

---

# amazon-braket-sdk

*Release 1.78.1.dev0*

unknown

Apr 18, 2024



# CONTENTS

- 1   Getting Started** **3**
  - 1.1   Getting Started with the Amazon Braket Python SDK . . . . . 3
- 2   Examples** **5**
  - 2.1   Examples . . . . . 5
- 3   Python SDK APIs** **11**
  - 3.1   API Reference . . . . . 11
- Python Module Index** **401**
- Index** **403**



The Amazon Braket Python SDK is an open source library to design and build quantum circuits, submit them to Amazon Braket devices as quantum tasks, and monitor their execution.

This documentation provides information about the Amazon Braket Python SDK library. The project homepage is in Github <https://github.com/amazon-braket/amazon-braket-sdk-python>. The project includes SDK source, installation instructions, and other information.



## GETTING STARTED

### 1.1 Getting Started with the Amazon Braket Python SDK

It is easy to get started with Amazon Braket Python SDK. You can get started using an Amazon Braket notebook instance or using your own environment.

For more information about Amazon Braket, see the full set of documentation at <https://docs.aws.amazon.com/braket/index.html>.

#### 1.1.1 Getting started using an Amazon Braket notebook

You can use the AWS Console to enable Amazon Braket, then create an Amazon Braket notebook instance and run your first circuit with the Amazon Braket Python SDK:

1. [Enable Amazon Braket](#).
2. [Create an Amazon Braket notebook instance](#).
3. [Run your first circuit using the Amazon Braket Python SDK](#).

When you use an Amazon Braket notebook, the Amazon Braket SDK and plugins are preloaded.

#### 1.1.2 Getting started in your environment

You can install the Amazon Braket Python SDK in your environment after enabling Amazon Braket and configuring the AWS SDK for Python:

1. [Enable Amazon Braket](#).
2. [Configure the AWS SDK for Python \(Boto3\) using the Quickstart](#).
3. [Run your first circuit using the Amazon Braket Python SDK](#).





## EXAMPLES

Explore Amazon Braket examples.

### 2.1 Examples

There are several examples available in the Amazon Braket repo: <https://github.com/amazon-braket/amazon-braket-examples>.

#### 2.1.1 Getting started

Get started on Amazon Braket with some introductory examples.

##### Getting started

A hello-world tutorial that shows you how to build a simple circuit and run it on a local simulator.

##### Running quantum circuits on simulators

This tutorial prepares a paradigmatic example for a multi-qubit entangled state, the so-called GHZ state (named after the three physicists Greenberger, Horne, and Zeilinger). The GHZ state is extremely non-classical, and therefore very sensitive to decoherence. It is often used as a performance benchmark for today's hardware. In many quantum information protocols it is used as a resource for quantum error correction, quantum communication, and quantum metrology.

**Note:** When a circuit is ran using a simulator, customers are required to use contiguous qubits/indices.

##### Running quantum circuits on QPU devices

This tutorial prepares a maximally-entangled Bell state between two qubits, for classical simulators and for QPUs. For classical devices, we can run the circuit on a local simulator or a cloud-based managed simulator. For the quantum devices, we run the circuit on the superconducting machine from Rigetti, and on the ion-trap machine provided by IonQ.

## Deep Dive into the anatomy of quantum circuits

This tutorial discusses in detail the anatomy of quantum circuits in the Amazon Braket SDK. You will learn how to build (parameterized) circuits and display them graphically, and how to append circuits to each other. Next, learn more about circuit depth and circuit size. Finally you will learn how to execute the circuit on a device of our choice (defining a quantum task) and how to track, log, recover, or cancel a quantum task efficiently.

## Superdense coding

This tutorial constructs an implementation of the superdense coding protocol using the Amazon Braket SDK. Superdense coding is a method of transmitting two classical bits by sending only one qubit. Starting with a pair of entangled qubits, the sender (aka Alice) applies a certain quantum gate to their qubit and sends the result to the receiver (aka Bob), who is then able to decode the full two-bit message.

## 2.1.2 Amazon Braket features

Learn more about the individual features of Amazon Braket.

### Getting notifications when a quantum task completes

This tutorial illustrates how Amazon Braket integrates with Amazon EventBridge for event-based processing. In the tutorial, you will learn how to configure Amazon Braket and Amazon Eventbridge to receive text notification about quantum task completions on your phone.

### Allocating Qubits on QPU Devices

This tutorial explains how you can use the Amazon Braket SDK to allocate the qubit selection for your circuits manually, when running on QPUs.

### Getting Devices and Checking Device Properties

This example shows how to interact with the Amazon Braket GetDevice API to retrieve Amazon Braket devices (such as simulators and QPUs) programmatically, and how to gain access to their properties.

### Using the tensor network simulator TN1

This notebook introduces the Amazon Braket managed tensor network simulator, TN1. You will learn about how TN1 works, how to use it, and which problems are best suited to run on TN1.

### Simulating noise on Amazon Braket

This notebook provides a detailed overview of noise simulation on Amazon Braket. You will learn how to define noise channels, apply noise to new or existing circuits, and run those circuits on the Amazon Braket noise simulators.

### 2.1.3 Advanced circuits and algorithms

Learn more about working with advanced circuits and algorithms.

#### Grover's search algorithm

This tutorial provides a step-by-step walkthrough of Grover's quantum algorithm. You learn how to build the corresponding quantum circuit with simple modular building blocks using the Amazon Braket SDK. You will learn how to build custom gates that are not part of the basic gate set provided by the SDK. A custom gate can be used as a core quantum gate by registering it as a subroutine.

#### Quantum amplitude amplification

This tutorial provides a detailed discussion and implementation of the Quantum Amplitude Amplification (QAA) algorithm using the Amazon Braket SDK. QAA is a routine in quantum computing which generalizes the idea behind Grover's famous search algorithm, with applications across many quantum algorithms. QAA uses an iterative approach to systematically increase the probability of finding one or multiple target states in a given search space. In a quantum computer, QAA can be used to obtain a quadratic speedup over several classical algorithms.

#### Quantum Fourier transform

This tutorial provides a detailed implementation of the Quantum Fourier Transform (QFT) and its inverse using Amazon Braket's SDK. The QFT is an important subroutine to many quantum algorithms, most famously Shor's algorithm for factoring and the quantum phase estimation (QPE) algorithm for estimating the eigenvalues of a unitary operator.

#### Quantum phase estimation

This tutorial provides a detailed implementation of the Quantum Phase Estimation (QPE) algorithm using the Amazon Braket SDK. The QPE algorithm is designed to estimate the eigenvalues of a unitary operator. Eigenvalue problems can be found across many disciplines and application areas, including principal component analysis (PCA) as used in machine learning and the solution of differential equations in mathematics, physics, engineering and chemistry.

### 2.1.4 Hybrid quantum algorithms

Learn more about hybrid quantum algorithms.

#### QAOA

This tutorial shows how to (approximately) solve binary combinatorial optimization problems using the Quantum Approximate Optimization Algorithm (QAOA).

## VQE Transverse Ising

This tutorial shows how to solve for the ground state of the Transverse Ising Model using the variational quantum eigenvalue solver (VQE).

## VQE Chemistry

This tutorial shows how to implement the Variational Quantum Eigensolver (VQE) algorithm in Amazon Braket SDK to compute the potential energy surface (PES) for the Hydrogen molecule.

## 2.1.5 Quantum machine learning and optimization with PennyLane

Learn more about how to combine PennyLane with Amazon Braket.

### Combining PennyLane with Amazon Braket

This tutorial shows you how to construct circuits and evaluate their gradients in PennyLane with execution performed using Amazon Braket.

### Computing gradients in parallel with PennyLane-Braket

Learn how to speed up training of quantum circuits by using parallel execution on Amazon Braket. Quantum circuit training involving gradients requires multiple device executions. The Amazon Braket SV1 simulator can be used to overcome this. The tutorial benchmarks SV1 against a local simulator, showing that SV1 outperforms the local simulator for both executions and gradient calculations. This illustrates how parallel capabilities can be combined between PennyLane and SV1.

### Graph optimization with QAOA

In this tutorial, you learn how quantum circuit training can be applied to a problem of practical relevance in graph optimization. It is easy to train a QAOA circuit in PennyLane to solve the maximum clique problem on a simple example graph. The tutorial then extends to a more difficult 20-node graph and uses the parallel capabilities of the Amazon Braket SV1 simulator to speed up gradient calculations and hence train the quantum circuit faster, using around 1-2 minutes per iteration.

### Hydrogen Molecule geometry with VQE

In this tutorial, you will learn how PennyLane and Amazon Braket can be combined to solve an important problem in quantum chemistry. The ground state energy of molecular hydrogen is calculated by optimizing a VQE circuit using the local Braket simulator. This tutorial highlights how qubit-wise commuting observables can be measured together in PennyLane and Amazon Braket, making optimization more efficient.

## 2.1.6 Amazon Braket Hybrid Jobs

Learn more about hybrid jobs on Amazon Braket.

### Creating your first Hybrid Job

This tutorial shows how to run your first Amazon Braket Hybrid Job.

### Quantum machine learning in Amazon Braket Hybrid Jobs

This notebook demonstrates a typical quantum machine learning workflow, including uploading data, monitoring training, and tuning hyperparameters.

### Using PennyLane with Braket Hybrid Jobs

In this tutorial, we use PennyLane within Amazon Braket Hybrid Jobs to run the Quantum Approximate Optimization Algorithm (QAOA) on a Max-Cut problem.

### Bring your own container

Amazon Braket has pre-configured containers for executing Amazon Braket Hybrid Jobs, which are sufficient for many use cases involving the Braket SDK and PennyLane. However, if we want to use custom packages outside the scope of pre-configured containers, we have the ability to supply a custom-built container. In this tutorial, we show how to use Braket Hybrid Jobs to train a quantum machine learning model using BYOC (Bring Your Own Container).



## PYTHON SDK APIS

The Amazon Braket Python SDK APIs:

### 3.1 API Reference

#### 3.1.1 braket namespace

Subpackages

**braket.ahs package**

Submodules

**braket.ahs.analog\_hamiltonian\_simulation module**

```
class braket.ahs.analog_hamiltonian_simulation.AnalogHamiltonianSimulation(register: AtomAr-  
                                                                           angement,  
                                                                           hamiltonian:  
                                                                           Hamiltonian)
```

Bases: object

Creates an AnalogHamiltonianSimulation with a given setup, and terms.

#### Parameters

- **register** (*AtomArrangement*) – The initial atom arrangement for the simulation.
- **hamiltonian** (*Hamiltonian*) – The hamiltonian to simulate.

**LOCAL\_DETUNING\_PROPERTY** = 'local\_detuning'

**DRIVING\_FIELDS\_PROPERTY** = 'driving\_fields'

**property register:** *AtomArrangement*

The initial atom arrangement for the simulation.

#### Type

*AtomArrangement*

**property hamiltonian:** *Hamiltonian*

The hamiltonian to simulate.

**Type***Hamiltonian***to\_ir()** → Program

Converts the Analog Hamiltonian Simulation into the canonical intermediate representation.

**Returns***ir.Program* – A representation of the circuit in the IR format.**discretize**(*device*: *AwsDevice*) → *AnalogHamiltonianSimulation*

Creates a new *AnalogHamiltonianSimulation* with all numerical values represented as Decimal objects with fixed precision based on the capabilities of the device.

**Parameters****device** (*AwsDevice*) – The device for which to discretize the program.**Returns***AnalogHamiltonianSimulation* – A discretized version of this program.**Raises***DiscretizationError* – If unable to discretize the program.**braket.ahs.atom\_arrangement module****class** `braket.ahs.atom_arrangement.SiteType`(*value*)

Bases: Enum

An enumeration.

**VACANT** = 'Vacant'**FILLED** = 'Filled'**class** `braket.ahs.atom_arrangement.AtomArrangementItem`(*coordinate*: *tuple*[*Number*, *Number*],  
*site\_type*: *SiteType*)

Bases: object

Represents an item (coordinate and metadata) in an atom arrangement.

**coordinate**: *tuple*[*Number*, *Number*]**site\_type**: *SiteType***class** `braket.ahs.atom_arrangement.AtomArrangement`

Bases: object

Represents a set of coordinates that can be used as a register to an *AnalogHamiltonianSimulation*.

**add**(*coordinate*: *tuple*[*Number*, *Number*] | *ndarray*, *site\_type*: *SiteType* = *SiteType.FILLED*) →  
*AtomArrangement*

Add a coordinate to the atom arrangement.

**Parameters**

- **coordinate** (*Union*[*tuple*[*Number*, *Number*], *ndarray*]) – The coordinate of the atom (in meters). The coordinates can be a numpy array of shape (2,) or a tuple of int, float, Decimal
- **site\_type** (*SiteType*) – The type of site. Optional. Default is FILLED.



**Returns**

*AtomArrangement* – returns self (to allow for chaining).

**coordinate\_list**(*coordinate\_index: Number*) → list[Number]

Returns all the coordinates at the given index.

**Parameters**

**coordinate\_index** (*Number*) – The index to get for each coordinate.

**Returns**

*list[Number]* – The list of coordinates at the given index.

**Example**

To get a list of all x-coordinates: `coordinate_list(0)` To get a list of all y-coordinates: `coordinate_list(1)`

**discretize**(*properties: DiscretizationProperties*) → *AtomArrangement*

Creates a discretized version of the atom arrangement, rounding all site coordinates to the closest multiple of the resolution. The types of the sites are unchanged.

**Parameters**

**properties** (*DiscretizationProperties*) – Capabilities of a device that represent the resolution with which the device can implement the parameters.

**Raises**

*DiscretizationError* – If unable to discretize the program.

**Returns**

*AtomArrangement* – A new discretized atom arrangement.

**braket.ahs.discretization\_types module**

**exception** `braket.ahs.discretization_types.DiscretizationError`

Bases: `Exception`

Raised if the discretization of the numerical values of the AHS program fails.

**class** `braket.ahs.discretization_types.DiscretizationProperties`(*lattice: Any, rydberg: Any*)

Bases: `object`

Capabilities of a device that represent the resolution with which the device can implement the parameters.

**Parameters**

- **(Any)** (*rydberg*) – configuration values for discretization of the lattice geometry, including the position resolution.
- **(Any)** – configuration values for discretization of Rydberg fields.

## Examples

```
lattice.geometry.positionResolution = Decimal("1E-7") rydberg.rydbergGlobal.timeResolution = Decimal("1E-9")
rydberg.rydbergGlobal.phaseResolution = Decimal("5E-7")
```

**lattice:** Any

**rydberg:** Any

## braket.ahs.driving\_field module

**class** `braket.ahs.driving_field.DrivingField`(*amplitude*: Field | TimeSeries, *phase*: Field | TimeSeries, *detuning*: Field | TimeSeries)

Bases: *Hamiltonian*

Creates a Hamiltonian term  $H_{drive}$  for the driving field that coherently transfers atoms from the ground state to the Rydberg state in an AnalogHamiltonianSimulation, defined by the formula

$$H_{drive}(t) := \frac{\Omega(t)}{2} e^{i\phi(t)} \left( \sum_k |g_k\rangle\langle r_k| + |r_k\rangle\langle g_k| \right) - \Delta(t) \sum_k |r_k\rangle\langle r_k|$$

where

$\Omega(t)$  is the global Rabi frequency in rad/s,

$\phi(t)$  is the global phase in rad/s,

$\Delta(t)$  is the global detuning in rad/s,

$|g_k\rangle$  is the ground state of atom  $k$ ,

$|r_k\rangle$  is the Rydberg state of atom  $k$ .

with the sum  $\sum_k$  taken over all target atoms.

### Parameters

- **amplitude** (*Union*[Field, TimeSeries]) – global amplitude ( $\Omega(t)$ ). Time is in s, and value is in rad/s.
- **phase** (*Union*[Field, TimeSeries]) – global phase ( $\phi(t)$ ). Time is in s, and value is in rad/s.
- **detuning** (*Union*[Field, TimeSeries]) – global detuning ( $\Delta(t)$ ). Time is in s, and value is in rad/s.

**property terms:** list[Hamiltonian]

The list of terms in this Hamiltonian.

### Type

list[Hamiltonian]

**property amplitude:** Field

The global amplitude ( $\Omega(t)$ ). Time is in s, and value is in rad/s.

### Type

Field

**property phase:** *Field*

The global phase ( $\phi(t)$ ). Time is in s, and value is in rad/s.

**Type**  
*Field*

**property detuning:** *Field*

global detuning ( $\Delta(t)$ ). Time is in s, and value is in rad/s.

**Type**  
*Field*

**stitch**(*other*: *DrivingField*, *boundary*: *StitchBoundaryCondition* = *StitchBoundaryCondition.MEAN*) → *DrivingField*

Stitches two driving fields based on `TimeSeries.stitch` method. The time points of the second *DrivingField* are shifted such that the first time point of the second *DrivingField* coincides with the last time point of the first *DrivingField*. The boundary point value is handled according to *StitchBoundaryCondition* argument value.

**Parameters**

- **other** (*DrivingField*) – The second shifting field to be stitched with.
- **boundary** (*StitchBoundaryCondition*) – {"mean", "left", "right"}. Boundary point handler.

Possible options are

- "mean" - take the average of the boundary value points of the first and the second time series.
- "left" - use the last value from the left time series as the boundary point.
- "right" - use the first value from the right time series as the boundary point.

**Returns**

*DrivingField* – The stitched *DrivingField* object.

**discretize**(*properties*: *DiscretizationProperties*) → *DrivingField*

Creates a discretized version of the Hamiltonian.

**Parameters**

**properties** (*DiscretizationProperties*) – Capabilities of a device that represent the resolution with which the device can implement the parameters.

**Returns**

*DrivingField* – A new discretized *DrivingField*.

**static from\_lists**(*times*: *list[float]*, *amplitudes*: *list[float]*, *detunings*: *list[float]*, *phases*: *list[float]*) → *DrivingField*

Builds *DrivingField* Hamiltonian from lists defining time evolution of Hamiltonian parameters (Rabi frequency, detuning, phase). The values of the parameters at each time points are global for all atoms.

**Parameters**

- **times** (*list[float]*) – The time points of the driving field
- **amplitudes** (*list[float]*) – The values of the amplitude
- **detunings** (*list[float]*) – The values of the detuning
- **phases** (*list[float]*) – The values of the phase

**Raises**

**ValueError** – If any of the input args length is different from the rest.

**Returns**

*DrivingField* – DrivingField Hamiltonian.

**braket.ahs.field module**

**class** `braket.ahs.field.Field`(*time\_series*: `TimeSeries`, *pattern*: `Pattern` | `None` = `None`)

Bases: `object`

A space and time dependent parameter of a program.

**Parameters**

- **time\_series** (`TimeSeries`) – The time series representing this field.
- **pattern** (`Optional[Pattern]`) – The local pattern of real numbers.

**property** `time_series`: `TimeSeries`

The time series representing this field.

**Type**

`TimeSeries`

**property** `pattern`: `Pattern` | `None`

The local pattern of real numbers.

**Type**

`Optional[Pattern]`

**discretize**(*time\_resolution*: `Decimal` | `None` = `None`, *value\_resolution*: `Decimal` | `None` = `None`,  
*pattern\_resolution*: `Decimal` | `None` = `None`) → `Field`

Creates a discretized version of the field, where time, value and pattern are rounded to the closest multiple of their corresponding resolutions.

**Parameters**

- **time\_resolution** (`Optional[Decimal]`) – Time resolution
- **value\_resolution** (`Optional[Decimal]`) – Value resolution
- **pattern\_resolution** (`Optional[Decimal]`) – Pattern resolution

**Returns**

*Field* – A new discretized field.

**braket.ahs.hamiltonian module**

**class** `braket.ahs.hamiltonian.Hamiltonian`(*terms*: `list[Hamiltonian]` | `None` = `None`)

Bases: `object`

A Hamiltonian representing a system to be simulated.

A Hamiltonian  $H$  may be expressed as a sum of multiple terms

$$H = \sum_i H_i$$

**property terms:** `list[Hamiltonian]`

The list of terms in this Hamiltonian.

**Type**

`list[Hamiltonian]`

**discretize**(*properties*: [DiscretizationProperties](#)) → [Hamiltonian](#)

Creates a discretized version of the Hamiltonian.

**Parameters**

**properties** ([DiscretizationProperties](#)) – Capabilities of a device that represent the resolution with which the device can implement the parameters.

**Returns**

[Hamiltonian](#) – A new discretized Hamiltonian.

## braket.ahs.local\_detuning module

**class** `braket.ahs.local_detuning.LocalDetuning`(*magnitude*: [Field](#))

Bases: [Hamiltonian](#)

Creates a Hamiltonian term  $H_{shift}$  representing the local detuning that changes the energy of the Rydberg level in an AnalogHamiltonianSimulation, defined by the formula

$$H_{shift}(t) := -\Delta(t) \sum_k h_k |r_k\rangle \langle r_k|$$

where

$\Delta(t)$  is the magnitude of the frequency shift in rad/s,

$h_k$  is the site coefficient of atom  $k$ , a dimensionless real number between 0 and 1,

$|r_k\rangle$  is the Rydberg state of atom  $k$ .

with the sum  $\sum_k$  taken over all target atoms.

**Parameters**

**magnitude** ([Field](#)) – containing the global magnitude time series  $\Delta(t)$ , where time is measured in seconds (s) and values are measured in rad/s, and the local pattern  $h_k$  of dimensionless real numbers between 0 and 1.

**property terms:** `list[Hamiltonian]`

The list of terms in this Hamiltonian.

**Type**

`list[Hamiltonian]`

**property magnitude:** [Field](#)

containing the global magnitude time series  $\Delta(t)$ , where time is measured in seconds (s) and values measured in rad/s and the local pattern  $h_k$  of dimensionless real numbers between 0 and 1.

**Type**

[Field](#)

**static from\_lists**(*times*: `list[float]`, *values*: `list[float]`, *pattern*: `list[float]`) → [LocalDetuning](#)

Get the shifting field from a set of time points, values and pattern

**Parameters**

- **times** (`list[float]`) – The time points of the shifting field

- **values** (*list[float]*) – The values of the shifting field
- **pattern** (*list[float]*) – The pattern of the shifting field

**Raises**

**ValueError** – If the length of times and values differs.

**Returns**

*LocalDetuning* – The shifting field obtained

**stitch**(*other: LocalDetuning*, *boundary: StitchBoundaryCondition = StitchBoundaryCondition.MEAN*) → *LocalDetuning*

Stitches two shifting fields based on `TimeSeries.stitch` method. The time points of the second `LocalDetuning` are shifted such that the first time point of the second `LocalDetuning` coincides with the last time point of the first `LocalDetuning`. The boundary point value is handled according to `StitchBoundaryCondition` argument value.

**Parameters**

- **other** (*LocalDetuning*) – The second local detuning to be stitched with.
- **boundary** (*StitchBoundaryCondition*) – {"mean", "left", "right"}. Boundary point handler.

Possible options are

- "mean" - take the average of the boundary value points of the first and the second time series.
- "left" - use the last value from the left time series as the boundary point.
- "right" - use the first value from the right time series as the boundary point.

**Raises**

**ValueError** – The `LocalDetuning` patterns differ.

**Returns**

*LocalDetuning* – The stitched `LocalDetuning` object.

Example (`StitchBoundaryCondition.MEAN`):

```
time_series_1 = TimeSeries.from_lists(times=[0, 0.1], values=[1, 2])
time_series_2 = TimeSeries.from_lists(times=[0.2, 0.4], values=[4, 5])
```

```
stitch_ts = time_series_1.stitch(time_series_2,
    ↪boundary=StitchBoundaryCondition.MEAN)
```

Result:

```
stitch_ts.times() = [0, 0.1, 0.3]
stitch_ts.values() = [1, 3, 5]
```

Example (`StitchBoundaryCondition.LEFT`):

```
stitch_ts = time_series_1.stitch(time_series_2,
    ↪boundary=StitchBoundaryCondition.LEFT)
```

Result:

```
stitch_ts.times() = [0, 0.1, 0.3]
stitch_ts.values() = [1, 2, 5]
```

Example (StitchBoundaryCondition.RIGHT):

```
stitch_ts = time_series_1.stitch(time_series_2,
    ↳boundary=StitchBoundaryCondition.RIGHT)
```

Result:

```
stitch_ts.times() = [0, 0.1, 0.3]
stitch_ts.values() = [1, 4, 5]
```

**discretize**(*properties*: [DiscretizationProperties](#)) → [LocalDetuning](#)

Creates a discretized version of the LocalDetuning.

**Parameters**

**properties** ([DiscretizationProperties](#)) – Capabilities of a device that represent the resolution with which the device can implement the parameters.

**Returns**

[LocalDetuning](#) – A new discretized LocalDetuning.

## braket.ahs.pattern module

**class** `braket.ahs.pattern.Pattern`(*series*: *list*[*Number*])

Bases: `object`

Represents the spatial dependence of a Field.

**Parameters**

**series** (*list*[*Number*]) – A series of numbers representing the the local pattern of real numbers.

**property series**: *list*[*Number*]

A series of numbers representing the local pattern of real numbers.

**Type**

*list*[*Number*]

**discretize**(*resolution*: *Decimal* | *None*) → [Pattern](#)

Creates a discretized version of the pattern, where each value is rounded to the closest multiple of the resolution.

**Parameters**

**resolution** (*Optional*[*Decimal*]) – Resolution of the discretization

**Returns**

[Pattern](#) – The new discretized pattern

## braket.ahs.shifting\_field module

## braket.annealing package

### Submodules

## braket.annealing.problem module

**class** `braket.annealing.problem.ProblemType`(*value*)

Bases: `str`, `Enum`

The type of annealing problem.

QUBO: Quadratic Unconstrained Binary Optimization, with values 1 and 0

ISING: Ising model, with values +/-1

**QUBO** = 'QUBO'

**ISING** = 'ISING'

**class** `braket.annealing.problem.Problem`(*problem\_type*: `ProblemType`, *linear*: `dict[int, float] | None = None`, *quadratic*: `dict[tuple[int, int], float] | None = None`)

Bases: `object`

Represents an annealing problem.

Initializes a `Problem`.

#### Parameters

- **problem\_type** (`ProblemType`) – The type of annealing problem
- **linear** (`dict[int, float] | None`) – The linear terms of this problem, as a map of variable to coefficient
- **quadratic** (`dict[tuple[int, int], float] | None`) – The quadratic terms of this problem, as a map of variables to coefficient

#### Examples

```
>>> problem = Problem(  
>>>     ProblemType.ISING,  
>>>     linear={1: 3.14},  
>>>     quadratic={(1, 2): 10.08},  
>>> )  
>>> problem.add_linear_term(2, 1.618).add_quadratic_term((3, 4), 1337)
```

**property** `problem_type`: `ProblemType`

The type of annealing problem.

#### Returns

`ProblemType` – The type of annealing problem

**property** `linear`: `dict[int, float]`

The linear terms of this problem.

#### Returns

`dict[int, float]` – The linear terms of this problem, as a map of variable to coefficient

**property** `quadratic`: `dict[tuple[int, int], float]`

The quadratic terms of this problem.

#### Returns

`dict[tuple[int, int], float]` – The quadratic terms of this problem, as a map of variables to coefficient



**add\_linear\_term**(*term: int, coefficient: float*) → *Problem*

Adds a linear term to the problem.

**Parameters**

- **term** (*int*) – The variable of the linear term
- **coefficient** (*float*) – The coefficient of the linear term

**Returns**

*Problem* – This problem object

**add\_linear\_terms**(*coefficients: dict[int, float]*) → *Problem*

Adds linear terms to the problem.

**Parameters**

**coefficients** (*dict[int, float]*) – A map of variable to coefficient

**Returns**

*Problem* – This problem object

**add\_quadratic\_term**(*term: tuple[int, int], coefficient: float*) → *Problem*

Adds a quadratic term to the problem.

**Parameters**

- **term** (*tuple[int, int]*) – The variables of the quadratic term
- **coefficient** (*float*) – The coefficient of the quadratic term

**Returns**

*Problem* – This problem object

**add\_quadratic\_terms**(*coefficients: dict[tuple[int, int], float]*) → *Problem*

Adds quadratic terms to the problem.

**Parameters**

**coefficients** (*dict[tuple[int, int], float]*) – A map of variables to coefficient

**Returns**

*Problem* – This problem object

**to\_ir**() → *Problem*

Converts this problem into IR representation.

**Returns**

*Problem* – IR representation of this problem object

## braket.aws package

### Submodules

#### braket.aws.aws\_device module

**class** braket.aws.aws\_device.**AwsDeviceType**(*value*)

Bases: str, Enum

Possible AWS device types

```
SIMULATOR = 'SIMULATOR'
```

```
QPU = 'QPU'
```

```
class braket.aws.aws_device.AwsDevice(arn: str, aws_session: AwsSession | None = None, noise_model: NoiseModel | None = None)
```

Bases: [Device](#)

Amazon Braket implementation of a device. Use this class to retrieve the latest metadata about the device and to run a quantum task on the device.

Initializes an [AwsDevice](#).

#### Parameters

- **arn** (*str*) – The ARN of the device
- **aws\_session** (*Optional*[[AwsSession](#)]) – An AWS session object. Default is `None`.
- **noise\_model** (*Optional*[[NoiseModel](#)]) – The Braket noise model to apply to the circuit before execution. Noise model can only be added to the devices that support noise simulation.

---

**Note:** Some devices (QPUs) are physically located in specific AWS Regions. In some cases, the current [aws\\_session](#) connects to a Region other than the Region in which the QPU is physically located. When this occurs, a cloned [aws\\_session](#) is created for the Region the QPU is located in.

See [braket.aws.aws\\_device.AwsDevice.REGIONS](#) for the AWS regions provider devices are located in across the AWS Braket service. This is not a device specific tuple.

---

```
REGIONS = ('us-east-1', 'us-west-1', 'us-west-2', 'eu-west-2')
```

```
DEFAULT_SHOTS_QPU = 1000
```

```
DEFAULT_SHOTS_SIMULATOR = 0
```

```
DEFAULT_MAX_PARALLEL = 10
```

```
run(task_specification: Circuit | Problem | OpenQasmProgram | BlackbirdProgram | PulseSequence | AnalogHamiltonianSimulation, s3_destination_folder: AwsSession.S3DestinationFolder | None = None, shots: int | None = None, poll_timeout_seconds: float = 432000, poll_interval_seconds: float | None = None, inputs: dict[str, float] | None = None, gate_definitions: dict[tuple[Gate, QubitSet], PulseSequence] | None = None, reservation_arn: str | None = None, *aws_quantum_task_args: Any, **aws_quantum_task_kwargs: Any) → AwsQuantumTask
```

Run a quantum task specification on this device. A quantum task can be a circuit or an annealing problem.

#### Parameters

- **task\_specification** (*Union*[[Circuit](#), [Problem](#), [OpenQasmProgram](#), [BlackbirdProgram](#), [PulseSequence](#), [AnalogHamiltonianSimulation](#)]) – Specification of quantum task (circuit, OpenQASM program or AHS program) to run on device.
- **s3\_destination\_folder** (*Optional*[[S3DestinationFolder](#)]) – The S3 location to save the quantum task's results to. Default is `<default_bucket>/tasks` if evoked outside a Braket Hybrid Job, `<Job Bucket>/jobs/<job name>/tasks` if evoked inside a Braket Hybrid Job.
- **shots** (*Optional*[*int*]) – The number of times to run the circuit or annealing problem. Default is 1000 for QPUs and 0 for simulators.

- **poll\_timeout\_seconds** (*float*) – The polling timeout for `AwsQuantumTask.result()`, in seconds. Default: 5 days.
- **poll\_interval\_seconds** (*Optional[float]*) – The polling interval for `AwsQuantumTask.result()`, in seconds. Defaults to the `getTaskPollIntervalMillis` value specified in `self.properties.service` (divided by 1000) if provided, otherwise 1 second.
- **inputs** (*Optional[dict[str, float]]*) – Inputs to be passed along with the IR. If the IR supports inputs, the inputs will be updated with this value. Default: {}.
- **gate\_definitions** (*Optional[dict[tuple[Gate, QubitSet], PulseSequence]]*) – A `dict[tuple[Gate, QubitSet], PulseSequence]` for a user defined gate calibration. The calibration is defined for a particular Gate on a particular QubitSet and is represented by a PulseSequence. Default: None.
- **reservation\_arn** (*str / None*) – The reservation ARN provided by Braket Direct to reserve exclusive usage for the device to run the quantum task on. Note: If you are creating tasks in a job that itself was created reservation ARN, those tasks do not need to be created with the reservation ARN. Default: None.
- **\*aws\_quantum\_task\_args** (*Any*) – Arbitrary arguments.
- **\*\*aws\_quantum\_task\_kwargs** (*Any*) – Arbitrary keyword arguments.

#### Returns

*AwsQuantumTask* – An `AwsQuantumTask` that tracks the execution on the device.

#### Examples

```
>>> circuit = Circuit().h(0).cnot(0, 1)
>>> device = AwsDevice("arn1")
>>> device.run(circuit, ("bucket-foo", "key-bar"))
```

```
>>> circuit = Circuit().h(0).cnot(0, 1)
>>> device = AwsDevice("arn2")
>>> device.run(task_specification=circuit,
>>>     s3_destination_folder=("bucket-foo", "key-bar"))
```

```
>>> circuit = Circuit().h(0).cnot(0, 1)
>>> device = AwsDevice("arn3")
>>> device.run(task_specification=circuit,
>>>     s3_destination_folder=("bucket-foo", "key-bar"), disable_qubit_
↳ rewiring=True)
```

```
>>> problem = Problem(
>>>     ProblemType.ISING,
>>>     linear={1: 3.14},
>>>     quadratic={(1, 2): 10.08},
>>> )
>>> device = AwsDevice("arn4")
>>> device.run(problem, ("bucket-foo", "key-bar"),
>>>     device_parameters={
>>>         "providerLevelParameters": {"postprocessingType": "SAMPLING"}}
>>> )
```

See also:

`braket.aws.aws_quantum_task.AwsQuantumTask.create()`

```
run_batch(task_specifications: Circuit | Problem | Program | Program | PulseSequence |
    AnalogHamiltonianSimulation | list[Circuit | Problem | Program | Program | PulseSequence |
    AnalogHamiltonianSimulation], s3_destination_folder: S3DestinationFolder | None = None,
    shots: int | None = None, max_parallel: int | None = None, max_connections: int = 100,
    poll_timeout_seconds: float = 432000, poll_interval_seconds: float = 1, inputs: dict[str, float] |
    list[dict[str, float]] | None = None, gate_definitions: dict[tuple[Gate, QubitSet], PulseSequence] |
    None = None, reservation_arn: str | None = None, *aws_quantum_task_args,
    **aws_quantum_task_kwargs) → AwsQuantumTaskBatch
```

Executes a batch of quantum tasks in parallel

#### Parameters

- **task\_specifications** (`Union[Union[Circuit, Problem, OpenQasmProgram, BlackbirdProgram, PulseSequence, AnalogHamiltonianSimulation], list[Union[Circuit, Problem, OpenQasmProgram, BlackbirdProgram, PulseSequence, AnalogHamiltonianSimulation]]]`) – # noqa Single instance or list of circuits, annealing problems, pulse sequences, or photonics program to run on device.
- **s3\_destination\_folder** (`Optional[S3DestinationFolder]`) – The S3 location to save the quantum tasks' results to. Default is <default\_bucket>/tasks if evoked outside a Braket Job, <Job Bucket>/jobs/<job name>/tasks if evoked inside a Braket Job.
- **shots** (`Optional[int]`) – The number of times to run the circuit or annealing problem. Default is 1000 for QPUs and 0 for simulators.
- **max\_parallel** (`Optional[int]`) – The maximum number of quantum tasks to run on AWS in parallel. Batch creation will fail if this value is greater than the maximum allowed concurrent quantum tasks on the device. Default: 10
- **max\_connections** (`int`) – The maximum number of connections in the Boto3 connection pool. Also the maximum number of thread pool workers for the batch. Default: 100
- **poll\_timeout\_seconds** (`float`) – The polling timeout for `AwsQuantumTask.result()`, in seconds. Default: 5 days.
- **poll\_interval\_seconds** (`float`) – The polling interval for `AwsQuantumTask.result()`, in seconds. Defaults to the `getTaskPollIntervalMillis` value specified in `self.properties.service` (divided by 1000) if provided, otherwise 1 second.
- **inputs** (`Optional[Union[dict[str, float], list[dict[str, float]]]]`) – Inputs to be passed along with the IR. If the IR supports inputs, the inputs will be updated with this value. Default: {}.
- **gate\_definitions** (`Optional[dict[tuple[Gate, QubitSet], PulseSequence]]`) – A `dict[tuple[Gate, QubitSet], PulseSequence]` for a user defined gate calibration. The calibration is defined for a particular Gate on a particular QubitSet and is represented by a PulseSequence. Default: None.
- **reservation\_arn** (`Optional[str]`) – The reservation ARN provided by Braket Direct to reserve exclusive usage for the device to run the quantum task on. Note: If you are creating tasks in a job that itself was created reservation ARN, those tasks do not need to be created with the reservation ARN. Default: None.

#### Returns

`AwsQuantumTaskBatch` – A batch containing all of the quantum tasks run

See also:

[`braket.aws.aws\_quantum\_task\_batch.AwsQuantumTaskBatch`](#)

**refresh\_metadata()** → None

Refresh the [`AwsDevice`](#) object with the most recent Device metadata.

**property type:** str

Return the device type

Type

str

**property provider\_name:** str

Return the provider name

Type

str

**property aws\_session:** [`AwsSession`](#)

**property arn:** str

Return the ARN of the device

Type

str

**property gate\_calibrations:** [`GateCalibrations`](#) | None

Calibration data for a QPU. Calibration data is shown for gates on particular qubits. If a QPU does not expose these calibrations, None is returned.

Returns

*Optional[GateCalibrations]* – The calibration object. Returns None if the data is not present.

**property is\_available:** bool

Returns true if the device is currently available.

Returns

*bool* – Return if the device is currently available.

**property properties:** [`DeviceCapabilities`](#)

Return the device properties

Please see `braket.device_schema` in [amazon-braket-schemas-python](#)

Type

[`DeviceCapabilities`](#)

**property topology\_graph:** [`DiGraph`](#)

topology of device as a networkx [`DiGraph`](#) object.

## Examples

```
>>> import networkx as nx
>>> device = AwsDevice("arn1")
>>> nx.draw_kamada_kawai(device.topology_graph, with_labels=True, font_weight=
↳ "bold")
```

```
>>> topology_subgraph = device.topology_graph.subgraph(range(8))
>>> nx.draw_kamada_kawai(topology_subgraph, with_labels=True, font_weight="bold
↳ ")
```

```
>>> print(device.topology_graph.edges)
```

### Returns

*DiGraph* – topology of QPU as a networkx DiGraph object. None if the topology is not available for the device.

### Type

DiGraph

**property frames:** dict[str, *Frame*]

Returns a dict mapping frame ids to the frame objects for predefined frames for this device.

**property ports:** dict[str, *Port*]

Returns a dict mapping port ids to the port objects for predefined ports for this device.

**static get\_devices**(arns: list[str] | None = None, names: list[str] | None = None, types: list[AwsDeviceType] | None = None, statuses: list[str] | None = None, provider\_names: list[str] | None = None, order\_by: str = 'name', aws\_session: AwsSession | None = None) → list[AwsDevice]

Get devices based on filters and desired ordering. The result is the AND of all the filters arns, names, types, statuses, provider\_names.

## Examples

```
>>> AwsDevice.get_devices(provider_names=['Rigetti'], statuses=['ONLINE'])
>>> AwsDevice.get_devices(order_by='provider_name')
>>> AwsDevice.get_devices(types=['SIMULATOR'])
```

### Parameters

- **arns** (*Optional*[list[str]]) – device ARN filter, default is None
- **names** (*Optional*[list[str]]) – device name filter, default is None
- **types** (*Optional*[list[AwsDeviceType]]) – device type filter, default is None QPUs will be searched for all regions and simulators will only be searched for the region of the current session.
- **statuses** (*Optional*[list[str]]) – device status filter, default is None. When None is used, RETIRED devices will not be returned. To include RETIRED devices in the results, use a filter that includes “RETIRED” for this parameter.
- **provider\_names** (*Optional*[list[str]]) – provider name filter, default is None

- **order\_by** (*str*) – field to order result by, default is `name`. Accepted values are ['arn', 'name', 'type', 'provider\_name', 'status']
- **aws\_session** (*Optional[AwsSession]*) – An AWS session object. Default is `None`.

**Raises**

**ValueError** – order\_by not in ['arn', 'name', 'type', 'provider\_name', 'status']

**Returns**

*list[AwsDevice]* – list of AWS devices

**static get\_device\_region**(*device\_arn: str*) → *str*

Gets the region from a device arn.

**Parameters**

**device\_arn** (*str*) – The device ARN.

**Raises**

**ValueError** – Raised if the ARN is not properly formatted

**Returns**

*str* – the region of the ARN.

**queue\_depth**() → *QueueDepthInfo*

Task queue depth refers to the total number of quantum tasks currently waiting to run on a particular device.

**Returns**

*QueueDepthInfo* – Instance of the QueueDepth class representing queue depth information for quantum tasks and hybrid jobs. Queue depth refers to the number of quantum tasks and hybrid jobs queued on a particular device. The normal tasks refers to the quantum tasks not submitted via Hybrid Jobs. Whereas, the priority tasks refers to the total number of quantum tasks waiting to run submitted through Amazon Braket Hybrid Jobs. These tasks run before the normal tasks. If the queue depth for normal or priority quantum tasks is greater than 4000, we display their respective queue depth as '>4000'. Similarly, for hybrid jobs if there are more than 1000 jobs queued on a device, display the hybrid jobs queue depth as '>1000'. Additionally, for QPUs if hybrid jobs queue depth is 0, we display information about priority and count of the running hybrid job.

**Example**

Queue depth information for a running job. >>> device = AwsDevice(Device.Amazon.SV1) >>> print(device.queue\_depth()) QueueDepthInfo(quantum\_tasks={<QueueType.NORMAL: 'Normal': '0', <QueueType.PRIORITY: 'Priority': '1'}, jobs='0 (1 prioritized job(s) running)')

If more than 4000 quantum tasks queued on a device. >>> device = AwsDevice(Device.Amazon.DM1) >>> print(device.queue\_depth()) QueueDepthInfo(quantum\_tasks={<QueueType.NORMAL: 'Normal': '>4000', <QueueType.PRIORITY: 'Priority': '2000'}, jobs='100')

**refresh\_gate\_calibrations**() → *GateCalibrations* | *None*

Refreshes the gate calibration data upon request.

If the device does not have calibration data, `None` is returned.

**Raises**

**URLError** – If the URL provided returns a non 2xx response.

**Returns**

*Optional[GateCalibrations]* – the calibration data for the device. `None` is returned if the device does not have a gate calibrations URL associated.

**braket.aws.aws\_quantum\_job module**

```
class braket.aws.aws_quantum_job.AwsQuantumJob(arn: str, aws_session: AwsSession | None = None,
                                              quiet: bool = False)
```

Bases: [QuantumJob](#)

Amazon Braket implementation of a quantum job.

Initializes an [AwsQuantumJob](#).

**Parameters**

- **arn** (str) – The ARN of the hybrid job.
- **aws\_session** ([AwsSession](#) | None) – The [AwsSession](#) for connecting to AWS services. Default is None, in which case an [AwsSession](#) object will be created with the region of the hybrid job.
- **quiet** (bool) – Sets the verbosity of the logger to low and does not report queue position. Default is False.

**Raises**

**ValueError** – Supplied region and session region do not match.

```
TERMINAL_STATES: ClassVar[set[str]] = {'CANCELLED', 'COMPLETED', 'FAILED'}
```

```
RESULTS_FILENAME = 'results.json'
```

```
RESULTS_TAR_FILENAME = 'model.tar.gz'
```

```
LOG_GROUP = '/aws/braket/jobs'
```

```
class LogState(value)
```

Bases: Enum

Log state enum.

```
TAILING = 'tailing'
```

```
JOB_COMPLETE = 'job_complete'
```

```
COMPLETE = 'complete'
```

```
classmethod create(device: str, source_module: str, entry_point: str | None = None, image_uri: str |
                  None = None, job_name: str | None = None, code_location: str | None = None,
                  role_arn: str | None = None, wait_until_complete: bool = False, hyperparameters:
                  dict[str, Any] | None = None, input_data: str | dict | S3DataSourceConfig | None =
                  None, instance_config: InstanceConfig | None = None, distribution: str | None =
                  None, stopping_condition: StoppingCondition | None = None, output_data_config:
                  OutputDataConfig | None = None, copy_checkpoints_from_job: str | None = None,
                  checkpoint_config: CheckpointConfig | None = None, aws_session: AwsSession |
                  None = None, tags: dict[str, str] | None = None, logger: Logger = <Logger
                  braket.aws.aws_quantum_job (WARNING)>, quiet: bool = False, reservation_arn:
                  str | None = None) → AwsQuantumJob
```

Creates a hybrid job by invoking the Braket CreateJob API.

**Parameters**

- **device** (str) – Device ARN of the QPU device that receives priority quantum task queueing once the hybrid job begins running. Each QPU has a separate hybrid jobs queue so that only one hybrid job is running at a time. The device string is accessible in the hybrid job



instance as the environment variable “AMZN\_BRAKET\_DEVICE\_ARN”. When using embedded simulators, you may provide the device argument as a string of the form: “local:<provider>/<simulator\_name>”.

- **source\_module** (*str*) – Path (absolute, relative or an S3 URI) to a python module to be tarred and uploaded. If `source_module` is an S3 URI, it must point to a tar.gz file. Otherwise, `source_module` may be a file or directory.
- **entry\_point** (*str* / *None*) – A str that specifies the entry point of the hybrid job, relative to the source module. The entry point must be in the format `importable.module` or `importable.module:callable`. For example, `source_module.submodule:start_here` indicates the `start_here` function contained in `source_module.submodule`. If `source_module` is an S3 URI, entry point must be given. Default: `source_module`’s name
- **image\_uri** (*str* / *None*) – A str that specifies the ECR image to use for executing the hybrid job. `image_uris.retrieve_image()` function may be used for retrieving the ECR image URIs for the containers supported by Braket. Default = `<Braket base image_uri>`.
- **job\_name** (*str* / *None*) – A str that specifies the name with which the hybrid job is created. Allowed pattern for hybrid job name: `^[a-zA-Z0-9](-*[a-zA-Z0-9]){0,50}$`. Default: `f'{image_uri_type}-{timestamp}'`.
- **code\_location** (*str* / *None*) – The S3 prefix URI where custom code will be uploaded. Default: `f's3://{default_bucket_name}/jobs/{job_name}/script'`.
- **role\_arn** (*str* / *None*) – A str providing the IAM role ARN used to execute the script. Default: IAM role returned by `AwsSession`’s `get_default_jobs_role()`.
- **wait\_until\_complete** (*bool*) – True if we should wait until the hybrid job completes. This would tail the hybrid job logs as it waits. Otherwise False. Default: False.
- **hyperparameters** (*dict[str, Any]* / *None*) – Hyperparameters accessible to the hybrid job. The hyperparameters are made accessible as a `dict[str, str]` to the hybrid job. For convenience, this accepts other types for keys and values, but `str()` is called to convert them before being passed on. Default: None.
- **input\_data** (*str* / *dict* / `S3DataSourceConfig` / *None*) – Information about the training data. Dictionary maps channel names to local paths or S3 URIs. Contents found at any local paths will be uploaded to S3 at `f's3://{default_bucket_name}/jobs/{job_name}/data/{channel_name}`. If a local path, S3 URI, or `S3DataSourceConfig` is provided, it will be given a default channel name “input”. Default: `{}`.
- **instance\_config** (`InstanceConfig` / *None*) – Configuration of the instance(s) for running the classical code for the hybrid job. Default: `InstanceConfig(instanceType='ml.m5.large', instanceCount=1, volumeSizeInGB=30)`.
- **distribution** (*str* / *None*) – A str that specifies how the hybrid job should be distributed. If set to “data\_parallel”, the hyperparameters for the hybrid job will be set to use data parallelism features for PyTorch or TensorFlow. Default: None.
- **stopping\_condition** (`StoppingCondition` / *None*) – The maximum length of time, in seconds, and the maximum number of quantum tasks that a hybrid job can run before being forcefully stopped. Default: `StoppingCondition(maxRuntimeInSeconds=5 * 24 * 60 * 60)`.

- **output\_data\_config** (`OutputDataConfig` / `None`) – Specifies the location for the output of the hybrid job. Default: `OutputDataConfig(s3Path=f's3://{default_bucket_name}/jobs/{job_name}/data', kmsKeyId=None)`.
- **copy\_checkpoints\_from\_job** (`str` / `None`) – A `str` that specifies the hybrid job ARN whose checkpoint you want to use in the current hybrid job. Specifying this value will copy over the checkpoint data from `use_checkpoints_from_job`'s `checkpoint_config` `s3Uri` to the current hybrid job's `checkpoint_config` `s3Uri`, making it available at `checkpoint_config.localPath` during the hybrid job execution. Default: `None`
- **checkpoint\_config** (`CheckpointConfig` / `None`) – Configuration that specifies the location where checkpoint data is stored. Default: `CheckpointConfig(localPath='/opt/jobs/checkpoints', s3Uri=f's3://{default_bucket_name}/jobs/{job_name}/checkpoints')`.
- **aws\_session** (`AwsSession` / `None`) – `AwsSession` for connecting to AWS Services. Default: `AwsSession()`
- **tags** (`dict[str, str]` / `None`) – Dict specifying the key-value pairs for tagging this hybrid job. Default: `{}`.
- **logger** (`Logger`) – Logger object with which to write logs, such as quantum task statuses while waiting for quantum task to be in a terminal state. Default is `getLogger(__name__)`
- **quiet** (`bool`) – Sets the verbosity of the logger to low and does not report queue position. Default is `False`.
- **reservation\_arn** (`str` / `None`) – the reservation window arn provided by Braket Direct to reserve exclusive usage for the device to run the hybrid job on. Default: `None`.

**Returns**

*AwsQuantumJob* – Hybrid job tracking the execution on Amazon Braket.

**Raises**

**ValueError** – Raises `ValueError` if the parameters are not valid.

**property arn:** `str`

The ARN (Amazon Resource Name) of the quantum hybrid job.

**Type**

`str`

**property name:** `str`

The name of the quantum job.

**Type**

`str`

**state**(*use\_cached\_value: bool = False*) → `str`

The state of the quantum hybrid job.

**Parameters**

**use\_cached\_value** (`bool`) – If `True`, uses the value most recently retrieved value from the Amazon Braket `GetJob` operation. If `False`, calls the `GetJob` operation to retrieve metadata, which also updates the cached value. Default = `False`.

**Returns**

*str* – The value of `status` in `metadata()`. This is the value of the `status` key in the Amazon Braket `GetJob` operation.

See also:

`metadata()`

`queue_position()` → *HybridJobQueueInfo*

The queue position details for the hybrid job.

#### Returns

*HybridJobQueueInfo* – Instance of *HybridJobQueueInfo* class representing the queue position information for the hybrid job. The `queue_position` is only returned when the hybrid job is not in `RUNNING/CANCELLING/TERMINAL` states, else `queue_position` is returned as `None`. If the queue position of the hybrid job is greater than 15, we return `>15` as the `queue_position` return value.

#### Examples

job status = QUEUED and position is 2 in the queue. `>>> job.queue_position() HybridJobQueueInfo(queue_position='2', message=None)`

job status = QUEUED and position is 18 in the queue. `>>> job.queue_position() HybridJobQueueInfo(queue_position='>15', message=None)`

job status = COMPLETED `>>> job.queue_position() HybridJobQueueInfo(queue_position=None, message='Job is in COMPLETED status. AmazonBraket does`

`not show queue position for this status.')`

`logs(wait: bool = False, poll_interval_seconds: int = 5)` → `None`

**Display logs for a given hybrid job, optionally tailing them until hybrid job is complete.**

If the output is a tty or a Jupyter cell, it will be color-coded based on which instance the log entry is from.

#### Parameters

- **wait** (*bool*) – True to keep looking for new log entries until the hybrid job completes; otherwise `False`. Default: `False`.
- **poll\_interval\_seconds** (*int*) – The interval of time, in seconds, between polling for new log entries and hybrid job completion (default: 5).

#### Raises

**exceptions.UnexpectedStatusException** – If waiting and the training hybrid job fails.

`metadata(use_cached_value: bool = False)` → `dict[str, Any]`

Gets the hybrid job metadata defined in Amazon Braket.

#### Parameters

**use\_cached\_value** (*bool*) – If `True`, uses the value most recently retrieved from the Amazon Braket `GetJob` operation, if it exists; if does not exist, `GetJob` is called to retrieve the metadata. If `False`, always calls `GetJob`, which also updates the cached value. Default: `False`.

#### Returns

`dict[str, Any]` – Dict that specifies the hybrid job metadata defined in Amazon Braket.

`metrics(metric_type: MetricType = MetricType.TIMESTAMP, statistic: MetricStatistic = MetricStatistic.MAX)` → `dict[str, list[Any]]`

Gets all the metrics data, where the keys are the column names, and the values are a list containing the values in each row. For example, the table:

**timestamp energy**

0 0.1 1 0.2

would be represented as: { “timestamp” : [0, 1], “energy” : [0.1, 0.2] } values may be integers, floats, strings or None.

#### Parameters

- **metric\_type** (`MetricType`) – The type of metrics to get. Default: `MetricType.TIMESTAMP`.
- **statistic** (`MetricStatistic`) – The statistic to determine which metric value to use when there is a conflict. Default: `MetricStatistic.MAX`.

#### Returns

*dict[str, list[Any]]* – The metrics data.

**cancel()** → str

Cancels the job.

#### Returns

*str* – Indicates the status of the job.

#### Raises

**ClientError** – If there are errors invoking the CancelJob API.

**result**(*poll\_timeout\_seconds: float = 864000, poll\_interval\_seconds: float = 5*) → dict[str, Any]

Retrieves the hybrid job result persisted using the `save_job_result` function.

#### Parameters

- **poll\_timeout\_seconds** (*float*) – The polling timeout, in seconds, for `result()`. Default: 10 days.
- **poll\_interval\_seconds** (*float*) – The polling interval, in seconds, for `result()`. Default: 5 seconds.

#### Returns

*dict[str, Any]* – Dict specifying the job results.

#### Raises

- **RuntimeError** – if hybrid job is in a FAILED or CANCELLED state.
- **TimeoutError** – if hybrid job execution exceeds the polling timeout period.

**download\_result**(*extract\_to: str | None = None, poll\_timeout\_seconds: float = 864000, poll\_interval\_seconds: float = 5*) → None

Downloads the results from the hybrid job output S3 bucket and extracts the tar.gz bundle to the location specified by `extract_to`. If no location is specified, the results are extracted to the current directory.

#### Parameters

- **extract\_to** (*str | None*) – The directory to which the results are extracted. The results are extracted to a folder titled with the hybrid job name within this directory. Default= Current working directory.
- **poll\_timeout\_seconds** (*float*) – The polling timeout, in seconds, for `download_result()`. Default: 10 days.
- **poll\_interval\_seconds** (*float*) – The polling interval, in seconds, for `download_result()`. Default: 5 seconds.

#### Raises

- **RuntimeError** – if hybrid job is in a FAILED or CANCELLED state.
- **TimeoutError** – if hybrid job execution exceeds the polling timeout period.

### braket.aws.aws\_quantum\_task module

```
class braket.aws.aws_quantum_task.AwsQuantumTask(arn: str, aws_session: AwsSession | None = None,
                                                  poll_timeout_seconds: float = 432000,
                                                  poll_interval_seconds: float = 1, logger: Logger =
<Logger braket.aws.aws_quantum_task
(WARNING)>, quiet: bool = False)
```

Bases: [QuantumTask](#)

Amazon Braket implementation of a quantum task. A quantum task can be a circuit, an OpenQASM program or an AHS program.

Initializes an [AwsQuantumTask](#).

#### Parameters

- **arn** (*str*) – The ARN of the quantum task.
- **aws\_session** ([AwsSession](#) / *None*) – The [AwsSession](#) for connecting to AWS services. Default is *None*, in which case an [AwsSession](#) object will be created with the region of the quantum task.
- **poll\_timeout\_seconds** (*float*) – The polling timeout for [result\(\)](#). Default: 5 days.
- **poll\_interval\_seconds** (*float*) – The polling interval for [result\(\)](#). Default: 1 second.
- **logger** (*Logger*) – Logger object with which to write logs, such as quantum task statuses while waiting for quantum task to be in a terminal state. Default is `getLogger(__name__)`
- **quiet** (*bool*) – Sets the verbosity of the logger to low and does not report queue position. Default is *False*.

### Examples

```
>>> task = AwsQuantumTask(arn='task_arn')
>>> task.state()
'COMPLETED'
>>> result = task.result()
AnnealingQuantumTaskResult(...)
```

```
>>> task = AwsQuantumTask(arn='task_arn', poll_timeout_seconds=300)
>>> result = task.result()
GateModelQuantumTaskResult(...)
```

```
NO_RESULT_TERMINAL_STATES: ClassVar[set[str]] = {'CANCELLED', 'FAILED'}
```

```
RESULTS_READY_STATES: ClassVar[set[str]] = {'COMPLETED'}
```

```
TERMINAL_STATES: ClassVar[set[str]] = {'CANCELLED', 'COMPLETED', 'FAILED'}
```

```
DEFAULT_RESULTS_POLL_TIMEOUT = 432000
```

```
DEFAULT_RESULTS_POLL_INTERVAL = 1
```

```
RESULTS_FILENAME = 'results.json'
```

```
static create(aws_session: AwsSession, device_arn: str, task_specification: Circuit | Problem |  
              OpenQASMProgram | BlackbirdProgram | PulseSequence | AnalogHamiltonianSimulation,  
              s3_destination_folder: AwsSession.S3DestinationFolder, shots: int, device_parameters:  
              dict[str, Any] | None = None, disable_qubit_rewiring: bool = False, tags: dict[str, str] |  
              None = None, inputs: dict[str, float] | None = None, gate_definitions: dict[tuple[Gate,  
              QubitSet], PulseSequence] | None = None, quiet: bool = False, reservation_arn: str | None  
              = None, *args, **kwargs) → AwsQuantumTask
```

AwsQuantumTask factory method that serializes a quantum task specification (either a quantum circuit or annealing problem), submits it to Amazon Braket, and returns back an AwsQuantumTask tracking the execution.

#### Parameters

- **aws\_session** (*AwsSession*) – AwsSession to connect to AWS with.
- **device\_arn** (*str*) – The ARN of the quantum device.
- **task\_specification** (*Union[Circuit, Problem, OpenQASMProgram, BlackbirdProgram, PulseSequence, AnalogHamiltonianSimulation]*) – #noqa The specification of the quantum task to run on device.
- **s3\_destination\_folder** (*AwsSession.S3DestinationFolder*) – NamedTuple, with bucket for index 0 and key for index 1, that specifies the Amazon S3 bucket and folder to store quantum task results in.
- **shots** (*int*) – The number of times to run the quantum task on the device. If the device is a simulator, this implies the state is sampled N times, where N = shots. shots=0 is only available on simulators and means that the simulator will compute the exact results based on the quantum task specification.
- **device\_parameters** (*dict[str, Any] | None*) – Additional parameters to send to the device.
- **disable\_qubit\_rewiring** (*bool*) – Whether to run the circuit with the exact qubits chosen, without any rewiring downstream, if this is supported by the device. Only applies to digital, gate-based circuits (as opposed to annealing problems). If True, no qubit rewiring is allowed; if False, qubit rewiring is allowed. Default: False
- **tags** (*dict[str, str] | None*) – Tags, which are Key-Value pairs to add to this quantum task. An example would be: {"state": "washington"}
- **inputs** (*dict[str, float] | None*) – Inputs to be passed along with the IR. If the IR supports inputs, the inputs will be updated with this value. Default: {}.
- **gate\_definitions** (*dict[tuple[Gate, QubitSet], PulseSequence] | None*) – A dict of user defined gate calibrations. Each calibration is defined for a particular Gate on a particular QubitSet and is represented by a PulseSequence. Default: None.
- **quiet** (*bool*) – Sets the verbosity of the logger to low and does not report queue position. Default is False.
- **reservation\_arn** (*str | None*) – The reservation ARN provided by Braket Direct to reserve exclusive usage for the device to run the quantum task on. Note: If you are creating tasks in a job that itself was created reservation ARN, those tasks do not need to be created with the reservation ARN. Default: None.

**Returns**

*AwsQuantumTask* – AwsQuantumTask tracking the quantum task execution on the device.

**Note:**

The following arguments are typically defined via clients of Device.

- `task_specification`
- `s3_destination_folder`
- `shots`

**See also:**

`braket.aws.aws_quantum_simulator.AwsQuantumSimulator.run()`    `braket.aws.aws_qpu.AwsQpu.run()`

**property id:** `str`

The ARN of the quantum task.

**Type**

`str`

**cancel()** → `None`

Cancel the quantum task. This cancels the future and the quantum task in Amazon Braket.

**metadata**(*use\_cached\_value: bool = False*) → `dict[str, Any]`

Get quantum task metadata defined in Amazon Braket.

**Parameters**

**use\_cached\_value** (*bool*) – If True, uses the value most recently retrieved from the Amazon Braket `GetQuantumTask` operation, if it exists; if not, `GetQuantumTask` will be called to retrieve the metadata. If False, always calls `GetQuantumTask`, which also updates the cached value. Default: False.

**Returns**

`dict[str, Any]` – The response from the Amazon Braket `GetQuantumTask` operation. If `use_cached_value` is True, Amazon Braket is not called and the most recently retrieved value is used, unless `GetQuantumTask` was never called, in which case it will still be called to populate the metadata for the first time.

**state**(*use\_cached\_value: bool = False*) → `str`

The state of the quantum task.

**Parameters**

**use\_cached\_value** (*bool*) – If True, uses the value most recently retrieved from the Amazon Braket `GetQuantumTask` operation. If False, calls the `GetQuantumTask` operation to retrieve metadata, which also updates the cached value. Default = False.

**Returns**

`str` – The value of `status` in `metadata()`. This is the value of the `status` key in the Amazon Braket `GetQuantumTask` operation. If `use_cached_value` is True, the value most recently returned from the `GetQuantumTask` operation is used.

**See also:**

[`metadata\(\)`](#)

**queue\_position()** → *QuantumTaskQueueInfo*

The queue position details for the quantum task.

**Returns**

*QuantumTaskQueueInfo* – Instance of *QuantumTaskQueueInfo* class representing the queue position information for the quantum task. The `queue_position` is only returned when quantum task is not in `RUNNING/CANCELLING/TERMINAL` states, else `queue_position` is returned as `None`. The normal tasks refers to the quantum tasks not submitted via Hybrid Jobs. Whereas, the priority tasks refers to the total number of quantum tasks waiting to run submitted through Amazon Braket Hybrid Jobs. These tasks run before the normal tasks. If the queue position for normal or priority quantum tasks is greater than 2000, we display their respective queue position as `'>2000'`.

**Examples**

```
task status = QUEUED and queue position is 2050 >>> task.queue_position() QuantumTaskQueue-
Info(queue_type=<QueueType.NORMAL: 'Normal'>, queue_position='>2000', message=None)
```

```
task status = COMPLETED >>> task.queue_position() QuantumTaskQueue-
Info(queue_type=<QueueType.NORMAL: 'Normal'>, queue_position=None, message='Task is in
COMPLETED status. AmazonBraket does not show queue position for this status.')
```

**result()** → *GateModelQuantumTaskResult* | *AnnealingQuantumTaskResult* | *PhotonicModelQuantumTaskResult*

Get the quantum task result by polling Amazon Braket to see if the task is completed. Once the quantum task is completed, the result is retrieved from S3 and returned as a *GateModelQuantumTaskResult* or *AnnealingQuantumTaskResult*

This method is a blocking thread call and synchronously returns a result. Call *async\_result()* if you require an asynchronous invocation. Consecutive calls to this method return a cached result from the preceding request.

**Returns**

*Union[GateModelQuantumTaskResult, AnnealingQuantumTaskResult, PhotonicModelQuantumTaskResult]* – The result of the quantum task, if the quantum task completed successfully; returns `None` if the quantum task did not complete successfully or the future timed out.

**async\_result()** → Task

Get the quantum task result asynchronously. Consecutive calls to this method return the result cached from the most recent request.

**braket.aws.aws\_quantum\_task\_batch module**



```

class braket.aws.aws_quantum_task_batch.AwsQuantumTaskBatch(aws_session: AwsSession,
                                                             device_arn: str, task_specifications:
                                                             Circuit | Problem |
                                                             OpenQasmProgram |
                                                             BlackbirdProgram |
                                                             AnalogHamiltonianSimulation |
                                                             list[Circuit | Problem |
                                                             OpenQasmProgram |
                                                             BlackbirdProgram |
                                                             AnalogHamiltonianSimulation],
                                                             s3_destination_folder:
                                                             AwsSession.S3DestinationFolder,
                                                             shots: int, max_parallel: int,
                                                             max_workers: int = 100,
                                                             poll_timeout_seconds: float =
                                                             432000, poll_interval_seconds: float
                                                             = 1, inputs: dict[str, float] |
                                                             list[dict[str, float]] | None = None,
                                                             gate_definitions: dict[tuple[Gate,
                                                             QubitSet], PulseSequence] |
                                                             list[dict[tuple[Gate, QubitSet],
                                                             PulseSequence]] | None = None,
                                                             reservation_arn: str | None = None,
                                                             *aws_quantum_task_args: Any,
                                                             **aws_quantum_task_kwargs: Any)

```

Bases: [QuantumTaskBatch](#)

Executes a batch of quantum tasks in parallel.

Using this class can yield vast speedups over executing quantum tasks sequentially, and is particularly useful for computations that can be parallelized, such as calculating quantum gradients or statistics of terms in a Hamiltonian.

Note: there is no benefit to using this method with QPUs outside of their execution windows, since results will not be available until the window opens.

Creates a batch of quantum tasks.

#### Parameters

- **aws\_session** ([AwsSession](#)) – AwsSession to connect to AWS with.
- **device\_arn** (*str*) – The ARN of the quantum device.
- **task\_specifications** ([Union\[Union\[Circuit, Problem, OpenQasmProgram, BlackbirdProgram, AnalogHamiltonianSimulation\], list\[Union\[Circuit, Problem, OpenQasmProgram, BlackbirdProgram, AnalogHamiltonianSimulation\]\]\]](#)) – # noqa Single instance or list of circuits, annealing problems, pulse sequences, or photonics program as specification of quantum task to run on device.
- **s3\_destination\_folder** ([AwsSession.S3DestinationFolder](#)) – NamedTuple, with bucket for index 0 and key for index 1, that specifies the Amazon S3 bucket and folder to store quantum task results in.
- **shots** (*int*) – The number of times to run the quantum task on the device. If the device is a simulator, this implies the state is sampled N times, where N = shots. shots=0 is only available on simulators and means that the simulator will compute the exact results based on the quantum task specification.

- **max\_parallel** (*int*) – The maximum number of quantum tasks to run on AWS in parallel. Batch creation will fail if this value is greater than the maximum allowed concurrent quantum tasks on the device.
- **max\_workers** (*int*) – The maximum number of thread pool workers. Default: 100
- **poll\_timeout\_seconds** (*float*) – The polling timeout for `AwsQuantumTask.result()`, in seconds. Default: 5 days.
- **poll\_interval\_seconds** (*float*) – The polling interval for results in seconds. Default: 1 second.
- **inputs** (*Union[dict[str, float], list[dict[str, float]] | None*) – Inputs to be passed along with the IR. If the IR supports inputs, the inputs will be updated with this value. Default: {}.
- **gate\_definitions** (*Union[dict[tuple[Gate, QubitSet], PulseSequence], list[dict[tuple[Gate, QubitSet], PulseSequence]] | None*) – # noqa: E501 User-defined gate calibration. The calibration is defined for a particular Gate on a particular QubitSet and is represented by a PulseSequence. Default: None.
- **reservation\_arn** (*str | None*) – The reservation ARN provided by Braket Direct to reserve exclusive usage for the device to run the quantum task on. Note: If you are creating tasks in a job that itself was created reservation ARN, those tasks do not need to be created with the reservation ARN. Default: None.
- **\*aws\_quantum\_task\_args** (*Any*) – Arbitrary args for QuantumTask.
- **\*\*aws\_quantum\_task\_kwargs** (*Any*) – Arbitrary kwargs for QuantumTask.,

**MAX\_CONNECTIONS\_DEFAULT = 100**

**MAX\_RETRIES = 3**

**results** (*fail\_unsuccessful: bool = False, max\_retries: int = 3, use\_cached\_value: bool = True*) → *list[AwsQuantumTask]*

Retrieves the result of every quantum task in the batch.

Polling for results happens in parallel; this method returns when all quantum tasks have reached a terminal state. The result of this method is cached.

#### Parameters

- **fail\_unsuccessful** (*bool*) – If set to `True`, this method will fail if any quantum task in the batch fails to return a result even after `max_retries` retries.
- **max\_retries** (*int*) – Maximum number of times to retry any failed quantum tasks, i.e. any quantum tasks in the `FAILED` or `CANCELLED` state or that didn't complete within the timeout. Default: 3.
- **use\_cached\_value** (*bool*) – If `False`, will refetch the results from S3, even when results have already been cached. Default: `True`.

#### Returns

*list[AwsQuantumTask]* – The results of all of the quantum tasks in the batch. `FAILED`, `CANCELLED`, or timed out quantum tasks will have a result of `None`

**retry\_unsuccessful\_tasks**() → *bool*

Retries any quantum tasks in the batch without valid results.

This method should only be called after `results()` has been called at least once. The method will generate new quantum tasks for any failed quantum tasks, so `self.task` and `self.results()` may return different values after a call to this method.

**Returns**

*bool* – Whether or not all retried quantum tasks completed successfully.

**property tasks:** `list[AwsQuantumTask]`

The quantum tasks in this batch, as a list of `AwsQuantumTask` objects

**Type**

`list[AwsQuantumTask]`

**property size:** `int`

The number of quantum tasks in the batch

**Type**

`int`

**property unfinished:** `set[str]`

Gets all the IDs of all the quantum tasks in the batch that have yet to complete.

**Returns**

*set[str]* – The IDs of all the quantum tasks in the batch that have yet to complete.

**property unsuccessful:** `set[str]`

The IDs of all the FAILED, CANCELLED, or timed out quantum tasks in the batch.

**Type**

`set[str]`

**braket.aws.aws\_session module**

```
class braket.aws.aws_session.AwsSession(boto_session: boto3.Session | None = None, braket_client: client
                                         | None = None, config: Config | None = None, default_bucket:
                                         str | None = None)
```

Bases: `object`

Manage interactions with AWS services.

Initializes an `AwsSession`.

**Parameters**

- **boto\_session** (*boto3.Session* | *None*) – A boto3 session object.
- **braket\_client** (*client* | *None*) – A boto3 Braket client.
- **config** (*Config* | *None*) – A botocore Config object.
- **default\_bucket** (*str* | *None*) – The name of the default bucket of the AWS Session.

**Raises**

**ValueError** – invalid `boto_session` or `braket_client`.

```
class S3DestinationFolder(bucket: str, key: str)
```

Bases: `NamedTuple`

A `NamedTuple` for an S3 bucket and object key.

Create new instance of `S3DestinationFolder(bucket, key)`

**bucket:** `str`

Alias for field number 0

**key:** `str`

Alias for field number 1

**property region:** `str`

**property account\_id:** `str`

Gets the caller's account number.

**Returns**

*str* – The account number of the caller.

**property iam\_client:** `client`

Gets the IAM client.

**Returns**

*client* – The IAM Client.

**property s3\_client:** `client`

Gets the S3 client.

**Returns**

*client* – The S3 Client.

**property sts\_client:** `client`

Gets the STS client.

**Returns**

*client* – The STS Client.

**property logs\_client:** `client`

Gets the CloudWatch logs client.

**Returns**

*client* – The CloudWatch logs Client.

**property ecr\_client:** `client`

Gets the ECR client.

**Returns**

*client* – The ECR Client.

**add\_braket\_user\_agent**(*user\_agent: str*) → None

Appends the user-agent value to the User-Agent header, if it does not yet exist in the header. This method is typically only relevant for libraries integrating with the Amazon Braket SDK.

**Parameters**

**user\_agent** (*str*) – The user-agent value to append to the header.

**cancel\_quantum\_task**(*arn: str*) → None

Cancel the quantum task.

**Parameters**

**arn** (*str*) – The ARN of the quantum task to cancel.

**create\_quantum\_task**(\*\**boto3\_kwargs*) → str

Create a quantum task.

**Parameters**

**\*\*boto3\_kwargs** – Keyword arguments for the Amazon Braket CreateQuantumTask operation.

**Returns**

*str* – The ARN of the quantum task.

**create\_job**(*\*\*boto3\_kwargs*) → *str*

Create a quantum hybrid job.

**Parameters**

**\*\*boto3\_kwargs** – Keyword arguments for the Amazon Braket CreateJob operation.

**Returns**

*str* – The ARN of the hybrid job.

**get\_quantum\_task**(*arn: str*) → *dict[str, Any]*

Gets the quantum task.

**Parameters**

**arn** (*str*) – The ARN of the quantum task to get.

**Returns**

*dict[str, Any]* – The response from the Amazon Braket GetQuantumTask operation.

**get\_default\_jobs\_role**() → *str*

This returns the role ARN for the default hybrid jobs role created in the Amazon Braket Console. It will pick the first role it finds with the RoleName prefix `AmazonBraketJobsExecutionRole` with a PathPrefix of `/service-role/`.

**Returns**

*str* – The ARN for the default IAM role for jobs execution created in the Amazon Braket console.

**Raises**

**RuntimeError** – If no roles can be found with the prefix `/service-role/AmazonBraketJobsExecutionRole`.

**get\_job**(*arn: str*) → *dict[str, Any]*

Gets the hybrid job.

**Parameters**

**arn** (*str*) – The ARN of the hybrid job to get.

**Returns**

*dict[str, Any]* – The response from the Amazon Braket GetQuantumJob operation.

**cancel\_job**(*arn: str*) → *dict[str, Any]*

Cancel the hybrid job.

**Parameters**

**arn** (*str*) – The ARN of the hybrid job to cancel.

**Returns**

*dict[str, Any]* – The response from the Amazon Braket CancelJob operation.

**retrieve\_s3\_object\_body**(*s3\_bucket: str, s3\_object\_key: str*) → *str*

Retrieve the S3 object body.

**Parameters**

- **s3\_bucket** (*str*) – The S3 bucket name.
- **s3\_object\_key** (*str*) – The S3 object key within the `s3_bucket`.

**Returns**

*str* – The body of the S3 object.

**upload\_to\_s3**(*filename: str, s3\_uri: str*) → None

Upload file to S3.

**Parameters**

- **filename** (*str*) – local file to be uploaded.
- **s3\_uri** (*str*) – The S3 URI where the file will be uploaded.

**upload\_local\_data**(*local\_prefix: str, s3\_prefix: str*) → None

Upload local data matching a prefix to a corresponding location in S3

**Parameters**

- **local\_prefix** (*str*) – a prefix designating files to be uploaded to S3. All files beginning with `local_prefix` will be uploaded.
- **s3\_prefix** (*str*) – the corresponding S3 prefix that will replace the local prefix when the data is uploaded. This will be an S3 URI and should include the bucket (i.e. `'s3://my-bucket/my/prefix-'`)

### Example

`local_prefix = "input"`, `s3_prefix = "s3://my-bucket/dir/input"` will upload:

- `'input.csv'` to `'s3://my-bucket/dir/input.csv'`
- `'input-2.csv'` to `'s3://my-bucket/dir/input-2.csv'`
- `'input/data.txt'` to `'s3://my-bucket/dir/input/data.txt'`
- `'input-dir/data.csv'` to `'s3://my-bucket/dir/input-dir/data.csv'` but will not upload:
- `'my-input.csv'`
- `'my-dir/input.csv'`

**To match all files within the directory “input” and upload them into**

`"s3://my-bucket/input"`, provide `local_prefix = "input"` and `s3_prefix = "s3://my-bucket/input/"`

**download\_from\_s3**(*s3\_uri: str, filename: str*) → None

Download file from S3

**Parameters**

- **s3\_uri** (*str*) – The S3 uri from where the file will be downloaded.
- **filename** (*str*) – filename to save the file to.

**copy\_s3\_object**(*source\_s3\_uri: str, destination\_s3\_uri: str*) → None

Copy object from another location in s3. Does nothing if source and destination URIs are the same.

**Parameters**

- **source\_s3\_uri** (*str*) – S3 URI pointing to the object to be copied.
- **destination\_s3\_uri** (*str*) – S3 URI where the object will be copied to.

**copy\_s3\_directory**(*source\_s3\_path: str, destination\_s3\_path: str*) → None

Copy all objects from a specified directory in S3. Does nothing if source and destination URIs are the same. Preserves nesting structure, will not overwrite other files in the destination location unless they share a name with a file being copied.

**Parameters**

- **source\_s3\_path** (*str*) – S3 URI pointing to the directory to be copied.
- **destination\_s3\_path** (*str*) – S3 URI where the contents of the source\_s3\_path directory will be copied to.

**list\_keys**(*bucket: str, prefix: str*) → list[str]

Lists keys matching prefix in bucket.

**Parameters**

- **bucket** (*str*) – Bucket to be queried.
- **prefix** (*str*) – The S3 path prefix to be matched

**Returns**

*list[str]* – A list of all keys matching the prefix in the bucket.

**default\_bucket**() → str

Returns the name of the default bucket of the AWS Session. In the following order of priority, it will return either the parameter `default_bucket` set during initialization of the `AwsSession` (if not `None`), the bucket being used by the currently running Braket Hybrid Job (if evoked inside of a Braket Hybrid Job), or a default value of “amazon-braket-<aws account id>-<aws session region>”. Except in the case of a user- specified bucket name, this method will create the default bucket if it does not exist.

**Returns**

*str* – Name of the default bucket.

**get\_device**(*arn: str*) → dict[str, Any]

Calls the Amazon Braket `get_device` API to retrieve device metadata.

**Parameters**

**arn** (*str*) – The ARN of the device.

**Returns**

*dict[str, Any]* – The response from the Amazon Braket `GetDevice` operation.

**search\_devices**(*arns: list[str] | None = None, names: list[str] | None = None, types: list[str] | None = None, statuses: list[str] | None = None, provider\_names: list[str] | None = None*) → list[dict[str, Any]]

Get devices based on filters. The result is the AND of all the filters `arns`, `names`, `types`, `statuses`, `provider_names`.

**Parameters**

- **arns** (*Optional[list[str]]*) – device ARN filter, default is `None`.
- **names** (*Optional[list[str]]*) – device name filter, default is `None`.
- **types** (*Optional[list[str]]*) – device type filter, default is `None`.
- **statuses** (*Optional[list[str]]*) – device status filter, default is `None`. When `None` is used, `RETIRED` devices will not be returned. To include `RETIRED` devices in the results, use a filter that includes “`RETIRED`” for this parameter.
- **provider\_names** (*Optional[list[str]]*) – provider name list, default is `None`.

**Returns**

*list[dict[str, Any]]* – The response from the Amazon Braket `SearchDevices` operation.

**static is\_s3\_uri**(string: str) → bool

Determines if a given string is an S3 URI.

**Parameters**

**string** (str) – the string to check.

**Returns**

bool – Returns True if the given string is an S3 URI.

**static parse\_s3\_uri**(s3\_uri: str) → tuple[str, str]

Parse S3 URI to get bucket and key

**Parameters**

**s3\_uri** (str) – S3 URI.

**Returns**

tuple[str, str] – Bucket and Key tuple.

**Raises**

- **ValueError** – Raises a ValueError if the provided string is not
- **a valid S3 URI.** –

**static construct\_s3\_uri**(bucket: str, \*dirs: str) → str

Create an S3 URI given a bucket and path.

**Parameters**

- **bucket** (str) – S3 URI.
- **\*dirs** (str) – directories to be appended in the resulting S3 URI

**Returns**

str – S3 URI

**Raises**

- **ValueError** – Raises a ValueError if the provided arguments are not
- **valid to generate an S3 URI** –

**describe\_log\_streams**(log\_group: str, log\_stream\_prefix: str, limit: int | None = None, next\_token: str | None = None) → dict[str, Any]

Describes CloudWatch log streams in a log group with a given prefix.

**Parameters**

- **log\_group** (str) – Name of the log group.
- **log\_stream\_prefix** (str) – Prefix for log streams to include.
- **limit** (Optional[int]) – Limit for number of log streams returned. default is 50.
- **next\_token** (Optional[str]) – The token for the next set of items to return. Would have been received in a previous call.

**Returns**

dict[str, Any] – Dictionary containing logStreams and nextToken

**get\_log\_events**(log\_group: str, log\_stream: str, start\_time: int, start\_from\_head: bool = True, next\_token: str | None = None) → dict[str, Any]

Gets CloudWatch log events from a given log stream.

**Parameters**



- **log\_group** (*str*) – Name of the log group.
- **log\_stream** (*str*) – Name of the log stream.
- **start\_time** (*int*) – Timestamp that indicates a start time to include log events.
- **start\_from\_head** (*bool*) – Bool indicating to return oldest events first. default is True.
- **next\_token** (*Optional[str]*) – The token for the next set of items to return. Would have been received in a previous call.

**Returns**

*dict[str, Any]* – Dictionary containing events, nextForwardToken, and nextBackwardToken

**copy\_session**(*region: str | None = None, max\_connections: int | None = None*) → *AwsSession*

Creates a new *AwsSession* based on the region.

**Parameters**

- **region** (*Optional[str]*) – Name of the region. Default = None.
- **max\_connections** (*Optional[int]*) – The maximum number of connections in the Boto3 connection pool. Default = None.

**Returns**

*AwsSession* – based on the region and boto config parameters.

**get\_full\_image\_tag**(*image\_uri: str*) → *str*

Get verbose image tag from image uri.

**Parameters**

**image\_uri** (*str*) – Image uri to get tag for.

**Returns**

*str* – Verbose image tag for given image.

**braket.aws.queue\_information module**

**class** *braket.aws.queue\_information.QueueType*(*value*)

Bases: *str, Enum*

Enumerates the possible priorities for the queue.

**Values:**

NORMAL: Represents normal queue for the device. PRIORITY: Represents priority queue for the device.

**NORMAL** = 'Normal'

**PRIORITY** = 'Priority'

**class** *braket.aws.queue\_information.QueueDepthInfo*(*quantum\_tasks: dict[QueueType, str], jobs: str*)

Bases: *object*

Represents quantum tasks and hybrid jobs queue depth information.

**quantum\_tasks**

number of quantum tasks waiting to run on a device. This includes both 'Normal' and 'Priority' tasks. For Example, {'quantum\_tasks': {QueueType.NORMAL: '7', QueueType.PRIORITY: '3'}}

**Type**

*dict[QueueType, str]*

**jobs**

number of hybrid jobs waiting to run on a device. Additionally, for QPUs if hybrid jobs queue depth is 0, we display information about priority and count of the running hybrid jobs. Example, 'jobs': '0 (1 prioritized job(s) running)'

**Type**

str

**quantum\_tasks:** dict[QueueType, str]

**jobs:** str

```
class braket.aws.queue_information.QuantumTaskQueueInfo(queue_type: QueueType, queue_position: str | None = None, message: str | None = None)
```

Bases: object

Represents quantum tasks queue information.

**queue\_type**

type of the quantum\_task queue either 'Normal' or 'Priority'.

**Type**

QueueType

**queue\_position**

current position of your quantum task within a respective device queue. This value can be None based on the state of the task. Default: None.

**Type**

Optional[str]

**message**

Additional message information. This key is present only if 'queue\_position' is None. Default: None.

**Type**

Optional[str]

**queue\_type:** QueueType

**queue\_position:** str | None = None

**message:** str | None = None

```
class braket.aws.queue_information.HybridJobQueueInfo(queue_position: str | None = None, message: str | None = None)
```

Bases: object

Represents hybrid job queue information.

**queue\_position**

current position of your hybrid job within a respective device queue. If the queue position of the hybrid job is greater than 15, we return '>15' as the queue\_position return value. The queue\_position is only returned when hybrid job is not in RUNNING/CANCELLING/TERMINAL states, else queue\_position is returned as None.

**Type**

Optional[str]

**message**

Additional message information. This key is present only if 'queue\_position' is None. Default: None.

**Type**

Optional[str]

**queue\_position:** str | None = None

**message:** str | None = None

**braket.circuits package****Subpackages****braket.circuits.noise\_model package****Submodules****braket.circuits.noise\_model.circuit\_instruction\_criteria module**

**class** braket.circuits.noise\_model.circuit\_instruction\_criteria.CircuitInstructionCriteria

Bases: [Criteria](#)

Criteria that implement these methods may be used to determine gate noise.

**abstract instruction\_matches**(*instruction*: [Instruction](#)) → bool

Returns True if an Instruction matches the criteria.

**Parameters**

**instruction** ([Instruction](#)) – An Instruction to match.

**Raises**

**NotImplementedError** – Not implemented.

**Returns**

*bool* – True if an Instruction matches the criteria.

**braket.circuits.noise\_model.criteria module**

**class** braket.circuits.noise\_model.criteria.CriteriaKey(*value*)

Bases: str, Enum

Specifies the types of keys that a criteria may use to match an instruction, observable, etc.

**QUBIT** = 'QUBIT'

**GATE** = 'GATE'

**UNITARY\_GATE** = 'UNITARY\_GATE'

**OBSERVABLE** = 'OBSERVABLE'

**class** `braket.circuits.noise_model.criteria.CriteriaKeyResult(value)`

Bases: `str`, `Enum`

The `get_keys()` method may return this enum instead of actual keys for a given criteria key type.

**ALL** = 'ALL'

**class** `braket.circuits.noise_model.criteria.Criteria`

Bases: `ABC`

Represents conditions that need to be met for a noise to apply to a circuit.

**abstract** `applicable_key_types()` → `Iterable[CriteriaKey]`

Returns the relevant set of keys for the Criteria

This informs what the Criteria operates on and can be used to optimize which Criteria is relevant.

**Returns**

*Iterable[CriteriaKey]* – The relevant set of keys for the Criteria.

**abstract** `get_keys(key_type: CriteriaKey)` → *CriteriaKeyResult* | `set[Any]`

Returns a set of key for a given key type.

**Parameters**

**key\_type** (*CriteriaKey*) – The criteria key type.

**Returns**

*Union[CriteriaKeyResult, set[Any]]* – Returns a set of keys for a key type. The actual returned keys will depend on the CriteriaKey. If the provided key type is not relevant the returned list will be empty. If the provided key type is relevant for all possible inputs, the string `CriteriaKeyResult.ALL` will be returned.

**abstract** `to_dict()` → `dict`

Converts this Criteria object into a dict representation

**Returns**

*dict* – A dictionary object representing the Criteria.

**classmethod** `from_dict(criteria: dict)` → *Criteria*

Converts a dictionary representing an object of this class into an instance of this class.

**Parameters**

**criteria** (*dict*) – A dictionary representation of an object of this class.

**Returns**

*Criteria* – An object of this class that corresponds to the passed in dictionary.

**classmethod** `register_criteria(criteria: type[Criteria])` → `None`

Register a criteria implementation by adding it into the Criteria class.

**Parameters**

**criteria** (*type[Criteria]*) – Criteria class to register.

**class** `GateCriteria(gates: Gate | Iterable[Gate] | None = None, qubits: Qubit | int | Iterable[Qubit | int] | None = None)`

Bases: *CircuitInstructionCriteria*

This class models noise Criteria based on named Braket SDK Gates.

Creates Gate-based Criteria. See `instruction_matches()` for more details.

**Parameters**

- **gates** (*Optional[Union[Gate, Iterable[Gate]]]*) – A set of relevant Gates. All the Gates must have the same `fixed_qubit_count()`. Optional. If gates are not provided this matcher will match on all gates.
- **qubits** (*Optional[QubitSetInput]*) – A set of relevant qubits. If no qubits are provided, all (possible) qubits are considered to be relevant.

#### Raises

- **ValueError** – If the gates don't all operate on the same number of qubits, or if
- **qubits are not valid targets for the provided gates.** –

**applicable\_key\_types()** → *Iterable[CriteriaKey]*

Returns an Iterable of criteria keys.

#### Returns

*Iterable[CriteriaKey]* – This Criteria operates on Gates and Qubits.

**classmethod from\_dict(criteria: dict)** → *Criteria*

Deserializes a dictionary into a Criteria object.

#### Parameters

**criteria** (*dict*) – A dictionary representation of a GateCriteria.

#### Returns

*Criteria* – A deserialized GateCriteria represented by the passed in serialized data.

**get\_keys(key\_type: CriteriaKey)** → *CriteriaKeyResult* | *set[Any]*

Gets the keys for a given CriteriaKey.

#### Parameters

**key\_type** (*CriteriaKey*) – The relevant Criteria Key.

#### Returns

*Union[CriteriaKeyResult, set[Any]]* – The return value is based on the key type: GATE will return a set of Gate classes that are relevant to this Criteria. QUBIT will return a set of qubit targets that are relevant to this Criteria, or *CriteriaKeyResult.ALL* if the Criteria is relevant for all (possible) qubits. All other keys will return an empty list.

**instruction\_matches(instruction: Instruction)** → *bool*

Returns true if an Instruction matches the criteria.

#### Parameters

**instruction** (*Instruction*) – An Instruction to match.

#### Returns

*bool* – Returns true if the operator is one of the Gates provided in the constructor and the target is a qubit (or set of qubits) provided in the constructor. If gates were not provided in the constructor, then this method will accept any Gate. If qubits were not provided in the constructor, then this method will accept any Instruction target.

**to\_dict()** → *dict*

Converts a dictionary representing an object of this class into an instance of this class.

#### Returns

*dict* – A dictionary representing the serialized version of this Criteria.

**class ObservableCriteria(observables: Observable | Iterable[Observable] | None = None, qubits: Qubit | int | Iterable[Qubit | int] | None = None)**

Bases: *ResultTypeCriteria*

This class models noise Criteria based on the Braket SDK Observable classes.

Creates Observable-based Criteria. See `instruction_matches()` for more details.

#### Parameters

- **observables** (*Optional[Union[Observable, Iterable[Observable]]]*) – A set of relevant Observables. Observables must only operate on a single qubit. Optional. If observables are not specified, this criteria will match on any observable.
- **qubits** (*Optional[QubitSetInput]*) – A set of relevant qubits. If no qubits are provided, all (possible) qubits are considered to be relevant.

**Throws:**

ValueError: If the operators operate on more than one qubit.

**applicable\_key\_types()** → Iterable[CriteriaKey]

Returns an Iterable of criteria keys.

**Returns**

Iterable[CriteriaKey] – This Criteria operates on Observables and Qubits.

**classmethod from\_dict(criteria: dict)** → Criteria

Deserializes a dictionary into a Criteria object.

**Parameters**

**criteria** (dict) – A dictionary representation of a GateCriteria.

**Returns**

Criteria – A deserialized GateCriteria represented by the passed in serialized data.

**get\_keys(key\_type: CriteriaKey)** → CriteriaKeyResult | set[Any]

Gets the keys for a given CriteriaKey.

**Parameters**

**key\_type** (CriteriaKey) – The relevant Criteria Key.

**Returns**

Union[CriteriaKeyResult, set[Any]] – The return value is based on the key type: OBSERVABLE will return a set of Observable classes that are relevant to this Criteria, or CriteriaKeyResult.ALL if the Criteria is relevant for all (possible) observables. QUBIT will return a set of qubit targets that are relevant to this Criteria, or CriteriaKeyResult.ALL if the Criteria is relevant for all (possible) qubits. All other keys will return an empty set.

**result\_type\_matches(result\_type: ResultType)** → bool

Returns true if a result type matches the criteria.

**Parameters**

**result\_type** (ResultType) – A result type or list of result types to match.

**Returns**

bool – Returns true if the result type is one of the Observables provided in the constructor and the target is a qubit (or set of qubits) provided in the constructor. If observables were not provided in the constructor, then this method will accept any Observable. If qubits were not provided in the constructor, then this method will accept any result type target.

**to\_dict()** → dict

Converts a dictionary representing an object of this class into an instance of this class.

**Returns**

dict – A dictionary representing the serialized version of this Criteria.

**class QubitInitializationCriteria(qubits: Qubit | int | Iterable[Qubit | int] | None = None)**

Bases: InitializationCriteria

This class models initialization noise Criteria based on qubits.

Creates initialization noise Qubit-based Criteria.

**Parameters**

**qubits** (*Optional*[*QubitSetInput*]) – A set of relevant qubits. If no qubits are provided, all (possible) qubits are considered to be relevant.

**applicable\_key\_types()** → *Iterable*[*CriteriaKey*]

Gets the QUBIT criteria key.

**Returns**

*Iterable*[*CriteriaKey*] – This Criteria operates on Qubits, but is valid for all Gates.

**classmethod from\_dict**(*criteria: dict*) → *Criteria*

Deserializes a dictionary into a Criteria object.

**Parameters**

**criteria** (*dict*) – A dictionary representation of a QubitCriteria.

**Returns**

*Criteria* – A deserialized QubitCriteria represented by the passed in serialized data.

**get\_keys**(*key\_type: CriteriaKey*) → *CriteriaKeyResult* | *set*[*Any*]

Gets the keys for a given CriteriaKey.

**Parameters**

**key\_type** (*CriteriaKey*) – The relevant Criteria Key.

**Returns**

*Union*[*CriteriaKeyResult*, *set*[*Any*]] – The return value is based on the key type: QUBIT will return a set of qubit targets that are relevant to this Criteria, or *CriteriaKeyResult.ALL* if the Criteria is relevant for all (possible) qubits. All other keys will return an empty set.

**qubit\_intersection**(*qubits: Qubit* | *int* | *Iterable*[*Qubit* | *int*]) → *Qubit* | *int* | *Iterable*[*Qubit* | *int*]

Returns subset of passed qubits that match the criteria.

**Parameters**

**qubits** (*QubitSetInput*) – A qubit or set of qubits that may match the criteria.

**Returns**

*QubitSetInput* – The subset of passed qubits that match the criteria.

**to\_dict()** → *dict*

Converts a dictionary representing an object of this class into an instance of this class.

**Returns**

*dict* – A dictionary representing the serialized version of this Criteria.

**class UnitaryGateCriteria**(*unitary: Unitary*, *qubits: Qubit* | *int* | *Iterable*[*Qubit* | *int*] | *None* = *None*)

Bases: *CircuitInstructionCriteria*

This class models noise Criteria based on unitary gates represented as a matrix.

Creates unitary gate-based Criteria. See *instruction\_matches()* for more details.

**Parameters**

- **unitary** (*Unitary*) – A unitary gate matrix represented as a Braket Unitary.
- **qubits** (*Optional*[*QubitSetInput*]) – A set of relevant qubits. If no qubits are provided, all (possible) qubits are considered to be relevant.

**Raises**

**ValueError** – If unitary is not a Unitary type.

**applicable\_key\_types()** → *Iterable*[*CriteriaKey*]

Returns keys based on criterion.

**Returns**

*Iterable*[*CriteriaKey*] – This Criteria operates on unitary gates and Qubits.

**classmethod** `from_dict(criteria: dict) → Criteria`

Deserializes a dictionary into a Criteria object.

**Parameters**

**criteria** (*dict*) – A dictionary representation of a UnitaryGateCriteria.

**Returns**

*Criteria* – A deserialized UnitaryGateCriteria represented by the passed in serialized data.

**get\_keys**(*key\_type: CriteriaKey*) → *CriteriaKeyResult* | set[Any]

Gets the keys for a given CriteriaKey.

**Parameters**

**key\_type** (*CriteriaKey*) – The relevant Criteria Key.

**Returns**

*Union[CriteriaKeyResult, set[Any]]* – The return value is based on the key type: UNITARY\_GATE will return a set containing the bytes of the unitary matrix representing the unitary gate. QUBIT will return a set of qubit targets that are relevant to this Criteria, or CriteriaKeyResult.ALL if the Criteria is relevant for all (possible) qubits. All other keys will return an empty list.

**instruction\_matches**(*instruction: Instruction*) → bool

Returns true if an Instruction matches the criteria.

**Parameters**

**instruction** (*Instruction*) – An Instruction to match.

**Returns**

*bool* – Returns true if the operator is one of the Unitary gates provided in the constructor and the target is a qubit (or set of qubits) provided in the constructor. If qubits were not provided in the constructor, then this method will ignore the Instruction target.

**to\_dict**() → dict

Converts a dictionary representing an object of this class into an instance of this class.

**Returns**

*dict* – A dictionary representing the serialized version of this Criteria.

## braket.circuits.noise\_model.criteria\_input\_parsing module

`braket.circuits.noise_model.criteria_input_parsing.parse_operator_input`(*operators: QuantumOperator | Iterable[QuantumOperator]*) → set[QuantumOperator] | None

Processes the quantum operator input to `__init__` to validate and return a set of QuantumOperators.

**Parameters**

**operators** (*Union[QuantumOperator, Iterable[QuantumOperator]]*) – QuantumOperator input.

**Returns**

*Optional[set[QuantumOperator]]* – The set of relevant QuantumOperators or None if none is specified.

**Throws:**

**ValueError:** If no quantum operator are provided, if the quantum operator don't all operate on the same number of qubits.



```
braket.circuits.noise_model.criteria_input_parsing.parse_qubit_input(qubits: Qubit | int |
                                                                    Iterable[Qubit | int] |
                                                                    None,
                                                                    expected_qubit_count: int
                                                                    | None = 0) → set[int |
                                                                    tuple[int]] | None
```

Processes the qubit input to `__init__` to validate and return a set of qubit targets.

#### Parameters

- **qubits** (*Optional*[*QubitSetInput*]) – Qubit input.
- **expected\_qubit\_count** (*Optional*[*int*]) – The expected number of qubits that the input gates operates on. If the value is non-zero, this method will validate that the expected qubit count matches the actual qubit count. Default is 0.

#### Returns

*Optional*[*set*[*Union*[*int*, *tuple*[*int*]]]] – The set of qubit targets, or None if no qubits are specified.

### braket.circuits.noise\_model.gate\_criteria module

```
class braket.circuits.noise_model.gate_criteria.GateCriteria(gates: Gate | Iterable[Gate] | None =
                                                            None, qubits: Qubit | int |
                                                            Iterable[Qubit | int] | None = None)
```

Bases: *CircuitInstructionCriteria*

This class models noise Criteria based on named Braket SDK Gates.

Creates Gate-based Criteria. See `instruction_matches()` for more details.

#### Parameters

- **gates** (*Optional*[*Union*[*Gate*, *Iterable*[*Gate*]]]) – A set of relevant Gates. All the Gates must have the same `fixed_qubit_count()`. Optional. If gates are not provided this matcher will match on all gates.
- **qubits** (*Optional*[*QubitSetInput*]) – A set of relevant qubits. If no qubits are provided, all (possible) qubits are considered to be relevant.

#### Raises

- **ValueError** – If the gates don't all operate on the same number of qubits, or if
- **qubits are not valid targets for the provided gates.** –

```
applicable_key_types() → Iterable[CriteriaKey]
```

Returns an Iterable of criteria keys.

#### Returns

*Iterable*[*CriteriaKey*] – This Criteria operates on Gates and Qubits.

```
get_keys(key_type: CriteriaKey) → CriteriaKeyResult | set[Any]
```

Gets the keys for a given CriteriaKey.

#### Parameters

**key\_type** (*CriteriaKey*) – The relevant Criteria Key.

#### Returns

*Union*[*CriteriaKeyResult*, *set*[*Any*]] – The return value is based on the key type: GATE will return a set of Gate classes that are relevant to this Criteria. QUBIT will return a set of qubit

targets that are relevant to this Criteria, or CriteriaKeyResult.ALL if the Criteria is relevant for all (possible) qubits. All other keys will return an empty list.

**to\_dict()** → dict

Converts a dictionary representing an object of this class into an instance of this class.

**Returns**

*dict* – A dictionary representing the serialized version of this Criteria.

**instruction\_matches**(*instruction*: [Instruction](#)) → bool

Returns true if an Instruction matches the criteria.

**Parameters**

**instruction** ([Instruction](#)) – An Instruction to match.

**Returns**

*bool* – Returns true if the operator is one of the Gates provided in the constructor and the target is a qubit (or set of qubits) provided in the constructor. If gates were not provided in the constructor, then this method will accept any Gate. If qubits were not provided in the constructor, then this method will accept any Instruction target.

**classmethod from\_dict**(*criteria*: dict) → [Criteria](#)

Deserializes a dictionary into a Criteria object.

**Parameters**

**criteria** (dict) – A dictionary representation of a GateCriteria.

**Returns**

*Criteria* – A deserialized GateCriteria represented by the passed in serialized data.

## **braket.circuits.noise\_model.initialization\_criteria module**

**class** `braket.circuits.noise_model.initialization_criteria.InitializationCriteria`

Bases: [Criteria](#)

Criteria that implement these methods may be used to determine initialization noise.

**abstract** **qubit\_intersection**(*qubits*: [Qubit](#) | int | Iterable[[Qubit](#) | int]) → [Qubit](#) | int | Iterable[[Qubit](#) | int]

Returns subset of passed qubits that match the criteria.

**Parameters**

**qubits** ([QubitSetInput](#)) – A qubit or set of qubits that may match the criteria.

**Returns**

[QubitSetInput](#) – The subset of passed qubits that match the criteria.

## **braket.circuits.noise\_model.noise\_model module**

**class** `braket.circuits.noise_model.noise_model.NoiseModelInstruction`(*noise*: [Noise](#), *criteria*: [Criteria](#))

Bases: object

Represents a single instruction for a Noise Model.

**noise**: [Noise](#)

**criteria:** *Criteria*

**to\_dict()** → dict

Converts this object to a dictionary.

**classmethod from\_dict**(*noise\_model\_item: dict*) → *NoiseModelInstruction*

Converts a dictionary representing an object of this class into an instance of this class.

**Parameters**

**noise\_model\_item** (*dict*) – A dictionary representation of an object of this class.

**Returns**

*NoiseModelInstruction* – An object of this class that corresponds to the passed in dictionary.

```
class braket.circuits.noise_model.noise_model.NoiseModelInstructions(initialization_noise:
                                                                    list[NoiseModelInstruction],
                                                                    gate_noise:
                                                                    list[NoiseModelInstruction],
                                                                    readout_noise:
                                                                    list[NoiseModelInstruction])
```

Bases: object

Represents the instructions in a noise model, separated by type.

**initialization\_noise:** list[*NoiseModelInstruction*]

**gate\_noise:** list[*NoiseModelInstruction*]

**readout\_noise:** list[*NoiseModelInstruction*]

```
class braket.circuits.noise_model.noise_model.NoiseModel(instructions: list[NoiseModelInstruction]
                                                         | None = None)
```

Bases: object

A Noise Model can be thought of as a set of Noise objects, and a corresponding set of criteria for how each Noise object should be applied to a circuit. For example, a noise model may represent that every H gate that acts on qubit 0 has a 10% probability of a bit flip, and every X gate that acts on qubit 0 has a 20% probability of a bit flip, and 5% probability of a phase flip.

**property instructions:** list[*NoiseModelInstruction*]

List all the noise in the NoiseModel.

**Returns**

list[*NoiseModelInstruction*] – The noise model instructions.

**add\_noise**(*noise: Noise, criteria: Criteria*) → *NoiseModel*

Modifies the NoiseModel to add noise with a given Criteria.

**Parameters**

- **noise** (*Noise*) – The noise to add.
- **criteria** (*Criteria*) – The criteria that determines when the noise will be applied.

**Returns**

*NoiseModel* – This NoiseModel object.

**insert\_noise**(*index: int, noise: Noise, criteria: Criteria*) → *NoiseModel*

Modifies the NoiseModel to insert a noise with a given Criteria at particular position.

**Parameters**

- **index** (*int*) – The index at which to insert.
- **noise** (*Noise*) – The noise to insert.
- **criteria** (*Criteria*) – The criteria that determines when the noise will be applied.

**Returns**

*NoiseModel* – This NoiseModel object.

**remove\_noise**(*index: int*) → *NoiseModel*

Removes the noise and corresponding criteria from the NoiseModel at the given index.

**Parameters**

**index** (*int*) – The index of the instruction to remove.

**Returns**

*NoiseModel* – This NoiseModel object.

**Throws:**

**IndexError:** If the provided index is not less than the number of noise (as returned from `items()`).

**get\_instructions\_by\_type**() → *NoiseModelInstructions*

Returns the noise in this model by noise type.

**Returns**

*NoiseModelInstructions* – The noise model instructions.

**from\_filter**(*qubit: QubitSetInput | None = None, gate: Gate | None = None, noise: type[Noise] | None = None*) → *NoiseModel*

Returns a new NoiseModel from this NoiseModel using a given filter. If no filters are specified, the returned NoiseModel will be the same as this one.

**Parameters**

- **qubit** (*Optional[QubitSetInput]*) – The qubit to filter. Default is None. If not None, the returned NoiseModel will only have Noise that might be applicable to the passed qubit (or qubit list, if the criteria acts on a multi-qubit Gate).
- **gate** (*Optional[Gate]*) – The gate to filter. Default is None. If not None, the returned NoiseModel will only have Noise that might be applicable to the passed Gate.
- **noise** (*Optional[type[Noise]]*) – The noise class to filter. Default is None. If not None, the returned NoiseModel will only have noise that is of the same class as the given noise class.

**Returns**

*NoiseModel* – A noise model containing Noise and Criteria that are applicable for the given filter.

**apply**(*circuit: Circuit*) → *Circuit*

Applies this noise model to a circuit, and returns a new circuit that's the noisy version of the given circuit. If multiple noise will act on the same instruction, they will be applied in the order they are added to the noise model.

**Parameters**

**circuit** (*Circuit*) – a circuit to apply noise to.

**Returns**

*Circuit* – A new circuit that's a noisy version of the passed in circuit.

**to\_dict()** → dict

Converts this object to a dictionary.

**classmethod from\_dict**(noise\_dict: dict) → *NoiseModel*

Converts a dictionary representing an object of this class into an instance of this class.

**Parameters**

**noise\_dict** (dict) – A dictionary representation of an object of this class.

**Returns**

*NoiseModel* – An object of this class that corresponds to the passed in dictionary.

## braket.circuits.noise\_model.observable\_criteria module

```
class braket.circuits.noise_model.observable_criteria.ObservableCriteria(observables:
    Observable |
    Iterable[Observable]
    | None = None,
    qubits: Qubit | int |
    Iterable[Qubit | int] |
    None = None)
```

Bases: *ResultTypeCriteria*

This class models noise Criteria based on the Braket SDK Observable classes.

Creates Observable-based Criteria. See instruction\_matches() for more details.

**Parameters**

- **observables** (*Optional[Union[Observable, Iterable[Observable]]]*) – A set of relevant Observables. Observables must only operate on a single qubit. Optional. If observables are not specified, this criteria will match on any observable.
- **qubits** (*Optional[QubitSetInput]*) – A set of relevant qubits. If no qubits are provided, all (possible) qubits are considered to be relevant.

**Throws:**

ValueError: If the operators operate on more than one qubit.

**applicable\_key\_types()** → Iterable[*CriteriaKey*]

Returns an Iterable of criteria keys.

**Returns**

*Iterable[CriteriaKey]* – This Criteria operates on Observables and Qubits.

**get\_keys**(key\_type: *CriteriaKey*) → *CriteriaKeyResult* | set[Any]

Gets the keys for a given CriteriaKey.

**Parameters**

**key\_type** (*CriteriaKey*) – The relevant Criteria Key.

**Returns**

*Union[CriteriaKeyResult, set[Any]]* – The return value is based on the key type: OBSERVABLE will return a set of Observable classes that are relevant to this Criteria, or CriteriaKeyResult.ALL if the Criteria is relevant for all (possible) observables. QUBIT will return a set of qubit targets that are relevant to this Criteria, or CriteriaKeyResult.ALL if the Criteria is relevant for all (possible) qubits. All other keys will return an empty set.

**to\_dict()** → dict

Converts a dictionary representing an object of this class into an instance of this class.

**Returns**

*dict* – A dictionary representing the serialized version of this Criteria.

**result\_type\_matches**(*result\_type*: [ResultType](#)) → bool

Returns true if a result type matches the criteria.

**Parameters**

**result\_type** ([ResultType](#)) – A result type or list of result types to match.

**Returns**

*bool* – Returns true if the result type is one of the Observables provided in the constructor and the target is a qubit (or set of qubits) provided in the constructor. If observables were not provided in the constructor, then this method will accept any Observable. If qubits were not provided in the constructor, then this method will accept any result type target.

**classmethod from\_dict**(*criteria*: dict) → [Criteria](#)

Deserializes a dictionary into a Criteria object.

**Parameters**

**criteria** (*dict*) – A dictionary representation of a GateCriteria.

**Returns**

*Criteria* – A deserialized GateCriteria represented by the passed in serialized data.

## **braket.circuits.noise\_model.qubit\_initialization\_criteria module**

```
class braket.circuits.noise_model.qubit_initialization_criteria.QubitInitializationCriteria(qubits:  
                                                                                        Qubit  
                                                                                        |  
                                                                                        int  
                                                                                        |  
                                                                                        Iterable  
                                                                                        able[Qubit  
                                                                                        |  
                                                                                        int]  
                                                                                        |  
                                                                                        None  
                                                                                        =  
                                                                                        None)
```

Bases: [InitializationCriteria](#)

This class models initialization noise Criteria based on qubits.

Creates initialization noise Qubit-based Criteria.

**Parameters**

**qubits** (*Optional*[[QubitSetInput](#)]) – A set of relevant qubits. If no qubits are provided, all (possible) qubits are considered to be relevant.

**applicable\_key\_types**() → [Iterable](#)[[CriteriaKey](#)]

Gets the QUBIT criteria key.

**Returns**

*Iterable*[[CriteriaKey](#)] – This Criteria operates on Qubits, but is valid for all Gates.

**get\_keys**(*key\_type*: *CriteriaKey*) → *CriteriaKeyResult* | set[Any]

Gets the keys for a given *CriteriaKey*.

**Parameters**

**key\_type** (*CriteriaKey*) – The relevant *Criteria Key*.

**Returns**

*Union[CriteriaKeyResult, set[Any]]* – The return value is based on the key type: QUBIT will return a set of qubit targets that are relevant to this *Criteria*, or *CriteriaKeyResult.ALL* if the *Criteria* is relevant for all (possible) qubits. All other keys will return an empty set.

**to\_dict**() → dict

Converts a dictionary representing an object of this class into an instance of this class.

**Returns**

*dict* – A dictionary representing the serialized version of this *Criteria*.

**qubit\_intersection**(*qubits*: *Qubit* | int | Iterable[*Qubit* | int]) → *Qubit* | int | Iterable[*Qubit* | int]

Returns subset of passed qubits that match the criteria.

**Parameters**

**qubits** (*QubitSetInput*) – A qubit or set of qubits that may match the criteria.

**Returns**

*QubitSetInput* – The subset of passed qubits that match the criteria.

**classmethod from\_dict**(*criteria*: dict) → *Criteria*

Deserializes a dictionary into a *Criteria* object.

**Parameters**

**criteria** (*dict*) – A dictionary representation of a *QubitCriteria*.

**Returns**

*Criteria* – A deserialized *QubitCriteria* represented by the passed in serialized data.

## braket.circuits.noise\_model.result\_type\_criteria module

**class** braket.circuits.noise\_model.result\_type\_criteria.**ResultTypeCriteria**

Bases: *Criteria*

*Criteria* that implement these methods may be used to determine readout noise.

**abstract result\_type\_matches**(*result\_type*: *ResultType*) → bool

Returns true if a result type matches the criteria.

**Parameters**

**result\_type** (*ResultType*) – A result type or list of result types to match.

**Returns**

*bool* – True if the result type matches the criteria.

**braket.circuits.noise\_model.unitary\_gate\_criteria module**

```
class braket.circuits.noise_model.unitary_gate_criteria.UnitaryGateCriteria(unitary: Unitary,  
                                                                           qubits: Qubit |  
                                                                           int |  
                                                                           Iterable[Qubit |  
                                                                           int] | None =  
                                                                           None)
```

Bases: [CircuitInstructionCriteria](#)

This class models noise Criteria based on unitary gates represented as a matrix.

Creates unitary gate-based Criteria. See `instruction_matches()` for more details.

**Parameters**

- **unitary** ([Unitary](#)) – A unitary gate matrix represented as a Braket Unitary.
- **qubits** (*Optional[QubitSetInput]*) – A set of relevant qubits. If no qubits are provided, all (possible) qubits are considered to be relevant.

**Raises**

**ValueError** – If unitary is not a Unitary type.

**applicable\_key\_types()** → [Iterable\[CriteriaKey\]](#)

Returns keys based on criterion.

**Returns**

[Iterable\[CriteriaKey\]](#) – This Criteria operates on unitary gates and Qubits.

**get\_keys**(*key\_type: CriteriaKey*) → [CriteriaKeyResult](#) | [set\[Any\]](#)

Gets the keys for a given CriteriaKey.

**Parameters**

**key\_type** ([CriteriaKey](#)) – The relevant Criteria Key.

**Returns**

[Union\[CriteriaKeyResult, set\[Any\]\]](#) – The return value is based on the key type: UNITARY\_GATE will return a set containing the bytes of the unitary matrix representing the unitary gate. QUBIT will return a set of qubit targets that are relevant to this Criteria, or [CriteriaKeyResult.ALL](#) if the Criteria is relevant for all (possible) qubits. All other keys will return an empty list.

**to\_dict()** → [dict](#)

Converts a dictionary representing an object of this class into an instance of this class.

**Returns**

[dict](#) – A dictionary representing the serialized version of this Criteria.

**instruction\_matches**(*instruction: Instruction*) → [bool](#)

Returns true if an Instruction matches the criteria.

**Parameters**

**instruction** ([Instruction](#)) – An Instruction to match.

**Returns**

[bool](#) – Returns true if the operator is one of the Unitary gates provided in the constructor and the target is a qubit (or set of qubits) provided in the constructor. If qubits were not provided in the constructor, then this method will ignore the Instruction target.



**classmethod** `from_dict(criteria: dict) → Criteria`

Deserializes a dictionary into a Criteria object.

**Parameters**

**criteria** (*dict*) – A dictionary representation of a UnitaryGateCriteria.

**Returns**

*Criteria* – A deserialized UnitaryGateCriteria represented by the passed in serialized data.

## braket.circuits.text\_diagram\_builders namespace

### Submodules

#### braket.circuits.text\_diagram\_builders.ascii\_circuit\_diagram module

**class** `braket.circuits.text_diagram_builders.ascii_circuit_diagram.AsciiCircuitDiagram`

Bases: *TextCircuitDiagram*

Builds ASCII string circuit diagrams.

**static** `build_diagram(circuit: Circuit) → str`

Build a text circuit diagram.

**Parameters**

**circuit** (*Circuit*) – Circuit for which to build a diagram.

**Returns**

*str* – string circuit diagram.

#### braket.circuits.text\_diagram\_builders.text\_circuit\_diagram module

**class** `braket.circuits.text_diagram_builders.text_circuit_diagram.TextCircuitDiagram`

Bases: *CircuitDiagram*, ABC

Abstract base class for text circuit diagrams.

#### braket.circuits.text\_diagram\_builders.text\_circuit\_diagram\_utils module

#### braket.circuits.text\_diagram\_builders.unicode\_circuit\_diagram module

**class** `braket.circuits.text_diagram_builders.unicode_circuit_diagram.UnicodeCircuitDiagram`

Bases: *TextCircuitDiagram*

Builds string circuit diagrams using box-drawing characters.

**static** `build_diagram(circuit: Circuit) → str`

Build a text circuit diagram.

**Parameters**

**circuit** (*Circuit*) – Circuit for which to build a diagram.

**Returns**

*str* – string circuit diagram.

## Submodules

### braket.circuits.angled\_gate module

**class** `braket.circuits.angled_gate.AngledGate`(*angle*: `FreeParameterExpression` | `float`, *qubit\_count*: `int` | `None`, *ascii\_symbols*: `Sequence[str]`)

Bases: `Gate`, `Parameterizable`

Class `AngledGate` represents a quantum gate that operates on N qubits and an angle.

Initializes an `AngledGate`.

#### Parameters

- **angle** (`Union[FreeParameterExpression, float]`) – The angle of the gate in radians or expression representation.
- **qubit\_count** (`Optional[int]`) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (`Sequence[str]`) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length != `qubit_count`, or `angle` is `None`

**property parameters:** `list[FreeParameterExpression | float]`

Returns the parameters associated with the object, either unbound free parameters or bound values.

#### Returns

`list[Union[FreeParameterExpression, float]]` – The free parameters or fixed value associated with the object.

**property angle:** `FreeParameterExpression | float`

Returns the angle of the gate

#### Returns

`Union[FreeParameterExpression, float]` – The angle of the gate in radians

**bind\_values**(*\*\*kwargs*) → `AngledGate`

Takes in parameters and attempts to assign them to values.

#### Returns

`AngledGate` – A new Gate of the same type with the requested parameters bound.

#### Raises

**NotImplementedError** – Subclasses should implement this function.

**adjoint**() → `list[Gate]`

Returns the adjoint of this gate as a singleton list.

#### Returns

`list[Gate]` – A list containing the gate with negated angle.

```
class braket.circuits.angled_gate.DoubleAngledGate(angle_1: FreeParameterExpression | float,
                                                    angle_2: FreeParameterExpression | float,
                                                    qubit_count: int | None, ascii_symbols:
                                                    Sequence[str])
```

Bases: *Gate*, *Parameterizable*

Class *DoubleAngledGate* represents a quantum gate that operates on N qubits and two angles.

Initiates a *DoubleAngledGate*.

#### Parameters

- **angle\_1** (*Union*[*FreeParameterExpression*, *float*]) – The first angle of the gate in radians or expression representation.
- **angle\_2** (*Union*[*FreeParameterExpression*, *float*]) – The second angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as *qubit\_count*, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – If *qubit\_count* is less than 1, *ascii\_symbols* are None, or *ascii\_symbols* length != *qubit\_count*, or *angle\_1* or *angle\_2* is None

**property parameters:** *list*[*FreeParameterExpression* | *float*]

Returns the parameters associated with the object, either unbound free parameters or bound values.

#### Returns

*list*[*Union*[*FreeParameterExpression*, *float*]] – The free parameters or fixed value associated with the object.

**property angle\_1:** *FreeParameterExpression* | *float*

Returns the first angle of the gate

#### Returns

*Union*[*FreeParameterExpression*, *float*] – The first angle of the gate in radians

**property angle\_2:** *FreeParameterExpression* | *float*

Returns the second angle of the gate

#### Returns

*Union*[*FreeParameterExpression*, *float*] – The second angle of the gate in radians

**bind\_values**(*\*\*kwargs*: *FreeParameterExpression* | *str*) → *AngledGate*

Takes in parameters and attempts to assign them to values.

#### Parameters

**\*\*kwargs** (*FreeParameterExpression* | *str*) – The parameters that are being assigned.

#### Returns

*AngledGate* – A new Gate of the same type with the requested parameters bound.

#### Raises

**NotImplementedError** – Subclasses should implement this function.

**adjoint()** → list[*Gate*]

Returns the adjoint of this gate as a singleton list.

**Returns**

list[*Gate*] – A list containing the gate with negated angle.

```
class braket.circuits.angled_gate.TripleAngledGate(angle_1: FreeParameterExpression | float,
                                                    angle_2: FreeParameterExpression | float,
                                                    angle_3: FreeParameterExpression | float,
                                                    qubit_count: int | None, ascii_symbols:
                                                    Sequence[str])
```

Bases: *Gate*, *Parameterizable*

Class *TripleAngledGate* represents a quantum gate that operates on N qubits and three angles.

Initiates a *TripleAngledGate*.

**Parameters**

- **angle\_1** (Union[FreeParameterExpression, float]) – The first angle of the gate in radians or expression representation.
- **angle\_2** (Union[FreeParameterExpression, float]) – The second angle of the gate in radians or expression representation.
- **angle\_3** (Union[FreeParameterExpression, float]) – The third angle of the gate in radians or expression representation.
- **qubit\_count** (Optional[int]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (Sequence[str]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as qubit\_count, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If qubit\_count is less than 1, ascii\_symbols are None, or ascii\_symbols length != qubit\_count, or *angle\_1* or *angle\_2* or *angle\_3* is None

**property parameters:** list[FreeParameterExpression | float]

Returns the parameters associated with the object, either unbound free parameters or bound values.

**Returns**

list[Union[FreeParameterExpression, float]] – The free parameters or fixed value associated with the object.

**property angle\_1:** FreeParameterExpression | float

Returns the first angle of the gate

**Returns**

Union[FreeParameterExpression, float] – The first angle of the gate in radians

**property angle\_2:** FreeParameterExpression | float

Returns the second angle of the gate

**Returns**

Union[FreeParameterExpression, float] – The second angle of the gate in radians

**property angle\_3:** *FreeParameterExpression* | float

Returns the third angle of the gate

**Returns**

*Union[FreeParameterExpression, float]* – The third angle of the gate in radians

**bind\_values**(\*\*kwargs: *FreeParameterExpression* | *str*) → *AngledGate*

Takes in parameters and attempts to assign them to values.

**Parameters**

**\*\*kwargs** (*FreeParameterExpression* | *str*) – The parameters that are being assigned.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**adjoint**() → list[*Gate*]

Returns the adjoint of this gate as a singleton list.

**Returns**

list[*Gate*] – A list containing the gate with negated angle.

**braket.circuits.angled\_gate.angled\_ascii\_characters**(gate: *str*, angle: *FreeParameterExpression* | float) → *str*

Generates a formatted ascii representation of an angled gate.

**Parameters**

- **gate** (*str*) – The name of the gate.
- **angle** (*Union[FreeParameterExpression, float]*) – The angle for the gate.

**Returns**

*str* – Returns the ascii representation for an angled gate.

**braket.circuits.angled\_gate.get\_angle**(gate: *AngledGate*, \*\*kwargs: *FreeParameterExpression* | *str*) → *AngledGate*

Gets the angle with all values substituted in that are requested.

**Parameters**

- **gate** (*AngledGate*) – The subclass of AngledGate for which the angle is being obtained.
- **\*\*kwargs** (*FreeParameterExpression* | *str*) – The named parameters that are being filled for a particular gate.

**Returns**

*AngledGate* – A new gate of the type of the AngledGate originally used with all angles updated.

**braket.circuits.ascii\_circuit\_diagram module****braket.circuits.basis\_state module**

```
class braket.circuits.basis_state.BasisState(state: int | list[int] | str | BasisState, size: int | None = None)
```

Bases: object

**property** size: int

**property** as\_tuple: tuple

**property** as\_int: int

**property** as\_string: str

**braket.circuits.braket\_program\_context module**

```
class braket.circuits.braket_program_context.BraketProgramContext(circuit: Circuit | None = None)
```

Bases: AbstractProgramContext

Initiates a *BraketProgramContext*.

**Parameters**

**circuit** (*Optional*[*Circuit*]) – A partially-built circuit to continue building with this context. Default: None.

**property circuit:** *Circuit*

The circuit being built in this context.

**is\_builtin\_gate**(name: str) → bool

Whether the gate is currently in scope as a built-in Braket gate.

**Parameters**

**name** (str) – name of the built-in Braket gate

**Returns**

bool – return TRUE if it is a built-in gate else FALSE.

**add\_phase\_instruction**(target: tuple[int], phase\_value: float) → None

Add a global phase to the circuit.

**Parameters**

- **target** (tuple[int]) – Unused
- **phase\_value** (float) – The phase value to be applied

**add\_gate\_instruction**(gate\_name: str, target: tuple[int], \*params, ctrl\_modifiers: list[int], power: float) → None

Add Braket gate to the circuit.

**Parameters**

- **gate\_name** (str) – name of the built-in Braket gate.
- **target** (tuple[int]) – control\_qubits + target\_qubits.

- **ctrl\_modifiers** (*list[int]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**-qubits in target. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state.
- **power** (*float*) – Integer or fractional power to raise the gate to.

**add\_custom\_unitary**(*unitary: ndarray, target: tuple[int]*) → None

Add a custom Unitary instruction to the circuit

#### Parameters

- **unitary** (*np.ndarray*) – unitary matrix
- **target** (*tuple[int]*) – control\_qubits + target\_qubits

**add\_noise\_instruction**(*noise\_instruction: str, target: list[int], probabilities: list[float]*) → None

Method to add a noise instruction to the circuit

#### Parameters

- **noise\_instruction** (*str*) – The name of the noise operation
- **target** (*list[int]*) – The target qubit or qubits to which the noise operation is applied.
- **probabilities** (*list[float]*) – The probabilities associated with each possible outcome of the noise operation.

**add\_kraus\_instruction**(*matrices: list[ndarray], target: list[int]*) → None

Method to add a Kraus instruction to the circuit

#### Parameters

- **matrices** (*list[ndarray]*) – The matrices defining the Kraus operation
- **target** (*list[int]*) – The target qubit or qubits to which the Kraus operation is applied.

**add\_result**(*result: Amplitude | Expectation | Probability | Sample | StateVector | DensityMatrix | Variance | AdjointGradient*) → None

Abstract method to add result type to the circuit

#### Parameters

**result** (*Results*) – The result object representing the measurement results

**handle\_parameter\_value**(*value: float | Expr*) → float | *FreeParameterExpression*

Convert parameter value to required format.

#### Parameters

**value** (*Union[float, Expr]*) – Value of the parameter

#### Returns

*Union[float, FreeParameterExpression]* – Return the value directly if numeric, otherwise wraps the symbolic expression as a *FreeParameterExpression*.

**add\_measure**(*target: tuple[int]*) → None

Add a measure instruction to the circuit

#### Parameters

**target** (*tuple[int]*) – the target qubits to be measured.

**braket.circuits.circuit module**

**class** `braket.circuits.circuit.Circuit`(*addable: AddableTypes | None = None, \*args, \*\*kwargs*)

Bases: `object`

A representation of a quantum circuit that contains the instructions to be performed on a quantum device and the requested result types.

See [braket.circuits.gates](#) module for all of the supported instructions.

See [braket.circuits.result\\_types](#) module for all of the supported result types.

AddableTypes are `Instruction`, iterable of `Instruction`, `ResultType`, iterable of `ResultType`, or `SubroutineCallable`

Initiates a `Circuit`.

**Parameters**

**addable** (*AddableTypes | None*) – The item(s) to add to self. Default = `None`.

**Raises**

**TypeError** – If `addable` is an unsupported type.

**Examples**

```
>>> circ = Circuit([Instruction(Gate.H(), 4), Instruction(Gate.CNot(), [4, 5])])
>>> circ = Circuit().h(0).cnot(0, 1)
>>> circ = Circuit().h(0).cnot(0, 1).probability([0, 1])
```

```
>>> @circuit.subroutine(register=True)
>>> def bell_pair(target):
...     return Circ().h(target[0]).cnot(target[0:2])
...
>>> circ = Circuit(bell_pair, [4,5])
>>> circ = Circuit().bell_pair([4,5])
```

**classmethod** `register_subroutine`(*func: SubroutineCallable*) → `None`

Register the subroutine `func` as an attribute of the `Circuit` class. The attribute name is the name of `func`.

**Parameters**

**func** (*SubroutineCallable*) – The function of the subroutine to add to the class.

**Examples**

```
>>> def h_on_all(target):
...     circ = Circuit()
...     for qubit in target:
...         circ += Instruction(Gate.H(), qubit)
...     return circ
...
>>> Circuit.register_subroutine(h_on_all)
>>> circ = Circuit().h_on_all(range(2))
>>> for instr in circ.instructions:
...     print(instr)
```

(continues on next page)



(continued from previous page)

```
...
Instruction('operator': 'H', 'target': QubitSet(Qubit(0),))
Instruction('operator': 'H', 'target': QubitSet(Qubit(1),))
```

**property depth: int**

Get the circuit depth.

**Type**

int

**property global\_phase: float**

Get the global phase of the circuit.

**Type**

float

**property instructions: list[Instruction]**

Get an iterable of instructions in the circuit.

**Type**

Iterable[Instruction]

**property result\_types: list[ResultType]**

Get a list of requested result types in the circuit.

**Type**

list[ResultType]

**property basis\_rotation\_instructions: list[Instruction]**

Gets a list of basis rotation instructions.

**Returns***list[Instruction]* – Get a list of basis rotation instructions in the circuit. These basis rotation instructions are added if result types are requested for an observable other than Pauli-Z.

This only makes sense if all observables are simultaneously measurable; if not, this method will return an empty list.

**property moments: Moments**Get the *moments* for this circuit. Note that this includes observables.**Type***Moments***property qubit\_count: int**

Get the qubit count for this circuit. Note that this includes observables.

**Returns***int* – The qubit count for this circuit.**property qubits: QubitSet**

Get a copy of the qubits for this circuit.

**Type***QubitSet*

**property parameters:** `set[FreeParameter]`

Gets a set of the parameters in the Circuit.

#### Returns

`set[FreeParameter]` – The FreeParameters in the Circuit.

**add\_result\_type**(*result\_type*: `ResultType`, *target*: `QubitSetInput` | `None` = `None`, *target\_mapping*: `dict[QubitInput, QubitInput]` | `None` = `None`) → `Circuit`

Add a requested result type to self, returns self for chaining ability.

#### Parameters

- **result\_type** (`ResultType`) – ResultType to add into self.
- **target** (`QubitSetInput` | `None`) – Target qubits for the result\_type. Default = `None`.
- **target\_mapping** (`dict[QubitInput, QubitInput]` | `None`) – A dictionary of qubit mappings to apply to the result\_type.target. Key is the qubit in result\_type.target and the value is what the key will be changed to. Default = `None`.

#### Returns

`Circuit` – self

---

**Note:** Target and target\_mapping will only be applied to those requested result types with the attribute target. The result\_type will be appended to the end of the dict keys of `circuit.result_types` only if it does not already exist in `circuit.result_types`

---

#### Raises

- **TypeError** – If both target\_mapping and target are supplied.
- **ValueError** – If a measure instruction exists on the current circuit.

### Examples

```
>>> result_type = ResultType.Probability(target=[0, 1])
>>> circ = Circuit().add_result_type(result_type)
>>> print(circ.result_types[0])
Probability(target=QubitSet([Qubit(0), Qubit(1)]))
```

```
>>> result_type = ResultType.Probability(target=[0, 1])
>>> circ = Circuit().add_result_type(result_type, target_mapping={0: 10, 1: 11})
>>> print(circ.result_types[0])
Probability(target=QubitSet([Qubit(10), Qubit(11)]))
```

```
>>> result_type = ResultType.Probability(target=[0, 1])
>>> circ = Circuit().add_result_type(result_type, target=[10, 11])
>>> print(circ.result_types[0])
Probability(target=QubitSet([Qubit(10), Qubit(11)]))
```

```
>>> result_type = ResultType.StateVector()
>>> circ = Circuit().add_result_type(result_type)
```

(continues on next page)

(continued from previous page)

```
>>> print(circ.result_types[0])
StateVector()
```

**add\_instruction**(*instruction*: [Instruction](#), *target*: [QubitSetInput](#) | *None* = *None*, *target\_mapping*: [dict](#)[[QubitInput](#), [QubitInput](#)] | *None* = *None*) → [Circuit](#)

Add an instruction to self, returns self for chaining ability.

#### Parameters

- **instruction** ([Instruction](#)) – Instruction to add into self.
- **target** ([QubitSetInput](#) | *None*) – Target qubits for the instruction. If a single qubit gate, an instruction is created for every index in target. Default = *None*.
- **target\_mapping** ([dict](#)[[QubitInput](#), [QubitInput](#)] | *None*) – A dictionary of qubit mappings to apply to the instruction.target. Key is the qubit in instruction.target and the value is what the key will be changed to. Default = *None*.

#### Returns

[Circuit](#) – self

#### Raises

- **TypeError** – If both target\_mapping and target are supplied.
- **ValueError** – If adding a gate or noise after a measure instruction.

### Examples

```
>>> instr = Instruction(Gate.CNot(), [0, 1])
>>> circ = Circuit().add_instruction(instr)
>>> print(circ.instructions[0])
Instruction('operator': 'CNOT', 'target': QubitSet(Qubit(0), Qubit(1)))
```

```
>>> instr = Instruction(Gate.CNot(), [0, 1])
>>> circ = Circuit().add_instruction(instr, target_mapping={0: 10, 1: 11})
>>> print(circ.instructions[0])
Instruction('operator': 'CNOT', 'target': QubitSet(Qubit(10), Qubit(11)))
```

```
>>> instr = Instruction(Gate.CNot(), [0, 1])
>>> circ = Circuit().add_instruction(instr, target=[10, 11])
>>> print(circ.instructions[0])
Instruction('operator': 'CNOT', 'target': QubitSet(Qubit(10), Qubit(11)))
```

```
>>> instr = Instruction(Gate.H(), 0)
>>> circ = Circuit().add_instruction(instr, target=[10, 11])
>>> print(circ.instructions[0])
Instruction('operator': 'H', 'target': QubitSet(Qubit(10),))
>>> print(circ.instructions[1])
Instruction('operator': 'H', 'target': QubitSet(Qubit(11),))
```

**add\_circuit**(*circuit*: [Circuit](#), *target*: [QubitSetInput](#) | *None* = *None*, *target\_mapping*: [dict](#)[[QubitInput](#), [QubitInput](#)] | *None* = *None*) → [Circuit](#)

Add a [Circuit](#) to self, returning self for chaining ability.

**Parameters**

- **circuit** (*Circuit*) – Circuit to add into self.
- **target** (*QubitSetInput* / *None*) – Target qubits for the supplied circuit. This is a macro over `target_mapping`; `target` is converted to a `target_mapping` by zipping together a sorted `circuit.qubits` and `target`. Default = *None*.
- **target\_mapping** (*dict[QubitInput, QubitInput]* / *None*) – A dictionary of qubit mappings to apply to the qubits of `circuit.instructions`. Key is the qubit to map, and the value is what to change it to. Default = *None*.

**Returns**

*Circuit* – self

**Raises**

**TypeError** – If both `target_mapping` and `target` are supplied.

---

**Note:** Supplying `target` sorts `circuit.qubits` to have deterministic behavior since `circuit.qubits` ordering is based on how instructions are inserted. Use caution when using this with circuits that with a lot of qubits, as the sort can be resource-intensive. Use `target_mapping` to use a linear runtime to remap the qubits.

Requested result types of the circuit that will be added will be appended to the end of the list for the existing requested result types. A result type to be added that is equivalent to an existing requested result type will not be added.

---

**Examples**

```
>>> widget = Circuit().h(0).cnot(0, 1)
>>> circ = Circuit().add_circuit(widget)
>>> instructions = list(circ.instructions)
>>> print(instructions[0])
Instruction('operator': 'H', 'target': QubitSet(Qubit(0),))
>>> print(instructions[1])
Instruction('operator': 'CNOT', 'target': QubitSet(Qubit(0), Qubit(1)))
```

```
>>> widget = Circuit().h(0).cnot(0, 1)
>>> circ = Circuit().add_circuit(widget, target_mapping={0: 10, 1: 11})
>>> instructions = list(circ.instructions)
>>> print(instructions[0])
Instruction('operator': 'H', 'target': QubitSet(Qubit(10),))
>>> print(instructions[1])
Instruction('operator': 'CNOT', 'target': QubitSet(Qubit(10), Qubit(11)))
```

```
>>> widget = Circuit().h(0).cnot(0, 1)
>>> circ = Circuit().add_circuit(widget, target=[10, 11])
>>> instructions = list(circ.instructions)
>>> print(instructions[0])
Instruction('operator': 'H', 'target': QubitSet(Qubit(10),))
>>> print(instructions[1])
Instruction('operator': 'CNOT', 'target': QubitSet(Qubit(10), Qubit(11)))
```

**add\_verbatim\_box**(*verbatim\_circuit*: [Circuit](#), *target*: *QubitSetInput* | *None* = *None*, *target\_mapping*: *dict*[*QubitInput*, *QubitInput*] | *None* = *None*) → [Circuit](#)

Add a verbatim [Circuit](#) to self, ensuring that the circuit is not modified in any way by the compiler.

#### Parameters

- **verbatim\_circuit** ([Circuit](#)) – Circuit to add into self.
- **target** (*QubitSetInput* | *None*) – Target qubits for the supplied circuit. This is a macro over *target\_mapping*; *target* is converted to a *target\_mapping* by zipping together a sorted *circuit.qubits* and *target*. Default = *None*.
- **target\_mapping** (*dict*[*QubitInput*, *QubitInput*] | *None*) – A dictionary of qubit mappings to apply to the qubits of *circuit.instructions*. Key is the qubit to map, and the value is what to change it to. Default = *None*.

#### Returns

*Circuit* – self

#### Raises

- **TypeError** – If both *target\_mapping* and *target* are supplied.
- **ValueError** – If *circuit* has result types attached

### Examples

```
>>> widget = Circuit().h(0).h(1)
>>> circ = Circuit().add_verbatim_box(widget)
>>> print(list(circ.instructions))
[Instruction('operator': StartVerbatimBox, 'target': QubitSet([])),
 Instruction('operator': H('qubit_count': 1), 'target': QubitSet([Qubit(0)])),
 Instruction('operator': H('qubit_count': 1), 'target': QubitSet([Qubit(1)])),
 Instruction('operator': EndVerbatimBox, 'target': QubitSet([]))]
```

```
>>> widget = Circuit().h(0).cnot(0, 1)
>>> circ = Circuit().add_verbatim_box(widget, target_mapping={0: 10, 1: 11})
>>> print(list(circ.instructions))
[Instruction('operator': StartVerbatimBox, 'target': QubitSet([])),
 Instruction('operator': H('qubit_count': 1), 'target': QubitSet([Qubit(10)])),
 Instruction('operator': H('qubit_count': 1), 'target': QubitSet([Qubit(11)])),
 Instruction('operator': EndVerbatimBox, 'target': QubitSet([]))]
```

```
>>> widget = Circuit().h(0).cnot(0, 1)
>>> circ = Circuit().add_verbatim_box(widget, target=[10, 11])
>>> print(list(circ.instructions))
[Instruction('operator': StartVerbatimBox, 'target': QubitSet([])),
 Instruction('operator': H('qubit_count': 1), 'target': QubitSet([Qubit(10)])),
 Instruction('operator': H('qubit_count': 1), 'target': QubitSet([Qubit(11)])),
 Instruction('operator': EndVerbatimBox, 'target': QubitSet([]))]
```

**measure**(*target\_qubits*: *QubitSetInput*) → [Circuit](#)

Add a *measure* operator to self ensuring only the target qubits are measured.

#### Parameters

**target\_qubits** (*QubitSetInput*) – target qubits to measure.

**Returns***Circuit* – self**Raises**

- **IndexError** – If self has no qubits.
- **IndexError** – If target qubits are not within the range of the current circuit.
- **ValueError** – If the current circuit contains any result types.
- **ValueError** – If the target qubit is already measured.

**Examples**

```
>>> circ = Circuit.h(0).cnot(0, 1).measure([0])
>>> circ.print(list(circ.instructions))
[Instruction('operator': H('qubit_count': 1), 'target': QubitSet([Qubit(0)]),
Instruction('operator': CNot('qubit_count': 2), 'target': QubitSet([Qubit(0),
Qubit(1)]),
Instruction('operator': H('qubit_count': 1), 'target': QubitSet([Qubit(2)]),
Instruction('operator': Measure, 'target': QubitSet([Qubit(0)]))]
```

**apply\_gate\_noise**(noise: type[Noise] | Iterable[type[Noise]], target\_gates: type[Gate] |  
 Iterable[type[Gate]] | None = None, target\_unitary: np.ndarray | None = None,  
 target\_qubits: QubitSetInput | None = None) → *Circuit*

Apply noise to the circuit according to target\_gates, target\_unitary and target\_qubits.

For any parameter that is None, that specification is ignored (e.g. if target\_gates is None then the noise is applied after every gate in target\_qubits). If target\_gates and target\_qubits are both None, then noise is applied to every qubit after every gate.

Noise is either applied to target\_gates or target\_unitary, so they cannot be provided at the same time.

When noise.qubit\_count == 1, ie. noise is single-qubit, noise is added to all qubits in target\_gates or target\_unitary (or to all qubits in target\_qubits if target\_gates is None).

When noise.qubit\_count > 1 and target\_gates is not None, the number of qubits of any gate in target\_gates must be the same as noise.qubit\_count.

When noise.qubit\_count > 1, target\_gates and target\_unitary is None, noise is only applied to gates with the same qubit\_count in target\_qubits.

**Parameters**

- **noise** (Union[type[Noise], Iterable[type[Noise]]]) – Noise channel(s) to be applied to the circuit.
- **target\_gates** (Optional[Union[type[Gate], Iterable[type[Gate]]]]) – Gate class or List of Gate classes which noise is applied to. Default=None.
- **target\_unitary** (Optional[np.ndarray]) – matrix of the target unitary gates. Default=None.
- **target\_qubits** (Optional[QubitSetInput]) – Index or indices of qubit(s). Default=None.

**Returns***Circuit* – self

## Raises

- **TypeError** – If noise is not Noise type. If target\_gates is not a Gate type, Iterable[Gate]. If target\_unitary is not a np.ndarray type. If target\_qubits has non integers or negative integers.
- **IndexError** – If applying noise to an empty circuit. If target\_qubits is out of range of circuit.qubits.
- **ValueError** – If both target\_gates and target\_unitary are provided. If target\_unitary is not a unitary. If noise is multi-qubit noise and target\_gates contain gates with the number of qubits not the same as noise.qubit\_count.

**Warning:** If noise is multi-qubit noise while there is no gate with the same number of qubits in `target_qubits` or in the whole circuit when `target_qubits` is not given. If no `target_gates` or `target_unitary` exist in `target_qubits` or in the whole circuit when they are not given.

Examples:

```
>>> circ = Circuit().x(0).y(1).z(0).x(1).cnot(0,1)
>>> print(circ)
T : |0|1|2|

q0 : -X-Z-C-
      |
q1 : -Y-X-X-

T : |0|1|2|

>>> noise = Noise.Depolarizing(probability=0.1)
>>> circ = Circuit().x(0).y(1).z(0).x(1).cnot(0,1)
>>> print(circ.apply_gate_noise(noise, target_gates = Gate.X))
T : |    0    |    1    |2|

q0 : -X-DEPO(0.1)-Z-----C-
      |
q1 : -Y-----X-DEPO(0.1)-X-

T : |    0    |    1    |2|

>>> circ = Circuit().x(0).y(1).z(0).x(1).cnot(0,1)
>>> print(circ.apply_gate_noise(noise, target_qubits = 1))
T : |    0    |    1    |    2    |

q0 : -X-----Z-----C-----
      |
q1 : -Y-DEPO(0.1)-X-DEPO(0.1)-X-DEPO(0.1)-

T : |    0    |    1    |    2    |

>>> circ = Circuit().x(0).y(1).z(0).x(1).cnot(0,1)
>>> print(circ.apply_gate_noise(noise,
...                               target_gates = [Gate.X, Gate.Y]
...                               target_qubits = [0,1]))
```

(continues on next page)

(continued from previous page)

```

... )
T : | 0 | 1 | 2 |

q0 : -X-DEPO(0.1)-Z-----C-
      |
q1 : -Y-DEPO(0.1)-X-DEPO(0.1)-X-
T : | 0 | 1 | 2 |

```

**apply\_initialization\_noise**(*noise*: *type[Noise] | Iterable[type[Noise]]*, *target\_qubits*: *QubitSetInput | None = None*) → *Circuit*

Apply noise at the beginning of the circuit for every qubit (default) or *target\_qubits*.

Only when *target\_qubits* is given can the noise be applied to an empty circuit.

When *noise.qubit\_count* > 1, the number of qubits in *target\_qubits* must be equal to *noise.qubit\_count*.

#### Parameters

- **noise** (*Union[type[Noise], Iterable[type[Noise]]*) – Noise channel(s) to be applied to the circuit.
- **target\_qubits** (*Optional[QubitSetInput]*) – Index or indices of qubit(s). Default=None.

#### Returns

*Circuit* – self

#### Raises

- **TypeError** – If *noise* is not *Noise* type. If *target\_qubits* has non-integers or negative integers.
- **IndexError** – If applying noise to an empty circuit when *target\_qubits* is not given.
- **ValueError** – If *noise.qubit\_count* > 1 and the number of qubits in *target\_qubits* is not the same as *noise.qubit\_count*.

#### Examples

```

>>> circ = Circuit().x(0).y(1).z(0).x(1).cnot(0,1)
>>> print(circ)

```

```

>>> noise = Noise.Depolarizing(probability=0.1)
>>> circ = Circuit().x(0).y(1).z(0).x(1).cnot(0,1)
>>> print(circ.apply_initialization_noise(noise))

```

```

>>> circ = Circuit().x(0).y(1).z(0).x(1).cnot(0,1)
>>> print(circ.apply_initialization_noise(noise, target_qubits = 1))

```

```

>>> circ = Circuit()
>>> print(circ.apply_initialization_noise(noise, target_qubits = [0, 1]))

```



**make\_bound\_circuit**(*param\_values: dict[str, Number], strict: bool = False*) → *Circuit*

Binds `FreeParameter`'s based upon their name and values passed in. If parameters share the same name, all the parameters of that name will be set to the mapped value.

#### Parameters

- **param\_values** (*dict[str, Number]*) – A mapping of `FreeParameter` names to a value to assign to them.
- **strict** (*bool*) – If `True`, raises a `ValueError` if any of the `FreeParameters` in `param_values` do not appear in the circuit. `False` by default.

#### Returns

*Circuit* – Returns a circuit with all present parameters fixed to their respective values.

**apply\_readout\_noise**(*noise: type[Noise] | Iterable[type[Noise]], target\_qubits: QubitSetInput | None = None*) → *Circuit*

Apply noise right before measurement in every qubit (default) or `target_qubits`.

Only when `target_qubits` is given can the noise be applied to an empty circuit.

When `noise.qubit_count > 1`, the number of qubits in `target_qubits` must be equal to `noise.qubit_count`.

#### Parameters

- **noise** (*Union[type[Noise], Iterable[type[Noise]]*) – Noise channel(s) to be applied to the circuit.
- **target\_qubits** (*Optional[QubitSetInput]*) – Index or indices of qubit(s). Default=`None`.

#### Returns

*Circuit* – self

#### Raises

- **TypeError** – If `noise` is not `Noise` type. If `target_qubits` has non-integers.
- **IndexError** – If applying noise to an empty circuit.
- **ValueError** – If `target_qubits` has negative integers. If `noise.qubit_count > 1` and the number of qubits in `target_qubits` is not the same as `noise.qubit_count`.

### Examples

```
>>> circ = Circuit().x(0).y(1).z(0).x(1).cnot(0,1)
>>> print(circ)
```

```
>>> noise = Noise.Depolarizing(probability=0.1)
>>> circ = Circuit().x(0).y(1).z(0).x(1).cnot(0,1)
>>> print(circ.apply_initialization_noise(noise))
```

```
>>> circ = Circuit().x(0).y(1).z(0).x(1).cnot(0,1)
>>> print(circ.apply_initialization_noise(noise, target_qubits = 1))
```

```
>>> circ = Circuit()
>>> print(circ.apply_initialization_noise(noise, target_qubits = [0, 1]))
```

**add**(*addable: AddableTypes, \*args, \*\*kwargs*) → *Circuit*

Generic add method for adding item(s) to self. Any arguments that `add_circuit()` and / or `add_instruction()` and / or `add_result_type` supports are supported by this method. If adding a subroutine, check with that subroutines documentation to determine what input it allows.

**Parameters**

**addable** (*AddableTypes*) – The item(s) to add to self. Default = None.

**Returns**

*Circuit* – self

**Raises**

**TypeError** – If addable is an unsupported type

See also:

`add_circuit()`

`add_instruction()`

`add_result_type()`

**Examples**

```
>>> circ = Circuit().add([Instruction(Gate.H(), 4), Instruction(Gate.CNot(), [4,
↪ 5])])
>>> circ = Circuit().add([ResultType.StateVector()])
```

```
>>> circ = Circuit().h(4).cnot([4, 5])
```

```
>>> @circuit.subroutine()
>>> def bell_pair(target):
...     return Circuit().h(target[0]).cnot(target[0: 2])
...
>>> circ = Circuit().add(bell_pair, [4,5])
```

**adjoint**() → *Circuit*

Returns the adjoint of this circuit.

This is the adjoint of every instruction of the circuit, in reverse order. Result types, and consequently basis rotations will stay in the same order at the end of the circuit.

**Returns**

*Circuit* – The adjoint of the circuit.

**diagram**(*circuit\_diagram\_class: type = <class 'braket.circuits.text\_diagram\_builders.unicode\_circuit\_diagram.UnicodeCircuitDiagram'>*) → str

Get a diagram for the current circuit.

**Parameters**

**circuit\_diagram\_class** (*type*) – A *CircuitDiagram* class that builds the diagram for this circuit. Default = *AsciiCircuitDiagram*.

**Returns**

*str* – An ASCII string circuit diagram.

**to\_ir**(*ir\_type*: `IRType` = `IRType.JAQCD`, *serialization\_properties*: `SerializationProperties` | `None` = `None`, *gate\_definitions*: `dict[tuple[Gate, QubitSet], PulseSequence]` | `None` = `None`) → `OpenQasmProgram` | `JaqcdProgram`

Converts the circuit into the canonical intermediate representation. If the circuit is sent over the wire, this method is called before it is sent.

#### Parameters

- **ir\_type** (`IRType`) – The `IRType` to use for converting the circuit object to its IR representation.
- **serialization\_properties** (`SerializationProperties` | `None`) – The serialization properties to use while serializing the object to the IR representation. The serialization properties supplied must correspond to the supplied `ir_type`. Defaults to `None`.
- **gate\_definitions** (`dict[tuple[Gate, QubitSet], PulseSequence]` | `None`) – The calibration data for the device. default: `None`.

#### Returns

`Union[OpenQasmProgram, JaqcdProgram]` – A representation of the circuit in the `ir_type` format.

#### Raises

**ValueError** – If the supplied `ir_type` is not supported, or if the supplied serialization properties don't correspond to the `ir_type`.

**static from\_ir**(*source*: `str` | `Program`, *inputs*: `dict[str, ConstrainedStrValue | ConstrainedFloatValue | int | List[ConstrainedStrValue | ConstrainedFloatValue | int]]` | `None` = `None`) → `Circuit`

Converts an OpenQASM program to a Braket Circuit object.

#### Parameters

- **source** (`Union[str, OpenQasmProgram]`) – OpenQASM string.
- **inputs** (`Optional[dict[str, io_type]]`) – Inputs to the circuit.

#### Returns

`Circuit` – Braket Circuit implementing the OpenQASM program.

**to\_unitary**() → `ndarray`

Returns the unitary matrix representation of the entire circuit.

---

**Note:** The performance of this method degrades with qubit count. It might be slow for `qubit_count > 10`.

---

#### Returns

`np.ndarray` – A numpy array with shape  $(2^{\text{qubit\_count}}, 2^{\text{qubit\_count}})$  representing the circuit as a unitary. For an empty circuit, an empty numpy array is returned (`array([], dtype=complex)`)

#### Raises

**TypeError** – If circuit is not composed only of `Gate` instances, i.e. a circuit with `Noise` operators will raise this error.

## Examples

```
>>> circ = Circuit().h(0).cnot(0, 1)
>>> circ.to_unitary()
array([[ 0.70710678+0.j,  0.          +0.j,  0.70710678+0.j,
         0.          +0.j],
       [ 0.          +0.j,  0.70710678+0.j,  0.          +0.j,
        0.70710678+0.j],
       [ 0.          +0.j,  0.70710678+0.j,  0.          +0.j,
        -0.70710678+0.j],
       [ 0.70710678+0.j,  0.          +0.j, -0.70710678+0.j,
         0.          +0.j]])
```

**property qubits\_frozen: bool**

Whether the circuit's qubits are frozen, that is, cannot be remapped.

This may happen because the circuit contains compiler directives preventing compilation of a part of the circuit, which consequently means that none of the other qubits can be rewired either for the program to still make sense.

**Type**

bool

**property observables\_simultaneously\_measurable: bool**

Whether the circuit's observables are simultaneously measurable

If this is False, then the circuit can only be run when shots = 0, as sampling (shots > 0) measures the circuit in the observables' shared eigenbasis.

**Type**

bool

**copy()** → *Circuit*

Return a shallow copy of the circuit.

**Returns**

*Circuit* – A shallow copy of the circuit.

**adjoint\_gradient(\*args, \*\*kwargs)** → SubroutineReturn

Registers this function into the circuit class.

**Parameters**

- **observable** (*Observable*) – The expectation value of this observable is the function against which parameters in the gradient are differentiated.
- **target** (*list[QubitSetInput] | None*) – Target qubits that the result type is requested for. Each term in the target list should have the same number of qubits as the corresponding term in the observable. Default is None, which means the observable must operate only on 1 qubit and it is applied to all qubits in parallel.
- **parameters** (*list[Union[str, FreeParameter]] | None*) – The free parameters in the circuit to differentiate with respect to. Default: all.

**Returns**

*ResultType* – gradient computed via adjoint differentiation as a requested result type

## Examples

```
>>> alpha, beta = FreeParameter('alpha'), FreeParameter('beta')
>>> circ = Circuit().h(0).h(1).rx(0, alpha).yy(0, 1, beta).adjoint_gradient(
>>>     observable=Observable.Z(), target=[0], parameters=[alpha, beta]
>>> )
```

**amplitude**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

### Parameters

**state** (*list[str]*) – list of quantum states as strings with “0” and “1”

### Returns

*ResultType* – state vector as a requested result type

## Examples

```
>>> circ = Circuit().amplitude(state=["01", "10"])
```

**amplitude\_damping**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

### Parameters

- **target** (*QubitSetInput*) – Target qubit(s).
- **gamma** (*float*) – decaying rate of the amplitude damping channel.

### Returns

*Iterable[Instruction]* – Iterable of AmplitudeDamping instructions.

## Examples

```
>>> circ = Circuit().amplitude_damping(0, gamma=0.1)
```

**bit\_flip**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **probability** (*float*) – Probability of bit flipping.

### Returns

*Iterable[Instruction]* – Iterable of BitFlip instructions.

## Examples

```
>>> circ = Circuit().bit_flip(0, probability=0.1)
```

**ccnot**(\*args, \*\*kwargs) → SubroutineReturn

CCNOT gate or Toffoli gate.

$$\text{CCNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

### Parameters

- **control1** (*QubitInput*) – Control qubit 1 index.
- **control2** (*QubitInput*) – Control qubit 2 index.
- **target** (*QubitInput*) – Target qubit index.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s), in addition to control1 and control2. Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Control state only applies to control qubits specified with the control argument, not control1 and control2. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Instruction* – CCNot instruction.

## Examples

```
>>> circ = Circuit().ccnot(0, 1, 2)
```

**ccnot**(\*args, \*\*kwargs) → SubroutineReturn

Controlled NOT gate.

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

### Parameters

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.

- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – CNot instruction.

**Examples**

```
>>> circ = Circuit().cnot(0, 1)
```

**cphaseshift**(\*args, \*\*kwargs) → SubroutineReturn

Controlled phase shift gate.

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

**Parameters**

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – CPhaseShift instruction.

**Examples**

```
>>> circ = Circuit().cphaseshift(0, 1, 0.15)
```

**cphaseshift00**(\*args, \*\*kwargs) → SubroutineReturn

Controlled phase shift gate for phasing the  $|00\rangle$  state.

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

**Parameters**

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – CPhaseShift00 instruction.

## Examples

```
>>> circ = Circuit().cphaseshift00(0, 1, 0.15)
```

**cphaseshift01**(\*args, \*\*kwargs) → SubroutineReturn

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

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

### Parameters

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Instruction* – CPhaseShift01 instruction.

## Examples

```
>>> circ = Circuit().cphaseshift01(0, 1, 0.15)
```

**cphaseshift10**(\*args, \*\*kwargs) → SubroutineReturn

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

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

### Parameters

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Instruction* – CPhaseShift10 instruction.



## Examples

```
>>> circ = Circuit().cphaseshift10(0, 1, 0.15)
```

**cswap**(\*args, \*\*kwargs) → SubroutineReturn

Controlled Swap gate.

$$\text{CSWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

### Parameters

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Instruction* – CSwap instruction.

## Examples

```
>>> circ = Circuit().cswap(0, 1, 2)
```

**cv**(\*args, \*\*kwargs) → SubroutineReturn

Controlled Sqrt of X gate.

$$\text{CV} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 + 0.5i & 0.5 - 0.5i \\ 0 & 0 & 0.5 - 0.5i & 0.5 + 0.5i \end{bmatrix}.$$

### Parameters

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Instruction* – CV instruction.

## Examples

```
>>> circ = Circuit().cv(0, 1)
```

**cy**(\*args, \*\*kwargs) → SubroutineReturn

Controlled Pauli-Y gate.

$$CY = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{bmatrix}.$$

### Parameters

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Instruction* – CY instruction.

## Examples

```
>>> circ = Circuit().cy(0, 1)
```

**cz**(\*args, \*\*kwargs) → SubroutineReturn

Controlled Pauli-Z gate.

$$CZ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}.$$

### Parameters

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Instruction* – CZ instruction.

## Examples

```
>>> circ = Circuit().cz(0, 1)
```

**density\_matrix**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

### Parameters

**target** (*QubitSetInput* / *None*) – The target qubits of the reduced density matrix. Default is *None*, and the full density matrix is returned.

### Returns

*ResultType* – density matrix as a requested result type

## Examples

```
>>> circ = Circuit().density_matrix(target=[0, 1])
```

**depolarizing**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **probability** (*float*) – Probability of depolarizing.

### Returns

*Iterable[Instruction]* – Iterable of Depolarizing instructions.

## Examples

```
>>> circ = Circuit().depolarizing(0, probability=0.1)
```

**ecr**(\*args, \*\*kwargs) → SubroutineReturn

An echoed RZX(pi/2) gate (ECR gate).

$$\text{ECR} = \begin{bmatrix} 0 & 0 & 1 & i \\ 0 & 0 & i & 1 \\ 1 & -i & 0 & 0 \\ -i & 1 & 0 & 0 \end{bmatrix}.$$

### Parameters

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default *None*.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).

- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – ECR instruction.

**Examples**

```
>>> circ = Circuit().ecr(0, 1)
```

**expectation**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

**Parameters**

- **observable** (*Observable*) – the observable for the result type
- **target** (*QubitSetInput* | *None*) – Target qubits that the result type is requested for. Default is None, which means the observable must operate only on 1 qubit and it is applied to all qubits in parallel.

**Returns**

*ResultType* – expectation as a requested result type

**Examples**

```
>>> circ = Circuit().expectation(observable=Observable.Z(), target=0)
```

**generalized\_amplitude\_damping**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s).
- **gamma** (*float*) – The damping rate of the amplitude damping channel.
- **probability** (*float*) – Probability of the system being excited by the environment.

**Returns**

*Iterable[Instruction]* – Iterable of GeneralizedAmplitudeDamping instructions.

**Examples**

```
>>> circ = Circuit().generalized_amplitude_damping(0, gamma=0.1, probability =
↳ 0.9)
```

**gphase**(\*args, \*\*kwargs) → SubroutineReturn

Global phase gate.

If the gate is applied with control/negative control modifiers, it is translated in an equivalent gate using the following definition: `phaseshift() = ctrl @ gphase()`. The rightmost control qubit is used for the translation. If the polarity of the rightmost control modifier is negative, the following identity is used: `negctrl @ gphase() q = x q; ctrl @ gphase() q; x q`.

Unitary matrix:

$$\text{gphase}(\gamma) = e^{i\gamma} I_1 = [e^{i\gamma}].$$

#### Parameters

- **angle** (*Union*[FreeParameterExpression, float]) – Phase in radians.
- **control** (*Optional*[QubitSetInput]) – Control qubit(s). Default None.
- **control\_state** (*Optional*[BasisStateInput]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (float) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* | *Iterable*[*Instruction*] – GPhase instruction.

#### Examples

```
>>> circ = Circuit().gphase(0.45)
```

**gpi**(\*args, \*\*kwargs) → SubroutineReturn

IonQ GPI gate.

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

#### Parameters

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union*[FreeParameterExpression, float]) – Angle in radians.
- **control** (*Optional*[QubitSetInput]) – Control qubit(s). Default None.
- **control\_state** (*Optional*[BasisStateInput]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (float) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Iterable*[*Instruction*] – GPi instruction.

## Examples

```
>>> circ = Circuit().gpi(0, 0.15)
```

**gpi2**(\*args, \*\*kwargs) → SubroutineReturn

IonQ GPi2 gate.

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

### Parameters

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union[FreeParameterExpression, float]*) – Angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Iterable[Instruction]* – GPi2 instruction.

## Examples

```
>>> circ = Circuit().gpi2(0, 0.15)
```

**h**(\*args, \*\*kwargs) → SubroutineReturn

Hadamard gate.

Unitary matrix:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Iterable[Instruction]* – Iterable of H instructions.

## Examples

```
>>> circ = Circuit().h(0)
>>> circ = Circuit().h([0, 1, 2])
```

**i**(\*args, \*\*kwargs) → SubroutineReturn

Identity gate.

Unitary matrix:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

### Parameters

- **target** (*QubitSetInput*) – Target qubit(s).
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Iterable[Instruction]* – Iterable of I instructions.

## Examples

```
>>> circ = Circuit().i(0)
>>> circ = Circuit().i([0, 1, 2])
```

**iswap**(\*args, \*\*kwargs) → SubroutineReturn

ISwap gate.

$$\text{iSWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & i & 0 \\ 0 & i & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

### Parameters

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).

- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – ISwap instruction.

#### Examples

```
>>> circ = Circuit().iswap(0, 1)
```

**kraus**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

#### Parameters

- **targets** (*QubitSetInput*) – Target qubit(s)
- **matrices** (*Iterable[array]*) – Matrices that define a general noise channel.
- **display\_name** (*str*) – The display name.

#### Returns

*Iterable[Instruction]* – Iterable of Kraus instructions.

#### Examples

```
>>> K0 = np.eye(4) * np.sqrt(0.9)
>>> K1 = np.kron([[1., 0.], [0., 1.]], [[0., 1.], [1., 0.]]) * np.sqrt(0.1)
>>> circ = Circuit().kraus([1, 0], matrices=[K0, K1])
```

**ms**(\*args, \*\*kwargs) → SubroutineReturn

IonQ Mølmer-Sørensen gate.

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

#### Parameters

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **angle\_1** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **angle\_2** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **angle\_3** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).



- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – MS instruction.

**Examples**

```
>>> circ = Circuit().ms(0, 1, 0.15, 0.34)
```

**pauli\_channel**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s) probability list[float]: Probabilities for the Pauli X, Y and Z noise happening in the Kraus channel.
- **probX** (*float*) – X rotation probability.
- **probY** (*float*) – Y rotation probability.
- **probZ** (*float*) – Z rotation probability.

**Returns**

*Iterable[Instruction]* – Iterable of PauliChannel instructions.

**Examples**

```
>>> circ = Circuit().pauli_channel(0, probX=0.1, probY=0.2, probZ=0.3)
```

**phase\_damping**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **gamma** (*float*) – Probability of phase damping.

**Returns**

*Iterable[Instruction]* – Iterable of PhaseDamping instructions.

**Examples**

```
>>> circ = Circuit().phase_damping(0, gamma=0.1)
```

**phase\_flip**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **probability** (*float*) – Probability of phase flipping.

**Returns**

*Iterable[Instruction]* – Iterable of PhaseFlip instructions.

## Examples

```
>>> circ = Circuit().phase_flip(0, probability=0.1)
```

**phaseshift**(\*args, \*\*kwargs) → SubroutineReturn

Phase shift gate.

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

### Parameters

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Iterable[Instruction]* – PhaseShift instruction.

## Examples

```
>>> circ = Circuit().phaseshift(0, 0.15)
```

**probability**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

### Parameters

**target** (*QubitSetInput | None*) – The target qubits that the result type is requested for. Default is None, which means all qubits for the circuit.

### Returns

*ResultType* – probability as a requested result type

## Examples

```
>>> circ = Circuit().probability(target=[0, 1])
```

**prx**(\*args, \*\*kwargs) → SubroutineReturn

PhaseRx gate.

$$\text{PRx}(\theta, \phi) = \begin{bmatrix} \cos(\theta/2) & -ie^{-i\phi} \sin(\theta/2) \\ -ie^{i\phi} \sin(\theta/2) & \cos(\theta/2) \end{bmatrix}.$$

### Parameters

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle\_1** (*Union[FreeParameterExpression, float]*) – First angle in radians.
- **angle\_2** (*Union[FreeParameterExpression, float]*) – Second angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Iterable[Instruction]* – PhaseRx instruction.

#### Examples

```
>>> circ = Circuit().prx(0, 0.15, 0.25)
```

**pswap**(\*args, \*\*kwargs) → SubroutineReturn

PSwap gate.

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

#### Parameters

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – PSwap instruction.

## Examples

```
>>> circ = Circuit().pswap(0, 1, 0.15)
```

**pulse\_gate**(\*args, \*\*kwargs) → SubroutineReturn

Arbitrary pulse gate which provides the ability to embed custom pulse sequences within circuits.

### Parameters

- **targets** (*QubitSet*) – Target qubits. Note: These are only for representational purposes. The actual targets are determined by the frames used in the pulse sequence.
- **pulse\_sequence** (*PulseSequence*) – PulseSequence to embed within the circuit.
- **display\_name** (*str*) – Name to be used for an instance of this pulse gate for circuit diagrams. Defaults to PG.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Instruction* – Pulse gate instruction.

## Examples

```
>>> pulse_seq = PulseSequence().set_frequency(frame, frequency) ...
>>> circ = Circuit().pulse_gate(pulse_sequence=pulse_seq, targets=[0])
```

**rx**(\*args, \*\*kwargs) → SubroutineReturn

X-axis rotation gate.

$$R_x(\phi) = \begin{bmatrix} \cos(\phi/2) & -i \sin(\phi/2) \\ -i \sin(\phi/2) & \cos(\phi/2) \end{bmatrix}.$$

### Parameters

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union[FreeParameterExpression, float]*) – Angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).

- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Iterable[Instruction]* – Rx instruction.

#### Examples

```
>>> circ = Circuit().rx(0, 0.15)
```

**ry**(\*args, \*\*kwargs) → SubroutineReturn

Y-axis rotation gate.

$$R_y(\phi) = \begin{bmatrix} \cos(\phi/2) & -\sin(\phi/2) \\ \sin(\phi/2) & \cos(\phi/2) \end{bmatrix}.$$

#### Parameters

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union[FreeParameterExpression, float]*) – Angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Iterable[Instruction]* – Rx instruction.

#### Examples

```
>>> circ = Circuit().ry(0, 0.15)
```

**rz**(\*args, \*\*kwargs) → SubroutineReturn

Z-axis rotation gate.

$$R_z(\phi) = \begin{bmatrix} e^{-i\phi/2} & 0 \\ 0 & e^{i\phi/2} \end{bmatrix}.$$

#### Parameters

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union[FreeParameterExpression, float]*) – Angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).

- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – Rx instruction.

**Examples**

```
>>> circ = Circuit().rz(0, 0.15)
```

**s**(\*args, \*\*kwargs) → SubroutineReturn

S gate.

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – Iterable of S instructions.

**Examples**

```
>>> circ = Circuit().s(0)
>>> circ = Circuit().s([0, 1, 2])
```

**sample**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

**Parameters**

- **observable** (*Observable*) – the observable for the result type
- **target** (*QubitSetInput | None*) – Target qubits that the result type is requested for. Default is None, which means the observable must operate only on 1 qubit and it is applied to all qubits in parallel.

**Returns**

*ResultType* – sample as a requested result type

## Examples

```
>>> circ = Circuit().sample(observable=Observable.Z(), target=0)
```

**si**(\*args, \*\*kwargs) → SubroutineReturn

Conjugate transpose of S gate.

$$S^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix}.$$

### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Iterable[Instruction]* – Iterable of Si instructions.

## Examples

```
>>> circ = Circuit().si(0)
>>> circ = Circuit().si([0, 1, 2])
```

**state\_vector**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

### Returns

*ResultType* – state vector as a requested result type

## Examples

```
>>> circ = Circuit().state_vector()
```

**swap**(\*args, \*\*kwargs) → SubroutineReturn

Swap gate.

$$\text{SWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

### Parameters

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.

- **control** (*Optional*[*QubitSetInput*]) – Control qubit(s). Default None.
- **control\_state** (*Optional*[*BasisStateInput*]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – Swap instruction.

**Examples**

```
>>> circ = Circuit().swap(0, 1)
```

**t**(\*args, \*\*kwargs) → SubroutineReturn

T gate.

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional*[*QubitSetInput*]) – Control qubit(s). Default None.
- **control\_state** (*Optional*[*BasisStateInput*]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable*[*Instruction*] – Iterable of T instructions.

**Examples**

```
>>> circ = Circuit().t(0)
>>> circ = Circuit().t([0, 1, 2])
```

**ti**(\*args, \*\*kwargs) → SubroutineReturn

Conjugate transpose of T gate.

$$T^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & e^{-i\pi/4} \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)



- **control** (*Optional*[*QubitSetInput*]) – Control qubit(s). Default None.
- **control\_state** (*Optional*[*BasisStateInput*]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable*[*Instruction*] – Iterable of Ti instructions.

**Examples**

```
>>> circ = Circuit().ti(0)
>>> circ = Circuit().ti([0, 1, 2])
```

**two\_qubit\_dephasing**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1.
- **target2** (*QubitInput*) – Target qubit 2.
- **probability** (*float*) – Probability of two-qubit dephasing.

**Returns**

*Iterable*[*Instruction*] – Iterable of Dephasing instructions.

**Examples**

```
>>> circ = Circuit().two_qubit_dephasing(0, 1, probability=0.1)
```

**two\_qubit\_depolarizing**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1.
- **target2** (*QubitInput*) – Target qubit 2.
- **probability** (*float*) – Probability of two-qubit depolarizing.

**Returns**

*Iterable*[*Instruction*] – Iterable of Depolarizing instructions.

## Examples

```
>>> circ = Circuit().two_qubit_depolarizing(0, 1, probability=0.1)
```

**two\_qubit\_pauli\_channel**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

### Parameters

- **target1** (*QubitInput*) – Target qubit 1.
- **target2** (*QubitInput*) – Target qubit 2.
- **probabilities** (*dict[str, float]*) – Probability of two-qubit Pauli channel.

### Returns

*Iterable[Instruction]* – Iterable of Depolarizing instructions.

## Examples

```
>>> circ = Circuit().two_qubit_pauli_channel(0, 1, {"XX": 0.1})
```

**u**(\*args, \*\*kwargs) → SubroutineReturn

Generalized single-qubit rotation gate.

Unitary matrix:

$$U(\theta, \phi, \lambda) = \begin{bmatrix} \cos(\theta/2) & -e^{i\lambda} \sin(\theta/2) \\ e^{i\phi} \sin(\theta/2) & -e^{i(\phi+\lambda)} \cos(\theta/2) \end{bmatrix}.$$

### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **angle\_1** (*Union[FreeParameterExpression, float]*) – theta angle in radians.
- **angle\_2** (*Union[FreeParameterExpression, float]*) – phi angle in radians.
- **angle\_3** (*Union[FreeParameterExpression, float]*) – lambda angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Iterable[Instruction]* – U instruction.

## Examples

```
>>> circ = Circuit().u(0, 0.15, 0.34, 0.52)
```

**unitary**(\*args, \*\*kwargs) → SubroutineReturn

Arbitrary unitary gate.

### Parameters

- **targets** (*QubitSet*) – Target qubits.
- **matrix** (*numpy.ndarray*) – Unitary matrix which defines the gate. Matrix should be compatible with the supplied targets, with  $2 ** \text{len}(\text{targets}) == \text{matrix.shape}[0]$ .
- **display\_name** (*str*) – Name to be used for an instance of this unitary gate for circuit diagrams. Defaults to U.

### Returns

*Instruction* – Unitary instruction.

### Raises

**ValueError** – If **matrix** is not a two-dimensional square matrix, or has a dimension length that is not compatible with the **targets**, or is not unitary,

## Examples

```
>>> circ = Circuit().unitary(matrix=np.array([[0, 1],[1, 0]]), targets=[0])
```

**v**(\*args, \*\*kwargs) → SubroutineReturn

Square root of X gate (V gate).

$$V = \frac{1}{2} \begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix}.$$

### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Iterable[Instruction]* – Iterable of V instructions.

## Examples

```
>>> circ = Circuit().v(0)
>>> circ = Circuit().v([0, 1, 2])
```

**variance**(\*args, \*\*kwargs) → SubroutineReturn

Registers this function into the circuit class.

### Parameters

- **observable** (*Observable*) – the observable for the result type
- **target** (*QubitSetInput* / *None*) – Target qubits that the result type is requested for. Default is *None*, which means the observable must only operate on 1 qubit and it will be applied to all qubits in parallel

### Returns

*ResultType* – variance as a requested result type

## Examples

```
>>> circ = Circuit().variance(observable=Observable.Z(), target=0)
```

**vi**(\*args, \*\*kwargs) → SubroutineReturn

Conjugate transpose of square root of X gate (conjugate transpose of V).

$$V^\dagger = \frac{1}{2} \begin{bmatrix} 1-i & 1+i \\ 1+i & 1-i \end{bmatrix}.$$

### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default *None*.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Iterable[Instruction]* – Iterable of Vi instructions.

## Examples

```
>>> circ = Circuit().vi(0)
>>> circ = Circuit().vi([0, 1, 2])
```

**x**(\*args, \*\*kwargs) → SubroutineReturn

Pauli-X gate.

Unitary matrix:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Iterable[Instruction]* – Iterable of X instructions.

### Examples

```
>>> circ = Circuit().x(0)
>>> circ = Circuit().x([0, 1, 2])
```

**xx**(\*args, \*\*kwargs) → SubroutineReturn

Ising XX coupling gate.

$$XX(\phi) = \begin{bmatrix} \cos(\phi/2) & 0 & 0 & -i \sin(\phi/2) \\ 0 & \cos(\phi/2) & -i \sin(\phi/2) & 0 \\ 0 & -i \sin(\phi/2) & \cos(\phi/2) & 0 \\ -i \sin(\phi/2) & 0 & 0 & \cos(\phi/2) \end{bmatrix}.$$

### Parameters

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Instruction* – XX instruction.

## Examples

```
>>> circ = Circuit().xx(0, 1, 0.15)
```

**xy**(\*args, \*\*kwargs) → SubroutineReturn

XY gate.

$$XY(\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\phi/2) & i \sin(\phi/2) & 0 \\ 0 & i \sin(\phi/2) & \cos(\phi/2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

### Parameters

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Returns

*Instruction* – XY instruction.

## Examples

```
>>> circ = Circuit().xy(0, 1, 0.15)
```

**y**(\*args, \*\*kwargs) → SubroutineReturn

Pauli-Y gate.

Unitary matrix:

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}.$$

### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).

- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Iterable[Instruction]* – Iterable of Y instructions.

#### Examples

```
>>> circ = Circuit().y(0)
>>> circ = Circuit().y([0, 1, 2])
```

**yy**(\*args, \*\*kwargs) → SubroutineReturn

Ising YY coupling gate.

$$YY(\phi) = \begin{bmatrix} \cos(\phi/2) & 0 & 0 & i \sin(\phi/2) \\ 0 & \cos(\phi/2) & -i \sin(\phi/2) & 0 \\ 0 & -i \sin(\phi/2) & \cos(\phi/2) & 0 \\ i \sin(\phi/2) & 0 & 0 & \cos(\phi/2) \end{bmatrix}.$$

#### Parameters

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – YY instruction.

#### Examples

```
>>> circ = Circuit().yy(0, 1, 0.15)
```

**z**(\*args, \*\*kwargs) → SubroutineReturn

Pauli-Z gate.

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

#### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.

- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – Iterable of Z instructions.

**Examples**

```
>>> circ = Circuit().z(0)
>>> circ = Circuit().z([0, 1, 2])
```

**zz**(\*args, \*\*kwargs) → SubroutineReturn

Ising ZZ coupling gate.

$$ZZ(\phi) = \begin{bmatrix} e^{-i\phi/2} & 0 & 0 & 0 \\ 0 & e^{i\phi/2} & 0 & 0 \\ 0 & 0 & e^{i\phi/2} & 0 \\ 0 & 0 & 0 & e^{-i\phi/2} \end{bmatrix}.$$

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – ZZ instruction.

**Examples**

```
>>> circ = Circuit().zz(0, 1, 0.15)
```

**braket.circuits.circuit.subroutine**(*register: bool = False*) → Callable

Subroutine is a function that returns instructions, result types, or circuits.

**Parameters**

**register** (*bool*) – If True, adds this subroutine into the *Circuit* class. Default = False.



**Returns***Callable* – The subroutine function.**Examples**

```

>>> @circuit.subroutine(register=True)
>>> def bell_circuit():
...     return Circuit().h(0).cnot(0, 1)
...
>>> circ = Circuit().bell_circuit()
>>> for instr in circ.instructions:
...     print(instr)
...
Instruction('operator': 'H', 'target': QubitSet(Qubit(0),))
Instruction('operator': 'H', 'target': QubitSet(Qubit(1),))

```

**braket.circuits.circuit\_diagram module****class** braket.circuits.circuit\_diagram.CircuitDiagram

Bases: ABC

A class that builds circuit diagrams.

**abstract static build\_diagram**(*circuit*: Circuit) → str

Build a diagram for the specified circuit.

**Parameters****circuit** (*cir.Circuit*) – The circuit to build a diagram for.**Returns***str* – String representation for the circuit diagram. An empty string is returned for an empty circuit.**braket.circuits.circuit\_helpers module****braket.circuits.circuit\_helpers.validate\_circuit\_and\_shots**(*circuit*: Circuit, *shots*: int) → None

Validates if circuit and shots are correct before running on a device

**Parameters**

- **circuit** (Circuit) – circuit to validate
- **shots** (int) – shots to validate

**Raises**

**ValueError** – If circuit has no instructions; if circuit has a non-gphase instruction; if no result types specified for circuit and shots=0. See `braket.circuit.result_types`; if circuit has observables that cannot be simultaneously measured and shots>0; or, if `StateVector` or `Amplitude` are specified as result types when shots>0.

**braket.circuits.compiler\_directive module**

**class** `braket.circuits.compiler_directive.CompilerDirective`(*ascii\_symbols: Sequence[str]*)

Bases: *Operator*

A directive specifying how the compiler should process a part of the circuit.

For example, a directive may tell the compiler not to modify some gates in the circuit.

Initiates a *CompilerDirective*.

**Parameters**

**ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the compiler directive. These are used when printing a diagram of circuits.

**property name:** *str*

The name of the operator.

**Returns**

*str* – The name of the operator.

**property ascii\_symbols:** *tuple[str, ...]*

Returns the ascii symbols for the compiler directive.

**Type**

*tuple[str, ...]*

**to\_ir**(*target: QubitSet | None = None, ir\_type: IRType = IRType.JAQCD, serialization\_properties: SerializationProperties | None = None, \*\*kwargs*) → *Any*

Returns IR object of the compiler directive.

**Parameters**

- **target** (*QubitSet | None*) – target qubit(s). Defaults to None
- **ir\_type** (*IRType*) – The IRType to use for converting the compiler directive object to its IR representation. Defaults to IRType.JAQCD.
- **serialization\_properties** (*SerializationProperties | None*) – The serialization properties to use while serializing the object to the IR representation. The serialization properties supplied must correspond to the supplied *ir\_type*. Defaults to None.

**Returns**

*Any* – IR object of the compiler directive.

**Raises**

**ValueError** – If the supplied *ir\_type* is not supported.

**counterpart**() → *CompilerDirective*

Returns the “opposite” counterpart to this compiler directive.

For example, the counterpart of a directive that starts a box is the directive that ends the box.

**Returns**

*CompilerDirective* – The counterpart compiler directive

**braket.circuits.compiler\_directives module****class** `braket.circuits.compiler_directives.StartVerbatimBox`Bases: *CompilerDirective*

Prevents the compiler from modifying any ensuing instructions until the appearance of a corresponding `EndVerbatimBox`.

Initiates a `CompilerDirective`.

**Parameters**

**ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the compiler directive. These are used when printing a diagram of circuits.

**counterpart()** → *CompilerDirective*

Returns the “opposite” counterpart to this compiler directive.

For example, the counterpart of a directive that starts a box is the directive that ends the box.

**Returns**

*CompilerDirective* – The counterpart compiler directive

**class** `braket.circuits.compiler_directives.EndVerbatimBox`Bases: *CompilerDirective*

Marks the end of a portion of code following a `StartVerbatimBox` that prevents the enclosed instructions from being modified by the compiler.

Initiates a `CompilerDirective`.

**Parameters**

**ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the compiler directive. These are used when printing a diagram of circuits.

**counterpart()** → *CompilerDirective*

Returns the “opposite” counterpart to this compiler directive.

For example, the counterpart of a directive that starts a box is the directive that ends the box.

**Returns**

*CompilerDirective* – The counterpart compiler directive

**braket.circuits.free\_parameter module****braket.circuits.free\_parameter\_expression module****braket.circuits.gate module****class** `braket.circuits.gate.Gate(qubit_count: int | None, ascii_symbols: Sequence[str])`Bases: *QuantumOperator*

Class `Gate` represents a quantum gate that operates on *N* qubits. Gates are considered the building blocks of quantum circuits. This class is considered the gate definition containing the metadata that defines what a gate is and what it does.

Initializes a `Gate`.

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

list[*Gate*] – The gates comprising the adjoint of this gate.

**to\_ir**(*target: QubitSet*, *ir\_type: IRType = IRType.JAQCD*, *serialization\_properties: OpenQASMSerializationProperties | None = None*, *\*, control: QubitSet | None = None*, *control\_state: int | list[int] | str | BasisState | None = None*, *power: float = 1*) → Any

Returns IR object of quantum operator and target

**Parameters**

- **target** (*QubitSet*) – target qubit(s).
- **ir\_type** (*IRType*) – The IRType to use for converting the gate object to its IR representation. Defaults to IRType.JAQCD.
- **serialization\_properties** (*Optional[SerializationProperties]*) – The serialization properties to use while serializing the object to the IR representation. The serialization properties supplied must correspond to the supplied **ir\_type**. Defaults to None.
- **control** (*Optional[QubitSet]*) – Control qubit(s). Only supported for OpenQASM. Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example "0101", [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default "1" \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

Any – IR object of the quantum operator and target

**Raises**

- **ValueError** – If the supplied **ir\_type** is not supported, or if the supplied serialization
- **properties don't correspond to the ir\_type.** –
- **ValueError** – If gate modifiers are supplied with **ir\_type** Jaqcd.

**property ascii\_symbols:** tuple[str, ...]

Returns the ascii symbols for the quantum operator.

**Type**

tuple[str, ...]

**classmethod register\_gate**(gate: type[Gate]) → None

Register a gate implementation by adding it into the Gate class.

**Parameters****gate** (type[Gate]) – Gate class to register.**class CCNot**

Bases: Gate

CCNOT gate or Toffoli gate.

Unitary matrix:

$$\text{CCNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

Initializes a Gate.

**Parameters**

- **qubit\_count** (Optional[int]) – Number of qubits this gate interacts with.
- **ascii\_symbols** (Sequence[str]) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises****ValueError** – qubit\_count is less than 1, ascii\_symbols are None, or ascii\_symbols length != qubit\_count**adjoint**() → list[Gate]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

list[Gate] – The gates comprising the adjoint of this gate.

**static cnot**(control1: QubitInput, control2: QubitInput, target: QubitInput, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1) → Instruction

CCNOT gate or Toffoli gate.

$$\text{CCNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

#### Parameters

- **control1** (*QubitInput*) – Control qubit 1 index.
- **control2** (*QubitInput*) – Control qubit 2 index.
- **target** (*QubitInput*) – Target qubit index.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s), in addition to control1 and control2. Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control1. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Control state only applies to control qubits specified with the control argument, not control1 and control2. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – CCNot instruction.

#### Examples

```
>>> circ = Circuit().ccnot(0, 1, 2)
```

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

#### Raises

**NotImplementedError** – Not Implemented.

#### Returns

*np.ndarray* – A matrix representation of the quantum operator

**class CNot**

Bases: [Gate](#)

Controlled NOT gate.

Unitary matrix:

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Initializes a [Gate](#).

#### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[[Gate](#)]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

#### Returns

list[[Gate](#)] – The gates comprising the adjoint of this gate.

**static cnot**(*control: QubitSetInput, target: QubitInput, power: float = 1*) → [Instruction](#)

Controlled NOT gate.

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

#### Parameters

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – CNot instruction.

### Examples

```
>>> circ = Circuit().cnot(0, 1)
```

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class CPhaseShift**(*angle*: [FreeParameterExpression](#) | *float*)

Bases: [AngledGate](#)

Controlled phase shift gate.

Unitary matrix:

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

**Parameters**

**angle** (*Union*[[FreeParameterExpression](#), *float*]) – angle in radians.

Initializes an [AngledGate](#).

**Parameters**

- **angle** (*Union*[[FreeParameterExpression](#), *float*]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If the **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**, or **angle** is None

**bind\_values**(**\*\*kwargs**) → [AngledGate](#)

Takes in parameters and attempts to assign them to values.

**Returns**

[AngledGate](#) – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static cphaseshift**(*control*: [QubitSetInput](#), *target*: [QubitInput](#), *angle*: [FreeParameterExpression](#) | *float*, *power*: *float* = 1) → [Instruction](#)



Controlled phase shift gate.

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

#### Parameters

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – CPhaseShift instruction.

### Examples

```
>>> circ = Circuit().cphaseshift(0, 1, 0.15)
```

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

#### Raises

**NotImplementedError** – Not Implemented.

#### Returns

*np.ndarray* – A matrix representation of the quantum operator

**class CPhaseShift00**(*angle: FreeParameterExpression | float*)

Bases: [AngledGate](#)

Controlled phase shift gate for phasing the  $|00\rangle$  state.

Unitary matrix:

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

#### Parameters

**angle** (*Union[FreeParameterExpression, float]*) – angle in radians.

Initializes an `AngledGate`.

#### Parameters

- **angle** (`Union[FreeParameterExpression, float]`) – The angle of the gate in radians or expression representation.
- **qubit\_count** (`Optional[int]`) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (`Sequence[str]`) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, or `angle` is `None`

**bind\_values**(*\*\*kwargs*) → *AngledGate*

Takes in parameters and attempts to assign them to values.

#### Returns

*AngledGate* – A new Gate of the same type with the requested parameters bound.

#### Raises

**NotImplementedError** – Subclasses should implement this function.

**static cphaseshift00**(*control: QubitSetInput, target: QubitInput, angle: FreeParameterExpression | float, power: float = 1*) → *Instruction*

Controlled phase shift gate for phasing the  $|00\rangle$  state.

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

#### Parameters

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **angle** (`Union[FreeParameterExpression, float]`) – angle in radians.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – CPhaseShift00 instruction.

#### Examples

```
>>> circ = Circuit().cphaseshift00(0, 1, 0.15)
```

**static fixed\_qubit\_count**() → *int*

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (Any) – Not Implemented.
- **\*\*kwargs** (Any) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

np.ndarray – A matrix representation of the quantum operator

**class CPhaseShift01**(angle: FreeParameterExpression | float)

Bases: AngledGate

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

Unitary matrix:

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

**Parameters**

**angle** (Union[FreeParameterExpression, float]) – angle in radians.

Initializes an AngledGate.

**Parameters**

- **angle** (Union[FreeParameterExpression, float]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (Optional[int]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (Sequence[str]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as qubit\_count, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If the qubit\_count is less than 1, ascii\_symbols are None, or ascii\_symbols length != qubit\_count, or angle is None

**bind\_values**(\*\*kwargs) → AngledGate

Takes in parameters and attempts to assign them to values.

**Returns**

AngledGate – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static cphaseshift01**(control: QubitSetInput, target: QubitInput, angle: FreeParameterExpression | float, power: float = 1) → Instruction

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

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

#### Parameters

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – CPhaseShift01 instruction.

### Examples

```
>>> circ = Circuit().cphaseshift01(0, 1, 0.15)
```

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

#### Raises

**NotImplementedError** – Not Implemented.

#### Returns

*np.ndarray* – A matrix representation of the quantum operator

**class CPhaseShift10**(*angle: FreeParameterExpression | float*)

Bases: [AngledGate](#)

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

Unitary matrix:

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

#### Parameters

**angle** (*Union[FreeParameterExpression, float]*) – angle in radians.

Initializes an `AngledGate`.

#### Parameters

- **angle** (*Union*[`FreeParameterExpression`, `float`]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[`int`]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[`str`]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, or `angle` is `None`

**bind\_values**(*\*\*kwargs*) → *AngledGate*

Takes in parameters and attempts to assign them to values.

#### Returns

*AngledGate* – A new Gate of the same type with the requested parameters bound.

#### Raises

**NotImplementedError** – Subclasses should implement this function.

**static cphaseshift10**(*control: QubitSetInput, target: QubitInput, angle: FreeParameterExpression | float, power: float = 1*) → *Instruction*

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

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

#### Parameters

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **angle** (*Union*[`FreeParameterExpression`, `float`]) – angle in radians.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – CPhaseShift10 instruction.

#### Examples

```
>>> circ = Circuit().cphaseshift10(0, 1, 0.15)
```

**static fixed\_qubit\_count**() → `int`

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class CSwap**

Bases: [Gate](#)

Controlled Swap gate.

Unitary matrix:

$$\text{CSWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Initializes a [Gate](#).

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[[Gate](#)]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

*list[Gate]* – The gates comprising the adjoint of this gate.

**static cswap**(*control: QubitSetInput, target1: QubitInput, target2: QubitInput, power: float = 1*) → [Instruction](#)

Controlled Swap gate.

$$\text{CSWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

#### Parameters

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – CSwap instruction.

#### Examples

```
>>> circ = Circuit().cswap(0, 1, 2)
```

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

#### Raises

**NotImplementedError** – Not Implemented.

#### Returns

*np.ndarray* – A matrix representation of the quantum operator

**class CV**

Bases: [Gate](#)

Controlled Sqrt of X gate.

Unitary matrix:

$$\text{CV} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 + 0.5i & 0.5 - 0.5i \\ 0 & 0 & 0.5 - 0.5i & 0.5 + 0.5i \end{bmatrix}.$$

Initializes a [Gate](#).

#### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[[Gate](#)]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

#### Returns

list[[Gate](#)] – The gates comprising the adjoint of this gate.

**static cv**(control: [QubitSetInput](#), target: [QubitInput](#), power: float = 1) → [Instruction](#)

Controlled Sqrt of X gate.

$$CV = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 + 0.5i & 0.5 - 0.5i \\ 0 & 0 & 0.5 - 0.5i & 0.5 + 0.5i \end{bmatrix}.$$

#### Parameters

- **control** ([QubitSetInput](#)) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** ([QubitInput](#)) – Target qubit index.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

[Instruction](#) – CV instruction.

### Examples

```
>>> circ = Circuit().cv(0, 1)
```

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

int – The number of qubits this quantum operator acts on.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.



**Raises****NotImplementedError** – Not Implemented.**Returns***np.ndarray* – A matrix representation of the quantum operator**class CY**Bases: *Gate*

Controlled Pauli-Y gate.

Unitary matrix:

$$CY = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{bmatrix}.$$

Initializes a *Gate*.**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises****ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count****adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns***list[Gate]* – The gates comprising the adjoint of this gate.**static cy**(*control: QubitSetInput, target: QubitInput, power: float = 1*) → *Instruction*

Controlled Pauli-Y gate.

$$CY = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{bmatrix}.$$

**Parameters**

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns***Instruction* – CY instruction.

## Examples

```
>>> circ = Circuit().cy(0, 1)
```

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class CZ**

Bases: [Gate](#)

Controlled Pauli-Z gate.

Unitary matrix:

$$\text{CZ} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}.$$

Initializes a [Gate](#).

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`

**adjoint()** → list[[Gate](#)]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

*list[Gate]* – The gates comprising the adjoint of this gate.

**static** `cz(control: QubitSetInput, target: QubitInput, power: float = 1) → Instruction`

Controlled Pauli-Z gate.

$$\text{CZ} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}.$$

#### Parameters

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – CZ instruction.

### Examples

```
>>> circ = Circuit().cz(0, 1)
```

**static** `fixed_qubit_count() → int`

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**to\_matrix()** → *ndarray*

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

#### Raises

**NotImplementedError** – Not Implemented.

#### Returns

*np.ndarray* – A matrix representation of the quantum operator

**class** `ECR`

Bases: *Gate*

An echoed RZX(pi/2) gate (ECR gate).

Unitary matrix:

$$\text{ECR} = \begin{bmatrix} 0 & 0 & 1 & i \\ 0 & 0 & i & 1 \\ 1 & -i & 0 & 0 \\ -i & 1 & 0 & 0 \end{bmatrix}.$$

Initializes a *Gate*.

#### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have [“C”, “X”] to correlate a symbol with that index.

**Raises**

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → *list[Gate]*

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

*list[Gate]* – The gates comprising the adjoint of this gate.

**static ecr**(*target1: QubitInput, target2: QubitInput, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → *Instruction*

An echoed RZX(pi/2) gate (ECR gate).

$$\text{ECR} = \begin{bmatrix} 0 & 0 & 1 & i \\ 0 & 0 & i & 1 \\ 1 & -i & 0 & 0 \\ -i & 1 & 0 & 0 \end{bmatrix}.$$

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – ECR instruction.

**Examples**

```
>>> circ = Circuit().ecr(0, 1)
```

**static fixed\_qubit\_count**() → *int*

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**to\_matrix**() → *ndarray*

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class** `GPhase`(*angle*: `FreeParameterExpression` | *float*)

Bases: `AngledGate`

Global phase gate.

Unitary matrix:

$$\text{gphase}(\gamma) = e^{i\gamma} I_1 = \begin{bmatrix} e^{i\gamma} \end{bmatrix}.$$

**Parameters**

**angle** (*Union*[`FreeParameterExpression`, *float*]) – angle in radians.

**Raises**

**ValueError** – If angle is not present

Initializes an `AngledGate`.

**Parameters**

- **angle** (*Union*[`FreeParameterExpression`, *float*]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length != `qubit_count`, or `angle` is `None`

**adjoint**() → *list*[`Gate`]

Returns the adjoint of this gate as a singleton list.

**Returns**

*list*[`Gate`] – A list containing the gate with negated angle.

**bind\_values**(**\*\*kwargs**) → `AngledGate`

Takes in parameters and attempts to assign them to values.

**Returns**

`AngledGate` – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static gphase**(*angle*: [FreeParameterExpression](#) | float, \*, *control*: [QubitSetInput](#) | None = None, *control\_state*: [BasisStateInput](#) | None = None, *power*: float = 1) → [Instruction](#) | [Iterable\[Instruction\]](#)

Global phase gate.

If the gate is applied with control/negative control modifiers, it is translated in an equivalent gate using the following definition: `phaseshift() = ctrl @ gphase()`. The rightmost control qubit is used for the translation. If the polarity of the rightmost control modifier is negative, the following identity is used: `negctrl @ gphase() q = x q; ctrl @ gphase() q; x q`.

Unitary matrix:

$$\text{gphase}(\gamma) = e^{i\gamma} I_1 = [e^{i\gamma}] .$$

**Parameters**

- **angle** ([Union\[FreeParameterExpression, float\]](#)) – Phase in radians.
- **control** ([Optional\[QubitSetInput\]](#)) – Control qubit(s). Default None.
- **control\_state** ([Optional\[BasisStateInput\]](#)) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** ([float](#)) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

[Instruction](#) | [Iterable\[Instruction\]](#) – GPhase instruction.

## Examples

```
>>> circ = Circuit().gphase(0.45)
```

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class GPI**(*angle*: [FreeParameterExpression](#) | float)

Bases: [AngledGate](#)

IonQ GPI gate.

Unitary matrix:

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

**Parameters**

**angle** (*Union*[*FreeParameterExpression*, *float*]) – angle in radians.

Initializes an *AngledGate*.

**Parameters**

- **angle** (*Union*[*FreeParameterExpression*, *float*]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as *qubit\_count*, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If the *qubit\_count* is less than 1, *ascii\_symbols* are None, or *ascii\_symbols* length != *qubit\_count*, or *angle* is None

**adjoint()** → *list*[*Gate*]

Returns the adjoint of this gate as a singleton list.

**Returns**

*list*[*Gate*] – A list containing the gate with negated angle.

**bind\_values(\*\*kwargs)** → *GPi*

Takes in parameters and attempts to assign them to values.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()** → *int*

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns *NotImplemented*.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static gpi**(*target*: *QubitSetInput*, *angle*: *FreeParameterExpression* | *float*, \*, *control*: *QubitSetInput* | *None* = *None*, *control\_state*: *BasisStateInput* | *None* = *None*, *power*: *float* = 1) → *Iterable*[*Instruction*]

IonQ GPI gate.

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

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union*[*FreeParameterExpression*, *float*]) – Angle in radians.

- **control** (*Optional*[*QubitSetInput*]) – Control qubit(s). Default None.
- **control\_state** (*Optional*[*BasisStateInput*]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable*[*Instruction*] – GPI instruction.

**Examples**

```
>>> circ = Circuit().gpi(0, 0.15)
```

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class GPI2**(*angle*: *FreeParameterExpression* | *float*)

Bases: *AngledGate*

IonQ GPI2 gate.

Unitary matrix:

$$\text{GPI2}(\phi) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -ie^{-i\phi} \\ -ie^{i\phi} & 1 \end{bmatrix}.$$

**Parameters**

**angle** (*Union*[*FreeParameterExpression*, *float*]) – angle in radians.

Initializes an *AngledGate*.

**Parameters**

- **angle** (*Union*[*FreeParameterExpression*, *float*]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.



**Raises**

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, or `angle` is `None`

**adjoint()**  $\rightarrow$  list[*Gate*]

Returns the adjoint of this gate as a singleton list.

**Returns**

list[*Gate*] – A list containing the gate with negated angle.

**bind\_values(\*\*kwargs)**  $\rightarrow$  *GPI2*

Takes in parameters and attempts to assign them to values.

**Returns**

*AngledGate* – A new *Gate* of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()**  $\rightarrow$  int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

int – The number of qubits this quantum operator acts on.

**static gpi2(target: *QubitSetInput*, angle: *FreeParameterExpression* | float, \*, control: *QubitSetInput* | None = None, control\_state: *BasisStateInput* | None = None, power: float = 1)  $\rightarrow$  Iterable[*Instruction*]**

IonQ GPI2 gate.

$$\text{GPI2}(\phi) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -ie^{-i\phi} \\ -ie^{i\phi} & 1 \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union*[*FreeParameterExpression*, float]) – Angle in radians.
- **control** (*Optional*[*QubitSetInput*]) – Control qubit(s). Default `None`.
- **control\_state** (*Optional*[*BasisStateInput*]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in `control`. Will be ignored if `control` is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (float) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

Iterable[*Instruction*] – GPI2 instruction.

**Examples**

```
>>> circ = Circuit().gpi2(0, 0.15)
```

**to\_matrix()**  $\rightarrow$  ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (Any) – Not Implemented.
- **\*\*kwargs** (Any) – Not Implemented.

**Raises****NotImplementedError** – Not Implemented.**Returns***np.ndarray* – A matrix representation of the quantum operator**class H**Bases: *Gate*

Hadamard gate.

Unitary matrix:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Initializes a *Gate*.**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have [“C”, “X”] to correlate a symbol with that index.

**Raises****ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count****adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns***list[Gate]* – The gates comprising the adjoint of this gate.**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.**Returns***int* – The number of qubits this quantum operator acts on.**static h**(*target: QubitSetInput*, \*, *control: QubitSetInput | None = None*, *control\_state: BasisStateInput | None = None*, *power: float = 1*) → Iterable[*Instruction*]

Hadamard gate.

Unitary matrix:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.

- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – Iterable of H instructions.

**Examples**

```
>>> circ = Circuit().h(0)
>>> circ = Circuit().h([0, 1, 2])
```

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class I**

Bases: [Gate](#)

Identity gate.

Unitary matrix:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Initializes a [Gate](#).

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have [“C”, “X”] to correlate a symbol with that index.

**Raises**

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[[Gate](#)]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

*list[Gate]* – The gates comprising the adjoint of this gate.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static i**(*target: QubitSetInput*, \*, *control: QubitSetInput | None = None*, *control\_state: BasisStateInput | None = None*, *power: float = 1*) → *Iterable[Instruction]*

Identity gate.

Unitary matrix:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default `None`.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – `Iterable` of `I` instructions.

**Examples**

```
>>> circ = Circuit().i(0)
>>> circ = Circuit().i([0, 1, 2])
```

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class ISwap**

Bases: *Gate*

ISwap gate.

Unitary matrix:

$$\text{iSWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & i & 0 \\ 0 & i & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Initializes a [Gate](#).

#### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[[Gate](#)]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

#### Returns

list[[Gate](#)] – The gates comprising the adjoint of this gate.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

int – The number of qubits this quantum operator acts on.

**static iswap**(*target1: QubitInput, target2: QubitInput, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → [Instruction](#)

ISwap gate.

$$\text{iSWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & i & 0 \\ 0 & i & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

#### Parameters

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example "0101", [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default "1" \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

[Instruction](#) – ISwap instruction.

## Examples

```
>>> circ = Circuit().iswap(0, 1)
```

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (Any) – Not Implemented.
- **\*\*kwargs** (Any) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

```
class MS(angle_1: FreeParameterExpression | float, angle_2: FreeParameterExpression | float, angle_3:
    FreeParameterExpression | float = 1.5707963267948966)
```

Bases: *TripleAngledGate*

IonQ Mølmer-Sørensen gate.

Unitary matrix:

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

**Parameters**

- **angle\_1** (*Union*[*FreeParameterExpression*, *float*]) – angle in radians.
- **angle\_2** (*Union*[*FreeParameterExpression*, *float*]) – angle in radians.
- **angle\_3** (*Union*[*FreeParameterExpression*, *float*]) – angle in radians. Default value is  $\text{angle}_3 = \pi/2$ .

Initiates a *TripleAngledGate*.

**Parameters**

- **angle\_1** (*Union*[*FreeParameterExpression*, *float*]) – The first angle of the gate in radians or expression representation.
- **angle\_2** (*Union*[*FreeParameterExpression*, *float*]) – The second angle of the gate in radians or expression representation.
- **angle\_3** (*Union*[*FreeParameterExpression*, *float*]) – The third angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length != `qubit_count`, or `angle_1` or `angle_2` or `angle_3` is `None`

**adjoint()** → list[*Gate*]

Returns the adjoint of this gate as a singleton list.

**Returns**

list[*Gate*] – A list containing the gate with negated angle.

**bind\_values(\*\*kwargs)** → *MS*

Takes in parameters and attempts to assign them to values.

**Parameters**

**\*\*kwargs** (*FreeParameterExpression* / *str*) – The parameters that are being assigned.

**Returns**

*AngledGate* – A new *Gate* of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

int – The number of qubits this quantum operator acts on.

**static ms**(*target1: QubitInput*, *target2: QubitInput*, *angle\_1: FreeParameterExpression* | *float*, *angle\_2: FreeParameterExpression* | *float*, *angle\_3: FreeParameterExpression* | *float* = 1.5707963267948966, \*, *control: QubitSetInput* | *None* = *None*, *control\_state: BasisStateInput* | *None* = *None*, *power: float* = 1) → Iterable[*Instruction*]

IonQ Mølmer-Sørensen gate.

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

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **angle\_1** (*Union*[*FreeParameterExpression*, *float*]) – angle in radians.
- **angle\_2** (*Union*[*FreeParameterExpression*, *float*]) – angle in radians.
- **angle\_3** (*Union*[*FreeParameterExpression*, *float*]) – angle in radians.
- **control** (*Optional*[*QubitSetInput*]) – Control qubit(s). Default `None`.
- **control\_state** (*Optional*[*BasisStateInput*]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if `control` is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

Iterable[*Instruction*] – MS instruction.

## Examples

```
>>> circ = Circuit().ms(0, 1, 0.15, 0.34)
```

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

### Raises

**NotImplementedError** – Not Implemented.

### Returns

*np.ndarray* – A matrix representation of the quantum operator

**class PRx**(*angle\_1*: [FreeParameterExpression](#) | *float*, *angle\_2*: [FreeParameterExpression](#) | *float*)

Bases: [DoubleAngledGate](#)

Phase Rx gate.

Unitary matrix:

$$\text{PRx}(\theta, \phi) = \begin{bmatrix} \cos(\theta/2) & -ie^{-i\phi} \sin(\theta/2) \\ -ie^{i\phi} \sin(\theta/2) & \cos(\theta/2) \end{bmatrix}.$$

### Parameters

- **angle\_1** (*Union*[[FreeParameterExpression](#), *float*]) – The first angle of the gate in radians or expression representation.
- **angle\_2** (*Union*[[FreeParameterExpression](#), *float*]) – The second angle of the gate in radians or expression representation.

Initiates a [DoubleAngledGate](#).

### Parameters

- **angle\_1** (*Union*[[FreeParameterExpression](#), *float*]) – The first angle of the gate in radians or expression representation.
- **angle\_2** (*Union*[[FreeParameterExpression](#), *float*]) – The second angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

### Raises

**ValueError** – If **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**, or **angle\_1** or **angle\_2** is None

**adjoint()** → list[[Gate](#)]

Returns the adjoint of this gate as a singleton list.



**Returns**

*list[Gate]* – A list containing the gate with negated angle.

**bind\_values**(\*\*kwargs) → *PRx*

Takes in parameters and attempts to assign them to values.

**Parameters**

**\*\*kwargs** (*FreeParameterExpression* | *str*) – The parameters that are being assigned.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count**() → *int*

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static prx**(target: *QubitSetInput*, angle\_1: *FreeParameterExpression* | *float*, angle\_2: *FreeParameterExpression* | *float*, \*, control: *QubitSetInput* | *None* = *None*, control\_state: *BasisStateInput* | *None* = *None*, power: *float* = 1) → *Iterable[Instruction]*

PhaseRx gate.

$$\text{PRx}(\theta, \phi) = \begin{bmatrix} \cos(\theta/2) & -ie^{-i\phi} \sin(\theta/2) \\ -ie^{i\phi} \sin(\theta/2) & \cos(\theta/2) \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle\_1** (*Union[FreeParameterExpression, float]*) – First angle in radians.
- **angle\_2** (*Union[FreeParameterExpression, float]*) – Second angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default *None*.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – PhaseRx instruction.

**Examples**

```
>>> circ = Circuit().prx(0, 0.15, 0.25)
```

**to\_matrix**() → *ndarray*

Returns a matrix representation of this gate.

**Returns**

*np.ndarray* – The matrix representation of this gate.

**class** PSwap(*angle*: FreeParameterExpression | float)

Bases: AngledGate

PSwap gate.

Unitary matrix:

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

#### Parameters

**angle** (Union[FreeParameterExpression, float]) – angle in radians.

Initializes an AngledGate.

#### Parameters

- **angle** (Union[FreeParameterExpression, float]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (Optional[int]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (Sequence[str]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – If the **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**, or **angle** is None

**bind\_values**(\*\*kwargs) → AngledGate

Takes in parameters and attempts to assign them to values.

#### Returns

AngledGate – A new Gate of the same type with the requested parameters bound.

#### Raises

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count**() → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

#### Returns

int – The number of qubits this quantum operator acts on.

**static pswap**(target1: QubitInput, target2: QubitInput, angle: FreeParameterExpression | float, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1) → Instruction

PSwap gate.

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

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – PSwap instruction.

**Examples**

```
>>> circ = Circuit().pswap(0, 1, 0.15)
```

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class PhaseShift**(*angle: FreeParameterExpression | float*)

Bases: *AngledGate*

Phase shift gate.

Unitary matrix:

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

**Parameters**

- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.

Initializes an AngledGate.

**Parameters**

- **angle** (*Union[FreeParameterExpression, float]*) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional[int]*) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on

the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, or `angle` is `None`

**bind\_values**(*\*\*kwargs*) → *AngledGate*

Takes in parameters and attempts to assign them to values.

#### Returns

*AngledGate* – A new Gate of the same type with the requested parameters bound.

#### Raises

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count**() → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**static phaseshift**(*target: QubitSetInput*, *angle: FreeParameterExpression | float, \*, control: QubitSetInput | None = None*, *control\_state: BasisStateInput | None = None*, *power: float = 1*) → *Iterable[Instruction]*

Phase shift gate.

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

#### Parameters

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default `None`.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in `control`. Will be ignored if `control` is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Iterable[Instruction]* – PhaseShift instruction.

### Examples

```
>>> circ = Circuit().phaseshift(0, 0.15)
```

**to\_matrix**() → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

#### Raises

**NotImplementError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class** `PulseGate`(*pulse\_sequence*: `PulseSequence`, *qubit\_count*: *int*, *display\_name*: *str* = 'PG')

Bases: `Gate`, `Parameterizable`

Arbitrary pulse gate which provides the ability to embed custom pulse sequences within circuits.

**Parameters**

- **pulse\_sequence** (`PulseSequence`) – `PulseSequence` to embed within the circuit.
- **qubit\_count** (*int*) – The number of qubits this pulse gate operates on.
- **display\_name** (*str*) – Name to be used for an instance of this pulse gate for circuit diagrams. Defaults to PG.

Initializes a `Gate`.

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`

**bind\_values**(*\*\*kwargs*) → `PulseGate`

Takes in parameters and returns an object with specified parameters replaced with their values.

**Returns**

`PulseGate` – A copy of this gate with the requested parameters bound.

**property parameters:** `list[FreeParameter]`

Returns the list of `FreeParameter`s associated with the gate.

**static pulse\_gate**(*targets*: `QubitSet`, *pulse\_sequence*: `PulseSequence`, *display\_name*: *str* = 'PG', \*, *control*: `QubitSetInput` | *None* = *None*, *control\_state*: `BasisStateInput` | *None* = *None*, *power*: *float* = 1) → `Instruction`

Arbitrary pulse gate which provides the ability to embed custom pulse sequences within circuits.

**Parameters**

- **targets** (`QubitSet`) – Target qubits. Note: These are only for representational purposes. The actual targets are determined by the frames used in the pulse sequence.
- **pulse\_sequence** (`PulseSequence`) – `PulseSequence` to embed within the circuit.
- **display\_name** (*str*) – Name to be used for an instance of this pulse gate for circuit diagrams. Defaults to PG.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if `control` is not present. May be represented as a string, list, or int. For example "0101", [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default "1" \* len(control).

- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – Pulse gate instruction.

**Examples**

```
>>> pulse_seq = PulseSequence().set_frequency(frame, frequency)....
>>> circ = Circuit().pulse_gate(pulse_sequence=pulse_seq, targets=[0])
```

**property pulse\_sequence:** *PulseSequence*

The underlying PulseSequence of this gate.

**Type**

*PulseSequence*

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class Rx**(*angle: FreeParameterExpression | float*)

Bases: *AngledGate*

X-axis rotation gate.

Unitary matrix:

$$R_x(\phi) = \begin{bmatrix} \cos(\phi/2) & -i \sin(\phi/2) \\ -i \sin(\phi/2) & \cos(\phi/2) \end{bmatrix}.$$

**Parameters**

**angle** (*Union[FreeParameterExpression, float]*) – angle in radians.

Initializes an AngledGate.

**Parameters**

- **angle** (*Union[FreeParameterExpression, float]*) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional[int]*) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`, or `angle` is None

**bind\_values**(\*\*kwargs) → *AngledGate*

Takes in parameters and attempts to assign them to values.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count**() → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static rx**(target: *QubitSetInput*, angle: *FreeParameterExpression* | *float*, \*, control: *QubitSetInput* | *None* = *None*, control\_state: *BasisStateInput* | *None* = *None*, power: *float* = 1) → *Iterable[Instruction]*

X-axis rotation gate.

$$R_x(\phi) = \begin{bmatrix} \cos(\phi/2) & -i \sin(\phi/2) \\ -i \sin(\phi/2) & \cos(\phi/2) \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union[FreeParameterExpression, float]*) – Angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – Rx instruction.

**Examples**

```
>>> circ = Circuit().rx(0, 0.15)
```

**to\_matrix**() → ndarray

Returns a matrix representation of this gate.

**Returns**

*np.ndarray* – The matrix representation of this gate.

**class Ry**(angle: *FreeParameterExpression* | *float*)

Bases: *AngledGate*

Y-axis rotation gate.

Unitary matrix:

$$R_y(\phi) = \begin{bmatrix} \cos(\phi/2) & -\sin(\phi/2) \\ \sin(\phi/2) & \cos(\phi/2) \end{bmatrix}.$$

**Parameters**

**angle** (*Union*[*FreeParameterExpression*, *float*]) – angle in radians.

Initializes an *AngledGate*.

**Parameters**

- **angle** (*Union*[*FreeParameterExpression*, *float*]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If the **qubit\_count** is less than 1, **ascii\_symbols** are *None*, or **ascii\_symbols** length != **qubit\_count**, or **angle** is *None*

**bind\_values**(\*\**kwargs*) → *AngledGate*

Takes in parameters and attempts to assign them to values.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count**() → *int*

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns *NotImplemented*.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static ry**(*target*: *QubitSetInput*, *angle*: *FreeParameterExpression* | *float*, \*, *control*: *QubitSetInput* | *None* = *None*, *control\_state*: *BasisStateInput* | *None* = *None*, *power*: *float* = 1) → *Iterable*[*Instruction*]

Y-axis rotation gate.

$$R_y(\phi) = \begin{bmatrix} \cos(\phi/2) & -\sin(\phi/2) \\ \sin(\phi/2) & \cos(\phi/2) \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union*[*FreeParameterExpression*, *float*]) – Angle in radians.
- **control** (*Optional*[*QubitSetInput*]) – Control qubit(s). Default *None*.
- **control\_state** (*Optional*[*BasisStateInput*]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string,



list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).

- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Iterable[Instruction]* – Rx instruction.

### Examples

```
>>> circ = Circuit().ry(0, 0.15)
```

**to\_matrix()** → ndarray

Returns a matrix representation of this gate.

#### Returns

*np.ndarray* – The matrix representation of this gate.

**class Rz**(*angle: FreeParameterExpression | float*)

Bases: *AngledGate*

Z-axis rotation gate.

Unitary matrix:

$$R_z(\phi) = \begin{bmatrix} e^{-i\phi/2} & 0 \\ 0 & e^{i\phi/2} \end{bmatrix}.$$

#### Parameters

**angle** (*Union[FreeParameterExpression, float]*) – angle in radians.

Initializes an AngledGate.

#### Parameters

- **angle** (*Union[FreeParameterExpression, float]*) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional[int]*) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – If the **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**, or **angle** is None

**bind\_values**(*\*\*kwargs*) → *AngledGate*

Takes in parameters and attempts to assign them to values.

#### Returns

*AngledGate* – A new Gate of the same type with the requested parameters bound.

#### Raises

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static rz**(*target: QubitSetInput*, *angle: FreeParameterExpression | float, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → *Iterable[Instruction]*

Z-axis rotation gate.

$$R_z(\phi) = \begin{bmatrix} e^{-i\phi/2} & 0 \\ 0 & e^{i\phi/2} \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union[FreeParameterExpression, float]*) – Angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default `None`.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – Rx instruction.

## Examples

```
>>> circ = Circuit().rz(0, 0.15)
```

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class S**

Bases: *Gate*

S gate.

Unitary matrix:

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}.$$

Initializes a [Gate](#).

#### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[[Gate](#)]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

#### Returns

list[[Gate](#)] – The gates comprising the adjoint of this gate.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

int – The number of qubits this quantum operator acts on.

**static s**(*target: QubitSetInput, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → Iterable[[Instruction](#)]

S gate.

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}.$$

#### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example "0101", [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default "1" \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

Iterable[[Instruction](#)] – Iterable of S instructions.

## Examples

```
>>> circ = Circuit().s(0)
>>> circ = Circuit().s([0, 1, 2])
```

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class Si**

Bases: [Gate](#)

Conjugate transpose of S gate.

Unitary matrix:

$$S^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix}.$$

Initializes a [Gate](#).

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have [“C”, “X”] to correlate a symbol with that index.

**Raises**

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[[Gate](#)]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

*list[Gate]* – The gates comprising the adjoint of this gate.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static si**(*target: QubitSetInput*, \*, *control: QubitSetInput | None = None*, *control\_state: BasisStateInput | None = None*, *power: float = 1*) → Iterable[*Instruction*]

Conjugate transpose of S gate.

$$S^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix}.$$

#### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Iterable[Instruction]* – Iterable of Si instructions.

#### Examples

```
>>> circ = Circuit().si(0)
>>> circ = Circuit().si([0, 1, 2])
```

**to\_matrix**() → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

#### Raises

**NotImplementedError** – Not Implemented.

#### Returns

*np.ndarray* – A matrix representation of the quantum operator

#### class Swap

Bases: *Gate*

Swap gate.

Unitary matrix:

$$\text{SWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Initializes a *Gate*.

#### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.

- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have [“C”, “X”] to correlate a symbol with that index.

**Raises**

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

list[*Gate*] – The gates comprising the adjoint of this gate.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

**Returns**

int – The number of qubits this quantum operator acts on.

**static swap**(*target1: QubitInput, target2: QubitInput, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → *Instruction*

Swap gate.

$$\text{SWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – Swap instruction.

## Examples

```
>>> circ = Circuit().swap(0, 1)
```

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class T**

Bases: [Gate](#)

T gate.

Unitary matrix:

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}.$$

Initializes a [Gate](#).

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have [“C”, “X”] to correlate a symbol with that index.

**Raises**

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[[Gate](#)]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

*list[Gate]* – The gates comprising the adjoint of this gate.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static** `t(target: QubitSetInput, *, control: QubitSetInput | None = None, control_state: BasisStateInput | None = None, power: float = 1) → Iterable[Instruction]`

T gate.

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}.$$

#### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default `None`.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Iterable[Instruction]* – Iterable of T instructions.

#### Examples

```
>>> circ = Circuit().t(0)
>>> circ = Circuit().t([0, 1, 2])
```

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

#### Raises

**NotImplementedError** – Not Implemented.

#### Returns

*np.ndarray* – A matrix representation of the quantum operator

#### class Ti

Bases: [Gate](#)

Conjugate transpose of T gate.

Unitary matrix:

$$T^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & e^{-i\pi/4} \end{bmatrix}.$$

Initializes a [Gate](#).

#### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if



CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – qubit\_count is less than 1, ascii\_symbols are None, or ascii\_symbols length != qubit\_count

**adjoint()** → list[Gate]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

#### Returns

list[Gate] – The gates comprising the adjoint of this gate.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

int – The number of qubits this quantum operator acts on.

**static ti**(target: QubitSetInput, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1) → Iterable[Instruction]

Conjugate transpose of T gate.

$$T^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & e^{-i\pi/4} \end{bmatrix}.$$

#### Parameters

- **target** (QubitSetInput) – Target qubit(s)
- **control** (Optional[QubitSetInput]) – Control qubit(s). Default None.
- **control\_state** (Optional[BasisStateInput]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example "0101", [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default "1" \* len(control).
- **power** (float) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

Iterable[Instruction] – Iterable of Ti instructions.

### Examples

```
>>> circ = Circuit().ti(0)
>>> circ = Circuit().ti([0, 1, 2])
```

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (Any) – Not Implemented.
- **\*\*kwargs** (Any) – Not Implemented.

#### Raises

**NotImplementedError** – Not Implemented.

#### Returns

np.ndarray – A matrix representation of the quantum operator

```
class U(angle_1: FreeParameterExpression | float, angle_2: FreeParameterExpression | float, angle_3:
    FreeParameterExpression | float)
```

Bases: *TripleAngledGate*

Generalized single-qubit rotation gate.

Unitary matrix:

$$U(\theta, \phi, \lambda) = \begin{bmatrix} \cos(\theta/2) & -e^{i\lambda} \sin(\theta/2) \\ e^{i\phi} \sin(\theta/2) & -e^{i(\phi+\lambda)} \cos(\theta/2) \end{bmatrix}.$$

#### Parameters

- **angle\_1** (*Union*[*FreeParameterExpression*, *float*]) – theta angle in radians.
- **angle\_2** (*Union*[*FreeParameterExpression*, *float*]) – phi angle in radians.
- **angle\_3** (*Union*[*FreeParameterExpression*, *float*]) – lambda angle in radians.

Initiates a *TripleAngledGate*.

#### Parameters

- **angle\_1** (*Union*[*FreeParameterExpression*, *float*]) – The first angle of the gate in radians or expression representation.
- **angle\_2** (*Union*[*FreeParameterExpression*, *float*]) – The second angle of the gate in radians or expression representation.
- **angle\_3** (*Union*[*FreeParameterExpression*, *float*]) – The third angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – If **qubit\_count** is less than 1, **ascii\_symbols** are *None*, or **ascii\_symbols** length != **qubit\_count**, or **angle\_1** or **angle\_2** or **angle\_3** is *None*

**adjoint**() → *list*[*Gate*]

Returns the adjoint of this gate as a singleton list.

#### Returns

*list*[*Gate*] – A list containing the gate with negated angle.

**bind\_values**(\*\**kwargs*) → *TripleAngledGate*

Takes in parameters and attempts to assign them to values.

#### Parameters

**\*\*kwargs** (*FreeParameterExpression* / *str*) – The parameters that are being assigned.

#### Returns

*AngledGate* – A new Gate of the same type with the requested parameters bound.

#### Raises

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**to\_matrix()** → ndarray

Returns a matrix representation of this gate.

**Returns**

*np.ndarray* – The matrix representation of this gate.

**static u**(target: *QubitSetInput*, angle\_1: *FreeParameterExpression* | *float*, angle\_2: *FreeParameterExpression* | *float*, angle\_3: *FreeParameterExpression* | *float*, \*, control: *QubitSetInput* | *None* = *None*, control\_state: *BasisStateInput* | *None* = *None*, power: *float* = 1) → *Iterable[Instruction]*

Generalized single-qubit rotation gate.

Unitary matrix:

$$U(\theta, \phi, \lambda) = \begin{bmatrix} \cos(\theta/2) & -e^{i\lambda} \sin(\theta/2) \\ e^{i\phi} \sin(\theta/2) & -e^{i(\phi+\lambda)} \cos(\theta/2) \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **angle\_1** (*Union[FreeParameterExpression, float]*) – theta angle in radians.
- **angle\_2** (*Union[FreeParameterExpression, float]*) – phi angle in radians.
- **angle\_3** (*Union[FreeParameterExpression, float]*) – lambda angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default *None*.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – U instruction.

## Examples

```
>>> circ = Circuit().u(0, 0.15, 0.34, 0.52)
```

**class Unitary**(matrix: ndarray, display\_name: str = 'U')

Bases: *Gate*

Arbitrary unitary gate.

**Parameters**

- **matrix** (*numpy.ndarray*) – Unitary matrix which defines the gate.
- **display\_name** (*str*) – Name to be used for an instance of this unitary gate for circuit diagrams. Defaults to U.

**Raises**

**ValueError** – If `matrix` is not a two-dimensional square matrix, or has a dimension length that is not a positive power of 2, or is not unitary.

Initializes a [Gate](#).

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`

**adjoint()** → list[[Gate](#)]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

list[[Gate](#)] – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

np.ndarray – A matrix representation of the quantum operator

**static unitary**(*targets: QubitSet, matrix: ndarray, display\_name: str = 'U'*) → [Instruction](#)

Arbitrary unitary gate.

**Parameters**

- **targets** ([QubitSet](#)) – Target qubits.
- **matrix** (*numpy.ndarray*) – Unitary matrix which defines the gate. Matrix should be compatible with the supplied targets, with `2 ** len(targets) == matrix.shape[0]`.
- **display\_name** (*str*) – Name to be used for an instance of this unitary gate for circuit diagrams. Defaults to U.

**Returns**

[Instruction](#) – Unitary instruction.

**Raises**

**ValueError** – If `matrix` is not a two-dimensional square matrix, or has a dimension length that is not compatible with the `targets`, or is not unitary,

## Examples

```
>>> circ = Circuit().unitary(matrix=np.array([[0, 1],[1, 0]]), targets=[0])
```

### class V

Bases: [Gate](#)

Square root of X gate (V gate).

Unitary matrix:

$$V = \frac{1}{2} \begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix}.$$

Initializes a [Gate](#).

#### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[[Gate](#)]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

#### Returns

list[[Gate](#)] – The gates comprising the adjoint of this gate.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

int – The number of qubits this quantum operator acts on.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

#### Raises

**NotImplementedError** – Not Implemented.

#### Returns

np.ndarray – A matrix representation of the quantum operator

**static** `v(target: QubitSetInput, *, control: QubitSetInput | None = None, control_state: BasisStateInput | None = None, power: float = 1) → Iterable[Instruction]`

Square root of X gate (V gate).

$$V = \frac{1}{2} \begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix}.$$

#### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default `None`.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in `control`. Will be ignored if `control` is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Iterable[Instruction]* – Iterable of V instructions.

#### Examples

```
>>> circ = Circuit().v(0)
>>> circ = Circuit().v([0, 1, 2])
```

#### class Vi

Bases: *Gate*

Conjugate transpose of square root of X gate (conjugate transpose of V).

Unitary matrix:

$$V^\dagger = \frac{1}{2} \begin{bmatrix} 1-i & 1+i \\ 1+i & 1-i \end{bmatrix}.$$

Initializes a *Gate*.

#### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have [“C”, “X”] to correlate a symbol with that index.

#### Raises

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length != `qubit_count`

**adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

*list[Gate]* – The gates comprising the adjoint of this gate.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**static vi**(*target: QubitSetInput, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → *Iterable[Instruction]*

Conjugate transpose of square root of X gate (conjugate transpose of  $V$ ).

$$V^\dagger = \frac{1}{2} \begin{bmatrix} 1-i & 1+i \\ 1+i & 1-i \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default `None`.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – Iterable of  $V_i$  instructions.

**Examples**

```
>>> circ = Circuit().vi(0)
>>> circ = Circuit().vi([0, 1, 2])
```

**class X**

Bases: [Gate](#)

Pauli-X gate.

Unitary matrix:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

Initializes a [Gate](#).

#### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have [“C”, “X”] to correlate a symbol with that index.

#### Raises

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[[Gate](#)]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

#### Returns

list[[Gate](#)] – The gates comprising the adjoint of this gate.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

int – The number of qubits this quantum operator acts on.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

#### Raises

**NotImplementedError** – Not Implemented.

#### Returns

np.ndarray – A matrix representation of the quantum operator

**static x**(*target: QubitSetInput, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → Iterable[[Instruction](#)]

Pauli-X gate.

Unitary matrix:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

#### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).



- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – Iterable of X instructions.

**Examples**

```
>>> circ = Circuit().x(0)
>>> circ = Circuit().x([0, 1, 2])
```

**class** **XX**(*angle*: [FreeParameterExpression](#) | *float*)

Bases: [AngledGate](#)

Ising XX coupling gate.

Unitary matrix:

$$XX(\phi) = \begin{bmatrix} \cos(\phi/2) & 0 & 0 & -i \sin(\phi/2) \\ 0 & \cos(\phi/2) & -i \sin(\phi/2) & 0 \\ 0 & -i \sin(\phi/2) & \cos(\phi/2) & 0 \\ -i \sin(\phi/2) & 0 & 0 & \cos(\phi/2) \end{bmatrix}.$$

Reference: <https://arxiv.org/abs/1707.06356>

**Parameters**

**angle** (*Union*[[FreeParameterExpression](#), *float*]) – angle in radians.

Initializes an [AngledGate](#).

**Parameters**

- **angle** (*Union*[[FreeParameterExpression](#), *float*]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If the **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**, or **angle** is None

**bind\_values**(\*\**kwargs*) → [AngledGate](#)

Takes in parameters and attempts to assign them to values.

**Returns**

[AngledGate](#) – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**to\_matrix()** → ndarray

Returns a matrix representation of this gate.

**Returns**

*np.ndarray* – The matrix representation of this gate.

**static xx**(*target1: QubitInput, target2: QubitInput, angle: FreeParameterExpression | float, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → *Instruction*

Ising XX coupling gate.

$$XX(\phi) = \begin{bmatrix} \cos(\phi/2) & 0 & 0 & -i \sin(\phi/2) \\ 0 & \cos(\phi/2) & -i \sin(\phi/2) & 0 \\ 0 & -i \sin(\phi/2) & \cos(\phi/2) & 0 \\ -i \sin(\phi/2) & 0 & 0 & \cos(\phi/2) \end{bmatrix}.$$

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default `None`.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in `control`. Will be ignored if `control` is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – XX instruction.

## Examples

```
>>> circ = Circuit().xx(0, 1, 0.15)
```

**class XY**(*angle: FreeParameterExpression | float*)

Bases: *AngledGate*

XY gate.

Unitary matrix:

$$XY(\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\phi/2) & i \sin(\phi/2) & 0 \\ 0 & i \sin(\phi/2) & \cos(\phi/2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Reference: <https://arxiv.org/abs/1912.04424v1>

#### Parameters

**angle** (*Union*[[FreeParameterExpression](#), *float*]) – angle in radians.

Initializes an `AngledGate`.

#### Parameters

- **angle** (*Union*[[FreeParameterExpression](#), *float*]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, or `angle` is `None`

**bind\_values**(*\*\*kwargs*) → [AngledGate](#)

Takes in parameters and attempts to assign them to values.

#### Returns

[AngledGate](#) – A new Gate of the same type with the requested parameters bound.

#### Raises

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count**() → *int*

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**to\_matrix**() → *ndarray*

Returns a matrix representation of this gate.

#### Returns

*np.ndarray* – The matrix representation of this gate.

**static xy**(*target1*: *QubitInput*, *target2*: *QubitInput*, *angle*: [FreeParameterExpression](#) | *float*, \*, *control*: *QubitSetInput* | *None* = *None*, *control\_state*: *BasisStateInput* | *None* = *None*, *power*: *float* = 1) → [Instruction](#)

XY gate.

$$XY(\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\phi/2) & i \sin(\phi/2) & 0 \\ 0 & i \sin(\phi/2) & \cos(\phi/2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

#### Parameters

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **angle** (*Union*[[FreeParameterExpression](#), *float*]) – angle in radians.

- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – XY instruction.

**Examples**

```
>>> circ = Circuit().xy(0, 1, 0.15)
```

**class Y**

Bases: *Gate*

Pauli-Y gate.

Unitary matrix:

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}.$$

Initializes a *Gate*.

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have [“C”, “X”] to correlate a symbol with that index.

**Raises**

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

list[*Gate*] – The gates comprising the adjoint of this gate.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

int – The number of qubits this quantum operator acts on.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (Any) – Not Implemented.
- **\*\*kwargs** (Any) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

np.ndarray – A matrix representation of the quantum operator

**static y**(target: QubitSetInput, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1) → Iterable[Instruction]

Pauli-Y gate.

Unitary matrix:

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}.$$

**Parameters**

- **target** (QubitSetInput) – Target qubit(s)
- **control** (Optional[QubitSetInput]) – Control qubit(s). Default None.
- **control\_state** (Optional[BasisStateInput]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (float) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

Iterable[Instruction] – Iterable of Y instructions.

## Examples

```
>>> circ = Circuit().y(0)
>>> circ = Circuit().y([0, 1, 2])
```

**class YY**(angle: FreeParameterExpression | float)

Bases: AngledGate

Ising YY coupling gate.

Unitary matrix:

$$YY(\phi) = \begin{bmatrix} \cos(\phi/2) & 0 & 0 & i \sin(\phi/2) \\ 0 & \cos(\phi/2) & -i \sin(\phi/2) & 0 \\ 0 & -i \sin(\phi/2) & \cos(\phi/2) & 0 \\ i \sin(\phi/2) & 0 & 0 & \cos(\phi/2) \end{bmatrix}.$$

Reference: <https://arxiv.org/abs/1707.06356>

**Parameters**

**angle** (Union[FreeParameterExpression, float]) – angle in radians.

Initializes an `AngledGate`.

#### Parameters

- **angle** (`Union[FreeParameterExpression, float]`) – The angle of the gate in radians or expression representation.
- **qubit\_count** (`Optional[int]`) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (`Sequence[str]`) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, or `angle` is `None`

**bind\_values**(*\*\*kwargs*) → *AngledGate*

Takes in parameters and attempts to assign them to values.

#### Returns

*AngledGate* – A new Gate of the same type with the requested parameters bound.

#### Raises

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count**() → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**to\_matrix**() → ndarray

Returns a matrix representation of this gate.

#### Returns

*np.ndarray* – The matrix representation of this gate.

**static yy**(*target1: QubitInput, target2: QubitInput, angle: FreeParameterExpression | float, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → *Instruction*

Ising YY coupling gate.

$$YY(\phi) = \begin{bmatrix} \cos(\phi/2) & 0 & 0 & i \sin(\phi/2) \\ 0 & \cos(\phi/2) & -i \sin(\phi/2) & 0 \\ 0 & -i \sin(\phi/2) & \cos(\phi/2) & 0 \\ i \sin(\phi/2) & 0 & 0 & \cos(\phi/2) \end{bmatrix}.$$

#### Parameters

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **angle** (`Union[FreeParameterExpression, float]`) – angle in radians.
- **control** (`Optional[QubitSetInput]`) – Control qubit(s). Default `None`.
- **control\_state** (`Optional[BasisStateInput]`) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if `control` is not present. May be represented as a string,

list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).

- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – YY instruction.

### Examples

```
>>> circ = Circuit().yy(0, 1, 0.15)
```

#### class Z

Bases: *Gate*

Pauli-Z gate.

Unitary matrix:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

Initializes a *Gate*.

#### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have [“C”, “X”] to correlate a symbol with that index.

#### Raises

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

#### Returns

list[*Gate*] – The gates comprising the adjoint of this gate.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

int – The number of qubits this quantum operator acts on.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.

- **\*\*kwargs** (Any) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**static z**(*target: QubitSetInput*, \*, *control: QubitSetInput | None = None*, *control\_state: BasisStateInput | None = None*, *power: float = 1*) → *Iterable[Instruction]*

Pauli-Z gate.

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if control is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – Iterable of Z instructions.

## Examples

```
>>> circ = Circuit().z(0)
>>> circ = Circuit().z([0, 1, 2])
```

**class ZZ**(*angle: FreeParameterExpression | float*)

Bases: *AngledGate*

Ising ZZ coupling gate.

Unitary matrix:

$$ZZ(\phi) = \begin{bmatrix} e^{-i\phi/2} & 0 & 0 & 0 \\ 0 & e^{i\phi/2} & 0 & 0 \\ 0 & 0 & e^{i\phi/2} & 0 \\ 0 & 0 & 0 & e^{-i\phi/2} \end{bmatrix}.$$

Reference: <https://arxiv.org/abs/1707.06356>

**Parameters**

**angle** (*Union[FreeParameterExpression, float]*) – angle in radians.

Initializes an AngledGate.

**Parameters**

- **angle** (*Union[FreeParameterExpression, float]*) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional[int]*) – The number of qubits that this gate interacts with.



- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`, or `angle` is None

**bind\_values**(*\*\*kwargs*) → *AngledGate*

Takes in parameters and attempts to assign them to values.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count**() → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**to\_matrix**() → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**static zz**(*target1: QubitInput, target2: QubitInput, angle: FreeParameterExpression | float, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → *Instruction*

Ising ZZ coupling gate.

$$ZZ(\phi) = \begin{bmatrix} e^{-i\phi/2} & 0 & 0 & 0 \\ 0 & e^{i\phi/2} & 0 & 0 \\ 0 & 0 & e^{i\phi/2} & 0 \\ 0 & 0 & 0 & e^{-i\phi/2} \end{bmatrix}.$$

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in control. Will be ignored if `control` is not present. May be represented as a string, list, or int. For example "0101", [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default "1" \* len(control).

- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – ZZ instruction.

**Examples**

```
>>> circ = Circuit().zz(0, 1, 0.15)
```

**braket.circuits.gate\_calibrations module**

```
class braket.circuits.gate_calibrations.GateCalibrations(pulse_sequences: dict[tuple[Gate, QubitSet], PulseSequence])
```

Bases: object

An object containing gate calibration data. The data represents the mapping on a particular gate on a set of qubits to its calibration to be used by a quantum device. This is represented by a dictionary with keys of `Tuple(Gate, QubitSet)` mapped to a `PulseSequence`.

Initiates a *GateCalibrations*.

**Parameters**

**pulse\_sequences** (*dict[tuple[Gate, QubitSet], PulseSequence]*) – A mapping containing a key of `(Gate, QubitSet)` mapped to the corresponding pulse sequence.

**property pulse\_sequences:** `dict[tuple[Gate, QubitSet], PulseSequence]`

Gets the mapping of `(Gate, Qubit)` to the corresponding `PulseSequence`.

**Returns**

*dict[tuple[Gate, QubitSet], PulseSequence]* – The calibration data Dictionary.

**copy()** → *GateCalibrations*

Returns a copy of the object.

**Returns**

*GateCalibrations* – a copy of the calibrations.

**filter**(*gates: list[Gate] | None = None, qubits: QubitSet | list[QubitSet] | None = None*) → *GateCalibrations*

Filters the data based on optional lists of gates and QubitSets.

**Parameters**

- **gates** (*list[Gate] | None*) – An optional list of gates to filter on.
- **qubits** (*QubitSet | list[QubitSet] | None*) – An optional `QubitSet` or list of `QubitSet` to filter on.

**Returns**

*GateCalibrations* – A filtered `GateCalibrations` object.

**to\_ir**(*calibration\_key: tuple[Gate, QubitSet] | None = None*) → str

Returns the default representation for the *GateCalibrations* object.

**Parameters**

**calibration\_key** (*tuple[Gate, QubitSet] | None*) – An optional key to get a specific default. Default: None

**Raises**

**ValueError** – Key does not exist in the *GateCalibrations* object.

**Returns**

*str* – the defcal string for the object.

**braket.circuits.gates module**

**class** `braket.circuits.gates.H`

Bases: *Gate*

Hadamard gate.

Unitary matrix:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Initializes a Gate.

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`

**adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

list[*Gate*] – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**static** `fixed_qubit_count()` → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**static** `h(target: QubitSetInput, *, control: QubitSetInput | None = None, control_state: BasisStateInput | None = None, power: float = 1)` → Iterable[*Instruction*]

Hadamard gate.

Unitary matrix:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

#### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default `None`.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Iterable[Instruction]* – Iterable of H instructions.

#### Examples

```
>>> circ = Circuit().h(0)
>>> circ = Circuit().h([0, 1, 2])
```

**class** `braket.circuits.gates.I`

Bases: *Gate*

Identity gate.

Unitary matrix:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Initializes a Gate.

#### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.

- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`

**adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

*list[Gate]* – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static i**(*target: QubitSetInput*, \*, *control: QubitSetInput | None = None*, *control\_state: BasisStateInput | None = None*, *power: float = 1*) → Iterable[*Instruction*]

Identity gate.

Unitary matrix:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in `control`. Will be ignored if `control` is not present. May be represented as a string, list, or int. For

example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).

- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Iterable[Instruction]* – Iterable of *I* instructions.

#### Examples

```
>>> circ = Circuit().i(0)
>>> circ = Circuit().i([0, 1, 2])
```

**class** `braket.circuits.gates.GPhase`(*angle*: `FreeParameterExpression` | *float*)

Bases: `AngledGate`

Global phase gate.

Unitary matrix:

$$\text{gphase}(\gamma) = e^{i\gamma} I_1 = \begin{bmatrix} e^{i\gamma} & \\ & \end{bmatrix}.$$

#### Parameters

**angle** (`Union[FreeParameterExpression, float]`) – angle in radians.

#### Raises

**ValueError** – If angle is not present

Initializes an `AngledGate`.

#### Parameters

- **angle** (`Union[FreeParameterExpression, float]`) – The angle of the gate in radians or expression representation.
- **qubit\_count** (`Optional[int]`) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (`Sequence[str]`) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`, or `angle` is None

**adjoint()** → `list[Gate]`

Returns the adjoint of this gate as a singleton list.

#### Returns

*list[Gate]* – A list containing the gate with negated angle.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**bind\_values(\*\*kwargs)** → *AngledGate*

Takes in parameters and attempts to assign them to values.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static gphase**(*angle*: *FreeParameterExpression* | *float*, \*, *control*: *QubitSetInput* | *None* = *None*, *control\_state*: *BasisStateInput* | *None* = *None*, *power*: *float* = *1*) → *Instruction* | *Iterable[Instruction]*

Global phase gate.

If the gate is applied with control/negative control modifiers, it is translated in an equivalent gate using the following definition: `phaseshift() = ctrl @ gphase()`. The rightmost control qubit is used for the translation. If the polarity of the rightmost control modifier is negative, the following identity is used: `negctrl @ gphase() q = x q; ctrl @ gphase() q; x q`.

Unitary matrix:

$$\text{gphase}(\gamma) = e^{i\gamma} I_1 = [e^{i\gamma}].$$

**Parameters**

- **angle** (*Union[FreeParameterExpression, float]*) – Phase in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default *None*.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* | *Iterable[Instruction]* – GPhase instruction.

**Examples**

```
>>> circ = Circuit().gphase(0.45)
```

**class** `braket.circuits.gates.X`

Bases: `Gate`

Pauli-X gate.

Unitary matrix:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

Initializes a Gate.

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`

**adjoint()** → list[`Gate`]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

*list[Gate]* – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator



**static** `fixed_qubit_count()` → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static** `x(target: QubitSetInput, *, control: QubitSetInput | None = None, control_state: BasisStateInput | None = None, power: float = 1)` → Iterable[*Instruction*]

Pauli-X gate.

Unitary matrix:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default `None`.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – Iterable of X instructions.

**Examples**

```
>>> circ = Circuit().x(0)
>>> circ = Circuit().x([0, 1, 2])
```

**class** `braket.circuits.gates.Y`

Bases: *Gate*

Pauli-Y gate.

Unitary matrix:

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}.$$

Initializes a Gate.

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.

- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`

**adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

list[*Gate*] – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

np.ndarray – A matrix representation of the quantum operator

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

int – The number of qubits this quantum operator acts on.

**static y**(*target: QubitSetInput*, \*, *control: QubitSetInput | None = None*, *control\_state: BasisStateInput | None = None*, *power: float = 1*) → Iterable[*Instruction*]

Pauli-Y gate.

Unitary matrix:

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in `control`. Will be ignored if `control` is not present. May be represented as a string, list, or int. For

example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).

- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Iterable[Instruction]* – Iterable of Y instructions.

#### Examples

```
>>> circ = Circuit().y(0)
>>> circ = Circuit().y([0, 1, 2])
```

**class** `braket.circuits.gates.Z`

Bases: `Gate`

Pauli-Z gate.

Unitary matrix:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

Initializes a Gate.

#### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have [“C”, “X”] to correlate a symbol with that index.

#### Raises

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`

**adjoint()** → list[`Gate`]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

#### Returns

*list[Gate]* – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

#### Raises

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static z**(*target: QubitSetInput, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → Iterable[[Instruction](#)]

Pauli-Z gate.

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default `None`.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – Iterable of Z instructions.

**Examples**

```
>>> circ = Circuit().z(0)
>>> circ = Circuit().z([0, 1, 2])
```

**class** `braket.circuits.gates.S`

Bases: [Gate](#)

S gate.

Unitary matrix:

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}.$$

Initializes a Gate.

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.

- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`

**adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

*list[Gate]* – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static s**(*target: QubitSetInput*, \*, *control: QubitSetInput | None = None*, *control\_state: BasisStateInput | None = None*, *power: float = 1*) → Iterable[*Instruction*]

S gate.

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in `control`. Will be ignored if `control` is not present. May be represented as a string, list, or int. For example "0101", [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default "1" \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – Iterable of S instructions.

**Examples**

```
>>> circ = Circuit().s(0)
>>> circ = Circuit().s([0, 1, 2])
```

**class** `braket.circuits.gates.Si`

Bases: `Gate`

Conjugate transpose of S gate.

Unitary matrix:

$$S^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix}.$$

Initializes a Gate.

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`

**adjoint()** → list[`Gate`]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

*list[Gate]* – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**static si**(*target: QubitSetInput*, \*, *control: QubitSetInput | None = None*, *control\_state: BasisStateInput | None = None*, *power: float = 1*) → Iterable[*Instruction*]

Conjugate transpose of S gate.

$$S^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix}.$$

#### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default `None`.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Iterable[Instruction]* – Iterable of Si instructions.

#### Examples

```
>>> circ = Circuit().si(0)
>>> circ = Circuit().si([0, 1, 2])
```

**class** braket.circuits.gates.T

Bases: *Gate*

T gate.

Unitary matrix:

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}.$$

Initializes a Gate.

#### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if

CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – qubit\_count is less than 1, ascii\_symbols are None, or ascii\_symbols length != qubit\_count

**adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

#### Returns

list[*Gate*] – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

#### Raises

**NotImplementedError** – Not Implemented.

#### Returns

np.ndarray – A matrix representation of the quantum operator

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

#### Returns

int – The number of qubits this quantum operator acts on.

**static t**(target: *QubitSetInput*, \*, control: *QubitSetInput* | *None* = *None*, control\_state: *BasisStateInput* | *None* = *None*, power: *float* = 1) → Iterable[*Instruction*]

T gate.

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}.$$

#### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional*[*QubitSetInput*]) – Control qubit(s). Default *None*.
- **control\_state** (*Optional*[*BasisStateInput*]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example "0101", [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default "1" \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

Iterable[*Instruction*] – Iterable of T instructions.



## Examples

```
>>> circ = Circuit().t(0)
>>> circ = Circuit().t([0, 1, 2])
```

**class** `braket.circuits.gates.Ti`

Bases: `Gate`

Conjugate transpose of T gate.

Unitary matrix:

$$T^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & e^{-i\pi/4} \end{bmatrix}.$$

Initializes a Gate.

### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

### Raises

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[`Gate`]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

### Returns

list[`Gate`] – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

### Raises

**NotImplementedError** – Not Implemented.

### Returns

np.ndarray – A matrix representation of the quantum operator

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static** **ti**(*target: QubitSetInput*, \*, *control: QubitSetInput | None = None*, *control\_state: BasisStateInput | None = None*, *power: float = 1*) → *Iterable[Instruction]*

Conjugate transpose of T gate.

$$T^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & e^{-i\pi/4} \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default *None*.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – Iterable of Ti instructions.

**Examples**

```
>>> circ = Circuit().ti(0)
>>> circ = Circuit().ti([0, 1, 2])
```

**class** **braket.circuits.gates.V**

Bases: *Gate*

Square root of X gate (V gate).

Unitary matrix:

$$V = \frac{1}{2} \begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix}.$$

Initializes a Gate.

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have [“C”, “X”] to correlate a symbol with that index.

**Raises**

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are *None*, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

*list[Gate]* – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static v**(*target: QubitSetInput, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → Iterable[*Instruction*]

Square root of X gate (V gate).

$$V = \frac{1}{2} \begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default **None**.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – Iterable of V instructions.

## Examples

```
>>> circ = Circuit().v(0)
>>> circ = Circuit().v([0, 1, 2])
```

**class** `braket.circuits.gates.Vi`

Bases: `Gate`

Conjugate transpose of square root of X gate (conjugate transpose of V).

Unitary matrix:

$$V^\dagger = \frac{1}{2} \begin{bmatrix} 1-i & 1+i \\ 1+i & 1-i \end{bmatrix}.$$

Initializes a Gate.

### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

### Raises

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[`Gate`]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

### Returns

list[`Gate`] – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

### Raises

**NotImplementedError** – Not Implemented.

### Returns

np.ndarray – A matrix representation of the quantum operator

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static vi**(*target: QubitSetInput*, \*, *control: QubitSetInput | None = None*, *control\_state: BasisStateInput | None = None*, *power: float = 1*) → *Iterable[Instruction]*

Conjugate transpose of square root of X gate (conjugate transpose of V).

$$V^\dagger = \frac{1}{2} \begin{bmatrix} 1-i & 1+i \\ 1+i & 1-i \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default *None*.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – Iterable of *Vi* instructions.

**Examples**

```
>>> circ = Circuit().vi(0)
>>> circ = Circuit().vi([0, 1, 2])
```

**class** `braket.circuits.gates.Rx`(*angle: FreeParameterExpression | float*)

Bases: [AngledGate](#)

X-axis rotation gate.

Unitary matrix:

$$R_x(\phi) = \begin{bmatrix} \cos(\phi/2) & -i \sin(\phi/2) \\ -i \sin(\phi/2) & \cos(\phi/2) \end{bmatrix}.$$

**Parameters**

**angle** (*Union[FreeParameterExpression, float]*) – angle in radians.

Initializes an `AngledGate`.

**Parameters**

- **angle** (*Union[FreeParameterExpression, float]*) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional[int]*) – The number of qubits that this gate interacts with.

- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`, or `angle` is None

**to\_matrix()** → ndarray

Returns a matrix representation of this gate.

**Returns**

*np.ndarray* – The matrix representation of this gate.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**bind\_values(\*\*kwargs)** → *AngledGate*

Takes in parameters and attempts to assign them to values.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static rx**(*target: QubitSetInput*, *angle: FreeParameterExpression | float*, \*, *control: QubitSetInput | None = None*, *control\_state: BasisStateInput | None = None*, *power: float = 1*) → Iterable[*Instruction*]

X-axis rotation gate.

$$R_x(\phi) = \begin{bmatrix} \cos(\phi/2) & -i \sin(\phi/2) \\ -i \sin(\phi/2) & \cos(\phi/2) \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union[FreeParameterExpression, float]*) – Angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in `control`. Will be ignored if `control` is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – Rx instruction.

## Examples

```
>>> circ = Circuit().rx(0, 0.15)
```

**class** `braket.circuits.gates.Ry`(*angle*: `FreeParameterExpression` | `float`)

Bases: `AngledGate`

Y-axis rotation gate.

Unitary matrix:

$$R_y(\phi) = \begin{bmatrix} \cos(\phi/2) & -\sin(\phi/2) \\ \sin(\phi/2) & \cos(\phi/2) \end{bmatrix}.$$

### Parameters

**angle** (`Union[FreeParameterExpression, float]`) – angle in radians.

Initializes an `AngledGate`.

### Parameters

- **angle** (`Union[FreeParameterExpression, float]`) – The angle of the gate in radians or expression representation.
- **qubit\_count** (`Optional[int]`) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (`Sequence[str]`) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, or `angle` is `None`

**to\_matrix**()  $\rightarrow$  `ndarray`

Returns a matrix representation of this gate.

### Returns

`np.ndarray` – The matrix representation of this gate.

**static fixed\_qubit\_count**()  $\rightarrow$  `int`

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

### Returns

`int` – The number of qubits this quantum operator acts on.

**bind\_values**(*\*\*kwargs*)  $\rightarrow$  `AngledGate`

Takes in parameters and attempts to assign them to values.

### Returns

`AngledGate` – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static ry**(*target*: *QubitSetInput*, *angle*: *FreeParameterExpression* | *float*, \*, *control*: *QubitSetInput* | *None* = *None*, *control\_state*: *BasisStateInput* | *None* = *None*, *power*: *float* = 1) → *Iterable[Instruction]*

Y-axis rotation gate.

$$R_y(\phi) = \begin{bmatrix} \cos(\phi/2) & -\sin(\phi/2) \\ \sin(\phi/2) & \cos(\phi/2) \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union[FreeParameterExpression, float]*) – Angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default *None*.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – Rx instruction.

**Examples**

```
>>> circ = Circuit().ry(0, 0.15)
```

**class** `braket.circuits.gates.Rz`(*angle*: *FreeParameterExpression* | *float*)

Bases: *AngledGate*

Z-axis rotation gate.

Unitary matrix:

$$R_z(\phi) = \begin{bmatrix} e^{-i\phi/2} & 0 \\ 0 & e^{i\phi/2} \end{bmatrix}.$$

**Parameters**

**angle** (*Union[FreeParameterExpression, float]*) – angle in radians.

Initializes an *AngledGate*.

**Parameters**

- **angle** (*Union[FreeParameterExpression, float]*) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional[int]*) – The number of qubits that this gate interacts with.



- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`, or `angle` is None

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**bind\_values(\*\*kwargs)** → *AngledGate*

Takes in parameters and attempts to assign them to values.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static rz**(*target: QubitSetInput*, *angle: FreeParameterExpression | float*, \*, *control: QubitSetInput | None = None*, *control\_state: BasisStateInput | None = None*, *power: float = 1*) → Iterable[*Instruction*]

Z-axis rotation gate.

$$R_z(\phi) = \begin{bmatrix} e^{-i\phi/2} & 0 \\ 0 & e^{i\phi/2} \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union[FreeParameterExpression, float]*) – Angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For

example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).

- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Iterable[Instruction]* – Rx instruction.

#### Examples

```
>>> circ = Circuit().rz(0, 0.15)
```

**class** `braket.circuits.gates.PhaseShift` (*angle*: `FreeParameterExpression` | *float*)

Bases: `AngledGate`

Phase shift gate.

Unitary matrix:

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

#### Parameters

**angle** (*Union*[`FreeParameterExpression`, *float*]) – angle in radians.

Initializes an `AngledGate`.

#### Parameters

- **angle** (*Union*[`FreeParameterExpression`, *float*]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length != `qubit_count`, or `angle` is `None`

**to\_matrix**() → *ndarray*

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

#### Raises

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**bind\_values**(\*\*kwargs) → *AngledGate*

Takes in parameters and attempts to assign them to values.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count**() → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static phaseshift**(target: *QubitSetInput*, angle: *FreeParameterExpression* | *float*, \*, control: *QubitSetInput* | *None* = *None*, control\_state: *BasisStateInput* | *None* = *None*, power: *float* = 1) → *Iterable[Instruction]*

Phase shift gate.

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

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default *None*.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – PhaseShift instruction.

**Examples**

```
>>> circ = Circuit().phaseshift(0, 0.15)
```

```
class braket.circuits.gates.U(angle_1: FreeParameterExpression | float, angle_2:
    FreeParameterExpression | float, angle_3: FreeParameterExpression | float)
```

Bases: *TripleAngledGate*

Generalized single-qubit rotation gate.

Unitary matrix:

$$U(\theta, \phi, \lambda) = \begin{bmatrix} \cos(\theta/2) & -e^{i\lambda} \sin(\theta/2) \\ e^{i\phi} \sin(\theta/2) & -e^{i(\phi+\lambda)} \cos(\theta/2) \end{bmatrix}.$$

**Parameters**

- **angle\_1** (*Union*[[FreeParameterExpression](#), *float*]) – theta angle in radians.
- **angle\_2** (*Union*[[FreeParameterExpression](#), *float*]) – phi angle in radians.
- **angle\_3** (*Union*[[FreeParameterExpression](#), *float*]) – lambda angle in radians.

Initializes a `TripleAngledGate`.

**Parameters**

- **angle\_1** (*Union*[[FreeParameterExpression](#), *float*]) – The first angle of the gate in radians or expression representation.
- **angle\_2** (*Union*[[FreeParameterExpression](#), *float*]) – The second angle of the gate in radians or expression representation.
- **angle\_3** (*Union*[[FreeParameterExpression](#), *float*]) – The third angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length != `qubit_count`, or `angle_1` or `angle_2` or `angle_3` is `None`

**to\_matrix()** → *ndarray*

Returns a matrix representation of this gate.

**Returns**

*np.ndarray* – The matrix representation of this gate.

**adjoint()** → *list*[[Gate](#)]

Returns the adjoint of this gate as a singleton list.

**Returns**

*list*[[Gate](#)] – A list containing the gate with negated angle.

**static fixed\_qubit\_count()** → *int*

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**bind\_values**(\*\*kwargs) → *TripleAngledGate*

Takes in parameters and attempts to assign them to values.

**Parameters**

**\*\*kwargs** (*FreeParameterExpression* / *str*) – The parameters that are being assigned.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static u**(target: *QubitSetInput*, angle\_1: *FreeParameterExpression* | *float*, angle\_2: *FreeParameterExpression* | *float*, angle\_3: *FreeParameterExpression* | *float*, \*, control: *QubitSetInput* | *None* = *None*, control\_state: *BasisStateInput* | *None* = *None*, power: *float* = 1) → *Iterable[Instruction]*

Generalized single-qubit rotation gate.

Unitary matrix:

$$U(\theta, \phi, \lambda) = \begin{bmatrix} \cos(\theta/2) & -e^{i\lambda} \sin(\theta/2) \\ e^{i\phi} \sin(\theta/2) & -e^{i(\phi+\lambda)} \cos(\theta/2) \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **angle\_1** (*Union[FreeParameterExpression, float]*) – theta angle in radians.
- **angle\_2** (*Union[FreeParameterExpression, float]*) – phi angle in radians.
- **angle\_3** (*Union[FreeParameterExpression, float]*) – lambda angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default *None*.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – U instruction.

**Examples**

```
>>> circ = Circuit().u(0, 0.15, 0.34, 0.52)
```

**class** `braket.circuits.gates.CNot`

Bases: *Gate*

Controlled NOT gate.

Unitary matrix:

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Initializes a Gate.

#### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – qubit\_count is less than 1, ascii\_symbols are None, or ascii\_symbols length != qubit\_count

**adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

#### Returns

list[*Gate*] – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

#### Raises

**NotImplementedError** – Not Implemented.

#### Returns

np.ndarray – A matrix representation of the quantum operator

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

#### Returns

int – The number of qubits this quantum operator acts on.

**static cnot**(control: *QubitSetInput*, target: *QubitInput*, power: float = 1) → *Instruction*

Controlled NOT gate.

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

**Parameters**

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – CNot instruction.

**Examples**

```
>>> circ = Circuit().cnot(0, 1)
```

**class** `braket.circuits.gates.Swap`

Bases: [Gate](#)

Swap gate.

Unitary matrix:

$$\text{SWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Initializes a Gate.

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`

**adjoint()** → list[[Gate](#)]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

*list[Gate]* – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.

- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**static fixed\_qubit\_count()** → *int*

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static swap**(*target1: QubitInput, target2: QubitInput, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → *Instruction*

Swap gate.

$$\text{SWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default *None*.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – Swap instruction.

**Examples**

```
>>> circ = Circuit().swap(0, 1)
```

**class** `braket.circuits.gates.ISwap`

Bases: *Gate*

ISwap gate.

Unitary matrix:



$$\text{iSWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & i & 0 \\ 0 & i & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Initializes a Gate.

#### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – qubit\_count is less than 1, ascii\_symbols are None, or ascii\_symbols length != qubit\_count

**adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

#### Returns

list[*Gate*] – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

#### Raises

**NotImplementedError** – Not Implemented.

#### Returns

np.ndarray – A matrix representation of the quantum operator

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

#### Returns

int – The number of qubits this quantum operator acts on.

**static iswap**(target1: *QubitInput*, target2: *QubitInput*, \*, control: *QubitSetInput* | None = None, control\_state: *BasisStateInput* | None = None, power: float = 1) → *Instruction*

ISwap gate.

$$\text{iSWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & i & 0 \\ 0 & i & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – ISwap instruction.

**Examples**

```
>>> circ = Circuit().iswap(0, 1)
```

**class** `braket.circuits.gates.PSwap`(*angle: FreeParameterExpression | float*)

Bases: [AngledGate](#)

PSwap gate.

Unitary matrix:

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

**Parameters**

**angle** (*Union[FreeParameterExpression, float]*) – angle in radians.

Initializes an AngledGate.

**Parameters**

- **angle** (*Union[FreeParameterExpression, float]*) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional[int]*) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If the **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**, or **angle** is None

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**bind\_values(\*\*kwargs)** → *AngledGate*

Takes in parameters and attempts to assign them to values.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static pswap**(*target1: QubitInput, target2: QubitInput, angle: FreeParameterExpression | float, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → *Instruction*

PSwap gate.

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

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default *None*.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – PSwap instruction.

## Examples

```
>>> circ = Circuit().pswap(0, 1, 0.15)
```

**class** `braket.circuits.gates.XY`(*angle*: `FreeParameterExpression` | `float`)

Bases: `AngledGate`

XY gate.

Unitary matrix:

$$XY(\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\phi/2) & i \sin(\phi/2) & 0 \\ 0 & i \sin(\phi/2) & \cos(\phi/2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Reference: <https://arxiv.org/abs/1912.04424v1>

### Parameters

**angle** (`Union[FreeParameterExpression, float]`) – angle in radians.

Initializes an `AngledGate`.

### Parameters

- **angle** (`Union[FreeParameterExpression, float]`) – The angle of the gate in radians or expression representation.
- **qubit\_count** (`Optional[int]`) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (`Sequence[str]`) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, or `angle` is `None`

**to\_matrix**()  $\rightarrow$  `ndarray`

Returns a matrix representation of this gate.

### Returns

`np.ndarray` – The matrix representation of this gate.

**bind\_values**(*\*\*kwargs*)  $\rightarrow$  `AngledGate`

Takes in parameters and attempts to assign them to values.

### Returns

`AngledGate` – A new Gate of the same type with the requested parameters bound.

### Raises

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**static xy**(*target1*: *QubitInput*, *target2*: *QubitInput*, *angle*: *FreeParameterExpression* | *float*, \*, *control*: *QubitSetInput* | *None* = *None*, *control\_state*: *BasisStateInput* | *None* = *None*, *power*: *float* = 1) → *Instruction*

XY gate.

$$XY(\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\phi/2) & i \sin(\phi/2) & 0 \\ 0 & i \sin(\phi/2) & \cos(\phi/2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

#### Parameters

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **angle** (*Union*[*FreeParameterExpression*, *float*]) – angle in radians.
- **control** (*Optional*[*QubitSetInput*]) – Control qubit(s). Default `None`.
- **control\_state** (*Optional*[*BasisStateInput*]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – XY instruction.

### Examples

```
>>> circ = Circuit().xy(0, 1, 0.15)
```

**class** `braket.circuits.gates.CPhaseShift`(*angle*: *FreeParameterExpression* | *float*)

Bases: *AngledGate*

Controlled phase shift gate.

Unitary matrix:

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

**Parameters**

**angle** (*Union*[*FreeParameterExpression*, *float*]) – angle in radians.

Initializes an AngledGate.

**Parameters**

- **angle** (*Union*[*FreeParameterExpression*, *float*]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If the **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**, or **angle** is None

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**bind\_values**(*\*\*kwargs*) → *AngledGate*

Takes in parameters and attempts to assign them to values.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static cphaseshift**(*control*: *QubitSetInput*, *target*: *QubitInput*, *angle*: *FreeParameterExpression* | *float*, *power*: *float* = 1) → *Instruction*

Controlled phase shift gate.

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

#### Parameters

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – CPhaseShift instruction.

#### Examples

```
>>> circ = Circuit().cphaseshift(0, 1, 0.15)
```

**class** `braket.circuits.gates.CPhaseShift00`(*angle*: *FreeParameterExpression* | *float*)

Bases: *AngledGate*

Controlled phase shift gate for phasing the  $|00\rangle$  state.

Unitary matrix:

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

#### Parameters

**angle** (*Union[FreeParameterExpression, float]*) – angle in radians.

Initializes an *AngledGate*.

#### Parameters

- **angle** (*Union[FreeParameterExpression, float]*) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional[int]*) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, or `angle` is `None`

**to\_matrix()**  $\rightarrow$  ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

np.ndarray – A matrix representation of the quantum operator

**bind\_values(\*\*kwargs)**  $\rightarrow$  *AngledGate*

Takes in parameters and attempts to assign them to values.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()**  $\rightarrow$  int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

int – The number of qubits this quantum operator acts on.

**static cphaseshift00**(*control: QubitSetInput, target: QubitInput, angle: FreeParameterExpression | float, power: float = 1*)  $\rightarrow$  *Instruction*

Controlled phase shift gate for phasing the  $|00\rangle$  state.

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

**Parameters**

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – CPhaseShift00 instruction.



## Examples

```
>>> circ = Circuit().cphaseshift00(0, 1, 0.15)
```

**class** `braket.circuits.gates.CPhaseShift01`(*angle*: `FreeParameterExpression` | `float`)

Bases: `AngledGate`

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

Unitary matrix:

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

### Parameters

**angle** (`Union[FreeParameterExpression, float]`) – angle in radians.

Initializes an `AngledGate`.

### Parameters

- **angle** (`Union[FreeParameterExpression, float]`) – The angle of the gate in radians or expression representation.
- **qubit\_count** (`Optional[int]`) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (`Sequence[str]`) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, or `angle` is `None`

**to\_matrix**()  $\rightarrow$  `ndarray`

Returns a matrix representation of the quantum operator.

### Parameters

- **\*args** (`Any`) – Not Implemented.
- **\*\*kwargs** (`Any`) – Not Implemented.

### Raises

**NotImplementedError** – Not Implemented.

### Returns

`np.ndarray` – A matrix representation of the quantum operator

**bind\_values**(**\*\*kwargs**)  $\rightarrow$  `AngledGate`

Takes in parameters and attempts to assign them to values.

### Returns

`AngledGate` – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static cphaseshift01**(*control: QubitSetInput, target: QubitInput, angle: FreeParameterExpression | float, power: float = 1*) → *Instruction*

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

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

**Parameters**

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – CPhaseShift01 instruction.

**Examples**

```
>>> circ = Circuit().cphaseshift01(0, 1, 0.15)
```

**class** `braket.circuits.gates.CPhaseShift10`(*angle: FreeParameterExpression | float*)

Bases: *AngledGate*

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

Unitary matrix:

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

**Parameters**

**angle** (*Union[FreeParameterExpression, float]*) – angle in radians.

Initializes an *AngledGate*.

**Parameters**

- **angle** (*Union*[*FreeParameterExpression*, *float*]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`, or `angle` is None

**to\_matrix()** → *ndarray*

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**bind\_values(\*\*kwargs)** → *AngledGate*

Takes in parameters and attempts to assign them to values.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()** → *int*

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static cphaseshift10**(*control: QubitSetInput*, *target: QubitInput*, *angle: FreeParameterExpression | float*, *power: float = 1*) → *Instruction*

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

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

**Parameters**

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.

- **target** (*QubitInput*) – Target qubit index.
- **angle** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – CPhaseShift10 instruction.

**Examples**

```
>>> circ = Circuit().cphaseshift10(0, 1, 0.15)
```

**class** `braket.circuits.gates.CV`

Bases: *Gate*

Controlled Sqrt of X gate.

Unitary matrix:

$$CV = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 + 0.5i & 0.5 - 0.5i \\ 0 & 0 & 0.5 - 0.5i & 0.5 + 0.5i \end{bmatrix}.$$

Initializes a Gate.

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`

**adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

list[*Gate*] – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises****NotImplementedError** – Not Implemented.**Returns***np.ndarray* – A matrix representation of the quantum operator**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.**Returns***int* – The number of qubits this quantum operator acts on.**static cv**(*control: QubitSetInput, target: QubitInput, power: float = 1*) → *Instruction*

Controlled Sqrt of X gate.

$$CV = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 + 0.5i & 0.5 - 0.5i \\ 0 & 0 & 0.5 - 0.5i & 0.5 + 0.5i \end{bmatrix}.$$

**Parameters**

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns***Instruction* – CV instruction.**Examples**

```
>>> circ = Circuit().cv(0, 1)
```

**class** `braket.circuits.gates.CY`Bases: *Gate*

Controlled Pauli-Y gate.

Unitary matrix:

$$CY = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{bmatrix}.$$

Initializes a Gate.

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.

- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`

**adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

*list[Gate]* – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static cy**(*control: QubitSetInput, target: QubitInput, power: float = 1*) → *Instruction*

Controlled Pauli-Y gate.

$$CY = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{bmatrix}.$$

**Parameters**

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – CY instruction.

## Examples

```
>>> circ = Circuit().cy(0, 1)
```

**class** `braket.circuits.gates.CZ`

Bases: `Gate`

Controlled Pauli-Z gate.

Unitary matrix:

$$CZ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}.$$

Initializes a Gate.

### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

### Raises

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**adjoint()** → list[`Gate`]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

### Returns

list[`Gate`] – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

### Raises

**NotImplementedError** – Not Implemented.

### Returns

np.ndarray – A matrix representation of the quantum operator

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static** **cz**(*control: QubitSetInput, target: QubitInput, power: float = 1*) → *Instruction*

Controlled Pauli-Z gate.

$$\text{CZ} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}.$$

**Parameters**

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target** (*QubitInput*) – Target qubit index.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – CZ instruction.

**Examples**

```
>>> circ = Circuit().cz(0, 1)
```

**class** `braket.circuits.gates.ECR`

Bases: *Gate*

An echoed RZX(pi/2) gate (ECR gate).

Unitary matrix:

$$\text{ECR} = \begin{bmatrix} 0 & 0 & 1 & i \\ 0 & 0 & i & 1 \\ 1 & -i & 0 & 0 \\ -i & 1 & 0 & 0 \end{bmatrix}.$$

Initializes a Gate.

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**



**adjoint()** → list[*Gate*]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

**Returns**

*list[Gate]* – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static ecr**(*target1: QubitInput, target2: QubitInput, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → *Instruction*

An echoed RZX(pi/2) gate (ECR gate).

$$\text{ECR} = \begin{bmatrix} 0 & 0 & 1 & i \\ 0 & 0 & i & 1 \\ 1 & -i & 0 & 0 \\ -i & 1 & 0 & 0 \end{bmatrix}.$$

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default **None**.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – ECR instruction.

## Examples

```
>>> circ = Circuit().ecr(0, 1)
```

**class** `braket.circuits.gates.XX`(*angle*: `FreeParameterExpression` | `float`)

Bases: `AngledGate`

Ising XX coupling gate.

Unitary matrix:

$$XX(\phi) = \begin{bmatrix} \cos(\phi/2) & 0 & 0 & -i \sin(\phi/2) \\ 0 & \cos(\phi/2) & -i \sin(\phi/2) & 0 \\ 0 & -i \sin(\phi/2) & \cos(\phi/2) & 0 \\ -i \sin(\phi/2) & 0 & 0 & \cos(\phi/2) \end{bmatrix}.$$

Reference: <https://arxiv.org/abs/1707.06356>

### Parameters

**angle** (`Union`[`FreeParameterExpression`, `float`]) – angle in radians.

Initializes an `AngledGate`.

### Parameters

- **angle** (`Union`[`FreeParameterExpression`, `float`]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (`Optional`[`int`]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (`Sequence`[`str`]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, or `angle` is `None`

**to\_matrix**()  $\rightarrow$  `ndarray`

Returns a matrix representation of this gate.

### Returns

`np.ndarray` – The matrix representation of this gate.

**bind\_values**(*\*\*kwargs*)  $\rightarrow$  `AngledGate`

Takes in parameters and attempts to assign them to values.

### Returns

`AngledGate` – A new Gate of the same type with the requested parameters bound.

### Raises

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**static xx**(*target1*: *QubitInput*, *target2*: *QubitInput*, *angle*: *FreeParameterExpression* | *float*, \*, *control*: *QubitSetInput* | *None* = *None*, *control\_state*: *BasisStateInput* | *None* = *None*, *power*: *float* = 1) → *Instruction*

Ising XX coupling gate.

$$XX(\phi) = \begin{bmatrix} \cos(\phi/2) & 0 & 0 & -i \sin(\phi/2) \\ 0 & \cos(\phi/2) & -i \sin(\phi/2) & 0 \\ 0 & -i \sin(\phi/2) & \cos(\phi/2) & 0 \\ -i \sin(\phi/2) & 0 & 0 & \cos(\phi/2) \end{bmatrix}.$$

#### Parameters

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **angle** (*Union*[*FreeParameterExpression*, *float*]) – angle in radians.
- **control** (*Optional*[*QubitSetInput*]) – Control qubit(s). Default *None*.
- **control\_state** (*Optional*[*BasisStateInput*]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – XX instruction.

#### Examples

```
>>> circ = Circuit().xx(0, 1, 0.15)
```

**class** `braket.circuits.gates.YY`(*angle*: *FreeParameterExpression* | *float*)

Bases: *AngledGate*

Ising YY coupling gate.

Unitary matrix:

$$YY(\phi) = \begin{bmatrix} \cos(\phi/2) & 0 & 0 & i \sin(\phi/2) \\ 0 & \cos(\phi/2) & -i \sin(\phi/2) & 0 \\ 0 & -i \sin(\phi/2) & \cos(\phi/2) & 0 \\ i \sin(\phi/2) & 0 & 0 & \cos(\phi/2) \end{bmatrix}.$$

Reference: <https://arxiv.org/abs/1707.06356>

**Parameters**

**angle** (*Union*[*FreeParameterExpression*, *float*]) – angle in radians.

Initializes an *AngledGate*.

**Parameters**

- **angle** (*Union*[*FreeParameterExpression*, *float*]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as *qubit\_count*, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If the *qubit\_count* is less than 1, *ascii\_symbols* are *None*, or *ascii\_symbols* length != *qubit\_count*, or *angle* is *None*

**to\_matrix()** → *ndarray*

Returns a matrix representation of this gate.

**Returns**

*np.ndarray* – The matrix representation of this gate.

**bind\_values(\*\*kwargs)** → *AngledGate*

Takes in parameters and attempts to assign them to values.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()** → *int*

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns *NotImplemented*.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static yy**(*target1*: *QubitInput*, *target2*: *QubitInput*, *angle*: *FreeParameterExpression* | *float*, \*, *control*: *QubitSetInput* | *None* = *None*, *control\_state*: *BasisStateInput* | *None* = *None*, *power*: *float* = 1) → *Instruction*

Ising YY coupling gate.

$$YY(\phi) = \begin{bmatrix} \cos(\phi/2) & 0 & 0 & i \sin(\phi/2) \\ 0 & \cos(\phi/2) & -i \sin(\phi/2) & 0 \\ 0 & -i \sin(\phi/2) & \cos(\phi/2) & 0 \\ i \sin(\phi/2) & 0 & 0 & \cos(\phi/2) \end{bmatrix}.$$

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.

- **angle** (*Union*[*FreeParameterExpression*, *float*]) – angle in radians.
- **control** (*Optional*[*QubitSetInput*]) – Control qubit(s). Default None.
- **control\_state** (*Optional*[*BasisStateInput*]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – YY instruction.

**Examples**

```
>>> circ = Circuit().yy(0, 1, 0.15)
```

**class** `braket.circuits.gates.ZZ`(*angle*: *FreeParameterExpression* | *float*)

Bases: *AngledGate*

Ising ZZ coupling gate.

Unitary matrix:

$$ZZ(\phi) = \begin{bmatrix} e^{-i\phi/2} & 0 & 0 & 0 \\ 0 & e^{i\phi/2} & 0 & 0 \\ 0 & 0 & e^{i\phi/2} & 0 \\ 0 & 0 & 0 & e^{-i\phi/2} \end{bmatrix}.$$

Reference: <https://arxiv.org/abs/1707.06356>

**Parameters**

**angle** (*Union*[*FreeParameterExpression*, *float*]) – angle in radians.

Initializes an *AngledGate*.

**Parameters**

- **angle** (*Union*[*FreeParameterExpression*, *float*]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If the **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**, or **angle** is None

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**bind\_values(\*\*kwargs)** → *AngledGate*

Takes in parameters and attempts to assign them to values.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static zz(target1: *QubitInput*, target2: *QubitInput*, angle: *FreeParameterExpression* | *float*, \*, control: *QubitSetInput* | *None* = *None*, control\_state: *BasisStateInput* | *None* = *None*, power: *float* = 1) → *Instruction***

Ising ZZ coupling gate.

$$ZZ(\phi) = \begin{bmatrix} e^{-i\phi/2} & 0 & 0 & 0 \\ 0 & e^{i\phi/2} & 0 & 0 \\ 0 & 0 & e^{i\phi/2} & 0 \\ 0 & 0 & 0 & e^{-i\phi/2} \end{bmatrix}.$$

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **angle** (*Union*[*FreeParameterExpression*, *float*]) – angle in radians.
- **control** (*Optional*[*QubitSetInput*]) – Control qubit(s). Default *None*.
- **control\_state** (*Optional*[*BasisStateInput*]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – ZZ instruction.

## Examples

```
>>> circ = Circuit().zz(0, 1, 0.15)
```

**class** `braket.circuits.gates.CCNot`

Bases: `Gate`

CCNOT gate or Toffoli gate.

Unitary matrix:

$$\text{CCNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

Initializes a Gate.

### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

### Raises

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`

**adjoint()** → list[`Gate`]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

### Returns

list[`Gate`] – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

### Raises

**NotImplementedError** – Not Implemented.

### Returns

np.ndarray – A matrix representation of the quantum operator

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**static ccnot**(*control1: QubitInput, control2: QubitInput, target: QubitInput, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → *Instruction*

CCNOT gate or Toffoli gate.

$$\text{CCNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

#### Parameters

- **control1** (*QubitInput*) – Control qubit 1 index.
- **control2** (*QubitInput*) – Control qubit 2 index.
- **target** (*QubitInput*) – Target qubit index.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s), in addition to control1 and control2. Default `None`.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Control state only applies to control qubits specified with the control argument, not control1 and control2. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – CCNot instruction.

#### Examples

```
>>> circ = Circuit().ccnot(0, 1, 2)
```

**class** `braket.circuits.gates.CSwap`

Bases: *Gate*

Controlled Swap gate.

Unitary matrix:



$$\text{CSWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Initializes a Gate.

#### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – qubit\_count is less than 1, ascii\_symbols are None, or ascii\_symbols length != qubit\_count

**adjoint()** → list[[Gate](#)]

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

#### Returns

list[[Gate](#)] – The gates comprising the adjoint of this gate.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

#### Raises

**NotImplementedError** – Not Implemented.

#### Returns

np.ndarray – A matrix representation of the quantum operator

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

#### Returns

int – The number of qubits this quantum operator acts on.

**static cswap**(*control*: *QubitSetInput*, *target1*: *QubitInput*, *target2*: *QubitInput*, *power*: *float* = 1) → *Instruction*

Controlled Swap gate.

$$\text{CSWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

#### Parameters

- **control** (*QubitSetInput*) – Control qubit(s). The last control qubit is absorbed into the target of the instruction.
- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – CSwap instruction.

#### Examples

```
>>> circ = Circuit().cswap(0, 1, 2)
```

**class** `braket.circuits.gates.GPi`(*angle*: *FreeParameterExpression* | *float*)

Bases: *AngledGate*

IonQ GPi gate.

Unitary matrix:

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

#### Parameters

**angle** (*Union*[*FreeParameterExpression*, *float*]) – angle in radians.

Initializes an *AngledGate*.

#### Parameters

- **angle** (*Union*[*FreeParameterExpression*, *float*]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and

index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`, or `angle` is None

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

#### Parameters

- **\*args** (Any) – Not Implemented.
- **\*\*kwargs** (Any) – Not Implemented.

#### Raises

**NotImplementedError** – Not Implemented.

#### Returns

*np.ndarray* – A matrix representation of the quantum operator

**adjoint()** → list[Gate]

Returns the adjoint of this gate as a singleton list.

#### Returns

*list[Gate]* – A list containing the gate with negated angle.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns **NotImplemented**.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**bind\_values(\*\*kwargs)** → GPI

Takes in parameters and attempts to assign them to values.

#### Returns

*AngledGate* – A new Gate of the same type with the requested parameters bound.

#### Raises

**NotImplementedError** – Subclasses should implement this function.

**static gpi**(*target: QubitSetInput, angle: FreeParameterExpression | float, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → Iterable[Instruction]

IonQ GPI gate.

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

#### Parameters

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union[FreeParameterExpression, float]*) – Angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.

- **control\_state** (*Optional* [*BasisStateInput*]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable* [*Instruction*] – GPI instruction.

**Examples**

```
>>> circ = Circuit().gpi(0, 0.15)
```

```
class braket.circuits.gates.PRx(angle_1: FreeParameterExpression | float, angle_2:
                                FreeParameterExpression | float)
```

Bases: *DoubleAngledGate*

Phase Rx gate.

Unitary matrix:

$$\text{PRx}(\theta, \phi) = \begin{bmatrix} \cos(\theta/2) & -ie^{-i\phi} \sin(\theta/2) \\ -ie^{i\phi} \sin(\theta/2) & \cos(\theta/2) \end{bmatrix}.$$

**Parameters**

- **angle\_1** (*Union* [*FreeParameterExpression*, *float*]) – The first angle of the gate in radians or expression representation.
- **angle\_2** (*Union* [*FreeParameterExpression*, *float*]) – The second angle of the gate in radians or expression representation.

Initiates a *DoubleAngledGate*.

**Parameters**

- **angle\_1** (*Union* [*FreeParameterExpression*, *float*]) – The first angle of the gate in radians or expression representation.
- **angle\_2** (*Union* [*FreeParameterExpression*, *float*]) – The second angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional* [*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence* [*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**, or **angle\_1** or **angle\_2** is None

**to\_matrix()** → ndarray

Returns a matrix representation of this gate.

**Returns**

*np.ndarray* – The matrix representation of this gate.

**adjoint()** → list[Gate]

Returns the adjoint of this gate as a singleton list.

**Returns**

*list[Gate]* – A list containing the gate with negated angle.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**bind\_values(\*\*kwargs)** → PRx

Takes in parameters and attempts to assign them to values.

**Parameters**

**\*\*kwargs** (*FreeParameterExpression* / *str*) – The parameters that are being assigned.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static prx**(*target: QubitSetInput*, *angle\_1: FreeParameterExpression | float*, *angle\_2: FreeParameterExpression | float*, \*, *control: QubitSetInput | None = None*, *control\_state: BasisStateInput | None = None*, *power: float = 1*) → Iterable[Instruction]

PhaseRx gate.

$$\text{PRx}(\theta, \phi) = \begin{bmatrix} \cos(\theta/2) & -ie^{-i\phi} \sin(\theta/2) \\ -ie^{i\phi} \sin(\theta/2) & \cos(\theta/2) \end{bmatrix}.$$

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle\_1** (*Union[FreeParameterExpression, float]*) – First angle in radians.
- **angle\_2** (*Union[FreeParameterExpression, float]*) – Second angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default `None`.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – PhaseRx instruction.

## Examples

```
>>> circ = Circuit().prx(0, 0.15, 0.25)
```

**class** `braket.circuits.gates.GPi2`(*angle*: `FreeParameterExpression` | `float`)

Bases: `AngledGate`

IonQ GPi2 gate.

Unitary matrix:

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

### Parameters

**angle** (`Union`[`FreeParameterExpression`, `float`]) – angle in radians.

Initializes an `AngledGate`.

### Parameters

- **angle** (`Union`[`FreeParameterExpression`, `float`]) – The angle of the gate in radians or expression representation.
- **qubit\_count** (`Optional`[`int`]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (`Sequence`[`str`]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, or `angle` is `None`

**to\_matrix**() → `ndarray`

Returns a matrix representation of the quantum operator.

### Parameters

- **\*args** (`Any`) – Not Implemented.
- **\*\*kwargs** (`Any`) – Not Implemented.

### Raises

**NotImplementedError** – Not Implemented.

### Returns

`np.ndarray` – A matrix representation of the quantum operator

**adjoint**() → `list`[`Gate`]

Returns the adjoint of this gate as a singleton list.

### Returns

`list`[`Gate`] – A list containing the gate with negated angle.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**bind\_values(\*\*kwargs)** → *GPi2*

Takes in parameters and attempts to assign them to values.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static gpi2**(target: *QubitSetInput*, angle: *FreeParameterExpression* | float, \*, control: *QubitSetInput* | None = None, control\_state: *BasisStateInput* | None = None, power: float = 1) → Iterable[*Instruction*]

IonQ GPi2 gate.

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

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s).
- **angle** (*Union*[*FreeParameterExpression*, float]) – Angle in radians.
- **control** (*Optional*[*QubitSetInput*]) – Control qubit(s). Default None.
- **control\_state** (*Optional*[*BasisStateInput*]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (float) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable*[*Instruction*] – GPi2 instruction.

**Examples**

```
>>> circ = Circuit().gpi2(0, 0.15)
```

```
class braket.circuits.gates.MS(angle_1: FreeParameterExpression | float, angle_2:
    FreeParameterExpression | float, angle_3: FreeParameterExpression | float
    = 1.5707963267948966)
```

Bases: *TripleAngledGate*

IonQ Mølmer-Sørensen gate.

Unitary matrix:

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

**Parameters**

- **angle\_1** (*Union*[[FreeParameterExpression](#), *float*]) – angle in radians.
- **angle\_2** (*Union*[[FreeParameterExpression](#), *float*]) – angle in radians.
- **angle\_3** (*Union*[[FreeParameterExpression](#), *float*]) – angle in radians. Default value is `angle_3=pi/2`.

Initiates a `TripleAngledGate`.

**Parameters**

- **angle\_1** (*Union*[[FreeParameterExpression](#), *float*]) – The first angle of the gate in radians or expression representation.
- **angle\_2** (*Union*[[FreeParameterExpression](#), *float*]) – The second angle of the gate in radians or expression representation.
- **angle\_3** (*Union*[[FreeParameterExpression](#), *float*]) – The third angle of the gate in radians or expression representation.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits that this gate interacts with.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the gate. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction. For instance, if a CNOT instruction has the control qubit on the first index and target qubit on the second index, the ASCII symbols should have ["C", "X"] to correlate a symbol with that index.

**Raises**

**ValueError** – If `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, or `angle_1` or `angle_2` or `angle_3` is `None`

**to\_matrix()**  $\rightarrow$  `ndarray`

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**adjoint()**  $\rightarrow$  `list`[[Gate](#)]

Returns the adjoint of this gate as a singleton list.

**Returns**

*list*[[Gate](#)] – A list containing the gate with negated angle.



**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**bind\_values(\*\*kwargs)** → *MS*

Takes in parameters and attempts to assign them to values.

**Parameters**

**\*\*kwargs** (*FreeParameterExpression* / *str*) – The parameters that are being assigned.

**Returns**

*AngledGate* – A new Gate of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**static ms**(*target1: QubitInput, target2: QubitInput, angle\_1: FreeParameterExpression | float, angle\_2: FreeParameterExpression | float, angle\_3: FreeParameterExpression | float = 1.5707963267948966, \*, control: QubitSetInput | None = None, control\_state: BasisStateInput | None = None, power: float = 1*) → *Iterable[Instruction]*

IonQ Mølmer-Sørensen gate.

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

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1 index.
- **target2** (*QubitInput*) – Target qubit 2 index.
- **angle\_1** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **angle\_2** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **angle\_3** (*Union[FreeParameterExpression, float]*) – angle in radians.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default `None`.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Iterable[Instruction]* – MS instruction.

## Examples

```
>>> circ = Circuit().ms(0, 1, 0.15, 0.34)
```

**class** `braket.circuits.gates.Unitary`(*matrix: ndarray, display\_name: str = 'U'*)

Bases: [Gate](#)

Arbitrary unitary gate.

### Parameters

- **matrix** (*numpy.ndarray*) – Unitary matrix which defines the gate.
- **display\_name** (*str*) – Name to be used for an instance of this unitary gate for circuit diagrams. Defaults to [U](#).

### Raises

**ValueError** – If **matrix** is not a two-dimensional square matrix, or has a dimension length that is not a positive power of 2, or is not unitary.

Initializes a Gate.

### Parameters

- **qubit\_count** (*Optional[int]*) – Number of qubits this gate interacts with.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as **qubit\_count**, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

### Raises

**ValueError** – **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**

**to\_matrix**() → *ndarray*

Returns a matrix representation of the quantum operator.

### Parameters

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

### Raises

**NotImplementedError** – Not Implemented.

### Returns

*np.ndarray* – A matrix representation of the quantum operator

**adjoint**() → *list[Gate]*

Returns a list of gates that implement the adjoint of this gate.

This is a list because some gates do not have an inverse defined by a single existing gate.

### Returns

*list[Gate]* – The gates comprising the adjoint of this gate.

**static unitary**(*targets: QubitSet, matrix: ndarray, display\_name: str = 'U'*) → [Instruction](#)

Arbitrary unitary gate.

### Parameters

- **targets** (`QubitSet`) – Target qubits.
- **matrix** (`numpy.ndarray`) – Unitary matrix which defines the gate. Matrix should be compatible with the supplied targets, with `2 ** len(targets) == matrix.shape[0]`.
- **display\_name** (`str`) – Name to be used for an instance of this unitary gate for circuit diagrams. Defaults to `U`.

**Returns**

*Instruction* – Unitary instruction.

**Raises**

**ValueError** – If `matrix` is not a two-dimensional square matrix, or has a dimension length that is not compatible with the `targets`, or is not unitary,

**Examples**

```
>>> circ = Circuit().unitary(matrix=np.array([[0, 1],[1, 0]]), targets=[0])
```

```
class braket.circuits.gates.PulseGate(pulse_sequence: PulseSequence, qubit_count: int, display_name: str = 'PG')
```

Bases: `Gate`, `Parameterizable`

Arbitrary pulse gate which provides the ability to embed custom pulse sequences within circuits.

**Parameters**

- **pulse\_sequence** (`PulseSequence`) – `PulseSequence` to embed within the circuit.
- **qubit\_count** (`int`) – The number of qubits this pulse gate operates on.
- **display\_name** (`str`) – Name to be used for an instance of this pulse gate for circuit diagrams. Defaults to `PG`.

Initializes a Gate.

**Parameters**

- **qubit\_count** (`Optional[int]`) – Number of qubits this gate interacts with.
- **ascii\_symbols** (`Sequence[str]`) – ASCII string symbols for the gate. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have [`"C"`, `"X"`] to correlate a symbol with that index.

**Raises**

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length  $\neq$  `qubit_count`

**property pulse\_sequence:** `PulseSequence`

The underlying `PulseSequence` of this gate.

**Type**

`PulseSequence`

**property parameters:** `list[FreeParameter]`

Returns the list of `FreeParameter` s associated with the gate.

**bind\_values**(\*\*kwargs) → *PulseGate*

Takes in parameters and returns an object with specified parameters replaced with their values.

**Returns**

*PulseGate* – A copy of this gate with the requested parameters bound.

**to\_matrix**() → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**static pulse\_gate**(targets: *QubitSet*, pulse\_sequence: *PulseSequence*, display\_name: str = 'PG', \*, control: *QubitSetInput* | None = None, control\_state: *BasisStateInput* | None = None, power: float = 1) → *Instruction*

**Arbitrary pulse gate which provides the ability to embed custom pulse sequences**  
within circuits.

**Parameters**

- **targets** (*QubitSet*) – Target qubits. Note: These are only for representational purposes. The actual targets are determined by the frames used in the pulse sequence.
- **pulse\_sequence** (*PulseSequence*) – *PulseSequence* to embed within the circuit.
- **display\_name** (*str*) – Name to be used for an instance of this pulse gate for circuit diagrams. Defaults to PG.
- **control** (*Optional[QubitSetInput]*) – Control qubit(s). Default None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in **control**. Will be ignored if **control** is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

**Returns**

*Instruction* – Pulse gate instruction.

## Examples

```
>>> pulse_seq = PulseSequence().set_frequency(frame, frequency)...
>>> circ = Circuit().pulse_gate(pulse_sequence=pulse_seq, targets=[0])
```

`braket.circuits.gates.format_complex(number: complex) → str`

Format a complex number into `<a> + <b>im` to be consumed by the braket unitary pragma

### Parameters

**number** (*complex*) – A complex number.

### Returns

*str* – The formatted string.

## braket.circuits.instruction module

```
class braket.circuits.instruction.Instruction(operator: InstructionOperator, target: QubitSetInput |
None = None, *, control: QubitSetInput | None = None,
control_state: BasisStateInput | None = None, power:
float = 1)
```

Bases: object

An instruction is a quantum directive that describes the quantum task to perform on a quantum device.

InstructionOperator includes objects of type Gate and Noise only.

### Parameters

- **operator** (*InstructionOperator*) – Operator for the instruction.
- **target** (*Optional[QubitSetInput]*) – Target qubits that the operator is applied to. Default is None.
- **control** (*Optional[QubitSetInput]*) – Target qubits that the operator is controlled on. Default is None.
- **control\_state** (*Optional[BasisStateInput]*) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in `control`. Will be ignored if `control` is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(control).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

### Raises

- **ValueError** – If `operator` is empty or any integer in `target` does not meet the Qubit or QubitSet class requirements. Also, if operator qubit count does not equal the size of the target qubit set.
- **TypeError** – If a Qubit class can’t be constructed from `target` due to an incorrect typing.

## Examples

```
>>> Instruction(Gate.CNOT(), [0, 1])
Instruction('operator': CNOT, 'target': QubitSet(Qubit(0), Qubit(1)))
>>> instr = Instruction(Gate.CNOT(), QubitSet([0, 1]))
Instruction('operator': CNOT, 'target': QubitSet(Qubit(0), Qubit(1)))
>>> instr = Instruction(Gate.H(), 0)
Instruction('operator': H, 'target': QubitSet(Qubit(0),))
>>> instr = Instruction(Gate.Rx(0.12), 0)
Instruction('operator': Rx, 'target': QubitSet(Qubit(0),))
>>> instr = Instruction(Gate.Rx(0.12, control=1), 0)
Instruction(
  'operator': Rx,
  'target': QubitSet(Qubit(0),),
  'control': QubitSet(Qubit(1),),
)
```

**property operator:** *Operator*

The operator for the instruction, for example, Gate.

**Type**

*Operator*

**property target:** *QubitSet*

Target qubits that the operator is applied to.

**Type**

*QubitSet*

**property control:** *QubitSet*

Target qubits that the operator is controlled on.

**Type**

*QubitSet*

**property control\_state:** *BasisState*

Quantum state that the operator is controlled to.

**Type**

*BasisState*

**property power:** float

Power that the operator is raised to.

**Type**

float

**adjoint()** → list[*Instruction*]

Returns a list of Instructions implementing adjoint of this instruction's own operator

This operation only works on Gate operators and compiler directives.

**Returns**

list[*Instruction*] – A list of new instructions that comprise the adjoint of this operator

**Raises**

**NotImplementedError** – If *operator* is not of type Gate or CompilerDirective

**to\_ir**(*ir\_type*: [IRType](#) = *IRType.JAQCD*, *serialization\_properties*: *SerializationProperties* | *None* = *None*)  
→ *Any*

Converts the operator into the canonical intermediate representation. If the operator is passed in a request, this method is called before it is passed.

#### Parameters

- **ir\_type** ([IRType](#)) – The *IRType* to use for converting the instruction object to its IR representation.
- **serialization\_properties** (*SerializationProperties* | *None*) – The serialization properties to use while serializing the object to the IR representation. The serialization properties supplied must correspond to the supplied *ir\_type*. Defaults to *None*.

#### Returns

*Any* – IR object of the instruction.

**property ascii\_symbols**: *tuple[str, ...]*

Returns the ascii symbols for the instruction’s operator.

#### Type

*tuple[str, ...]*

**copy**(*target\_mapping*: *dict*[*QubitInput*, *QubitInput*] | *None* = *None*, *target*: *QubitSetInput* | *None* = *None*, *control\_mapping*: *dict*[*QubitInput*, *QubitInput*] | *None* = *None*, *control*: *QubitSetInput* | *None* = *None*, *control\_state*: *BasisStateInput* | *None* = *None*, *power*: *float* = 1) → [Instruction](#)

Return a shallow copy of the instruction.

---

**Note:** If *target\_mapping* is specified, then *self.target* is mapped to the specified qubits. This is useful apply an instruction to a circuit and change the target qubits. Same relationship holds for *control\_mapping*.

---

#### Parameters

- **target\_mapping** (*Optional*[*dict*[*QubitInput*, *QubitInput*]]) – A dictionary of qubit mappings to apply to the target. Key is the qubit in this [target](#) and the value is what the key is changed to. Default = *None*.
- **target** (*Optional*[*QubitSetInput*]) – Target qubits for the new instruction. Default is *None*.
- **control\_mapping** (*Optional*[*dict*[*QubitInput*, *QubitInput*]]) – A dictionary of qubit mappings to apply to the control. Key is the qubit in this [control](#) and the value is what the key is changed to. Default = *None*.
- **control** (*Optional*[*QubitSetInput*]) – Control qubits for the new instruction. Default is *None*.
- **control\_state** (*Optional*[*BasisStateInput*]) – Quantum state on which to control the operation. Must be a binary sequence of same length as number of qubits in [control](#). Will be ignored if [control](#) is not present. May be represented as a string, list, or int. For example “0101”, [0, 1, 0, 1], 5 all represent controlling on qubits 0 and 2 being in the  $|0\rangle$  state and qubits 1 and 3 being in the  $|1\rangle$  state. Default “1” \* len(*control*).
- **power** (*float*) – Integer or fractional power to raise the gate to. Negative powers will be split into an inverse, accompanied by the positive power. Default 1.

#### Returns

*Instruction* – A shallow copy of the instruction.

**Raises**

**TypeError** – If both `target_mapping` and `target` are supplied.

**Examples**

```
>>> instr = Instruction(Gate.H(), 0)
>>> new_instr = instr.copy()
>>> new_instr.target
QubitSet(Qubit(0))
>>> new_instr = instr.copy(target_mapping={0: 5})
>>> new_instr.target
QubitSet(Qubit(5))
>>> new_instr = instr.copy(target=[5])
>>> new_instr.target
QubitSet(Qubit(5))
```

**braket.circuits.measure module**

**class** `braket.circuits.measure.Measure`(\*\*kwargs)

Bases: `QuantumOperator`

Class `Measure` represents a measure operation on targeted qubits

Initiates a `Measure`.

**Raises**

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are None, or `ascii_symbols` length != `qubit_count`

**property** `ascii_symbols`: `tuple[str]`

Returns the ascii symbols for the measure.

**Type**

`tuple[str]`

**to\_ir**(`target`: `QubitSet` | `None` = `None`, `ir_type`: `IRType` = `IRType.OPENQASM`, `serialization_properties`: `SerializationProperties` | `None` = `None`, \*\*kwargs) → `Any`

Returns IR object of the measure operator.

**Parameters**

- **target** (`QubitSet` / `None`) – target qubit(s). Defaults to `None`
- **ir\_type** (`IRType`) – The `IRType` to use for converting the measure object to its IR representation. Defaults to `IRType.OpenQASM`.
- **serialization\_properties** (`SerializationProperties` / `None`) – The serialization properties to use while serializing the object to the IR representation. The serialization properties supplied must correspond to the supplied `ir_type`. Defaults to `None`.

**Returns**

`Any` – IR object of the measure operator.

**Raises**

**ValueError** – If the supplied `ir_type` is not supported.



**braket.circuits.moments module**

```
class braket.circuits.moments.MomentType(value)
```

Bases: str, Enum

The type of moments. GATE: a gate NOISE: a noise channel added directly to the circuit GATE\_NOISE: a gate-based noise channel INITIALIZATION\_NOISE: a initialization noise channel READOUT\_NOISE: a readout noise channel COMPILER\_DIRECTIVE: an instruction to the compiler, external to the quantum program itself MEASURE: a measurement

```
GATE = 'gate'
```

```
NOISE = 'noise'
```

```
GATE_NOISE = 'gate_noise'
```

```
INITIALIZATION_NOISE = 'initialization_noise'
```

```
READOUT_NOISE = 'readout_noise'
```

```
COMPILER_DIRECTIVE = 'compiler_directive'
```

```
GLOBAL_PHASE = 'global_phase'
```

```
MEASURE = 'measure'
```

```
class braket.circuits.moments.MomentsKey(time: int, qubits: QubitSet, moment_type: MomentType,  
noise_index: int, subindex: int = 0)
```

Bases: NamedTuple

Key of the Moments mapping.

**Parameters**

- **time** – moment
- **qubits** – qubit set
- **moment\_type** – The type of the moment
- **noise\_index** – the number of noise channels at the same moment. For gates, this is the number of gate\_noise channels associated with that gate. For all other noise types, noise\_index starts from 0; but for gate noise, it starts from 1.

Create new instance of MomentsKey(time, qubits, moment\_type, noise\_index, subindex)

```
time: int
```

Alias for field number 0

```
qubits: QubitSet
```

Alias for field number 1

```
moment_type: MomentType
```

Alias for field number 2

```
noise_index: int
```

Alias for field number 3

```
subindex: int
```

Alias for field number 4

**class** `braket.circuits.moments.Moments`(*instructions: Iterable[Instruction] | None = None*)

Bases: `Mapping[MomentsKey, Instruction]`

An ordered mapping of `MomentsKey` or `NoiseMomentsKey` to `Instruction`. The core data structure that contains instructions, ordering they are inserted in, and time slices when they occur. `Moments` implements `Mapping` and functions the same as a read-only dictionary. It is mutable only through the `add()` method.

This data structure is useful to determine a dependency of instructions, such as printing or optimizing circuit structure, before sending it to a quantum device. The original insertion order is preserved and can be retrieved via the `values()` method.

#### Parameters

**instructions** (`Iterable[Instruction] | None`) – Instructions to initialize self. Default = None.

#### Examples

```
>>> moments = Moments()
>>> moments.add([Instruction(Gate.H(), 0), Instruction(Gate.CNOT(), [0, 1])])
>>> moments.add([Instruction(Gate.H(), 0), Instruction(Gate.H(), 1)])
>>> for i, item in enumerate(moments.items()):
...     print(f"Item {i}")
...     print(f"\tKey: {item[0]}")
...     print(f"\tValue: {item[1]}")
...
Item 0
  Key: MomentsKey(time=0, qubits=QubitSet([Qubit(0)]))
  Value: Instruction('operator': H, 'target': QubitSet([Qubit(0)]))
Item 1
  Key: MomentsKey(time=1, qubits=QubitSet([Qubit(0), Qubit(1)]))
  Value: Instruction('operator': CNOT, 'target': QubitSet([Qubit(0), Qubit(1)]))
Item 2
  Key: MomentsKey(time=2, qubits=QubitSet([Qubit(0)]))
  Value: Instruction('operator': H, 'target': QubitSet([Qubit(0)]))
Item 3
  Key: MomentsKey(time=2, qubits=QubitSet([Qubit(1)]))
  Value: Instruction('operator': H, 'target': QubitSet([Qubit(1)]))
```

**property depth:** `int`

Get the depth (number of slices) of self.

**Type**

`int`

**property qubit\_count:** `int`

Get the number of qubits used across all of the instructions.

**Type**

`int`

**property qubits:** `QubitSet`

Get the qubits used across all of the instructions. The order of qubits is based on the order in which the instructions were added.

---

**Note:** Don't mutate this object, any changes may impact the behavior of this class and / or consumers. If you need to mutate this, then copy it via `QubitSet(moments.qubits())`.

---

### Type

*QubitSet*

**time\_slices()** → dict[int, list[*Instruction*]]

Get instructions keyed by time.

### Returns

*dict[int, list[Instruction]]* – Key is the time and value is a list of instructions that occur at that moment in time. The order of instructions is in no particular order.

---

**Note:** This is a computed result over self and can be freely mutated. This is re-computed with every call, with a computational runtime  $O(N)$  where  $N$  is the number of instructions in self.

---

**add**(*instructions: Iterable[Instruction] | Instruction, noise\_index: int = 0*) → None

Add one or more instructions to self.

### Parameters

- **instructions** (*Union[Iterable[Instruction], Instruction]*) – Instructions to add to self. The instruction is added to the max time slice in which the instruction fits.
- **noise\_index** (*int*) – the number of noise channels at the same moment. For gates, this is the number of `gate_noise` channels associated with that gate. For all other noise types, `noise_index` starts from 0; but for gate noise, it starts from 1.

**add\_noise**(*instruction: Instruction, input\_type: str = 'noise', noise\_index: int = 0*) → None

Adds noise to a moment.

### Parameters

- **instruction** (*Instruction*) – Instruction to add.
- **input\_type** (*str*) – One of `MomentType`.
- **noise\_index** (*int*) – The number of noise channels at the same moment. For gates, this is the number of `gate_noise` channels associated with that gate. For all other noise types, `noise_index` starts from 0; but for gate noise, it starts from 1.

**sort\_moments()** → None

Make the disordered moments in order.

1. Make the readout noise in the end
2. Make the initialization noise at the beginning

**keys()** → *KeysView[MomentsKey]*

Return a view of self's keys.

**items()** → *ItemsView[MomentsKey, Instruction]*

Return a view of self's (key, instruction).

**values()** → `ValuesView[Instruction]`

Return a view of self's instructions.

**Returns**

`ValuesView[Instruction]` – The (in-order) instructions.

**get**(key: `MomentsKey`, default: `Any | None = None`) → `Instruction`

Get the instruction in self by key.

**Parameters**

- **key** (`MomentsKey`) – Key of the instruction to fetch.
- **default** (`Any | None`) – Value to return if key is not in moments. Default = None.

**Returns**

`Instruction` – moments[key] if key in moments, else default is returned.

## braket.circuits.noise module

**class** `braket.circuits.noise.Noise`(qubit\_count: `int | None`, ascii\_symbols: `Sequence[str]`)

Bases: `QuantumOperator`

Class `Noise` represents a noise channel that operates on one or multiple qubits. Noise are considered as building blocks of quantum circuits that simulate noise. It can be used as an operator in an `Instruction` object. It appears in the diagram when user prints a circuit with `Noise`. This class is considered the noise channel definition containing the metadata that defines what the noise channel is and what it does.

Initializes a `Noise` object.

**Parameters**

- **qubit\_count** (`Optional[int]`) – Number of qubits this noise channel interacts with.
- **ascii\_symbols** (`Sequence[str]`) – ASCII string symbols for this noise channel. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction.

**Raises**

**ValueError** – `qubit_count` is less than 1, `ascii_symbols` are None, or length of `ascii_symbols` is not equal to `qubit_count`

**property name:** `str`

Returns the name of the quantum operator

**Returns**

`str` – The name of the quantum operator as a string

**to\_ir**(target: `QubitSet`, ir\_type: `IRType` = `IRType.JAQCD`, serialization\_properties: `SerializationProperties | None` = `None`) → `Any`

Returns IR object of quantum operator and target

**Parameters**

- **target** (`QubitSet`) – target qubit(s)
- **ir\_type** (`IRType`) – The `IRType` to use for converting the noise object to its IR representation. Defaults to `IRType.JAQCD`.

- **serialization\_properties** (*SerializationProperties* | *None*) – The serialization properties to use while serializing the object to the IR representation. The serialization properties supplied must correspond to the supplied *ir\_type*. Defaults to *None*.

**Returns**

*Any* – IR object of the quantum operator and target

**Raises**

- **ValueError** – If the supplied *ir\_type* is not supported, or if the supplied serialization
- **properties don't correspond to the ir\_type.** –

**to\_matrix**(\*args, \*\*kwargs) → *Iterable*[*ndarray*]

Returns a list of matrices defining the Kraus matrices of the noise channel.

**Returns**

*Iterable*[*ndarray*] – list of matrices defining the Kraus matrices of the noise channel.

**classmethod from\_dict**(noise: *dict*) → *Noise*

Converts a dictionary representing an object of this class into an instance of this class.

**Parameters**

**noise** (*dict*) – A dictionary representation of an object of this class.

**Returns**

*Noise* – An object of this class that corresponds to the passed in dictionary.

**classmethod register\_noise**(noise: *type*[*Noise*]) → *None*

Register a noise implementation by adding it into the *Noise* class.

**Parameters**

**noise** (*type*[*Noise*]) – Noise class to register.

**class AmplitudeDamping**(gamma: *FreeParameterExpression* | *float*)

Bases: *DampingNoise*

AmplitudeDamping noise channel which transforms a density matrix *rho* according to:

$$\begin{aligned} & \rho \\ \rightarrow & E_0 \rho E_0^\dagger + E_1 \rho E_1^\dagger \end{aligned}$$

where

$$E_0 = \begin{matrix} \text{left}(\begin{matrix} \text{beginmatrix} 10 \\ 0 \\ \sqrt{1-\gamma} \end{matrix} \end{matrix}$$

$$\begin{matrix} \text{endmatrix} \\ \text{right} \end{matrix} E_1 = \begin{matrix} \text{left}(\begin{matrix} \text{beginmatrix} 0 \\ \sqrt{\gamma} \end{matrix} \end{matrix}$$

$$\begin{matrix} \text{endmatrix} \\ \text{right} \end{matrix}$$

This noise channel is shown as **AD** in circuit diagrams.

Initializes a [\*DampingNoise\*](#).

#### Parameters

- **gamma** (*Union*[[FreeParameterExpression](#), *float*]) – Probability of damping.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits to apply noise.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction.

#### Raises

**ValueError** – If **qubit\_count** < 1, **ascii\_symbols** is *None*, **len**(**ascii\_symbols**) != **qubit\_count**, **gamma** is not *float* or *FreeParameterExpression*, or **gamma** > 1.0 or **gamma** < 0.0.

**static amplitude\_damping**(*target*: [Qubit](#) | *int* | *Iterable*[[Qubit](#) | *int*], *gamma*: *float*) → *Iterable*[[Instruction](#)]

Registers this function into the circuit class.

#### Parameters

- **target** (*QubitSetInput*) – Target qubit(s).
- **gamma** (*float*) – decaying rate of the amplitude damping channel.

#### Returns

*Iterable*[[Instruction](#)] – *Iterable* of *AmplitudeDamping* instructions.

## Examples

```
>>> circ = Circuit().amplitude_damping(0, gamma=0.1)
```

**bind\_values**(\*\*kwargs) → *Noise*

Takes in parameters and attempts to assign them to values.

**Returns**

*Noise* – A new Noise object of the same type with the requested parameters bound.

**static fixed\_qubit\_count**() → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**classmethod from\_dict**(noise: dict) → *Noise*

Converts a dictionary representation of this class into this class.

**Parameters**

**noise** (*dict*) – The dictionary representation of this noise.

**Returns**

*Noise* – A Noise object that represents the passed in dictionary.

**to\_matrix**() → Iterable[ndarray]

Returns a matrix representation of this noise.

**Returns**

*Iterable[ndarray]* – A list of matrix representations of this noise.

**class BitFlip**(probability: *FreeParameterExpression* | float)

Bases: *SingleProbabilisticNoise*

Bit flip noise channel which transforms a density matrix

*rho* according to:

$$\begin{aligned} & \rho \\ & \rightarrow (1-p)\rho + pX\rho X \\ & \text{dagger} \end{aligned}$$

where

```

I =
left(
beginmatrix10

01
endmatrix
right)
X =
left(
beginmatrix01

10
endmatrix
right)
p =
textprobability

```

This noise channel is shown as BF in circuit diagrams.

Initializes a [SingleProbabilisticNoise](#).

#### Parameters

- **probability** (*Union*[[FreeParameterExpression](#), *float*]) – The probability that the noise occurs.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits to apply noise.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction.
- **max\_probability** (*float*) – Maximum allowed probability of the noise channel. Default: 0.5

#### Raises

**ValueError** – If the **qubit\_count** is less than 1, **ascii\_symbols** are *None*, or **ascii\_symbols** length  $\neq$  **qubit\_count**, **probability** is not *float* or *FreeParameterExpression*, **probability**  $> 1/2$ , or **probability**  $< 0$

**bind\_values**(*\*\*kwargs*: [FreeParameter](#) | *str*)  $\rightarrow$  *Noise*

Takes in parameters and attempts to assign them to values.

#### Parameters

**\*\*kwargs** (*Union*[[FreeParameter](#), *str*]) – Arbitrary keyword arguments.

#### Returns

*Noise* – A new Noise object of the same type with the requested parameters bound.

**static bit\_flip**(*target*: [Qubit](#) | *int* | *Iterable*[[Qubit](#) | *int*], *probability*: *float*)  $\rightarrow$  *Iterable*[[Instruction](#)]

Registers this function into the circuit class.

#### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **probability** (*float*) – Probability of bit flipping.

#### Returns

*Iterable*[[Instruction](#)] – Iterable of BitFlip instructions.



## Examples

```
>>> circ = Circuit().bit_flip(0, probability=0.1)
```

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**classmethod from\_dict(noise: dict)** → *Noise*

Converts a dictionary representation of this class into this class.

**Parameters**

**noise** (*dict*) – The dictionary representation of this noise.

**Returns**

*Noise* – A Noise object that represents the passed in dictionary.

**to\_matrix()** → Iterable[ndarray]

Returns a matrix representation of this noise.

**Returns**

*Iterable[ndarray]* – A list of matrix representations of this noise.

**class Depolarizing(probability: FreeParameterExpression | float)**

Bases: *SingleProbabilisticNoise\_34*

Depolarizing noise channel which transforms a density matrix *rho* according to:

$$\begin{aligned} & \rho \\ \rightarrow & (1-p)\rho \\ & + \frac{p}{3}X\rho X \\ & + \frac{p}{3}Y\rho Y \\ & + \frac{p}{3}Z\rho Z \end{aligned}$$

where

```

        I =
        left(
beginmatrix10

        01
        endmatrix
        right)
        X =
        left(
beginmatrix01

        10
        endmatrix
        right)
        Y =
        left(
beginmatrix0 - i

        i0
        endmatrix
        right)
        Z =
        left(
beginmatrix10

        0 - 1
        endmatrix
        right)
        p =
        textprobability
```

This noise channel is shown as DEPO in circuit diagrams.

Initializes a [\*SingleProbabilisticNoise\\_34\*](#).

#### Parameters

- **probability** (*Union[FreeParameterExpression, float]*) – The probability that the noise occurs.
- **qubit\_count** (*Optional[int]*) – The number of qubits to apply noise.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction.

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`,

or `ascii_symbols` length  $\neq$  `qubit_count`, `probability` is not float or `FreeParameterExpression`, `probability > 3/4`, or `probability < 0`

**bind\_values**(\*\*kwargs) → *Noise*

Takes in parameters and attempts to assign them to values.

**Returns**

*Noise* – A new Noise object of the same type with the requested parameters bound.

**static depolarizing**(target: *Qubit* | *int* | *Iterable*[*Qubit* | *int*], probability: *float*) → *Iterable*[*Instruction*]

Registers this function into the circuit class.

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **probability** (*float*) – Probability of depolarizing.

**Returns**

*Iterable*[*Instruction*] – Iterable of Depolarizing instructions.

## Examples

```
>>> circ = Circuit().depolarizing(0, probability=0.1)
```

**static fixed\_qubit\_count**() → *int*

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**classmethod from\_dict**(noise: *dict*) → *Noise*

Converts a dictionary representation of this class into this class.

**Parameters**

**noise** (*dict*) – The dictionary representation of this noise.

**Returns**

*Noise* – A Noise object that represents the passed in dictionary.

**to\_matrix**() → *Iterable*[*ndarray*]

Returns a matrix representation of this noise.

**Returns**

*Iterable*[*ndarray*] – A list of matrix representations of this noise.

**class GeneralizedAmplitudeDamping**(gamma: *FreeParameterExpression* | *float*, probability: *FreeParameterExpression* | *float*)

Bases: *GeneralizedAmplitudeDampingNoise*

**Generalized AmplitudeDamping noise channel which transforms a density matrix  $\rho$  according to:**

$\rho$   
 $\rightarrow E_0$   
 $\rho E_0$   
 $\dagger + E_1$   
 $\rho E_1$   
 $\dagger + E_2$   
 $\rho E_2$   
 $\dagger + E_3$   
 $\rho E_3$   
 $\dagger$

where

$$E_0 = \sqrt{\text{probability}} \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1 - \text{probability}} \end{pmatrix}$$

$$E_1 = \sqrt{\text{probability}} \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1 - \text{probability}} \end{pmatrix}$$

$$E_2 = \sqrt{1 - \text{probability}} \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1 - \text{probability}} \end{pmatrix}$$

$$E_3 = \sqrt{1 - \text{probability}} \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1 - \text{probability}} \end{pmatrix}$$

This noise channel is shown as GAD in circuit diagrams.

Initiates a `GeneralizedAmplitudeDampingNoise`.

**Parameters**

- **gamma** (*Union*[[FreeParameterExpression](#), *float*]) – Probability of damping.
- **probability** (*Union*[[FreeParameterExpression](#), *float*]) – Probability of the system being excited by the environment.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits to apply noise.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction.

**Raises**

**ValueError** – If **qubit\_count** < 1, **ascii\_symbols** is None, `len(ascii_symbols) != qubit_count`, **probability** or **gamma** is not *float* or [FreeParameterExpression](#), **probability** > 1.0 or **probability** < 0.0, or **gamma** > 1.0 or **gamma** < 0.0.

**bind\_values**(*\*\*kwargs*) → [Noise](#)

Takes in parameters and attempts to assign them to values.

**Returns**

*Noise* – A new Noise object of the same type with the requested parameters bound.

**static fixed\_qubit\_count**() → *int*

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**classmethod from\_dict**(*noise: dict*) → [Noise](#)

Converts a dictionary representation of this class into this class.

**Parameters**

**noise** (*dict*) – The dictionary representation of this noise.

**Returns**

*Noise* – A Noise object that represents the passed in dictionary.

**static generalized\_amplitude\_damping**(*target: Qubit | int | Iterable[Qubit | int]*, *gamma: float*, *probability: float*) → *Iterable[Instruction]*

Registers this function into the circuit class.

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s).
- **gamma** (*float*) – The damping rate of the amplitude damping channel.
- **probability** (*float*) – Probability of the system being excited by the environment.

**Returns**

*Iterable[Instruction]* – Iterable of GeneralizedAmplitudeDamping instructions.

## Examples

```
>>> circ = Circuit().generalized_amplitude_damping(0, gamma=0.1,
↪ probability = 0.9)
```

**to\_matrix()** → Iterable[ndarray]

Returns a matrix representation of this noise.

### Returns

*Iterable[ndarray]* – A list of matrix representations of this noise.

**class Kraus**(*matrices: Iterable[ndarray]*, *display\_name: str = 'KR'*)

Bases: *Noise*

User-defined noise channel that uses the provided matrices as Kraus operators This noise channel is shown as NK in circuit diagrams.

Init: Kraus.

### Parameters

- **matrices** (*Iterable[ndarray]*) – A list of matrices that define a noise channel. These matrices need to satisfy the requirement of CPTP map.
- **display\_name** (*str*) – Name to be used for an instance of this general noise channel for circuit diagrams. Defaults to KR.

### Raises

**ValueError** – If any matrix in **matrices** is not a two-dimensional square matrix, or has a dimension length which is not a positive exponent of 2, or the **matrices** do not satisfy CPTP condition.

**classmethod from\_dict**(*noise: dict*) → *Noise*

Converts a dictionary representation of this class into this class.

### Parameters

**noise** (*dict*) – The dictionary representation of this noise.

### Returns

*Noise* – A Noise object that represents the passed in dictionary.

**static kraus**(*targets: Qubit | int | Iterable[Qubit | int]*, *matrices: Iterable[array]*, *display\_name: str = 'KR'*) → Iterable[*Instruction*]

Registers this function into the circuit class.

### Parameters

- **targets** (*QubitSetInput*) – Target qubit(s)
- **matrices** (*Iterable[array]*) – Matrices that define a general noise channel.
- **display\_name** (*str*) – The display name.

### Returns

*Iterable[Instruction]* – Iterable of Kraus instructions.

## Examples

```
>>> K0 = np.eye(4) * np.sqrt(0.9)
>>> K1 = np.kron([[1., 0.], [0., 1.]], [[0., 1.], [1., 0.]]) * np.sqrt(0.1)
>>> circ = Circuit().kraus([1, 0], matrices=[K0, K1])
```

**to\_dict()** → dict

Converts this object into a dictionary representation. Not implemented at this time.

**Returns**

*dict* – Not implemented at this time..

**to\_matrix()** → Iterable[ndarray]

Returns a matrix representation of this noise.

**Returns**

*Iterable[ndarray]* – A list of matrix representations of this noise.

**class PauliChannel**(*probX*: [FreeParameterExpression](#) | float, *probY*: [FreeParameterExpression](#) | float, *probZ*: [FreeParameterExpression](#) | float)

Bases: [PauliNoise](#)

Pauli noise channel which transforms a density matrix

*rho* according to:

$$\begin{aligned} & \rho \\ \rightarrow & (1 - \text{probX} - \text{probY} - \text{probZ}) \\ & \rho + \text{probXX} \\ & \rho X \\ & \text{dagger} + \text{probYY} \\ & \rho Y \\ & \text{dagger} + \text{probZZ} \\ & \rho Z \\ & \text{dagger} \end{aligned}$$



where

$$\begin{aligned}
 I &= \\
 &\text{left}( \\
 &\text{beginmatrix} 10 \\
 &01 \\
 &\text{endmatrix} \\
 &\text{right}) \\
 X &= \\
 &\text{left}( \\
 &\text{beginmatrix} 01 \\
 &10 \\
 &\text{endmatrix} \\
 &\text{right}) \\
 Y &= \\
 &\text{left}( \\
 &\text{beginmatrix} 0-i \\
 &i0 \\
 &\text{endmatrix} \\
 &\text{right}) \\
 Z &= \\
 &\text{left}( \\
 &\text{beginmatrix} 10 \\
 &0-1 \\
 &\text{endmatrix} \\
 &\text{right})
 \end{aligned}$$

This noise channel is shown as PC in circuit diagrams.

Creates PauliChannel noise.

#### Parameters

- **probX** (*Union* [[FreeParameterExpression](#), *float*]) – X rotation probability.
- **probY** (*Union* [[FreeParameterExpression](#), *float*]) – Y rotation probability.
- **probZ** (*Union* [[FreeParameterExpression](#), *float*]) – Z rotation probability.

**bind\_values**(*\*\*kwargs*) → *Noise*

Takes in parameters and attempts to assign them to values.

#### Returns

*Noise* – A new Noise object of the same type with the requested parameters bound.

**static fixed\_qubit\_count**() → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**classmethod** `from_dict`(*noise: dict*) → *Noise*

Converts a dictionary representation of this class into this class.

**Parameters**

**noise** (*dict*) – The dictionary representation of this noise.

**Returns**

*Noise* – A Noise object that represents the passed in dictionary.

**static** `pauli_channel`(*target: Qubit | int | Iterable[Qubit | int]*, *probX: float*, *probY: float*, *probZ: float*) → *Iterable[Instruction]*

Registers this function into the circuit class.

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s) probability list[*float*]: Probabilities for the Pauli X, Y and Z noise happening in the Kraus channel.
- **probX** (*float*) – X rotation probability.
- **probY** (*float*) – Y rotation probability.
- **probZ** (*float*) – Z rotation probability.

**Returns**

*Iterable[Instruction]* – Iterable of PauliChannel instructions.

## Examples

```
>>> circ = Circuit().pauli_channel(0, probX=0.1, probY=0.2, probZ=0.3)
```

**to\_matrix**() → *Iterable[ndarray]*

Returns a matrix representation of this noise.

**Returns**

*Iterable[ndarray]* – A list of matrix representations of this noise.

**class** `PhaseDamping`(*gamma: FreeParameterExpression | float*)

Bases: *DampingNoise*

Phase damping noise channel which transforms a density matrix *rho* according to:

$$\begin{aligned} &rho \\ \rightarrow &E_0 \rho E_0^\dagger + E_1 \rho E_1^\dagger \end{aligned}$$



**Parameters**

**noise** (*dict*) – The dictionary representation of this noise.

**Returns**

*Noise* – A Noise object that represents the passed in dictionary.

**static phase\_damping**(*target: Qubit | int | Iterable[Qubit | int], gamma: float*) →  
Iterable[*Instruction*]

Registers this function into the circuit class.

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **gamma** (*float*) – Probability of phase damping.

**Returns**

Iterable[*Instruction*] – Iterable of PhaseDamping instructions.

**Examples**

```
>>> circ = Circuit().phase_damping(0, gamma=0.1)
```

**to\_matrix**() → Iterable[ndarray]

Returns a matrix representation of this noise.

**Returns**

Iterable[ndarray] – A list of matrix representations of this noise.

**class PhaseFlip**(*probability: FreeParameterExpression | float*)

Bases: [SingleProbabilisticNoise](#)

Phase flip noise channel which transforms a density matrix  
*rho* according to:

$$\begin{aligned} & \rho \\ \rightarrow & (1-p)\rho + pX\rho X \\ & \rho \end{aligned}$$

where

```

I =
left(
beginmatrix10

01
endmatrix
right)
Z =
left(
beginmatrix10

0 - 1
endmatrix
right)
p =
textprobability

```

This noise channel is shown as PF in circuit diagrams.

Initializes a [\*SingleProbabilisticNoise\*](#).

#### Parameters

- **probability** (*Union*[[FreeParameterExpression](#), *float*]) – The probability that the noise occurs.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits to apply noise.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction.
- **max\_probability** (*float*) – Maximum allowed probability of the noise channel. Default: 0.5

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length `!=` `qubit_count`, `probability` is not `float` or `FreeParameterExpression`, `probability > 1/2`, or `probability < 0`

**bind\_values**(*\*\*kwargs*: [FreeParameter](#) | *str*) → *Noise*

Takes in parameters and attempts to assign them to values.

#### Parameters

**\*\*kwargs** (*Union*[[FreeParameter](#), *str*]) – Arbitrary keyword arguments.

#### Returns

*Noise* – A new *Noise* object of the same type with the requested parameters bound.

**static fixed\_qubit\_count**() → *int*

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**classmethod** `from_dict(noise: dict) → Noise`

Converts a dictionary representation of this class into this class.

**Parameters**

**noise** (*dict*) – The dictionary representation of this noise.

**Returns**

*Noise* – A Noise object that represents the passed in dictionary.

**static** `phase_flip(target: Qubit | int | Iterable[Qubit | int], probability: float) → Iterable[Instruction]`

Registers this function into the circuit class.

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **probability** (*float*) – Probability of phase flipping.

**Returns**

*Iterable[Instruction]* – Iterable of PhaseFlip instructions.

## Examples

```
>>> circ = Circuit().phase_flip(0, probability=0.1)
```

**to\_matrix()** → *Iterable[ndarray]*

Returns a matrix representation of this noise.

**Returns**

*Iterable[ndarray]* – A list of matrix representations of this noise.

**class TwoQubitDephasing**(*probability: FreeParameterExpression | float*)

Bases: *SingleProbabilisticNoise\_34*

**Two-Qubit Dephasing noise channel which transforms a**  
density matrix  
*rho* according to:

$$\begin{aligned} & \rho \\ \rightarrow & (1-p)\rho + p\left(\frac{1}{3}\left(\rho + \rho_{IZ} + \rho_{ZI} + \rho_{ZZ}\right)\right) \end{aligned}$$

where

$$I = \begin{matrix} & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \end{matrix}$$

$$Z = \begin{matrix} & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \end{matrix}$$

$$p = \text{probability}$$

This noise channel is shown as DEPH in circuit diagrams.

Initializes a [SingleProbabilisticNoise\\_34](#).

#### Parameters

- **probability** ([Union\[FreeParameterExpression, float\]](#)) – The probability that the noise occurs.
- **qubit\_count** ([Optional\[int\]](#)) – The number of qubits to apply noise.
- **ascii\_symbols** ([Sequence\[str\]](#)) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction.

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, `probability` is not float or `FreeParameterExpression`, `probability > 3/4`, or `probability < 0`

**bind\_values**(\*\**kwargs*) → [Noise](#)

Takes in parameters and attempts to assign them to values.

#### Returns

[Noise](#) – A new Noise object of the same type with the requested parameters bound.

**static fixed\_qubit\_count**() → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

int – The number of qubits this quantum operator acts on.

**classmethod from\_dict**(noise: dict) → [Noise](#)

Converts a dictionary representation of this class into this class.

#### Parameters

**noise** (dict) – The dictionary representation of this noise.

**Returns**

*Noise* – A Noise object that represents the passed in dictionary.

**to\_matrix()** → Iterable[ndarray]

Returns a matrix representation of this noise.

**Returns**

*Iterable[ndarray]* – A list of matrix representations of this noise.

**static two\_qubit\_dephasing**(*target1: Qubit | int, target2: Qubit | int, probability: float*) → Iterable[*Instruction*]

Registers this function into the circuit class.

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1.
- **target2** (*QubitInput*) – Target qubit 2.
- **probability** (*float*) – Probability of two-qubit dephasing.

**Returns**

*Iterable[Instruction]* – Iterable of Dephasing instructions.

**Examples**

```
>>> circ = Circuit().two_qubit_dephasing(0, 1, probability=0.1)
```

**class TwoQubitDepolarizing**(*probability: FreeParameterExpression | float*)

Bases: *SingleProbabilisticNoise\_1516*

**Two-Qubit Depolarizing noise channel which transforms a**  
density matrix  
*rho* according to:



$\rho$   
 $\rightarrow(1-p)$   
 $\rho + p/15(IX$   
 $\rho IX$   
 $\dagger + IY$   
 $\rho IY$   
 $\dagger + IZ$   
 $\rho IZ$   
 $\dagger + XI$   
 $\rho XI$   
 $\dagger + XX$   
 $\rho XX$   
 $\dagger + XY$   
 $\rho XY$   
 $\dagger + XZ$   
 $\rho XZ$   
 $\dagger + YI$   
 $\rho YI$   
 $\dagger + YX$   
 $\rho YX$   
 $\dagger + YY$   
 $\rho YY$   
 $\dagger + YZ$   
 $\rho YZ$   
 $\dagger + ZI$   
 $\rho ZI$   
 $\dagger + ZX$   
 $\rho ZX$   
 $\dagger + ZY$   
 $\rho ZY$   
 $\dagger + ZZ$   
 $\rho ZZ$   
 $\dagger)$

where

```

        I =
        left(
beginmatrix10

        01
        endmatrix
        right)
        X =
        left(
beginmatrix01

        10
        endmatrix
        right)
        Y =
        left(
beginmatrix0 - i

        i0
        endmatrix
        right)
        Z =
        left(
beginmatrix10

        0 - 1
        endmatrix
        right)
        p =
        textprobability
```

This noise channel is shown as DEPO in circuit diagrams.

Initializes a [SingleProbabilisticNoise\\_1516](#).

#### Parameters

- **probability** (*Union*[[FreeParameterExpression](#), *float*]) – The probability that the noise occurs.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits to apply noise.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction.

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`,

or `ascii_symbols` length  $\neq$  `qubit_count`, `probability` is not float or `FreeParameterExpression`, `probability > 15/16`, or `probability < 0`

**bind\_values**(\*\*kwargs) → *Noise*

Takes in parameters and attempts to assign them to values.

**Returns**

*Noise* – A new Noise object of the same type with the requested parameters bound.

**static fixed\_qubit\_count**() → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

int – The number of qubits this quantum operator acts on.

**classmethod from\_dict**(noise: dict) → *Noise*

Converts a dictionary representation of this class into this class.

**Parameters**

**noise** (dict) – The dictionary representation of this noise.

**Returns**

*Noise* – A Noise object that represents the passed in dictionary.

**to\_matrix**() → Iterable[ndarray]

Returns a matrix representation of this noise.

**Returns**

Iterable[ndarray] – A list of matrix representations of this noise.

**static two\_qubit\_depolarizing**(target1: Qubit | int, target2: Qubit | int, probability: float) → Iterable[Instruction]

Registers this function into the circuit class.

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1.
- **target2** (*QubitInput*) – Target qubit 2.
- **probability** (*float*) – Probability of two-qubit depolarizing.

**Returns**

Iterable[Instruction] – Iterable of Depolarizing instructions.

## Examples

```
>>> circ = Circuit().two_qubit_depolarizing(0, 1, probability=0.1)
```

**class TwoQubitPauliChannel**(probabilities: dict[str, float])

Bases: *MultiQubitPauliNoise*

**Two-Qubit Pauli noise channel which transforms a density matrix  $\rho$  according to:**

$\rho$   
 $\rightarrow(1-p)$   
 $\rho + p_{IX}IX$   
 $\rho_{IX}$   
 $\text{dagger} + p_{IY}IY$   
 $\rho_{IY}$   
 $\text{dagger} + p_{IZ}IZ$   
 $\rho_{IZ}$   
 $\text{dagger} + p_{XI}XI$   
 $\rho_{XI}$   
 $\text{dagger} + p_{XX}XX$   
 $\rho_{XX}$   
 $\text{dagger} + p_{XY}XY$   
 $\rho_{XY}$   
 $\text{dagger} + p_{XZ}XZ$   
 $\rho_{XZ}$   
 $\text{dagger} + p_{YI}YI$   
 $\rho_{YI}$   
 $\text{dagger} + p_{YX}YX$   
 $\rho_{YX}$   
 $\text{dagger} + p_{YY}YY$   
 $\rho_{YY}$   
 $\text{dagger} + p_{YZ}YZ$   
 $\rho_{YZ}$   
 $\text{dagger} + p_{ZI}ZI$   
 $\rho_{ZI}$   
 $\text{dagger} + p_{ZX}ZX$   
 $\rho_{ZX}$   
 $\text{dagger} + p_{ZY}ZY$   
 $\rho_{ZY}$   
 $\text{dagger} + p_{ZZ}ZZ$   
 $\rho_{ZZ}$   
 $\text{dagger})$

where

$$\begin{aligned}
 I &= \\
 &\text{left}( \\
 &\text{beginmatrix} 1 & 0 \\
 & 0 & 1 \\
 &\text{endmatrix} \\
 &\text{right}) \\
 X &= \\
 &\text{left}( \\
 &\text{beginmatrix} 0 & 1 \\
 & 1 & 0 \\
 &\text{endmatrix} \\
 &\text{right}) \\
 Y &= \\
 &\text{left}( \\
 &\text{beginmatrix} 0 & -i \\
 & i & 0 \\
 &\text{endmatrix} \\
 &\text{right}) \\
 Z &= \\
 &\text{left}( \\
 &\text{beginmatrix} 1 & 0 \\
 & 0 & -1 \\
 &\text{endmatrix} \\
 &\text{right}) \\
 p &= \\
 &\text{textsumofallprobabilities}
 \end{aligned}$$

This noise channel is shown as `PC_2({"pauli_string": pauli_string, "probability": probability})` in circuit diagrams.

[summary]

#### Parameters

- **probabilities** (`dict[str, Union[FreeParameterExpression, float]]`) – A dictionary with Pauli strings as keys and the probabilities as values, i.e. `{"XX": 0.1, "IZ": 0.2}`.
- **qubit\_count** (`Optional[int]`) – The number of qubits the Pauli noise acts on.
- **ascii\_symbols** (`Sequence[str]`) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction.

#### Raises

- **ValueError** – If `qubit_count < 1`, `ascii_symbols` is `None`, `ascii_symbols` length != `qubit_count`, `probabilities` are not `float`s` or `FreeParameterExpressions`, any of `probabilities > 1` or `probabilities < 0`, the sum of all `probabilities` is `> 1`, if “II” is specified as a Pauli string, if any Pauli string contains invalid strings, or if the length of `probabilities` is greater than `4**qubit_count`.
- **TypeError** – If the type of the dictionary keys are not strings. If the `probabilities` are not floats.

**bind\_values**(\*\*kwargs) → *Noise*

Takes in parameters and attempts to assign them to values.

**Returns**

*Noise* – A new *Noise* object of the same type with the requested parameters bound.

**static fixed\_qubit\_count**() → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**classmethod from\_dict**(noise: dict) → *Noise*

Converts a dictionary representation of this class into this class.

**Parameters**

**noise** (dict) – The dictionary representation of this noise.

**Returns**

*Noise* – A *Noise* object that represents the passed in dictionary.

**to\_matrix**() → Iterable[ndarray]

Returns a matrix representation of this noise.

**Returns**

*Iterable[ndarray]* – A list of matrix representations of this noise.

**static two\_qubit\_pauli\_channel**(target1: *Qubit* | int, target2: *Qubit* | int, probabilities: dict[str, float]) → Iterable[*Instruction*]

Registers this function into the circuit class.

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1.
- **target2** (*QubitInput*) – Target qubit 2.
- **probabilities** (dict[str, float]) – Probability of two-qubit Pauli channel.

**Returns**

*Iterable[Instruction]* – Iterable of Depolarizing instructions.

## Examples

```
>>> circ = Circuit().two_qubit_pauli_channel(0, 1, {"XX": 0.1})
```

```
class braket.circuits.noise.SingleProbabilisticNoise(probability: FreeParameterExpression | float,
                                                    qubit_count: int | None, ascii_symbols:
                                                    Sequence[str], max_probability: float = 0.5)
```

Bases: *Noise*, *Parameterizable*

Class *SingleProbabilisticNoise* represents the bit/phase flip noise channel on N qubits parameterized by a single probability.

Initializes a *SingleProbabilisticNoise*.

#### Parameters

- **probability** (*Union*[*FreeParameterExpression*, *float*]) – The probability that the noise occurs.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits to apply noise.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as *qubit\_count*, and index ordering is expected to correlate with the target ordering on the instruction.
- **max\_probability** (*float*) – Maximum allowed probability of the noise channel. Default: 0.5

#### Raises

**ValueError** – If the *qubit\_count* is less than 1, *ascii\_symbols* are *None*, or *ascii\_symbols* length  $\neq$  *qubit\_count*, *probability* is not *float* or *FreeParameterExpression*, *probability* > 1/2, or *probability* < 0

**property probability:** *float*

The probability that parametrizes the noise channel.

#### Returns

*float* – The probability that parametrizes the noise channel.

**property parameters:** *list*[*FreeParameterExpression* | *float*]

Returns the parameters associated with the object, either unbound free parameter expressions or bound values.

#### Returns

*list*[*Union*[*FreeParameterExpression*, *float*]] – The free parameter expressions or fixed values associated with the object.

**bind\_values**(*\*\*kwargs*) → *SingleProbabilisticNoise*

Takes in parameters and attempts to assign them to values.

#### Returns

*SingleProbabilisticNoise* – A new Noise object of the same type with the requested parameters bound.

#### Raises

**NotImplementedError** – Subclasses should implement this function.

**to\_dict**() → *dict*

Converts this object into a dictionary representation.

#### Returns

*dict* – A dictionary object that represents this object. It can be converted back into this object using the *from\_dict*() method.

```
class braket.circuits.noise.SingleProbabilisticNoise_34(probability: FreeParameterExpression |
float, qubit_count: int | None,
ascii_symbols: Sequence[str])
```

Bases: *SingleProbabilisticNoise*

Class *SingleProbabilisticNoise* represents the Depolarizing and TwoQubitDephasing noise channels parameterized by a single probability.

Initializes a *SingleProbabilisticNoise\_34*.

**Parameters**

- **probability** (*Union*[[FreeParameterExpression](#), *float*]) – The probability that the noise occurs.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits to apply noise.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction.

**Raises**

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, `probability` is not `float` or `FreeParameterExpression`, `probability`  $> 3/4$ , or `probability`  $< 0$

```
class braket.circuits.noise.SingleProbabilisticNoise_1516(probability: FreeParameterExpression |
                                                         float, qubit_count: int | None,
                                                         ascii_symbols: Sequence[str])
```

Bases: [SingleProbabilisticNoise](#)

Class [SingleProbabilisticNoise](#) represents the TwoQubitDepolarizing noise channel parameterized by a single probability.

Initializes a [SingleProbabilisticNoise\\_1516](#).

**Parameters**

- **probability** (*Union*[[FreeParameterExpression](#), *float*]) – The probability that the noise occurs.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits to apply noise.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction.

**Raises**

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, `probability` is not `float` or `FreeParameterExpression`, `probability`  $> 15/16$ , or `probability`  $< 0$

```
class braket.circuits.noise.MultiQubitPauliNoise(probabilities: dict[str, FreeParameterExpression |
                                                                    float], qubit_count: int | None, ascii_symbols:
                                                                    Sequence[str])
```

Bases: [Noise](#), [Parameterizable](#)

Class [MultiQubitPauliNoise](#) represents a general multi-qubit Pauli channel, parameterized by up to  $4^{**N} - 1$  probabilities.

[summary]

**Parameters**

- **probabilities** (*dict*[*str*, *Union*[[FreeParameterExpression](#), *float*]]) – A dictionary with Pauli strings as keys and the probabilities as values, i.e. {"XX": 0.1, "IZ": 0.2}.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits the Pauli noise acts on.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction.



**Raises**

- **ValueError** – If `qubit_count < 1`, `ascii_symbols` is `None`, `ascii_symbols` length  $\neq$  `qubit_count`, `probabilities` are not float's or `FreeParameterExpressions`, any of `probabilities > 1` or `probabilities < 0`, the sum of all probabilities is  $> 1$ , if "II" is specified as a Pauli string, if any Pauli string contains invalid strings, or if the length of probabilities is greater than  $4^{**}qubit\_count$ .
- **TypeError** – If the type of the dictionary keys are not strings. If the probabilities are not floats.

**property probabilities:** `dict[str, float]`

A map of a Pauli string to its corresponding probability.

**Returns**

`dict[str, float]` – A map of a Pauli string to its corresponding probability.

**property parameters:** `list[FreeParameterExpression | float]`

Returns the parameters associated with the object, either unbound free parameter expressions or bound values.

Parameters are in alphabetical order of the Pauli strings in `probabilities`.

**Returns**

`list[Union[FreeParameterExpression, float]]` – The free parameter expressions or fixed values associated with the object.

**bind\_values(\*\*kwargs)** → `MultiQubitPauliNoise`

Takes in parameters and attempts to assign them to values.

**Returns**

`MultiQubitPauliNoise` – A new Noise object of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**to\_dict()** → `dict`

Converts this object into a dictionary representation.

**Returns**

`dict` – A dictionary object that represents this object. It can be converted back into this object using the `from_dict()` method.

```
class braket.circuits.noise.PauliNoise(probX: FreeParameterExpression | float, probY:
    FreeParameterExpression | float, probZ: FreeParameterExpression
    | float, qubit_count: int | None, ascii_symbols: Sequence[str])
```

Bases: `Noise`, `Parameterizable`

Class `PauliNoise` represents the a single-qubit Pauli noise channel acting on one qubit. It is parameterized by three probabilities.

Initializes a `PauliNoise`.

**Parameters**

- **probX** (`Union[FreeParameterExpression, float]`) – The X coefficient of the Kraus operators in the channel.
- **probY** (`Union[FreeParameterExpression, float]`) – The Y coefficient of the Kraus operators in the channel.

- **probZ** (*Union*[*FreeParameterExpression*, *float*]) – The Z coefficient of the Kraus operators in the channel.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits to apply noise.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction.

**Raises**

**ValueError** – If the **qubit\_count** is less than 1, **ascii\_symbols** are None, or **ascii\_symbols** length != **qubit\_count**, **probX** or **probY** or **probZ** is not float or *FreeParameterExpression*, **probX** or **probY** or **probZ** > 1.0, or **probX** or **probY** or **probZ** < 0.0, or **probX** + **probY** + **probZ** > 1

**property probX:** *FreeParameterExpression* | float

The probability of a Pauli X error.

**Returns**

*Union*[*FreeParameterExpression*, *float*] – The probability of a Pauli X error.

**property probY:** *FreeParameterExpression* | float

The probability of a Pauli Y error.

**Returns**

*Union*[*FreeParameterExpression*, *float*] – The probability of a Pauli Y error.

**property probZ:** *FreeParameterExpression* | float

The probability of a Pauli Z error.

**Returns**

*Union*[*FreeParameterExpression*, *float*] – The probability of a Pauli Z error.

**property parameters:** *list*[*FreeParameterExpression* | float]

Returns the parameters associated with the object, either unbound free parameter expressions or bound values.

Parameters are in the order [probX, probY, probZ]

**Returns**

*list*[*Union*[*FreeParameterExpression*, *float*]] – The free parameter expressions or fixed values associated with the object.

**bind\_values**(*\*\*kwargs*) → *PauliNoise*

Takes in parameters and attempts to assign them to values.

**Returns**

*PauliNoise* – A new Noise object of the same type with the requested parameters bound.

**Raises**

**NotImplementedError** – Subclasses should implement this function.

**to\_dict**() → dict

Converts this object into a dictionary representation.

**Returns**

*dict* – A dictionary object that represents this object. It can be converted back into this object using the **from\_dict**() method.

```
class braket.circuits.noise.DampingNoise(gamma: FreeParameterExpression | float, qubit_count: int |
                                         None, ascii_symbols: Sequence[str])
```

Bases: *Noise*, *Parameterizable*

Class *DampingNoise* represents a damping noise channel on N qubits parameterized by gamma.

Initializes a *DampingNoise*.

#### Parameters

- **gamma** (*Union*[*FreeParameterExpression*, *float*]) – Probability of damping.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits to apply noise.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as *qubit\_count*, and index ordering is expected to correlate with the target ordering on the instruction.

#### Raises

**ValueError** – If *qubit\_count* < 1, *ascii\_symbols* is None, *len(ascii\_symbols)* != *qubit\_count*, *gamma* is not float or *FreeParameterExpression*, or *gamma* > 1.0 or *gamma* < 0.0.

**property gamma:** float

Probability of damping.

#### Returns

*float* – Probability of damping.

**property parameters:** list[*FreeParameterExpression* | float]

Returns the parameters associated with the object, either unbound free parameter expressions or bound values.

#### Returns

list[*Union*[*FreeParameterExpression*, *float*]] – The free parameter expressions or fixed values associated with the object.

**bind\_values**(\*\**kwargs*) → *DampingNoise*

Takes in parameters and attempts to assign them to values.

#### Returns

*DampingNoise* – A new Noise object of the same type with the requested parameters bound.

#### Raises

**NotImplementedError** – Subclasses should implement this function.

**to\_dict**() → dict

Converts this object into a dictionary representation.

#### Returns

dict – A dictionary object that represents this object. It can be converted back into this object using the *from\_dict*() method.

```
class braket.circuits.noise.GeneralizedAmplitudeDampingNoise(gamma: FreeParameterExpression |
                                                             float, probability:
                                                             FreeParameterExpression | float,
                                                             qubit_count: int | None,
                                                             ascii_symbols: Sequence[str])
```

Bases: *DampingNoise*

Class *GeneralizedAmplitudeDampingNoise* represents the generalized amplitude damping noise channel on N qubits parameterized by gamma and probability.

Initializes a *GeneralizedAmplitudeDampingNoise*.

#### Parameters

- **gamma** (*Union*[*FreeParameterExpression*, *float*]) – Probability of damping.
- **probability** (*Union*[*FreeParameterExpression*, *float*]) – Probability of the system being excited by the environment.
- **qubit\_count** (*Optional*[*int*]) – The number of qubits to apply noise.
- **ascii\_symbols** (*Sequence*[*str*]) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as **qubit\_count**, and index ordering is expected to correlate with the target ordering on the instruction.

#### Raises

**ValueError** – If **qubit\_count** < 1, **ascii\_symbols** is None, `len(ascii_symbols) != qubit_count`, **probability** or **gamma** is not *float* or *FreeParameterExpression*, **probability** > 1.0 or **probability** < 0.0, or **gamma** > 1.0 or **gamma** < 0.0.

**property probability:** *float*

Probability of the system being excited by the environment.

#### Returns

*float* – Probability of the system being excited by the environment.

**property parameters:** *list*[*FreeParameterExpression* | *float*]

Returns the parameters associated with the object, either unbound free parameter expressions or bound values.

Parameters are in the order [gamma, probability]

#### Returns

*list*[*Union*[*FreeParameterExpression*, *float*]] – The free parameter expressions or fixed values associated with the object.

**to\_dict()** → *dict*

Converts this object into a dictionary representation.

#### Returns

*dict* – A dictionary object that represents this object. It can be converted back into this object using the `from_dict()` method.

## braket.circuits.noise\_helpers module

`braket.circuits.noise_helpers.no_noise_applied_warning(noise_applied: bool) → None`

Helper function to give a warning if noise is not applied.

#### Parameters

**noise\_applied** (*bool*) – True if the noise has been applied.

`braket.circuits.noise_helpers.wrap_with_list(an_item: Any) → list[Any]`

Helper function to make the input parameter a list.

#### Parameters

**an\_item** (*Any*) – The item to wrap.

#### Returns

*list*[*Any*] – The item wrapped in a list.

`braket.circuits.noise_helpers.check_noise_target_gates`(*noise*: [Noise](#), *target\_gates*: *Iterable[type[Gate]]*) → *None*

Helper function to check 1. whether all the elements in *target\_gates* are a *Gate* type; 2. if *noise* is multi-qubit noise and *target\_gates* contain gates with the number of qubits is the same as *noise.qubit\_count*.

#### Parameters

- **noise** ([Noise](#)) – A Noise class object to be applied to the circuit.
- **target\_gates** (*Iterable[type[Gate]]*) – Gate class or List of Gate classes which noise is applied to.

`braket.circuits.noise_helpers.check_noise_target_unitary`(*noise*: [Noise](#), *target\_unitary*: *ndarray*) → *None*

Helper function to check 1. whether the input matrix is a *np.ndarray* type; 2. whether the *target\_unitary* is a unitary;

#### Parameters

- **noise** ([Noise](#)) – A Noise class object to be applied to the circuit.
- **target\_unitary** (*ndarray*) – matrix of the target unitary gates

`braket.circuits.noise_helpers.check_noise_target_qubits`(*circuit*: [Circuit](#), *target\_qubits*: *QubitSetInput* | *None* = *None*) → [QubitSet](#)

Helper function to check whether all the *target\_qubits* are positive integers.

#### Parameters

- **circuit** ([Circuit](#)) – A circuit where noise is to be checked.
- **target\_qubits** (*Optional[QubitSetInput]*) – Index or indices of qubit(s).

#### Returns

[QubitSet](#) – The target qubits.

`braket.circuits.noise_helpers.apply_noise_to_moments`(*circuit*: [Circuit](#), *noise*: *Iterable[type[Noise]]*, *target\_qubits*: [QubitSet](#), *position*: *str*) → [Circuit](#)

Apply initialization/readout noise to the circuit.

When *noise.qubit\_count* == 1, noise is added to all qubits in *target\_qubits*.

When *noise.qubit\_count* > 1, *noise.qubit\_count* must be the same as the length of *target\_qubits*.

#### Parameters

- **circuit** ([Circuit](#)) – A circuit to noise is applied to.
- **noise** (*Iterable[type[Noise]]*) – Noise channel(s) to be applied to the circuit.
- **target\_qubits** ([QubitSet](#)) – Index or indices of qubits. noise is applied to.
- **position** (*str*) – The position to add the noise to. May be ‘initialization’ or ‘readout\_noise’.

#### Returns

[Circuit](#) – modified circuit.

`braket.circuits.noise_helpers.apply_noise_to_gates`(*circuit*: [Circuit](#), *noise*: *Iterable[type[Noise]]*, *target\_gates*: *Iterable[type[Gate]]* | *np.ndarray*, *target\_qubits*: [QubitSet](#)) → [Circuit](#)

Apply noise after target gates in target qubits.

When *noise.qubit\_count* == 1, noise is applied to *target\_qubits* after *target\_gates*.

When `noise.qubit_count > 1`, all elements in `target_gates`, if is given, must have the same number of qubits as `noise.qubit_count`.

**Parameters**

- **circuit** (`Circuit`) – A circuit where noise is applied to.
- **noise** (`Iterable[type[Noise]]`) – Noise channel(s) to be applied to the circuit.
- **target\_gates** (`Union[Iterable[type[Gate]], ndarray]`) – List of gates, or a unitary matrix which noise is applied to.
- **target\_qubits** (`QubitSet`) – Index or indices of qubits which noise is applied to.

**Returns**

*Circuit* – modified circuit.

**Raises**

**Warning** – If noise is multi-qubit noise while there is no gate with the same number of qubits in `target_qubits` or in the whole circuit when `target_qubits` is not given. If no `target_gates` exist in `target_qubits` or in the whole circuit when `target_qubits` is not given.

**braket.circuits.noises module**

**class** `braket.circuits.noises.BitFlip(probability: FreeParameterExpression | float)`

Bases: `SingleProbabilisticNoise`

Bit flip noise channel which transforms a density matrix *rho* according to:

$$\begin{aligned} & \rho \\ \rightarrow & (1-p)\rho + pX\rho X \\ & \text{dagger} \end{aligned}$$

where

$$I = \text{left}(\text{beginmatrix} 10 \\ 01 \\ \text{endmatrix} \text{right})$$

$$X = \text{left}(\text{beginmatrix} 01 \\ 10 \\ \text{endmatrix} \text{right})$$

$$p = \text{textprobability}$$

This noise channel is shown as BF in circuit diagrams.

Initializes a `SingleProbabilisticNoise`.

#### Parameters

- **probability** (`Union[FreeParameterExpression, float]`) – The probability that the noise occurs.
- **qubit\_count** (`Optional[int]`) – The number of qubits to apply noise.
- **ascii\_symbols** (`Sequence[str]`) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction.
- **max\_probability** (`float`) – Maximum allowed probability of the noise channel. Default: 0.5

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, `probability` is not `float` or `FreeParameterExpression`, `probability`  $> 1/2$ , or `probability`  $< 0$

**to\_matrix()**  $\rightarrow$  `Iterable[ndarray]`

Returns a matrix representation of this noise.

#### Returns

`Iterable[ndarray]` – A list of matrix representations of this noise.

**static fixed\_qubit\_count()**  $\rightarrow$  `int`

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

`int` – The number of qubits this quantum operator acts on.

**static** `bit_flip`(*target*: `Qubit` | `int` | `Iterable`[`Qubit` | `int`], *probability*: `float`) → `Iterable`[`Instruction`]

Registers this function into the circuit class.

#### Parameters

- **target** (`QubitSetInput`) – Target qubit(s)
- **probability** (`float`) – Probability of bit flipping.

#### Returns

`Iterable`[`Instruction`] – Iterable of BitFlip instructions.

### Examples

```
>>> circ = Circuit().bit_flip(0, probability=0.1)
```

**bind\_values**(*\*\*kwargs*: `FreeParameter` | `str`) → `Noise`

Takes in parameters and attempts to assign them to values.

#### Parameters

**\*\*kwargs** (`Union`[`FreeParameter`, `str`]) – Arbitrary keyword arguments.

#### Returns

`Noise` – A new Noise object of the same type with the requested parameters bound.

**classmethod** `from_dict`(*noise*: `dict`) → `Noise`

Converts a dictionary representation of this class into this class.

#### Parameters

**noise** (`dict`) – The dictionary representation of this noise.

#### Returns

`Noise` – A Noise object that represents the passed in dictionary.

**class** `braket.circuits.noises.PhaseFlip`(*probability*: `FreeParameterExpression` | `float`)

Bases: `SingleProbabilisticNoise`

Phase flip noise channel which transforms a density matrix  $\rho$  according to:

$$\begin{aligned} &\rho \\ &\rightarrow (1-p)\rho + pX\rho X \\ &\quad \text{dagger} \end{aligned}$$



where

$$I = \begin{matrix} & \text{left} \\ \begin{matrix} \text{beginmatrix} 1 & 0 \\ 0 & 1 \end{matrix} & \end{matrix}$$

$$Z = \begin{matrix} & \text{left} \\ \begin{matrix} \text{beginmatrix} 1 & 0 \\ 0 & -1 \end{matrix} & \end{matrix}$$

$$p = \text{probability}$$

This noise channel is shown as PF in circuit diagrams.

Initializes a `SingleProbabilisticNoise`.

#### Parameters

- **probability** (`Union[FreeParameterExpression, float]`) – The probability that the noise occurs.
- **qubit\_count** (`Optional[int]`) – The number of qubits to apply noise.
- **ascii\_symbols** (`Sequence[str]`) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction.
- **max\_probability** (`float`) – Maximum allowed probability of the noise channel. Default: 0.5

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, `probability` is not `float` or `FreeParameterExpression`, `probability`  $> 1/2$ , or `probability`  $< 0$

**to\_matrix()**  $\rightarrow$  `Iterable[ndarray]`

Returns a matrix representation of this noise.

#### Returns

`Iterable[ndarray]` – A list of matrix representations of this noise.

**static fixed\_qubit\_count()**  $\rightarrow$  `int`

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

`int` – The number of qubits this quantum operator acts on.

**static** `phase_flip(target: Qubit | int | Iterable[Qubit | int], probability: float) → Iterable[Instruction]`

Registers this function into the circuit class.

#### Parameters

- **target** (*QubitSetInput*) – Target qubit(s)
- **probability** (*float*) – Probability of phase flipping.

#### Returns

*Iterable[Instruction]* – Iterable of PhaseFlip instructions.

### Examples

```
>>> circ = Circuit().phase_flip(0, probability=0.1)
```

**bind\_values**(\*\*kwargs: FreeParameter | str) → Noise

Takes in parameters and attempts to assign them to values.

#### Parameters

**\*\*kwargs** (*Union[FreeParameter, str]*) – Arbitrary keyword arguments.

#### Returns

*Noise* – A new Noise object of the same type with the requested parameters bound.

**classmethod** `from_dict(noise: dict) → Noise`

Converts a dictionary representation of this class into this class.

#### Parameters

**noise** (*dict*) – The dictionary representation of this noise.

#### Returns

*Noise* – A Noise object that represents the passed in dictionary.

**class** `braket.circuits.noises.PauliChannel(probX: FreeParameterExpression | float, probY: FreeParameterExpression | float, probZ: FreeParameterExpression | float)`

Bases: *PauliNoise*

Pauli noise channel which transforms a density matrix

*rho* according to:

$$\begin{aligned} & \rho \\ & \rightarrow (1 - \text{probX} - \text{probY} - \text{probZ})\rho \\ & \quad + \text{probXX}\rho \\ & \quad + \text{probXY}\rho \\ & \quad + \text{probYY}\rho \\ & \quad + \text{probXZ}\rho \\ & \quad + \text{probYZ}\rho \\ & \quad + \text{probZZ}\rho \end{aligned}$$

where

$$\begin{aligned}
 I &= \\
 &\text{left}( \\
 &\text{beginmatrix} 10 \\ \\ 01 \\ \\ \text{endmatrix} \\
 &\text{right}) \\
 X &= \\
 &\text{left}( \\
 &\text{beginmatrix} 01 \\ \\ 10 \\ \\ \text{endmatrix} \\
 &\text{right}) \\
 Y &= \\
 &\text{left}( \\
 &\text{beginmatrix} 0-i \\ \\ i0 \\ \\ \text{endmatrix} \\
 &\text{right}) \\
 Z &= \\
 &\text{left}( \\
 &\text{beginmatrix} 10 \\ \\ 0-1 \\ \\ \text{endmatrix} \\
 &\text{right})
 \end{aligned}$$

This noise channel is shown as PC in circuit diagrams.

Creates PauliChannel noise.

#### Parameters

- **probX** (*Union*[FreeParameterExpression, float]) – X rotation probability.
- **probY** (*Union*[FreeParameterExpression, float]) – Y rotation probability.
- **probZ** (*Union*[FreeParameterExpression, float]) – Z rotation probability.

**to\_matrix()** → Iterable[ndarray]

Returns a matrix representation of this noise.

#### Returns

*Iterable*[ndarray] – A list of matrix representations of this noise.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

*int* – The number of qubits this quantum operator acts on.

**static** `pauli_channel`(*target*: `Qubit` | `int` | `Iterable[Qubit | int]`, *probX*: `float`, *probY*: `float`, *probZ*: `float`) → `Iterable[Instruction]`

Registers this function into the circuit class.

#### Parameters

- **target** (`QubitSetInput`) – Target qubit(s) probability list[`float`]: Probabilities for the Pauli X, Y and Z noise happening in the Kraus channel.
- **probX** (`float`) – X rotation probability.
- **probY** (`float`) – Y rotation probability.
- **probZ** (`float`) – Z rotation probability.

#### Returns

`Iterable[Instruction]` – `Iterable` of `PauliChannel` instructions.

### Examples

```
>>> circ = Circuit().pauli_channel(0, probX=0.1, probY=0.2, probZ=0.3)
```

**bind\_values**(*\*\*kwargs*) → `Noise`

Takes in parameters and attempts to assign them to values.

#### Returns

`Noise` – A new `Noise` object of the same type with the requested parameters bound.

**classmethod** `from_dict`(*noise*: `dict`) → `Noise`

Converts a dictionary representation of this class into this class.

#### Parameters

**noise** (`dict`) – The dictionary representation of this noise.

#### Returns

`Noise` – A `Noise` object that represents the passed in dictionary.

**class** `braket.circuits.noises.Depolarizing`(*probability*: `FreeParameterExpression` | `float`)

Bases: `SingleProbabilisticNoise_34`

Depolarizing noise channel which transforms a density matrix *rho* according to:

$$\begin{aligned} & \rho \\ \rightarrow & (1-p)\rho + p/3X\rho X \\ & + p/3Y\rho Y \\ & + p/3Z\rho Z \end{aligned}$$

where

$$\begin{aligned}
 I &= \\
 &\text{left}( \\
 &\text{beginmatrix} 10 \\ \\ \\ 01 \\ \text{endmatrix} \\
 &\text{right}) \\
 X &= \\
 &\text{left}( \\
 &\text{beginmatrix} 01 \\ \\ \\ 10 \\ \text{endmatrix} \\
 &\text{right}) \\
 Y &= \\
 &\text{left}( \\
 &\text{beginmatrix} 0-i \\ \\ \\ i0 \\ \text{endmatrix} \\
 &\text{right}) \\
 Z &= \\
 &\text{left}( \\
 &\text{beginmatrix} 10 \\ \\ \\ 0-1 \\ \text{endmatrix} \\
 &\text{right}) \\
 p &= \\
 &\text{textprobability}
 \end{aligned}$$

This noise channel is shown as DEPO in circuit diagrams.

Initializes a `SingleProbabilisticNoise_34`.

#### Parameters

- **probability** (`Union[FreeParameterExpression, float]`) – The probability that the noise occurs.
- **qubit\_count** (`Optional[int]`) – The number of qubits to apply noise.
- **ascii\_symbols** (`Sequence[str]`) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction.

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`,

or `ascii_symbols` length `!=` `qubit_count`, `probability` is not `float` or `FreeParameterExpression`, `probability > 3/4`, or `probability < 0`

**to\_matrix()** → `Iterable[ndarray]`

Returns a matrix representation of this noise.

**Returns**

*Iterable[ndarray]* – A list of matrix representations of this noise.

**static fixed\_qubit\_count()** → `int`

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static depolarizing(target: `Qubit` | `int` | `Iterable[Qubit | int]`, probability: `float`)** → `Iterable[Instruction]`

Registers this function into the circuit class.

**Parameters**

- **target** (`QubitSetInput`) – Target qubit(s)
- **probability** (`float`) – Probability of depolarizing.

**Returns**

*Iterable[Instruction]* – Iterable of Depolarizing instructions.

## Examples

```
>>> circ = Circuit().depolarizing(0, probability=0.1)
```

**bind\_values(\*\*kwargs)** → *Noise*

Takes in parameters and attempts to assign them to values.

**Returns**

*Noise* – A new Noise object of the same type with the requested parameters bound.

**classmethod from\_dict(noise: dict)** → *Noise*

Converts a dictionary representation of this class into this class.

**Parameters**

**noise** (`dict`) – The dictionary representation of this noise.

**Returns**

*Noise* – A Noise object that represents the passed in dictionary.

**class** `braket.circuits.noises.TwoQubitDepolarizing(probability: FreeParameterExpression | float)`

Bases: *SingleProbabilisticNoise\_1516*

**Two-Qubit Depolarizing noise channel which transforms a density matrix  $\rho$  according to:**

$\rho$   
 $\text{Rightarrow}(1 - p)$   
 $\rho + p/15(IX$   
 $\rho IX$   
 $\text{dagger} + IY$   
 $\rho IY$   
 $\text{dagger} + IZ$   
 $\rho IZ$   
 $\text{dagger} + XI$   
 $\rho XI$   
 $\text{dagger} + XX$   
 $\rho XX$   
 $\text{dagger} + XY$   
 $\rho XY$   
 $\text{dagger} + XZ$   
 $\rho XZ$   
 $\text{dagger} + YI$   
 $\rho YI$   
 $\text{dagger} + YX$   
 $\rho YX$   
 $\text{dagger} + YY$   
 $\rho YY$   
 $\text{dagger} + YZ$   
 $\rho YZ$   
 $\text{dagger} + ZI$   
 $\rho ZI$   
 $\text{dagger} + ZX$   
 $\rho ZX$   
 $\text{dagger} + ZY$   
 $\rho ZY$   
 $\text{dagger} + ZZ$   
 $\rho ZZ$   
 $\text{dagger})$

where

$$\begin{aligned} I &= \\ &\text{left}( \\ &\text{beginmatrix} 10 \\ \\ 01 \\ &\text{endmatrix} \\ &\text{right}) \\ X &= \\ &\text{left}( \\ &\text{beginmatrix} 01 \\ \\ 10 \\ &\text{endmatrix} \\ &\text{right}) \\ Y &= \\ &\text{left}( \\ &\text{beginmatrix} 0-i \\ \\ i0 \\ &\text{endmatrix} \\ &\text{right}) \\ Z &= \\ &\text{left}( \\ &\text{beginmatrix} 10 \\ \\ 0-1 \\ &\text{endmatrix} \\ &\text{right}) \\ p &= \\ &\text{textprobability} \end{aligned}$$

This noise channel is shown as DEPO in circuit diagrams.

Initializes a `SingleProbabilisticNoise_1516`.

#### Parameters

- **probability** (`Union[FreeParameterExpression, float]`) – The probability that the noise occurs.
- **qubit\_count** (`Optional[int]`) – The number of qubits to apply noise.
- **ascii\_symbols** (`Sequence[str]`) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction.

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`,



or `ascii_symbols` length `!=` `qubit_count`, `probability` is not `float` or `FreeParameterExpression`, `probability > 15/16`, or `probability < 0`

**to\_matrix()** → `Iterable[ndarray]`

Returns a matrix representation of this noise.

#### Returns

`Iterable[ndarray]` – A list of matrix representations of this noise.

**static fixed\_qubit\_count()** → `int`

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

`int` – The number of qubits this quantum operator acts on.

**static two\_qubit\_depolarizing**(*target1*: `Qubit` | `int`, *target2*: `Qubit` | `int`, *probability*: `float`) → `Iterable[Instruction]`

Registers this function into the circuit class.

#### Parameters

- **target1** (`QubitInput`) – Target qubit 1.
- **target2** (`QubitInput`) – Target qubit 2.
- **probability** (`float`) – Probability of two-qubit depolarizing.

#### Returns

`Iterable[Instruction]` – Iterable of Depolarizing instructions.

### Examples

```
>>> circ = Circuit().two_qubit_depolarizing(0, 1, probability=0.1)
```

**bind\_values**(\*\**kwargs*) → `Noise`

Takes in parameters and attempts to assign them to values.

#### Returns

`Noise` – A new Noise object of the same type with the requested parameters bound.

**classmethod from\_dict**(*noise*: `dict`) → `Noise`

Converts a dictionary representation of this class into this class.

#### Parameters

**noise** (`dict`) – The dictionary representation of this noise.

#### Returns

`Noise` – A Noise object that represents the passed in dictionary.

**class** `braket.circuits.noises.TwoQubitDephasing`(*probability*: `FreeParameterExpression` | `float`)

Bases: `SingleProbabilisticNoise_34`

**Two-Qubit Dephasing noise channel which transforms a density matrix  $\rho$  according to:**

$$\begin{aligned}
 & \text{rho} \\
 & \text{Rightarrow}(1 - p) \\
 & \text{rho} + p/3(\text{IZ} \\
 & \text{rhoIZ} \\
 & \text{dagger} + \text{ZI} \\
 & \text{rhoZI} \\
 & \text{dagger} + \text{ZZ} \\
 & \text{rhoZZ} \\
 & \text{dagger})
 \end{aligned}$$

where

$$\begin{aligned}
 I &= \\
 & \text{left}( \\
 & \text{beginmatrix} 10 \\ \\ \\ 01 \\ \\ \\ \text{endmatrix} \\
 & \text{right}) \\
 Z &= \\
 & \text{left}( \\
 & \text{beginmatrix} 10 \\ \\ 0 - 1 \\ \\ \text{endmatrix} \\
 & \text{right}) \\
 p &= \\
 & \text{textprobability}
 \end{aligned}$$

This noise channel is shown as DEPH in circuit diagrams.

Initializes a `SingleProbabilisticNoise_34`.

#### Parameters

- **probability** (`Union[FreeParameterExpression, float]`) – The probability that the noise occurs.
- **qubit\_count** (`Optional[int]`) – The number of qubits to apply noise.
- **ascii\_symbols** (`Sequence[str]`) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction.

#### Raises

**ValueError** – If the `qubit_count` is less than 1, `ascii_symbols` are `None`, or `ascii_symbols` length  $\neq$  `qubit_count`, `probability` is not `float` or `FreeParameterExpression`, `probability`  $> 3/4$ , or `probability`  $< 0$

**to\_matrix()**  $\rightarrow$  `Iterable[ndarray]`

Returns a matrix representation of this noise.

**Returns**

*Iterable[ndarray]* – A list of matrix representations of this noise.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static two\_qubit\_dephasing**(*target1: Qubit | int, target2: Qubit | int, probability: float*) → *Iterable[Instruction]*

Registers this function into the circuit class.

**Parameters**

- **target1** (*QubitInput*) – Target qubit 1.
- **target2** (*QubitInput*) – Target qubit 2.
- **probability** (*float*) – Probability of two-qubit dephasing.

**Returns**

*Iterable[Instruction]* – Iterable of Dephasing instructions.

**Examples**

```
>>> circ = Circuit().two_qubit_dephasing(0, 1, probability=0.1)
```

**bind\_values**(*\*\*kwargs*) → *Noise*

Takes in parameters and attempts to assign them to values.

**Returns**

*Noise* – A new Noise object of the same type with the requested parameters bound.

**classmethod from\_dict**(*noise: dict*) → *Noise*

Converts a dictionary representation of this class into this class.

**Parameters**

**noise** (*dict*) – The dictionary representation of this noise.

**Returns**

*Noise* – A Noise object that represents the passed in dictionary.

**class** `braket.circuits.noises.TwoQubitPauliChannel`(*probabilities: dict[str, float]*)

Bases: *MultiQubitPauliNoise*

**Two-Qubit Pauli noise channel which transforms a**

density matrix

*rho* according to:

*rho*  
*Rightarrow*(1 - *p*)  
*rho* + *p*<sub>*I**X*</sub>*I**X*  
*rho**I**X*  
*dagger* + *p*<sub>*I**Y*</sub>*I**Y*  
*rho**I**Y*  
*dagger* + *p*<sub>*I**Z*</sub>*I**Z*  
*rho**I**Z*  
*dagger* + *p*<sub>*X**I*</sub>*X**I*  
*rho**X**I*  
*dagger* + *p*<sub>*X**X*</sub>*X**X*  
*rho**X**X*  
*dagger* + *p*<sub>*X**Y*</sub>*X**Y*  
*rho**X**Y*  
*dagger* + *p*<sub>*X**Z*</sub>*X**Z*  
*rho**X**Z*  
*dagger* + *p*<sub>*Y**I*</sub>*Y**I*  
*rho**Y**I*  
*dagger* + *p*<sub>*Y**X*</sub>*Y**X*  
*rho**Y**X*  
*dagger* + *p*<sub>*Y**Y*</sub>*Y**Y*  
*rho**Y**Y*  
*dagger* + *p*<sub>*Y**Z*</sub>*Y**Z*  
*rho**Y**Z*  
*dagger* + *p*<sub>*Z**I*</sub>*Z**I*  
*rho**Z**I*  
*dagger* + *p*<sub>*Z**X*</sub>*Z**X*  
*rho**Z**X*  
*dagger* + *p*<sub>*Z**Y*</sub>*Z**Y*  
*rho**Z**Y*  
*dagger* + *p*<sub>*Z**Z*</sub>*Z**Z*  
*rho**Z**Z*  
*dagger*)

where

$$\begin{aligned}
 I &= \\
 &\text{left}( \\
 &\quad \text{beginmatrix} 10 \\
 &\quad \quad 01 \\
 &\quad \text{endmatrix} \\
 &\quad \text{right}) \\
 X &= \\
 &\text{left}( \\
 &\quad \text{beginmatrix} 01 \\
 &\quad \quad 10 \\
 &\quad \text{endmatrix} \\
 &\quad \text{right}) \\
 Y &= \\
 &\text{left}( \\
 &\quad \text{beginmatrix} 0 - i \\
 &\quad \quad i0 \\
 &\quad \text{endmatrix} \\
 &\quad \text{right}) \\
 Z &= \\
 &\text{left}( \\
 &\quad \text{beginmatrix} 10 \\
 &\quad \quad 0 - 1 \\
 &\quad \text{endmatrix} \\
 &\quad \text{right}) \\
 p &= \\
 &\text{textsumofallprobabilities}
 \end{aligned}$$

This noise channel is shown as `PC_2({"pauli_string": probability})` in circuit diagrams.

[summary]

#### Parameters

- **probabilities** (`dict[str, Union[FreeParameterExpression, float]]`) – A dictionary with Pauli strings as keys and the probabilities as values, i.e. `{"XX": 0.1, "IZ": 0.2}`.
- **qubit\_count** (`Optional[int]`) – The number of qubits the Pauli noise acts on.
- **ascii\_symbols** (`Sequence[str]`) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction.

#### Raises

- **ValueError** – If `qubit_count < 1`, `ascii_symbols` is `None`, `ascii_symbols` length  $\neq$  `qubit_count`, `probabilities` are not `float`'s or `FreeParameterExpressions`,

any of `probabilities > 1` or `probabilities < 0`, the sum of all probabilities is `> 1`, if “II” is specified as a Pauli string, if any Pauli string contains invalid strings, or if the length of probabilities is greater than `4**qubit_count`.

- **TypeError** – If the type of the dictionary keys are not strings. If the probabilities are not floats.

**to\_matrix()** → `Iterable[ndarray]`

Returns a matrix representation of this noise.

#### Returns

`Iterable[ndarray]` – A list of matrix representations of this noise.

**static fixed\_qubit\_count()** → `int`

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

#### Returns

`int` – The number of qubits this quantum operator acts on.

**static two\_qubit\_pauli\_channel**(*target1*: `Qubit` | `int`, *target2*: `Qubit` | `int`, *probabilities*: `dict[str, float]`) → `Iterable[Instruction]`

Registers this function into the circuit class.

#### Parameters

- **target1** (`QubitInput`) – Target qubit 1.
- **target2** (`QubitInput`) – Target qubit 2.
- **probabilities** (`dict[str, float]`) – Probability of two-qubit Pauli channel.

#### Returns

`Iterable[Instruction]` – Iterable of Depolarizing instructions.

### Examples

```
>>> circ = Circuit().two_qubit_pauli_channel(0, 1, {"XX": 0.1})
```

**bind\_values**(\*\**kwargs*) → `Noise`

Takes in parameters and attempts to assign them to values.

#### Returns

`Noise` – A new Noise object of the same type with the requested parameters bound.

**classmethod from\_dict**(*noise*: `dict`) → `Noise`

Converts a dictionary representation of this class into this class.

#### Parameters

**noise** (`dict`) – The dictionary representation of this noise.

#### Returns

`Noise` – A Noise object that represents the passed in dictionary.

**class** `braket.circuits.noises.AmplitudeDamping`(*gamma*: `FreeParameterExpression` | `float`)

Bases: `DampingNoise`

AmplitudeDamping noise channel which transforms a density matrix  $\rho$  according to:

$$\rho \rightarrow E_0 \rho E_0^\dagger + E_1 \rho E_1^\dagger$$

where

$$E_0 = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-\gamma} \end{pmatrix}$$

$$E_1 = \begin{pmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{pmatrix}$$

This noise channel is shown as AD in circuit diagrams.

Initializes a DampingNoise.

#### Parameters

- **gamma** (*Union[FreeParameterExpression, float]*) – Probability of damping.
- **qubit\_count** (*Optional[int]*) – The number of qubits to apply noise.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction.

#### Raises

**ValueError** – If `qubit_count < 1`, `ascii_symbols` is `None`, `len(ascii_symbols) != qubit_count`, `gamma` is not `float` or `FreeParameterExpression`, or `gamma > 1.0` or `gamma < 0.0`.

**to\_matrix()** → Iterable[ndarray]

Returns a matrix representation of this noise.

**Returns**

*Iterable[ndarray]* – A list of matrix representations of this noise.

**static fixed\_qubit\_count()** → int

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static amplitude\_damping**(*target*: Qubit | int | Iterable[Qubit | int], *gamma*: float) → Iterable[Instruction]

Registers this function into the circuit class.

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s).
- **gamma** (*float*) – decaying rate of the amplitude damping channel.

**Returns**

*Iterable[Instruction]* – Iterable of AmplitudeDamping instructions.

## Examples

```
>>> circ = Circuit().amplitude_damping(0, gamma=0.1)
```

**bind\_values**(\*\**kwargs*) → Noise

Takes in parameters and attempts to assign them to values.

**Returns**

*Noise* – A new Noise object of the same type with the requested parameters bound.

**classmethod from\_dict**(*noise*: dict) → Noise

Converts a dictionary representation of this class into this class.

**Parameters**

**noise** (*dict*) – The dictionary representation of this noise.

**Returns**

*Noise* – A Noise object that represents the passed in dictionary.

**class** braket.circuits.noises.**GeneralizedAmplitudeDamping**(*gamma*: FreeParameterExpression | float, *probability*: FreeParameterExpression | float)

Bases: *GeneralizedAmplitudeDampingNoise*

**Generalized AmplitudeDamping noise channel which transforms a density matrix  $\rho$  according to:**



$\rho$   
 $\rightarrow E_0$   
 $\rho E_0$   
 $\dagger + E_1$   
 $\rho E_1$   
 $\dagger + E_2$   
 $\rho E_2$   
 $\dagger + E_3$   
 $\rho E_3$   
 $\dagger$

where

$$E_0 = \sqrt{\text{probability}} \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1 - \text{probability}} \end{pmatrix}$$
$$E_1 = \sqrt{\text{probability}} \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1 - \text{probability}} \end{pmatrix}$$
$$E_2 = \sqrt{1 - \text{probability}} \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1 - \text{probability}} \end{pmatrix}$$
$$E_3 = \sqrt{1 - \text{probability}} \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1 - \text{probability}} \end{pmatrix}$$

This noise channel is shown as GAD in circuit diagrams.

Initializes a GeneralizedAmplitudeDampingNoise.

**Parameters**

- **gamma** (*Union[FreeParameterExpression, float]*) – Probability of damping.
- **probability** (*Union[FreeParameterExpression, float]*) – Probability of the system being excited by the environment.
- **qubit\_count** (*Optional[int]*) – The number of qubits to apply noise.
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction.

**Raises**

**ValueError** – If `qubit_count < 1`, `ascii_symbols` is `None`, `len(ascii_symbols) != qubit_count`, `probability` or `gamma` is not `float` or `FreeParameterExpression`, `probability > 1.0` or `probability < 0.0`, or `gamma > 1.0` or `gamma < 0.0`.

**to\_matrix()** → *Iterable[ndarray]*

Returns a matrix representation of this noise.

**Returns**

*Iterable[ndarray]* – A list of matrix representations of this noise.

**static fixed\_qubit\_count()** → *int*

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static generalized\_amplitude\_damping**(*target: Qubit | int | Iterable[Qubit | int], gamma: float, probability: float*) → *Iterable[Instruction]*

Registers this function into the circuit class.

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s).
- **gamma** (*float*) – The damping rate of the amplitude damping channel.
- **probability** (*float*) – Probability of the system being excited by the environment.

**Returns**

*Iterable[Instruction]* – Iterable of GeneralizedAmplitudeDamping instructions.

**Examples**

```
>>> circ = Circuit().generalized_amplitude_damping(0, gamma=0.1, probability =
↳ 0.9)
```

**bind\_values**(*\*\*kwargs*) → *Noise*

Takes in parameters and attempts to assign them to values.

**Returns**

*Noise* – A new Noise object of the same type with the requested parameters bound.

**classmethod** `from_dict(noise: dict) → Noise`

Converts a dictionary representation of this class into this class.

**Parameters**

**noise** (dict) – The dictionary representation of this noise.

**Returns**

Noise – A Noise object that represents the passed in dictionary.

**class** `braket.circuits.noises.PhaseDamping(gamma: FreeParameterExpression | float)`

Bases: `DampingNoise`

Phase damping noise channel which transforms a density matrix  $\rho$  according to:

$$\rho \rightarrow E_0 \rho E_0^\dagger + E_1 \rho E_1^\dagger$$

where

$$E_0 = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-\gamma} \end{pmatrix}, \quad E_1 = \begin{pmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{pmatrix}$$

$\gamma$  = probability

This noise channel is shown as PD in circuit diagrams.

Initializes a DampingNoise.

**Parameters**

- **gamma** (Union[FreeParameterExpression, float]) – Probability of damping.
- **qubit\_count** (Optional[int]) – The number of qubits to apply noise.

- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the noise. These are used when printing a diagram of a circuit. The length must be the same as `qubit_count`, and index ordering is expected to correlate with the target ordering on the instruction.

**Raises**

**ValueError** – If `qubit_count < 1`, `ascii_symbols` is `None`, `len(ascii_symbols) != qubit_count`, `gamma` is not `float` or `FreeParameterExpression`, or `gamma > 1.0` or `gamma < 0.0`.

**to\_matrix()** → *Iterable[ndarray]*

Returns a matrix representation of this noise.

**Returns**

*Iterable[ndarray]* – A list of matrix representations of this noise.

**static fixed\_qubit\_count()** → `int`

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**static phase\_damping**(*target: Qubit | int | Iterable[Qubit | int]*, *gamma: float*) → *Iterable[Instruction]*

Registers this function into the circuit class.

**Parameters**

- **target** (*QubitSetInput*) – Target qubit(s)
- **gamma** (*float*) – Probability of phase damping.

**Returns**

*Iterable[Instruction]* – *Iterable* of `PhaseDamping` instructions.

**Examples**

```
>>> circ = Circuit().phase_damping(0, gamma=0.1)
```

**bind\_values**(*\*\*kwargs*) → *Noise*

Takes in parameters and attempts to assign them to values.

**Returns**

*Noise* – A new `Noise` object of the same type with the requested parameters bound.

**classmethod from\_dict**(*noise: dict*) → *Noise*

Converts a dictionary representation of this class into this class.

**Parameters**

**noise** (*dict*) – The dictionary representation of this noise.

**Returns**

*Noise* – A `Noise` object that represents the passed in dictionary.

**class** `braket.circuits.noises.Kraus`(*matrices: Iterable[ndarray]*, *display\_name: str = 'KR'*)

Bases: *Noise*

User-defined noise channel that uses the provided matrices as Kraus operators This noise channel is shown as NK in circuit diagrams.

Initializes *Kraus*.

#### Parameters

- **matrices** (*Iterable[ndarray]*) – A list of matrices that define a noise channel. These matrices need to satisfy the requirement of CPTP map.
- **display\_name** (*str*) – Name to be used for an instance of this general noise channel for circuit diagrams. Defaults to KR.

#### Raises

**ValueError** – If any matrix in **matrices** is not a two-dimensional square matrix, or has a dimension length which is not a positive exponent of 2, or the **matrices** do not satisfy CPTP condition.

**to\_matrix()** → *Iterable[ndarray]*

Returns a matrix representation of this noise.

#### Returns

*Iterable[ndarray]* – A list of matrix representations of this noise.

**static kraus**(*targets: Qubit | int | Iterable[Qubit | int], matrices: Iterable[array], display\_name: str = 'KR'*) → *Iterable[Instruction]*

Registers this function into the circuit class.

#### Parameters

- **targets** (*QubitSetInput*) – Target qubit(s)
- **matrices** (*Iterable[array]*) – Matrices that define a general noise channel.
- **display\_name** (*str*) – The display name.

#### Returns

*Iterable[Instruction]* – Iterable of Kraus instructions.

### Examples

```
>>> K0 = np.eye(4) * np.sqrt(0.9)
>>> K1 = np.kron([[1., 0.], [0., 1.]], [[0., 1.], [1., 0.]]) * np.sqrt(0.1)
>>> circ = Circuit().kraus([1, 0], matrices=[K0, K1])
```

**to\_dict()** → *dict*

Converts this object into a dictionary representation. Not implemented at this time.

#### Returns

*dict* – Not implemented at this time..

**classmethod from\_dict**(*noise: dict*) → *Noise*

Converts a dictionary representation of this class into this class.

#### Parameters

**noise** (*dict*) – The dictionary representation of this noise.

#### Returns

*Noise* – A Noise object that represents the passed in dictionary.

**braket.circuits.observable module**

**class** `braket.circuits.observable.Observable`(*qubit\_count*: *int*, *ascii\_symbols*: *Sequence[str]*)

Bases: `QuantumOperator`

Class `Observable` to represent a quantum observable.

Objects of this type can be used as input to `ResultType.Sample`, `ResultType.Variance`, `ResultType.Expectation` to specify the measurement basis.

Initializes a `QuantumOperator`.

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this quantum operator acts on. If all instances of the operator act on the same number of qubits, this argument should be `None`, and `fixed_qubit_count` should be implemented to return the qubit count; if `fixed_qubit_count` is implemented and an `int` is passed in, it must equal `fixed_qubit_count`, or instantiation will raise a `ValueError`. An `int` must be passed in if instances can have a varying number of qubits, in which case `fixed_qubit_count` should not be implemented,
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the quantum operator. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have [`"C"`, `"X"`] to correlate a symbol with that index.

**Raises**

- **TypeError** – `qubit_count` is not an `int`
- **ValueError** – `qubit_count` is less than 1, `ascii_symbols` are `None`, `fixed_qubit_count` is implemented and not equal to `qubit_count`, or `len(ascii_symbols) != qubit_count`

**to\_ir**(*target*: `QubitSet` | `None` = `None`, *ir\_type*: `IRType` = `IRType.JAQCD`, *serialization\_properties*: `SerializationProperties` | `None` = `None`) → `str` | `list[str]` | `list[list[float]]`

Returns the IR representation for the observable

**Parameters**

- **target** (`QubitSet` | `None`) – target qubit(s). Defaults to `None`.
- **ir\_type** (`IRType`) – The `IRType` to use for converting the result type object to its IR representation. Defaults to `IRType.JAQCD`.
- **serialization\_properties** (`SerializationProperties` | `None`) – The serialization properties to use while serializing the object to the IR representation. The serialization properties supplied must correspond to the supplied `ir_type`. Defaults to `None`.

**Returns**

`Union[str, list[Union[str, list[list[list[float]]]]]` – The IR representation for the observable.

**Raises**

**ValueError** – If the supplied `ir_type` is not supported, or if the supplied serialization properties don't correspond to the `ir_type`.

**property coefficient:** `int`

The coefficient of the observable.

**Returns**

*int* – coefficient value of the observable.

**property basis\_rotation\_gates:** `tuple[Gate, ...]`

Returns the basis rotation gates for this observable.

**Returns**

*tuple[Gate, ...]* – The basis rotation gates for this observable.

**property eigenvalues:** `ndarray`

Returns the eigenvalues of this observable.

**Returns**

*np.ndarray* – The eigenvalues of this observable.

**eigenvalue**(*index: int*) → `float`

Returns the eigenvalue of this observable at the given index.

The eigenvalues are ordered by their corresponding computational basis state after diagonalization.

**Parameters**

**index** (*int*) – The index of the desired eigenvalue

**Returns**

*float* – The index th eigenvalue of the observable.

**classmethod register\_observable**(*observable: Observable*) → `None`

Register an observable implementation by adding it into the *Observable* class.

**Parameters**

**observable** (*Observable*) – Observable class to register.

**class H**

Bases: *StandardObservable*

Hadamard operation as an observable.

Examples: >>> Observable.H()

**property basis\_rotation\_gates:** `tuple[Gate, ...]`

Returns the basis rotation gates for this observable.

**Returns**

*tuple[Gate, ...]* – The basis rotation gates for this observable.

**to\_matrix**() → `ndarray`

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class Hermitian**(*matrix: ndarray, display\_name: str = 'Hermitian'*)

Bases: *Observable*

Hermitian matrix as an observable.

Initiates a Hermitian.

**Parameters**



- **matrix** (*np.ndarray*) – Hermitian matrix that defines the observable.
- **display\_name** (*str*) – Name to use for an instance of this Hermitian matrix observable for circuit diagrams. Defaults to *Hermitian*.

**Raises**

**ValueError** – If **matrix** is not a two-dimensional square matrix, or has a dimension length that is not a positive power of 2, or is not Hermitian.

**Examples**

```
>>> Observable.Hermitian(matrix=np.array([[0, 1],[1, 0]]))
```

**property basis\_rotation\_gates:** *tuple[Gate, ...]*

Returns the basis rotation gates for this observable.

**Returns**

*tuple[Gate, ...]* – The basis rotation gates for this observable.

**eigenvalue**(*index: int*) → *float*

Returns the eigenvalue of this observable at the given index.

The eigenvalues are ordered by their corresponding computational basis state after diagonalization.

**Parameters**

**index** (*int*) – The index of the desired eigenvalue

**Returns**

*float* – The index th eigenvalue of the observable.

**property eigenvalues:** *ndarray*

Returns the eigenvalues of this observable.

**Returns**

*np.ndarray* – The eigenvalues of this observable.

**to\_matrix**() → *ndarray*

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class I**

Bases: *Observable*

Identity operation as an observable.

Examples: >>> Observable.I()

**property basis\_rotation\_gates:** *tuple[Gate, ...]*

Returns the basis rotation gates for this observable.

**Returns**

*tuple[Gate, ...]* – The basis rotation gates for this observable.

**eigenvalue**(*index: int*) → *float*

Returns the eigenvalue of this observable at the given index.

The eigenvalues are ordered by their corresponding computational basis state after diagonalization.

**Parameters****index** (*int*) – The index of the desired eigenvalue**Returns***float* – The index th eigenvalue of the observable.**property eigenvalues: ndarray**

Returns the eigenvalues of this observable.

**Returns***np.ndarray* – The eigenvalues of this observable.**to\_matrix()** → *ndarray*

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises****NotImplementedError** – Not Implemented.**Returns***np.ndarray* – A matrix representation of the quantum operator**class Sum**(*observables: list[Observable], display\_name: str = 'Hamiltonian'*)Bases: *Observable*

Sum of observables

Inits a Sum.

**Parameters**

- **observables** (*list[Observable]*) – List of observables for Sum
- **display\_name** (*str*) – Name to use for an instance of this Sum observable for circuit diagrams. Defaults to Hamiltonian.

**Examples**

```
>>> t1 = -3 * Observable.Y() + 2 * Observable.X()
Sum(X('qubit_count': 1), Y('qubit_count': 1))
>>> t1.summands
(X('qubit_count': 1), Y('qubit_count': 1))
```

**property basis\_rotation\_gates: tuple[Gate, ...]**

Returns the basis rotation gates for this observable.

**Returns***tuple[Gate, ...]* – The basis rotation gates for this observable.**eigenvalue(index: int)** → *float*

Returns the eigenvalue of this observable at the given index.

The eigenvalues are ordered by their corresponding computational basis state after diagonalization.

**Parameters****index** (*int*) – The index of the desired eigenvalue**Returns***float* – The index th eigenvalue of the observable.

**property eigenvalues:** `ndarray`

Returns the eigenvalues of this observable.

**Returns**

`np.ndarray` – The eigenvalues of this observable.

**property summands:** `tuple[Observable, ...]`

The observables that comprise this sum.

**Type**

`tuple[Observable]`

**to\_matrix()** → `ndarray`

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

`np.ndarray` – A matrix representation of the quantum operator

**class TensorProduct**(*observables: list[Observable]*)

Bases: `Observable`

Tensor product of observables

Initializes a TensorProduct.

**Parameters**

**observables** (*list[Observable]*) – List of observables for tensor product

## Examples

```
>>> t1 = Observable.Y() @ Observable.X()
>>> t1.to_matrix()
array([[0.+0.j, 0.+0.j, 0.-0.j, 0.-1.j],
       [0.+0.j, 0.+0.j, 0.-1.j, 0.-0.j],
       [0.+0.j, 0.+1.j, 0.+0.j, 0.+0.j],
       [0.+1.j, 0.+0.j, 0.+0.j, 0.+0.j]])
>>> t2 = Observable.Z() @ t1
>>> t2.factors
(Z('qubit_count': 1), Y('qubit_count': 1), X('qubit_count': 1))
```

Note: You must provide the list of observables for the tensor product to be evaluated in the order that you want the tensor product to be calculated. For `TensorProduct(observables=[ob1, ob2, ob3])`, the tensor product's matrix is the result of the tensor product of `ob1`, `ob2`, `ob3`, or `np.kron(np.kron(ob1.to_matrix(), ob2.to_matrix()), ob3.to_matrix())`.

**property ascii\_symbols:** `tuple[str, ...]`

Returns the ascii symbols for the quantum operator.

**Type**

`tuple[str, ...]`

**property basis\_rotation\_gates:** `tuple[Gate, ...]`

Returns the basis rotation gates for this observable.

**Returns**

`tuple[Gate, ...]` – The basis rotation gates for this observable.

**eigenvalue**(*index: int*) → float

Returns the eigenvalue of this observable at the given index.

The eigenvalues are ordered by their corresponding computational basis state after diagonalization.

**Parameters**

**index** (*int*) – The index of the desired eigenvalue

**Returns**

*float* – The index th eigenvalue of the observable.

**property eigenvalues: ndarray**

Returns the eigenvalues of this observable.

**Returns**

*np.ndarray* – The eigenvalues of this observable.

**property factors: tuple[Observable, ...]**

The observables that comprise this tensor product.

**Type**

*tuple[Observable]*

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class X**

Bases: *StandardObservable*

Pauli-X operation as an observable.

Examples: >>> Observable.X()

**property basis\_rotation\_gates: tuple[Gate, ...]**

Returns the basis rotation gates for this observable.

**Returns**

*tuple[Gate, ...]* – The basis rotation gates for this observable.

**to\_matrix()** → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class Y**

Bases: *StandardObservable*

Pauli-Y operation as an observable.

Examples: >>> Observable.Y()

**property basis\_rotation\_gates:** `tuple[Gate, ...]`

Returns the basis rotation gates for this observable.

**Returns**

*tuple[Gate, ...]* – The basis rotation gates for this observable.

**to\_matrix()**  $\rightarrow$  ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class Z**

Bases: *StandardObservable*

Pauli-Z operation as an observable.

Examples: `>>> Observable.Z()`

**property basis\_rotation\_gates:** `tuple[Gate, ...]`

Returns the basis rotation gates for this observable.

**Returns**

*tuple[Gate, ...]* – The basis rotation gates for this observable.

**to\_matrix()**  $\rightarrow$  ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**class** `braket.circuits.observable.StandardObservable(ascii_symbols: Sequence[str])`

Bases: *Observable*

Class *StandardObservable* to represent a Pauli-like quantum observable with eigenvalues of (+1, -1).

Initializes a QuantumOperator.

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this quantum operator acts on. If all instances of the operator act on the same number of qubits, this argument should be `None`, and `fixed_qubit_count` should be implemented to return the qubit count; if `fixed_qubit_count` is implemented and an `int` is passed in, it must equal `fixed_qubit_count`, or instantiation will raise a `ValueError`. An `int` must be passed in if instances can have a varying number of qubits, in which case `fixed_qubit_count` should not be implemented,
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the quantum operator. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and

target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises**

- **TypeError** – qubit\_count is not an int
- **ValueError** – qubit\_count is less than 1, `ascii_symbols` are None, fixed\_qubit\_count is implemented and not equal to qubit\_count, or `len(ascii_symbols) != qubit_count`

**property eigenvalues:** ndarray

Returns the eigenvalues of this observable.

**Returns**

*np.ndarray* – The eigenvalues of this observable.

**eigenvalue**(index: int) → float

Returns the eigenvalue of this observable at the given index.

The eigenvalues are ordered by their corresponding computational basis state after diagonalization.

**Parameters**

**index** (int) – The index of the desired eigenvalue

**Returns**

*float* – The index th eigenvalue of the observable.

**property ascii\_symbols:** tuple[str, ...]

Returns the ascii symbols for the quantum operator.

**Type**

tuple[str, ...]

## braket.circuits.observables module

**class** braket.circuits.observables.H

Bases: *StandardObservable*

Hadamard operation as an observable.

Examples: >>> Observable.H()

**to\_matrix**() → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (Any) – Not Implemented.
- **\*\*kwargs** (Any) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**property basis\_rotation\_gates:** tuple[Gate, ...]

Returns the basis rotation gates for this observable.

**Returns**

*tuple[Gate, ...]* – The basis rotation gates for this observable.

**class** `braket.circuits.observables.I`

Bases: *Observable*

Identity operation as an observable.

Examples: `>>> Observable.I()`

**to\_matrix()** → `ndarray`

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**property** `basis_rotation_gates:` `tuple[Gate, ...]`

Returns the basis rotation gates for this observable.

**Returns**

*tuple[Gate, ...]* – The basis rotation gates for this observable.

**property** `eigenvalues:` `ndarray`

Returns the eigenvalues of this observable.

**Returns**

*np.ndarray* – The eigenvalues of this observable.

**eigenvalue**(*index: int*) → `float`

Returns the eigenvalue of this observable at the given index.

The eigenvalues are ordered by their corresponding computational basis state after diagonalization.

**Parameters**

**index** (*int*) – The index of the desired eigenvalue

**Returns**

*float* – The index th eigenvalue of the observable.

**class** `braket.circuits.observables.X`

Bases: *StandardObservable*

Pauli-X operation as an observable.

Examples: `>>> Observable.X()`

**to\_matrix()** → `ndarray`

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**property basis\_rotation\_gates:** **tuple**[*Gate*, ...]

Returns the basis rotation gates for this observable.

**Returns**

*tuple*[*Gate*, ...] – The basis rotation gates for this observable.

**class** `braket.circuits.observables.Y`

Bases: *StandardObservable*

Pauli-Y operation as an observable.

Examples: >>> `Observable.Y()`

**to\_matrix()** → *ndarray*

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**property basis\_rotation\_gates:** **tuple**[*Gate*, ...]

Returns the basis rotation gates for this observable.

**Returns**

*tuple*[*Gate*, ...] – The basis rotation gates for this observable.

**class** `braket.circuits.observables.Z`

Bases: *StandardObservable*

Pauli-Z operation as an observable.

Examples: >>> `Observable.Z()`

**to\_matrix()** → *ndarray*

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator



**property basis\_rotation\_gates:** `tuple[Gate, ...]`

Returns the basis rotation gates for this observable.

**Returns**

`tuple[Gate, ...]` – The basis rotation gates for this observable.

**class** `braket.circuits.observables.TensorProduct(observables: list[Observable])`

Bases: `Observable`

Tensor product of observables

Initializes a `TensorProduct`.

**Parameters**

**observables** (`list[Observable]`) – List of observables for tensor product

**Examples**

```
>>> t1 = Observable.Y() @ Observable.X()
>>> t1.to_matrix()
array([[0.+0.j, 0.+0.j, 0.-0.j, 0.-1.j],
       [0.+0.j, 0.+0.j, 0.-1.j, 0.-0.j],
       [0.+0.j, 0.+1.j, 0.+0.j, 0.+0.j],
       [0.+1.j, 0.+0.j, 0.+0.j, 0.+0.j]])
>>> t2 = Observable.Z() @ t1
>>> t2.factors
(Z('qubit_count': 1), Y('qubit_count': 1), X('qubit_count': 1))
```

Note: You must provide the list of observables for the tensor product to be evaluated in the order that you want the tensor product to be calculated. For `TensorProduct(observables=[ob1, ob2, ob3])`, the tensor product's matrix is the result of the tensor product of ob1, ob2, ob3, or `np.kron(np.kron(ob1.to_matrix(), ob2.to_matrix()), ob3.to_matrix())`.

**property ascii\_symbols:** `tuple[str, ...]`

Returns the ascii symbols for the quantum operator.

**Type**

`tuple[str, ...]`

**property factors:** `tuple[Observable, ...]`

The observables that comprise this tensor product.

**Type**

`tuple[Observable]`

**to\_matrix()**  $\rightarrow$  `ndarray`

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

`np.ndarray` – A matrix representation of the quantum operator

**property** `basis_rotation_gates: tuple[Gate, ...]`

Returns the basis rotation gates for this observable.

**Returns**

`tuple[Gate, ...]` – The basis rotation gates for this observable.

**property** `eigenvalues: ndarray`

Returns the eigenvalues of this observable.

**Returns**

`np.ndarray` – The eigenvalues of this observable.

**eigenvalue**(`index: int`) → float

Returns the eigenvalue of this observable at the given index.

The eigenvalues are ordered by their corresponding computational basis state after diagonalization.

**Parameters**

**index** (`int`) – The index of the desired eigenvalue

**Returns**

`float` – The index th eigenvalue of the observable.

**class** `braket.circuits.observables.Sum(observables: list[Observable], display_name: str = 'Hamiltonian')`

Bases: `Observable`

Sum of observables

Initiates a `Sum`.

**Parameters**

- **observables** (`list[Observable]`) – List of observables for Sum
- **display\_name** (`str`) – Name to use for an instance of this Sum observable for circuit diagrams. Defaults to `Hamiltonian`.

## Examples

```
>>> t1 = -3 * Observable.Y() + 2 * Observable.X()
Sum(X('qubit_count': 1), Y('qubit_count': 1))
>>> t1.summands
(X('qubit_count': 1), Y('qubit_count': 1))
```

**property** `summands: tuple[Observable, ...]`

The observables that comprise this sum.

**Type**

`tuple[Observable]`

**to\_matrix**() → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (`Any`) – Not Implemented.
- **\*\*kwargs** (`Any`) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**property basis\_rotation\_gates:** `tuple[Gate, ...]`

Returns the basis rotation gates for this observable.

**Returns**

*tuple[Gate, ...]* – The basis rotation gates for this observable.

**property eigenvalues:** `ndarray`

Returns the eigenvalues of this observable.

**Returns**

*np.ndarray* – The eigenvalues of this observable.

**eigenvalue**(*index: int*) → float

Returns the eigenvalue of this observable at the given index.

The eigenvalues are ordered by their corresponding computational basis state after diagonalization.

**Parameters**

**index** (*int*) – The index of the desired eigenvalue

**Returns**

*float* – The index th eigenvalue of the observable.

**class** `braket.circuits.observables.Hermitian`(*matrix: ndarray, display\_name: str = 'Hermitian'*)

Bases: [Observable](#)

Hermitian matrix as an observable.

Initiates a [Hermitian](#).

**Parameters**

- **matrix** (*np.ndarray*) – Hermitian matrix that defines the observable.
- **display\_name** (*str*) – Name to use for an instance of this Hermitian matrix observable for circuit diagrams. Defaults to [Hermitian](#).

**Raises**

**ValueError** – If *matrix* is not a two-dimensional square matrix, or has a dimension length that is not a positive power of 2, or is not Hermitian.

**Examples**

```
>>> Observable.Hermitian(matrix=np.array([[0, 1],[1, 0]]))
```

**to\_matrix**() → `ndarray`

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (*Any*) – Not Implemented.
- **\*\*kwargs** (*Any*) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**property basis\_rotation\_gates:** `tuple[Gate, ...]`

Returns the basis rotation gates for this observable.

**Returns**

*tuple[Gate, ...]* – The basis rotation gates for this observable.

**property eigenvalues:** `ndarray`

Returns the eigenvalues of this observable.

**Returns**

*np.ndarray* – The eigenvalues of this observable.

**eigenvalue**(*index: int*)  $\rightarrow$  float

Returns the eigenvalue of this observable at the given index.

The eigenvalues are ordered by their corresponding computational basis state after diagonalization.

**Parameters**

**index** (*int*) – The index of the desired eigenvalue

**Returns**

*float* – The index th eigenvalue of the observable.

**braket.circuits.observables.observable\_from\_ir**(*ir\_observable: list[str | list[list[list[float]]]]*)  $\rightarrow$  *Observable*

Create an observable from the IR observable list. This can be a tensor product of observables or a single observable.

**Parameters**

**ir\_observable** (*list[Union[str, list[list[list[float]]]]*) – observable as defined in IR

**Returns**

*Observable* – observable object

## braket.circuits.operator module

**class** `braket.circuits.operator.Operator`

Bases: ABC

An operator is the abstract definition of an operation for a quantum device.

**abstract property name:** `str`

The name of the operator.

**Returns**

*str* – The name of the operator.

**abstract to\_ir**(*\*args, \*\*kwargs*)  $\rightarrow$  Any

Converts the operator into the canonical intermediate representation. If the operator is passed in a request, this method is called before it is passed.

**Returns**

*Any* – The canonical intermediate representation of the operator.

**braket.circuits.parameterizable module****braket.circuits.quantum\_operator module**

**class** `braket.circuits.quantum_operator.QuantumOperator`(*qubit\_count: int | None, ascii\_symbols: Sequence[str]*)

Bases: *Operator*

A quantum operator is the definition of a quantum operation for a quantum device.

Initializes a *QuantumOperator*.

**Parameters**

- **qubit\_count** (*Optional[int]*) – Number of qubits this quantum operator acts on. If all instances of the operator act on the same number of qubits, this argument should be `None`, and `fixed_qubit_count` should be implemented to return the qubit count; if `fixed_qubit_count` is implemented and an `int` is passed in, it must equal `fixed_qubit_count`, or instantiation will raise a `ValueError`. An `int` must be passed in if instances can have a varying number of qubits, in which case `fixed_qubit_count` should not be implemented,
- **ascii\_symbols** (*Sequence[str]*) – ASCII string symbols for the quantum operator. These are used when printing a diagram of circuits. Length must be the same as `qubit_count`, and index ordering is expected to correlate with target ordering on the instruction. For instance, if CNOT instruction has the control qubit on the first index and target qubit on the second index. Then ASCII symbols would have ["C", "X"] to correlate a symbol with that index.

**Raises**

- **TypeError** – `qubit_count` is not an `int`
- **ValueError** – `qubit_count` is less than 1, `ascii_symbols` are `None`, `fixed_qubit_count` is implemented and not equal to `qubit_count`, or `len(ascii_symbols) != qubit_count`

**static** `fixed_qubit_count()` → `int`

Returns the number of qubits this quantum operator acts on, if instances are guaranteed to act on the same number of qubits.

If different instances can act on a different number of qubits, this method returns `NotImplemented`.

**Returns**

*int* – The number of qubits this quantum operator acts on.

**property** `qubit_count: int`

The number of qubits this quantum operator acts on.

**Type**

`int`

**property** `ascii_symbols: tuple[str, ...]`

Returns the ascii symbols for the quantum operator.

**Type**

`tuple[str, ...]`

**property name:** `str`

Returns the name of the quantum operator

**Returns**

*str* – The name of the quantum operator as a string

**to\_ir**(\*args: Any, \*\*kwargs: Any) → Any

Returns IR representation of quantum operator.

**Parameters**

- **\*args** (Any) – Not Implemented.
- **\*\*kwargs** (Any) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*Any* – The canonical intermediate representation of the operator.

**to\_matrix**(\*args: Any, \*\*kwargs: Any) → ndarray

Returns a matrix representation of the quantum operator.

**Parameters**

- **\*args** (Any) – Not Implemented.
- **\*\*kwargs** (Any) – Not Implemented.

**Raises**

**NotImplementedError** – Not Implemented.

**Returns**

*np.ndarray* – A matrix representation of the quantum operator

**matrix\_equivalence**(other: [QuantumOperator](#)) → bool

Whether the matrix form of two quantum operators are equivalent

**Parameters**

**other** ([QuantumOperator](#)) – Quantum operator instance to compare this quantum operator to

**Returns**

*bool* – If matrix forms of this quantum operator and the other quantum operator are equivalent

## **braket.circuits.quantum\_operator\_helpers module**

`braket.circuits.quantum_operator_helpers.verify_quantum_operator_matrix_dimensions`(*matrix*:  
*ndarray*) →  
*None*

Verifies matrix is square and matrix dimensions are positive powers of 2, raising `ValueError` otherwise.

**Parameters**

**matrix** (*ndarray*) – matrix to verify

**Raises**

**ValueError** – If **matrix** is not a two-dimensional square matrix, or has a dimension length that is not a positive power of 2

`braket.circuits.quantum_operator_helpers.is_hermitian(matrix: ndarray) → bool`

Whether matrix is Hermitian

A square matrix  $U$  is Hermitian if

$$U = U^\dagger$$

where  $U^\dagger$  is the conjugate transpose of  $U$ .

**Parameters**

**matrix** (*ndarray*) – matrix to verify

**Returns**

*bool* – If matrix is Hermitian

`braket.circuits.quantum_operator_helpers.is_square_matrix(matrix: ndarray) → bool`

Whether matrix is square, meaning it has exactly two dimensions and the dimensions are equal

**Parameters**

**matrix** (*np.ndarray*) – matrix to verify

**Returns**

*bool* – If matrix is square

`braket.circuits.quantum_operator_helpers.is_unitary(matrix: ndarray) → bool`

Whether matrix is unitary

A square matrix  $U$  is unitary if

$$UU^\dagger = I$$

where  $U^\dagger$  is the conjugate transpose of  $U$  and  $I$  is the identity matrix.

**Parameters**

**matrix** (*np.ndarray*) – matrix to verify

**Returns**

*bool* – If matrix is unitary

`braket.circuits.quantum_operator_helpers.is_cptp(matrices: Iterable[ndarray]) → bool`

Whether a transformation defined by these matrices as Kraus operators is a completely positive trace preserving (CPTP) map. This is the requirement for a transformation to be a quantum channel. Reference: Section 8.2.3 in Nielsen & Chuang (2010) 10th edition.

**Parameters**

**matrices** (*Iterable[ndarray]*) – List of matrices representing Kraus operators.

**Returns**

*bool* – If the matrices define a CPTP map.

`braket.circuits.quantum_operator_helpers.get_pauli_eigenvalues(num_qubits: int) → ndarray`

Get the eigenvalues of Pauli operators and their tensor products as an immutable Numpy ndarray.

**Parameters**

**num\_qubits** (*int*) – the number of qubits the operator acts on

**Returns**

*np.ndarray* – the eigenvalues of a Pauli product operator of the given size

**braket.circuits.qubit module****braket.circuits.qubit\_set module****braket.circuits.result\_type module****class** `braket.circuits.result_type.ResultType`(*ascii\_symbols: list[str]*)Bases: `object`

Class `ResultType` represents a requested result type for the circuit. This class is considered the result type definition containing the metadata that defines what a requested result type is and what it does.

Initializes a `ResultType`.

**Parameters**

**ascii\_symbols** (*list[str]*) – ASCII string symbols for the result type. This is used when printing a diagram of circuits.

**Raises**

**ValueError** – *ascii\_symbols* is None

**property** `ascii_symbols: list[str]`

Returns the ascii symbols for the requested result type.

**Type**

`list[str]`

**property** `name: str`

Returns the name of the result type

**Returns**

*str* – The name of the result type as a string

**to\_ir**(*ir\_type: IRType = IRType.JAQCD, serialization\_properties: SerializationProperties | None = None, \*\*kwargs*) → *Any*

Returns IR object of the result type

**Parameters**

- **ir\_type** (*IRType*) – The *IRType* to use for converting the result type object to its IR representation. Defaults to *IRType.JAQCD*.
- **serialization\_properties** (*SerializationProperties | None*) – The serialization properties to use while serializing the object to the IR representation. The serialization properties supplied must correspond to the supplied *ir\_type*. Defaults to None.

**Returns**

*Any* – IR object of the result type

**Raises**

**ValueError** – If the supplied *ir\_type* is not supported, or if the supplied serialization properties don't correspond to the *ir\_type*.

**copy**(*target\_mapping: dict[QubitInput, QubitInput] | None = None, target: QubitSetInput | None = None*) → `ResultType`

Return a shallow copy of the result type.



---

**Note:** If `target_mapping` is specified, then `self.target` is mapped to the specified qubits. This is useful apply an instruction to a circuit and change the target qubits.

---

#### Parameters

- **target\_mapping** (*dict*[*QubitInput*, *QubitInput*] | *None*) – A dictionary of qubit mappings to apply to the target. Key is the qubit in this target and the value is what the key is changed to. Default = *None*.
- **target** (*QubitSetInput* | *None*) – Target qubits for the new instruction.

#### Returns

*ResultType* – A shallow copy of the result type.

#### Raises

**TypeError** – If both `target_mapping` and `target` are supplied.

### Examples

```
>>> result_type = ResultType.Probabilities(targets=[0])
>>> new_result_type = result_type.copy()
>>> new_result_type.targets
QubitSet(Qubit(0))
>>> new_result = result_type.copy(target_mapping={0: 5})
>>> new_result_type.target
QubitSet(Qubit(5))
>>> new_result = result_type.copy(target=[5])
>>> new_result_type.target
QubitSet(Qubit(5))
```

**classmethod** `register_result_type`(*result\_type: type*[*ResultType*]) → *None*

Register a result type implementation by adding it into the *ResultType* class.

#### Parameters

**result\_type** (*type*[*ResultType*]) – *ResultType* class to register.

**class** `AdjointGradient`(*observable: Observable*, *target: list*[*QubitSetInput*] | *None* = *None*, *parameters: list*[*str* | *FreeParameter*] | *None* = *None*)

Bases: *ObservableParameterResultType*

The gradient of the expectation value of the provided observable, applied to target, with respect to the given parameter.

Initiates an `AdjointGradient`.

#### Parameters

- **observable** (*Observable*) – The expectation value of this observable is the function against which parameters in the gradient are differentiated.
- **target** (*list*[*QubitSetInput*] | *None*) – Target qubits that the result type is requested for. Each term in the target list should have the same number of qubits as the corresponding term in the observable. Default is *None*, which means the observable must operate only on 1 qubit and it is applied to all qubits in parallel.

- **parameters** (*list[Union[str, FreeParameter]]* | *None*) – The free parameters in the circuit to differentiate with respect to. Default: all.

#### Raises

**ValueError** – If the observable’s qubit count does not equal the number of target qubits, or if `target=None` and the observable’s qubit count is not 1.

### Examples

```
>>> ResultType.AdjointGradient(observable=Observable.Z(),
                                target=0, parameters=["alpha", "beta"])
```

```
>>> tensor_product = Observable.Y() @ Observable.Z()
>>> hamiltonian = Observable.Y() @ Observable.Z() + Observable.H()
>>> ResultType.AdjointGradient(
>>>     observable=tensor_product,
>>>     target=[[0, 1], [2]],
>>>     parameters=["alpha", "beta"],
>>> )
```

**static adjoint\_gradient**(*observable: Observable, target: list[QubitSetInput] | None = None, parameters: list[str | FreeParameter] | None = None*) → *ResultType*

Registers this function into the circuit class.

#### Parameters

- **observable** (*Observable*) – The expectation value of this observable is the function against which parameters in the gradient are differentiated.
- **target** (*list[QubitSetInput]* | *None*) – Target qubits that the result type is requested for. Each term in the target list should have the same number of qubits as the corresponding term in the observable. Default is *None*, which means the observable must operate only on 1 qubit and it is applied to all qubits in parallel.
- **parameters** (*list[Union[str, FreeParameter]]* | *None*) – The free parameters in the circuit to differentiate with respect to. Default: all.

#### Returns

*ResultType* – gradient computed via adjoint differentiation as a requested result type

### Examples

```
>>> alpha, beta = FreeParameter('alpha'), FreeParameter('beta')
>>> circ = Circuit().h(0).h(1).rx(0, alpha).yy(0, 1, beta).adjoint_gradient(
>>>     observable=Observable.Z(), target=[0], parameters=[alpha, beta]
>>> )
```

**class Amplitude**(*state: list[str]*)

Bases: *ResultType*

The amplitude of the specified quantum states as a requested result type. This is available on simulators only when `shots=0`.

Initializes an *Amplitude*.

#### Parameters

**state** (*list[str]*) – list of quantum states as strings with “0” and “1”

**Raises**

**ValueError** – If state is None or an empty list, or state is not a list of strings of ‘0’ and ‘1’

**Examples**

```
>>> ResultType.Amplitude(state=['01', '10'])
```

**static** `amplitude(state: list[str]) → ResultType`

Registers this function into the circuit class.

**Parameters**

**state** (`list[str]`) – list of quantum states as strings with “0” and “1”

**Returns**

*ResultType* – state vector as a requested result type

**Examples**

```
>>> circ = Circuit().amplitude(state=["01", "10"])
```

**property** `state: list[str]`

**class** `DensityMatrix(target: QubitSetInput | None = None)`

Bases: *ResultType*

The full density matrix as a requested result type. This is available on simulators only when shots=0.

Init a DensityMatrix.

**Parameters**

**target** (`QubitSetInput | None`) – The target qubits of the reduced density matrix. Default is None, and the full density matrix is returned.

**Examples**

```
>>> ResultType.DensityMatrix(target=[0, 1])
```

**static** `density_matrix(target: QubitSetInput | None = None) → ResultType`

Registers this function into the circuit class.

**Parameters**

**target** (`QubitSetInput | None`) – The target qubits of the reduced density matrix. Default is None, and the full density matrix is returned.

**Returns**

*ResultType* – density matrix as a requested result type

## Examples

```
>>> circ = Circuit().density_matrix(target=[0, 1])
```

property **target**: *QubitSet*

**class Expectation**(*observable*: *Observable*, *target*: *QubitSetInput* | *None* = *None*)

Bases: *ObservableResultType*

Expectation of the specified target qubit set and observable as the requested result type.

If no targets are specified, the observable must operate only on 1 qubit and it is applied to all qubits in parallel. Otherwise, the number of specified targets must be equivalent to the number of qubits the observable can be applied to.

For `shots>0`, this is calculated by measurements. For `shots=0`, this is supported only by simulators and represents the exact result.

See `braket.circuits.observables` module for all of the supported observables.

Initiates an Expectation.

### Parameters

- **observable** (*Observable*) – the observable for the result type
- **target** (*QubitSetInput* | *None*) – Target qubits that the result type is requested for. Default is *None*, which means the observable must operate only on 1 qubit and it is applied to all qubits in parallel.

## Examples

```
>>> ResultType.Expectation(observable=Observable.Z(), target=0)
```

```
>>> tensor_product = Observable.Y() @ Observable.Z()
>>> ResultType.Expectation(observable=tensor_product, target=[0, 1])
```

**static expectation**(*observable*: *Observable*, *target*: *QubitSetInput* | *None* = *None*) → *ResultType*

Registers this function into the circuit class.

### Parameters

- **observable** (*Observable*) – the observable for the result type
- **target** (*QubitSetInput* | *None*) – Target qubits that the result type is requested for. Default is *None*, which means the observable must operate only on 1 qubit and it is applied to all qubits in parallel.

### Returns

*ResultType* – expectation as a requested result type

## Examples

```
>>> circ = Circuit().expectation(observable=Observable.Z(), target=0)
```

**class** `Probability`(*target: QubitSetInput | None = None*)

Bases: `ResultType`

Probability in the computational basis as the requested result type.

It can be the probability of all states if no targets are specified, or the marginal probability of a restricted set of states if only a subset of all qubits are specified as targets.

For `shots>0`, this is calculated by measurements. For `shots=0`, this is supported only on simulators and represents the exact result.

Initiates a `Probability`.

### Parameters

**target** (*QubitSetInput | None*) – The target qubits that the result type is requested for. Default is `None`, which means all qubits for the circuit.

## Examples

```
>>> ResultType.Probability(target=[0, 1])
```

**static** `probability`(*target: QubitSetInput | None = None*) → `ResultType`

Registers this function into the circuit class.

### Parameters

**target** (*QubitSetInput | None*) – The target qubits that the result type is requested for. Default is `None`, which means all qubits for the circuit.

### Returns

`ResultType` – probability as a requested result type

## Examples

```
>>> circ = Circuit().probability(target=[0, 1])
```

**property** `target`: `QubitSet`

**class** `Sample`(*observable: Observable, target: QubitSetInput | None = None*)

Bases: `ObservableResultType`

Sample of specified target qubit set and observable as the requested result type.

If no targets are specified, the observable must operate only on 1 qubit and it is applied to all qubits in parallel. Otherwise, the number of specified targets must equal the number of qubits the observable can be applied to.

This is only available for `shots>0`.

See `braket.circuits.observables` module for all of the supported observables.

Initiates a `Sample`.

### Parameters

- **observable** (`Observable`) – the observable for the result type

- **target** (*QubitSetInput* | *None*) – Target qubits that the result type is requested for. Default is *None*, which means the observable must operate only on 1 qubit and it is applied to all qubits in parallel.

### Examples

```
>>> ResultType.Sample(observable=Observable.Z(), target=0)
```

```
>>> tensor_product = Observable.Y() @ Observable.Z()
>>> ResultType.Sample(observable=tensor_product, target=[0, 1])
```

**static sample**(*observable*: *Observable*, *target*: *QubitSetInput* | *None* = *None*) → *ResultType*

Registers this function into the circuit class.

#### Parameters

- **observable** (*Observable*) – the observable for the result type
- **target** (*QubitSetInput* | *None*) – Target qubits that the result type is requested for. Default is *None*, which means the observable must operate only on 1 qubit and it is applied to all qubits in parallel.

#### Returns

*ResultType* – sample as a requested result type

### Examples

```
>>> circ = Circuit().sample(observable=Observable.Z(), target=0)
```

## class StateVector

Bases: *ResultType*

The full state vector as a requested result type. This is available on simulators only when *shots=0*.

Initializes a *ResultType*.

#### Parameters

**ascii\_symbols** (*list[str]*) – ASCII string symbols for the result type. This is used when printing a diagram of circuits.

#### Raises

**ValueError** – *ascii\_symbols* is *None*

**static state\_vector**() → *ResultType*

Registers this function into the circuit class.

#### Returns

*ResultType* – state vector as a requested result type

## Examples

```
>>> circ = Circuit().state_vector()
```

**class** `Variance`(*observable*: `Observable`, *target*: `QubitSetInput` | `None` = `None`)

Bases: `ObservableResultType`

Variance of specified target qubit set and observable as the requested result type.

If no targets are specified, the observable must operate only on 1 qubit and it is applied to all qubits in parallel. Otherwise, the number of targets specified must equal the number of qubits that the observable can be applied to.

For `shots>0`, this is calculated by measurements. For `shots=0`, this is supported only by simulators and represents the exact result.

See `braket.circuits.observables` module for all of the supported observables.

Initiates a `Variance`.

### Parameters

- **observable** (`Observable`) – the observable for the result type
- **target** (`QubitSetInput` | `None`) – Target qubits that the result type is requested for. Default is `None`, which means the observable must operate only on 1 qubit and it is applied to all qubits in parallel.

### Raises

**ValueError** – If the observable’s qubit count does not equal the number of target qubits, or if `target=None` and the observable’s qubit count is not 1.

## Examples

```
>>> ResultType.Variance(observable=Observable.Z(), target=0)
```

```
>>> tensor_product = Observable.Y() @ Observable.Z()
>>> ResultType.Variance(observable=tensor_product, target=[0, 1])
```

**static variance**(*observable*: `Observable`, *target*: `QubitSetInput` | `None` = `None`) → `ResultType`

Registers this function into the circuit class.

### Parameters

- **observable** (`Observable`) – the observable for the result type
- **target** (`QubitSetInput` | `None`) – Target qubits that the result type is requested for. Default is `None`, which means the observable must only operate on 1 qubit and it will be applied to all qubits in parallel

### Returns

`ResultType` – variance as a requested result type

## Examples

```
>>> circ = Circuit().variance(observable=Observable.Z(), target=0)
```

```
class braket.circuits.result_type.ObservableResultType(ascii_symbols: list[str], observable:
    Observable, target: QubitSetInput | None =
    None)
```

Bases: [ResultType](#)

Result types with observables and targets. If no targets are specified, the observable must only operate on 1 qubit and it will be applied to all qubits in parallel. Otherwise, the number of specified targets must be equivalent to the number of qubits the observable can be applied to.

See [braket.circuits.observables](#) module for all of the supported observables.

Initializes an [ObservableResultType](#).

### Parameters

- **ascii\_symbols** (*list[str]*) – ASCII string symbols for the result type. This is used when printing a diagram of circuits.
- **observable** ([Observable](#)) – the observable for the result type
- **target** (*QubitSetInput | None*) – Target qubits that the result type is requested for. Default is None, which means the observable must only operate on 1 qubit and it will be applied to all qubits in parallel

### Raises

**ValueError** – if target=None and the observable’s qubit count is not 1. Or, if target!=None and the observable’s qubit count and the number of target qubits are not equal. Or, if target!=None and the observable’s qubit count and the number of ascii\_symbols are not equal.

property observable: [Observable](#)

property target: [QubitSet](#)

```
class braket.circuits.result_type.ObservableParameterResultType(ascii_symbols: list[str],
    observable: Observable, target:
    QubitSetInput | None = None,
    parameters: list[str] |
    FreeParameter | None = None)
```

Bases: [ObservableResultType](#)

Result types with observables, targets and parameters. If no targets are specified, the observable must only operate on 1 qubit and it will be applied to all qubits in parallel. Otherwise, the number of specified targets must be equivalent to the number of qubits the observable can be applied to. If no parameters are specified, observable will be applied to all the free parameters.

See [braket.circuits.observables](#) module for all of the supported observables.

Initializes an [ObservableResultType](#).

### Parameters

- **ascii\_symbols** (*list[str]*) – ASCII string symbols for the result type. This is used when printing a diagram of circuits.
- **observable** ([Observable](#)) – the observable for the result type



- **target** (*QubitSetInput* | *None*) – Target qubits that the result type is requested for. Default is *None*, which means the observable must only operate on 1 qubit and it will be applied to all qubits in parallel

**Raises**

**ValueError** – if `target=None` and the observable's qubit count is not 1. Or, if `target!=None` and the observable's qubit count and the number of target qubits are not equal. Or, if `target!=None` and the observable's qubit count and the number of `ascii_symbols` are not equal.

**property parameters:** `list[str]`

**braket.circuits.result\_types module**

**class** `braket.circuits.result_types.StateVector`

Bases: *ResultType*

The full state vector as a requested result type. This is available on simulators only when `shots=0`.

Initializes a *ResultType*.

**Parameters**

**ascii\_symbols** (*list[str]*) – ASCII string symbols for the result type. This is used when printing a diagram of circuits.

**Raises**

**ValueError** – `ascii_symbols` is *None*

**static** `state_vector()` → *ResultType*

Registers this function into the circuit class.

**Returns**

*ResultType* – state vector as a requested result type

**Examples**

```
>>> circ = Circuit().state_vector()
```

**class** `braket.circuits.result_types.DensityMatrix(target: QubitSetInput | None = None)`

Bases: *ResultType*

The full density matrix as a requested result type. This is available on simulators only when `shots=0`.

Initiates a *DensityMatrix*.

**Parameters**

**target** (*QubitSetInput* | *None*) – The target qubits of the reduced density matrix. Default is *None*, and the full density matrix is returned.

## Examples

```
>>> ResultType.DensityMatrix(target=[0, 1])
```

property target: *QubitSet*

static *density\_matrix*(target: *QubitSetInput* | *None* = *None*) → *ResultType*

Registers this function into the circuit class.

### Parameters

**target** (*QubitSetInput* | *None*) – The target qubits of the reduced density matrix. Default is *None*, and the full density matrix is returned.

### Returns

*ResultType* – density matrix as a requested result type

## Examples

```
>>> circ = Circuit().density_matrix(target=[0, 1])
```

class *braket.circuits.result\_types.AdjointGradient*(*observable: Observable*, *target: list[QubitSetInput] | None = None*, *parameters: list[str | FreeParameter] | None = None*)

Bases: *ObservableParameterResultType*

The gradient of the expectation value of the provided observable, applied to target, with respect to the given parameter.

Initiates an *AdjointGradient*.

### Parameters

- **observable** (*Observable*) – The expectation value of this observable is the function against which parameters in the gradient are differentiated.
- **target** (*list[QubitSetInput]* | *None*) – Target qubits that the result type is requested for. Each term in the target list should have the same number of qubits as the corresponding term in the observable. Default is *None*, which means the observable must operate only on 1 qubit and it is applied to all qubits in parallel.
- **parameters** (*list[Union[str, FreeParameter]]* | *None*) – The free parameters in the circuit to differentiate with respect to. Default: *all*.

### Raises

**ValueError** – If the observable's qubit count does not equal the number of target qubits, or if *target=None* and the observable's qubit count is not 1.

## Examples

```
>>> ResultType.AdjointGradient(observable=Observable.Z(),
                               target=0, parameters=["alpha", "beta"])
```

```
>>> tensor_product = Observable.Y() @ Observable.Z()
>>> hamiltonian = Observable.Y() @ Observable.Z() + Observable.H()
>>> ResultType.AdjointGradient(
>>>     observable=tensor_product,
>>>     target=[[0, 1], [2]],
>>>     parameters=["alpha", "beta"],
>>> )
```

**static adjoint\_gradient**(*observable*: [Observable](#), *target*: *list*[*QubitSetInput*] | *None* = *None*,  
*parameters*: *list*[*str* | [FreeParameter](#)] | *None* = *None*) → [ResultType](#)

Registers this function into the circuit class.

### Parameters

- **observable** ([Observable](#)) – The expectation value of this observable is the function against which parameters in the gradient are differentiated.
- **target** (*list*[*QubitSetInput*] | *None*) – Target qubits that the result type is requested for. Each term in the target list should have the same number of qubits as the corresponding term in the observable. Default is *None*, which means the observable must operate only on 1 qubit and it is applied to all qubits in parallel.
- **parameters** (*list*[*Union*[*str*, [FreeParameter](#)]] | *None*) – The free parameters in the circuit to differentiate with respect to. Default: all.

### Returns

[ResultType](#) – gradient computed via adjoint differentiation as a requested result type

## Examples

```
>>> alpha, beta = FreeParameter('alpha'), FreeParameter('beta')
>>> circ = Circuit().h(0).h(1).rx(0, alpha).yy(0, 1, beta).adjoint_gradient(
>>>     observable=Observable.Z(), target=[0], parameters=[alpha, beta]
>>> )
```

**class** `braket.circuits.result_types.Amplitude`(*state*: *list*[*str*])

Bases: [ResultType](#)

The amplitude of the specified quantum states as a requested result type. This is available on simulators only when `shots=0`.

Initializes an [Amplitude](#).

### Parameters

**state** (*list*[*str*]) – list of quantum states as strings with “0” and “1”

### Raises

**ValueError** – If state is *None* or an empty list, or state is not a list of strings of ‘0’ and ‘1’

## Examples

```
>>> ResultType.Amplitude(state=['01', '10'])
```

**property state:** `list[str]`

**static amplitude**(state: list[str]) → *ResultType*

Registers this function into the circuit class.

### Parameters

**state** (list[str]) – list of quantum states as strings with “0” and “1”

### Returns

*ResultType* – state vector as a requested result type

## Examples

```
>>> circ = Circuit().amplitude(state=["01", "10"])
```

**class** `braket.circuits.result_types.Probability`(target: *QubitSetInput* | *None* = *None*)

Bases: *ResultType*

Probability in the computational basis as the requested result type.

It can be the probability of all states if no targets are specified, or the marginal probability of a restricted set of states if only a subset of all qubits are specified as targets.

For shots>0, this is calculated by measurements. For shots=0, this is supported only on simulators and represents the exact result.

Initiates a *Probability*.

### Parameters

**target** (*QubitSetInput* | *None*) – The target qubits that the result type is requested for. Default is *None*, which means all qubits for the circuit.

## Examples

```
>>> ResultType.Probability(target=[0, 1])
```

**property target:** *QubitSet*

**static probability**(target: *QubitSetInput* | *None* = *None*) → *ResultType*

Registers this function into the circuit class.

### Parameters

**target** (*QubitSetInput* | *None*) – The target qubits that the result type is requested for. Default is *None*, which means all qubits for the circuit.

### Returns

*ResultType* – probability as a requested result type

## Examples

```
>>> circ = Circuit().probability(target=[0, 1])
```

**class** `braket.circuits.result_types.Expectation`(*observable*: [Observable](#), *target*: *QubitSetInput* | *None* = *None*)

Bases: [ObservableResultType](#)

Expectation of the specified target qubit set and observable as the requested result type.

If no targets are specified, the observable must operate only on 1 qubit and it is applied to all qubits in parallel. Otherwise, the number of specified targets must be equivalent to the number of qubits the observable can be applied to.

For `shots>0`, this is calculated by measurements. For `shots=0`, this is supported only by simulators and represents the exact result.

See [braket.circuits.observables](#) module for all of the supported observables.

Initiates an [Expectation](#).

### Parameters

- **observable** ([Observable](#)) – the observable for the result type
- **target** (*QubitSetInput* | *None*) – Target qubits that the result type is requested for. Default is *None*, which means the observable must operate only on 1 qubit and it is applied to all qubits in parallel.

## Examples

```
>>> ResultType.Expectation(observable=Observable.Z(), target=0)
```

```
>>> tensor_product = Observable.Y() @ Observable.Z()
>>> ResultType.Expectation(observable=tensor_product, target=[0, 1])
```

**static expectation**(*observable*: [Observable](#), *target*: *QubitSetInput* | *None* = *None*) → [ResultType](#)

Registers this function into the circuit class.

### Parameters

- **observable** ([Observable](#)) – the observable for the result type
- **target** (*QubitSetInput* | *None*) – Target qubits that the result type is requested for. Default is *None*, which means the observable must operate only on 1 qubit and it is applied to all qubits in parallel.

### Returns

[ResultType](#) – expectation as a requested result type

## Examples

```
>>> circ = Circuit().expectation(observable=Observable.Z(), target=0)
```

```
class braket.circuits.result_types.Sample(observable: Observable, target: QubitSetInput | None = None)
```

Bases: [ObservableResultType](#)

Sample of specified target qubit set and observable as the requested result type.

If no targets are specified, the observable must operate only on 1 qubit and it is applied to all qubits in parallel. Otherwise, the number of specified targets must equal the number of qubits the observable can be applied to.

This is only available for `shots>0`.

See [braket.circuits.observables](#) module for all of the supported observables.

Initiates a *Sample*.

### Parameters

- **observable** ([Observable](#)) – the observable for the result type
- **target** ([QubitSetInput](#) | *None*) – Target qubits that the result type is requested for. Default is *None*, which means the observable must operate only on 1 qubit and it is applied to all qubits in parallel.

## Examples

```
>>> ResultType.Sample(observable=Observable.Z(), target=0)
```

```
>>> tensor_product = Observable.Y() @ Observable.Z()
>>> ResultType.Sample(observable=tensor_product, target=[0, 1])
```

```
static sample(observable: Observable, target: QubitSetInput | None = None) → ResultType
```

Registers this function into the circuit class.

### Parameters

- **observable** ([Observable](#)) – the observable for the result type
- **target** ([QubitSetInput](#) | *None*) – Target qubits that the result type is requested for. Default is *None*, which means the observable must operate only on 1 qubit and it is applied to all qubits in parallel.

### Returns

*ResultType* – sample as a requested result type

## Examples

```
>>> circ = Circuit().sample(observable=Observable.Z(), target=0)
```

**class** `braket.circuits.result_types.Variance`(*observable*: [Observable](#), *target*: *QubitSetInput* | *None* = *None*)

Bases: [ObservableResultType](#)

Variance of specified target qubit set and observable as the requested result type.

If no targets are specified, the observable must operate only on 1 qubit and it is applied to all qubits in parallel. Otherwise, the number of targets specified must equal the number of qubits that the observable can be applied to.

For `shots>0`, this is calculated by measurements. For `shots=0`, this is supported only by simulators and represents the exact result.

See [braket.circuits.observables](#) module for all of the supported observables.

Initiates a [Variance](#).

### Parameters

- **observable** ([Observable](#)) – the observable for the result type
- **target** (*QubitSetInput* | *None*) – Target qubits that the result type is requested for. Default is *None*, which means the observable must operate only on 1 qubit and it is applied to all qubits in parallel.

### Raises

**ValueError** – If the observable’s qubit count does not equal the number of target qubits, or if `target=None` and the observable’s qubit count is not 1.

## Examples

```
>>> ResultType.Variance(observable=Observable.Z(), target=0)
```

```
>>> tensor_product = Observable.Y() @ Observable.Z()
>>> ResultType.Variance(observable=tensor_product, target=[0, 1])
```

**static variance**(*observable*: [Observable](#), *target*: *QubitSetInput* | *None* = *None*) → [ResultType](#)

Registers this function into the circuit class.

### Parameters

- **observable** ([Observable](#)) – the observable for the result type
- **target** (*QubitSetInput* | *None*) – Target qubits that the result type is requested for. Default is *None*, which means the observable must only operate on 1 qubit and it will be applied to all qubits in parallel

### Returns

[ResultType](#) – variance as a requested result type

## Examples

```
>>> circ = Circuit().variance(observable=Observable.Z(), target=0)
```

### braket.circuits.serialization module

```
class braket.circuits.serialization.IRType(value)
```

Bases: str, Enum

Defines the available IRTypes for circuit serialization.

**OPENQASM** = 'OPENQASM'

**JAQCD** = 'JAQCD'

```
class braket.circuits.serialization.QubitReferenceType(value)
```

Bases: str, Enum

Defines how qubits should be referenced in the generated OpenQASM string. See <https://qiskit.github.io/openqasm/language/types.html#quantum-types> for details.

**VIRTUAL** = 'VIRTUAL'

**PHYSICAL** = 'PHYSICAL'

```
class braket.circuits.serialization.OpenQASMSerializationProperties(qubit_reference_type:
                                                                    QubitReferenceType =
                                                                    QubitReference-
                                                                    Type.VIRTUAL)
```

Bases: object

Properties for serializing a circuit to OpenQASM.

**qubit\_reference\_type** (*QubitReferenceType*): determines whether to use logical qubits or physical qubits (q[i] vs \$i).

**qubit\_reference\_type**: *QubitReferenceType* = 'VIRTUAL'

**format\_target**(*target: int*) → str

Format a target qubit to the appropriate OpenQASM representation.

**Parameters**

**target** (*int*) – The target qubit.

**Returns**

*str* – The OpenQASM representation of the target qubit.

```
braket.circuits.serialization.SerializationProperties
```

alias of *OpenQASMSerializationProperties*



**braket.circuits.translations module**

`braket.circuits.translations.get_observable(obs: Observable | list) → Observable`

Gets the observable.

**Parameters**

**obs** (*Union*[*Observable*, *list*]) – The observable(s) to get translated.

**Returns**

*Observable* – The translated observable.

`braket.circuits.translations.get_tensor_product(observable: Observable | list) → Observable`

Generate an braket circuit observable

**Parameters**

**observable** (*Union*[*Observable*, *list*]) – ir observable or a matrix

**Returns**

*Observable* – braket circuit observable

`braket.circuits.translations.braket_result_to_result_type(result: Amplitude | Expectation | Probability | Sample | StateVector | DensityMatrix | Variance | AdjointGradient) → None`

**braket.circuits.unitary\_calculation module**

`braket.circuits.unitary_calculation.calculate_unitary_big_endian(instructions: Iterable[Instruction], qubits: QubitSet) → ndarray`

Returns the unitary matrix representation for all the instruction`s on qubits `qubits.

---

**Note:** The performance of this method degrades with qubit count. It might be slow for `len(qubits) > 10`.

---

**Parameters**

- **instructions** (*Iterable*[*Instruction*]) – The instructions for which the unitary matrix will be calculated.
- **qubits** (*QubitSet*) – The actual qubits used by the instructions.

**Returns**

*np.ndarray* – A numpy array with shape  $(2^{\text{qubit\_count}}, 2^{\text{qubit\_count}})$  representing the instructions as a unitary matrix.

**Raises**

**TypeError** – If instructions is not composed only of Gate instances, i.e. a circuit with Noise operators will raise this error. Any *CompilerDirective* instructions will be ignored, as these should not affect the unitary representation of the circuit.

## braket.devices package

### Submodules

#### braket.devices.device module

**class** `braket.devices.device.Device(name: str, status: str)`

Bases: `ABC`

An abstraction over quantum devices that includes quantum computers and simulators.

Initializes a *Device*.

##### Parameters

- **name** (*str*) – Name of quantum device
- **status** (*str*) – Status of quantum device

**abstract run**(*task\_specification: Circuit | Problem, shots: int | None, inputs: dict[str, float] | None, \*args, \*\*kwargs*) → *QuantumTask*

Run a quantum task specification on this quantum device. A quantum task can be a circuit or an annealing problem.

##### Parameters

- **task\_specification** (*Union[Circuit, Problem]*) – Specification of a quantum task to run on device.
- **shots** (*Optional[int]*) – The number of times to run the quantum task on the device. Default is `None`.
- **inputs** (*Optional[dict[str, float]]*) – Inputs to be passed along with the IR. If IR is an OpenQASM Program, the inputs will be updated with this value. Not all devices and IR formats support inputs. Default: `{}`.
- **\*args** (*Any*) – Arbitrary arguments.
- **\*\*kwargs** (*Any*) – Arbitrary keyword arguments.

##### Returns

*QuantumTask* – The *QuantumTask* tracking task execution on this device

**abstract run\_batch**(*task\_specifications: Circuit | Problem | list[Circuit | Problem], shots: int | None, max\_parallel: int | None, inputs: dict[str, float] | list[dict[str, float]] | None, \*args: Any, \*\*kwargs: Any*) → *QuantumTaskBatch*

Executes a batch of quantum tasks in parallel

##### Parameters

- **task\_specifications** (*Union[Union[Circuit, Problem], list[Union[Circuit, Problem]]]*) – Single instance or list of circuits or problems to run on device.
- **shots** (*Optional[int]*) – The number of times to run the circuit or annealing problem.
- **max\_parallel** (*Optional[int]*) – The maximum number of quantum tasks to run in parallel. Batch creation will fail if this value is greater than the maximum allowed concurrent quantum tasks on the device.

- **inputs** (*Optional[Union[dict[str, float], list[dict[str, float]]]*) – Inputs to be passed along with the IR. If the IR supports inputs, the inputs will be updated with this value.
- **\*args** (*Any*) – Arbitrary arguments.
- **\*\*kwargs** (*Any*) – Arbitrary keyword arguments.

**Returns**

*QuantumTaskBatch* – A batch containing all of the quantum tasks run

**property name: str**

Return the name of this Device.

**Returns**

*str* – The name of this Device

**property status: str**

Return the status of this Device.

**Returns**

*str* – The status of this Device

**braket.devices.devices module**

**class** `braket.devices.devices.Devices`

Bases: `object`

**Amazon**

alias of `_Amazon`

**IonQ**

alias of `_IonQ`

**OQC**

alias of `_OQC`

**QuEra**

alias of `_QuEra`

**Rigetti**

alias of `_Rigetti`

**braket.devices.local\_simulator module**

**class** `braket.devices.local_simulator.LocalSimulator`(*backend: str | BraketSimulator = 'default',  
noise\_model: NoiseModel | None = None*)

Bases: `Device`

A simulator meant to run directly on the user's machine.

This class wraps a `BraketSimulator` object so that it can be run and returns results using constructs from the SDK rather than Braket IR.

Initializes a `LocalSimulator`.

**Parameters**

- **backend** (*Union*[*str*, *BraketSimulator*]) – The name of the simulator backend or the actual simulator instance to use for simulation. Defaults to the `default` simulator backend name.
- **noise\_model** (*Optional*[*NoiseModel*]) – The Braket noise model to apply to the circuit before execution. Noise model can only be added to the devices that support noise simulation.

**run**(*task\_specification*: *Circuit* | *Problem* | *Program* | *AnalogHamiltonianSimulation*, *shots*: *int* = 0, *inputs*: *dict*[*str*, *float*] | *None* = *None*, *\*args*: *Any*, *\*\*kwargs*: *Any*) → *LocalQuantumTask*

Runs the given task with the wrapped local simulator.

#### Parameters

- **task\_specification** (*Union*[*Circuit*, *Problem*, *Program*, *AnalogHamiltonianSimulation*]) – The quantum task specification.
- **shots** (*int*) – The number of times to run the circuit or annealing problem. Default is 0, which means that the simulator will compute the exact results based on the quantum task specification. Sampling is not supported for shots=0.
- **inputs** (*Optional*[*dict*[*str*, *float*]]) – Inputs to be passed along with the IR. If the IR supports inputs, the inputs will be updated with this value. Default: {}.
- **\*args** (*Any*) – Arbitrary arguments.
- **\*\*kwargs** (*Any*) – Arbitrary keyword arguments.

#### Returns

*LocalQuantumTask* – A *LocalQuantumTask* object containing the results of the simulation

---

**Note:** If running a circuit, the number of qubits will be passed to the backend as the argument after the circuit itself.

---

#### Examples

```
>>> circuit = Circuit().h(0).cnot(0, 1)
>>> device = LocalSimulator("default")
>>> device.run(circuit, shots=1000)
```

**run\_batch**(*task\_specifications*: *Circuit* | *Problem* | *Program* | *AnalogHamiltonianSimulation* | *list*[*Circuit* | *Problem* | *Program* | *AnalogHamiltonianSimulation*], *shots*: *int* | *None* = 0, *max\_parallel*: *int* | *None* = *None*, *inputs*: *dict*[*str*, *float*] | *list*[*dict*[*str*, *float*]] | *None* = *None*, *\*args*, *\*\*kwargs*) → *LocalQuantumTaskBatch*

Executes a batch of quantum tasks in parallel

#### Parameters

- **task\_specifications** (*Union*[*Union*[*Circuit*, *Problem*, *Program*, *AnalogHamiltonianSimulation*], *list*[*Union*[*Circuit*, *Problem*, *Program*, *AnalogHamiltonianSimulation*]]]) – Single instance or list of quantum task specification.
- **shots** (*Optional*[*int*]) – The number of times to run the quantum task. Default: 0.
- **max\_parallel** (*Optional*[*int*]) – The maximum number of quantum tasks to run in parallel. Default is the number of CPU.

- **inputs** (*Optional[Union[dict[str, float], list[dict[str, float]]]]*) – Inputs to be passed along with the IR. If the IR supports inputs, the inputs will be updated with this value. Default: {}.

**Returns**

*LocalQuantumTaskBatch* – A batch containing all of the quantum tasks run

**See also:**

*braket.tasks.local\_quantum\_task\_batch.LocalQuantumTaskBatch*

**property properties: DeviceCapabilities**

Return the device properties

Please see `braket.device_schema` in [amazon-braket-schemas-python](#)

**Type**

*DeviceCapabilities*

**static registered\_backends() → set[str]**

Gets the backends that have been registered as entry points

**Returns**

*set[str]* – The names of the available backends that can be passed into `LocalSimulator`’s constructor

**braket.error\_mitigation package****Submodules****braket.error\_mitigation.debias module****class braket.error\_mitigation.debias.Debias**

Bases: *ErrorMitigation*

The debias error mitigation scheme. This scheme takes no parameters.

**serialize() → list[Debias]**

This returns a list of service-readable error mitigation scheme descriptions.

**Returns**

*list[ErrorMitigationScheme]* – A list of service-readable error mitigation scheme descriptions.

**Raises**

**NotImplementedError** – Not implemented in the base class.

## braket.error\_mitigation.error\_mitigation module

**class** `braket.error_mitigation.error_mitigation.ErrorMitigation`

Bases: `object`

**serialize()** → `list[ErrorMitigationScheme]`

This returns a list of service-readable error mitigation scheme descriptions.

**Returns**

*list[ErrorMitigationScheme]* – A list of service-readable error mitigation scheme descriptions.

**Raises**

**NotImplementedError** – Not implemented in the base class.

## braket.jobs package

### Subpackages

### braket.jobs.local package

### Submodules

## braket.jobs.local.local\_job module

**class** `braket.jobs.local.local_job.LocalQuantumJob(arn: str, run_log: str | None = None)`

Bases: `QuantumJob`

Amazon Braket implementation of a hybrid job that runs locally.

Initializes a `LocalQuantumJob`.

**Parameters**

- **arn** (`str`) – The ARN of the hybrid job.
- **run\_log** (`str | None`) – The container output log of running the hybrid job with the given arn.

**Raises**

**ValueError** – Local job is not found.

**classmethod** **create**(*device: str, source\_module: str, entry\_point: str | None = None, image\_uri: str | None = None, job\_name: str | None = None, code\_location: str | None = None, role\_arn: str | None = None, hyperparameters: dict[str, Any] | None = None, input\_data: str | dict | S3DataSourceConfig | None = None, output\_data\_config: OutputDataConfig | None = None, checkpoint\_config: CheckpointConfig | None = None, aws\_session: AwsSession | None = None, local\_container\_update: bool = True*) → `LocalQuantumJob`

Creates and runs hybrid job by setting up and running the customer script in a local docker container.

**Parameters**

- **device** (`str`) – Device ARN of the QPU device that receives priority quantum task queueing once the hybrid job begins running. Each QPU has a separate hybrid jobs queue so that only one hybrid job is running at a time. The device string is accessible in the hybrid job

instance as the environment variable “AMZN\_BRAKET\_DEVICE\_ARN”. When using embedded simulators, you may provide the device argument as a string of the form: “local:<provider>/<simulator\_name>”.

- **source\_module** (*str*) – Path (absolute, relative or an S3 URI) to a python module to be tarred and uploaded. If `source_module` is an S3 URI, it must point to a tar.gz file. Otherwise, `source_module` may be a file or directory.
- **entry\_point** (*str* / *None*) – A *str* that specifies the entry point of the hybrid job, relative to the source module. The entry point must be in the format `importable.module` or `importable.module:callable`. For example, `source_module.submodule:start_here` indicates the `start_here` function contained in `source_module.submodule`. If `source_module` is an S3 URI, entry point must be given. Default: `source_module`’s name
- **image\_uri** (*str* / *None*) – A *str* that specifies the ECR image to use for executing the hybrid job. `image_uris.retrieve_image()` function may be used for retrieving the ECR image URIs for the containers supported by Braket. Default = `<Braket base image_uri>`.
- **job\_name** (*str* / *None*) – A *str* that specifies the name with which the hybrid job is created. Default: `f'{image_uri_type}-{timestamp}'`.
- **code\_location** (*str* / *None*) – The S3 prefix URI where custom code will be uploaded. Default: `f's3://{default_bucket_name}/jobs/{job_name}/script'`.
- **role\_arn** (*str* / *None*) – This field is currently not used for local hybrid jobs. Local hybrid jobs will use the current role’s credentials. This may be subject to change.
- **hyperparameters** (*dict[str, Any]* / *None*) – Hyperparameters accessible to the hybrid job. The hyperparameters are made accessible as a `Dict[str, str]` to the hybrid job. For convenience, this accepts other types for keys and values, but `str()` is called to convert them before being passed on. Default: `None`.
- **input\_data** (*str* / *dict* / `S3DataSourceConfig` / *None*) – Information about the training data. Dictionary maps channel names to local paths or S3 URIs. Contents found at any local paths will be uploaded to S3 at `f's3://{default_bucket_name}/jobs/{job_name}/data/{channel_name}`. If a local path, S3 URI, or `S3DataSourceConfig` is provided, it will be given a default channel name “input”. Default: `{}`.
- **output\_data\_config** (`OutputDataConfig` / *None*) – Specifies the location for the output of the hybrid job. Default: `OutputDataConfig(s3Path=f's3://{default_bucket_name}/jobs/{job_name}/data', kmsKeyId=None)`.
- **checkpoint\_config** (`CheckpointConfig` / *None*) – Configuration that specifies the location where checkpoint data is stored. Default: `CheckpointConfig(localPath='/opt/jobs/checkpoints', s3Uri=f's3://{default_bucket_name}/jobs/{job_name}/checkpoints')`.
- **aws\_session** (`AwsSession` / *None*) – `AwsSession` for connecting to AWS Services. Default: `AwsSession()`
- **local\_container\_update** (*bool*) – Perform an update, if available, from ECR to the local container image. Optional. Default: `True`.

#### Raises

**ValueError** – Local directory with the job name already exists.

#### Returns

*LocalQuantumJob* – The representation of a local Braket Hybrid Job.

**property arn:** `str`

The ARN (Amazon Resource Name) of the hybrid job.

**Type**

`str`

**property name:** `str`

The name of the hybrid job.

**Type**

`str`

**property run\_log:** `str`

Gets the run output log from running the hybrid job.

**Raises**

**ValueError** – The log file is not found.

**Returns**

`str` – The container output log from running the hybrid job.

**state**(*use\_cached\_value: bool = False*) → `str`

The state of the hybrid job.

**Parameters**

**use\_cached\_value** (*bool*) – If True, uses the value most recently retrieved value from the Amazon Braket GetJob operation. If False, calls the GetJob operation to retrieve metadata, which also updates the cached value. Default = False.

**Returns**

`str` – Returns “COMPLETED”.

**metadata**(*use\_cached\_value: bool = False*) → `dict[str, Any]`

When running the hybrid job in local mode, the metadata is not available.

**Parameters**

**use\_cached\_value** (*bool*) – If True, uses the value most recently retrieved from the Amazon Braket GetJob operation, if it exists; if does not exist, GetJob is called to retrieve the metadata. If False, always calls GetJob, which also updates the cached value. Default: False.

**Returns**

`dict[str, Any]` – None

**cancel**() → `str`

When running the hybrid job in local mode, the cancelling a running is not possible.

**Returns**

`str` – None

**download\_result**(*extract\_to: str | None = None, poll\_timeout\_seconds: float = 864000, poll\_interval\_seconds: float = 5*) → `None`

When running the hybrid job in local mode, results are automatically stored locally.

**Parameters**

- **extract\_to** (*str | None*) – The directory to which the results are extracted. The results are extracted to a folder titled with the hybrid job name within this directory. Default= Current working directory.



- **poll\_timeout\_seconds** (*float*) – The polling timeout, in seconds, for `result()`. Default: 10 days.
- **poll\_interval\_seconds** (*float*) – The polling interval, in seconds, for `result()`. Default: 5 seconds.

**result**(*poll\_timeout\_seconds: float = 864000, poll\_interval\_seconds: float = 5*) → dict[str, Any]

Retrieves the `LocalQuantumJob` result persisted using `save_job_result` function.

#### Parameters

- **poll\_timeout\_seconds** (*float*) – The polling timeout, in seconds, for `result()`. Default: 10 days.
- **poll\_interval\_seconds** (*float*) – The polling interval, in seconds, for `result()`. Default: 5 seconds.

#### Raises

**ValueError** – The local job directory does not exist.

#### Returns

*dict[str, Any]* – Dict specifying the hybrid job results.

**metrics**(*metric\_type: MetricType = MetricType.TIMESTAMP, statistic: MetricStatistic = MetricStatistic.MAX*) → dict[str, list[Any]]

Gets all the metrics data, where the keys are the column names, and the values are a list containing the values in each row.

#### Parameters

- **metric\_type** (*MetricType*) – The type of metrics to get. Default: `MetricType.TIMESTAMP`.
- **statistic** (*MetricStatistic*) – The statistic to determine which metric value to use when there is a conflict. Default: `MetricStatistic.MAX`.

### Example

**timestamp energy**

0 0.1 1 0.2

would be represented as: { “timestamp” : [0, 1], “energy” : [0.1, 0.2] } values may be integers, floats, strings or None.

#### Returns

*dict[str, list[Any]]* – The metrics data.

**logs**(*wait: bool = False, poll\_interval\_seconds: int = 5*) → None

Display container logs for a given hybrid job

#### Parameters

- **wait** (*bool*) – True to keep looking for new log entries until the hybrid job completes; otherwise False. Default: False.
- **poll\_interval\_seconds** (*int*) – The interval of time, in seconds, between polling for new log entries and hybrid job completion (default: 5).

## braket.jobs.local.local\_job\_container module

### braket.jobs.local.local\_job\_container\_setup module

```
braket.jobs.local.local_job_container_setup.setup_container(container: _LocalJobContainer,
                                                            aws_session: AwsSession,
                                                            **creation_kwargs: str) → dict[str,
                                                            str]
```

Sets up a container with prerequisites for running a Braket Hybrid Job. The prerequisites are based on the options the customer has chosen for the hybrid job. Similarly, any environment variables that are needed during runtime will be returned by this function.

#### Parameters

- **container** (*\_LocalJobContainer*) – The container that will run the braket hybrid job.
- **aws\_session** (*AwsSession*) – AwsSession for connecting to AWS Services.
- **\*\*creation\_kwargs** (*str*) – Arbitrary keyword arguments.

#### Returns

*dict[str, str]* – A dictionary of environment variables that reflect Braket Hybrid Jobs options requested by the customer.

## braket.jobs.metrics\_data package

### Submodules

#### braket.jobs.metrics\_data.cwl\_insights\_metrics\_fetcher module

```
class braket.jobs.metrics_data.cwl_insights_metrics_fetcher.CwlInsightsMetricsFetcher(aws_session:
~braket.aws.aws_session.AwsSession,
poll_timeout_seconds:
float
=
10,
poll_interval_seconds:
float
= 1,
logger:
~logging.Logger
=
<Logger
braket.jobs.metrics_data.cwl_insights_metrics_fetcher.CwlInsightsMetricsFetcher
(WARNING)>)
```

Bases: object

Initializes a *CwlInsightsMetricsFetcher*.

#### Parameters

- **aws\_session** (*AwsSession*) – *AwsSession* to connect to AWS with.
- **poll\_timeout\_seconds** (*float*) – The polling timeout for retrieving the metrics, in seconds. Default: 10 seconds.
- **poll\_interval\_seconds** (*float*) – The interval of time, in seconds, between polling for results. Default: 1 second.
- **logger** (*Logger*) – *Logger* object with which to write logs, such as quantum task statuses while waiting for a quantum task to be in a terminal state. Default is `getLogger(__name__)`

`LOG_GROUP_NAME = '/aws/braket/jobs'`

`QUERY_DEFAULT_JOB_DURATION = 10800`

`get_metrics_for_job(job_name: str, metric_type: MetricType = MetricType.TIMESTAMP, statistic: MetricStatistic = MetricStatistic.MAX, job_start_time: int | None = None, job_end_time: int | None = None, stream_prefix: str | None = None) → dict[str, list[str | float | int]]`

Synchronously retrieves all the algorithm metrics logged by a given Hybrid Job.

#### Parameters

- **job\_name** (*str*) – The name of the Hybrid Job. The name must be exact to ensure only the relevant metrics are retrieved.
- **metric\_type** (*MetricType*) – The type of metrics to get. Default is *MetricType.TIMESTAMP*.
- **statistic** (*MetricStatistic*) – The statistic to determine which metric value to use when there is a conflict. Default is *MetricStatistic.MAX*.
- **job\_start\_time** (*int* / *None*) – The time when the hybrid job started. Default: 3 hours before `job_end_time`.
- **job\_end\_time** (*int* / *None*) – If the hybrid job is complete, this should be the time at which the hybrid job finished. Default: current time.
- **stream\_prefix** (*str* / *None*) – If a logs prefix is provided, it will be used instead of the job name.

#### Returns

`dict[str, list[Union[str, float, int]]]` – The metrics data, where the keys are the column names and the values are a list containing the values in each row.

#### Example

timestamp energy 0 0.1 1 0.2 would be represented as: { “timestamp” : [0, 1], “energy” : [0.1, 0.2] } The values may be integers, floats, strings or *None*.

**braket.jobs.metrics\_data.cwl\_metrics\_fetcher module**

```
class braket.jobs.metrics_data.cwl_metrics_fetcher.CwlMetricsFetcher(aws_session:
    ~braket.aws.aws_session.AwsSession,
    poll_timeout_seconds:
        float = 10, logger:
        ~logging.Logger =
        <Logger
        braket.jobs.metrics_data.cwl_metrics_fetcher
        (WARNING)>)
```

Bases: object

Initializes a `CwlMetricsFetcher`.

**Parameters**

- **aws\_session** (`AwsSession`) – `AwsSession` to connect to AWS with.
- **poll\_timeout\_seconds** (`float`) – The polling timeout for retrieving the metrics, in seconds. Default: 10 seconds.
- **logger** (`Logger`) – Logger object with which to write logs, such as quantum task statuses while waiting for quantum task to be in a terminal state. Default is `getLogger(__name__)`

`LOG_GROUP_NAME = '/aws/braket/jobs'`

**get\_metrics\_for\_job**(*job\_name: str, metric\_type: `MetricType` = `MetricType.TIMESTAMP`, statistic: `MetricStatistic` = `MetricStatistic.MAX`*) → dict[str, list[str | float | int]]

Synchronously retrieves all the algorithm metrics logged by a given Hybrid Job.

**Parameters**

- **job\_name** (`str`) – The name of the Hybrid Job. The name must be exact to ensure only the relevant metrics are retrieved.
- **metric\_type** (`MetricType`) – The type of metrics to get. Default is `MetricType.TIMESTAMP`.
- **statistic** (`MetricStatistic`) – The statistic to determine which metric value to use when there is a conflict. Default is `MetricStatistic.MAX`.

**Returns**

`dict[str, list[Union[str, float, int]]]` – The metrics data, where the keys are the column names and the values are a list containing the values in each row.

**Example**

timestamp energy 0 0.1 1 0.2 would be represented as: { “timestamp”: [0, 1], “energy”: [0.1, 0.2] } values may be integers, floats, strings or None.

**braket.jobs.metrics\_data.definitions module**

```
class braket.jobs.metrics_data.definitions.MetricPeriod(value)
```

Bases: Enum

Period over which the cloudwatch metric is aggregated.

```
ONE_MINUTE: int = 60
```

```
class braket.jobs.metrics_data.definitions.MetricStatistic(value)
```

Bases: Enum

Metric data aggregation to use over the specified period.

```
MIN: str = 'Min'
```

```
MAX: str = 'Max'
```

```
class braket.jobs.metrics_data.definitions.MetricType(value)
```

Bases: Enum

Metric type.

```
TIMESTAMP: str = 'Timestamp'
```

```
ITERATION_NUMBER: str = 'IterationNumber'
```

**braket.jobs.metrics\_data.exceptions module**

```
exception braket.jobs.metrics_data.exceptions.MetricsRetrievalError
```

Bases: Exception

Raised when retrieving metrics fails.

**braket.jobs.metrics\_data.log\_metrics\_parser module**

```
class braket.jobs.metrics_data.log_metrics_parser.LogMetricsParser(logger: ~logging.Logger =
    <Logger
    braket.jobs.metrics_data.log_metrics_parser
    (WARNING)>)
```

Bases: object

This class is used to parse metrics from log lines, and return them in a more convenient format.

```
METRICS_DEFINITIONS = re.compile('(\w+)\s*=\s*([^\s;]+)\s*;',')
```

```
TIMESTAMP = 'timestamp'
```

```
ITERATION_NUMBER = 'iteration_number'
```

```
NODE_ID = 'node_id'
```

```
NODE_TAG = re.compile('^\s*([^\s\]]*)\s*\s*')
```

**parse\_log\_message**(*timestamp: str, message: str*) → None

Parses a line from logs, adding all the metrics that have been logged on that line. The timestamp is also added to match the corresponding values.

**Parameters**

- **timestamp** (*str*) – A formatted string representing the timestamp for any found metrics.
- **message** (*str*) – A log line from a log.

**get\_columns\_and\_pivot\_indices**(*pivot: str*) → tuple[dict[str, list[str | float | int]], dict[tuple[int, str], int]]

Parses the metrics to find all the metrics that have the pivot column. The values of the pivot column are paired with the node\_id and assigned a row index, so that all metrics with the same pivot value and node\_id are stored in the same row.

**Parameters**

**pivot** (*str*) – The name of the pivot column. Must be `TIMESTAMP` or `ITERATION_NUMBER`.

**Returns**

*tuple[dict[str, list[Union[str, float, int]]], dict[tuple[int, str], int]]* – Contains: The dict[str, list[Any]] is the result table with all the metrics values initialized to None. The dict[tuple[int, str], int] is the list of pivot indices, where the value of a pivot column and node\_id is mapped to a row index.

**get\_metric\_data\_with\_pivot**(*pivot: str, statistic: MetricStatistic*) → dict[str, list[str | float | int]]

Gets the metric data for a given pivot column name. Metrics without the pivot column are not included in the results. Metrics that have the same value in the pivot column from the same node are returned in the same row. Metrics from different nodes are stored in different rows. If the metric has multiple values for the row, the statistic is used to determine which value is returned. For example, for the metrics: “iteration\_number” : 0, “metricA” : 2, “metricB” : 1, “iteration\_number” : 0, “metricA” : 1, “no\_pivot\_column” : 0, “metricA” : 0, “iteration\_number” : 1, “metricA” : 2, “iteration\_number” : 1, “node\_id” : “nodeB”, “metricB” : 0,

**The result with iteration\_number as the pivot, statistic of MIN the result will be:**

iteration\_number node\_id metricA metricB 0 None 1 1 1 None 2 None 1 nodeB None 0

**Parameters**

- **pivot** (*str*) – The name of the pivot column. Must be `TIMESTAMP` or `ITERATION_NUMBER`.
- **statistic** (*MetricStatistic*) – The statistic to determine which value to use.

**Returns**

*dict[str, list[Union[str, float, int]]]* – The metrics data.

**get\_parsed\_metrics**(*metric\_type: MetricType, statistic: MetricStatistic*) → dict[str, list[str | float | int]]

Gets all the metrics data, where the keys are the column names and the values are a list containing the values in each row.

**Parameters**

- **metric\_type** (*MetricType*) – The type of metrics to get.
- **statistic** (*MetricStatistic*) – The statistic to determine which metric value to use when there is a conflict.

**Returns**

*dict[str, list[Union[str, float, int]]]* – The metrics data.

## Example

```
timestamp energy
0 0.1 1 0.2
```

would be represented as: { “timestamp” : [0, 1], “energy” : [0.1, 0.2] } values may be integers, floats, strings or None.

## Submodules

### braket.jobs.config module

```
class braket.jobs.config.CheckpointConfig(localPath: str = '/opt/jobs/checkpoints', s3Uri: str | None = None)
```

Bases: object

Configuration that specifies the location where checkpoint data is stored.

**localPath:** str = '/opt/jobs/checkpoints'

**s3Uri:** str | None = None

```
class braket.jobs.config.InstanceConfig(instanceType: str = 'ml.m5.large', volumeSizeInGb: int = 30, instanceCount: int = 1)
```

Bases: object

Configuration of the instance(s) used to run the hybrid job.

**instanceType:** str = 'ml.m5.large'

**volumeSizeInGb:** int = 30

**instanceCount:** int = 1

```
class braket.jobs.config.OutputDataConfig(s3Path: str | None = None, kmsKeyId: str | None = None)
```

Bases: object

Configuration that specifies the location for the output of the hybrid job.

**s3Path:** str | None = None

**kmsKeyId:** str | None = None

```
class braket.jobs.config.StoppingCondition(maxRuntimeInSeconds: int = 432000)
```

Bases: object

Conditions that specify when the hybrid job should be forcefully stopped.

**maxRuntimeInSeconds:** int = 432000

```
class braket.jobs.config.DeviceConfig(device: 'str')
```

Bases: object

**device:** str

```
class braket.jobs.config.S3DataSourceConfig(s3_data: str, content_type: str | None = None)
```

Bases: object

Data source for data that lives on S3.

#### config

config passed to the Braket API

#### Type

dict[str, dict]

Create a definition for input data used by a Braket Hybrid job.

#### Parameters

- **s3\_data** (*str*) – Defines the location of s3 data to train on.
- **content\_type** (*str* | *None*) – MIME type of the input data (default: *None*).

### braket.jobs.data\_persistence module

```
braket.jobs.data_persistence.save_job_checkpoint(checkpoint_data: dict[str, Any],
                                                  checkpoint_file_suffix: str = "", data_format:
                                                  PersistedJobDataFormat =
                                                  PersistedJobDataFormat.PLAINTEXT) → None
```

Saves the specified `checkpoint_data` to the local output directory, specified by the container environment variable `CHECKPOINT_DIR`, with the filename `f"{job_name}_{checkpoint_file_suffix}.json"`. The `job_name` refers to the name of the current job and is retrieved from the container environment variable `JOB_NAME`. The `checkpoint_data` values are serialized to the specified `data_format`.

**Note:** This function for storing the checkpoints is only for use inside the job container

as it writes data to directories and references env variables set in the containers.

#### Parameters

- **checkpoint\_data** (*dict[str, Any]*) – Dict that specifies the checkpoint data to be persisted.
- **checkpoint\_file\_suffix** (*str*) – str that specifies the file suffix to be used for the checkpoint filename. The resulting filename `f"{job_name}_{checkpoint_file_suffix}.json"` is used to save the checkpoints. Default: ""
- **data\_format** (*PersistedJobDataFormat*) – The data format used to serialize the values. Note that for `PICKLED` data formats, the values are base64 encoded after serialization. Default: `PersistedJobDataFormat.PLAINTEXT`

#### Raises

**ValueError** – If the supplied `checkpoint_data` is *None* or empty.

```
braket.jobs.data_persistence.load_job_checkpoint(job_name: str | None = None, checkpoint_file_suffix:
                                                  str = "") → dict[str, Any]
```

Loads the job checkpoint data stored for the job named '`job_name`', with the checkpoint file that ends with the `checkpoint_file_suffix`. The `job_name` can refer to any job whose checkpoint data you expect to be available in the file path specified by the `CHECKPOINT_DIR` container environment variable. If not provided, this function will use the currently running job's name.

**Note:** This function for loading hybrid job checkpoints is only for use inside the job container

as it writes data to directories and references env variables set in the containers.



**Parameters**

- **job\_name** (*str* / *None*) – str that specifies the name of the job whose checkpoints are to be loaded. Default: current job name.
- **checkpoint\_file\_suffix** (*str*) – str specifying the file suffix that is used to locate the checkpoint file to load. The resulting file name `f"{job_name}(_{checkpoint_file_suffix}).json"` is used to locate the checkpoint file. Default: ""

**Returns**

*dict[str, Any]* – Dict that contains the checkpoint data persisted in the checkpoint file.

**Raises**

- **FileNotFoundError** – If the file `f"{job_name}(_{checkpoint_file_suffix})"` could not be found in the directory specified by the container environment variable `CHECKPOINT_DIR`.
- **ValueError** – If the data stored in the checkpoint file can't be deserialized (possibly due to corruption).

`braket.jobs.data_persistence.load_job_result(filename: str | Path | None = None) → dict[str, Any]`

Loads job result of currently running job.

**Parameters**

**filename** (*str* / *Path* / *None*) – Location of job results. Default `results.json` in job results directory in a job instance or in working directory locally. This file must be in the format used by [save\\_job\\_result](#).

**Returns**

*dict[str, Any]* – Job result data of current job

`braket.jobs.data_persistence.save_job_result(result_data: dict[str, Any] | Any, data_format: PersistedJobDataFormat | None = None) → None`

Saves the `result_data` to the local output directory that is specified by the container environment variable `AMZN_BRAKET_JOB_RESULTS_DIR`, with the filename 'results.json'. The `result_data` values are serialized to the specified `data_format`.

**Note:** This function for storing the results is only for use inside the job container

as it writes data to directories and references env variables set in the containers.

**Parameters**

- **result\_data** (*dict[str, Any]* / *Any*) – Dict that specifies the result data to be persisted. If result data is not a dict, then it will be wrapped as `{"result": result_data}`.
- **data\_format** (*PersistedJobDataFormat* / *None*) – The data format used to serialize the values. Note that for `PICKLED` data formats, the values are base64 encoded after serialization. Default: `PersistedJobDataFormat.PLAINTEXT`.

**Raises**

**TypeError** – Unsupported data format.

**braket.jobs.environment\_variables module**

`braket.jobs.environment_variables.get_job_name()` → str

Get the name of the current job.

**Returns**

*str* – The name of the job if in a job, else an empty string.

`braket.jobs.environment_variables.get_job_device_arn()` → str

Get the device ARN of the current job. If not in a job, default to “local:none/none”.

**Returns**

*str* – The device ARN of the current job or “local:none/none”.

`braket.jobs.environment_variables.get_input_data_dir(channel: str = 'input')` → str

Get the job input data directory.

**Parameters**

**channel** (*str*) – The name of the input channel. Default value corresponds to the default input channel name, `input`.

**Returns**

*str* – The input directory, defaulting to current working directory.

`braket.jobs.environment_variables.get_results_dir()` → str

Get the job result directory.

**Returns**

*str* – The results directory, defaulting to current working directory.

`braket.jobs.environment_variables.get_checkpoint_dir()` → str

Get the job checkpoint directory.

**Returns**

*str* – The checkpoint directory, defaulting to current working directory.

`braket.jobs.environment_variables.get_hyperparameters()` → dict[str, str]

Get the job hyperparameters as a dict, with the values stringified.

**Returns**

*dict[str, str]* – The hyperparameters of the job.

**braket.jobs.hybrid\_job module**

`braket.jobs.hybrid_job.hybrid_job(*, device: str | None, include_modules: str | ModuleType | Iterable[str | ModuleType] | None = None, dependencies: str | Path | list[str] | None = None, local: bool = False, job_name: str | None = None, image_uri: str | None = None, input_data: str | dict | S3DataSourceConfig | None = None, wait_until_complete: bool = False, instance_config: InstanceConfig | None = None, distribution: str | None = None, copy_checkpoints_from_job: str | None = None, checkpoint_config: CheckpointConfig | None = None, role_arn: str | None = None, stopping_condition: StoppingCondition | None = None, output_data_config: OutputDataConfig | None = None, aws_session: AwsSession | None = None, tags: dict[str, str] | None = None, logger: Logger = <Logger braket.jobs.hybrid_job (WARNING)>, quiet: bool | None = None, reservation_arn: str | None = None) → Callable`

Defines a hybrid job by decorating the entry point function. The job will be created when the decorated function is called.

The job created will be a `LocalQuantumJob` when `local` is set to `True`, otherwise an `AwsQuantumJob`. The following parameters will be ignored when running a job with `local` set to `True`: `wait_until_complete`, `instance_config`, `distribution`, `copy_checkpoints_from_job`, `stopping_condition`, `tags`, `logger`, and `quiet`.

### Parameters

- **device** (*str* / *None*) – Device ARN of the QPU device that receives priority quantum task queueing once the hybrid job begins running. Each QPU has a separate hybrid jobs queue so that only one hybrid job is running at a time. The device string is accessible in the hybrid job instance as the environment variable “AMZN\_BRAKET\_DEVICE\_ARN”. When using embedded simulators, you may provide the device argument as string of the form: “local:<provider>/<simulator\_name>” or *None*.
- **include\_modules** (*str* / *ModuleType* / *Iterable[str / ModuleType]* / *None*) – Either a single module or module name or a list of module or module names referring to local modules to be included. Any references to members of these modules in the hybrid job algorithm code will be serialized as part of the algorithm code. Default: []
- **dependencies** (*str* / *Path* / *list[str]* / *None*) – Path (absolute or relative) to a `requirements.txt` file, or alternatively a list of strings, with each string being a [requirement specifier](#), to be used for the hybrid job.
- **local** (*bool*) – Whether to use local mode for the hybrid job. Default: `False`
- **job\_name** (*str* / *None*) – A string that specifies the name with which the job is created. Allowed pattern for job name: `^[a-zA-Z0-9](-*[a-zA-Z0-9]){0,50}$`. Defaults to `f'{decorated-function-name}-{timestamp}'`.
- **image\_uri** (*str* / *None*) – A str that specifies the ECR image to use for executing the job. `retrieve_image()` function may be used for retrieving the ECR image URIs for the containers supported by Braket. Default: `<Braket base image_uri>`.
- **input\_data** (*str* / *dict* / *S3DataSourceConfig* / *None*) – Information about the training data. Dictionary maps channel names to local paths or S3 URIs. Contents found at any local paths will be uploaded to S3 at `f's3://{default_bucket_name}/jobs/{job_name}/data/{channel_name}'`. If a local path, S3 URI, or `S3DataSourceConfig` is provided, it will be given a default channel name “input”. Default: {}.
- **wait\_until\_complete** (*bool*) – True if we should wait until the job completes. This would tail the job logs as it waits. Otherwise `False`. Ignored if using local mode. Default: `False`.
- **instance\_config** (*InstanceConfig* / *None*) – Configuration of the instance(s) for running the classical code for the hybrid job. Default: `InstanceConfig(instanceType='ml.m5.large', instanceCount=1, volumeSizeInGB=30)`.
- **distribution** (*str* / *None*) – A str that specifies how the job should be distributed. If set to “data\_parallel”, the hyperparameters for the job will be set to use data parallelism features for PyTorch or TensorFlow. Default: `None`.
- **copy\_checkpoints\_from\_job** (*str* / *None*) – A str that specifies the job ARN whose checkpoint you want to use in the current job. Specifying this value will copy over the checkpoint data from `use_checkpoints_from_job's` `checkpoint_config` s3Uri to the current

job's `checkpoint_config` `s3Uri`, making it available at `checkpoint_config.localPath` during the job execution. Default: `None`

- **checkpoint\_config** (`CheckpointConfig` / `None`) – Configuration that specifies the location where checkpoint data is stored. Default: `CheckpointConfig(localPath='/opt/jobs/checkpoints', s3Uri=f's3://{default_bucket_name}/jobs/{job_name}/checkpoints')`.
- **role\_arn** (`str` / `None`) – A `str` providing the IAM role ARN used to execute the script. Default: IAM role returned by `AwsSession's get_default_jobs_role()`.
- **stopping\_condition** (`StoppingCondition` / `None`) – The maximum length of time, in seconds, and the maximum number of tasks that a job can run before being forcefully stopped. Default: `StoppingCondition(maxRuntimeInSeconds=5 * 24 * 60 * 60)`.
- **output\_data\_config** (`OutputDataConfig` / `None`) – Specifies the location for the output of the job. Default: `OutputDataConfig(s3Path=f's3://{default_bucket_name}/jobs/{job_name}/data', kmsKeyId=None)`.
- **aws\_session** (`AwsSession` / `None`) – `AwsSession` for connecting to AWS Services. Default: `AwsSession()`
- **tags** (`dict[str, str]` / `None`) – Dict specifying the key-value pairs for tagging this job. Default: `{}`.
- **logger** (`Logger`) – Logger object with which to write logs, such as task statuses while waiting for task to be in a terminal state. Default: `getLogger(__name__)`
- **quiet** (`bool` / `None`) – Sets the verbosity of the logger to low and does not report queue position. Default is `False`.
- **reservation\_arn** (`str` / `None`) – the reservation window arn provided by Braket Direct to reserve exclusive usage for the device to run the hybrid job on. Default: `None`.

#### Returns

*Callable* – the callable for creating a Hybrid Job.

### braket.jobs.image\_uris module

```
class braket.jobs.image_uris.Framework(value)
```

Bases: `str`, `Enum`

Supported Frameworks for pre-built containers

**BASE** = 'BASE'

**PL\_TENSORFLOW** = 'PL\_TENSORFLOW'

**PL\_PYTORCH** = 'PL\_PYTORCH'

```
braket.jobs.image_uris.built_in_images(region: str) → set[str]
```

Checks a region for built in Braket images.

#### Parameters

**region** (`str`) – The AWS region to check for images

#### Returns

*set[str]* – returns a set of built images

`braket.jobs.image_uris.retrieve_image(framework: Framework, region: str) → str`

Retrieves the ECR URI for the Docker image matching the specified arguments.

#### Parameters

- **framework** (`Framework`) – The name of the framework.
- **region** (`str`) – The AWS region for the Docker image.

#### Returns

`str` – The ECR URI for the corresponding Amazon Braket Docker image.

#### Raises

**ValueError** – If any of the supplied values are invalid or the combination of inputs specified is not supported.

### braket.jobs.logs module

`class braket.jobs.logs.ColorWrap(force: bool = False)`

Bases: `object`

A callable that prints text in a different color depending on the instance. Up to 5 if the standard output is a terminal or a Jupyter notebook cell.

Initialize a `ColorWrap`.

#### Parameters

**force** (`bool`) – If True, the render output is colorized wherever the output is. Default: False.

`class braket.jobs.logs.Position(timestamp, skip)`

Bases: `tuple`

Create new instance of `Position(timestamp, skip)`

#### skip

Alias for field number 1

#### timestamp

Alias for field number 0

`braket.jobs.logs.multi_stream_iter(aws_session: AwsSession, log_group: str, streams: list[str], positions: dict[str, Position]) → Generator[tuple[int, dict]]`

Iterates over the available events coming from a set of log streams. Log streams are in a single log group interleaving the events from each stream, so they yield in timestamp order.

#### Parameters

- **aws\_session** (`AwsSession`) – The `AwsSession` for interfacing with CloudWatch.
- **log\_group** (`str`) – The name of the log group.
- **streams** (`list[str]`) – A list of the log stream names. The stream number is the position of the stream in this list.
- **positions** (`dict[str, Position]`) – A list of (timestamp, skip) pairs which represent the last record read from each stream.

#### Yields

`Generator[tuple[int, dict]]` – A tuple of (stream number, cloudwatch log event).

```
braket.jobs.logs.log_stream(aws_session: AwsSession, log_group: str, stream_name: str, start_time: int = 0, skip: int = 0) → Generator[dict]
```

A generator for log items in a single stream. This yields all the items that are available at the current moment.

#### Parameters

- **aws\_session** ([AwsSession](#)) – The AwsSession for interfacing with CloudWatch.
- **log\_group** (*str*) – The name of the log group.
- **stream\_name** (*str*) – The name of the specific stream.
- **start\_time** (*int*) – The time stamp value to start reading the logs from. Default: 0.
- **skip** (*int*) – The number of log entries to skip at the start. Default: 0 (This is for when there are multiple entries at the same timestamp.)

#### Yields

*Generator[dict]* – A CloudWatch log event with the following key-value pairs: ‘timestamp’ (*int*): The time of the event. ‘message’ (*str*): The log event data. ‘ingestionTime’ (*int*): The time the event was ingested.

```
braket.jobs.logs.flush_log_streams(aws_session: AwsSession, log_group: str, stream_prefix: str, stream_names: list[str], positions: dict[str, Position], stream_count: int, has_streams: bool, color_wrap: ColorWrap, state: list[str], queue_position: str | None = None) → bool
```

Flushes log streams to stdout.

#### Parameters

- **aws\_session** ([AwsSession](#)) – The AwsSession for interfacing with CloudWatch.
- **log\_group** (*str*) – The name of the log group.
- **stream\_prefix** (*str*) – The prefix for log streams to flush.
- **stream\_names** (*list[str]*) – A list of the log stream names. The position of the stream in this list is the stream number. If incomplete, the function will check for remaining streams and mutate this list to add stream names when available, up to the **stream\_count** limit.
- **positions** (*dict[str, Position]*) – A dict mapping stream numbers to (timestamp, skip) pairs which represent the last record read from each stream. The function will update this list after being called to represent the new last record read from each stream.
- **stream\_count** (*int*) – The number of streams expected.
- **has\_streams** (*bool*) – Whether the function has already been called once all streams have been found. This value is possibly updated and returned at the end of execution.
- **color\_wrap** ([ColorWrap](#)) – An instance of ColorWrap to potentially color-wrap print statements from different streams.
- **state** (*list[str]*) – The previous and current state of the job.
- **queue\_position** (*Optional[str]*) – The current queue position. This is not passed in if the job is ran with `quiet=True`

#### Raises

**Exception** – Any exception found besides a `ResourceNotFoundException`.

#### Returns

*bool* – Returns ‘True’ if any streams have been flushed.

**braket.jobs.metrics module**

**braket.jobs.metrics.log\_metric**(*metric\_name: str, value: float | int, timestamp: float | None = None, iteration\_number: int | None = None*) → None

Records Braket Hybrid Job metrics.

**Parameters**

- **metric\_name** (*str*) – The name of the metric.
- **value** (*Union[float, int]*) – The value of the metric.
- **timestamp** (*Optional[float]*) – The time the metric data was received, expressed as the number of seconds since the epoch. Default: Current system time.
- **iteration\_number** (*Optional[int]*) – The iteration number of the metric.

**braket.jobs.quantum\_job module**

**class** `braket.jobs.quantum_job.QuantumJob`

Bases: ABC

**DEFAULT\_RESULTS\_POLL\_TIMEOUT** = 864000

**DEFAULT\_RESULTS\_POLL\_INTERVAL** = 5

**abstract property** `arn: str`

The ARN (Amazon Resource Name) of the hybrid job.

**Returns**

*str* – The ARN (Amazon Resource Name) of the hybrid job.

**abstract property** `name: str`

The name of the hybrid job.

**Returns**

*str* – The name of the hybrid job.

**abstract state**(*use\_cached\_value: bool = False*) → str

The state of the hybrid job.

**Parameters**

**use\_cached\_value** (*bool*) – If True, uses the value most recently retrieved value from the Amazon Braket Get Job operation. If False, calls the Get Job operation to retrieve metadata, which also updates the cached value. Default = False.

**Returns**

*str* – The value of status in `metadata()`. This is the value of the status key in the Amazon Braket Get Job operation.

**See also:**

`metadata()`

**abstract logs**(*wait: bool = False, poll\_interval\_seconds: int = 5*) → None

Display logs for a given hybrid job, optionally tailing them until hybrid job is complete.

If the output is a tty or a Jupyter cell, it will be color-coded based on which instance the log entry is from.

**Parameters**

- **wait** (*bool*) – True to keep looking for new log entries until the hybrid job completes; otherwise False. Default: False.
- **poll\_interval\_seconds** (*int*) – The interval of time, in seconds, between polling for new log entries and hybrid job completion (default: 5).

**Raises**

**RuntimeError** – If waiting and the hybrid job fails.

**abstract metadata**(*use\_cached\_value: bool = False*) → dict[str, Any]

Gets the job metadata defined in Amazon Braket.

**Parameters**

**use\_cached\_value** (*bool*) – If True, uses the value most recently retrieved from the Amazon Braket GetJob operation, if it exists; if does not exist, GetJob is called to retrieve the metadata. If False, always calls GetJob, which also updates the cached value. Default: False.

**Returns**

dict[str, Any] – Dict that specifies the hybrid job metadata defined in Amazon Braket.

**abstract metrics**(*metric\_type: MetricType = MetricType.TIMESTAMP, statistic: MetricStatistic = MetricStatistic.MAX*) → dict[str, list[Any]]

Gets all the metrics data, where the keys are the column names, and the values are a list containing the values in each row.

**Parameters**

- **metric\_type** (*MetricType*) – The type of metrics to get. Default: MetricType.TIMESTAMP.
- **statistic** (*MetricStatistic*) – The statistic to determine which metric value to use when there is a conflict. Default: MetricStatistic.MAX.

**Returns**

dict[str, list[Any]] – The metrics data.

**Example**

**timestamp energy**  
0 0.1 1 0.2

would be represented as: { “timestamp” : [0, 1], “energy” : [0.1, 0.2] } values may be integers, floats, strings or None.

**abstract cancel**() → str

Cancels the hybrid job.

**Returns**

str – Indicates the status of the hybrid job.

**Raises**

**ClientError** – If there are errors invoking the CancelJob API.

**abstract result**(*poll\_timeout\_seconds: float = 864000, poll\_interval\_seconds: float = 5*) → dict[str, Any]

Retrieves the hybrid job result persisted using save\_job\_result() function.

**Parameters**



- **poll\_timeout\_seconds** (*float*) – The polling timeout, in seconds, for `result()`. Default: 10 days.
- **poll\_interval\_seconds** (*float*) – The polling interval, in seconds, for `result()`. Default: 5 seconds.

**Returns**

`dict[str, Any]` – Dict specifying the hybrid job results.

**Raises**

- **RuntimeError** – if hybrid job is in a FAILED or CANCELLED state.
- **TimeoutError** – if hybrid job execution exceeds the polling timeout period.

**abstract download\_result**(*extract\_to: str | None = None, poll\_timeout\_seconds: float = 864000, poll\_interval\_seconds: float = 5*) → None

Downloads the results from the hybrid job output S3 bucket and extracts the tar.gz bundle to the location specified by `extract_to`. If no location is specified, the results are extracted to the current directory.

**Parameters**

- **extract\_to** (*str | None*) – The directory to which the results are extracted. The results are extracted to a folder titled with the hybrid job name within this directory. Default= Current working directory.
- **poll\_timeout\_seconds** (*float*) – The polling timeout, in seconds, for `download_result()`. Default: 10 days.
- **poll\_interval\_seconds** (*float*) – The polling interval, in seconds, for `download_result()`. Default: 5 seconds.

**Raises**

- **RuntimeError** – if hybrid job is in a FAILED or CANCELLED state.
- **TimeoutError** – if hybrid job execution exceeds the polling timeout period.

**braket.jobs.quantum\_job\_creation module**

`braket.jobs.quantum_job_creation.prepare_quantum_job`(*device: str, source\_module: str, entry\_point: str | None = None, image\_uri: str | None = None, job\_name: str | None = None, code\_location: str | None = None, role\_arn: str | None = None, hyperparameters: dict[str, Any] | None = None, input\_data: str | dict | S3DataSourceConfig | None = None, instance\_config: InstanceConfig | None = None, distribution: str | None = None, stopping\_condition: StoppingCondition | None = None, output\_data\_config: OutputDataConfig | None = None, copy\_checkpoints\_from\_job: str | None = None, checkpoint\_config: CheckpointConfig | None = None, aws\_session: AwsSession | None = None, tags: dict[str, str] | None = None, reservation\_arn: str | None = None*) → dict

Creates a hybrid job by invoking the Braket CreateJob API.

## Parameters

- **device** (*str*) – Device ARN of the QPU device that receives priority quantum task queuing once the hybrid job begins running. Each QPU has a separate hybrid jobs queue so that only one hybrid job is running at a time. The device string is accessible in the hybrid job instance as the environment variable “AMZN\_BRAKET\_DEVICE\_ARN”. When using embedded simulators, you may provide the device argument as string of the form: “local:<provider>/<simulator\_name>”.
- **source\_module** (*str*) – Path (absolute, relative or an S3 URI) to a python module to be tarred and uploaded. If `source_module` is an S3 URI, it must point to a tar.gz file. Otherwise, `source_module` may be a file or directory.
- **entry\_point** (*str / None*) – A `str` that specifies the entry point of the hybrid job, relative to the source module. The entry point must be in the format `importable.module` or `importable.module:callable`. For example, `source_module.submodule:start_here` indicates the `start_here` function contained in `source_module.submodule`. If `source_module` is an S3 URI, entry point must be given. Default: `source_module`’s name
- **image\_uri** (*str / None*) – A `str` that specifies the ECR image to use for executing the hybrid job. `image_uris.retrieve_image()` function may be used for retrieving the ECR image URIs for the containers supported by Braket. Default = `<Braket base image_uri>`.
- **job\_name** (*str / None*) – A `str` that specifies the name with which the hybrid job is created. The hybrid job name must be between 0 and 50 characters long and cannot contain underscores. Default: `f’{image_uri_type}-{timestamp}’`.
- **code\_location** (*str / None*) – The S3 prefix URI where custom code will be uploaded. Default: `f’s3://{default_bucket_name}/jobs/{job_name}/script’`.
- **role\_arn** (*str / None*) – A `str` providing the IAM role ARN used to execute the script. Default: IAM role returned by `AwsSession`’s `get_default_jobs_role()`.
- **hyperparameters** (*dict[str, Any] / None*) – Hyperparameters accessible to the hybrid job. The hyperparameters are made accessible as a `Dict[str, str]` to the hybrid job. For convenience, this accepts other types for keys and values, but `str()` is called to convert them before being passed on. Default: `None`.
- **input\_data** (*str / dict / S3DataSourceConfig / None*) – Information about the training data. Dictionary maps channel names to local paths or S3 URIs. Contents found at any local paths will be uploaded to S3 at `f’s3://{default_bucket_name}/jobs/{job_name}/data/{channel_name}`. If a local path, S3 URI, or `S3DataSourceConfig` is provided, it will be given a default channel name “input”. Default: `{}`.
- **instance\_config** (*InstanceConfig / None*) – Configuration of the instance(s) for running the classical code for the hybrid job. Defaults to `InstanceConfig(instanceType='ml.m5.large', instanceCount=1, volumeSizeInGB=30)`.
- **distribution** (*str / None*) – A `str` that specifies how the hybrid job should be distributed. If set to “data\_parallel”, the hyperparameters for the hybrid job will be set to use data parallelism features for PyTorch or TensorFlow. Default: `None`.
- **stopping\_condition** (*StoppingCondition / None*) – The maximum length of time, in seconds, and the maximum number of quantum tasks that a hybrid job can run before being forcefully stopped. Default: `StoppingCondition(maxRuntimeInSeconds=5 * 24 * 60 * 60)`.

- **output\_data\_config** (`OutputDataConfig` / `None`) – Specifies the location for the output of the hybrid job. Default: `OutputDataConfig(s3Path=f's3://{default_bucket_name}/jobs/{job_name}/data', kmsKeyId=None)`.
- **copy\_checkpoints\_from\_job** (`str` / `None`) – A `str` that specifies the hybrid job ARN whose checkpoint you want to use in the current hybrid job. Specifying this value will copy over the checkpoint data from `use_checkpoints_from_job`'s `checkpoint_config` `s3Uri` to the current hybrid job's `checkpoint_config` `s3Uri`, making it available at `checkpoint_config.localPath` during the hybrid job execution. Default: `None`
- **checkpoint\_config** (`CheckpointConfig` / `None`) – Configuration that specifies the location where checkpoint data is stored. Default: `CheckpointConfig(localPath='/opt/jobs/checkpoints', s3Uri=f's3://{default_bucket_name}/jobs/{job_name}/checkpoints')`.
- **aws\_session** (`AwsSession` / `None`) – `AwsSession` for connecting to AWS Services. Default: `AwsSession()`
- **tags** (`dict[str, str]` / `None`) – Dict specifying the key-value pairs for tagging this hybrid job. Default: `{}`.
- **reservation\_arn** (`str` / `None`) – the reservation window arn provided by Braket Direct to reserve exclusive usage for the device to run the hybrid job on. Default: `None`.

**Returns**

*dict* – Hybrid job tracking the execution on Amazon Braket.

**Raises**

**ValueError** – Raises `ValueError` if the parameters are not valid.

**braket.jobs.serialization module**

`braket.jobs.serialization.serialize_values(data_dictionary: dict[str, Any], data_format: PersistedJobDataFormat) → dict[str, Any]`

Serializes the `data_dictionary` values to the format specified by `data_format`.

**Parameters**

- **data\_dictionary** (`dict[str, Any]`) – Dict whose values are to be serialized.
- **data\_format** (`PersistedJobDataFormat`) – The data format used to serialize the values. Note that for `PICKLED` data formats, the values are base64 encoded after serialization, so that they represent valid UTF-8 text and are compatible with `PersistedJobData.json()`.

**Returns**

*dict[str, Any]* – Dict with same keys as `data_dictionary` and values serialized to the specified `data_format`.

`braket.jobs.serialization.deserialize_values(data_dictionary: dict[str, Any], data_format: PersistedJobDataFormat) → dict[str, Any]`

Deserializes the `data_dictionary` values from the format specified by `data_format`.

**Parameters**

- **data\_dictionary** (`dict[str, Any]`) – Dict whose values are to be deserialized.
- **data\_format** (`PersistedJobDataFormat`) – The data format that the `data_dictionary` values are currently serialized with.

**Returns**

*dict[str, Any]* – Dict with same keys as *data\_dictionary* and values deserialized from the specified *data\_format* to plaintext.

**braket.parametric package****Submodules****braket.parametric.free\_parameter module**

**class** `braket.parametric.free_parameter.FreeParameter(name: str)`

Bases: *FreeParameterExpression*

Class 'FreeParameter'

Free parameters can be used in parameterized circuits. Objects that can take a parameter all inherit from :class:'Parameterizable'. The FreeParameter can be swapped in to a circuit for a numerical value later on. Circuits with FreeParameters must have all the inputs provided at execution or substituted prior to execution.

**Examples**

```
>>> alpha, beta = FreeParameter("alpha"), FreeParameter("beta")
>>> circuit = Circuit().rx(target=0, angle=alpha).ry(target=1, angle=beta)
>>> circuit = circuit(alpha=0.3)
>>> device = LocalSimulator()
>>> device.run(circuit, inputs={'beta': 0.5} shots=10)
```

Initializes a new :class:'FreeParameter' object.

**Parameters**

**name** (*str*) – Name of the :class:'FreeParameter'. Can be a unicode value.

**Examples**

```
>>> param1 = FreeParameter("theta")
>>> param1 = FreeParameter("")
```

**property name:** *str*

Name of this parameter.

**Type**

*str*

**subs**(*parameter\_values: dict[str, Number]*) → *FreeParameter* | *Number*

Substitutes a value in if the parameter exists within the mapping.

**Parameters**

**parameter\_values** (*dict[str, Number]*) – A mapping of parameter to its corresponding value.

**Returns**

*Union[FreeParameter, Number]* – The substituted value if this parameter is in *parameter\_values*, otherwise returns self

`to_dict()` → dict

`classmethod from_dict(parameter: dict)` → *FreeParameter*

## braket.parametric.free\_parameter\_expression module

`class braket.parametric.free_parameter_expression.FreeParameterExpression(expression: FreeParameterExpression | Number | Expr | str)`

Bases: object

Class 'FreeParameterExpression'

Objects that can take a parameter all inherit from :class:'Parameterizable'. FreeParametersExpressions can hold FreeParameters that can later be swapped out for a number. Circuits or PulseSequences with FreeParameters present will NOT run. Values must be substituted prior to execution.

Initializes a FreeParameterExpression. Best practice is to initialize using FreeParameters and Numbers. Not meant to be initialized directly.

Below are examples of how FreeParameterExpressions should be made.

### Parameters

**expression** (*Union*[*FreeParameterExpression*, *Number*, *Expr*, *str*]) – The expression to use.

### Raises

**NotImplementedError** – Raised if the expression is not of type [*FreeParameterExpression*, *Number*, *Expr*, *str*]

## Examples

```
>>> expression_1 = FreeParameter("theta") * FreeParameter("alpha")
>>> expression_2 = 1 + FreeParameter("beta") + 2 * FreeParameter("alpha")
```

**property expression:** *Number* | *Expr*

Gets the expression.

### Returns

*Union*[*Number*, *Expr*] – The expression for the FreeParameterExpression.

**subs**(*parameter\_values: dict*[*str*, *Number*]) → *FreeParameterExpression* | *Number* | *Expr*

Similar to a substitution in SymPy. Parameters are swapped for corresponding values or expressions from the dictionary.

### Parameters

**parameter\_values** (*dict*[*str*, *Number*]) – A mapping of parameters to their corresponding values to be assigned.

### Returns

*Union*[*FreeParameterExpression*, *Number*, *Expr*] – A numerical value if there are no symbols left in the expression otherwise returns a new FreeParameterExpression.

`braket.parametric.free_parameter_expression.subs_if_free_parameter`(*parameter*: Any, *\*\*kwargs*: [FreeParameterExpression](#) | *str*) → Any

Substitute a free parameter with the given kwargs, if any.

#### Parameters

- **parameter** (Any) – The parameter.
- **\*\*kwargs** (Union[[FreeParameterExpression](#), *str*]) – The kwargs to use to substitute.

#### Returns

Any – The substituted parameters.

### `braket.parametric.parameterizable` module

**class** `braket.parametric.parameterizable.Parameterizable`

Bases: ABC

A parameterized object is the abstract definition of an object that can take in FreeParameterExpressions.

**abstract property** `parameters`: list[[FreeParameterExpression](#) | [FreeParameter](#) | float]

Get the parameters.

#### Returns

list[Union[FreeParameterExpression, FreeParameter, float]] – The parameters associated with the object, either unbound free parameter expressions or bound values. The order of the parameters is determined by the subclass.

**abstract** `bind_values`(*\*\*kwargs*: [FreeParameter](#) | *str*) → Any

Takes in parameters and returns an object with specified parameters replaced with their values.

#### Parameters

**\*\*kwargs** (Union[[FreeParameter](#), *str*]) – Arbitrary keyword arguments.

#### Returns

Any – The result object will depend on the implementation of the object being bound.

### `braket.pulse` package

#### Subpackages

#### `braket.pulse.ast` namespace

#### Submodules

#### `braket.pulse.ast.approximation_parser` module

#### `braket.pulse.ast.free_parameters` module

#### `braket.pulse.ast.qasm_parser` module

`braket.pulse.ast.qasm_parser.ast_to_qasm(ast: Program) → str`

Converts an AST program to OpenQASM

#### Parameters

**ast** (*ast.Program*) – The AST program.

#### Returns

*str* – a str representing the OpenPulse program encoding the program.

## braket.pulse.ast.qasm\_transformer module

### Submodules

## braket.pulse.frame module

**class** `braket.pulse.frame.Frame`(*frame\_id: str, port: Port, frequency: float, phase: float = 0, is\_predefined: bool = False, properties: dict[str, Any] | None = None*)

Bases: object

Frame tracks the frame of reference, when interacting with the qubits, throughout the execution of a program. See <https://openqasm.com/language/openpulse.html#frames> for more details.

Initializes a Frame.

#### Parameters

- **frame\_id** (*str*) – str identifying a unique frame.
- **port** (*Port*) – port that this frame is attached to.
- **frequency** (*float*) – frequency to which this frame should be initialized.
- **phase** (*float*) – phase to which this frame should be initialized. Defaults to 0.
- **is\_predefined** (*bool*) – bool indicating whether this is a predefined frame on the device. Defaults to False.
- **properties** (*Optional[dict[str, Any]]*) – Dict containing properties of this frame. Defaults to None.

**property id:** *str*

Returns a str indicating the frame id.

## braket.pulse.port module

**class** `braket.pulse.port.Port`(*port\_id: str, dt: float, properties: dict[str, Any] | None = None*)

Bases: object

Ports represent any input or output component meant to manipulate and observe qubits on a device. See <https://openqasm.com/language/openpulse.html#ports> for more details.

Initializes a Port.

#### Parameters

- **port\_id** (*str*) – str identifying a unique port on the device.
- **dt** (*float*) – The smallest time step that may be used on the control hardware.

- **properties** (*Optional[dict[str, Any]]*) – Dict containing properties of this port. Defaults to None.

**property id:** `str`

Returns a str indicating the port id.

**property dt:** `float`

Returns the smallest time step that may be used on the control hardware.

## braket.pulse.pulse\_sequence module

**class** `braket.pulse.pulse_sequence.PulseSequence`

Bases: `object`

A representation of a collection of instructions to be performed on a quantum device and the requested results.

**to\_time\_trace()** → *PulseSequenceTrace*

Generate an approximate trace of the amplitude, frequency, phase for each frame contained in the `PulseSequence`, under the action of the instructions contained in the pulse sequence.

### Returns

*PulseSequenceTrace* – The approximation information with each attribute (amplitude, frequency and phase) mapping a str (frame id) to a `TimeSeries` (containing the time evolution of that attribute).

**property parameters:** `set[FreeParameter]`

Returns the set of `FreeParameter`s in the `PulseSequence`.

**set\_frequency**(*frame*: `Frame`, *frequency*: `float` | `FreeParameterExpression`) → *PulseSequence*

Adds an instruction to set the frequency of the frame to the specified `frequency` value.

### Parameters

- **frame** (`Frame`) – Frame for which the frequency needs to be set.
- **frequency** (`Union[float, FreeParameterExpression]`) – frequency value to set for the specified frame.

### Returns

*PulseSequence* – self, with the instruction added.

**shift\_frequency**(*frame*: `Frame`, *frequency*: `float` | `FreeParameterExpression`) → *PulseSequence*

Adds an instruction to shift the frequency of the frame by the specified `frequency` value.

### Parameters

- **frame** (`Frame`) – Frame for which the frequency needs to be shifted.
- **frequency** (`Union[float, FreeParameterExpression]`) – frequency value by which to shift the frequency for the specified frame.

### Returns

*PulseSequence* – self, with the instruction added.

**set\_phase**(*frame*: `Frame`, *phase*: `float` | `FreeParameterExpression`) → *PulseSequence*

Adds an instruction to set the phase of the frame to the specified phase value.

### Parameters

- **frame** (`Frame`) – Frame for which the frequency needs to be set.



- **phase** (*Union*[*float*, *FreeParameterExpression*]) – phase value to set for the specified frame.

**Returns**

*PulseSequence* – self, with the instruction added.

**shift\_phase**(*frame*: *Frame*, *phase*: *float* | *FreeParameterExpression*) → *PulseSequence*

Adds an instruction to shift the phase of the frame by the specified phase value.

**Parameters**

- **frame** (*Frame*) – Frame for which the phase needs to be shifted.
- **phase** (*Union*[*float*, *FreeParameterExpression*]) – phase value by which to shift the phase for the specified frame.

**Returns**

*PulseSequence* – self, with the instruction added.

**set\_scale**(*frame*: *Frame*, *scale*: *float* | *FreeParameterExpression*) → *PulseSequence*

Adds an instruction to set the scale on the frame to the specified scale value.

**Parameters**

- **frame** (*Frame*) – Frame for which the scale needs to be set.
- **scale** (*Union*[*float*, *FreeParameterExpression*]) – scale value to set on the specified frame.

**Returns**

*PulseSequence* – self, with the instruction added.

**delay**(*qubits\_or\_frames*: *Frame* | *list*[*Frame*] | *QubitSet*, *duration*: *float* | *FreeParameterExpression*) → *PulseSequence*

Adds an instruction to advance the frame clock by the specified duration value.

**Parameters**

- **qubits\_or\_frames** (*Union*[*Frame*, *list*[*Frame*], *QubitSet*]) – Qubits or frame(s) on which the delay needs to be introduced.
- **duration** (*Union*[*float*, *FreeParameterExpression*]) – value (in seconds) defining the duration of the delay.

**Returns**

*PulseSequence* – self, with the instruction added.

**barrier**(*qubits\_or\_frames*: *list*[*Frame*] | *QubitSet*) → *PulseSequence*

Adds an instruction to align the frame clocks to the latest time across all the specified frames.

**Parameters**

**qubits\_or\_frames** (*Union*[*list*[*Frame*], *QubitSet*]) – Qubits or frames which the delay needs to be introduced.

**Returns**

*PulseSequence* – self, with the instruction added.

**play**(*frame*: *Frame*, *waveform*: *Waveform*) → *PulseSequence*

Adds an instruction to play the specified waveform on the supplied frame.

**Parameters**

- **frame** (*Frame*) – Frame on which the specified waveform signal would be output.

- **waveform** (*Waveform*) – Waveform envelope specifying the signal to output on the specified frame.

**Returns**

*PulseSequence* – returns self.

**capture\_v0**(*frame: Frame*) → *PulseSequence*

Adds an instruction to capture the bit output from measuring the specified frame.

**Parameters**

**frame** (*Frame*) – Frame on which the capture operation needs to be performed.

**Returns**

*PulseSequence* – self, with the instruction added.

**make\_bound\_pulse\_sequence**(*param\_values: dict[str, float]*) → *PulseSequence*

Binds FreeParameters based upon their name and values passed in. If parameters share the same name, all the parameters of that name will be set to the mapped value.

**Parameters**

**param\_values** (*dict[str, float]*) – A mapping of FreeParameter names to a value to assign to them.

**Returns**

*PulseSequence* – Returns a PulseSequence with all present parameters fixed to their respective values.

**to\_ir**(*sort\_input\_parameters: bool = False*) → str

Converts this OpenPulse problem into IR representation.

**Parameters**

**sort\_input\_parameters** (*bool*) – whether input parameters should be printed in a sorted order. Defaults to False.

**Returns**

*str* – a str representing the OpenPulse program encoding the PulseSequence.

## braket.pulse.pulse\_sequence\_trace module

```
class braket.pulse.pulse_sequence_trace.PulseSequenceTrace(amplitudes: dict[str, TimeSeries],  
                                                           frequencies: dict[str, TimeSeries],  
                                                           phases: dict[str, TimeSeries])
```

Bases: object

This class encapsulates the data representing the PulseSequence execution. It contains the trace of amplitude, frequency and phase information for each frame in the PulseSequence.

**Parameters**

- **amplitudes** (*dict*) – A dictionary of frame ID to a TimeSeries of complex values specifying the waveform amplitude.
- **frequencies** (*dict*) – A dictionary of frame ID to a TimeSeries of float values specifying the waveform frequency.
- **phases** (*dict*) – A dictionary of frame ID to a TimeSeries of float values specifying the waveform phase.

**amplitudes:** *dict[str, TimeSeries]*

```
frequencies: dict[str, TimeSeries]
phases: dict[str, TimeSeries]
```

## braket.pulse.waveforms module

### class braket.pulse.waveforms.Waveform

Bases: ABC

A waveform is a time-dependent envelope that can be used to emit signals on an output port or receive signals from an input port. As such, when transmitting signals to the qubit, a frame determines time at which the waveform envelope is emitted, its carrier frequency, and its phase offset. When capturing signals from a qubit, at minimum a frame determines the time at which the signal is captured. See <https://openqasm.com/language/openpulse.html#waveforms> for more details.

**abstract sample**(*dt: float*) → ndarray

Generates a sample of amplitudes for this Waveform based on the given time resolution.

#### Parameters

**dt** (*float*) – The time resolution.

#### Returns

*np.ndarray* – The sample amplitudes for this waveform.

### class braket.pulse.waveforms.ArbitraryWaveform(*amplitudes: list[complex], id: str | None = None*)

Bases: *Waveform*

An arbitrary waveform with amplitudes at each timestep explicitly specified using an array.

Initializes an *ArbitraryWaveform*.

#### Parameters

- **amplitudes** (*list[complex]*) – Array of complex values specifying the waveform amplitude at each timestep. The timestep is determined by the sampling rate of the frame to which waveform is applied to.
- **id** (*Optional[str]*) – The identifier used for declaring this waveform. A random string of ascii characters is assigned by default.

**sample**(*dt: float*) → ndarray

Generates a sample of amplitudes for this Waveform based on the given time resolution.

#### Parameters

**dt** (*float*) – The time resolution.

#### Raises

**NotImplementedError** – This class does not implement sample.

#### Returns

*np.ndarray* – The sample amplitudes for this waveform.

### class braket.pulse.waveforms.ConstantWaveform(*length: float | FreeParameterExpression, iq: complex, id: str | None = None*)

Bases: *Waveform, Parameterizable*

A constant waveform which holds the supplied iq value as its amplitude for the specified length.

Initializes a *ConstantWaveform*.

#### Parameters

- **length** (*Union[float, FreeParameterExpression]*) – Value (in seconds) specifying the duration of the waveform.
- **iq** (*complex*) – complex value specifying the amplitude of the waveform.
- **id** (*Optional[str]*) – The identifier used for declaring this waveform. A random string of ascii characters is assigned by default.

**property parameters:** `list[FreeParameterExpression | FreeParameter | float]`

Returns the parameters associated with the object, either unbound free parameter expressions or bound values.

**Returns**

*list[Union[FreeParameterExpression, FreeParameter, float]]* – a list of parameters.

**bind\_values**(*\*\*kwargs: FreeParameter | str*) → *ConstantWaveform*

Takes in parameters and returns an object with specified parameters replaced with their values.

**Parameters**

**\*\*kwargs** (*Union[FreeParameter, str]*) – Arbitrary keyword arguments.

**Returns**

*ConstantWaveform* – A copy of this waveform with the requested parameters bound.

**sample**(*dt: float*) → *ndarray*

Generates a sample of amplitudes for this Waveform based on the given time resolution.

**Parameters**

**dt** (*float*) – The time resolution.

**Returns**

*np.ndarray* – The sample amplitudes for this waveform.

```
class braket.pulse.waveforms.DragGaussianWaveform(length: float | FreeParameterExpression, sigma:
float | FreeParameterExpression, beta: float |
FreeParameterExpression, amplitude: float |
FreeParameterExpression = 1, zero_at_edges: bool
= False, id: str | None = None)
```

Bases: *Waveform, Parameterizable*

A gaussian waveform with an additional gaussian derivative component and lifting applied.

Initializes a *DragGaussianWaveform*.

**Parameters**

- **length** (*Union[float, FreeParameterExpression]*) – Value (in seconds) specifying the duration of the waveform.
- **sigma** (*Union[float, FreeParameterExpression]*) – A measure (in seconds) of how wide or narrow the Gaussian peak is.
- **beta** (*Union[float, FreeParameterExpression]*) – The correction amplitude.
- **amplitude** (*Union[float, FreeParameterExpression]*) – The amplitude of the waveform envelope. Defaults to 1.
- **zero\_at\_edges** (*bool*) – bool specifying whether the waveform amplitude is clipped to zero at the edges. Defaults to False.
- **id** (*Optional[str]*) – The identifier used for declaring this waveform. A random string of ascii characters is assigned by default.

**property parameters:** `list[FreeParameterExpression | FreeParameter | float]`

Returns the parameters associated with the object, either unbound free parameter expressions or bound values.

**bind\_values**(\*\*kwargs: FreeParameter | str) → DragGaussianWaveform

Takes in parameters and returns an object with specified parameters replaced with their values.

**Parameters**

**\*\*kwargs** (Union[FreeParameter, str]) – Arbitrary keyword arguments.

**Returns**

DragGaussianWaveform – A copy of this waveform with the requested parameters bound.

**sample**(dt: float) → ndarray

Generates a sample of amplitudes for this Waveform based on the given time resolution.

**Parameters**

**dt** (float) – The time resolution.

**Returns**

np.ndarray – The sample amplitudes for this waveform.

**class** braket.pulse.waveforms.GaussianWaveform(length: float | FreeParameterExpression, sigma: float | FreeParameterExpression, amplitude: float | FreeParameterExpression = 1, zero\_at\_edges: bool = False, id: str | None = None)

Bases: Waveform, Parameterizable

A waveform with amplitudes following a gaussian distribution for the specified parameters.

Initializes a GaussianWaveform.

**Parameters**

- **length** (Union[float, FreeParameterExpression]) – Value (in seconds) specifying the duration of the waveform.
- **sigma** (Union[float, FreeParameterExpression]) – A measure (in seconds) of how wide or narrow the Gaussian peak is.
- **amplitude** (Union[float, FreeParameterExpression]) – The amplitude of the waveform envelope. Defaults to 1.
- **zero\_at\_edges** (bool) – bool specifying whether the waveform amplitude is clipped to zero at the edges. Defaults to False.
- **id** (Optional[str]) – The identifier used for declaring this waveform. A random string of ascii characters is assigned by default.

**property parameters:** `list[FreeParameterExpression | FreeParameter | float]`

Returns the parameters associated with the object, either unbound free parameter expressions or bound values.

**bind\_values**(\*\*kwargs: FreeParameter | str) → GaussianWaveform

Takes in parameters and returns an object with specified parameters replaced with their values.

**Parameters**

**\*\*kwargs** (Union[FreeParameter, str]) – Arbitrary keyword arguments.

**Returns**

GaussianWaveform – A copy of this waveform with the requested parameters bound.

**sample**(*dt: float*) → ndarray

Generates a sample of amplitudes for this Waveform based on the given time resolution.

**Parameters**

**dt** (*float*) – The time resolution.

**Returns**

*np.ndarray* – The sample amplitudes for this waveform.

## braket.quantum\_information package

### Submodules

#### braket.quantum\_information.pauli\_string module

**class** braket.quantum\_information.pauli\_string.**PauliString**(*pauli\_string: str | PauliString*)

Bases: object

A lightweight representation of a Pauli string with its phase.

Initializes a *PauliString*.

**Parameters**

**pauli\_string** (*Union[str, PauliString]*) – The representation of the pauli word, either a string or another PauliString object. A valid string consists of an optional phase, specified by an optional sign +/- followed by an uppercase string in {I, X, Y, Z}. Example valid strings are: XYZ, +YIZY, -YX

**Raises**

**ValueError** – If the Pauli String is empty.

**property phase: int**

The phase of the Pauli string.

Can be one of +/-1

**Type**

int

**property qubit\_count: int**

The number of qubits this Pauli string acts on.

**Type**

int

**to\_unsigned\_observable**(*include\_trivial: bool = False*) → *TensorProduct*

Returns the observable corresponding to the unsigned part of the Pauli string.

For example, for a Pauli string -XYZ, the corresponding observable is X Y Z.

**Parameters**

**include\_trivial** (*bool*) – Whether to include explicit identity factors in the observable.  
Default: False.

**Returns**

*TensorProduct* – The tensor product of the unsigned factors in the Pauli string.

**weight\_n\_substrings**(*weight*: *int*) → tuple[*PauliString*, ...]

Returns every substring of this Pauli string with exactly *weight* nontrivial factors.

The number of substrings is equal to  $\binom{n}{w}$ , where  $n$  is the number of nontrivial (non-identity) factors in the Pauli string and  $w$  is *weight*.

**Parameters**

**weight** (*int*) – The number of non-identity factors in the substrings.

**Returns**

tuple[*PauliString*, ...] – A tuple of weight-n Pauli substrings.

**eigenstate**(*signs*: *str* | list[*int*] | tuple[*int*, ...] | None = None) → *Circuit*

Returns the eigenstate of this Pauli string with the given factor signs.

The resulting eigenstate has each qubit in the +1 eigenstate of its corresponding signed Pauli operator. For example, a Pauli string +XYZ and signs +- has factors +X, +Y and -Z, with the corresponding qubits in states  $|+\rangle$ ,  $|i\rangle$ , and  $|1\rangle$  respectively (the global phase of the Pauli string is ignored).

**Parameters**

**signs** (Optional[Union[*str*, list[*int*], tuple[*int*, ...]]]) – The sign of each factor of the eigenstate, specified either as a string of “+” and “\_”, or as a list or tuple of +/-1. The length of signs must be equal to the length of the Pauli string. If not specified, it is assumed to be all +. Default: None.

**Returns**

*Circuit* – A circuit that prepares the desired eigenstate of the Pauli string.

**Raises**

**ValueError** – If the length of signs is not equal to that of the Pauli string or the signs are invalid.

**dot**(*other*: *PauliString*, *inplace*: *bool* = False) → *PauliString*

Right multiplies this Pauli string with the argument.

Returns the result of multiplying the current circuit by the argument on its right. For example, if called on -XYZ with argument ZYX, then YIY is the result. In-place computation is off by default.

**Parameters**

- **other** (*PauliString*) – The right multiplicand.
- **inplace** (*bool*) – If True, self is updated to hold the product.

**Returns**

*PauliString* – The resultant circuit from right multiplying self with other.

**Raises**

**ValueError** – If the lengths of the Pauli strings being multiplied differ.

**power**(*n*: *int*, *inplace*: *bool* = False) → *PauliString*

Composes Pauli string with itself n times.

**Parameters**

- **n** (*int*) – The number of times to self-multiply. Can be any integer value.
- **inplace** (*bool*) – Update self if True

**Returns**

*PauliString* – If n is positive, result from self-multiplication n times. If zero, identity. If negative, self-multiplication from trivial inverse (recall Pauli operators are involutory).

**Raises**

**ValueError** – If `n` isn't a plain Python `int`.

`to_circuit()` → *Circuit*

Returns circuit represented by this *PauliString*.

**Returns**

*Circuit* – The circuit for this *PauliString*.

**braket.registers package****Submodules****braket.registers.qubit module**

**class** `braket.registers.qubit.Qubit(index: int)`

Bases: `int`

A quantum bit index. The index of this qubit is locally scoped towards the contained circuit. This may not be the exact qubit index on the quantum device.

Creates a new *Qubit*.

**Parameters**

**index** (*int*) – Index of the qubit.

**Raises**

**ValueError** – If `index` is less than zero.

**Returns**

*Qubit* – Returns a new *Qubit* object.

**Examples**

```
>>> Qubit(0)
>>> Qubit(1)
```

**static new**(*qubit: Qubit | int*) → *Qubit*

Helper constructor - if input is a *Qubit* it returns the same value, else a new *Qubit* is constructed.

**Parameters**

**qubit** (*QubitInput*) – *Qubit* index. If `type == Qubit` then the *qubit* is returned.

**Returns**

*Qubit* – The qubit.



**braket.registers.qubit\_set module**

**class** `braket.registers.qubit_set.QubitSet`(*qubits: QubitSetInput | None = None*)

Bases: `IndexedSet`

An ordered, unique set of quantum bits.

---

**Note:** `QubitSet` implements `__hash__()` but is a mutable object, therefore be careful when mutating this object.

---

Initializes a `QubitSet`.

**Parameters**

**qubits** (`QubitSetInput` | `None`) – Qubits to be included in the `QubitSet`. Default is `None`.

**Examples**

```
>>> qubits = QubitSet([0, 1])
>>> for qubit in qubits:
...     print(qubit)
...
Qubit(0)
Qubit(1)
```

```
>>> qubits = QubitSet([0, 1, [2, 3]])
>>> for qubit in qubits:
...     print(qubit)
...
Qubit(0)
Qubit(1)
Qubit(2)
Qubit(3)
```

**map**(*mapping: dict[Qubit | int, Qubit | int]*) → `QubitSet`

Creates a new `QubitSet` where this instance's qubits are mapped to the values in `mapping`. If this instance contains a qubit that is not in the `mapping` that qubit is not modified.

**Parameters**

**mapping** (`dict[QubitInput, QubitInput]`) – A dictionary of qubit mappings to apply. Key is the qubit in this instance to target, and the value is what the key will be changed to.

**Returns**

`QubitSet` – A new `QubitSet` with the mapping applied.

## Examples

```
>>> qubits = QubitSet([0, 1])
>>> mapping = {0: 10, Qubit(1): Qubit(11)}
>>> qubits.map(mapping)
QubitSet([Qubit(10), Qubit(11)])
```

## braket.tasks package

### Submodules

#### braket.tasks.analog\_hamiltonian\_simulation\_quantum\_task\_result module

```
class braket.tasks.analog_hamiltonian_simulation_quantum_task_result.AnalogHamiltonianSimulationShotSta
```

Bases: str, Enum

An enumeration.

**SUCCESS** = 'Success'

**PARTIAL\_SUCCESS** = 'Partial Success'

**FAILURE** = 'Failure'

```
class braket.tasks.analog_hamiltonian_simulation_quantum_task_result.ShotResult(status:
    'AnalogHamiltonianSimulationShotStatus',
    pre_sequence:
    'np.ndarray'
    = None,
    post_sequence:
    'np.ndarray'
    = None)
```

Bases: object

**status:** *AnalogHamiltonianSimulationShotStatus*

**pre\_sequence:** ndarray = None

**post\_sequence:** ndarray = None

```
class braket.tasks.analog_hamiltonian_simulation_quantum_task_result.AnalogHamiltonianSimulationQuantumTaskResult:
```

Bases: object

**task\_metadata:** TaskMetadata

**additional\_metadata:** AdditionalMetadata

**measurements:** list[ShotResult] = None

**static from\_object**(result: AnalogHamiltonianSimulationTaskResult) →  
AnalogHamiltonianSimulationQuantumTaskResult

**static from\_string**(result: str) → AnalogHamiltonianSimulationQuantumTaskResult

**get\_counts**() → dict[str, int]

Aggregate state counts from AHS shot results.

## Notes

We use the following convention to denote the state of an atom (site). e: empty site r: Rydberg state atom  
g: ground state atom

### Returns

*dict[str, int]* – number of times each state configuration is measured. Returns None if none of shot measurements are successful. Only successful shots contribute to the state count.

**get\_avg\_density**() → ndarray

Get the average Rydberg state densities from the result

### Returns

*np.ndarray* – The average densities from the result

**braket.tasks.annealing\_quantum\_task\_result module**

```
class braket.tasks.annealing_quantum_task_result.AnnealingQuantumTaskResult(record_array:
    recarray,
    variable_count:
    int,
    problem_type:
    ProblemType,
    task_metadata:
    TaskMetadata,
    addi-
    tional_metadata:
    AdditionalMeta-
    data)
```

Bases: object

Result of an annealing problem quantum task execution. This class is intended to be initialized by a QuantumTask class.

**Parameters**

- **record\_array** (*np.recarray*) – numpy array with keys ‘solution’ (np.ndarray) where row is solution, column is value of the variable, ‘solution\_count’ (numpy.ndarray) the number of times the solutions occurred, and ‘value’ (numpy.ndarray) the output or energy of the solutions.
- **variable\_count** (*int*) – the number of variables
- **problem\_type** (*ProblemType*) – the type of annealing problem
- **task\_metadata** (*TaskMetadata*) – Quantum task metadata.
- **additional\_metadata** (*AdditionalMetadata*) – Additional metadata about the quantum task

**record\_array:** recarray

**variable\_count:** int

**problem\_type:** ProblemType

**task\_metadata:** TaskMetadata

**additional\_metadata:** AdditionalMetadata

**data**(*selected\_fields: list[str] | None = None, sorted\_by: str = 'value', reverse: bool = False*) → Generator[tuple]

Yields the data in record\_array

**Parameters**

- **selected\_fields** (*Optional[list[str]]*) – selected fields to return. Options are ‘solution’, ‘value’, and ‘solution\_count’. Default is None.
- **sorted\_by** (*str*) – Sorts the data by this field. Options are ‘solution’, ‘value’, and ‘solution\_count’. Default is ‘value’.
- **reverse** (*bool*) – If True, returns the data in reverse order. Default is False.

**Yields**

*Generator[tuple]* – data in record\_array

**static from\_object**(*result: AnnealingTaskResult*) → *AnnealingQuantumTaskResult*

Create AnnealingQuantumTaskResult from AnnealingTaskResult object

**Parameters**

**result** (*AnnealingTaskResult*) – AnnealingTaskResult object

**Returns**

*AnnealingQuantumTaskResult* – An AnnealingQuantumTaskResult based on the given result object

**static from\_string**(*result: str*) → *AnnealingQuantumTaskResult*

Create AnnealingQuantumTaskResult from string

**Parameters**

**result** (*str*) – JSON object string

**Returns**

*AnnealingQuantumTaskResult* – An AnnealingQuantumTaskResult based on the given string

**braket.tasks.gate\_model\_quantum\_task\_result module**

```
class braket.tasks.gate_model_quantum_task_result.GateModelQuantumTaskResult(task_metadata:
                                                                              TaskMetadata,
                                                                              addi-
                                                                              tional_metadata:
                                                                              Additional-
                                                                              Metadata,
                                                                              result_types:
                                                                              list[ResultTypeValue]
                                                                              | None = None,
                                                                              values:
                                                                              list[Any] | None
                                                                              = None,
                                                                              measurements:
                                                                              ndarray | None
                                                                              = None, mea-
                                                                              sured_qubits:
                                                                              list[int] | None
                                                                              = None,
                                                                              measure-
                                                                              ment_counts:
                                                                              Counter | None
                                                                              = None,
                                                                              measure-
                                                                              ment_probabilities:
                                                                              dict[str, float] |
                                                                              None = None,
                                                                              measure-
                                                                              ments_copied_from_device:
                                                                              bool | None =
                                                                              None, measure-
                                                                              ment_counts_copied_from_device:
                                                                              bool | None =
                                                                              None, measure-
                                                                              ment_probabilities_copied_from_device:
                                                                              bool | None =
                                                                              None, _re-
                                                                              sult_types_indices:
                                                                              dict[str, int] |
                                                                              None = None)
```

Bases: object

Result of a gate model quantum task execution. This class is intended to be initialized by a QuantumTask class.

#### Parameters

- **task\_metadata** (*TaskMetadata*) – Quantum task metadata.
- **additional\_metadata** (*AdditionalMetadata*) – Additional metadata about the quantum task
- **result\_types** (*list[dict[str, Any]]*) – List of dictionaries where each dictionary has two keys: ‘Type’ (the result type in IR JSON form) and ‘Value’ (the result value for this result type). This can be an empty list if no result types are specified in the IR. This is calculated from [measurements](#) and the IR of the circuit program when `shots>0`.
- **values** (*list[Any]*) – The values for result types requested in the circuit. This can be an empty list if no result types are specified in the IR. This is calculated from [measurements](#)

and the IR of the circuit program when shots>0.

- **measurements** (*numpy.ndarray*, *optional*) – 2d array - row is shot and column is qubit. Default is None. Only available when shots > 0. The qubits in *measurements* are the ones in *GateModelQuantumTaskResult.measured\_qubits*.
- **measured\_qubits** (*list[int]*, *optional*) – The indices of the measured qubits. Default is None. Only available when shots > 0. Indicates which qubits are in *measurements*.
- **measurement\_counts** (*Counter*, *optional*) – A Counter of measurements. Key is the measurements in a big endian binary string. Value is the number of times that measurement occurred. Default is None. Only available when shots > 0. Note that the keys in Counter are unordered.
- **measurement\_probabilities** (*dict[str, float]*, *optional*) – A dictionary of probabilistic results. Key is the measurements in a big endian binary string. Value is the probability the measurement occurred. Default is None. Only available when shots > 0.
- **measurements\_copied\_from\_device** (*bool*, *optional*) – flag whether *measurements* were copied from device. If false, *measurements* are calculated from device data. Default is None. Only available when shots > 0.
- **measurement\_counts\_copied\_from\_device** (*bool*, *optional*) – flag whether *measurement\_counts* were copied from device. If False, *measurement\_counts* are calculated from device data. Default is None. Only available when shots > 0.
- **measurement\_probabilities\_copied\_from\_device** (*bool*, *optional*) – flag whether *measurement\_probabilities* were copied from device. If false, *measurement\_probabilities* are calculated from device data. Default is None. Only available when shots > 0.

**task\_metadata:** TaskMetadata

**additional\_metadata:** AdditionalMetadata

**result\_types:** list[ResultTypeValue] = None

**values:** list[Any] = None

**measurements:** ndarray = None

**measured\_qubits:** list[int] = None

**measurement\_counts:** Counter = None

**measurement\_probabilities:** dict[str, float] = None

**measurements\_copied\_from\_device:** bool = None

**measurement\_counts\_copied\_from\_device:** bool = None

**measurement\_probabilities\_copied\_from\_device:** bool = None

**get\_value\_by\_result\_type**(*result\_type*: ResultType) → Any

Get value by result type. The result type must have already been requested in the circuit sent to the device for this quantum task result.

#### Parameters

**result\_type** (ResultType) – result type requested

#### Returns

Any – value of the result corresponding to the result type

**Raises**

**ValueError** – If result type is not found in result. Result types must be added to the circuit before the circuit is run on a device.

**get\_compiled\_circuit()** → str | None

Get the compiled circuit, if one is available.

**Returns**

*Optional[str]* – The compiled circuit or None.

**static measurement\_counts\_from\_measurements**(*measurements: ndarray*) → Counter

Creates measurement counts from measurements

**Parameters**

**measurements** (*np.ndarray*) – 2d array - row is shot and column is qubit.

**Returns**

*Counter* – A Counter of measurements. Key is the measurements in a big endian binary string. Value is the number of times that measurement occurred.

**static measurement\_probabilities\_from\_measurement\_counts**(*measurement\_counts: Counter*) → dict[str, float]

Creates measurement probabilities from measurement counts

**Parameters**

**measurement\_counts** (*Counter*) – A Counter of measurements. Key is the measurements in a big endian binary string. Value is the number of times that measurement occurred.

**Returns**

*dict[str, float]* – A dictionary of probabilistic results. Key is the measurements in a big endian binary string. Value is the probability the measurement occurred.

**static measurements\_from\_measurement\_probabilities**(*measurement\_probabilities: dict[str, float], shots: int*) → ndarray

Creates measurements from measurement probabilities.

**Parameters**

- **measurement\_probabilities** (*dict[str, float]*) – A dictionary of probabilistic results. Key is the measurements in a big endian binary string. Value is the probability the measurement occurred.
- **shots** (*int*) – number of iterations on device.

**Returns**

*np.ndarray* – A dictionary of probabilistic results. Key is the measurements in a big endian binary string. Value is the probability the measurement occurred.

**static from\_object**(*result: GateModelTaskResult*) → *GateModelQuantumTaskResult*

Create GateModelQuantumTaskResult from GateModelTaskResult object.

**Parameters**

**result** (*GateModelTaskResult*) – GateModelTaskResult object

**Returns**

*GateModelQuantumTaskResult* – A GateModelQuantumTaskResult based on the given dict

**Raises**

**ValueError** – If neither “Measurements” nor “MeasurementProbabilities” is a key in the result dict



**static from\_string**(*result: str*) → *GateModelQuantumTaskResult*

Create GateModelQuantumTaskResult from string.

**Parameters**

**result** (*str*) – JSON object string, with GateModelQuantumTaskResult attributes as keys.

**Returns**

*GateModelQuantumTaskResult* – A GateModelQuantumTaskResult based on the given string

**Raises**

**ValueError** – If neither “Measurements” nor “MeasurementProbabilities” is a key in the result dict

**static cast\_result\_types**(*gate\_model\_task\_result: GateModelTaskResult*) → None

Casts the result types to the types expected by the SDK.

**Parameters**

**gate\_model\_task\_result** (*GateModelTaskResult*) – GateModelTaskResult representing the results.

### braket.tasks.local\_quantum\_task module

**class** `braket.tasks.local_quantum_task.LocalQuantumTask`(*result: GateModelQuantumTaskResult | AnnealingQuantumTaskResult | PhotonicModelQuantumTaskResult*)

Bases: *QuantumTask*

A quantum task containing the results of a local simulation.

Since this class is instantiated with the results, `cancel()` and `run_async()` are unsupported.

**property id:** *str*

Gets the task ID.

**Returns**

*str* – The ID of the task.

**cancel()** → None

Cancel the quantum task.

**state()** → *str*

Gets the state of the task.

**Returns**

*str* – Returns COMPLETED

**result()** → *GateModelQuantumTaskResult | AnnealingQuantumTaskResult | PhotonicModelQuantumTaskResult*

Get the quantum task result.

**Returns**

*Union[GateModelQuantumTaskResult, AnnealingQuantumTaskResult, PhotonicModelQuantumTaskResult]* – Get the quantum task result. Call `async_result` if you want the result in an asynchronous way.

**async\_result()** → Task

Get the quantum task result asynchronously.

**Raises**

**NotImplementedError** – Asynchronous local simulation unsupported

**Returns**

*asyncio.Task* – Get the quantum task result asynchronously.

**braket.tasks.local\_quantum\_task\_batch module**

```
class braket.tasks.local_quantum_task_batch.LocalQuantumTaskBatch(results:
                                                                    list[GateModelQuantumTaskResult
                                                                    |
                                                                    AnnealingQuantumTaskResult
                                                                    | PhotonicModelQuantum-
                                                                    TaskResult])
```

Bases: *QuantumTaskBatch*

Executes a batch of quantum tasks in parallel.

Since this class is instantiated with the results, `cancel()` and `run_async()` are unsupported.

**results()** → list[*GateModelQuantumTaskResult* | *AnnealingQuantumTaskResult* |  
*PhotonicModelQuantumTaskResult*]

Get the quantum task results.

**Returns**

list[Union[*GateModelQuantumTaskResult*, *AnnealingQuantumTaskResult*, *PhotonicMod-*  
*elQuantumTaskResult*]] – Get the quantum task results.

**braket.tasks.photonic\_model\_quantum\_task\_result module**

```
class braket.tasks.photonic_model_quantum_task_result.PhotonicModelQuantumTaskResult(task_metadata:
                                                                                          'TaskMeta-
                                                                                          data',
                                                                                          addi-
                                                                                          tional_metadata:
                                                                                          'Ad-
                                                                                          di-
                                                                                          tional-
                                                                                          Meta-
                                                                                          data',
                                                                                          mea-
                                                                                          sure-
                                                                                          ments:
                                                                                          'np.ndarray'
                                                                                          =
                                                                                          None)
```

Bases: object

**task\_metadata:** TaskMetadata

**additional\_metadata:** AdditionalMetadata

**measurements:** ndarray = None

**static from\_object**(*result: PhotonicModelTaskResult*) → *PhotonicModelQuantumTaskResult*

Create PhotonicModelQuantumTaskResult from PhotonicModelTaskResult object.

**Parameters**

**result** (*PhotonicModelTaskResult*) – PhotonicModelTaskResult object

**Returns**

*PhotonicModelQuantumTaskResult* – A PhotonicModelQuantumTaskResult based on the given dict

**Raises**

**ValueError** – If “measurements” is not a key in the result dict

**static from\_string**(*result: str*) → *PhotonicModelQuantumTaskResult*

## braket.tasks.quantum\_task module

**class** `braket.tasks.quantum_task.QuantumTask`

Bases: ABC

An abstraction over a quantum task on a quantum device.

**abstract property id:** `str`

Get the quantum task ID.

**Returns**

*str* – The quantum task ID.

**abstract cancel**() → None

Cancel the quantum task.

**abstract state**() → `str`

Get the state of the quantum task.

**Returns**

*str* – State of the quantum task.

**abstract result**() → *GateModelQuantumTaskResult* | *AnnealingQuantumTaskResult* | *PhotonicModelQuantumTaskResult*

Get the quantum task result.

**Returns**

*Union[GateModelQuantumTaskResult, AnnealingQuantumTaskResult, PhotonicModelQuantumTaskResult]* – Get the quantum task result. Call `async_result` if you want the result in an asynchronous way.

**abstract async\_result**() → Task

Get the quantum task result asynchronously.

**Returns**

*asyncio.Task* – Get the quantum task result asynchronously.

**metadata**(*use\_cached\_value: bool = False*) → dict[str, Any]

Get task metadata.

**Parameters**

**use\_cached\_value** (*bool*) – If True, uses the value retrieved from the previous request. Default is False.

**Returns**

*dict[str, Any]* – The metadata regarding the quantum task. If *use\_cached\_value* is *True*, then the value retrieved from the most recent request is used.

**braket.tasks.quantum\_task\_batch module**

**class** `braket.tasks.quantum_task_batch.QuantumTaskBatch`

Bases: *ABC*

An abstraction over a quantum task batch on a quantum device.

**abstract results()** → *list[GateModelQuantumTaskResult | AnnealingQuantumTaskResult | PhotonicModelQuantumTaskResult]*

Get the quantum task results.

**Returns**

*list[Union[GateModelQuantumTaskResult, AnnealingQuantumTaskResult, PhotonicModelQuantumTaskResult]]* – Get the quantum task results.

**braket.timings package****Submodules****braket.timings.time\_series module**

**class** `braket.timings.time_series.TimeSeriesItem`(*time: 'Number', value: 'Number'*)

Bases: *object*

**time:** *Number*

**value:** *Number*

**class** `braket.timings.time_series.StitchBoundaryCondition`(*value*)

Bases: *str, Enum*

An enumeration.

**MEAN** = *'mean'*

**LEFT** = *'left'*

**RIGHT** = *'right'*

**class** `braket.timings.time_series.TimeSeries`

Bases: *object*

**put**(*time: Number, value: Number*) → *TimeSeries*

Puts a value to the time series at the given time. A value passed to an existing time will overwrite the current value.

**Parameters**

- **time** (*Number*) – The time of the value.
- **value** (*Number*) – The value to add to the time series.

**Returns***TimeSeries* – returns self (to allow for chaining).**times()** → list[Number]

Returns the times in the time series.

**Returns***list[Number]* – The times in the time series.**values()** → list[Number]

Returns the values in the time series.

**Returns***list[Number]* – The values in the time series.**static from\_lists**(times: list[float], values: list[float]) → *TimeSeries*

Create a time series from the list of time and value points

**Parameters**

- **times** (*list[float]*) – list of time points
- **values** (*list[float]*) – list of value points

**Returns***TimeSeries* – time series constructed from lists**Raises****ValueError** – If the len of *times* does not equal len of *values*.**static constant\_like**(times: list | float | *TimeSeries*, constant: float = 0.0) → *TimeSeries*

Obtain a constant time series given another time series or the list of time points, and the constant values.

**Parameters**

- **times** (*list | float | TimeSeries*) – list of time points or a time series
- **constant** (*float*) – constant value

**Returns***TimeSeries* – A constant time series**concatenate**(other: *TimeSeries*) → *TimeSeries*

Concatenates two time series into a single time series. The time points in the final time series are obtained by concatenating two lists of time points from the first and the second time series. Similarly, the values in the final time series is a concatenated list of the values in the first and the second time series.

**Parameters**

**other** (*TimeSeries*) – The second time series to be concatenated Notes: Keeps the time points in both time series unchanged. Assumes that the time points in the first *TimeSeries* are at earlier times than the time points in the second *TimeSeries*.

**Returns***TimeSeries* – The concatenated time series.**Raises**

**ValueError** – If the timeseries is not empty and time points in the first *TimeSeries* are not strictly smaller than in the second.

Example:

```
time_series_1 = TimeSeries.from_lists(times=[0, 0.1], values=[1, 2])
time_series_2 = TimeSeries.from_lists(times=[0.2, 0.3], values=[4, 5])

concat_ts = time_series_1.concatenate(time_series_2)

Result:
concat_ts.times() = [0, 0.1, 0.2, 0.3]
concat_ts.values() = [1, 2, 4, 5]
```

**stitch**(*other*: [TimeSeries](#), *boundary*: [StitchBoundaryCondition](#) = [StitchBoundaryCondition.MEAN](#)) → [TimeSeries](#)

Stitch two time series to a single time series. The time points of the second time series are shifted such that the first time point of the second series coincides with the last time point of the first series. The boundary point value is handled according to [StitchBoundaryCondition](#) argument value.

#### Parameters

- **other** ([TimeSeries](#)) – The second time series to be stitched with.
- **boundary** ([StitchBoundaryCondition](#)) – {"mean", "left", "right"}. Boundary point handler.

Possible options are

- "mean" - take the average of the boundary value points of the first and the second time series.
- "left" - use the last value from the left time series as the boundary point.
- "right" - use the first value from the right time series as the boundary point.

#### Returns

[TimeSeries](#) – The stitched time series.

#### Raises

**ValueError** – If boundary is not one of {"mean", "left", "right"}.

Example ([StitchBoundaryCondition.MEAN](#)):

```
time_series_1 = TimeSeries.from_lists(times=[0, 0.1], values=[1, 2])
time_series_2 = TimeSeries.from_lists(times=[0.2, 0.4], values=[4, 5])

stitch_ts = time_series_1.stitch(time_series_2,
    ↪boundary=StitchBoundaryCondition.MEAN)

Result:
stitch_ts.times() = [0, 0.1, 0.3]
stitch_ts.values() = [1, 3, 5]
```

Example ([StitchBoundaryCondition.LEFT](#)):

```
stitch_ts = time_series_1.stitch(time_series_2,
    ↪boundary=StitchBoundaryCondition.LEFT)

Result:
stitch_ts.times() = [0, 0.1, 0.3]
stitch_ts.values() = [1, 2, 5]
```

Example (StitchBoundaryCondition.RIGHT):

```
stitch_ts = time_series_1.stitch(time_series_2,
    ↪boundary=StitchBoundaryCondition.RIGHT)
```

Result:

```
stitch_ts.times() = [0, 0.1, 0.3]
stitch_ts.values() = [1, 4, 5]
```

**discretize**(*time\_resolution: Decimal | None, value\_resolution: Decimal | None*) → *TimeSeries*

Creates a discretized version of the time series, rounding all times and values to the closest multiple of the corresponding resolution.

#### Parameters

- **time\_resolution** (*Optional [Decimal]*) – Time resolution
- **value\_resolution** (*Optional [Decimal]*) – Value resolution

#### Returns

*TimeSeries* – A new discretized time series.

**static periodic\_signal**(*times: list[float], values: list[float], num\_repeat: int = 1*) → *TimeSeries*

Create a periodic time series by repeating the same block multiple times.

#### Parameters

- **times** (*list [float]*) – List of time points in a single block
- **values** (*list [float]*) – Values for the time series in a single block
- **num\_repeat** (*int*) – Number of block repetitions

#### Raises

**ValueError** – If the first and last values are not the same

#### Returns

*TimeSeries* – A new periodic time series.

**static trapezoidal\_signal**(*area: float, value\_max: float, slew\_rate\_max: float, time\_separation\_min: float = 0.0*) → *TimeSeries*

Get a trapezoidal time series with specified area, maximum value, maximum slew rate and minimum separation of time points

#### Parameters

- **area** (*float*) – Total area under the time series
- **value\_max** (*float*) – The maximum value of the time series
- **slew\_rate\_max** (*float*) – The maximum slew rate
- **time\_separation\_min** (*float*) – The minimum separation of time points

#### Raises

**ValueError** – If the time separation is negative

#### Returns

*TimeSeries* – A trapezoidal time series

Notes: The area of a time series  $f(t)$  is defined as the time integral of  $f(t)$  from  $t=0$  to  $t=T$ , where  $T$  is the duration. We also assume the trapezoidal time series starts and ends at zero.

## braket.tracking package

### Submodules

#### braket.tracking.pricing module

**class** `braket.tracking.pricing.Pricing`

Bases: `object`

**get\_prices()** → `None`

Retrieves the price list.

**price\_search(\*\*kwargs: str)** → `list[dict[str, str]]`

Searches the price list for a given set of parameters.

**Parameters**

**\*\*kwargs** (*str*) – Arbitrary keyword arguments.

**Returns**

*list[dict[str, str]]* – The price list.

`braket.tracking.pricing.price_search(**kwargs: str)` → `list[dict[str, str]]`

Searches the price list for a given set of parameters.

**Parameters**

**\*\*kwargs** (*str*) – Arbitrary keyword arguments.

**Returns**

*list[dict[str, str]]* – The price list.

#### braket.tracking.tracker module

**class** `braket.tracking.tracker.Tracker`

Bases: `object`

Amazon Braket cost tracker. Use this class to track costs incurred from quantum tasks on Amazon Braket.

**start()** → *Tracker*

Start tracking resources with this tracker.

**Returns**

*Tracker* – self.

**stop()** → *Tracker*

Stop tracking resources with this tracker.

**Returns**

*Tracker* – self.

**receive\_event(event: \_TaskCreationEvent)** → `None`

Process a Task Creation Event.

**Parameters**

**event** (*\_TaskCreationEvent*) – The event to process.



**tracked\_resources()** → list[str]

Resources tracked by this tracker.

**Returns**

*list[str]* – The list of quantum task ids for quantum tasks tracked by this tracker.

**qpu\_tasks\_cost()** → Decimal

Estimate cost of all quantum tasks tracked by this tracker that use Braket qpu devices.

Note: Charges shown are estimates based on your Amazon Braket simulator and quantum processing unit (QPU) task usage. Estimated charges shown may differ from your actual charges. Estimated charges do not factor in any discounts or credits, and you may experience additional charges based on your use of other services such as Amazon Elastic Compute Cloud (Amazon EC2).

**Returns**

*Decimal* – The estimated total cost in USD

**simulator\_tasks\_cost()** → Decimal

**Estimate cost of all quantum tasks tracked by this tracker using Braket simulator devices.**

Note: The cost of a simulator quantum task is not available until after the results for the task have been fetched. Call `result()` on an `AwsQuantumTask` before estimating its cost to ensure that the simulator usage is included in the cost estimate.

Note: Charges shown are estimates based on your Amazon Braket simulator and quantum processing unit (QPU) task usage. Estimated charges shown may differ from your actual charges. Estimated charges do not factor in any discounts or credits, and you may experience additional charges based on your use of other services such as Amazon Elastic Compute Cloud (Amazon EC2).

**Returns**

*Decimal* – The estimated total cost in USD

**quantum\_tasks\_statistics()** → dict[str, dict[str, Any]]

Get a summary of quantum tasks grouped by device.

**Returns**

*dict[str, dict[str, Any]]* – A dictionary where each key is a device arn, and maps to a dictionary summarizing the quantum tasks run on the device. The summary includes the total shots sent to the device and the most recent status of the quantum tasks created on this device. For finished quantum tasks on simulator devices, the summary also includes the duration of the simulation.

### Example

```
>>> tracker.quantum_tasks_statistics()
{'qpu_device_foo':
  {'shots' : 1000,
   'tasks' : { 'COMPLETED' : 4,
                'QUEUED' : 1 },
  },
 'simulator_device_bar':
  {'shots' : 1000
   'tasks' : { 'COMPLETED' : 2,
                'CREATED' : 1},
   'execution_duration' : datetime.timedelta(seconds=5, microseconds=654321),
```

(continues on next page)

(continued from previous page)

```
'billed_execution_duration' : datetime.timedelta(seconds=6,
↪microseconds=123456)}}
```

## braket.tracking.tracking\_context module

**class** `braket.tracking.tracking_context.TrackingContext`

Bases: `object`

**register\_tracker**(*tracker*: `Tracker`) → `None`

Registers a tracker.

### Parameters

**tracker** (`Tracker`) – The tracker.

**deregister\_tracker**(*tracker*: `Tracker`) → `None`

Deregisters a tracker.

### Parameters

**tracker** (`Tracker`) – The tracker.

**broadcast\_event**(*event*: `_TrackingEvent`) → `None`

Broadcasts an event to all trackers.

### Parameters

**event** (`_TrackingEvent`) – The event to broadcast.

**active\_trackers**() → `set`

Gets the active trackers.

### Returns

*set* – The set of active trackers.

`braket.tracking.tracking_context.register_tracker`(*tracker*: `Tracker`) → `None`

Registers a tracker.

### Parameters

**tracker** (`Tracker`) – The tracker.

`braket.tracking.tracking_context.deregister_tracker`(*tracker*: `Tracker`) → `None`

Deregisters a tracker.

### Parameters

**tracker** (`Tracker`) – The tracker.

`braket.tracking.tracking_context.broadcast_event`(*event*: `_TrackingEvent`) → `None`

Broadcasts an event to all trackers.

### Parameters

**event** (`_TrackingEvent`) – The event to broadcast.

`braket.tracking.tracking_context.active_trackers`() → `set`

Gets the active trackers.

### Returns

*set* – The set of active trackers.

**braket.tracking.tracking\_events module****Submodules****braket.ipython\_utils module**

`braket.ipython_utils.running_in_jupyter()` → bool

Determine if running within Jupyter.

Inspired by <https://github.com/ipython/ipython/issues/11694>

**Returns**

*bool* – True if running in Jupyter, else False.



## PYTHON MODULE INDEX

### b

- braket, 11
- braket.ahs, 11
  - braket.ahs.analog\_hamiltonian\_simulation, 11
  - braket.ahs.atom\_arrangement, 12
  - braket.ahs.discretization\_types, 13
  - braket.ahs.driving\_field, 14
  - braket.ahs.field, 16
  - braket.ahs.hamiltonian, 16
  - braket.ahs.local\_detuning, 17
  - braket.ahs.pattern, 19
  - braket.ahs.shifting\_field, 19
- braket.annealing, 19
  - braket.annealing.problem, 19
- braket.aws, 21
  - braket.aws.aws\_device, 21
  - braket.aws.aws\_quantum\_job, 28
  - braket.aws.aws\_quantum\_task, 33
  - braket.aws.aws\_quantum\_task\_batch, 36
  - braket.aws.aws\_session, 39
  - braket.aws.queue\_information, 45
- braket.circuits, 47
  - braket.circuits.angled\_gate, 62
  - braket.circuits.ascii\_circuit\_diagram, 66
  - braket.circuits.basis\_state, 66
  - braket.circuits.braket\_program\_context, 66
  - braket.circuits.circuit, 68
  - braket.circuits.circuit\_diagram, 109
  - braket.circuits.circuit\_helpers, 109
  - braket.circuits.compiler\_directive, 110
  - braket.circuits.compiler\_directives, 111
  - braket.circuits.free\_parameter, 111
  - braket.circuits.free\_parameter\_expression, 111
  - braket.circuits.gate, 111
  - braket.circuits.gate\_calibrations, 174
  - braket.circuits.gates, 175
  - braket.circuits.instruction, 241
  - braket.circuits.measure, 244
  - braket.circuits.moments, 245
  - braket.circuits.noise, 248
  - braket.circuits.noise\_helpers, 280
  - braket.circuits.noise\_model, 47
  - braket.circuits.noise\_model.circuit\_instruction\_criteria, 47
  - braket.circuits.noise\_model.criteria, 47
  - braket.circuits.noise\_model.criteria\_input\_parsing, 52
  - braket.circuits.noise\_model.gate\_criteria, 53
  - braket.circuits.noise\_model.initialization\_criteria, 54
  - braket.circuits.noise\_model.noise\_model, 54
  - braket.circuits.noise\_model.observable\_criteria, 57
  - braket.circuits.noise\_model.qubit\_initialization\_criteria, 58
  - braket.circuits.noise\_model.result\_type\_criteria, 59
  - braket.circuits.noise\_model.unitary\_gate\_criteria, 60
  - braket.circuits.noises, 282
  - braket.circuits.observable, 307
  - braket.circuits.observables, 314
  - braket.circuits.operator, 320
  - braket.circuits.parameterizable, 321
  - braket.circuits.quantum\_operator, 321
  - braket.circuits.quantum\_operator\_helpers, 322
  - braket.circuits.qubit, 324
  - braket.circuits.qubit\_set, 324
  - braket.circuits.result\_type, 324
  - braket.circuits.result\_types, 333
  - braket.circuits.serialization, 340
  - braket.circuits.text\_diagram\_builders, 61
  - braket.circuits.text\_diagram\_builders.ascii\_circuit\_diagram, 61
  - braket.circuits.text\_diagram\_builders.text\_circuit\_diagram, 61
  - braket.circuits.text\_diagram\_builders.text\_circuit\_diagram, 61
  - braket.circuits.text\_diagram\_builders.unicode\_circuit\_diagram, 61
  - braket.circuits.translations, 341
  - braket.circuits.unitary\_calculation, 341
- braket.devices, 342

- braket.devices.device, 342
- braket.devices.devices, 343
- braket.devices.local\_simulator, 343
- braket.error\_mitigation, 345
- braket.error\_mitigation.debias, 345
- braket.error\_mitigation.error\_mitigation, 346
- braket.ipython\_utils, 399
- braket.jobs, 346
- braket.jobs.config, 355
- braket.jobs.data\_persistence, 356
- braket.jobs.environment\_variables, 358
- braket.jobs.hybrid\_job, 358
- braket.jobs.image\_uris, 360
- braket.jobs.local, 346
- braket.jobs.local.local\_job, 346
- braket.jobs.local.local\_job\_container, 350
- braket.jobs.local.local\_job\_container\_setup, 350
- braket.jobs.logs, 361
- braket.jobs.metrics, 363
- braket.jobs.metrics\_data, 350
- braket.jobs.metrics\_data.cwl\_insights\_metrics\_fetcher, 350
- braket.jobs.metrics\_data.cwl\_metrics\_fetcher, 352
- braket.jobs.metrics\_data.definitions, 353
- braket.jobs.metrics\_data.exceptions, 353
- braket.jobs.metrics\_data.log\_metrics\_parser, 353
- braket.jobs.quantum\_job, 363
- braket.jobs.quantum\_job\_creation, 365
- braket.jobs.serialization, 367
- braket.parametric, 368
- braket.parametric.free\_parameter, 368
- braket.parametric.free\_parameter\_expression, 369
- braket.parametric.parameterizable, 370
- braket.pulse, 370
- braket.pulse.ast, 370
- braket.pulse.ast.approximation\_parser, 370
- braket.pulse.ast.free\_parameters, 370
- braket.pulse.ast.qasm\_parser, 370
- braket.pulse.ast.qasm\_transformer, 371
- braket.pulse.frame, 371
- braket.pulse.port, 371
- braket.pulse.pulse\_sequence, 372
- braket.pulse.pulse\_sequence\_trace, 374
- braket.pulse.waveforms, 375
- braket.quantum\_information, 378
- braket.quantum\_information.pauli\_string, 378
- braket.registers, 380
- braket.registers.qubit, 380
- braket.registers.qubit\_set, 381
- braket.tasks, 382
- braket.tasks.analog\_hamiltonian\_simulation\_quantum\_task\_result, 382
- braket.tasks.annealing\_quantum\_task\_result, 384
- braket.tasks.gate\_model\_quantum\_task\_result, 385
- braket.tasks.local\_quantum\_task, 389
- braket.tasks.local\_quantum\_task\_batch, 390
- braket.tasks.photonic\_model\_quantum\_task\_result, 390
- braket.tasks.quantum\_task, 391
- braket.tasks.quantum\_task\_batch, 392
- braket.timings, 392
- braket.timings.time\_series, 392
- braket.tracking, 396
- braket.tracking.pricing, 396
- braket.tracking.tracker, 396
- braket.tracking.tracking\_context, 398
- braket.tracking.tracking\_events, 399

## A

`account_id` (*braket.aws.aws\_session.AwsSession* property), 40  
`active_trackers()` (*braket.tracking.tracking\_context.TrackingContext* method), 398  
`active_trackers()` (in module *braket.tracking.tracking\_context*), 398  
`add()` (*braket.ahs.atom\_arrangement.AtomArrangement* method), 12  
`add()` (*braket.circuits.circuit.Circuit* method), 77  
`add()` (*braket.circuits.moments.Moments* method), 247  
`add_braket_user_agent()` (*braket.aws.aws\_session.AwsSession* method), 40  
`add_circuit()` (*braket.circuits.circuit.Circuit* method), 71  
`add_custom_unitary()` (*braket.circuits.braket\_program\_context.BraketProgramContext* method), 67  
`add_gate_instruction()` (*braket.circuits.braket\_program\_context.BraketProgramContext* method), 66  
`add_instruction()` (*braket.circuits.circuit.Circuit* method), 71  
`add_kraus_instruction()` (*braket.circuits.braket\_program\_context.BraketProgramContext* method), 67  
`add_linear_term()` (*braket.annealing.problem.Problem* method), 20  
`add_linear_terms()` (*braket.annealing.problem.Problem* method), 21  
`add_measure()` (*braket.circuits.braket\_program\_context.BraketProgramContext* method), 67  
`add_noise()` (*braket.circuits.moments.Moments* method), 247  
`add_noise()` (*braket.circuits.noise\_model.noise\_model.NoiseModel* method), 55  
`add_noise_instruction()` (*braket.circuits.braket\_program\_context.BraketProgramContext* method), 67  
`add_phase_instruction()` (*braket.circuits.braket\_program\_context.BraketProgramContext* method), 66  
`add_quadratic_term()` (*braket.annealing.problem.Problem* method), 21  
`add_quadratic_terms()` (*braket.annealing.problem.Problem* method), 21  
`add_result()` (*braket.circuits.braket\_program\_context.BraketProgramContext* method), 67  
`add_result_type()` (*braket.circuits.circuit.Circuit* method), 70  
`add_verbatim_box()` (*braket.circuits.circuit.Circuit* method), 72  
`additional_metadata` (*braket.tasks.analog\_hamiltonian\_simulation\_quantum\_task\_result.AnalogHamiltonianSimulationQuantumTaskResult* attribute), 383  
`additional_metadata` (*braket.tasks.annealing\_quantum\_task\_result.AnnealingQuantumTaskResult* attribute), 384  
`additional_metadata` (*braket.tasks.gate\_model\_quantum\_task\_result.GateModelQuantumTaskResult* attribute), 387  
`additional_metadata` (*braket.tasks.photonic\_model\_quantum\_task\_result.PhotonicModelQuantumTaskResult* attribute), 390  
`adjoint()` (*braket.circuits.angled\_gate.AngledGate* method), 62  
`adjoint()` (*braket.circuits.angled\_gate.DoubleAngledGate* method), 63  
`adjoint()` (*braket.circuits.angled\_gate.TripleAngledGate* method), 65  
`adjoint()` (*braket.circuits.circuit.Circuit* method), 78  
`adjoint()` (*braket.circuits.gate.Gate* method), 112  
`adjoint()` (*braket.circuits.gate.Gate.CCNot* method), 113  
`adjoint()` (*braket.circuits.gate.Gate.CNot* method), 115  
`adjoint()` (*braket.circuits.gate.Gate.CSwap* method), 122  
`adjoint()` (*braket.circuits.gate.Gate.CV* method), 124  
`adjoint()` (*braket.circuits.gate.Gate.CY* method), 125  
`adjoint()` (*braket.circuits.gate.Gate.CZ* method), 126  
`adjoint()` (*braket.circuits.gate.Gate.ECR* method), 128

- `adjoint()` (*braket.circuits.gate.Gate.GPhase method*), 129
- `adjoint()` (*braket.circuits.gate.Gate.GPi method*), 131
- `adjoint()` (*braket.circuits.gate.Gate.GPi2 method*), 133
- `adjoint()` (*braket.circuits.gate.Gate.H method*), 134
- `adjoint()` (*braket.circuits.gate.Gate.I method*), 135
- `adjoint()` (*braket.circuits.gate.Gate.ISwap method*), 137
- `adjoint()` (*braket.circuits.gate.Gate.MS method*), 139
- `adjoint()` (*braket.circuits.gate.Gate.PRx method*), 140
- `adjoint()` (*braket.circuits.gate.Gate.S method*), 151
- `adjoint()` (*braket.circuits.gate.Gate.Si method*), 152
- `adjoint()` (*braket.circuits.gate.Gate.Swap method*), 154
- `adjoint()` (*braket.circuits.gate.Gate.T method*), 155
- `adjoint()` (*braket.circuits.gate.Gate.Ti method*), 157
- `adjoint()` (*braket.circuits.gate.Gate.U method*), 158
- `adjoint()` (*braket.circuits.gate.Gate.Unitary method*), 160
- `adjoint()` (*braket.circuits.gate.Gate.V method*), 161
- `adjoint()` (*braket.circuits.gate.Gate.Vi method*), 162
- `adjoint()` (*braket.circuits.gate.Gate.X method*), 164
- `adjoint()` (*braket.circuits.gate.Gate.Y method*), 168
- `adjoint()` (*braket.circuits.gate.Gate.Z method*), 171
- `adjoint()` (*braket.circuits.gates.CCNot method*), 227
- `adjoint()` (*braket.circuits.gates.CNot method*), 202
- `adjoint()` (*braket.circuits.gates.CSwap method*), 229
- `adjoint()` (*braket.circuits.gates.CV method*), 216
- `adjoint()` (*braket.circuits.gates.CY method*), 218
- `adjoint()` (*braket.circuits.gates.CZ method*), 219
- `adjoint()` (*braket.circuits.gates.ECR method*), 220
- `adjoint()` (*braket.circuits.gates.GPhase method*), 178
- `adjoint()` (*braket.circuits.gates.GPi method*), 231
- `adjoint()` (*braket.circuits.gates.GPi2 method*), 234
- `adjoint()` (*braket.circuits.gates.H method*), 175
- `adjoint()` (*braket.circuits.gates.I method*), 177
- `adjoint()` (*braket.circuits.gates.ISwap method*), 205
- `adjoint()` (*braket.circuits.gates.MS method*), 236
- `adjoint()` (*braket.circuits.gates.PRx method*), 233
- `adjoint()` (*braket.circuits.gates.S method*), 185
- `adjoint()` (*braket.circuits.gates.Si method*), 186
- `adjoint()` (*braket.circuits.gates.Swap method*), 203
- `adjoint()` (*braket.circuits.gates.T method*), 188
- `adjoint()` (*braket.circuits.gates.Ti method*), 189
- `adjoint()` (*braket.circuits.gates.U method*), 200
- `adjoint()` (*braket.circuits.gates.Unitary method*), 238
- `adjoint()` (*braket.circuits.gates.V method*), 190
- `adjoint()` (*braket.circuits.gates.Vi method*), 192
- `adjoint()` (*braket.circuits.gates.X method*), 180
- `adjoint()` (*braket.circuits.gates.Y method*), 182
- `adjoint()` (*braket.circuits.gates.Z method*), 183
- `adjoint()` (*braket.circuits.instruction.Instruction method*), 242
- `adjoint_gradient()` (*braket.circuits.circuit.Circuit method*), 80
- `adjoint_gradient()` (*braket.circuits.result\_type.ResultType.AdjointGradient static method*), 326
- `adjoint_gradient()` (*braket.circuits.result\_types.AdjointGradient static method*), 335
- `AdjointGradient` (class in *braket.circuits.result\_types*), 334
- `ALL` (*braket.circuits.noise\_model.criteria.CriteriaKeyResult attribute*), 48
- `Amazon` (*braket.devices.devices.Devices attribute*), 343
- `amplitude` (*braket.ahs.driving\_field.DrivingField property*), 14
- `Amplitude` (class in *braket.circuits.result\_types*), 335
- `amplitude()` (*braket.circuits.circuit.Circuit method*), 81
- `amplitude()` (*braket.circuits.result\_type.ResultType.Amplitude static method*), 327
- `amplitude()` (*braket.circuits.result\_types.Amplitude static method*), 336
- `amplitude_damping()` (*braket.circuits.circuit.Circuit method*), 81
- `amplitude_damping()` (*braket.circuits.noise.Noise.AmplitudeDamping static method*), 250
- `amplitude_damping()` (*braket.circuits.noises.AmplitudeDamping static method*), 300
- `AmplitudeDamping` (class in *braket.circuits.noises*), 298
- `amplitudes` (*braket.pulse.pulse\_sequence\_trace.PulseSequenceTrace attribute*), 374
- `AnalogHamiltonianSimulation` (class in *braket.ahs.analog\_hamiltonian\_simulation*), 11
- `AnalogHamiltonianSimulationQuantumTaskResult` (class in *braket.tasks.analog\_hamiltonian\_simulation\_quantum\_task\_result*), 382
- `AnalogHamiltonianSimulationShotStatus` (class in *braket.tasks.analog\_hamiltonian\_simulation\_quantum\_task\_result*), 382
- `angle` (*braket.circuits.angled\_gate.AngledGate property*), 62
- `angle_1` (*braket.circuits.angled\_gate.DoubleAngledGate property*), 63
- `angle_1` (*braket.circuits.angled\_gate.TripleAngledGate property*), 64
- `angle_2` (*braket.circuits.angled\_gate.DoubleAngledGate property*), 63
- `angle_2` (*braket.circuits.angled\_gate.TripleAngledGate property*), 64
- `angle_3` (*braket.circuits.angled\_gate.TripleAngledGate property*), 64
- `angled_ascii_characters()` (in module *braket.circuits.angled\_gate*), 65
- `AngledGate` (class in *braket.circuits.angled\_gate*), 62
- `AnnealingQuantumTaskResult` (class in *braket.tasks.annealing\_quantum\_task\_result*),



384

`applicable_key_types()`  
(`braket.circuits.noise_model.criteria.Criteria`  
`method`), 48

`applicable_key_types()`  
(`braket.circuits.noise_model.criteria.Criteria.GateCriteria`  
`method`), 49

`applicable_key_types()`  
(`braket.circuits.noise_model.criteria.Criteria.ObservableCriteria`  
`method`), 50

`applicable_key_types()`  
(`braket.circuits.noise_model.criteria.Criteria.QubitCriteria`  
`method`), 51

`applicable_key_types()`  
(`braket.circuits.noise_model.criteria.Criteria.UnitaryGateCriteria`  
`method`), 51

`applicable_key_types()`  
(`braket.circuits.noise_model.gate_criteria.GateCriteria`  
`method`), 53

`applicable_key_types()`  
(`braket.circuits.noise_model.observable_criteria.ObservableCriteria`  
`method`), 57

`applicable_key_types()`  
(`braket.circuits.noise_model.qubit_initialization_criteria.QubitInitializationCriteria`  
`method`), 58

`applicable_key_types()`  
(`braket.circuits.noise_model.unitary_gate_criteria.UnitaryGateCriteria`  
`method`), 60

`apply()` (`braket.circuits.noise_model.noise_model.NoiseModel`  
`method`), 56

`apply_gate_noise()` (`braket.circuits.circuit.Circuit`  
`method`), 74

`apply_initialization_noise()`  
(`braket.circuits.circuit.Circuit` `method`), 76

`apply_noise_to_gates()` (in `module`  
`braket.circuits.noise_helpers`), 281

`apply_noise_to_moments()` (in `module`  
`braket.circuits.noise_helpers`), 281

`apply_readout_noise()`  
(`braket.circuits.circuit.Circuit` `method`), 77

`ArbitraryWaveform` (class in `braket.pulse.waveforms`), 375

`arn` (`braket.aws.aws_device.AwsDevice` `property`), 25

`arn` (`braket.aws.aws_quantum_job.AwsQuantumJob`  
`property`), 30

`arn` (`braket.jobs.local.local_job.LocalQuantumJob` `property`), 347

`arn` (`braket.jobs.quantum_job.QuantumJob` `property`), 363

`as_int` (`braket.circuits.basis_state.BasisState` `property`), 66

`as_string` (`braket.circuits.basis_state.BasisState` `property`), 66

`as_tuple` (`braket.circuits.basis_state.BasisState` `property`), 66

`ascii_symbols` (`braket.circuits.compiler_directive.CompilerDirective`  
`property`), 110

`ascii_symbols` (`braket.circuits.gate.Gate` `property`), 112

`ascii_symbols` (`braket.circuits.instruction.Instruction`  
`property`), 243

`ascii_symbols` (`braket.circuits.measure.Measure` `property`), 244

`ascii_symbols` (`braket.circuits.observable.Observable.TensorProduct`  
`property`), 311

`ascii_symbols` (`braket.circuits.observable.StandardObservable`  
`property`), 314

`ascii_symbols` (`braket.circuitsobservables.TensorProduct`  
`property`), 317

`ascii_symbols` (`braket.circuits.quantum_operator.QuantumOperator`  
`property`), 321

`ascii_symbols` (`braket.circuits.result_type.ResultType`  
`property`), 324

`AsciiCircuitDiagram` (class in  
`braket.circuits.text_diagram_builders.ascii_circuit_diagram`), 61

`ast_to_qasm()` (in `module`  
`braket.pulse.transpiler.parser`), 370

`async_result()` (`braket.aws.aws_quantum_task.AwsQuantumTask`  
`method`), 36

`async_result()` (`braket.tasks.local_quantum_task.LocalQuantumTask`  
`method`), 389

`async_result()` (`braket.tasks.quantum_task.QuantumTask`  
`method`), 391

`AtomArrangement` (class in  
`braket.ahs.atom_arrangement`), 12

`AtomArrangementItem` (class in  
`braket.ahs.atom_arrangement`), 12

`aws_session` (`braket.aws.aws_device.AwsDevice` `property`), 25

`AwsDevice` (class in `braket.aws.aws_device`), 22

`AwsDeviceType` (class in `braket.aws.aws_device`), 21

`AwsQuantumJob` (class in `braket.aws.aws_quantum_job`), 28

`AwsQuantumJob.LogState` (class in  
`braket.aws.aws_quantum_job`), 28

`AwsQuantumTask` (class in  
`braket.aws.aws_quantum_task`), 33

`AwsQuantumTaskBatch` (class in  
`braket.aws.aws_quantum_task_batch`), 36

`AwsSession` (class in `braket.aws.aws_session`), 39

`AwsSession.S3DestinationFolder` (class in  
`braket.aws.aws_session`), 39

## B

`barrier()` (`braket.pulse.pulse_sequence.PulseSequence`  
`method`), 373

`BASE` (`braket.jobs.image_uris.Framework` `attribute`), 360

basis_rotation_gates (braket.circuits.observable.Observable property), 308	bind_values() (braket.circuits.angled_gate.TripleAngledGate method), 65
basis_rotation_gates (braket.circuits.observable.Observable.H property), 308	bind_values() (braket.circuits.gate.Gate.CPhaseShift method), 116
basis_rotation_gates (braket.circuits.observable.Observable.Hermitian property), 309	bind_values() (braket.circuits.gate.Gate.CPhaseShift00 method), 118
basis_rotation_gates (braket.circuits.observable.Observable.I property), 309	bind_values() (braket.circuits.gate.Gate.CPhaseShift01 method), 119
basis_rotation_gates (braket.circuits.observable.Observable.Sum property), 310	bind_values() (braket.circuits.gate.Gate.CPhaseShift10 method), 121
basis_rotation_gates (braket.circuits.observable.Observable.TensorProduct property), 311	bind_values() (braket.circuits.gate.Gate.GPhase method), 129
basis_rotation_gates (braket.circuits.observable.Observable.X property), 312	bind_values() (braket.circuits.gate.Gate.GPi method), 131
basis_rotation_gates (braket.circuits.observable.Observable.Y property), 312	bind_values() (braket.circuits.gate.Gate.GPi2 method), 133
basis_rotation_gates (braket.circuits.observable.Observable.Z property), 313	bind_values() (braket.circuits.gate.Gate.MS method), 139
basis_rotation_gates (braket.circuitsobservables.H property), 314	bind_values() (braket.circuits.gate.Gate.PhaseShift method), 144
basis_rotation_gates (braket.circuitsobservables.Hermitian property), 319	bind_values() (braket.circuits.gate.Gate.PRx method), 141
basis_rotation_gates (braket.circuitsobservables.I property), 315	bind_values() (braket.circuits.gate.Gate.PSwap method), 142
basis_rotation_gates (braket.circuitsobservables.Sum property), 319	bind_values() (braket.circuits.gate.Gate.PulseGate method), 145
basis_rotation_gates (braket.circuitsobservables.TensorProduct property), 317	bind_values() (braket.circuits.gate.Gate.Rx method), 147
basis_rotation_gates (braket.circuitsobservables.X property), 316	bind_values() (braket.circuits.gate.Gate.Ry method), 148
basis_rotation_gates (braket.circuitsobservables.Y property), 316	bind_values() (braket.circuits.gate.Gate.Rz method), 149
basis_rotation_gates (braket.circuitsobservables.Z property), 316	bind_values() (braket.circuits.gate.Gate.U method), 158
basis_rotation_instructions (braket.circuits.circuit.Circuit property), 69	bind_values() (braket.circuits.gate.Gate.XX method), 165
BasisState (class in braket.circuits.basis_state), 66	bind_values() (braket.circuits.gate.Gate.XY method), 167
bind_values() (braket.circuits.angled_gate.AngledGate method), 62	bind_values() (braket.circuits.gate.Gate.YY method), 170
bind_values() (braket.circuits.angled_gate.DoubleAngledGate method), 63	bind_values() (braket.circuits.gate.Gate.ZZ method), 173
	bind_values() (braket.circuits.gates.CPhaseShift method), 210
	bind_values() (braket.circuits.gates.CPhaseShift00 method), 212
	bind_values() (braket.circuits.gates.CPhaseShift01 method), 213
	bind_values() (braket.circuits.gates.CPhaseShift10 method), 215
	bind_values() (braket.circuits.gates.GPhase method), 179
	bind_values() (braket.circuits.gates.GPi method), 231
	bind_values() (braket.circuits.gates.GPi2 method),

235

`bind_values()` (*braket.circuits.gates.MS method*), 237

`bind_values()` (*braket.circuits.gates.PhaseShift method*), 199

`bind_values()` (*braket.circuits.gates.PRx method*), 233

`bind_values()` (*braket.circuits.gates.PSwap method*), 207

`bind_values()` (*braket.circuits.gates.PulseGate method*), 239

`bind_values()` (*braket.circuits.gates.Rx method*), 194

`bind_values()` (*braket.circuits.gates.Ry method*), 195

`bind_values()` (*braket.circuits.gates.Rz method*), 197

`bind_values()` (*braket.circuits.gates.U method*), 200

`bind_values()` (*braket.circuits.gates.XX method*), 222

`bind_values()` (*braket.circuits.gates.XY method*), 208

`bind_values()` (*braket.circuits.gates.YY method*), 224

`bind_values()` (*braket.circuits.gates.ZZ method*), 226

`bind_values()` (*braket.circuits.noise.DampingNoise method*), 279

`bind_values()` (*braket.circuits.noise.MultiQubitPauliNoise method*), 277

`bind_values()` (*braket.circuits.noise.Noise.AmplitudeDamping method*), 251

`bind_values()` (*braket.circuits.noise.Noise.BitFlip method*), 252

`bind_values()` (*braket.circuits.noise.Noise.Depolarizing method*), 255

`bind_values()` (*braket.circuits.noise.Noise.GeneralizedAmplitudeDamping method*), 258

`bind_values()` (*braket.circuits.noise.Noise.PauliChannel method*), 261

`bind_values()` (*braket.circuits.noise.Noise.PhaseDamping method*), 263

`bind_values()` (*braket.circuits.noise.Noise.PhaseFlip method*), 265

`bind_values()` (*braket.circuits.noise.Noise.TwoQubitDephasing method*), 267

`bind_values()` (*braket.circuits.noise.Noise.TwoQubitDepolarizing method*), 271

`bind_values()` (*braket.circuits.noise.Noise.TwoQubitPauliChannel method*), 274

`bind_values()` (*braket.circuits.noise.PauliNoise method*), 278

`bind_values()` (*braket.circuits.noise.SingleProbabilisticNoise method*), 275

`bind_values()` (*braket.circuits.noises.AmplitudeDamping method*), 300

`bind_values()` (*braket.circuits.noises.BitFlip method*), 284

`bind_values()` (*braket.circuits.noises.Depolarizing method*), 290

`bind_values()` (*braket.circuits.noises.GeneralizedAmplitudeDamping method*), 303

`bind_values()` (*braket.circuits.noises.PauliChannel method*), 288

`bind_values()` (*braket.circuits.noises.PhaseDamping method*), 305

`bind_values()` (*braket.circuits.noises.PhaseFlip method*), 286

`bind_values()` (*braket.circuits.noises.TwoQubitDephasing method*), 295

`bind_values()` (*braket.circuits.noises.TwoQubitDepolarizing method*), 293

`bind_values()` (*braket.circuits.noises.TwoQubitPauliChannel method*), 298

`bind_values()` (*braket.parametric.parameterizable.Parameterizable method*), 370

`bind_values()` (*braket.pulse.waveforms.ConstantWaveform method*), 376

`bind_values()` (*braket.pulse.waveforms.DragGaussianWaveform method*), 377

`bind_values()` (*braket.pulse.waveforms.GaussianWaveform method*), 377

`bit_flip()` (*braket.circuits.circuit.Circuit method*), 81

`bit_flip()` (*braket.circuits.noise.Noise.BitFlip static method*), 252

`bit_flip()` (*braket.circuits.noises.BitFlip static method*), 283

`BitFlip` (class in *braket.circuits.noises*), 282

**braket**

- module, 11
- braket.ahs**
  - module, 11
  - braket.ahs.analog\_hamiltonian\_simulation**
    - module, 11
  - braket.ahs.atom\_arrangement**
    - module, 12
  - braket.ahs.discretization\_types**
    - module, 13
  - braket.ahs.driving\_field**
    - module, 14
  - braket.ahs.field**
    - module, 16
  - braket.ahs.hamiltonian**
    - module, 16
  - braket.ahs.local\_detuning**
    - module, 17
  - braket.ahs.pattern**
    - module, 19
  - braket.ahs.shifting\_field**
    - module, 19
- braket.annealing**
  - module, 19
- braket.annealing.problem**
  - module, 19
- braket.aws**
  - braket.aws.aws\_device**
    - module, 21

module, 21  
braket.aws.aws\_quantum\_job  
    module, 28  
braket.aws.aws\_quantum\_task  
    module, 33  
braket.aws.aws\_quantum\_task\_batch  
    module, 36  
braket.aws.aws\_session  
    module, 39  
braket.aws.queue\_information  
    module, 45  
braket.circuits  
    module, 47  
braket.circuits.angled\_gate  
    module, 62  
braket.circuits.ascii\_circuit\_diagram  
    module, 66  
braket.circuits.basis\_state  
    module, 66  
braket.circuits.braket\_program\_context  
    module, 66  
braket.circuits.circuit  
    module, 68  
braket.circuits.circuit\_diagram  
    module, 109  
braket.circuits.circuit\_helpers  
    module, 109  
braket.circuits.compiler\_directive  
    module, 110  
braket.circuits.compiler\_directives  
    module, 111  
braket.circuits.free\_parameter  
    module, 111  
braket.circuits.free\_parameter\_expression  
    module, 111  
braket.circuits.gate  
    module, 111  
braket.circuits.gate\_calibrations  
    module, 174  
braket.circuits.gates  
    module, 175  
braket.circuits.instruction  
    module, 241  
braket.circuits.measure  
    module, 244  
braket.circuits.moments  
    module, 245  
braket.circuits.noise  
    module, 248  
braket.circuits.noise\_helpers  
    module, 280  
braket.circuits.noise\_model  
    module, 47  
braket.circuits.noise\_model.circuit\_instruction\_criteria  
    module, 47  
braket.circuits.noise\_model.criteria  
    module, 47  
braket.circuits.noise\_model.criteria\_input\_parsing  
    module, 52  
braket.circuits.noise\_model.gate\_criteria  
    module, 53  
braket.circuits.noise\_model.initialization\_criteria  
    module, 54  
braket.circuits.noise\_model.noise\_model  
    module, 54  
braket.circuits.noise\_model.observable\_criteria  
    module, 57  
braket.circuits.noise\_model.qubit\_initialization\_criteria  
    module, 58  
braket.circuits.noise\_model.result\_type\_criteria  
    module, 59  
braket.circuits.noise\_model.unitary\_gate\_criteria  
    module, 60  
braket.circuits.noises  
    module, 282  
braket.circuits.observable  
    module, 307  
braket.circuits.observables  
    module, 314  
braket.circuits.operator  
    module, 320  
braket.circuits.parameterizable  
    module, 321  
braket.circuits.quantum\_operator  
    module, 321  
braket.circuits.quantum\_operator\_helpers  
    module, 322  
braket.circuits.qubit  
    module, 324  
braket.circuits.qubit\_set  
    module, 324  
braket.circuits.result\_type  
    module, 324  
braket.circuits.result\_types  
    module, 333  
braket.circuits.serialization  
    module, 340  
braket.circuits.text\_diagram\_builders  
    module, 61  
braket.circuits.text\_diagram\_builders.ascii\_circuit\_diagram  
    module, 61  
braket.circuits.text\_diagram\_builders.text\_circuit\_diagram  
    module, 61  
braket.circuits.text\_diagram\_builders.text\_circuit\_diagram  
    module, 61  
braket.circuits.text\_diagram\_builders.unicode\_circuit\_diagram  
    module, 61  
braket.circuits.translations

---

module, 341	module, 353
braket.circuits.unitary_calculation	braket.jobs.quantum_job
module, 341	module, 363
braket.devices	braket.jobs.quantum_job_creation
module, 342	module, 365
braket.devices.device	braket.jobs.serialization
module, 342	module, 367
braket.devices.devices	braket.parametric
module, 343	module, 368
braket.devices.local_simulator	braket.parametric.free_parameter
module, 343	module, 368
braket.error_mitigation	braket.parametric.free_parameter_expression
module, 345	module, 369
braket.error_mitigation.debias	braket.parametric.parameterizable
module, 345	module, 370
braket.error_mitigation.error_mitigation	braket.pulse
module, 346	module, 370
braket.ipython_utils	braket.pulse.ast
module, 399	module, 370
braket.jobs	braket.pulse.ast.approximation_parser
module, 346	module, 370
braket.jobs.config	braket.pulse.ast.free_parameters
module, 355	module, 370
braket.jobs.data_persistence	braket.pulse.ast.qasm_parser
module, 356	module, 370
braket.jobs.environment_variables	braket.pulse.ast.qasm_transformer
module, 358	module, 371
braket.jobs.hybrid_job	braket.pulse.frame
module, 358	module, 371
braket.jobs.image_uris	braket.pulse.port
module, 360	module, 371
braket.jobs.local	braket.pulse.pulse_sequence
module, 346	module, 372
braket.jobs.local.local_job	braket.pulse.pulse_sequence_trace
module, 346	module, 374
braket.jobs.local.local_job_container	braket.pulse.waveforms
module, 350	module, 375
braket.jobs.local.local_job_container_setup	braket.quantum_information
module, 350	module, 378
braket.jobs.logs	braket.quantum_information.pauli_string
module, 361	module, 378
braket.jobs.metrics	braket.registers
module, 363	module, 380
braket.jobs.metrics_data	braket.registers.qubit
module, 350	module, 380
braket.jobs.metrics_data.cwl_insights_metrics_fetcher	braket.registers.qubit_set
module, 350	module, 381
braket.jobs.metrics_data.cwl_metrics_fetcher	braket.tasks
module, 352	module, 382
braket.jobs.metrics_data.definitions	braket.tasks.analog_hamiltonian_simulation_quantum_task_result
module, 353	module, 382
braket.jobs.metrics_data.exceptions	braket.tasks.annealing_quantum_task_result
module, 353	module, 384
braket.jobs.metrics_data.log_metrics_parser	braket.tasks.gate_model_quantum_task_result



module, 385  
 braket.tasks.local\_quantum\_task  
   module, 389  
 braket.tasks.local\_quantum\_task\_batch  
   module, 390  
 braket.tasks.photonic\_model\_quantum\_task\_result  
   module, 390  
 braket.tasks.quantum\_task  
   module, 391  
 braket.tasks.quantum\_task\_batch  
   module, 392  
 braket.timings  
   module, 392  
 braket.timings.time\_series  
   module, 392  
 braket.tracking  
   module, 396  
 braket.tracking.pricing  
   module, 396  
 braket.tracking.tracker  
   module, 396  
 braket.tracking.tracking\_context  
   module, 398  
 braket.tracking.tracking\_events  
   module, 399  
 braket\_result\_to\_result\_type() (in module  
   braket.circuits.translations), 341  
 BracketProgramContext (class in  
   braket.circuits.braket\_program\_context),  
   66  
 broadcast\_event() (braket.tracking.tracking\_context.TrackingContext  
   method), 398  
 broadcast\_event() (in module  
   braket.tracking.tracking\_context), 398  
 bucket (braket.aws.aws\_session.AwsSession.S3DestinationFolder  
   attribute), 39  
 build\_diagram() (braket.circuits.circuit\_diagram.CircuitDiagram  
   static method), 109  
 build\_diagram() (braket.circuits.text\_diagram\_builders.as\_text\_diagram\_builder  
   static method), 61  
 build\_diagram() (braket.circuits.text\_diagram\_builders.unrolled\_circuit\_diagram\_builder  
   static method), 61  
 built\_in\_images() (in module  
   braket.jobs.image\_uris), 360  
**C**  
 calculate\_unitary\_big\_endian() (in module  
   braket.circuits.unitary\_calculation), 341  
 cancel() (braket.aws.aws\_quantum\_job.AwsQuantumJob  
   method), 32  
 cancel() (braket.aws.aws\_quantum\_task.AwsQuantumTask  
   method), 35  
 cancel() (braket.jobs.local.local\_job.LocalQuantumJob  
   method), 348  
 cancel() (braket.jobs.quantum\_job.QuantumJob  
   method), 364  
 cancel() (braket.tasks.local\_quantum\_task.LocalQuantumTask  
   method), 389  
 cancel() (braket.tasks.quantum\_task.QuantumTask  
   method), 391  
 cancel\_job() (braket.aws.aws\_session.AwsSession  
   method), 41  
 cancel\_quantum\_task()  
   (braket.aws.aws\_session.AwsSession method),  
   40  
 capture\_v0() (braket.pulse.pulse\_sequence.PulseSequence  
   method), 374  
 cast\_result\_types()  
   (braket.tasks.gate\_model\_quantum\_task\_result.GateModelQuantumTaskResult  
   static method), 389  
 CCNot (class in braket.circuits.gates), 227  
 ccnot() (braket.circuits.circuit.Circuit method), 82  
 ccnot() (braket.circuits.gate.Gate.CCNot static  
   method), 113  
 ccnot() (braket.circuits.gates.CCNot static method), 228  
 check\_noise\_target\_gates() (in module  
   braket.circuits.noise\_helpers), 280  
 check\_noise\_target\_qubits() (in module  
   braket.circuits.noise\_helpers), 281  
 check\_noise\_target\_unitary() (in module  
   braket.circuits.noise\_helpers), 281  
 CheckpointConfig (class in braket.jobs.config), 355  
 circuit (braket.circuits.braket\_program\_context.BraketProgramContext  
   property), 66  
 Circuit (class in braket.circuits.circuit), 68  
 CircuitDiagram (class in  
   braket.circuits.circuit\_diagram), 109  
 CircuitInstructionCriteria (class in  
   braket.circuits.noise\_model.circuit\_instruction\_criteria),  
   47  
 CNOT (class in braket.circuits.gates), 201  
 cnot() (braket.circuits.circuit.Circuit method), 82  
 cnot() (braket.circuits.gate.Gate.CNOT static method),  
   115  
 cnot() (braket.circuits.gates.CNOT static method), 202  
 coefficient (braket.circuits.observable.Observable  
   property), 307  
 ColorWrap (class in braket.jobs.logs), 361  
 COMPILER\_DIRECTIVE (braket.circuits.moments.MomentType  
   attribute), 245  
 CompilerDirective (class in  
   braket.circuits.compiler\_directive), 110  
 COMPLETE (braket.aws.aws\_quantum\_job.AwsQuantumJob.LogState  
   attribute), 28  
 concatenate() (braket.timings.time\_series.TimeSeries  
   method), 393  
 config (braket.jobs.config.S3DataSourceConfig at-  
   tribute), 356

`constant_like()` (*braket.timings.time\_series.TimeSeries* static method), 393  
`ConstantWaveform` (class in *braket.pulse.waveforms*), 375  
`construct_s3_uri()` (*braket.aws.aws\_session.AwsSession* static method), 44  
`control` (*braket.circuits.instruction.Instruction* property), 242  
`control_state` (*braket.circuits.instruction.Instruction* property), 242  
`coordinate` (*braket.ahs.atom\_arrangement.AtomArrangement* attribute), 12  
`coordinate_list()` (*braket.ahs.atom\_arrangement.AtomArrangement* method), 13  
`copy()` (*braket.circuits.circuit.Circuit* method), 80  
`copy()` (*braket.circuits.gate\_calibrations.GateCalibrations* method), 174  
`copy()` (*braket.circuits.instruction.Instruction* method), 243  
`copy()` (*braket.circuits.result\_type.ResultType* method), 324  
`copy_s3_directory()` (*braket.aws.aws\_session.AwsSession* method), 42  
`copy_s3_object()` (*braket.aws.aws\_session.AwsSession* method), 42  
`copy_session()` (*braket.aws.aws\_session.AwsSession* method), 45  
`counterpart()` (*braket.circuits.compiler\_directive.CompilerDirective* method), 110  
`counterpart()` (*braket.circuits.compiler\_directives.EndVerbatimBox* method), 111  
`counterpart()` (*braket.circuits.compiler\_directives.StartVerbatimBox* method), 111  
`CPhaseShift` (class in *braket.circuits.gates*), 209  
`cphaseshift()` (*braket.circuits.circuit.Circuit* method), 83  
`cphaseshift()` (*braket.circuits.gate.Gate.CPhaseShift* static method), 116  
`cphaseshift()` (*braket.circuits.gates.CPhaseShift* static method), 210  
`CPhaseShift00` (class in *braket.circuits.gates*), 211  
`cphaseshift00()` (*braket.circuits.circuit.Circuit* method), 83  
`cphaseshift00()` (*braket.circuits.gate.Gate.CPhaseShift00* static method), 118  
`cphaseshift00()` (*braket.circuits.gates.CPhaseShift00* static method), 212  
`CPhaseShift01` (class in *braket.circuits.gates*), 213  
`cphaseshift01()` (*braket.circuits.circuit.Circuit* method), 84  
`cphaseshift01()` (*braket.circuits.gate.Gate.CPhaseShift01* static method), 119  
`cphaseshift01()` (*braket.circuits.gates.CPhaseShift01* static method), 214  
`CPhaseShift10` (class in *braket.circuits.gates*), 214  
`cphaseshift10()` (*braket.circuits.circuit.Circuit* method), 84  
`cphaseshift10()` (*braket.circuits.gate.Gate.CPhaseShift10* static method), 121  
`cphaseshift10()` (*braket.circuits.gates.CPhaseShift10* static method), 215  
`create()` (*braket.aws.aws\_quantum\_job.AwsQuantumJob* class method), 28  
`create()` (*braket.aws.aws\_quantum\_task.AwsQuantumTask* static method), 34  
`create()` (*braket.jobs.local.local\_job.LocalQuantumJob* class method), 346  
`create_job()` (*braket.aws.aws\_session.AwsSession* method), 41  
`create_quantum_task()` (*braket.aws.aws\_session.AwsSession* method), 40  
`criteria` (*braket.circuits.noise\_model.noise\_model.NoiseModelInstruction* attribute), 54  
`Criteria` (class in *braket.circuits.noise\_model.criteria*), 48  
`Criteria.GateCriteria` (class in *braket.circuits.noise\_model.criteria*), 48  
`Criteria.ObservableCriteria` (class in *braket.circuits.noise\_model.criteria*), 49  
`Criteria.QubitInitializationCriteria` (class in *braket.circuits.noise\_model.criteria*), 50  
`Criteria.UnitaryGateCriteria` (class in *braket.circuits.noise\_model.criteria*), 51  
`CriteriaKey` (class in *braket.circuits.noise\_model.criteria*), 47  
`CriteriaKeyResult` (class in *braket.circuits.noise\_model.criteria*), 47  
`CSwap` (class in *braket.circuits.gates*), 228  
`cswap()` (*braket.circuits.circuit.Circuit* method), 85  
`cswap()` (*braket.circuits.gate.Gate.CSwap* static method), 122  
`cswap()` (*braket.circuits.gates.CSwap* static method), 229  
`CV` (class in *braket.circuits.gates*), 216  
`cv()` (*braket.circuits.circuit.Circuit* method), 85  
`cv()` (*braket.circuits.gate.Gate.CV* static method), 124  
`cv()` (*braket.circuits.gates.CV* static method), 217  
`CwlInsightsMetricsFetcher` (class in *braket.jobs.metrics\_data.cwl\_insights\_metrics\_fetcher*), 350  
`CwlMetricsFetcher` (class in *braket.jobs.metrics\_data.cwl\_metrics\_fetcher*), 352  
`CY` (class in *braket.circuits.gates*), 217  
`cy()` (*braket.circuits.circuit.Circuit* method), 86  
`cy()` (*braket.circuits.gate.Gate.CY* static method), 125

cy() (*braket.circuits.gates.CY static method*), 218  
 CZ (*class in braket.circuits.gates*), 219  
 cz() (*braket.circuits.circuit.Circuit method*), 86  
 cz() (*braket.circuits.gate.Gate.CZ static method*), 126  
 cz() (*braket.circuits.gates.CZ static method*), 220

## D

DampingNoise (*class in braket.circuits.noise*), 278  
 data() (*braket.tasks.annealing\_quantum\_task\_result.AnnealingQuantumTaskResult method*), 384  
 Debias (*class in braket.error\_mitigation.debias*), 345  
 default\_bucket() (*braket.aws.aws\_session.AwsSession method*), 43  
 DEFAULT\_MAX\_PARALLEL (*braket.aws.aws\_device.AwsDevice attribute*), 22  
 DEFAULT\_RESULTS\_POLL\_INTERVAL (*braket.aws.aws\_quantum\_task.AwsQuantumTask attribute*), 33  
 DEFAULT\_RESULTS\_POLL\_INTERVAL (*braket.jobs.quantum\_job.QuantumJob attribute*), 363  
 DEFAULT\_RESULTS\_POLL\_TIMEOUT (*braket.aws.aws\_quantum\_task.AwsQuantumTask attribute*), 33  
 DEFAULT\_RESULTS\_POLL\_TIMEOUT (*braket.jobs.quantum\_job.QuantumJob attribute*), 363  
 DEFAULT\_SHOTS\_QPU (*braket.aws.aws\_device.AwsDevice attribute*), 22  
 DEFAULT\_SHOTS\_SIMULATOR (*braket.aws.aws\_device.AwsDevice attribute*), 22  
 delay() (*braket.pulse.pulse\_sequence.PulseSequence method*), 373  
 density\_matrix() (*braket.circuits.circuit.Circuit method*), 87  
 density\_matrix() (*braket.circuits.result\_type.ResultType static method*), 327  
 density\_matrix() (*braket.circuits.result\_types.DensityMatrix static method*), 334  
 DensityMatrix (*class in braket.circuits.result\_types*), 333  
 Depolarizing (*class in braket.circuits.noises*), 288  
 depolarizing() (*braket.circuits.circuit.Circuit method*), 87  
 depolarizing() (*braket.circuits.noise.Noise.Depolarizing static method*), 255  
 depolarizing() (*braket.circuits.noises.Depolarizing static method*), 290  
 depth (*braket.circuits.circuit.Circuit property*), 69  
 depth (*braket.circuits.moments.Moments property*), 246  
 deregister\_tracker() (*braket.tracking.tracking\_context.TrackingContext method*), 398  
 deregister\_tracker() (*in module braket.tracking.tracking\_context*), 398  
 describe\_log\_streams() (*braket.aws.aws\_session.AwsSession method*), 44  
 deserialize\_values() (*in module braket.jobs.serialization*), 367  
 detuning (*braket.ahs.driving\_field.DrivingField property*), 15  
 device (*braket.jobs.config.DeviceConfig attribute*), 355  
 Device (*class in braket.devices.device*), 342  
 DeviceConfig (*class in braket.jobs.config*), 355  
 Devices (*class in braket.devices.devices*), 343  
 diagram() (*braket.circuits.circuit.Circuit method*), 78  
 DiscretizationError, 13  
 DiscretizationProperties (*class in braket.ahs.discretization\_types*), 13  
 discretize() (*braket.ahs.analog\_hamiltonian\_simulation.AnalogHamiltonian method*), 12  
 discretize() (*braket.ahs.atom\_arrangement.AtomArrangement method*), 13  
 discretize() (*braket.ahs.driving\_field.DrivingField method*), 15  
 discretize() (*braket.ahs.field.Field method*), 16  
 discretize() (*braket.ahs.hamiltonian.Hamiltonian method*), 17  
 discretize() (*braket.ahs.local\_detuning.LocalDetuning method*), 19  
 discretize() (*braket.ahs.pattern.Pattern method*), 19  
 discretize() (*braket.timings.time\_series.TimeSeries method*), 395  
 dot() (*braket.quantum\_information.pauli\_string.PauliString method*), 379  
 DoubleAngledGate (*class in braket.circuits.angled\_gate*), 62  
 download\_from\_s3() (*braket.aws.aws\_session.AwsSession method*), 42  
 download\_result() (*braket.aws.aws\_quantum\_job.AwsQuantumJob method*), 32  
 download\_result() (*braket.jobs.local.local\_job.LocalQuantumJob method*), 348  
 download\_result() (*braket.jobs.quantum\_job.QuantumJob method*), 365  
 DragGaussianWaveform (*class in braket.pulse.waveforms*), 376  
 DRIVING\_FIELDS\_PROPERTY (*braket.ahs.analog\_hamiltonian\_simulation.AnalogHamiltonianS attribute*), 11  
 DrivingField (*class in braket.ahs.driving\_field*), 14  
 dt (*braket.pulse.port.Port property*), 372

## E

ECR (*class in braket.circuits.gates*), 220



ecr() (*braket.circuits.circuit.Circuit method*), 87  
 ecr() (*braket.circuits.gate.Gate.ECR static method*), 128  
 ecr() (*braket.circuits.gates.ECR static method*), 221  
 ecr\_client (*braket.aws.aws\_session.AwsSession property*), 40  
 eigenstate() (*braket.quantum\_information.pauli\_string.PauliString method*), 379  
 eigenvalue() (*braket.circuits.observable.Observable method*), 308  
 eigenvalue() (*braket.circuits.observable.Observable.Hermitian method*), 309  
 eigenvalue() (*braket.circuits.observable.Observable.I method*), 309  
 eigenvalue() (*braket.circuits.observable.Observable.Sum method*), 310  
 eigenvalue() (*braket.circuits.observable.Observable.TensorProduct method*), 312  
 eigenvalue() (*braket.circuits.observable.StandardObservable method*), 314  
 eigenvalue() (*braket.circuitsobservables.Hermitian method*), 320  
 eigenvalue() (*braket.circuitsobservables.I method*), 315  
 eigenvalue() (*braket.circuitsobservables.Sum method*), 319  
 eigenvalue() (*braket.circuitsobservables.TensorProduct method*), 318  
 eigenvalues (*braket.circuits.observable.Observable property*), 308  
 eigenvalues (*braket.circuits.observable.Observable.Hermitian property*), 309  
 eigenvalues (*braket.circuits.observable.Observable.I property*), 310  
 eigenvalues (*braket.circuits.observable.Observable.Sum property*), 310  
 eigenvalues (*braket.circuits.observable.Observable.TensorProduct property*), 312  
 eigenvalues (*braket.circuits.observable.StandardObservable property*), 314  
 eigenvalues (*braket.circuitsobservables.Hermitian property*), 320  
 eigenvalues (*braket.circuitsobservables.I property*), 315  
 eigenvalues (*braket.circuitsobservables.Sum property*), 319  
 eigenvalues (*braket.circuitsobservables.TensorProduct property*), 318  
 EndVerbatimBox (class in *braket.circuits.compiler\_directives*), 111  
 ErrorMitigation (class in *braket.error\_mitigation.error\_mitigation*), 346  
 Expectation (class in *braket.circuits.result\_types*), 337  
 expectation() (*braket.circuits.circuit.Circuit method*), 88  
 expectation() (*braket.circuits.result\_type.ResultType.Expectation static method*), 328  
 expectation() (*braket.circuits.result\_types.Expectation static method*), 337  
 FreeParameterExpression (*braket.parametric.free\_parameter\_expression.FreeParameterExpression property*), 369

## F

factors (*braket.circuits.observable.Observable.TensorProduct property*), 312  
 factors (*braket.circuitsobservables.TensorProduct property*), 317  
 FAILURE (*braket.tasks.analog\_hamiltonian\_simulation\_quantum\_task\_result.Failure attribute*), 382  
 Field (class in *braket.ahs.field*), 16  
 FILLED (*braket.ahs.atom\_arrangement.SiteType attribute*), 12  
 filter() (*braket.circuits.gate\_calibrations.GateCalibrations method*), 174  
 fixed\_qubit\_count() (*braket.circuits.gate.Gate.CCNot static method*), 114  
 fixed\_qubit\_count() (*braket.circuits.gate.Gate.CNot static method*), 115  
 fixed\_qubit\_count() (*braket.circuits.gate.Gate.CPhaseShift static method*), 117  
 fixed\_qubit\_count() (*braket.circuits.gate.Gate.CPhaseShift00 static method*), 118  
 fixed\_qubit\_count() (*braket.circuits.gate.Gate.CPhaseShift01 static method*), 120  
 fixed\_qubit\_count() (*braket.circuits.gate.Gate.CPhaseShift10 static method*), 121  
 fixed\_qubit\_count() (*braket.circuits.gate.Gate.CSwap static method*), 123  
 fixed\_qubit\_count() (*braket.circuits.gate.Gate.CV static method*), 124  
 fixed\_qubit\_count() (*braket.circuits.gate.Gate.CY static method*), 126  
 fixed\_qubit\_count() (*braket.circuits.gate.Gate.CZ static method*), 127  
 fixed\_qubit\_count() (*braket.circuits.gate.Gate.ECR static method*), 128  
 fixed\_qubit\_count() (*braket.circuits.gate.Gate.GPhase static method*), 129  
 fixed\_qubit\_count() (*braket.circuits.gate.Gate.GPi static method*), 131

`fixed_qubit_count()` (*braket.circuits.gate.Gate.GPi2 static method*), 133  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.H static method*), 134  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.I static method*), 136  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.ISwap static method*), 137  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.MS static method*), 139  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.PhaseShift static method*), 144  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.PRx static method*), 141  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.PSwap static method*), 142  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.Rx static method*), 147  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.Ry static method*), 148  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.Rz static method*), 149  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.S static method*), 151  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.Si static method*), 152  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.Swap static method*), 154  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.T static method*), 155  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.Ti static method*), 157  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.U static method*), 158  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.V static method*), 161  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.Vi static method*), 163  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.X static method*), 164  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.XX static method*), 165  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.XY static method*), 167  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.Y static method*), 168  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.YY static method*), 170  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.Z static method*), 171  
`fixed_qubit_count()` (*braket.circuits.gate.Gate.ZZ static method*), 173  
`fixed_qubit_count()` (*braket.circuits.gates.CCNot static method*), 227  
`fixed_qubit_count()` (*braket.circuits.gates.CNot static method*), 202  
`fixed_qubit_count()` (*braket.circuits.gates.CPhaseShift static method*), 210  
`fixed_qubit_count()` (*braket.circuits.gates.CPhaseShift00 static method*), 212  
`fixed_qubit_count()` (*braket.circuits.gates.CPhaseShift01 static method*), 214  
`fixed_qubit_count()` (*braket.circuits.gates.CPhaseShift10 static method*), 215  
`fixed_qubit_count()` (*braket.circuits.gates.CSwap static method*), 229  
`fixed_qubit_count()` (*braket.circuits.gates.CV static method*), 217  
`fixed_qubit_count()` (*braket.circuits.gates.CY static method*), 218  
`fixed_qubit_count()` (*braket.circuits.gates.CZ static method*), 219  
`fixed_qubit_count()` (*braket.circuits.gates.ECR static method*), 221  
`fixed_qubit_count()` (*braket.circuits.gates.GPhase static method*), 179  
`fixed_qubit_count()` (*braket.circuits.gates.GPi static method*), 231  
`fixed_qubit_count()` (*braket.circuits.gates.GPi2 static method*), 234  
`fixed_qubit_count()` (*braket.circuits.gates.H static method*), 175  
`fixed_qubit_count()` (*braket.circuits.gates.I static method*), 177  
`fixed_qubit_count()` (*braket.circuits.gates.ISwap static method*), 205  
`fixed_qubit_count()` (*braket.circuits.gates.MS static method*), 237  
`fixed_qubit_count()` (*braket.circuits.gates.PhaseShift static method*), 199  
`fixed_qubit_count()` (*braket.circuits.gates.PRx static method*), 233  
`fixed_qubit_count()` (*braket.circuits.gates.PSwap static method*), 207  
`fixed_qubit_count()` (*braket.circuits.gates.Rx static method*), 194  
`fixed_qubit_count()` (*braket.circuits.gates.Ry static method*), 195  
`fixed_qubit_count()` (*braket.circuits.gates.Rz static method*), 197

`fixed_qubit_count()` (`braket.circuits.gates.S` static method), 185  
`fixed_qubit_count()` (`braket.circuits.gates.Si` static method), 186  
`fixed_qubit_count()` (`braket.circuits.gates.Swap` static method), 204  
`fixed_qubit_count()` (`braket.circuits.gates.T` static method), 188  
`fixed_qubit_count()` (`braket.circuits.gates.Ti` static method), 189  
`fixed_qubit_count()` (`braket.circuits.gates.U` static method), 200  
`fixed_qubit_count()` (`braket.circuits.gates.V` static method), 191  
`fixed_qubit_count()` (`braket.circuits.gates.Vi` static method), 192  
`fixed_qubit_count()` (`braket.circuits.gates.X` static method), 180  
`fixed_qubit_count()` (`braket.circuits.gates.XX` static method), 222  
`fixed_qubit_count()` (`braket.circuits.gates.XY` static method), 208  
`fixed_qubit_count()` (`braket.circuits.gates.Y` static method), 182  
`fixed_qubit_count()` (`braket.circuits.gates.YY` static method), 224  
`fixed_qubit_count()` (`braket.circuits.gates.Z` static method), 184  
`fixed_qubit_count()` (`braket.circuits.gates.ZZ` static method), 226  
`fixed_qubit_count()` (`braket.circuits.noise.Noise.AmplitudeDamping` static method), 251  
`fixed_qubit_count()` (`braket.circuits.noise.Noise.BitFlip` static method), 253  
`fixed_qubit_count()` (`braket.circuits.noise.Noise.Depolarizing` static method), 255  
`fixed_qubit_count()` (`braket.circuits.noise.Noise.GeneralizedAmplitudeDamping` static method), 258  
`fixed_qubit_count()` (`braket.circuits.noise.Noise.PauliChannel` static method), 261  
`fixed_qubit_count()` (`braket.circuits.noise.Noise.PhaseDamping` static method), 263  
`fixed_qubit_count()` (`braket.circuits.noise.Noise.PhaseFlip` static method), 265  
`fixed_qubit_count()` (`braket.circuits.noise.Noise.TwoQubitDephasing` static method), 267  
`fixed_qubit_count()` (`braket.circuits.noise.Noise.TwoQubitDepolarizing` static method), 271  
`fixed_qubit_count()` (`braket.circuits.noise.Noise.TwoQubitPauliChannel` static method), 274  
`fixed_qubit_count()` (`braket.circuits.noises.AmplitudeDamping` static method), 300  
`fixed_qubit_count()` (`braket.circuits.noises.BitFlip` static method), 283  
`fixed_qubit_count()` (`braket.circuits.noises.Depolarizing` static method), 290  
`fixed_qubit_count()` (`braket.circuits.noises.GeneralizedAmplitudeDamping` static method), 303  
`fixed_qubit_count()` (`braket.circuits.noises.PauliChannel` static method), 287  
`fixed_qubit_count()` (`braket.circuits.noises.PhaseDamping` static method), 305  
`fixed_qubit_count()` (`braket.circuits.noises.PhaseFlip` static method), 285  
`fixed_qubit_count()` (`braket.circuits.noises.TwoQubitDephasing` static method), 295  
`fixed_qubit_count()` (`braket.circuits.noises.TwoQubitDepolarizing` static method), 293  
`fixed_qubit_count()` (`braket.circuits.noises.TwoQubitPauliChannel` static method), 298  
`fixed_qubit_count()` (`braket.circuits.quantum_operator.QuantumOperator` static method), 321  
`flush_log_streams()` (in module `braket.jobs.logs`), 362  
`format_complex()` (in module `braket.circuits.gates`), 241  
`format_target()` (`braket.circuits.serialization.OpenQASMSerializationP` method), 340  
`Frame` (class in `braket.pulse.frame`), 371  
`frames` (`braket.aws.aws_device.AwsDevice` property), 26  
`Framework` (class in `braket.jobs.image_uris`), 360  
`FreeParameter` (class in `braket.parametric.free_parameter`), 368  
`FreeParameterExpression` (class in `braket.parametric.free_parameter_expression`), 369  
`frequencies` (`braket.pulse.pulse_sequence_trace.PulseSequenceTrace` attribute), 374

`from_dict()` (`braket.circuits.noise.Noise` class method), 249  
`from_dict()` (`braket.circuits.noise.Noise.AmplitudeDamping` class method), 251  
`from_dict()` (`braket.circuits.noise.Noise.BitFlip` class method), 253  
`from_dict()` (`braket.circuits.noise.Noise.Depolarizing` class method), 255  
`from_dict()` (`braket.circuits.noise.Noise.GeneralizedAmplitudeDamping` class method), 258  
`from_dict()` (`braket.circuits.noise.Noise.Kraus` class method), 259  
`from_dict()` (`braket.circuits.noise.Noise.PauliChannel` class method), 262  
`from_dict()` (`braket.circuits.noise.Noise.PhaseDamping` class method), 263  
`from_dict()` (`braket.circuits.noise.Noise.PhaseFlip` class method), 266  
`from_dict()` (`braket.circuits.noise.Noise.TwoQubitDephasing` class method), 267  
`from_dict()` (`braket.circuits.noise.Noise.TwoQubitDepolarizing` class method), 271  
`from_dict()` (`braket.circuits.noise.Noise.TwoQubitPauliChannel` class method), 274  
`from_dict()` (`braket.circuits.noise_model.criteria.Criteria` class method), 48  
`from_dict()` (`braket.circuits.noise_model.criteria.CriteriaFromObject` class method), 49  
`from_dict()` (`braket.circuits.noise_model.criteria.CriteriaFromObservable` class method), 50  
`from_dict()` (`braket.circuits.noise_model.criteria.CriteriaFromQubitList` class method), 51  
`from_dict()` (`braket.circuits.noise_model.criteria.CriteriaFromQubitListAndCriteria` class method), 51  
`from_dict()` (`braket.circuits.noise_model.gate_criteria.GateCriteria` class method), 54  
`from_dict()` (`braket.circuits.noise_model.noise_model.NoiseModel` class method), 57  
`from_dict()` (`braket.circuits.noise_model.noise_model.NoiseModelFromList` class method), 55  
`from_dict()` (`braket.circuits.noise_model.observable_criteria.ObservableCriteria` class method), 58  
`from_dict()` (`braket.circuits.noise_model.qubit_initialization_criteria.QubitInitializationCriteria` class method), 59  
`from_dict()` (`braket.circuits.noise_model.unitary_gate_criteria.UnitaryGateCriteria` class method), 60  
`from_dict()` (`braket.circuits.noises.AmplitudeDamping` class method), 300  
`from_dict()` (`braket.circuits.noises.BitFlip` class method), 284  
`from_dict()` (`braket.circuits.noises.Depolarizing` class method), 290  
`from_dict()` (`braket.circuits.noises.GeneralizedAmplitudeDamping` class method), 303  
`from_dict()` (`braket.circuits.noises.Kraus` class method), 306  
`from_dict()` (`braket.circuits.noises.PauliChannel` class method), 288  
`from_dict()` (`braket.circuits.noises.PhaseDamping` class method), 305  
`from_dict()` (`braket.circuits.noises.PhaseFlip` class method), 286  
`from_dict()` (`braket.circuits.noises.TwoQubitDephasing` class method), 295  
`from_dict()` (`braket.circuits.noises.TwoQubitDepolarizing` class method), 293  
`from_dict()` (`braket.circuits.noises.TwoQubitPauliChannel` class method), 298  
`from_dict()` (`braket.parametric.free_parameter.FreeParameter` class method), 369  
`from_filter()` (`braket.circuits.noise_model.noise_model.NoiseModel` method), 56  
`from_ir()` (`braket.circuits.circuit.Circuit` static method), 79  
`from_lists()` (`braket.ahs.driving_field.DrivingField` static method), 15  
`from_lists()` (`braket.ahs.local_detuning.LocalDetuning` static method), 17  
`from_lists()` (`braket.timings.time_series.TimeSeries` static method), 393  
`from_object()` (`braket.tasks.analog_hamiltonian_simulation_quantum_task.AnalogHamiltonianSimulationQuantumTask` static method), 383  
`from_object()` (`braket.tasks.annealing_quantum_task_result.AnnealingQuantumTaskResult` static method), 384  
`from_object()` (`braket.tasks.gate_model_quantum_task_result.GateModelQuantumTaskResult` static method), 388  
`from_object()` (`braket.tasks.photonic_model_quantum_task_result.PhotonicModelQuantumTaskResult` static method), 390  
`from_string()` (`braket.tasks.analog_hamiltonian_simulation_quantum_task.AnalogHamiltonianSimulationQuantumTask` static method), 383  
`from_string()` (`braket.tasks.annealing_quantum_task_result.AnnealingQuantumTaskResult` static method), 385  
`from_string()` (`braket.tasks.gate_model_quantum_task_result.GateModelQuantumTaskResult` static method), 388  
`from_string()` (`braket.tasks.photonic_model_quantum_task_result.PhotonicModelQuantumTaskResult` static method), 391

## G

`gamma` (`braket.circuits.noise.DampingNoise` property), 279  
`GATE` (`braket.circuits.moments.MomentType` attribute), 245  
`GATE` (`braket.circuits.noise_model.criteria.CriteriaKey` attribute), 47  
`Gate` (class in `braket.circuits.gate`), 111  
`Gate.CCNot` (class in `braket.circuits.gate`), 113  
`Gate.CNNot` (class in `braket.circuits.gate`), 114  
`Gate.CPhaseShift` (class in `braket.circuits.gate`), 116



- Gate.CPhaseShift00 (class in *braket.circuits.gate*), 117  
 Gate.CPhaseShift01 (class in *braket.circuits.gate*), 119  
 Gate.CPhaseShift10 (class in *braket.circuits.gate*), 120  
 Gate.CSwap (class in *braket.circuits.gate*), 122  
 Gate.CV (class in *braket.circuits.gate*), 123  
 Gate.CY (class in *braket.circuits.gate*), 125  
 Gate.CZ (class in *braket.circuits.gate*), 126  
 Gate.ECR (class in *braket.circuits.gate*), 127  
 Gate.GPhase (class in *braket.circuits.gate*), 129  
 Gate.GPi (class in *braket.circuits.gate*), 130  
 Gate.GPi2 (class in *braket.circuits.gate*), 132  
 Gate.H (class in *braket.circuits.gate*), 134  
 Gate.I (class in *braket.circuits.gate*), 135  
 Gate.ISwap (class in *braket.circuits.gate*), 136  
 Gate.MS (class in *braket.circuits.gate*), 138  
 Gate.PhaseShift (class in *braket.circuits.gate*), 143  
 Gate.PRx (class in *braket.circuits.gate*), 140  
 Gate.PSwap (class in *braket.circuits.gate*), 141  
 Gate.PulseGate (class in *braket.circuits.gate*), 145  
 Gate.Rx (class in *braket.circuits.gate*), 146  
 Gate.Ry (class in *braket.circuits.gate*), 147  
 Gate.Rz (class in *braket.circuits.gate*), 149  
 Gate.S (class in *braket.circuits.gate*), 150  
 Gate.Si (class in *braket.circuits.gate*), 152  
 Gate.Swap (class in *braket.circuits.gate*), 153  
 Gate.T (class in *braket.circuits.gate*), 155  
 Gate.Ti (class in *braket.circuits.gate*), 156  
 Gate.U (class in *braket.circuits.gate*), 158  
 Gate.Unitary (class in *braket.circuits.gate*), 159  
 Gate.V (class in *braket.circuits.gate*), 161  
 Gate.Vi (class in *braket.circuits.gate*), 162  
 Gate.X (class in *braket.circuits.gate*), 163  
 Gate.XX (class in *braket.circuits.gate*), 165  
 Gate.XY (class in *braket.circuits.gate*), 166  
 Gate.Y (class in *braket.circuits.gate*), 168  
 Gate.YY (class in *braket.circuits.gate*), 169  
 Gate.Z (class in *braket.circuits.gate*), 171  
 Gate.ZZ (class in *braket.circuits.gate*), 172  
 gate\_calibrations (*braket.aws.aws\_device.AwsDevice* property), 25  
 GATE\_NOISE (*braket.circuits.moments.MomentType* attribute), 245  
 gate\_noise (*braket.circuits.noise\_model.noise\_model.NoiseModelIntrinsic* attribute), 55  
 GateCalibrations (class in *braket.circuits.gate\_calibrations*), 174  
 GateCriteria (class in *braket.circuits.noise\_model.gate\_criteria*), 53  
 GateModelQuantumTaskResult (class in *braket.tasks.gate\_model\_quantum\_task\_result*), 385  
 GaussianWaveform (class in *braket.pulse.waveforms*), 377  
 generalized\_amplitude\_damping() (*braket.circuits.circuit.Circuit* method), 88  
 generalized\_amplitude\_damping() (*braket.circuits.noise.Noise.GeneralizedAmplitudeDamping* static method), 258  
 generalized\_amplitude\_damping() (*braket.circuits.noises.GeneralizedAmplitudeDamping* static method), 303  
 GeneralizedAmplitudeDamping (class in *braket.circuits.noises*), 300  
 GeneralizedAmplitudeDampingNoise (class in *braket.circuits.noise*), 279  
 get() (*braket.circuits.moments.Moments* method), 248  
 get\_angle() (in module *braket.circuits.angled\_gate*), 65  
 get\_avg\_density() (*braket.tasks.analog\_hamiltonian\_simulation\_quantum\_task* method), 383  
 get\_checkpoint\_dir() (in module *braket.jobs.environment\_variables*), 358  
 get\_columns\_and\_pivot\_indices() (*braket.jobs.metrics\_data.log\_metrics\_parser.LogMetricsParser* method), 354  
 get\_compiled\_circuit() (*braket.tasks.gate\_model\_quantum\_task\_result.GateModelQuantumTaskResult* method), 388  
 get\_counts() (*braket.tasks.analog\_hamiltonian\_simulation\_quantum\_task* method), 383  
 get\_default\_jobs\_role() (*braket.aws.aws\_session.AwsSession* method), 41  
 get\_device() (*braket.aws.aws\_session.AwsSession* method), 43  
 get\_device\_region() (*braket.aws.aws\_device.AwsDevice* static method), 27  
 get\_devices() (*braket.aws.aws\_device.AwsDevice* static method), 26  
 get\_full\_image\_tag() (*braket.aws.aws\_session.AwsSession* method), 45  
 get\_hyperparameters() (in module *braket.jobs.environment\_variables*), 358  
 get\_input\_data\_dir() (in module *braket.jobs.environment\_variables*), 358  
 get\_instructions\_by\_type() (*braket.circuits.noise\_model.noise\_model.NoiseModelIntrinsic* method), 56  
 get\_job() (*braket.aws.aws\_session.AwsSession* method), 41  
 get\_job\_device\_arn() (in module *braket.jobs.environment\_variables*), 358  
 get\_job\_name() (in module *braket.jobs.environment\_variables*), 358  
 get\_keys() (*braket.circuits.noise\_model.criteria.Criteria* method), 48

[get\\_keys\(\)](#) (*braket.circuits.noise\_model.criteria.Criteria.GPhase* static method), 49  
[get\\_keys\(\)](#) (*braket.circuits.noise\_model.criteria.Criteria.QubitInitializationCriteria* static method), 50  
[get\\_keys\(\)](#) (*braket.circuits.noise\_model.criteria.Criteria.QubitInitializationCriteria* static method), 51  
[get\\_keys\(\)](#) (*braket.circuits.noise\_model.criteria.Criteria.QubitInitializationCriteria* static method), 52  
[get\\_keys\(\)](#) (*braket.circuits.noise\_model.gate\_criteria.GateCriteria.GPi2* static method), 53  
[get\\_keys\(\)](#) (*braket.circuits.noise\_model.observable\_criteria.ObservableCriteria.GPi2* static method), 57  
[get\\_keys\(\)](#) (*braket.circuits.noise\_model.qubit\_initialization\_criteria.QubitInitializationCriteria* static method), 59  
[get\\_keys\(\)](#) (*braket.circuits.noise\_model.unitary\_gate\_criteria.UnitaryGateCriteria* static method), 60  
[get\\_log\\_events\(\)](#) (*braket.aws.aws\_session.AwsSession* static method), 44  
[get\\_metric\\_data\\_with\\_pivot\(\)](#) (*braket.jobs.metrics\_data.log\_metrics\_parser.LogMetricsParser* static method), 354  
[get\\_metrics\\_for\\_job\(\)](#) (*braket.jobs.metrics\_data.cwl\_insights\_metrics\_fetcher.CwlInsightsMetricsFetcher* static method), 351  
[get\\_metrics\\_for\\_job\(\)](#) (*braket.jobs.metrics\_data.cwl\_metrics\_fetcher.CwlMetricsFetcher* static method), 352  
[get\\_observable\(\)](#) (in module *braket.circuits.translations*), 341  
[get\\_parsed\\_metrics\(\)](#) (*braket.jobs.metrics\_data.log\_metrics\_parser.LogMetricsParser* static method), 354  
[get\\_pauli\\_eigenvalues\(\)](#) (in module *braket.circuits.quantum\_operator\_helpers*), 323  
[get\\_prices\(\)](#) (*braket.tracking.pricing.Pricing* static method), 396  
[get\\_quantum\\_task\(\)](#) (*braket.aws.aws\_session.AwsSession* static method), 41  
[get\\_results\\_dir\(\)](#) (in module *braket.jobs.environment\_variables*), 358  
[get\\_tensor\\_product\(\)](#) (in module *braket.circuits.translations*), 341  
[get\\_value\\_by\\_result\\_type\(\)](#) (*braket.tasks.gate\_model\_quantum\_task\_result.GateModelQuantumTaskResult* static method), 387  
[global\\_phase](#) (*braket.circuits.circuit.Circuit* property), 69  
[GLOBAL\\_PHASE](#) (*braket.circuits.moments.MomentType* attribute), 245  
[GPhase](#) (class in *braket.circuits.gates*), 178  
[gphase\(\)](#) (*braket.circuits.circuit.Circuit* static method), 88  
[gphase\(\)](#) (*braket.circuits.gate.Gate.GPhase* static method), 130  
[GPhase](#) (class in *braket.circuits.gates*), 179  
[GPhase](#) (class in *braket.circuits.gates*), 230  
[gpi\(\)](#) (*braket.circuits.circuit.Circuit* static method), 89  
[gpi\(\)](#) (*braket.circuits.gates.GPi* static method), 131  
[gpi\(\)](#) (*braket.circuits.gates.GPi* static method), 231  
[gpi2\(\)](#) (*braket.circuits.circuit.Circuit* static method), 90  
[gpi2\(\)](#) (*braket.circuits.gates.GPi2* static method), 133  
[gpi2\(\)](#) (*braket.circuits.gates.GPi2* static method), 235  
[H](#) (class in *braket.circuits.gates*), 175  
[h\(\)](#) (*braket.circuits.circuit.Circuit* static method), 90  
[h\(\)](#) (*braket.circuits.gate.Gate.H* static method), 134  
[h\(\)](#) (*braket.circuits.gates.H* static method), 176  
[hamiltonian](#) (*braket.ahs.analog\_hamiltonian\_simulation.AnalogHamiltonian* property), 11  
[Hamiltonian](#) (class in *braket.ahs.hamiltonian*), 16  
[handle\\_parameter\\_value\(\)](#) (*braket.circuits.braket\_program\_context.BraketProgramContext* static method), 67  
[Hermitian](#) (class in *braket.circuits.observable*), 319  
[HybridJob](#) (in module *braket.jobs.hybrid\_job*), 358  
[HybridJobQueueInfo](#) (class in *braket.aws.queue\_information*), 46  
[I](#) (class in *braket.circuits.gates*), 176  
[I](#) (class in *braket.circuits.observable*), 315  
[i\(\)](#) (*braket.circuits.circuit.Circuit* static method), 91  
[i\(\)](#) (*braket.circuits.gate.Gate.I* static method), 136  
[i\(\)](#) (*braket.circuits.gates.I* static method), 177  
[iam\\_client](#) (*braket.aws.aws\_session.AwsSession* property), 40  
[id](#) (*braket.aws.aws\_quantum\_task.AwsQuantumTask* property), 35  
[id](#) (*braket.pulse.frame.Frame* property), 371  
[id](#) (*braket.pulse.port.Port* property), 372  
[id](#) (*braket.tasks.local\_quantum\_task.LocalQuantumTask* property), 389  
[id](#) (*braket.tasks.quantum\_task.QuantumTask* property), 391  
[INITIALIZATION\\_NOISE](#) (*braket.circuits.moments.MomentType* attribute), 245  
[initialization\\_noise](#) (*braket.circuits.noise\_model.noise\_model.NoiseModelInstruction* attribute), 55  
[InitializationCriteria](#) (class in *braket.circuits.noise\_model.initialization\_criteria*), 54

`insert_noise()` (`braket.circuits.noise_model.noise_model.NoiseModel` method), 55  
`InstanceConfig` (class in `braket.jobs.config`), 355  
`instanceCount` (`braket.jobs.config.InstanceConfig` attribute), 355  
`instanceType` (`braket.jobs.config.InstanceConfig` attribute), 355  
`Instruction` (class in `braket.circuits.instruction`), 241  
`instruction_matches()` (`braket.circuits.noise_model.circuit_instruction_criteria.CircuitInstructionCriteria` method), 47  
`instruction_matches()` (`braket.circuits.noise_model.criteria.Criteria.GateCriteria` method), 49  
`instruction_matches()` (`braket.circuits.noise_model.criteria.Criteria.UnitaryGateCriteria` method), 52  
`instruction_matches()` (`braket.circuits.noise_model.gate_criteria.GateCriteria` method), 54  
`instruction_matches()` (`braket.circuits.noise_model.unitary_gate_criteria.UnitaryGateCriteria` method), 60  
`instructions` (`braket.circuits.circuit.Circuit` property), 69  
`instructions` (`braket.circuits.noise_model.noise_model.NoiseModel` property), 55  
`IonQ` (`braket.devices.devices.Devices` attribute), 343  
`IRType` (class in `braket.circuits.serialization`), 340  
`is_available` (`braket.aws.aws_device.AwsDevice` property), 25  
`is_builtin_gate()` (`braket.circuits.braket_program_context.BraketProgramContext` method), 66  
`is_cptp()` (in module `braket.circuits.quantum_operator_helpers`), 323  
`is_hermitian()` (in module `braket.circuits.quantum_operator_helpers`), 322  
`is_s3_uri()` (`braket.aws.aws_session.AwsSession` static method), 43  
`is_square_matrix()` (in module `braket.circuits.quantum_operator_helpers`), 323  
`is_unitary()` (in module `braket.circuits.quantum_operator_helpers`), 323  
`ISING` (`braket.annealing.problem.ProblemType` attribute), 20  
`ISwap` (class in `braket.circuits.gates`), 204  
`iswap()` (`braket.circuits.circuit.Circuit` method), 91  
`iswap()` (`braket.circuits.gate.Gate.ISwap` static method), 137  
`iswap()` (`braket.circuits.gates.ISwap` static method), 205  
`IterationsModel` (`braket.circuits.moments.Moments` method), 247  
`ITERATION_NUMBER` (`braket.jobs.metrics_data.definitions.MetricType` attribute), 353  
`ITERATION_NUMBER` (`braket.jobs.metrics_data.log_metrics_parser.LogMetricsParser` attribute), 353  
**J**  
`JAQCD` (`braket.circuits.serialization.IRType` attribute), 340  
`JOB_COMPLETE` (`braket.aws.aws_quantum_job.AwsQuantumJob.LogState` attribute), 28  
`Jobs` (`braket.aws.queue_information.QueueDepthInfo` attribute), 45, 46  
**K**  
`key` (`braket.aws.aws_session.AwsSession.S3DestinationFolder` attribute), 39  
`keys()` (`braket.circuits.moments.Moments` method), 247  
`kmsKeyId` (`braket.jobs.config.OutputDataConfig` attribute), 355  
`Kraus` (class in `braket.circuits.noises`), 305  
`kraus()` (`braket.circuits.circuit.Circuit` method), 92  
`kraus()` (`braket.circuits.noise.Noise.Kraus` static method), 259  
`Kraus()` (`braket.circuits.noises.Kraus` static method), 306  
**L**  
`lattice` (`braket.ahs.discretization_types.DiscretizationProperties` attribute), 14  
`LEFT` (`braket.time_series.StitchBoundaryCondition` attribute), 392  
`linear` (`braket.annealing.problem.Problem` property), 20  
`list_keys()` (`braket.aws.aws_session.AwsSession` method), 43  
`load_job_checkpoint()` (in module `braket.jobs.data_persistence`), 356  
`load_job_result()` (in module `braket.jobs.data_persistence`), 357  
`LOCAL_DETUNING_PROPERTY` (`braket.ahs.analog_hamiltonian_simulation.AnalogHamiltonianSimulation` attribute), 11  
`LocalDetuning` (class in `braket.ahs.local_detuning`), 17  
`localPath` (`braket.jobs.config.CheckpointConfig` attribute), 355  
`LocalQuantumJob` (class in `braket.jobs.local.local_job`), 346  
`LocalQuantumTask` (class in `braket.tasks.local_quantum_task`), 389  
`LocalQuantumTaskBatch` (class in `braket.tasks.local_quantum_task_batch`), 390

LocalSimulator (class in `braket.devices.local_simulator`), 343

LOG\_GROUP (`braket.aws.aws_quantum_job.AwsQuantumJob` attribute), 28

LOG\_GROUP\_NAME (`braket.jobs.metrics_data.cwl_insights_metrics_fetcher.CwlInsightsMetricFetcher` attribute), 351

LOG\_GROUP\_NAME (`braket.jobs.metrics_data.cwl_metrics_fetcher.CwlMetricsFetcher` attribute), 352

log\_metric() (in module `braket.jobs.metrics`), 363

log\_stream() (in module `braket.jobs.logs`), 361

LogMetricsParser (class in `braket.jobs.metrics_data.log_metrics_parser`), 353

logs() (`braket.aws.aws_quantum_job.AwsQuantumJob` method), 31

logs() (`braket.jobs.local.local_job.LocalQuantumJob` method), 349

logs() (`braket.jobs.quantum_job.QuantumJob` method), 363

logs\_client (`braket.aws.aws_session.AwsSession` property), 40

## M

magnitude (`braket.ahs.local_detuning.LocalDetuning` property), 17

make\_bound\_circuit() (`braket.circuits.circuit.Circuit` method), 76

make\_bound\_pulse\_sequence() (`braket.pulse.pulse_sequence.PulseSequence` method), 374

map() (`braket.registers.qubit_set.QubitSet` method), 381

matrix\_equivalence() (`braket.circuits.quantum_operator.QuantumOperator` method), 322

MAX (`braket.jobs.metrics_data.definitions.MetricStatistic` attribute), 353

MAX\_CONNECTIONS\_DEFAULT (`braket.aws.aws_quantum_task_batch.AwsQuantumTaskBatch` attribute), 38

MAX\_RETRIES (`braket.aws.aws_quantum_task_batch.AwsQuantumTaskBatch` attribute), 38

maxRuntimeInSeconds (`braket.jobs.config.StoppingCondition` attribute), 355

MEAN (`braket.timings.time_series.StitchBoundaryCondition` attribute), 392

MEASURE (`braket.circuits.moments.MomentType` attribute), 245

Measure (class in `braket.circuits.measure`), 244

measure() (`braket.circuits.circuit.Circuit` method), 73

measured\_qubits (`braket.tasks.gate_model_quantum_task_result.GateModelQuantumTaskResult` attribute), 387

measurement\_counts (`braket.tasks.gate_model_quantum_task_result.GateModelQuantumTaskResult` attribute), 387

measurement\_counts\_copied\_from\_device (`braket.tasks.gate_model_quantum_task_result.GateModelQuantumTaskResult` attribute), 387

measurement\_counts\_from\_measurements() (`braket.tasks.gate_model_quantum_task_result.GateModelQuantumTaskResult` static method), 388

measurement\_probabilities (`braket.tasks.gate_model_quantum_task_result.GateModelQuantumTaskResult` attribute), 387

measurement\_probabilities\_copied\_from\_device (`braket.tasks.gate_model_quantum_task_result.GateModelQuantumTaskResult` attribute), 387

measurement\_probabilities\_from\_measurement\_counts() (`braket.tasks.gate_model_quantum_task_result.GateModelQuantumTaskResult` static method), 388

measurements (`braket.tasks.analog_hamiltonian_simulation_quantum_task_result.AnalogHamiltonianSimulationQuantumTaskResult` attribute), 383

measurements (`braket.tasks.gate_model_quantum_task_result.GateModelQuantumTaskResult` attribute), 387

measurements (`braket.tasks.photonic_model_quantum_task_result.PhotonicModelQuantumTaskResult` attribute), 390

measurements\_copied\_from\_device (`braket.tasks.gate_model_quantum_task_result.GateModelQuantumTaskResult` attribute), 387

measurements\_from\_measurement\_probabilities() (`braket.tasks.gate_model_quantum_task_result.GateModelQuantumTaskResult` static method), 388

message (`braket.aws.queue_information.HybridJobQueueInfo` attribute), 46, 47

message (`braket.aws.queue_information.QuantumTaskQueueInfo` attribute), 46

metadata() (`braket.aws.aws_quantum_job.AwsQuantumJob` method), 31

metadata() (`braket.aws.aws_quantum_task.AwsQuantumTask` method), 35

metadata() (`braket.jobs.local.local_job.LocalQuantumJob` method), 348

metadata() (`braket.jobs.quantum_job.QuantumJob` method), 364

metadata() (`braket.tasks.quantum_task.QuantumTask` method), 391

MetricPeriod (class in `braket.jobs.metrics_data.definitions`), 353

metrics() (`braket.aws.aws_quantum_job.AwsQuantumJob` method), 31

metrics() (`braket.jobs.local.local_job.LocalQuantumJob` method), 349

metrics() (`braket.jobs.quantum_job.QuantumJob` method), 364

METRICS\_DEFINITIONS (`braket.jobs.metrics_data.log_metrics_parser.LogMetricsParser` attribute), 353

MetricsRetrievalError (class in `braket.jobs.metrics_data.log_metrics_parser`), 353

MetricStatistic (class in `braket.jobs.metrics_data.definitions`), 353



*braket.jobs.metrics\_data.definitions*), 353  
 MetricType (class in *braket.jobs.metrics\_data.definitions*), 353  
 MIN (*braket.jobs.metrics\_data.definitions.MetricStatistic* attribute), 353  
 module  
   braket, 11  
   braket.ahs, 11  
   braket.ahs.analog\_hamiltonian\_simulation, 11  
   braket.ahs.atom\_arrangement, 12  
   braket.ahs.discretization\_types, 13  
   braket.ahs.driving\_field, 14  
   braket.ahs.field, 16  
   braket.ahs.hamiltonian, 16  
   braket.ahs.local\_detuning, 17  
   braket.ahs.pattern, 19  
   braket.ahs.shifting\_field, 19  
   braket.annealing, 19  
   braket.annealing.problem, 19  
   braket.aws, 21  
   braket.aws.aws\_device, 21  
   braket.aws.aws\_quantum\_job, 28  
   braket.aws.aws\_quantum\_task, 33  
   braket.aws.aws\_quantum\_task\_batch, 36  
   braket.aws.aws\_session, 39  
   braket.aws.queue\_information, 45  
   braket.circuits, 47  
   braket.circuits.angled\_gate, 62  
   braket.circuits.ascii\_circuit\_diagram, 66  
   braket.circuits.basis\_state, 66  
   braket.circuits.braket\_program\_context, 66  
   braket.circuits.circuit, 68  
   braket.circuits.circuit\_diagram, 109  
   braket.circuits.circuit\_helpers, 109  
   braket.circuits.compiler\_directive, 110  
   braket.circuits.compiler\_directives, 111  
   braket.circuits.free\_parameter, 111  
   braket.circuits.free\_parameter\_expression, 111  
   braket.circuits.gate, 111  
   braket.circuits.gate\_calibrations, 174  
   braket.circuits.gates, 175  
   braket.circuits.instruction, 241  
   braket.circuits.measure, 244  
   braket.circuits.moments, 245  
   braket.circuits.noise, 248  
   braket.circuits.noise\_helpers, 280  
   braket.circuits.noise\_model, 47  
   braket.circuits.noise\_model.circuit\_instruction\_directives, 47  
   braket.circuits.noise\_model.criteria, 47

braket.circuits.noise\_model.criteria\_input\_parsing, 52  
 braket.circuits.noise\_model.gate\_criteria, 53  
 braket.circuits.noise\_model.initialization\_criteria, 54  
 braket.circuits.noise\_model.noise\_model, 54  
 braket.circuits.noise\_model.observable\_criteria, 57  
 braket.circuits.noise\_model.qubit\_initialization\_criteria, 58  
 braket.circuits.noise\_model.result\_type\_criteria, 59  
 braket.circuits.noise\_model.unitary\_gate\_criteria, 60  
 braket.circuits.noises, 282  
 braket.circuits.observable, 307  
 braket.circuits.observables, 314  
 braket.circuits.operator, 320  
 braket.circuits.parameterizable, 321  
 braket.circuits.quantum\_operator, 321  
 braket.circuits.quantum\_operator\_helpers, 322  
 braket.circuits.qubit, 324  
 braket.circuits.qubit\_set, 324  
 braket.circuits.result\_type, 324  
 braket.circuits.result\_types, 333  
 braket.circuits.serialization, 340  
 braket.circuits.text\_diagram\_builders, 61  
 braket.circuits.text\_diagram\_builders.ascii\_circuit\_diagram, 61  
 braket.circuits.text\_diagram\_builders.text\_circuit\_diagram, 61  
 braket.circuits.text\_diagram\_builders.text\_circuit\_diagram\_helpers, 61  
 braket.circuits.text\_diagram\_builders.unicode\_circuit\_diagram, 61  
 braket.circuits.translations, 341  
 braket.circuits.unitary\_calculation, 341  
 braket.devices, 342  
 braket.devices.device, 342  
 braket.devices.devices, 343  
 braket.devices.local\_simulator, 343  
 braket.error\_mitigation, 345  
 braket.error\_mitigation.debias, 345  
 braket.error\_mitigation.error\_mitigation, 346  
 braket.ipython\_utils, 399  
 braket.jobs, 346  
 braket.jobs.config, 355  
 braket.jobs.data\_persistence, 356  
 braket.jobs.environment\_variables, 358  
 braket.jobs.hybrid\_job, 358

braket.jobs.image\_uris, 360  
 braket.jobs.local, 346  
 braket.jobs.local.local\_job, 346  
 braket.jobs.local.local\_job\_container, 350  
 braket.jobs.local.local\_job\_container\_setup, 350  
 braket.jobs.logs, 361  
 braket.jobs.metrics, 363  
 braket.jobs.metrics\_data, 350  
 braket.jobs.metrics\_data.cwl\_insights\_metrics\_fetcher, 350  
 braket.jobs.metrics\_data.cwl\_metrics\_fetcher, 352  
 braket.jobs.metrics\_data.definitions, 353  
 braket.jobs.metrics\_data.exceptions, 353  
 braket.jobs.metrics\_data.log\_metrics\_parser, 353  
 braket.jobs.quantum\_job, 363  
 braket.jobs.quantum\_job\_creation, 365  
 braket.jobs.serialization, 367  
 braket.parametric, 368  
 braket.parametric.free\_parameter, 368  
 braket.parametric.free\_parameter\_expression, 369  
 braket.parametric.parameterizable, 370  
 braket.pulse, 370  
 braket.pulse.ast, 370  
 braket.pulse.ast.approximation\_parser, 370  
 braket.pulse.ast.free\_parameters, 370  
 braket.pulse.ast.qasm\_parser, 370  
 braket.pulse.ast.qasm\_transformer, 371  
 braket.pulse.frame, 371  
 braket.pulse.port, 371  
 braket.pulse.pulse\_sequence, 372  
 braket.pulse.pulse\_sequence\_trace, 374  
 braket.pulse.waveforms, 375  
 braket.quantum\_information, 378  
 braket.quantum\_information.pauli\_string, 378  
 braket.registers, 380  
 braket.registers.qubit, 380  
 braket.registers.qubit\_set, 381  
 braket.tasks, 382  
 braket.tasks.analog\_hamiltonian\_simulation, 382  
 braket.tasks.annealing\_quantum\_task\_result, 384  
 braket.tasks.gate\_model\_quantum\_task\_result, 385  
 braket.tasks.local\_quantum\_task, 389  
 braket.tasks.local\_quantum\_task\_batch, 390  
 braket.tasks.photonic\_model\_quantum\_task\_result, 390  
 braket.tasks.quantum\_task, 391  
 braket.tasks.quantum\_task\_batch, 392  
 braket.timings, 392  
 braket.timings.time\_series, 392  
 braket.tracking, 396  
 braket.tracking.pricing, 396  
 braket.tracking.tracker, 396  
 braket.tracking.tracking\_context, 398  
 braket.tracking.tracking\_events, 399  
 moment\_type (*braket.circuits.moments.MomentsKey* attribute), 245  
 moments (*braket.circuits.circuit.Circuit* property), 69  
 Moments (*class in braket.circuits.moments*), 245  
 MomentsKey (*class in braket.circuits.moments*), 245  
 MomentType (*class in braket.circuits.moments*), 245  
 MS (*class in braket.circuits.gates*), 235  
 ms() (*braket.circuits.circuit.Circuit* method), 92  
 ms() (*braket.circuits.gate.Gate* static method), 139  
 ms() (*braket.circuits.gates* static method), 237  
 multi\_stream\_iter() (*in module braket.jobs.logs*), 361  
 MultiQubitPauliNoise (*class in braket.circuits.noise*), 276

## N

name (*braket.aws.aws\_quantum\_job.AwsQuantumJob* property), 30  
 name (*braket.circuits.compiler\_directive.CompilerDirective* property), 110  
 name (*braket.circuits.noise.Noise* property), 248  
 name (*braket.circuits.operator.Operator* property), 320  
 name (*braket.circuits.quantum\_operator.QuantumOperator* property), 321  
 name (*braket.circuits.result\_type.ResultType* property), 324  
 name (*braket.devices.device.Device* property), 343  
 name (*braket.jobs.local.local\_job.LocalQuantumJob* property), 348  
 name (*braket.jobs.quantum\_job.QuantumJob* property), 363  
 name (*braket.parametric.free\_parameter.FreeParameter* property), 368  
 new() (*braket.registers.qubit.Qubit* static method), 380  
 noise\_applied\_warning() (*in module braket.circuits.noise\_helpers*), 280  
 NO\_RESULT\_TERMINAL\_STATES  
 (*braket.aws.aws\_quantum\_task.AwsQuantumTask* attribute), 33  
 NODE\_ID (*braket.jobs.metrics\_data.log\_metrics\_parser.LogMetricsParser* attribute), 353  
 NODE\_TAG (*braket.jobs.metrics\_data.log\_metrics\_parser.LogMetricsParser* attribute), 353

**NOISE** (*braket.circuits.moments.MomentType* attribute), 245  
**noise** (*braket.circuits.noise\_model.noise\_model.NoiseModel* attribute), 54  
**Noise** (class in *braket.circuits.noise*), 248  
**Noise.AmplitudeDamping** (class in *braket.circuits.noise*), 249  
**Noise.BitFlip** (class in *braket.circuits.noise*), 251  
**Noise.Depolarizing** (class in *braket.circuits.noise*), 253  
**Noise.GeneralizedAmplitudeDamping** (class in *braket.circuits.noise*), 255  
**Noise.Kraus** (class in *braket.circuits.noise*), 259  
**Noise.PauliChannel** (class in *braket.circuits.noise*), 260  
**Noise.PhaseDamping** (class in *braket.circuits.noise*), 262  
**Noise.PhaseFlip** (class in *braket.circuits.noise*), 264  
**Noise.TwoQubitDephasing** (class in *braket.circuits.noise*), 266  
**Noise.TwoQubitDepolarizing** (class in *braket.circuits.noise*), 268  
**Noise.TwoQubitPauliChannel** (class in *braket.circuits.noise*), 271  
**noise\_index** (*braket.circuits.moments.MomentsKey* attribute), 245  
**NoiseModel** (class in *braket.circuits.noise\_model.noise\_model*), 55  
**NoiseModelInstruction** (class in *braket.circuits.noise\_model.noise\_model*), 54  
**NoiseModelInstructions** (class in *braket.circuits.noise\_model.noise\_model*), 55  
**NORMAL** (*braket.aws.queue\_information.QueueType* attribute), 45

**O**

**OBSERVABLE** (*braket.circuits.noise\_model.criteria.CriteriaKey* attribute), 47  
**observable** (*braket.circuits.result\_type.ObservableResultType* property), 332  
**Observable** (class in *braket.circuits.observable*), 307  
**Observable.H** (class in *braket.circuits.observable*), 308  
**Observable.Hermitian** (class in *braket.circuits.observable*), 308  
**Observable.I** (class in *braket.circuits.observable*), 309  
**Observable.Sum** (class in *braket.circuits.observable*), 310  
**Observable.TensorProduct** (class in *braket.circuits.observable*), 311  
**Observable.X** (class in *braket.circuits.observable*), 312  
**Observable.Y** (class in *braket.circuits.observable*), 312  
**Observable.Z** (class in *braket.circuits.observable*), 313  
**observable\_from\_ir()** (in *braket.circuitsobservables*), 320  
**ObservableCriteria** (class in *braket.circuits.noise\_model.observable\_criteria*), 57  
**ObservableParameterResultType** (class in *braket.circuits.result\_type*), 332  
**ObservableResultType** (class in *braket.circuits.result\_type*), 332  
**observables\_simultaneously\_measurable** (*braket.circuits.circuit.Circuit* property), 80  
**ONE\_MINUTE** (*braket.jobs.metrics\_data.definitions.MetricPeriod* attribute), 353  
**OPENQASM** (*braket.circuits.serialization.IRType* attribute), 340  
**OpenQASMSerializationProperties** (class in *braket.circuits.serialization*), 340  
**operator** (*braket.circuits.instruction.Instruction* property), 242  
**Operator** (class in *braket.circuits.operator*), 320  
**OQC** (*braket.devices.devices.Devices* attribute), 343  
**OutputDataConfig** (class in *braket.jobs.config*), 355

**P**

**Parameterizable** (class in *braket.parametric.parameterizable*), 370  
**parameters** (*braket.circuits.angled\_gate.AngledGate* property), 62  
**parameters** (*braket.circuits.angled\_gate.DoubleAngledGate* property), 63  
**parameters** (*braket.circuits.angled\_gate.TripleAngledGate* property), 64  
**parameters** (*braket.circuits.circuit.Circuit* property), 69  
**parameters** (*braket.circuits.gate.Gate.PulseGate* property), 145  
**parameters** (*braket.circuits.gates.PulseGate* property), 239  
**parameters** (*braket.circuits.noise.DampingNoise* property), 279  
**parameters** (*braket.circuits.noise.GeneralizedAmplitudeDampingNoise* property), 280  
**parameters** (*braket.circuits.noise.MultiQubitPauliNoise* property), 277  
**parameters** (*braket.circuits.noise.PauliNoise* property), 278  
**parameters** (*braket.circuits.noise.SingleProbabilisticNoise* property), 275  
**parameters** (*braket.circuits.result\_type.ObservableParameterResultType* property), 333  
**parameters** (*braket.parametric.parameterizable.Parameterizable* property), 370  
**parameters** (*braket.pulse.pulse\_sequence.PulseSequence* property), 372

parameters (*braket.pulse.waveforms.ConstantWaveform* property), 376  
 parameters (*braket.pulse.waveforms.DragGaussianWaveform* property), 376  
 parameters (*braket.pulse.waveforms.GaussianWaveform* property), 377  
 parse\_log\_message() (*braket.jobs.metrics\_data.log\_metrics\_parser.LogMetricsParser* method), 353  
 parse\_operator\_input() (in module *braket.circuits.noise\_model.criteria\_input\_parsing*), 52  
 parse\_qubit\_input() (in module *braket.circuits.noise\_model.criteria\_input\_parsing*), 53  
 parse\_s3\_uri() (*braket.aws.aws\_session.AwsSession* static method), 44  
 PARTIAL\_SUCCESS (*braket.tasks.analog\_hamiltonian\_simulation\_quantum\_task\_result.AnalogHamiltonianSimulationShotStatus* attribute), 382  
 pattern (*braket.ahs.field.Field* property), 16  
 Pattern (class in *braket.ahs.pattern*), 19  
 pauli\_channel() (*braket.circuits.circuit.Circuit* method), 93  
 pauli\_channel() (*braket.circuits.noise.Noise.PauliChannel* static method), 262  
 pauli\_channel() (*braket.circuits.noises.PauliChannel* static method), 288  
 PauliChannel (class in *braket.circuits.noises*), 286  
 PauliNoise (class in *braket.circuits.noise*), 277  
 PauliString (class in *braket.quantum\_information.pauli\_string*), 378  
 periodic\_signal() (*braket.timings.time\_series.TimeSeries* static method), 395  
 phase (*braket.ahs.driving\_field.DrivingField* property), 14  
 phase (*braket.quantum\_information.pauli\_string.PauliString* property), 378  
 phase\_damping() (*braket.circuits.circuit.Circuit* method), 93  
 phase\_damping() (*braket.circuits.noise.Noise.PhaseDamping* static method), 264  
 phase\_damping() (*braket.circuits.noises.PhaseDamping* static method), 305  
 phase\_flip() (*braket.circuits.circuit.Circuit* method), 93  
 phase\_flip() (*braket.circuits.noise.Noise.PhaseFlip* static method), 266  
 phase\_flip() (*braket.circuits.noises.PhaseFlip* static method), 285  
 PhaseDamping (class in *braket.circuits.noises*), 304  
 PhaseFlip (class in *braket.circuits.noises*), 284  
 phases (*braket.pulse.pulse\_sequence\_trace.PulseSequenceTrace* attribute), 375  
 PhaseShift (class in *braket.circuits.gates*), 198  
 phaseshift() (*braket.circuits.circuit.Circuit* method), 94  
 phaseshift() (*braket.circuits.gate.Gate.PhaseShift* static method), 144  
 phaseshift() (*braket.circuits.gates.PhaseShift* static method), 199  
 PhotonicModelQuantumTaskResult (class in *braket.tasks.photonic\_model\_quantum\_task\_result*), 390  
 PHYSICAL (*braket.circuits.serialization.QubitReferenceType* attribute), 340  
 PL\_PYTORCH (*braket.jobs.image\_uris.Framework* attribute), 360  
 PL\_TENSORFLOW (*braket.jobs.image\_uris.Framework* attribute), 360  
 play() (*braket.pulse.pulse\_sequence.PulseSequence* method), 377  
 Port (class in *braket.pulse.port*), 371  
 ports (*braket.aws.aws\_device.AwsDevice* property), 26  
 Position (class in *braket.jobs.logs*), 361  
 post\_sequence (*braket.tasks.analog\_hamiltonian\_simulation\_quantum\_task\_result.AnalogHamiltonianSimulationShotStatus* attribute), 382  
 power (*braket.circuits.instruction.Instruction* property), 242  
 power() (*braket.quantum\_information.pauli\_string.PauliString* method), 379  
 pre\_sequence (*braket.tasks.analog\_hamiltonian\_simulation\_quantum\_task\_result.AnalogHamiltonianSimulationShotStatus* attribute), 382  
 prepare\_quantum\_job() (in module *braket.jobs.quantum\_job\_creation*), 365  
 price\_search() (*braket.tracking.pricing.Pricing* method), 396  
 price\_search() (in module *braket.tracking.pricing*), 396  
 Pricing (class in *braket.tracking.pricing*), 396  
 PRIORITY (*braket.aws.queue\_information.QueueType* attribute), 45  
 probabilities (*braket.circuits.noise.MultiQubitPauliNoise* property), 277  
 probability (*braket.circuits.noise.GeneralizedAmplitudeDampingNoise* property), 280  
 probability (*braket.circuits.noise.SingleProbabilisticNoise* property), 275  
 Probability (class in *braket.circuits.result\_types*), 336  
 probability() (*braket.circuits.circuit.Circuit* method), 94  
 probability() (*braket.circuits.result\_type.ResultType.Probability* static method), 329  
 probability() (*braket.circuits.result\_types.Probability* static method), 336  
 Problem (class in *braket.annealing.problem*), 20  
 problem\_type (*braket.annealing.problem.Problem* property), 20



**problem\_type** (*braket.tasks.annealing\_quantum\_task\_result.QuantumTaskResult* attribute), 384  
**ProblemType** (class in *braket.annealing.problem*), 19  
**probX** (*braket.circuits.noise.PauliNoise* property), 278  
**probY** (*braket.circuits.noise.PauliNoise* property), 278  
**probZ** (*braket.circuits.noise.PauliNoise* property), 278  
**properties** (*braket.aws.aws\_device.AwsDevice* property), 25  
**properties** (*braket.devices.local\_simulator.LocalSimulator* property), 345  
**provider\_name** (*braket.aws.aws\_device.AwsDevice* property), 25  
**PRx** (class in *braket.circuits.gates*), 232  
**prx()** (*braket.circuits.circuit.Circuit* method), 94  
**prx()** (*braket.circuits.gate.Gate.PRx* static method), 141  
**prx()** (*braket.circuits.gates.PRx* static method), 233  
**PSwap** (class in *braket.circuits.gates*), 206  
**pswap()** (*braket.circuits.circuit.Circuit* method), 95  
**pswap()** (*braket.circuits.gate.Gate.PSwap* static method), 142  
**pswap()** (*braket.circuits.gates.PSwap* static method), 207  
**pulse\_gate()** (*braket.circuits.circuit.Circuit* method), 96  
**pulse\_gate()** (*braket.circuits.gate.Gate.PulseGate* static method), 145  
**pulse\_gate()** (*braket.circuits.gates.PulseGate* static method), 240  
**pulse\_sequence** (*braket.circuits.gate.Gate.PulseGate* property), 146  
**pulse\_sequence** (*braket.circuits.gates.PulseGate* property), 239  
**pulse\_sequences** (*braket.circuits.gate\_calibrations.GateCalibration* property), 174  
**PulseGate** (class in *braket.circuits.gates*), 239  
**PulseSequence** (class in *braket.pulse.pulse\_sequence*), 372  
**PulseSequenceTrace** (class in *braket.pulse.pulse\_sequence\_trace*), 374  
**put()** (*braket.timings.time\_series.TimeSeries* method), 392  
**Q**  
**QPU** (*braket.aws.aws\_device.AwsDeviceType* attribute), 22  
**qpu\_tasks\_cost()** (*braket.tracking.tracker.Tracker* method), 397  
**quadratic** (*braket.annealing.problem.Problem* property), 20  
**quantum\_tasks** (*braket.aws.queue\_information.QueueDepth* attribute), 45, 46  
**quantum\_tasks\_statistics()** (*braket.tracking.tracker.Tracker* method), 397  
**QuantumJob** (class in *braket.jobs.quantum\_job*), 363  
**QuantumOperator** (class in *braket.circuits.quantum\_operator*), 321  
**QuantumTask** (class in *braket.tasks.quantum\_task*), 391  
**QuantumTaskBatch** (class in *braket.tasks.quantum\_task\_batch*), 392  
**QuantumTaskQueueInfo** (class in *braket.aws.queue\_information*), 46  
**QUBIT** (*braket.circuits.noise\_model.criteria.CriteriaKey* attribute), 47  
**Qubit** (class in *braket.registers.qubit*), 380  
**qubit\_count** (*braket.circuits.circuit.Circuit* property), 69  
**qubit\_count** (*braket.circuits.moments.Moments* property), 246  
**qubit\_count** (*braket.circuits.quantum\_operator.QuantumOperator* property), 321  
**qubit\_count** (*braket.quantum\_information.pauli\_string.PauliString* property), 378  
**qubit\_intersection()** (*braket.circuits.noise\_model.criteria.Criteria.QubitInitializationCriteria* method), 51  
**qubit\_intersection()** (*braket.circuits.noise\_model.initialization\_criteria.InitializationCriteria* method), 54  
**qubit\_intersection()** (*braket.circuits.noise\_model.qubit\_initialization\_criteria.QubitInitializationCriteria* method), 59  
**qubit\_reference\_type** (*braket.circuits.serialization.OpenQASMSerializationProperties* attribute), 340  
**QubitInitializationCriteria** (class in *braket.circuits.noise\_model.qubit\_initialization\_criteria*), 58  
**QubitReferenceType** (class in *braket.circuits.serialization*), 340  
**qubits** (*braket.circuits.circuit.Circuit* property), 69  
**qubits** (*braket.circuits.moments.Moments* property), 246  
**qubits** (*braket.circuits.moments.MomentsKey* attribute), 245  
**qubits\_frozen** (*braket.circuits.circuit.Circuit* property), 80  
**QubitSet** (class in *braket.registers.qubit\_set*), 381  
**QUBO** (*braket.annealing.problem.ProblemType* attribute), 20  
**QuEra** (*braket.devices.devices.Devices* attribute), 343  
**QUERY\_DEFAULT\_JOB\_DURATION** (*braket.jobs.metrics\_data.cwl\_insights\_metrics\_fetcher.CwlInsightsMetricsFetcher* attribute), 351  
**queue\_depth()** (*braket.aws.aws\_device.AwsDevice* method), 27  
**queue\_position** (*braket.aws.queue\_information.HybridJobQueueInfo* attribute), 46, 47  
**queue\_position** (*braket.aws.queue\_information.QuantumTaskQueueInfo* attribute), 46

`queue_position()` (*braket.aws.aws\_quantum\_job.AwsQuantumJob* method), 31  
`queue_position()` (*braket.aws.aws\_quantum\_task.AwsQuantumTask* method), 35  
`queue_type` (*braket.aws.queue\_information.QuantumTaskQueueType* attribute), 46  
`QueueDepthInfo` (class in *braket.aws.queue\_information*), 45  
`QueueType` (class in *braket.aws.queue\_information*), 45  
**R**  
`READOUT_NOISE` (*braket.circuits.moments.MomentType* attribute), 245  
`readout_noise` (*braket.circuits.noise\_model.noise\_model.NoiseModelWithFunctions* attribute), 55  
`receive_event()` (*braket.tracking.tracker.Tracker* method), 396  
`record_array` (*braket.tasks.annealing\_quantum\_task\_result.AnnouncingQuantumTaskResult* attribute), 384  
`refresh_gate_calibrations()` (*braket.aws.aws\_device.AwsDevice* method), 27  
`refresh_metadata()` (*braket.aws.aws\_device.AwsDevice* method), 25  
`region` (*braket.aws.aws\_session.AwsSession* property), 40  
`REGIONS` (*braket.aws.aws\_device.AwsDevice* attribute), 22  
`register` (*braket.ahs.analog\_hamiltonian\_simulation.AnalogHamiltonianSimulation* property), 11  
`register_criteria()` (*braket.circuits.noise\_model.criteria.Criteria* class method), 48  
`register_gate()` (*braket.circuits.gate.Gate* class method), 113  
`register_noise()` (*braket.circuits.noise.Noise* class method), 249  
`register_observable()` (*braket.circuits.observable.Observable* class method), 308  
`register_result_type()` (*braket.circuits.result\_type.ResultType* class method), 325  
`register_subroutine()` (*braket.circuits.circuit.Circuit* class method), 68  
`register_tracker()` (*braket.tracking.tracking\_context.TrackingContext* method), 398  
`register_tracker()` (in module *braket.tracking.tracking\_context*), 398  
`registered_backends()` (*braket.devices.local\_simulator.LocalSimulator* static method), 345  
`remove_noise()` (*braket.circuits.noise\_model.noise\_model.NoiseModel* method), 56  
`result()` (*braket.aws.aws\_quantum\_job.AwsQuantumJob* method), 32  
`result()` (*braket.aws.aws\_quantum\_task.AwsQuantumTask* method), 36  
`result()` (*braket.jobs.local.local\_job.LocalQuantumJob* method), 349  
`result()` (*braket.jobs.quantum\_job.QuantumJob* method), 364  
`result()` (*braket.tasks.local\_quantum\_task.LocalQuantumTask* method), 389  
`result()` (*braket.tasks.quantum\_task.QuantumTask* method), 391  
`result_type_matches()` (*braket.circuits.noise\_model.criteria.Criteria.ObservableCriteria* method), 50  
`result_type_matches()` (*braket.circuits.noise\_model.observable\_criteria.ObservableCriteria* method), 58  
`result_type_matches()` (*braket.circuits.noise\_model.result\_type\_criteria.ResultTypeCriteria* method), 59  
`result_types` (*braket.circuits.circuit.Circuit* property), 69  
`result_types` (*braket.tasks.gate\_model\_quantum\_task\_result.GateModelQuantumTaskResult* attribute), 387  
`results()` (*braket.aws.aws\_quantum\_task\_batch.AwsQuantumTaskBatch* method), 38  
`results()` (*braket.tasks.local\_quantum\_task\_batch.LocalQuantumTaskBatch* method), 390  
`results()` (*braket.tasks.quantum\_task\_batch.QuantumTaskBatch* method), 392  
`RESULTS_FILENAME` (*braket.aws.aws\_quantum\_job.AwsQuantumJob* attribute), 28  
`RESULTS_FILENAME` (*braket.aws.aws\_quantum\_task.AwsQuantumTask* attribute), 34  
`RESULTS_READY_STATES` (*braket.aws.aws\_quantum\_task.AwsQuantumTask* attribute), 33  
`RESULTS_TAR_FILENAME` (*braket.aws.aws\_quantum\_job.AwsQuantumJob* attribute), 28  
`ResultType` (class in *braket.circuits.result\_type*), 324  
`ResultType.AdjointGradient` (class in *braket.circuits.result\_type*), 325  
`ResultType.Amplitude` (class in *braket.circuits.result\_type*), 326  
`ResultType.DensityMatrix` (class in *braket.circuits.result\_type*), 327  
`ResultType.Expectation` (class in *braket.circuits.result\_type*), 328  
`ResultType.Probability` (class in *braket.circuits.result\_type*), 329

ResultType.Sample (class in *braket.circuits.result\_type*), 329  
 ResultType.StateVector (class in *braket.circuits.result\_type*), 330  
 ResultType.Variance (class in *braket.circuits.result\_type*), 331  
 ResultTypeCriteria (class in *braket.circuits.noise\_model.result\_type\_criteria*), 59  
 retrieve\_image() (in module *braket.jobs.image\_uris*), 360  
 retrieve\_s3\_object\_body() (in *braket.aws.aws\_session.AwsSession* method), 41  
 retry\_unsuccessful\_tasks() (in *braket.aws.aws\_quantum\_task\_batch.AwsQuantumTaskBatch* method), 38  
 Rigetti (in *braket.devices.devices.Devices* attribute), 343  
 RIGHT (in *braket.timings.time\_series.StitchBoundaryCondition* attribute), 392  
 run() (in *braket.aws.aws\_device.AwsDevice* method), 22  
 run() (in *braket.devices.device.Device* method), 342  
 run() (in *braket.devices.local\_simulator.LocalSimulator* method), 344  
 run\_batch() (in *braket.aws.aws\_device.AwsDevice* method), 24  
 run\_batch() (in *braket.devices.device.Device* method), 342  
 run\_batch() (in *braket.devices.local\_simulator.LocalSimulator* method), 344  
 run\_log (in *braket.jobs.local.local\_job.LocalQuantumJob* property), 348  
 running\_in\_jupyter() (in *braket.ipython\_utils*), 399  
 Rx (class in *braket.circuits.gates*), 193  
 rx() (in *braket.circuits.circuit.Circuit* method), 96  
 rx() (in *braket.circuits.gate.Gate* static method), 147  
 rx() (in *braket.circuits.gates.Rx* static method), 194  
 Ry (class in *braket.circuits.gates*), 195  
 ry() (in *braket.circuits.circuit.Circuit* method), 97  
 ry() (in *braket.circuits.gate.Gate* static method), 148  
 ry() (in *braket.circuits.gates.Ry* static method), 196  
 rydberg (in *braket.ahs.discretization\_types.DiscretizationProperties* attribute), 14  
 Rz (class in *braket.circuits.gates*), 196  
 rz() (in *braket.circuits.circuit.Circuit* method), 97  
 rz() (in *braket.circuits.gate.Gate* static method), 150  
 rz() (in *braket.circuits.gates.Rz* static method), 197  
**S**  
 S (class in *braket.circuits.gates*), 184  
 s() (in *braket.circuits.circuit.Circuit* method), 98  
 s() (in *braket.circuits.gate.Gate* static method), 151  
 s() (in *braket.circuits.gates.S* static method), 185  
 s3\_client (in *braket.aws.aws\_session.AwsSession* property), 40  
 S3DataSourceConfig (class in *braket.jobs.config*), 355  
 s3Path (in *braket.jobs.config.OutputDataConfig* attribute), 355  
 s3Uri (in *braket.jobs.config.CheckpointConfig* attribute), 355  
 Sample (class in *braket.circuits.result\_types*), 338  
 sample() (in *braket.circuits.circuit.Circuit* method), 98  
 sample() (in *braket.circuits.result\_type.ResultType.Sample* static method), 330  
 sample() (in *braket.circuits.result\_types.Sample* static method), 338  
 sample() (in *braket.pulse.waveforms.ArbitraryWaveform* method), 375  
 sample() (in *braket.pulse.waveforms.ConstantWaveform* method), 376  
 sample() (in *braket.pulse.waveforms.DragGaussianWaveform* method), 377  
 sample() (in *braket.pulse.waveforms.GaussianWaveform* method), 377  
 sample() (in *braket.pulse.waveforms.Waveform* method), 375  
 save\_job\_checkpoint() (in *braket.jobs.data\_persistence*), 356  
 save\_job\_result() (in *braket.jobs.data\_persistence*), 357  
 search\_devices() (in *braket.aws.aws\_session.AwsSession* method), 43  
 SerializationProperties (in *braket.circuits.serialization*), 340  
 serialize() (in *braket.error\_mitigation.debias.Debias* method), 345  
 serialize() (in *braket.error\_mitigation.error\_mitigation.ErrorMitigation* method), 346  
 serialize\_values() (in *braket.jobs.serialization*), 367  
 series (in *braket.ahs.pattern.Pattern* property), 19  
 set\_frequency() (in *braket.pulse.pulse\_sequence.PulseSequence* method), 372  
 set\_phase() (in *braket.pulse.pulse\_sequence.PulseSequence* method), 372  
 set\_scale() (in *braket.pulse.pulse\_sequence.PulseSequence* method), 373  
 setup\_container() (in *braket.jobs.local.local\_job\_container\_setup*), 350  
 shift\_frequency() (in *braket.pulse.pulse\_sequence.PulseSequence* method), 372  
 shift\_phase() (in *braket.pulse.pulse\_sequence.PulseSequence* method), 373  
 ShotResult (class in *braket.tasks.analog\_hamiltonian\_simulation\_quantum*), 382  
 Si (class in *braket.circuits.gates*), 186

- [si\(\)](#) (*braket.circuits.circuit.Circuit* method), 99  
[si\(\)](#) (*braket.circuits.gate.Gate.Si* static method), 152  
[si\(\)](#) (*braket.circuits.gates.Si* static method), 187  
[SIMULATOR](#) (*braket.aws.aws\_device.AwsDeviceType* attribute), 21  
[simulator\\_tasks\\_cost\(\)](#) (*braket.tracking.tracker.Tracker* method), 397  
[SingleProbabilisticNoise](#) (class in *braket.circuits.noise*), 274  
[SingleProbabilisticNoise\\_1516](#) (class in *braket.circuits.noise*), 276  
[SingleProbabilisticNoise\\_34](#) (class in *braket.circuits.noise*), 275  
[site\\_type](#) (*braket.ahs.atom\_arrangement.AtomArrangement* attribute), 12  
[SiteType](#) (class in *braket.ahs.atom\_arrangement*), 12  
[size](#) (*braket.aws.aws\_quantum\_task\_batch.AwsQuantumTaskBatch* property), 39  
[size](#) (*braket.circuits.basis\_state.BasisState* property), 66  
[skip](#) (*braket.jobs.logs.Position* attribute), 361  
[sort\\_moments\(\)](#) (*braket.circuits.moments.Moments* method), 247  
[StandardObservable](#) (class in *braket.circuits.observable*), 313  
[start\(\)](#) (*braket.tracking.tracker.Tracker* method), 396  
[StartVerbatimBox](#) (class in *braket.circuits.compiler\_directives*), 111  
[state](#) (*braket.circuits.result\_type.ResultType.Amplitude* property), 327  
[state](#) (*braket.circuits.result\_types.Amplitude* property), 336  
[state\(\)](#) (*braket.aws.aws\_quantum\_job.AwsQuantumJob* method), 30  
[state\(\)](#) (*braket.aws.aws\_quantum\_task.AwsQuantumTask* method), 35  
[state\(\)](#) (*braket.jobs.local.local\_job.LocalQuantumJob* method), 348  
[state\(\)](#) (*braket.jobs.quantum\_job.QuantumJob* method), 363  
[state\(\)](#) (*braket.tasks.local\_quantum\_task.LocalQuantumTask* method), 389  
[state\(\)](#) (*braket.tasks.quantum\_task.QuantumTask* method), 391  
[state\\_vector\(\)](#) (*braket.circuits.circuit.Circuit* method), 99  
[state\\_vector\(\)](#) (*braket.circuits.result\_type.ResultType.StateVector* static method), 330  
[state\\_vector\(\)](#) (*braket.circuits.result\_types.StateVector* static method), 333  
[StateVector](#) (class in *braket.circuits.result\_types*), 333  
[status](#) (*braket.devices.device.Device* property), 343  
[status](#) (*braket.tasks.analog\_hamiltonian\_simulation\_quantum\_task\_result.ShotResult* attribute), 382  
[stitch\(\)](#) (*braket.ahs.driving\_field.DrivingField* method), 15  
[stitch\(\)](#) (*braket.ahs.local\_detuning.LocalDetuning* method), 18  
[stitch\(\)](#) (*braket.timings.time\_series.TimeSeries* method), 394  
[StitchBoundaryCondition](#) (class in *braket.timings.time\_series*), 392  
[stop\(\)](#) (*braket.tracking.tracker.Tracker* method), 396  
[StoppingCondition](#) (class in *braket.jobs.config*), 355  
[sts\\_client](#) (*braket.aws.aws\_session.AwsSession* property), 40  
[subindex](#) (*braket.circuits.moments.MomentsKey* attribute), 245  
[subroutine\(\)](#) (in module *braket.circuits.circuit*), 108  
[subs\(\)](#) (*braket.parametric.free\_parameter.FreeParameter* method), 368  
[subs\(\)](#) (*braket.parametric.free\_parameter\_expression.FreeParameterExpression* method), 369  
[subs\\_if\\_free\\_parameter\(\)](#) (in module *braket.parametric.free\_parameter\_expression*), 369  
[SUCCESS](#) (*braket.tasks.analog\_hamiltonian\_simulation\_quantum\_task\_result.LogState* attribute), 382  
[Sum](#) (class in *braket.circuits.observable*), 318  
[summands](#) (*braket.circuits.observable.Observable* property), 311  
[summands](#) (*braket.circuits.observable.Sum* property), 318  
[Swap](#) (class in *braket.circuits.gates*), 203  
[swap\(\)](#) (*braket.circuits.circuit.Circuit* method), 99  
[swap\(\)](#) (*braket.circuits.gate.Gate.Swap* static method), 154  
[swap\(\)](#) (*braket.circuits.gates.Swap* static method), 204  

## T

[T](#) (class in *braket.circuits.gates*), 187  
[t\(\)](#) (*braket.circuits.circuit.Circuit* method), 100  
[t\(\)](#) (*braket.circuits.gate.Gate.T* static method), 155  
[t\(\)](#) (*braket.circuits.gates.T* static method), 188  
[TALLING](#) (*braket.aws.aws\_quantum\_job.AwsQuantumJob.LogState* attribute), 28  
[target](#) (*braket.circuits.instruction.Instruction* property), 242  
[target](#) (*braket.circuits.result\_type.ObservableResultType* property), 332  
[target](#) (*braket.circuits.result\_type.ResultType.DensityMatrix* property), 328  
[target](#) (*braket.circuits.result\_type.ResultType.Probability* property), 329  
[target](#) (*braket.circuits.result\_types.DensityMatrix* property), 334  
[target](#) (*braket.circuits.result\_types.Probability* property), 336



`task_metadata` (`braket.tasks.analog_hamiltonian_simulation.QuantumTaskResult` attribute), 383  
`task_metadata` (`braket.tasks.annealing_quantum_task_result.AnnalingQuantumTaskResult` attribute), 384  
`task_metadata` (`braket.tasks.gate_model_quantum_task_result.GateModelQuantumTaskResult` attribute), 387  
`task_metadata` (`braket.tasks.photonic_model_quantum_task_result.PhotonicModelQuantumTaskResult` attribute), 390  
`tasks` (`braket.aws.aws_quantum_task_batch.AwsQuantumTaskBatch` property), 39  
`TensorProduct` (class in `braket.circuits.observables`), 317  
`TERMINAL_STATES` (`braket.aws.aws_quantum_job.AwsQuantumJob` attribute), 28  
`TERMINAL_STATES` (`braket.aws.aws_quantum_task.AwsQuantumTask` attribute), 33  
`terms` (`braket.ahs.driving_field.DrivingField` property), 14  
`terms` (`braket.ahs.hamiltonian.Hamiltonian` property), 16  
`terms` (`braket.ahs.local_detuning.LocalDetuning` property), 17  
`TextCircuitDiagram` (class in `braket.circuits.text_diagram_builders.text_circuit_diagram`), 61  
`Ti` (class in `braket.circuits.gates`), 189  
`ti` () (`braket.circuits.circuit.Circuit` method), 100  
`ti` () (`braket.circuits.gate.Gate` static method), 157  
`ti` () (`braket.circuits.gates.Ti` static method), 190  
`time` (`braket.circuits.moments.MomentsKey` attribute), 245  
`time` (`braket.timings.time_series.TimeSeriesItem` attribute), 392  
`time_series` (`braket.ahs.field.Field` property), 16  
`time_slices` () (`braket.circuits.moments.Moments` method), 247  
`times` () (`braket.timings.time_series.TimeSeries` method), 393  
`TimeSeries` (class in `braket.timings.time_series`), 392  
`TimeSeriesItem` (class in `braket.timings.time_series`), 392  
`timestamp` (`braket.jobs.logs.Position` attribute), 361  
`TIMESTAMP` (`braket.jobs.metrics_data.definitions.MetricType` attribute), 353  
`TIMESTAMP` (`braket.jobs.metrics_data.log_metrics_parser.LogMetricsParser` attribute), 353  
`to_circuit` () (`braket.quantum_information.pauli_string.PauliString` method), 380  
`to_dict` () (`braket.circuits.noise.DampingNoise` method), 279  
`to_dict` () (`braket.circuits.noise.GeneralizedAmplitudeDampingNoise` method), 280  
`to_dict` () (`braket.circuits.noise.MultiQubitPauliNoise` method), 277  
`to_dict` () (`braket.circuits.noise_model.criteria.Criteria` method), 260  
`to_dict` () (`braket.circuits.noise_model.criteria.Criteria` method), 278  
`to_dict` () (`braket.circuits.noise_model.criteria.Criteria` method), 275  
`to_dict` () (`braket.circuits.noise_model.criteria.Criteria` method), 48  
`to_dict` () (`braket.circuits.noise_model.criteria.Criteria` method), 49  
`to_dict` () (`braket.circuits.noise_model.criteria.Criteria` method), 50  
`to_dict` () (`braket.circuits.noise_model.criteria.Criteria` method), 51  
`to_dict` () (`braket.circuits.noise_model.criteria.Criteria` method), 52  
`to_dict` () (`braket.circuits.noise_model.gate_criteria.GateCriteria` method), 54  
`to_dict` () (`braket.circuits.noise_model.noise_model.NoiseModel` method), 56  
`to_dict` () (`braket.circuits.noise_model.noise_model.NoiseModelInstruction` method), 55  
`to_dict` () (`braket.circuits.noise_model.observable_criteria.ObservableCriteria` method), 57  
`to_dict` () (`braket.circuits.noise_model.qubit_initialization_criteria.QubitInitializationCriteria` method), 59  
`to_dict` () (`braket.circuits.noise_model.unitary_gate_criteria.UnitaryGateCriteria` method), 60  
`to_dict` () (`braket.circuits.noises.Kraus` method), 306  
`to_dict` () (`braket.parametric.free_parameter.FreeParameter` method), 368  
`to_ir` () (`braket.ahs.analog_hamiltonian_simulation.AnalogHamiltonianSimulation` method), 12  
`to_ir` () (`braket.annealing.problem.Problem` method), 21  
`to_ir` () (`braket.circuits.circuit.Circuit` method), 78  
`to_ir` () (`braket.circuits.compiler_directive.CompilerDirective` method), 110  
`to_ir` () (`braket.circuits.gate.Gate` method), 112  
`to_ir` () (`braket.circuits.gate_calibrations.GateCalibrations` method), 174  
`to_ir` () (`braket.circuits.instruction.Instruction` method), 242  
`to_ir` () (`braket.circuits.measure.Measure` method), 244  
`to_ir` () (`braket.circuits.noise.Noise` method), 248  
`to_ir` () (`braket.circuits.observable.Observable` method), 307  
`to_ir` () (`braket.circuits.operator.Operator` method), 320  
`to_ir` () (`braket.circuits.quantum_operator.QuantumOperator` method), 322  
`to_ir` () (`braket.circuits.result_type.ResultType` method), 324  
`to_ir` () (`braket.pulse.pulse_sequence.PulseSequence` method), 374

`to_matrix()` (*braket.circuits.gate.Gate.CCNot method*), 114  
`to_matrix()` (*braket.circuits.gate.Gate.CNot method*), 116  
`to_matrix()` (*braket.circuits.gate.Gate.CPhaseShift method*), 117  
`to_matrix()` (*braket.circuits.gate.Gate.CPhaseShift00 method*), 118  
`to_matrix()` (*braket.circuits.gate.Gate.CPhaseShift01 method*), 120  
`to_matrix()` (*braket.circuits.gate.Gate.CPhaseShift10 method*), 121  
`to_matrix()` (*braket.circuits.gate.Gate.CSwap method*), 123  
`to_matrix()` (*braket.circuits.gate.Gate.CV method*), 124  
`to_matrix()` (*braket.circuits.gate.Gate.CY method*), 126  
`to_matrix()` (*braket.circuits.gate.Gate.CZ method*), 127  
`to_matrix()` (*braket.circuits.gate.Gate.ECR method*), 128  
`to_matrix()` (*braket.circuits.gate.Gate.GPhase method*), 130  
`to_matrix()` (*braket.circuits.gate.Gate.GPi method*), 132  
`to_matrix()` (*braket.circuits.gate.Gate.GPi2 method*), 133  
`to_matrix()` (*braket.circuits.gate.Gate.H method*), 135  
`to_matrix()` (*braket.circuits.gate.Gate.I method*), 136  
`to_matrix()` (*braket.circuits.gate.Gate.ISwap method*), 138  
`to_matrix()` (*braket.circuits.gate.Gate.MS method*), 140  
`to_matrix()` (*braket.circuits.gate.Gate.PhaseShift method*), 144  
`to_matrix()` (*braket.circuits.gate.Gate.PRx method*), 141  
`to_matrix()` (*braket.circuits.gate.Gate.PSwap method*), 143  
`to_matrix()` (*braket.circuits.gate.Gate.PulseGate method*), 146  
`to_matrix()` (*braket.circuits.gate.Gate.Rx method*), 147  
`to_matrix()` (*braket.circuits.gate.Gate.Ry method*), 149  
`to_matrix()` (*braket.circuits.gate.Gate.Rz method*), 150  
`to_matrix()` (*braket.circuits.gate.Gate.S method*), 152  
`to_matrix()` (*braket.circuits.gate.Gate.Si method*), 153  
`to_matrix()` (*braket.circuits.gate.Gate.Swap method*), 155  
`to_matrix()` (*braket.circuits.gate.Gate.T method*), 156  
`to_matrix()` (*braket.circuits.gate.Gate.Ti method*), 157  
`to_matrix()` (*braket.circuits.gate.Gate.U method*), 159  
`to_matrix()` (*braket.circuits.gate.Gate.Unitary method*), 160  
`to_matrix()` (*braket.circuits.gate.Gate.V method*), 161  
`to_matrix()` (*braket.circuits.gate.Gate.Vi method*), 163  
`to_matrix()` (*braket.circuits.gate.Gate.X method*), 164  
`to_matrix()` (*braket.circuits.gate.Gate.XX method*), 166  
`to_matrix()` (*braket.circuits.gate.Gate.XY method*), 167  
`to_matrix()` (*braket.circuits.gate.Gate.Y method*), 168  
`to_matrix()` (*braket.circuits.gate.Gate.YY method*), 170  
`to_matrix()` (*braket.circuits.gate.Gate.Z method*), 171  
`to_matrix()` (*braket.circuits.gate.Gate.ZZ method*), 173  
`to_matrix()` (*braket.circuits.gates.CCNot method*), 227  
`to_matrix()` (*braket.circuits.gates.CNot method*), 202  
`to_matrix()` (*braket.circuits.gates.CPhaseShift method*), 210  
`to_matrix()` (*braket.circuits.gates.CPhaseShift00 method*), 212  
`to_matrix()` (*braket.circuits.gates.CPhaseShift01 method*), 213  
`to_matrix()` (*braket.circuits.gates.CPhaseShift10 method*), 215  
`to_matrix()` (*braket.circuits.gates.CSwap method*), 229  
`to_matrix()` (*braket.circuits.gates.CV method*), 216  
`to_matrix()` (*braket.circuits.gates.CY method*), 218  
`to_matrix()` (*braket.circuits.gates.CZ method*), 219  
`to_matrix()` (*braket.circuits.gates.ECR method*), 221  
`to_matrix()` (*braket.circuits.gates.GPhase method*), 178  
`to_matrix()` (*braket.circuits.gates.GPi method*), 231  
`to_matrix()` (*braket.circuits.gates.GPi2 method*), 234  
`to_matrix()` (*braket.circuits.gates.H method*), 175  
`to_matrix()` (*braket.circuits.gates.I method*), 177  
`to_matrix()` (*braket.circuits.gates.ISwap method*), 205  
`to_matrix()` (*braket.circuits.gates.MS method*), 236  
`to_matrix()` (*braket.circuits.gates.PhaseShift method*), 198  
`to_matrix()` (*braket.circuits.gates.PRx method*), 232  
`to_matrix()` (*braket.circuits.gates.PSwap method*), 206  
`to_matrix()` (*braket.circuits.gates.PulseGate method*), 240  
`to_matrix()` (*braket.circuits.gates.Rx method*), 194  
`to_matrix()` (*braket.circuits.gates.Ry method*), 195  
`to_matrix()` (*braket.circuits.gates.Rz method*), 197  
`to_matrix()` (*braket.circuits.gates.S method*), 185  
`to_matrix()` (*braket.circuits.gates.Si method*), 186  
`to_matrix()` (*braket.circuits.gates.Swap method*), 203  
`to_matrix()` (*braket.circuits.gates.T method*), 188  
`to_matrix()` (*braket.circuits.gates.Ti method*), 189  
`to_matrix()` (*braket.circuits.gates.U method*), 200  
`to_matrix()` (*braket.circuits.gates.Unitary method*), 238  
`to_matrix()` (*braket.circuits.gates.V method*), 191  
`to_matrix()` (*braket.circuits.gates.Vi method*), 192  
`to_matrix()` (*braket.circuits.gates.X method*), 180  
`to_matrix()` (*braket.circuits.gates.XX method*), 222

[to\\_matrix\(\)](#) ([braket.circuits.gates.XY method](#)), 208  
[to\\_matrix\(\)](#) ([braket.circuits.gates.Y method](#)), 182  
[to\\_matrix\(\)](#) ([braket.circuits.gates.YY method](#)), 224  
[to\\_matrix\(\)](#) ([braket.circuits.gates.Z method](#)), 183  
[to\\_matrix\(\)](#) ([braket.circuits.gates.ZZ method](#)), 225  
[to\\_matrix\(\)](#) ([braket.circuits.noise.Noise method](#)), 249  
[to\\_matrix\(\)](#) ([braket.circuits.noise.Noise.AmplitudeDamping method](#)), 251  
[to\\_matrix\(\)](#) ([braket.circuits.noise.Noise.BitFlip method](#)), 253  
[to\\_matrix\(\)](#) ([braket.circuits.noise.Noise.Depolarizing method](#)), 255  
[to\\_matrix\(\)](#) ([braket.circuits.noise.Noise.GeneralizedAmplitudeDamping method](#)), 259  
[to\\_matrix\(\)](#) ([braket.circuits.noise.Noise.Kraus method](#)), 260  
[to\\_matrix\(\)](#) ([braket.circuits.noise.Noise.PauliChannel method](#)), 262  
[to\\_matrix\(\)](#) ([braket.circuits.noise.Noise.PhaseDamping method](#)), 264  
[to\\_matrix\(\)](#) ([braket.circuits.noise.Noise.PhaseFlip method](#)), 266  
[to\\_matrix\(\)](#) ([braket.circuits.noise.Noise.TwoQubitDephasing method](#)), 268  
[to\\_matrix\(\)](#) ([braket.circuits.noise.Noise.TwoQubitDepolarizing method](#)), 271  
[to\\_matrix\(\)](#) ([braket.circuits.noise.Noise.TwoQubitPauliChannel method](#)), 274  
[to\\_matrix\(\)](#) ([braket.circuits.noises.AmplitudeDamping method](#)), 299  
[to\\_matrix\(\)](#) ([braket.circuits.noises.BitFlip method](#)), 283  
[to\\_matrix\(\)](#) ([braket.circuits.noises.Depolarizing method](#)), 290  
[to\\_matrix\(\)](#) ([braket.circuits.noises.GeneralizedAmplitudeDamping method](#)), 303  
[to\\_matrix\(\)](#) ([braket.circuits.noises.Kraus method](#)), 306  
[to\\_matrix\(\)](#) ([braket.circuits.noises.PauliChannel method](#)), 287  
[to\\_matrix\(\)](#) ([braket.circuits.noises.PhaseDamping method](#)), 305  
[to\\_matrix\(\)](#) ([braket.circuits.noises.PhaseFlip method](#)), 285  
[to\\_matrix\(\)](#) ([braket.circuits.noises.TwoQubitDephasing method](#)), 294  
[to\\_matrix\(\)](#) ([braket.circuits.noises.TwoQubitDepolarizing method](#)), 293  
[to\\_matrix\(\)](#) ([braket.circuits.noises.TwoQubitPauliChannel method](#)), 298  
[to\\_matrix\(\)](#) ([braket.circuits.observable.Observable.H method](#)), 308  
[to\\_matrix\(\)](#) ([braket.circuits.observable.Observable.Hermitian method](#)), 309  
[to\\_matrix\(\)](#) ([braket.circuits.observable.Observable.I method](#)), 310  
[to\\_matrix\(\)](#) ([braket.circuits.observable.Observable.Sum method](#)), 311  
[to\\_matrix\(\)](#) ([braket.circuits.observable.Observable.TensorProduct method](#)), 312  
[to\\_matrix\(\)](#) ([braket.circuits.observable.Observable.X method](#)), 312  
[to\\_matrix\(\)](#) ([braket.circuits.observable.Observable.Y method](#)), 313  
[to\\_matrix\(\)](#) ([braket.circuits.observable.Observable.Z method](#)), 313  
[to\\_matrix\(\)](#) ([braket.circuitsobservables.H method](#)), 314  
[to\\_matrix\(\)](#) ([braket.circuitsobservables.Hermitian method](#)), 319  
[to\\_matrix\(\)](#) ([braket.circuitsobservables.I method](#)), 315  
[to\\_matrix\(\)](#) ([braket.circuitsobservables.Sum method](#)), 318  
[to\\_matrix\(\)](#) ([braket.circuitsobservables.TensorProduct method](#)), 317  
[to\\_matrix\(\)](#) ([braket.circuitsobservables.X method](#)), 315  
[to\\_matrix\(\)](#) ([braket.circuitsobservables.Y method](#)), 316  
[to\\_matrix\(\)](#) ([braket.circuitsobservables.Z method](#)), 316  
[to\\_matrix\(\)](#) ([braket.circuits.quantum\\_operator.QuantumOperator method](#)), 322  
[to\\_time\\_trace\(\)](#) ([braket.pulse.pulse\\_sequence.PulseSequence method](#)), 372  
[to\\_unitary\(\)](#) ([braket.circuits.circuit.Circuit method](#)), 79  
[to\\_unsigned\\_observable\(\)](#) ([braket.quantum\\_information.pauli\\_string.PauliString method](#)), 378  
[topology\\_graph](#) ([braket.aws.aws\\_device.AwsDevice property](#)), 25  
[tracked\\_resources\(\)](#) ([braket.tracking.tracker.Tracker method](#)), 396  
[Tracker](#) (class in [braket.tracking.tracker](#)), 396  
[TrackingContext](#) (class in [braket.tracking.tracking\\_context](#)), 398  
[trapezoidal\\_signal\(\)](#) ([braket.timings.time\\_series.TimeSeries static method](#)), 395  
[TripleAngledGate](#) (class in [braket.circuits.angled\\_gate](#)), 64  
[two\\_qubit\\_dephasing\(\)](#) ([braket.circuits.circuit.Circuit method](#)), 101  
[two\\_qubit\\_dephasing\(\)](#) ([braket.circuits.noise.Noise.TwoQubitDephasing static method](#)), 268  
[two\\_qubit\\_dephasing\(\)](#)

(*braket.circuits.noises.TwoQubitDephasing* static method), 295

`two_qubit_depolarizing()` (*braket.circuits.circuit.Circuit* method), 101

`two_qubit_depolarizing()` (*braket.circuits.noise.Noise.TwoQubitDepolarizing* static method), 271

`two_qubit_depolarizing()` (*braket.circuits.noises.TwoQubitDepolarizing* static method), 293

`two_qubit_pauli_channel()` (*braket.circuits.circuit.Circuit* method), 102

`two_qubit_pauli_channel()` (*braket.circuits.noise.Noise.TwoQubitPauliChannel* static method), 274

`two_qubit_pauli_channel()` (*braket.circuits.noises.TwoQubitPauliChannel* static method), 298

`TwoQubitDephasing` (class in *braket.circuits.noises*), 293

`TwoQubitDepolarizing` (class in *braket.circuits.noises*), 290

`TwoQubitPauliChannel` (class in *braket.circuits.noises*), 295

type (*braket.aws.aws\_device.AwsDevice* property), 25

## U

`U` (class in *braket.circuits.gates*), 199

`u()` (*braket.circuits.circuit.Circuit* method), 102

`u()` (*braket.circuits.gate.Gate.U* static method), 159

`u()` (*braket.circuits.gates.U* static method), 201

`unfinished` (*braket.aws.aws\_quantum\_task\_batch.AwsQuantumTaskBatch* property), 39

`UnicodeCircuitDiagram` (class in *braket.circuits.text\_diagram\_builders.unicode\_circuit\_diagram\_builders*), 61

`Unitary` (class in *braket.circuits.gates*), 238

`unitary()` (*braket.circuits.circuit.Circuit* method), 103

`unitary()` (*braket.circuits.gate.Gate.Unitary* static method), 160

`unitary()` (*braket.circuits.gates.Unitary* static method), 238

`UNITARY_GATE` (*braket.circuits.noise\_model.criteria.CriteriaKey* attribute), 47

`UnitaryGateCriteria` (class in *braket.circuits.noise\_model.unitary\_gate\_criteria*), 60

`unsuccessful` (*braket.aws.aws\_quantum\_task\_batch.AwsQuantumTaskBatch* property), 39

`upload_local_data()` (*braket.aws.aws\_session.AwsSession* method), 42

`upload_to_s3()` (*braket.aws.aws\_session.AwsSession* method), 42

## V

`V` (class in *braket.circuits.gates*), 190

`v()` (*braket.circuits.circuit.Circuit* method), 103

`v()` (*braket.circuits.gate.Gate.V* static method), 161

`v()` (*braket.circuits.gates.V* static method), 191

`VACANT` (*braket.ahs.atom\_arrangement.SiteType* attribute), 12

`validate_circuit_and_shots()` (in module *braket.circuits.circuit\_helpers*), 109

`value` (*braket.timings.time\_series.TimeSeriesItem* attribute), 392

`values` (*braket.tasks.gate\_model\_quantum\_task\_result.GateModelQuantumTaskResult* attribute), 387

`values()` (*braket.circuits.moments.Moments* method), 247

`values()` (*braket.timings.time\_series.TimeSeries* method), 393

`variable_count` (*braket.tasks.annealing\_quantum\_task\_result.AnnealingQuantumTaskResult* attribute), 384

`Variance` (class in *braket.circuits.result\_types*), 339

`variance()` (*braket.circuits.circuit.Circuit* method), 104

`variance()` (*braket.circuits.result\_type.ResultType.Variance* static method), 331

`variance()` (*braket.circuits.result\_types.Variance* static method), 339

`verify_quantum_operator_matrix_dimensions()` (in module *braket.circuits.quantum\_operator\_helpers*), 322

`Vi` (class in *braket.circuits.gates*), 192

`vi()` (*braket.circuits.circuit.Circuit* method), 104

`vi()` (*braket.circuits.gate.Gate.Vi* static method), 163

`vi()` (*braket.circuits.gates.Vi* static method), 193

`VIRTUAL` (*braket.circuits.serialization.QubitReferenceType* attribute), 340

`volume_size_in_gb` (*braket.jobs.config.InstanceConfig* attribute), 355

## W

`Waveform` (class in *braket.pulse.waveforms*), 375

`weight_n_substrings()` (*braket.quantum\_information.pauli\_string.PauliString* method), 378

`wrap_with_list()` (in module *braket.circuits.noise\_helpers*), 280

## X

`X` (class in *braket.circuits.gates*), 180

`X` (class in *braket.circuits.observables*), 315

`x()` (*braket.circuits.circuit.Circuit* method), 104

`x()` (*braket.circuits.gate.Gate.X* static method), 164

`x()` (*braket.circuits.gates.X* static method), 181

`XX` (class in *braket.circuits.gates*), 222

`xx()` (*braket.circuits.circuit.Circuit* method), 105

`xx()` (*braket.circuits.gate.Gate.XX static method*), 166  
`xx()` (*braket.circuits.gates.XX static method*), 223  
`XY` (*class in braket.circuits.gates*), 208  
`xy()` (*braket.circuits.circuit.Circuit method*), 106  
`xy()` (*braket.circuits.gate.Gate.XY static method*), 167  
`xy()` (*braket.circuits.gates.XY static method*), 209

## Y

`Y` (*class in braket.circuits.gates*), 181  
`Y` (*class in braket.circuits.observables*), 316  
`y()` (*braket.circuits.circuit.Circuit method*), 106  
`y()` (*braket.circuits.gate.Gate.Y static method*), 169  
`y()` (*braket.circuits.gates.Y static method*), 182  
`YY` (*class in braket.circuits.gates*), 223  
`yy()` (*braket.circuits.circuit.Circuit method*), 107  
`yy()` (*braket.circuits.gate.Gate.YY static method*), 170  
`yy()` (*braket.circuits.gates.YY static method*), 224

## Z

`Z` (*class in braket.circuits.gates*), 183  
`Z` (*class in braket.circuits.observables*), 316  
`z()` (*braket.circuits.circuit.Circuit method*), 107  
`z()` (*braket.circuits.gate.Gate.Z static method*), 172  
`z()` (*braket.circuits.gates.Z static method*), 184  
`ZZ` (*class in braket.circuits.gates*), 225  
`zz()` (*braket.circuits.circuit.Circuit method*), 108  
`zz()` (*braket.circuits.gate.Gate.ZZ static method*), 173  
`zz()` (*braket.circuits.gates.ZZ static method*), 226