute), 35	reset() (PasqalDevice method), 69
par_domain ( <i>PhaseDamp attribute</i> ), 73	reset() (SimulatorDevice method), 103
par_domain ( <i>PhaseFlip attribute</i> ), 83	S
<pre>parameter_frequencies (AmplitudeDamp attribute),</pre>	
14	<pre>sample() (MixedStateSimulatorDevice method), 55</pre>
parameter_frequencies (BitFlip attribute), 25	<pre>sample() (PasqalDevice method), 69</pre>
parameter_frequencies (Depolarize attribute), 35	<pre>sample() (SimulatorDevice method), 103</pre>
parameter_frequencies ( <i>PhaseDamp attribute</i> ), 73	<pre>sample_basis_states() (MixedStateSimulatorDevice</pre>
parameter_frequencies ( <i>PhaseFlip attribute</i> ), 83	method), 55
parameters (AmplitudeDamp attribute), 15	<pre>sample_basis_states() (PasqalDevice method), 69</pre>
parameters (BitFlip attribute), 25	<pre>sample_basis_states() (SimulatorDevice method),</pre>
parameters (Depolarize attribute), 35	103
parameters (MixedStateSimulatorDevice attribute), 44	short_name (MixedStateSimulatorDevice attribute), 44
parameters (PasqalDevice attribute), 58	short_name (PasqalDevice attribute), 58
parameters (PhaseDamp attribute), 73	short_name (SimulatorDevice attribute), 93
parameters ( <i>PhaseFlip attribute</i> ), 84	$\verb shot_vector  (\textit{MixedStateSimulatorDevice attribute}), 44$
parameters (SimulatorDevice attribute), 92	shot_vector (PasqalDevice attribute), 58
PasgalDevice (class in pennylane cirg), 57	shot_vector (SimulatorDevice attribute), 93

```
shots (MixedStateSimulatorDevice attribute), 45
                                                      var() (PasgalDevice method), 70
shots (PasqalDevice attribute), 59
                                                      var() (SimulatorDevice method), 105
shots (SimulatorDevice attribute), 93
                                                      version (MixedStateSimulatorDevice attribute), 45
SimulatorDevice (class in pennylane_cirq), 91
                                                      version (PasqalDevice attribute), 59
single_qubit_rot_angles
                              (AmplitudeDamp
                                                 at-
                                                      version (SimulatorDevice attribute), 93
         tribute), 15
                                                      W
single_qubit_rot_angles (BitFlip attribute), 25
single_qubit_rot_angles (Depolarize attribute), 35
                                                      wire_map (MixedStateSimulatorDevice attribute), 45
single_qubit_rot_angles (PhaseDamp attribute), 74
                                                      wire_map (PasqalDevice attribute), 59
single_qubit_rot_angles (PhaseFlip attribute), 84
                                                      wire_map (SimulatorDevice attribute), 93
sparse_matrix() (AmplitudeDamp method), 21
                                                      wires (AmplitudeDamp attribute), 15
sparse_matrix() (BitFlip method), 32
                                                      wires (BitFlip attribute), 25
sparse_matrix() (Depolarize method), 42
                                                      wires (Depolarize attribute), 35
sparse_matrix() (PhaseDamp method), 80
                                                      wires (MixedStateSimulatorDevice attribute), 45
sparse_matrix() (PhaseFlip method), 90
                                                      wires (PasqalDevice attribute), 59
state (MixedStateSimulatorDevice attribute), 45
                                                      wires (PhaseDamp attribute), 74
state (PasqalDevice attribute), 59
                                                      wires (PhaseFlip attribute), 84
state (Simulator Device attribute), 93
                                                      wires (SimulatorDevice attribute), 93
states_to_binary()
                         (MixedStateSimulatorDevice
         static method), 55
states_to_binary() (PasqalDevice static method), 69
states_to_binary() (SimulatorDevice static method),
         104
statistics() (MixedStateSimulatorDevice method), 55
statistics() (PasqalDevice method), 70
statistics() (Simulator Device method), 104
stopping_condition (MixedStateSimulatorDevice at-
        tribute), 45
stopping_condition (PasqalDevice attribute), 59
stopping_condition (SimulatorDevice attribute), 93
supports_observable() (MixedStateSimulatorDevice
         method), 56
supports_observable() (PasqalDevice method), 70
supports_observable() (SimulatorDevice method),
         104
supports_operation()
                         (MixedStateSimulatorDevice
        method), 56
supports_operation() (PasqalDevice method), 70
supports_operation() (SimulatorDevice method),
         104
terms() (AmplitudeDamp method), 22
terms() (BitFlip method), 32
terms() (Depolarize method), 42
terms() (PhaseDamp method), 80
terms() (PhaseFlip method), 90
to_paulistring()
                         (MixedStateSimulatorDevice
         method), 56
to_paulistring() (PasqalDevice method), 70
to_paulistring() (SimulatorDevice method), 104
V
```

var() (MixedStateSimulatorDevice method), 56