## CONTENTS

1 Alpha Disclaimer 3

2 User Documentation 5
   2.1 Installing Cirq 5
   2.2 Tutorial 8
   2.3 Circuits 17
   2.4 Gates 24
   2.5 Noise 28
   2.6 Simulation 29
   2.7 Schedules and Devices 34
   2.8 Qudits 36
   2.9 Development 38
   2.10 Examples 43
   2.11 Recommended interface and behavior of a plot method 61

3 API Reference 63
   3.1 API Reference 63

Python Module Index 429

Index 431
Cirq is a software library for writing, manipulating, and optimizing quantum circuits and then running them against quantum computers and simulators. Cirq attempts to expose the details of hardware, instead of abstracting them away, because, in the Noisy Intermediate-Scale Quantum (NISQ) regime, these details determine whether or not it is possible to execute a circuit at all.
Cirq is currently in alpha. We are still making breaking changes. We will break your code when we make new releases. We recommend that you target a specific version of Cirq, and periodically bump to the latest release. That way you have control over when a breaking change affects you.
2.1 Installing Cirq

Choose your operating system:

- *Installing on Linux*
- *Installing on Mac OS X*
- *Installing on Windows*
- *Installing on Docker*

If you want to create a development environment, see `development.md`.

2.1.1 Alpha Disclaimer

*Cirq is currently in alpha.* We are still making breaking changes. We *will* break your code when we make new releases. We recommend that you target a specific version of Cirq, and periodically bump to the latest release. That way you have control over when a breaking change affects you.

2.1.2 Installing on Linux

1. Make sure you have python 3.6.0 or greater.
   
   See *Installing Python 3 on Linux* @ the hitchhiker's guide to python.

2. Consider using a virtual environment.

3. Use `pip` to install `cirq`:

   ```
   python -m pip install --upgrade pip
   python -m pip install cirq
   ```

4. (Optional) install other dependencies.

   Install dependencies of features in `cirq.contrib`.

   ```
   python -m pip install cirq[contrib]
   ```

   Install system dependencies that pip can’t handle.
4. (Optional) install dependencies of features in cirq.contrib.

```bash
python -m pip install cirq[contrib]
```

Install system dependencies that pip can’t handle.

```bash
brew cask install mactex
```

- Without mactex, pdf writing functionality will not work.

5. Check that it works!

```bash
python -c 'import cirq; print(cirq.google.Foxtail)'
```

# should print:

```
# (0, 0) ——(0, 1) ——(0, 2) ——(0, 3) ——(0, 4) ——(0, 5) ——(0, 6) ——(0, 7) ——(0, 8) ——(0, 9) ——(0, 10)
# |     |     |     |     |     |     |     |     |     |     | # (1, 0) ——(1, 1) ——(1, 2) ——(1, 3) ——(1, 4) ——(1, 5) ——(1, 6) ——(1, 7) ——(1, 8) ——(1, 9) ——(1, 10)
```

---

### 2.1.4 Installing on Windows

1. If you are using the Windows Subsystem for Linux, use the Linux install instructions instead of these instructions.
2. Make sure you have python 3.5 or greater.

See Installing Python 3 on Windows @ the hitchhiker’s guide to python.

3. Use pip to install cirq:

```bash
python -m pip install --upgrade pip
python -m pip install cirq
```

4. (Optional) install dependencies of features in cirq.contrib.

```bash
python -m pip install cirq[contrib]
```

5. Check that it works!

```bash
python -c "import cirq; print(cirq.google.Foxtail)"
```

# should print:

```
(0, 0)——(0, 1)——(0, 2)——(0, 3)——(0, 4)——(0, 5)——(0, 6)——(0, 7)——(0, 8)——(0, 9)——(0, 10)
#   |   |   |   |   |   |   |   |   |   |   |
#   |   |   |   |   |   |   |   |   |   |   |
#   |   |   |   |   |   |   |   |   |   |   |
#   |   |   |   |   |   |   |   |   |   |   |
#   |   |   |   |   |   |   |   |   |   |   |
```

2.1.5 Installing on Docker

This will use a Docker image that will isolate Cirq’s installation from the rest of the system.

1. Install Docker on your host sytem.

2. Pull the docker image:

```bash
docker pull quantumlib/cirq
```

3. Check that it works!

```bash
docker run -it quantumlib/cirq python -c "import cirq; print(cirq.google.Foxtail)"
```

# should print:

```
(0, 0)——(0, 1)——(0, 2)——(0, 3)——(0, 4)——(0, 5)——(0, 6)——(0, 7)——(0, 8)——(0, 9)——(0, 10)
#   |   |   |   |   |   |   |   |   |   |   |
#   |   |   |   |   |   |   |   |   |   |   |
#   |   |   |   |   |   |   |   |   |   |   |
#   |   |   |   |   |   |   |   |   |   |   |
#   |   |   |   |   |   |   |   |   |   |   |
```

4. To run the image:

```bash
docker run -it quantumlib/cirq
```
2.2 Tutorial

In this tutorial we will go from knowing nothing about Cirq to creating a quantum variational algorithm. Note that this tutorial isn’t a quantum computing 101 tutorial, we assume familiarity of quantum computing at about the level of the textbook “Quantum Computation and Quantum Information” by Nielsen and Chuang.

To begin, please follow the instructions for installing Cirq.

2.2.1 Background: Variational quantum algorithms

The variational method in quantum theory is a classical method for finding low energy states of a quantum system. The rough idea of this method is that one defines a trial wave function (sometimes called an ansatz) as a function of some parameters, and then one finds the values of these parameters that minimize the expectation value of the energy with respect to these parameters. This minimized ansatz is then an approximation to the lowest energy eigenstate, and the expectation value serves as an upper bound on the energy of the ground state.

In the last few years (see arXiv:1304.3061 and arXiv:1507.08969 for example), it has been realized that quantum computers can mimic the classical technique and that a quantum computer does so with certain advantages. In particular, when one applies the classical variational method to a system of \( n \) qubits, an exponential number (in \( n \)) of complex numbers are necessary to generically represent the wave function of the system. However with a quantum computer one can directly produce this state using a parameterized quantum circuit, and then by repeated measurements estimate the expectation value of the energy.

This idea has led to a class of algorithms known as variational quantum algorithms. Indeed this approach is not just limited to finding low energy eigenstates, but minimizing any objective function that can be expressed as a quantum observable. It is an open question to identify under what conditions these quantum variational algorithms will succeed, and exploring this class of algorithms is a key part of research for noisy intermediate scale quantum computers.

The classical problem we will focus on is the 2D +/- Ising model with transverse field (ISING). This problem is NP-complete. So it is highly unlikely that quantum computers will be able to efficiently solve it across all instances. Yet this type of problem is illustrative of the general class of problems that Cirq is designed to tackle.

Consider the energy function

\[
E(s_1, \ldots, s_n) = \sum_{<i,j>} J_{i,j} s_i s_j + \sum_i h_i s_i
\]

where here each si, Ji,j, and hi are either +1 or -1. Here each index i is associated with a bit on a square lattice, and the <i,j> notation means sums over neighboring bits on this lattice. The problem we would like to solve is, given Ji,j, and hi, find an assignment of si values that minimize E.

How does a variational quantum algorithm work for this? One approach is to consider \( n \) qubits and associate them with each of the bits in the classical problem. This maps the classical problem onto the quantum problem of minimizing the expectation value of the observable

\[
H = \sum_{<i,j>} J_{i,j} Z_i Z_j + \sum_i h_i Z_i
\]

Then one defines a set of parameterized quantum circuits, i.e. a quantum circuit where the gates (or more general quantum operations) are parameterized by some values. This produces an ansatz state

\[
|\psi(p_1, p_2, \ldots, p_k)\rangle
\]

where pi are the parameters that produce this state (here we assume a pure state, but mixed states are of course possible).

The variational algorithm then works by noting that one can obtain the value of the objective function for a given ansatz state by

1. Prepare the ansatz state.
2. Make a measurement which samples from some terms in H.

Note that one cannot always measure $H$ directly (without the use of quantum phase estimation). So one often relies on the linearity of expectation values to measure parts of $H$ in step 2. One always needs to repeat the measurements to obtain an estimate of the expectation value. How many measurements needed to achieve a given accuracy is beyond the scope of this tutorial, but Cirq can help investigate this question.

The above shows that one can use a quantum computer to obtain estimates of the objective function for the ansatz. This can then be used in an outer loop to try to obtain parameters for the the lowest value of the objective function. For these values, one can then use that best ansatz to produce samples of solutions to the problem which obtain a hopefully good approximation for the lowest possible value of the objective function.

### 2.2.2 Create a circuit on a Grid

To build the above variational quantum algorithm using Cirq, one begins by building the appropriate circuit. In Cirq circuits are represented either by a `Circuit` object or a `Schedule` object. Schedules offer more control over quantum gates and circuits at the timing level, which we do not need, so here we will work with `Circuit`s instead.

Conceptually: a `Circuit` is a collection of `Moments`. A `Moment` is a collection of `Operations` that all act during the same abstract time slice. An `Operation` is an effect that operates on a specific subset of `Qubits`. The most common type of `Operation` is a `Gate` applied to several qubits (a `GateOperation`). The following diagram should help illustrate these concepts.

![Circuit Diagram](image)

Because the problem we have defined has a natural structure on a grid, we will use Cirq’s built in `GridQubits` as our qubits. We will demonstrate some of how this works in an interactive Python environment, the following code can be run in series in a Python environment where you have Cirq installed.

Let’s begin by talking about our qubits. In an interactive Python environment run
Here we see that we’ve created a bunch of GridQubits. GridQubits implement the Qid class, which just means that they are equatable and hashable. Qid has an abstract comparison_key method that must be implemented by child types in order to ensure there’s a reasonable sorting order for diagrams and that this matches what happens when sorted(qubits) is called. GridQubits in addition have a row and column, indicating their position on a grid.

Now that we have some qubits, let us construct a Circuit on these qubits. For example, suppose we want to apply the Hadamard gate $\text{H}$ to every qubit whose row index plus column index is even and an $\text{X}$ gate to every qubit whose row index plus column index is odd. To do this we write

```python
import cirq

definition the length of the grid.
length = 3
# define qubits on the grid.
qubits = [cirq.GridQubit(i, j) for i in range(length) for j in range(length)]
print(qubits)
# prints
# [cirq.GridQubit(0, 0), cirq.GridQubit(0, 1), cirq.GridQubit(0, 2), cirq.GridQubit(1, 0), cirq.GridQubit(1, 1), cirq.GridQubit(1, 2), cirq.GridQubit(2, 0), cirq.GridQubit(2, 1), cirq.GridQubit(2, 2)]

circuit = cirq.Circuit()
circuit.append(cirq.H(q) for q in qubits if (q.row + q.col) % 2 == 0)
circuit.append(cirq.X(q) for q in qubits if (q.row + q.col) % 2 == 1)
print(circuit)
# (0, 0): —H—
# (0, 1): —X—
# (0, 2): —H—
# (1, 0): —X—
# (1, 1): —H—
# (1, 2): —X—
# (2, 0): —H—
# (2, 1): —X—
# (2, 2): —H—
```

One thing to notice here. First `cirq.X` is a Gate object. There are many different gates supported by Cirq. A good place to look at gates that are defined is in common_gates.py. One common confusion to avoid is the difference between a gate class and a gate object (which is an instantiation of a class). The second is that gate objects are transformed into Operations (technically GateOperations) via either the method `on(qubit)` or, as we see for the X gates, via simply applying the gate to the qubits (qubit). Here we only apply single qubit gates, but a similar pattern applies for multiple qubit gates with a sequence of qubits as parameters.

Another thing to notice about the above circuit is that the gates from both the append instructions appear on the same vertical line. Gates appearing on the same vertical line constitute a Moment.

We can modify this by changing the InsertStrategy of the append method. InsertStrategy describes how new insertions into Circuits place their gates. Details of these strategies can be found in the circuit documentation. The default InsertStrategy used in the above circuit is EARLIEST which resulted in the X gates sliding over to act at the earliest Moment they can. If we wanted to insert the gates so that they form individual Moments, we could instead use the NEW_THEN_INLINE insertion strategy:
circuit = cirq.Circuit()
circuit.append([cirq.H(q) for q in qubits if (q.row + q.col) % 2 == 0],
               strategy=cirq.InsertStrategy.EARLIEST)
circuit.append([cirq.X(q) for q in qubits if (q.row + q.col) % 2 == 1],
               strategy=cirq.InsertStrategy.NEW_THEN_INLINE)
print(circuit)
# prints
# (0, 0): ___H_____
# (0, 1): ______X___
# (0, 2): ___H_____
# (1, 0): ______X___
# (1, 1): ___H_____
# (1, 2): ______X___
# (2, 0): ___H_____
# (2, 1): ______X___
# (2, 2): ___H_____

This circuit has now has staggered gates created by two Moments.

for i, m in enumerate(circuit):
    print('Moment {}: {}'.format(i, m))
# prints
# Moment 0: H((0, 0)) and H((0, 2)) and H((1, 1)) and H((2, 0)) and H((2, 2))
# Moment 1: X((0, 1)) and X((1, 0)) and X((1, 2)) and X((2, 1))

Here we see that we can iterate over a Circuit’s Moments.

### 2.2.3 Creating the Ansatz

If you look closely at the circuit creation code above you will see that we applied the append method to both a generator and a list (recall that in Python one can use generator comprehensions in method calls). Inspecting the code for append one sees that the append method generally takes an OP_TREE (or a Moment). What is an OP_TREE?

It is not a class but a contract. Roughly an OP_TREE is anything that can be flattened, perhaps recursively, into a list of operations, or into a single operation. Examples of an OP_TREE are

- A single Operation.
- A list of Operations.
- A tuple of Operations.
- A list of a list of Operations.
- A generator yielding Operations.

This last case yields a nice pattern for defining sub-circuits / layers: define a function that takes in the relevant parameters and then yields the operations for the sub circuit and then this can be appended to the Circuit:

```python
def rot_x_layer(length, half_turns):
    """Yields X rotations by half_turns on a square grid of given length."""
```

(continues on next page)
Another important concept here is that the rotation gate is specified in “half turns”. For a rotation about X this is the gate \( \cos(h_{\text{half turns}} \times \pi) \mathbb{I} + i \sin(h_{\text{half turns}} \times \pi) X \).

There is a lot of freedom defining a variational ansatz. Here we will do a variation on a QAOA strategy and define an ansatz related to the problem we are trying to solve.

First we need to choose how the instances of the problem are represented. These are the values \( J \) and \( h \) in the Hamiltonian definition. We will represent these as two dimensional arrays (lists of lists). For \( J \) we will use two such lists, one for the row links and one for the column links.

Here is code that we can use to generate random problem instances

```python
import random

def rand2d(rows, cols):
    return [[random.choice([+1, -1]) for _ in range(cols)] for _ in range(rows)]

def random_instance(length):
    # transverse field terms
    h = rand2d(length, length)
    # links within a row
    jr = rand2d(length - 1, length)
    # links within a column
    jc = rand2d(length, length - 1)
    return (h, jr, jc)

h, jr, jc = random_instance(3)
print('transverse fields: {}'.format(h))
print('row j fields: {}'.format(jr))
print('column j fields: {}'.format(jc))
# prints something like
# transverse fields: [[-1, 1, -1], [-1, -1, -1], [-1, 1, -1]]
# row j fields: [[1, 1, -1], [-1, 1, 1]]
# column j fields: [[1, -1], [-1, 1], [-1, 1]]
```

where the actual values will be different for an individual run because they are using `random.choice`.

Given this definition of the problem instance we can now introduce our ansatz. Our ansatz will consist of one step of a circuit made up of

1. Apply an XPowGate for the same parameter for all qubits. This is the method we have written above.
2. Apply a ZPowGate for the same parameter for all qubits where the transverse field term \( h \) is +1.
def rot_z_layer(h, half_turns):
    """Yields Z rotations by half_turns conditioned on the field h."
    gate = cirq.ZPowGate(exponent=half_turns)
    for i, h_row in enumerate(h):
        for j, h_ij in enumerate(h_row):
            if h_ij == 1:
                yield gate(cirq.GridQubit(i, j))

1. Apply a CZPowGate for the same parameter between all qubits where the coupling field term J is +1. If the field is -1 apply CZPowGate conjugated by X gates on all qubits.

def rot_11_layer(jr, jc, half_turns):
    """Yields rotations about |11> conditioned on the jr and jc fields."
    gate = cirq.CZPowGate(exponent=half_turns)
    for i, jr_row in enumerate(jr):
        for j, jr_ij in enumerate(jr_row):
            if jr_ij == -1:
                yield cirq.X(cirq.GridQubit(i, j))
                yield cirq.X(cirq.GridQubit(i + 1, j))
                yield gate(cirq.GridQubit(i, j),
                           cirq.GridQubit(i + 1, j))
            if jr_ij == -1:
                yield cirq.X(cirq.GridQubit(i, j))
                yield cirq.X(cirq.GridQubit(i, j + 1))
    for i, jc_row in enumerate(jc):
        for j, jc_ij in enumerate(jc_row):
            if jc_ij == -1:
                yield cirq.X(cirq.GridQubit(i, j))
                yield cirq.X(cirq.GridQubit(i, j + 1))
                yield gate(cirq.GridQubit(i, j),
                           cirq.GridQubit(i, j + 1))
            if jc_ij == -1:
                yield cirq.X(cirq.GridQubit(i, j))
                yield cirq.X(cirq.GridQubit(i, j + 1))

Putting this together we can create a step that uses just three parameters. The code to do this uses the generator for each of the layers (note to advanced Python users that this code is not a bug in using yield due to the auto flattening of the OP_TREE concept. Normally one would want to use yield from here, but this is not necessary):

def one_step(h, jr, jc, x_half_turns, h_half_turns, j_half_turns):
    length = len(h)
    yield rot_x_layer(length, x_half_turns, h, h_half_turns, j_half_turns)
    yield rot_z_layer(h, h_half_turns)
    yield rot_11_layer(jr, jc, j_half_turns)

h, jr, jc = random_instance(3)
circuit = cirq.Circuit()
circuit.append(one_step(h, jr, jc, 0.1, 0.2, 0.3),
               strategy=cirq.InsertStrategy.EARLIEST)
print(circuit)
# prints something like
# (0, 0): —X^0.1—X—@—X—@—X—@^0.
# (0, 1): —X^0.1—Z^(1/5)—@—X—@^0.
# (1, 1): —X—X—X—@—X—@—X—@—X
Here we see that we have chosen particular parameter values (0.1, 0.2, 0.3).

### 2.2.4 Simulation

Now let’s see how to simulate the circuit corresponding to creating our ansatz. In Cirq the simulators make a distinction between a “run” and a “simulation”. A “run” only allows for a simulation that mimics the actual quantum hardware. For example, it does not allow for access to the amplitudes of the wave function of the system, since that is not experimentally accessible. “Simulate” commands, however, are more broad and allow different forms of simulation. When prototyping small circuits it is useful to execute “simulate” methods, but one should be wary of relying on them when run against actual hardware.

Currently Cirq ships with a simulator tied strongly to the gate set of the Google xmon architecture. However, for convenience, the simulator attempts to automatically convert unknown operations into XmonGates (as long as the operation specifies a matrix or a decomposition into XmonGates). This can in principle allows us to simulate any circuit that has gates that implement one and two qubit KnownMatrix gates. Future releases of Cirq will expand these simulators.

Because the simulator is tied to the xmon gate set, the simulator lives, in contrast to core Cirq, in the `cirq.google` module. To run a simulation of the full circuit we simply create a simulator, and pass the circuit to the simulator.

```python
simulator = cirq.Simulator()
circuit = cirq.Circuit()
circuit.append(one_step(h, jr, jc, 0.1, 0.2, 0.3))
circuit.append(cirq.measure(*qubits, key='x'))
results = simulator.run(circuit, repetitions=100)
print(results.histogram(key='x'))
# prints something like
# Counter({0: 85, 128: 5, 32: 3, 1: 2, 4: 1, 2: 1, 8: 1, 18: 1, 20: 1})
```
Note that we have run the simulation 100 times and produced a histogram of the counts of the measurement results. What are the keys in the histogram counter? Note that we have passed in the order of the qubits. This ordering is then used to translate the order of the measurement results to a register using a big endian representation.

For our optimization problem we will want to calculate the value of the objective function for a given result run. One way to do this is use the raw measurement data from the result of simulator.run. Another way to do this is to provide to the histogram a method to calculate the objective: this will then be used as the key for the returned Counter.

```python
import numpy as np

def energy_func(length, h, jr, jc):
    def energy(measurements):
        # Reshape measurement into array that matches grid shape.
        meas_list_of_lists = [measurements[i * length:(i + 1) * length] for i in range(length)]

        # Convert true/false to +1/-1.
        pm_meas = 1 - 2 * np.array(meas_list_of_lists).astype(np.int32)

        tot_energy = np.sum(pm_meas * h)

        for i, jr_row in enumerate(jr):
            for j, jr_ij in enumerate(jr_row):
                tot_energy += jr_ij * pm_meas[i, j] * pm_meas[i + 1, j]

        for i, jc_row in enumerate(jc):
            for j, jc_ij in enumerate(jc_row):
                tot_energy += jc_ij * pm_meas[i, j] * pm_meas[i, j + 1]

        return tot_energy
    return energy

print(results.histogram(key='x', fold_func=energy_func(3, h, jr, jc)))
# prints something like
# Counter({7: 79, 5: 12, -1: 4, 1: 3, 13: 1, -3: 1})
```

One can then calculate the expectation value over all repetitions

```python
def obj_func(result):
    energy_hist = result.histogram(key='x', fold_func=energy_func(3, h, jr, jc))
    return np.sum((k * v for k, v in energy_hist.items())) / result.repetitions
print('Value of the objective function %.2f' % obj_func(results))
# prints something like
# Value of the objective function 6.21
```

### 2.2.5 Parameterizing the Ansatz

Now that we have constructed a variational ansatz and shown how to simulate it using Cirq, we can now think about optimizing the value. On quantum hardware one would most likely want to have the optimization code as close to the hardware as possible. As the classical hardware that is allowed to inter-operate with the quantum hardware becomes better specified, this language will be better defined. Without this specification, however, Cirq also provides a useful concept for optimizing the looping in many optimization algorithms. This is the fact that many of the values in the gate sets can, instead of being specified by a float, be specified by a Symbol and this Symbol can be substituted for a value specified at execution time.

Luckily for us, we have written our code so that using parameterized values is as simple as passing Symbol objects where we previously passed float values.

```python
import sympy

circuit = cirq.Circuit()
alpha = sympy.Symbol('alpha')
```

(continues on next page)
beta = sympy.Symbol('beta')
gamma = sympy.Symbol('gamma')
circuit.append(one_step(h, jr, jc, alpha, beta, gamma))
circuit.append(cirq.measure(*qubits, key='x'))
print(circuit)
# prints something like
# (0, 0):
  # alpha
  # 'x'
  # →
# (0, 1):
  # β
  # →
# (0, 2):
  # γ
  # →
# (1, 0):
  # α
  # →
# (1, 1):
  # β
  # →
# (1, 2):
  # γ
  # →
# (2, 0):
  # α
  # →
# (2, 1):
  # β
  # →
# (2, 2):
  # γ
  # →

Note now that the circuit’s gates are parameterized.
Parameters are specified at runtime using a `ParamResolver` which is just a dictionary from `Symbol` keys to runtime values. For example,

```python
resolver = cirq.ParamResolver({'alpha': 0.1, 'beta': 0.3, 'gamma': 0.7})
resolved_circuit = cirq.resolve_parameters(circuit, resolver)
```

resolves the parameters to actual values in the above circuit.
More usefully, Cirq also has the concept of a “sweep”. A sweep is essentially a collection of parameter resolvers. This runtime information is very useful when one wants to run many circuits for many different parameter values. Sweeps can be created to specify values directly (this is one way to get classical information into a circuit), or a variety of helper methods. For example suppose we want to evaluate our circuit over an equally spaced grid of parameter values. We can easily create this using \texttt{LinSpace}.

\begin{verbatim}
 sweep = (cirq.Linspace(key='alpha', start=0.1, stop=0.9, length=5)
 * cirq.Linspace(key='beta', start=0.1, stop=0.9, length=5)
 * cirq.Linspace(key='gamma', start=0.1, stop=0.9, length=5))
 results = simulator.run_sweep(circuit, params=sweep, repetitions=100)
 for result in results:
     print(result.params.param_dict, obj_func(result))
 # prints something like
 # OrderedDict([('alpha', 0.1), ('beta', 0.1), ('gamma', 0.1)]) 6.42
 # OrderedDict([('alpha', 0.1), ('beta', 0.1), ('gamma', 0.30000000000000004)]) 6.48
 # OrderedDict([('alpha', 0.1), ('beta', 0.1), ('gamma', 0.5)]) 6.44
 # OrderedDict([('alpha', 0.1), ('beta', 0.1), ('gamma', 0.70000000000000001)]) 6.58
 # OrderedDict([('alpha', 0.1), ('beta', 0.1), ('gamma', 0.9)]) 6.58
 ... # OrderedDict([('alpha', 0.9), ('beta', 0.9), ('gamma', 0.70000000000000001)]) 0.76
 # OrderedDict([('alpha', 0.9), ('beta', 0.9), ('gamma', 0.9)]) 0.94``
\end{verbatim}

2.2.6 Finding the Minimum

Now we have all the code to we need to do a simple grid search over values to find a minimal value. Grid search is most definitely not the best optimization algorithm, but is here simply illustrative.

\begin{verbatim}
 sweep_size = 10
 sweep = (cirq.Linspace(key='alpha', start=0.0, stop=1.0, length=10)
 * cirq.Linspace(key='beta', start=0.0, stop=1.0, length=10)
 * cirq.Linspace(key='gamma', start=0.0, stop=1.0, length=10))
 results = simulator.run_sweep(circuit, params=sweep, repetitions=100)

 min = None
 min_params = None
 for result in results:
     value = obj_func(result)
     if min is None or value < min:
         min = value
         min_params = result.params
 print('Minimum objective value is \{\}.'.format(min))
 # prints something like
 # Minimum objective value is -1.42.
\end{verbatim}

We’ve created a simple variational quantum algorithm using Cirq. Where to go next? Perhaps you can play around with the above code and work on analyzing the algorithms performance. Add new parameterized circuits and build an end to end program for analyzing these circuits.

2.3 Circuits

2.3.1 Conceptual overview

There are two primary representations of quantum programs in Cirq, each of which are represented by a class: \texttt{Circuit} and \texttt{Schedule}. Conceptually a Circuit object is very closely related to the abstract quantum circuit
model, while a Schedule object is like the abstract quantum circuit model but includes detailed timing information.

Conceptually: a Circuit is a collection of Moments. A Moment is a collection of Operations that all act during the same abstract time slice. An Operation is a some effect that operates on a specific subset of Qubits, the most common type of Operation is a GateOperation.

![Circuit Diagram]

Let’s unpack this.

At the base of this construction is the notion of a qubit. In Cirq, qubits and other quantum objects are identified by instances of subclasses of the Qid base class. Different subclasses of Qid can be used for different purposes. For example the qubits that Google’s Xmon devices use are often arranged on the vertices of a square grid. For this the class GridQubit subclasses Qid. For example, we can create a 3 by 3 grid of qubits using

```python
qubits = [cirq.GridQubit(x, y) for x in range(3) for y in range(3)]
print(qubits[0])  # prints "(0, 0)"
```

The next level up conceptually is the notion of a Gate. A Gate represents a physical process that occurs on a Qubit. The important property of a Gate is that it can be applied on to one or more qubits. This can be done via the Gate.on method itself or via () and doing this turns the Gate into a GateOperation.

```python
# This is an Pauli X gate. It is an object instance.
x_gate = cirq.X
# Applying it to the qubit at location (0, 0) (defined above)
# turns it into an operation.
x_op = x_gate(qubits[0])

print(x_op)  # prints "X((0, 0))"
```
A Moment is quite simply a collection of operations, each of which operates on a different set of qubits, and which conceptually represents these operations as occurring during this abstract time slice. The Moment structure itself is not required to be related to the actual scheduling of the operations on a quantum computer, or via a simulator, though it can be. For example, here is a Moment in which Pauli X and a CZ gate operate on three qubits:

```python
cz = cirq.CZ(qubits[0], qubits[1])
x = cirq.X(qubits[2])
moment = cirq.Moment([x, cz])
print(moment)
# prints "X((0, 2)) and CZ((0, 0), (0, 1))"
```

Note that is not the only way to construct moments, nor even the typical method, but illustrates that a Moment is just a collection of operations on disjoint sets of qubits.

Finally at the top level a Circuit is an ordered series of Moments. The first Moment in this series is, conceptually, contains the first Operations that will be applied. Here, for example, is a simple circuit made up of two moments

```python
cz01 = cirq.CZ(qubits[0], qubits[1])
x2 = cirq.X(qubits[2])
cz12 = cirq.CZ(qubits[1], qubits[2])
moment0 = cirq.Moment([cz01, x2])
moment1 = cirq.Moment([cz12])
circuit = cirq.Circuit((moment0, moment1))
print(circuit)
# prints the text diagram for the circuit:
# (0, 0): @
# (0, 1): @ @
# (0, 2): X @
```

Again, note that this is only one way to construct a Circuit but illustrates the concept that a Circuit is an iterable of Moments.

### 2.3.2 Constructing circuits

Constructing Circuits as a series of Moments with each Moment being hand-crafted is tedious. Instead we provide a variety of different manners to create a Circuit.

One of the most useful ways to construct a Circuit is by appending onto the Circuit with the Circuit.append method.

```python
from cirq.ops import CZ, H
q0, q1, q2 = [cirq.GridQubit(i, 0) for i in range(3)]
circuit = cirq.Circuit()
circuit.append([CZ(q0, q1), H(q2)])
print(circuit)
# prints
# (0, 0): @
# (1, 0): @ @
# (2, 0): H
```

### 2.3. Circuits
This appended an entire new moment to the qubit, which we can continue to do,

```
circuit.append([H(q0), CZ(q1, q2)])

print(circuit)
# prints
# (0, 0): @ H
# (1, 0): @ @
# (2, 0): H @
```

In these two examples, we have appending full moments, what happens when we append all of these at once?

```
circuit = cirq.Circuit()
circuit.append([CZ(q0, q1), H(q2), H(q0), CZ(q1, q2)])

print(circuit)
# prints
# (0, 0): @ H
# (1, 0): @ @
# (2, 0): H @
```

We see that here we have again created two Moments. How did Circuit know how to do this? Circuit's Circuit.append method (and its cousin Circuit.insert) both take an argument called the InsertStrategy. By default the InsertStrategy is InsertStrategy.NEW_THEN_INLINE.

### 2.3.3 InsertStrategies

InsertStrategy defines how Operations are placed in a Circuit when requested to be inserted at a given location. Here a location is identified by the index of the Moment (in the Circuit) where the insertion is requested to be placed at (in the case of Circuit.append this means inserting at the Moment at an index one greater than the maximum moment index in the Circuit). There are four such strategies: InsertStrategy.EARLIEST, InsertStrategy.NEW, InsertStrategy.INLINE and InsertStrategy.NEW_THEN_INLINE.

InsertStrategy.EARLIEST is defined as

- InsertStrategy.EARLIEST: Scans backward from the insert location until a moment with operations touching qubits affected by the operation to insert is found. The operation is added into the moment just after that location.

For example, if we first create an Operation in a single moment, and then use InsertStrategy.EARLIEST the Operation can slide back to this first Moment if there is space:

```
from cirq.circuits import InsertStrategy
circuit = cirq.Circuit()
circuit.append([CZ(q0, q1)])
circuit.append([H(q0), H(q2)], strategy=InsertStrategy.EARLIEST)

print(circuit)
# prints
# (0, 0): @ H
# (1, 0): @ @
# (2, 0): H @
```

(continues on next page)
After creating the first moment with a $\text{CZ}$ gate, the second append uses the `InsertStrategy.EARLIEST` strategy. The $\text{H}$ on $q_0$ cannot slide back, while the $\text{H}$ on $q_2$ can and so ends up in the first moment.

Contrast this with the `InsertStrategy.NEW` InsertStrategy:

```python
circuit = cirq.Circuit() circuit.append([H(q0), H(q1), H(q2)], strategy=InsertStrategy.NEW) print(circuit)
```

Here every operator processed by the append ends up in a new moment. `InsertStrategy.NEW` is most useful when you are inserting a single operation and don’t want it to interfere with other moments.

Another strategy is `InsertStrategy.INLINE`:

```python
circuit = cirq.Circuit() circuit.append([CZ(q1, q2)]) circuit.append([CZ(q1, q2)]) circuit.append([H(q0), H(q1), H(q2)], strategy=InsertStrategy.INLINE) print(circuit)
```

After two initial $\text{CZ}$ between the second and third qubit, we try to insert 3 $\text{H}$ operations. We see that the $\text{H}$ on the first qubit is inserted into the previous moment, but the $\text{H}$ on the second and third qubits cannot be inserted into the previous moment, so a new moment is created.

Finally we turn to the default strategy:

```python
circuit = cirq.Circuit() circuit.append([H(q0)]) circuit.append([CZ(q1,q2), H(q0)], strategy=InsertStrategy.NEW_THEN_INLINE) print(circuit)
```

2.3. Circuits
The first append creates a single moment with a $H$ on the first qubit. Then the append with the `InsertStrategy.NEW_THEN_INLINE` strategy begins by inserting the $CZ$ in a new `Moment` (the `InsertStrategy.NEW` in `InsertStrategy.NEW_THEN_INLINE`). Subsequent appending is done `InsertStrategy.INLINE` so the next $H$ on the first qubit is appending in the just created `Moment`.

Here is a helpful diagram for the different `InsertStrategies`.

```plaintext
# (0, 0): ────────H───────
# (1, 0): ────────H─@
# (2, 0): ────────H─@
```

### 2.3.4 Patterns for Arguments to Append and Insert

Above we have used a series of `Circuit.append` calls with a list of different `Operations` we are adding to the circuit. But the argument where we have supplied a list can also take more than just list values.

Example:

```python
def my_layer():
    yield CZ(q0, q1)
    yield [H(q) for q in (q0, q1, q2)]
    yield [CZ(q1, q2)]
    yield [H(q0), [CZ(q1, q2)]]

circuit = cirq.Circuit()
circuit.append(my_layer())

for x in my_layer():
    print(x)
```

Recall that in Python functions that have a `yield` are `generators`. Generators are functions that act as `iterators`. Above we see that we can iterate over `my_layer()`. We see that when we do this each of the `yields` produces what was yielded, and here these are `Operations`, lists of `Operations` or lists of `Operations` mixed with lists of `Operations`. But when we pass this iterator to the append method, something magical happens. `Circuit` is able to flatten all of these and pass them as one giant list to `Circuit.append` (this also works for `Circuit.insert`).

The above idea uses a concept we call an `OP_TREE`. An `OP_TREE` is not a class, but a contract. The basic idea is that, if the input can be iteratively flattened into a list of operations, then the input is an `OP_TREE`. 
A very nice pattern emerges from this structure: define *generators* for sub-circuits, which can vary by size or Operation parameters.

Another useful method is to construct a Circuit fully formed from an **OP_TREE** is to pass the **OP_TREE** into Circuit when initializing it:

```python
circuit = cirq.Circuit(H(q0), H(q1))
print(circuit)
# prints
# (0, 0): ───H
# (1, 0): ───H
```

### 2.3.5 Slicing and Iterating over Circuits

Circuits can be iterated over and sliced. When they are iterated over each item in the iteration is a moment:

```python
circuit = cirq.Circuit(H(q0), CZ(q0, q1))
for moment in circuit:
    print(moment)
# prints
# H((0, 0))
# CZ((0, 0), (1, 0))
```

Slicing a Circuit on the other hand, produces a new Circuit with only the moments corresponding to the slice:

```python
circuit = cirq.Circuit(H(q0), CZ(q0, q1), H(q1), CZ(q0, q1))
print(circuit[1:3])
# prints
# (0, 0): ──@
# (1, 0): ──H
```

Especially useful is dropping the last moment (which are often just measurements): `circuit[:-1]`, or reversing a circuit: `circuit[::-1]`.

### 2.3.6 Importing from OpenQASM

The QASM importer is in experimental state and currently only supports a subset of the full OpenQASM spec. Amongst others, classical control, arbitrary gate definitions and even some of the gates that don’t have a one-to-one representation in Cirq are not yet supported. The functionality should be sufficient to import interesting quantum circuits. Error handling is very simple - on any lexical or syntactical error a `QasmException` is raised.

**Importing cirq.Circuit from QASM format**

**Requirements:** ply

```bash
pip install ply==3.4
```

The following call will create a circuit defined by the input QASM string:

```python
from cirq.contrib.qasm_import import circuit_from_qasm
circuit = circuit_from_qasm('"
(continues on next page)')
OPENQASM 2.0;
include "qelib1.inc";
qreg q[3];
creg meas[3];
h q;
measure q -> meas;
"

print(circuit)

Results:

q_0: —H—M('meas_0')—
q_1: —H—M('meas_1')—
q_2: —H—M('meas_2')—

**Supported Control statements**

**Gate conversion rules**

Note: The standard Quantum Experience standard gates are defined in \(qelib.inc\) based on the \(U\) and \(CX\) built-in instructions and we could generate them dynamically. Instead, we chose to map them to native cirq gates explicitly which results in a more user friendly circuit.

**Mapping quantum registers to qubits**

For a quantum register \(qreg qfoo[n];\) the QASM importer will create \(cirq.NamedQubits\) named \(qfoo_0..qfoo_{<n-1}>\).

**Mapping classical registers to measurement keys**

For a classical register \(creg cbar[n];\) the QASM importer will create measurement keys named \(cbar_0..cbar_{<n-1}>\).

### 2.4 Gates

A Gate is an operation that can be applied to a collection of qubits (objects with a Qid). Gates can be applied to qubits by calling their on method, or, alternatively calling the gate on the qubits. The object created by such calls is an Operation.

```python
from cirq.ops import CNOT
from cirq.devices import GridQubit
q0, q1 = (GridQubit(0, 0), GridQubit(0, 1))
print(CNOT.on(q0, q1))
print(CNOT(q0, q1))
# prints
# CNOT((0, 0), (0, 1))
# CNOT((0, 0), (0, 1))
```
Gates operate on a specific number of qubits and classes that implement Gate must supply either the _num_qubits_ or _qid_shape_ magic method. For convenience one can use the SingleQubitGate, TwoQubitGate, and ThreeQubitGate classes for these common gate sizes.

The most common type of Gate is one that corresponds to applying a unitary evolution on the qubits that the gate acts on. Gates can also correspond to noisy evolution on the qubits. This version of a gate is not used when sending the circuit to a quantum computer for execution, but it can be used with various simulators. See noise documentation.

### 2.4.1 Magic Methods

A class that implements Gate can be applied to qubits to produce an Operation. In order to support functionality beyond that basic task, it is necessary to implement several magic methods.

Standard magic methods in python are __add__, __eq__, and __len__. Cirq defines several additional magic methods, for functionality such as parameterization, diagramming, and simulation. For example, if a gate specifies a _unitary_ method that returns a matrix for the gate, then simulators will be able to simulate applying the gate. Or, if a gate specifies a _pow_ method that works for an exponent of -1, then cirq.inverse will start to work on lists including the gate.

We describe some magic methods below.

**cirq.num_qubits and def _num_qubits_**

A Gate must implement the _num_qubits_ method. This method returns an integer and is used by cirq.num_qubits to determine how many qubits this gate operates on.

**cirq.qid_shape and def _qid_shape_**

A qudit gate or operation must implement the _qid_shape_ method that returns a tuple of integers. This method is used to determine how many qudits the gate or operation operates on and what dimension each qudit must be. If only the _num_qubits_ method is implemented, the object is assumed to operate only on qubits. Callers can query the qid shape of the object by calling cirq.qid_shape on it. See qudit documentation for more information.

**cirq.unitary and def _unitary_**

When an object can be described by a unitary matrix, it can expose that unitary matrix by implementing a _unitary_(self) -> np.ndarray method. Callers can query whether or not an object has a unitary matrix by calling cirq.unitary on it. The _unitary_ method may also return NotImplemented, in which case cirq.unitary behaves as if the method is not implemented.

**cirq.decompose and def _decompose_**

Operations and gates can be defined in terms of other operations by implementing a _decompose_ method that returns those other operations. Operations implement _decompose_(self) whereas gates implement _decompose_(self, qubits) (since gates don’t know their qubits ahead of time).

The main requirements on the output of _decompose_ methods are:

1. **DO NOT CREATE CYCLES.** The cirq.decompose method will iterative decompose until it finds values satisfying a keep predicate. Cycles cause it to enter an infinite loop.
2. Head towards operations defined by Cirq, because these operations have good decomposition methods that terminate in single-qubit and two qubit gates. These gates can be understood by the simulator, optimizers, and other code.

3. All that matters is functional equivalence. Don’t worry about staying within or reaching a particular gate set; it’s too hard to predict what the caller will want. Gate-set-aware decomposition is useful, but **this is not the protocol that does that**. Gate-set-aware decomposition may be added in the future, but doesn’t exist within Cirq at the moment.

For example, `cirq.CCZ` decomposes into a series of `cirq.CNOT` and `cirq.T` operations. This allows code that doesn’t understand three-qubit operation to work with `cirq.CCZ`; by decomposing it into operations they do understand. As another example, `cirq.TOFFOLI` decomposes into a `cirq.H` followed by a `cirq.CCZ` followed by a `cirq.H`. Although the output contains a three qubit operation (the CCZ), that operation can be decomposed into two qubit and one qubit operations. So code that doesn’t understand three qubit operations can deal with Toffolis by decomposing them, and then decomposing the CCZs that result from the initial decomposition.

In general, decomposition-aware code consuming operations is expected to recursively decompose unknown operations until the code either hits operations it understands or hits a dead end where no more decomposition is possible. The `cirq.decompose` method implements logic for performing exactly this kind of recursive decomposition. Callers specify a keep predicate, and optionally specify intercepting and fallback decomposers, and then `cirq.decompose` will repeatedly decompose whatever operations it was given until the operations satisfy the given keep. If `cirq.decompose` hits a dead end, it raises an error.

Cirq doesn’t make any guarantees about the “target gate set” decomposition is heading towards. `cirq.decompose` is not a method Decompositions within Cirq happen to converge towards X, Y, Z, CZ, PhasedX, specified-matrix gates, and others. But this set will vary from release to release, and so it is important for consumers of decompositions to look for generic properties of gates, such as “two qubit gate with a unitary matrix”, instead of specific gate types such as CZ gates.

**cirq.inverse and __pow__**

Gates and operations are considered to be **invertible** when they implement a `__pow__` method that returns a result besides `NotImplemented` for an exponent of -1. This inverse can be accessed either directly as `value**-1`, or via the utility method `cirq.inverse(value)`. If you are sure that `value` has an inverse, saying `value**-1` is more convenient than saying `cirq.inverse(value)}. cirq.inverse is for cases where you aren’t sure if `value` is invertable, or where `value` might be a sequence of invertible operations.

`cirq.inverse` has a default parameter used as a fallback when `value` isn’t invertable. For example, `cirq.inverse(value, default=None)` returns the inverse of `value`, or else returns `None` if `value` isn’t invertable. (If no default is specified and `value` isn’t invertable, a `TypeError` is raised.)

When you give `cirq.inverse` a list, or any other kind of iterable thing, it will return a sequence of operations that (if run in order) undoes the operations of the original sequence (if run in order). Basically, the items of the list are individually inverted and returned in reverse order. For example, the expression `cirq.inverse([cirq.S(b), cirq.CNOT(a, b)])` will return the tuple `(cirq.CNOT(a, b), cirq.S(b)**-1)`.

Gates and operations can also return values beside `NotImplemented` from their `__pow__` method for exponents besides -1. This pattern is used often by Cirq. For example, the square root of X gate can be created by raising `cirq.X` to 0.5:

```python
import cirq
print(cirq.unitary(cirq.X))
# prints
# [[0.+0.j 1.+0.j]
# [1.+0.j 0.+0.j]]

sqrt_x = cirq.X**0.5
```

(continues on next page)
```python
print(cirq.unitary(sqrt_x))
# prints
#=[[0.5+0.5j 0.5-0.5j]
# [0.5-0.5j 0.5+0.5j]]
```

The Pauli gates included in Cirq use the convention Z**0.5  S  np.diag(1, i), Z**-0.5  S**-1, X**0.5  H•S•H, and the square root of Y is inferred via the right hand rule.

```python
_circuit_diagram_info_(self, args) and cirq.circuit_diagram_info(val, [args], [default])
```

Circuit diagrams are useful for visualizing the structure of a Circuit. Gates can specify compact representations to use in diagrams by implementing a _circuit_diagram_info_ method. For example, this is why SWAP gates are shown as linked ‘×’ characters in diagrams.

The _circuit_diagram_info_ method takes an args parameter of type cirq.CircuitDiagramInfoArgs and returns either a string (typically the gate’s name), a sequence of strings (a label to use on each qubit targeted by the gate), or an instance of cirq.CircuitDiagramInfo (which can specify more advanced properties such as exponents and will expand in the future).

You can query the circuit diagram info of a value by passing it into cirq.circuit_diagram_info.

### 2.4.2 Xmon gates

Google’s Xmon devices support a specific gate set. Gates in this gate set operate on GridQubits, which are qubits arranged on a square grid and which have an x and y coordinate.

The native Xmon gates are

- **cirq.PhasedXPowGate** This gate is a rotation about an axis in the XY plane of the Bloch sphere. The PhasedXPowGate takes two parameters, exponent and phase_exponent. The gate is equivalent to the circuit —Z^-p—X^t—Z^p— where p is the phase_exponent and t is the exponent.

- **cirq.Z / cirq.Rz** Rotations about the Pauli Z axis. The matrix of cirq.Z**t is \(\exp(i \pi |1><1| t)\) whereas the matrix of cirq.Rz() is \(\exp(-i \pi /2)\). Note that in quantum computing hardware, this gate is often implemented in the classical control hardware as a phase change on later operations, instead of as a physical modification applied to the qubits. (TODO: explain this in more detail)

- **cirq.CZ** The controlled-Z gate. A two qubit gate that phases the \(|11\rangle\) state. The matrix of cirq.CZ**t is \(\exp(i \pi |11><11| t)\).

- **cirq.MeasurementGate** This is a single qubit measurement in the computational basis.

### 2.4.3 Other Common Gates

Cirq comes with a number of common named gates:

- **CNOT** the controlled-X gate
- **SWAP** the swap gate
- **H** the Hadamard gate
- **S** the square root of Z gate
- and our error correcting friend the **T** gate
2.5 Noise

For simulation, it is useful to have gates that enact noisy quantum evolution. Cirq supports modeling noise via operator sum representations of noise (these evolutions are also known as quantum operations, quantum dynamical maps, or superoperators). This formalism models evolution of the density matrix via

$$\rho \rightarrow \sum_k A_k \rho A_k^\dagger$$

Where $A_k$ are Krauss operators. These operators are not necessarily unitary and must satisfy the trace preserving property

$$\sum_k A_k^\dagger A_k = I$$

As a noisy channel, Krauss operators are not unique. For more details of these operators see John Preskill’s notes.

2.5.1 Magic methods

A gate can represent an operator sum representation by supporting the channel protocol. Alternatively, for channels that represent probabilistic mixtures of unitaries, one can implement the mixture protocol.

cirq.channel and def channel

To represent an operator sum evolution, a gate should implement the SupportsChannel protocol. To do this, the gate should implement the _channel_(self) -> Sequence[np.ndarray]: method. This method should return the sequence of numpy matrices corresponding to the Krauss operators. The basis in which this matrix is expressed is always implicit with respect to the object being called. For example, in GateOperations, these matrices must be ordered with respect to the list of qubits that the channel is applied to. The qubit-to-amplitude order mapping matches the ordering of numpy.kron(A, B), where $A$ is a qubit earlier in the list than the qubit $B$.

If one has defined _channel_, then that gate and any GateOperation that uses that gate can be used as an argument to cirq.channel and cirq.channel will return this sequence of matrices.

Besides objects that support _channel_, cirq.channel will also fall back to other objects that can be interpreted as channels. For example, if a channel is a probabilistic mixture of unitary gates (see below), then cirq.channel will fall back to seeing if the object supports _mixture_. If _mixture_ is not supported, then cirq.channel checks to see if _unitary_ is supported.

In addition to supporting _channel_, objects that are channels should also implement _has_channel_(self) -> bool to return True. This method is used to determine whether an object has a _channel_ or not without having to do the potentially expensive creation of the matrices for the channel.

cirq.mixture, cirq.mixture_channel, and def mixture

Some channels can be interpreted as probabilistically selecting between different unitary evolutions.

$$\rho \rightarrow \sum_k p_k U_k \rho U_k^\dagger$$

In this case, it is possible to perform Monte Carlo simulations of these gates using a wave function based simulator (and not a density matrix based simulator). Instead of implementing the SupportsChannel protocol, one should implement the SupportsMixture protocol. To do this, one should implement the _mixture_(self) -> Sequence[Tuple[float, np.ndarray]] protocol. This returns a sequence of tuples. The first element of
Cirq Documentation, Release 0.6.1

Each tuple is the probability of the unitary and the second element is the unitary. Like the \_channel\_ method described above, the basis for these matrices is implicit with respect to the object being called. One should also make \_has_mixture\_ return True to indicate to callers that the object supports the mixture protocol. If one wants to get the mixture channel directly, one can call cirq.mixture_channel.

### 2.5.2 Common Channels

Cirq supports many commonly used quantum channels out of the box, see \texttt{ops/common_channels.py}.

**AsymmetricDepolarizingChannel, DepolarizingChannel, BitFlipChannel, and PhaseFlipChannel**

The asymmetric depolarizing channel represents probabilistically selecting one of three Pauli gates to apply or doing nothing to the state. This is implemented via a \_mixture\_ method so that a Monte Carlo simulation with a wave function simulator can be used.

This channel implements the evolution

\[
\rho \rightarrow (1-p_x-p_y-p_z) \rho + p_x X \rho X + p_y Y \rho Y + p_z Z \rho Z
\]

Here \(p_x\) is the probability that the X Pauli gate is applied and no other gate is applied, and similarly for \(p_y\) and \(p_z\).

A particular case of the asymmetric depolarizing channel is the case where each of the different Paulis occur with the same probability. This is encapsulated in the \texttt{DepolarizingChannel} gate, which takes a probability \(p\) such that each Pauli gate occurs with the probability \(p/3\).

To construct channels, useful helpers are provided \texttt{cirq.asymmetric_depolarize} and \texttt{cirq.depolarize}.

Another common case is when only a Pauli X (bit flip) can occur, or when only a Pauli Y (phase flip) can occur. These correspond to BitFlipChannel and PhaseFlipChannel with helpers \texttt{cirq.bit_flip} and \texttt{cirq.phase_flip}.

**GeneralizedAmplitudeDampingChannel and AmplitudeDampingChannel**

The generalized amplitude damping channel models the effect of energy dissipation to a surrounding environment as well as dephasing that does not exchange energy. The amplitude damping channel only models dissipation of energy to a surrounding environment. Cirq has implementations of both of these channels. The generalized amplitude damping channel corresponds to

\[
\begin{aligned}
M_0 =& \sqrt{p} \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1 - \gamma} \end{bmatrix} \\
M_1 =& \sqrt{p} \begin{bmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{bmatrix} \\
M_2 =& \sqrt{1-p} \begin{bmatrix} \sqrt{1-\gamma} & 0 \\ 0 & 1 \end{bmatrix} \\
M_3 =& \sqrt{1-p} \begin{bmatrix} 0 & 0 \\ \sqrt{\gamma} & 0 \end{bmatrix}
\end{aligned}
\]

Where is the probability of the interaction being dissipative and \(p\) is the probability that the qubit and environment exchange energy. The amplitude damping channel corresponds to \(p=1\).

Cirq provides the helpers \texttt{cirq.generalized_amplitude_damp} and \texttt{cirq.amplitude_damp} to construct these noisy gates.

### 2.6 Simulation

Cirq comes with built in Python simulators for testing small circuits. The two main types of simulations that Cirq supports are pure state and mixed state. The pure state simulations are supported by \texttt{cirq.Simulator} and the mixed state simulators are supported by \texttt{cirq.DensityMatrixSimulator}.
The names *pure state simulator* and *mixed state simulators* refers to the fact that these simulations are for quantum circuits; including unitary, measurements, and noise that keeps the evolution in a pure state or a mixed state. In other words, there are some noisy evolutions that are supported by the pure state simulator as long as they preserve the purity of the state.

### 2.6.1 Introduction to pure state simulation

Here is a simple circuit

```python
import cirq
def basic_circuit(meas=True):
    sqrt_x = cirq.X**0.5
    yield sqrt_x(q0), sqrt_x(q1)
    yield cirq.CZ(q0, q1)
    yield sqrt_x(q0), sqrt_x(q1)
    if meas:
        yield cirq.measure(q0, key='q0'), cirq.measure(q1, key='q1')

circuit = cirq.Circuit()
circuit.append(basic_circuit())
print(circuit)
```

We can simulate this by creating a `cirq.Simulator` and passing the circuit into its `run` method:

```python
from cirq import Simulator
simulator = Simulator()
result = simulator.run(circuit)
print(result)
```

Run returns an `TrialResult`. As you can see the result contains the result of any measurements for the simulation run.

The actual measurement results here depend on the seeding `numpy` random seed generator. (You can set this using `numpy.random_seed`) Another run, can result in a different set of measurement results:

```python
result = simulator.run(circuit)
print(result)
```

The simulator is designed to mimic what running a program on a quantum computer is actually like. In particular the `run` methods (run and run_sweep) on the simulator do not give access to the wave function of the quantum computer (since one doesn’t have access to this on the actual quantum hardware). Instead the `simulate` methods
(simulate, simulate_sweep, simulate_moment_steps) should be used if one wants to debug the circuit and get access to the full wave function:

```python
import numpy as np
circuit = cirq.Circuit()
circuit.append(basic_circuit(False))
result = simulator.simulate(circuit, qubit_order=[q0, q1])

print(np.around(result.final_state, 3))
# prints
# [0.5+0.j 0. +0.5j 0. +0.5j 0.5+0.j]
```

Note that the simulator uses numpy's float32 precision (which is complex64 for complex numbers) by default, but that the simulator can take in a a dtype of np.complex128 if higher precision is needed.

### 2.6.2 Qubit and Amplitude Ordering

The `qubit_order` argument to the simulator’s run method determines the ordering of some results, such as the amplitudes in the final wave function. The `qubit_order` argument is optional. When it is omitted, qubits are ordered ascending by their name (i.e. what their `__str__` method returns).

The simplest `qubit_order` value you can provide is a list of the qubits in the desired ordered. Any qubits from the circuit that are not in the list will be ordered using the default `__str__` ordering, but come after qubits that are in the list. Be aware that all qubits in the list are included in the simulation, even if they are not operated on by the circuit.

The mapping from the order of the qubits to the order of the amplitudes in the wave function can be tricky to understand. Basically, it is the same as the ordering used by numpy.kron:

```python
outside = [1, 10]
inside = [1, 2]
print(np.kron(outside, inside))
# prints
# [ 1  2 10 20]
```

More concretely, the $k$’th amplitude in the wave function will correspond to the $k$’th case that would be encountered when nesting loops over the possible values of each qubit. The first qubit’s computational basis values are looped over in the outermost loop, the last qubit’s computational basis values are looped over in the inner-most loop, etc:

```python
i = 0
for first in [0, 1]:
    for second in [0, 1]:
        print('amps[{}] is for first={}, second={}'.format(i, first, second))
        i += 1
# prints
# amps[0] is for first=0, second=0
# amps[1] is for first=0, second=1
# amps[2] is for first=1, second=0
# amps[3] is for first=1, second=1
```

We can check that this is in fact the ordering with a circuit that flips one qubit out of two:

```python
q_stay = cirq.NamedQubit('q_stay')
q_flip = cirq.NamedQubit('q_flip')
c = cirq.Circuit(cirq.X(q_flip))

# first qubit in order flipped
```

(continues on next page)
result = simulator.simulate(c, qubit_order=[q_flip, q_stay])
print(abs(result.final_state).round(3))
# prints
# [0. 0. 1. 0.]

# second qubit in order flipped
result = simulator.simulate(c, qubit_order=[q_stay, q_flip])
print(abs(result.final_state).round(3))
# prints
# [0. 1. 0. 0.]

## 2.6.3 Stepping through a Circuit

Often when debugging it is useful to not just see the end result of a circuit, but to inspect, or even modify, the state of the system at different steps in the circuit. To support this Cirq provides a method to return an iterator over a Moment by Moment simulation. This is the method `simulate_moment_steps`:

```python
for i, step in enumerate(simulator.simulate_moment_steps(circuit)):
    print('state at step %d: %s' % (i, np.around(step.state_vector(), 3)))
# prints something like
# state at step 0: [ 0.5+0.j 0.0+0.5j 0.0+0.5j -0.5+0.j ]
# state at step 1: [ 0.5+0.j 0.0+0.5j 0.0+0.5j 0.5+0.j ]
# state at step 2: [-0.5+0.j -0.0+0.5j -0.0+0.5j -0.5+0.j ]
# state at step 3: [ 0.+0.j 0.+0.j -0.+1.j 0.+0.j]
```

The object returned by the `moment_steps` iterator is a `StepResult`. This object has the state along with any measurements that occurred during that step (so does not include measurement results from previous Moments). In addition, the `StepResult` contains `set_state()` which can be used to set the state. One can pass a valid full state to this method by passing a numpy array. Or alternatively one can pass an integer and then the state will be set entirely in the computation basis state for the binary expansion of the passed integer.

### 2.6.4 Monte Carlo simulations of noise

Some noise models can be thought of as randomly applying unitary evolutions with different probabilities. Such noise models are amenable to Monte Carlo simulation. An example of such a noise model is the bit flip channel. This channel randomly applied either does nothing (identity) or applies a Pauli \( \text{cirq.X} \) gate:

\[
\rho \rightarrow (1-p) \rho + p X \rho X
\]

Let's see a use of this in a simulator

```python
q = cirq.NamedQubit('a')
circuit = cirq.Circuit(cirq.bit_flip(p=0.2)(q), cirq.measure(q))
simulator = cirq.Simulator()
result = simulator.run(circuit, repetitions=100)
print(result.histogram(key='a'))
# prints something like
# Counter({1: 17, 0: 83})
```

As expected, the bit is flipped about 20 percent of the time.
Channels that support this sort of evolution implement the `SupportsMixture` protocol. Also note that this functionality is currently only supported in the pure state simulator and not in the density state simulator. If the mixed state simulator encounters a mixture, it will treat it as a general channel.

### 2.6.5 Parameterized Values and Studies

In addition to circuit gates with fixed values, Cirq also supports gates which can have `Symbol` value (see *Gates*). These are values that can be resolved at run-time. For simulators these values are resolved by providing a `ParamResolver`. A `ParamResolver` provides a map from the `Symbol`’s name to its assigned value.

```python
import sympy
rot_w_gate = cirq.X**sympy.Symbol('x')
circuit = cirq.Circuit()
circuit.append([rot_w_gate(q0), rot_w_gate(q1)])
for y in range(5):
    resolver = cirq.ParamResolver({'x': y / 4.0})
    result = simulator.simulate(circuit, resolver)
    print(np.round(result.final_state, 2))
# prints something like
# [1. +0.j 0.+0.j 0.+0.j 0. +0.j]
# [0.85+0.j 0.-0.35j 0.-0.35j -0.15+0.j]
# [0.5 +0.j 0.-0.5j 0.-0.5j -0.5 +0.j]
# [0.15+0.j 0.-0.35j 0.-0.35j -0.85+0.j]
# [0. +0.j 0.-0.j 0.-0.j -1. +0.j]
```

Here we see that the `Symbol` is used in two gates, and then the resolver provide this value at run time.

Parameterized values are most useful in defining what we call a **Study**. A **Study** is a collection of trials, where each trial is a run with a particular set of configurations and which may be run repeatedly. Running a study returns one `TrialContext` and `TrialResult` per set of fixed parameter values and repetitions (which are reported as the `repetition_id` in the `TrialContext` object). Example:

```python
resolvers = [cirq.ParamResolver({'x': y / 2.0}) for y in range(3)]
circuit = cirq.Circuit()
circuit.append([rot_w_gate(q0), rot_w_gate(q1)])
circuit.append([cirq.measure(q0, key='q0'), cirq.measure(q1, key='q1')])
results = simulator.run_sweep(program=circuit, params=resolvers, repetitions=2)
for result in results:
    print(result)
# prints something like
# repetition_id=0 x=0.0 q0=0 q1=0
# repetition_id=1 x=0.0 q0=0 q1=0
# repetition_id=0 x=0.5 q0=0 q1=1
# repetition_id=1 x=0.5 q0=1 q1=1
# repetition_id=0 x=1.0 q0=1 q1=1
# repetition_id=1 x=1.0 q0=1 q1=1
```

where we see that different repetitions for the case that the qubit has been rotated into a superposition over computational basis states yield different measurement results per run. Also note that we now see the use of the `TrialContext` returned as the first tuple from `run`: it contains the `param_dict` describing what values were actually used in resolving the `Symbols`.

TODO(dabacon): Describe the iterable of parameterized resolvers supported by Google’s API.
2.6.6 Mixed state simulations

In addition to pure state simulation, Cirq also supports simulation of mixed states. Even though this simulator is not as efficient as the pure state simulators, they allow for a larger class of noisy circuits to be run as well as keeping track of the simulation’s density matrix. This later fact can allow for more exact simulations (for example the pure state simulator’s Monte Carlo simulation only allows sampling from the density matrix, not explicitly giving the entries of the density matrix like the mixed state simulator can do). Mixed state simulation is supported by the `cirq.DensityMatrixSimulator` class.

Here is a simple example of simulating a channel using the mixed state simulator:

```python
q = cirq.NamedQubit('a')
circuit = cirq.Circuit(cirq.H(q), cirq.amplitude_damp(0.2)(q), cirq.measure(q))
simulator = cirq.DensityMatrixSimulator()
result = simulator.run(circuit, repetitions=100)
print(result.histogram(key='a'))
# prints something like
# Counter({0: 61, 1: 39})
```

We create a state in an equal superposition of 0 and 1 then apply amplitude damping which takes 1 to 0 with something like a probability of 0.2. We see that instead of about 50 percent of the timing being in 0, about 20 percent of the 1 has been converted into 0, so we end up with total around 60 percent in the 0 state.

Like the pure state simulators, the mixed state simulator supports `run` and `run_sweeps` methods. The `cirq.DensityMatrixSimulator` also supports getting access to the density matrix of the circuit at the end of simulating the circuit, or when stepping through the circuit. These are done by the `simulate` and `simulate_sweep` methods, or, for stepping through the circuit, via the `simulate_moment_steps` method. For example, we can simulate creating an equal superposition followed by an amplitude damping channel with a gamma of 0.2 by:

```python
q = cirq.NamedQubit('a')
circuit = cirq.Circuit(cirq.H(q), cirq.amplitude_damp(0.2)(q))
simulator = cirq.DensityMatrixSimulator()
result = simulator.simulate(circuit)
print(np.around(result.final_density_matrix, 3))
# prints
# [[0.6 +0.j 0.447+0.j]
#  [0.447+0.j 0.4 +0.j]]
```

We see that we have access to the density matrix at the end of the simulation via `final_density_matrix`.

2.7 Schedules and Devices

`Schedule` and `Circuit` are the two major container classes for quantum circuits. In contrast to `Circuit`, a Schedule includes detailed information about the timing and duration of the gates.

Conceptually a Schedule is made up of a set of ScheduledOperations as well as a description of the Device on which the schedule is intended to be run. Each ScheduledOperation is made up of a time when the operation starts and a duration describing how long the operation takes, in addition to the Operation itself (like in a Circuit an Operation is made up of a Gate and the Qids upon which the gate acts.)

2.7.1 Devices

The `Device` class is an abstract class which encapsulates constraints (or lack thereof) that come when running a circuit on actual hardware. For instance, most hardware only allows certain gates to be enacted on qubits. Or, as
another example, some gates may be constrained to not be able to run at the same time as neighboring gates. Further
the Device class knows more about the scheduling of Operations.

Here for example is a Device made up of 10 qubits on a line:

```python
import cirq
from cirq.devices import GridQubit
class Xmon10Device(cirq.Device):
    def __init__(self):
        self.qubits = [GridQubit(i, 0) for i in range(10)]
    def duration_of(self, operation):
        # Wouldn't it be nice if everything took 10ns?
        return cirq.Duration(nanos=10)
    def validate_operation(self, operation):
        if not isinstance(operation.cirq.GateOperation):
            raise ValueError(' {!r} is not a supported operation'.format(operation))
        if not isinstance(operation.gate, (cirq.CZPowGate, cirq.XPowGate, cirq.PhasedXPowGate, cirq.YPowGate)):
            raise ValueError(' {!r} is not a supported gate'.format(operation.gate))
        if len(operation.qubits) == 2:
            p, q = operation.qubits
            if not p.is_adjacent(q):
                raise ValueError('Non-local interaction: {!r}'.format(repr(operation)))
    def validate_scheduled_operation(self, schedule, scheduled_operation):
        self.validate_operation(scheduled_operation.operation)
    def validate_circuit(self, circuit):
        for moment in circuit:
            for operation in moment.operations:
                self.validate_operation(operation)
    def validate_schedule(self, schedule):
        for scheduled_operation in schedule.scheduled_operations:
            self.validate_scheduled_operation(schedule, scheduled_operation)
```

This device, for example, knows that two qubit gates between next-nearest-neighbors is not valid:

```python
device = Xmon10Device()
circuit = cirq.Circuit()
circuit.append([cirq.CZ(device.qubits[0], device.qubits[2])])
try:
    device.validate_circuit(circuit)
except ValueError as e:
    print(e)
# prints something like
# ValueError: Non-local interaction: Operation(cirq.CZ, (GridQubit(0, 0), GridQubit(2, ˓→ 0)))
```
2.7.2 Schedules

A Schedule contains more timing information above and beyond that which is provided by the Moment structure of a Circuit. This can be used both for fine grained timing control, but also to optimize a circuit for a particular device. One can work directly with Schedules or, more common, use a custom scheduler that converts a Circuit to a Schedule. A simple example of such a scheduler is the moment_by_moment_schedule method of schedulers.py. This scheduler attempts to keep the Moment structure of the underlying Circuit as much as possible: each Operation in a Moment is scheduled to start at the same time (such a schedule may not be possible, in which case this method raises an exception.)

Here, for example, is a simple Circuit on the Xmon10Device defined above

```python
circuit = cirq.Circuit()
circuit.append([cirq.CZ(device.qubits[0], device.qubits[1]), cirq.X(device.qubits[0])])
print(circuit)
# prints:
# (0, 0):   @  X
# (1, 0):   @
```

This can be converted over into a schedule using the moment by moment schedule

```python
schedule = cirq.moment_by_moment_schedule(device, circuit)
```

Schedules have an attributed scheduled_operations which contains all the scheduled operations in a SortedListWithKey, where the key is the start time of the SortedOperation. Schedules support nice helpers for querying about the time-space layout of the schedule. For instance, the Schedule behaves as if it has an index corresponding to time. So, we can look up which operations occur at a specific time

```python
print(schedule[cirq.Timestamp(nanos=15)])
# prints something like
# [ScheduledOperation(Timestamp(picos=10000), Duration(picos=10000),...)]
```

or even a start and end time using slicing notation

```python
slice = schedule[cirq.Timestamp(nanos=5):cirq.Timestamp(nanos=15)]
slice_schedule = cirq.Schedule(device, slice)
print(slice_schedule == schedule)
# prints True
```

More complicated queries across Schedules can be done using the query.

Schedules are usually built by converting from Circuits, but one can also directly manipulate the schedule using the include and exclude methods. include will check if there are any collisions with other schedule operations.

2.8 Qudits

Most of the time in quantum computation, we work with qubits which are 2-level quantum systems. A qu-\(d\)-it is a generalization of a qubit to a \(d\)-level or \(d\)-dimension system. Qudits with known values for \(d\) have specific names. A qubit has dimension 2, a qutrit has dimension 3, a ququart has dimension 4, and so on. In Cirq, qudits work exactly like qubits except they have a dimension attribute other than 2 and they can only be used with gates specific to that dimension. Both qubits and qudits are represented by a Qid object.
To apply a gate to some qudits, the dimensions of the qudits must match the dimensions it works on. For example if a gate represents a unitary evolution on three qudits, a qubit, a qutrit, and another qutrit, the gate’s “qid shape” is \((2, 3, 3)\) and its `on` method will accept exactly 3 Qids with dimension 2, 3, and 3. This is an example single qutrit gate used in a circuit:

```python
class QutritPlusGate(cirq.SingleQubitGate):
    def _qid_shape_(self):
        return (3,)
    def _unitary_(self):
        return np.array([[0, 0, 1],
                         [1, 0, 0],
                         [0, 1, 0]])
    def _circuit_diagram_info_(self, args):
        return '+1'
q0 = cirq.LineQid(0, dimension=3)
circuit = cirq.Circuit(QutritPlusGate().on(q0))
print(circuit)
# prints
# 0 (d=3):
#     +1
```

### 2.8.1 Qids

Qid is the type representing qubits and qudits. By default a qid like `cirq.NamedQubit('a')` is a qubit. To create a qutrit named ‘a’, specify the dimension with `cirq.NamedQubit('a').with_dimension(3)`. In addition, the `LineQid` constructor supports a dimension argument directly `cirq.LineQid(0, dimension=4)`.

### 2.8.2 `cirq.qid_shape` and `def _qid_shape_`

Quantum gates, operations, and other types that act on a sequence of qudits can specify the dimension of each qudit they act on by implementing the `_qid_shape_` magic method. This method returns a tuple of integers corresponding to the required dimension of each qudit it operates on e.g. \((2, 3, 3)\) means an object that acts on a qubit, a qutrit, and another qutrit.

When Qids are used with Gates, Operations, and Circuits, the dimension of each qid must match the corresponding entry in the qid shape. An error is raised otherwise.

Callers can query the qid shape of an object or a list of Qids by calling `cirq.qid_shape` on it. By default, `cirq.qid_shape` will return the equivalent qid shape for qubits if `_qid_shape_` is not defined. For a qubit-only gate the qid shape is a tuple of 2s containing one 2 for each qubit e.g. \((2,) \times \text{cirq.num_qubits(gate)}\).

### 2.8.3 Unitaries, Mixtures, and Channels

The magic methods `_unitary_`, `_apply_unitary_`, `_mixture_`, and `_channel_` used to define unitary operations, mixtures, and channels can be used with qudits with one caveat. The matrix dimensions for qudits will be larger than for qubits based on the values of the qudit dimensions (the object’s qid shape). The size of the matrix is determined from the product of the qudit dimensions. For example, a single qubit unitary is a 2x2 matrix whereas a single qutrit unitary is a 3x3 matrix. A two qutrit unitary is a 9x9 matrix \((3 \times 3 = 9)\) and a qubit-ququart unitary is a 8x8 matrix \((2 \times 4 = 8)\). The size of the matrices for mixtures and channels follow the same rule.
2.8.4 Simulators and Samplers

Simulators like `cirq.Simulator` and `cirq.DensityMatrixSimulator` will return simulation results with larger matrices than the same size qubit circuit when simulating qudit circuits. The size of the matrix is determined by the product of the dimensions of the qudits being simulated. The state vector output of `cirq.Simulator` after simulating a circuit on a qubit, a qutrit, and a qutrit will have \(2 \times 3 \times 3 = 18\) elements. Call `cirq.qid_shape(simulation_result)` to check the qudit dimensions.

Measurement results from running a qudit circuit are integers in the range 0 to `qid.dimension-1`.

2.9 Development

This document is a summary of how to do various tasks one runs into as a developer of Cirq. Note that all commands assume a Debian environment, and all commands (except the initial repository cloning command) assume your current working directory is the cirq repo root.

2.9.1 Cloning the repository

The simplest way to get a local copy of cirq that you can edit is by cloning Cirq’s github repository:

```bash
git clone git@github.com:quantumlib/cirq.git
cd cirq
```

To do your development in a Docker image, you can build one with Cirq/dev_tools/Dockerfile or pull an existing image:

```bash
docker pull quantumlib/cirq:dev
docker run -it quantumlib/cirq:dev python -c "import cirq; print(cirq.google.Foxtail)"
```

If you want to contribute changes to Cirq, you will instead want to fork the repository and submit pull requests from your fork.

2.9.2 Forking the repository

1. Fork the Cirq repo (Fork button in upper right corner of repo page). Forking creates a new github repo at the location https://github.com/USERNAME/cirq where USERNAME is your github id.

2. Clone the fork you created to your local machine at the directory where you would like to store your local copy of the code, and cd into the newly created directory.

   ```bash
git clone git@github.com:USERNAME/cirq.git
cd cirq
   ```

   (Alternatively, you can clone the repository using the URL provided on your repo page under the green “Clone or Download” button)

3. Add a remote called `upstream` to git. This remote will represent the main git repo for cirq (as opposed to the clone, which you just created, which will be the `origin` remote). This remote can be used to merge changes from Cirq’s main repository into your local development copy.

   ```bash
git remote add upstream https://github.com/quantumlib/cirq.git
```

To verify the remote, run `git remote -v`. You should see both the `origin` and `upstream` remotes.
4. Sync up your local git with the upstream remote:

```
  git fetch upstream
```

You can check the branches that are on the upstream remote by running `git remote -va` or `git branch -r`. Most importantly you should see `upstream/master` listed.

5. Merge the upstream master into your local master so that it is up to date.

```
  git checkout master
  git merge upstream/master
```

At this point your local git master should be synced with the master from the main cirq repo.

### 2.9.3 Setting up an environment

1. First clone the repository, if you have not already done so. See the previous section for instructions.

2. Install system dependencies.

   Make sure you have python 3.5 or greater. You can install most other dependencies via `apt-get`:

   ```bash
   cat apt-system-requirements.txt dev_tools/conf/apt-list-dev-tools.txt | xargs sudo apt-get install --yes
   ```

   There are some extra steps if protocol buffers are changed; see the next section.

3. Prepare a virtual environment including the dev tools (such as mypy).

   One of the system dependencies we installed was `virtualenvwrapper`, which makes it easy to create virtual environments. If you did not have `virtualenvwrapper` previously, you may need to re-open your terminal or run `source ~/.bashrc` before these commands will work:

   ```bash
   mkvirtualenv cirq-py3 --python=/usr/bin/python3
   python -m pip install --upgrade pip
   python -m pip install -e .[dev_env]
   ```

   (When you later open another terminal, you can activate the virtualenv with `workon cirq-py3`.)

4. Check that the tests pass.

   ```bash
   pytest .
   ```

5. *(OPTIONAL)* include your development copy of cirq in your python path.

   ```bash
   PYTHONPATH="$(pwd):${PYTHONPATH}"
   ```

   or add it to the python path, but only in the virtualenv.

   ```bash
   add2virtualenv ./
   ```

### 2.9.4 Protocol buffers

Protocol buffers are used in Cirq for converting circuits, gates, and other objects into a standard form that can be written and read by other programs. Cirq’s protobufs live at `cirq/api/google` and may need to be changed or extended from time to time.
If any protos are updated, their dependents can be rebuilt by calling the script `dev_tools/build-protos.sh`. This script uses grpcio-tools and protobuf version 3.8.0 to generate the python proto api.

Additionally, for workflows that use bazel (relevant for C/C++ code depending on Cirq), we have made available bazel rulesets for generating both python and C/C++ proto apis. These rules live in the BUILD files here and here. Downstream projects should load Cirq as an external dependency, allowing rules from those BUILD files to be used directly.

### 2.9.5 Continuous integration and local testing

There are a few options for running continuous integration checks, varying from easy and fast to slow and reliable.

The simplest way to run checks is to invoke `pytest`, `pylint`, or `mypy` for yourself as follows:

```
pytest
pylint --rcfile=dev_tools/conf/.pylintrc cirq
mypy --config-file=dev_tools/conf/mypy.ini
```

This can be a bit tedious, because you have to specify the configuration files each time. A more convenient way to run checks is to via the scripts in the `check/` directory, which specify configuration arguments for you and cover more use cases:

```
# Run all tests in the repository.
./check/pytest [files-and-flags-for-pytest]

# Check all relevant files in the repository for lint.
./check/pylint [files-and-flags-for-pylint]

# Typecheck all python files in the repository.
./check/mypy [files-and-flags-for-mypy]

# Compute incremental coverage vs master (or a custom revision of your choice).
./check/pytest-and-incremental-coverage [BASE_REVISION]

# Only run tests associated with files that have changed when diffed vs master (or a custom revision of your choice).
./check/pytest-changed-files [BASE_REVISION]

# Run the documentation tests.
./check/doctest

# Check the format of the files. Use --apply to apply the suggested format changes.
./check/format-incremental [--apply]

# Run all of the above tests. Which pytest is run is set by the --only-changed-files.
./check/all [BASE_REVISION] [--only-changed-files] [--apply-format-changes]
```

The above scripts are convenient and reasonably fast, but they often won’t exactly match the results computed by the continuous integration builds run on Travis. For example, you may be running an older version of `pylint` or `numpy`. If you need to test against the actual continuous integration check, open up a pull request. For this pull request you may want to mark it as `[Testing]` so that it is not reviewed.

### 2.9.6 Writing docstrings and generating documentation

Cirq uses Google style doc strings with a markdown flavor and support for latex. Here is an example docstring:
```python
def some_method(a: int, b: str) -> float:
    r'''One line summary of method.

    Additional information about the method, perhaps with some sort of latex
    equation to make it clearer:
    $$
    M = \begin{bmatrix}
    0 & 1 \\
    1 & 0
    \end{bmatrix}
    $$
    Notice that this docstring is an r-string, since the latex has backslashes.
    We can also include example code:
    ```python
    print(cirq.google.Foxtail)
    ```
    You can also do inline latex like $y = x^2$ and inline code like
    `cirq.unitary(cirq.X)`.
    And of course there's the standard sections.
    Args:
    a: The first argument.
    b: Another argument.
    Returns:
    An important value.
    Raises:
    ValueError: The value of `a` wasn't quite right.
    '''
```

Documentation is generated automatically by readthedocs when pushing to master, but you can also generated a
local copy by running:

```bash
dev_tools/build-docs.sh
```

The HTML output will go into the docs/_build directory.

### 2.9.7 Producing a pypi package

1. Do a dry run with test pypi.

   If you’re making a release, you should have access to a test pypi account capable of uploading packages to cirq.
   Put its credentials into the environment variables TEST_TWINE_USERNAME and TEST_TWINE_PASSWORD
   then run

   ```bash
   ./dev_tools/packaging/publish-dev-package.sh EXPECTED_VERSION --test
   ```

   You must specify the EXPECTED_VERSION argument to match the version in cirq/_version.py, and it
   must contain the string dev. This is to prevent accidentally uploading the wrong version.

   The script will append the current date and time to the expected version number before uploading to test pypi.
   It will print out the full version that it uploaded. Take not of this value.

2.9. Development
Once the package has uploaded, verify that it works

```bash
./dev_tools/packaging/verify-published-package.sh FULL_VERSION_REPORTED_BY_PUBLISH_SCRIPT --test
```

The script will create fresh virtual environments, install cirq and its dependencies, check that code importing cirq executes, and run the tests over the installed code. If everything goes smoothly, the script will finish by printing **VERIFIED**.

2. Do a dry run with prod pypi

This step is essentially identical to the test dry run, but with production pypi. You should have access to a production pypi account capable of uploading packages to cirq. Put its credentials into the environment variables `PROD_TWINE_USERNAME` and `PROD_TWINE_PASSWORD` then run

```bash
./dev_tools/packaging/publish-dev-package.sh EXPECTED_VERSION --prod
```

Once the package has uploaded, verify that it works

```bash
./dev_tools/packaging/verify-published-package.sh FULL_VERSION_REPORTED_BY_PUBLISH_SCRIPT --prod
```

If everything goes smoothly, the script will finish by printing **VERIFIED**.

3. Set the version number in `cirq/_version.py`.

Development versions end with `.dev` or `.dev#`. For example, `0.0.4.dev500` is a development version of the release version `0.0.4`. For a release, create a pull request turning `#.#.#.dev*` into `#.#.#` and a follow up pull request turning `#.#.#` into `(#+1).#.#.dev`.

4. Run `dev_tools/packaging/produce-package.sh` to produce pypi artifacts.

```bash
dev_tools/packaging/produce-package.sh dist
```

The output files will be placed in the directory `dist/`.

5. Create a github release.

Describe major changes (especially breaking changes) in the summary. Make sure you point the tag being created at the one and only revision with the non-dev version number. Attach the package files you produced to the release.

6. Upload to pypi.

You can use a tool such as `twine` for this. For example:

```bash
twine upload -u "${PROD_TWINE_USERNAME}" -p "${PROD_TWINE_PASSWORD}" dist/*
```

You should then run the verification script to check that the uploaded package works:

```bash
./dev_tools/packaging/verify-published-package.sh VERSION_YOU_UPLOADED --prod
```

And try it out for yourself:

```bash
python -m pip install cirq
python -c "import cirq; print(cirq.google.Foxtail)"
python -c "import cirq; print(cirq.__version__)"
```
Cirq Documentation, Release 0.6.1

2.10 Examples
2.10.1 BCS Mean Field
# coding=utf-8
r"""Quantum circuit to prepare the BCS ground states for
superconductors/superfluids. Such states can be prepared by
applying pairwise Bogoliubov transformations on basis states
with opposite spins and momenta, followed by the fermionic Fourier
transformations. In this simple example, we consider a 1D 4-site Hubbard model.
The fermionic quantum state is mapped that of a qubit ladder (two coupled
chains) using the Jordan-Wigner transformation, the upper (lower) chain
represent spin-up (down) basis states.
The Bogoliubov transformation can be readily implemented by
applying quantum gates on vertical pairs of qubits, which takes the form
|BCS = \prod_k (u_k + v_k c^\dag_{k,↑} c^\dag_{k,↓}|vac where |vac is
the vacuum state and u_k^2 = (1+ _k/(_k^2+_k^2)^{1/2})/2 and v_k^2
= (1 - _k/(_k^2+_k^2)^{1/2})/2.
We use the fast fermionic Fourier transformation (FFFT) to implement the basis
transformation from the momentum picture to the position picture.
This is an attempt to reduce the number of the gates that have to be
calibrated in experiments (compared to the Givens rotation approach); one
only needs to calibrate a couple of two-qubit gates using FFFT, i.e.,
the iSWAP gate and its square root iSWAP. We use the single-qubit S gate to
convert the iSWAP gate and the iSWAP gate to fermionic gates.
=== REFERENCE ===
F. Verstraete, J. I. Cirac, and J. I. Latorre, “Quantum circuits for strongly
Zhang Jiang, Kevin J. Sung, Kostyantyn Kechedzhi, Vadim N. Smelyanskiy,
=== EXAMPLE OUTPUT ===
Quantum circuits to prepare the BCS mean field state.
Number of sites = 4
Number of fermions = 4
Tunneling strength = 1.0
On-site interaction strength = -4.0
Superconducting gap = 1.1261371093950703
Circuit for the Bogoliubov transformation:
(0, 0) (0, 1) (0, 2) (0, 3) (1, 0)

(1, 1)

(1, 2)

(1, 3)

W(.25)
iSwap

iSwap^-1.83

W(.625)
W(.25)
iSwap

iSwap^-1.67
(continues on next page)

2.10. Examples

43


Circuit for the inverse fermionic Fourier transformation on the spin-up states:
\[
\begin{align*}
\text{(0, 0)} & \quad \text{(0, 1)} & \quad \text{(0, 2)} & \quad \text{(0, 3)} \\
S^{-1} & \quad i\text{Swap} & \quad i\text{Swap} & \quad S^{-1} & \quad 2 \\
i\text{Swap} & \quad i\text{Swap}^0.5 & \quad i\text{Swap} & \quad i\text{Swap}^0.5 & \quad S^{-1} \\
i\text{Swap} & \quad i\text{Swap} & \quad S^{-1} & \quad S^{-1} & \quad i\text{Swap} & \quad i\text{Swap}^0.5 & \quad S^{-1} & \quad i\text{Swap}^0.5 & \quad S^{-1} & \quad i\text{Swap} \quad S^{-1} & \quad S^{-1} & \quad i\text{Swap} \quad i\text{Swap}^0.5 \\
\end{align*}
\]

Circuit for the inverse fermionic Fourier transformation on the spin-down states:
\[
\begin{align*}
\text{(1, 0)} & \quad \text{(1, 1)} & \quad \text{(1, 2)} & \quad \text{(1, 3)} \\
S^{-1} & \quad i\text{Swap} & \quad i\text{Swap} & \quad S^{-1} & \quad 2 \\
i\text{Swap} & \quad i\text{Swap}^0.5 & \quad i\text{Swap} & \quad i\text{Swap}^0.5 & \quad S^{-1} \\
i\text{Swap} & \quad S^{-1} & \quad S^{-1} & \quad S^{-1} & \quad S^{-1} & \quad S^{-1} & \quad S^{-1} & \quad S^{-1} \\
\end{align*}
\]
import numpy as np
import scipy.optimize
import cirq

def main():
    # Number of sites in the Fermi-Hubbard model (2*n_site spin orbitals)
    n_site = 4
    # Number of fermions
    n_fermi = 4
    # Hopping strength between neighboring sites
    t = 1.
    # On-site interaction strength. It has to be negative (attractive) for the
    # BCS theory to work.
    u = -4.
    # Calculate the superconducting gap and the angles for BCS
    delta, bog_theta = bcs_parameters(n_site, n_fermi, u, t)
    # Initializing the qubits on a ladder
    upper_qubits = [cirq.GridQubit(0, i) for i in range(n_fermi)]
    lower_qubits = [cirq.GridQubit(1, i) for i in range(n_fermi)]

    print('Quantum circuits to prepare the BCS meanfield state.')
    print('Number of sites = ', n_site)
    print('Number of fermions = ', n_fermi)
    print('Tunneling strength = ', t)
    print('On-site interaction strength = ', u)
    print('Superconducting gap = ', delta, '\n')

    bog_circuit = cirq.Circuit(
        bogoliubov_trans(upper_qubits[i], lower_qubits[i], bog_theta[i])
        for i in range(n_site))
    bog_circuit = cirq.google.optimized_for_xmon(bog_circuit)

    # (continues on next page)
print('Circuit for the Bogoliubov transformation:
)
print(bog_circuit.to_text_diagram(transpose=True), 'n')

# The inverse fermionic Fourier transformation on the spin-up states
print('Circuit for the inverse fermionic Fourier transformation on the ' 'spin-up states:',
)
fourier_circuit_spin_up = cirq.Circuit(
    fermi_fourier_trans_inverse_4(upper_qubits),
    strategy=cirq.InsertStrategy.EARLIEST)
fourier_circuit_spin_up = cirq.google.optimized_for_xmon(fourier_circuit_spin_up)
print(fourier_circuit_spin_up.to_text_diagram(transpose=True), 'n')

# The inverse fermionic Fourier transformation on the spin-down states
print('Circuit for the inverse fermionic Fourier transformation on the ' 'spin-down states:',
)
fourier_circuit_spin_down = cirq.Circuit(
    fermi_fourier_trans_inverse_conjugate_4(lower_qubits),
    strategy=cirq.InsertStrategy.EARLIEST)
fourier_circuit_spin_down = cirq.google.optimized_for_xmon(fourier_circuit_spin_down)
print(fourier_circuit_spin_down.to_text_diagram(transpose=True))

def fswap(p, q):
    """Decompose the Fermionic SWAP gate into two single-qubit gates and
    one iSWAP gate.
    Args:
        p: the id of the first qubit
        q: the id of the second qubit
    """

    yield cirq.ISWAP(q, p), cirq.Z(p) ** 1.5
    yield cirq.Z(q) ** 1.5

def bogoliubov_trans(p, q, theta):
    r"""The 2-mode Bogoliubov transformation is mapped to two-qubit operations. We use the identity X S^dagi S X X = Y X S^dag Y S X = X to transform the Hamiltonian XY+XY to XX+YY type. The time evolution of the XX + YY Hamiltonian can be expressed as a power of the iSWAP gate.
    Args:
        p: the first qubit
        q: the second qubit
        theta: The rotational angle that specifies the Bogoliubov transformation, which is a function of the kinetic energy and the superconducting gap.
    """

    # The iSWAP gate corresponds to evolve under the Hamiltonian XX+YY for # time -pi/4.
    expo = -4 * theta / np.pi

    yield cirq.X(p)
    yield cirq.S(p)
yield cirq.ISWAP(p, q)**expo
yield cirq.S(p)**1.5
yield cirq.X(p)

def fermi_fourier_trans_2(p, q):
    """The 2-mode fermionic Fourier transformation can be implemented straightforwardly by the iSWAP gate. The iSWAP gate can be readily implemented with the gmon qubits using the XX + YY Hamiltonian. The matrix representation of the 2-qubit fermionic Fourier transformation is:
    \[
    \begin{bmatrix}
    1 & 0 & 0 & 0 \\
    0 & 1/2 & 1/2 & 0 \\
    0 & 1/2 & -1/2 & 0 \\
    0 & 0 & 0 & -1
    \end{bmatrix}
    \]
    The square root of the iSWAP gate is:
    \[
    \begin{bmatrix}
    1 & 0 & 0 & 0 \\
    0 & 0.5 + 0.5j & 0.5 - 0.5j & 0 \\
    0 & 0.5 - 0.5j & 0.5 + 0.5j & 0 \\
    0 & 0 & 0 & 1
    \end{bmatrix}
    \]
    Args:
    p: the first qubit
    q: the second qubit
    ""
    yield cirq.Z(p)**1.5
    yield cirq.ISWAP(q, p)**0.5
    yield cirq.Z(p)**1.5

def fermi_fourier_trans_inverse_4(qubits):
    """The reverse fermionic Fourier transformation implemented on 4 qubits on a line, which maps the momentum picture to the position picture. Using the fast Fourier transformation algorithm, the circuit can be decomposed into 2-mode fermionic Fourier transformation, the fermionic SWAP gates, and single-qubit rotations.
    Args:
    qubits: list of four qubits
    ""
    yield fswap(qubits[1], qubits[2]),
    yield fermi_fourier_trans_2(qubits[0], qubits[1])
    yield fermi_fourier_trans_2(qubits[2], qubits[3])
    yield fswap(qubits[1], qubits[2])
    yield fermi_fourier_trans_2(qubits[0], qubits[1])
    yield cirq.S(qubits[2])
    yield fermi_fourier_trans_2(qubits[2], qubits[3])
    yield fswap(qubits[1], qubits[2])

def fermi_fourier_trans_inverse_conjugate_4(qubits):
    """We will need to map the momentum states in the reversed order for spin-down states to the position picture. This transformation can be simply implemented the complex conjugate of the former one. We only need to change the S gate to \( S^* = S^{**3} \).
    ""
    (continues on next page)
def bcs_parameters(n_site, n_fermi, u, t):
    """Generate the parameters for the BCS ground state, i.e., the superconducting gap and the rotational angles in the Bogoliubov transformation.
    
    Args:
    n_site: the number of sites in the Hubbard model
    n_fermi: the number of fermions
    u: the interaction strength
    t: the tunneling strength
    
    Returns:
    float delta, List[float] bog_theta
    """

    # The wave numbers satisfy the periodic boundary condition.
    wave_num = np.linspace(0, 1, n_site, endpoint=False)
    # The hopping energy as a function of wave numbers
    hop_erg = -2 * t * np.cos(2 * np.pi * wave_num)
    # Finding the Fermi energy
    fermi_erg = hop_erg[n_fermi // 2]
    # Set the Fermi energy to zero
    hop_erg = hop_erg - fermi_erg
    
    def _bcs_gap(x):
        """Defines the self-consistent equation for the BCS wavefunction.
        
        Args:
        x: the superconducting gap
        """

        s = 0.
        for i in range(n_site):
            s += 1. / np.sqrt(hop_erg[i] ** 2 + x ** 2)
        return 1 + s * u / (2 * n_site)

    # Superconducting gap
    delta = scipy.optimize.bisect(_bcs_gap, 0.01, 10000. * abs(u))
    # The amplitude of the double excitation state
    bcs_v = np.sqrt(0.5 * (1 - hop_erg / np.sqrt(hop_erg ** 2 + delta ** 2)))
    # The rotational angle in the Bogoliubov transformation.
    bog_theta = np.arcsin(bcs_v)
return delta, bog_theta

if __name__ == "__main__":
    main()

2.10.2 Bell Inequality


import numpy as np
import cirq

def main():
    # Create circuit.
    circuit = make_bell_test_circuit()
    print('Circuit:')
    print(circuit)

    # Run simulations.
    print()  
    repetitions = 75
    print('Simulating {} repetitions...'.format(repetitions))
    result = cirq.Simulator().run(program=circuit, repetitions=repetitions)

    # Collect results.
    a = np.array(result.measurements['a'][:, 0])
    b = np.array(result.measurements['b'][:, 0])
    x = np.array(result.measurements['x'][:, 0])
    y = np.array(result.measurements['y'][:, 0])
outcomes = a ^ b == x & y
win_percent = len([e for e in outcomes if e]) * 100 / repetitions

# Print data.
print()
print('Results')
print('a:', bitstring(a))
print('b:', bitstring(b))
print('x:', bitstring(x))
print('y:', bitstring(y))
print('(a XOR b) == (x AND y):

', bitstring(outcomes))
print('Win rate: {} \%'.format(win_percent))

def make_bell_test_circuit():
    alice = cirq.GridQubit(0, 0)
bob = cirq.GridQubit(1, 0)
alice_referee = cirq.GridQubit(0, 1)
bob_referee = cirq.GridQubit(1, 1)
circuit = cirq.Circuit()

    # Prepare shared entangled state.
    circuit.append([
        cirq.H(alice),
cirq.CNOT(alice, bob),
cirq.X(alice)**-0.25,
    ])

    # Referees flip coins.
    circuit.append([
        cirq.H(alice_referee),
cirq.H(bob_referee),
    ])

    # Players do a sqrt(X) based on their referee's coin.
    circuit.append([
        cirq.CNOT(alice_referee, alice)**0.5,
cirq.CNOT(bob_referee, bob)**0.5,
    ])

    # Then results are recorded.
    circuit.append([
        cirq.measure(alice, key='a'),
cirq.measure(bob, key='b'),
cirq.measure(alice_referee, key='x'),
cirq.measure(bob_referee, key='y'),
    ])

    return circuit

def bitstring(bits):
    return ''.join('1' if e else '_' for e in bits)

if __name__ == '__main__':
2.10.3 Bernstein Vazirani

"""Demonstrates the Bernstein-Vazirani algorithm.

The (non-recursive) Bernstein-Vazirani algorithm takes a black-box oracle implementing a function \( f(a) = a \cdot \text{factors} + \text{bias} \) (mod 2), where 'bias' is 0 or 1, 'a' and 'factors' are vectors with all elements equal to 0 or 1, and the algorithm solves for 'factors' in a single query to the oracle.

=== REFERENCE ===


=== EXAMPLE OUTPUT ===

Secret function:
\( f(a) = a \cdot \langle 0, 1, 1, 0, 1, 0, 0, 1 \rangle + 1 \) (mod 2)
Circuit:
(0, 0): \[ H \quad H \quad M \]
(1, 0): \[ H \quad @ \quad H \quad M \]
(2, 0): \[ H \quad @ \quad H \quad M \]
(3, 0): \[ H \quad @ \quad H \quad M \]
(4, 0): \[ H \quad @ \quad H \quad M \]
(5, 0): \[ H \quad @ \quad H \quad M \]
(6, 0): \[ H \quad @ \quad H \quad M \]
(7, 0): \[ H \quad @ \quad H \quad M \]
(8, 0): \[ X \quad H \quad X \quad X \quad X \quad X \quad X \quad X \quad M \]

Sampled results:
Counter({'01110010': 3})
Most common matches secret factors:
True
"""

import random

import cirq

def main(qubit_count = 8):
    circuit_sample_count = 3

    # Choose qubits to use.
    input_qubits = [cirq.GridQubit(i, 0) for i in range(qubit_count)]
    output_qubit = cirq.GridQubit(qubit_count, 0)
# Pick coefficients for the oracle and create a circuit to query it.
secret_bias_bit = random.randint(0, 1)
secret_factor_bits = [random.randint(0, 1) for _ in range(qubit_count)]
oracle = make_oracle(input_qubits, output_qubit, secret_factor_bits, secret_bias_bit)
print('Secret function:

\[ f(a) = a \cdot \langle{} \rangle + \langle{} \rangle \pmod{2} \]
'.format(', '.join(str(e) for e in secret_factor_bits), secret_bias_bit))

# Embed the oracle into a special quantum circuit querying it exactly once.
circuit = make_bernstein_vazirani_circuit(input_qubits, output_qubit, oracle)
print('Circuit:)
print(circuit)

# Sample from the circuit a couple times.
simulator = cirq.Simulator()
result = simulator.run(circuit, repetitions=circuit_sample_count)
frequencies = result.histogram(key='result', fold_func=bitstring)
print('Sampled results:

\[
\]
'.format(frequencies))

# Check if we actually found the secret value.
most_common_bitstring = frequencies.most_common(1)[0][0]
print('Most common matches secret factors:

\[
\]
'.format(most_common_bitstring == bitstring(secret_factor_bits)))

def make_oracle(input_qubits, output_qubit, secret_factor_bits, secret_bias_bit):
    """Gates implementing the function \( f(a) = a \cdot \text{factors} + \text{bias} \pmod{2} \).""
    if secret_bias_bit:
        yield cirq.X(output_qubit)
    for qubit, bit in zip(input_qubits, secret_factor_bits):
        if bit:
            yield cirq.CNOT(qubit, output_qubit)

def make_bernstein_vazirani_circuit(input_qubits, output_qubit, oracle):
    """Solves for factors in \( f(a) = a \cdot \text{factors} + \text{bias} \pmod{2} \) with one query.""
    c = cirq.Circuit()
    # Initialize qubits.
c.append([
        cirq.X(output_qubit),
        cirq.H(output_qubit),
        cirq.H.on_each(*input_qubits),
    ])
    # Query oracle.
```
c.append(oracle)

# Measure in X basis.
c.append([
    cirq.H.on_each(*input_qubits),
    cirq.measure(*input_qubits, key='result')
])

return c

def bitstring(bits):
    return ''.join(str(int(b)) for b in bits)

if __name__ == '__main__':
    main()
```

### 2.10.4 Grover

"""Demonstrates Grover algorithm.

The Grover algorithm takes a black-box oracle implementing a function \( f(x) = 1 \) if \( x = x' \), \( f(x) = 0 \) if \( x \neq x' \) and finds \( x' \) within a randomly ordered sequence of \( N \) items using \( O(\sqrt{N}) \) operations and \( O(N \log(N)) \) gates, with the probability \( p \geq 2/3 \).

At the moment, only 2-bit sequences (for which one pass through Grover operator is enough) are considered.

--- REFERENCE ---
Coles, Eidenbenz et al. Quantum Algorithm Implementations for Beginners
https://arxiv.org/abs/1804.03719

--- EXAMPLE OUTPUT ---
Secret bit sequence: [1, 0]
Circuit:
(0, 0):

(1, 0):

(2, 0):

Sampled results:
Counter({"10": 10})
Most common bitstring: 10
Found a match: True

"""

import random
import cirq

def set_io_qubits(qubit_count):
    """Add the specified number of input and output qubits."""
```python
input_qubits = [cirq.GridQubit(i, 0) for i in range(qubit_count)]
output_qubit = cirq.GridQubit(qubit_count, 0)
return (input_qubits, output_qubit)

def make_oracle(input_qubits, output_qubit, x_bits):
    """Implement function f(x) = 1 if x=x', f(x) = 0 if x!= x'.""
    # Make oracle.
    # for (1, 1) it's just a Toffoli gate
    # otherwise negate the zero-bits.
    yield (cirq.X(q) for (q, bit) in zip(input_qubits, x_bits) if not bit)
    yield (cirq.TOFFOLI(input_qubits[0], input_qubits[1], output_qubit))
    yield (cirq.X(q) for (q, bit) in zip(input_qubits, x_bits) if not bit)

def make_grover_circuit(input_qubits, output_qubit, oracle):
    """Find the value recognized by the oracle in sqrt(N) attempts.""
    # For 2 input qubits, that means using Grover operator only once.
    c = cirq.Circuit()

    # Initialize qubits.
    c.append(
        [cirq.X(output_qubit),
         cirq.H(output_qubit),
         cirq.H.on_each(*input_qubits),
        ]
    )

    # Query oracle.
    c.append(oracle)

    # Construct Grover operator.
    c.append(cirq.H.on_each(*input_qubits))
    c.append(cirq.X.on_each(*input_qubits))
    c.append(cirq.H.on(input_qubits[1]))
    c.append(cirq.CNOT(input_qubits[0], input_qubits[1]))
    c.append(cirq.H.on(input_qubits[1]))
    c.append(cirq.X.on_each(*input_qubits))
    c.append(cirq.H.on_each(*input_qubits))

    # Measure the result.
    c.append(cirq.measure(*input_qubits, key='result'))

    return c

def bitstring(bits):
    return ''.join(str(int(b)) for b in bits)

def main():
    qubit_count = 2
    circuit_sample_count = 10

    # Set up input and output qubits.
    (input_qubits, output_qubit) = set_io_qubits(qubit_count)

    # Choose the x’ and make an oracle which can recognize it.
```

(continues on next page)
x_bits = [random.randint(0, 1) for _ in range(qubit_count)]
print('Secret bit sequence: {}'.format(x_bits))

# Make oracle (black box)
oracle = make_oracle(input_qubits, output_qubit, x_bits)

# Embed the oracle into a quantum circuit implementing Grover's algorithm.
circuit = make_grover_circuit(input_qubits, output_qubit, oracle)
print('Circuit: ')
print(circuit)

# Sample from the circuit a couple times.
simulator = cirq.Simulator()
result = simulator.run(circuit, repetitions=circuit_sample_count)

frequencies = result.histogram(key='result', fold_func=bitstring)
print('Sampled results: 

{} 
'.format(frequencies))

# Check if we actually found the secret value.
most_common_bitstring = frequencies.most_common(1)[0][0]
print('Most common bitstring: {}'.format(most_common_bitstring))
print('Found a match: {} '.format(most_common_bitstring == bitstring(x_bits)))

if __name__ == '__main__':
    main()
```python
simulator = cirq.Simulator()
result = simulator.run(circuit, repetitions=20)
print("Results:")
print(result)

if __name__ == '__main__':
    main()
```

### 2.10.6 Phase Estimator

""
Creates and simulates a phase estimator circuit.

--- EXAMPLE OUTPUT ---
Testing with 8 qubits.
target=0.0000, estimate=0.0000=0/256
target=0.1000, estimate=0.1016=26/256
target=0.2000, estimate=0.1992=51/256
target=0.3000, estimate=0.3008=77/256
target=0.4000, estimate=0.3984=102/256
target=0.5000, estimate=0.5000=128/256
target=0.6000, estimate=0.6016=154/256
target=0.7000, estimate=0.6992=179/256
target=0.8000, estimate=0.8008=205/256
target=0.9000, estimate=0.8984=230/256
RMS Error: 0.0011
""

```python
import numpy as np
import cirq
def run_estimate(unknown_gate, qnum, repetitions):
    ""
    Construct the following phase estimator circuit and execute simulations.
    ""
    ancilla = cirq.LineQubit(-1)
    qubits = cirq.LineQubit.range(qnum)
    # Circuit diagram
    # H - - - - - - - - - - - - - - - - - - M
    # H - - - - - - - - - - - - - - - - - M
    # H - - - - - - - - - - - - - - - - - M
    # H - - - - - - - - - - - - - - - - - M
    # H - - - - - - - - - - - - - - - - - M
    # H - - - - - - - - - - - - - - - - - M
    # H - - - - - - - - - - - - - - - - - M
    # H - - - - - - - - - - - - - - - - - M
    # U - - - - - - - - - - - - - - - - - - - - - -

    The measurement results M=[m1, m2,...] are translated to the estimated
    phase with the following formula:
    phi = m1*(1/2) + m2*(1/2)^2 + m3*(1/2)^3 + ...
    """
```
oracle_raised_to_power = [
    unknown_gate.on(ancilla).controlled_by(qubits[i])**(2**i)
    for i in range(qnum)
]
circuit = cirq.Circuit(cirq.H.on_each(*qubits), oracle_raised_to_power,
cirq.QFT(*qubits, without_reverse=True)**-1,
cirq.measure(*qubits, key='phase'))

return cirq.sample(circuit, repetitions=repetitions)

def experiment(qnum, repetitions=100):
    """Execute the phase estimator circuit with multiple settings and show results.
    """
    def example_gate(phi):
        """An example unitary 1-qubit gate U with an eigen vector |0> and an eigen value exp(2*Pi*i*phi)
        """
        gate = cirq.SingleQubitMatrixGate(
            matrix=np.array([[np.exp(2*np.pi*1.0j*phi), 0], [0, 1]])
        )
        return gate

    print(f'Testing with {qnum} qubits.
    errors = []
    for target in np.arange(0, 1, 0.1):
        result = run_estimate(example_gate(target), qnum, repetitions)
        mode = result.data['phase'].mode()[0]
        guess = mode / 2**qnum
        print(f'target={target:0.4f}, estimate={guess:0.4f}={mode}/2**qnum')
        errors.append((target - guess)**2)
    rms = np.sqrt(sum(errors) / len(errors))
    print(f'RMS Error: {rms:0.4f}

2.10.7 Place on Bristlecone

# pylint: disable=line-too-long
"""Create a circuit, optimize it, and map it onto a Bristlecone chip.
"""

===EXAMPLE_OUTPUT===
Length 10 line on Bristlecone:
(5, 0) (5, 1)
Initial circuit:

0: \(\times\) \(M('all')\)

1: \(\times\)

2: \(\times\)

3: \(\times\)

4: \(\times\)

5: \(\times\)

6: \(\times\)

7: \(\times\)

8: \(\times\)

9: \(\times\)

Xmon circuit:

(5, 0): \(Y^{-0.5}\) \@ \(Y^0.5\) \@ \(Y^{-0.5}\) \@ \(Y^0.5\) \@ \(Y^{-0.5}\) \@ \(Y^0.5\) \@ \(Y^{-0.5}\) \@ \(Y^0.5\) \@ \(Y^{-0.5}\) \@ \(Y^0.5\) \@ \(Y^{-0.5}\) \@ \(Y^0.5\) \\
= \(M('all')\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\)

(5, 1): \(X^{-0.5}\) \@ \(X^0.5\) \@ \(X^{-0.5}\) \@ \(X^0.5\) \@ \(X^{-0.5}\) \@ \(X^0.5\) \@ \(X^{-0.5}\) \@ \(X^0.5\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\)

(6, 2): \(X^{-0.5}\) \@ \(X^0.5\) \@ \(X^{-0.5}\) \@ \(X^0.5\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\) \\
\(\times\)
import cirq

def main():
    print("Length 10 line on Bristlecone:")
    line = cirq.google.line_on_device(cirq.google.Bristlecone, length=10)
    print(line)

    print("Initial circuit:")
    n = 10
    depth = 2
    circuit = cirq.Circuit(
        cirq.SWAP(cirq.LineQubit(j), cirq.LineQubit(j + 1))
        for i in range(depth)
        for j in range(i % 2, n - 1, 2)
    )
    circuit.append(cirq.measure(*cirq.LineQubit.range(n), key='all'))
    print(circuit)

    print()
    print("Xmon circuit:")
    translated = cirq.google.optimized_for_xmon(
        circuit=circuit,
        new_device=cirq.google.Bristlecone,
        qubit_map=lambda q: line[q.x])
    print(translated)

if __name__ == '__main__':
    main()

2.10.8 Quantum Fourier Transform

""
Creates and simulates a circuit for Quantum Fourier Transform(QFT) on a 4 qubit system.

In this example we demonstrate Fourier Transform on (1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0) vector. To do the same, we prepare the input state of the qubits as |0000>.

=== EXAMPLE OUTPUT ===

Circuit:
(0, 0): ─H┤0^0.5 ├──H──────────┤0^0.5 ├──H──────────┤0^0.5 ──H

```python
import numpy as np
import cirq

def main():
    """Demonstrates Quantum Fourier transform. """
    # Create circuit
    qft_circuit = generate_2x2_grid_qft_circuit()
    print('Circuit:
    print(qft_circuit)
    # Simulate and collect final_state
    simulator = cirq.Simulator()
    result = simulator.simulate(qft_circuit)
    print()
    print('FinalState
    print(np.around(result.final_state, 3))

def _cz_and_swap(q0, q1, rot):
    yield cirq.CZ(q0, q1)**rot
    yield cirq.SWAP(q0,q1)

# Create a quantum fourier transform circuit for 2x2 planar qubit architecture. # Circuit is adopted from https://arxiv.org/pdf/quant-ph/0402196.pdf
def generate_2x2_grid_qft_circuit():
    # Define a 2x2 square grid of qubits.
    a,b,c,d = [cirq.GridQubit(0, 0), cirq.GridQubit(0, 1),
               cirq.GridQubit(1, 1), cirq.GridQubit(1, 0)]

    circuit = cirq.Circuit(cirq.H(a),
                           _cz_and_swap(a, b, 0.5),
                           _cz_and_swap(b, c, 0.25),
                           _cz_and_swap(c, d, 0.125),
                           cirq.H(a),
                           _cz_and_swap(a, b, 0.5),
                           _cz_and_swap(b, c, 0.25),
                           cirq.H(a),
                           _cz_and_swap(a, b, 0.5),
                           cirq.H(a),
                           strategy=cirq.InsertStrategy.EARLIEST)

    return circuit

if __name__ == '__main__':
    main()
```

2.11 Recommended interface and behavior of a plot method

Here we recommend the input arguments, return value, and behavior of the `plot` method of a class.

2.11.1 Requirements

1. **Convenience to interactive users.** This is the highest priority. Compared to being called in a batch script as a library (for composing more complicated plots or other purposes), the `plot` method is mainly used in interactive sessions like ipython, jupyter, colab, PyCharm, and python interpreter.

2. **Plot is customizable.** The plot should be customizable by the user after `plot` returns. This is important because user may need to change the look for presentation, paper, or just the style they prefer. One plot style does not fit all.

3. **No unnecessary messages in interactive sessions.** It should not produce any warning/error messages in normal usages in an interactive sessions. See #1890 for an example of such message.

4. **No popups during tests.** It should not produce any pop-up windows during tests.

2.11.2 Recommendation

The `plot` method must produce a plot when there is no arguments in an interactive session. The recommended way to achieve that is illustrated in the example below.

```python
from typing import Any, List, Optional
import matplotlib.pyplot as plt
class Foo:
    ...
    def plot(self, ax: Optional[plt.Axes]=None, **plot_kwargs: Any) -> plt.Axes:
        show_plot = not ax
        if not ax:
            fig, ax = plt.subplots(1, 1)  # or your favorite figure setup
            # Call methods of the ax instance like ax.plot to plot on it.
        ...
        if show_plot:
            fig.show()
        return ax
```

This `plot` method works in 2 modes: *memory mode* and *interactive mode*, signalled by the presence of the `ax` argument. When present, the method is instructed to plot on the provided `ax` instance in memory. No plot is shown on the screen. When absent, the code is in *interactive mode*, and it creates a figure and shows it.

The returned `ax` instance can be used to further customize the plot if the user wants to. Note that if we were to call `plt.show` instead of `fig.show`, the customizations on the returned `ax` does not show up on subsequent call to `plt.show`.

To satisfy requirement number 4, unit test codes should create an `ax` object and pass it into the `plot` method like the following example.

```python
def test_foo_plot():
    # make a Foo instance foo
    figure, ax = plt.subplots(1, 1)
    foo.plot(ax)
    # assert on the content of ax here if necessary.
```
This does not produce a pop-up window because `fig.show` is not called.

### 2.11.3 Classes that produce multi-axes plot

Some classes contain complicated data and plotting on a single `ax` is not sufficient. The `plot` method of such a class should take an optional `axes` argument that is a list of `plt.Axes` instances.

```python
class Foo:
    ...
    def plot(self, axes: Optional[List[plt.Axes]]=None, **plot_kwargs: Any) -> List[plt.Axes]:
        show_plot = not axes
        if not axes:
            fig, axes = plt.subplots(1, 2)  # or your favorite figure setup
        elif len(axes) != 2:
            raise ValueError('your error message')
        # Call methods of the axes[i] objects to plot on it.
        ...
        if show_plot:
            fig.show()
        return axes
```

The reason we don’t recommend passing a `plt.Figure` argument is that, the `plot` method has no information on which `plt.Axes` objects to plot on if there are more `plt.Axes` in the figure than what the method needs. The caller is responsible for passing in correct number of `Axes` instances.

The `plot` method can be tested similarly.

### 2.11.4 PyCharm issue

As of this writing in October 2019, running a script calling a `plot` method in PyCharm does not pop up a window with the figure. A call to `plt.show()` is needed to show it. We believe this is a PyCharm-specific issue because the same code works in Python interpreter.

### 2.11.5 References

- Issue #1890 “Plotting code should not call `show`”
- PR #2097
- PR #2286
CHAPTER THREE

API REFERENCE

3.1 API Reference

3.1.1 Devices and Qubits

Classes for identifying the qubits and hardware you want to operate on.

<table>
<thead>
<tr>
<th>Device</th>
<th>Hardware constraints for validating circuits and schedules.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GridQubit(row, col)</td>
<td>A qubit on a 2d square lattice.</td>
</tr>
<tr>
<td>LineQubit(x)</td>
<td>A qubit on a 1d lattice with nearest-neighbor connectivity.</td>
</tr>
<tr>
<td>NamedQubit(name)</td>
<td>A qubit identified by name.</td>
</tr>
<tr>
<td>Qid</td>
<td>Identifies a quantum object such as a qubit, qudit, resonator, etc.</td>
</tr>
</tbody>
</table>

```python
cirq.Device

class cirq.Device
    Hardware constraints for validating circuits and schedules.
    
    __init__(self, ...) Initialize self. See help(type(self)) for accurate signature.

Methods

    can_add_operation_into_moment(operation, moment) Determines if it’s possible to add an operation into a moment.
    decompose_operation(operation) Returns a device-valid decomposition for the given operation.
    duration_of(operation)                                      |
    validate_circuit(circuit) Raises an exception if a circuit is not valid.
    validate_operation(operation) Raises an exception if an operation is not valid.
    validate_schedule(schedule) Raises an exception if a schedule is not valid.
    validate_scheduled_operation(schedule, ...) Raises an exception if the scheduled operation is not valid.
```
cirq.Device.can_add_operation_into_moment

Device.can_add_operation_into_moment(operation: cirq.Operation, moment: cirq.Moment) → bool

Determines if it’s possible to add an operation into a moment.

For example, on the XmonDevice two CZs shouldn’t be placed in the same moment if they are on adjacent qubits.

Parameters

- **operation** – The operation being added.
- **moment** – The moment being transformed.

Returns Whether or not the moment will validate after adding the operation.

cirq.Device.decompose_operation

Device.decompose_operation(operation: cirq.Operation) → cirq.OP_TREE

Returns a device-valid decomposition for the given operation.

This method is used when adding operations into circuits with a device specified, to avoid spurious failures due to e.g. using a Hadamard gate that must be decomposed into native gates.

cirq.Device.duration_of

abstract Device.duration_of(operation: cirq.Operation) → cirq.value.duration.Duration

cirq.Device.validate_circuit

Device.validate_circuit(circuit: cirq.Circuit) → None

Raises an exception if a circuit is not valid.

Parameters **circuit** – The circuit to validate.

Raises **ValueError** – The circuit isn’t valid for this device.

cirq.Device.validate_moment

Device.validate_moment(moment: cirq.Moment) → None

Raises an exception if a moment is not valid.

Parameters **moment** – The moment to validate.

Raises **ValueError** – The moment isn’t valid for this device.
cirq.Device.validate_operation

abstract Device.validate_operation (operation: cirq.Operation) → None

Raises an exception if an operation is not valid.

Parameters  
operation – The operation to validate.

Raises  
ValueError – The operation isn’t valid for this device.

cirq.Device.validate_schedule

abstract Device.validate_schedule (schedule: cirq.Schedule) → None

Raises an exception if a schedule is not valid.

Parameters  
schedule – The schedule to validate.

Raises  
ValueError – The schedule isn’t valid for this device.

cirq.Device.validate_scheduled_operation

abstract Device.validate_scheduled_operation (schedule: cirq.Schedule, scheduled_operation: cirq.ScheduledOperation) → None

Raises an exception if the scheduled operation is not valid.

Parameters

- schedule – The schedule to validate against.

- scheduled_operation – The scheduled operation to validate.

Raises  
ValueError – If the scheduled operation is not valid for the schedule.

cirq.GridQubit

class cirq.GridQubit (row: int, col: int)
A qubit on a 2d square lattice.

GridQubits use row-major ordering:

<table>
<thead>
<tr>
<th>GridQubit(0, 0)</th>
<th>GridQubit(0, 1)</th>
<th>GridQubit(1, 0)</th>
<th>GridQubit(1, 1)</th>
</tr>
</thead>
</table>

New GridQubits can be constructed by adding or subtracting tuples

```python
>>> cirq.GridQubit(2, 3) + (3, 1)
cirq.GridQubit(5, 4)
>>> cirq.GridQubit(2, 3) - (1, 2)
cirq.GridQubit(1, 1)
```

__init__ (row: int, col: int)

Initialize self. See help(type(self)) for accurate signature.

Methods
from_diagram(diagram)

Parse ASCII art device layout into info about qubits and

is_adjacent(other)

Determines if two qubits are adjacent qubits.

neighbors([qids])

Returns qubits that are potential neighbors to this GridQubit

rect(rows, cols[, top, left])

Returns a rectangle of GridQubits.

square(diameter[, top, left])

Returns a square of GridQubits.

validate_dimension(dimension)

Raises an exception if dimension is not positive.

with_dimension(dimension)

Returns a new qid with a different dimension.

cirq.GridQubit.from_diagram

static GridQubit.from_diagram(diagram: str) → List[cirq.devices.grid_qubit.GridQubit]

Parse ASCII art device layout into info about qubits and
connectivity. As an example, the below diagram will create a list of
GridQubits in a pyramid structure.

—A—
–AAA–
-AAAAA-
AAAAAAA

You can use any character other than a hyphen to mark a qubit. As an
example, the qubits for the Bristlecone device could be represented by
the below diagram. This produces a diamond-shaped grid of qubits, and
qubits with the same letter correspond to the same readout line.

.....AB.....
....ABCD....
....ABCDEF...
..ABCDEFGH..
.ABCDEFGHIJ
.ABCDEFGHIJKL
.CDEFGHIJKL.
..EFGHIJKL..
...GHIJKL...
....IJKL....
.....KL.....

Args:

diagram: String representing the qubit layout. Each line represents
a row. Alphanumeric characters are assigned as qubits.
Dots ('.'), dashes ('-'), and spaces (' ') are treated as
empty locations in the grid. If diagram has characters other
than alphanumerics, spacers, and newlines ('')

), an error will
be thrown. The top-left corner of the diagram will be have
coordinate (0,0).
cirq.GridQubit.is_adjacent

GridQubit.is_adjacent(other: cirq.ops.raw_types.Qid) → bool
Determines if two qubits are adjacent qubits.

cirq.GridQubit.neighbors

GridQubit.neighbors(qids: Optional[Iterable[cirq.ops.raw_types.Qid]] = None) → Set[cirq.devices.grid_qubit.GridQubit]
Returns qubits that are potential neighbors to this GridQubit

Parameters
qids – optional Iterable of qubits to constrain neighbors to.

cirq.GridQubit.rect

static GridQubit.rect(rows: int, cols: int, top: int = 0, left: int = 0) → List[cirq.devices.grid_qubit.GridQubit]
Returns a rectangle of GridQubits.

Parameters
• rows – Number of rows in the rectangle
• cols – Number of columns in the rectangle
• top – Row number of the topmost row
• left – Column number of the leftmost row

Returns A list of GridQubits filling in a rectangular grid

cirq.GridQubit.square

static GridQubit.square(diameter: int, top: int = 0, left: int = 0) → List[cirq.devices.grid_qubit.GridQubit]
Returns a square of GridQubits.

Parameters
• diameter – Length of a side of the square
• top – Row number of the topmost row
• left – Column number of the leftmost row

Returns A list of GridQubits filling in a square grid
cirq.GridQubit.validate_dimension

```python
static GridQubit.validate_dimension(dimension: int) -> None
```

Raises an exception if `dimension` is not positive.

Raises `ValueError` – `dimension` is not positive.

cirq.GridQubit.with_dimension

```python
GridQubit.with_dimension(dimension: int) -> cirq.ops.raw_types.Qid
```

Returns a new qid with a different dimension.

Child classes can override. Wraps the qubit object by default.

**Parameters**

`dimension` – The new dimension or number of levels.

Attributes

```python

dimension
```

Returns the dimension or the number of quantum levels this qid has.

---

cirq.GridQubit.dimension

```python
property GridQubit.dimension
```

Returns the dimension or the number of quantum levels this qid has.

E.g. 2 for a qubit, 3 for a qutrit, etc.

cirq.LineQubit

```python
class cirq.LineQubit(x: int)
```

A qubit on a 1d lattice with nearest-neighbor connectivity.

LineQubits have a single attribute, and integer coordinate ‘x’, which identifies the qubits location on the line. LineQubits are ordered by this integer.

One can construct new LineQubits by adding or subtracting integers:

```python
>>> cirq.LineQubit(1) + 3
cirq.LineQubit(4)

>>> cirq.LineQubit(2) - 1
cirq.LineQubit(1)
```

```python
__init__(x: int) -> None
```

Initializes a line qubit at the given x coordinate.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_adjacent(other)</code></td>
<td>Determines if two qubits are adjacent line qubits.</td>
</tr>
<tr>
<td><code>neighbors([qids])</code></td>
<td>Returns qubits that are potential neighbors to this LineQubit.</td>
</tr>
<tr>
<td><code>range(*range_args)</code></td>
<td>Returns a range of line qubits.</td>
</tr>
<tr>
<td><code>validate_dimension(dimension)</code></td>
<td>Raises an exception if <code>dimension</code> is not positive.</td>
</tr>
<tr>
<td><code>with_dimension(dimension)</code></td>
<td>Returns a new qid with a different dimension.</td>
</tr>
</tbody>
</table>

### `cirq.LineQubit.is_adjacent`

```python
LineQubit.is_adjacent(other: cirq.ops.raw_types.Qid) → bool
```

Determines if two qubits are adjacent line qubits.

### `cirq.LineQubit.neighbors`

```python
LineQubit.neighbors(qids: Optional[Iterable[cirq.ops.raw_types.Qid]] = None) → Set[cirq.devices.line_qubit._BaseLineQid]
```

Returns qubits that are potential neighbors to this LineQubit.

**Parameters**

- `qubits` – optional Iterable of qubits to constrain neighbors to.

### `cirq.LineQubit.range`

```python
static LineQubit.range(*range_args) → List[cirq.devices.line_qubit.LineQubit]
```

Returns a range of line qubits.

**Parameters**

- `*range_args` – Same arguments as python’s built-in range method.

**Returns**

A list of line qubits.

### `cirq.LineQubit.validate_dimension`

```python
static LineQubit.validate_dimension(dimension: int) → None
```

Raises an exception if `dimension` is not positive.

**Raises**

`ValueError` – `dimension` is not positive.

### `cirq.LineQubit.with_dimension`

```python
LineQubit.with_dimension(dimension: int) → cirq.devices.line_qubit.LineQid
```

Returns a new qid with a different dimension.

**Parameters**

- `dimension` – The new dimension or number of levels.

### Attributes

- `is_adjacent`
- `neighbors`
- `range`
- `validate_dimension`
- `with_dimension`
dimension

Returns the dimension or the number of quantum levels this qid has.

cirq.LineQubit.dimension

property LineQubit.dimension

Returns the dimension or the number of quantum levels this qid has.
E.g. 2 for a qubit, 3 for a qutrit, etc.

cirq.NamedQubit

class cirq.NamedQubit(name: str)

A qubit identified by name.

By default, NamedQubit has a lexicographic order. However, numbers within the name are handled correctly. So, for example, if you print a circuit containing cirq.NamedQubit('qubit22') and cirq.NamedQubit('qubit3'), the wire for ‘qubit3’ will correctly come before ‘qubit22’.

__init__ (name: str) → None

Initialize self. See help(type(self)) for accurate signature.

Methods

range(*args, prefix) Returns a range of NamedQubits.
validate_dimension(dimension) Raises an exception if dimension is not positive.
with_dimension(dimension) Returns a new qid with a different dimension.

cirq.NamedQubit.range

static NamedQubit.range (*args, prefix: str) Returns a range of NamedQubits.

The range returned starts with the prefix, and followed by a qubit for each number in the range, e.g.:

NamedQubit.range(3, prefix="a") -> ["a1", "a2", "a3"
NamedQubit.range(2, 4, prefix="a") -> ["a2", "a3"

Parameters
• *args – Args to be passed to Python’s standard range function.
• prefix – A prefix for constructed NamedQubits.

Returns A list of NamedQubits.

cirq.NamedQubit.validate_dimension

static NamedQubit.validate_dimension(dimension: int) → None

Raises an exception if dimension is not positive.

Raises ValueError – dimension is not positive.

cirq.NamedQubit.with_dimension

NamedQubit.with_dimension(dimension: int) → cirq.ops.raw_types.Qid

Returns a new qid with a different dimension.

Child classes can override. Wraps the qubit object by default.

Parameters dimension – The new dimension or number of levels.

Attributes

dimension

Returns the dimension or the number of quantum levels this qid has.

name

cirq.NamedQubit.dimension

property NamedQubit.dimension

Returns the dimension or the number of quantum levels this qid has.
E.g. 2 for a qubit, 3 for a qutrit, etc.

cirq.NamedQubit.name

property NamedQubit.name

cirq.Qid

class cirq.Qid

Identifies a quantum object such as a qubit, qudit, resonator, etc.

Child classes represent specific types of objects, such as a qubit at a particular location on a chip or a qubit with a particular name.
The main criteria that a custom qid must satisfy is *comparability*. Child classes meet this criteria by implementing the `_comparison_key` method. For example, `cirq.LineQubit`'s `_comparison_key` method returns `self.x`. This ensures that line qubits with the same `x` are equal, and that line qubits will be sorted ascending by `x`. `Qid` implements all equality, comparison, and hashing methods via `_comparison_key`.

```
__init__()
    Initialize self. See help(type(self)) for accurate signature.
```

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>validate_dimension(dimensions)</td>
<td>Raises an exception if <code>dimensions</code> is not positive.</td>
</tr>
<tr>
<td>with_dimension(dimensions)</td>
<td>Returns a new qid with a different dimension.</td>
</tr>
</tbody>
</table>

```python
cirq.Qid.validate_dimension

static Qid.validate_dimension(dimensions: int) → None
    Raises an exception if `dimensions` is not positive.
    Raises `ValueError` – `dimensions` is not positive.
```

cirq.Qid.with_dimension

```python
cirq.Qid.with_dimension(dimensions: int) → cirq.ops.raw_types.Qid
    Returns a new qid with a different dimension.
    Child classes can override. Wraps the qubit object by default.
    Parameters dimension – The new dimension or number of levels.
```

**Attributes**

```python
dimension
    Returns the dimension or the number of quantum levels this qid has.
```

cirq.Qid.dimension

```python
abstract property Qid.dimension

    Returns the dimension or the number of quantum levels this qid has.
    E.g. 2 for a qubit, 3 for a qutrit, etc.
```
### 3.1.2 Single Qubit Unitary Gates

Unitary operations you can apply to a single qubit.

<table>
<thead>
<tr>
<th>Gate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>H</code></td>
<td>A Gate that performs a rotation around the X+Z axis of the Bloch sphere.</td>
</tr>
<tr>
<td><code>HPowGate(*[, exponent, global_shift])</code></td>
<td>A Gate that performs a rotation around the X+Z axis of the Bloch sphere.</td>
</tr>
<tr>
<td><code>measure(*target[, key, invert_mask])</code></td>
<td>Returns a single MeasurementGate applied to all the given qubits.</td>
</tr>
<tr>
<td><code>measure_each(*qubits[, key_func])</code></td>
<td>Returns a list of operations individually measuring the given qubits.</td>
</tr>
<tr>
<td><code>MeasurementGate([num_qubits, key, . . .])</code></td>
<td>A gate that measures qubits in the computational basis.</td>
</tr>
<tr>
<td><code>PhasedXPowGate(*, phase_exponent[,...])</code></td>
<td>A gate equivalent to the circuit $Z^p X^t Z^p$.</td>
</tr>
<tr>
<td><code>Rx(rads)</code></td>
<td>Returns a gate with the matrix $e^{-i X rads / 2}$.</td>
</tr>
<tr>
<td><code>Ry(rads)</code></td>
<td>Returns a gate with the matrix $e^{-i Y rads / 2}$.</td>
</tr>
<tr>
<td><code>Rz(rads)</code></td>
<td>Returns a gate with the matrix $e^{-i Z rads / 2}$.</td>
</tr>
<tr>
<td><code>S</code></td>
<td>A gate that rotates around the Z axis of the Bloch sphere.</td>
</tr>
<tr>
<td><code>SingleQubitMatrixGate(matrix)</code></td>
<td>A 1-qubit or qudit gate defined by its matrix.</td>
</tr>
<tr>
<td><code>T</code></td>
<td>A gate that rotates around the Z axis of the Bloch sphere.</td>
</tr>
<tr>
<td><code>TwoQubitMatrixGate(matrix)</code></td>
<td>A 2-qubit gate defined only by its matrix.</td>
</tr>
<tr>
<td><code>X</code></td>
<td></td>
</tr>
<tr>
<td><code>XPowGate(*[, exponent, global_shift])</code></td>
<td>A gate that rotates around the X axis of the Bloch sphere.</td>
</tr>
<tr>
<td><code>Y</code></td>
<td></td>
</tr>
<tr>
<td><code>YPowGate(*[, exponent, global_shift])</code></td>
<td>A gate that rotates around the Y axis of the Bloch sphere.</td>
</tr>
<tr>
<td><code>Z</code></td>
<td></td>
</tr>
<tr>
<td><code>ZPowGate(*[, exponent, global_shift])</code></td>
<td>A gate that rotates around the Z axis of the Bloch sphere.</td>
</tr>
</tbody>
</table>

**cirq.H**

`cirq.H = cirq.H`  
A Gate that performs a rotation around the X+Z axis of the Bloch sphere.

The unitary matrix of `HPowGate(exponent=t)` is:

\[
\begin{bmatrix}
g \cdot (c-i \cdot s / \sqrt{2}), & -i \cdot g \cdot s / \sqrt{2} \\
-i \cdot g \cdot s / \sqrt{2}, & g \cdot (c+i \cdot s / \sqrt{2})
\end{bmatrix}
\]

where

- $c = \cos(\cdot t / 2)$
- $s = \sin(\cdot t / 2)$
- $g = \exp(i \cdot t / 2)$

Note in particular that for $t=1$, this gives the Hadamard matrix.

`cirq.H`, the Hadamard gate, is an instance of this gate at $exponent=1$.  

---

3.1. API Reference
cirq.HPowGate

class cirq.HPowGate(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0)

A Gate that performs a rotation around the X+Z axis of the Bloch sphere.

The unitary matrix of $\text{HPowGate}(exponent=t)$ is:

\[
\begin{bmatrix}
g \cdot (c - i \cdot s / \sqrt{2}), & -i \cdot g \cdot s / \sqrt{2} \\
-i \cdot g \cdot s / \sqrt{2}, & g \cdot (c + i \cdot s / \sqrt{2})
\end{bmatrix}
\]

where

\[
c = \cos(t/2) \\
s = \sin(t/2) \\
g = \exp(i \cdot t).
\]

Note in particular that for $t=1$, this gives the Hadamard matrix.

cirq.H, the Hadamard gate, is an instance of this gate at $exponent=1$.

__init__(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0) → None

Initializes the parameters used to compute the gate’s matrix.

The eigenvalue of each eigenspace of a gate is computed by

1. Starting with an angle in half turns as returned by the gate’s
   \_eigen_components method:

2. Shifting the angle by \text{global\_shift}:

\[+ s\]

3. Scaling the angle by \text{exponent}:

\[(+ s) \cdot e\]

4. Converting from half turns to a complex number on the unit circle:

\[\exp(i \cdot \pi \cdot (+ s) \cdot e)\]

Parameters

- **exponent** – The $t$ in $\text{gate}^t$. Determines how much the eigenvalues of the gate are scaled by. For example, eigenvectors phased by -1 when $\text{gate}^*I$ is applied will gain a relative phase of $e^{i \cdot \pi \cdot \text{exponent}}$ when $\text{gate}^\text{exponent}$ is applied (relative to eigenvectors unaffected by $\text{gate}^*I$).

- **global_shift** – Offsets the eigenvalues of the gate at $exponent=1$. In effect, this controls a global phase factor on the gate’s unitary matrix. The factor is:

\[\exp(i \cdot \pi \cdot \text{global\_shift} \cdot \text{exponent})\]

For example, $\text{cirq.X}^*t$ uses a \text{global\_shift} of 0 but $\text{cirq.Rx}(t)$ uses a \text{global\_shift} of -0.5, which is why $\text{cirq.unitary(cirq.Rx(pi))}$ equals -iX instead of X.
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlled(num_controls, control_values, ...)</code></td>
<td>Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.</td>
</tr>
<tr>
<td><code>num_qubits()</code></td>
<td>The number of qubits this gate acts on.</td>
</tr>
<tr>
<td><code>on(*qubits)</code></td>
<td>Returns an application of this gate to the given qubits.</td>
</tr>
<tr>
<td><code>on_each(*targets)</code></td>
<td>Returns a list of operations applying the gate to all targets.</td>
</tr>
<tr>
<td><code>validate_args(qubits)</code></td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
<tr>
<td><code>wrap_in_linear_combination([coefficient])</code></td>
<td></td>
</tr>
</tbody>
</table>

### cirq.HPowGate.controlled

```
HPowGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) -> cirq.ops.raw_types.Gate
```

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- **num_controls**: Total number of control qubits.
- **control_values**: For which control qubit values to apply the sub gate. A sequence of length `num_controls` where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- **control_qid_shape**: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to `(2,) * num_controls`. Specify this argument when using qudits.

### cirq.HPowGate.num_qubits

```
HPowGate.num_qubits() -> int
```

The number of qubits this gate acts on.

### cirq.HPowGate.on

```
HPowGate.on(*qubits: cirq.ops.raw_types.Qid) -> cirq.ops.raw_types.Operation
```

Returns an application of this gate to the given qubits.

- **Parameters** `*qubits` – The collection of qubits to potentially apply the gate to.
cirq.HPowGate.on_each

HPowGate.on_each(*targets: Union[cirq.ops.raw_types.Qid, Iterable[Any]]) → List[cirq.ops.raw_types.Operation]

Returns a list of operations applying the gate to all targets.

Parameters  *targets – The qubits to apply this gate to.

Returns  Operations applying this gate to the target qubits.

Raises ValueError if targets are not instances of Qid or List[Qid] –

cirq.HPowGate.validate_args

HPowGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None

Checks if this gate can be applied to the given qubits.

By default checks that:
• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call super().validate_args(qubits) then do custom checks.

Parameters  qubits – The sequence of qubits to potentially apply the gate to.

Throws:  ValueError: The gate can’t be applied to the qubits.

cirq.HPowGate.wrap_in_linear_combination

HPowGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates

Attributes

exponent

global_shift

property HPowGate.exponent

cirq.HPowGate.global_shift

property HPowGate.global_shift
cirq.measure

```python
```

Returns a single MeasurementGate applied to all the given qubits.

The qubits are measured in the computational basis.

Parameters

- **target** – The qubits that the measurement gate should measure.
- **key** – The string key of the measurement. If this is None, it defaults to a comma-separated list of the target qubits’ str values.
- **invert_mask** – A list of Truthy or Falsey values indicating whether the corresponding qubits should be flipped. None indicates no inverting should be done.

Returns An operation targeting the given qubits with a measurement.

Raises `ValueError` if the qubits are not instances of `Qid`.

---

cirq.measure_each

```python
```

Returns a list of operations individually measuring the given qubits.

The qubits are measured in the computational basis.

Parameters

- **qubits** – The qubits to measure.
- **key_func** – Determines the key of the measurements of each qubit. Takes the qubit and returns the key for that qubit. Defaults to str.

Returns A list of operations individually measuring the given qubits.

---

cirq.MeasurementGate

```python
class cirq.MeasurementGate(num_qubits: Optional[int] = None, key: str = '', invert_mask: Tuple[bool, ...] = (), qid_shape: Tuple[int, ...] = None)
```

A gate that measures qubits in the computational basis.

The measurement gate contains a key that is used to identify results of measurements.

```python
__init__(num_qubits: Optional[int] = None, key: str = '', invert_mask: Tuple[bool, ...] = (), qid_shape: Tuple[int, ...] = None) → None
```

Parameters

- **num_qubits** – The number of qubits to act upon.
- **key** – The string key of the measurement.
• **invert_mask** – A list of values indicating whether the corresponding qubits should be flipped. The list's length must not be longer than the number of qubits, but it is permitted to be shorter. Qubits with indices past the end of the mask are not flipped.

• **qid_shape** – Specifies the dimension of each qid the measurement applies to. The default is 2 for every qubit.

**Raises** `ValueError` – If the length of `invert_mask` is greater than `num_qubits`. or if the length of `qid_shape` doesn’t equal `num_qubits`.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlled([num_controls, control_values, ...])</code></td>
<td>Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.</td>
</tr>
<tr>
<td><code>full_invert_mask()</code></td>
<td>Returns the invert mask for all qubits.</td>
</tr>
<tr>
<td><code>num_qubits()</code></td>
<td>The number of qubits this gate acts on.</td>
</tr>
<tr>
<td><code>on(*qubits)</code></td>
<td>Returns an application of this gate to the given qubits.</td>
</tr>
<tr>
<td><code>validate_args(qubits)</code></td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
<tr>
<td><code>with_bits_flipped(*bit_positions)</code></td>
<td>Toggles whether or not the measurement inverts various outputs.</td>
</tr>
<tr>
<td><code>wrap_in_linear_combination([coefficient])</code></td>
<td></td>
</tr>
</tbody>
</table>

```python
measurementGate.controlled
```

**MeasurementGate.controlled**(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- **num_controls**: Total number of control qubits.
- **control_values**: For which control qubit values to apply the sub gate. A sequence of length `num_controls` where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- **control_qid_shape**: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to `(2,) * num_controls`. Specify this argument when using qudits.
**cirq.MeasurementGate.full_invert_mask**

```python
MeasurementGate.full_invert_mask()
```

Returns the invert mask for all qubits.

If the user supplies a partial invert_mask, this returns that mask padded by False.

Similarly if no invert_mask is supplied this returns a tuple of size equal to the number of qubits with all entries False.

**cirq.MeasurementGate.num_qubits**

```python
MeasurementGate.num_qubits() -> int
```

The number of qubits this gate acts on.

**cirq.MeasurementGate.on**

```python
MeasurementGate.on(*qubits: cirq.ops.raw_types.Qid) -> cirq.ops.raw_types.Operation
```

Returns an application of this gate to the given qubits.

**Parameters**

- **qubits** – The collection of qubits to potentially apply the gate to.

**cirq.MeasurementGate.validate_args**

```python
MeasurementGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) -> None
```

Checks if this gate can be applied to the given qubits.

By default checks that:
- inputs are of type Qid
- len(qubits) == num_qubits()
- qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call super().validate_args(qubits) then do custom checks.

**Parameters**

- **qubits** – The sequence of qubits to potentially apply the gate to.

**Throws**:

ValueError: The gate can’t be applied to the qubits.

**cirq.MeasurementGate.with_bits_flipped**

```python
MeasurementGate.with_bits_flipped(*bit_positions: int) -> cirq.ops.measurement_gate.MeasurementGate
```

Toggles whether or not the measurement inverts various outputs.
cirq.MeasurementGate.wrap_in_linear_combination

MeasurementGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) \rightarrow\text{linear_combinations.LinearCombinationOfGates}

cirq.PhasedXPowGate

class cirq.PhasedXPowGate(*, phase_exponent: Union[float, sympy.core.symbol.Symbol], exponent: Union[float, sympy.core.symbol.Symbol] = 1.0, global_shift: float = 0.0)

A gate equivalent to the circuit $Z^{-p}X^{t}Z^{p}$.

__init__(*, phase_exponent: Union[float, sympy.core.symbol.Symbol], exponent: Union[float, sympy.core.symbol.Symbol] = 1.0, global_shift: float = 0.0) \rightarrow None

Parameters

• phase_exponent – The exponent on the $Z$ gates conjugating the $X$ gate.
• exponent – The exponent on the $X$ gate conjugated by $Z$s.
• global_shift – How much to shift the operation’s eigenvalues at exponent=1.

Methods

controlled([num_controls, control_values, ...])  
Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

num_qubits()  
The number of qubits this gate acts on.

on(*qubits)  
Returns an application of this gate to the given qubits.

on_each(*targets)  
Returns a list of operations applying the gate to all targets.

validate_args(qubits)  
Checks if this gate can be applied to the given qubits.

wrap_in_linear_combination(coefficients)

PhasedXPowGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) \rightarrow cirq.ops.raw_types.Gate

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

num_controls: Total number of control qubits.
control_values: For which control qubit values to apply the sub gate. A sequence of length num_controls where each entry is an integer (or set of integers) corresponding to the
qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.

control_qid_shape: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to (2,) * num_controls. Specify this argument when using qudits.

cirq.PhasedXPowGate.num_qubits

PhasedXPowGate.num_qubits() \rightarrow int
  The number of qubits this gate acts on.

cirq.PhasedXPowGate.on

PhasedXPowGate.on(*qubits: cirq.ops.raw_types.Qid) \rightarrow cirq.ops.raw_types.Operation
  Returns an application of this gate to the given qubits.

  Parameters *qubits – The collection of qubits to potentially apply the gate to.

cirq.PhasedXPowGate.on_each

PhasedXPowGate.on_each(*targets: Union[cirq.ops.raw_types.Qid, Iterable[Any]]) \rightarrow List[cirq.ops.raw_types.Operation]
  Returns a list of operations applying the gate to all targets.

  Parameters *targets – The qubits to apply this gate to.

  Returns Operations applying this gate to the target qubits.

  Raises ValueError if targets are not instances of Qid or List[Qid] –

cirq.PhasedXPowGate.validate_args

PhasedXPowGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) \rightarrow None
  Checks if this gate can be applied to the given qubits.

  By default checks that:
  • inputs are of type Qid
  • len(qubits) == num_qubits()
  • qubit_i.dimension == qid_shape[i] for all qubits

  Child classes can override. The child implementation should call super().validate_args(qubits) then do custom checks.

  Parameters qubits – The sequence of qubits to potentially apply the gate to.

  Throws: ValueError: The gate can’t be applied to the qubits.
cirq.PhasedXPowGate.wrap_in_linear_combination

PhasedXPowGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates

Attributes

exponent

The exponent on the central X gate conjugated by the Z gates.

global_shift

phase_exponent

The exponent on the Z gates conjugating the X gate.

cirq.PhasedXPowGate.exponent

property PhasedXPowGate.exponent

The exponent on the central X gate conjugated by the Z gates.

cirq.PhasedXPowGate.global_shift

property PhasedXPowGate.global_shift

cirq.PhasedXPowGate.phase_exponent

property PhasedXPowGate.phase_exponent

The exponent on the Z gates conjugating the X gate.

cirq.Rx

cirq.Rx(rads: Union[float, sympy.core.basic.Basic]) → cirq.ops.common_gates.XPowGate

Returns a gate with the matrix $e^{-i X \text{rads} / 2}$.

cirq.Ry

cirq.Ry(rads: Union[float, sympy.core.basic.Basic]) → cirq.ops.common_gates.YPowGate

Returns a gate with the matrix $e^{-i Y \text{rads} / 2}$.

cirq.Rz

cirq.Rz(rads: Union[float, sympy.core.basic.Basic]) → cirq.ops.common_gates.ZPowGate

Returns a gate with the matrix $e^{-i Z \text{rads} / 2}$.

cirq.S

cirq.S = cirq.S

A gate that rotates around the Z axis of the Bloch sphere.

The unitary matrix of ZPowGate(exponent=t) is:
where:

\[ g = \exp(i \cdot t). \]

Note in particular that this gate has a global phase factor of 
\( e^{i \cdot t/2} \) vs the traditionally defined rotation matrices
about the Pauli Z axis. See cirq.Rz for rotations without the global
phase. The global phase factor can be adjusted by using the `global_shift` parameter when initializing.

cirq.Z, the Pauli Z gate, is an instance of this gate at exponent=1.

cirq.SingleQubitMatrixGate

class cirq.SingleQubitMatrixGate(matrix: numpy.ndarray)

A 1-qubit or qudit gate defined by its matrix.

More general than specialized classes like ZPowGate, but more expensive
and more float-error sensitive to work with (due to using
eigendecompositions).

__init__(matrix: numpy.ndarray) -> None

Initializes the 2-qubit matrix gate.

Parameters

- **matrix** – The matrix that defines the gate.

Methods

- **controlled([num_controls, control_values, ...])**
  Returns a controlled version of this gate. If no arguments are

- **num_qubits()**
  The number of qubits this gate acts on.

- **on(*qubits)**
  Returns an application of this gate to the given qubits.

- **on_each(*targets)**
  Returns a list of operations applying the gate to all targets.

- **validate_args(qubits)**
  Checks if this gate can be applied to the given qubits.

- **wrap_in_linear_combination([coefficient])**
cirq.SingleQubitMatrixGate.controlled

SingleQubitMatrixGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) \rightarrow cirq.ops.raw_types.Gate

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

num_controls: Total number of control qubits.
control_values: For which control qubit values to apply the sub gate. A sequence of length num_controls where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
control_qid_shape: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to (2,) * num_controls. Specify this argument when using qudits.

cirq.SingleQubitMatrixGate.num_qubits

SingleQubitMatrixGate.num_qubits() \rightarrow int

The number of qubits this gate acts on.

cirq.SingleQubitMatrixGate.on

SingleQubitMatrixGate.on(*qubits: cirq.ops.raw_types.Qid) \rightarrow cirq.ops.raw_types.Operation

Returns an application of this gate to the given qubits.

Parameters *qubits – The collection of qubits to potentially apply the gate to.

cirq.SingleQubitMatrixGate.on_each

SingleQubitMatrixGate.on_each(*targets: Union[cirq.ops.raw_types.Qid, Iterable[Any]]) \rightarrow List[cirq.ops.raw_types.Operation]

Returns a list of operations applying the gate to all targets.

Parameters *targets – The qubits to apply this gate to.

Returns Operations applying this gate to the target qubits.

Raises ValueError if targets are not instances of Qid or List[Qid] –
cirq.SingleQubitMatrixGate.validate_args

SingleQubitMatrixGate.validate_args(qubits)

Checks if this gate can be applied to the given qubits.

By default checks that:

• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call super().validate_args(qubits) then do custom checks.

Parameters qubits – The sequence of qubits to potentially apply the gate to.

Throws: ValueError: The gate can’t be applied to the qubits.

cirq.SingleQubitMatrixGate.wrap_in_linear_combination

SingleQubitMatrixGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) \rightarrow LinearCombinationOfGates

cirq.T

cirq.T = cirq.T

A gate that rotates around the Z axis of the Bloch sphere.

The unitary matrix of ZPowGate(exponent=t) is:

\[
\begin{bmatrix}
1 & 0 \\
0 & g
\end{bmatrix}
\]

where:

\[ g = \exp(i \cdot t). \]

Note in particular that this gate has a global phase factor of \( e^{i \cdot t/2} \) vs the traditionally defined rotation matrices about the Pauli Z axis. See cirq.Rz for rotations without the global phase. The global phase factor can be adjusted by using the global_shift parameter when initializing.

cirq.Z, the Pauli Z gate, is an instance of this gate at exponent=1.
`cirq.TwoQubitMatrixGate`

**class** `cirq.TwoQubitMatrixGate(matrix: numpy.ndarray)`
A 2-qubit gate defined only by its matrix.

More general than specialized classes like `CZPowGate`, but more expensive and more float-error sensitive to work with (due to using eigendecompositions).

**__init__**(matrix: numpy.ndarray) → None
Initializes the 2-qubit matrix gate.

**Parameters**
- **matrix** – The matrix that defines the gate.

**Methods**

- `controlled([num_controls, control_values, ...])` Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- `num_qubits()` The number of qubits this gate acts on.

- `or(*qubits)` Returns an application of this gate to the given qubits.

- `validate_args(qubits)` Checks if this gate can be applied to the given qubits.

- `wrap_in_linear_combination([coefficient])`

**cirq.TwoQubitMatrixGate.controlled**

`TwoQubitMatrixGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate`

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- **num_controls** – Total number of control qubits.
- **control_values** – For which control qubit values to apply the sub gate. A sequence of length `num_controls` where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- **control_qid_shape** – The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to `(2,) * num_controls`. Specify this argument when using qudits.
cirq.TwoQubitMatrixGate.num_qubits

TwoQubitMatrixGate.num_qubits() → int
The number of qubits this gate acts on.

cirq.TwoQubitMatrixGate.on

TwoQubitMatrixGate.on(*qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation
Returns an application of this gate to the given qubits.

Parameters *qubits – The collection of qubits to potentially apply the gate to.

cirq.TwoQubitMatrixGate.validate_args

TwoQubitMatrixGate.validate_args(qubits)
Checks if this gate can be applied to the given qubits.
By default checks that:
• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call
super().validate_args(qubits) then do custom checks.

Parameters qubits – The sequence of qubits to potentially apply the gate to.

Throws: ValueError: The gate can’t be applied to the qubits.

cirq.TwoQubitMatrixGate.wrap_in_linear_combination

TwoQubitMatrixGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates

cirq.X

cirq.X = cirq.X

cirq.XPowGate

class cirq.XPowGate(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0)
A gate that rotates around the X axis of the Bloch sphere.
The unitary matrix of XPowGate(exponent=t) is:
$$[[g\cdot c, -i\cdot g\cdot s],$$
$$[-i\cdot g\cdot s, g\cdot c]]$$

where:

\[
\begin{align*}
c &= \cos(\pi t/2) \\
s &= \sin(\pi t/2) \\
g &= \exp(i\pi t/2).
\end{align*}
\]

Note in particular that this gate has a global phase factor of
\(e^{i\pi t/2}\) vs the traditionally defined rotation matrices
about the Pauli X axis. See \texttt{cirq.Rx} for rotations without the global
phase. The global phase factor can be adjusted by using the \texttt{global_shift}
parameter when initializing.

cirq.X, the Pauli X gate, is an instance of this gate at exponent=1.

```python
__init__(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0) -> None
```

Initializes the parameters used to compute the gate’s matrix.

The eigenvalue of each eigenspace of a gate is computed by

1. Starting with an angle in half turns as returned by the gate’s
   \_eigen\_components method:

   ```python
   + s
   ```

2. Shifting the angle by \texttt{global\_shift}:

   ```python
   ( + s) * e
   ```

3. Scaling the angle by \texttt{exponent}:

   ```python
   exp(i * pi * ( + s) * e)
   ```

4. Converting from half turns to a complex number on the unit circle:

   ```python
   exp(i * pi * ( + s) * e)
   ```

Parameters

- \texttt{exponent} – The \(t\) in gate**t. Determines how much the eigenvalues of the gate are
  scaled by. For example, eigenvectors phased by -1 when gate**1 is applied will gain a rel-
  ative phase of \(e^{i\pi exponent}\) when gate**exponent is applied (relative to eigenvectors
  unaffected by gate**1).

- \texttt{global\_shift} – Offsets the eigenvalues of the gate at exponent=1. In effect, this con-
  trols a global phase factor on the gate’s unitary matrix. The factor is:

  ```python
  exp(i * pi * global\_shift * exponent)
  ```

  For example, cirq.X**t uses a \texttt{global\_shift} of 0 but cirq.Rx(t) uses a \texttt{global\_shift} of -0.5,
  which is why cirq.unitary(cirq.Rx(pi)) equals -iX instead of X.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlled([num_controls, control_values, ...])</code></td>
<td>Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.</td>
</tr>
<tr>
<td><code>in_su2()</code></td>
<td>Returns an equal-up-global-phase gate from the group SU2.</td>
</tr>
<tr>
<td><code>num_qubits()</code></td>
<td>The number of qubits this gate acts on.</td>
</tr>
<tr>
<td><code>on(*qubits)</code></td>
<td>Returns an application of this gate to the given qubits.</td>
</tr>
<tr>
<td><code>on_each(*targets)</code></td>
<td>Returns a list of operations applying the gate to all targets.</td>
</tr>
<tr>
<td><code>validate_args(qubits)</code></td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
<tr>
<td><code>with_canonical_global_phase()</code></td>
<td>Returns an equal-up-global-phase standardized form of the gate.</td>
</tr>
<tr>
<td><code>wrap_in_linear_combination([coefficient])</code></td>
<td></td>
</tr>
</tbody>
</table>

```python
from cirq import XPowGate

XPowGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate
```

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- **num_controls**: Total number of control qubits.
- **control_values**: For which control qubit values to apply the sub gate. A sequence of length `num_controls` where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- **control_qid_shape**: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to `(2,) * num_controls`. Specify this argument when using qudits.

```python
from cirq import XPowGate

XPowGate.in_su2() → cirq.ops.common_gates.XPowGate
```

Returns an equal-up-global-phase gate from the group SU2.

```python
from cirq import XPowGate

XPowGate.num_qubits() → int
```

The number of qubits this gate acts on.
cirq.XPowGate.on

XPowGate.on (*qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation
Returns an application of this gate to the given qubits.

**Parameters**

+ **qubits** – The collection of qubits to potentially apply the gate to.

---

cirq.XPowGate.on_each

XPowGate.on_each (*targets: Union[cirq.ops.raw_types.Qid, Iterable[Any]]) → List[cirq.ops.raw_types.Operation]
Returns a list of operations applying the gate to all targets.

**Parameters**

+ **targets** – The qubits to apply this gate to.

**Returns**

Operations applying this gate to the target qubits.

**Raises**

ValueError if targets are not instances of Qid or List[Qid] –

---

cirq.XPowGate.validate_args

XPowGate.validate_args (qubits: Sequence[cirq.ops.raw_types.Qid]) → None
Checks if this gate can be applied to the given qubits.

By default checks that:

• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call super().validate_args(qubits) then do custom checks.

**Parameters**

+ **qubits** – The sequence of qubits to potentially apply the gate to.

**Throws**: ValueError: The gate can’t be applied to the qubits.

---

cirq.XPowGate.with_canonical_global_phase

XPowGate.with_canonical_global_phase () → cirq.ops.common_gates.XPowGate
Returns an equal-up-global-phase standardized form of the gate.

---

cirq.XPowGate.wrap_in_linear_combination

XPowGate.wrap_in_linear_combination (coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates
Attributes

```python
exponent
global_shift
phase_exponent
```

cirq.XPowGate.exponent

```python
property XPowGate.exponent
```

cirq.XPowGate.global_shift

```python
property XPowGate.global_shift
```

cirq.XPowGate.phase_exponent

```python
property XPowGate.phase_exponent
```

cirq.Y
cirq.Y = cirq.Y

cirq.YPowGate

```python
class cirq.YPowGate(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0)
A gate that rotates around the Y axis of the Bloch sphere.
The unitary matrix of YPowGate(exponent=t) is:

$$
\begin{bmatrix}
 g \cdot c, & -g \cdot s \\
 g \cdot s, & g \cdot c
\end{bmatrix}
$$

where:

c = \cos(\cdot t/2) \\
s = \sin(\cdot t/2) \\
g = \exp(i \cdot t/2).
```

Note in particular that this gate has a global phase factor of $e^{i \cdot t/2}$ vs the traditionally defined rotation matrices about the Pauli Y axis. See cirq.Ry for rotations without the global phase. The global phase factor can be adjusted by using the `global_shift` parameter when initializing.

cirq.Y, the Pauli Y gate, is an instance of this gate at exponent=1.
__init__(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0) → None

Initializes the parameters used to compute the gate’s matrix.

The eigenvalue of each eigenspace of a gate is computed by

1. Starting with an angle in half turns as returned by the gate’s
   _eigen_components method:

2. Shifting the angle by global_shift:

3. Scaling the angle by exponent:

4. Converting from half turns to a complex number on the unit circle:

Parameters

- **exponent** – The t in gate**t. Determines how much the eigenvalues of the gate are scaled by. For example, eigenvectors phased by -1 when gate**1 is applied will gain a relative phase of $e^{i \pi \text{exponent}}$ when gate**exponent is applied (relative to eigenvectors unaffected by gate**1).

- **global_shift** – Offsets the eigenvalues of the gate at exponent=1. In effect, this controls a global phase factor on the gate’s unitary matrix. The factor is:

  $\text{exp}(i \pi \text{global_shift} \times \text{exponent})$

  For example, cirq.X**t uses a global_shift of 0 but cirq.Rx(t) uses a global_shift of -0.5, which is why cirq.unitary(cirq.Rx(pi)) equals -iX instead of X.

Methods

- controlled([num_controls, control_values, ...])

  Returns a controlled version of this gate. If no arguments are

- in_su2()

  Returns an equal-up-global-phase gate from the group SU2.

- num_qubits()

  The number of qubits this gate acts on.

- on(*qubits)

  Returns an application of this gate to the given qubits.

- on_each(*targets)

  Returns a list of operations applying the gate to all targets.

- validate_args(qubits)

  Checks if this gate can be applied to the given qubits.

- with_canonical_global_phase()

  Returns an equal-up-global-phase standardized form of the gate.

- wrap_in_linear_combination([coefficient])


cirq.YPowGate.controlled

YPowGate.controlled \((\text{num\_controls}: \text{int} = \text{None}, \text{control\_values}: \text{Optional[Sequence[Union[int, Collection[int]]]]} = \text{None}, \text{control\_qid\_shape}: \text{Optional[Tuple[int, ...]]} = \text{None}) \rightarrow \text{cirq.ops.raw\_types.Gate}\)

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- \text{num\_controls}: Total number of control qubits.
- \text{control\_values}: For which control qubit values to apply the sub gate. A sequence of length \text{num\_controls} where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- \text{control\_qid\_shape}: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to \((2,) \ast \text{num\_controls}\). Specify this argument when using qudits.

cirq.YPowGate.in_su2

YPowGate.in_su2 () \rightarrow \text{cirq.ops.common\_gates.YPowGate}

Returns an equal-up-global-phase gate from the group SU2.

cirq.YPowGate.num_qubits

YPowGate.num_qubits () \rightarrow \text{int}

The number of qubits this gate acts on.

cirq.YPowGate.on

YPowGate.on \((*\text{qubits}: \text{cirq.ops.raw\_types.Qid}) \rightarrow \text{cirq.ops.raw\_types.Operation}\)

Returns an application of this gate to the given qubits.

- \text{Parameters \ *} \text{qubits} – The collection of qubits to potentially apply the gate to.

cirq.YPowGate.on_each

YPowGate.on_each \((*\text{targets}: \text{Union[Iterable[Qid], List[cirq.ops.raw\_types.Operation]]}) \rightarrow \text{List[cirq.ops.raw\_types.Operation]}\)

Returns a list of operations applying the gate to all targets.

- \text{Parameters \ *} \text{targets} – The qubits to apply this gate to.

- \text{Returns \ }: Operations applying this gate to the target qubits.

- \text{Raises ValueError if targets are not instances of Qid or List[Qid]} –


cirq.YPowGate.validate_args

YPowGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None
Checks if this gate can be applied to the given qubits.

By default checks that:
• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call
super().validate_args(qubits) then do custom checks.

Parameters qubits – The sequence of qubits to potentially apply the gate to.

Throws: ValueError: The gate can’t be applied to the qubits.

cirq.YPowGate.with_canonical_global_phase

YPowGate.with_canonical_global_phase() → cirq.ops.common_gates.YPowGate
Returns an equal-up-global-phase standardized form of the gate.

cirq.YPowGate.wrap_in_linear_combination

YPowGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates

Attributes

exponent

global_shift

phase_exponent


cirq.YPowGate.exponent

property YPowGate.exponent

cirq.YPowGate.global_shift

property YPowGate.global_shift

cirq.YPowGate.phase_exponent

property YPowGate.phase_exponent
cirq.Z

cirq.Z = cirq.Z

cirq.ZPowGate

class cirq.ZPowGate(*)
    exponent: Union[float, sympy.core.basic.Basic] = 1.0,
    global_shift: float = 0.0)

    A gate that rotates around the Z axis of the Bloch sphere.

    The unitary matrix of ZPowGate(exponent=t) is:

    $\begin{bmatrix}
    1 & 0 
    0 & g
    \end{bmatrix}$

    where:
    
g = \exp(i \cdot t).

    Note in particular that this gate has a global phase factor of
    \( e^{i \cdot t/2} \) vs the traditionally defined rotation matrices
    about the Pauli Z axis. See cirq.Rz for rotations without the global
    phase. The global phase factor can be adjusted by using the global_shift
    parameter when initializing.

cirq.Z, the Pauli Z gate, is an instance of this gate at exponent=1.

__init__(*
    exponent: Union[float, sympy.core.basic.Basic] = 1.0,
    global_shift: float = 0.0) → None

    Initializes the parameters used to compute the gate’s matrix.

    The eigenvalue of each eigenspace of a gate is computed by

    1. Starting with an angle in half turns as returned by the gate’s
        _eigen_components method:

        $\begin{bmatrix}
        1 & 0 
        0 & g
        \end{bmatrix}$

    2. Shifting the angle by global_shift:

        $+ s$

    3. Scaling the angle by exponent:

        $( + s) * e$

    4. Converting from half turns to a complex number on the unit circle:

        $\exp(i * \pi * ( + s) * e)$

    Parameters
• **exponent** – The t in gate**t**. Determines how much the eigenvalues of the gate are scaled by. For example, eigenvectors phased by -1 when gate**l** is applied will gain a relative phase of e^{i \pi \text{exponent}} when gate**exponent** is applied (relative to eigenvectors unaffected by gate**1**).

• **global_shift** – Offsets the eigenvalues of the gate at exponent=1. In effect, this controls a global phase factor on the gate’s unitary matrix. The factor is:

\[
\exp(i \pi \text{global_shift} \times \text{exponent})
\]

For example, `cirq.X**t` uses a `global_shift` of 0 but `cirq.Rx(t)` uses a `global_shift` of -0.5, which is why `cirq.unitary(cirq.Rx(pi))` equals -iX instead of X.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlled([num_controls, control_values, ...])</code></td>
<td>Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.</td>
</tr>
<tr>
<td><code>in_su2()</code></td>
<td>Returns an equal-up-global-phase gate from the group SU2.</td>
</tr>
<tr>
<td><code>num_qubits()</code></td>
<td>The number of qubits this gate acts on.</td>
</tr>
<tr>
<td><code>on(*qubits)</code></td>
<td>Returns an application of this gate to the given qubits.</td>
</tr>
<tr>
<td><code>on_each(*targets)</code></td>
<td>Returns a list of operations applying the gate to all targets.</td>
</tr>
<tr>
<td><code>validate_args(qubits)</code></td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
<tr>
<td><code>with_canonical_global_phase()</code></td>
<td>Returns an equal-up-global-phase standardized form of the gate.</td>
</tr>
<tr>
<td><code>wrap_in_linear_combination([coefficient])</code></td>
<td></td>
</tr>
</tbody>
</table>

### `cirq.ZPowGate.controlled`

`ZPowGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) -> cirq.ops.raw_types.Gate`

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- **num_controls**: Total number of control qubits.
- **control_values**: For which control qubit values to apply the sub gate. A sequence of length `num_controls` where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- **control_qid_shape**: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to `(2,) * num_controls`. Specify this argument when using qudits.
```python
cirq.ZPowGate.in_su2

ZPowGate.in_su2() → cirq.ops.common_gates.ZPowGate
Returns an equal-up-global-phase gate from the group SU2.

cirq.ZPowGate.num_qubits

ZPowGate.num_qubits() → int
The number of qubits this gate acts on.

cirq.ZPowGate.on

ZPowGate.on(*qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation
Returns an application of this gate to the given qubits.

Parameters

*qubits – The collection of qubits to potentially apply the gate to.

cirq.ZPowGate.on_each

ZPowGate.on_each(*targets: Union[cirq.ops.raw_types.Qid, Iterable[Any]]) → List[cirq.ops.raw_types.Operation]
Returns a list of operations applying the gate to all targets.

Parameters

*targets – The qubits to apply this gate to.

Returns

Operations applying this gate to the target qubits.

Raises

ValueError if targets are not instances of Qid or list, or

List[Qid] –

cirq.ZPowGate.validate_args

ZPowGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None
Checks if this gate can be applied to the given qubits.

By default checks that:

• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call
super().validate_args(qubits) then do custom checks.

Parameters

qubits – The sequence of qubits to potentially apply the gate to.

Throws:

ValueError: The gate can’t be applied to the qubits.
```
cirq.ZPowGate.with_canonical_global_phase

```
ZPowGate().with_canonical_global_phase() \to cirq.ops.common_gates.ZPowGate
```

Returns an equal-up-global-phase standardized form of the gate.

---

cirq.ZPowGate.wrap_in_linear_combination

```
ZPowGate().wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) \to linear_combinations.LinearCombinationOfGates
```

**Attributes**

- `exponent`
- `global_shift`

---

```
cirq.ZPowGate.exponent
```

**property** `ZPowGate.exponent`

```
cirq.ZPowGate.global_shift
```

**property** `ZPowGate.global_shift`

---

### 3.1.3 Two Qubit Unitary Gates

Unitary operations you can apply to pairs of qubits.

<table>
<thead>
<tr>
<th>Gate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNOT</td>
<td>A gate that applies a controlled power of an X gate.</td>
</tr>
<tr>
<td>CNotPowGate(*[, exponent, global_shift])</td>
<td>A gate that applies a controlled power of an X gate.</td>
</tr>
<tr>
<td>CZ</td>
<td>A gate that applies a phase to the</td>
</tr>
<tr>
<td>CZPowGate(*[, exponent, global_shift])</td>
<td>A gate that applies a phase to the</td>
</tr>
<tr>
<td>ISWAP</td>
<td>Rotates the</td>
</tr>
<tr>
<td>ISwapPowGate(*[, exponent, global_shift])</td>
<td>Rotates the</td>
</tr>
<tr>
<td>MS(rads)</td>
<td>The Mølmer–Sørensen gate, a native two-qubit operation in ion traps.</td>
</tr>
<tr>
<td>SWAP</td>
<td>The SWAP gate, possibly raised to a power.</td>
</tr>
<tr>
<td>SwapPowGate(*[, exponent, global_shift])</td>
<td>The SWAP gate, possibly raised to a power.</td>
</tr>
<tr>
<td>XX</td>
<td>The X-parity gate, possibly raised to a power.</td>
</tr>
<tr>
<td>XXPowGate(*[, exponent, global_shift])</td>
<td>The X-parity gate, possibly raised to a power.</td>
</tr>
<tr>
<td>YY</td>
<td>The Y-parity gate, possibly raised to a power.</td>
</tr>
<tr>
<td>YYPowGate(*[, exponent, global_shift])</td>
<td>The Y-parity gate, possibly raised to a power.</td>
</tr>
<tr>
<td>ZZ</td>
<td>The Z-parity gate, possibly raised to a power.</td>
</tr>
<tr>
<td>ZZPowGate(*[, exponent, global_shift])</td>
<td>The Z-parity gate, possibly raised to a power.</td>
</tr>
</tbody>
</table>
cirq.CNOT

cirq.CNOT = cirq.CNOT

A gate that applies a controlled power of an X gate.

When applying CNOT (controlled-not) to qubits, you can either use positional arguments CNOT(q1, q2), where q2 is toggled when q1 is on, or named arguments CNOT(control=q1, target=q2).

(Mixing the two is not permitted.)

The unitary matrix of CNotPowGate(exponent=t) is:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & g \cdot c & -i \cdot g \cdot s \\
0 & 0 & -i \cdot g \cdot s & g \cdot c
\end{bmatrix}
\]

where:

\[
c = \cos(\cdot t/2) \\
s = \sin(\cdot t/2) \\
g = \exp(i \cdot t/2)
\]

cirq.CNOT, the controlled NOT gate, is an instance of this gate at exponent=1.

cirq.CNotPowGate

class cirq.CNotPowGate(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0)

A gate that applies a controlled power of an X gate.

When applying CNOT (controlled-not) to qubits, you can either use positional arguments CNOT(q1, q2), where q2 is toggled when q1 is on, or named arguments CNOT(control=q1, target=q2).

(Mixing the two is not permitted.)

The unitary matrix of CNotPowGate(exponent=t) is:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & g \cdot c & -i \cdot g \cdot s \\
0 & 0 & -i \cdot g \cdot s & g \cdot c
\end{bmatrix}
\]

where:
c = cos(\cdot t/2)
\[g = \exp(i \cdot t/2).
\]

cirq.CNOT, the controlled NOT gate, is an instance of this gate at exponent=1.

```python
__init__(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0) → None
```

Initializes the parameters used to compute the gate’s matrix.

The eigenvalue of each eigenspace of a gate is computed by

1. Starting with an angle in half turns as returned by the gate’s `_eigen_components` method:

2. Shifting the angle by `global_shift`:

3. Scaling the angle by `exponent`:

4. Converting from half turns to a complex number on the unit circle:

   ```python
   \exp(i \pi (\cdot s) \cdot e)
   ```

**Parameters**

- `exponent` – The t in gate**t. Determines how much the eigenvalues of the gate are scaled by. For example, eigenvectors phased by -1 when gate**1 is applied will gain a relative phase of \exp\{i \pi \text{exponent}\} when gate**exponent is applied (relative to eigenvectors unaffected by gate**1).

- `global_shift` – Offsets the eigenvalues of the gate at exponent=1. In effect, this controls a global phase factor on the gate’s unitary matrix. The factor is:

  \exp(i \pi \cdot \text{global_shift} \cdot \text{exponent})

  For example, cirq.X**t uses a `global_shift` of 0 but cirq.Rx(t) uses a `global_shift` of -0.5, which is why cirq.unitary(cirq.Rx(pi)) equals -iX instead of X.

**Methods**

```python
controlled([num_controls, control_values, ...])
```

Returns a controlled version of this gate. If no arguments are provided...

```python
num_qubits()
```

The number of qubits this gate acts on.

Continued on next page
Cirq Documentation, Release 0.6.1

Table 26 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>on(*args, **kwargs)</code></td>
<td>Returns an application of this gate to the given qubits.</td>
</tr>
<tr>
<td><code>validate_args(qubits)</code></td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
<tr>
<td><code>wrap_in_linear_combination([coefficient])</code></td>
<td></td>
</tr>
</tbody>
</table>

**cirq.CNotPowGate.controlled**

```python
cNotPowGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) \rightarrow cirq.ops.raw_types.Gate
```

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- `num_controls`: Total number of control qubits.
- `control_values`: For which control qubit values to apply the sub gate. A sequence of length `num_controls` where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- `control_qid_shape`: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to `(2,) * num_controls`. Specify this argument when using qudits.

**cirq.CNotPowGate.num_qubits**

```python
cNotPowGate.num_qubits() \rightarrow int
```

The number of qubits this gate acts on.

**cirq.CNotPowGate.on**

```python
cNotPowGate.on(*args: cirq.ops.raw_types.Qid, **kwargs: cirq.ops.raw_types.Qid) \rightarrow cirq.ops.raw_types.Operation
```

Returns an application of this gate to the given qubits.

**Parameters**

- `*qubits` – The collection of qubits to potentially apply the gate to.

**cirq.CNotPowGate.validate_args**

```python
cNotPowGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) \rightarrow None
```

Checks if this gate can be applied to the given qubits.

By default checks that:

- inputs are of type `Qid`
- `len(qubits) == num_qubits()`
- qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call
\texttt{super().validate_args(qubits)} then do custom checks.

**Parameters** 
\texttt{qubits} – The sequence of qubits to potentially apply the gate to.

**Throws**: ValueError: The gate can’t be applied to the qubits.

\texttt{cirq.CNotPowGate.wrap_in_linear_combination}

\texttt{CNotPowGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) \rightarrow linear_combinations.LinearCombinationOfGates}

**Attributes**

- \texttt{exponent}
- \texttt{global_shift}

\texttt{cirq.CNotPowGate.exponent}

**property** \texttt{CNotPowGate.exponent}

\texttt{cirq.CNotPowGate.global_shift}

**property** \texttt{CNotPowGate.global_shift}

\texttt{cirq.CZ}

\texttt{cirq.CZ = cirq.CZ}

A gate that applies a phase to the $|11$ state of two qubits.

The unitary matrix of \texttt{CZPowGate(exponent=t)} is:

$$
\begin{bmatrix}
1, & 0, & 0, & 0, \\
0, & 1, & 0, & 0, \\
0, & 0, & 1, & 0, \\
0, & 0, & 0, & g
\end{bmatrix}
$$

where:

$$g = \exp(i \cdot t)$$

\texttt{cirq.CZ}, the controlled Z gate, is an instance of this gate at
\texttt{exponent=1}. 
cirq.CZPowGate

```python
cirq.CZPowGate(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0)
```

A gate that applies a phase to the |11 state of two qubits.

The unitary matrix of `CZPowGate(exponent=t)` is:

```
[[1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 1, 0],
 [0, 0, 0, g]]
```

where:

```
g = exp(i*pi*t).
```

cirq.CZ, the controlled Z gate, is an instance of this gate at `exponent=1`.

```python
cirq.CZPowGate.__init__(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0) → None
```

Initializes the parameters used to compute the gate’s matrix.

The eigenvalue of each eigenspace of a gate is computed by

1. Starting with an angle in half turns as returned by the gate’s `eigen_components` method:

   ```python
   _eigen_components
   ```

2. Shifting the angle by `global_shift`:

   ```python
   + s
   ```

3. Scaling the angle by `exponent`:

   ```python
   ( + s) * e
   ```

4. Converting from half turns to a complex number on the unit circle:

   ```python
   exp(i * pi * ( + s) * e)
   ```

**Parameters**

- **exponent** – The t in `gate**t`. Determines how much the eigenvalues of the gate are scaled by. For example, eigenvectors phased by -1 when `gate**I` is applied will gain a relative phase of $e^{i\pi\text{exponent}}$ when `gate**exponent` is applied (relative to eigenvectors unaffected by `gate**I`).

- **global_shift** – Offsets the eigenvalues of the gate at `exponent=1`. In effect, this controls a global phase factor on the gate’s unitary matrix. The factor is:

  ```python
  exp(i * pi * global_shift * exponent)
  ```

For example, `cirq.X**t` uses a `global_shift` of 0 but `cirq.Rx(t)` uses a `global_shift` of -0.5, which is why `cirq.unitary(cirq.Rx(pi))` equals -iX instead of X.
Methods

controlled([num_controls, control_values, ...])
Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

num_qubits()
The number of qubits this gate acts on.

on(*qubits)
Returns an application of this gate to the given qubits.

qubit_index_to_equivalence_group_key(index)
Returns a key that differs between non-interchangeable qubits.

validate_args(qubits)
Checks if this gate can be applied to the given qubits.

wrap_in_linear_combination([coefficient])

---

cirq.CZPowGate.controlled

CZPowGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- **num_controls**: Total number of control qubits.
- **control_values**: For which control qubit values to apply the sub gate. A sequence of length num_controls where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- **control_qid_shape**: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to (2,) * num_controls. Specify this argument when using qudits.

---

cirq.CZPowGate.num_qubits

CZPowGate.num_qubits() → int
The number of qubits this gate acts on.

---

cirq.CZPowGate.on

CZPowGate.on(*qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation
Returns an application of this gate to the given qubits.

**Parameters**
- **qubits**: The collection of qubits to potentially apply the gate to.
**cirq.CZPowGate.qubit_index_to_equivalence_group_key**

CZPowGate.qubit_index_to_equivalence_group_key(index: int) → int

Returns a key that differs between non-interchangeable qubits.

**cirq.CZPowGate.validate_args**

CZPowGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None

Checks if this gate can be applied to the given qubits.

By default checks that:

- inputs are of type Qid
- len(qubits) == num_qubits()
- qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call super().validate_args(qubits) then do custom checks.

**Parameters qubits** – The sequence of qubits to potentially apply the gate to.

**Throws:** ValueError: The gate can’t be applied to the qubits.

**cirq.CZPowGate.wrap_in_linear_combination**

CZPowGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates

**Attributes**

- **exponent**
- **global_shift**

**cirq.CZPowGate.exponent**

property CZPowGate.exponent

**cirq.CZPowGate.global_shift**

property CZPowGate.global_shift

**cirq.ISWAP**

cirq.ISWAP = cirq.ISWAP

Rotates the |01 vs |10 subspace of two qubits around its Bloch X-axis.
When exponent=1, swaps the two qubits and phases |01 and |10 by i. More generally, this gate’s matrix is defined as follows:

\[
\text{ISWAP}^\ast t \ exp\left(±i \ \frac{t (\text{XX} + \text{YY})}{4}\right)
\]

which is given by the matrix:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & c & i \cdot s & 0 \\
0 & i \cdot s & c & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

where:

\[c = \cos\left(\cdot \frac{t}{2}\right)\]
\[s = \sin\left(\cdot \frac{t}{2}\right)\]

\texttt{cirq.ISWAP}, the swap gate that applies i to the |01 and |10 states, is an instance of this gate at exponent=1.

\textbf{References}

“What is the matrix of the iSwap gate?”
https://quantumcomputing.stackexchange.com/questions/2594/

\texttt{cirq.ISwapPowGate}

class \texttt{cirq.ISwapPowGate}(*, exponent: \texttt{Union[float, sympy.core.basic.Basic]} = 1.0, \texttt{global_shift}: \texttt{float} = 0.0)

Rotates the |01 vs |10 subspace of two qubits around its Bloch X-axis.

When exponent=1, swaps the two qubits and phases |01 and |10 by i. More generally, this gate’s matrix is defined as follows:

\[
\text{ISWAP}^\ast t \ exp\left(±i \ \frac{t (\text{XX} + \text{YY})}{4}\right)
\]

which is given by the matrix:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & c & i \cdot s & 0 \\
0 & i \cdot s & c & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

where:

\[c = \cos\left(\cdot \frac{t}{2}\right)\]
\[s = \sin\left(\cdot \frac{t}{2}\right)\]
cirq.ISWAP, the swap gate that applies i to the |01 and |10 states, is an instance of this gate at exponent=1.

References

“What is the matrix of the iSwap gate?”
https://quantumcomputing.stackexchange.com/questions/2594/

```python
__init__(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0) → None

Initializes the parameters used to compute the gate’s matrix.

The eigenvalue of each eigenspace of a gate is computed by

1. Starting with an angle in half turns as returned by the gate’s
   `_eigen_components` method:

2. Shifting the angle by `global_shift`:

3. Scaling the angle by `exponent`:

4. Converting from half turns to a complex number on the unit circle:

   \[ \exp(i \pi \cdot (s + e)) \]

Parameters

- `exponent` – The t in gate**t. Determines how much the eigenvalues of the gate are scaled by. For example, eigenvectors phased by -1 when gate**1 is applied will gain a relative phase of \( e^{i \pi \text{exponent}} \) when gate**exponent is applied (relative to eigenvectors unaffected by gate**1).

- `global_shift` – Offsets the eigenvalues of the gate at exponent=1. In effect, this controls a global phase factor on the gate’s unitary matrix. The factor is:

   \[ \exp(i \pi \text{global_shift} \cdot \text{exponent}) \]

For example, cirq.X**t uses a `global_shift` of 0 but cirq.Rx(t) uses a `global_shift` of -0.5, which is why cirq.unitary(cirq.Rx(pi)) equals -iX instead of X.

Methods

- `controlled([num_controls, control_values, ...])` Returns a controlled version of this gate. If no arguments are

- `num_qubits()` The number of qubits this gate acts on.

Continued on next page
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>on(*qubits)</code></td>
<td>Returns an application of this gate to the given qubits.</td>
</tr>
<tr>
<td><code>qubit_index_to_equivalence_group_key(index)</code></td>
<td>Returns a key that differs between non-interchangeable qubits.</td>
</tr>
<tr>
<td><code>validate_args(qubits)</code></td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
<tr>
<td><code>wrap_in_linear_combination([coefficient])</code></td>
<td></td>
</tr>
</tbody>
</table>

**Cirq.ISwapPowGate.controlled**

```python
ISwapPowGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) \rightarrow cirq.ops.raw_types.Gate
```

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- **num_controls**: Total number of control qubits.
- **control_values**: For which control qubit values to apply the sub gate. A sequence of length `num_controls` where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- **control_qid_shape**: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to `(2,) * num_controls`. Specify this argument when using qudits.

**Cirq.ISwapPowGate.num_qubits**

```python
ISwapPowGate.num_qubits() \rightarrow int
```

The number of qubits this gate acts on.

**Cirq.ISwapPowGate.on**

```python
ISwapPowGate.on(*qubits: cirq.ops.raw_types.Qid) \rightarrow cirq.ops.raw_types.Operation
```

Returns an application of this gate to the given qubits.

- **Parameters**
  - *qubits*: The collection of qubits to potentially apply the gate to.

**Cirq.ISwapPowGate.qubit_index_to_equivalence_group_key**

```python
ISwapPowGate.qubit_index_to_equivalence_group_key(index: int) \rightarrow int
```

Returns a key that differs between non-interchangeable qubits.
**cirq.ISwapPowGate.validate_args**

```python
ISwapPowGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None
```
Checks if this gate can be applied to the given qubits.

By default checks that:

• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call
`super().validate_args(qubits)` then do custom checks.

**Parameters qubits** – The sequence of qubits to potentially apply the gate to.

**Throws:** ValueError: The gate can’t be applied to the qubits.

**cirq.ISwapPowGate.wrap_in_linear_combination**

```python
ISwapPowGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates
```

**Attributes**

```python
exponent
```

```python
global_shift
```

**cirq.ISwapPowGate.exponent**

```python
property ISwapPowGate.exponent
```

**cirq.ISwapPowGate.global_shift**

```python
property ISwapPowGate.global_shift
```

**cirq.MS**

```python
cirq.MS(rads: float) → cirq.ops.parity_gates.XXPowGate
```

The Mølmer–Sørensen gate, a native two-qubit operation in ion traps.

A rotation around the XX axis in the two-qubit bloch sphere.

The gate implements the following unitary:

\[
\exp(-i t XX) = \begin{bmatrix}
\cos(t) & 0 & 0 & -\sin(t) \\
0 & \cos(t) & -\sin(t) & 0 \\
0 & \sin(t) & \cos(t) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

(continues on next page)
\[
\begin{bmatrix}
0 & -\sin(t) & \cos(t) & 0 \\
-\sin(t) & 0 & 0 & \cos(t)
\end{bmatrix}
\]

Parameters **rads** – The rotation angle in radians.

Returns Mølmer–Sørensen gate rotating by the desired amount.

cirq.SWAP

cirq.SWAP = cirq.SWAP

The SWAP gate, possibly raised to a power. Exchanges qubits.

\[
\text{SwapPowGate()}^{*t} = \text{SwapPowGate(\text{exponent}=t)} \text{ and acts on two qubits in the computational basis as the matrix:}
\]

\[
\begin{bmatrix}
1, & 0, & 0, & 0, \\
0, & g \cdot c, & -i \cdot g \cdot s, & 0, \\
0, & -i \cdot g \cdot s, & g \cdot c, & 0, \\
0, & 0, & 0, & 1
\end{bmatrix}
\]

where:

\[
c = \cos(\cdot t/2)
\]
\[
s = \sin(\cdot t/2)
\]
\[
g = \exp(i \cdot \cdot t/2).
\]

cirq.SWAP, the swap gate, is an instance of this gate at exponent=1.

cirq.SwapPowGate

class cirq.SwapPowGate(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0)

The SWAP gate, possibly raised to a power. Exchanges qubits.

\[
\text{SwapPowGate()}^{*t} = \text{SwapPowGate(\text{exponent}=t)} \text{ and acts on two qubits in the computational basis as the matrix:}
\]

\[
\begin{bmatrix}
1, & 0, & 0, & 0, \\
0, & g \cdot c, & -i \cdot g \cdot s, & 0, \\
0, & -i \cdot g \cdot s, & g \cdot c, & 0, \\
0, & 0, & 0, & 1
\end{bmatrix}
\]

where:

\[
c = \cos(\cdot t/2)
\]
\[
s = \sin(\cdot t/2)
\]
\[
g = \exp(i \cdot \cdot t/2).
\]
cirq.SWAP, the swap gate, is an instance of this gate at exponent=1.

```python
__init__(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0) → None
```

Initializes the parameters used to compute the gate’s matrix.

The eigenvalue of each eigenspace of a gate is computed by

1. Starting with an angle in half turns as returned by the gate’s
   `_eigen_components` method:

2. Shifting the angle by `global_shift`:

   \(+ s\)

3. Scaling the angle by `exponent`:

   \(( + s) \times e\)

4. Converting from half turns to a complex number on the unit circle:

   \(\exp(i \times \pi \times ( + s) \times e)\)

**Parameters**

- `exponent` – The t in gate**t**. Determines how much the eigenvalues of the gate are scaled by. For example, eigenvectors phased by -1 when gate**1** is applied will gain a relative phase of \(e^{i \pi \text{exponent}}\) when gate**exponent** is applied (relative to eigenvectors unaffected by gate**1**).

- `global_shift` – Offsets the eigenvalues of the gate at exponent=1. In effect, this controls a global phase factor on the gate’s unitary matrix. The factor is:

  \(\exp(i \times \pi \times \text{global_shift} \times \text{exponent})\)

For example, cirq.X**t uses a `global_shift` of 0 but cirq.Rx(t) uses a `global_shift` of -0.5, which is why cirq.unitary(cirq.Rx(pi)) equals -iX instead of X.

**Methods**

- `controlled([num_controls, control_values, ...])` Returns a controlled version of this gate. If no arguments are

  `num_qubits()` The number of qubits this gate acts on.

  `on(*qubits)` Returns an application of this gate to the given qubits.

  `qubit_index_to_equivalence_group_key(index)` Returns a key that differs between non-interchangeable qubits.

  `validate_args(qubits)` Checks if this gate can be applied to the given qubits.

  `wrap_in_linear_combination([coefficient])`
Cirq Documentation, Release 0.6.1

```python
import cirq

cirq.SwapPowGate.controlled

SwapPowGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) \n\n@cirq.ops.raw_types.Gate

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

num_controls: Total number of control qubits.
control_values: For which control qubit values to apply the sub gate. A sequence of length num_controls where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
control_qid_shape: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to (2,) * num_controls. Specify this argument when using qudits.
```

```python
cirq.SwapPowGate.num_qubits

SwapPowGate.num_qubits() \n\n@cirq.ops.raw_types.Int

The number of qubits this gate acts on.
```

```python
cirq.SwapPowGate.on

SwapPowGate.on(*qubits: cirq.ops.raw_types.Qid) \n\n@cirq.ops.raw_types.Operation

Returns an application of this gate to the given qubits.

Parameters
• *qubits – The collection of qubits to potentially apply the gate to.
```

```python
cirq.SwapPowGate.qubit_index_to_equivalence_group_key

SwapPowGate.qubit_index_to_equivalence_group_key(index: int) \n\n@cirq.ops.raw_types.Int

Returns a key that differs between non-interchangeable qubits.
```

```python
cirq.SwapPowGate.validate_args

SwapPowGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) \n\nNone

Checks if this gate can be applied to the given qubits.

By default checks that:
• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit_i.dimension == qid_shape[i] for all qubits
```
Child classes can override. The child implementation should call 
`super().validate_args(qubits)` then do custom checks.

**Parameters**  
`qubits` – The sequence of qubits to potentially apply the gate to.

**Throws:**  
`ValueError`: The gate can’t be applied to the qubits.

cirq.SwapPowGate.wrap_in_linear_combination

`SwapPowGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates`

**Attributes**

`exponent`

`global_shift`

cirq.SwapPowGate.exponent

**property**  
`SwapGate.exponent`

cirq.SwapPowGate.global_shift

**property**  
`SwapGate.global_shift`

**cirq.XX**

`cirq.XX = cirq.XX`

The X-parity gate, possibly raised to a power.

At exponent=1, this gate implements the following unitary:

\[
\begin{pmatrix}
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0
\end{pmatrix}
\]

See also: `cirq.MS` (the Mølmer–Sørensen gate), which is implemented via this class.

cirq.XXPowGate

**class**  
`cirq.XXPowGate(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0)`

The X-parity gate, possibly raised to a power.

At exponent=1, this gate implements the following unitary:
\[XX = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}\]

See also: \texttt{cirq.MS} (the Mølmer–Sørensen gate), which is implemented via this class.

```python
__init__(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0) \rightarrow None
```

Initializes the parameters used to compute the gate’s matrix.

The eigenvalue of each eigenspace of a gate is computed by

1. Starting with an angle in half turns as returned by the gate’s
   \_eigen_components method:

2. Shifting the angle by \texttt{global_shift}:

3. Scaling the angle by \texttt{exponent}:

4. Converting from half turns to a complex number on the unit circle:

   \[\exp(i \times \pi \times ( + s) \times e)\]

**Parameters**

- \texttt{exponent} – The t in gate**t. Determines how much the eigenvalues of the gate are scaled by. For example, eigenvectors phased by -1 when gate**1 is applied will gain a relative phase of \(\exp(i \pi \times e)\) when gate**exponent is applied (relative to eigenvectors unaffected by gate**1).

- \texttt{global_shift} – Offsets the eigenvalues of the gate at exponent=1. In effect, this controls a global phase factor on the gate’s unitary matrix. The factor is:

\[
\exp(i \times \pi \times \text{global_shift} \times \text{exponent})
\]

For example, \texttt{cirq.X**t} uses a \texttt{global_shift} of 0 but \texttt{cirq.Rx(t)} uses a \texttt{global_shift} of -0.5, which is why \texttt{cirq.unitary(cirq.Rx(pi))} equals -iX instead of X.

**Methods**

```python
controlled([num_controls, control_values, ...])
```

Returns a controlled version of this gate. If no arguments are

```python
num_qubits()
```

The number of qubits this gate acts on.
Table 34 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>on(*qubits)</code></td>
<td>Returns an application of this gate to the given qubits.</td>
</tr>
<tr>
<td><code>qubit_index_to_equivalence_group_key(index)</code></td>
<td>Returns a key that differs between non-interchangeable qubits.</td>
</tr>
<tr>
<td><code>validate_args(qubits)</code></td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
<tr>
<td><code>wrap_in_linear_combination([coefficient])</code></td>
<td></td>
</tr>
</tbody>
</table>

`cirq.XXPowGate.controlled`

`XXPowGate.controlled` *(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate*

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- `num_controls`: Total number of control qubits.
- `control_values`: For which control qubit values to apply the sub gate. A sequence of length `num_controls` where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- `control_qid_shape`: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to `(2,) * num_controls`. Specify this argument when using qudits.

`cirq.XXPowGate.num_qubits`

`XXPowGate.num_qubits()` → int

The number of qubits this gate acts on.

`cirq.XXPowGate.on`

`XXPowGate.on (*qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation`

Returns an application of this gate to the given qubits.

**Parameters** `*qubits` – The collection of qubits to potentially apply the gate to.

`cirq.XXPowGate.qubit_index_to_equivalence_group_key`

`XXPowGate.qubit_index_to_equivalence_group_key(index: int) → int`

Returns a key that differs between non-interchangeable qubits.
**cirq.XXPowGate.validate_args**

`XXPowGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None`

Checks if this gate can be applied to the given qubits.

By default checks that:
- inputs are of type `Qid`
- `len(qubits) == num_qubits()`  
- `qubit_i.dimension == qid_shape[i]` for all qubits

Child classes can override. The child implementation should call `super().validate_args(qubits)` then do custom checks.

**Parameters**
- **qubits** – The sequence of qubits to potentially apply the gate to.

**Throws:**
- `ValueError`: The gate can’t be applied to the qubits.

**cirq.XXPowGate.wrap_in_linear_combination**

`XXPowGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates`

**Attributes**

- `exponent`
- `global_shift`

**cirq.XXPowGate.exponent**

**property** `XXPowGate.exponent`

**cirq.XXPowGate.global_shift**

**property** `XXPowGate.global_shift`

**cirq.YY**

cirq.YY = cirq.YY

The Y-parity gate, possibly raised to a power.

**cirq.YYPowGate**

**class** `cirq.YYPowGate(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0)`

The Y-parity gate, possibly raised to a power.
__init__ (*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0) → None

Initializes the parameters used to compute the gate’s matrix.

The eigenvalue of each eigenspace of a gate is computed by

1. Starting with an angle in half turns as returned by the gate’s
   _eigen_components method:

```
```

2. Shifting the angle by global_shift:

```
+ s
```

3. Scaling the angle by exponent:

```
( + s) * e
```

4. Converting from half turns to a complex number on the unit circle:

```
exp(i * pi * ( + s) * e)
```

Parameters

- **exponent** – The t in gate**t. Determines how much the eigenvalues of the gate are scaled by. For example, eigenvectors phased by -1 when gate**1 is applied will gain a relative phase of e**(i pi exponent) when gate**exponent is applied (relative to eigenvectors unaffected by gate**1).

- **global_shift** – Offsets the eigenvalues of the gate at exponent=1. In effect, this controls a global phase factor on the gate’s unitary matrix. The factor is:

```
exp(i * pi * global_shift * exponent)
```

For example, cirq.X**t uses a global_shift of 0 but cirq.Rx(t) uses a global_shift of -0.5, which is why cirq.unitary(cirq.Rx(pi)) equals -iX instead of X.

Methods

- **controlled([num_controls, control_values, ...])**
  Returns a controlled version of this gate. If no arguments are

- **num_qubits()**
  The number of qubits this gate acts on.

- **on(*qubits)**
  Returns an application of this gate to the given qubits.

- **qubit_index_to_equivalence_group_key(index)**
  Returns a key that differs between non-interchangeable qubits.

- **validate_args(qubits)**
  Checks if this gate can be applied to the given qubits.

- **wrap_in_linear_combination([coefficient])**
cirq.YYPowGate.controlled

YYPowGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- num_controls: Total number of control qubits.
- control_values: For which control qubit values to apply the sub gate. A sequence of length num_controls where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- control_qid_shape: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to (2,) * num_controls. Specify this argument when using qudits.

cirq.YYPowGate.num_qubits

YYPowGate.num_qubits() → int

The number of qubits this gate acts on.

cirq.YYPowGate.on

YYPowGate.on(*qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation

Returns an application of this gate to the given qubits.

Parameters

- *qubits – The collection of qubits to potentially apply the gate to.

cirq.YYPowGate.qubit_index_to_equivalence_group_key

YYPowGate.qubit_index_to_equivalence_group_key(index: int) → int

Returns a key that differs between non-interchangeable qubits.

cirq.YYPowGate.validate_args

YYPowGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None

Checks if this gate can be applied to the given qubits.

By default checks that:

- inputs are of type Qid
- len(qubits) == num_qubits()
- qubit_i.dimension == qid_shape[i] for all qubits
Child classes can override. The child implementation should call 
super().validate_args(qubits) then do custom checks.

Parameters qubits – The sequence of qubits to potentially apply the gate to.

Throws: ValueError: The gate can’t be applied to the qubits.

cirq.YYPowGate.wrap_in_linear_combination

YYPowGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates

Attributes

exponent
global_shift

cirq.YYPowGate.exponent

property YYPowGate.exponent

cirq.YYPowGate.global_shift

property YYPowGate.global_shift

cirq.ZZ

cirq.ZZ = cirq.ZZ

The Z-parity gate, possibly raised to a power.

The ZZ**t gate implements the following unitary:

\[
(ZZ)^t = \begin{bmatrix} 1 & \ldots \\
. & w & \ldots \\
. & . & w & . \\
. & . & . & 1 \\
\end{bmatrix}
\]

where \( w = e^{i \pi t} \) and '.' means '0'.

cirq.ZZPowGate

class cirq.ZZPowGate(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0)

The Z-parity gate, possibly raised to a power.

The ZZ**t gate implements the following unitary:
(2Z)^t = [1, . . . ]
    [., w, .]
    [., ., w.]
    [., ., ., 1]
where w = e^{i \pi t} and '.' means '0'.

__init__(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0) → None

Initializes the parameters used to compute the gate’s matrix.

The eigenvalue of each eigenspace of a gate is computed by

1. Starting with an angle in half turns as returned by the gate’s
   _eigen_components method:

2. Shifting the angle by global_shift:

3. Scaling the angle by exponent:

4. Converting from half turns to a complex number on the unit circle:

Parameters

• exponent – The t in gate**t. Determines how much the eigenvalues of the gate are
  scaled by. For example, eigenvectors phased by -1 when gate**1 is applied will gain a rel-
  ative phase of e^{i \pi exponent} when gate**exponent is applied (relative to eigenvectors
  unaffected by gate**1).

• global_shift – Offsets the eigenvalues of the gate at exponent=1. In effect, this con-
  trols a global phase factor on the gate’s unitary matrix. The factor is:

For example, cirq.X**t uses a global_shift of 0 but cirq.Rx(t) uses a global_shift of -0.5,
which is why cirq.unitary(cirq.Rx(pi)) equals -iX instead of X.

Methods

controlled([num_controls, control_values, ...])
Returns a controlled version of this gate. If no arguments are

num_qubits() The number of qubits this gate acts on.

on(*qubits) Returns an application of this gate to the given
qubits.

qubit_index_to_equivalence_group_key() Returns a key that differs between non-
interchangeable qubits.

Continued on next page
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>validate_args(qubits)</code></td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
<tr>
<td><code>wrap_in_linear_combination([coefficient])</code></td>
<td></td>
</tr>
</tbody>
</table>

**cirq.ZZPowGate.controlled**

```python
ZZPowGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) -> cirq.ops.raw_types.Gate
```

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- **num_controls**: Total number of control qubits.
- **control_values**: For which control qubit values to apply the sub gate. A sequence of length `num_controls` where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- **control_qid_shape**: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to `(2,) * num_controls`. Specify this argument when using qudits.

**cirq.ZZPowGate.num_qubits**

```python
ZZPowGate.num_qubits() -> int
```

The number of qubits this gate acts on.

**cirq.ZZPowGate.on**

```python
ZZPowGate.on(*qubits: cirq.ops.raw_types.Qid) -> cirq.ops.raw_types.Operation
```

Returns an application of this gate to the given qubits.

- **Parameters**: `*qubits` – The collection of qubits to potentially apply the gate to.

**cirq.ZZPowGate.qubit_index_to_equivalence_group_key**

```python
ZZPowGate.qubit_index_to_equivalence_group_key(index: int) -> int
```

Returns a key that differs between non-interchangeable qubits.

**cirq.ZZPowGate.validate_args**

```python
ZZPowGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) -> None
```

Checks if this gate can be applied to the given qubits.

By default checks that:
• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call
super().validate_args(qubits) then do custom checks.

**Parameters qubits** – The sequence of qubits to potentially apply the gate to.

**Throws:** ValueError: The gate can’t be applied to the qubits.

```python
cirq.ZZPowGate.wrap_in_linear_combination
```

`ZZPowGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates`

**Attributes**

```python
cirq.ZZPowGate.exponent
```

**property ZZPowGate.exponent**

```python
cirq.ZZPowGate.global_shift
```

**property ZZPowGate.global_shift**

### 3.1.4 Three Qubit Unitary Gates

Unitary operations you can apply to triplets of qubits, with helpful adjacency-respecting decompositions.

<table>
<thead>
<tr>
<th>Gate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CCX</strong></td>
<td>A Toffoli (doubly-controlled-NOT) that can be raised to a power.</td>
</tr>
<tr>
<td><strong>CCXPowGate</strong></td>
<td>A Toffoli (doubly-controlled-NOT) that can be raised to a power.</td>
</tr>
<tr>
<td><strong>CCZ</strong></td>
<td>A doubly-controlled-Z that can be raised to a power.</td>
</tr>
<tr>
<td><strong>CCZPowGate</strong></td>
<td>A doubly-controlled-Z that can be raised to a power.</td>
</tr>
<tr>
<td><strong>CSWAP</strong></td>
<td>A controlled swap gate.</td>
</tr>
<tr>
<td><strong>CSwapGate</strong></td>
<td>A controlled swap gate.</td>
</tr>
<tr>
<td><strong>FREDKIN</strong></td>
<td>A controlled swap gate.</td>
</tr>
<tr>
<td><strong>TOFFOLI</strong></td>
<td>A Toffoli (doubly-controlled-NOT) that can be raised to a power.</td>
</tr>
</tbody>
</table>
cirq.CCX

cirq.CCX = cirq.TOFOLI
A Toffoli (doubly-controlled-NOT) that can be raised to a power.

The matrix of $CCX^t$ is an 8x8 identity except the bottom right 2x2 area is the matrix of $X^t$.

cirq.CCXPowGate

class cirq.CCXPowGate(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0)
A Toffoli (doubly-controlled-NOT) that can be raised to a power.

The matrix of $CCX^t$ is an 8x8 identity except the bottom right 2x2 area is the matrix of $X^t$.

__init__(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0) → None
Initializes the parameters used to compute the gate’s matrix.

The eigenvalue of each eigenspace of a gate is computed by

1. Starting with an angle in half turns as returned by the gate’s
   _eigen_components method:

   \[
   + s
   \]

2. Shifting the angle by global_shift:

   \[
   + s
   \]

3. Scaling the angle by exponent:

   \[
   ( + s) * e
   \]

4. Converting from half turns to a complex number on the unit circle:

   \[
   \exp(i * \pi * ( + s) * e)
   \]

Parameters

- **exponent** – The t in gate**t. Determines how much the eigenvalues of the gate are scaled by. For example, eigenvectors phased by -1 when gate**I is applied will gain a relative phase of $e^{i \pi \text{exponent}}$ when gate**exponent is applied (relative to eigenvectors unaffected by gate**I).

- **global_shift** – Offsets the eigenvalues of the gate at exponent=1. In effect, this controls a global phase factor on the gate’s unitary matrix. The factor is:

   \[
   \exp(i * \pi * \text{global_shift} * \text{exponent})
   \]

For example, cirq.X**t uses a global_shift of 0 but cirq.Rx(t) uses a global_shift of -0.5, which is why cirq.unitary(cirq.Rx(pi)) equals -iX instead of X.
Methods

<table>
<thead>
<tr>
<th>controlled([num_controls, control_values, ...])</th>
<th>Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_qubits()</td>
<td>The number of qubits this gate acts on.</td>
</tr>
<tr>
<td>on(*qubits)</td>
<td>Returns an application of this gate to the given qubits.</td>
</tr>
<tr>
<td>qubit_index_to_equivalence_group_key(index)</td>
<td>Returns a key that differs between non-interchangeable qubits.</td>
</tr>
<tr>
<td>validate_args(qubits)</td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
<tr>
<td>wrap_in_linear_combination([coefficient])</td>
<td></td>
</tr>
</tbody>
</table>

cirq.CCXPowGate.controlled

CCXPowGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

num_controls: Total number of control qubits.
control_values: For which control qubit values to apply the sub gate. A sequence of length num_controls where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
control_qid_shape: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to (2,) * num_controls. Specify this argument when using qudits.

cirq.CCXPowGate.num_qubits

CCXPowGate.num_qubits() → int

The number of qubits this gate acts on.

cirq.CCXPowGate.on

CCXPowGate.on(*qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation

Returns an application of this gate to the given qubits.

Parameters *qubits – The collection of qubits to potentially apply the gate to.
**cirq.CCXPowGate.qubit_index_to_equivalence_group_key**

CCXPowGate.qubit_index_to_equivalence_group_key(index)

Returns a key that differs between non-interchangeable qubits.

**cirq.CCXPowGate.validate_args**

CCXPowGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None

Checks if this gate can be applied to the given qubits.

By default checks that:
- inputs are of type Qid
- len(qubits) == num_qubits()
- qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call super().validate_args(qubits) then do custom checks.

**Parameters qubits** – The sequence of qubits to potentially apply the gate to.

**Throws**: ValueError: The gate can’t be applied to the qubits.

**cirq.CCXPowGate.wrap_in_linear_combination**

CCXPowGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates

**Attributes**

```
exponent

property CCXPowGate.exponent

global_shift

property CCXPowGate.global_shift
```

**cirq.CCZ**

cirq.CCZ = cirq.CCZ

A doubly-controlled-Z that can be raised to a power.
The matrix of $CCZ^t$ is \( \text{diag}(1, 1, 1, 1, 1, 1, 1, \exp(i \pi t)) \).

cirq.CCZPowGate

class cirq.CCZPowGate(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0)
A doubly-controlled-Z that can be raised to a power.

The matrix of $CCZ^t$ is \( \text{diag}(1, 1, 1, 1, 1, 1, 1, \exp(i \pi t)) \).

__init__(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0) \rightarrow None

Initializes the parameters used to compute the gate’s matrix.

The eigenvalue of each eigenspace of a gate is computed by

1. Starting with an angle in half turns as returned by the gate’s _eigen_components method:

2. Shifting the angle by global_shift:

3. Scaling the angle by exponent:

4. Converting from half turns to a complex number on the unit circle:

\[ \exp(i \pi \times ( + s) \times e) \]

Parameters

- \textbf{exponent} – The t in \( \text{gate}^t \). Determines how much the eigenvalues of the gate are scaled by. For example, eigenvectors phased by -1 when \( \text{gate}^1 \) is applied will gain a relative phase of \( \exp(i \pi \times \text{exponent}) \) when \( \text{gate}^\text{exponent} \) is applied (relative to eigenvectors unaffected by \( \text{gate}^1 \)).

- \textbf{global_shift} – Offsets the eigenvalues of the gate at exponent=1. In effect, this controls a global phase factor on the gate’s unitary matrix. The factor is:

\[ \exp(i \times \pi \times ( + s) \times e) \]

Methods

controlled([num_controls, control_values, ...])
Returns a controlled version of this gate. If no arguments are provided, defaults are used.

num_qubits()
The number of qubits this gate acts on.

Continued on next page
Table 43 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>on(*qubits)</code></td>
<td>Returns an application of this gate to the given qubits.</td>
</tr>
<tr>
<td><code>qubit_index_to_equivalence_group_key(index)</code></td>
<td>Returns a key that differs between non-interchangeable qubits.</td>
</tr>
<tr>
<td><code>validate_args(qubits)</code></td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
<tr>
<td><code>wrap_in_linear_combination([coefficient])</code></td>
<td></td>
</tr>
</tbody>
</table>

```python
cirq.CCZPowGate.controlled
```

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

```python
cirq.CCZPowGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate
```

- `num_controls`: Total number of control qubits.
- `control_values`: For which control qubit values to apply the sub gate. A sequence of length `num_controls` where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- `control_qid_shape`: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to `(2,) * num_controls`. Specify this argument when using qudits.

```python
cirq.CCZPowGate.num_qubits
```

The number of qubits this gate acts on.

```python
cirq.CCZPowGate.on
```

Returns an application of this gate to the given qubits.

```python
cirq.CCZPowGate.on(*qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation
```

- **Parameters**
  - `*qubits` – The collection of qubits to potentially apply the gate to.

```python
cirq.CCZPowGate.qubit_index_to_equivalence_group_key
```

Returns a key that differs between non-interchangeable qubits.


```python
cirq.CCZPowGate.validate_args

CCZPowGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None

Checks if this gate can be applied to the given qubits.

By default checks that:
  • inputs are of type Qid
  • len(qubits) == num_qubits()
  • qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call super().validate_args(qubits) then do custom checks.

Parameters qubits – The sequence of qubits to potentially apply the gate to.

Throws: ValueError: The gate can’t be applied to the qubits.
```

```python
cirq.CCZPowGate.wrap_in_linear_combination

CCZPowGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates

Attributes

``` exponent

cirq.CCZPowGate.exponent

``` property CCZPowGate.exponent

cirq.CCZPowGate.global_shift

``` property CCZPowGate.global_shift

cirq.CSWAP

cirq.CSWAP = cirq.FREDKIN

A controlled swap gate. The Fredkin gate.

cirq.CSwapGate

class cirq.CSwapGate

A controlled swap gate. The Fredkin gate.
__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

circq.CSwapGate.controlled

CSwapGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → circq.ops.raw_types.Gate

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

num_controls: Total number of control qubits.
control_values: For which control qubit values to apply the sub gate. A sequence of length num_controls where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
control_qid_shape: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to (2,) * num_controls. Specify this argument when using qudits.

circq.CSwapGate.num_qubits

CSwapGate.num_qubits() → int
The number of qubits this gate acts on.

circq.CSwapGate.on

CSwapGate.on(*qubits) → circq.ops.raw_types.Operation
Returns an application of this gate to the given qubits.

Parameters *qubits – The collection of qubits to potentially apply the gate to.
Cirq Documentation, Release 0.6.1

### `cirq.CSwapGate.qubit_index_to_equivalence_group_key`

`CSwapGate.qubit_index_to_equivalence_group_key(index)`

Returns a key that differs between non-interchangeable qubits.

### `cirq.CSwapGate.validate_args`

`CSwapGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None`

Checks if this gate can be applied to the given qubits.

By default checks that:

- inputs are of type `Qid`
- `len(qubits) == num_qubits()`  
- `qubit_i.dimension == qid_shape[i]` for all qubits

Child classes can override. The child implementation should call `super().validate_args(qubits)` then do custom checks.

**Parameters** `qubits` – The sequence of qubits to potentially apply the gate to.

**Throws:** `ValueError`: The gate can’t be applied to the qubits.

### `cirq.CSwapGate.wrap_in_linear_combination`

`CSwapGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates`

### `cirq.FREDKIN`

`cirq.FREDKIN = cirq.FREDKIN`

A controlled swap gate. The Fredkin gate.

### `cirq.TOFFOLI`

`cirq.TOFFOLI = cirq.TOFFOLI`

A Toffoli (doubly-controlled-NOT) that can be raised to a power.

The matrix of $\text{CCX}^n$ is an 8x8 identity except the bottom right 2x2 area is the matrix of $\text{X}^n$.

#### 3.1.5 Multiqubit Unitary Gates

Some gates can be applied to arbitrary number of qubits
A Gate that perform no operation on qubits.

`cirq.IdentityGate(num_qubits, qid_shape)` A Gate that perform no operation on qubits.

The unitary matrix of this gate is a diagonal matrix with all 1s on the diagonal and all 0s off the diagonal in any basis.

cirq.I is the single qubit identity gate.

### cirq.IdentityGate

**class cirq.IdentityGate**(num_qubits: Optional[int] = None, qid_shape: Tuple[int, ...] = None)

A Gate that perform no operation on qubits.

The unitary matrix of this gate is a diagonal matrix with all 1s on the diagonal and all 0s off the diagonal in any basis.

cirq.I is the single qubit identity gate.

**__init__**(num_qubits: Optional[int] = None, qid_shape: Tuple[int, ...] = None)

**Parameters**

- **num_qubits** –
- **qid_shape** – Specifies the dimension of each qid the measurement applies to. The default is 2 for every qubit.

**Raises** `ValueError` – If the length of qid_shape doesn’t equal num_qubits.

**Methods**

**controlled**([num_controls, control_values, ...]) Returns a controlled version of this gate. If no arguments are

**num_qubits()** The number of qubits this gate acts on.

**on**(*qubits) Returns an application of this gate to the given qubits.

**on_each**(*targets) Returns a list of operations that applies the single qubit identity
Table 47 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>validate_args(qubits)</code></td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
<tr>
<td><code>wrap_in_linear_combination([coefficient])</code></td>
<td></td>
</tr>
</tbody>
</table>

**cirq.IdentityGate.controlled**

```python
IdentityGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) -> cirq.ops.raw_types.Gate
```

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- **num_controls**: Total number of control qubits.
- **control_values**: For which control qubit values to apply the sub gate. A sequence of length `num_controls` where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- **control_qid_shape**: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to `(2,) * num_controls`. Specify this argument when using qudits.

**cirq.IdentityGate.num_qubits**

```python
IdentityGate.num_qubits() -> int
```

The number of qubits this gate acts on.

**cirq.IdentityGate.on**

```python
IdentityGate.on(*qubits: cirq.ops.raw_types.Qid) -> cirq.ops.raw_types.Operation
```

Returns an application of this gate to the given qubits.

- **Parameters** `*qubits` – The collection of qubits to potentially apply the gate to.

**cirq.IdentityGate.on_each**

```python
IdentityGate.on_each(*targets: Union[cirq.ops.raw_types.Qid, Iterable[Any]]) -> List[cirq.ops.raw_types.Operation]
```

Returns a list of operations that applies the single qubit identity to each of the targets.

- **Parameters** `*targets` – The qubits to apply this gate to.
- **Returns** Operations applying this gate to the target qubits.
Raises

- ValueError if targets are not instances of Qid or List[Qid]
- the gate from which this is applied is not a single qubit identity
- gate.

**cirq.IdentityGate.validate_args**

IdentityGate.validate_args (qubits: Sequence[cirq.ops.raw_types.Qid]) → None

Checks if this gate can be applied to the given qubits.

By default checks that:

- inputs are of type Qid
- len(qubits) == num_qubits()
- qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call
super().validate_args(qubits) then do custom checks.

**Parameters** qubits – The sequence of qubits to potentially apply the gate to.

**Throws:** ValueError: The gate can’t be applied to the qubits.

**cirq.IdentityGate.wrap_in_linear_combination**

IdentityGate.wrap_in_linear_combination (coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates

3.1.6 Measurements

Measurement can be on multiple qubits. Currently only measurement in computational basis is supported.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>measure(*target[, key, invert_mask])</td>
<td>Returns a single MeasurementGate applied to all the given qubits.</td>
</tr>
<tr>
<td>measure_each(*qubits[, key_func])</td>
<td>Returns a list of operations individually measuring the given qubits.</td>
</tr>
<tr>
<td>MeasurementGate([num_qubits, key, ...])</td>
<td>A gate that measures qubits in the computational basis.</td>
</tr>
</tbody>
</table>

3.1.7 Channels and Mixture Gate

Non-unitary gates. Mixture gates are those that can be interpreted as applying a unitary for a fixed probability while channel encompasses the more general concept of a noisy open system quantum evolution.
### Generalized Amplitude Damping

Amplitude damping is a widely used model that simulates the dissipation of quantum information. The `amplitude_damp` function in Cirq provides a straightforward way to model amplitude damping in your quantum circuits.

#### Function Definition

```python
cirq.amplitude_damp(gamma: float) -> cirq.ops.common_channels.AmplitudeDampingChannel
```

- **Purpose**: Returns an AmplitudeDampingChannel with the given probability gamma.
- **Parameters**:
  - `gamma`: the probability of the interaction being dissipative.

#### Mathematical Description

Amplitude damping is a type of dephasing channel where the amplitude of a qubit's state is reduced, typically due to dissipative effects. The evolution of a density matrix under amplitude damping is described by the following equation:

$$ \rho \rightarrow M_0 \rho M_0^\dagger + M_1 \rho M_1^\dagger $$

where:

- $M_0 = \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1 - \gamma} \end{bmatrix}$
- $M_1 = \begin{bmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{bmatrix}$

#### Example

Consider a qubit with a density matrix $\rho$. Under amplitude damping, the evolution of $\rho$ can be computed using the AmplitudeDampingChannel. The channel effectively reduces the amplitude of the qubit's state, simulating the loss of quantum coherence.

#### Notes

- It's important to ensure that `gamma` is a valid probability, otherwise a `ValueError` will be raised.
- The channel `amplitude_damp` is a convenient interface for incorporating amplitude damping into quantum circuits, allowing for the simulation of real-world quantum systems where qubits lose coherence due to environmental effects.
cirq.AmplitudeDampingChannel

class cirq.AmplitudeDampingChannel (gamma: float)
Dampen qubit amplitudes through dissipation.

This channel models the effect of energy dissipation to the surrounding environment.

__init__ (gamma: float) → None
The amplitude damping channel.

Construct a channel that dissipates energy. The probability of energy exchange occurring is given by gamma.

This channel evolves a density matrix as follows:

$$ \rho \rightarrow M_0 \rho M_0^\dagger + M_1 \rho M_1^\dagger $$

With:

$$ \begin{aligned} M_0 &= \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1 - \gamma} \end{bmatrix} \\
M_1 &= \begin{bmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{bmatrix} \end{aligned} $$

Parameters

- gamma – the probability of the interaction being dissipative.

Raises

- ValueError – is gamma is not a valid probability.

Methods

controlled ([num_controls, control_values, ...])
Returns a controlled version of this gate. If no arguments are

num_qubits ()
The number of qubits this gate acts on.

on (*qubits)
Returns an application of this gate to the given qubits.

on_each (*targets)
Returns a list of operations applying the gate to all targets.

validate_args (qubits)
Checks if this gate can be applied to the given qubits.

wrap_in_linear_combination ([coefficient])

cirq.AmplitudeDampingChannel.controlled

AmplitudeDampingChannel.controlled (num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate
Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

num_controls: Total number of control qubits.
control_values: For which control qubit values to apply the sub gate. A sequence of length num_controls where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
control_qid_shape: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to (2,) * num_controls. Specify this argument when using qudits.

cirq.AmplitudeDampingChannel.num_qubits

AmplitudeDampingChannel.num_qubits() \rightarrow \text{int}

The number of qubits this gate acts on.

cirq.AmplitudeDampingChannel.on

AmplitudeDampingChannel.on(qubits: \text{cirq.ops.raw_types.Qid}) \rightarrow \text{cirq.ops.raw_types.Operation}

Returns an application of this gate to the given qubits.

Parameters *qubits – The collection of qubits to potentially apply the gate to.

cirq.AmplitudeDampingChannel.on_each

AmplitudeDampingChannel.on_each(targets: \text{Union[cirq.ops.raw_types.Qid, Iterable[\text{Any}]]}) \rightarrow \text{List[cirq.ops.raw_types.Operation]}

Returns a list of operations applying the gate to all targets.

Parameters *targets – The qubits to apply this gate to.

Returns Operations applying this gate to the target qubits.

Raises ValueError if targets are not instances of Qid or List[Qid] –

cirq.AmplitudeDampingChannel.validate_args

AmplitudeDampingChannel.validate_args(qubits: \text{Sequence[cirq.ops.raw_types.Qid]}) \rightarrow \text{None}

Checks if this gate can be applied to the given qubits.

By default checks that:
• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call
super().validate_args(qubits) then do custom checks.

Parameters qubits – The sequence of qubits to potentially apply the gate to.

Throws: ValueError: The gate can’t be applied to the qubits.

cirq.AmplitudeDampingChannel.wrap_in_linear_combination

AmplitudeDampingChannel.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) \rightarrow linear_combinations.LinearCombinationOfGates

Attributes

gamma The probability of the interaction being dissipative.

cirq.AmplitudeDampingChannel.gamma

property AmplitudeDampingChannel.gamma
The probability of the interaction being dissipative.

cirq.asymmetric_depolarize

cirq.asymmetric depolarize(p_x: float, p_y: float, p_z: float) \rightarrow cirq.ops.common_channels.AsymmetricDepolarizingChannel

Returns a AsymmetricDepolarizingChannel with given parameter.

This channel evolves a density matrix via

\[ \rho \rightarrow (1 - p_x - p_y - p_z) \rho + p_x X \rho X + p_y Y \rho Y + p_z Z \rho Z \]

Parameters

• p_x – The probability that a Pauli X and no other gate occurs.

• p_y – The probability that a Pauli Y and no other gate occurs.

• p_z – The probability that a Pauli Z and no other gate occurs.

Raises ValueError – if the args or the sum of the args are not probabilities.

cirq.AsymmetricDepolarizingChannel

class cirq.AsymmetricDepolarizingChannel(p_x: float, p_y: float, p_z: float)

A channel that depolarizes asymmetrically along different directions.
__init__(p_x: float, p_y: float, p_z: float) \rightarrow None
The asymmetric depolarizing channel.
This channel applies one of four disjoint possibilities: nothing (the identity channel) or one of the three pauli gates. The disjoint probabilities of the three gates are $p_x$, $p_y$, and $p_z$ and the identity is done with probability $1 - p_x - p_y - p_z$. The supplied probabilities must be valid probabilities and the sum $p_x + p_y + p_z$ must be a valid probability or else this constructor will raise a ValueError.

This channel evolves a density matrix via

$$ \rho \rightarrow (1 - p_x - p_y - p_z) \rho + p_x X \rho X + p_y Y \rho Y + p_z Z \rho Z $$

**Parameters**

- $p_x$ – The probability that a Pauli X and no other gate occurs.
- $p_y$ – The probability that a Pauli Y and no other gate occurs.
- $p_z$ – The probability that a Pauli Z and no other gate occurs.

** Raises ** ValueError – if the args or the sum of args are not probabilities.

**Methods**

- `controlled([num_controls, control_values, ...])` Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- `num_qubits()` The number of qubits this gate acts on.

- `on(*qubits)` Returns an application of this gate to the given qubits.

- `on_each(*targets)` Returns a list of operations applying the gate to all targets.

- `validate_args(qubits)` Checks if this gate can be applied to the given qubits.

- `wrap_in_linear_combination([coefficient])`

cirq.AsymmetricDepolarizingChannel.controlled

AsymmetricDepolarizingChannel.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- num_controls: Total number of control qubits.
- control_values: For which control qubit values to apply the sub gate. A sequence of length num_controls where each entry is an integer (or set of integers) corresponding to the
qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.

control_qid_shape: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to \((2,) \ast \text{num}\_\text{controls}\). Specify this argument when using qudits.

cirq.AsymmetricDepolarizingChannel.num_qubits

AsymmetricDepolarizingChannel.num_qubits() \(\rightarrow\) int

The number of qubits this gate acts on.

cirq.AsymmetricDepolarizingChannel.on

AsymmetricDepolarizingChannel.on(*qubits: cirq.ops.raw_types.Qid) \(\rightarrow\) cirq.ops.raw_types.Operation

Returns an application of this gate to the given qubits.

Parameters *qubits – The collection of qubits to potentially apply the gate to.

cirq.AsymmetricDepolarizingChannel.on_each

AsymmetricDepolarizingChannel.on_each(*targets: Union[cirq.ops.raw_types.Qid, Iterable[Any]]) \(\rightarrow\) List[cirq.ops.raw_types.Operation]

Returns a list of operations applying the gate to all targets.

Parameters *targets – The qubits to apply this gate to.

Returns Operations applying this gate to the target qubits.

Raises ValueError if targets are not instances of Qid or List[Qid] –

cirq.AsymmetricDepolarizingChannel.validate_args

AsymmetricDepolarizingChannel.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) \(\rightarrow\) None

Checks if this gate can be applied to the given qubits.

By default checks that:

• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call super().validate_args(qubits) then do custom checks.
**Parameters** `qubits` – The sequence of qubits to potentially apply the gate to.

**Throws:** `ValueError`: The gate can’t be applied to the qubits.

```python
cirq.AsymmetricDepolarizingChannel.wrap_in_linear_combination
```

`AsymmetricDepolarizingChannel.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates`

**Attributes**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>p_x</code></td>
<td>The probability that a Pauli X and no other gate occurs.</td>
</tr>
<tr>
<td><code>p_y</code></td>
<td>The probability that a Pauli Y and no other gate occurs.</td>
</tr>
<tr>
<td><code>p_z</code></td>
<td>The probability that a Pauli Z and no other gate occurs.</td>
</tr>
</tbody>
</table>

```python
cirq.AsymmetricDepolarizingChannel.p_x
```

**property** `AsymmetricDepolarizingChannel.p_x`  
The probability that a Pauli X and no other gate occurs.

```python
cirq.AsymmetricDepolarizingChannel.p_y
```

**property** `AsymmetricDepolarizingChannel.p_y`  
The probability that a Pauli Y and no other gate occurs.

```python
cirq.AsymmetricDepolarizingChannel.p_z
```

**property** `AsymmetricDepolarizingChannel.p_z`  
The probability that a Pauli Z and no other gate occurs.

```python
cirq.bit_flip
```

`cirq.bit_flip(p: Optional[float] = None) → Union[cirq.ops.common_gates.XPowGate, cirq.ops.common_channels.BitFlipChannel]`

Construct a BitFlipChannel that flips a qubit state with probability of a flip given by p. If p is None, return a guaranteed flip in the form of an X operation.

This channel evolves a density matrix via

$$ \rho \rightarrow M_0 \rho M_0^\dagger + M_1 \rho M_1^\dagger $$
With

\[
\begin{aligned}
M_0 &= \sqrt{p} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\
M_1 &= \sqrt{1-p} \begin{bmatrix} 0 & 1 \\ 1 & -0 \end{bmatrix}
\end{aligned}
\]

**Parameters** `p` – the probability of a bit flip.

**Raises** `ValueError` – if `p` is not a valid probability.

cirq.BitFlipChannel

class cirq.BitFlipChannel (p: float)

Probabilistically flip a qubit from 1 to 0 state or vice versa.

__init__ (p: float) → None

The bit flip channel.

Construct a channel that flips a qubit with probability `p`.

This channel evolves a density matrix via:

\[
\rho \rightarrow M_0 \rho M_0^\dagger + M_1 \rho M_1^\dagger
\]

With:

\[
\begin{aligned}
M_0 &= \sqrt{1 - p} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\
M_1 &= \sqrt{p} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}
\end{aligned}
\]

**Parameters** `p` – the probability of a bit flip.

**Raises** `ValueError` – if `p` is not a valid probability.

**Methods**

controlled([num_controls, control_values, ...])

Returns a controlled version of this gate. If no arguments are

num_qubits() The number of qubits this gate acts on.

on(*qubits) Returns an application of this gate to the given qubits.

on_each(*targets) Returns a list of operations applying the gate to all targets.

validate_args(qubits) Checks if this gate can be applied to the given qubits.

wrap_in_linear_combination([coefficient])

cirq.BitFlipChannel.controlled

BitFlipChannel.controlled (num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate
Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

**num_controls**: Total number of control qubits.
**control_values**: For which control qubit values to apply the sub gate. A sequence of length `num_controls` where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
**control_qid_shape**: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to `(2,) * num_controls`. Specify this argument when using qudits.

cirq.BitFlipChannel.num_qubits

```python
BitFlipChannel.num_qubits() \to int
The number of qubits this gate acts on.
```

cirq.BitFlipChannel.on

```python
BitFlipChannel.on(*qubits: cirq.ops.raw_types.Qid) \to cirq.ops.raw_types.Operation
Returns an application of this gate to the given qubits.

Parameters

* qubits – The collection of qubits to potentially apply the gate to.
```

cirq.BitFlipChannel.on_each

```python
BitFlipChannel.on_each(*targets: Union[cirq.ops.raw_types.Qid, Iterable[any]]) \to List[cirq.ops.raw_types.Operation]
Returns a list of operations applying the gate to all targets.

Parameters

* targets – The qubits to apply this gate to.

Returns

Operations applying this gate to the target qubits.

Raises `ValueError` if targets are not instances of `Qid` or `List[Qid]` –

```

cirq.BitFlipChannel.validate_args

```python
BitFlipChannel.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) \to None
Checks if this gate can be applied to the given qubits.

By default checks that:

* inputs are of type `Qid`
* len(qubits) == num_qubits()
* qubit_i.dimension == qid_shape[i] for all qubits
Child classes can override. The child implementation should call `super().validate_args(qubits)` then do custom checks.

**Parameters**

default.qubits – The sequence of qubits to potentially apply the gate to.

**Throws:** ValueError: The gate can’t be applied to the qubits.

```
cirq.BitFlipChannel.wrap_in_linear_combination
```

BitFlipChannel.\texttt{wrap\_in\_linear\_combination}\texttt{(coefficient:\ Union[complex, float, int] = I)} \rightarrow \texttt{linear\_combinations.LinearCombinationOfGates}

**Attributes**

\texttt{p} \hspace{1cm} The probability of a bit flip.

```
cirq.BitFlipChannel.p
```

**property** BitFlipChannel.\texttt{p}

The probability of a bit flip.

```
cirq.depolarize
```

cirq.\texttt{depolarize}(p: float) \rightarrow cirq.\texttt{ops.common\_channels.DebolizingChannel}

Returns a DepolarizingChannel with given probability of error.

This channel applies one of four disjoint possibilities: nothing (the identity channel) or one of the three pauli gates. The disjoint probabilities of the three gates are all the same, \(p / 3\), and the identity is done with probability \(1 - p\). The supplied probability must be a valid probability or else this constructor will raise a ValueError.

This channel evolves a density matrix via

$$
\rho \rightarrow (1 - p) \rho + (p / 3) X \rho X + (p / 3) Y \rho Y + (p / 3) Z \rho Z
$$

**Parameters**
p – The probability that one of the Pauli gates is applied. Each of the Pauli gates is applied independently with probability \(p / 3\).

**Raises** ValueError – if \(p\) is not a valid probability.

```
cirq.DebolizingChannel
```

cirq.\texttt{DepolarizingChannel}(p: float)

A channel that depolarizes a qubit.
__init__ (p: float) → None

The symmetric depolarizing channel.

This channel applies one of four disjoint possibilities: nothing (the identity channel) or one of the three pauli gates. The disjoint probabilities of the three gates are all the same, p / 3, and the identity is done with probability 1 - p. The supplied probability must be a valid probability or else this constructor will raise a ValueError.

This channel evolves a density matrix via

$$ \rho \rightarrow (1 - p) \rho + (p / 3) X \rho X + (p / 3) Y \rho Y + (p / 3) Z \rho Z $$

**Parameters**

\(p\) – The probability that one of the Pauli gates is applied. Each of the Pauli gates is applied independently with probability \(p / 3\).

**Raises**

ValueError – if \(p\) is not a valid probability.

**Methods**

**controlled([num_controls, control_values, ...])**

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

**num_qubits()**

The number of qubits this gate acts on.

**on(*qubits)**

Returns an application of this gate to the given qubits.

**on_each(*targets)**

Returns a list of operations applying the gate to all targets.

**validate_args(qubits)**

Checks if this gate can be applied to the given qubits.

**wrap_in_linear_combination([coefficient])**

**cirq.DependorizingChannel.controlled**

DepolarizingChannel.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

num_controls: Total number of control qubits.
control_values: For which control qubit values to apply the sub gate. A sequence of length num_controls where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
control_qid_shape: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to (2,) * num_controls. Specify this argument when using qudits.

```
cirq.DepolarizingChannel.num_qubits
```

DepolarizingChannel.num_qubits() → int

The number of qubits this gate acts on.

```
cirq.DepolarizingChannel.on
```

DepolarizingChannel.on(*qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation

Returns an application of this gate to the given qubits.

**Parameters**

*qubits – The collection of qubits to potentially apply the gate to.

```
cirq.DepolarizingChannel.on_each
```

DepolarizingChannel.on_each(*targets: Union[cirq.ops.raw_types.Qid, Iterable[Any]]) → List[cirq.ops.raw_types.Operation]

Returns a list of operations applying the gate to all targets.

**Parameters**

*targets – The qubits to apply this gate to.

**Returns**

Operations applying this gate to the target qubits.

**Raises**

ValueError if targets are not instances of Qid or List[Qid] –

```
cirq.DepolarizingChannel.validate_args
```

DepolarizingChannel.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None

Checks if this gate can be applied to the given qubits.

By default checks that:

- inputs are of type Qid
- len(qubits) == num_qubits()
- qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call super().validate_args(qubits) then do custom checks.

**Parameters**

qubits – The sequence of qubits to potentially apply the gate to.

**Throws:**

ValueError: The gate can’t be applied to the qubits.
cirq.DepolarizingChannel.wrap_in_linear_combination

```
DepolarizingChannel.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates
```

**Attributes**

- `p`: The probability that one of the Pauli gates is applied.

**cirq.DepolarizingChannel.p**

```
property DepolarizingChannel.p
The probability that one of the Pauli gates is applied.
Each of the Pauli gates is applied independently with probability p / 3.
```

cirq.generalized_amplitude_damp

```
cirq.generalized_amplitude_damp(p: float, gamma: float) → cirq.ops.common_channels.GeneralizedAmplitudeDampingChannel
```

Returns a GeneralizedAmplitudeDampingChannel with the given probabilities `gamma` and `p`.
This channel evolves a density matrix via:

$$ \rho \rightarrow M_0 \rho M_0^\dagger + M_1 \rho M_1^\dagger + M_2 \rho M_2^\dagger + M_3 \rho M_3^\dagger $$

With:

```
M_0 = \sqrt{p} \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1 - \gamma} \end{bmatrix} \\
M_1 = \sqrt{p} \begin{bmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{bmatrix} \\
M_2 = \sqrt{1-p} \begin{bmatrix} \sqrt{1-\gamma} & 0 \\ 0 & 1 \end{bmatrix} \\
M_3 = \sqrt{1-p} \begin{bmatrix} 0 & 0 \\ \sqrt{\gamma} & 0 \end{bmatrix}
```

**Parameters**

- `gamma` – the probability of the interaction being dissipative.
- `p` – the probability of the qubit and environment exchanging energy.

**Raises** `ValueError` – `gamma` or `p` is not a valid probability.

cirq.GeneralizedAmplitudeDampingChannel

```
cirq.GeneralizedAmplitudeDampingChannel(p: float, gamma: float)
```

Dampen qubit amplitudes through non ideal dissipation.

This channel models the effect of energy dissipation into the environment.
as well as the environment depositing energy into the system.

```python
__init__(p: float, gamma: float) \rightarrow None

The generalized amplitude damping channel.

Construct a channel to model energy dissipation into the environment as well as the environment depositing energy into the system. The probabilities with which the energy exchange occur are given by \( \gamma \), and the probability of the environment being not excited is given by \( p \).

The stationary state of this channel is the diagonal density matrix with probability \( p \) of being \( |0 \rangle \) and probability \( 1 - p \) of being \( |1 \rangle \).

This channel evolves a density matrix via

\[
\rho \rightarrow M_0 \rho M_0^\dagger + M_1 \rho M_1^\dagger + M_2 \rho M_2^\dagger + M_3 \rho M_3^\dagger
\]

With

\[
\begin{aligned}
M_0 &= \sqrt{p} \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1 - \gamma} \end{bmatrix} \\
M_1 &= \sqrt{p} \begin{bmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{bmatrix} \\
M_2 &= \sqrt{1-p} \begin{bmatrix} \sqrt{1-\gamma} & 0 \\ 0 & 1 \end{bmatrix} \\
M_3 &= \sqrt{1-p} \begin{bmatrix} 0 & 0 \\ \sqrt{\gamma} & 0 \end{bmatrix}
\end{aligned}
\]

Parameters

- \( \gamma \) – the probability of the interaction being dissipative.
- \( p \) – the probability of the qubit and environment exchanging energy.

Raises \texttt{ValueError} – if \( \gamma \) or \( p \) is not a valid probability.

Methods

```python
controlled([num_controls, control_values, ...])

Returns a controlled version of this gate. If no arguments are

```python
num_qubits()

The number of qubits this gate acts on.

```python
on(*qubits)

Returns an application of this gate to the given qubits.

```python
on_each(*targets)

Returns a list of operations applying the gate to all targets.

```python
validate_args(qubits)

Checks if this gate can be applied to the given qubits.

Continued on next page
Table 58 – continued from previous page

\[
\text{\textit{wrap_in_linear_combination}}(\text{[coefficient]})
\]

cirq.GeneralizedAmplitudeDampingChannel.controlled

\[
\text{GeneralizedAmplitudeDampingChannel.controlled}(\text{num_controls: int } = \text{None}, \text{ control_values: Optional[Sequence[Union[int, Collection[int]]]] } = \text{None}, \text{ control_qid_shape: Optional[Tuple[int, ...]] } = \text{None}) \rightarrow \text{cirq.ops.raw_types.Gate}
\]

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- \text{num_controls}: Total number of control qubits.
- \text{control_values}: For which control qubit values to apply the sub gate. A sequence of length \text{num_controls} where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- \text{control_qid_shape}: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to \((2,) \ast \text{num_contacts}\). Specify this argument when using qudits.

cirq.GeneralizedAmplitudeDampingChannel.num_qubits

\[
\text{GeneralizedAmplitudeDampingChannel.num_qubits()} \rightarrow \text{int}
\]

The number of qubits this gate acts on.

cirq.GeneralizedAmplitudeDampingChannel.on

\[
\text{GeneralizedAmplitudeDampingChannel.on}(\ast\text{qubits: cirq.ops.raw_types.Qid} \rightarrow \text{cirq.ops.raw_types.Operation}
\]

Returns an application of this gate to the given qubits.

- \text{Parameters} \ast\text{qubits} – The collection of qubits to potentially apply the gate to.

cirq.GeneralizedAmplitudeDampingChannel.on_each

\[
\text{GeneralizedAmplitudeDampingChannel.on_each}(\ast\text{targets: Union[cirq.ops.raw_types.Qid, Iterable[Any]]}) \rightarrow \text{List[cirq.ops.raw_types.Operation]}
\]

Returns a list of operations applying the gate to all targets.

- \text{Parameters} \ast\text{targets} – The qubits to apply this gate to.
- \text{Returns} Operations applying this gate to the target qubits.
Raises `ValueError` if targets are not instances of `Qid` or `List[Qid]` –

```python
cirq.GeneralizedAmplitudeDampingChannel.validate_args
```

Checks if this gate can be applied to the given qubits.

By default checks that:

- inputs are of type `Qid`
- `len(qubits) == num_qubits()` for all qubits
- `qubit_i.dimension == qid_shape[i]` for all qubits

Child classes can override. The child implementation should call `super().validate_args(qubits)` then do custom checks.

**Parameters** `qubits` – The sequence of qubits to potentially apply the gate to.

**Throws:** `ValueError`: The gate can’t be applied to the qubits.

```python
cirq.GeneralizedAmplitudeDampingChannel.wrap_in_linear_combination
```

```python
GeneralizedAmplitudeDampingChannel.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombination
```

**Attributes**

- `gamma` The probability of the interaction being dissipative.
- `p` The probability of the qubit and environment exchanging energy.

```python
cirq.GeneralizedAmplitudeDampingChannel.gamma
```

**property** `GeneralizedAmplitudeDampingChannel.gamma`

The probability of the interaction being dissipative.

```python
cirq.GeneralizedAmplitudeDampingChannel.p
```

**property** `GeneralizedAmplitudeDampingChannel.p`

The probability of the qubit and environment exchanging energy.
cirq.reset

cirq.reset \(\text{qubit: cirq.ops.raw_types.Qid}\) \(\rightarrow\) cirq.ops.raw_types.Operation

Returns a ResetChannel on the given qubit.

cirq.ResetChannel

class cirq.ResetChannel \((\text{dimension: int} = 2)\)

Reset a qubit back to its \(\ket{0}\) state.

The reset channel is equivalent to performing an unobserved measurement which then controls a bit flip onto the targeted qubit.

__init__ \((\text{dimension: int} = 2)\) \(\rightarrow\) None

The reset channel.

Construct a channel that resets the qubit.

This channel evolves a density matrix as follows:

\[
\rho \rightarrow M_0 \rho M_0^\dagger + M_1 \rho M_1^\dagger
\]

With:

\[
\begin{aligned}
M_0 &= \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \\
M_1 &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}
\end{aligned}
\]

Parameters dimension – Specify this argument when resetting a qudit. There will be \text{dimension} number of dimension by dimension matrices describing the channel, each with a 1 at a different position in the top row.

Methods

controlled([num_controls, control_values, ...])

Returns a controlled version of this gate. If no arguments are

num_qubits() The number of qubits this gate acts on.

on(*qubits) Returns an application of this gate to the given qubits.

on_each(*targets) Returns a list of operations applying the gate to all targets.

validate_args(qubits) Checks if this gate can be applied to the given qubits.

wrap_in_linear_combination([coefficient])
cirq.ResetChannel.controlled

```
ResetChannel.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) \rightarrow cirq.ops.raw_types.Gate
```

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

**num_controls:** Total number of control qubits.

**control_values:** For which control qubit values to apply the sub gate. A sequence of length num_controls where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.

**control_qid_shape:** The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to (2,) * num_controls. Specify this argument when using qudits.

cirq.ResetChannel.num_qubits

```
ResetChannel.num_qubits() \rightarrow int
```

The number of qubits this gate acts on.

cirq.ResetChannel.on

```
ResetChannel.on(*qubits: cirq.ops.raw_types.Qid) \rightarrow cirq.ops.raw_types.Operation
```

Returns an application of this gate to the given qubits.

**Parameters**  
`*qubits` – The collection of qubits to potentially apply the gate to.

cirq.ResetChannel.on_each

```
ResetChannel.on_each(*targets: Union[cirq.ops.raw_types.Qid, Iterable[Any]]) \rightarrow List[cirq.ops.raw_types.Operation]
```

Returns a list of operations applying the gate to all targets.

**Parameters**  
`*targets` – The qubits to apply this gate to.

**Returns**  
Operations applying this gate to the target qubits.

**Raises**  
`ValueError if targets are not instances of Qid or List[Qid]`
cirq.ResetChannel.validate_args

ResetChannel.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None
Checks if this gate can be applied to the given qubits.

By default checks that:
• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call
super().validate_args(qubits) then do custom checks.

Parameters qubits – The sequence of qubits to potentially apply the gate to.

Throws: ValueError: The gate can’t be applied to the qubits.

cirq.ResetChannel.wrap_in_linear_combination

ResetChannel.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates

Attributes

dimension The dimension of the qudit being reset.

property ResetChannel.dimension
The dimension of the qudit being reset.

cirq.phase_damp

cirq.phase_damp(gamma: float) → cirq.ops.common_channels.PhaseDampingChannel

Creates a PhaseDampingChannel with damping constant gamma.

This channel evolves a density matrix via:

\[ \rho \rightarrow M_0 \rho M_0^\dagger + M_1 \rho M_1^\dagger \]

With:

\[ M_0 = \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1 - \gamma} \end{bmatrix} \]
\[ M_1 = \begin{bmatrix} 0 & 0 \\ 0 & \sqrt{\gamma} \end{bmatrix} \]

Parameters gamma – The damping constant.
cirq.phase_flip

cirq.phase_flip(p: Optional[float] = None) \rightarrow Union[cirq.ops.common_gates.ZPowGate, cirq.ops.common_channels.PhaseFlipChannel]

Returns a PhaseFlipChannel that flips a qubit’s phase with probability p
if p is None, return a guaranteed phase flip in the form of a Z operation.
This channel evolves a density matrix via:

$$ \rho \rightarrow M_0 \rho M_0^\dagger + M_1 \rho M_1^\dagger $$

With:

$$ \begin{aligned} M_0 &= \sqrt{p} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\
M_1 &= \sqrt{1-p} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \end{aligned} $$

Parameters

- **p** – the probability of a phase flip.

Raises `ValueError` – if p is not a valid probability.

cirq.PhaseDampingChannel

class cirq.PhaseDampingChannel(gamma: float)

Dampen qubit phase.

This channel models phase damping which is the loss of quantum information without the loss of energy.

__init__ (gamma: float) \rightarrow None

The phase damping channel.

Construct a channel that enacts a phase damping constant gamma.

This channel evolves a density matrix via:

$$ \rho \rightarrow M_0 \rho M_0^\dagger + M_1 \rho M_1^\dagger $$

With:

$$ \begin{aligned} M_0 &= \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1 - \gamma} \end{bmatrix} \\
M_1 &= \begin{bmatrix} 0 & 0 \\ 0 & \sqrt{\gamma} \end{bmatrix} \end{aligned} $$

Parameters

- **gamma** – The damping constant.

Raises `ValueError` – if gamma is not a valid probability.
Methods

```python
circled([num_controls, control_values, ...])
```
Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

```python
num_qubits()
```
The number of qubits this gate acts on.

```python
on(*qubits)
```
Returns an application of this gate to the given qubits.

```python
on_each(*targets)
```
Returns a list of operations applying the gate to all targets.

```python
validate_args(qubits)
```
Checks if this gate can be applied to the given qubits.

```python
wrap_in_linear_combination([coefficient])
```

**circled.PhaseDampingChannel.controlled**

```python
PhaseDampingChannel.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate
```
Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- **num_controls**: Total number of control qubits.
- **control_values**: For which control qubit values to apply the sub gate. A sequence of length `num_controls` where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- **control_qid_shape**: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to `(2,) * num_controls`. Specify this argument when using qudits.

**circled.PhaseDampingChannel.num_qubits**

```python
PhaseDampingChannel.num_qubits() → int
```
The number of qubits this gate acts on.

**circled.PhaseDampingChannel.on**

```python
PhaseDampingChannel.on(*qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation
```
Returns an application of this gate to the given qubits.

- **Parameters** `*qubits` – The collection of qubits to potentially apply the gate to.
cirq.PhaseDampingChannel.on_each

PhaseDampingChannel.on_each(*targets: Union[cirq.ops.raw_types.Qid, Iterable[Any]]) \rightarrow List[cirq.ops.raw_types.Operation]

Returns a list of operations applying the gate to all targets.

- **Parameters** `*targets` – The qubits to apply this gate to.
- **Returns** Operations applying this gate to the target qubits.
- **Raises** `ValueError` if targets are not instances of `Qid` or `List[Qid]` –

cirq.PhaseDampingChannel.validate_args

PhaseDampingChannel.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) \rightarrow None

Checks if this gate can be applied to the given qubits.

By default checks that:

- inputs are of type `Qid`
- `len(qubits) == num_qubits()`
- `qubit_i.dimension == qid_shape[i]` for all qubits

Child classes can override. The child implementation should call `super().validate_args(qubits)` then do custom checks.

- **Parameters** `qubits` – The sequence of qubits to potentially apply the gate to.
- **Throws**: `ValueError`: The gate can’t be applied to the qubits.

cirq.PhaseDampingChannel.wrap_in_linear_combination

PhaseDampingChannel.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) \rightarrow linear_combinations.LinearCombinationOfGates

Attributes

- **gamma** The damping constant.


cirq.PhaseDampingChannel.gamma

- **property** `PhaseDampingChannel.gamma` The damping constant.
cirq.PhaseFlipChannel

**class cirq.PhaseFlipChannel (p: float)**

Proabilistically flip the sign of the phase of a qubit.

```python
__init__(p: float) -> None
```

The phase flip channel.

Construct a channel to flip the phase with probability p.

This channel evolves a density matrix via:

$$ \rho \rightarrow M_0 \rho M_0^\dagger + M_1 \rho M_1^\dagger $$

With:

$$
\begin{aligned}
M_0 &= \sqrt{1 - p} \begin{bmatrix} 1 & 0 \\
0 & 1 \end{bmatrix} \\
M_1 &= \sqrt{p} \begin{bmatrix} 1 & 0 \\
0 & -1 \end{bmatrix}
\end{aligned}
$$

**Parameters**

- **p** – the probability of a phase flip.

**Raises**

- **ValueError** – if p is not a valid probability.

**Methods**

- **controlled([num_controls, control_values, ...])**
  Returns a controlled version of this gate. If no arguments are
  specified, defaults to a single qubit control.

- **num_qubits()**
  The number of qubits this gate acts on.

- **on(*qubits)**
  Returns an application of this gate to the given qubits.

- **on_each(*targets)**
  Returns a list of operations applying the gate to all targets.

- **validate_args(qubits)**
  Checks if this gate can be applied to the given qubits.

- **wrap_in_linear_combination([coefficient])**

**cirq.PhaseFlipChannel.controlled**

```python
PhaseFlipChannel.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) -> cirq.ops.raw_types.Gate
```

Returns a controlled version of this gate. If no arguments are
specified, defaults to a single qubit control.

- **num_controls**: Total number of control qubits.
- **control_values**: For which control qubit values to apply the sub
gate. A sequence of length num_controls where each
  entry is an integer (or set of integers) corresponding to the
qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.

control_qid_shape: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to (2,) * num_controls. Specify this argument when using qudits.

cirq.PhaseFlipChannel.num_qubits

PhaseFlipChannel.num_qubits() \rightarrow \text{int}

The number of qubits this gate acts on.

cirq.PhaseFlipChannel.on

PhaseFlipChannel.on(*qubits: cirq.ops.raw_types.Qid) \rightarrow \text{cirq.ops.raw_types.Operation}

Returns an application of this gate to the given qubits.

Parameters *qubits – The collection of qubits to potentially apply the gate to.

cirq.PhaseFlipChannel.on_each

PhaseFlipChannel.on_each(*targets: Union[cirq.ops.raw_types.Qid, Iterable[\text{Any}]]) \rightarrow \text{List[cirq.ops.raw_types.Operation]}

Returns a list of operations applying the gate to all targets.

Parameters *targets – The qubits to apply this gate to.

Returns Operations applying this gate to the target qubits.

Raises ValueError if targets are not instances of Qid or List[Qid] –

cirq.PhaseFlipChannel.validate_args

PhaseFlipChannel.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) \rightarrow \text{None}

Checks if this gate can be applied to the given qubits.

By default checks that:

• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call super().validate_args(qubits) then do custom checks.

Parameters qubits – The sequence of qubits to potentially apply the gate to.

Throws: ValueError: The gate can’t be applied to the qubits.
Cirq Documentation, Release 0.6.1

cirq.PhaseFlipChannel.wrap_in_linear_combination

PhaseFlipChannel.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) \rightarrow \text{linear_combinations.LinearCombinationOfGates}

Attributes

\[ p \]

The probability of a phase flip.

cirq.PhaseFlipChannel.p

@property PhaseFlipChannel.p

The probability of a phase flip.

3.1.8 Other Gate and Operation Classes

Generic classes for creating new kinds of gates and operations.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cirq.ControlledGate(sub_gate[,num_controls,...])</td>
<td>Augments existing gates with a control qubit.</td>
</tr>
<tr>
<td>cirq.ControlledOperation(controls, sub_operation)</td>
<td></td>
</tr>
<tr>
<td>cirq.EigenGate(*[,exponent,global_shift])</td>
<td>A gate with a known eigendecomposition.</td>
</tr>
<tr>
<td>cirq.Gate</td>
<td>An operation type that can be applied to a collection of qubits.</td>
</tr>
<tr>
<td>cirq.GateOperation(gate,qubits)</td>
<td>An application of a gate to a sequence of qubits.</td>
</tr>
<tr>
<td>cirq.InterchangeableQubitsGate</td>
<td>Indicates operations should be equal under some qubit permutations.</td>
</tr>
<tr>
<td>cirq.LinearCombinationOfGates(terms)</td>
<td>Represents linear operator defined by a linear combination of gates.</td>
</tr>
<tr>
<td>cirq.Operation</td>
<td>An effect applied to a collection of qubits.</td>
</tr>
<tr>
<td>cirq.SingleQubitGate</td>
<td>A gate that must be applied to exactly one qubit.</td>
</tr>
<tr>
<td>cirq.ThreeQubitGate</td>
<td>A gate that must be applied to exactly three qubits.</td>
</tr>
<tr>
<td>cirq.TwoQubitGate</td>
<td>A gate that must be applied to exactly two qubits.</td>
</tr>
</tbody>
</table>

cirq.ControlledGate

class cirq.ControlledGate(sub_gate: cirq.ops.raw_types.Gate, num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Sequence[int]] = None)

Augments existing gates with a control qubit.

__init__(sub_gate: cirq.ops.raw_types.Gate, num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Sequence[int]] = None) \rightarrow \text{None}

Initializes the controlled gate. If no arguments are specified for the controls, defaults to a single qubit control.

Parameters
• **sub_gate** – The gate to add a control qubit to.

• **num_controls** – Total number of control qubits.

• **control_values** – For which control qubit values to apply the sub gate. A sequence of length `num_controls` where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.

• **control_qid_shape** – The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to `(2,) * num_controls`. Specify this argument when using qudits.

### Methods

- `controlled([num_controls, control_values, ...])`  
  Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- `num_controls()`  
  The number of qubits this gate acts on.

- `on(*qubits)`  
  Returns an application of this gate to the given qubits.

- `validate_args(qubits)`  
  Checks if this gate can be applied to the given qubits.

- `wrap_in_linear_combination([coefficient])`  
  Cirq.ControlledGate.controlled

```python
ControlledGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate
```

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- `num_controls`:
  - Total number of control qubits.
  - control_values: For which control qubit values to apply the sub gate. A sequence of length `num_controls` where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
  - control_qid_shape: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to `(2,) * num_controls`. Specify this argument when using qudits.

- `cirq.ControlledGate.num_controls`

```python
ControlledGate.num_controls() → int
```
cirq.ControlledGate.num_qubits

ControlledGate.num_qubits() → int
The number of qubits this gate acts on.

cirq.ControlledGate.on

ControlledGate.on(*qubits: cirq.ops.raw_types.Qid) → cirq.ops.controlled_operation.ControlledOperation
Returns an application of this gate to the given qubits.

Parameters
*qubits – The collection of qubits to potentially apply the gate to.

cirq.ControlledGate.validate_args

ControlledGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None
Checks if this gate can be applied to the given qubits.

By default checks that:
• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call
super().validate_args(qubits) then do custom checks.

Parameters
qubits – The sequence of qubits to potentially apply the gate to.

Throws: ValueError: The gate can’t be applied to the qubits.

cirq.ControlledGate.wrap_in_linear_combination

ControlledGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates

cirq.ControlledOperation

class cirq.ControlledOperation(controls: Sequence[cirq.ops.raw_types.Qid], sub_operation: cirq.ops.raw_types.Operation, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None)

__init__(controls: Sequence[cirq.ops.raw_types.Qid], sub_operation: cirq.ops.raw_types.Operation, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None)
Initialize self. See help(type(self)) for accurate signature.

Methods
controlled_by(*control_qubits[, control_values]) Returns a controlled version of this operation. If no control_qubits are specified, returns self.

Parameters
• control_qubits – Qubits to control the operation by. Required.
• control_values – For which control qubit values to apply the operation. A sequence of the same length as control_qubits where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the operation is applied. If unspecified, control values default to 1.

cirq.ControlledOperation.transform_qubits

transform_qubits(func) Returns the same operation, but with different qubits.

Parameters func – The function to use to turn each current qubit into a desired new qubit.

Returns
The receiving operation but with qubits transformed by the given function.

cirq.ControlledOperation.validate_args

validate_args(qubits) Raises an exception if the qubits don’t match this operation’s qid

Parameters qubits – The new qids for the operation.
**Raises ValueError** – The operation had qids that don’t match it’s qid shape.

```python
cirq.ControlledOperation.with_qubits
```

ControlledOperation.with_qubits(*new_qubits)

**Attributes**

```python
cirq.EigenGate
```

cirq.EigenGate(*, exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0)

A gate with a known eigendecomposition.

EigenGate is particularly useful when one wishes for different parts of the same eigenspace to be extrapolated differently. For example, if a gate has a 2-dimensional eigenspace with eigenvalue -1, but one wishes for the square root of the gate to split this eigenspace into a part with eigenvalue i and a part with eigenvalue -i, then EigenGate allows this functionality to be unambiguously specified via the _eigen_components method.

```python
__init__(* exponent: Union[float, sympy.core.basic.Basic] = 1.0, global_shift: float = 0.0) → None
```

Initializes the parameters used to compute the gate’s matrix.

The eigenvalue of each eigenspace of a gate is computed by

1. Starting with an angle in half turns as returned by the gate’s _eigen_components method:

```python
+ s
```

2. Shifting the angle by global_shift:

3. Scaling the angle by exponent:
4. Converting from half turns to a complex number on the unit circle:

\[ \exp(i \times \pi \times (s) \times e) \]

**Parameters**

- **exponent** – The \( t \) in gate**\( t \). Determines how much the eigenvalues of the gate are scaled by. For example, eigenvectors phased by -1 when gate**1 is applied will gain a relative phase of \( e^{i \pi \text{exponent}} \) when gate**\( t \) is applied (relative to eigenvectors unaffected by gate**1).

- **global_shift** – Offsets the eigenvalues of the gate at exponent=1. In effect, this controls a global phase factor on the gate’s unitary matrix. The factor is:

\[ \exp(i \times \pi \times \text{global_shift} \times \text{exponent}) \]

For example, cirq.X**\( t \) uses a global_shift of 0 but cirq.Rx(\( t \)) uses a global_shift of -0.5, which is why cirq.unitary(cirq.Rx(\( \pi \))) equals -iX instead of X.

**Methods**

```python
EigenGate.controlled([num_controls, control_values, ...])
```

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- **num_controls**: Total number of control qubits.
- **control_values**: For which control qubit values to apply the sub gate. A sequence of length **num_controls** where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- **control_qid_shape**: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to None.

**Examples**

```python
circuit = cirq.Circuit(cirq.measure(cirq.NamedQubit('q'), key='m'))
circuit.append(cirq.X(cirq.NamedQubit('q')).controlled_by(cirq.NamedQubit('a'), cirq.NamedQubit('b')))```
(2, num_controls). Specify this argument when using qudits.

**cirq.EigenGate.num_qubits**

```python
EigenGate.num_qubits() -> int
```

The number of qubits this gate acts on.

**cirq.EigenGate.on**

```python
EigenGate.on(*qubits: cirq.ops.raw_types.Qid) -> cirq.ops.raw_types.Operation
```

Returns an application of this gate to the given qubits.

**Parameters**

*qubits* – The collection of qubits to potentially apply the gate to.

**cirq.EigenGate.validate_args**

```python
EigenGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) -> None
```

Checks if this gate can be applied to the given qubits.

By default checks that:

* inputs are of type Qid
* len(qubits) == num_qubits()
* qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call
`super().validate_args(qubits)` then do custom checks.

**Parameters**

qubits – The sequence of qubits to potentially apply the gate to.

**Throws:**

`ValueError`: The gate can’t be applied to the qubits.

**cirq.EigenGate.wrap_in_linear_combination**

```python
EigenGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) -> linear_combinations.LinearCombinationOfGates
```

**Attributes**

**exponent**

**global_shift**

**cirq.EigenGate.exponent**

```python
property EigenGate.exponent
```
**cirq.EigenGate.global_shift**

**property** EigenGate.global_shift

**cirq.Gate**

**class cirq.Gate**

An operation type that can be applied to a collection of qubits.

Gates can be applied to qubits by calling their `on()` method with the qubits to be applied to supplied, or, alternatively, by simply calling the gate on the qubits. In other words calling `MyGate.on(q1, q2)` to create an Operation on q1 and q2 is equivalent to `MyGate(q1,q2)`.

Gates operate on a certain number of qubits. All implementations of gate must implement the `num_qubits` method declaring how many qubits they act on. The gate feature classes `SingleQubitGate` and `TwoQubitGate` can be used to avoid writing this boilerplate.

Linear combinations of gates can be created by adding gates together and multiplying them by scalars.

```python
__init__()
```

Initialize self. See help(type(self)) for accurate signature.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cirq.Gate.controlled([num_controls, control_values, ...])</code></td>
<td>Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.</td>
</tr>
<tr>
<td><code>num_qubits()</code></td>
<td>The number of qubits this gate acts on.</td>
</tr>
<tr>
<td><code>on(*qubits)</code></td>
<td>Returns an application of this gate to the given qubits.</td>
</tr>
<tr>
<td><code>validate_args(qubits)</code></td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
<tr>
<td><code>wrap_in_linear_combination([coefficient])</code></td>
<td></td>
</tr>
</tbody>
</table>
num_controls: Total number of control qubits.
control_values: For which control qubit values to apply the sub
gate. A sequence of length num_controls where each
entry is an integer (or set of integers) corresponding to the
qubit value (or set of possible values) where that control is
enabled. When all controls are enabled, the sub gate is
applied. If unspecified, control values default to 1.
control_qid_shape: The qid shape of the controls. A tuple of the
expected dimension of each control qid. Defaults to
(2,) * num_controls. Specify this argument when using qudits.

cirq.Gate.num_qubits

gate.num_qubits() → int
The number of qubits this gate acts on.

cirq.Gate.on

gate.on(*qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation
Returns an application of this gate to the given qubits.

Parameters *qubits – The collection of qubits to potentially apply the gate to.

cirq.Gate.validate_args

gate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None
Checks if this gate can be applied to the given qubits.

By default checks that:
• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call
super().validate_args(qubits) then do custom checks.

Parameters qubits – The sequence of qubits to potentially apply the gate to.

Throws: ValueError: The gate can’t be applied to the qubits.

cirq.Gate.wrap_in_linear_combination

gate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → lin-
ear_combinations.LinearCombinationOfGates
class cirq.GateOperation(gate: cirq.ops.raw_types.Gate, qubits: Sequence[cirq.ops.raw_types.Qid])

An application of a gate to a sequence of qubits.

__init__(gate: cirq.ops.raw_types.Gate, qubits: Sequence[cirq.ops.raw_types.Qid]) → None

Parameters

• gate – The gate to apply.
• qubits – The qubits to operate on.

Methods

controlled_by(*control_qubits[, control_values])

Returns a controlled version of this operation. If no control_qubits are specified, returns self.

Parameters

• control_qubits – Qubits to control the operation by. Required.
• control_values – For which control qubit values to apply the operation. A sequence of the same length as control_qubits where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the operation is applied. If unspecified, control values default to 1.

cirq.GateOperation.transform_qubits

GateOperation.transform_qubits(func: Callable[cirq.ops.raw_types.Qid, cirq.ops.raw_types.Qid]) → cirq.ops.raw_types.Operation

Returns the same operation, but with different qubits.

Parameters func – The function to use to turn each current qubit into a desired new qubit.
Returns

The receiving operation but with qubits transformed by the given function.

cirq.GateOperation.validate_args

GateOperation.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid])

Raises an exception if the qubits don’t match this operation’s qid shape.
Call this method from a subclass’s with_qubits method.

Parameters qubits – The new qids for the operation.

Raises ValueError – The operation had qids that don’t match it’s qid shape.

cirq.GateOperation.with_gate

GateOperation.with_gate(new_gate: cirq.ops.raw_types.Gate) → cirq.ops.raw_types.Operation

cirq.GateOperation.with_qubits

GateOperation.with_qubits(*new_qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation

Attributes

<table>
<thead>
<tr>
<th>gate</th>
<th>The gate applied by the operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>qubits</td>
<td>The qubits targeted by the operation.</td>
</tr>
</tbody>
</table>

cirq.GateOperation.gate

property GateOperation.gate

The gate applied by the operation.

cirq.GateOperation.qubits

property GateOperation.qubits

The qubits targeted by the operation.

cirq.InterchangeableQubitsGate

class cirq.InterchangeableQubitsGate

Indicates operations should be equal under some qubit permutations.

__init__()

Initialize self. See help(type(self)) for accurate signature.
Methods

```python
qubit_index_to_equivalence_group_key(index)
```

Returns a key that differs between non-interchangeable qubits.

```python
cirq.InterchangeableQubitsGate.qubit_index_to_equivalence_group_key(index: int) → int
```

Returns a key that differs between non-interchangeable qubits.

```python
cirq.LinearCombinationOfGates
```

class cirq.LinearCombinationOfGates(terms: Mapping[cirq.ops.raw_types.Gate, Union[complex, float]])

Represents linear operator defined by a linear combination of gates.

Suppose \(G_1, G_2, \ldots, G_n\) are gates and \(b_1, b_2, \ldots, b_n\) are complex numbers. Then

```
LinearCombinationOfGates({G1: b1, G2: b2, \ldots, Gn: bn})
```

represents the linear operator

\[
A = b_1 G_1 + b_2 G_2 + \ldots + b_n G_n
\]

Note that \(A\) may not be unitary or even normal.

Rather than creating `LinearCombinationOfGates` instance explicitly, one may use overloaded arithmetic operators. For example,

```
cirq.LinearCombinationOfGates({cirq.X: 2, cirq.Z: -2})
```

is equivalent to

```
2 * cirq.X - 2 * cirq.Z
```

```python
__init__(terms: Mapping[cirq.ops.raw_types.Gate, Union[complex, float]]) → None
```

Initializes linear combination from a collection of terms.

- **Parameters** terms – Mapping of gates to coefficients in the linear combination being initialized.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clean(*[, atol])</code></td>
<td>Remove terms with coefficients of absolute value atol or less.</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>None. Remove all items from D.</td>
</tr>
<tr>
<td><code>copy()</code></td>
<td>TSelf</td>
</tr>
<tr>
<td><code>fromkeys(vectors[, coefficient])</code></td>
<td>Returns number of qubits in the domain if known, None if unknown.</td>
</tr>
<tr>
<td><code>get(k[,d])</code></td>
<td>If key is not found, d is returned if given, otherwise KeyError is raised.</td>
</tr>
<tr>
<td><code>keys()</code></td>
<td>as a 2-tuple; but raise KeyError if D is empty.</td>
</tr>
<tr>
<td><code>matrix()</code></td>
<td>Reconstructs matrix of self using unitaries of underlying gates.</td>
</tr>
<tr>
<td><code>num_qubits()</code></td>
<td>If E present and has a .keys() method, does: for k in E: D[k] = E[k]</td>
</tr>
<tr>
<td><code>pop(k[,d])</code></td>
<td>If key is not found, d is returned if given, otherwise KeyError is raised.</td>
</tr>
<tr>
<td><code>popitem()</code></td>
<td>Returns number of qubits in the domain if known, None if unknown.</td>
</tr>
<tr>
<td><code>setdefault(k[,d])</code></td>
<td>key is not found, d is returned if given, otherwise KeyError is raised.</td>
</tr>
<tr>
<td><code>update([E, ]**F)</code></td>
<td>as a 2-tuple; but raise KeyError if D is empty.</td>
</tr>
<tr>
<td><code>values()</code></td>
<td>a set-like object providing a view on D’s items</td>
</tr>
</tbody>
</table>

**cirq.LinearCombinationOfGates.clean**

LinearCombinationOfGates.clean (*, atol: float = 1e-09) → TSelf

Remove terms with coefficients of absolute value atol or less.

**cirq.LinearCombinationOfGates.clear**

LinearCombinationOfGates.clear () → None. Remove all items from D.

**cirq.LinearCombinationOfGates.copy**

LinearCombinationOfGates.copy () → TSelf

**cirq.LinearCombinationOfGates.fromkeys**

classmethod LinearCombinationOfGates.fromkeys(vectors, coefficient=0)

**cirq.LinearCombinationOfGates.get**

LinearCombinationOfGates.get (k[, d]) → D[k] if k in D, else d. d defaults to None.

**cirq.LinearCombinationOfGates.items**

LinearCombinationOfGates.items () → a set-like object providing a view on D’s items


cirq.LinearCombinationOfGates.keys

LinearCombinationOfGates.keys() → a set-like object providing a view on D’s keys

cirq.LinearCombinationOfGates.matrix

LinearCombinationOfGates.matrix() → numpy.ndarray
Reconstructs matrix of self using unitaries of underlying gates.

    Raises TypeError – if any of the gates in self does not provide a unitary.

cirq.LinearCombinationOfGates.num_qubits

LinearCombinationOfGates.num_qubits() → Optional[int]
Returns number of qubits in the domain if known, None if unknown.

cirq.LinearCombinationOfGates.pop

LinearCombinationOfGates.pop(k[, d]) → v, remove specified key and return the correspond-
ing value.
If key is not found, d is returned if given, otherwise KeyError is raised.

cirq.LinearCombinationOfGates.popitem

LinearCombinationOfGates.popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

cirq.LinearCombinationOfGates.setdefault

LinearCombinationOfGates.setdefault(k[, d]) → D.get(k,d), also set D[k]=d if k not in D

cirq.LinearCombinationOfGates.update

LinearCombinationOfGates.update(E, **F) → None. Update D from mapping/iterable E
and F.

    If E present and has a .keys() method, does: for k in E: D[k] = E[k]
    If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v
    In either case, this is followed by: for k, v in F.items(): D[k] = v


cirq.LinearCombinationOfGates.values

LinearCombinationOfGates.values() → an object providing a view on D’s values

Attributes

3.1. API Reference 171
TSelf

```
cirq.LinearCombinationOfGates.TSelf
```

```
LinearCombinationOfGates.TSelf = ~TSelf
```

cirq.Operation

class cirq.Operation

An effect applied to a collection of qubits.

The most common kind of Operation is a GateOperation, which separates its effect into a qubit-independent Gate and the qubits it should be applied to.

```
__init__()
```

Initialize self. See help(type(self)) for accurate signature.

Methods

```
controlled_by(*control_qubits[, control_values])
```

Returns a controlled version of this operation. If no control_qubits are specified, returns self.

**Parameters**

- **control_qubits** – Qubits to control the operation by. Required.
- **control_values** – For which control qubit values to apply the operation. A sequence of the same length as control_qubits where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the operation is applied. If unspecified, control values default to 1.

```
transform_qubits(func)
```

Returns the same operation, but with different qubits.

```
validate_args(qubits)
```

Raises an exception if the qubits don’t match this operation’s qid.

```
with_qubits(*new_qubits)
```

cirq.Operation.controlled_by

```
Operation.controlled_by(*control_qubits: cirq.ops.raw_types.Qid, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None) → cirq.ops.raw_types.Operation
```

Returns a controlled version of this operation. If no control_qubits are specified, returns self.
cirq.Operation.transform_qubits

Operation.transform_qubits(func: Callable[cirq.ops.raw_types.Qid, cirq.ops.raw_types.Qid]) → cirq.ops.raw_types.Operation

Returns the same operation, but with different qubits.

Parameters

func – The function to use to turn each current qubit into a desired new qubit.

Returns

The receiving operation but with qubits transformed by the given function.

cirq.Operation.validate_args

Operation.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid])

 Raises an exception if the qubits don’t match this operation’s qid shape.

Call this method from a subclass’s with_qubits method.

Parameters

qubits – The new qids for the operation.

Raises

ValueError – The operation had qids that don’t match it’s qid shape.

cirq.Operation.with_qubits

abstract Operation.with_qubits(*new_qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation

Attributes


gate
qubits

cirq.Operation.gate

property Operation.gate

cirq.Operation.qubits

abstract property Operation.qubits

cirq.SingleQubitGate

class cirq.SingleQubitGate

A gate that must be applied to exactly one qubit.

__init__()  
Initialize self. See help(type(self)) for accurate signature.
## Methods

### controlled([num_controls, control_values, ...])

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- **num_controls**: Total number of control qubits.
- **control_values**: For which control qubit values to apply the sub gate. A sequence of length num_controls where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- **control_qid_shape**: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to (2,) * num_controls. Specify this argument when using qudits.

### num_qubits()

The number of qubits this gate acts on.

### on(*qubits)

Returns an application of this gate to the given qubits.

### on_each(*targets)

Returns a list of operations applying the gate to all targets.

### validate_args(*qubits)

Checks if this gate can be applied to the given qubits.

### wrap_in_linear_combination([coefficient])

### cirq.SingleQubitGate.controlled

**SingleQubitGate.controlled**(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

### cirq.SingleQubitGate.num_qubits

**SingleQubitGate.num_qubits**() → int

The number of qubits this gate acts on.

### cirq.SingleQubitGate.on

**SingleQubitGate.on**(qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation

Returns an application of this gate to the given qubits.

**Parameters**

- **qubits**: The collection of qubits to potentially apply the gate to.
cirq.SingleQubitGate.on_each

SingleQubitGate.on_each(*targets: Union[cirq.ops.raw_types.Qid, Iterable[Any]]) → List[cirq.ops.raw_types.Operation]

Returns a list of operations applying the gate to all targets.

Parameters *targets – The qubits to apply this gate to.

Returns Operations applying this gate to the target qubits.

Raises ValueError if targets are not instances of Qid or List[Qid] –

cirq.SingleQubitGate.validate_args

SingleQubitGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None

Checks if this gate can be applied to the given qubits.

By default checks that:

• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit.i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call super().validate_args(qubits) then do custom checks.

Parameters qubits – The sequence of qubits to potentially apply the gate to.

Throws: ValueError: The gate can’t be applied to the qubits.

cirq.SingleQubitGate.wrap_in_linear_combination

SingleQubitGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates

cirq.ThreeQubitGate

class cirq.ThreeQubitGate

A gate that must be applied to exactly three qubits.

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods
controlled([num_controls, control_values, ...])

Returns a controlled version of this gate. If no arguments are
specified, defaults to a single qubit control.

num_qubits()

The number of qubits this gate acts on.

on(*qubits)

Returns an application of this gate to the given qubits.

validate_args(qubits)

Checks if this gate can be applied to the given qubits.

wrap_in_linear_combination([coefficient])

cirq.ThreeQubitGate.controlled

ThreeQubitGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate

Returns a controlled version of this gate. If no arguments are
specified, defaults to a single qubit control.

num_controls: Total number of control qubits.
control_values: For which control qubit values to apply the sub
gate. A sequence of length num_controls where each
entry is an integer (or set of integers) corresponding to the
qubit value (or set of possible values) where that control is
enabled. When all controls are enabled, the sub gate is
applied. If unspecified, control values default to 1.
control_qid_shape: The qid shape of the controls. A tuple of the
expected dimension of each control qid. Defaults to
(2,) * num_controls. Specify this argument when using qudits.

cirq.ThreeQubitGate.num_qubits

ThreeQubitGate.num_qubits() → int

The number of qubits this gate acts on.

cirq.ThreeQubitGate.on

ThreeQubitGate.on(*qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation

Returns an application of this gate to the given qubits.

Parameters  *qubits – The collection of qubits to potentially apply the gate to.

cirq.ThreeQubitGate.validate_args

ThreeQubitGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None

Checks if this gate can be applied to the given qubits.
By default checks that:

- inputs are of type \texttt{Qid}
- \texttt{len(qubits)} == \texttt{num_qubits()}
- \texttt{qubit_i.dimension} == \texttt{qid_shape[i]} for all qubits

Child classes can override. The child implementation should call \texttt{super().validate_args(qubits)} then do custom checks.

**Parameters** \texttt{qubits} – The sequence of qubits to potentially apply the gate to.

**Throws:** \texttt{ValueError}: The gate can’t be applied to the qubits.

cirq.ThreeQubitGate.wrap_in_linear_combination

\texttt{ThreeQubitGate}.\texttt{wrap_in_linear_combination}(\texttt{coefficient: Union[complex, float, int] = 1}) \rightarrow \texttt{linear_combinations.LinearCombinationOfGates}

cirq.TwoQubitGate

**class** cirq.TwoQubitGate

A gate that must be applied to exactly two qubits.

\texttt{__init__}()

Initialize self. See help(type(self)) for accurate signature.

**Methods**

\texttt{controlled}([\texttt{num_controls, control_values, ...}])

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

\texttt{num_qubits}()

The number of qubits this gate acts on.

\texttt{on}(\ast qubits)

Returns an application of this gate to the given qubits.

\texttt{validate_args}(qubits)

Checks if this gate can be applied to the given qubits.

\texttt{wrap_in_linear_combination}([\texttt{coefficient}])

cirq.TwoQubitGate.controlled

\texttt{TwoQubitGate}.\texttt{controlled}([\texttt{num_controls}: int = \texttt{None}, \texttt{control_values}: Optional[Sequence[Union[int, Collection[int]]]] = \texttt{None}, \texttt{control_qid_shape}: Optional[Tuple[int, ...]] = \texttt{None}) \rightarrow \texttt{cirq.ops.raw_types.Gate}

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.
num_controls: Total number of control qubits.
control_values: For which control qubit values to apply the sub
gate. A sequence of length num_controls where each
entry is an integer (or set of integers) corresponding to the
qubit value (or set of possible values) where that control is
enabled. When all controls are enabled, the sub gate is
applied. If unspecified, control values default to 1.
control_qid_shape: The qid shape of the controls. A tuple of the
expected dimension of each control qid. Defaults to
(2,) * num_controls. Specify this argument when using qudits.

cirq.TwoQubitGate.num_qubits

TwoQubitGate.num_qubits() → int
The number of qubits this gate acts on.

cirq.TwoQubitGate.on

TwoQubitGate.on(*qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation
Returns an application of this gate to the given qubits.

Parameters *qubits – The collection of qubits to potentially apply the gate to.

cirq.TwoQubitGate.validate_args

TwoQubitGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None
Checks if this gate can be applied to the given qubits.

By default checks that:
• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call
super().validate_args(qubits) then do custom checks.

Parameters qubits – The sequence of qubits to potentially apply the gate to.

Throws: ValueError: The gate can’t be applied to the qubits.

cirq.TwoQubitGate.wrap_in_linear_combination

TwoQubitGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates
3.1.9 Pauli and Clifford Group Concepts

Pauli(index, name) Represents the Pauli gates.

PauliInteractionGate(pauli0, invert0, ...[, ...])

PauliString(*contents[, qubit_pauli_map, ...])

PauliTransform(to, flip)

SingleQubitCliffordGate(*, _rotation_map, Any single qubit Clifford rotation.

...)

cirq.Pauli

class cirq.Pauli(index: int, name: str)
    Represents the Pauli gates.

    This is an abstract class with no public subclasses. The only instances
    of private subclasses are the X, Y, or Z Pauli gates defined below.

    __init__(index: int, name: str) → None
        Initialize self. See help(type(self)) for accurate signature.

    Methods

    by_index(index)
    by_relative_index(p, relative_index)
    commutes_with(other)
    controlled([num_controls, control_values, ...])
        Returns a controlled version of this gate. If no
        arguments are

    num_qubits() The number of qubits this gate acts on.
    or(*qubits) Returns an application of this gate to the given
                qubits.
    phased_pauli_product(other)
    relative_index(second) Relative index of self w.r.t.
    third(second)
    validate_args(qubits) Checks if this gate can be applied to the given qubits.
    wrap_in_linear_combination([coefficient])

cirq.Pauli.by_index

    static Pauli.by_index(index: int) → cirq.ops.pauli_gates.Pauli

cirq.Pauli.by_relative_index

    static Pauli.by_relative_index(p: cirq.ops.pauli_gates.Pauli, relative_index: int) →
        cirq.ops.pauli_gates.Pauli
cirq.Pauli.commutes_with

Pauli.commutes_with(\text{other}: \text{cirq.ops.pauli.gates.Pauli}) \rightarrow \text{bool}

cirq.Pauli.controlled

Pauli.controlled(\text{num_controls}: \text{int} = \text{None}, \text{control_values}: \text{Optional[Sequence[Union[int, Collection[int]]]}} = \text{None}, \text{control_qid_shape}: \text{Optional[Tuple[int, ...]]} = \text{None}) \rightarrow \text{cirq.ops.raw_types.Gate}

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

\text{num\_controls}: \text{Total number of control qubits.}
\text{control\_values}: \text{For which control qubit values to apply the sub gate. A sequence of length \text{num\_controls} where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.}
\text{control\_qid\_shape}: \text{The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to (2,)^{\text{num\_controls}}. Specify this argument when using qudits.}

cirq.Pauli.num_qubits

Pauli.num_qubits()

The number of qubits this gate acts on.

cirq.Pauli.on

Pauli.on(\text{*qubits}: \text{cirq.ops.raw_types.Qid}) \rightarrow \text{SingleQubitPauliStringGateOperation}

Returns an application of this gate to the given qubits.

\text{Parameters} \text{*qubits} – The collection of qubits to potentially apply the gate to.

cirq.Pauli.phased_pauli_product

Pauli.phased_pauli_product(\text{other}: \text{Union[cirq.Pauli, identity.IdentityGate]}) \rightarrow \text{Tuple[complex, Union[cirq.Pauli, identity.IdentityGate]]}

cirq.Pauli.relative_index

Pauli.relative_index(\text{second}: \text{cirq.ops.pauli.gates.Pauli}) \rightarrow \text{int}

Relative index of self w.r.t. second in the (X, Y, Z) cycle.
**cirq.Pauli.third**

Pauli.third(second: cirq.ops.pauli_gates.Pauli) → cirq.ops.pauli_gates.Pauli

**cirq.Pauli.validate_args**

Pauli.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None

Checks if this gate can be applied to the given qubits.

By default checks that:

- inputs are of type Qid
- len(qubits) == num_qubits()
- qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call super().validate_args(qubits) then do custom checks.

**Parameters**

- **qubits** – The sequence of qubits to potentially apply the gate to.

**Throws:** ValueError: The gate can’t be applied to the qubits.

**cirq.Pauli.wrap_in_linear_combination**

Pauli.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates

**cirq.PauliInteractionGate**

class cirq.PauliInteractionGate(pauli0: cirq.ops.pauli_gates.Pauli, invert0: bool, pauli1: cirq.ops.pauli_gates.Pauli, invert1: bool, *, exponent: Union[float, sympy.core.basic.Basic] = 1.0)

__init__(pauli0: cirq.ops.pauli_gates.Pauli, invert0: bool, pauli1: cirq.ops.pauli_gates.Pauli, invert1: bool, *, exponent: Union[float, sympy.core.basic.Basic] = 1.0) → None

**Parameters**

- **pauli0** – The interaction axis for the first qubit.
- **invert0** – Whether to condition on the +1 or -1 eigenvector of the first qubit’s interaction axis.
- **pauli1** – The interaction axis for the second qubit.
- **invert1** – Whether to condition on the +1 or -1 eigenvector of the second qubit’s interaction axis.
- **exponent** – Determines the amount of phasing to apply to the vector equal to the tensor product of the two conditions.
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlled([num_controls, control_values, ...])</code></td>
<td>Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.</td>
</tr>
<tr>
<td><code>num_qubits()</code></td>
<td>The number of qubits this gate acts on.</td>
</tr>
<tr>
<td><code>on(*qubits)</code></td>
<td>Returns an application of this gate to the given qubits.</td>
</tr>
<tr>
<td><code>qubit_index_to_equivalence_group_key(index)</code></td>
<td>Returns a key that differs between non-interchangeable qubits.</td>
</tr>
<tr>
<td><code>validate_args(qubits)</code></td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
<tr>
<td><code>wrap_in_linear_combination([coefficient])</code></td>
<td></td>
</tr>
</tbody>
</table>

### `cirq.PauliInteractionGate.controlled`

**PauliInteractionGate.controlled**

```python
PauliInteractionGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate
```

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- **num_controls**: Total number of control qubits.
- **control_values**: For which control qubit values to apply the sub gate. A sequence of length `num_controls` where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- **control_qid_shape**: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to `(2,) * num_controls`. Specify this argument when using qudits.

### `cirq.PauliInteractionGate.num_qubits`

**PauliInteractionGate.num_qubits**

```python
PauliInteractionGate.num_qubits() → int
```

The number of qubits this gate acts on.

### `cirq.PauliInteractionGate.on`

**PauliInteractionGate.on**

```python
PauliInteractionGate.on(*qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation
```

Returns an application of this gate to the given qubits.

**Parameters**

- ***qubits**: The collection of qubits to potentially apply the gate to.
cirq.PauliInteractionGate.qubit_index_to_equivalence_group_key

PauliInteractionGate.qubit_index_to_equivalence_group_key(index: int) → int

Returns a key that differs between non-interchangeable qubits.

cirq.PauliInteractionGate.validate_args

PauliInteractionGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None

Checks if this gate can be applied to the given qubits.

By default checks that:

* inputs are of type Qid
* len(qubits) == num_qubits()
* qubit_i.dimension == qid_shape[i] for all qubits

Child classes can override. The child implementation should call super().validate_args(qubits) then do custom checks.

Parameters:
qubits – The sequence of qubits to potentially apply the gate to.

Throws: ValueError: The gate can’t be applied to the qubits.

cirq.PauliInteractionGate.wrap_in_linear_combination

PauliInteractionGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) → linear_combinations.LinearCombinationOfGates

Attributes

CNOT

CZ

exponent

global_shift

cirq.PauliInteractionGate.CNOT

PauliInteractionGate.CNOT = cirq.PauliInteractionGate(cirq.Z, False, cirq.X, False)

cirq.PauliInteractionGate.CZ

cirq.PauliInteractionGate.exponent

**property** PauliInteractionGate.exponent

cirq.PauliInteractionGate.global_shift

**property** PauliInteractionGate.global_shift

cirq.PauliString


Initializes a new PauliString.

**Parameters**

- ***contents** – A value or values to convert into a pauli string. This can be a number, a pauli operation, a dictionary from qubit to pauli/identity gates, or collections thereof. If a list of values is given, they are each individually converted and then multiplied from left to right in order.

- **qubit_pauli_map** – Initial dictionary mapping qubits to pauli operations. Defaults to the empty dictionary. Note that, unlike dictionaries passed to contents, this dictionary must not contain any identity gate values. Further note that this argument specifies values that are logically before factors specified in `contents`; `contents` are right multiplied onto the values in this dictionary.

- **coefficient** – Initial scalar coefficient. Defaults to 1.

**Examples**

```python
>>> a, b, c = cirq.LineQubit.range(3)

>>> print(cirq.PauliString([cirq.X(a), cirq.X(a)]))
I

>>> print(cirq.PauliString(-1, cirq.X(a), cirq.Y(b), cirq.Z(c)))
-X(0)*Y(1)*Z(2)

>>> print(cirq.PauliString((a: cirq.X), [-2, 3, cirq.Y(a)]))
-6j*Z(0)

>>> print(cirq.PauliString((a: cirq.I, b: cirq.X())))
X(1)

>>> print(cirq.PauliString((a: cirq.Y),
...                          ... qubit_pauli_map={a: cirq.X}))
1j*Z(0)
```
Methods

```python
circuit.PauliString.commutes_with(other)
```

Returns the Pauli string conjugated by a clifford operation.

```python
circuit.PauliString.conjugated_by(clifford)
```

Returns the Pauli string conjugated by a clifford operation.

```python
circuit.PauliString.controlled_by(*control_qubits[, control_values])
```

Returns a controlled version of this operation. If no control_qubits

```python
circuit.PauliString.dense(qubits)
```

Returns a `cirq.DensePauliString` version of this Pauli string.

```python
circuit.PauliString.equal_up_to_coefficient(other)
```

```python
circuit.PauliString.expectation_from_density_matrix(state, . . .)
```

Evaluate the expectation of this PauliString given a density matrix.

```python
circuit.PauliString.expectation_from_wavefunction(state, . . .[, . . .])
```

Evaluate the expectation of this PauliString given a wavefunction.

```python
circuit.PauliString.from_single(qubit, pauli)
```

THIS FUNCTION IS DEPRECATED.

```python
circuit.PauliString.get(key[, default])
```

```python
circuit.PauliString.items()
```

```python
circuit.PauliString.keys()
```

```python
circuit.PauliString.map_qubits(qubit_map)
```

```python
circuit.PauliString.pass_operations_over(ops[, after_to_before])
```

Determines how the Pauli string changes when conjugated by Cliffords.

```python
circuit.PauliString.to_z_basis_ops()
```

Returns operations to convert the qubits to the computational basis.

```python
circuit.PauliString.transform_qubits(func)
```

Returns the same operation, but with different qubits.

```python
circuit.PauliString.validate_args(qubits)
```

Raises an exception if the qubits don’t match this operation’s qid

```python
circuit.PauliString.values()
```

```python
circuit.PauliString.with_qubits(*new_qubits)
```

```python
circuit.PauliString.zip_items(other)
```

```python
circuit.PauliString.zip_paulis(other)
```

---

### 3.1. API Reference

- **circuit.PauliString.commutes_with**
  
  ```python
  PauliString.commutes_with(other: circuit.ops.pauli_string.PauliString) \rightarrow bool
  ```

- **circuit.PauliString.conjugated_by**
  
  ```python
  PauliString.conjugated_by(clifford: circuit.OP_TREE) \rightarrow PauliString
  ```

  Returns the Pauli string conjugated by a clifford operation.

  The product-of-Paulis $SP$ conjugated by the Clifford operation $S^C$ is

  $$ C^\dagger C P C $$

  For example, conjugating a $+Y$ operation by an $S$ operation results in a $+X$ operation.
In a circuit diagram where \( P \) is a pauli string observable immediately after a Clifford operation \( C \), the pauli string \( P\text{.conjugated\_by}(C) \) is the equivalent pauli string observable just before \( C \).

\[
\begin{align*}
\text{—————} & \quad \text{C} & \quad \text{——P——} \\
= & \quad \text{——C—P————} \\
= & \quad \text{——C—P——C\text{\_}1——C——} \\
= & \quad \text{——C—P——C\text{\_}1——C——} \\
= & \quad \text{——C—P——C\text{\_}1——C——} \\
= & \quad \text{——P\text{.conjugated\_by}(C)——C——} \\
\end{align*}
\]

Analogously, a Pauli product \( P \) can be moved from before a Clifford \( C \) in a circuit diagram to after the Clifford \( C \) by conjugating \( P \) by the inverse of \( C \):

\[
\begin{align*}
\text{——P—C———————} \\
= & \quad \text{——C—P\text{.conjugated\_by}(C\text{\_}1)——} \\
\end{align*}
\]

**Parameters**

- **clifford** – The Clifford operation to conjugate by. This can be an individual operation, or a tree of operations.

Note that the composite Clifford operation defined by a sequence of operations is equivalent to a circuit containing those operations in the given order. Somewhat counterintuitively, this means that the operations in the sequence are conjugated onto the Pauli string in reverse order. For example, \( P\text{.conjugated\_by}([C_1, C_2]) \) is equivalent to \( P\text{.conjugated\_by}(C_2)\text{.conjugated\_by}(C_1) \).

**Examples**

```python
>>> a, b = cirq.LineQubit.range(2)
>>> print(cirq.X(a).conjugated_by(cirq.CZ(a, b)))
X(0)*Z(1)
>>> print(cirq.X(a).conjugated_by(cirq.S(a)))
-Y(0)
>>> print(cirq.X(a).conjugated_by([cirq.H(a), cirq.CNOT(a, b)]))
Z(0)*X(1)
```

**Returns** The Pauli string conjugated by the given Clifford operation.
cirq.PauliString.controlled_by

PauliString.controlled_by(*control_qubits: cirq.ops.raw_types.Qid, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None) → cirq.ops.raw_types.Operation

Returns a controlled version of this operation. If no control_qubits are specified, returns self.

Parameters

- **control_qubits** – Qubits to control the operation by. Required.
- **control_values** – For which control qubit values to apply the operation. A sequence of the same length as control_qubits where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the operation is applied. If unspecified, control values default to 1.

cirq.PauliString.dense

PauliString.dense(qubits: Sequence[cirq.Qid]) → cirq.DensePauliString

Returns a `cirq.DensePauliString` version of this Pauli string.

This method satisfies the invariant `P.dense(qubits).on(*qubits) == P`.

Parameters **qubits** – The implicit sequence of qubits used by the dense pauli string. Specifically, if the returned dense Pauli string was applied to these qubits (via its `on` method) then the result would be a Pauli string equivalent to the receiving Pauli string.

Returns A `cirq.DensePauliString` instance `D` such that `D.on(*qubits)` equals the receiving `cirq.PauliString` instance `P`.

cirq.PauliString.equal_up_to_coefficient

PauliString.equal_up_to_coefficient(other: cirq.ops.pauli_string.PauliString) → bool

cirq.PauliString.expectation_from_density_matrix

PauliString.expectation_from_density_matrix(state: numpy.ndarray, qubit_map: Mapping[cirq.ops.raw_types.Qid, int], *, atol: float = 1e-07, check Preconditions: bool = True) → float

Evaluate the expectation of this PauliString given a density matrix.

Compute the expectation value of this PauliString with respect to an array representing a density matrix. By convention expectation values are defined for Hermitian operators, and so this method will fail if this PauliString is non-Hermitian.
state must be an array representation of a density matrix and have shape \((2 \times n, 2 \times n)\) or \((2, 2, \ldots, 2)\) \((2^n\text{ entries})\), where state is expressed over \(n\) qubits.

qubit_map must assign an integer index to each qubit in this PauliString that determines which bit position of a computational basis state that qubit corresponds to. For example if state represents \(|0\rangle |+\rangle\) and \(q_0, q_1 = \text{cirq.LineQubit}\.range(2)\) then:

```python
cirq.X(q0).expectation(state, qubit_map={q0: 0, q1: 1}) = 0
cirq.X(q0).expectation(state, qubit_map={q0: 1, q1: 0}) = 1
```

Parameters

- **state** – An array representing a valid density matrix.
- **qubit_map** – A map from all qubits used in this PauliString to the indices of the qubits that state is defined over.
- **atol** – Absolute numerical tolerance.
- **check_preconditions** – Whether to check that state represents a valid density matrix.

Returns

The expectation value of the input state.

Raises **NotImplementedError** if this PauliString is non-Hermitian.

**cirq.PauliString.expectation_from_wavefunction**

**PauliString.expectation_from_wavefunction**(state: numpy.ndarray, qubit_map: Mapping[cirq.ops.raw_types.Qid, int], *, atol: float = 1e-07, check_preconditions: bool = True) \(\rightarrow\) float

Evaluate the expectation of this PauliString given a wavefunction.

Compute the expectation value of this PauliString with respect to a wavefunction. By convention expectation values are defined for Hermitian operators, and so this method will fail if this PauliString is non-Hermitian.

state must be an array representation of a wavefunction and have shape \((2 \times n, )\) or \((2, 2, \ldots, 2)\) \((n\text{ entries})\) where state is expressed over \(n\) qubits.

qubit_map must assign an integer index to each qubit in this PauliString that determines which bit position of a computational basis state that qubit corresponds to. For example if state represents \(|0\rangle |+\rangle\) and \(q_0, q_1 = \text{cirq.LineQubit}\.range(2)\) then:
cirq.X(q0).expectation(state, qubit_map={q0: 0, q1: 1}) = 0

cirq.X(q0).expectation(state, qubit_map={q0: 1, q1: 0}) = 1

Parameters

- **state** – An array representing a valid wavefunction.
- **qubit_map** – A map from all qubits used in this PauliString to the indices of the qubits that *state* is defined over.
- **atol** – Absolute numerical tolerance.
- **check_preconditions** – Whether to check that *state* represents a valid wavefunction.

Returns

The expectation value of the input state.

Raises **NotImplementedError** if this PauliString is non-Hermitian.

### cirq.PauliString.from_single

**static** `PauliString.from_single`(qubit: `cirq.ops.raw_types.Qid`, pauli: `cirq.ops.pauli_gates.Pauli`) → `PauliString`

THIS FUNCTION IS DEPRECATED. IT WILL BE REMOVED IN cirq v0.7.0.

Create a PauliString with a single qubit.

### cirq.PauliString.get

`PauliString.get`(key: `cirq.ops.raw_types.Qid`, default=None)

### cirq.PauliString.items

`PauliString.items`() → `ItemsView`

### cirq.PauliString.keys

`PauliString.keys`() → `KeysView[cirq.ops.raw_types.Qid]`

### cirq.PauliString.map_qubits

`PauliString.map_qubits`(qubit_map: `Dict[cirq.ops.raw_types.Qid, cirq.ops.raw_types.Qid]`) → `cirq.ops.pauli_string.PauliString`
cirq.PauliString.pass_operations_over

PauliString.pass_operations_over(ops: Iterable[cirq.Operation], after_to_before: bool = False) → PauliString

Determines how the Pauli string changes when conjugated by Cliffords.

The output and input pauli strings are related by a circuit equivalence. In particular, this circuit:

```
ops——INPUT_PAULI_STRING——
```

will be equivalent to this circuit:

```
OUTPUT_PAULI_STRING——ops——
```

up to global phase (assuming after_to_before is not set).

If ops together have matrix C, the Pauli string has matrix P, and the output Pauli string has matrix P’, then P’ = C^-1 P C up to global phase.

Setting after_to_before inverts the relationship, so that the output is the input and the input is the output. Equivalently, it inverts C.

Parameters

- **ops** – The operations to move over the string.
- **after_to_before** – Determines whether the operations start after the pauli string, instead of before (and so are moving in the opposite direction).

cirq.PauliString.to_z_basis_ops

PauliString.to_z_basis_ops() → Iterator[cirq.ops.raw_types.Operation]

Returns operations to convert the qubits to the computational basis.

cirq.PauliString.transform_qubits

PauliString.transform_qubits(func: Callable[cirq.ops.raw_types.Qid, cirq.ops.raw_types.Qid]) → cirq.ops.raw_types.Operation

Returns the same operation, but with different qubits.

Parameters **func** – The function to use to turn each current qubit into a desired new qubit.

Returns

The receiving operation but with qubits transformed by the given function.
cirq.PauliString.validate_args

PauliString.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid])

 Raises an exception if the qubits don’t match this operation’s qid shape.
 Call this method from a subclass’s with_qubits method.

 Parameters qubits – The new qids for the operation.

 Raises ValueError – The operation had qids that don’t match it’s qid shape.

cirq.PauliString.values

PauliString.values() → ValuesView[cirq.ops.pauli_gates.Pauli]

cirq.PauliString.with_qubits

PauliString.with_qubits(*new_qubits: cirq.ops.raw_types.Qid) → cirq.ops.pauli_string.PauliString

cirq.PauliString.zip_items


cirq.PauliString.zip_paulis


Attributes

coefficient
gate
qubits

cirq.PauliString.coefficient

property PauliString.coefficient

cirq.PauliString.gate

property PauliString.gate
cirq.PauliString.qubits

property PauliString.qubits

cirq.PauliTransform

class cirq.PauliTransform

    __init__(self)
        Initialize self. See help(type(self)) for accurate signature.

Methods

    count(value)
    index(value, [start, [stop]])
        Raises ValueError if the value is not present.

cirq.PauliTransform.count

    PauliTransform.count(value) \rightarrow integer – return number of occurrences of value

cirq.PauliTransform.index

    PauliTransform.index(value[, start[, stop]]) \rightarrow integer – return first index of value.
        Raises ValueError if the value is not present.

Attributes

    flip
        Alias for field number 1
    to
        Alias for field number 0

cirq.PauliTransform.flip

    property PauliTransform.flip
        Alias for field number 1

cirq.PauliTransform.to

    property PauliTransform.to
        Alias for field number 0
cirq.SingleQubitCliffordGate

class cirq.SingleQubitCliffordGate(*, _rotation_map: Dict[cirq.ops.pauli_gates.Pauli, cirq.ops.clifford_gate.PauliTransform], _inverse_map: Dict[cirq.ops.pauli_gates.Pauli, cirq.ops.clifford_gate.PauliTransform])

Any single qubit Clifford rotation.

__init__(*_, _rotation_map: Dict[cirq.ops.pauli_gates.Pauli, cirq.ops.clifford_gate.PauliTransform], _inverse_map: Dict[cirq.ops.pauli_gates.Pauli, cirq.ops.clifford_gate.PauliTransform]) → None

Initialize self. See help(type(self)) for accurate signature.

Methods

commutes_with(gate_or_pauli)

commutes_with_pauli(pauli)

commutes_with_single_qubit_gate(gate)

Tests if the two circuits would be equivalent up to global phase:

controlled([num_controls, control_values, ...])

Returns a controlled version of this gate. If no arguments are

decompose_rotation()

Returns ((first_rotation_axis, first_rotation_quarter_turns), ...)

equivalent_gate_before(after)

Returns a SingleQubitCliffordGate such that the circuits

from_double_map([pauli_map_to, x_to, y_to, z_to])

Returns a SingleQubitCliffordGate for the

from_pauli(pauli[, sqrt])

from_quarter_turns(pauli, quarter_turns)

from_single_map([pauli_map_to, x_to, y_to, z_to])

Returns a SingleQubitCliffordGate for the

from_xz_map(x_to, z_to)

Returns a SingleQubitCliffordGate for the specified transforms.

merged_with(second)

Returns a SingleQubitCliffordGate such that the circuits

num_qubits()

num_qubits() The number of qubits this gate acts on.

crd(*qubits)

Returns an application of this gate to the given qubits.

on_each(*targets)

Returns a list of operations applying the gate to all targets.

Continued on next page
Table 91 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| `transform(pauli)`                          | Modify the Pauli gate to an equivalent Circuit.
| `validate_args(qubits)`                     | Checks if this gate can be applied to the given qubits.                     |
| `wrap_in_linear_combination([coefficient])` | Wrap the gate in a linear combination.                                      |

### `cirq.SingleQubitCliffordGate.commutes_with`

`SingleQubitCliffordGate.commutes_with(gate_or_pauli: Union[SingleQubitCliffordGate, cirq.ops.pauli_gates.Pauli]) → bool`

### `cirq.SingleQubitCliffordGate.commutes_with_pauli`

`SingleQubitCliffordGate.commutes_with_pauli(pauli: cirq.ops.pauli_gates.Pauli) → bool`

### `cirq.SingleQubitCliffordGate.commutes_with_single_qubit_gate`

`SingleQubitCliffordGate.commutes_with_single_qubit_gate(gate: cirq.ops.clifford_gate.SingleQubitCliffordGate) → bool`

Tests if the two circuits would be equivalent up to global phase:

`~self~ · gate · ~gate~ · self~`

### `cirq.SingleQubitCliffordGate.controlled`

`SingleQubitCliffordGate.controlled(num_controls: int = None, control_values: Optional[Sequence[Union[int, Collection[int]]]] = None, control_qid_shape: Optional[Tuple[int, ...]] = None) → cirq.ops.raw_types.Gate`

Returns a controlled version of this gate. If no arguments are specified, defaults to a single qubit control.

- `num_controls`: Total number of control qubits.
- `control_values`: For which control qubit values to apply the sub gate. A sequence of length `num_controls` where each entry is an integer (or set of integers) corresponding to the qubit value (or set of possible values) where that control is enabled. When all controls are enabled, the sub gate is applied. If unspecified, control values default to 1.
- `control_qid_shape`: The qid shape of the controls. A tuple of the expected dimension of each control qid. Defaults to `(2,) * num_controls`. Specify this argument when using qudits.
**cirq.SingleQubitCliffordGate.decompose_rotation**

```python
cirq.SingleQubitCliffordGate.decompose_rotation() → Sequence[Tuple[cirq.ops.pauli_gates.Pauli, int]]
```

Returns `((first_rotation_axis, first_rotation_quarter_turns), . . . )`.

This is a sequence of zero, one, or two rotations.

**cirq.SingleQubitCliffordGate.equivalent_gate_before**

```python
cirq.SingleQubitCliffordGate.equivalent_gate_before(after: cirq.ops.clifford_gate.SingleQubitCliffordGate) → cirq.ops.clifford_gate.SingleQubitCliffordGate
```

Returns a SingleQubitCliffordGate such that the circuits

- output
- self
- gate

are equivalent up to global phase.

**cirq.SingleQubitCliffordGate.from_double_map**

```python
```

Returns a SingleQubitCliffordGate for the specified transform with a 90 or 180 degree rotation.

Either `pauli_map_to` or two of `(x_to, y_to, z_to)` may be specified.

**Parameters**

- `pauli_map_to` – A dictionary with two key value pairs describing two transforms.
- `x_to` – The transform from `cirq.X`
- `y_to` – The transform from `cirq.Y`
- `z_to` – The transform from `cirq.Z`

**cirq.SingleQubitCliffordGate.from_pauli**

```python
static cirq.SingleQubitCliffordGate.from_pauli(pauli: cirq.ops.pauli_gates.Pauli, sqrt: bool = False) → cirq.ops.clifford_gate.SingleQubitCliffordGate
```

Either `pauli_map_to` or two of `(x_to, y_to, z_to)` may be specified.
cirq.SingleQubitCliffordGate.from_quarter_turns

static SingleQubitCliffordGate.from_quarter_turns(pauli: cirq.ops.pauli_gates.Pauli, quarter_turns: int) → cirq.ops.clifford_gate.SingleQubitCliffordGate

cirq.SingleQubitCliffordGate.from_single_map

static SingleQubitCliffordGate.from_single_map(pauli_map_to: Optional[Dict[cirq.ops.pauli_gates.Pauli, Tuple[cirq.ops.pauli_gates.Pauli, bool]]] = None, *x_to: Optional[Tuple[cirq.ops.pauli_gates.Pauli, bool]] = None, *y_to: Optional[Tuple[cirq.ops.pauli_gates.Pauli, bool]] = None, *z_to: Optional[Tuple[cirq.ops.pauli_gates.Pauli, bool]] = None) → cirq.ops.clifford_gate.SingleQubitCliffordGate

Returns a SingleQubitCliffordGate for the specified transform with a 90 or 180 degree rotation.

The arguments are exclusive, only one may be specified.

Parameters

- **pauli_map_to** – A dictionary with a single key value pair describing the transform.
- **x_to** – The transform from cirq.X
- **y_to** – The transform from cirq.Y
- **z_to** – The transform from cirq.Z


cirq.SingleQubitCliffordGate.from_xz_map

static SingleQubitCliffordGate.from_xz_map(x_to: Tuple[cirq.ops.pauli_gates.Pauli, bool], z_to: Tuple[cirq.ops.pauli_gates.Pauli, bool]) → cirq.ops.clifford_gate.SingleQubitCliffordGate

Returns a SingleQubitCliffordGate for the specified transforms.

The Y transform is derived from the X and Z.

Parameters

- **x_to** – Which Pauli to transform X to and if it should negate.
- **z_to** – Which Pauli to transform Z to and if it should negate.
cirq.SingleQubitCliffordGate.merged_with

SingleQubitCliffordGate.merged_with(second: cirq.ops.clifford_gate.SingleQubitCliffordGate) → cirq.ops.clifford_gate.SingleQubitCliffordGate

Returns a SingleQubitCliffordGate such that the circuits
–output– and –self–second–
are equivalent up to global phase.

cirq.SingleQubitCliffordGate.num_qubits

SingleQubitCliffordGate.num_qubits() → int

The number of qubits this gate acts on.

cirq.SingleQubitCliffordGate.on

SingleQubitCliffordGate.on(*qubits: cirq.ops.raw_types.Qid) → cirq.ops.raw_types.Operation

Returns an application of this gate to the given qubits.

Parameters:
qubits – The collection of qubits to potentially apply the gate to.

cirq.SingleQubitCliffordGate.on_each

SingleQubitCliffordGate.on_each(*targets: Union[cirq.ops.raw_types.Qid, Iterable[Any]]) → List[cirq.ops.raw_types.Operation]

Returns a list of operations applying the gate to all targets.

Parameters:
targets – The qubits to apply this gate to.

Returns:
Operations applying this gate to the target qubits.

Raises ValueError if targets are not instances of Qid or List[Qid] –

cirq.SingleQubitCliffordGate.transform

SingleQubitCliffordGate.transform(pauli: cirq.ops.pauli_gates.Pauli) → cirq.ops.clifford_gate.PauliTransform

cirq.SingleQubitCliffordGate.validate_args

SingleQubitCliffordGate.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid]) → None

Checks if this gate can be applied to the given qubits.

By default checks that:
• inputs are of type Qid
• len(qubits) == num_qubits()
• qubit_i.dimension == qid_shape[i] for all qubits
Child classes can override. The child implementation should call `super().validate_args(qubits)` then do custom checks.

**Parameters**  
`qubits` – The sequence of qubits to potentially apply the gate to.

**Throws:**  
ValueError: The gate can’t be applied to the qubits.

cirq.SingleQubitCliffordGate.wrap_in_linear_combination

```python
SingleQubitCliffordGate.wrap_in_linear_combination(coefficient: Union[complex, float, int] = 1) \rightarrow linear_combinations.LinearCombinationOfGates
```

**Attributes**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H</strong></td>
<td></td>
</tr>
<tr>
<td><strong>I</strong></td>
<td></td>
</tr>
<tr>
<td><strong>X</strong></td>
<td></td>
</tr>
<tr>
<td><strong>X_nsqrt</strong></td>
<td></td>
</tr>
<tr>
<td><strong>X_sqrt</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Y</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Y_nsqrt</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Y_sqrt</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Z</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Z_nsqrt</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Z_sqrt</strong></td>
<td></td>
</tr>
</tbody>
</table>

cirq.SingleQubitCliffordGate.H

```python
SingleQubitCliffordGate.H = cirq.SingleQubitCliffordGate(X:+Z, Y:-Y, Z:+X)
```

cirq.SingleQubitCliffordGate.I

```python
SingleQubitCliffordGate.I = cirq.SingleQubitCliffordGate(X:+X, Y:+Y, Z:+Z)
```

cirq.SingleQubitCliffordGate.X

```python
SingleQubitCliffordGate.X = cirq.SingleQubitCliffordGate(X:+X, Y:-Y, Z:-Z)
```

cirq.SingleQubitCliffordGate.X_nsqrt

```python
SingleQubitCliffordGate.X_nsqrt = cirq.SingleQubitCliffordGate(X:+X, Y:-Z, Z:+Y)
```

cirq.SingleQubitCliffordGate.X_sqrt

```python
SingleQubitCliffordGate.X_sqrt = cirq.SingleQubitCliffordGate(X:+X, Y:+Z, Z:-Y)
```
cirq.SingleQubitCliffordGate.Y

SingleQubitCliffordGate.Y = cirq.SingleQubitCliffordGate(X:-X, Y:+Y, Z:-Z)

cirq.SingleQubitCliffordGate.Y_nsqrt

SingleQubitCliffordGate.Y_nsqrt = cirq.SingleQubitCliffordGate(X:+Z, Y:+Y, Z:-X)

cirq.SingleQubitCliffordGate.Y_sqrt

SingleQubitCliffordGate.Y_sqrt = cirq.SingleQubitCliffordGate(X:-Z, Y:+Y, Z:+X)

cirq.SingleQubitCliffordGate.Z

SingleQubitCliffordGate.Z = cirq.SingleQubitCliffordGate(X:-X, Y:-Y, Z:+Z)

cirq.SingleQubitCliffordGate.Z_nsqrt

SingleQubitCliffordGate.Z_nsqrt = cirq.SingleQubitCliffordGate(X:-Y, Y:+X, Z:+Z)

cirq.SingleQubitCliffordGate.Z_sqrt


### 3.1.10 Circuits and Schedules

Utilities for representing and manipulating quantum computations via Circuits, Operations, and Moments.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circuit</td>
<td>A mutable list of groups of operations to apply to some qubits.</td>
</tr>
<tr>
<td>CircuitDag</td>
<td>A representation of a Circuit as a directed acyclic graph.</td>
</tr>
<tr>
<td>flatten_op_tree(root[, preserve_moments])</td>
<td>Performs an in-order iteration of the operations (leaves) in an OP_TREE.</td>
</tr>
<tr>
<td>freeze_op_tree(root)</td>
<td>Replaces all iterables in the OP_TREE with tuples.</td>
</tr>
<tr>
<td>GateOperation(gate, qubits)</td>
<td>An application of a gate to a sequence of qubits.</td>
</tr>
<tr>
<td>InsertStrategy(name, doc)</td>
<td>Indicates preferences on how to add multiple operations to a circuit.</td>
</tr>
<tr>
<td>Moment([operations])</td>
<td>A simplified time-slice of operations within a sequenced circuit.</td>
</tr>
<tr>
<td>moment_by_moment_schedule(device, circuit)</td>
<td>Returns a schedule aligned with the moment structure of the Circuit.</td>
</tr>
<tr>
<td>op_gate_of_type(op, gate_type)</td>
<td>THIS FUNCTION IS DEPRECATED.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP_TREE</td>
<td>Union type; Union[X, Y] means either X or Y.</td>
</tr>
<tr>
<td>Operation</td>
<td>An effect applied to a collection of qubits.</td>
</tr>
<tr>
<td>ParallelGateOperation(gate, qubits)</td>
<td>An application of several copies of a gate to a group of qubits.</td>
</tr>
</tbody>
</table>

Continued on next page
### cirq.Circuit


A mutable list of groups of operations to apply to some qubits.

Methods returning information about the circuit:
- `next_moment_operating_on`
- `prev_moment_operating_on`
- `next_moments_operating_on`
- `operation_at`
- `all_qubits`
- `all_operations`
- `findall_operations`
- `findall_operations_between`
- `findall_operations_until_blocked`
- `findall_operations_with_gate_type`
- `reachable_frontier_from`
- `has_measurements`
- `are_all_matches_terminal`
- `are_all_measurements_terminal`
- `unitary`
- `final_wavefunction`
- `to_text_diagram`
- `to_text_diagram_drawer`

Methods for mutation:
- `insert`
- `append`
- `insert_into_range`
- `clear_operations_touching`
- `batch_insert`
- `batch_remove`
- `batch_insert_into`
- `insert_at_frontier`
Circuits can also be iterated over, for moment in circuit:
...
and sliced,
circuit[1:3] is a new Circuit made up of two moments, the first being circuit[1] and the second being circuit[2];
and concatenated,
circuit1 + circuit2 is a new Circuit made up of the moments in circuit1 followed by the moments in circuit2;
and multiplied by an integer,
circuit * k is a new Circuit made up of the moments in circuit repeated k times.
and mutated,
circuit[1:7] = [Moment(…)]

```python
```
Initializes a circuit.

Parameters

- `contents` – The initial list of moments and operations defining the circuit. You can also pass in operations, lists of operations, or generally anything meeting the `cirq.OP_TREE` contract. Non-moment entries will be inserted according to the specified insertion strategy.
- `strategy` – When initializing the circuit with operations and moments from `contents`, this determines how the operations are packed together. This option does not affect later insertions into the circuit.
- `device` – Hardware that the circuit should be able to run on.

Methods

- `all_operations()` _iters over the operations applied by this circuit.
- `all_qubits()` _Returns the qubits acted upon by Operations in this circuit.
- `append(moment_or_operation_tree[, strategy])` _Appends operations onto the end of the circuit.
- `apply_unitary_effect_to_state([...])` _THIS FUNCTION IS DEPRECATED.
- `are_all_matches_terminal(predicate)` _Check whether all of the ops that satisfy a predicate are terminal.
- `are_all_measurements_terminal()`  WHETHER all measurement gates are at the end of the circuit.
- `batch_insert(insertions)`  Applies a batched insert operation to the circuit.
- `batch_insert_into(insert_intos)`  Inserts operations into empty spaces in existing moments.
- `batch_remove(removals)`  Removes several operations from a circuit.
- `clear_operations_touching(qubits, moment_indices)`  Clears operations that are touching given qubits at given moments.
- `copy()`  _Continued on next page_
Table 94 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>final_wavefunction</code>([<code>initial_state</code>, ...])</td>
<td>Left-multiplies a state vector by the circuit’s unitary effect.</td>
</tr>
<tr>
<td><code>findall_operations</code>(<code>predicate</code>)</td>
<td>Find the locations of all operations that satisfy a given condition.</td>
</tr>
<tr>
<td><code>findall_operations_between</code>(start_frontier, ...)</td>
<td>Finds operations between the two given frontiers.</td>
</tr>
<tr>
<td><code>findall_operations_until_blocked</code>(start_frontier)</td>
<td>Finds all operations until a blocking operation is hit. This returns</td>
</tr>
<tr>
<td><code>findall_operations_with_gate_type</code>(gate_type)</td>
<td>the locations of all gate operations of a given type.</td>
</tr>
<tr>
<td><code>from_ops</code>(*operations[, strategy, device])</td>
<td>THIS FUNCTION IS DEPRECATED.</td>
</tr>
<tr>
<td><code>has_measurements</code>()</td>
<td></td>
</tr>
<tr>
<td><code>insert</code>(index, moment_or_operation_tree[,...])</td>
<td>Inserts operations into the circuit.</td>
</tr>
<tr>
<td><code>insert_at_frontier</code>(operations, start[, frontier])</td>
<td>Inserts operations inline at frontier.</td>
</tr>
<tr>
<td><code>insert_into_range</code>(operations, start, end)</td>
<td>Writes operations inline into an area of the circuit.</td>
</tr>
<tr>
<td><code>next_moment_operating_on</code>(qubits[, ...])</td>
<td>Finds the index of the next moment that touches the given qubits.</td>
</tr>
<tr>
<td><code>next_moments_operating_on</code>(qubits[, ...])</td>
<td>Finds the index of the next moment that touches each qubit.</td>
</tr>
<tr>
<td><code>operation_at</code>(qubit, moment_index)</td>
<td>Finds the operation on a qubit within a moment, if any.</td>
</tr>
<tr>
<td><code>prev_moment_operating_on</code>(qubits[, ...])</td>
<td>Finds the index of the next moment that touches the given qubits.</td>
</tr>
<tr>
<td><code>qid_shape</code>([<code>qubit_order</code>, ...])</td>
<td></td>
</tr>
<tr>
<td><code>reachable_frontier_from</code>(start_frontier, *[, ...])</td>
<td>Determines how far can be reached into a circuit under certain rules.</td>
</tr>
<tr>
<td><code>save_qasm</code>(file_path[, header, precision,...])</td>
<td>Save a QASM file equivalent to the circuit.</td>
</tr>
<tr>
<td><code>to_qasm</code>([header, precision, qubit_order])</td>
<td>Returns QASM equivalent to the circuit.</td>
</tr>
<tr>
<td><code>to_text_diagram</code>(*[, use_unicode_characters, ...])</td>
<td>Returns text containing a diagram describing the circuit.</td>
</tr>
<tr>
<td><code>to_text_diagram_drawer</code>(*[, ...])</td>
<td>Returns a TextDiagramDrawer with the circuit drawn into it.</td>
</tr>
<tr>
<td><code>to_unitary_matrix</code>([<code>qubit_order</code>, ...])</td>
<td>THIS FUNCTION IS DEPRECATED.</td>
</tr>
<tr>
<td><code>transform_qubits</code>(func, *[, new_device])</td>
<td>Returns the same circuit, but with different qubits.</td>
</tr>
<tr>
<td><code>unitary</code>([<code>qubit_order</code>, ...])</td>
<td>Converts the circuit into a unitary matrix, if possible.</td>
</tr>
<tr>
<td><code>with_device</code>(new_device[, qubit_mapping])</td>
<td>Maps the current circuit onto a new device, and validates.</td>
</tr>
<tr>
<td><code>with_noise</code>(noise)</td>
<td>Make a noisy version of the circuit.</td>
</tr>
</tbody>
</table>

### cirq.Circuit.all_operations

`Circuit.all_operations()` → `Iterator[cirq.ops.raw_types.Operation]`

Iterates over the operations applied by this circuit.

Operations from earlier moments will be iterated over first. Operations within a moment are iterated in the order they were given to the
moment’s constructor.

```python
Cirq Documentation, Release 0.6.1

cirq.Circuit.all_qubits

Circuit.all_qubits() \rightarrow \text{FrozenSet[cirq.ops.raw_types.Qid]}

Returns the qubits acted upon by Operations in this circuit.

cirq.Circuit.append


Appends operations onto the end of the circuit.

Moments within the operation tree are appended intact.

Parameters

• moment_or_operation_tree – The moment or operation tree to append.
• strategy – How to pick/create the moment to put operations into.

cirq.Circuit.apply_unitary_effect_to_state

Circuit.apply_unitary_effect_to_state(initial_state: \text{Union[int, numpy.ndarray]} = 0, qubit_order: cirq.QubitOrderOrList = \langle\text{cirq.ops.qubit_order.QubitOrder object}\rangle, qubits_that_should_be_present: \text{Iterable[cirq.Qid]} = (), ignore_terminal_measurements: \text{bool} = True, dtype: \text{Type[numpy.number]} = \langle\text{class \text{'numpy.complex128'}\rangle\rangle \rightarrow \text{numpy.ndarray}

THIS FUNCTION IS DEPRECATED.

IT WILL BE REMOVED IN cirq v0.7.0.

Use cirq.final_wavefunction(circuit) or Circuit.final_wavefunction() instead

Left-multiplies a state vector by the circuit’s unitary effect.

A circuit’s "unitary effect" is the unitary matrix produced by multiplying together all of its gates’ unitary matrices. A circuit with non-unitary gates (such as measurement or parameterized gates) does not have a well-defined unitary effect, and the method will fail if such operations are present.

For convenience, terminal measurements are automatically ignored instead of causing a failure. Set the `ignore_terminal_measurements` argument to False to disable this behavior.

This method is equivalent to left-multiplying the input state by `cirq.unitary(circuit)` but it’s computed in a more efficient way.

Args:

initial_state: The input state for the circuit. This can be an int

(continues on next page)
or a vector. When this is an int, it refers to a computational basis state (e.g. 5 means initialize to `|5 = |...000101`). If this is a state vector, it directly specifies the initial state's amplitudes. The vector must be a flat numpy array with a type that can be converted to np.complex128.

qubit_order: Determines how qubits are ordered when passing matrices into np.kron.

qubits_that_should_be_present: Qubits that may or may not appear in operations within the circuit, but that should be included regardless when generating the matrix.

ignore_terminal_measurements: When set, measurements at the end of the circuit are ignored instead of causing the method to fail.

dtype: The numpy dtype for the returned unitary. Defaults to np.complex128. Specifying np.complex64 will run faster at the cost of precision. `dtype` must be a complex np.dtype, unless all operations in the circuit have unitary matrices with exclusively real coefficients (e.g. an H + TOFFOLI circuit).

Returns:
A (possibly gigantic) numpy array storing the superposition that came out of the circuit for the given input state.

Raises:
ValueError: The circuit contains measurement gates that are not ignored.
TypeError: The circuit contains gates that don't have a known unitary matrix, e.g. gates parameterized by a Symbol.

cirq.Circuit.are_all_matches_terminal

Circuit.are_all_matches_terminal (predicate: Callable[cirq.Operation, bool])
Check whether all of the ops that satisfy a predicate are terminal.

Parameters
predicate – A predicate on ops. Operations which is being checked.

Returns
Whether or not all Operations in a circuit that satisfy the given predicate are terminal.

cirq.Circuit.are_all_measurements_terminal

Circuit.are_all_measurements_terminal ()
Whether all measurement gates are at the end of the circuit.

cirq.Circuit.batch_insert

Circuit.batch_insert (insertions: Iterable[Tuple[int, cirq.OP_TREE]]) → None
Applies a batched insert operation to the circuit.

Transparently handles the fact that earlier insertions may shift the index that later insertions should occur at. For example, if you insert an operation at index 2 and at index 4, but the insert at index 2
causes a new moment to be created, then the insert at “4” will actually occur at index 5 to account for the shift from the new moment.

All insertions are done with the strategy ‘EARLIEST’.

When multiple inserts occur at the same index, the gates from the later inserts end up before the gates from the earlier inserts (exactly as if you’d called list.insert several times with the same index: the later inserts shift the earliest inserts forward).

**Parameters insertions** – A sequence of (insert_index, operations) pairs indicating operations to add into the circuit at specific places.

**cirq.Circuit.batch_insert_into**

Circuit.batch_insert_into(insert_intos: Iterable[Tuple[int, cirq.ops.raw_types.Operation]]) → None

Inserts operations into empty spaces in existing moments.

If any of the insertions fails (due to colliding with an existing operation), this method fails without making any changes to the circuit.

**Parameters insert_intos** – A sequence of (moment_index, new_operation) pairs indicating a moment to add a new operation into.

**ValueError**: One of the insertions collided with an existing operation.

**IndexError**: Inserted into a moment index that doesn’t exist.

**cirq.Circuit.batch_remove**

Circuit.batch_remove(removals: Iterable[Tuple[int, cirq.Operation]]) → None

Removes several operations from a circuit.

**Parameters removals** – A sequence of (moment_index, operation) tuples indicating operations to delete from the moments that are present. All listed operations must actually be present or the edit will fail (without making any changes to the circuit).

**ValueError**: One of the operations to delete wasn’t present to start with.

**IndexError**: Deleted from a moment that doesn’t exist.

**cirq.Circuit.clear_operations_touching**

Circuit.clear_operations_touching(qubits: Iterable[cirq.Qid], moment_indices: Iterable[int])

Clears operations that are touching given qubits at given moments.
• **qubits** – The qubits to check for operations on.
• **moment_indices** – The indices of moments to check for operations within.

```python
cirq.Circuit.copy
```

```python
Circuit.copy() → cirq.circuits.circuit.Circuit
```

```python
cirq.Circuit.final_wavefunction
```

```python
```

Left-multiplies a state vector by the circuit’s unitary effect.

A circuit’s “unitary effect” is the unitary matrix produced by multiplying together all of its gates’ unitary matrices. A circuit with non-unitary gates (such as measurement or parameterized gates) does not have a well-defined unitary effect, and the method will fail if such operations are present.

For convenience, terminal measurements are automatically ignored instead of causing a failure. Set the `ignore_terminal_measurements` argument to False to disable this behavior.

This method is equivalent to left-multiplying the input state by `cirq.unitary(circuit)` but it’s computed in a more efficient way.

**Parameters**

• **initial_state** – The input state for the circuit. This can be an int or a vector. When this is an int, it refers to a computational basis state (e.g., 5 means initialize to |5⟩ = .. .000101). If this is a state vector, it directly specifies the initial state’s amplitudes. The vector must be a flat numpy array with a type that can be converted to np.complex128.

• **qubit_order** – Determines how qubits are ordered when passing matrices into np.kron.

• **qubits_that_should_be_present** – Qubits that may or may not appear in operations within the circuit, but that should be included regardless when generating the matrix.

• **ignore_terminal_measurements** – When set, measurements at the end of the circuit are ignored instead of causing the method to fail.
• **dtype** – The numpy dtype for the returned unitary. Defaults to np.complex128. Specifying np.complex64 will run faster at the cost of precision. *dtype* must be a complex np.dtype, unless all operations in the circuit have unitary matrices with exclusively real coefficients (e.g. an H + TOFFOLI circuit).

**Returns** A (possibly gigantic) numpy array storing the superposition that came out of the circuit for the given input state.

**Raises**

• **ValueError** – The circuit contains measurement gates that are not ignored.

• **TypeError** – The circuit contains gates that don’t have a known unitary matrix, e.g. gates parameterized by a Symbol.

cirq.Circuit.findall_operations

*Circuit*. `findall_operations` *(predicate: Callable[cirq.Operation, bool]) → Iterable[Tuple[int, cirq.ops.raw_types.Operation]]*

Find the locations of all operations that satisfy a given condition.

This returns an iterator of (index, operation) tuples where each operation satisfies op_cond(operation) is truthy. The indices are in order of the moments and then order of the ops within that moment.

**Parameters** `predicate` – A method that takes an Operation and returns a Truthy value indicating the operation meets the find condition.

**Returns** An iterator (index, operation)`'s that satisfy the op_condition.

cirq.Circuit.findall_operations_between

*Circuit*. `findall_operations_between` *(start_frontier: Dict[cirq.Qid, int], end_frontier: Dict[cirq.Qid, int], omit_crossing_operations: bool = False) → List[Tuple[int, cirq.ops.raw_types.Operation]]*

Finds operations between the two given frontiers.

If a qubit is in `start_frontier` but not `end_frontier`, its end index defaults to the end of the circuit. If a qubit is in `end_frontier` but not `start_frontier`, its start index defaults to the start of the circuit. Operations on qubits not mentioned in either frontier are not included in the results.

**Parameters**

• `start_frontier` – Just before where to start searching for operations, for each qubit of interest. Start frontier indices are inclusive.

• `end_frontier` – Just before where to stop searching for operations, for each qubit of interest. End frontier indices are exclusive.
• **omit_crossing_operations** – Determines whether or not operations that cross from a location between the two frontiers to a location outside the two frontiers are included or excluded. (Operations completely inside are always included, and operations completely outside are always excluded.)

**Returns** A list of tuples. Each tuple describes an operation found between the two frontiers. The first item of each tuple is the index of the moment containing the operation, and the second item is the operation itself. The list is sorted so that the moment index increases monotonically.

>cirq.Circuit.findall_operations_until_blocked

```python
Circuit.findall_operations_until_blocked(start_frontier: Dict[cirq.Qid, int],
is_blocker: Callable[cirq.Operation, bool] = <function Circuit.<lambda>>) -> List[Tuple[int, cirq.ops.raw_types.Operation]]
```

Finds all operations until a blocking operation is hit. This returns a list of all operations from the starting frontier until a blocking operation is encountered. An operation is part of the list if it is involves a qubit in the start_frontier dictionary, comes after the moment listed in that dictionary, and before any blocking operations that involve that qubit. Operations are only considered to be blocking the qubits that they operate on, so a blocking operation that does not operate on any qubit in the starting frontier is not actually considered blocking. See reachable_frontier_from for a more in depth example of reachable states.

**Parameters**

- **start_frontier** – A starting set of reachable locations.
- **is_blocker** – A predicate that determines if operations block reachability. Any location covered by an operation that causes is_blocker to return True is considered to be an unreachable location.

**Returns** A list of tuples. Each tuple describes an operation found between the start frontier and a blocking operation. The first item of each tuple is the index of the moment containing the operation, and the second item is the operation itself.

>cirq.Circuit.findall_operations_with_gate_type

```python
Circuit.findall_operations_with_gate_type(gate_type: Type[T_DESIRED_GATE_TYPE]) -> Iterable[Tuple[int, cirq.ops.gate_operation.GateOperation, T_DESIRED_GATE_TYPE]]
```

Find the locations of all gate operations of a given type.

**Parameters** **gate_type** – The type of gate to find, e.g. XPowGate or MeasurementGate.

**Returns** An iterator (index, operation, gate)’s for operations with the given gate type.
cirq.Circuit.from_ops

```python
```

**THIS FUNCTION IS DEPRECATED.**

IT WILL BE REMOVED IN cirq v0.8.0.

use cirq.Circuit(*ops) instead.

Creates an empty circuit and appends the given operations.

**Args:**
- `operations`: The operations to append to the new circuit.
- `strategy`: How to append the operations.
- `device`: Hardware that the circuit should be able to run on.

**Returns:**
The constructed circuit containing the operations.

cirq.Circuit.has_measurements

```python
Circuit.has_measurements()
```

cirq.Circuit.insert

```python
```

Inserts operations into the circuit.

Operations are inserted into the moment specified by the index and 'InsertStrategy'.

Moments within the operation tree are inserted intact.

**Parameters**
- `index` – The index to insert all of the operations at.
- `moment_or_operation_tree` – The moment or operation tree to insert.
- `strategy` – How to pick/create the moment to put operations into.

**Returns**
The insertion index that will place operations just after the operations that were inserted by this method.

**Raises** `ValueError` – Bad insertion strategy.

cirq.Circuit.insert_at_frontier

```python
```

Inserts operations inline at frontier.

**Parameters**
- `operations` – The operations to insert at the frontier.
- `start` – The index at which to start inserting operations.
- `frontier` – A dictionary mapping qubits to their indices in the frontier.

**Returns**
The updated frontier dictionary with the operations inserted.

**Raises** `ValueError` – If the operation tree has a moment in a wider area than the frontier.

3.1. API Reference
Parameters

• **operations** – the operations to insert
  
• **start** – the moment at which to start inserting the operations
  
• **frontier** – frontier[q] is the earliest moment in which an operation acting on qubit q can be placed.

```python
Circuit.insert_into_range(operations: Union[cirq.ops.raw_types.Operation, cirq.ops.op_tree.OpTree], start: int, end: int) → int
```

Writes operations inline into an area of the circuit.

**Parameters**

• **start** – The start of the range (inclusive) to write the given operations into.

• **end** – The end of the range (exclusive) to write the given operations into. If there are still operations remaining, new moments are created to fit them.

• **operations** – An operation or tree of operations to insert.

**Returns** An insertion index that will place operations after the operations that were inserted by this method.

**Raises** `IndexError` – Bad inline_start and/or inline_end.

```python
Circuit.next_moment_operating_on(qubits: Iterable[cirq.Qid], start_moment_index: int = 0, max_distance: int = None) → Optional[int]
```

Finds the index of the next moment that touches the given qubits.

**Parameters**

• **qubits** – We’re looking for operations affecting any of these qubits.

• **start_moment_index** – The starting point of the search.

• **max_distance** – The number of moments (starting from the start index and moving forward) to check. Defaults to no limit.

**Returns** None if there is no matching moment, otherwise the index of the earliest matching moment.

**Raises** `ValueError` – negative max_distance.

```python
Circuit.next_moments_operating_on(qubits: Iterable[cirq.Qid], start_moment_index: int = 0) → Dict[cirq.ops.raw_types.Qid, int]
```

Finds the index of the next moment that touches each qubit.

**Parameters**

• **qubits** – The qubits to find the next moments acting on.

• **start_moment_index** – The starting point of the search.
Returns The index of the next moment that touches each qubit. If there is no such moment, the next moment is specified as the number of moments in the circuit. Equivalently, can be characterized as one plus the index of the last moment after start_moment_index (inclusive) that does not act on a given qubit.

cirq.Circuit.operation_at

Circuit.operation_at(qubit: cirq.ops.raw_types.Qid, moment_index: int) → Optional[cirq.ops.raw_types.Operation]
Finds the operation on a qubit within a moment, if any.

Parameters

• qubit – The qubit to check for an operation on.
• moment_index – The index of the moment to check for an operation within. Allowed to be beyond the end of the circuit.

Returns None if there is no operation on the qubit at the given moment, or else the operation.

cirq.Circuit.prev_moment_operating_on

Circuit.prev_moment_operating_on(qubits: Sequence[cirq.Qid], end_moment_index: Optional[int] = None, max_distance: Optional[int] = None) → Optional[int]
Finds the index of the next moment that touches the given qubits.

Parameters

• qubits – We’re looking for operations affecting any of these qubits.
• end_moment_index – The moment index just after the starting point of the reverse search. Defaults to the length of the list of moments.
• max_distance – The number of moments (starting just before from the end index and moving backward) to check. Defaults to no limit.

Returns None if there is no matching moment, otherwise the index of the latest matching moment.

Raises ValueError – negative max_distance.

cirq.Circuit.qid_shape

Circuit.qid_shape(qubit_order: Union[cirq.ops.qubit_order.QubitOrder, Iterable[cirq.ops.raw_types.Qid]] = <cirq.ops.qubit_order.QubitOrder object>) → Tuple[int, ...]

cirq.Circuit.reachable_frontier_from

Circuit.reachable_frontier_from(start_frontier: Dict[cirq.Qid, int], *, is_blocker: Callable[cirq.Operation, bool] = <function Circuit.<lambda>>) → Dict[cirq.ops.raw_types.Qid, int]
Determines how far can be reached into a circuit under certain rules.
The location \( L = (\text{qubit}, \text{moment\_index}) \) is reachable if and only if:

\[
\text{a) There is not a blocking operation covering } L. \\
\text{AND} \\
\begin{aligned}
\text{b1) qubit is in start frontier and } & \text{moment\_index} = \\
& \max(\text{start\_frontier}[\text{qubit}], 0). \\
\text{OR} \\
\text{b2) There is no operation at } L & \text{ and } \text{prev}(L) = (\text{qubit}, \\
& \text{moment\_index-1}) \text{ is reachable.} \\
\text{OR} \\
\text{b3) There is an (non-blocking) operation } P & \text{ covering } L \text{ such that} \\
& (q', \text{moment\_index - 1}) \text{ is reachable for every } q' \text{ on which } P \\
& \text{acts.}
\end{aligned}
\]

An operation in moment \text{moment\_index} is blocking if

\[
\text{a) } '\text{is\_blocker}' \text{ returns a truthy value.} \\
\text{OR} \\
\text{b) The operation acts on a qubit not in start\_frontier.} \\
\text{OR} \\
\text{c) The operation acts on a qubit } q \text{ such that} \\
& \text{start\_frontier}[q] > \text{moment\_index.}
\]

In other words, the reachable region extends forward through time along each qubit in start\_frontier until it hits a blocking operation. Any location involving a qubit not in start\_frontier is unreachable.

For each qubit \( q \) in start\_frontier, the reachable locations will correspond to a contiguous range starting at start\_frontier\[q\] and ending just before some index \text{end\_q}. The result of this method is a dictionary, and that dictionary maps each qubit \( q \) to its \text{end\_q}.

**Examples**

If start\_frontier is 
\{ 
\text{cirq.LineQubit(0): 6,} 
\text{cirq.LineQubit(1): 2,} 
\text{cirq.LineQubit(2): 2,} 
\} 
then the reachable wire locations in the following circuit are highlighted with ‘‘ characters:
And the computed end_frontier is {
    cirq.LineQubit(0): 11,
    cirq.LineQubit(1): 9,
    cirq.LineQubit(2): 6,
    cirq.LineQubit(3): 0,
}

Note that the frontier indices (shown above the circuit) are best thought of (and shown) as happening between moment indices.

If we specify a blocker as follows:

```python
is_blocker=lambda: op == cirq.CZ(cirq.LineQubit(1), cirq.LineQubit(2))
```

and use this start_frontier:

```python
{
    cirq.LineQubit(0): 0,
    cirq.LineQubit(1): 0,
    cirq.LineQubit(2): 0,
    cirq.LineQubit(3): 0,
}
```

Then this is the reachable area:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13
0: -H@--@---H---
|   |   |
1: @-H@--@---H---
|   |   |
2: @H@--@---H---
|   |   |
3: ---H@--@---H---
```
and the computed end_frontier is:

```python
{
    cirq.LineQubit(0): 11,
    cirq.LineQubit(1): 3,
    cirq.LineQubit(2): 3,
    cirq.LineQubit(3): 5,
}
```

**Parameters**

- **start_frontier** – A starting set of reachable locations.
- **is_blocker** – A predicate that determines if operations block reachability. Any location covered by an operation that causes *is_blocker* to return True is considered to be an unreachable location.

**Returns**

An end_frontier dictionary, containing an end index for each qubit *q* mapped to a start index by the given *start_frontier* dictionary.

To determine if a location (q, i) was reachable, you can use this expression:

```
q in start_frontier and start_frontier[q] <= i < end_frontier[q]
```

where *i* is the moment index, *q* is the qubit, and end_frontier is the result of this method.

cirq.Circuit.save_qasm

```python
```

Save a QASM file equivalent to the circuit.

**Parameters**

- **file_path** – The location of the file where the qasm will be written.
- **header** – A multi-line string that is placed in a comment at the top of the QASM. Defaults to a cirq version specifier.
- **precision** – Number of digits to use when representing numbers.
- **qubit_order** – Determines how qubits are ordered in the QASM register.

cirq.Circuit.to_qasm

```python
```

Returns QASM equivalent to the circuit.

**Parameters**

- **header** – A multi-line string that is placed in a comment at the top of the QASM. Defaults to a cirq version specifier.
- **precision** – Number of digits to use when representing numbers.
• qubit_order – Determines how qubits are ordered in the QASM register.

cirq.Circuit.to_text_diagram


Returns text containing a diagram describing the circuit.

Parameters

• use_unicode_characters – Determines if unicode characters are allowed (as opposed to ascii-only diagrams).

• transpose – Arranges qubit wires vertically instead of horizontally.

• precision – Number of digits to display in text diagram

• qubit_order – Determines how qubits are ordered in the diagram.

Returns The text diagram.

cirq.Circuit.to_text_diagram_drawer


Returns a TextDiagramDrawer with the circuit drawn into it.

Parameters

• use_unicode_characters – Determines if unicode characters are allowed (as opposed to ascii-only diagrams).

• qubit_namer – Names qubits in diagram. Defaults to str.

• transpose – Arranges qubit wires vertically instead of horizontally.

• precision – Number of digits to use when representing numbers.

• qubit_order – Determines how qubits are ordered in the diagram.

• get_circuit_diagram_info – Gets circuit diagram info. Defaults to protocol with fallback.

Returns The TextDiagramDrawer instance.
cirq.Circuit.to_unitary_matrix

Circuit.to_unitary_matrix(qubit_order: cirq.QubitOrderOrList = <cirq.ops.qubit_order.QubitOrder object>, qubits_that_should_be_present: Iterable[cirq.Qid] = (), ignore_terminal_measurements: bool = True, dtype: Type[numpy.number] = <class 'numpy.complex128'>) \rightarrow numpy.ndarray

THIS FUNCTION IS DEPRECATED.
IT WILL BE REMOVED IN cirq v0.7.0.

Use Circuit.unitary() instead.

Converts the circuit into a unitary matrix, if possible.

Returns the same result as `cirq.unitary`, but provides more options.

Args:
qubit_order: Determines how qubits are ordered when passing matrices into np.kron.
qubits_that_should_be_present: Qubits that may or may not appear in operations within the circuit, but that should be included regardless when generating the matrix.
ignore_terminal_measurements: When set, measurements at the end of the circuit are ignored instead of causing the method to fail.
dtype: The numpy dtype for the returned unitary. Defaults to np.complex128. Specifying np.complex64 will run faster at the cost of precision. `dtype` must be a complex np.dtype, unless all operations in the circuit have unitary matrices with exclusively real coefficients (e.g. an H + TOFFOLI circuit).

Returns:
A (possibly gigantic) 2d numpy array corresponding to a matrix equivalent to the circuit's effect on a quantum state.

Raises:
ValueError: The circuit contains measurement gates that are not ignored.
TypeError: The circuit contains gates that don't have a known unitary matrix, e.g. gates parameterized by a Symbol.

---

cirq.Circuit.transform_qubits


Returns the same circuit, but with different qubits.

Note that this method does essentially the same thing as cirq.Circuit.with_device. It is included regardless because there are also transform_qubits methods on cirq.Operation and cirq.Moment.

Parameters
• **func** – The function to use to turn each current qubit into a desired new qubit.

• **new_device** – The device to use for the new circuit, if different. If this is not set, the new device defaults to the current device.

Returns

The receiving circuit but with qubits transformed by the given function, and with an updated device (if specified).

cirq.Circuit.unitary

```python
```

Converts the circuit into a unitary matrix, if possible.

Returns the same result as `cirq.unitary`, but provides more options.

Parameters

• **qubit_order** – Determines how qubits are ordered when passing matrices into np.kron.

• **qubits_that_should_be_present** – Qubits that may or may not appear in operations within the circuit, but that should be included regardless when generating the matrix.

• **ignore_terminal_measurements** – When set, measurements at the end of the circuit are ignored instead of causing the method to fail.

• **dtype** – The numpy dtype for the returned unitary. Defaults to np.complex128. Specifying np.complex64 will run faster at the cost of precision. `dtype` must be a complex np.dtype, unless all operations in the circuit have unitary matrices with exclusively real coefficients (e.g. an H + TOFFOLI circuit).

Returns A (possibly gigantic) 2d numpy array corresponding to a matrix equivalent to the circuit’s effect on a quantum state.

Raises

• **ValueError** – The circuit contains measurement gates that are not ignored.

• **TypeError** – The circuit contains gates that don’t have a known unitary matrix, e.g. gates parameterized by a Symbol.

cirq.Circuit.with_device

```python
```

Maps the current circuit onto a new device, and validates.

Parameters

• **new_device** – The new device that the circuit should be on.

• **qubit_mapping** – How to translate qubits from the old device into qubits on the new device.

Returns The translated circuit.
cirq.Circuit.with_noise


Make a noisy version of the circuit.

**Parameters**

`noise` – The noise model to use. This describes the kind of noise to add to the circuit.

**Returns**

A new circuit with the same moment structure but with new moments inserted where needed when more than one noisy operation is generated for an input operation. Emptied moments are removed.

**Attributes**

device
cirq.Circuit.device

property Circuit.device

cirq.Circuit.moments

property Circuit.moments

cirq.CircuitDag


A representation of a Circuit as a directed acyclic graph.

Nodes of the graph are instances of Unique containing each operation of a circuit.

Edges of the graph are tuples of nodes. Each edge specifies a required application order between two operations. The first must be applied before the second.

The graph is maximalist (transitive completion).

```python
__init__(can_reorder: Callable[[cirq.Operation, cirq.Operation], bool] = <function _disjoint_qubits>, incoming_graph_data: Any = None, device: cirq.devices.device.Device = cirq.UNCONSTRAINED_DEVICE) → None
```

Initializes a CircuitDag.

**Parameters**
- **can_reorder** – A predicate that determines if two operations may be reordered. Graph edges are created for pairs of operations where this returns False.

The default predicate allows reordering only when the operations don’t share common qubits.

- **incoming_graph_data** – Data in initialize the graph. This can be any value supported by networkx.DiGraph() e.g. an edge list or another graph.

- **device** – Hardware that the circuit should be able to run on.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_cycle(nodes, **attr)</code></td>
<td>Add a cycle</td>
</tr>
<tr>
<td><code>add_edge(u_of_edge, v_of_edge, **attr)</code></td>
<td>Add an edge between u and v.</td>
</tr>
<tr>
<td><code>add_edges_from(ebunch_to_add, **attr)</code></td>
<td>Add all the edges in ebunch_to_add.</td>
</tr>
<tr>
<td><code>add_node(node_for_adding, **attr)</code></td>
<td>Add a single node node_for_adding and update node attributes.</td>
</tr>
<tr>
<td><code>add_nodes_from(nodes_for_adding, **attr)</code></td>
<td>Add multiple nodes.</td>
</tr>
<tr>
<td><code>add_path(nodes, **attr)</code></td>
<td>Add multiple edges between nodes.</td>
</tr>
<tr>
<td><code>add_star(nodes, **attr)</code></td>
<td>Add a single node and update node attributes.</td>
</tr>
<tr>
<td><code>add_weighted_edges_from(ebunch_to_add, weight)</code></td>
<td>Add weighted edges in ebunch_to_add with specified weight attr</td>
</tr>
<tr>
<td><code>adjacency()</code></td>
<td>Returns an iterator over (node, adjacency dict) tuples for all nodes.</td>
</tr>
<tr>
<td><code>all_operations()</code></td>
<td></td>
</tr>
<tr>
<td><code>all_qubits()</code></td>
<td></td>
</tr>
<tr>
<td><code>append(op)</code></td>
<td></td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>Remove all nodes and edges from the graph.</td>
</tr>
<tr>
<td><code>copy(as_view)</code></td>
<td>Returns a copy of the graph.</td>
</tr>
<tr>
<td><code>disjoint_qubits(op1, op2)</code></td>
<td>Returns true only if the operations have qubits in common.</td>
</tr>
<tr>
<td><code>edge_subgraph(edges)</code></td>
<td>Returns the subgraph induced by the specified edges.</td>
</tr>
<tr>
<td><code>findall_nodes_until_blocked(is_blocker)</code></td>
<td>Finds all nodes before blocking ones.</td>
</tr>
<tr>
<td><code>fresh_copy()</code></td>
<td></td>
</tr>
<tr>
<td><code>from_circuit(circuit[, can_reorder])</code></td>
<td></td>
</tr>
<tr>
<td><code>from_ops(*operations[, can_reorder, device])</code></td>
<td></td>
</tr>
<tr>
<td><code>get_edge_data(u, v[, default])</code></td>
<td>Returns the attribute dictionary associated with edge (u, v).</td>
</tr>
<tr>
<td><code>has_edge(u, v)</code></td>
<td>Returns True if the edge (u, v) is in the graph.</td>
</tr>
<tr>
<td><code>has_node(n)</code></td>
<td>Returns True if the graph contains the node n.</td>
</tr>
<tr>
<td><code>has_predecessor(u, v)</code></td>
<td>Returns True if node u has predecessor v.</td>
</tr>
<tr>
<td><code>has_successor(u, v)</code></td>
<td>Returns True if node u has successor v.</td>
</tr>
<tr>
<td><code>is_directed()</code></td>
<td>Returns True if graph is directed, False otherwise.</td>
</tr>
<tr>
<td><code>is_multigraph()</code></td>
<td>Returns True if graph is a multigraph, False otherwise.</td>
</tr>
<tr>
<td><code>make_node(op)</code></td>
<td></td>
</tr>
<tr>
<td><code>nbunch_iter([nbunch])</code></td>
<td>Returns an iterator over nodes contained in nbunch that are also in the graph.</td>
</tr>
<tr>
<td><code>neighbors(n)</code></td>
<td>Returns an iterator over successor nodes of n.</td>
</tr>
<tr>
<td><code>nodes_with_selfloops()</code></td>
<td></td>
</tr>
<tr>
<td><code>number_of_edges([u, v])</code></td>
<td>Returns the number of edges between two nodes.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 96 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number_of_nodes()</td>
<td>Returns the number of nodes in the graph.</td>
</tr>
<tr>
<td>number_of_selfloops()</td>
<td></td>
</tr>
<tr>
<td>order()</td>
<td>Returns the number of nodes in the graph.</td>
</tr>
<tr>
<td>ordered_nodes()</td>
<td></td>
</tr>
<tr>
<td>predecessors(n)</td>
<td>Returns an iterator over predecessor nodes of n.</td>
</tr>
<tr>
<td>remove_edge(u, v)</td>
<td>Remove the edge between u and v.</td>
</tr>
<tr>
<td>remove_edges_from(ebunch)</td>
<td>Remove all edges specified in ebunch.</td>
</tr>
<tr>
<td>remove_node(n)</td>
<td>Remove node n.</td>
</tr>
<tr>
<td>remove_nodes_from(nodes)</td>
<td>Remove multiple nodes.</td>
</tr>
<tr>
<td>reverse([copy])</td>
<td>Returns the reverse of the graph.</td>
</tr>
<tr>
<td>selfloop_edges([data, keys, default])</td>
<td></td>
</tr>
<tr>
<td>size([weight])</td>
<td>Returns the number of edges or total of all edge weights.</td>
</tr>
<tr>
<td>subgraph(nodes)</td>
<td>Returns a SubGraph view of the subgraph induced on nodes.</td>
</tr>
<tr>
<td>successors(n)</td>
<td>Returns an iterator over successor nodes of n.</td>
</tr>
<tr>
<td>to_circuit()</td>
<td></td>
</tr>
<tr>
<td>to_directed([as_view])</td>
<td>Returns a directed representation of the graph.</td>
</tr>
<tr>
<td>to_directed_class()</td>
<td>Returns the class to use for empty directed copies.</td>
</tr>
<tr>
<td>to_undirected([reciprocal, as_view])</td>
<td>Returns an undirected representation of the digraph.</td>
</tr>
<tr>
<td>to_undirected_class()</td>
<td>Returns the class to use for empty undirected copies.</td>
</tr>
<tr>
<td>update([edges, nodes])</td>
<td>Update the graph using nodes/edges/graphs as input.</td>
</tr>
</tbody>
</table>

**cirq.CircuitDag.add_cycle**

CircuitDag.add_cycle(nodes, **attr)**

**cirq.CircuitDag.add_edge**

CircuitDag.add_edge(u_of_edge, v_of_edge, **attr)**

Add an edge between u and v.

The nodes u and v will be automatically added if they are not already in the graph.

Edge attributes can be specified with keywords or by directly accessing the edge’s attribute dictionary. See examples below.

**Parameters**

- **v(u,)** – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.
- **attr(keyword arguments, optional)** – Edge data (or labels or objects) can be assigned using keyword arguments.

**See also:**

- **add_edges_from()** add a collection of edges

**Notes**

Adding an edge that already exists updates the edge data.
Many NetworkX algorithms designed for weighted graphs use an edge attribute (by default `weight`) to hold a numerical value.

**Examples**

The following all add the edge e=(1, 2) to graph G:

```python
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> e = (1, 2)
>>> G.add_edge(1, 2)  # explicit two-node form
>>> G.add_edge(*e)   # single edge as tuple of two nodes
>>> G.add_edges_from([(1, 2)])  # add edges from iterable container
```

Associate data to edges using keywords:

```python
>>> G.add_edge(1, 2, weight=3)
>>> G.add_edge(1, 3, weight=7, capacity=15, length=342.7)
```

For non-string attribute keys, use subscript notation.

```python
>>> G.add_edge(1, 2)
>>> G[1][2].update({0: 5})
>>> G.edges[1, 2].update({0: 5})
```

cirq.CircuitDag.add_edges_from

CircuitDag.add_edges_from(ebunch_to_add, **attr)

Add all the edges in ebunch_to_add.

**Parameters**

- `ebunch_to_add (container of edges)` – Each edge given in the container will be added to the graph. The edges must be given as 2-tuples (u, v) or 3-tuples (u, v, d) where d is a dictionary containing edge data.
- `attr (keyword arguments, optional)` – Edge data (or labels or objects) can be assigned using keyword arguments.

See also:

- `add_edge()` add a single edge
- `add_weighted_edges_from()` convenient way to add weighted edges

**Notes**

Adding the same edge twice has no effect but any edge data will be updated when each duplicate edge is added.

Edge attributes specified in an ebunch take precedence over attributes specified via keyword arguments.

**Examples**
```python
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edges_from([(0, 1), (1, 2)])  # using a list of edge tuples
>>> e = zip(range(0, 3), range(1, 4))
>>> G.add_edges_from(e)  # Add the path graph 0-1-2-3

Associate data to edges
```}

```python
>>> G.add_edges_from([(1, 2), (2, 3)], weight=3)
>>> G.add_edges_from([(3, 4), (1, 4)], label='WN2898')
```

**CircuitDag.add_node**

CircuitDag.add_node(node_for_adding, **attr)**

Add a single node node_for_adding and update node attributes.

**Parameters**

- node_for_adding (node) – A node can be any hashable Python object except None.
- attr (keyword arguments, optional) – Set or change node attributes using key=value.

**Examples**

```python
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> K3 = nx.Graph([(0, 1), (1, 2), (2, 0)])
>>> G.add_node(K3)
>>> G.number_of_nodes()
3
```

Use keywords set/change node attributes:

```python
>>> G.add_node(1, size=10)
>>> G.add_node(3, weight=0.4, UTM=('13S', 382871, 3972649))
```

**Notes**

A hashable object is one that can be used as a key in a Python dictionary. This includes strings, numbers, tuples of strings and numbers, etc.

On many platforms hashable items also include mutables such as NetworkX Graphs, though one should be careful that the hash doesn’t change on mutables.

**CircuitDag.add_nodes_from**

CircuitDag.add_nodes_from(nodes_for_adding, **attr)**

Add multiple nodes.
Parameters

- **nodes_for_adding** *(iterable container)* – A container of nodes (list, dict, set, etc.). OR A container of (node, attribute dict) tuples. Node attributes are updated using the attribute dict.

- **attr** *(keyword arguments, optional (default= no attributes))* – Update attributes for all nodes in nodes. Node attributes specified in nodes as a tuple take precedence over attributes specified via keyword arguments.

See also:
add_node()

Examples

```python
g = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
g.add_nodes_from('Hello')
k3 = nx.Graph([(0, 1), (1, 2), (2, 0)])
g.add_nodes_from(k3)
sorted(g.nodes(), key=str)
[0, 1, 2, 'H', 'e', 'l', 'o']
```

Use keywords to update specific node attributes for every node.

```python
g.add_nodes_from([(1, 2), size=10])
g.add_nodes_from([(3, 4), weight=0.4])
```

Use (node, attrdict) tuples to update attributes for specific nodes.

```python
g.add_nodes_from([(1, dict(size=11)), (2, {'color':'blue'})])
g.nodes[1]['size']
11
h = nx.Graph()
h.add_nodes_from(g.nodes(data=True))
h.nodes[1]['size']
11
```

```

cirq.CircuitDag.add_path
circuitDag.add_path(nodes, **attr)

cirq.CircuitDag.add_star
circuitDag.add_star(nodes, **attr)

cirq.CircuitDag.add_weighted_edges_from
circuitDag.add_weighted_edges_from(ebunch_to_add, weight='weight', **attr)
Add weighted edges in ebunch_to_add with specified weight attr

Parameters

```
• **ebunch_to_add** (*container of edges*) – Each edge given in the list or container will be added to the graph. The edges must be given as 3-tuples \((u, v, w)\) where \(w\) is a number.

• **weight** (*string, optional (default= 'weight')*) – The attribute name for the edge weights to be added.

• **attr** (*keyword arguments, optional (default= no attributes)*) – Edge attributes to add/update for all edges.

**See also:**

- **add_edge()** add a single edge
- **add_edges_from()** add multiple edges

**Notes**

Adding the same edge twice for Graph/DiGraph simply updates the edge data. For MultiGraph/MultiDiGraph, duplicate edges are stored.

**Examples**

```python
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_weighted_edges_from([(0, 1, 3.0), (1, 2, 7.5)])
```

**cirq.CircuitDag.adjacency**

**CircuitDag.adjacency()**

Returns an iterator over (node, adjacency dict) tuples for all nodes.

For directed graphs, only outgoing neighbors/adjacencies are included.

**Returns** **adj_iter** – An iterator over (node, adjacency dictionary) for all nodes in the graph.

**Return type** iterator

**Examples**

```python
>>> G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> [(n, ndict) for n, ndict in G.adjacency()]
[(0, {1: {}}), (1, {0: {}, 2: {}}), (2, {1: {}, 3: {}}), (3, {2: {}})]
```

**cirq.CircuitDag.all_operations**

**CircuitDag.all_operations()** \to**Iterator[cirq.ops.raw_types.Operation]**

**cirq.CircuitDag.all_qubits**

**CircuitDag.all_qubits()**
### cirq.CircuitDag.append

**CircuitDag.append**(op: cirq.Operation) → None

### cirq.CircuitDag.clear

**CircuitDag.clear()**

Remove all nodes and edges from the graph. This also removes the name, and all graph, node, and edge attributes.

**Examples**

```python
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.clear()
>>> list(G.nodes)
[]
>>> list(G.edges)
[]
```

### cirq.CircuitDag.copy

**CircuitDag.copy**(as_view=False)

Returns a copy of the graph.

The copy method by default returns an independent shallow copy of the graph and attributes. That is, if an attribute is a container, that container is shared by the original and the copy. Use Python’s `copy.deepcopy` for new containers.

If `as_view` is True then a view is returned instead of a copy.

**Notes**

All copies reproduce the graph structure, but data attributes may be handled in different ways. There are four types of copies of a graph that people might want.

Deepcopy – A “deepcopy” copies the graph structure as well as all data attributes and any objects they might contain. The entire graph object is new so that changes in the copy do not affect the original object. (see Python’s `copy.deepcopy`)

Data Reference (Shallow) – For a shallow copy the graph structure is copied but the edge, node and graph attribute dicts are references to those in the original graph. This saves time and memory but could cause confusion if you change an attribute in one graph and it changes the attribute in the other. NetworkX does not provide this level of shallow copy.

Independent Shallow – This copy creates new independent attribute dicts and then does a shallow copy of the attributes. That is, any attributes that are containers are shared between the new graph and the original. This is exactly what `dict.copy()` provides. You can obtain this style copy using:

```python
>>> G = nx.path_graph(5)
>>> H = G.copy()
>>> H = G.copy(as_view=False)
```
Fresh Data – For fresh data, the graph structure is copied while new empty data attribute dicts are created. The resulting graph is independent of the original and it has no edge, node or graph attributes. Fresh copies are not enabled. Instead use:

```python
>>> H = G.__class__(G)
```  

View – Inspired by dict-views, graph-views act like read-only versions of the original graph, providing a copy of the original structure without requiring any memory for copying the information.

See the Python copy module for more information on shallow and deep copies, https://docs.python.org/2/library/copy.html.

Parameters

```python
Parameters as_view (bool, optional (default=False)) – If True, the returned graph-view provides a read-only view of the original graph without actually copying any data.
```  

Returns G – A copy of the graph.

See also:

to_directed() return a directed copy of the graph.

Examples

```python
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> H = G.copy()
```  

cirq.CircuitDag.disjoint_qubits

```python
static CircuitDag.disjoint_qubits (op1: cirq.Operation, op2: cirq.Operation) → bool
```

Returns true only if the operations have qubits in common.

cirq.CircuitDag.edge_subgraph

```python
CircuitDag.edge_subgraph (edges)
```

Returns the subgraph induced by the specified edges.

The induced subgraph contains each edge in `edges` and each node incident to any one of those edges.

Parameters

```python
Parameters edges (iterable) – An iterable of edges in this graph.
```

Returns G – An edge-induced subgraph of this graph with the same edge attributes.

Return type Graph
Notes

The graph, edge, and node attributes in the returned subgraph view are references to the corresponding attributes in the original graph. The view is read-only.

To create a full graph version of the subgraph with its own copy of the edge or node attributes, use:

```python
>>> G.edge_subgraph(edges).copy()
```

Examples

```python
>>> G = nx.path_graph(5)
>>> H = G.edge_subgraph([(0, 1), (3, 4)])
>>> list(H.nodes)
[0, 1, 3, 4]
>>> list(H.edges)
[(0, 1), (3, 4)]
```

cirq.CircuitDag.findall_nodes_until_blocked


Finds all nodes before blocking ones.

Parameters

- `is_blocker` – The predicate that indicates whether or not an
  operation is blocking.

```
cirq.CircuitDag.fresh_copy
```

CircuitDag.fresh_copy()

cirq.CircuitDag.from_circuit


cirq.CircuitDag.from_ops

**cirq.CircuitDag.get_edge_data**

CircuitDag.get_edge_data(u, v, default=None)
Returns the attribute dictionary associated with edge (u, v).

This is identical to G[u][v] except the default is returned instead of an exception if the edge doesn’t exist.

**Parameters**

- v (u, )
- default (any Python object (default=None)) – Value to return if the edge (u, v) is not found.

**Returns** edge_dict – The edge attribute dictionary.

**Return type** dictionary

**Examples**

```python
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G[0][1]
{}
```

Warning: Assigning to G[u][v] is not permitted. But it is safe to assign attributes G[u][v][‘foo’]

```python
>>> G[0][1][‘weight’] = 7
>>> G[0][1][‘weight’]
7
>>> G[1][0][‘weight’]
7
```

```python
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.get_edge_data(0, 1)  # default edge data is {}
{}
>>> e = (0, 1)
>>> G.get_edge_data(*e)  # tuple form
{}
>>> G.get_edge_data(‘a’, ‘b’, default=0)  # edge not in graph, return 0
0
```

**cirq.CircuitDag.has_edge**

CircuitDag.has_edge(u, v)
Returns True if the edge (u, v) is in the graph.

This is the same as v in G[u] without KeyError exceptions.

**Parameters** v (u, ) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.

**Returns** edge_ind – True if edge is in the graph, False otherwise.

**Return type** bool
Examples

```python
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.has_edge(0, 1)  # using two nodes
True
>>> e = (0, 1)
>>> G.has_edge(*e)  # e is a 2-tuple (u, v)
True
>>> e = (0, 1, {'weight':7})
>>> G.has_edge(*e[:2])  # e is a 3-tuple (u, v, data_dictionary)
True
```

The following syntax are equivalent:

```python
>>> G.has_edge(0, 1)
True
>>> 1 in G[0]  # though this gives KeyError if 0 not in G
True
```

Cirq.CircuitDag.has_node

CircuitDag.**has_node**(n)

Returns True if the graph contains the node n.

Identical to *n in G*

**Parameters**

- **n** (node) –

**Examples**

```python
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.has_node(0)
True
```

It is more readable and simpler to use

```python
>>> 0 in G
True
```

Cirq.CircuitDag.has_predecessor

CircuitDag.**has_predecessor**(u, v)

Returns True if node u has predecessor v.

This is true if graph has the edge u<-v.

Cirq.CircuitDag.has_successor

CircuitDag.**has_successor**(u, v)

Returns True if node u has successor v.

This is true if graph has the edge u->v.
cirq.CircuitDag.is_directed

CircuitDag.is_directed()  
Returns True if graph is directed, False otherwise.

cirq.CircuitDag.is_multigraph

CircuitDag.is_multigraph()  
Returns True if graph is a multigraph, False otherwise.

cirq.CircuitDag.make_node

static CircuitDag.make_node(op: cirq.Operation) → cirq.circuits.circuit_dag.Unique

cirq.CircuitDag.nbunch_iter

CircuitDag.nbunch_iter(nbunch=None)  
Returns an iterator over nodes contained in nbunch that are also in the graph.  
The nodes in nbunch are checked for membership in the graph and if not are silently ignored.  
Parameters nbunch (single node, container, or all nodes (default=all nodes)) – The view will only report edges incident to these nodes.  
Returns niter – An iterator over nodes in nbunch that are also in the graph. If nbunch is None, iterate over all nodes in the graph.  
Return type iterator  
Raises NetworkXError – If nbunch is not a node or or sequence of nodes. If a node in nbunch is not hashable.  
See also:  
Graph.__iter__()  
Notes  
When nbunch is an iterator, the returned iterator yields values directly from nbunch, becoming exhausted when nbunch is exhausted.  
To test whether nbunch is a single node, one can use “if nbunch in self:”, even after processing with this routine.  
If nbunch is not a node or a (possibly empty) sequence/iterator or None, a NetworkXError is raised. Also, if any object in nbunch is not hashable, a NetworkXError is raised.

cirq.CircuitDag.neighbors

CircuitDag.neighbors(n)  
Returns an iterator over successor nodes of n.  
A successor of n is a node m such that there exists a directed edge from n to m.
Parameters \( n \) (node) – A node in the graph

Raises NetworkXError – If \( n \) is not in the graph.

See also:
predecessors()

Notes
neighbors() and successors() are the same.

cirq.CircuitDag.nodes_with_selfloops

CircuitDag.nodes_with_selfloops()

cirq.CircuitDag.number_of_edges

CircuitDag.number_of_edges (u=None, v=None)

Returns the number of edges between two nodes.

Parameters \( v \) (u,) – If \( u \) and \( v \) are specified, return the number of edges between \( u \) and \( v \).
Otherwise return the total number of all edges.

Returns nedges – The number of edges in the graph. If nodes \( u \) and \( v \) are specified return the number of edges between those nodes. If the graph is directed, this only returns the number of edges from \( u \) to \( v \).

Return type int

See also:
size()

Examples

For undirected graphs, this method counts the total number of edges in the graph:

```python
>>> G = nx.path_graph(4)
>>> G.number_of_edges()
3
```

If you specify two nodes, this counts the total number of edges joining the two nodes:

```python
>>> G.number_of_edges(0, 1)
1
```

For directed graphs, this method can count the total number of directed edges from \( u \) to \( v \):

```python
>>> G = nx.DiGraph()
>>> G.add_edge(0, 1)
>>> G.add_edge(1, 0)
>>> G.number_of_edges(0, 1)
1
```
**cirq.CircuitDag.number_of_nodes**

CircuitDag.\texttt{number\_of\_nodes}()

Returns the number of nodes in the graph.

\textbf{Returns} \texttt{nnodes} – The number of nodes in the graph.

\textbf{Return type} \texttt{int}

\textbf{See also:}

\texttt{order()}, \texttt{\_\_len\_\_}()

\textbf{Examples}

```python
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> len(G)
3
```

**cirq.CircuitDag.number_of_selfloops**

CircuitDag.\texttt{number\_of\_selfloops}()

**cirq.CircuitDag.order**

CircuitDag.\texttt{order}()

Returns the number of nodes in the graph.

\textbf{Returns} \texttt{nnodes} – The number of nodes in the graph.

\textbf{Return type} \texttt{int}

\textbf{See also:}

\texttt{number\_of\_nodes()}, \texttt{\_\_len\_\_}()

**cirq.CircuitDag.ordered_nodes**

CircuitDag.\texttt{ordered\_nodes}() \rightarrow \texttt{Iterator[cirq.circuits.circuit\_dag.Unique[cirq.ops.raw\_types.Operation]]}

**cirq.CircuitDag.predecessors**

CircuitDag.\texttt{predecessors}(n)

Returns an iterator over predecessor nodes of \texttt{n}.

\textbf{A predecessor of} \texttt{n} \textbf{is a node} \texttt{m} \textbf{such that there exists a directed edge from} \texttt{m} \textbf{to} \texttt{n}.

\textbf{Parameters} \texttt{n (node)} – A node in the graph

\textbf{Raises} NetworkXError – If \texttt{n} is not in the graph.

\textbf{See also:}

\texttt{successors}()
cirq.CircuitDag.remove_edge

CircuitDag.remove_edge(u, v)
Remove the edge between u and v.

Parameters v (u,) – Remove the edge between nodes u and v.

Raises NetworkXError – If there is not an edge between u and v.

See also:

remove_edges_from() remove a collection of edges

Examples

```python
>>> G = nx.Graph()   # or DiGraph, etc
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.remove_edge(0, 1)
>>> e = (1, 2)
>>> G.remove_edge(*e) # unpacks e from an edge tuple
>>> e = (2, 3, {'weight':7}) # an edge with attribute data
>>> G.remove_edge(*e[:2]) # select first part of edge tuple
```

cirq.CircuitDag.remove_edges_from

CircuitDag.remove_edges_from(ebunch)
Remove all edges specified in ebunch.

Parameters ebunch (list or container of edge tuples) – Each edge given in
the list or container will be removed from the graph. The edges can be:

- 2-tuples (u, v) edge between u and v.
- 3-tuples (u, v, k) where k is ignored.

See also:

remove_edge() remove a single edge

Notes

Will fail silently if an edge in ebunch is not in the graph.

Examples

```python
>>> G = nx.path_graph(4)   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> ebunch = [(1, 2), (2, 3)]
>>> G.remove_edges_from(ebunch)
```
cirq.CircuitDag.remove_node

CircuitDag.remove_node(n)
Remove node n.
Removes the node n and all adjacent edges. Attempting to remove a non-existent node will raise an exception.

Parameters n (node) – A node in the graph

Raises NetworkXError – If n is not in the graph.

See also:
remove_nodes_from()

Examples

```python
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> list(G.edges)
[(0, 1), (1, 2)]
>>> G.remove_node(1)
>>> list(G.edges)
[]
```

cirq.CircuitDag.remove_nodes_from

CircuitDag.remove_nodes_from(nodes)
Remove multiple nodes.

Parameters nodes (iterable container) – A container of nodes (list, dict, set, etc.). If a node in the container is not in the graph it is silently ignored.

See also:
remove_node()

Examples

```python
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> e = list(G.nodes)
>>> e
[0, 1, 2]
>>> G.remove_nodes_from(e)
>>> list(G.nodes)
[]
```

cirq.CircuitDag.reverse

CircuitDag.reverse(copy=True)
Returns the reverse of the graph.

The reverse is a graph with the same nodes and edges but with the directions of the edges reversed.
Parameters `copy` *(bool optional (default=True))* – If True, return a new DiGraph holding the reversed edges. If False, the reverse graph is created using a view of the original graph.

### CircuitDag.selfloop_edges

**CircuitDag.selfloop_edges** *(data=False, keys=False, default=None)*

### CircuitDag.size

**CircuitDag.size** *(weight=None)*

Returns the number of edges or total of all edge weights.

Parameters `weight` *(string or None, optional (default=None))* – The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1.

Returns

- `size` – The number of edges or (if weight keyword is provided) the total weight sum.

  If weight is None, returns an int. Otherwise a float (or more general numeric if the weights are more general).

Return type `numeric`

See also:

- `number_of_edges()`

#### Examples

```python
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.size()
3

>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge('a', 'b', weight=2)
>>> G.add_edge('b', 'c', weight=4)
>>> G.size()
2
>>> G.size(weight='weight')
6.0
```

### CircuitDag.subgraph

**CircuitDag.subgraph** *(nodes)*

Returns a SubGraph view of the subgraph induced on `nodes`.

The induced subgraph of the graph contains the nodes in `nodes` and the edges between those nodes.

Parameters `nodes` *(list, iterable)* – A container of nodes which will be iterated through once.
Returns G – A subgraph view of the graph. The graph structure cannot be changed but node/edge attributes can and are shared with the original graph.

Return type SubGraph View

Notes

The graph, edge and node attributes are shared with the original graph. Changes to the graph structure is ruled out by the view, but changes to attributes are reflected in the original graph.

To create a subgraph with its own copy of the edge/node attributes use: G.subgraph(nodes).copy()

For an inplace reduction of a graph to a subgraph you can remove nodes: G.remove_nodes_from([n for n in G if n not in set(nodes)])

Subgraph views are sometimes NOT what you want. In most cases where you want to do more than simply look at the induced edges, it makes more sense to just create the subgraph as its own graph with code like:

```python
# Create a subgraph SG based on a (possibly multigraph) G
SG = G.__class__()
SG.add_nodes_from((n, G.nodes[n]) for n in largest_wcc)
if SG.is_multigraph:
    SG.add_edges_from((n, nbr, key, d)
        for n, nbrs in G.adj.items() if n in largest_wcc
        for nbr, keydict in nbrs.items() if nbr in largest_wcc
        for key, d in keydict.items())
else:
    SG.add_edges_from((n, nbr, d)
        for n,_nbrs in G.adj.items() if n in largest_wcc
        for nbr, d in nbrs.items() if nbr in largest_wcc)
SG.graph.update(G.graph)
```

Examples

```python
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> H = G.subgraph([0, 1, 2])
>>> list(H.edges)
[(0, 1), (1, 2)]
```

cirq.CircuitDag.successors

CircuitDag.successors(n)

Returns an iterator over successor nodes of n.

A successor of n is a node m such that there exists a directed edge from n to m.

Parameters n (node) – A node in the graph

Raises NetworkXError – If n is not in the graph.

See also:

predecessors()
Notes

neighbors() and successors() are the same.

cirq.CircuitDag.to_circuit

CircuitDag.to_circuit() → cirq.circuits.circuit.Circuit

cirq.CircuitDag.to_directed

CircuitDag.to_directed(as_view=False)
Returns a directed representation of the graph.

Returns G – A directed graph with the same name, same nodes, and with each edge (u, v, data)
replaced by two directed edges (u, v, data) and (v, u, data).

Return type DiGraph

Notes

This returns a “deepcopy” of the edge, node, and graph attributes which attempts to completely copy all of
the data and references.

This is in contrast to the similar D=DiGraph(G) which returns a shallow copy of the data.

See the Python copy module for more information on shallow and deep copies, https://docs.python.org/2/
library/copy.html.

Warning: If you have subclassed Graph to use dict-like objects in the data structure, those changes do not
transfer to the DiGraph created by this method.

Examples

```python
>>> G = nx.Graph()  # or MultiGraph, etc
>>> G.add_edge(0, 1)
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1), (1, 0)]
```

If already directed, return a (deep) copy

```python
>>> G = nx.DiGraph()  # or MultiDiGraph, etc
>>> G.add_edge(0, 1)
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1)]
```

cirq.CircuitDag.to_directed_class

CircuitDag.to_directed_class()
Returns the class to use for empty directed copies.

If you subclass the base classes, use this to designate what directed class to use for to_directed() copies.
cirq.CircuitDag.to_undirected

CircuitDag.to_undirected(reciprocal=False, as_view=False)
Returns an undirected representation of the digraph.

Parameters
• reciprocal (bool (optional)) – If True only keep edges that appear in both di-
  rections in the original digraph.
• as_view (bool (optional, default=False)) – If True return an undirected
  view of the original directed graph.

Returns G – An undirected graph with the same name and nodes and with edge (u, v, data) if
either (u, v, data) or (v, u, data) is in the digraph. If both edges exist in digraph and their edge
data is different, only one edge is created with an arbitrary choice of which edge data to use.
You must check and correct for this manually if desired.

Return type Graph

See also: Graph(), copy(), add_edge(), add_edges_from()

Notes
If edges in both directions (u, v) and (v, u) exist in the graph, attributes for the new undirected edge will
be a combination of the attributes of the directed edges. The edge data is updated in the (arbitrary) order
that the edges are encountered. For more customized control of the edge attributes use add_edge().
This returns a “deepcopy” of the edge, node, and graph attributes which attempts to completely copy all of
the data and references.
This is in contrast to the similar G=DiGraph(D) which returns a shallow copy of the data.
See the Python copy module for more information on shallow and deep copies, https://docs.python.org/2/
library/copy.html.
Warning: If you have subclassed DiGraph to use dict-like objects in the data structure, those changes do
not transfer to the Graph created by this method.

Examples
>>> G = nx.path_graph(2)  # or MultiGraph, etc
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1), (1, 0)]
>>> G2 = H.to_undirected()
>>> list(G2.edges)
[(0, 1)]

CircuitDag.to_undirected_class

CircuitDag.to_undirected_class()
Returns the class to use for empty undirected copies.
If you subclass the base classes, use this to designate what directed class to use for to_directed() copies.
cirq.CircuitDag.update

CircuitDag.\texttt{update}(\texttt{edges=None, nodes=None})
Update the graph using nodes/edges/graphs as input.

Like \texttt{dict.update}, this method takes a graph as input, adding the graph’s nodes and edges to this graph. It can also take two inputs: edges and nodes. Finally it can take either nodes or edges. To specify only nodes the keyword \texttt{nodes} must be used.

The collections of edges and nodes are treated similarly to the \texttt{add_edges_from/add_nodes_from} methods. When iterated, they should yield 2-tuples (u, v) or 3-tuples (u, v, datadict).

**Parameters**

- **\texttt{edges} (Graph object, collection of edges, or None)** – The first parameter can be a graph or some edges. If it has attributes \texttt{nodes} and \texttt{edges}, then it is taken to be a Graph-like object and those attributes are used as collections of nodes and edges to be added to the graph. If the first parameter does not have those attributes, it is treated as a collection of edges and added to the graph. If the first argument is \texttt{None}, no edges are added.

- **\texttt{nodes} (collection of nodes, or None)** – The second parameter is treated as a collection of nodes to be added to the graph unless it is \texttt{None}. If \texttt{edges is None} and \texttt{nodes} is \texttt{None} an exception is raised. If the first parameter is a Graph, then \texttt{nodes} is ignored.

**Examples**

```python
>>> G = nx.path_graph(5)
>>> G.update(nx.complete_graph(range(4,10)))
>>> from itertools import combinations
>>> edges = {u, v, {'power': u * v})
... for u, v in combinations(range(10, 20), 2)
... if u * v < 225)
>>> nodes = [1000] # for singleton, use a container
>>> G.update(edges=edges, nodes=nodes)
```

**Notes**

It you want to update the graph using an adjacency structure it is straightforward to obtain the edges/nodes from adjacency. The following examples provide common cases, your adjacency may be slightly different and require tweaks of these examples.

```python
>>> # dict-of-set/list/tuple
>>> adj = {1: {2, 3}, 2: {1, 3}, 3: {1, 2}}
>>> e = [(u, v) for u, nbrs in adj.items() for v in nbrs]
>>> G.update(edges=e, nodes=adj)
```

```python
>>> DG = nx.DiGraph()
>>> # dict-of-dict-of-attribute
>>> adj = {1: {2: 1.3, 3: 0.7}, 2: {1: 1.4}, 3: {1: 0.7}}
>>> e = [(u, v, {'weight': d}) for u, nbrs in adj.items() for v in nbrs.items()
... for v, d in nbrs.items()]
>>> DG.update(edges=e, nodes=adj)
```
>>> # dict-of-dict-of-dict
>>> adj = {1: {2: {'weight': 1.3}, 3: {'color': 0.7, 'weight': 1.2}}
... for u, v in adj.items()}
>>> e = [(u, v, {'weight': d}) for u, v, d in adj.items()]
>>> DG.update(edges=e, nodes=adj)

>>> # predecessor adjacency (dict-of-set)
>>> pred = {1: {2, 3}, 2: {3}, 3: {3}}
>>> e = [(v, u) for u, v, d in pred.items() for v in v]

>>> # MultiGraph dict-of-dict-of-dict-of-attribute
>>> MDG = nx.MultiDiGraph()
>>> adj = {1: {2: {0: {'weight': 1.3}, 1: {'weight': 1.2}}},
... 3: {2: {0: {'weight': 0.7}}}}
>>> e = [(u, v, ekey, d) for u, v, ekey, d in adj.items()]
>>> MDG.update(edges=e)

See also:

- `add_edges_from()` add multiple edges to a graph
- `add_nodes_from()` add multiple nodes to a graph

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>adj</code></td>
<td>Graph adjacency object holding the neighbors of each node.</td>
</tr>
<tr>
<td><code>degree</code></td>
<td>A DegreeView for the Graph as G.degree or G.degree().</td>
</tr>
<tr>
<td><code>edges</code></td>
<td>An OutEdgeView of the DiGraph as G.edges or G.edges().</td>
</tr>
<tr>
<td><code>in_degree</code></td>
<td>An InDegreeView for (node, in_degree) or in_degree for single node.</td>
</tr>
<tr>
<td><code>in_edges</code></td>
<td>An InEdgeView of the Graph as G.in_edges or G.in_edges().</td>
</tr>
<tr>
<td><code>name</code></td>
<td>String identifier of the graph.</td>
</tr>
<tr>
<td><code>node</code></td>
<td>A NodeView of the Graph as G.nodes or G.nodes().</td>
</tr>
<tr>
<td><code>nodes</code></td>
<td>A NodeView of the Graph as G.nodes or G.nodes().</td>
</tr>
<tr>
<td><code>out_degree</code></td>
<td>An OutDegreeView for (node, out_degree)</td>
</tr>
<tr>
<td><code>out_edges</code></td>
<td>An OutEdgeView of the DiGraph as G.edges or G.edges().</td>
</tr>
<tr>
<td><code>pred</code></td>
<td>Graph adjacency object holding the predecessors of each node.</td>
</tr>
<tr>
<td><code>succ</code></td>
<td>Graph adjacency object holding the successors of each node.</td>
</tr>
</tbody>
</table>

**cirq.CircuitDag.adj**

- Property `CircuitDag.adj` Graph adjacency object holding the neighbors of each node.
This object is a read-only dict-like structure with node keys and neighbor-dict values. The neighbor-dict is keyed by neighbor to the edge-data-dict. So \( G.adj[3][2]['color'] = 'blue' \) sets the color of the edge \((3, 2)\) to "blue".

Iterating over \( G.adj \) behaves like a dict. Useful idioms include

```python
for nbr, datadict in G.adj[n].items():
```

The neighbor information is also provided by subscripting the graph.

So

```python
for nbr, foovalue in G[node].data('foo', default=1): works.
```

For directed graphs, \( G.adj \) holds outgoing (successor) info.

```python
cirq.CircuitDag.degree
```

**property** CircuitDag.degree

A DegreeView for the Graph as \( G.degree \) or \( G.degree() \).

The node degree is the number of edges adjacent to the node.

The weighted node degree is the sum of the edge weights for edges incident to that node.

This object provides an iterator for (node, degree) as well as lookup for the degree for a single node.

**Parameters**

- **nbunch** *(single node, container, or all nodes (default= all nodes))* – The view will only report edges incident to these nodes.

- **weight** *(string or None, optional (default=None))* – The name of an edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1. The degree is the sum of the edge weights adjacent to the node.

**Returns**

- **If a single node is requested**

- **deg** *(int)* – Degree of the node

- **OR if multiple nodes are requested**

- **nd_iter** *(iterator)* – The iterator returns two-tuples of (node, degree).

**See also:**

- in_degree
- out_degree
Examples

```python
>>> G = nx.DiGraph()  # or MultiDiGraph
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.degree(0)  # node 0 with degree 1
1
>>> list(G.degree([0, 1, 2]))
[(0, 1), (1, 2), (2, 2)]
```

cirq.CircuitDag.edges

**property** CircuitDag.edges

An OutEdgeView of the DiGraph as G.edges or G.edges().

```python
defaults(self, nbunch=None, data=False, default=None)
```

The OutEdgeView provides set-like operations on the edge-tuples as well as edge attribute lookup. When called, it also provides an EdgeDataView object which allows control of access to edge attributes (but does not provide set-like operations).

Hence, `G.edges[u, v]['color']` provides the value of the color attribute for edge `(u, v)` while

```python
for (u, v, c) in G.edges.data('color', default='red'):
```

iterates through all the edges yielding the color attribute with default 'red' if no color attribute exists.

**Parameters**

- `nbunch` *(single node, container, or all nodes (default= all nodes)) – The view will only report edges incident to these nodes.*
- `data` *(string or bool, optional (default=False)) – The edge attribute returned in 3-tuple (u, v, ddict[data]). If True, return edge attribute dict in 3-tuple (u, v, ddict). If False, return 2-tuple (u, v).*
- `default` *(value, optional (default=None)) – Value used for edges that don’t have the requested attribute. Only relevant if data is not True or False.*

**Returns** edges – A view of edge attributes, usually it iterates over (u, v) or (u, v, d) tuples of edges, but can also be used for attribute lookup as `edges[u, v]['foo']`.

**Return type** OutEdgeView

See also:

`in_edges`, `out_edges`

**Notes**

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-edges.
Examples

```python
>>> G = nx.DiGraph()  # or MultiDiGraph, etc
>>> nx.add_path(G, [0, 1, 2])
>>> G.add_edge(2, 3, weight=5)
>>> [e for e in G.edges]
[(0, 1), (1, 2), (2, 3)]
>>> G.edges.data()  # default data is {} (empty dict)
OutEdgeDataView([(0, 1), {}, (1, 2), {}, (2, 3), {'weight': 5})]
>>> G.edges.data('weight', default=1)
OutEdgeDataView([(0, 1), (1, 2), (2, 3), (2, 3, 5)])
```
Examples

```python
>>> G = nx.DiGraph()
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.in_degree(0)  # node 0 with degree 0
0
>>> list(G.in_degree([0, 1, 2]))
[(0, 0), (1, 1), (2, 1)]
```

cirq.CircuitDag.in_edges

property CircuitDag.in_edges

An InEdgeView of the Graph as G.in_edges or G.in_edges().

```python
in_edges(self, nbunch=None, data=False, default=None):
```

Parameters

- • `nbunch`  (single node, container, or all nodes (default= all nodes)) – The view will only report edges incident to these nodes.
- • `data`  (string or bool, optional (default=False)) – The edge attribute returned in 3-tuple (u, v, ddict[data]). If True, return edge attribute dict in 3-tuple (u, v, ddict). If False, return 2-tuple (u, v).
- • `default`  (value, optional (default=None)) – Value used for edges that don’t have the requested attribute. Only relevant if data is not True or False.

Returns in_edges – A view of edge attributes, usually it iterates over (u, v) or (u, v, d) tuples of edges, but can also be used for attribute lookup as edges[u, v][‘foo’].

Return type InEdgeView

See also:

edges

cirq.CircuitDag.name

property CircuitDag.name

String identifier of the graph.

This graph attribute appears in the attribute dict G.graph keyed by the string "name". as well as an attribute (technically a property) G.name. This is entirely user controlled.

cirq.CircuitDag.node

property CircuitDag.node

A NodeView of the Graph as G.nodes or G.nodes().

Can be used as G.nodes for data lookup and for set-like operations.
Can also be used as `G.nodes(data='color', default=None)` to return a NodeDataView which reports specific node data but no set operations. It presents a dict-like interface as well with `G.nodes.items()` iterating over `(node, nodedata)` 2-tuples and `G.nodes[3]['foo']` providing the value of the `foo` attribute for node 3. In addition, a view `G.nodes.data('foo')` provides a dict-like interface to the `foo` attribute of each node. `G.nodes.data('foo', default=1)` provides a default for nodes that do not have attribute `foo`.

**Parameters**

- **data** *(string or bool, optional (default=False)) –* The node attribute returned in 2-tuple `(n, ddict[data])`. If True, return entire node attribute dict as `(n, ddict)`. If False, return just the nodes `n`.

- **default** *(value, optional (default=None)) –* Value used for nodes that don’t have the requested attribute. Only relevant if data is not True or False.

**Returns**

Allows set-like operations over the nodes as well as node attribute dict lookup and calling to get a NodeDataView. A NodeDataView iterates over `(n, data)` and has no set operations. A NodeView iterates over `n` and includes set operations.

When called, if data is False, an iterator over nodes. Otherwise an iterator of 2-tuples (node, attribute value) where the attribute is specified in `data`. If data is True then the attribute becomes the entire data dictionary.

**Return type**  NodeView

**Notes**

If your node data is not needed, it is simpler and equivalent to use the expression `for n in G`, or `list(G)`.

**Examples**

There are two simple ways of getting a list of all nodes in the graph:

```python
>>> G = nx.path_graph(3)
>>> list(G.nodes)
[0, 1, 2]
```

To get the node data along with the nodes:

```python
>>> G.add_node(1, time='5pm')
>>> G.nodes[0]['foo'] = 'bar'
>>> list(G.nodes(data=True))
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
>>> list(G.nodes.data())
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
```
>>> list(G.nodes(data='foo'))
[(0, 'bar'), (1, None), (2, None)]

>>> list(G.nodes.data('foo'))
[(0, 'bar'), (1, None), (2, None)]

>>> list(G.nodes(data='time'))
[(0, None), (1, '5pm'), (2, None)]

>>> list(G.nodes.data('time'))
[(0, None), (1, '5pm'), (2, None)]

>>> list(G.nodes(data='time', default='Not Available'))
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]

>>> list(G.nodes.data('time', default='Not Available'))
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]

If some of your nodes have an attribute and the rest are assumed to have a default attribute value you can create a dictionary from node/attribute pairs using the default keyword argument to guarantee the value is never None:

```python
>>> G = nx.Graph()
>>> G.add_node(0)
>>> G.add_node(1, weight=2)
>>> G.add_node(2, weight=3)
>>> dict(G.nodes(data='weight', default=1))
{0: 1, 1: 2, 2: 3}
```

cirq.CircuitDag.nodes

**property CircuitDag.nodes**

A NodeView of the Graph as G.nodes or G.nodes().

Can be used as G.nodes for data lookup and for set-like operations. Can also be used as G.nodes(data='color', default=None) to return a NodeDataView which reports specific node data but no set operations. It presents a dict-like interface as well with G.nodes.items() iterating over (node, nodedata) 2-tuples and G.nodes[3]['foo'] providing the value of the foo attribute for node 3. In addition, a view G.nodes.data('foo') provides a dict-like interface to the foo attribute of each node. G.nodes.data('foo', default=1) provides a default for nodes that do not have attribute foo.

**Parameters**

- **data** *(string or bool, optional (default=False))* – The node attribute returned in 2-tuple (n, ddict[data]). If True, return entire node attribute dict as (n, ddict). If False, return just the nodes n.

- **default** *(value, optional (default=None))* – Value used for nodes that don’t have the requested attribute. Only relevant if data is not True or False.

**Returns**
Allows set-like operations over the nodes as well as node attribute dict lookup and calling to get a NodeDataView. A NodeDataView iterates over \((n, data)\) and has no set operations. A NodeView iterates over \(n\) and includes set operations.

When called, if data is False, an iterator over nodes. Otherwise an iterator of 2-tuples (node, attribute value) where the attribute is specified in \(data\). If data is True then the attribute becomes the entire data dictionary.

**Return type**  
NodeView

**Notes**

If your node data is not needed, it is simpler and equivalent to use the expression for \(n\) in \(G\), or `list(G)`.

**Examples**

There are two simple ways of getting a list of all nodes in the graph:

```python
>>> G = nx.path_graph(3)
>>> list(G.nodes)
[0, 1, 2]
>>> list(G)
[0, 1, 2]
```

To get the node data along with the nodes:

```python
>>> G.add_node(1, time='5pm')
>>> G.nodes[0]['foo'] = 'bar'
>>> list(G.nodes(data=True))
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
>>> list(G.nodes.data())
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
>>> list(G.nodes(data='foo'))
[(0, 'bar'), (1, None), (2, None)]
>>> list(G.nodes.data('foo'))
[(0, 'bar'), (1, None), (2, None)]
>>> list(G.nodes(data='time'))
[(0, None), (1, '5pm'), (2, None)]
>>> list(G.nodes.data('time'))
[(0, None), (1, '5pm'), (2, None)]
>>> list(G.nodes(data='time', default='Not Available'))
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
>>> list(G.nodes.data('time', default='Not Available'))
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
```

If some of your nodes have an attribute and the rest are assumed to have a default attribute value you can create a dictionary from node/attribute pairs using the `default` keyword argument to guarantee the value is never None:
```python
>>> G = nx.Graph()
>>> G.add_node(0)
>>> G.add_node(1, weight=2)
>>> G.add_node(2, weight=3)
>>> dict(G.nodes(data='weight', default=1))
{0: 1, 1: 2, 2: 3}
```

cirq.CircuitDag.out_degree

**property CircuitDag.out_degree**

An OutDegreeView for (node, out_degree)

The node out_degree is the number of edges pointing out of the node. The weighted node degree is the sum of the edge weights for edges incident to that node.

This object provides an iterator over (node, out_degree) as well as lookup for the degree for a single node.

**Parameters**

- `nbunch` *(single node, container, or all nodes (default= all nodes)) –* The view will only report edges incident to these nodes.
- `weight` *(string or None, optional (default=None)) –* The name of an edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1. The degree is the sum of the edge weights adjacent to the node.

**Returns**

- **If a single node is requested**
  - `deg` *(int) –* Out-degree of the node
  - **OR if multiple nodes are requested**
  - `nd_iter` *(iterator) –* The iterator returns two-tuples of (node, out-degree).

**See also:**

degree, in_degree

**Examples**

```python
>>> G = nx.DiGraph()
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.out_degree(0)  # node 0 with degree 1
1
>>> list(G.out_degree([0, 1, 2]))
[(0, 1), (1, 1), (2, 1)]
```
cirq.CircuitDag.out_edges

property CircuitDag.out_edges
An OutEdgeView of the DiGraph as G.edges or G.edges()

edges(self, nbunch=None, data=False, default=None)

The OutEdgeView provides set-like operations on the edge-tuples as well as edge attribute lookup. When called, it also provides an EdgeDataView object which allows control of access to edge attributes (but does not provide set-like operations).
Hence, G.edges[u, v]['color'] provides the value of the color attribute for edge (u, v) while
for (u, v, c) in G.edges.data('color', default='red'):
iterates through all the edges yielding the color attribute with default 'red' if no color attribute exists.

Parameters

• nbunch (single node, container, or all nodes (default= all nodes)) – The view will only report edges incident to these nodes.
• data (string or bool, optional (default=False)) – The edge attribute returned in 3-tuple (u, v, ddict[data]). If True, return edge attribute dict in 3-tuple (u, v, ddict). If False, return 2-tuple (u, v).
• default (value, optional (default=None)) – Value used for edges that don’t have the requested attribute. Only relevant if data is not True or False.

Returns edges – A view of edge attributes, usually it iterates over (u, v) or (u, v, d) tuples of edges, but can also be used for attribute lookup as edges[u, v]['foo'].

Return type OutEdgeView

See also:
in_edges, out_edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-edges.

Examples

```python
>>> G = nx.DiGraph()  # or MultiDiGraph, etc
>>> nx.add_path(G, [0, 1, 2])
>>> G.add_edge(2, 3, weight=5)
>>> [e for e in G.edges]
[(0, 1), (1, 2), (2, 3)]
>>> G.edges.data()  # default data is {} (empty dict)
OutEdgeDataView([(0, 1), {}], [(1, 2), {}], [(2, 3), {'weight': 5}])
```

(continues on next page)
OutEdgeDataView([(0, 1, 1), (1, 2, 1), (2, 3, 5)])

```python
>>> G.edges([0, 2])  # only edges incident to these nodes
OutEdgeDataView([(0, 1), (2, 3)])
```

```python
>>> G.edges(0)  # only edges incident to a single node (use G.adj[0]?)
OutEdgeDataView([(0, 1)])
```

### cirq.CircuitDag.pred

**property** `CircuitDag.pred`

Graph adjacency object holding the predecessors of each node.

This object is a read-only dict-like structure with node keys and neighbor-dict values. The neighbor-dict is keyed by neighbor to the edge-data-dict. So `G.pred[2][3]['color'] = 'blue'` sets the color of the edge `(3, 2)` to "blue".

Iterating over `G.pred` behaves like a dict. Useful idioms include:

```python
for nbr, datadict in G.pred[n].items():
```

A data-view not provided by dicts also exists:

```python
for nbr, foovalue in G.pred[node].data('foo'):
```

A default can be set via a `default` argument to the `data` method.

### cirq.CircuitDag.succ

**property** `CircuitDag.succ`

Graph adjacency object holding the successors of each node.

This object is a read-only dict-like structure with node keys and neighbor-dict values. The neighbor-dict is keyed by neighbor to the edge-data-dict. So `G.succ[3][2]['color'] = 'blue'` sets the color of the edge `(3, 2)` to "blue".

Iterating over `G.succ` behaves like a dict. Useful idioms include:

```python
for nbr, datadict in G.succ[n].items():
```

A data-view not provided by dicts also exists:

```python
for nbr, foovalue in G.succ[node].data('foo'):
```

and a default can be set via a `default` argument to the `data` method.

The neighbor information is also provided by subscripting the graph. So `for nbr, foovalue in G[node].data('foo', default=1):` works.

For directed graphs, `G.adj` is identical to `G.succ`.

For more information, refer to the documentation.
cirq.flatten_op_tree


Performs an in-order iteration of the operations (leaves) in an OP_TREE.

**Parameters**

- **root** – The operation or tree of operations to iterate.
- **preserve_moments** – Whether to yield Moments intact instead of flattening them

**Yields** Operations from the tree.

**Raises** `TypeError` – root isn’t a valid OP_TREE.

cirq.freeze_op_tree


Replaces all iterables in the OP_TREE with tuples.

**Parameters** `root` – The operation or tree of operations to freeze.

**Returns** An OP_TREE with the same operations and branching structure, but where all internal nodes are tuples instead of arbitrary iterables.

cirq.InsertStrategy

class cirq.InsertStrategy (name, doc)

Indicates preferences on how to add multiple operations to a circuit.

**Init**

```python
__init__(name, doc)
    Initialize self. See help(type(self)) for accurate signature.
```

**Methods**

---

**Attributes**

<table>
<thead>
<tr>
<th>EARLIEST</th>
<th>INLINE</th>
<th>NEW</th>
<th>NEW_THEN_INLINE</th>
</tr>
</thead>
</table>

**cirq.InsertStrategy.EARLIEST**

`InsertStrategy.EARLIEST = cirq.InsertStrategy.EARLIEST`

**cirq.InsertStrategy.INLINE**

`InsertStrategy.INLINE = cirq.InsertStrategy.INLINE`
cirq.InsertStrategy.NEW

InsertStrategy.NEW = cirq.InsertStrategy.NEW

cirq.InsertStrategy.NEW_THEN_INLINE

InsertStrategy.NEW_THEN_INLINE = cirq.InsertStrategy.NEW_THEN_INLINE

cirq.Moment

class cirq.Moment (operations: Iterable[cirq.ops.raw_types.Operation] = ())
A simplified time-slice of operations within a sequenced circuit.

Note that grouping sequenced circuits into moments is an abstraction that may not carry over directly to the scheduling on the hardware or simulator. Operations in the same moment may or may not actually end up scheduled to occur at the same time. However the topological quantum circuit ordering will be preserved, and many schedulers or consumers will attempt to maximize the moment representation.

__init__ (operations: Iterable[cirq.ops.raw_types.Operation] = ()) → None
Constructs a moment with the given operations.

Parameters operations – The operations applied within the moment. Will be frozen into a tuple before storing.

Raises ValueError – A qubit appears more than once.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>operates_on(qubits)</td>
<td>Determines if the moment has operations touching the given qubits.</td>
</tr>
<tr>
<td>operates_on_single_qubit(qubit)</td>
<td>Determines if the moment has operations touching the given qubit.</td>
</tr>
<tr>
<td>transform_qubits(func)</td>
<td>Returns the same moment, but with different qubits.</td>
</tr>
<tr>
<td>with_operation(operation)</td>
<td>Returns an equal moment, but with the given op added.</td>
</tr>
<tr>
<td>without_operations_touching(qubits)</td>
<td>Returns an equal moment, but without ops on the given qubits.</td>
</tr>
</tbody>
</table>

cirq.Moment.operates_on

Moment.operates_on (qubits: Iterable[cirq.ops.raw_types.Qid]) → bool
Determines if the moment has operations touching the given qubits.

Parameters qubits – The qubits that may or may not be touched by operations.

Returns Whether this moment has operations involving the qubits.
**cirq.Moment.operates_on_single_qubit**

Moment.\texttt{operates\_on\_single\_qubit}(\textit{qubit}: \texttt{cirq.ops.raw\_types.Qid}) $\rightarrow$ bool
Determine if the moment has operations touching the given qubit.

- **Parameters** \textit{qubit} – The qubit that may or may not be touched by operations.
- **Returns** Whether this moment has operations involving the qubit.

**cirq.Moment.transform_qubits**

Moment.\texttt{transform\_qubits}(\textit{func}: Callable[\texttt{cirq.ops.raw\_types.Qid, cirq.ops.raw\_types.Qid}]) $\rightarrow$ TSelf\_Moment
Returns the same moment, but with different qubits.

- **Parameters** \textit{func} – The function to use to turn each current qubit into a desired new qubit.
- **Returns**
  
  The receiving moment but with qubits transformed by the given function.

**cirq.Moment.with_operation**

Moment.\texttt{with\_operation}(\textit{operation}: \texttt{cirq.ops.raw\_types.Operation})
Returns an equal moment, but with the given op added.

- **Parameters** \textit{operation} – The operation to append.
- **Returns** The new moment.

**cirq.Moment.without_operations.touching**

Moment.\texttt{without\_operations\_touching}(\textit{qubits}: \texttt{Iterable[cirq.ops.raw\_types.Qid]})
Returns an equal moment, but without ops on the given qubits.

- **Parameters** \textit{qubits} – Operations that touch these will be removed.
- **Returns** The new moment.

**Attributes**

- \texttt{operations}

- \texttt{qubits}

**cirq.Moment.operations**

**property** Moment.\texttt{operations}

**cirq.Moment.qubits**

**property** Moment.\texttt{qubits}
**cirq.moment_by_moment_schedule**

This method attempts to create a schedule in which each moment of a circuit is scheduled starting at the same time. Given the constraints of the given device, such a schedule may not be possible, in this case the method will raise a ValueError with a description of the conflict.

The schedule that is produced will take each moments and schedule the operations in this moment in a time slice of length equal to the maximum time of an operation in the moment.

**Returns** A Schedule for the circuit.

**Raises** ValueError – if the scheduling cannot be done.

**cirq.op_gate_of_type**

**cirq.OP_TREE**

To define a union, use e.g. Union[int, str]. Details:

- The arguments must be types and there must be at least one.
- None as an argument is a special case and is replaced by type(None).
- Unions of unions are flattened, e.g.:

  ```python
  Union[Union[int, str], float] == Union[int, str, float]
  ```

- Unions of a single argument vanish, e.g.:

  ```python
  Union[int] == int  # The constructor actually returns int
  ```

- Redundant arguments are skipped, e.g.:

  ```python
  Union[int, str, int] == Union[int, str]
  ```

- When comparing unions, the argument order is ignored, e.g.:
• When two arguments have a subclass relationship, the least derived argument is kept, e.g.:

```python
class Employee: pass
class Manager(Employee): pass
```

```python
Union[int, Employee, Manager] == Union[int, Employee]
Union[Manager, int, Employee] == Union[int, Employee]
Union[Employee, Manager] == Employee
```

• Similar for object:

```python
Union[int, object] == object
```

• You cannot subclass or instantiate a union.

• You can use Optional[X] as a shorthand for Union[X, None].

cirq.ParallelGateOperation

class cirq.ParallelGateOperation (gate: cirq.ops.raw_types.Gate, qubits: Sequence[cirq.ops.raw_types.Qid])

An application of several copies of a gate to a group of qubits.

```python
__init__ (gate: cirq.ops.raw_types.Gate, qubits: Sequence[cirq.ops.raw_types.Qid]) → None
```

Parameters

• `gate` – the gate to apply.

• `qubits` – list of qubits to apply the gate to.

Methods

controlled_by(`*control_qubits[, control_values]`) Returns a controlled version of this operation. If no control_qubits

transform_qubits(func) Returns the same operation, but with different qubits.

validate_args(qubits) Raises an exception if the qubits don’t match this operation’s qid

with_gate(new_gate) ParallelGateOperation with same qubits but a new gate

with_qubits(`*new_qubits`) ParallelGateOperation with same the gate but new qubits
Returns a controlled version of this operation. If no control_qubits
are specified, returns self.

Parameters

- **control_qubits** – Qubits to control the operation by. Required.
- **control_values** – For which control qubit values to apply the operation. A sequence
  of the same length as control_qubits where each entry is an integer (or set of integers)
  corresponding to the qubit value (or set of possible values) where that control is enabled.
  When all controls are enabled, the operation is applied. If unspecified, control values
default to 1.

cirq.ParallelGateOperation.transform_qubits

ParallelGateOperation.transform_qubits(func: Callable[[cirq.ops.raw_types.Qid, cirq.ops.raw_types.Qid]], 
                                        ) -> cirq.ops.raw_types.Operation

Returns the same operation, but with different qubits.

Parameters

- **func** – The function to use to turn each current qubit into a desired new qubit.

Returns

The receiving operation but with qubits transformed by the given function.

cirq.ParallelGateOperation.validate_args

ParallelGateOperation.validate_args(qubits: Sequence[cirq.ops.raw_types.Qid])

 Raises an exception if the qubits don’t match this operation’s qid shape.

Call this method from a subclass’s with_qubits method.

Parameters

- **qubits** – The new qids for the operation.

Raises **ValueError** – The operation had qids that don’t match it’s qid shape.

cirq.ParallelGateOperation.with_gate

ParallelGateOperation.with_gate(new_gate: cirq.ops.raw_types.Gate) 
                                        ) -> cirq.ops.parallel_gate_operation.ParallelGateOperation

ParallelGateOperation with same qubits but a new gate

cirq.ParallelGateOperation.with_qubits

ParallelGateOperation.with_qubits(*new_qubits: cirq.ops.raw_types.Qid) 
                                        ) -> cirq.ops.parallel_gate_operation.ParallelGateOperation

ParallelGateOperation with same the gate but new qubits
Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gate</code></td>
<td>The single qubit gate applied by the operation.</td>
</tr>
<tr>
<td><code>qubits</code></td>
<td>The qubits targeted by the operation.</td>
</tr>
</tbody>
</table>

`cirq.ParallelGateOperation.gate`

**property** `ParallelGateOperation.gate`

The single qubit gate applied by the operation.

`cirq.ParallelGateOperation.qubits`

**property** `ParallelGateOperation.qubits`

The qubits targeted by the operation.

`cirq.QubitOrder`

**class** `cirq.QubitOrder` *(explicit_func: Callable[Iterable[cirq.ops.raw_types.Qid], Tuple[cirq.ops.raw_types.Qid, ...]])*  
Defines the kronecker product order of qubits.

**__init__** *(explicit_func: Callable[Iterable[cirq.ops.raw_types.Qid], Tuple[cirq.ops.raw_types.Qid, ...]]) → None*  
Initialize self. See help(type(self)) for accurate signature.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>as_qubit_order(val)</code></td>
<td>Converts a value into a basis.</td>
</tr>
<tr>
<td><code>explicit(fixed_qubits[, fallback])</code></td>
<td>A basis that contains exactly the given qubits in the given order.</td>
</tr>
<tr>
<td><code>map(internalize, externalize)</code></td>
<td>Transforms the Basis so that it applies to wrapped qubits.</td>
</tr>
<tr>
<td><code>order_for(qubits)</code></td>
<td>Returns a qubit tuple ordered corresponding to the basis.</td>
</tr>
<tr>
<td><code>sorted_by(key)</code></td>
<td>A basis that orders qubits ascending based on a key function.</td>
</tr>
</tbody>
</table>

`cirq.QubitOrder.as_qubit_order`

**static** `QubitOrder.as_qubit_order` *(val: qubit_order_or_list.QubitOrderOrList) → QubitOrder*  
Converts a value into a basis.

**Parameters** `val` – An iterable or a basis.

**Returns** The basis implied by the value.
cirq.QubitOrder.explicit

```python
static QubitOrder.explicit(fixed_qubits: Iterable[cirq.ops.raw_types.Qid], fallback: Optional[QubitOrder] = None) → cirq.ops.qubit_order.QubitOrder
```

A basis that contains exactly the given qubits in the given order.

**Parameters**

- `fixed_qubits` - The qubits in basis order.
- `fallback` - A fallback order to use for extra qubits not in the fixed_qubits list. Extra qubits will always come after the fixed_qubits, but will be ordered based on the fallback. If no fallback is specified, a ValueError is raised when extra qubits are specified.

**Returns** A Basis instance that forces the given qubits in the given order.

cirq.QubitOrder.map

```python
QubitOrder.map(internalize: Callable[TExternalQubit, TInternalQubit], externalize: Callable[TInternalQubit, TExternalQubit]) → cirq.ops.qubit_order.QubitOrder
```

Transforms the Basis so that it applies to wrapped qubits.

**Parameters**

- `externalize` - Converts an internal qubit understood by the underlying basis into an external qubit understood by the caller.
- `internalize` - Converts an external qubit understood by the caller into an internal qubit understood by the underlying basis.

**Returns** A basis that transforms qubits understood by the caller into qubits understood by an underlying basis, uses that to order the qubits, then wraps the ordered qubits back up for the caller.

cirq.QubitOrder.order_for

```python
QubitOrder.order_for(qubits: Iterable[cirq.ops.raw_types.Qid]) → Tuple[cirq.ops.raw_types.Qid, ...]
```

Returns a qubit tuple ordered corresponding to the basis.

**Parameters** `qubits` - Qubits that should be included in the basis. (Additional qubits may be added into the output by the basis.)

**Returns** A tuple of qubits in the same order that their single-qubit matrices would be passed into `np.kron` when producing a matrix for the entire system.

cirq.QubitOrder.sorted_by

```python
static QubitOrder.sorted_by(key: Callable[cirq.ops.raw_types.Qid, Any]) → cirq.ops.qubit_order.QubitOrder
```

A basis that orders qubits ascending based on a key function.

**Parameters** `key` - A function that takes a qubit and returns a key value. The basis will be ordered ascending according to these key values.

**Returns** A basis that orders qubits ascending based on a key function.
Attributes

**cirq.QubitOrder.DEFAULT**

```python
QubitOrder.DEFAULT = <cirq.ops.qubit_order.QubitOrder object>
```

A basis that orders qubits in the same way that calling `sorted` does.

Specifically, qubits are ordered first by their type name and then by whatever comparison value qubits of a given type provide (e.g. for `LineQubit` it is the x coordinate of the qubit).

---

**cirq.QubitOrderOrList**

```python
cirq.QubitOrderOrList = typing.Union[cirq.ops.qubit_order.QubitOrder, typing.Iterable[cirq.ops.raw_types.Qid]]
```

Union type; `Union[X, Y]` means either `X` or `Y`.

To define a union, use e.g. `Union[int, str]`. Details:

- The arguments must be types and there must be at least one.
- None as an argument is a special case and is replaced by `type(None)`.
- Unions of unions are flattened, e.g.:
  ```python
  Union[Union[int, str], float] == Union[int, str, float]
  ```
- Unions of a single argument vanish, e.g.:
  ```python
  Union[int] == int  # The constructor actually returns int
  ```
- Redundant arguments are skipped, e.g.:
  ```python
  Union[int, str, int] == Union[int, str]
  ```
- When comparing unions, the argument order is ignored, e.g.:
  ```python
  Union[int, str] == Union[str, int]
  ```
- When two arguments have a subclass relationship, the least derived argument is kept, e.g.:
  ```python
  class Employee: pass
class Manager(Employee): pass
Union[int, Employee, Manager] == Union[int, Employee]
Union[Manager, int, Employee] == Union[int, Employee]
Union[Employee, Manager] == Employee
  ```
- Similar for object:
  ```python
  Union[int, object] == object
  ```
• You cannot subclass or instantiate a union.
• You can use Optional[X] as a shorthand for Union[X, None].

cirq.Schedule

A quantum program with operations happening at specific times.

Supports schedule[time] point lookups and
schedule[inclusive_start_time:exclusive_end_time] slice lookups.

device
The hardware this will schedule on.

scheduled_operations

A SortedListWithKey containing the
ScheduledOperations for this schedule. The key is the start time
of the ScheduledOperation.

Initializes a new schedule.

Parameters

• device – The hardware this schedule will run on.
• scheduled_operations – Initial list of operations to apply. These will be moved
into a sorted list, with a key equal to each operation’s start time.

Methods

exclude(scheduled_operation) Omits a scheduled operation from the schedule, if present.

include(scheduled_operation) Adds a scheduled operation to the schedule.

operations_happening_at_same_time_as(. . .) Finds operations happening at the same time as the
given operation.

query(*, time[, duration, qubits, . . .]) Finds operations by time and qubit.

to_circuit() Convert the schedule to a circuit.

cirq.Schedule.exclude

Schedule.exclude(scheduled_operation: cirq.schedules.scheduled_operation.ScheduledOperation) -> bool
Omits a scheduled operation from the schedule, if present.

Parameters scheduled_operation – The operation to try to remove.
**Returns** True if the operation was present and is now removed, False if it was already not present.

**cirq.Schedule.include**

Schedule.include(scheduled_operation: cirq.schedules.scheduled_operation.ScheduledOperation)

Adds a scheduled operation to the schedule.

**Parameters** scheduled_operation – The operation to add.

**Raises** ValueError – The operation collided with something already in the schedule.

**cirq.Schedule.operations_happening_at_same_time_as**

Schedule.operations_happening_at_same_time_as(scheduled_operation: cirq.schedules.scheduled_operation.ScheduledOperation)

→ List[cirq.schedules.scheduled_operation.ScheduledOperation]

Finds operations happening at the same time as the given operation.

**Parameters** scheduled_operation – The operation specifying the time to query.

**Returns** Scheduled operations that overlap with the given operation.

**cirq.Schedule.query**

Schedule.query(*, time: cirq.value.timestamp.Timestamp, duration: cirq.DURATION_LIKE = None, qubits: Iterable[cirq.ops.raw_types.Qid] = None, include_query_end_time=False, include_op_end_times=False) → List[cirq.schedules.scheduled_operation.ScheduledOperation]

Finds operations by time and qubit.

**Parameters**

- **time** – Operations must end after this time to be returned.
- **duration** – Operations must start by time+duration to be returned.
- **qubits** – If specified, only operations touching one of the included qubits will be returned.
- **include_query_end_time** – Determines if the query interval includes its end time. Defaults to no.
- **include_op_end_times** – Determines if the scheduled operation intervals include their end times or not. Defaults to no.

**Returns** A list of scheduled operations meeting the specified conditions.

**cirq.Schedule.to_circuit**

Schedule.to_circuit() → cirq.circuits.circuit.Circuit

Convert the schedule to a circuit.

This discards most timing information from the schedule, but does place operations that are scheduled at the same time in the same Moment.
cirq.ScheduledOperation

```python
```

An operation that happens over a specified time interval.

```python
__init__(time: cirq.value.timestamp.Timestamp, duration: cirq.DURATION_LIKE, operation: cirq.ops.raw_types.Operation) ➔ None
```

Initializes the scheduled operation.

**Parameters**

- **time** – When the operation starts.
- **duration** – How long the operation lasts.
- **operation** – The operation.

**Methods**

```python
cirq.ScheduledOperation.op_at_on(operation, time, device)
```

Creates a scheduled operation with a device-determined duration.

```python
```

Creates a scheduled operation with a device-determined duration.

**cirq.transform_op_tree**

```python
```

Maps transformation functions onto the nodes of an OP_TREE.

**Parameters**

- **root** – The operation or tree of operations to transform.
- **op_transformation** – How to transform the operations (i.e. leaves).
- **iter_transformation** – How to transform the iterables (i.e. internal nodes).
- **preserve_moments** – Whether to leave Moments alone. If True, the transformation functions will not be applied to Moments or the operations within them.

**Returns** A transformed operation tree.

** Raises ** **TypeError** – root isn’t a valid OP_TREE.
cirq.Unique

class cirq.Unique(val: T)
A wrapper for a value that doesn’t compare equal to other instances.

For example: 5 == 5 but Unique(5) != Unique(5).

Unique is used by CircuitDag to wrap operations because nodes in a graph
are considered the same node if they compare equal to each other. X(q0)
in one moment of a Circuit and X(q0) in another moment of the Circuit are
wrapped by Unique(X(q0)) so they are distinct nodes in the graph.

__init__(val: T) → None
Initialize self. See help(type(self)) for accurate signature.

Methods

3.1.11 Trials and Simulations

Classes for simulations and results.

bloch_vector_from_state_vector(state, index) Returns the bloch vector of a qubit.
density_matrix_from_state_vector(state[, ...]) Returns the density matrix of the wavefunction.

DensityMatrixSimulator(*[, dtype, noise, seed]) A simulator for density matrices and noisy quantum circuits.
DensityMatrixSimulatorState(density_matrix, ...) The simulator state for DensityMatrixSimulator
DensityMatrixStepResult(density_matrix, ...) A single step in the simulation of the DensityMatrixSimulator.
DensityMatrixTrialResult(params, ...) A SimulationTrialResult for DensityMatrixSimulator runs.
dirac_notation(state[, decimals, qid_shape]) Returns the wavefunction as a string in Dirac notation.
measure_density_matrix(density_matrix, indices) Performs a measurement of the density matrix in the computational basis.
measure_state_vector(state, indices, *[...]) Performs a measurement of the state in the computational basis.
sample(program, *[noise, param_resolver, ...]) Simulates sampling from the given circuit or schedule.
sample_density_matrix(density_matrix, indices, *) Samples repeatedly from measurements in the computational basis.
sample_state_vector(state, indices, *[...]) Samples repeatedly from measurements in the computational basis.
sample_sweep(program, params, *[noise, ...]) Runs the supplied Circuit or Schedule, mimicking quantum hardware.
Sampler Something capable of sampling quantum circuits.

Continued on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimulatesFinalState</td>
<td>Simulator that allows access to a quantum computer’s final state.</td>
</tr>
<tr>
<td>SimulatesIntermediateState</td>
<td>A SimulatesFinalState that simulates a circuit by moments.</td>
</tr>
<tr>
<td>SimulatesIntermediateWaveFunction</td>
<td>A simulator that accesses its wave function as it does its simulation.</td>
</tr>
<tr>
<td>SimulatesSamples</td>
<td>Simulator that mimics running on quantum hardware.</td>
</tr>
<tr>
<td>SimulationTrialResult**(params, measurements,...)**</td>
<td>Results of a simulation by a SimulatesFinalState.</td>
</tr>
<tr>
<td>Simulator**(dtype, seed)**</td>
<td>A sparse matrix wave function simulator that uses numpy.</td>
</tr>
<tr>
<td>SparseSimulatorStep**(state_vector,...)**</td>
<td>A StepResult that includes StateVectorMixin methods.</td>
</tr>
<tr>
<td>StateVectorMixin**(qubit_map)**</td>
<td>A mixin that provide methods for objects that have a state vector.</td>
</tr>
<tr>
<td>StepResult**(measurements)**</td>
<td>Results of a step of a SimulatesIntermediateState.</td>
</tr>
<tr>
<td>TrialResult**(params, measurements)**</td>
<td>The results of multiple executions of a circuit with fixed parameters.</td>
</tr>
<tr>
<td>to_valid_density_matrix**(density_matrix_rep)**</td>
<td>Verifies the density_matrix_rep is valid and converts it to ndarray form.</td>
</tr>
<tr>
<td>to_valid_state_vector**(state_rep, num_qubits,***)</td>
<td>Verifies the state_rep is valid and converts it to ndarray form.</td>
</tr>
<tr>
<td>validate_normalized_state**(state,***)</td>
<td>Validates that the given state is a valid wave function.</td>
</tr>
<tr>
<td>validate_probability**(p, p_str)**</td>
<td>Validates that a probability is between 0 and 1 inclusively.</td>
</tr>
<tr>
<td>WaveFunctionSimulatorState**(state_vector,***)</td>
<td></td>
</tr>
<tr>
<td>WaveFunctionStepResult**(measurements)**</td>
<td></td>
</tr>
<tr>
<td>WaveFunctionTrialResult**(params,***)</td>
<td>A SimulationTrialResult that includes the StateVectorMixin methods.</td>
</tr>
</tbody>
</table>

```
cirq.bloch_vector_from_state_vector
```

Returns the bloch vector of a qubit.

Calculates the bloch vector of the qubit at index in the wavefunction given by state, assuming state follows the standard Kronecker convention of numpy.kron.

**Parameters**

- **state** – A sequence representing a wave function in which the ordering mapping to qubits follows the standard Kronecker convention of numpy.kron.
- **index** – index of qubit who’s bloch vector we want to find. follows the standard Kronecker convention of numpy.kron.
cirq.density_matrix_from_state_vector

cirq.density_matrix_from_state_vector(state: Sequence, indices: Optional[Iterable[int]] = None, qid_shape: Optional[Tuple[int, ...]] = None) → numpy.ndarray

Returns the density matrix of the wavefunction.

Calculate the density matrix for the system on the given qubit indices, with the qubits not in indices that are present in state traced out. If indices is None the full density matrix for state is returned. We assume state follows the standard Kronecker convention of numpy.kron.

For example: state = np.array([1/np.sqrt(2), 1/np.sqrt(2)], dtype=np.complex64) indices = None gives us

$$ \rho = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix} $$

Parameters

• **state** – A sequence representing a wave function in which the ordering mapping to qubits follows the standard Kronecker convention of numpy.kron.

• **indices** – list containing indices for qubits that you would like to include in the density matrix (i.e.) qubits that WON’T be traced out. follows the standard Kronecker convention of numpy.kron.

Returns A numpy array representing the density matrix.

Raises

• **ValueError** – if the size of state is not a power of 2.

• **ValueError** – if the size of the state represents more than 25 qubits.

• **IndexError** – if the indices are out of range for the number of qubits corresponding to the state.

cirq.DensityMatrixSimulator

class cirq.DensityMatrixSimulator(*, dtype: Type[numpy.number] = <class 'numpy.complex64'>, noise: cirq.NOISE_MODEL_LIKE = None, seed: Union[numpy.random.mtrand.RandomState, int, None] = None)

A simulator for density matrices and noisy quantum circuits.
This simulator can be applied on circuits that are made up of operations
that have:
* a \_channel\_ method
* a \_mixture\_ method for a probabilistic combination of unitary gates.
* a \_unitary\_ method
* a \_has_unitary\_ and \_apply_unitary\_ method.
* measurements
* a \_decompose\_ that eventually yields one of the above
That is, the circuit must have elements that follow on of the protocols:
* cirq.SupportsChannel
* cirq.SupportsMixture
* cirq.SupportsConsistentApplyUnitary
* cirq.SupportsUnitary
* cirq.SupportsDecompose
or is a measurement.

This simulator supports three types of simulation.

Run simulations which mimic running on actual quantum hardware. These
simulations do not give access to the density matrix (like actual hardware).
There are two variations of run methods, one which takes in a single
(optional) way to resolve parameterized circuits, and a second which
takes in a list or sweep of parameter resolver:

run(circuit, param_resolver, repetitions)
run_sweep(circuit, params, repetitions)

These methods return TrialResults which contain both the measurement
results, but also the parameters used for the parameterized
circuit operations. The initial state of a run is always the all 0s state
in the computational basis.

By contrast the simulate methods of the simulator give access to the density
matrix of the simulation at the end of the simulation of the circuit.
Note that if the circuit contains measurements then the density matrix
is that result for those particular measurement results. For example
if there is one measurement, then the simulation may result in the
measurement result for this measurement, and the density matrix will
be that conditional on that result. It will not be the density matrix formed
by summing over the different measurements and their probabilities.
The simulate methods take in two parameters that the run methods do not: a
qubit order and an initial state. The qubit order is necessary because an
ordering must be chosen for the kronecker product (see
DensityMatrixTrialResult for details of this ordering). The initial state can be either the full density matrix, the full wave function (for pure states), or an integer which represents the initial state of being in a computational basis state for the binary representation of that integer. Similar to run methods, there are two simulate methods that run for single simulations or for sweeps across different parameters:

```python
simulate(circuit, param_resolver, qubit_order, initial_state)
simulate_sweep(circuit, params, qubit_order, initial_state)
```

The simulate methods in contrast to the run methods do not perform repetitions. The result of these simulations is a DensityMatrixTrialResult which contains, in addition to measurement results and information about the parameters that were used in the simulation, access to the density matrix via the `density_matrix` method.

If one wishes to perform simulations that have access to the density matrix as one steps through running the circuit there is a generator which can be iterated over and each step is an object that gives access to the density matrix. This stepping through a Circuit is done on a Moment by Moment manner.

```python
simulate_moment_steps(circuit, param_resolver, qubit_order, initial_state)
```

One can iterate over the moments via

```python
for step_result in simulate_moments(circuit):
    # do something with the density matrix via
    # step_result.density_matrix()
```

```python
__init__(
*args, dtype: Type[numpy.number] = <class 'numpy.complex64'>, noise: cirq.NOISE_MODELLIKE = None, seed: Union[numpy.random.mtrand.RandomState, int, None] = None)
```

Density matrix simulator.

**Parameters**

- `dtype` – The `numpy.dtype` used by the simulation. One of `numpy.complex64` or `numpy.complex128`
- `noise` – A noise model to apply while simulating.
- `seed` – The random seed to use for this simulator.

**Methods**
### run

Samples from the given Circuit or Schedule.

#### run_async

Asynchronously samples from the given Circuit or Schedule.

#### run_sweep

Runs the supplied Circuit or Schedule, mimicking quantum hardware.

#### run_sweep_async

Asynchronously sweeps and samples from the given Circuit or Schedule.

#### sample

Samples the given Circuit or Schedule, producing a pandas DataFrame.

#### simulate

Simulates the supplied Circuit or Schedule.

#### simulate_sweep

Simulates the supplied Circuit or Schedule.

#### cirq.DensityMatrixSimulator.run

Samples from the given Circuit or Schedule.

By default, the run_async method invokes this method on another thread. So this method is supposed to be thread safe.

**Parameters**

- **program** — The circuit or schedule to sample from.
- **param_resolver** — Parameters to run with the program.
- **repetitions** — The number of times to sample.

**Returns**  
TrialResult for a run.

#### cirq.DensityMatrixSimulator.run_async

Asynchronously samples from the given Circuit or Schedule.

By default, this method calls run on another thread and yields the result via the asyncio event loop. However, child classes are free to override it to use other strategies.

**Parameters**

- **program** — The circuit or schedule to sample from.
- **repetitions** — The number of times to sample.

**Returns**  
An awaitable TrialResult.
cirq.DensityMatrixSimulator.run_sweep

```
DensityMatrixSimulator.run_sweep(program: Union[cirq.circuits.circuit.Circuit,
cirq.schedules.schedule.Schedule],
params: Union[Dict[str, float],
cirq.study.resolver.ParamResolver,
cirq.study.sweeps.Sweep, Iterable[Union[Dict[str,
float], cirq.study.resolver.ParamResolver,
cirq.study.sweeps.Sweep]], None], repetitions: int
= 1) → List[cirq.study.trial_result.TrialResult]
```

Runs the supplied Circuit or Schedule, mimicking quantum hardware.

In contrast to run, this allows for sweeping over different parameter values.

**Parameters**

- **program** – The circuit or schedule to simulate.
- **params** – Parameters to run with the program.
- **repetitions** – The number of repetitions to simulate.

**Returns** TrialResult list for this run; one for each possible parameter resolver.

cirq.DensityMatrixSimulator.run_sweep_async

```
async DensityMatrixSimulator.run_sweep_async(program: Union[cirq.Circuit,
cirq.Schedule], params: cirq.Sweepable, repetitions: int
= 1) → List[cirq.TrialResult]
```

Asynchronously sweeps and samples from the given Circuit or Schedule.

By default, this method calls `run_sweep` on another thread and yields the result via the asyncio event loop. However, child classes are free to override it to use other strategies.

**Parameters**

- **program** – The circuit or schedule to sample from.
- **params** – One or more mappings from parameter keys to parameter values to use. For each parameter assignment, `repetitions` samples will be taken.
- **repetitions** – The number of times to sample.

**Returns** An awaitable TrialResult.

cirq.DensityMatrixSimulator.sample

```
DensityMatrixSimulator.sample(program: Union[cirq.Circuit, cirq.Schedule], *, repetitions:
int = 1, params: cirq.Sweepable = None) → pd.DataFrame
```

Samples the given Circuit or Schedule, producing a pandas data frame.
### Parameters

- **program** – The circuit or schedule to sample from.
- **repetitions** – The number of times to sample the program, for each parameter mapping.
- **params** – Maps symbols to one or more values. This argument can be a dictionary, a list of dictionaries, a `cirq.Sweep`, a list of `cirq.Sweep`, etc. The program will be sampled `repetition` times for each mapping. Defaults to a single empty mapping.

### Returns

A `pandas.DataFrame` with a row for each sample, and a column for each measurement result as well as a column for each symbolic parameter. There is an also index column containing the repetition number, for each parameter assignment.

### Examples

```python
>>> a, b, c = cirq.LineQubit.range(3)
>>> sampler = cirq.Simulator()
>>> circuit = cirq.Circuit(cirq.X(a), ...
...     cirq.measure(a, key='out'))
>>> print(sampler.sample(circuit, repetitions=4))
out
0 1
1 1
2 1
3 1
```

```python
>>> circuit = cirq.Circuit(cirq.X(a), ...
...     cirq.CNOT(a, b), ...
...     cirq.measure(a, b, c, key='out'))
>>> print(sampler.sample(circuit, repetitions=4))
out
0 6
1 6
2 6
3 6
```

```python
>>> circuit = cirq.Circuit(cirq.X(a)**sympy.Symbol('t'), ...
...     cirq.measure(a, key='out'))
>>> print(sampler.sample(circuit, repetitions=3, params=[{'t': 0}, {'t': 1}]))
t out
0 0 0
1 0 0
2 0 0
0 1 1
1 1 1
2 1 1
```
**cirq.DensityMatrixSimulator.simulate**


Simulates the supplied Circuit or Schedule.

This method returns a result which allows access to the entire wave function.

**Parameters**

- **program** – The circuit or schedule to simulate.
- **param_resolver** – Parameters to run with the program.
- **qubit_order** – Determines the canonical ordering of the qubits. This is often used in specifying the initial state, i.e. the ordering of the computational basis states.
- **initial_state** – The initial state for the simulation. The form of this state depends on the simulation implementation. See documentation of the implementing class for details.

**Returns** SimulationTrialResults for the simulation. Includes the final state.

**cirq.DensityMatrixSimulator.simulate_moment_steps**


Returns an iterator of StepResults for each moment simulated.

If the circuit being simulated is empty, a single step result should be returned with the state being set to the initial state.

**Parameters**

- **circuit** – The Circuit to simulate.
• **param_resolver** – A ParamResolver for determining values of Symbols.

• **qubit_order** – Determines the canonical ordering of the qubits. This is often used in specifying the initial state, i.e. the ordering of the computational basis states.

• **initial_state** – The initial state for the simulation. The form of this state depends on the simulation implementation. See documentation of the implementing class for details.

**Returns** Iterator that steps through the simulation, simulating each moment and returning a StepResult for each moment.

**cirq.DensityMatrixSimulator.simulate_sweep**

```python
DensityMatrixSimulator.simulate_sweep(program: Union[cirq.circuits.circuit.Circuit, cirq.schedules.schedule.Schedule],
```

Simulates the supplied Circuit or Schedule.

This method returns a result which allows access to the entire wave function. In contrast to simulate, this allows for sweeping over different parameter values.

**Parameters**

• **program** – The circuit or schedule to simulate.

• **params** – Parameters to run with the program.

• **qubit_order** – Determines the canonical ordering of the qubits. This is often used in specifying the initial state, i.e. the ordering of the computational basis states.

• **initial_state** – The initial state for the simulation. The form of this state depends on the simulation implementation. See documentation of the implementing class for details.

**Returns** List of SimulationTrialResults for this run, one for each possible parameter resolver.

**cirq.DensityMatrixSimulatorState**

```python
class cirq.DensityMatrixSimulatorState(density_matrix: numpy.ndarray, qubit_map: Dict[cirq.ops.raw_types.Qid, int])
```

The simulator state for DensityMatrixSimulator

**Parameters**

• **density_matrix** – The density matrix of the simulation.
• **qubit_map** – A map from qid to index used to define the ordering of the basis in density_matrix.

```python
__init__(density_matrix: numpy.ndarray, qubit_map: Dict[cirq.ops.raw_types.Qid, int])
```

Initialize self. See help(type(self)) for accurate signature.

### Methods

#### cirq.DensityMatrixStepResult

**class cirq.DensityMatrixStepResult**

```python
density_matrix: numpy.ndarray, measurements: Dict[str, numpy.ndarray], qubit_map: Dict[cirq.ops.raw_types.Qid, int], dtype: Type[numpy.number] = <class 'numpy.complex64'>
```

A single step in the simulation of the DensityMatrixSimulator.

**qubit_map**

A map from the Qubits in the Circuit to the the index of this qubit for a canonical ordering. This canonical ordering is used to define the state vector (see the state_vector() method).

**measurements**

A dictionary from measurement gate key to measurement results, ordered by the qubits that the measurement operates on.

```python
__init__(density_matrix: numpy.ndarray, measurements: Dict[str, numpy.ndarray], qubit_map: Dict[cirq.ops.raw_types.Qid, int], dtype: Type[numpy.number] = <class 'numpy.complex64'>)
```

DensityMatrixStepResult.

**Parameters**

- **density_matrix** – The density matrix at this step. Can be mutated.
- **measurements** – The measurements for this step of the simulation.
- **qubit_map** – A map from qid to index used to define the ordering of the basis in density_matrix.
- **dtype** – The numpy dtype for the density matrix.

**Methods**

```python
density_matrix()
```

Returns the density matrix at this step in the simulation.

Continued on next page
### Table 112 – continued from previous page

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sample(qubits[, repetitions, seed])</code></td>
<td>Samples from the system at this point in the computation.</td>
</tr>
<tr>
<td><code>sample_measurement_ops(measurement_ops[, ...])</code></td>
<td>Samples from the system at this point in the computation.</td>
</tr>
<tr>
<td><code>set_density_matrix(density_matrix_repr)</code></td>
<td>Set the density matrix to a new density matrix.</td>
</tr>
</tbody>
</table>

---

**cirq.DensityMatrixStepResult.density_matrix**

DensityMatrixStepResult.density_matrix()

Returns the density matrix at this step in the simulation.

The density matrix that is stored in this result is returned in the computational basis with these basis states defined by the qubit_map. In particular the value in the qubit_map is the index of the qubit, and these are translated into binary vectors where the last qubit is the 1s bit of the index, the second-to-last is the 2s bit of the index, and so forth (i.e. big endian ordering). The density matrix is a $2^{**\text{num_qubits}}$ square matrix, with rows and columns ordered by the computational basis as just described.

#### Example

$qubit\_map: \{QubitA: 0, QubitB: 1, QubitC: 2\}$

Then the returned density matrix will have (row and column) indices mapped to qubit basis states like the following table

<table>
<thead>
<tr>
<th>QubitA</th>
<th>QubitB</th>
<th>QubitC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>1 0</td>
</tr>
<tr>
<td>2 0</td>
<td>1 0</td>
<td>0 0</td>
</tr>
<tr>
<td>3 0</td>
<td>1 1</td>
<td>1 1</td>
</tr>
<tr>
<td>4 1</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>5 1</td>
<td>0 1</td>
<td>1 1</td>
</tr>
<tr>
<td>6 1</td>
<td>1 0</td>
<td>0 0</td>
</tr>
<tr>
<td>7 1</td>
<td>1 1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

---

**cirq.DensityMatrixStepResult.sample**

DensityMatrixStepResult.sample(qubits: List[cirq.ops.raw_types.Qid], repetitions: int = 1, seed: Union[numpy.random.mtrand.RandomState, int, None] = None) → numpy.ndarray

Samples from the system at this point in the computation.

Note that this does not collapse the wave function.

#### Parameters
• **qubits** – The qubits to be sampled in an order that influence the returned measurement results.

• **repetitions** – The number of samples to take.

• **seed** – A seed for the pseudorandom number generator.

**Returns** Measurement results with True corresponding to the $|1\rangle$ state. The outer list is for repetitions, and the inner corresponds to measurements ordered by the supplied qubits. These lists are wrapped as an numpy ndarray.

cirq.DensityMatrixStepResult.sample_measurement_ops

DensityMatrixStepResult.sample_measurement_ops(measurement_ops: List[cirq.ops.gate_operation.GateOperation], repetitions: int = 1, seed: Union[numpy.random.mtrand.RandomState, int, None] = None) → Dict[str, numpy.ndarray]

Samples from the system at this point in the computation.

Note that this does not collapse the wave function.

In contrast to `sample` which samples qubits, this takes a list of cirq.GateOperation instances whose gates are cirq.MeasurementGate instances and then returns a mapping from the key in the measurement gate to the resulting bit strings. Different measurement operations must not act on the same qubits.

**Parameters**

• **measurement_ops** – GateOperation instances whose gates are MeasurementGate instances to be sampled form.

• **repetitions** – The number of samples to take.

• **seed** – A seed for the pseudorandom number generator.

**Returns**: A dictionary from measurement gate key to measurement results. Measurement results are stored in a 2-dimensional numpy array, the first dimension corresponding to the repetition and the second to the actual boolean measurement results (ordered by the qubits being measured.)

**Raises** ValueError – If the operation’s gates are not MeasurementGate instances or a qubit is acted upon multiple times by different operations from measurement_ops.

cirq.DensityMatrixStepResult.set_density_matrix

DensityMatrixStepResult.set_density_matrix(density_matrix_repr: Union[int, numpy.ndarray])

Set the density matrix to a new density matrix.

**Parameters**

• **density_matrix_repr** – If this is an int, the density matrix is set to
• computational basis state corresponding to this state. Otherwise (the)
• this is a np.ndarray it is the full state, either a pure state (if)
• the full density matrix. If it is the pure state it must be the (or)
• size, be normalized (correct)
• to an appropriate dtype for the simulator. If it is a (castable)
• state it must be correctly sized and positive semidefinite (mixed)
• trace one. (with)

cirq.DensityMatrixTrialResult
A SimulationTrialResult for DensityMatrixSimulator runs.

The density matrix that is stored in this result is returned in the computational basis with these basis states defined by the qubit_map. In particular the value in the qubit_map is the index of the qubit, and these are translated into binary vectors where the last qubit is the 1s bit of the index, the second-to-last is the 2s bit of the index, and so forth (i.e. big endian ordering). The density matrix is a 2 ** num_qubits square matrix, with rows and columns ordered by the computational basis as just described.

Example
qubit_map: {QubitA: 0, QubitB: 1, QubitC: 2}
Then the returned density matrix will have (row and column) indices mapped to qubit basis states like the following table

<table>
<thead>
<tr>
<th>QubitA</th>
<th>QubitB</th>
<th>QubitC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**params**

A ParamResolver of settings used for this result.

**measurements**

A dictionary from measurement gate key to measurement results. Measurement results are a numpy ndarray of actual boolean measurement results (ordered by the qubits acted on by the measurement gate.)

**final_simulator_state**

The final simulator state of the system after the trial finishes.

**final_density_matrix**

The final density matrix of the system.

```python
```

Initialize self. See help(type(self)) for accurate signature.

**Methods**

**Attributes**

**qubit_map**

A map from Qid to index used to define the ordering of the basis in

```python
cirq.DensityMatrixTrialResult.qubit_map
```

**property**

```python
DensityMatrixTrialResult.qubit_map
```

A map from Qid to index used to define the ordering of the basis in the result.

**cirqdirac_notation**

```python
cirq.dirac_notation(state: Sequence, decimals: int = 2, qid_shape: Optional[Tuple[int, ...]] = None) → str
```

Returns the wavefunction as a string in Dirac notation.
For example:

```python
state = np.array([1/np.sqrt(2), 1/np.sqrt(2)], dtype=np.complex64)
print(dirac_notation(state)) -> 0.71|0 + 0.71|1
```

**Parameters**

- `state` – A sequence representing a wave function in which the ordering mapping to qubits follows the standard Kronecker convention of `numpy.kron`.
- `decimals` – How many decimals to include in the pretty print.

**Returns** A pretty string consisting of a sum of computational basis kets and non-zero floats of the specified accuracy.

cirq.measure_density_matrix
cirq.measure_density_matrix(density_matrix: numpy.ndarray, indices: List[int], qid_shape: Optional[Tuple[int, ...]] = None, out: numpy.ndarray = None, seed: Union[numpy.random.mtrand.RandomState, int, None] = None) → Tuple[List[int], numpy.ndarray]

Performs a measurement of the density matrix in the computational basis.

This does not modify `density_matrix` unless the optional `out` is `density_matrix`.

**Parameters**

- `density_matrix` – The density matrix to be measured. This matrix is assumed to be positive semidefinite and trace one. The matrix is assumed to be of shape `(2 ** integer, 2 ** integer)` or `(2, 2, ..., 2)`.
- `indices` – Which qubits are measured. The matrix is assumed to be supplied in big endian order. That is the `x`th index of `v`, when expressed as a bitstring, has the largest values in the `0`th index.
- `qid_shape` – The qid shape of the density matrix. Specify this argument when using qudits.
- `out` – An optional place to store the result. If `out` is the same as the `density_matrix` parameter, then `density_matrix` will be modified inline. If `out` is not None, then the result is put into `out`. If `out` is None a new value will be allocated. In all of these cases `out` will be the same as the returned `ndarray` of the method. The shape and dtype of `out` will match that of `density_matrix` if `out` is None, otherwise it will match the shape and dtype of `out`.
- `seed` – A seed for the pseudorandom number generator.

**Returns** A tuple of a list and an `numpy` array. The list is an array of booleans corresponding to the measurement values (ordered by the indices). The `numpy` array is the post measurement matrix. This matrix has the same shape and dtype as the input matrix.

**Raises**

- `ValueError` if the dimension of the matrix is not compatible with a – matrix of n qubits.
- `IndexError` if the indices are out of range for the number of qubits – corresponding to the density matrix.
cirq.measure_state_vector

cirq.measure_state_vector(state: numpy.ndarray, indices: List[int], *, qid_shape: Optional[Tuple[int, ...]] = None, out: numpy.ndarray = None, seed: Union[numpy.random.mtrand.RandomState, int, None] = None) → Tuple[List[int], numpy.ndarray]

Performs a measurement of the state in the computational basis.

This does not modify state unless the optional out is state.

Parameters

• state – The state to be measured. This state is assumed to be normalized. The state must be of size $2^{\text{integer}}$. The state can be of shape $(2^{\text{integer}})$ or $(2, 2, \ldots, 2)$.

• indices – Which qubits are measured. The state is assumed to be supplied in big endian order. That is the xth index of v, when expressed as a bitstring, has the largest values in the 0th index.

• qid_shape – The qid shape of the state vector. Specify this argument when using qudits.

• out – An optional place to store the result. If out is the same as the state parameter, then state will be modified inline. If out is not None, then the result is put into out. If out is None a new value will be allocated. In all of these case out will be the same as the returned ndarray of the method. The shape and dtype of out will match that of state if out is None, otherwise it will match the shape and dtype of state.

• seed – A seed for the pseudorandom number generator.

Returns A tuple of a list and an numpy array. The list is an array of booleans corresponding to the measurement values (ordered by the indices). The numpy array is the post measurement state. This state has the same shape and dtype as the input state.

Raises

• ValueError if the size of state is not a power of 2.

• IndexError if the indices are out of range for the number of qubits – corresponding to the state.

cirq.sample


Simulates sampling from the given circuit or schedule.

Parameters

• program – The circuit or schedule to sample from.

• noise – Noise model to use while running the simulation.

• param_resolver – Parameters to run with the program.

• repetitions – The number of samples to take.

• dtype – The numpy.dtype used by the simulation. Typically one of numpy.complex64 or numpy.complex128. Favors speed over precision by default, i.e. uses numpy.complex64.
• **seed** – The random seed to use for this simulator.

### `cirq.sample_density_matrix`

```python
cirq.sample_density_matrix(density_matrix: numpy.ndarray, indices: List[int], *, qid_shape: Optional[Tuple[int, ...]] = None, repetitions: int = 1, seed: Union[numpy.random.mtrand.RandomState, int, None] = None) → numpy.ndarray
```

Samples repeatedly from measurements in the computational basis.

Note that this does not modify the density_matrix.

**Parameters**

- **density_matrix** – The density matrix to be measured. This matrix is assumed to be positive semidefinite and trace one. The matrix is assumed to be of shape \((2 ** \text{integer}, 2 ** \text{integer})\) or \((2, 2, \ldots, 2)\).

- **indices** – Which qubits are measured. The density matrix rows and columns are assumed to be supplied in big endian order. That is the xth index of v, when expressed as a bitstring, has its largest values in the 0th index.

- **qid_shape** – The qid shape of the density matrix. Specify this argument when using qudits.

- **repetitions** – The number of times to sample the density matrix.

- **seed** – A seed for the pseudorandom number generator.

**Returns** Measurement results with True corresponding to the \(|1\rangle\) state. The outer list is for repetitions, and the inner corresponds to measurements ordered by the supplied qubits. These lists are wrapped as an numpy ndarray.

**Raises**

- **ValueError** – repetitions is less than one or size of matrix is not a power of 2.

- **IndexError** – An index from indices is out of range, given the number of qubits corresponding to the density matrix.

### `cirq.sample_state_vector`

```python
cirq.sample_state_vector(state: numpy.ndarray, indices: List[int], *, qid_shape: Optional[Tuple[int, ...]] = None, repetitions: int = 1, seed: Union[numpy.random.mtrand.RandomState, int, None] = None) → numpy.ndarray
```

Samples repeatedly from measurements in the computational basis.

Note that this does not modify the passed in state.

**Parameters**

- **state** – The multi-qubit wavefunction to be sampled. This is an array of 2 to the power of the number of qubit complex numbers, and so state must be of size \(2 ** \text{integer}\). The state can be a vector of size \(2 ** \text{integer}\) or a tensor of shape \((2, 2, \ldots, 2)\).

- **indices** – Which qubits are measured. The state is assumed to be supplied in big endian order. That is the xth index of v, when expressed as a bitstring, has its largest values in the 0th index.

- **qid_shape** – The qid shape of the state vector. Specify this argument when using qudits.
• **repetitions** – The number of times to sample the state.

• **seed** – A seed for the pseudorandom number generator.

**Returns** Measurement results with True corresponding to the |1 state. The outer list is for repetitions, and the inner corresponds to measurements ordered by the supplied qubits. These lists are wrapped as an numpy ndarray.

**Raises**

• **ValueError** – repetitions is less than one or size of state is not a power of 2.

• **IndexError** – An index from indices is out of range, given the number of qubits corresponding to the state.

cirq.sample_sweep


Runs the supplied Circuit or Schedule, mimicking quantum hardware.

In contrast to run, this allows for sweeping over different parameter values.

**Parameters**

• **program** – The circuit or schedule to simulate.

• **params** – Parameters to run with the program.

• **noise** – Noise model to use while running the simulation.

• **repetitions** – The number of repetitions to simulate, per set of parameter values.

• **dtype** – The numpy.dtype used by the simulation. Typically one of numpy.complex64 or numpy.complex128. Favors speed over precision by default, i.e. uses numpy.complex64.

• **seed** – The random seed to use for this simulator.

**Returns** TrialResult list for this run; one for each possible parameter resolver.

cirq.Sampler

class cirq.Sampler

Something capable of sampling quantum circuits. Simulator or hardware.

```python
__init__(self)
    Initialize self. See help(type(self)) for accurate signature.
```
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>run</code></td>
<td>Samples from the given Circuit or Schedule.</td>
</tr>
<tr>
<td><code>run_async</code></td>
<td>Asynchronously samples from the given Circuit or Schedule.</td>
</tr>
<tr>
<td><code>run_sweep</code></td>
<td>Samples from the given Circuit or Schedule.</td>
</tr>
<tr>
<td><code>run_sweep_async</code></td>
<td>Asynchronously sweeps and samples from the given Circuit or Schedule.</td>
</tr>
<tr>
<td><code>sample</code></td>
<td>Samples the given Circuit or Schedule, producing a pandas data frame.</td>
</tr>
</tbody>
</table>

**cirq.Sampler.run**

```python
Sampler.run(program: Union[cirq.Circuit, cirq.Schedule], param_resolver: cirq.ParamResolverOrSimilarType = None, repetitions: int = 1) → cirq.TrialResult
```

Samples from the given Circuit or Schedule.

By default, the `run_async` method invokes this method on another thread. So this method is supposed to be thread safe.

**Parameters**

- `program` – The circuit or schedule to sample from.
- `param_resolver` – Parameters to run with the program.
- `repetitions` – The number of times to sample.

**Returns** TrialResult for a run.

**cirq.Sampler.run_async**

```python
async Sampler.run_async(program: Union[cirq.Circuit, cirq.Schedule], *, repetitions: int) → cirq.TrialResult
```

Asynchronously samples from the given Circuit or Schedule.

By default, this method calls `run` on another thread and yields the result via the asyncio event loop. However, child classes are free to override it to use other strategies.

**Parameters**

- `program` – The circuit or schedule to sample from.
- `repetitions` – The number of times to sample.

**Returns** An awaitable TrialResult.
cirq.Sampler.run_sweep


Samples from the given Circuit or Schedule.

In contrast to run, this allows for sweeping over different parameter values.

Parameters

• program – The circuit or schedule to sample from.
• params – Parameters to run with the program.
• repetitions – The number of times to sample.

Returns TrialResult list for this run; one for each possible parameter resolver.

cirq.Sampler.run_sweep_async


Asynchronously sweeps and samples from the given Circuit or Schedule.

By default, this method calls run_sweep on another thread and yields the result via the asyncio event loop. However, child classes are free to override it to use other strategies.

Parameters

• program – The circuit or schedule to sample from.
• params – One or more mappings from parameter keys to parameter values to use. For each parameter assignment, repetitions samples will be taken.
• repetitions – The number of times to sample.

Returns An awaitable TrialResult.

cirq.Sampler.sample


Samples the given Circuit or Schedule, producing a pandas data frame.

Parameters

• program – The circuit or schedule to sample from.
• repetitions – The number of times to sample the program, for each parameter mapping.
• **params** – Maps symbols to one or more values. This argument can be a dictionary, a list of dictionaries, a `cirq.Sweep`, a list of `cirq.Sweep`, etc. The program will be sampled `repetition` times for each mapping. Defaults to a single empty mapping.

**Returns** A `pandas.DataFrame` with a row for each sample, and a column for each measurement result as well as a column for each symbolic parameter. There is an also index column containing the repetition number, for each parameter assignment.

**Examples**

```python
circuit = cirq.Circuit(cirq.X(a),
...   cirq.measure(a, key='out'))
print(sampler.sample(circuit, repetitions=4))

out
0 1
1 1
2 1
3 1
```

```python
circuit = cirq.Circuit(cirq.X(a),
...   cirq.CNOT(a, b),
...   cirq.measure(a, b, c, key='out'))
print(sampler.sample(circuit, repetitions=4))

out
0 6
1 6
2 6
3 6
```

```python
circuit = cirq.Circuit(cirq.X(a)**sympy.Symbol('t'),
...   cirq.measure(a, key='out'))
print(sampler.sample(circuit, repetitions=3,
...   params=[{'t': 0}, {'t': 1}])

  t out
0 0 0
1 0 0
2 0 0
0 1 1
1 1 1
2 1 1
```

**cirq.SimulatesFinalState**

**class cirq.SimulatesFinalState**

Simulator that allows access to a quantum computer’s final state.

Implementors of this interface should implement the simulate_sweep method. This simulator only returns the state of the quantum system for the final step of a simulation. This simulator state may be a wave
function, the density matrix, or another representation, depending on the implementation. For simulators that also allow stepping through a circuit see SimulatesIntermediateState.

```python
__init__()
    Initialize self. See help(type(self)) for accurate signature.
```

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>simulate(program[, param_resolver, ...])</code></td>
<td>Simulates the supplied Circuit or Schedule.</td>
</tr>
<tr>
<td><code>simulate_sweep(program, params[, ...])</code></td>
<td>Simulates the supplied Circuit or Schedule.</td>
</tr>
</tbody>
</table>

**cirq.SimulatesFinalState.simulate**

```python
```

This method returns a result which allows access to the entire wave function.

**Parameters**

- **program** – The circuit or schedule to simulate.
- **param_resolver** – Parameters to run with the program.
- **qubit_order** – Determines the canonical ordering of the qubits. This is often used in specifying the initial state, i.e. the ordering of the computational basis states.
- **initial_state** – The initial state for the simulation. The form of this state depends on the simulation implementation. See documentation of the implementing class for details.

**Returns** SimulationTrialResults for the simulation. Includes the final state.
SimulatesFinalState.simulate_sweep

Simulates the supplied Circuit or Schedule.

This method returns a result which allows access to the entire wave function. In contrast to simulate, this allows for sweeping over different parameter values.

Parameters

- program – The circuit or schedule to simulate.
- params – Parameters to run with the program.
- qubit_order – Determines the canonical ordering of the qubits. This is often used in specifying the initial state, i.e. the ordering of the computational basis states.
- initial_state – The initial state for the simulation. The form of this state depends on the simulation implementation. See documentation of the implementing class for details.

Returns List of SimulationTrialResults for this run, one for each possible parameter resolver.

SimulatesIntermediateState

A SimulatesFinalState that simulates a circuit by moments.

Whereas a general SimulatesFinalState may return the entire wave function at the end of a circuit, a SimulatesIntermediateState can simulate stepping through the moments of a circuit.

Implementors of this interface should implement the _simulator_iterator method.
__init__()  
Initialize self. See help(type(self)) for accurate signature.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>simulate(program[, param_resolver, ...])</td>
<td>Simulates the supplied Circuit or Schedule.</td>
</tr>
<tr>
<td>simulate_moment_steps(circuit[, ...])</td>
<td>Returns an iterator of StepResults for each moment simulated.</td>
</tr>
<tr>
<td>simulate_sweep(program, params[, ...])</td>
<td>Simulates the supplied Circuit or Schedule.</td>
</tr>
</tbody>
</table>

cirq.SimulatesIntermediateState.simulate


Simulates the supplied Circuit or Schedule.

This method returns a result which allows access to the entire wave function.

Parameters

- program – The circuit or schedule to simulate.
- param_resolver – Parameters to run with the program.
- qubit_order – Determines the canonical ordering of the qubits. This is often used in specifying the initial state, i.e. the ordering of the computational basis states.
- initial_state – The initial state for the simulation. The form of this state depends on the simulation implementation. See documentation of the implementing class for details.

Returns SimulationTrialResults for the simulation. Includes the final state.
SimulatesIntermediateState.simulate_moment_steps


Returns an iterator of StepResults for each moment simulated.

If the circuit being simulated is empty, a single step result should be returned with the state being set to the initial state.

**Parameters**

- **circuit** – The Circuit to simulate.
- **param_resolver** – A ParamResolver for determining values of Symbols.
- **qubit_order** – Determines the canonical ordering of the qubits. This is often used in specifying the initial state, i.e. the ordering of the computational basis states.
- **initial_state** – The initial state for the simulation. The form of this state depends on the simulation implementation. See documentation of the implementing class for details.

**Returns** Iterator that steps through the simulation, simulating each moment and returning a StepResult for each moment.
Simulates the supplied Circuit or Schedule.

This method returns a result which allows access to the entire wave function. In contrast to simulate, this allows for sweeping over different parameter values.

**Parameters**

- **program** – The circuit or schedule to simulate.
- **params** – Parameters to run with the program.
- **qubit_order** – Determines the canonical ordering of the qubits. This is often used in specifying the initial state, i.e. the ordering of the computational basis states.
- **initial_state** – The initial state for the simulation. The form of this state depends on the simulation implementation. See documentation of the implementing class for details.

**Returns** List of SimulationTrialResults for this run, one for each possible parameter resolver.

### SimulatesIntermediateWaveFunction

A simulator that accesses its wave function as it does its simulation.

Implementors of this interface should implement the _simulator_iterator method.

__init__()

Initialize self. See help(type(self)) for accurate signature.

**Methods**
**compute_amplitudes**

\[
\text{compute_amplitudes}(\text{program, bitstrings[, \ldots]} \quad \text{Computes the desired amplitudes.})
\]

**compute_amplitudes_sweep**

\[
\text{compute_amplitudes_sweep}(\text{program, \ldots[, \ldots]} \quad \text{Computes the desired amplitudes.})
\]

**simulate**

\[
\text{simulate}(\text{program[, param_resolver, \ldots]}) \quad \text{Simulates the supplied Circuit or Schedule.}
\]

**simulate_moment_steps**

\[
\text{simulate_moment_steps}(\text{circuit[, \ldots]}) \quad \text{Returns an iterator of StepResults for each moment simulated.}
\]

**simulate_sweep**

\[
\text{simulate_sweep}(\text{program, params[, \ldots]}) \quad \text{Simulates the supplied Circuit or Schedule.}
\]

---

**cirq.SimulatesIntermediateWaveFunction.compute_amplitudes**


Computes the desired amplitudes.

The initial state is assumed to be the all zeros state.

**Parameters**

- **program** – The circuit or schedule to simulate.
- **bitstrings** – The bitstrings whose amplitudes are desired, input as an integer array where each integer is formed from measured qubit values according to qubit_order from most to least significant qubit, i.e. in big-endian ordering.
- **param_resolver** – Parameters to run with the program.
- **qubit_order** – Determines the canonical ordering of the qubits. This is often used in specifying the initial state, i.e. the ordering of the computational basis states.

**Returns** List of amplitudes.
cirq.SimulatesIntermediateWaveFunction.compute_amplitudes_sweep


Computes the desired amplitudes.

The initial state is assumed to be the all zeros state.

**Parameters**

- **program** – The circuit or schedule to simulate.

- **bitstrings** – The bitstrings whose amplitudes are desired, input as an integer array where each integer is formed from measured qubit values according to `qubit_order` from most to least significant qubit, i.e. in big-endian ordering.

- **params** – Parameters to run with the program.

- **qubit_order** – Determines the canonical ordering of the qubits. This is often used in specifying the initial state, i.e. the ordering of the computational basis states.

**Returns** List of lists of amplitudes. The outer dimension indexes the circuit parameters and the inner dimension indexes the bitstrings.
SimulatesIntermediateWaveFunction.simulate


Simulates the supplied Circuit or Schedule.

This method returns a result which allows access to the entire wave function.

**Parameters**

- **program** – The circuit or schedule to simulate.
- **param_resolver** – Parameters to run with the program.
- **qubit_order** – Determines the canonical ordering of the qubits. This is often used in specifying the initial state, i.e. the ordering of the computational basis states.
- **initial_state** – The initial state for the simulation. The form of this state depends on the simulation implementation. See documentation of the implementing class for details.

**Returns** SimulationTrialResults for the simulation. Includes the final state.
SimulatesIntermediateWaveFunction.simulate_moment_steps


Returns an iterator of StepResults for each moment simulated.

If the circuit being simulated is empty, a single step result should be returned with the state being set to the initial state.

Parameters

- **circuit** – The Circuit to simulate.
- **param_resolver** – A ParamResolver for determining values of Symbols.
- **qubit_order** – Determines the canonical ordering of the qubits. This is often used in specifying the initial state, i.e. the ordering of the computational basis states.
- **initial_state** – The initial state for the simulation. The form of this state depends on the simulation implementation. See documentation of the implementing class for details.

Returns Iterator that steps through the simulation, simulating each moment and returning a StepResult for each moment.
cirq.SimulatesIntermediateWaveFunction.simulate_sweep

SimulatesIntermediateWaveFunction.simulate_sweep(program:
Union[cirq.circuits.circuit.Circuit,
cirq.schedules.schedule.Schedule],
params: Union[Dict[str, float],
cirq.studyresolver.ParamResolver,
cirq.study.sweeps.Sweep, Iter-
erable[Union[Dict[str, float],
cirq.studyresolver.ParamResolver,
cirq.study.sweeps.Sweep]],
None], qubit_order:
Union[cirq.ops.qubit_order.QubitOrder,
Iterable[cirq.ops.raw_types.Qid]]
= <cirq.ops.qubit_order.QubitOrder
object>, initial_state:
Any = None) →
List[cirq.simulatorSimulationTrialResult]

Simulates the supplied Circuit or Schedule.

This method returns a result which allows access to the entire
wave function. In contrast to simulate, this allows for sweeping
over different parameter values.

Parameters

- **program** – The circuit or schedule to simulate.
- **params** – Parameters to run with the program.
- **qubit_order** – Determines the canonical ordering of the qubits. This is often used in
  specifying the initial state, i.e. the ordering of the computational basis states.
- **initial_state** – The initial state for the simulation. The form of this state depends on
  the simulation implementation. See documentation of the implementing class for details.

Returns List of SimulationTrialResults for this run, one for each possible parameter resolver.

cirq.SimulatesSamples

class cirq.SimulatesSamples
  Simulator that mimics running on quantum hardware.
  Implementors of this interface should implement the _run method.

  __init__()
  Initialize self. See help(type(self)) for accurate signature.

Methods
run(program[, param_resolver, repetitions])  Samples from the given Circuit or Schedule.

run_async(program, *, repetitions)  Asynchronously samples from the given Circuit or Schedule.

run_sweep(program, params[, repetitions])  Runs the supplied Circuit or Schedule, mimicking quantum hardware.

run_sweep_async(program, params[, repetitions])  Asynchronously sweeps and samples from the given Circuit or Schedule.

cirq.SimulatesSamples.run

SimulatesSamples.run(program: Union[cirq.Circuit, cirq.Schedule], param_resolver: cirq.ParamResolverOrSimilarType = None, repetitions: int = 1) → cirq.TrialResult

Samples from the given Circuit or Schedule.

By default, the run_async method invokes this method on another thread. So this method is supposed to be thread safe.

Parameters

• program – The circuit or schedule to sample from.
• param_resolver – Parameters to run with the program.
• repetitions – The number of times to sample.

Returns  TrialResult for a run.

cirq.SimulatesSamples.run_async

async SimulatesSamples.run_async(program: Union[cirq.Circuit, cirq.Schedule], *, repetitions: int) → cirq.TrialResult

Asynchronously samples from the given Circuit or Schedule.

By default, this method calls run on another thread and yields the result via the asyncio event loop. However, child classes are free to override it to use other strategies.

Parameters

• program – The circuit or schedule to sample from.
• repetitions – The number of times to sample.

Returns  An awaitable TrialResult.
**cirq.SimulatesSamples.run_sweep**


Runs the supplied Circuit or Schedule, mimicking quantum hardware.

In contrast to run, this allows for sweeping over different parameter values.

**Parameters**

- **program** – The circuit or schedule to simulate.
- **params** – Parameters to run with the program.
- **repetitions** – The number of repetitions to simulate.

**Returns** TrialResult list for this run; one for each possible parameter resolver.

**cirq.SimulatesSamples.run_sweep_async**

async SimulatesSamples.run_sweep_async(program: Union[cirq.Circuit, cirq.Schedule], params: cirq.Sweepable, repetitions: int = 1) → List[cirq.TrialResult]

Asynchronously sweeps and samples from the given Circuit or Schedule.

By default, this method calls run_sweep on another thread and yields the result via the asyncio event loop. However, child classes are free to override it to use other strategies.

**Parameters**

- **program** – The circuit or schedule to sample from.
- **params** – One or more mappings from parameter keys to parameter values to use. For each parameter assignment, repetitions samples will be taken.
- **repetitions** – The number of times to sample.

**Returns** An awaitable TrialResult.

**cirq.SimulatesSamples.sample**

SimulatesSamples.sample(program: Union[cirq.Circuit, cirq.Schedule], *, repetitions: int = 1, params: cirq.Sweepable = None) → pd.DataFrame

Samples the given Circuit or Schedule, producing a pandas data frame.

**Parameters**
• **program** – The circuit or schedule to sample from.

• **repetitions** – The number of times to sample the program, for each parameter mapping.

• **params** – Maps symbols to one or more values. This argument can be a dictionary, a list of dictionaries, a `cirq.Sweep`, a list of `cirq.Sweep`, etc. The program will be sampled repetition times for each mapping. Defaults to a single empty mapping.

**Returns** A `pandas.DataFrame` with a row for each sample, and a column for each measurement result as well as a column for each symbolic parameter. There is also an index column containing the repetition number, for each parameter assignment.

**Examples**

```python
g, b, c = cirq.LineQubit.range(3)
sampler = cirq.Simulator()
circuit = cirq.Circuit(cirq.X(g),
    ...  cirq.measure(g, key='out'))
print(sampler.sample(circuit, repetitions=4))
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```python
circuit = cirq.Circuit(cirq.X(g),
    ...  cirq.CNOT(g, b),
    ...  cirq.measure(g, b, c, key='out'))
print(sampler.sample(circuit, repetitions=4))
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>
```

```python
circuit = cirq.Circuit(cirq.X(g)**sympy.Symbol('t'),
    ...  cirq.measure(g, key='out'))
print(sampler.sample(
    ...  circuit,
    ...  repetitions=3,
    ...  params=[{'t': 0}, {'t': 1}]))
```

```
t |   |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**cirq.SimulationTrialResult**

```python
class cirq.SimulationTrialResult(params: cirq.study.resolver.ParamResolver, measurements: Dict[str, numpy.ndarray], final_simulator_state: Any)
```

Results of a simulation by a `SimulatesFinalState`.

3.1. API Reference  297
Unlike TrialResult these results contain the final simulator_state of the system. This simulator_state is dependent on the simulation implementation and may be, for example, the wave function of the system or the density matrix of the system.

**params**

A ParamResolver of settings used for this result.

**measurements**

A dictionary from measurement gate key to measurement results. Measurement results are a numpy ndarray of actual boolean measurement results (ordered by the qubits acted on by the measurement gate.)

```python
def __init__(params: cirq.study.resolver.ParamResolver, measurements: Dict[str, numpy.ndarray], final_simulator_state: Any) -> None:
    Initialize self. See help(type(self)) for accurate signature.
```

### Methods

#### Attributes

**qubit_map**

A map from Qid to index used to define the ordering of the basis in

cirq.Simulator

```python
class cirq.Simulator(*, dtype: Type[numpy.number] = <class 'numpy.complex64'>, seed: Union[numpy.random.mtrand.RandomState, int, None] = None)
```

A sparse matrix wave function simulator that uses numpy.

This simulator can be applied on circuits that are made up of operations that have a _unitary_ method, or _has_unitary_ and
_apply_unitary_ or _mixture_ methods, are measurements, or support a _decompose_ method that returns operations satisfying these same conditions. That is to say, the operations should follow the cirq.SupportsConsistentApplyUnitary protocol, the cirq.SupportsUnitary protocol, the cirq.SupportsMixture protocol, or the cirq.CompositeOperation protocol. It is also permitted for the circuit to contain measurements which are operations that support cirq.SupportsChannel and cirq.SupportsMeasurementKey

This simulator supports three types of simulation.

Run simulations which mimic running on actual quantum hardware. These simulations do not give access to the wave function (like actual hardware). There are two variations of run methods, one which takes in a single (optional) way to resolve parameterized circuits, and a second which takes in a list or sweep of parameter resolver:

```python
run(circuit, param_resolver, repetitions)
run_sweep(circuit, params, repetitions)
```

The simulation performs optimizations if the number of repetitions is greater than one and all measurements in the circuit are terminal (at the end of the circuit). These methods return TrialResults which contain both the measurement results, but also the parameters used for the parameterized circuit operations. The initial state of a run is always the all 0s state in the computational basis.

By contrast the simulate methods of the simulator give access to the wave function of the simulation at the end of the simulation of the circuit. These methods take in two parameters that the run methods do not: a qubit order and an initial state. The qubit order is necessary because an ordering must be chosen for the kronecker product (see SparseSimulationTrialResult for details of this ordering). The initial state can be either the full wave function, or an integer which represents the initial state of being in a computational basis state for the binary representation of that integer. Similar to run methods, there are two simulate methods that run for single runs or for sweeps across different

Parameters

- **simulate**(circuit, param_resolver, qubit_order, initial_state)
- **simulate_sweep**(circuit, params, qubit_order, initial_state)
The simulate methods in contrast to the run methods do not perform repetitions. The result of these simulations is a `SparseSimulationTrialResult` which contains, in addition to measurement results and information about the parameters that were used in the simulation, access to the state via the `state` method and `StateVectorMixin` methods.

If one wishes to perform simulations that have access to the wave function as one steps through running the circuit there is a generator which can be iterated over and each step is an object that gives access to the wave function. This stepping through a `Circuit` is done on a `Moment` by `Moment` manner.

```python
simulate_moment_steps(circuit, param_resolver, qubit_order, initial_state)
```

One can iterate over the moments via

```python
for step_result in simulate_moments(circuit):
    # do something with the wave function via step_result.state
```

Note also that simulations can be stochastic, i.e. return different results for different runs. The first version of this occurs for measurements, where the results of the measurement are recorded. This can also occur when the circuit has mixtures of unitaries.

See `Simulator` for the definitions of the supported methods.

```python
__init__(*, dtype: Type[numpy.number] = <class 'numpy.complex64'>, seed: Union[numpy.random.mtrand.RandomState, int, None] = None)
```

A sparse matrix simulator.

Parameters

- `dtype` – The `numpy.dtype` used by the simulation. One of `numpy.complex64` or `numpy.complex128`.
- `seed` – The random seed to use for this simulator.

Methods

```
compute_amplitudes(program, bitstrings[, ...])
compute_amplitudes_sweep(program, ...[, ...])
run(program[, param_resolver, repetitions])
run_async(program, *, repetitions)
run_sweep(program, params[, repetitions])
run_sweep_async(program, params[, repetitions])
sample(program, *, repetitions, params)
simulate(program[, param_resolver, ...])
simulate_moment_steps(circuit[, ...])
simulate_sweep(program, params[, ...])
```

Computes the desired amplitudes.

Computes the desired amplitudes.

Samples from the given Circuit or Schedule.

Asynchronously samples from the given Circuit or Schedule.

Runs the supplied Circuit or Schedule, mimicking quantum hardware.

Asynchronously sweeps and samples from the given Circuit or Schedule.

Samples the given Circuit or Schedule, producing a pandas data frame.

Simulates the supplied Circuit or Schedule.

Returns an iterator of StepResults for each moment simulated.

Simulates the supplied Circuit or Schedule.
**cirq.Simulator.compute_amplitudes**


Computes the desired amplitudes.

The initial state is assumed to be the all zeros state.

**Parameters**

- **program** – The circuit or schedule to simulate.
- **bitstrings** – The bitstrings whose amplitudes are desired, input as an integer array where each integer is formed from measured qubit values according to `qubit_order` from most to least significant qubit, i.e. in big-endian ordering.
- **param_resolver** – Parameters to run with the program.
- **qubit_order** – Determines the canonical ordering of the qubits. This is often used in specifying the initial state, i.e. the ordering of the computational basis states.

**Returns** List of amplitudes.

**cirq.Simulator.compute_amplitudes_sweep**


Computes the desired amplitudes.

The initial state is assumed to be the all zeros state.

**Parameters**

- **program** – The circuit or schedule to simulate.
- **bitstrings** – The bitstrings whose amplitudes are desired, input as an integer array where each integer is formed from measured qubit values according to `qubit_order` from most to least significant qubit, i.e. in big-endian ordering.
- **params** – Parameters to run with the program.
- **qubit_order** – Determines the canonical ordering of the qubits. This is often used in specifying the initial state, i.e. the ordering of the computational basis states.
**cirq.Simulator.run**

```python
Simulator.run(program: Union[cirq.Circuit, cirq.Schedule], param_resolver: cirq.ParamResolverOrSimilarType = None, repetitions: int = 1) → cirq.TrialResult
```

Samples from the given Circuit or Schedule.

By default, the `run_async` method invokes this method on another thread. So this method is supposed to be thread safe.

**Parameters**

- **program** – The circuit or schedule to sample from.
- **param_resolver** – Parameters to run with the program.
- **repetitions** – The number of times to sample.

**Returns** TrialResult for a run.

**cirq.Simulator.run_async**

```python
async Simulator.run_async(program: Union[cirq.Circuit, cirq.Schedule], *, repetitions: int) → cirq.TrialResult
```

Asynchronously samples from the given Circuit or Schedule.

By default, this method calls `run` on another thread and yields the result via the asyncio event loop. However, child classes are free to override it to use other strategies.

**Parameters**

- **program** – The circuit or schedule to sample from.
- **repetitions** – The number of times to sample.

**Returns** An awaitable TrialResult.

**cirq.Simulator.run_sweep**

```python
```

Runs the supplied Circuit or Schedule, mimicking quantum hardware.

In contrast to `run`, this allows for sweeping over different parameter
values.

Parameters

- **program** – The circuit or schedule to simulate.
- **params** – Parameters to run with the program.
- **repetitions** – The number of repetitions to simulate.

Returns TrialResult list for this run; one for each possible parameter resolver.

cirq.Simulator.run_sweep_async

```python
async Simulator.run_sweep_async(program: Union[cirq.Circuit, cirq.Schedule],
params: cirq.Sweepable, repetitions: int = 1) → List[cirq.TrialResult]
```

Asynchronously sweeps and samples from the given Circuit or Schedule.

By default, this method calls run_sweep on another thread and yields the result via the asyncio event loop. However, child classes are free to override it to use other strategies.

Parameters

- **program** – The circuit or schedule to sample from.
- **params** – One or more mappings from parameter keys to parameter values to use. For each parameter assignment, **repetitions** samples will be taken.
- **repetitions** – The number of times to sample.

Returns An awaitable TrialResult.

cirq.Simulator.sample

```python
```

Samples the given Circuit or Schedule, producing a pandas data frame.

Parameters

- **program** – The circuit or schedule to sample from.
- **repetitions** – The number of times to sample the program, for each parameter mapping.
- **params** – Maps symbols to one or more values. This argument can be a dictionary, list of dictionaries, a cirq.Sweep, a list of cirq.Sweep, etc. The program will be sampled **repetition** times for each mapping. Defaults to a single empty mapping.

Returns A pandas.DataFrame with a row for each sample, and a column for each measurement result as well as a column for each symbolic parameter. There is an also index column containing the repetition number, for each parameter assignment.
Examples

```python
>>> a, b, c = cirq.LineQubit.range(3)
>>> sampler = cirq.Simulator()
>>> circuit = cirq.Circuit(cirq.X(a),
...     cirq.measure(a, key='out'))
>>> print(sampler.sample(circuit, repetitions=4))
  out
0  1
1  1
2  1
3  1

>>> circuit = cirq.Circuit(cirq.X(a),
...     cirq.CNOT(a, b),
...     cirq.measure(a, b, c, key='out'))
>>> print(sampler.sample(circuit, repetitions=4))
  out
0  6
1  6
2  6
3  6

>>> circuit = cirq.Circuit(cirq.X(a)**sympy.Symbol('t'),
...     cirq.measure(a, key='out'))
>>> print(sampler.sample(circuit,
...     repetitions=3,
...     params=[{'t': 0}, {'t': 1}]))
  t  out
0  0  0
1  0  0
2  0  0
0  1  1
1  1  1
2  1  1
```
cirq.Simulator.simulate

```
Simulator.simulate(program: Union[cirq.circuits.circuit.Circuit,
cirq.schedules.schedule.Schedule], param_resolver: Union[cirq.ParamResolver,
Dict[Union[str, sympy.core.basic.Basic],
Union[float, str, sympy.core.symbol.Symbol], None] = None, qubit_order:
Union[cirq.ops.qubit_order.QubitOrder, Iterable[cirq.ops.raw_types.Qid]]] =
<cirq.ops.qubit_order.QubitOrder object>, initial_state: Any = None) →
cirq.sim.simulator.SimulationTrialResult

Simulates the supplied Circuit or Schedule.
```

This method returns a result which allows access to the entire wave function.

Parameters
- **program** – The circuit or schedule to simulate.
- **param_resolver** – Parameters to run with the program.
- **qubit_order** – Determines the canonical ordering of the qubits. This is often used in specifying the initial state, i.e. the ordering of the computational basis states.
- **initial_state** – The initial state for the simulation. The form of this state depends on the simulation implementation. See documentation of the implementing class for details.

Returns SimulationTrialResults for the simulation. Includes the final state.

**cirq.Simulator.simulate_moment_steps**


Returns an iterator of StepResults for each moment simulated.

If the circuit being simulated is empty, a single step result should be returned with the state being set to the initial state.

Parameters

- **circuit** – The Circuit to simulate.
- **param_resolver** – A ParamResolver for determining values of Symbols.
- **qubit_order** – Determines the canonical ordering of the qubits. This is often used in specifying the initial state, i.e. the ordering of the computational basis states.
- **initial_state** – The initial state for the simulation. The form of this state depends on the simulation implementation. See documentation of the implementing class for details.

Returns Iterator that steps through the simulation, simulating each moment and returning a StepResult for each moment.

**cirq.Simulator.simulate_sweep**


Simulates the supplied Circuit or Schedule.
This method returns a result which allows access to the entire wave function. In contrast to simulate, this allows for sweeping over different parameter values.

**Parameters**

- **program** – The circuit or schedule to simulate.
- **params** – Parameters to run with the program.
- **qubit_order** – Determines the canonical ordering of the qubits. This is often used in specifying the initial state, i.e. the ordering of the computational basis states.
- **initial_state** – The initial state for the simulation. The form of this state depends on the simulation implementation. See documentation of the implementing class for details.

**Returns** List of SimulationTrialResults for this run, one for each possible parameter resolver.

cirq.SparseSimulatorStep

class cirq.SparseSimulatorStep(state_vector, measurements, qubit_map, dtype)

A StepResult that includes StateVectorMixin methods.

__init__(state_vector, measurements, qubit_map, dtype)

Results of a step of the simulator.

**Parameters**

- **qubit_map** – A map from the Qubits in the Circuit to the the index of this qubit for a canonical ordering. This canonical ordering is used to define the state vector (see the state_vector() method).
- **measurements** – A dictionary from measurement gate key to measurement results, ordered by the qubits that the measurement operates on.

**Methods**

- **bloch_vector_of(qubit)**: Returns the bloch vector of a qubit in the state.
- **density_matrix_of([qubits])**: Returns the density matrix of the state.
- **dirac_notation([decimals])**: Returns the state vector as a string in Dirac notation.
- **sample(qubits[, repetitions, seed])**: Samples from the system at this point in the computation.
- **sample_measurement_ops(measurement_ops[, ...])**: Samples from the system at this point in the computation.
- **set_state_vector(state)**
- **state_vector()**: Return the wave function at this point in the computation.

cirq.SparseSimulatorStep.bloch_vector_of

SparseSimulatorStep.bloch_vector_of(qubit: cirq.ops.raw_types.Qid) → numpy.ndarray

Returns the bloch vector of a qubit in the state.

Calculates the bloch vector of the given qubit
in the state given by self.state_vector(), given that
self.state_vector() follows the standard Kronecker convention of
numpy.kron.

**Parameters** qubit – qubit who’s bloch vector we want to find.

**Returns** A length 3 numpy array representing the qubit’s bloch vector.

**Raises**
- *ValueError* – if the size of the state represents more than 25 qubits.
- *IndexError* – if index is out of range for the number of qubits corresponding to the state.

**cirq.SparseSimulatorStep.density_matrix_of**

**SparseSimulatorStep.density_matrix_of**(qubits: List[cirq.ops.raw_types.Qid] = None) → numpy.ndarray

Returns the density matrix of the state.

Calculate the density matrix for the system on the list, qubits. Any
qubits not in the list that are present in self.state_vector() will be
traced out. If qubits is None the full density matrix for
self.state_vector() is returned, given self.state_vector() follows
standard Kronecker convention of numpy.kron.

For example: self.state_vector() = np.array([1/np.sqrt(2),
1/np.sqrt(2)], dtype=np.complex64) qubits = None gives us

$$
\rho = \begin{bmatrix} 0.5 & 0.5 \\
0.5 & 0.5 
\end{bmatrix}
$$

**Parameters** qubits – list containing qubit IDs that you would like to include in the density
matrix (i.e.) qubits that WON’T be traced out.

**Returns** A numpy array representing the density matrix.

**Raises**
- *ValueError* – if the size of the state represents more than 25 qubits.
- *IndexError* – if the indices are out of range for the number of qubits corresponding to the state.

**cirq.SparseSimulatorStep.dirac_notation**

**SparseSimulatorStep.dirac_notation**(decimals: int = 2) → str

Returns the state vector as a string in Dirac notation.

**Parameters** decimals – How many decimals to include in the pretty print.

**Returns** A pretty string consisting of a sum of computational basis kets and non-zero floats of
the specified accuracy.
**cirq.SparseSimulatorStep.sample**

```python
SparseSimulatorStep.sample(qubits: List[cirq.ops.raw_types.Qid], repetitions: int = 1, seed: Union[numpy.random.mtrand.RandomState, int, None] = None) -> numpy.ndarray
```

Samples from the system at this point in the computation.

Note that this does not collapse the wave function.

**Parameters**

- **qubits** – The qubits to be sampled in an order that influence the returned measurement results.
- **repetitions** – The number of samples to take.
- **seed** – A seed for the pseudorandom number generator.

**Returns** Measurement results with True corresponding to the $|1\rangle$ state. The outer list is for repetitions, and the inner corresponds to measurements ordered by the supplied qubits. These lists are wrapped as a numpy ndarray.

**cirq.SparseSimulatorStep.sample_measurement_ops**

```python
SparseSimulatorStep.sample_measurement_ops(measurement_ops: List[cirq.ops.gate_operation.GateOperation], repetitions: int = 1, seed: Union[numpy.random.mtrand.RandomState, int, None] = None) -> Dict[str, numpy.ndarray]
```

Samples from the system at this point in the computation.

Note that this does not collapse the wave function.

In contrast to `sample` which samples qubits, this takes a list of `cirq.GateOperation` instances whose gates are `cirq.MeasurementGate` instances and then returns a mapping from the key in the measurement gate to the resulting bit strings. Different measurement operations must not act on the same qubits.

**Parameters**

- **measurement_ops** – `GateOperation` instances whose gates are `MeasurementGate` instances to be sampled form.
- **repetitions** – The number of samples to take.
- **seed** – A seed for the pseudorandom number generator.

**Returns:** A dictionary from measurement gate key to measurement results. Measurement results are stored in a 2-dimensional numpy array, the first dimension corresponding to the repetition and the second to the actual boolean measurement results (ordered by the qubits being measured.)

**Raises** `ValueError` – If the operation’s gates are not `MeasurementGate` instances or a qubit is acted upon multiple times by different operations from `measurement_ops`. 
The state is returned in the computational basis with these basis states defined by the qubit_map. In particular the value in the qubit_map is the index of the qubit, and these are translated into binary vectors where the last qubit is the 1s bit of the index, the second-to-last is the 2s bit of the index, and so forth (i.e. big endian ordering).

**Example**

qubit_map: {QubitA: 0, QubitB: 1, QubitC: 2}

Then the returned vector will have indices mapped to qubit basis states like the following table

<table>
<thead>
<tr>
<th>QubitA</th>
<th>QubitB</th>
<th>QubitC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Cirq Documentation, Release 0.6.1

```python
__init__(qubit_map: Optional[Dict[cirq.ops.raw_types.Qid, int]] = None, *args, **kwargs)

Parameters
qubit_map -- A map from the Qubits in the Circuit to the index of this qubit for a canonical ordering. This canonical ordering is used to define the state (see the state_vector() method).

Methods

bloch_vector_of(qubit) Returns the bloch vector of a qubit in the state.

density_matrix_of([qubits]) Returns the density matrix of the state.

dirac_notation([decimals]) Returns the state vector as a string in Dirac notation.

state_vector() Return the state vector (wave function).
```

cirq.StateVectorMixin.bloch_vector_of

StateVectorMixin.bloch_vector_of(qubit: cirq.ops.raw_types.Qid) → numpy.ndarray

Returns the bloch vector of a qubit in the state.

Calculates the bloch vector of the given qubit in the state given by self.state_vector(), given that self.state_vector() follows the standard Kronecker convention of numpy.kron.

Parameters
qubit -- qubit who's bloch vector we want to find.

Returns
A length 3 numpy array representing the qubit's bloch vector.

Raises
- ValueError -- if the size of the state represents more than 25 qubits.
- IndexError -- if index is out of range for the number of qubits corresponding to the state.

cirq.StateVectorMixin.density_matrix_of

StateVectorMixin.density_matrix_of(qubits: List[cirq.ops.raw_types.Qid] = None) → numpy.ndarray

Returns the density matrix of the state.

Calculate the density matrix for the system on the list, qubits. Any qubits not in the list that are present in self.state_vector() will be traced out. If qubits is None the full density matrix for self.state_vector() is returned, given self.state_vector() follows standard Kronecker convention of numpy.kron.

For example: self.state_vector() = np.array([1/np.sqrt(2), 1/np.sqrt(2)], dtype=np.complex64) qubits = None gives us

310 Chapter 3. API Reference
\[
\rho = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}
\]

**Parameters**
- **qubits** – list containing qubit IDs that you would like to include in the density matrix (i.e.) qubits that WON’T be traced out.

**Returns**
A numpy array representing the density matrix.

**Raises**
- **ValueError** – if the size of the state represents more than 25 qubits.
- **IndexError** – if the indices are out of range for the number of qubits corresponding to the state.

---

**cirq.StateVectorMixin.dirac_notation**

StateVectorMixin.dirac_notation(*decimals: int = 2*) → str

Returns the state vector as a string in Dirac notation.

**Parameters**
- **decimals** – How many decimals to include in the pretty print.

**Returns**
A pretty string consisting of a sum of computational basis kets and non-zero floats of the specified accuracy.

---

**cirq.StateVectorMixin.state_vector**

abstract StateVectorMixin.state_vector() → numpy.ndarray

Return the state vector (wave function).

The vector is returned in the computational basis with these basis states defined by the `qubit_map`. In particular the value in the `qubit_map` is the index of the qubit, and these are translated into binary vectors where the last qubit is the 1s bit of the index, the second-to-last is the 2s bit of the index, and so forth (i.e. big endian ordering).

**Example**

- `qubit_map: {QubitA: 0, QubitB: 1, QubitC: 2}`

Then the returned vector will have indices mapped to qubit basis states like the following table

<table>
<thead>
<tr>
<th>QubitA</th>
<th>QubitB</th>
<th>QubitC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Attributes

qubit_map

```python
qubit_map
```

**cirq.StateVectorMixin.qubit_map**

**property StateVectorMixin.qubit_map**

cirq.StepResult

circle StepResult (measurements: Optional[Dict[str, List[int]]] = None)

Results of a step of a SimulatesIntermediateState.

```python
measurements
```

A dictionary from measurement gate key to measurement results, ordered by the qubits that the measurement operates on.

```python
__init__(measurements: Optional[Dict[str, List[int]]] = None) → None
```

Initialize self. See help(type(self)) for accurate signature.

Methods

```python
sample(qubits[, repetitions, seed]) Samples from the system at this point in the computation.
```

```python
sample_measurement_ops(measurement_ops[, ...]) Samples from the system at this point in the computation.
```

circle StepResult.sample

```python
abstract StepResult.sample (qubits: List[cirq.ops.raw_types.Qid], repetitions: int = 1, seed:
Union[numpy.random.mtrand.RandomState, int, None] = None) → numpy.ndarray
```

Samples from the system at this point in the computation.

Note that this does not collapse the wave function.

**Parameters**

- **qubits** – The qubits to be sampled in an order that influence the returned measurement results.
- **repetitions** – The number of samples to take.
- **seed** – A seed for the pseudorandom number generator.

**Returns** Measurement results with True corresponding to the |1 state. The outer list is for repetitions, and the inner corresponds to measurements ordered by the supplied qubits. These lists are wrapped as an numpy ndarray.
cirq.StepResult.sample_measurement_ops

```python
cirq.StepResult.sample_measurement_ops(measurement_ops: List[cirq.ops.gate_operation.GateOperation], repetitions: int = 1, seed:
Union[numpy.random.mtrand.RandomState, int, None] = None) → Dict[str, numpy.ndarray]
```

Samples from the system at this point in the computation.

Note that this does not collapse the wave function.

In contrast to `sample` which samples qubits, this takes a list of cirq.GateOperation instances whose gates are cirq.MeasurementGate instances and then returns a mapping from the key in the measurement gate to the resulting bit strings. Different measurement operations must not act on the same qubits.

**Parameters**

- **measurement_ops** – GateOperation instances whose gates are MeasurementGate instances to be sampled form.
- **repetitions** – The number of samples to take.
- **seed** – A seed for the pseudorandom number generator.

**Returns**: A dictionary from measurement gate key to measurement results. Measurement results are stored in a 2-dimensional numpy array, the first dimension corresponding to the repetition and the second to the actual boolean measurement results (ordered by the qubits being measured.)

**Raises**`ValueError` – If the operation’s gates are not MeasurementGate instances or a qubit is acted upon multiple times by different operations from measurement_ops.

cirq.TrialResult

```python
cirq.TrialResult(*, params: cirq.study.resolver.ParamResolver, measurements: Dict[str, numpy.ndarray])
```

The results of multiple executions of a circuit with fixed parameters.
Stored as a Pandas DataFrame that can be accessed through the “data” attribute. The repetition number is the row index and measurement keys are the columns of the DataFrame. Each element is a big endian integer representation of measurement outcomes for the measurement key in that repetition.

**params**

A ParamResolver of settings used when sampling result.

```python
def __init__(*params: cirq.study.resolver.ParamResolver, measurements: Dict[str, numpy.ndarray]) → None
```

**Parameters**

- **params** – A ParamResolver of settings used for this result.
• **measurements** – A dictionary from measurement gate key to measurement results. The value for each key is a 2-D array of booleans, with the first index running over the repetitions, and the second index running over the qubits for the corresponding measurements.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>from_single_parameter_set(*, params, ...)</code></td>
<td>Packages runs of a single parameterized circuit into a TrialResult.</td>
</tr>
<tr>
<td><code>histogram(*, key[, fold_func])</code></td>
<td>Counts the number of times a measurement result occurred.</td>
</tr>
<tr>
<td><code>multi_measurement_histogram(*, keys[, fold_func])</code></td>
<td>Counts the number of times combined measurement results occurred.</td>
</tr>
</tbody>
</table>

#### cirq.TrialResult.from_single_parameter_set

```python
cirq.TrialResult.from_single_parameter_set(*, params: cirq.study.resolver.ParamResolver, measurements: Dict[str, numpy.ndarray]) → cirq.study.trial_result.TrialResult
```

Packages runs of a single parameterized circuit into a TrialResult.

**Parameters**

- **params** – A ParamResolver of settings used for this result.
- **measurements** – A dictionary from measurement gate key to measurement results. The value for each key is a 2-D array of booleans, with the first index running over the repetitions, and the second index running over the qubits for the corresponding measurements.

#### cirq.TrialResult.histogram

```python
cirq.TrialResult.histogram(*, key: Union[str, cirq.Qid, Iterable[cirq.Qid]], fold_func: Callable[Tuple, T] = <function big_endian_bits_to_int>) → collections.Counter
```

Counts the number of times a measurement result occurred.

For example, suppose that:

- `fold_func` is not specified
- `key='abc'`
- the measurement with key 'abc' measures qubits a, b, and c.
- the circuit was sampled 3 times.
- the sampled measurement values were:
  1. `a=1 b=0 c=0`
  2. `a=0 b=1 c=0`
  3. `a=1 b=0 c=0`

Then the counter returned by this method will be:

```python
collections.Counter({
    0b100: 2,
    0b010: 1
})
```
Where ‘0b100’ is binary for ‘4’ and ‘0b010’ is binary for ‘2’. Notice that the bits are combined in a big-endian way by default, with the first measured qubit determining the highest-value bit.

Parameters

- **key** – Keys of measurements to include in the histogram.
- **fold_func** – A function used to convert a sampled measurement result into a countable value. The input is a list of bits sampled together by a measurement. If this argument is not specified, it defaults to interpreting the bits as a big endian integer.

**Returns** A counter indicating how often a measurement sampled various results.

cirq.TrialResult.multi_measurement_histogram


Counts the number of times combined measurement results occurred.

This is a more general version of the ‘histogram’ method. Instead of only counting how often results occurred for one specific measurement, this method tensors multiple measurement results together and counts how often the combined results occurred.

For example, suppose that:

- fold_func is not specified
- keys=['abc', 'd']
- the measurement with key 'abc' measures qubits a, b, and c.
- the measurement with key 'd' measures qubit d.
- the circuit was sampled 3 times.
- the sampled measurement values were:
  1. a=1 b=0 c=0 d=0
  2. a=0 b=1 c=0 d=1
  3. a=1 b=0 c=0 d=0

Then the counter returned by this method will be:

collections.Counter({
    (0b100, 0): 2,
    (0b010, 1): 1
})

Where ‘0b100’ is binary for ‘4’ and ‘0b010’ is binary for ‘2’. Notice that the bits are combined in a big-endian way by default, with the first measured qubit determining the highest-value bit.

Parameters
• **fold_func** – A function used to convert sampled measurement results into countable values. The input is a tuple containing the list of bits measured by each measurement specified by the keys argument. If this argument is not specified, it defaults to returning tuples of integers, where each integer is the big endian interpretation of the bits a measurement sampled.

• **keys** – Keys of measurements to include in the histogram.

**Returns** A counter indicating how often measurements sampled various results.

**Attributes**

data
measurements
repetitions

cirq.TrialResult.data

```python
property TrialResult.data
```

cirq.TrialResult.measurements

```python
property TrialResult.measurements
```

cirq.TrialResult.repetitions

```python
property TrialResult.repetitions
```

cirq.to_valid_density_matrix

cirq.to_valid_density_matrix(density_matrix_rep: Union[int, numpy.ndarray], num_qubits: Optional[int] = None, *, qid_shape: Optional[ Tuple[int, ...]] = None, dtype: Type[numpy.number] = <class 'numpy.complex64'>, atol: float = 1e-07) → numpy.ndarray

Verifies the density_matrix_rep is valid and converts it to ndarray form.

This method is used to support passing a matrix, a vector (wave function), or a computational basis state as a representation of a state.

**Parameters**

• **density_matrix_rep** – If an numpy array, if it is of rank 2 (a matrix), then this is the density matrix. If it is a numpy array of rank 1 (a vector) then this is a wave function. If this is an int, then this is the computation basis state.

• **num_qubits** – The number of qubits for the density matrix. The density_matrix_rep must be valid for this number of qubits.

• **qid_shape** – The qid shape of the state vector. Specify this argument when using qudits.
• **dtype** – The numpy dtype of the density matrix, will be used when creating the state for a computational basis state (int), or validated against if density_matrix_rep is a numpy array.

• **atol** – Numerical tolerance for verifying density matrix properties.

**Returns** A numpy matrix corresponding to the density matrix on the given number of qubits.

**Raises** `ValueError` if the density_matrix_rep is not valid.

---

cirq.to_valid_state_vector

cirq.to_valid_state_vector(state_rep: Union[int, numpy.ndarray], num_qubits: int, *, qid_shape: Optional[Tuple[int, ...]] = None, dtype: Type[numpy.number] = <class 'numpy.complex64'>, atol: float = 1e-07) → numpy.ndarray

Verifies the state_rep is valid and converts it to ndarray form.

This method is used to support passing in an integer representing a computational basis state or a full wave function as a representation of a state.

**Parameters**

• **state_rep** – If an int, the state returned is the state corresponding to a computational basis state. If an numpy array this is the full wave function. Both of these are validated for the given number of qubits, and the state must be properly normalized and of the appropriate dtype.

• **num_qubits** – The number of qubits for the state. The state_rep must be valid for this number of qubits.

• **qid_shape** – The expected qid shape of the state vector. Specify this argument when using qudits.

• **dtype** – The numpy dtype of the state, will be used when creating the state for a computational basis state, or validated against if state_rep is a numpy array.

• **atol** – Numerical tolerance for verifying that the norm of the state is close to 1.

**Returns** A numpy ndarray corresponding to the state on the given number of qubits.

**Raises** `ValueError` if the state is not valid or num_qubits != len(qid_shape)

---

cirq.validate_normalized_state

cirq.validate_normalized_state(state: numpy.ndarray, *, qid_shape: Tuple[int, ...], dtype: Type[numpy.number] = <class 'numpy.complex64'>, atol: float = 1e-07) → None

Validates that the given state is a valid wave function.

---

cirq.validate_probability

cirq.validate_probability(p: float, p_str: str) → float

Validates that a probability is between 0 and 1 inclusively.

**Parameters**
• p – The value to validate.
• p_str – What to call the probability in error messages.

Returns The probability p if the probability if valid.

Raises ValueError if the probability is invalid.

cirq.WaveFunctionSimulatorState

class cirq.WaveFunctionSimulatorState(state_vector: numpy.ndarray, qubit_map:
Dict[cirq.ops.raw_types.Qid, int])

__init__(state_vector: numpy.ndarray, qubit_map: Dict[cirq.ops.raw_types.Qid, int])
Initialize self. See help(type(self)) for accurate signature.

Methods


cirq.WaveFunctionStepResult

class cirq.WaveFunctionStepResult(measurements: Optional[Dict[str, List[int]]] = None)

__init__(measurements: Optional[Dict[str, List[int]]] = None) → None
Initialize self. See help(type(self)) for accurate signature.

Methods

sample(qubits[, repetitions, seed]) Samples from the system at this point in the computation.
sample_measurement_ops(measurement_ops[, ...]) Samples from the system at this point in the computation.

cirq.WaveFunctionStepResult.sample

abstract WaveFunctionStepResult.sample(qubits: List[cirq.ops.raw_types.Qid], repetitions: int = 1, seed:
Union[numpy.random.mtrand.RandomState, int, None] = None) → numpy.ndarray

Samples from the system at this point in the computation.

Note that this does not collapse the wave function.

Parameters

• qubits – The qubits to be sampled in an order that influence the returned measurement results.
• repetitions – The number of samples to take.
• seed – A seed for the pseudorandom number generator.

Returns Measurement results with True corresponding to the \(|1\rangle\) state. The outer list is for repetitions, and the inner corresponds to measurements ordered by the supplied qubits. These
lists are wrapped as an numpy ndarray.

**cirq.WaveFunctionStepResult.sample_measurement_ops**

`WaveFunctionStepResult.sample_measurement_ops(measurement_ops: List[cirq.ops.gate_operation.GateOperation], repetitions: int = 1, seed: Union[numpy.random.mtrand.RandomState, int, None] = None) → Dict[str, numpy.ndarray]`

Samples from the system at this point in the computation.

Note that this does not collapse the wave function.

In contrast to `sample` which samples qubits, this takes a list of `cirq.GateOperation` instances whose gates are `cirq.MeasurementGate` instances and then returns a mapping from the key in the measurement gate to the resulting bit strings. Different measurement operations must not act on the same qubits.

**Parameters**

- `measurement_ops` – `GateOperation` instances whose gates are `MeasurementGate` instances to be sampled form.
- `repetitions` – The number of samples to take.
- `seed` – A seed for the pseudorandom number generator.

**Returns:** A dictionary from measurement gate key to measurement results. Measurement results are stored in a 2-dimensional numpy array, the first dimension corresponding to the repetition and the second to the actual boolean measurement results (ordered by the qubits being measured.)

**Raises** `ValueError` – If the operation’s gates are not `MeasurementGate` instances or a qubit is acted upon multiple times by different operations from `measurement_ops`.

**cirq.WaveFunctionTrialResult**

**class cirq.WaveFunctionTrialResult**


A SimulationTrialResult that includes the StateVectorMixin methods.

**final_state**

The final wave function of the system.

**__init__**


Args:
- `qubit_map`: A map from the Qubits in the Circuit to the the index of this qubit for a canonical ordering. This canonical ordering
is used to define the state (see the state_vector() method).

## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bloch_vector_of(qubit)</code></td>
<td>Returns the bloch vector of a qubit in the state.</td>
</tr>
<tr>
<td><code>density_matrix_of([qubits])</code></td>
<td>Returns the density matrix of the state.</td>
</tr>
<tr>
<td><code>dirac_notation([decimals])</code></td>
<td>Returns the state vector as a string in Dirac notation.</td>
</tr>
<tr>
<td><code>state_vector()</code></td>
<td>Return the wave function at the end of the computation.</td>
</tr>
</tbody>
</table>

### `cirq.WaveFunctionTrialResult.bloch_vector_of`

```python
WaveFunctionTrialResult.bloch_vector_of(qubit: cirq.ops.raw_types.Qid) → numpy.ndarray
```

Returns the bloch vector of a qubit in the state.

Calculates the bloch vector of the given qubit in the state given by self.state_vector(), given that self.state_vector() follows the standard Kronecker convention of numpy.kron.

**Parameters**
- `qubit` – qubit who’s bloch vector we want to find.

**Returns**
- A length 3 numpy array representing the qubit’s bloch vector.

**Raises**
- `ValueError` – if the size of the state represents more than 25 qubits.
- `IndexError` – if index is out of range for the number of qubits corresponding to the state.

### `cirq.WaveFunctionTrialResult.density_matrix_of`

```python
WaveFunctionTrialResult.density_matrix_of(qubits: List[cirq.ops.raw_types.Qid] = None) → numpy.ndarray
```

Returns the density matrix of the state.

Calculate the density matrix for the system on the list, qubits. Any qubits not in the list that are present in self.state_vector() will be traced out. If qubits is None the full density matrix for self.state_vector() is returned, given self.state_vector() follows standard Kronecker convention of numpy.kron.

For example: self.state_vector() = np.array([1/np.sqrt(2), 1/np.sqrt(2)], dtype=np.complex64) qubits = None gives us
$$ \rho = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix} $$

**Parameters**
- **qubits** – list containing qubit IDs that you would like to include in the density matrix (i.e.) qubits that WON'T be traced out.

**Returns**
A numpy array representing the density matrix.

**Raises**
- **ValueError** – if the size of the state represents more than 25 qubits.
- **IndexError** – if the indices are out of range for the number of qubits corresponding to the state.

**cirq.WaveFunctionTrialResult.dirac_notation**

WaveFunctionTrialResult.dirac_notation(decimals: int = 2) → str

Returns the state vector as a string in Dirac notation.

**Parameters**
- **decimals** – How many decimals to include in the pretty print.

**Returns**
A pretty string consisting of a sum of computational basis kets and non-zero floats of the specified accuracy.

**cirq.WaveFunctionTrialResult.state_vector**

WaveFunctionTrialResult.state_vector()

Return the wave function at the end of the computation.

The state is returned in the computational basis with these basis states defined by the qubit_map. In particular the value in the qubit_map is the index of the qubit, and these are translated into binary vectors where the last qubit is the 1s bit of the index, the second-to-last is the 2s bit of the index, and so forth (i.e. big endian ordering).

**Example**

qubit_map: \{QubitA: 0, QubitB: 1, QubitC: 2\}

Then the returned vector will have indices mapped to qubit basis states like the following table:

<table>
<thead>
<tr>
<th>QubitA</th>
<th>QubitB</th>
<th>QubitC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Attributes

---

**qubit_map**

cirq.WaveFunctionTrialResult.qubit_map

---

**property** WaveFunctionTrialResult.qubit_map

### 3.1.12 Parameterization

Handling of parameterized values.

<table>
<thead>
<tr>
<th>Class/Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linspace</td>
<td>A simple sweep over linearly-spaced values.</td>
</tr>
<tr>
<td>ParamResolver</td>
<td>Resolves sympy.Symbols to actual values.</td>
</tr>
<tr>
<td>plot_state_histogram</td>
<td>Plot the state histogram from a single result with repetitions.</td>
</tr>
<tr>
<td>Points</td>
<td>A simple sweep with explicitly supplied values.</td>
</tr>
<tr>
<td>Sweep</td>
<td>A sweep is an iterator over ParamResolvers.</td>
</tr>
<tr>
<td>Sweepable</td>
<td>Union type; Union[X, Y] means either X or Y.</td>
</tr>
<tr>
<td>to_resolvers</td>
<td>Convert a Sweepable to a list of ParamResolvers.</td>
</tr>
<tr>
<td>UnitSweep</td>
<td>A sweep with a single element that assigns no parameter values.</td>
</tr>
</tbody>
</table>

---

**cirq.Linspace**

class cirq.Linspace (key: Union[str, sympy.core.symbol.Symbol], start: float, stop: float, length: int)

A simple sweep over linearly-spaced values.

__init__ (key: Union[str, sympy.core.symbol.Symbol], start: float, stop: float, length: int) → None

Creates a linear-spaced sweep for a given key.

For the given args, assigns to the list of values
start, start + (stop - start) / (length - 1), . . . , stop

Methods

---

**param_tuples()**

An iterator over (key, value) pairs assigning Symbol key to value.

---

**cirq.Linspace.param_tuples**

Linspace.param_tuples () → Iterator[Iterable[Tuple[str, float]]]

An iterator over (key, value) pairs assigning Symbol key to value.

Attributes
keys

The keys for the all of the sympy.Symbols that are resolved.

cirq.Linspace.keys

property Linspace.keys

The keys for the all of the sympy.Symbols that are resolved.

cirq.ParamResolver

class cirq.ParamResolver

(param_dict: Union[cirq.ParamResolver, Dict[Union[str, sympy.core.basic.Basic], Union[float, str, sympy.core.symbol.Symbol]], None] = None)

Resolves sympy.Symbols to actual values.

A Symbol is a wrapped parameter name (str). A ParamResolver is an object that can be used to assign values for these keys.

ParamResolvers are hashable.
	param_dict

A dictionary from the ParameterValue key (str) to its assigned value.

__init__(param_dict: Union[cirq.ParamResolver, Dict[Union[str, sympy.core.basic.Basic], Union[float, str, sympy.core.symbol.Symbol]], None] = None) → None

Initialize self. See help(type(self)) for accurate signature.

Methods

value_of(value)

Attempt to resolve a Symbol, string, or float to its assigned value.

cirq.ParamResolver.value_of

ParamResolver.value_of(value: Union[sympy.core.basic.Basic, float, str]) → Union[float, sympy.core.basic.Basic]

Attempt to resolve a Symbol, string, or float to its assigned value.

Floats are returned without modification. Strings are resolved via the parameter dictionary with exact match only. Otherwise, strings are considered to be sympy.Symbols with the name as the input string.
sympy.Symbols are first checked for exact match in the parameter dictionary. Otherwise, the symbol is resolved using sympy substitution.

Note that passing a formula to this resolver can be slow due to the underlying sympy library. For circuits relying on quick performance, it is recommended that all formulas are flattened before-hand using cirq.flatten or other means so that formula resolution is avoided. If unable to resolve a sympy.Symbol, returns it unchanged. If unable to resolve a name, returns a sympy.Symbol with that name.

**Parameters**

- **value** – The sympy.Symbol or name or float to try to resolve into just a float.

**Returns**

The value of the parameter as resolved by this resolver.

### cirq.plot_state_histogram

**cirq.plot_state_histogram**

(cirq.plot_state_histogram (result: cirq.study.trial_result.TrialResult) \rightarrow numpy.ndarray)

Plot the state histogram from a single result with repetitions.

States is a bitstring representation of all the qubit states in a single result.
Currently this function assumes each measurement gate applies to only a single qubit.

**Parameters**

- **result** – The trial results to plot.

**Returns**

The histogram. A list of values plotted on the y-axis.

### cirq.Points

**class cirq.Points**

(cirq.Points (key: Union[str, sympy.core.symbol.Symbol], points: Sequence[float])

A simple sweep with explicitly supplied values.

**__init__**

(__init__ (key: Union[str, sympy.core.symbol.Symbol], points: Sequence[float]) \rightarrow None)

Initialize self. See help(type(self)) for accurate signature.

**Methods**

- **param_tuples()**

  An iterator over (key, value) pairs assigning Symbol key to value.

**cirq.Points.param_tuples**

Points.param_tuples () \rightarrow Iterator[Iterable[str, float]]

An iterator over (key, value) pairs assigning Symbol key to value.
Attributes

<table>
<thead>
<tr>
<th>keys</th>
<th>The keys for the all of the sympy.Symbols that are resolved.</th>
</tr>
</thead>
</table>

**cirq.Points.keys**

**property Points.keys**
The keys for the all of the sympy.Symbols that are resolved.

**cirq.Sweep**

**class cirq.Sweep**
A sweep is an iterator over ParamResolvers.

A ParamResolver assigns values to Symbols. For sweeps, each ParamResolver must specify the same Symbols that are assigned. So a sweep is a way to iterate over a set of different values for a fixed set of Symbols. This is useful for a circuit, where there are a fixed set of Symbols, and you want to iterate over an assignment of all values to all symbols.

For example, a sweep can explicitly assign a set of equally spaced points between two endpoints using a Linspace,
```
sweep = Linspace("angle", start=0.0, end=2.0, length=10)
```
This can then be used with a circuit that has an ‘angle’ sympy.Symbol to run simulations multiple simulations, one for each of the values in the sweep
```
result = simulator.run_sweep(program=circuit, params=sweep)
```

Sweeps support Cartesian and Zip products using the ‘*’ and ‘+’ operators, see the Product and Zip documentation.

```python
__init__()
Initialize self. See help(type(self)) for accurate signature.
```

**Methods**

<table>
<thead>
<tr>
<th>param_tuples()</th>
<th>An iterator over (key, value) pairs assigning Symbol key to value.</th>
</tr>
</thead>
</table>

**cirq.Sweep.param_tuples**

**abstract Sweep.param_tuples()** → Iterator[Iterable[Tuple[str, float]]]
An iterator over (key, value) pairs assigning Symbol key to value.
Attributes

<table>
<thead>
<tr>
<th>keys</th>
<th>The keys for all of the sympy.Symbols that are resolved.</th>
</tr>
</thead>
</table>

cirq.Sweep.keys

**abstract property** Sweep.keys
The keys for all of the sympy.Symbols that are resolved.

cirq.Sweepable


Union type; Union[X, Y] means either X or Y.

To define a union, use e.g. Union[int, str]. Details:

- The arguments must be types and there must be at least one.
- None as an argument is a special case and is replaced by type(None).
- Unions of unions are flattened, e.g.:
  \[
  \text{Union[Union[int, str], float]} = \text{Union[int, str, float]}
  \]
- Unions of a single argument vanish, e.g.:
  \[
  \text{Union[int]} = \text{int} \quad \# \text{The constructor actually returns int}
  \]
- Redundant arguments are skipped, e.g.:
  \[
  \text{Union[int, str, int]} = \text{Union[int, str]}
  \]
- When comparing unions, the argument order is ignored, e.g.:
  \[
  \text{Union[int, str]} = \text{Union[str, int]}
  \]
- When two arguments have a subclass relationship, the least derived argument is kept, e.g.:
  \[
  \begin{align*}
  \text{class Employee: pass} \\
  \text{class Manager(Employee): pass}
  \end{align*}
  \[
  \text{Union[int, Employee, Manager]} = \text{Union[int, Employee]}
  \]
- Similar for object:
  \[
  \text{Union[int, object]} = \text{object}
  \]
- You cannot subclass or instantiate a union.
- You can use Optional[X] as a shorthand for Union[X, None].
cirq.to_resolvers


Convert a Sweepable to a list of ParamResolvers.

cirq.UnitSweep

cirq.UnitSweep = cirq.UnitSweep

A sweep with a single element that assigns no parameter values.

This is useful as a base sweep, instead of special casing None.

3.1.13 Magic Method Protocols

Utility methods for accessing generic functionality exposed by some gates, operations, and other types.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply_channel(val, args[, default])</td>
<td>High performance evolution under a channel evolution.</td>
</tr>
<tr>
<td>apply_unitary(unitary_value, args[, default])</td>
<td>High performance left-multiplication of a unitary effect onto a tensor.</td>
</tr>
<tr>
<td>approx_eq(val, other, *[atol])</td>
<td>Approximately compares two objects.</td>
</tr>
<tr>
<td>channel(val[, default])</td>
<td>Returns a list of matrices describing the channel for the given value.</td>
</tr>
<tr>
<td>circuit_diagram_info(val[, args, default, ...])</td>
<td>Requests information on drawing an operation in a circuit diagram.</td>
</tr>
<tr>
<td>decompose(val, *[intercepting_decomposer, ...])</td>
<td>Recursively decomposes a value into cirq.Operations meeting a criteria.</td>
</tr>
<tr>
<td>decompose_once(val[, default])</td>
<td>Decomposes a value into operations, if possible.</td>
</tr>
<tr>
<td>decompose_once_with_qubits(val, qubits[, ...])</td>
<td>Decomposes a value into operations on the given qubits.</td>
</tr>
<tr>
<td>has_channel(val)</td>
<td>Returns whether the value has a channel representation.</td>
</tr>
<tr>
<td>has_mixture(val)</td>
<td>Returns whether the value has a mixture representation.</td>
</tr>
<tr>
<td>has_mixture_channel(val)</td>
<td>Returns whether the value has a mixture channel representation.</td>
</tr>
<tr>
<td>has_unitary(val)</td>
<td>Determines whether the value has a unitary effect.</td>
</tr>
<tr>
<td>inverse(val[, default])</td>
<td>Returns the inverse val**-1 of the given value, if defined.</td>
</tr>
<tr>
<td>is_measurement(val)</td>
<td>Returns whether or not the given value is a measurement.</td>
</tr>
<tr>
<td>is_parameterized(val)</td>
<td>Returns whether the object is parameterized with any Symbols.</td>
</tr>
<tr>
<td>measurement_key(val[, default])</td>
<td>Get the measurement key for the given value.</td>
</tr>
<tr>
<td>mixture(val[, default])</td>
<td>Return a sequence of tuples representing a probabilistic combination.</td>
</tr>
<tr>
<td>mixture_channel(val[, default])</td>
<td>Return a sequence of tuples for a channel that is a mixture of unitaries.</td>
</tr>
<tr>
<td>mul(lhs, rhs[, default])</td>
<td>Returns lhs * rhs, or else a default if the operator is not implemented.</td>
</tr>
</tbody>
</table>

Continued on next page
num_qubits(val[, default])

Returns the number of qubits, qudits, or qids val operates on.

pauli Expansion(val, *[, default, atol])

Returns coefficients of the expansion of val in the Pauli basis.

phase_by(val, phase_turns, qubit_index[, . . . ])

Returns a phased version of the effect.

pow(val, exponent[, default])

Returns \( \text{val}^{\text{factor}} \) of the given value, if defined.

qasm(val, *[, args, qubits, default])

Returns QASM code for the given value, if possible.

qid_shape(val[, default])

Returns a tuple describing the number of quantum levels of each

resolve_parameters(val, param_resolver)

Resolves symbol parameters in the effect using the param resolver.

trace_distance_bound(val)

Returns a maximum on the trace distance between this effect's input

unitary(val[, default])

Returns a unitary matrix describing the given value.

validate_mixture(supports_mixture)

Validates that the mixture's tuple are valid probabilities.

cirq.apply_channel

cirq.apply_channel(val: Any, args: cirq.protocols.apply_channel_protocol.ApplyChannelArgs, default: TDefault = array([], dtype=float64) → Union[numpy.ndarray, TDefault]

High performance evolution under a channel evolution.

If \( \text{val} \) defines an \_apply_channel\_ method, that method will be used to apply \( \text{val} \)'s channel effect to the target tensor. Otherwise, if \( \text{val} \) defines an \_apply_unitary\_ method, that method will be used to apply \( \text{val} \)s channel effect to the target tensor. Otherwise, if \( \text{val} \) returns a non-default channel with cirq.channel, that channel will be applied using a generic method. If none of these cases apply, an exception is raised or the specified default value is returned.

Parameters

- **val** – The value with a channel to apply to the target.
- **args** – A mutable cirq.ApplyChannelArgs object describing the target tensor, available workspace, and left and right axes to operate on. The attributes of this object will be mutated as part of computing the result.
- **default** – What should be returned if \( \text{val} \) doesn’t have a channel. If not specified, a TypeError is raised instead of returning a default value.

Returns

If the receiving object is not able to apply a channel, the specified default value is returned (or a TypeError is raised). If this occurs, then \( \text{target_tensor} \) should not have been mutated.

If the receiving object was able to work inline, directly mutating \( \text{target_tensor} \) it will return \( \text{target_tensor} \). The caller is responsible for checking if the result is \( \text{target_tensor} \).
If the receiving object wrote its output over `out_buffer`, the result will be `out_buffer`. The caller is responsible for checking if the result is `out_buffer` (and e.g. swapping the buffer for the target tensor before the next call).

Note that it is an error for the return object to be either of the auxiliary buffers, and the method will raise an AssertionError if this contract is violated.

The receiving object may also write its output over a new buffer that it created, in which case that new array is returned.

**Raises**

- **TypeError** – `val` doesn’t have a channel and `default` wasn’t specified.
- **ValueError** – Different left and right shapes of `args.target_tensor` selected by `left_axes` and `right_axes` or `qid_shape(val)` doesn’t equal the left and right shapes.
- **AssertionError** – `_apply_channel_` returned an auxiliary buffer.

[cirq.apply_unitary](#)

cirq.apply_unitary

cirq.apply_unitary (unitary_value: Any, args: cirq.protocols.apply_unitary_protocol.ApplyUnitaryArgs, default: TDefault = array([], dtype=float64)) → Union[numpy.ndarray, TDefault]

High performance left-multiplication of a unitary effect onto a tensor.

Applies the unitary effect of `unitary_value` to the tensor specified in `args` by using the following strategies:

A. Try to use `unitary_value._apply_unitary_(args)`.
   Case a) Method not present or returns `NotImplemented`. Continue to next strategy.
   Case b) Method returns `None`. Conclude `unitary_value` has no unitary effect.
   Case c) Method returns a numpy array. Forward the successful result to the caller.

B. Try to use `unitary_value._unitary_()`.
   Case a) Method not present or returns `NotImplemented`. Continue to next strategy.
   Case b) Method returns `None`. Conclude `unitary_value` has no unitary effect.
   Case c) Method returns a numpy array. Multiply the matrix onto the target tensor and return to the caller.

C. Try to use `unitary_value._decompose_()`.
   Case a) Method not present or returns `NotImplemented` or `None`. Continue to next strategy.
   Case b) Method returns an OP_TREE. Delegate to cirq.apply_unitaries.
D. Conclude that `unitary_value` has no unitary effect.

The order that the strategies are tried depends on the number of qubits being operated on. For small numbers of qubits (4 or less) the order is ABCD. For larger numbers of qubits the order is ACBD (because it is expected that decomposing will outperform generating the raw matrix).

**Parameters**

- `unitary_value` – The value with a unitary effect to apply to the target.
- `args` – A mutable `cirq.ApplyUnitaryArgs` object describing the target tensor, available workspace, and axes to operate on. The attributes of this object will be mutated as part of computing the result.
- `default` – What should be returned if `unitary_value` doesn’t have a unitary effect. If not specified, a `TypeError` is raised instead of returning a default value.

**Returns**

If the receiving object does not have a unitary effect, then the specified default value is returned (or a `TypeError` is raised). If this occurs, then `target_tensor` should not have been mutated.

Otherwise the result is the `np.ndarray` instance storing the result. This may be `args.target_tensor`, `args.available_workspace`, or some other numpy array. It is the caller’s responsibility to correctly handle all of these cases. In all cases `args.target_tensor` and `args.available_buffer` may have been mutated.

**Raises** `TypeError` – `unitary_value` doesn’t have a unitary effect and `default` wasn’t specified.

cirq.approx_eq
cirq.approx_eq(val: Any, other: Any, *, atol: Union[int, float] = 1e-08) → bool

Approximately compares two objects.

If `val` implements `SupportsApproxEquality` protocol then it is invoked and takes precedence over all other checks:

- For primitive numeric types `int` and `float` approximate equality is delegated to `math.isclose()`.
- For complex primitive type the real and imaginary parts are treated independently and compared using `math.isclose()`.
- For `val` and `other` both iterable of the same length, consecutive elements are compared recursively. Types of `val` and `other` does not necessarily needs to match each other. They just need to be iterable and have the same structure.

**Parameters**

- `val` – Source object for approximate comparison.
- `other` – Target object for approximate comparison.
- `atol` – The minimum absolute tolerance. See `np.isclose()` documentation for details. Defaults to 1e-8 which matches `np.isclose()` default absolute tolerance.
Returns True if objects are approximately equal, False otherwise.

**cirq.channel**

cirq.channel(val: Any, default: Any = (array([], dtype=float64), )) → Union[Tuple[numpy.ndarray], Sequence[TDefault]]

Returns a list of matrices describing the channel for the given value.

These matrices are the terms in the operator sum representation of a quantum channel. If the returned matrices are \{A_0,A_1,\ldots,A_{r-1}\}, then this describes the channel:

\[
\rho \rightarrow \sum_{k=0}^{r-1} A_0 \rho A_0^\dagger
\]

These matrices are required to satisfy the trace preserving condition

\[
\sum_{k=0}^{r-1} A_i^\dagger A_i = I
\]

where I is the identity matrix. The matrices A_i are sometimes called Krauss or noise operators.

**Parameters**

- **val** – The value to describe by a channel.
- **default** – Determines the fallback behavior when val doesn’t have a channel. If default is not set, a TypeError is raised. If default is set to a value, that value is returned.

**Returns** If val has a _channel_ method and its result is not NotImplemented, that result is returned. Otherwise, if val has a _mixture_ method and its results is not NotImplement a tuple made up of channel corresponding to that mixture being a probabilistic mixture of unitaries is returned. Otherwise, if val has a _unitary_ method and its result is not NotImplemented a tuple made up of that result is returned. Otherwise, if a default value was specified, the default value is returned.

**Raises** TypeError – val doesn’t have a _channel_ or _unitary_ method (or that method returned NotImplemented) and also no default value was specified.

**cirq.circuit_diagram_info**

cirq.circuit_diagram_info(val: Any, args: Optional[cirq.protocols.circuit_diagram_info_protocol.CircuitDiagramInfoArgs] = None, default=cirq.CircuitDiagramInfo(wire_symbols=(), exponent=1, connected=True, exponent_qubit_index=None, auto_exponent_parens=True))

Requests information on drawing an operation in a circuit diagram.

Calls circuit_diagram_info on val. If val doesn’t have circuit_diagram_info, or it returns NotImplemented, that indicates that diagram information is not available.

**Parameters**

- **val** – The operation or gate that will need to be drawn.
- **args** – A CircuitDiagramInfoArgs describing the desired drawing style.
• **default** – A default result to return if the value doesn’t have circuit diagram information. If not specified, a TypeError is raised instead.

**Returns**

If `val` has no `_circuit_diagram_info_` method or it returns `NotImplemented`, then `default` is returned (or a TypeError is raised if no `default` is specified).

Otherwise, the value returned by `_circuit_diagram_info_` is returned.

**Raises** **TypeError** – `val` doesn’t have circuit diagram information and `default` was not specified.

cirq.decompose


Recursively decomposes a value into `cirq.Operation`s meeting a criteria.

**Parameters**

• **val** – The value to decompose into operations.

• **intercepting_decomposer** – An optional method that is called before the default decomposer (the value’s `_decompose_` method). If `intercepting_decomposer` is specified and returns a result that isn’t `NotImplemented` or `None`, that result is used. Otherwise the decomposition falls back to the default decomposer.

  Note that `val` will be passed into `intercepting_decomposer`, even if `val` isn’t a `cirq.Operation`.

• **fallback_decomposer** – An optional decomposition that used after the `intercepting_decomposer` and the default decomposer (the value’s `_decompose_` method) both fail.

• **keep** – A predicate that determines if the initial operation or intermediate decomposed operations should be kept or else need to be decomposed further. If `keep` isn’t specified, it defaults to “value can’t be decomposed anymore”.

• **on_stuck_raise** – If there is an operation that can’t be decomposed and also can’t be kept, `on_stuck_raise` is used to determine what error to raise. `on_stuck_raise` can either directly be an `Exception`, or a method that takes the problematic operation and returns an `Exception`. If `on_stuck_raise` is set to `None` or a method that returns `None`, undecomposable operations are simply silently kept. `on_stuck_raise` defaults to a `ValueError` describing the unwanted undecomposable operation.

**Returns** A list of operations that the given value was decomposed into. If `on_stuck_raise` isn’t set to `None`, all operations in the list will satisfy the predicate specified by `keep`.

**Raises**

• **TypeError** – `val` isn’t a `cirq.Operation` and can’t be decomposed even once. (So it’s not possible to return a list of operations.)

• **ValueError** – Default type of error raised if there’s an undecomposable operation that doesn’t satisfy the given `keep` predicate.

• **TError** – Custom type of error raised if there’s an undecomposable operation that doesn’t satisfy the given `keep` predicate.
cirq.decompose_once

**cirq.decompose_once** *(val: Any, default=([], ), *args, **kwargs)*

Decomposes a value into operations, if possible.

This method decomposes the value exactly once, instead of decomposing it and then continuing to decomposing the decomposed operations recursively until some criteria is met (which is what **cirq.decompose** does).

**Parameters**

- **val** – The value to call _decompose_ on, if possible.
- **default** – A default result to use if the value doesn’t have a _decompose_ method or that method returns *NotImplemented* or *None*. If not specified, undecomposable values cause a *TypeError*.
- **args** – Positional arguments to forward into the _decompose_ method of val. For example, this is used to tell gates what qubits they are being applied to.
- **kwargs** – Keyword arguments to forward into the _decompose_ method of val.

**Returns**

The result of *val._decompose_(qubits)*, if *val* has a _decompose_ method and it didn’t return *NotImplemented* or *None*. Otherwise *default* is returned, if it was specified. Otherwise an error is raised.

*TypeError*: *val* didn’t have a _decompose_ method (or that method returned *NotImplemented* or *None*) and *default* wasn’t set.

cirq.decompose_once_with_qubits

**cirq.decompose_once_with_qubits** *(val: Any, qubits: Iterable[cirq.Qid], default=([],))*

Decomposes a value into operations on the given qubits.

This method is used when decomposing gates, which don’t know which qubits they are being applied to unless told. It decomposes the gate exactly once, instead of decomposing it and then continuing to decomposing the decomposed operations recursively until some criteria is met.

**Parameters**

- **val** – The value to call _decompose_(qubits) on, if possible.
- **qubits** – The value to pass into the named qubits parameter of val._decompose_.
- **default** – A default result to use if the value doesn’t have a _decompose_ method or that method returns *NotImplemented* or *None*. If not specified, undecomposable values cause a *TypeError*.

**Returns**

The result of *val._decompose_(qubits)*, if *val* has a _decompose_ method and it didn’t return *NotImplemented* or *None*. Otherwise *default* is returned, if it was specified. Otherwise an error is raised.
**TypeError:** `val` didn’t have a `_decompose_` method (or that method returned `NotImplemented` or `None`) and `default` wasn’t set.

### `cirq.has_channel`

```python
cirq.has_channel(val: Any) → bool
```

Returns whether the value has a channel representation.

**Returns** If `val` has a `_has_channel_` method and its result is not `NotImplemented`, that result is returned. Otherwise, if `val` has a `_has_mixture_` method and its result is not `NotImplemented`, that result is returned. Otherwise if `val` has a `_has_unitary_` method and its results is not `NotImplemented`, that result is returned. Otherwise, if the value has a `_channel_` method return if that has a non-default value. Returns False if none of these functions exists.

### `cirq.has_mixture`

```python
cirq.has_mixture(val: Any) → bool
```

Returns whether the value has a mixture representation.

**Returns** If `val` has a `_has_mixture_` method and its result is not `NotImplemented`, that result is returned. Otherwise, if the value has a `_mixture_` method return True if that has a non-default value. Returns False if neither function exists.

### `cirq.has_mixture_channel`

```python
cirq.has_mixture_channel(val: Any) → bool
```

Returns whether the value has a mixture channel representation.

In contrast to `has_mixture` this method falls back to checking whether the value has a unitary representation via `has_channel`.

**Returns** If `val` has a `_has_mixture_` method and its result is not `NotImplemented`, that result is returned. Otherwise, if `val` has a `_has_unitary_` method and its results is not `NotImplemented`, that result is returned. Otherwise, if the value has a `_mixture_` method that is not a non-default value, True is returned. Returns False if none of these functions.

### `cirq.has_unitary`

```python
cirq.has_unitary(val: Any) → bool
```

Determines whether the value has a unitary effect.

Determines whether `val` has a unitary effect by attempting the following strategies:

1. Try to use `val.has_unitary()`.
   - Case a) Method not present or returns `NotImplemented`. Inconclusive.
   - Case b) Method returns `True`.
Unitary.
Case c) Method returns False.
Not unitary.

2. Try to use `val._decompose_()`.
   Case a) Method not present or returns `NotImplemented` or `None`.
   Inconclusive.
   Case b) Method returns an OP_TREE containing only unitary operations.
   Unitary.
   Case c) Method returns an OP_TREE containing non-unitary operations.
   Not Unitary.

3. Try to use `val._apply_unitary_(args)`.
   Case a) Method not present or returns `NotImplemented`.
   Inconclusive.
   Case b) Method returns a numpy array.
   Unitary.
   Case c) Method returns `None`.
   Not unitary.

4. Try to use `val._unitary_()`.
   Case a) Method not present or returns `NotImplemented`.
   Continue to next strategy.
   Case b) Method returns a numpy array.
   Unitary.
   Case c) Method returns `None`.
   Not unitary.

It is assumed that, when multiple of these strategies give a conclusive result, that these results will all be consistent with each other. If all strategies are inconclusive, the value is classified as non-unitary.

Parameters `value that may or may not have a unitary effect`. *(The)*

Returns Whether or not `val` has a unitary effect.

cirq.inverse

cirq.inverse(val: Any, default: Any = ([], )) → Any

Returns the inverse `val**-1` of the given value, if defined.

An object can define an inverse by defining a `pow(self, exponent)` method that returns something besides `NotImplemented` when given the exponent -1. The inverse of iterables is by default defined to be the iterable’s items, each inverted, in reverse order.

Parameters

- `val` – The value (or iterable of invertible values) to invert.
• **default** – Determines the fallback behavior when *val* doesn’t have an inverse defined. If *default* is not set, a `TypeError` is raised. If *default* is set to a value, that value is returned.

**Returns** If *val* has a `__pow__` method that returns something besides `NotImplemented` when given an exponent of `-1`, that result is returned. Otherwise, if *val* is iterable, the result is a tuple with the same items as *val* but in reverse order and with each item inverted. Otherwise, if a *default* argument was specified, it is returned.

**Raises** `TypeError` – *val* doesn’t have a `__pow__` method, or that method returned `NotImplemented` when given `-1`. Furthermore *val* isn’t an iterable containing invertible items. Also, no *default* argument was specified.

### `cirq.is_measurement`

`cirq.is_measurement(val: Any) → bool`

Returns whether or not the given value is a measurement.

A measurement must implement the `measurement_key` protocol and have a channel, as represented by the `has_channel` protocol returning true.

### `cirq.is_parameterized`

`cirq.is_parameterized(val: Any) → bool`

Returns whether the object is parameterized with any Symbols.

A value is parameterized when it has an `__is_parameterized__` method and that method returns a truthy value, or if the value is an instance of `sympy.Basic`.

**Returns** True if the gate has any unresolved Symbols and False otherwise. If no implementation of the magic method above exists or if that method returns `NotImplemented`, this will default to False.

### `cirq.measurement_key`

`cirq.measurement_key(val: Any, default: Any = ([]),)`

Get the measurement key for the given value.

**Parameters**

• **val** – The value which has the measurement key.

• **default** – Determines the fallback behavior when *val* doesn’t have a measurement key. If *default* is not set, a `TypeError` is raised. If default is set to a value, that value is returned if the value does not have `measurement_key`.

**Returns** If *val* has a `measurement_key` method and its result is not `NotImplemented`, that result is returned. Otherwise, if a default value was specified, the default value is returned.

**Raises** `TypeError` – *val* doesn’t have a `measurement_key` method (or that method returned `NotImplemented`) and also no default value was specified.
cirq.mixture

\[
\text{cirq.mixture} \; (\text{val}: \text{Any}, \text{default}: \text{Any} = \langle 0.0, , \rangle) \rightarrow \text{Sequence}[\text{Tuple}[\text{float, Any}]]
\]

Return a sequence of tuples representing a probabilistic combination.

A mixture is described by an iterable of tuples of the form

\[
\langle \text{probability of object, object} \rangle
\]

The probability components of the tuples must sum to 1.0 and be between 0 and 1 (inclusive).

**Parameters**

- **val** – The value whose mixture is being computed.
- **default** – A default value if val does not support mixture.

**Returns** An iterable of tuples of size 2. The first element of the tuple is a probability (between 0 and 1) and the second is the object that occurs with that probability in the mixture. The probabilities will sum to 1.0.

---

cirq.mixture_channel

\[
\text{cirq.mixture_channel} \; (\text{val}: \text{Any}, \text{default}: \text{Any} = \langle 0.0, , \rangle) \rightarrow \text{Sequence}[\text{Tuple}[\text{float, numpy.ndarray}]]
\]

Return a sequence of tuples for a channel that is a mixture of unitaries.

In contrast to mixture this method falls back to unitary if \_mixture\_ is not implemented.

A mixture channel is described by an iterable of tuples of the form

\[
\langle \text{probability of unitary, unitary} \rangle
\]

The probability components of the tuples must sum to 1.0 and be between 0 and 1 (inclusive) and the unitary must be a unitary matrix.

**Parameters**

- **val** – The value whose mixture_channel is being computed.
- **default** – A default value if val does not support mixture.

**Returns** An iterable of tuples of size 2. The first element of the tuple is a probability (between 0 and 1) and the second is the unitary that occurs with that probability. The probabilities will sum to 1.0.
### `cirq.mul`

```python
cirq.mul(lhs: Any, rhs: Any, default: Any = ([], )) → Any
Returns lhs * rhs, or else a default if the operator is not implemented.
```

This method is mostly used by `pow` methods trying to return `NotImplemented` instead of causing a `TypeError`.

**Parameters**
- `lhs` – Left hand side of the multiplication.
- `rhs` – Right hand side of the multiplication.
- `default` – Default value to return if the multiplication is not defined. If not default is specified, a type error is raised when the multiplication fails.

**Returns**
The product of the two inputs, or else the default value if the product is not defined, or else raises a `TypeError` if no default is defined.

**Raises**
- `TypeError` – `lhs` doesn’t have `__mul__` or it returned `NotImplemented` AND `lhs` doesn’t have `__rmul__` or it returned `NotImplemented` AND a default value isn’t specified.

### `cirq.num_qubits`

```python
cirq.num_qubits(val: Any, *, default: Union[cirq.value.linear_dict.LinearDict[str], TDefault] = cirq.LinearDict({}), atol: float = 1e-09) → Union[int, TDefault]
Returns the number of qubits, qudits, or qids `val` operates on.
```

**Parameters**
- `val` – The value to get the number of qubits from.
- `default` – Determines the fallback behavior when `val` doesn’t have a number of qubits. If `default` is not set, a `TypeError` is raised. If default is set to a value, that value is returned.

**Returns**
If `val` has a `_num_qubits_` method and its result is not `NotImplemented`, that result is returned. Otherwise, if `val` has a `_qid_shape_` method, the number of qubits is computed from the length of the shape and returned e.g. `len(shape)`. If neither method returns a value other than `NotImplemented` and a default value was specified, the default value is returned.

**Raises**
- `TypeError` – `val` doesn’t have either a `_num_qubits_` or a `_qid_shape_` method (or they returned `NotImplemented`) and also no default value was specified.

### `cirq.pauli_expansion`

```python
cirq.pauli_expansion(val: Any, *, default: Union[cirq.value.linear_dict.LinearDict[str], TDefault] = cirq.LinearDict({}), atol: float = 1e-09) → Union[cirq.value.linear_dict.LinearDict[str], TDefault]
Returns coefficients of the expansion of `val` in the Pauli basis.
```

**Parameters**
- `val` – The value whose Pauli expansion is to be returned.
- `default` – Determines what happens when `val` does not have methods that allow Pauli expansion to be obtained (see below). If set, the value is returned in that case. Otherwise, `TypeError` is raised.
atol – Ignore coefficients whose absolute value is smaller than this.

**Returns** If `val` has a `.pauli_expansion_` method, then its result is returned. Otherwise, if `val` has a small unitary then that unitary is expanded in the Pauli basis and coefficients are returned. Otherwise, if default is set to None or other value then default is returned. Otherwise, `TypeError` is raised.

**Raises**

- `TypeError if val has none of the methods necessary to obtain its Pauli – expansion and no default value has been provided.`

---

### cirq.phase_by

```python
cirq.phase_by(val: Any, phase_turns: float, qubit_index: int, default: TDefault = ((),))
```

Returns a phased version of the effect.

For example, an X gate phased by 90 degrees would be a Y gate. This works by calling `val`'s `phase_by` method and returning the result.

**Parameters**

- **val** – The value to describe with a unitary matrix.
- **phase_turns** – The amount to phase the gate, in fractions of a whole turn. Multiply by 2 to get radians.
- **qubit_index** – The index of the target qubit the phasing applies to. For operations this is the index of the qubit within the operation’s qubit list. For gates it’s the index of the qubit within the tuple of qubits taken by the gate’s `on` method.
- **default** – The default value to return if `val` can’t be phased. If not specified, an error is raised when `val` can’t be phased.

**Returns** If `val` has a `.phase_by_` method and its result is not `NotImplemented`, that result is returned. Otherwise, the function will return the default value provided or raise a `TypeError` if none was provided.

**Raises** `TypeError` – `val` doesn’t have a `.phase_by_` method (or that method returned `NotImplemented`) and no `default` was specified.

---

### cirq.pow

```python
cirq.pow(val: Any, exponent: Any, default: Any = ((),)) → Any
```

Returns `val**factor` of the given value, if defined.

Values define an extrapolation by defining a `pow(self, exponent)` method. Note that the method may return `NotImplemented` to indicate a particular extrapolation can’t be done.

**Parameters**
• **val** – The value or iterable of values to invert.

• **exponent** – The extrapolation factor. For example, if this is 0.5 and val is a gate then the caller is asking for a square root of the gate.

• **default** – Determines the fallback behavior when `val` doesn’t have an extrapolation defined. If `default` is not set and that occurs, a `TypeError` is raised instead.

**Returns** If `val` has a `__pow__` method that returns something besides `NotImplemented`, that result is returned. Otherwise, if a default value was specified, the default value is returned.

**Raises** `TypeError` – `val` doesn’t have a `__pow__` method (or that method returned `NotImplemented`) and no default value was specified.

### `cirq.qasm`

```python
```

Returns QASM code for the given value, if possible.

Different values require different sets of arguments. The general rule of thumb is that circuits don’t need any, operations need a `QasmArgs`, and gates need both a `QasmArgs` and `qubits`.

**Parameters**

• **val** – The value to turn into QASM code.

• **args** – A `QasmArgs` object to pass into the value’s `__qasm__` method. This is for needed for objects that only have a local idea of what’s going on, e.g. a `cirq.Operation` in a bigger `cirq.Circuit` involving qubits that the operation wouldn’t otherwise know about.

• **qubits** – A list of qubits that the value is being applied to. This is needed for `cirq.Gate` values, which otherwise wouldn’t know what qubits to talk about.

• **default** – A default result to use if the value doesn’t have a `__qasm__` method or that method returns `NotImplemented` or `None`. If not specified, undecomposable values cause a `TypeError`.

**Returns** The result of `val.__qasm__(...)`, if `val` has a `__qasm__` method and it didn’t return `NotImplemented` or `None`. Otherwise default is returned, if it was specified. Otherwise an error is raised.

**TypeError:** `val` didn’t have a `__qasm__` method (or that method returned `NotImplemented` or `None`) and `default` wasn’t set.

### `cirq.qid_shape`

```python
def cirq.qid_shape(val: Any, default: TDefault = (0,)) → Union[Tuple[int, ...], TDefault]
```

Returns a tuple describing the number of quantum levels of each qubit/qudit/qid `val` operates on.

**Parameters**

• **val** – The value to get the shape of.
- **default** – Determines the fallback behavior when `val` doesn’t have a shape. If `default` is not set, a TypeError is raised. If default is set to a value, that value is returned.

**Returns** If `val` has a `_qid_shape_` method and its result is not Not Implemented, that result is returned. Otherwise, if `val` has a `_num_qubits_` method, the shape with `num_qubits` qubits is returned e.g. `(2,)^num_qubits`. If neither method returns a value other than Not Implemented and a default value was specified, the default value is returned.

**Raises** TypeError – `val` doesn’t have either a `_qid_shape_` or a `_num_qubits_` method (or they returned Not Implemented) and also no default value was specified.

cirq.resolve_parameters

cirq.resolve_parameters(val: Any, param_resolver: cirq.ParamResolverOrSimilarType) → Any

Resolves symbol parameters in the effect using the param resolver.

**Parameters**

- **val** – The object to resolve (e.g. the gate, operation, etc)
- **param_resolver** – the object to use for resolving all symbols

**Returns** a gate or operation of the same type, but with all Symbols replaced with floats according to the given ParamResolver. If `val` has no `_resolve_parameters_` method or if it returns Not Implemented, `val` itself is returned.

cirq.trace_distance_bound

cirq.trace_distance_bound(val: Any) → float

Returns a maximum on the trace distance between this effect’s input and output.

This method attempts a number of strategies to calculate this value.

Strategy 1:
Use the effect’s `_trace_distance_bound_` method.

Strategy 2:
If the effect is unitary, calculate the trace distance bound from the eigenvalues of the unitary matrix.

**Parameters** `val` – The effect of which the bound should be calculated

**Returns** If any of the strategies return a result that is not Not Implemented and not None, that result is returned. Otherwise, 1.0 is returned. Result is capped at a maximum of 1.0, even if the underlying function produces a result greater than 1.0
cirq.unitary

cirq.unitary(val: Any, default: TDefault = array([], dtype=float64)) → Union[numpy.ndarray, TDefault]

Returns a unitary matrix describing the given value.

The matrix is determined by any one of the following techniques:

- The value has a _unitary_ method that returns something besides None or NotImplemented. The matrix is whatever the method returned.
- The value has a _decompose_ method that returns a list of operations, and each operation in the list has a unitary effect. The matrix is created by aggregating the sub-operations’ unitary effects.
- The value has an _apply_unitary_ method, and it returns something besides None or NotImplemented. The matrix is created by applying _apply_unitary_ to an identity matrix.

If none of these techniques succeeds, it is assumed that val doesn’t have a unitary effect. The order in which techniques are attempted is unspecified.

**Parameters**

- **val** – The value to describe with a unitary matrix.
- **default** – Determines the fallback behavior when val doesn’t have a unitary effect. If default is not set, a TypeError is raised. If default is set to a value, that value is returned.

**Returns**

If val has a unitary effect, the corresponding unitary matrix. Otherwise, if default is specified, it is returned.

**Raises**

TypeError – val doesn’t have a unitary effect and no default value was specified.

cirq.validate_mixture

cirq.validate_mixture(supports_mixture: cirq.protocols.mixture_protocol.SupportsMixture)

Validates that the mixture’s tuple are valid probabilities.

### 3.1.14 Magic Method Protocol Types

Classes defining and used by the magic method protocols.

<table>
<thead>
<tr>
<th>Protocol Type</th>
<th>Arguments/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplyChannelArgs</td>
<td>(target_tensor, out_buffer, ...) Arguments for efficiently performing a channel.</td>
</tr>
<tr>
<td>ApplyUnitaryArgs</td>
<td>(target_tensor, ...) Arguments for performing an efficient left-multiplication by a unitary.</td>
</tr>
<tr>
<td>CircuitDiagramInfo</td>
<td>(wire_symbols[, exponent, ...]) Describes how to draw an operation in a circuit diagram.</td>
</tr>
<tr>
<td>CircuitDiagramInfoArgs</td>
<td>(known_qubits, ...) A request for information on drawing an operation in a circuit diagram.</td>
</tr>
<tr>
<td>QasmArgs</td>
<td>([precision, version, qubit_id_map, ...])</td>
</tr>
<tr>
<td>QasmOutput</td>
<td>(operations, qubits[, header, ...])</td>
</tr>
<tr>
<td>SupportsApplyChannel</td>
<td>(*args, **kwargs) An object that can efficiently implement a channel.</td>
</tr>
</tbody>
</table>
### Table 143 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SupportsConsistentApplyUnitary(*args, **kwargs)</code></td>
<td>An object that can be efficiently left-multiplied into tensors.</td>
</tr>
<tr>
<td><code>SupportsApproximateEquality(*args, **kwargs)</code></td>
<td>Object which can be compared approximately.</td>
</tr>
<tr>
<td><code>SupportsChannel(*args, **kwargs)</code></td>
<td>An object that may be describable as a quantum channel.</td>
</tr>
<tr>
<td><code>SupportsCircuitDiagramInfo(*args, **kwargs)</code></td>
<td>A diagrammable operation on qubits.</td>
</tr>
<tr>
<td><code>SupportsDecompose(*args, **kwargs)</code></td>
<td>An object that can be decomposed into simpler operations.</td>
</tr>
<tr>
<td><code>SupportsDecomposeWithQubits(*args, **kwargs)</code></td>
<td>An object that can be decomposed into operations on given qubits.</td>
</tr>
<tr>
<td><code>SupportsExplicitHasUnitary(*args, **kwargs)</code></td>
<td>An object that explicitly specifies whether it has a unitary effect.</td>
</tr>
<tr>
<td><code>SupportsExplicitNumQubits(*args, **kwargs)</code></td>
<td>A unitary, channel, mixture or other object that operates on a known number of qubits.</td>
</tr>
<tr>
<td><code>SupportsExplicitQidShape(*args, **kwargs)</code></td>
<td>A unitary, channel, mixture or other object that operates on a known qid shape.</td>
</tr>
<tr>
<td><code>SupportsMixture(*args, **kwargs)</code></td>
<td>An object that may be describable as a probabilistic combination.</td>
</tr>
<tr>
<td><code>SupportsParameterization(*args, **kwargs)</code></td>
<td>An object that can be parameterized by Symbols and resolved.</td>
</tr>
<tr>
<td><code>SupportsPhase(*args, **kwargs)</code></td>
<td>An effect that can be phased around the Z axis of target qubits.</td>
</tr>
<tr>
<td><code>SupportsQasm(*args, **kwargs)</code></td>
<td>An object that can be turned into QASM code.</td>
</tr>
<tr>
<td><code>SupportsQasmWithArgs(*args, **kwargs)</code></td>
<td>An object that can be turned into QASM code with arguments.</td>
</tr>
<tr>
<td><code>SupportsQasmWithArgsAndQubits(*args, **kwargs)</code></td>
<td>An object that can be turned into QASM code if it knows its qubits.</td>
</tr>
<tr>
<td><code>SupportsTraceDistanceBound(*args, **kwargs)</code></td>
<td>An effect with known bounds on how easy it is to detect.</td>
</tr>
<tr>
<td><code>SupportsUnitary(*args, **kwargs)</code></td>
<td>An object that may be describable by a unitary matrix.</td>
</tr>
</tbody>
</table>

**cirq.ApplyChannelArgs**

**class cirq.ApplyChannelArgs**

```python
class cirq.ApplyChannelArgs:
    target_tensor: numpy.ndarray, out_buffer: numpy.ndarray, auxiliary_buffer0: numpy.ndarray, auxiliary_buffer1: numpy.ndarray, left_axes: Iterable[int], right_axes: Iterable[int]
```

Arguments for efficiently performing a channel.

A channel performs the mapping

$$ X \rightarrow \sum_i A_i X A_i^\dagger $$

for operators $A_i$ that satisfy the normalization condition...
The receiving object is expected to mutate `target_tensor` so that it contains the density matrix after multiplication, and then return `target_tensor`. Alternatively, if workspace is required, the receiving object can overwrite `out_buffer` with the results and return `out_buffer`. Or, if the receiving object is attempting to be simple instead of fast, it can create an entirely new array and return that.

**target_tensor**

The input tensor that needs to be left and right multiplied and summed, representing the effect of the channel. The tensor will have the shape $(2, 2, 2, \ldots, 2)$. It usually corresponds to a multi-qubit density matrix, with the first $n$ indices corresponding to the rows of the density matrix and the last $n$ indices corresponding to the columns of the density matrix.

**out_buffer**

Pre-allocated workspace with the same shape and dtype as the target tensor. If buffers are used, the result should end up in this buffer. It is the responsibility of calling code to notice if the result is this buffer.

**auxiliary_buffer0**

Pre-allocated workspace with the same shape and dtype as the target tensor.

**auxiliary_buffer1**

Pre-allocated workspace with the same shape and dtype as the target tensor.

**left_axes**

Which axes to multiply the left action of the channel upon.

**right_axes**

Which axes to multiply the right action of the channel upon.
__init__(target_tensor: numpy.ndarray, out_buffer: numpy.ndarray, auxiliary_buffer0: numpy.ndarray, auxiliary_buffer1: numpy.ndarray, left_axes: Iterable[int], right_axes: Iterable[int])

Args for apply channel.

Parameters

• **target_tensor** – The input tensor that needs to be left and right multiplied and summed representing the effect of the channel. The tensor will have the shape (2, 2, 2, ..., 2). It usually corresponds to a multi-qubit density matrix, with the first \( n \) indices corresponding to the rows of the density matrix and the last \( n \) indices corresponding to the columns of the density matrix.

• **out_buffer** – Pre-allocated workspace with the same shape and dtype as the target tensor. If buffers are used, the result should end up in this buffer. It is the responsibility of calling code to notice if the result is this buffer.

• **auxiliary_buffer0** – Pre-allocated workspace with the same shape and dtype as the target tensor.

• **auxiliary_buffer1** – Pre-allocated workspace with the same shape and dtype as the target tensor.

• **left_axes** – Which axes to multiply the left action of the channel upon.

• **right_axes** – Which axes to multiply the right action of the channel upon.

Methods

cirq.ApplyUnitaryArgs

class cirq.ApplyUnitaryArgs(target_tensor: numpy.ndarray, available_buffer: numpy.ndarray, axes: Iterable[int])

Arguments for performing an efficient left-multiplication by a unitary.

The receiving object is expected to mutate **target_tensor** so that it contains the state after multiplication, and then return **target_tensor**. Alternatively, if workspace is required, the receiving object can overwrite **available_buffer** with the results and return **available_buffer**. Or, if the receiving object is attempting to be simple instead of fast, it can create an entirely new array and return that.

**target_tensor**

The input tensor that needs to be left-multiplied by the unitary effect of the receiving object. The tensor will have the shape (2, 2, 2, ..., 2). It usually corresponds to a multi-qubit superposition, but it could also be a multi-qubit unitary transformation or some other concept.
available_buffer

Pre-allocated workspace with the same shape and dtype as the target tensor.

axes

Which axes the unitary effect is being applied to (e.g. the qubits that the gate is operating on).

__init__ (target_tensor: numpy.ndarray, available_buffer: numpy.ndarray, axes: Iterable[int])

Parameters

• target_tensor – The input tensor that needs to be left-multiplied by the unitary effect of the receiving object. The tensor will have the shape (2, 2, 2, ..., 2). It usually corresponds to a multi-qubit superposition, but it could also be a multi-qubit unitary transformation or some other concept.

• available_buffer – Pre-allocated workspace with the same shape and dtype as the target tensor.

• axes – Which axes the unitary effect is being applied to (e.g. the qubits that the gate is operating on).

Methods

default([num_qubits, qid_shape]) A default instance starting in state |0.

subspace_index([little_endian_bits_int, ...]) An index for the subspace where the target axes equal a value.

with_axes_transposed_to_start() Returns a transposed view of the same arguments.

cirq.ApplyUnitaryArgs.default

static ApplyUnitaryArgs.default (num_qubits: Optional[int] = None, *
qid_shape: Optional[Tuple[int, ...]] = None) →
cirq.protocols.apply_unitary_protocol.ApplyUnitaryArgs

A default instance starting in state |0.

Specify exactly one argument.

Parameters

• num_qubits – The number of qubits to make space for in the state.

• qid_shape – The shape of the state, specifying the dimension of each qid.
**cirq.ApplyUnitaryArgs.subspace_index**

`ApplyUnitaryArgs.subspace_index` (little_endian_bits_int: int = 0, big_endian_bits_int: int = 0) → Tuple[Union[slice, int, ellipsis], ...]

An index for the subspace where the target axes equal a value.

**Parameters**

- **little_endian_bits_int** – The desired value of the qubits at the targeted axes, packed into an integer. The least significant bit of the integer is the desired bit for the first axis, and so forth in increasing order. Can’t be specified at the same time as `big_endian_bits_int`.

- **big_endian_bits_int** – The desired value of the qubits at the targeted axes, packed into an integer. The most significant bit of the integer is the desired bit for the first axis, and so forth in decreasing order. Can’t be specified at the same time as `little_endian_bits_int`.

- **value_tuple** – The desired value of the qids at the targeted axes, packed into a tuple. Specify either `little_endian_bits_int` or `value_tuple`.

**Returns** A value that can be used to index into `target_tensor` and `available_buffer`, and manipulate only the part of Hilbert space corresponding to a given bit assignment.

**Example**

If `target_tensor` is a 4 qubit tensor and `axes` is [1, 3] and then this method will return the following when given `little_endian_bits=0b01`:

```python
(slice(None), 0, slice(None), 1, Ellipsis)
```

Therefore the following two lines would be equivalent:

```python
args.target_tensor[args.subspace_index(0b01)] += 1
args.target_tensor[:, 0, :, 1] += 1
```

**cirq.ApplyUnitaryArgs.with_axes_transposed_to_start**

`ApplyUnitaryArgs.with_axes_transposed_to_start()` → `cirq.protocols.apply_unitary_protocol.ApplyUnitaryArgs`

Returns a transposed view of the same arguments.

**Returns** A view over the same target tensor and available workspace, but with the numpy arrays transposed such that the axes field is guaranteed to equal `range(len(result.axes))`. This allows one to say e.g. `result.target_tensor[0, 1, 0, ...]` instead of `result.target_tensor[result.subspace_index(0b010)]`.

**cirq.CircuitDiagramInfo**

**class cirq.CircuitDiagramInfo**

```python
circuit_diagram_info = cirq.CircuitDiagramInfo(wire_symbols: Tuple[str, ...], exponent: Any = 1, connected: bool = True, exponent_qubit_index: Optional[int] = None, auto_exponent_parens: bool = True)
```

Describes how to draw an operation in a circuit diagram.

**__init__**

```python
__init__(wire_symbols: Tuple[str, ...], exponent: Any = 1, connected: bool = True, exponent_qubit_index: Optional[int] = None, auto_exponent_parens: bool = True) → None
```

**Parameters**
• **wire_symbols** – The symbols that should be shown on the qubits affected by this operation. Must match the number of qubits that the operation is applied to.

• **exponent** – An optional convenience value that will be appended onto an operation’s final gate symbol with a caret in front (unless it’s equal to 1). For example, the square root of X gate has a text diagram exponent of 0.5 and symbol of ‘X’ so it is drawn as ‘X^0.5’.

• **connected** – Whether or not to draw a line connecting the qubits.

• **exponent_qubit_index** – The qubit to put the exponent on. (The k’th qubit is the k’th target of the gate.) Defaults to the bottom qubit in the diagram.

• **auto_exponent_parens** – When this is True, diagram making code will add parentheses around exponents whose contents could look ambiguous (e.g. if the exponent contains a dash character that could be mistaken for an identity wire). Defaults to True.

**Methods**

cirq.CircuitDiagramInfoArgs

class cirq.CircuitDiagramInfoArgs(known_qubits: Optional[Iterable[cirq.Qid]], known_qubit_count: Optional[int], use_unicode_characters: bool, precision: Optional[int], qubit_map: Optional[Dict[cirq.Qid, int]])

A request for information on drawing an operation in a circuit diagram.

**known_qubits**

The qubits the gate is being applied to. None means this information is not known by the caller.

**known_qubit_count**

The number of qubits the gate is being applied to. None means this information is not known by the caller.

**use_unicode_characters**

If true, the wire symbols are permitted to include unicode characters (as long as they work well in fixed width fonts). If false, use only ascii characters. ASCII is preferred in cases where UTF8 support is done poorly, or where the fixed-width font being used to show the diagrams does not properly handle unicode characters.

**precision**
The number of digits after the decimal to show for numbers in the text diagram. None means use full precision.

**qubit_map**
The map from qubits to diagram positions.

```
__init__(known_qubits: Optional[Iterable[cirq.Qid]], known_qubit_count: Optional[int], use_unicode_characters: bool, precision: Optional[int], qubit_map: Optional[Dict[cirq.Qid, int]]) → None
```
Initialize self. See help(type(self)) for accurate signature.

**Methods**

```
cirq.CircuitDiagramInfoArgs.copy
```
CircuitDiagramInfoArgs.copy()

```
cirq.CircuitDiagramInfoArgs.with_args(**kwargs)
```
CircuitDiagramInfoArgs.with_args(**kwargs)

**Attributes**

```
cirq.CircuitDiagramInfoArgs.UNINFORMED_DEFAULT
```
CircuitDiagramInfoArgs.UNINFORMED_DEFAULT = cirq.CircuitDiagramInfoArgs(known_qubits=None, known_qubit_count=None, use_unicode_characters=True, precision=3, qubit_map=None)

**cirq.QasmArgs**

```
class cirq.QasmArgs(precision: int = 10, version: str = '2.0', qubit_id_map: Dict[cirq.Qid, str] = None, meas_key_id_map: Dict[str, str] = None)
```
```
__init__(precision: int = 10, version: str = '2.0', qubit_id_map: Dict[cirq.Qid, str] = None, meas_key_id_map: Dict[str, str] = None) → None
```
```
Parameters

- **precision** – The number of digits after the decimal to show for numbers in the qasm code.
- **version** – The QASM version to target. Objects may return different qasm depending on version.
- **qubit_id_map** – A dictionary mapping qubits to qreg QASM identifiers.
• **meas_key_id_map** – A dictionary mapping measurement keys to qreg QASM identifiers.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>check_unused_args(used_args, args, kwargs)</code></td>
<td>Checks if the provided arguments have unused parameters.</td>
</tr>
<tr>
<td><code>convert_field(value, conversion)</code></td>
<td>Converts a given value to a specified conversion.</td>
</tr>
<tr>
<td><code>format(**kwargs)</code></td>
<td>Formats the provided arguments.</td>
</tr>
<tr>
<td><code>format_field(value, spec)</code></td>
<td>Method of <code>format()</code> that specifies the output of <code>format()</code>.</td>
</tr>
<tr>
<td><code>get_field(field_name, args, kwargs)</code></td>
<td>Gets a field value based on the provided schema.</td>
</tr>
<tr>
<td><code>get_value(key, args, kwargs)</code></td>
<td>Gets a value based on the provided key.</td>
</tr>
<tr>
<td><code>parse(format_string)</code></td>
<td>Parses the provided format string.</td>
</tr>
<tr>
<td><code>validate_version(*supported_versions)</code></td>
<td>Validates the version of the provided arguments.</td>
</tr>
<tr>
<td><code>vformat(format_string, args, kwargs)</code></td>
<td>Variadic version of <code>format()</code>.</td>
</tr>
</tbody>
</table>

**cirq.QasmArgs.check_unused_args**

`QasmArgs.check_unused_args(used_args, args, kwargs)`

**cirq.QasmArgs.convert_field**

`QasmArgs.convert_field(value, conversion)`

**cirq.QasmArgs.format**

`QasmArgs.format(**kwargs)`

**cirq.QasmArgs.format_field**

`QasmArgs.format_field(value: Any, spec: str) → str`

  Method of `format()` that specifies the output of `format()`.

**cirq.QasmArgs.get_field**

`QasmArgs.get_field(field_name, args, kwargs)`

**cirq.QasmArgs.get_value**

`QasmArgs.get_value(key, args, kwargs)`

**cirq.QasmArgs.parse**

`QasmArgs.parse(format_string)`
cirq.QasmArgs.validate_version

QasmArgs.validate_version(*supported_versions: str) → None

cirq.QasmArgs.vformat

QasmArgs.vformat(format_string, args, kwargs)

cirq.QasmOutput

class cirq.QasmOutput(operations: cirq.OP_TREE, qubits: Tuple[cirq.ops.raw_types.Qid, ...], header: str = ", precision: int = 10, version: str = '2.0')

__init__(operations: cirq.OP_TREE, qubits: Tuple[cirq.ops.raw_types.Qid, ...], header: str = ", precision: int = 10, version: str = '2.0') → None
Initialize self. See help(type(self)) for accurate signature.

Methods

- is_valid_qasm_id(id_str)
  Test if id_str is a valid id in QASM grammar.
- save(path)
  Write QASM output to a file specified by path.

cirq.QasmOutput.is_valid_qasm_id

QasmOutput.is_valid_qasm_id(id_str: str) → bool
  Test if id_str is a valid id in QASM grammar.

cirq.QasmOutput.save

QasmOutput.save(path: Union[str, bytes, int]) → None
  Write QASM output to a file specified by path.

Attributes

- valid_id_re

Cirq Documentation, Release 0.6.1

cirq.QasmOutput.valid_id_re

QasmOutput.valid_id_re = re.compile('[a-z][a-zA-Z0-9_]*\Z')

cirq.SupportsApplyChannel

class cirq.SupportsApplyChannel(*args, **kwargs)
An object that can efficiently implement a channel.

__init__(*args, **kwargs)
Methods

__init__(*args, **kwargs)

### cirq.SupportsApproximateEquality

class cirq.SupportsApproximateEquality(*args, **kwargs)
Object which can be compared approximately.

__init__(*args, **kwargs)

Methods

### cirq.SupportsChannel

class cirq.SupportsChannel(*args, **kwargs)
An object that may be describable as a quantum channel.

__init__(*args, **kwargs)

Methods

### cirq.SupportsCircuitDiagramInfo

class cirq.SupportsCircuitDiagramInfo(*args, **kwargs)
A diagrammable operation on qubits.

__init__(*args, **kwargs)

Methods

### cirq.SupportsDecompose

class cirq.SupportsDecompose(*args, **kwargs)
An object that can be decomposed into simpler operations.
All decomposition methods should ultimately terminate on basic 1-qubit and 2-qubit gates included by default in Cirq. Cirq does not make any guarantees about what the final gate set is. Currently, decompositions within Cirq happen to converge towards the X, Y, Z, CZ, PhasedX, specified-matrix gates, and others. This set will vary from release to release. Because of this variability, it is important for consumers of decomposition to look for generic properties of gates, such as “two qubit gate with a unitary matrix”, instead of specific gate types such as CZ gates (though a consumer is of course free to handle CZ gates in a special way, and consumers can give an intercepting_decomposer to cirq.decompose that attempts to target a specific gate set).

For example, cirq.TOFFOLI has a _decompose_ method that returns a pair of Hadamard gates surrounding a cirq.CCZ. Although cirq.CCZ is not a 1-qubit or 2-qubit operation, it specifies its own _decompose_ method that only returns 1-qubit or 2-qubit operations. This means that iteratively decomposing cirq.TOFFOLI terminates in 1-qubit and 2-qubit operations, and so almost all decomposition-aware code will be able to handle cirq.TOFFOLI instances.

Callers are responsible for iteratively decomposing until they are given operations that they understand. The cirq.decompose method is a simple way to do this, because it has logic to recursively decompose until a given keep predicate is satisfied.

Code implementing _decompose_ MUST NOT create cycles, such as a gate A decomposes into a gate B which decomposes back into gate A. This will result in infinite loops when calling cirq.decompose.

It is permitted (though not recommended) for the chain of decompositions resulting from an operation to hit a dead end before reaching 1-qubit or 2-qubit operations. When this happens, cirq.decompose will raise a TypeError by default, but can be configured to ignore the issue or raise a caller-provided error.

__init__(*args, **kwargs)

Methods
cirq.SupportsDecomposeWithQubits

```python
class cirq.SupportsDecomposeWithQubits(*args, **kwargs):
    An object that can be decomposed into operations on given qubits.
```

Returning `NotImplemented` or `None` means “not decomposable”. Otherwise an operation, list of operations, or generally anything meeting the `OP_TREE` contract can be returned.

For example, a SWAP gate can be turned into three CNOTs. But in order to describe those CNOTs one must be able to talk about “the target qubit” and “the control qubit”. This can only be done once the qubits-to-be-swapped are known.

The main user of this protocol is `GateOperation`, which decomposes itself by delegating to its gate. The qubits argument is needed because gates are specified independently of target qubits and so must be told the relevant qubits. A `GateOperation` implements `SupportsDecompose` as long as its gate implements `SupportsDecomposeWithQubits`.

```python
__init__(*args, **kwargs)
```

Methods

---

cirq.SupportsExplicitHasUnitary

```python
class cirq.SupportsExplicitHasUnitary(*args, **kwargs):
    An object that explicitly specifies whether it has a unitary effect.
```

```python
__init__(*args, **kwargs)
```

Methods

---

cirq.SupportsExplicitNumQubits

```python
class cirq.SupportsExplicitNumQubits(*args, **kwargs)
    A unitary, channel, mixture or other object that operates on a known number of qubits.
```

```python
__init__(*args, **kwargs)
```
Methods

## cirq.SupportsExplicitQidShape

class cirq.SupportsExplicitQidShape(*args, **kwargs)

A unitary, channel, mixture or other object that operates on a known number qubits/qudits/qids, each with a specific number of quantum levels.

__init__(*args, **kwargs)

Methods

## cirq.SupportsMixture

class cirq.SupportsMixture(*args, **kwargs)

An object that may be describable as a probabilistic combination.

__init__(*args, **kwargs)

Methods

## cirq.SupportsParameterization

class cirq.SupportsParameterization(*args, **kwargs)

An object that can be parameterized by Symbols and resolved via a ParamResolver

__init__(*args, **kwargs)

Methods

## cirq.SupportsPhase

class cirq.SupportsPhase(*args, **kwargs)

An effect that can be phased around the Z axis of target qubits.

__init__(*args, **kwargs)

Methods

3.1. API Reference
cirq.SupportsQasm

```python
class cirq.SupportsQasm(*args, **kwargs):
    An object that can be turned into QASM code.
```

Returning `NotImplemented` or `None` means “don’t know how to turn into QASM”. In that case fallbacks based on decomposition and known unitaries will be used instead.

```python
__init__(*args, **kwargs)
```

Methods

---

cirq.SupportsQasmWithArgs

```python
class cirq.SupportsQasmWithArgs(*args, **kwargs):
    An object that can be turned into QASM code.
```

Returning `NotImplemented` or `None` means “don’t know how to turn into QASM”. In that case fallbacks based on decomposition and known unitaries will be used instead.

```python
__init__(*args, **kwargs)
```

Methods

---

cirq.SupportsQasmWithArgsAndQubits

```python
class cirq.SupportsQasmWithArgsAndQubits(*args, **kwargs):
    An object that can be turned into QASM code if it knows its qubits.
```

Returning `NotImplemented` or `None` means “don’t know how to turn into QASM”. In that case fallbacks based on decomposition and known unitaries will be used instead.

```python
__init__(*args, **kwargs)
```
Methods

cirq.SupportsTraceDistanceBound

class cirq.SupportsTraceDistanceBound(*args, **kwargs)

An effect with known bounds on how easy it is to detect.

__init__(*args, **kwargs)

Methods

cirq.SupportsUnitary

class cirq.SupportsUnitary(*args, **kwargs)

An object that may be describable by a unitary matrix.

__init__(*args, **kwargs)

Methods

3.1.15 Optimization

Classes and methods for rewriting circuits.

ConvertToCzAndSingleGates([ignore_failures, ...])

Attempts to convert strange multi-qubit gates into CZ and single qubit

DropEmptyMoments

Removes empty moments from a circuit.

DropNegligible([tolerance])

An optimization pass that removes operations with tiny effects.

EjectPhasedPaulis([tolerance, ...])

Pushes X, Y, and PhasedX gates towards the end of the circuit.

EjectZ([tolerance, eject_parameterized])

Pushes Z gates towards the end of the circuit.

ExpandComposite([no_decomp])

An optimizer that expands composite operations via cirq.decompose.

google.optimized_for_xmon(circuit, ...)

Optimizes a circuit with XmonDevice in mind.

merge_single_qubit_gates_into_phased_x_z(circuit)

Canonicalizes runs of single-qubit rotations in a circuit.

Continued on next page
Table 170 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MergeInteractions([tolerance, ...])</code></td>
<td>Combines series of adjacent one and two-qubit gates operating on a</td>
</tr>
<tr>
<td><code>MergeSingleQubitGates(*[rewriter, synthesizer])</code></td>
<td>Optimizes runs of adjacent unitary 1-qubit operations.</td>
</tr>
<tr>
<td><code>PointOptimizationSummary(clear_span,...)</code></td>
<td>A description of a local optimization to perform.</td>
</tr>
<tr>
<td><code>PointOptimizer([post_clean_up])</code></td>
<td>Makes circuit improvements focused on a specific location.</td>
</tr>
<tr>
<td><code>single_qubit_matrix_to_gates(mat[, tolerance])</code></td>
<td>Implements a single-qubit operation with few gates.</td>
</tr>
<tr>
<td><code>single_qubit_matrix_to_pauli_rotations(mat)</code></td>
<td>Implements a single-qubit operation with few rotations.</td>
</tr>
<tr>
<td><code>single_qubit_matrix_to_phased_x_z(mat[, atol])</code></td>
<td>Implements a single-qubit operation with a PhasedX and Z gate.</td>
</tr>
<tr>
<td><code>single_qubit_op_to_framed_phase_form(mat)</code></td>
<td>Decomposes a 2x2 unitary M into $U^\dagger I \times \text{diag}(1, r) \times U$ * $\text{diag}(g, g)$.</td>
</tr>
<tr>
<td><code>two_qubit_matrix_to_operations(q0, q1, ...)</code></td>
<td>Decomposes a two-qubit operation into Z/XY/CZ gates.</td>
</tr>
</tbody>
</table>

### cirq.ConvertToCzAndSingleGates

```python
class cirq.ConvertToCzAndSingleGates (ignore_failures: bool = False, allow_partial_czs: bool = False)
```

Attempts to convert strange multi-qubit gates into CZ and single qubit gates.

First, checks if the operation has a unitary effect. If so, and the gate is a 1-qubit or 2-qubit gate, then performs circuit synthesis of the operation.

Second, attempts to `cirq.decompose` to the operation.

Third, if `ignore_failures` is set, gives up and returns the gate unchanged. Otherwise raises a TypeError.

```python
__init__ (ignore_failures: bool = False, allow_partial_czs: bool = False) → None
```

**Parameters**

- **ignore_failures** – If set, gates that fail to convert are forwarded unchanged. If not set, conversion failures raise a TypeError.
- **allow_partial_czs** – If set, the decomposition is permitted to use gates of the form `cirq.CZ**t`, instead of only `cirq.CZ`. 
Methods

**optimization_at** *(circuit, index, op)*  
Describes how to change operations near the given location.

**optimize_circuit** *(circuit)*

---

cirq.ConvertToCzAndSingleGates.optimization_at


Describes how to change operations near the given location.

For example, this method could realize that the given operation is an X gate and that in the very next moment there is a Z gate. It would indicate that they should be combined into a Y gate by returning:

PointOptimizationSummary(clear_span=2, clear_qubits=op.qubits, new_operations=cirq.Y(op.qubits[0]))

**Parameters**

- **circuit** – The circuit to improve.
- **index** – The index of the moment with the operation to focus on.
- **op** – The operation to focus improvements upon.

**Returns**  
A description of the optimization to perform, or else None if no change should be made.

---

cirq.ConvertToCzAndSingleGates.optimize_circuit

ConvertToCzAndSingleGates.optimize_circuit *(circuit: cirq.circuits.circuit.Circuit)*

---

cirq.DropEmptyMoments

circle DropEmptyMoments

Removes empty moments from a circuit.

**__init__**

Initialize self. See help(type(self)) for accurate signature.

---

**Methods**

**optimize_circuit** *(circuit)*
cirq.DropEmptyMoments.optimize_circuit

DropEmptyMoments.optimize_circuit(circuit: cirq.circuits.circuit.Circuit)

cirq.DropNegligible

class cirq.DropNegligible(tolerance: float = 1e-08)
An optimization pass that removes operations with tiny effects.

__init__(tolerance: float = 1e-08) → None
Initialize self. See help(type(self)) for accurate signature.

Methods

optimize_circuit(circuit)

cirq.DropNegligible.optimize_circuit

DropNegligible.optimize_circuit(circuit: cirq.circuits.circuit.Circuit) → None

cirq.EjectPhasedPaulis

class cirq.EjectPhasedPaulis(tolerance: float = 1e-08, eject_parameterized: bool = False)
Pushes X, Y, and PhasedX gates towards the end of the circuit.

As the gates get pushed, they may absorb Z gates, cancel against other
X, Y, or PhasedX gates with exponent=1, get merged into measurements (as
output bit flips), and cause phase kickback operations across CZs (which can
then be removed by the EjectZ optimization).

__init__(tolerance: float = 1e-08, eject_parameterized: bool = False) → None

Parameters

• tolerance – Maximum absolute error tolerance. The optimization is permitted to sim-
  ply drop negligible combinations gates with a threshold determined by this tolerance.

• eject_parameterized – If True, the optimization will attempt to eject parametrized
  gates as well. This may result in other gates parameterized by symbolic expressions.

Methods

optimize_circuit(circuit)

cirq.EjectPhasedPaulis.optimize_circuit

EjectPhasedPaulis.optimize_circuit(circuit: cirq.circuits.circuit.Circuit)
cirq.EjectZ

**class** cirq.EjectZ (tolerance: float = 0.0, eject_parameterized: bool = False)

Pushes Z gates towards the end of the circuit.

As the Z gates get pushed they may absorb other Z gates, get absorbed into measurements, cross CZ gates, cross W gates (by phasing them), etc.

**__init__** (tolerance: float = 0.0, eject_parameterized: bool = False) → None

**Parameters**

- **tolerance** – Maximum absolute error tolerance. The optimization is permitted to simply drop negligible combinations of Z gates, with a threshold determined by this tolerance.

- **eject_parameterized** – If True, the optimization will attempt to eject parametrized Z gates as well. This may result in other gates parameterized by symbolic expressions.

**Methods**

`cirq.EjectZ.optimize_circuit`

EjectZ.optimize_circuit (circuit: cirq.circuits.circuit.Circuit)

**cirq.ExpandComposite**

**class** cirq.ExpandComposite (no_decomp: Callable[cirq.ops.raw_types.Operation, bool] = <function ExpandComposite.<lambda>>)

An optimizer that expands composite operations via cirq.decompose.

For each operation in the circuit, this pass examines if the operation can be decomposed. If it can be, the operation is cleared out and and replaced with its decomposition using a fixed insertion strategy.

**__init__** (no_decomp: Callable[cirq.ops.raw_types.Operation, bool] = <function ExpandComposite.<lambda>>) → None

Construct the optimization pass.

**Parameters**

- **no_decomp** – A predicate that determines whether an operation should be decomposed or not. Defaults to decomposing everything.

**Methods**
optimization_at(circuit, index, op) Describes how to change operations near the given location.

optimize_circuit(circuit)

cirq.ExpandComposite.optimization_at

ExpandComposite.optimization_at(circuit, index, op)
Describes how to change operations near the given location.

For example, this method could realize that the given operation is an X gate and that in the very next moment there is a Z gate. It would indicate that they should be combined into a Y gate by returning

PointOptimizationSummary(clear_span=2,
clear_qubits=op.qubits,
new_operations=cirq.Y(op.qubits[0]))

Parameters
- **circuit** – The circuit to improve.
- **index** – The index of the moment with the operation to focus on.
- **op** – The operation to focus improvements upon.

Returns A description of the optimization to perform, or else None if no change should be made.

cirq.ExpandComposite.optimize_circuit

ExpandComposite.optimize_circuit(circuit: cirq.circuits.circuit.Circuit)

cirq.google.optimized_for_xmon


Optimizes a circuit with XmonDevice in mind.

Starts by converting the circuit’s operations to the xmon gate set, then begins merging interactions and rotations, ejecting pi-rotations and phasing operations, dropping unnecessary operations, and pushing operations earlier.

Parameters
- **circuit** – The circuit to optimize.
- **new_device** – The device the optimized circuit should be targeted at. If set to None, the circuit’s current device is used.
• **qubit_map** – Transforms the qubits (e.g. so that they are GridQubits).

• **allow_partial_czs** – If true, the optimized circuit may contain partial CZ gates. Otherwise all partial CZ gates will be converted to full CZ gates. At worst, two CZ gates will be put in place of each partial CZ from the input.

**Returns** The optimized circuit.

cirq.merge_single_qubit_gates_into_phased_x_z

cirq.merge_single_qubit_gates_into_phased_x_z(circuit: cirq.circuits.circuit.Circuit, atol: float = 1e-08) → None

Canonicalizes runs of single-qubit rotations in a circuit.

Specifically, any run of non-parameterized circuits will be replaced by an optional PhasedX operation followed by an optional Z operation.

**Parameters**

• **circuit** – The circuit to rewrite. This value is mutated in-place.

• **atol** – Absolute tolerance to angle error. Larger values allow more negligible gates to be dropped, smaller values increase accuracy.

cirq.MergeInteractions


Combines series of adjacent one and two-qubit gates operating on a pair of qubits.

__init__(tolerance: float = 1e-08, allow_partial_czs: bool = True, post_clean_up: Callable[Sequence[cirq.ops.raw_types.Operation], Union[cirq.ops.raw_types.Operation, cirq.ops.op_tree.OpTree]] = <function MergeInteractions.<lambda>>) → None

Args:
post_clean_up: This function is called on each set of optimized operations before they are put into the circuit to replace the old operations.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>optimization_at</strong></td>
<td>(circuit, index, op) Describes how to change operations near the given location.</td>
</tr>
<tr>
<td><strong>optimize_circuit</strong></td>
<td>(circuit)</td>
</tr>
</tbody>
</table>
cirq.MergeInteractions.optimization_at


Describes how to change operations near the given location.

For example, this method could realize that the given operation is an X gate and that in the very next moment there is a Z gate. It would indicate that they should be combined into a Y gate by returning a `PointOptimizationSummary(clear_span=2, clear_qubits=op.qubits, new_operations=cirq.Y(op.qubits[0]))`

Parameters

- **circuit** – The circuit to improve.
- **index** – The index of the moment with the operation to focus on.
- **op** – The operation to focus improvements upon.

Returns

A description of the optimization to perform, or else None if no change should be made.

cirq.MergeInteractions.optimize_circuit

MergeInteractions.optimize_circuit(circuit: cirq.circuits.circuit.Circuit)

cirq.MergeSingleQubitGates

Optimizes runs of adjacent unitary 1-qubit operations.


Parameters

- **rewriter** – Specifies how to merge runs of single-qubit operations into a more desirable form. Takes a list of operations and produces a list of operations. The default rewriter computes the matrix of the run and returns a `cirq.SingleQubitMatrixGate`. If `rewriter` returns `None`, that means “do not rewrite the operations”.

- **synthesizer** – A special kind of rewriter that operates purely on the unitary matrix of the intended operation. Takes a qubit and a unitary matrix and returns a list of operations. Can’t be specified at the same time as `rewriter`. If `synthesizer` returns `None`, that means “do not rewrite the operations used to make this matrix”.

364 Chapter 3. API Reference
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>optimization_at(circuit, index, op)</code></td>
<td>Describes how to change operations near the given location.</td>
</tr>
<tr>
<td><code>optimize_circuit(circuit)</code></td>
<td></td>
</tr>
</tbody>
</table>

### `cirq.MergeSingleQubitGates.optimization_at`


Describes how to change operations near the given location.

For example, this method could realize that the given operation is an X gate and that in the very next moment there is a Z gate. It would indicate that they should be combined into a Y gate by returning `PointOptimizationSummary(clear_span=2, clear_qubits=op.qubits, new_operations=cirq.Y(op.qubits[0]))`

**Parameters**

- **circuit** – The circuit to improve.
- **index** – The index of the moment with the operation to focus on.
- **op** – The operation to focus improvements upon.

**Returns** A description of the optimization to perform, or else None if no change should be made.

### `cirq.MergeSingleQubitGates.optimize_circuit`

`MergeSingleQubitGates.optimize_circuit(circuit: cirq.circuits.circuit.Circuit)`

### `cirq.PointOptimizationSummary`

**class cirq.PointOptimizationSummary**

`cirq.PointOptimizationSummary(clear_span: int, clear_qubits: Iterable[cirq.Qid], new_operations: cirq.OP_TREE)`

A description of a local optimization to perform.

**__init__** (clear_span: int, clear_qubits: Iterable[cirq.Qid], new_operations: cirq.OP_TREE) → None

**Parameters**

- **clear_span** – Defines the range of moments to affect. Specifically, refers to the indices in range(start, start+clear_span) where start is an index known from surrounding context.
- **clear_qubits** – Defines the set of qubits that should be cleared with each affected moment.
- **new_operations** – The operations to replace the cleared out operations with.
Methods

cirq.PointOptimizer

class cirq.PointOptimizer (post_clean_up: Callable[Sequence[cirq.Operation], Union[cirq.ops.raw_types.Operation, cirq.ops.op_tree.OpTree]] = <function PointOptimizer.<lambda>>)  

Makes circuit improvements focused on a specific location.

__init__ (post_clean_up: Callable[Sequence[cirq.Operation], Union[cirq.ops.raw_types.Operation, cirq.ops.op_tree.OpTree]] = <function PointOptimizer.<lambda>>) → None

Parameters post_clean_up – This function is called on each set of optimized operations before they are put into the circuit to replace the old operations.

Methods

optimization_at(circuit, index, op)  

Describes how to change operations near the given location.

optimize_circuit(circuit)

For example, this method could realize that the given operation is an X gate and that in the very next moment there is a Z gate. It would indicate that they should be combined into a Y gate by returning PointOptimizationSummary(clear_span=2, clear_qubits=op.qubits, new_operations=cirq.Y(op.qubits[0]))

Parameters

- circuit – The circuit to improve.
- index – The index of the moment with the operation to focus on.
- op – The operation to focus improvements upon.

Returns A description of the optimization to perform, or else None if no change should be made.

cirq.PointOptimizer.optimize_circuit

PointOptimizer.optimize_circuit (circuit: cirq.circuits.circuit.Circuit)
**cirq.single_qubit_matrix_to_gates**

**cirq.single_qubit_matrix_to_gates** *(mat: numpy.ndarray, tolerance: float = 0) → List[cirq.ops.gate_features.SingleQubitGate]*

Implements a single-qubit operation with few gates.

**Parameters**

- **mat** – The 2x2 unitary matrix of the operation to implement.
- **tolerance** – A limit on the amount of error introduced by the construction.

**Returns**

A list of gates that, when applied in order, perform the desired operation.

**cirq.single_qubit_matrix_to_pauli_rotations**

**cirq.single_qubit_matrix_to_pauli_rotations** *(mat: numpy.ndarray, atol: float = 0) → List[Tuple[cirq.ops.pauli_gates.Pauli, float]]*

Implements a single-qubit operation with few rotations.

**Parameters**

- **mat** – The 2x2 unitary matrix of the operation to implement.
- **atol** – A limit on the amount of absolute error introduced by the construction.

**Returns**

A list of (Pauli, half_turns) tuples that, when applied in order, perform the desired operation.

**cirq.single_qubit_matrix_to_phased_x_z**

**cirq.single_qubit_matrix_to_phased_x_z** *(mat: numpy.ndarray, atol: float = 0) → List[cirq.ops.gate_features.SingleQubitGate]*

Implements a single-qubit operation with a PhasedX and Z gate. If one of the gates isn’t needed, it will be omitted.

**Parameters**

- **mat** – The 2x2 unitary matrix of the operation to implement.
- **atol** – A limit on the amount of error introduced by the construction.

**Returns**

A list of gates that, when applied in order, perform the desired operation.

**cirq.single_qubit_op_to_framed_phase_form**

**cirq.single_qubit_op_to_framed_phase_form** *(mat: numpy.ndarray) → Tuple[numpy.ndarray, complex, complex]*

Decomposes a 2x2 unitary M into $U^{-1} * \text{diag}(1, r) * U * \text{diag}(g, g)$.

- $U$ translates the rotation axis of $M$ to the Z axis.
- $g$ fixes a global phase factor difference caused by the translation.
- $r$’s phase is the amount of rotation around $M$’s rotation axis.
This decomposition can be used to decompose controlled single-qubit rotations into controlled-Z operations bordered by single-qubit operations.

**Parameters** `mat` – The qubit operation as a 2x2 unitary matrix.

**Returns** A 2x2 unitary U, the complex relative phase factor r, and the complex global phase factor g. Applying M is equivalent (up to global phase) to applying U, rotating around the Z axis to apply r, then un-applying U. When M is controlled, the control must be rotated around the Z axis to apply g.

cirq.two_qubit_matrix_to_operations

cirq.two_qubit_matrix_to_operations

Decomposes a two-qubit operation into Z/XY/CZ gates.

**Parameters**

- `q0` – The first qubit being operated on.
- `q1` – The other qubit being operated on.
- `mat` – Defines the operation to apply to the pair of qubits.
- `allow_partial_czs` – Enables the use of Partial-CZ gates.
- `atol` – A limit on the amount of absolute error introduced by the construction.
- `clean_operations` – Enables optimizing resulting operation list by merging operations and ejecting phased Paulis and Z operations.

**Returns** A list of operations implementing the matrix.

### 3.1.16 Utilities

General utility methods, mostly related to performing relevant linear algebra operations and decompositions.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allclose_up_to_global_phase</td>
<td>Determines if a ( \approx b \exp(i \cdot t) ) for some ( t ).</td>
</tr>
<tr>
<td>apply_matrix_to_slices</td>
<td>Left-multiplies an N( \times )N matrix onto N slices of a numpy array.</td>
</tr>
<tr>
<td>bidiagonalize_real_matrix_pair_with_symmetric_products</td>
<td>Finds orthogonal matrices ( L ) and ( R ) such that ( L \cdot \text{matrix} \cdot R ) is diagonal.</td>
</tr>
<tr>
<td>bidiagonalize_unitary_with_special_orthogonals</td>
<td>Finds orthogonal matrices ( L ) and ( R ) such that ( L \cdot \text{matrix} \cdot R ) is diagonal.</td>
</tr>
<tr>
<td>block_diag</td>
<td>Concatenates blocks into a block diagonal matrix.</td>
</tr>
<tr>
<td>commutes</td>
<td>Determines if two matrices approximately commute.</td>
</tr>
<tr>
<td>canonicalize_half_turns</td>
<td>Wraps the input into the range ((-1, +1]).</td>
</tr>
<tr>
<td>chosen_angle_to_canonical_half_turns</td>
<td>Returns a canonicalized half_turns value based on the given arguments.</td>
</tr>
<tr>
<td>chosen_angle_to_half_turns</td>
<td>Returns a half_turns value based on the given arguments.</td>
</tr>
<tr>
<td>commutes</td>
<td>Determines if two matrices approximately commute.</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 181 – continued from previous page

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>diagonalize_real_symmetric_and_sorted_diagonal_matrices</code></td>
<td>Returns an orthogonal matrix that diagonalizes both given matrices.</td>
</tr>
<tr>
<td><code>diagonalize_real_symmetric_matrix</code></td>
<td>Returns an orthogonal matrix that diagonalizes the given matrix.</td>
</tr>
<tr>
<td><code>dot(*values)</code></td>
<td>Computes the dot/matrix product of a sequence of values.</td>
</tr>
<tr>
<td><code>Duration([value, picos, nanos, micros, millis])</code></td>
<td>A time delta that supports symbols and picosecond accuracy.</td>
</tr>
<tr>
<td><code>expand_matrix_in_orthogonal_basis(m, basis)</code></td>
<td>Computes coefficients of expansion of m in basis.</td>
</tr>
<tr>
<td><code>hilbert_schmidt_inner_product(m1, m2)</code></td>
<td>Computes Hilbert-Schmidt inner product of two matrices.</td>
</tr>
<tr>
<td><code>is_diagonal(matrix, *[atol])</code></td>
<td>Determines if a matrix is a approximately diagonal.</td>
</tr>
<tr>
<td><code>is_hermitian(matrix, *[rtol, atol])</code></td>
<td>Determines if a matrix is approximately Hermitian.</td>
</tr>
<tr>
<td><code>is_negligible_turn(turns, tolerance)</code></td>
<td>Determines if a matrix is approximately negligible.</td>
</tr>
<tr>
<td><code>is_orthogonal(matrix, *[rtol, atol])</code></td>
<td>Determines if a matrix is approximately orthogonal.</td>
</tr>
<tr>
<td><code>is_special_orthogonal(matrix, *[rtol, atol])</code></td>
<td>Determines if a matrix is approximately special orthogonal.</td>
</tr>
<tr>
<td><code>is_special_unitary(matrix, *[rtol, atol])</code></td>
<td>Determines if a matrix is approximately unitary with unit determinant.</td>
</tr>
<tr>
<td><code>is_unitary(matrix, *[rtol, atol])</code></td>
<td>Determines if a matrix is approximately unitary.</td>
</tr>
<tr>
<td><code>kak_canonicalize_vector(x, y, z[, atol])</code></td>
<td>Canonicalizes an XX/YY/ZZ interaction by swap/negate/shift-ing axes.</td>
</tr>
<tr>
<td><code>kak_decomposition(unitary_object, *[rtol, ...])</code></td>
<td>Decomposes a 2-qubit unitary into 1-qubit ops and XX/YY/ZZ interactions.</td>
</tr>
<tr>
<td><code>kak_vector(unitary, *[rtol, atol, ...])</code></td>
<td>Compute the KAK vectors of one or more two qubit unitaries.</td>
</tr>
<tr>
<td><code>KakDecomposition(*, global_phase, ...)</code></td>
<td>A convenient description of an arbitrary two-qubit operation.</td>
</tr>
<tr>
<td><code>kron(*factors)</code></td>
<td>Computes the kronecker product of a sequence of values.</td>
</tr>
<tr>
<td><code>kron_factor_4x4_to_2x2s(matrix)</code></td>
<td>Splits a 4x4 matrix $U = \text{kron}(A, B)$ into A, B, and a global factor.</td>
</tr>
<tr>
<td><code>kron_with_controls(*factors)</code></td>
<td>Computes the kronecker product of a sequence of values and control tags.</td>
</tr>
<tr>
<td><code>LinearDict([terms, validator])</code></td>
<td>Represents linear combination of things.</td>
</tr>
<tr>
<td><code>map_eigenvalues(matrix, func, *[rtol, atol])</code></td>
<td>Applies a function to the eigenvalues of a matrix.</td>
</tr>
<tr>
<td><code>match_global_phase(a, b)</code></td>
<td>Phases the given matrices so that they agree on the phase of one entry.</td>
</tr>
<tr>
<td><code>matrix_from_basis_coefficients(expansion, basis)</code></td>
<td>Computes linear combination of basis vectors with given coefficients.</td>
</tr>
<tr>
<td><code>partial_trace(tensor, keep_indices)</code></td>
<td>Takes the partial trace of a given tensor.</td>
</tr>
<tr>
<td><code>PeriodicValue(value, period)</code></td>
<td>Wrapper for periodic numerical values.</td>
</tr>
<tr>
<td><code>reflection_matrix_pow(reflection_matrix, ...)</code></td>
<td>Raises a matrix with two opposing eigenvalues to a power.</td>
</tr>
<tr>
<td><code>slice_for_qubits_equal_to(target_qubit_axes)</code></td>
<td>Returns an index corresponding to a desired subset of an np.ndarray.</td>
</tr>
<tr>
<td><code>so4_to_magic_su2s(mat, *[rtol, atol, ...])</code></td>
<td>Finds 2x2 special-unitaries A, B where $\text{mat} = \text{Mag.H @ kron}(A, B) @ \text{Mag}$.</td>
</tr>
</tbody>
</table>

Continued on next page
**Table 181 – continued from previous page**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>targeted_conjugate_about(tensor, target, indices)</code></td>
<td>Conjugates the given tensor about the target tensor.</td>
</tr>
<tr>
<td><code>targeted_left_multiply(left_matrix, ... [, out])</code></td>
<td>Left-multiplies the given axes of the target tensor by the given matrix.</td>
</tr>
<tr>
<td><code>TextDiagramDrawer()</code></td>
<td>A utility class for creating simple text diagrams.</td>
</tr>
<tr>
<td><code>Timestamp(*[, picos, nanos])</code></td>
<td>A location in time with picosecond accuracy.</td>
</tr>
<tr>
<td><code>value_equality([cls, unhashable, ...])</code></td>
<td>Implements <code>eq/ne/hash</code> via a <code>value_equality_values</code> method.</td>
</tr>
</tbody>
</table>

### `cirq.allclose_up_to_global_phase`

`cirq.allclose_up_to_global_phase(a: numpy.ndarray, b: numpy.ndarray, *, rtol: float = 1e-05, atol: float = 1e-08, equal_nan: bool = False) → bool`

Determines if \(a \sim b \exp(i t)\) for some \(t\).

**Parameters**

- \(a\) – A numpy array.
- \(b\) – Another numpy array.
- \(rtol\) – Relative error tolerance.
- \(atol\) – Absolute error tolerance.
- \(equal_nan\) – Whether or not NaN entries should be considered equal to other NaN entries.

### `cirq.apply_matrix_to_slices`

`cirq.apply_matrix_to_slices(target: numpy.ndarray, matrix: numpy.ndarray, slices: Sequence[Union[int, slice, ellipsis, Sequence[Union[int, slice, ellipsis]]]], *, out: Optional[numpy.ndarray] = None) → numpy.ndarray`

Left-multiplies an \(N\times N\) matrix onto \(N\) slices of a numpy array.

**Example**

The 4x4 matrix of a fractional SWAP gate can be expressed as

\[
\begin{bmatrix}
1 & X^{**t} \\
X^{**t} & 1
\end{bmatrix}
\]

Where \(X\) is the 2x2 Pauli X gate and \(t\) is the power of the swap with \(t=1\) being a full swap. \(X^{**t}\) is a power of the Pauli X gate’s matrix. Applying the fractional swap is equivalent to applying a fractional \(X\) within the inner 2x2 subspace; the rest of the matrix is identity. This can be expressed using `apply_matrix_to_slices` as follows:
def fractional_swap(target):
    assert target.shape == (4,)
    return apply_matrix_to_slices(
        target=target,
        matrix=cirq.unitary(cirq.X**t),
        slices=[1, 2]
    )

Parameters

- **target** – The input array with slices that need to be left-multiplied.
- **matrix** – The linear operation to apply to the subspace defined by the slices.
- **slices** – The parts of the tensor that correspond to the “vector entries” that the matrix should operate on. May be integers or complicated multi-dimensional slices into a tensor. The slices must refer to non-overlapping sections of the input all with the same shape.
- **out** – Where to write the output. If not specified, a new numpy array is created, with the same shape and dtype as the target, to store the output.

Returns The transformed array.

cirq.bidiagonalize_real_matrix_pair_with_symmetric_products
cirq.bidiagonalize_real_matrix_pair_with_symmetric_products(mat1:
    numpy.ndarray,
    mat2:
    numpy.ndarray,
    *, rtol: float = 1e-05,
    atol: float = 1e-08,
    check Preconditions: bool = True) → Tuple[numpy.ndarray,
    numpy.ndarray]

Finds orthogonal matrices that diagonalize both mat1 and mat2.

Requires mat1 and mat2 to be real.
Requires mat1.T @ mat2 to be symmetric.
Requires mat1 @ mat2.T to be symmetric.

Parameters

- **mat1** – One of the real matrices.
- **mat2** – The other real matrix.
- **rtol** – Relative numeric error threshold.
- **atol** – Absolute numeric error threshold.
- **check preconditions** – If set, verifies that the inputs are real, and that mat1.T @ mat2 and mat1 @ mat2.T are both symmetric. Defaults to set.

Returns A tuple (L, R) of two orthogonal matrices, such that both L @ mat1 @ R and L @ mat2 @ R are diagonal matrices.
Cirq Documentation, Release 0.6.1

Raises **ValueError** – Matrices don’t meet preconditions (e.g. not real).

### cirq.bidiagonalize_unitary_with_special_orthogonals

```python
cirq.bidiagonalize_unitary_with_special_orthogonals(mat: numpy.ndarray, *, rtol: float = 1e-05, atol: float = 1e-08, check_preconditions: bool = True) → Tuple[numpy.ndarray, numpy.array, numpy.ndarray]
```

Finds orthogonal matrices L, R such that L @ matrix @ R is diagonal.

**Parameters**

- `mat` – A unitary matrix.
- `rtol` – Relative numeric error threshold.
- `atol` – Absolute numeric error threshold.
- `check_preconditions` – If set, verifies that the input is a unitary matrix (to the given tolerances). Defaults to set.

**Returns**

A triplet (L, d, R) such that L @ mat @ R = diag(d). Both L and R will be orthogonal matrices with determinant equal to 1.

Raises **ValueError** – Matrices don’t meet preconditions (e.g. not real).

### cirq.block_diag

```python
cirq.block_diag(*blocks: numpy.ndarray) → numpy.ndarray
```

Concatenates blocks into a block diagonal matrix.

**Parameters**

- `blocks` – Square matrices to place along the diagonal of the result.

**Returns**

A block diagonal matrix with the given blocks along its diagonal.

**Raises** **ValueError** – A block isn’t square.

### cirq.commutes

```python
cirq.commutes(m1: numpy.ndarray, m2: numpy.ndarray, *, rtol: float = 1e-05, atol: float = 1e-08) → bool
```

Determines if two matrices approximately commute.

Two matrices A and B commute if they are square and have the same size and AB = BA.

**Parameters**

- `m1` – One of the matrices.
- `m2` – The other matrix.
- `rtol` – The per-matrix-entry relative tolerance on equality.
- `atol` – The per-matrix-entry absolute tolerance on equality.
**Returns** Whether the two matrices have compatible sizes and a commutator equal to zero within tolerance.

**cirq.canonicalize_half_turns**

```python
cirq.canonicalize_half_turns(half_turns: Union[float, sympy.core.basic.Basic]) → Union[float, sympy.core.basic.Basic]
```

Wraps the input into the range (-1, +1].

**cirq.chosen_angle_to_canonical_half_turns**

```python
cirq.chosen_angle_to_canonical_half_turns(half_turns: Union[float, sympy.core.basic.Basic, None] = None, rads: Optional[float] = None, degs: Optional[float] = None, default: float = 1.0) → Union[float, sympy.core.basic.Basic]
```

Returns a canonicalized half_turns based on the given arguments.

At most one of half_turns, rads, degs must be specified. If none are specified, the output defaults to half_turns=1.

**Parameters**

- **half_turns** – The number of half turns to rotate by.
- **rads** – The number of radians to rotate by.
- **degs** – The number of degrees to rotate by
- **default** – The half turns angle to use if nothing else is specified.

**Returns** A number of half turns.

**cirq.chosen_angle_to_half_turns**

```python
cirq.chosen_angle_to_half_turns(half_turns: Union[float, sympy.core.basic.Basic, None] = None, rads: Optional[float] = None, degs: Optional[float] = None, default: float = 1.0) → Union[float, sympy.core.basic.Basic]
```

Returns a half_turns value based on the given arguments.

At most one of half_turns, rads, degs must be specified. If none are specified, the output defaults to half_turns=1.

**Parameters**

- **half_turns** – The number of half turns to rotate by.
- **rads** – The number of radians to rotate by.
- **degs** – The number of degrees to rotate by
- **default** – The half turns angle to use if nothing else is specified.
Returns
A number of half turns.

cirq.CONTROL_TAG

cirq.CONTROL_TAG = array([[nan, 0.], [0., 1.]])

cirq.diagonalize_real_symmetric_and_sorted_diagonal_matrices

cirq.diagonalize_real_symmetric_and_sorted_diagonal_matrices
(symmetric_matrix: numpy.ndarray,
 diagonal_matrix: numpy.ndarray, *,
 rtol: float = 1e-05,
 atol: float = 1e-08,
 check_preconditions: bool = True) →
numpy.ndarray

Returns an orthogonal matrix that diagonalizes both given matrices.

The given matrices must commute.
Guarantees that the sorted diagonal matrix is not permuted by the
diagonalization (except for nearly-equal values).

Parameters

• symmetric_matrix – A real symmetric matrix.
• diagonal_matrix – A real diagonal matrix with entries along the diagonal sorted into
descending order.
• rtol – Relative numeric error threshold.
• atol – Absolute numeric error threshold.
• check_preconditions – If set, verifies that the input matrices commute and are re-
spectively symmetric and diagonal descending.

Returns
An orthogonal matrix \( P \) such that \( P^T \times \text{symmetric\_matrix} \times P \) is diagonal and \( P^T \times \text{diagonal\_matrix} \times P = \text{diagonal\_matrix} \) (up to tolerance).

 Raises
ValueError – Matrices don’t meet preconditions (e.g. not symmetric).

cirq.diagonalize_real_symmetric_matrix

cirq.diagonalize_real_symmetric_matrix
(matrix: numpy.ndarray, *,
 rtol: float = 1e-05,
 atol: float = 1e-08,
 check_preconditions: bool = True) →
numpy.ndarray

Returns an orthogonal matrix that diagonalizes the given matrix.

Parameters

• matrix – A real symmetric matrix to diagonalize.
• rtol – Relative error tolerance.
• atol – Absolute error tolerance.
check_preconditions – If set, verifies that the input matrix is real and symmetric.

Returns An orthogonal matrix P such that P.T @ matrix @ P is diagonal.

Raises ValueError – Matrix isn’t real symmetric.

cirq.dot

cirq.dot(*values: Union[float, complex, numpy.ndarray]) → Union[float, complex, numpy.ndarray]

Computes the dot/matrix product of a sequence of values.

Performs the computation in serial order without regard to the matrix sizes. If you are using this for matrices of large and differing sizes, consider using np.linalg.multi_dot for better performance.

Parameters *values – The values to combine with the dot/matrix product.

Returns The resulting value or matrix.

cirq.Duration


A time delta that supports symbols and picosecond accuracy.


Initializes a Duration with a time specified in some unit.

If multiple arguments are specified, their contributions are added.

Parameters

• value – A value with a pre-specified time unit. Currently only supports 0 and datetime.timedelta instances.

• picos – A number of picoseconds to add to the time delta.

• nanos – A number of nanoseconds to add to the time delta.

• micros – A number of microseconds to add to the time delta.

• millis – A number of milliseconds to add to the time delta.

Examples

```python
>>> print(cirq.Duration(nanos=100))
100 ns
>>> print(cirq.Duration(micros=1.5 * sympy.Symbol('t')))  
(1500.0*t) ns
```
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create(duration)</code></td>
<td>THIS FUNCTION IS DEPRECATED. Returns the number of microseconds that the duration spans.</td>
</tr>
<tr>
<td><code>total_micros()</code></td>
<td>Returns the number of microseconds that the duration spans.</td>
</tr>
<tr>
<td><code>total_millis()</code></td>
<td>Returns the number of milliseconds that the duration spans.</td>
</tr>
<tr>
<td><code>total_nanos()</code></td>
<td>Returns the number of nanoseconds that the duration spans.</td>
</tr>
<tr>
<td><code>total_picos()</code></td>
<td>Returns the number of picoseconds that the duration spans.</td>
</tr>
</tbody>
</table>

### `cirq.Duration.create`

```python
classmethod Duration.create(duration: Union[None, datetime.timedelta, cirq.Duration]) → Duration
```

THIS FUNCTION IS DEPRECATED. IT WILL BE REMOVED IN cirq v0.7. Use `cirq.Duration(...)` instead.

Creates a Duration from `datetime.timedelta` if necessary.

### `cirq.Duration.total_micros`

```python
Duration.total_micros() → Union[sympy.core.basic.Basic, float]
```

Returns the number of microseconds that the duration spans.

### `cirq.Duration.total_millis`

```python
Duration.total_millis() → Union[sympy.core.basic.Basic, float]
```

Returns the number of milliseconds that the duration spans.

### `cirq.Duration.total_nanos`

```python
Duration.total_nanos() → Union[sympy.core.basic.Basic, float]
```

Returns the number of nanoseconds that the duration spans.

### `cirq.Duration.total_picos`

```python
Duration.total_picos() → Union[sympy.core.basic.Basic, float]
```

Returns the number of picoseconds that the duration spans.

### `cirq.expand_matrix_in_orthogonal_basis`

```python
cirq.expand_matrix_in_orthogonal_basis(m: numpy.ndarray, basis: Dict[str, numpy.ndarray]) → cirq.value.linear_dict.LinearDict[str]
```

Computes coefficients of expansion of m in basis.
We require that basis be orthogonal w.r.t. the Hilbert-Schmidt inner product. We do not require that basis be orthonormal. Note that Pauli basis (I, X, Y, Z) is orthogonal, but not orthonormal.

**cirq.hilbert_schmidt_inner_product**

cirq.hilbert_schmidt_inner_product (m1: numpy.ndarray, m2: numpy.ndarray) → complex

Computes Hilbert-Schmidt inner product of two matrices.

Linear in second argument.

**cirq.is_diagonal**

cirq.is_diagonal (matrix: numpy.ndarray, *, atol: float = 1e-08) → bool

Determines if a matrix is a approximately diagonal.

A matrix is diagonal if i≠j implies m[i,j]==0.

**Parameters**

- **matrix** – The matrix to check.

- **atol** – The per-matrix-entry absolute tolerance on equality.

**Returns** Whether the matrix is diagonal within the given tolerance.

**cirq.is_hermitian**

cirq.is_hermitian (matrix: numpy.ndarray, *, rtol: float = 1e-05, atol: float = 1e-08) → bool

Determines if a matrix is approximately Hermitian.

A matrix is Hermitian if it’s square and equal to its adjoint.

**Parameters**

- **matrix** – The matrix to check.

- **rtol** – The per-matrix-entry relative tolerance on equality.

- **atol** – The per-matrix-entry absolute tolerance on equality.

**Returns** Whether the matrix is Hermitian within the given tolerance.

**cirq.is_negligible_turn**

cirq.is_negligible_turn (turns: float, tolerance: float) → bool

**cirq.is_orthogonal**

cirq.is_orthogonal (matrix: numpy.ndarray, *, rtol: float = 1e-05, atol: float = 1e-08) → bool

Determines if a matrix is approximately orthogonal.

A matrix is orthogonal if it’s square and real and its transpose is its inverse.
Parameters

- **matrix** – The matrix to check.
- **rtol** – The per-matrix-entry relative tolerance on equality.
- **atol** – The per-matrix-entry absolute tolerance on equality.

Returns Whether the matrix is orthogonal within the given tolerance.

cirq.is_special_orthogonal

cirq.is_special_orthogonal(matrix: numpy.ndarray, *, rtol: float = 1e-05, atol: float = 1e-08) → bool

Determines if a matrix is approximately special orthogonal.

A matrix is special orthogonal if it is square and real and its transpose is its inverse and its determinant is one.

Parameters

- **matrix** – The matrix to check.
- **rtol** – The per-matrix-entry relative tolerance on equality.
- **atol** – The per-matrix-entry absolute tolerance on equality.

Returns Whether the matrix is special orthogonal within the given tolerance.

cirq.is_special_unitary

cirq.is_special_unitary(matrix: numpy.ndarray, *, rtol: float = 1e-05, atol: float = 1e-08) → bool

Determines if a matrix is approximately unitary with unit determinant.

A matrix is special-unitary if it is square and its adjoint is its inverse and its determinant is one.

Parameters

- **matrix** – The matrix to check.
- **rtol** – The per-matrix-entry relative tolerance on equality.
- **atol** – The per-matrix-entry absolute tolerance on equality.

Returns Whether the matrix is unitary with unit determinant within the given tolerance.

cirq.is_unitary

cirq.is_unitary(matrix: numpy.ndarray, *, rtol: float = 1e-05, atol: float = 1e-08) → bool

Determines if a matrix is approximately unitary.

A matrix is unitary if it’s square and its adjoint is its inverse.

Parameters
• **matrix** – The matrix to check.
• **rtol** – The per-matrix-entry relative tolerance on equality.
• **atol** – The per-matrix-entry absolute tolerance on equality.

**Returns** Whether the matrix is unitary within the given tolerance.

cirq.kak_canonicalize_vector

cirq.kak_canonicalize_vector(x: float, y: float, z: float, atol: float = 1e-09) →
cirq.linalg.decompositions.KakDecomposition

Canonicalizes an XX/YY/ZZ interaction by swap/negate/shift-ing axes.

**Parameters**
• **x** – The strength of the XX interaction.
• **y** – The strength of the YY interaction.
• **z** – The strength of the ZZ interaction.
• **atol** – How close x2 must be to /4 to guarantee z2 >= 0

**Returns**
The canonicalized decomposition, with vector coefficients (x2, y2, z2) satisfying:

\[
0 \quad \text{abs}(z2) \quad y2 \quad x2 /4 \text{ if } x2 = /4, z2 >= 0
\]

Guarantees that the implied output matrix:

\[
g \cdot (a1 \quad a0) \cdot \exp(i \cdot (x2 \cdot XX + y2 \cdot YY + z2 \cdot ZZ)) \cdot (b1 \quad b0)
\]

is approximately equal to the implied input matrix:

\[
\exp(i \cdot (x \cdot XX + y \cdot YY + z \cdot ZZ))
\]

cirq.kak_decomposition

cirq.kak_decomposition(unitary_object: Union[numpy.ndarray, cirq.SupportsUnitary], *, rtol: float = 1e-05, atol: float = 1e-08, check_preconditions: bool = True) →
cirq.linalg.decompositions.KakDecomposition

Decomposes a 2-qubit unitary into 1-qubit ops and XX/YY/ZZ interactions.

**Parameters**
• **unitary_object** – The value to decompose. Can either be a 4x4 unitary matrix, or an object that has a 4x4 unitary matrix (via the cirq.SupportsUnitary protocol).
• **rtol** – Per-matrix-entry relative tolerance on equality.
• **atol** – Per-matrix-entry absolute tolerance on equality.
• **check_preconditions** – If set, verifies that the input corresponds to a 4x4 unitary before decomposing.

**Returns**
A cirq.KakDecomposition canonicalized such that the interaction coefficients x, y, z satisfy:

\[
0 \quad \text{abs}(z2) \quad y2 \quad x2 /4 \text{ if } x2 = /4, z2 >= 0
\]

**Raises**
• **ValueError** – Bad matrix.
• **ArithmeticError** – Failed to perform the decomposition.

**References**


cirq.kak_vector

cirq.kak_vector(unitary: Union[Iterable[numpy.ndarray], numpy.ndarray], *, rtol: float = 1e-05, atol: float = 1e-08, check_preconditions: bool = True) → numpy.ndarray

Compute the KAK vectors of one or more two qubit unitaries.

Any 2 qubit unitary may be expressed as

$$ U = k_l A k_r $$

where $k_l, k_r$ are single qubit (local) unitaries and

$$ A= \exp \left( i \sum_{s=x,y,z} k_s \sigma_{s}^{(0)} \sigma_{s}^{(1)} \right) $$

The vector entries are ordered such that

$$ 0 |k_z| k_y k_x /4 $$

if $k_x$ = /4, $k_z \geq 0$.

**Parameters**

• **unitary** – A unitary matrix, or a multi-dimensional array of unitary matrices. Must have shape $(\ldots, 4, 4)$, where the last two axes are for the unitary matrix and other axes are for broadcasting the kak vector computation.

• **rtol** – Per-matrix-entry relative tolerance on equality. Used in unitarity check of input.

• **atol** – Per-matrix-entry absolute tolerance on equality. Used in unitarity check of input.

  This also determines how close $k_x$ must be to /4 to guarantee $k_z \geq 0$. Must be non-negative.

• **check_preconditions** – When set to False, skips verifying that the input is unitary in order to increase performance.

**Returns** The KAK vector of the given unitary or unitaries. The output shape is the same as the input shape, except the two unitary matrix axes are replaced by the kak vector axis (i.e. the output has shape `unitary.shape[:-2] + (3,)`).

**References**

Examples

```python
>>> cirq.kak_vector(np.eye(4))
array([0., 0., 0.])
>>> unitaries = [cirq.unitary(cirq.CZ), cirq.unitary(cirq.ISWAP)]
>>> cirq.kak_vector(unitaries) * 4 / np.pi
array([[ 1., 0., -0.],
       [ 1., 1., 0.]])
```

cirq.KakDecomposition

class cirq.KakDecomposition(*, global_phase: complex, single_qubit_operations_before: Tuple[numpy.ndarray, numpy.ndarray], interaction_coefficients: Tuple[float, float, float], single_qubit_operations_after: Tuple[numpy.ndarray, numpy.ndarray])

A convenient description of an arbitrary two-qubit operation. Any two qubit operation \( U \) can be decomposed into the form

\[
U = g \cdot (a1 \ a0) \cdot \exp(i \cdot (x \cdot XX + y \cdot YY + z \cdot ZZ)) \cdot (b1 \ b0)
\]

This class stores \( g, (b0, b1), (x, y, z), \) and \( (a0, a1) \).

**global_phase**

\( g \) from the above equation.

**single_qubit_operations_before**

\( b0, b1 \) from the above equation.

**interaction_coefficients**

\( x, y, z \) from the above equation.

**single_qubit_operations_after**

\( a0, a1 \) from the above equation.

References

‘An Introduction to Cartan’s KAK Decomposition for QC Programmers’

```python
__init__(*, global_phase: complex, single_qubit_operations_before: Tuple[numpy.ndarray, numpy.ndarray], interaction_coefficients: Tuple[float, float, float], single_qubit_operations_after: Tuple[numpy.ndarray, numpy.ndarray])
```

Initializes a decomposition for a two-qubit operation \( U \).

\[
U = g \cdot (a1 \ a0) \cdot \exp(i \cdot (x \cdot XX + y \cdot YY + z \cdot ZZ)) \cdot (b1 \ b0)
\]

**Parameters**

- **global_phase** – \( g \) from the above equation.
- **single_qubit_operations_before** – \( b0, b1 \) from the above equation.
- **interaction_coefficients** – \( x, y, z \) from the above equation.
• `single_qubit_operations_after` – a0, a1 from the above equation.

**Methods**

---

cirq.kron

cirq.kron(*factors: Union[numpy.ndarray, complex, float]) → numpy.ndarray

Computes the kronecker product of a sequence of values.

A *args version of lambda args: functools.reduce(np.kron, args).

- **Parameters** `*factors` – The matrices, tensors, and/or scalars to combine together using np.kron.
- **Returns** The kronecker product of all the inputs.

cirq.kron_factor_4x4_to_2x2s

cirq.kron_factor_4x4_to_2x2s(matrix: numpy.ndarray) → Tuple[complex, numpy.ndarray, numpy.ndarray]

Splits a 4x4 matrix U = kron(A, B) into A, B, and a global factor.

Requires the matrix to be the kronecker product of two 2x2 unitaries.
Requires the matrix to have a non-zero determinant.
Giving an incorrect matrix will cause garbage output.

- **Parameters** `matrix` – The 4x4 unitary matrix to factor.
- **Returns** A scalar factor and a pair of 2x2 unit-determinant matrices. The kronecker product of all three is equal to the given matrix.
- **Raises** ValueError – The given matrix can’t be tensor-factored into 2x2 pieces.

cirq.kron_with_controls

cirq.kron_with_controls(*factors: Union[numpy.ndarray, complex, float]) → numpy.ndarray

Computes the kronecker product of a sequence of values and control tags.

Use `cirq.CONTROL_TAG` to represent controls. Any entry of the output corresponding to a situation where the control is not satisfied will be overwritten by identity matrix elements.

The control logic works by imbuing NaN with the meaning “failed to meet one or more controls”. The normal kronecker product then spreads the per-item NaNs to all the entries in the product that need to be replaced by identity matrix elements. This method rewrites those NaNs. Thus CONTROL_TAG can be the matrix `[[NaN, 0], [0, 1]]` or equivalently `[[NaN, NaN], [NaN, 1]]`. 
Because this method re-interprets NaNs as control-failed elements, it won’t propagate error-indicating NaNs from its input to its output in the way you’d otherwise expect.

Examples

```python
result = cirq.kron_with_controls(cirq.CONTROL_TAG, cirq.unitary(cirq.X))
print(result.astype(np.int32))
```

# prints:
# [[1 0 0 0]
#  [0 1 0 0]
#  [0 0 0 1]
#  [0 0 1 0]]

**Parameters**

*+factors* – The matrices, tensors, scalars, and/or control tags to combine together using np.kron.

**Returns**
The resulting matrix.

cirq.LinearDict

```python
class cirq.LinearDict(terms: Optional[Mapping[TVector, Union[complex, float]]] = None, validator: Callable[TVector, bool] = <function LinearDict.<lambda>>) -> None
```

Initializes linear combination from a collection of terms.

LinearDict implements the basic linear algebraic operations of vector addition and scalar multiplication for linear combinations of abstract vectors. Keys represent the vectors, values represent their coefficients. The only requirement on the keys is that they be hashable (i.e. are immutable and implement hash and eq with equal objects hashing to equal values).

A consequence of treating keys as opaque is that all relationships between the keys other than equality are ignored. In particular, keys are allowed to be linearly dependent.

```python
__init__(terms: Optional[Mapping[TVector, Union[complex, float]]] = None, validator: Callable[TVector, bool] = <function LinearDict.<lambda>>) -> None
```

Initializes linear combination from a collection of terms.

**Parameters**

* terms – Mapping of abstract vectors to coefficients in the linear combination being initialized.
• **validator** – Optional predicate that determines whether a vector is valid or not. Dictionary and linear algebra operations that would lead to the inclusion of an invalid vector into the combination raise ValueError exception. By default all vectors are valid.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clean(*[, atol])</code></td>
<td>Remove terms with coefficients of absolute value atol or less.</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>LinearDict.clear() → None. Remove all items from D.</td>
</tr>
<tr>
<td><code>copy()</code></td>
<td>LinearDict.copy() → TSelf</td>
</tr>
<tr>
<td><code>fromkeys(vectors[, coefficient])</code></td>
<td>classmethod LinearDict.fromkeys(vectors, coefficient=0)</td>
</tr>
<tr>
<td><code>get(k[, d])</code></td>
<td>LinearDict.get(k[, d]) → D[k] if k in D, else d. d defaults to None.</td>
</tr>
<tr>
<td><code>items()</code></td>
<td></td>
</tr>
<tr>
<td><code>keys()</code></td>
<td></td>
</tr>
<tr>
<td><code>pop(k[, d])</code></td>
<td>If key is not found, d is returned if given, otherwise KeyError is raised.</td>
</tr>
<tr>
<td><code>popitem()</code></td>
<td>as a 2-tuple; but raise KeyError if D is empty.</td>
</tr>
<tr>
<td><code>setdefault(k[, d])</code></td>
<td></td>
</tr>
<tr>
<td><code>update([E, ]**F)</code></td>
<td>If E present and has a .keys() method, does: for k in E: D[k] = E[k]</td>
</tr>
<tr>
<td><code>values()</code></td>
<td></td>
</tr>
</tbody>
</table>

**cirq.LinearDict.clean**

LinearDict.clean(*, atol: float = 1e-09) → TSelf
Remove terms with coefficients of absolute value atol or less.

**cirq.LinearDict.clear**

LinearDict.clear() → None. Remove all items from D.

**cirq.LinearDict.copy**

LinearDict.copy() → TSelf

**cirq.LinearDict.fromkeys**

classmethod LinearDict.fromkeys(vectors, coefficient=0)

**cirq.LinearDict.get**

LinearDict.get(k[, d]) → D[k] if k in D, else d. d defaults to None.
cirq.LinearDict.items

LinearDict.items() → a set-like object providing a view on D’s items

cirq.LinearDict.keys

LinearDict.keys() → a set-like object providing a view on D’s keys

cirq.LinearDict.pop

LinearDict.pop(k[, d]) → v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised.

cirq.LinearDict.popitem

LinearDict.popitem() → (k, v), remove and return some (key, value) pair as a 2-tuple; but raise KeyError if D is empty.

cirq.LinearDict.setdefault

LinearDict.setdefault(k[, d]) → D.get(k,d), also set D[k]=d if k not in D

cirq.LinearDict.update

LinearDict.update([E], **F) → None. Update D from mapping/iterable E and F.

    If E present and has a .keys() method, does: for k in E: D[k] = E[k]
    If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v
    In either case, this is followed by: for k, v in F.items(): D[k] = v

cirq.LinearDict.values

LinearDict.values() → an object providing a view on D’s values

Attributes

TSelf

cirq.LinearDict.TSelf

LinearDict.TSelf = ~TSelf
cirq.map_eigenvalues

cirq.map_eigenvalues(matrix: numpy.ndarray, func: Callable[complex, complex], *, rtol: float = 1e-05, atol: float = 1e-08) \rightarrow numpy.ndarray

Applies a function to the eigenvalues of a matrix.

Given \( M = \sum_k a_k |v_k><v_k| \).

Parameters

- matrix – The matrix to modify with the function.
- func – The function to apply to the eigenvalues of the matrix.
- rtol – Relative threshold used when separating eigenspaces.
- atol – Absolute threshold used when separating eigenspaces.

Returns

The transformed matrix.

cirq.match_global_phase

cirq.match_global_phase(a: numpy.ndarray, b: numpy.ndarray) \rightarrow Tuple[numpy.ndarray, numpy.ndarray]

Phases the given matrices so that they agree on the phase of one entry.

To maximize precision, the position with the largest entry from one of the matrices is used when attempting to compute the phase difference between the two matrices.

Parameters

- a – A numpy array.
- b – Another numpy array.

Returns

A tuple (a’, b’) where a’ == b’ implies a == b*exp(i t) for some t.

cirq.matrix_from_basis_coefficients

cirq.matrix_from_basis_coefficients(expansion: cirq.value.linear_dict.LinearDict[str], basis: Dict[str, numpy.ndarray]) \rightarrow numpy.ndarray

Computes linear combination of basis vectors with given coefficients.

cirq.partial_trace

cirq.partial_trace(tensor: numpy.ndarray, keep_indices: List[int]) \rightarrow numpy.ndarray

Takes the partial trace of a given tensor.

The input tensor must have shape \( (d_0, \ldots, d_{k-1}, d_0, \ldots, d_{k-1}) \).

The trace is done over all indices that are not in keep_indices. The resulting tensor has shape \( (d_{i_0}, \ldots, d_{i_r}, d_{i_0}, \ldots, d_{i_r}) \)

where \( i_j \) is the jth element of keep_indices.
Parameters

• **tensor** – The tensor to sum over. This tensor must have a shape \((d_0, \ldots, d_{k-1}, d_0, \ldots, d_{k-1})\).

• **keep_indices** – Which indices to not sum over. These are only the indices of the first half of the tensors indices (i.e. all elements must be between 0 and tensor.ndims / 2 - 1 inclusive).

Raises **ValueError** – if the tensor is not of the correct shape or the indices are not from the first half of valid indices for the tensor.

cirq.PeriodicValue

class cirq.PeriodicValue(value: Union[int, float], period: Union[int, float])
Wrapper for periodic numerical values.

Wrapper for periodic numerical types which implements \_eq\_, \_ne\_, \_hash\_ and \_approx_eq\_ so that values which are in the same equivalence class are treated as equal.

Internally the \texttt{value} passed to \_\texttt{init}\_ is normalized to the interval \([0, \text{period})\) and stored as that. Specialized version of \_approx_eq\_ is provided to cover values which end up at the opposite edges of this interval.

\_\texttt{init}\_. (value: Union[int, float], period: Union[int, float])
Initializes the equivalence class.

Parameters

• **value** – numerical value to wrap.

• **period** – periodicity of the numerical value.

Methods

cirq.reflection_matrix_pow

cirq.reflection_matrix_pow(reflection_matrix: numpy.ndarray, exponent: float)
Raises a matrix with two opposing eigenvalues to a power.

Parameters

• **reflection_matrix** – The matrix to raise to a power.

• **exponent** – The power to raise the matrix to.

Returns The given matrix raised to the given power.
cirq.slice_for_qubits_equal_to

cirq.slice_for_qubits_equal_to(target_qubit_axes: Sequence[int], little_endian_qureg_value: int = 0, *, big_endian_qureg_value: int = 0, num_qubits: Optional[int] = None, qid_shape: Optional[Tuple[int, ...]] = None) → Tuple[Union[slice, int, ellipsis], ...]

Returns an index corresponding to a desired subset of an np.ndarray.

It is assumed that the np.ndarray’s shape is of the form (2, 2, 2, ..., 2).

Example

```python
# A '4 qubit' tensor with values from 0 to 15.
r = np.array(range(16)).reshape((2,) * 4)

# We want to index into the subset where qubit #1 and qubit #3 are ON.
s = cirq.slice_for_qubits_equal_to([1, 3], 0b11)
print(s)
# (slice(None, None, None), 1, slice(None, None, None), 1, Ellipsis)

# Get that subset. It corresponds to numbers of the form 0b*1*1.
# where here '*' indicates any possible value.
print(r[s])
# [[ 5  7]
#  [13 15]]
```

Parameters

- **target_qubit_axes** – The qubits that are specified by the index bits. All other axes of the slice are unconstrained.

- **little_endian_qureg_value** – An integer whose bits specify what value is desired for of the target qubits. The integer is little endian w.r.t. the target qubit axes, meaning the low bit of the integer determines the desired value of the first targeted qubit, and so forth with the k’th targeted qubit’s value set to bool(qureg_value & (1 << k)).

- **big_endian_qureg_value** – Same as **little_endian_qureg_value** but big endian w.r.t. to target qubit axes, meaning the low bit of the integer determines the desired value of the last target qubit, and so forth. Specify exactly one of the *_qureg_value arguments.

- **num_qubits** – If specified the slices will extend all the way up to this number of qubits, otherwise if it is None, the final element return will be Ellipsis. Optional and defaults to using Ellipsis.

- **qid_shape** – The qid shape of the state vector being sliced. Specify this instead of **num_qubits** when using qids with dimension != 2. The qureg value is interpreted to store digits with corresponding bases packed into an int.

Returns An index object that will slice out a mutable view of the desired subset of a tensor.

cirq.so4_to_magic_su2s

cirq.so4_to_magic_su2s(mat: numpy.ndarray, *, rtol: float = 1e-05, atol: float = 1e-08, check_preconditions: bool = True) → Tuple[numpy.ndarray,.numpy.ndarray]

Finds 2x2 special-unitaries A, B where mat = Mag.H @ kron(A, B) @ Mag.
Mag is the magic basis matrix:

\[
\begin{pmatrix}
1 & 0 & 0 & i \\
0 & i & 1 & 0 \\
0 & i & -1 & 0 \\
1 & 0 & 0 & -i
\end{pmatrix}
\] (times sqrt(0.5) to normalize)

**Parameters**

- **mat** – A real 4x4 orthogonal matrix.
- **rtol** – Per-matrix-entry relative tolerance on equality.
- **atol** – Per-matrix-entry absolute tolerance on equality.
- **check_preconditions** – When set, the code verifies that the given matrix is from SO(4). Defaults to set.

**Returns** A pair (A, B) of matrices in SU(2) such that Mag.H @ kron(A, B) @ Mag is approximately equal to the given matrix.

**Raises** ValueError – Bad matrix.

cirq.targeted_conjugate_about

cirq\._targeted_conjugate\_about(tensor: \_numpy.ndarray, target: \_numpy.ndarray, indices: Sequence[int], conj_indices: Sequence[int] = None, buffer: Optional[\_numpy.ndarray] = None, out: Optional[\_numpy.ndarray] = None) → \_numpy.ndarray

Conjugates the given tensor about the target tensor.

This method computes a target tensor conjugated by another tensor. Here conjugate is used in the sense of conjugating by a matrix, i.e.

A conjugated about B is $A B A^\dagger$ where $A^\dagger$ represents the conjugate transpose.

Abstractly this compute $A \cdot B \cdot A^\dagger$ where A and B are multi-dimensional arrays, and instead of matrix multiplication $\cdot$ is a contraction between the given indices (indices for first $\cdot$, conj_indices for second $\cdot$).

More specifically this computes

\[
\text{sum } tensor_{i_0,\ldots,i_{r-1},j_0,\ldots,j_{r-1}} \times \text{target}_{k_0,\ldots,k_{r-1},l_0,\ldots,l_{r-1}} \times tensor_{m_0,\ldots,m_{r-1},n_0,\ldots,n_{r-1}}^* \\
\]

where the sum is over indices where $j_s = k_s$ and $s$ is in indices and $l_s = m_s$ and $s$ is in conj_indices.

**Parameters**

- **tensor** – The tensor that will be conjugated about the target tensor.
• **target** – The tensor that will receive the conjugation.

• **indices** – The indices which will be contracted between the tensor and target.

• **The indices which will be contracted between the**
  
  (conj_indices;)
  
  – complex conjugate of the tensor and the target. If this is None, then these will be the values in indices plus half the number of dimensions of the target (ndim). This is the most common case and corresponds to the case where the target is an operator on a n-dimensional tensor product space (here n would be ndim).

• **buffer** – A buffer to store partial results in. If not specified or None, a new buffer is used.

• **out** – The buffer to store the results in. If not specified or None, a new buffer is used. Must have the same shape as target.

**Returns** The result the conjugation.

cirq.targeted_left_multiply
cirq.
cirq.targeted_left_multiply(leaf_matrix: numpy.ndarray, right_target: numpy.ndarray, target_axes: Sequence[int], out: Optional[numpy.ndarray] = None) → numpy.ndarray

Left-multiplies the given axes of the target tensor by the given matrix.

Note that the matrix must have a compatible tensor structure.

For example, if you have an 6-qubit state vector input_state with shape (2, 2, 2, 2, 2), and a 2-qubit unitary operation op with shape (2, 2, 2), and you want to apply op to the 5'th and 3'rd qubits within input_state, then the output state vector is computed as follows:

```
output_state = cirq.targeted_left_multiply(op, input_state, [5, 3])
```

This method also works when the right hand side is a matrix instead of a vector. If a unitary circuit’s matrix is old_effect, and you append a CNOT(q1, q4) operation onto the circuit, where the control q1 is the qubit at offset 1 and the target q4 is the qubit at offset 4, then the appended circuit’s unitary matrix is computed as follows:

```
new_effect = cirq.targeted_left_multiply(
    left_matrix=cirq.unitary(cirq.CNOT).reshape((2, 2, 2, 2)),
    right_target=old_effect,
    target_axes=[1, 4])
```

**Parameters**

• **left_matrix** – What to left-multiply the target tensor by.

• **right_target** – A tensor to carefully broadcast a left-multiply over.

• **target_axes** – Which axes of the target are being operated on.
• **out** – The buffer to store the results in. If not specified or None, a new buffer is used. Must have the same shape as right_target.

**Returns**  The output tensor.

---

### cirq.TextDiagramDrawer

**class cirq.TextDiagramDrawer**

A utility class for creating simple text diagrams.

```python
def __init__()
    Initialize self. See help(type(self)) for accurate signature.
```

**Methods**

- **content_present(x, y)** Determines if a line or printed text is at the given location.

- **force_horizontal_padding_after(index, padding)** Change the padding after the given column.

- **force_vertical_padding_after(index, padding)** Change the padding after the given row.

- **grid_line(x1, y1, x2, y2[, emphasize])** Adds a vertical or horizontal line from (x1, y1) to (x2, y2).

- **height()** Determines how many entry rows are in the diagram.

- **horizontal_line(y, x1, x2[, emphasize])** Adds a line from (x1, y) to (x2, y).

- **insert_empty_columns(x[, amount])** Insert a number of columns after the given column.

- **insert_empty_rows(y[, amount])** Insert a number of rows after the given row.

- **render([horizontal_spacing, ...])** Outputs text containing the diagram.

- **transpose()** Returns the same diagram, but mirrored across its diagonal.

- **vertical_line(x, y1, y2[, emphasize])** Adds a line from (x, y1) to (x, y2).

- **width()** Determines how many entry columns are in the diagram.

- **write(x, y, text[, transposed_text])** Adds text to the given location.

---

### cirq.TextDiagramDrawer.content_present

```python
class DiagramDrawer.content_present(x: int, y: int) -> bool
    Determines if a line or printed text is at the given location.
```

### cirq.TextDiagramDrawer.force_horizontal_padding_after

```python
class DiagramDrawer.force_horizontal_padding_after(index: int, padding: Union[int, float]) -> None
    Change the padding after the given column.
```
**cirq.TextDiagramDrawer.force_vertical_padding_after**

TextDiagramDrawer\texttt{.force\_vertical\_padding\_after}(index: int, padding: Union[int, float]) → None

Change the padding after the given row.

**cirq.TextDiagramDrawer.grid_line**

TextDiagramDrawer\texttt{.grid\_line}(x1: int, y1: int, x2: int, y2: int, emphasize: bool = False)

\begin{verbatim}
Adds a vertical or horizontal line from (x1, y1) to (x2, y2).

Horizontal line is selected on equality in the second coordinate and vertical line is selected on equality in the first coordinate.

\textbf{Raises \texttt{ValueError}} – If line is neither horizontal nor vertical.
\end{verbatim}

**cirq.TextDiagramDrawer.height**

TextDiagramDrawer\texttt{.height}() → int

Determined how many entry rows are in the diagram.

**cirq.TextDiagramDrawer.horizontal_line**

TextDiagramDrawer\texttt{.horizontal\_line}(y: Union[int, float], x1: Union[int, float], x2: Union[int, float], emphasize: bool = False) → None

\begin{verbatim}
Adds a line from (x1, y) to (x2, y).
\end{verbatim}

**cirq.TextDiagramDrawer.insert_empty_columns**

TextDiagramDrawer\texttt{.insert\_empty\_columns}(x: int, amount: int = 1) → None

Insert a number of columns after the given column.

**cirq.TextDiagramDrawer.insert_empty_rows**

TextDiagramDrawer\texttt{.insert\_empty\_rows}(y: int, amount: int = 1) → None

Insert a number of rows after the given row.

**cirq.TextDiagramDrawer.render**

TextDiagramDrawer\texttt{.render}(horizontal\_spacing: int = 1, vertical\_spacing: int = 1, crossing\_char: str = None, use\_unicode\_characters: bool = True) → str

\begin{verbatim}
Outputs text containing the diagram.
\end{verbatim}
cirq.TextDiagramDrawer.transpose

TextDiagramDrawer.transpose() \rightarrow cirq.circuits.text_diagram_drawer.TextDiagramDrawer

Returns the same diagram, but mirrored across its diagonal.

cirq.TextDiagramDrawer.vertical_line

TextDiagramDrawer.vertical_line(x: Union[int, float], y1: Union[int, float], y2: Union[int, float], emphasize: bool = False) \rightarrow None

Adds a line from (x, y1) to (x, y2).

cirq.TextDiagramDrawer.width

TextDiagramDrawer.width() \rightarrow int

Determines how many entry columns are in the diagram.

cirq.TextDiagramDrawer.write

TextDiagramDrawer.write(x: int, y: int, text: str, transposed_text: Optional[str] = None)

Adds text to the given location.

Parameters

• **x** – The column in which to write the text.
• **y** – The row in which to write the text.
• **text** – The text to write at location (x, y).
• **transposed_text** – Optional text to write instead, if the text diagram is transposed.

cirq.Timestamp

class cirq.Timestamp(*, picos: Union[int, float] = 0, nanos: Union[int, float] = 0)

A location in time with picosecond accuracy.

Supports affine operations against Duration.

__init__(*, picos: Union[int, float] = 0, nanos: Union[int, float] = 0) \rightarrow None

Initializes a Timestamp with a time specified in ns and/or ps.

The time is relative to some unspecified “time zero”. If both picos and nanos are specified, their contributions away from zero are added.

Parameters

• **picos** – How many picoseconds away from time zero?
• **nanos** – How many nanoseconds away from time zero?

Methods
The timestamp’s location in picoseconds from arbitrary time zero.

**cirq.Timestamp.raw_picos**

```python
Timestamp.raw_picos() \rightarrow float
```

The timestamp’s location in picoseconds from arbitrary time zero.

**cirq.value_equality**

```python
cirq.value_equality (cls: type = None, *, unhashable: bool = False, distinct_child_types: bool = False, manual_cls: bool = False, approximate: bool = False) \rightarrow Union[Callable[type, type], type]
```

Implements `eq/ne/hash` via a `value_equality_values` method.

`value_equality_values` is a method that the decorated class must implement.

`value_equality_approximate_values` is a method that the decorated class might implement if special support for approximate equality is required. This is only used when approximate argument is set. When approximate argument is set and `value_equality_approximate_values` is not defined, `value_equality_values` values are used for approximate equality.

For example, this can be used to compare periodic values like angles: the angle value can be wrapped with `PeriodicValue`. When returned as part of approximate values a special normalization will be done automatically to guarantee correctness.

Note that the type of the decorated value is included as part of the value equality values. This is so that completely separate classes with identical equality values (e.g. a `Point2D` and a `Vector2D`) don’t compare as equal. Further note that this means that child types of the decorated type will be considered equal to each other, though this behavior can be changed via the ‘distinct_child_types’ argument. The type logic is implemented behind the scenes by a ‘`value_equality_values_cls`’ method added to the class.

**Parameters**

- `cls` – The type to decorate. Automatically passed in by python when using the `@cirq.value_equality` decorator notation on a class.
- `unhashable` – When set, the `__hash__` method will be set to None instead of to a hash of the equality class and equality values. Useful for mutable types such as dictionaries.
- `distinct_child_types` – When set, classes that inherit from the decorated class will not be considered equal to it. Also, different child classes will not be considered equal to each other. Useful for when the decorated class is an abstract class or trait that is helping to define equality for many conceptually distinct concrete classes.
- `manual_cls` – When set, the method `value_equality_values_cls` must be implemented. This allows a new class to compare as equal to another existing class that is also
using value equality, by having the new class return the existing class’ type. Incompatible with \texttt{distinct_child_types}.

• \texttt{approximate} – When set, the decorated class will be enhanced with \_\texttt{approx_eq\_} implementation and thus start to support the \texttt{SupportsApproximateEquality} protocol.

\subsection*{3.1.17 Experiments}

Utilities for running experiments on hardware, or producing things required to run experiments.

\begin{verbatim}

generate_boixo_2018_supremacy_circuits_v2(...)  # Generates Google Random Circuits v2 as in github.com/sboixo/GRCS

generate_boixo_2018_supremacy_circuits_v2_bristlecone(...)  # Generates Google Random Circuits v2 in Bristlecone.

generate_boixo_2018_supremacy_circuits_v2_grid(...)  # Generates Google Random Circuits v2 as in github.com/sboixo/GRCS


cirq.generate_boixo_2018_supremacy_circuits_v2

cirq.generate_boixo_2018_supremacy_circuits_v2(qubits: Iterable[cirq.devices.grid_qubit.GridQubit],
                                      cz_depth: int, seed: int) → cirq.circuits.circuit.Circuit

Generates Google Random Circuits v2 as in github.com/sboixo/GRCS \texttt{cz\_v2}.
See also \url{https://arxiv.org/abs/1807.10749}

Parameters

\begin{itemize}
\item \texttt{qubits} – qubit grid in which to generate the circuit.
\item \texttt{cz_depth} – number of layers with CZ gates.
\item \texttt{seed} – seed for the random instance.
\end{itemize}

Returns A circuit corresponding to instance \texttt{inst_{n_rows\times c_{cz_depth+1}}(seed)}

The mapping of qubits is cirq.GridQubit(j,k) \rightarrow q[j*n\_cols+k] (as in the QASM mapping)

\begin{verbatim}

cirq.generate_boixo_2018_supremacy_circuits_v2_bristlecone

cirq.generate_boixo_2018_supremacy_circuits_v2_bristlecone(n_rows: int, cz_depth: int, seed: int) →
                                      cirq.circuits.circuit.Circuit

Generates Google Random Circuits v2 in Bristlecone.
See also \url{https://arxiv.org/abs/1807.10749}

Parameters
\end{verbatim}
• **n_rows** – number of rows in a Bristlecone lattice. Note that we do not include single qubit corners.
• **cz_depth** – number of layers with CZ gates.
• **seed** – seed for the random instance.

**Returns** A circuit with given size and seed.

cirq.generate_boixo_2018_supremacy_circuits_v2_grid

cirq.generate_boixo_2018_supremacy_circuits_v2_grid(n_rows: int, n_cols: int, cz_depth: int, seed: int) → cirq.circuits.circuit.Circuit

Generates Google Random Circuits v2 as in github.com/sboixo/GRCS cz_v2.
See also https://arxiv.org/abs/1807.10749

**Parameters**

• **n_rows** – number of rows of a 2D lattice.
• **n_cols** – number of columns.
• **cz_depth** – number of layers with CZ gates.
• **seed** – seed for the random instance.

**Returns** A circuit corresponding to instance inst_{n_rows}x{n_cols}_{cz_depth+1}_{seed}

The mapping of qubits is cirq.GridQubit(j,k) -> q[j*n_cols+k] (as in the QASM mapping)

### 3.1.18 Ion traps and neutral atoms

Support for ion trap an neutral atom devices.

<table>
<thead>
<tr>
<th><strong>Class</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConvertToIonGates</strong>([ignore_failures])</td>
<td>Attempts to convert non-native gates into IonGates.</td>
</tr>
<tr>
<td><strong>IonDevice</strong>(measurement_duration, ...)</td>
<td>A device with qubits placed on a line.</td>
</tr>
<tr>
<td><strong>MS</strong>(rads)</td>
<td>The Mølmer–Sørensen gate, a native two-qubit operation in ion traps.</td>
</tr>
<tr>
<td><strong>two_qubit_matrix_to_ion_operations</strong>(q0, q1, mat)</td>
<td>Decomposes a two-qubit operation into MS/single-qubit rotation gates.</td>
</tr>
<tr>
<td><strong>ConvertToNeutralAtomGates</strong>([ignore_failures])</td>
<td>Attempts to convert gates into native Atom gates.</td>
</tr>
<tr>
<td><strong>NeutralAtomDevice</strong>(measurement_duration, ...)</td>
<td>A device with qubits placed on a grid.</td>
</tr>
</tbody>
</table>

#### cirq.ConvertToIonGates

**class cirq.ConvertToIonGates** *(ignore_failures: bool = False)*

Attempts to convert non-native gates into IonGates.

**__init__**(ignore_failures: bool = False) → None

**Parameters** **ignore_failures** – If set, gates that fail to convert are forwarded unchanged. If not set, conversion failures raise a TypeError.
### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>convert_circuit(circuit)</code></td>
<td>Convert a single (one- or two-qubit) operation</td>
</tr>
<tr>
<td><code>convert_one(op)</code></td>
<td>Convert a single (one- or two-qubit) operation</td>
</tr>
</tbody>
</table>

#### cirq.ConvertToIonGates.convert_circuit

```python
```

Convert to ion trap native gates

**Parameters**
- `circuit` – Gate operation to be converted

**Returns**
- The desired operation implemented with ion trap gates

#### cirq.ConvertToIonGates.convert_one

```python
```

Convert a single (one- or two-qubit) operation into ion trap native gates

**Parameters**
- `op` – Gate operation to be converted

**Returns**
- The desired operation implemented with ion trap gates

#### cirq.IonDevice

```python
class cirq.IonDevice(measurement_duration: cirq.DURATION_LIKE, twoq_gates_duration: cirq.DURATION_LIKE, oneq_gates_duration: cirq.DURATION_LIKE, qubits: Iterable[cirq.devices.line_qubit.LineQubit])
```

A device with qubits placed on a line.

Qubits have all-to-all connectivity.

```python
cirq.IonDevice.__init__(measurement_duration: cirq.DURATION_LIKE, twoq_gates_duration: cirq.DURATION_LIKE, oneq_gates_duration: cirq.DURATION_LIKE, qubits:Iterable[cirq.devices.line_qubit.LineQubit]) → None
```

Initialize the description of an ion trap device.

**Parameters**
- `measurement_duration` – The maximum duration of a measurement.
- `twoq_gates_duration` – The maximum duration of a two qubit operation.
- `oneq_gates_duration` – The maximum duration of a single qubit operation.
- `qubits` – Qubits on the device, identified by their x, y location.

#### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>at(position)</code></td>
<td>Returns the qubit at the given position, if there is one, else None.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 192 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>can_add_operation_into_moment</code></td>
<td>Determines if it’s possible to add an operation into a moment.</td>
</tr>
<tr>
<td><code>decompose_circuit</code></td>
<td>Returns a device-valid decomposition for the given operation.</td>
</tr>
<tr>
<td><code>decompose_operation</code></td>
<td></td>
</tr>
<tr>
<td><code>duration_of</code></td>
<td></td>
</tr>
<tr>
<td><code>neighbors_of</code></td>
<td></td>
</tr>
<tr>
<td><code>validate_circuit</code></td>
<td>Raises an exception if a circuit is not valid.</td>
</tr>
<tr>
<td><code>validate_gate</code></td>
<td></td>
</tr>
<tr>
<td><code>validate_moment</code></td>
<td></td>
</tr>
<tr>
<td><code>validate_operation</code></td>
<td></td>
</tr>
<tr>
<td><code>validate_schedule</code></td>
<td></td>
</tr>
<tr>
<td><code>validate_scheduled_operation</code></td>
<td></td>
</tr>
</tbody>
</table>

**cirq.IonDevice.at**

IionDevice.at (position: int) → Optional[cirq.devices.line_qubit.LineQubit]

Returns the qubit at the given position, if there is one, else None.

**cirq.IonDevice.can_add_operation_into_moment**

IionDevice.can_add_operation_into_moment (operation: cirq.ops.raw_types.Operation, moment: cirq.ops.moment.Moment) → bool

Determines if it’s possible to add an operation into a moment.

For example, on the XmonDevice two CZs shouldn’t be placed in the same moment if they are on adjacent qubits.

**Parameters**

- operation – The operation being added.
- moment – The moment being transformed.

**Returns** Whether or not the moment will validate after adding the operation.

**cirq.IonDevice.decompose_circuit**


**cirq.IonDevice.decompose_operation**


Returns a device-valid decomposition for the given operation.
This method is used when adding operations into circuits with a device specified, to avoid spurious failures due to e.g. using a Hadamard gate that must be decomposed into native gates.

```python
cirq.IonDevice.duration_of
```

```python
IonDevice.duration_of(operation)
```

```python
cirq.IonDevice.neighbors_of
```

```python
IonDevice.neighbors_of(qubit: cirq.devices.line_qubit.LineQubit)
Returns the qubits that the given qubit can interact with.
```

```python
cirq.IonDevice.validate_circuit
```

```python
IonDevice.validate_circuit(circuit: cirq.circuits.circuit.Circuit)
Raises an exception if a circuit is not valid.

Parameters:

- `circuit` – The circuit to validate.

Raises: `ValueError` – The circuit isn’t valid for this device.

```python
cirq.IonDevice.validate_gate
```

```python
IonDevice.validate_gate(gate: cirq.ops.raw_types.Gate)
```

```python
cirq.IonDevice.validate_moment
```

```python
IonDevice.validate_moment(moment: cirq.Moment) → None
```

Raises an exception if a moment is not valid.

Parameters:

- `moment` – The moment to validate.

Raises: `ValueError` – The moment isn’t valid for this device.

```python
cirq.IonDevice.validate_operation
```

```python
IonDevice.validate_operation(operation)
```

Raises an exception if an operation is not valid.

Parameters:

- `operation` – The operation to validate.

Raises: `ValueError` – The operation isn’t valid for this device.

```python
cirq.IonDevice.validate_schedule
```

```python
IonDevice.validate_schedule(schedule)
```

Raises an exception if a schedule is not valid.

Parameters:

- `schedule` – The schedule to validate.
**Raises** *ValueError* – The schedule isn’t valid for this device.

**cirq.IonDevice.validate_scheduled_operation**

*IonDevice.validate_scheduled_operation(schedule, scheduled_operation)*

Raises an exception if the scheduled operation is not valid.

**Parameters**

- **schedule** – The schedule to validate against.
- **scheduled_operation** – The scheduled operation to validate.

**Raises** *ValueError* – If the scheduled operation is not valid for the schedule.

**cirq.two_qubit_matrix_to_ion_operations**

* cirq.two_qubit_matrix_to_ion_operations(q0: cirq.ops.raw_types.Qid, q1: cirq.ops.raw_types.Qid, mat: numpy.ndarray, atol: float = 1e-08) → List[cirq.ops.raw_types.Operation]

Decomposes a two-qubit operation into MS/single-qubit rotation gates.

**Parameters**

- **q0** – The first qubit being operated on.
- **q1** – The other qubit being operated on.
- **mat** – Defines the operation to apply to the pair of qubits.
- **tolerance** – A limit on the amount of error introduced by the construction.

**Returns** A list of operations implementing the matrix.

**cirq.ConvertToNeutralAtomGates**

*class cirq.ConvertToNeutralAtomGates(ignore_failures=False)*

Attempts to convert gates into native Atom gates.

First, checks if the given operation is already a native neutral atom operation.

Second, checks if the operation has a known unitary. If so, and the gate is a 1-qubit or 2-qubit gate, then performs circuit synthesis of the operation. The 2-qubit gates are decomposed using CZ gates because CZ gates are the highest fidelity 2-qubit gates for neutral atoms.

Third, attempts to *cirq.decompose* to the operation.

Fourth, if *ignore_failures* is set, gives up and returns the gate unchanged. Otherwise raises a *TypeError*. 
__init__(ignore_failures=False) \rightarrow\) None

**Parameters**

- **ignore_failures** – If set, gates that fail to convert are forwarded unchanged.
  If not set, conversion failures raise a TypeError.

**Methods**

- **convert(op)**
- **optimization_at(circuit, index, op)** Describes how to change operations near the given location.
- **optimize_circuit(circuit)**

**cirq.ConvertToNeutralAtomGates.convert**

`ConvertToNeutralAtomGates.convert(op: cirq.ops.raw_types.Operation) \rightarrow List[cirq.ops.raw_types.Operation]`

**cirq.ConvertToNeutralAtomGates.optimization_at**

`ConvertToNeutralAtomGates.optimization_at(circuit, index, op)` Describes how to change operations near the given location.

For example, this method could realize that the given operation is an X gate and that in the very next moment there is a Z gate. It would indicate that they should be combined into a Y gate by returning

```python
PointOptimizationSummary(clear_span=2,
clear_qubits=op.qubits,
new_operations=cirq.Y(op.qubits[0]))
```

**Parameters**

- **circuit** – The circuit to improve.
- **index** – The index of the moment with the operation to focus on.
- **op** – The operation to focus improvements upon.

**Returns** A description of the optimization to perform, or else None if no change should be made.

**cirq.ConvertToNeutralAtomGates.optimize_circuit**

`ConvertToNeutralAtomGates.optimize_circuit(circuit: cirq.circuits.circuit.Circuit)`

**cirq.NeutralAtomDevice**

```python
```

A device with qubits placed on a grid.

Initializes the description of the AQuA device.

Parameters

- **measurement_duration** – the maximum duration of a measurement.
- **gate_duration** – the maximum duration of a gate
- **control_radius** – the maximum distance between qubits for a controlled gate. Distance is measured in units of the indices passed into the GridQubit constructor.
- **max_parallel_z** – The maximum number of qubits that can be acted on in parallel by a Z gate
- **max_parallel_xy** – The maximum number of qubits that can be acted on in parallel by a local XY gate
- **max_parallel_c** – the maximum number of qubits that can be acted on in parallel by a controlled gate. Must be less than or equal to the lesser of max_parallel_z and max_parallel_xy
- **qubits** – Qubits on the device, identified by their x, y location. Must be of type GridQubit

Raises **ValueError** – if the wrong qubit type is provided or if invalid parallel parameters are provided

Methods

- **can_add_operation_into_moment(operation, moment)**
  Determines if it’s possible to add an operation into a moment. An

- **decompose_operation(operation)**
  Returns a device-valid decomposition for the given operation.

- **distance(p, q)**
  Provides the duration of the given operation on this device.

- **neighbors_of(qubit)**
  Returns the qubits that the given qubit can interact with.

- **qubit_list()**

- **validate_circuit(circuit)**
  Raises an error if the given circuit is invalid on this device. A

- **validate_gate(gate)**
  Raises an error if the provided gate isn’t part of the native gate set.

- **validate_moment(moment)**
  Raises an error if the given moment is invalid on this device

- **validate_operation(operation)**
  Raises an error if the given operation is invalid on this device.

- **validate_schedule(schedule)**
  Raises an error if the given schedule is invalid on this device.

Continued on next page
Table 194 – continued from previous page

validate_scheduled_operation(schedule, ...) Raises an error if the given scheduled_operation is 

isn’t valid in

cirq.NeutralAtomDevice.can_add_operation_into_moment

NeutralAtomDevice.can_add_operation_into_moment (operation: 
cirq.ops.raw_types.Operation, 
moment: 
cirq.ops.moment.Moment) 
→ bool

Determines if it’s possible to add an operation into a moment. An 
operation can be added if the moment with the operation added is valid

Parameters

• operation – The operation being added.
• moment – The moment being transformed.

Returns Whether or not the moment will validate after adding the operation.

Raises ValueError – If either of the given moment or operation is invalid

cirq.NeutralAtomDevice.decompose_operation

NeutralAtomDevice.decompose_operation (operation: 
cirq.ops.raw_types.Operation) 
→ Union[cirq.ops.raw_types.Operation, 
cirq.ops.op_tree.OpTree]

Returns a device-valid decomposition for the given operation.

This method is used when adding operations into circuits with a device 
specified, to avoid spurious failures due to e.g. using a Hadamard gate 
that must be decomposed into native gates.

cirq.NeutralAtomDevice.distance

NeutralAtomDevice.distance (p: cirq.ops.raw_types.Qid, q: cirq.ops.raw_types.Qid) → float

cirq.NeutralAtomDevice.duration_of

NeutralAtomDevice.duration_of (operation: cirq.ops.raw_types.Operation) 
Provides the duration of the given operation on this device.

Parameters operation – the operation to get the duration of

Returns The duration of the given operation on this device

Raises ValueError – If the operation provided doesn’t correspond to a native gate
cirq.NeutralAtomDevice.neighbors_of

NeutralAtomDevice.neighbors_of(qubit: cirq.devices.grid_qubit.GridQubit)
Returns the qubits that the given qubit can interact with.

cirq.NeutralAtomDevice.qubit_list

NeutralAtomDevice.qubit_list()

cirq.NeutralAtomDevice.validate_circuit

NeutralAtomDevice.validate_circuit(circuit: cirq.circuits.circuit.Circuit)
Raisers an error if the given circuit is invalid on this device. A circuit is invalid if any of its moments are invalid or if there is a non-empty moment after a moment with a measurement.

Parameters

circuit – The circuit to validate

Raises

ValueError – If the given circuit can’t be run on this device

cirq.NeutralAtomDevice.validate_gate

NeutralAtomDevice.validate_gate(gate: cirq.ops.raw_types.Gate)
Raisers an error if the provided gate isn’t part of the native gate set.

Parameters

gate – the gate to validate

Raises

ValueError – If the given gate is not part of the native gate set.

cirq.NeutralAtomDevice.validate_moment

NeutralAtomDevice.validate_moment(moment: cirq.ops.moment.Moment)
Raisers an error if the given moment is invalid on this device

Parameters

moment – The moment to validate

Raises

ValueError – If the given moment is invalid

cirq.NeutralAtomDevice.validate_operation

NeutralAtomDevice.validate_operation(operation: cirq.ops.raw_types.Operation)
Raisers an error if the given operation is invalid on this device.

Parameters

operation – the operation to validate

Raises

ValueError – If the operation is not valid
cirq.NeutralAtomDevice.validate_schedule

NeutralAtomDevice.validate_schedule(schedule)
Renames an error if the given schedule is invalid on this device.

Parameters schedule – The schedule to validate

Raises ValueError – If the schedule is invalid

cirq.NeutralAtomDevice.validate_scheduled_operation

NeutralAtomDevice.validate_scheduled_operation(schedule, scheduled_operation)

Raises an error if the given scheduled_operation is isn’t valid in the
device. Also raises an error if the operations that overlap with the
given operation would form an invalid moment on the device.

Parameters

• schedule – The schedule the scheduled operation is part of
• scheduled_operation – The operation to validate

Raises ValueError – If the scheduled operation is invalid in the schedule

3.1.19 Google

Functionality specific to quantum hardware and services from Google.

| google.AnnealSequenceSearchStrategy([...]|Linearized sequence search using simulated annealing method. |
| google.Bristlecone | |
| google.ConvertToXmonGates([ignore_failures]) | Attempts to convert strange gates into XmonGates. |
| google.Engine([project_id[, version,...]]) | Runs programs via the Quantum Engine API. |
| google.engine_from_environment() | Returns an Engine instance configured using environment variables. |
| google.Foxtail | |
| google.fox_to_proto_dict(gate, qubits) | |
| google.is_native_xmon_op(op) | Check if the gate corresponding to an operation is a native xmon gate. |
| google.JobConfig([job_id, gcs_prefix,...]) | Configuration for a job to run on the Quantum Engine API. |
| google.line_on_device(device, length[,method]) | Searches for linear sequence of qubits on device. |
| google.LinePlacementStrategy | Choice and options for the line placement calculation method. |
| google.optimized_for_xmon(circuit[,...]) | Optimizes a circuit with XmonDevice in mind. |
| google.pack_results(measurements) | Pack measurement results into a byte string. |
| google.schedule_from_proto_dicts(device, ops) | Convert proto dictionaries into a Schedule for the given device. |

Continued on next page
Table 195 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>google.schedule_to_proto_dicts(schedule)</code></td>
<td>Convert a schedule into an iterable of proto dictionaries.</td>
</tr>
<tr>
<td><code>google.unpack_results(data, repetitions, ...)</code></td>
<td>Unpack data from a bitstring into individual measurement results.</td>
</tr>
<tr>
<td><code>google.xmon_op_from_proto_dict(proto_dict)</code></td>
<td>Convert the proto dictionary to the corresponding operation.</td>
</tr>
</tbody>
</table>

```python

cirq.google.AnnealSequenceSearchStrategy
class cirq.google.AnnealSequenceSearchStrategy:
    Linearized sequence search using simulated annealing method.

    TODO: This line search strategy is still work in progress and requires efficiency improvements.

    __init__(trace_func: Callable[[List[List[cirq.devices.grid_qubit.GridQubit]], float, float, float, bool], None] = None, seed: int = None) → None
    Linearized sequence search using simulated annealing method.

    Parameters
    • `trace_func` – Optional callable which will be called for each simulated annealing step with arguments: solution candidate (list of linear sequences on the chip), current temperature (float), candidate cost (float), probability of accepting candidate (float), and acceptance decision (boolean).
    • `seed` – Optional seed value for random number generator.

    Returns List of linear sequences on the chip found by simulated annealing method.

Methods

place_line(device, length) Runs line sequence search.
```

```python
cirq.google.AnnealSequenceSearchStrategy.place_line

AnnealSequenceSearchStrategy.place_line(device: cirq.google.XmonDevice, length: int) → cirq.google.line.placement.sequence.GridQubitLineTuple

Runs line sequence search.

Parameters
• `device` – Chip description.
• `length` – Required line length.

Returns List of linear sequences on the chip found by simulated annealing method.
```
cirq.google.Bristlecone

cirq.google.Bristlecone = cirq.google.Bristlecone

cirq.google.ConvertToXmonGates

class cirq.google.ConvertToXmonGates(ignore_failures=False)
    Attempts to convert strange gates into XmonGates.

    First, checks if the given operation is already a native xmon operation.

    Second, checks if the operation has a known unitary. If so, and the gate
    is a 1-qubit or 2-qubit gate, then performs circuit synthesis of the
    operation.

    Third, attempts to cirq.decompose to the operation.

    Fourth, if ignore_failures is set, gives up and returns the gate unchanged.
    Otherwise raises a TypeError.

    __init__(ignore_failures=False) → None
    Parameters ignore_failures – If set, gates that fail to convert are forwarded unchanged.
    If not set, conversion failures raise a TypeError.

    Methods

    convert(op)
    optimization_at(circuit, index, op)  # Describes how to change operations near the given location.
    optimize_circuit(circuit)

    cirq.google.ConvertToXmonGates.convert

    ConvertToXmonGates.convert(op: cirq.Operation) → List[cirq.Operation]

    cirq.google.ConvertToXmonGates.optimization_at

    ConvertToXmonGates.optimization_at(circuit, index, op)
    # Describes how to change operations near the given location.

    For example, this method could realize that the given operation is an
    X gate and that in the very next moment there is a Z gate. It would
    indicate that they should be combined into a Y gate by returning
    PointOptimizationSummary(clear_span=2,
clear_qubits=op.qubits,
new_operations=cirq.Y(op.qubits[0]))

**Parameters**

- **circuit** – The circuit to improve.
- **index** – The index of the moment with the operation to focus on.
- **op** – The operation to focus improvements upon.

**Returns**  
A description of the optimization to perform, or else None if no change should be made.

cirq.google.ConvertToXmonGates.optimize_circuit

ConvertToXmonGates.optimize_circuit(circuit: cirq.circuits.circuit.Circuit)

cirq.google.Engine


Runs programs via the Quantum Engine API.

This class has methods for creating programs and jobs that execute on Quantum Engine:
create_program
run
run_sweep

Another set of methods return information about programs and jobs that have been previously created on the Quantum Engine, as well as metadata about available processors:
get_calibration
get_job
get_job_results
get_latest_calibration
get_program
list_processors

Finally, the engine has methods to update existing programs and jobs:
add_job_labels
add_program_labels
cancel_job
remove_job_labels
remove_program_labels
set_job_labels
set_program_labels

```python
```

Engine service client.

**Parameters**

- **project_id** – A project_id string of the Google Cloud Project to use. API interactions will be attributed to this project and any resources created will be owned by the project. See https://cloud.google.com/resource-manager/docs/creating-managing-projects#identifying_projects
- **version** – API version.
- **discovery_url** – Discovery url for the API to select a non-default backend for the Engine. Incompatible with `version` argument.
- **default_gcs_prefix** – A fallback gcs_prefix to use when one isn’t specified in the JobConfig given to ‘run’ methods. See JobConfig for more information on gcs_prefix.
- **service_args** – A dictionary of arguments that can be used to configure options on the underlying apiclient. See https://github.com/googleapis/google-api-python-client
- **verbose** – Supresses stderr messages when set to False. Default is true.

**Methods**

```python
add_job_labels(job_resource_name, labels)
add_program_labels(program_id, labels)
cancel_job(job_resource_name) Cancels the given job.
create_job(*program_name[, job_config,…])
create_program(program[, program_id, gate_set])
gate_calibration(calibration_name)
gate_job(job_resource_name)
gate_job_results(job_resource_name)
gate_latest_calibration(processor_id)
gate_program(program_id)
implied_job_config(job_config)
list_processors()
create_job
```

Returns a list of Processors that the user has visibility to in the

```python
program_as_schedule(program)
remove_job_labels(job_resource_name, label_keys)
remove_program_labels(program_id, label_keys)
```

Continued on next page
Table 198 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>run(*, program[, program_id, job_config, ...])</code></td>
<td>Runs the supplied Circuit or Schedule via Quantum Engine.</td>
</tr>
<tr>
<td><code>run_sweep(*, program[, program_id, ...])</code></td>
<td>Runs the supplied Circuit or Schedule via Quantum Engine.</td>
</tr>
<tr>
<td><code>sampler(processor_id, gate_set)</code></td>
<td>Returns a sampler backed by the engine.</td>
</tr>
<tr>
<td><code>set_job_labels(job_resource_name, labels)</code></td>
<td></td>
</tr>
<tr>
<td><code>set_program_labels(program_id, labels)</code></td>
<td></td>
</tr>
</tbody>
</table>

`cirq.google.Engine.add_job_labels`

`Engine.add_job_labels` (*job_resource_name: str, labels: Dict[str, str]*)

`cirq.google.Engine.add_program_labels`

`Engine.add_program_labels` (*program_id: str, labels: Dict[str, str]*)

`cirq.google.Engine.cancel_job`

`Engine.cancel_job` (*job_resource_name: str*)

Cancels the given job.

See also the cancel method on EngineJob.

Params:

- `job_resource_name`: A string of the form `projects/project_id/programs/program_id/jobs/job_id`.

`cirq.google.Engine.create_job`


`cirq.google.Engine.create_program`


Wraps a Circuit or Scheduler for use with the Quantum Engine.

**Parameters**
• **program** – The Circuit or Schedule to execute. If a circuit is provided, a moment by moment schedule will be used.

• **program_id** – A user-provided identifier for the program. This must be unique within the Google Cloud project being used. If this parameter is not provided, a random id of the format ‘prog-######’ will be generated.

• **gate_set** – The gate set used to serialize the circuit. The gate set must be supported by the selected processor

**cirq.google.Engine.get_calibration**

Engine.get_calibration(calibration_name: str) → cirq.google.engine.calibration.Calibration

Retrieve metadata about a specific calibration run.

**Params:**
calibration_name: A string of the form
<processor name>/calibrations/<ms since epoch>

**Returns** A dictionary containing the metadata.

**cirq.google.Engine.get_job**

Engine.get_job(job_resource_name: str) → Dict

Returns metadata about a previously created job.

See get_job_result if you want the results of the job and not just metadata about the job.

**Params:**
job_resource_name: A string of the form
projects/project_id/programs/program_id/jobs/job_id.

**Returns** A dictionary containing the metadata.

**cirq.google.Engine.get_job_results**

Engine.get_job_results(job_resource_name: str) → List[cirq.study.trial_result.TrialResult]

Returns the actual results (not metadata) of a completed job.

**Params:**
job_resource_name: A string of the form
projects/project_id/programs/program_id/jobs/job_id.

**Returns** An iterable over the TrialResult, one per parameter in the parameter sweep.
cirq.google.Engine.get_latest_calibration


Returns metadata about the latest known calibration for a processor.

Params:
processor_id: The processor identifier within the resource name, where name has the format:
projects/<project_id>/processors/<processor_id>.

Returns A dictionary containing the calibration data or None if there are no calibrations.

cirq.google.Engine.get_program

Engine.get_program(program_id: str) → Dict

Returns the previously created quantum program.

Params:
program_id: A string containing the unique ID of a program within the project specified for the Engine.

Returns A dictionary containing the metadata and the program.

cirq.google.Engine.implied_job_config


cirq.google.Engine.list_processors

Engine.list_processors() → List[Dict]

Returns a list of Processors that the user has visibility to in the current Engine project. The names of these processors are used to identify devices when scheduling jobs and gathering calibration metrics.

Returns A list of dictionaries containing the metadata of each processor.

cirq.google.Engine.program_as_schedule

**cirq.google.Engine.remove_job_labels**

Engine.remove_job_labels(job_resource_name: str, label_keys: List[str])

**cirq.google.Engine.remove_program_labels**

Engine.remove_program_labels(program_id: str, label_keys: List[str])

**cirq.google.Engine.run**


Runs the supplied Circuit or Schedule via Quantum Engine.

**Parameters**

- **program** – The Circuit or Schedule to execute. If a circuit is provided, a moment by moment schedule will be used.

- **program_id** – A user-provided identifier for the program. This must be unique within the Google Cloud project being used. If this parameter is not provided, a random id of the format ‘prog-######’ will be generated.

- **job_config** – Configures the names and properties of jobs.

- **param_resolver** – Parameters to run with the program.

- **repetitions** – The number of repetitions to simulate.

- **priority** – The priority to run at, 0-100.

- **processor_ids** – The engine processors that should be candidates to run the program. Only one of these will be scheduled for execution.

- **gate_set** – The gate set used to serialize the circuit. The gate set must be supported by the selected processor.

**Returns** A single TrialResult for this run.

**cirq.google.Engine.run_sweep**


Runs the supplied Circuit or Schedule via Quantum Engine.
In contrast to run, this runs across multiple parameter sweeps, and does not block until a result is returned.

**Parameters**

- **program** – The Circuit or Schedule to execute. If a circuit is provided, a moment by moment schedule will be used.
- **program_id** – A user-provided identifier for the program. This must be unique within the Google Cloud project being used. If this parameter is not provided, a random id of the format ‘prog-#####’ will be generated.
- **job_config** – Configures the names and properties of jobs.
- **params** – Parameters to run with the program.
- **repetitions** – The number of circuit repetitions to run.
- **priority** – The priority to run at, 0-100.
- **processor_ids** – The engine processors that should be candidates to run the program. Only one of these will be scheduled for execution.

**Returns** An EngineJob. If this is iterated over it returns a list of TrialResults, one for each parameter sweep.

cirq.google.Engine.sampler

```python
```

Returns a sampler backed by the engine.

**Parameters**

- **processor_id** – String identifier, or list of string identifiers, determining which processors may be used when sampling.
- **gate_set** – Determines how to serialize circuits when requesting samples.

cirq.google.Engine.set_job_labels

```python
Engine.set_job_labels(job_resource_name: str, labels: Dict[str, str])
```

cirq.google.Engine.set_program_labels

```python
Engine.set_program_labels(program_id: str, labels: Dict[str, str])
```

cirq.google.engine_from_environment

```python
cirq.google.engine_from_environment() → cirq.google.engine.engine.Engine
```

Returns an Engine instance configured using environment variables.

If the environment variables are set, but incorrect, an authentication
failure will occur when attempting to run jobs on the engine.

Required Environment Variables:
QUANTUM_ENGINE_PROJECT: The name of a google cloud project, with the quantum engine enabled, that you have access to.

Raises EnvironmentError – The environment variables are not set.

cirq.google.Foxtail
cirq.google.Foxtail = cirq.google.Foxtail
cirq.google.gate_to_proto_dict
cirq.google.gate_to_proto_dict(gate: cirq.Gate, qubits: Tuple[cirq.Qid, ...]) → Dict
cirq.google.GreedySequenceSearchStrategy
class cirq.google.GreedySequenceSearchStrategy(algorithm: str = 'best')
Greedy search method for linear sequence of qubits on a chip.
__init__(algorithm: str = 'best') → None
Initializes greedy sequence search strategy.

Parameters
• algorithm – Greedy algorithm to be used. Available options are:
  • – runs all heuristics and chooses the best result, (best)–
  • – on every step takes the qubit which has connection (largest_area)–
  • the largest number of unassigned qubits, and (with)–
  • – on every step takes the qubit with minimal (minimal_connectivity)–
  • of unassigned neighbouring qubits. (number)–

Methods

place_line(device, length) Runs line sequence search.

cirq.google.GreedySequenceSearchStrategy.place_line

GreedySequenceSearchStrategy.place_line(device: cirq.google.XmonDevice,
length: int) →
cirq.google.line.placement.sequence.GridQubitLineTuple

Runs line sequence search.

Parameters

• **device** – Chip description.
• **length** – Required line length.

**Returns** Linear sequences found on the chip.

**Raises** `ValueError` – If search algorithm passed on initialization is not recognized.

cirq.google.is_native_xmon_op

cirq.google.is_native_xmon_op(op: cirq.Operation) → bool

Check if the gate corresponding to an operation is a native xmon gate.

**Parameters**
- `op` – Input operation.

**Returns** True if the operation is native to the xmon, false otherwise.

cirq.google.JobConfig


Configuration for a job to run on the Quantum Engine API.

An instance of a program that has been scheduled on the Quantum Engine is called a Job. This object contains the configuration for a job.

__init__(job_id: Optional[str] = None, gcs_prefix: Optional[str] = None, gcs_results: Optional[str] = None) → None

Configuration for a job that is run on Quantum Engine.

**Parameters**
- `job_id` – Id of the job to create, defaults to ‘job-0’.
- `gcs_prefix` – Google Cloud Storage bucket and object prefix to use for storing programs and results. The bucket will be created if needed. Must be in the form “gs://bucket-name/object-prefix/”.
- `gcs_results` – Explicit override for the results storage location.

**Methods**

copy()

cirq.google.JobConfig.copy

JobConfig.copy() → cirq.google.engine.engine.JobConfig
**cirq.google.line_on_device**

```
```

Searches for linear sequence of qubits on device.

**Parameters**

- **device** – Google Xmon device instance.
- **length** – Desired number of qubits making up the line.

**Returns** Line sequences search results.

---

**cirq.google.LinePlacementStrategy**

```python
class cirq.google.LinePlacementStrategy
```

Choice and options for the line placement calculation method.

Currently two methods are available: cirq.line.GreedySequenceSearchMethod and cirq.line.AnnealSequenceSearchMethod.

```python
__init__()

Initialize self. See help(type(self)) for accurate signature.
```

**Methods**

```python
cirq.google.LinePlacementStrategy.place_line(device, length)
```

Runs line sequence search.

**Parameters**

- **device** – Chip description.
- **length** – Required line length.

**Returns** Linear sequences found on the chip.

---

**cirq.google.pack_results**

```
cirq.google.pack_results(measurements: Sequence[Tuple[str, numpy.ndarray]]) → bytes
```

Pack measurement results into a byte string.
Parameters measurements – A sequence of tuples, one for each measurement, consisting of a string key and an array of boolean data. The data should be a 2-D array indexed by (repetition, qubit_index). All data for all measurements must have the same number of repetitions.

Returns Packed bytes, as described in the unpack_results docstring below.

Raises ValueError if the measurement data do not have the compatible shapes.

---

cirq.google.schedule_from_proto_dicts
cirq.google.schedule_from_proto_dicts

cirq.google.schedule_from_proto_dicts(device: cirq.google.XmonDevice, ops: Iterable[Dict]) → cirq.schedules.schedule.Schedule

Convert proto dictionaries into a Schedule for the given device.

---

cirq.google.schedule_to_proto_dicts
cirq.google.schedule_to_proto_dicts

cirq.google.schedule_to_proto_dicts(schedule: cirq.schedules.schedule.Schedule) → Iterable[Dict]

Convert a schedule into an iterable of proto dictionaries.

Parameters schedule – The schedule to convert to a proto dict. Must contain only gates that can be cast to xmon gates.

Yields A proto dictionary corresponding to an Operation proto.

---

cirq.google.unpack_results
cirq.google.unpack_results

cirq.google.unpack_results(data: bytes, repetitions: int, key_sizes: Sequence[Tuple[str, int]]) → Dict[str, numpy.ndarray]

Unpack data from a bitstring into individual measurement results.

Parameters

• data – Packed measurement results, in the form <rep0><rep1>... where each repetition is <key0_0><key0_{size0-1}><key1_0><key1_{size1-1}>... with bits packed in little-endian order in each byte.

• repetitions – number of repetitions.

• key_sizes – Keys and sizes of the measurements in the data.

Returns Dict mapping measurement key to a 2D array of boolean results. Each array has shape (repetitions, size) with size for that measurement.

---

cirq.google.xmon_op_from_proto_dict
cirq.google.xmon_op_from_proto_dict

cirq.google.xmon_op_from_proto_dict(proto_dict: Dict) → cirq.Operation

Convert the proto dictionary to the corresponding operation.

See protos in api/google/v1 for specification of the protos.

Parameters proto_dict – Dictionary representing the proto. Keys are always strings, but values may be types correspond to a raw proto type or another dictionary (for messages).

Returns The operation.

Raises
3.1.20 Testing

Functionality for writing unit tests involving objects from Cirq, and also some general testing utilities.

- `testing.assert_allclose_up_to_global_phase(a, b)` Checks if $a \approx b \exp(i t)$ for some $t$.
- `testing.assert_circuits_with_terminal_measurements_are_equivalent(circuit1, circuit2)` Determines if two circuits have equivalent effects.
- `testing.assert_decompose_is_consistent_with_unitary(val)` Uses `val._unitary_` to check `val._phase_by_`'s behavior.
- `testing.assert_eigengate_implements_consistent_protocols(gate)` Checks that an EigenGate subclass is internally consistent and has a good `repr`.
- `testing.assert_pauli_expansion_is_consistent_with_unitary(val)` Checks Pauli expansion against unitary matrix.
- `testing.assert_phase_by_is_consistent_with_unitary(val)` Uses `val._unitary_` to check `val._phase_by_`'s behavior.
- `testing.assert_qasm_is_consistent_with_unitary(val)` Uses `val._unitary_` to check `val._qasm_`'s behavior.
- `testing.assert_same_circuits(actual_circuit, expected_circuit)` Asserts that two circuits are identical, with a descriptive error.
- `testing.EqualsTester()` Tests equality against user-provided disjoint equivalence groups.
- `testing.nonoptimal_toffoli_circuit(q0, q1)` Tests ordering against user-provided disjoint ordered groups or items.
- `testing.random_circuit(qubits, n_moments)` Generates a random circuit.
- `testing.random_orthogonal(dim)` Returns a random orthogonal matrix distributed with Haar measure.
- `testing.random_special_orthogonal(dim)` Returns a random special orthogonal matrix distributed with Haar measure.
- `testing.random_special_unitary(dim, *[, qubits])` Returns a random special unitary distributed with Haar measure.
- `testing.random_superposition(dim)` Returns a random unit-length vector from the uniform distribution.
Table 202 – continued from previous page

| **testing.random_unitary** *(dim, random_state)* | Returns a random unitary matrix distributed with Haar measure. |

---

cirq.testing.assert_allclose_up_to_global_phase


Checks if \( a \approx b \exp(i t) \) for some \( t \).

**Parameters**

- **actual** – A numpy array.
- **desired** – Another numpy array.
- **rtol** – Relative error tolerance.
- **atol** – Absolute error tolerance.
- **equal_nan** – Whether or not NaN entries should be considered equal to other NaN entries.
- **err_msg** – The error message to be printed in case of failure.
- **verbose** – If True, the conflicting values are appended to the error message.

**Raises** [AssertionError](https://docs.python.org/3/library/exceptions.html#AssertionError) – The matrices aren’t nearly equal up to global phase.

---

cirq.testing.assert_circuits_with_terminal_measurements_are_equivalent


Determined if two circuits have equivalent effects.

The circuits can contain measurements, but the measurements must be at the end of the circuit. Circuits are equivalent if, for all possible inputs, their outputs (classical bits for lines terminated with measurement and qubits for lines without measurement) are observationally indistinguishable up to a tolerance. Note that under this definition of equivalence circuits that differ solely in the overall phase of the post-measurement state of measured qubits are considered equivalent.

For example, applying an extra Z gate to an unmeasured qubit changes the
effect of a circuit. But inserting a Z gate operation just before a measurement does not.

### Parameters

- **actual** – The circuit that was actually computed by some process.
- **reference** – A circuit with the correct function.
- **atol** – Absolute error tolerance.

#### `cirq.testing.assert_decompose_is_consistent_with_unitary`

```python
assert_decompose_is_consistent_with_unitary(val: Any, ignoring_global_phase: bool = False)
```

Uses `val._unitary_` to check `val._phase_by_`’s behavior.

#### `cirq.testing.assert_eigengate_implements_consistent_protocols`

```python
assert_eigengate_implements_consistent_protocols(eigen_gate_type: Type[cirq.ops.eigen_gate.EigenGate], *, exponents: Sequence[Union[float, sympy.core.basic.Basic]] = (0, 1, -1, 0.25, -0.5, 0.1, s), global_shifts: Sequence[float] = (0, -0.5, 0.1), qubit_count: Optional[int] = None, ignoring_global_phase: bool = False, setup_code: str = 'import cirq
import numpy as np
import sympy',
```

Checks that an EigenGate subclass is internally consistent and has a good `repr`.

---

3.1. API Reference 421
cirq.testing.assert_equivalent_repr

Checks that eval(repr(v)) == v.

Parameters

- **value** – A value whose repr should be evaluable python code that produces an equivalent value.
- **setup_code** – Code that must be executed before the repr can be evaluated. Ideally this should just be a series of 'import' lines.
- **global_vals** – Pre-defined values that should be in the global scope when evaluating the repr.
- **local_vals** – Pre-defined values that should be in the local scope when evaluating the repr.

cirq.testing.assert_has_consistent_apply_unitary

Tests whether a value’s apply_unitary is correct.

Contrasts the effects of the value’s apply_unitary with the matrix returned by the value’s unitary method.

Parameters

- **val** – The value under test. Should have a __pow__ method.
- **atol** – Absolute error tolerance.

cirq.testing.assert_has_consistent_apply_unitary_for_various_exponents

Tests whether a value’s apply_unitary is correct.
Contrasts the effects of the value’s \_apply\_unitary\_ with the matrix returned by the value’s \_unitary\_ method. Attempts this after attempting to raise the value to several exponents.

**Parameters**

- **val** – The value under test. Should have a \_pow\_ method.
- **exponents** – The exponents to try. Defaults to a variety of special and arbitrary angles, as well as a parameterized angle (a symbol). If the value’s \_pow\_ returns \_NotImplemented\_ for any of these, they are skipped.

### Cirq.testing.assert\_has\_diagram

```python
cirq.testing.assert_has_diagram(actual: cirq.circuits.circuit.Circuit, desired: str, **kwargs) -> None
```

Determines if a given circuit has the desired text diagram.

**Parameters**

- **actual** – The circuit that was actually computed by some process.
- **desired** – The desired text diagram as a string. Newlines at the beginning and whitespace at the end are ignored.
- ****\_\_kwarg\_s** – Keyword arguments to be passed to actual.to\_text\_diagram().

### Cirq.testing.assert\_implements\_consistent\_protocols

```python
cirq.testing.assert_implements_consistent_protocols(val: Any, *, exponents: Sequence[Any] = (0, 1, -1, 0.5, 0.25, -0.5, 0.1, s), qubit\_count: Optional[int] = None, ignoring\_global\_phase: bool = False, setup\_code: str = 'import cirq
import numpy as np
import sympy', global\_vals: Optional[Dict[str, Any]] = None, local\_vals: Optional[Dict[str, Any]] = None) -> None
```

Checks that a value is internally consistent and has a good \_repr\_.

### Cirq.testing.assert\_pauli\_expansion\_is\_consistent\_with\_unitary

```python
cirq.testing.assert_pauli_expansion_is_consistent_with_unitary(val: Any) -> None
```

Checks Pauli expansion against unitary matrix.

### Cirq.testing.assert\_phase\_by\_is\_consistent\_with\_unitary

```python
cirq.testing.assert_phase_by_is_consistent_with_unitary(val: Any)
```

Uses \_phase\_by\_'s behavior to check \_phase\_by\_'s behavior.
cirq.testing.assert_qasm_is_consistent_with_unitary

cirq.testing.assert_qasm_is_consistent_with_unitary(val: Any)
Uses val._unitary_ to check val._qasm_'s behavior.

cirq.testing.assert_same_circuits

cirq.testing.assert_same_circuits(actual: cirq.circuits.circuit.Circuit, expected: cirq.circuits.circuit.Circuit) → None
Asserts that two circuits are identical, with a descriptive error.

Parameters

- **actual** – A circuit computed by some code under test.
- **expected** – The circuit that should have been computed.

cirq.testing.EqualsTester

class cirq.testing.EqualsTester
Tests equality against user-provided disjoint equivalence groups.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*<em>add_equality_group(<em>group_items)</em></em></td>
<td>Tries to add a disjoint equivalence group to the equality tester.</td>
</tr>
<tr>
<td>*<em>make_equality_group(<em>factories)</em></em></td>
<td>Tries to add a disjoint equivalence group to the equality tester.</td>
</tr>
</tbody>
</table>

cirq.testing.EqualsTester.add_equality_group

EqualsTester.add_equality_group(*group_items: Any)
Tries to add a disjoint equivalence group to the equality tester.

This method asserts that items within the group must all be equal to each other, but not equal to any items in other groups that have been or will be added.

Parameters **group_items** – The items making up the equivalence group.

Raises AssertionError – Items within the group are not equal to each other, or items in another group are equal to items within the new group, or the items violate the equals-implies-same-hash rule.
cirq.testing.EqualsTester.make_equality_group

EqualsTester.make_equality_group(*factories: Callable[\[Any\]])
Tries to add a disjoint equivalence group to the equality tester.

Uses the factory methods to produce two different objects with the same initialization for each factory. Asserts that the objects are equal, but not equal to any items in other groups that have been or will be added. Adds the objects as a group.

**Parameters factories** – Methods for producing independent copies of an item.

**Raises AssertionError** – The factories produce items not equal to the others, or items in another group are equal to items from the factory, or the items violate the equal-implies-same-hash rule.

cirq.testing.highlight_text_differences

cirq.testing.highlight_text_differences(actual: str, expected: str) → str

cirq.testing.nonoptimal_toffoli_circuit

**cirq.testing.nonoptimal_toffoli_circuit(q0: cirq.ops.raw_types.Qid, q1: cirq.ops.raw_types.Qid, q2: cirq.ops.raw_types.Qid, device: cirq.devices.device.Device = cirq.UNCONSTRAINED_DEVICE)** → cirq.circuits.circuit.Circuit

cirq.testing.OrderTester

class cirq.testing.OrderTester
Tests ordering against user-provided disjoint ordered groups or items.

**__init__()**
Initialize self. See help(type(self)) for accurate signature.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_ascending(*items)</td>
<td>Tries to add a sequence of ascending items to the order tester.</td>
</tr>
<tr>
<td>add_ascending_equivalence_group(*group_items)</td>
<td>To add an ascending equivalence group to the order tester.</td>
</tr>
</tbody>
</table>

**cirq.testing.OrderTester.add_ascending**

**OrderTester.add_ascending(*items: Any)**
Tries to add a sequence of ascending items to the order tester.
This method asserts that items must all be ascending with regard to both each other and the elements which have already been added during previous calls. Some of the previously added elements might be equivalence groups, which are supposed to be equal to each other within that group.

**Parameters**

- *items* – The sequence of strictly ascending items.

**Raises**

- AssertionError – Items are not ascending either with regard to each other, or with regard to the elements which have been added before.

### `cirq.testing.OrderTester.add_ascending_equivalence_group`

`OrderTester.add_ascending_equivalence_group(*group_items: Any)`

Tries to add an ascending equivalence group to the order tester. Asserts that the group items are equal to each other, but strictly ascending with regard to the already added groups.

**Parameters**

- *group_items* – items making the equivalence group

**Raises**

- AssertionError – The group elements aren’t equal to each other, or items in another group overlap with the new group.

### `cirq.testing.random_circuit`

`cirq.testing.random_circuit(qubits: Union[Sequence[cirq.ops.raw_types.Qid], int], n_moments: int, op_density: float, gate_domain: Optional[Dict[cirq.ops.raw_types.Gate, int]] = None, random_state: Union[numpy.random.mtrand.RandomState, int, None] = None) → cirq.circuits.circuit.Circuit`

Generates a random circuit.

**Parameters**

- *qubits* – If a sequence of qubits, then these are the qubits that the circuit should act on. Because the qubits on which an operation acts are chosen randomly, not all given qubits may be acted upon. If an int, then this number of qubits will be automatically generated.

- *n_moments* – the number of moments in the generated circuit.

- *op_density* – the expected proportion of qubits that are acted on in any moment.

- *gate_domain* – The set of gates to choose from, with a specified arity.

- *random_state* – Random state or random state seed.

**Raises**

- ValueError –
  - *op_density* is not in (0, 1).
  - *gate_domain* is empty.
  - *qubits* is an int less than 1 or an empty sequence.

**Returns**

The randomly generated Circuit.
cirq.testing.random_orthogonal

cirq.testing.random_orthogonal(dim: int) → numpy.ndarray
Returns a random orthogonal matrix distributed with Haar measure.

Parameters
- **dim** – The width and height of the matrix.

Returns The sampled orthogonal matrix.

References


cirq.testing.random_special_orthogonal

cirq.testing.random_special_orthogonal(dim: int) → numpy.ndarray
Returns a random special orthogonal matrix distributed with Haar measure.

Parameters
- **dim** – The width and height of the matrix.

Returns The sampled special orthogonal matrix.

cirq.testing.random_special_unitary

cirq.testing.random_special_unitary(dim: int, *, random_state: Optional[numpy.random.mtrand.RandomState] = None) → numpy.ndarray
Returns a random special unitary distributed with Haar measure.

Parameters
- **dim** – The width and height of the matrix.
- **random_state** – A seed to use for random number generation.

Returns The sampled special unitary.

cirq.testing.random_superposition

cirq.testing.random_superposition(dim: int) → numpy.ndarray
Returns a random unit-length vector from the uniform distribution.

Parameters
- **dim** – The dimension of the vector.

Returns The sampled unit-length vector.

cirq.testing.random_unitary

cirq.testing.random_unitary(dim: int, *, random_state: Union[numpy.random.mtrand.RandomState, int, None] = None) → numpy.ndarray
Returns a random unitary matrix distributed with Haar measure.

Parameters
- **dim** – The width and height of the matrix.
- **random_state** – A seed to use for random number generation.
Returns The sampled unitary matrix.

References


3.1.21 Contrib

Contributed code that requires extra dependencies to be installed, code that may be unstable, and code that may or may not be a fit for the main library. A waiting area.

<table>
<thead>
<tr>
<th>contrib.acquaintance</th>
<th>Tools for creating and using acquaintance strategies.</th>
</tr>
</thead>
<tbody>
<tr>
<td>contrib.paulistring</td>
<td>Methods related to optimizing and transforming PauliStrings.</td>
</tr>
<tr>
<td>contrib.qcircuit</td>
<td>Converts cirq circuits into latex using qcircuit.</td>
</tr>
<tr>
<td>contrib.quirk</td>
<td>Converts cirq circuits into quirk circuits.</td>
</tr>
</tbody>
</table>

**cirq.contrib.acquaintance**

Tools for creating and using acquaintance strategies.

**cirq.contrib.paulistring**

Methods related to optimizing and transforming PauliStrings.

**cirq.contrib.qcircuit**

Converts cirq circuits into latex using qcircuit.

**cirq.contrib.quirk**

Converts cirq circuits into quirk circuits.
C

cirq.contrib.acquaintance, 428
cirq.contrib.paulistring, 428
cirq.contrib.qcircuit, 428
cirq.contrib.quirk, 428
**Symbols**

 initiating() (cirq.AmplitudeDampingChannel method), 135
 initiating() (cirq.ApplyChannelArgs method), 344
 initiating() (cirq.ApplyUnitaryArgs method), 346
 initiating() (cirq.AsymmetricDepolarizingChannel method), 137
 initiating() (cirq.BitFlipChannel method), 141
 initiating() (cirq.CCXPowGate method), 123
 initiating() (cirq.CCZPowGate method), 126
 initiating() (cirq.CNotPowGate method), 100
 initiating() (cirq.CSwapGate method), 128
 initiating() (cirq.CZPowGate method), 103
 initiating() (cirq.Circuit method), 201
 initiating() (cirq.CircuitDag method), 218
 initiating() (cirq.CircuitDiagramInfo method), 347
 initiating() (cirq.CircuitDiagramInfoArgs method), 349
 initiating() (cirq.ControlledGate method), 158
 initiating() (cirq.ControlledOperation method), 160
 initiating() (cirq.ConvertToCzAndSingleGates method), 358
 initiating() (cirq.ConvertToIonGates method), 396
 initiating() (cirq.ConvertToNeutralAtomGates method), 401
 initiating() (cirq.DensityMatrixSimulator method), 267
 initiating() (cirq.DensityMatrixSimulatorState method), 273
 initiating() (cirq.DensityMatrixStepResult method), 273
 initiating() (cirq.DensityMatrixTrialResult method), 277
 initiating() (cirq.DepolarizingChannel method), 143
 initiating() (cirq.Device method), 63
 initiating() (cirq.DropEmptyMoments method), 359
 initiating() (cirq.DropNegligible method), 360
 initiating() (cirq.Duration method), 375
 initiating() (cirq.EigenGate method), 162
 initiating() (cirq.EjectPhasedPaulis method), 360
 initiating() (cirq.EjectZ method), 361
 initiating() (cirq.ExpandComposite method), 361
 initiating() (cirq.Gate method), 165
 initiating() (cirq.GateOperation method), 167
 initiating() (cirq.GeneralizedAmplitudeDampingChannel method), 147
 initiating() (cirq.GridQubit method), 65
 initiating() (cirq.HPowGate method), 74
 initiating() (cirq.ISwapPowGate method), 107
 initiating() (cirq.IdentityGate method), 131
 initiating() (cirq.InsertStrategy method), 251
 initiating() (cirq.InterchangeableQubitsGate method), 168
 initiating() (cirq.IonDevice method), 397
 initiating() (cirq.KakDecomposition method), 381
 initiating() (cirq.LineQubit method), 68
 initiating() (cirq.LinearCombinationOfGates method), 169
 initiating() (cirq.LinearDict method), 383
 initiating() (cirq.Linspace method), 322
 initiating() (cirq.MeasurementGate method), 77
 initiating() (cirq.MergeInteractions method), 363
 initiating() (cirq.MergeSingleQubitGates method), 364
 initiating() (cirq.Moment method), 252
 initiating() (cirq.NamedQubit method), 70
 initiating() (cirq.NeutralAtomDevice method), 401
 initiating() (cirq.Operation method), 172
 initiating() (cirq.ParallelGateOperation method), 255
 initiating() (cirq.ParamResolver method), 323
 initiating() (cirq.Pauli method), 179
 initiating() (cirq.PauliInteractionGate method), 181
 initiating() (cirq.PauliString method), 184
 initiating() (cirq.PauliTransform method), 192
 initiating() (cirq.PeriodicValue method), 387
 initiating() (cirq.PhaseDampingChannel method), 153
 initiating() (cirq.PhaseFlipChannel method), 156
 initiating() (cirq.PhasedXPowGate method), 80
 initiating() (cirq.PointOptimizationSummary method), 365
 initiating() (cirq.PointOptimizer method), 366
 initiating() (cirq.Points method), 324
add_accepting()
add_accepting_equivalence_group()
add_cycle()
add_edge()
add_edges_from()
add_equivalence_group()
add_job_labels()
add_node()
add_nodes_from()
add_path()
add_program_labels()
add_star()
add_weighted_edges_from()
adj()
adjacency() (cirq.CircuitDag method), 224
all_operations() (cirq.Circuit method), 202
all_operations() (cirq.CircuitDag method), 224
all_qubits() (cirq.Circuit method), 203
all_qubits() (cirq.CircuitDag method), 224
allclose_up_to_global_phase() (in module cirq), 370
amplitude_damp() (in module cirq), 134
AmplitudeDampingChannel (class in cirq), 135
AnnealSequenceSearchStrategy (class in cirq.google), 406
append() (cirq.Circuit method), 203
apply_channel() (in module cirq), 328
apply_matrix_to_slices() (in module cirq), 370
apply_unitary() (in module cirq), 329
apply_unitary_effect_to_state() (cirq.Circuit method), 203
ApplyChannelArgs (class in cirq), 343
ApplyUnitaryArgs (class in cirq), 345
approx_eq() (in module cirq), 330
are_all_matches_terminal() (cirq.Circuit method), 204
are_all_measurements_terminal() (cirq.Circuit method), 204
as_qubit_order() (cirq.QubitOrder static method), 257
assert_allclose_up_to_global_phase() (in module cirq.testing), 420
assert_circuits_with_terminal_measurement_are_equivalent() (in module cirq.testing), 420
assert_decompose_is_consistent_with_unitary() (in module cirq.testing), 421
assert_eigengate_implements_consistent_protocols() (in module cirq.testing), 421
assert_equivalent_repr() (in module cirq.testing), 422
assert_has_consistent_apply_unitary() (in module cirq.testing), 422
assert_has_consistent_apply_unitary_for() (in module cirq.testing), 422
assert_has_diagram() (in module cirq.testing), 423
assert_implements_consistent_protocols() (in module cirq.testing), 423
assert_pauli_expansion_is_consistent_with_unitary() (in module cirq.testing), 423
assert_phase_by_is_consistent_with_unitary() (in module cirq.testing), 423
assert_qasm_is_consistent_with_unitary() (in module cirq.testing), 424
assert_same_circuits() (in module cirq.testing), 424
asymmetric_depolarize() (in module cirq), 137
AsymmetricDepolarizingChannel (class in cirq), 137
at() (cirq.IonDevice method), 398
auxiliary_buffer0 (cirq.ApplyChannelArgs attribute), 344
auxiliary_buffer1 (cirq.ApplyChannelArgs attribute), 344
available_buffer (cirq.ApplyUnitaryArgs attribute), 345
axes (cirq.ApplyUnitaryArgs attribute), 346

B
batch_insert() (cirq.Circuit method), 204
batch_insert_into() (cirq.Circuit method), 205
batch_remove() (cirq.Circuit method), 205
bidiagonalize_real_matrix_pair_with_symmetric_products() (in module cirq), 371
bidiagonalize_unitary_with_special_orthogonals() (in module cirq), 372
bit_flip() (in module cirq), 140
BitFlipChannel (class in cirq), 141
bloch_vector_from_state_vector() (in module cirq), 264
bloch_vector_of() (cirq.SparseSimulatorStep method), 306
bloch_vector_of() (cirq.StateVectorMixIn method), 310
bloch_vector_of() (cirq.WaveFunctionTrialResult method), 320
block_diag() (in module cirq), 372
Bristlecone (in module cirq.google), 407
by_index() (cirq.Pauli static method), 179
by_relative_index() (cirq.Pauli static method), 190

C
can_add_operation_into_moment() (cirq.Device method), 64
can_add_operation_into_moment() (cirq.IonDevice method), 398
can_add_operation_into_moment() (cirq.NeutralAtomDevice method), 403
cancel_job() (cirq.google.Engine method), 410
canonicalize_half_turns() (in module cirq), 373
clear() (in module cirq), 123
CCXPowGate (class in cirq), 123
cancel_job() (in module cirq), 125
CCZPowGate (class in cirq), 126
channel() (in module cirq), 331
check_unused_args() (cirq.QasmArgs method), 350
chosen_angle_to_canonical_half_turns() (in module cirq), 373
chosen_angle_to_half_turns() (in module cirq), 373
Circuit (class in cirq), 200
circuit_diagram_info() (in module cirq), 331
CircuitDag (class in cirq), 218
CircuitDiagramInfo (class in cirq), 347
CircuitDiagramInfoArgs (class in cirq), 348
cirq.contrib.acquaintance (module), 428
cirq.contrib.paulistring (module), 428
cirq.contrib.qcircuit (module), 428
cirq.contrib.quirk (module), 428
clean() (cirq.LinearCombinationOfGates method), 170
clean() (cirq.LinearDict method), 384
clear() (cirq.CircuitDag method), 225
clear() (cirq.LinearCombinationOfGates method), 170
clear() (cirq.LinearDict method), 384
clear_operations_touching() (cirq.Circuit method), 205
CNOT (cirq.PauliInteractionGate attribute), 183
CNOT (in module cirq), 99
CNotPowGate (class in cirq), 99
coefficient() (cirq.PauliString property), 191
commutes() (in module cirq), 372
commutes_with() (cirq.Pauli method), 180
commutes_with() (cirq.PauliString method), 185
commutes_with_pauli() (cirq.SingleQubitCliffordGate method), 194
commutes_with_single_qubit_gate() (cirq.SingleQubitCliffordGate method), 194
compute_amplitudes() (cirq.SimulatesIntermediateWaveFunction method), 290
compute_amplitudes() (cirq.Simulator method), 301
compute_amplitudes_sweep() (cirq.SimulatesIntermediateWaveFunction method), 291
compute_amplitudes_sweep() (cirq.Simulator method), 301
conjugated_by() (cirq.PauliString method), 185
content_present() (cirq.TextDiagramDrawer method), 391
CONTROL_TAG (in module cirq), 374
controlled() (cirq.AmplitudeDampingChannel method), 135
controlled() (cirq.AsymmetricDepolarizingChannel method), 138
controlled() (cirq.BitFlipChannel method), 141
controlled() (cirq.CCXPowGate method), 124
controlled() (cirq.CCZPowGate method), 127
controlled() (cirq.CNotPowGate method), 101
controlled() (cirq.ControlledGate method), 159
controlled() (cirq.CSwapGate method), 129
controlled() (cirq.CZPowGate method), 104
controlled() (cirq.DepolarizingChannel method), 144
controlled() (cirq.EigenGate method), 163
controlled() (cirq.Gate method), 165
controlled() (cirq.GeneralizedAmplitudeDampingChannel method), 148
controlled() (cirq.HPowGate method), 75
controlled() (cirq.IdentityGate method), 132
controlled() (cirq.ISwapGate method), 108
controlled() (cirq.MeasurementGate method), 78
controlled() (cirq.Pauli method), 180
controlled() (cirq.PauliInteractionGate method), 182
controlled() (cirq.PhaseDampingChannel method), 154
controlled() (cirq.PhasedXPowGate method), 80
controlled() (cirq.PhaseFlipChannel method), 156
controlled() (cirq.ResetChannel method), 151
controlled() (cirq.SingleQubitCliffordGate method), 194
controlled() (cirq.SingleQubitGate method), 174
controlled() (cirq.SingleQubitMatrixGate method), 84
controlled() (cirq.SwapGate method), 112
controlled() (cirq.ThreeQubitGate method), 176
controlled() (cirq.TwoQubitGate method), 177
controlled() (cirq.TwoQubitMatrixGate method), 86
controlled() (cirq.XPowGate method), 89
controlled() (cirq.XXPowGate method), 115
controlled() (cirq.YPowGate method), 93
controlled() (cirq.YYPowGate method), 118
controlled() (cirq.ZPowGate method), 96
controlled() (cirq.ZZPowGate method), 121
controlled_by() (cirq.ControlledOperation method), 161
controlled_by() (cirq.GateOperation method), 167
controlled_by() (cirq.Operation method), 172
controlled_by() (cirq.ParallelGateOperation method), 255
controlled_by() (cirq.PauliString method), 187
ControlledGate (class in cirq), 158
ControlledOperation (class in cirq), 160
convert() (cirq.ConvertToNeutralAtomGates method), 401
convert() (cirq.google.ConvertToXmonGates method), 407
convert_circuit() (cirq.ConvertToIonGates method), 397
convert_field() (cirq.QasmArgs method), 350
expand_matrix_in_orthogonal_basis() (in module cirq), 376
ExpandComposite (class in cirq), 361
expectation_from_density_matrix() (cirq.PauliString method), 187
expectation_from_wavefunction() (cirq.PauliString method), 188
explicit() (cirq.QubitOrder static method), 258
exponent() (cirq.CCXPowGate property), 125
exponent() (cirq.CCZPowGate property), 128
exponent() (cirq.CNotPowGate property), 102
exponent() (cirq.CZPowGate property), 105
exponent() (cirq.EigenGate property), 164
exponent() (cirq.HPowGate property), 76
exponent() (cirq.ISwapPowGate property), 109
exponent() (cirq.PauliInteractionGate property), 184
exponent() (cirq.PhasedXPowGate property), 82
exponent() (cirq.SwapPowGate property), 113
exponent() (cirq.XPowGate property), 91
exponent() (cirq.XXPowGate property), 116
exponent() (cirq.YPowGate property), 94
exponent() (cirq.YYPowGate property), 119
exponent() (cirq.ZPowGate property), 98
exponent() (cirq.ZZPowGate property), 122

F
final_density_matrix
(cirq.DensityMatrixTrialResult attribute), 277

final_simulator_state
(cirq.DensityMatrixTrialResult attribute), 277

final_state (cirq.WaveFunctionTrialResult attribute), 319

final_wavefunction() (cirq.Circuit method), 206
findall_nodes_until_blocked()
(cirq.CircuitDag method), 227
findall_operations() (cirq.Circuit method), 207
findall_operations_between() (cirq.Circuit method), 207
findall_operations_until_blocked()
(cirq.Circuit method, 208
findall_operations_with_gate_type()
(cirq.Circuit method), 208
flatten_op_tree() (in module cirq), 251
flip() (cirq.PauliTransform property), 192
force_horizontal_padding_after()
(cirq.TextDiagramDrawer method), 391
force_vertical_padding_after()
(cirq.TextDiagramDrawer method), 392
format() (cirq.QasmArgs method), 350
format_field() (cirq.QasmArgs method), 350
Foxtail (in module cirq.google), 415
FREDKIN (in module cirq), 130
freeze_op_tree() (in module cirq), 251
fresh_copy() (cirq.CircuitDag method), 227
from_circuit() (cirq.CircuitDag static method), 227
from_diagram() (cirq.GridQubit static method), 66
from_double_map() (cirq.SingleQubitCliffordGate static method), 195
from_ops() (cirq.Circuit static method), 209
from_op() (cirq.CircuitDag static method), 227
from_pauli() (cirq.SingleQubitCliffordGate static method), 195
from_quarter_turns()
(cirq.SingleQubitCliffordGate static method), 196
from_single() (cirq.PauliString static method), 189
from_single_map() (cirq.SingleQubitCliffordGate static method), 196
from_single_parameter_set()
(cirq.TrialResult static method), 314
from_xz_map() (cirq.SingleQubitCliffordGate static method), 196
fromkeys() (cirq.LinearCombinationOfGates class method), 170
fromkeys() (cirq.LinearDict class method), 384
full_invert_mask() (cirq.MeasurementGate method), 79

G
gamma() (cirq.AmplitudeDampingChannel property), 137
gamma() (cirq.GeneralizedAmplitudeDampingChannel property), 149
gamma() (cirq.PhaseDampingChannel property), 155
Gate (class in cirq), 165
gate() (cirq.ControlledOperation property), 162
gate() (cirq.GateOperation property), 168
gate() (cirq.Operation property), 173
gate() (cirq.ParallelGateOperation property), 257
gate() (cirq.PauliString property), 191
gate_to_proto_dict() (in module cirq.google), 415
GateOperation (class in cirq), 167
generalized_amplitude_damp() (in module cirq), 146
GeneralizedAmplitudeDampingChannel (class in cirq), 146
generate_boixo_2018_supremacy_circuits_v2()
(in module cirq), 395
generate_boixo_2018_supremacy_circuits_v2_bristlecone()
(in module cirq), 395
generate_boixo_2018_supremacy_circuits_v2_grid()
(in module cirq), 396
get() (cirq.LinearCombinationOfGates method), 170
get() (cirq.LinearDict method), 384
on_each() (cirq.SingleQubitMatrixGate method), 84
on_each() (cirq.XPowGate method), 90
on_each() (cirq.YPowGate method), 93
on_each() (cirq.ZPowGate method), 97
op_at_on() (cirq.ScheduledOperation static method), 262
op_gate_of_type() (in module cirq), 254
OP_TREE (in module cirq), 254
operates_on() (cirq.Moment method), 252
operates_on_single_qubit() (cirq.Moment method), 253
Operation (class in cirq), 172
operation_at() (cirq.Circuit method), 211
operations() (cirq.Moment property), 253
operations_happening_at_same_time_as() (cirq.Schedule method), 261
optimization_at() (cirq.ConvertToCzAndSingleGates method), 359
optimization_at() (cirq.ConvertToNeutralAtomGates method), 401
optimization_at() (cirq.ExpandComposite method), 362
optimization_at() (cirq.google.ConvertToXmonGates method), 407
optimization_at() (cirq.MergeInteractions method), 364
optimization_at() (cirq.MergeSingleQubitGates method), 365
optimization_at() (cirq.PointOptimizer method), 365
optimize_circuit() (cirq.PointOptimizer method), 366
optimized_for_xmon() (in module cirq.google), 362
order() (cirq.CircuitDag method), 232
order_for() (cirq.QubitOrder method), 258
ordered_nodes() (cirq.CircuitDag method), 232
OrderTester (class in cirq.testing), 425
out_buffer (cirq.ApplyChannelArgs attribute), 344
out_degree() (cirq.CircuitDag property), 248
out_edges() (cirq.CircuitDag property), 249

P
p() (cirq.BitFlipChannel property), 143
p() (cirq.DepolarizingChannel property), 146
p() (cirq.GeneralizedAmplitudeDampingChannel property), 149
p() (cirq.PhaseFlipChannel property), 158
p_x() (cirq.AsymmetricDepolarizingChannel property), 140
p_y() (cirq.AsymmetricDepolarizingChannel property), 140
p_z() (cirq.AsymmetricDepolarizingChannel property), 140
pack_results() (in module cirq.google), 417
ParallelGateOperation (class in cirq), 255
param_dict (cirq.ParamResolver attribute), 323
param_tuples() (cirq.Linspace method), 322
param_tuples() (cirq.Points method), 324
param_tuples() (cirq.Sweep method), 325
param_resolver (class in cirq), 323
params (cirq.DensityMatrixTrialResult attribute), 276
params (cirq.SimulationTrialResult attribute), 298
params (cirq.TrialResult attribute), 313
parse() (cirq.QasmArgs method), 350
partial_trace() (in module cirq), 386
pass_operations_over() (cirq.PauliString method), 190
Pauli (class in cirq), 179
pauliExpansion() (in module cirq), 338
PauliInteractionGate (class in cirq), 181
PauliString (class in cirq), 184
PauliTransform (class in cirq), 192
PeriodicValue (class in cirq), 387
phase_by() (in module cirq), 339
phase_damp() (in module cirq), 152
phase_exponent() (cirq.PhasedXPowGate property), 82
phase_exponent() (cirq.XPowGate property), 91
phase_exponent() (cirq.YPowGate property), 94
phase_flip() (in module cirq), 153
phased_pauli_product() (cirq.Pauli method), 180
PhaseDampingChannel (class in cirq), 153
PhasedXPowGate (class in cirq), 80
PhaseFlipChannel (class in cirq), 156
plot_state_histogram() (in module cirq), 324
PointOptimizationSummary (class in cirq), 365
PointOptimizer (class in cirq), 366
Points (class in cirq), 324
pop() (cirq.LinearCombinationOfGates method), 171
pop() (cirq.LinearDict method), 385
popitem() (cirq.LinearCombinationOfGates method), 171
popitem() (cirq.LinearDict method), 385
precision (cirq.CircuitDiagramInfoArgs attribute), 348
pred() (cirq.CircuitDag property), 250
predecessors() (cirq.CircuitDag method), 232
prev_moment_operating_on() (cirq.Circuit method), 211
program_as_schedule() (cirq.google.Engine method), 412
qubit_index_to_equivalence_group_key() (cirq.CCXPowGate method), 125
qubit_index_to_equivalence_group_key() (cirq.CCZPowGate method), 127
qubit_index_to_equivalence_group_key() (cirq.CSwapGate method), 130
qubit_index_to_equivalence_group_key() (cirq.CZPowGate method), 105
qubit_index_to_equivalence_group_key() (cirq.InterchangeableQubitsGate method), 169
qubit_index_to_equivalence_group_key() (cirq.ISwapPowGate method), 108
qubit_index_to_equivalence_group_key() (cirq.PauliInteractionGate method), 183
qubit_index_to_equivalence_group_key() (cirq.SwapPowGate method), 112
qubit_index_to_equivalence_group_key() (cirq.XXPowGate method), 115
qubit_index_to_equivalence_group_key() (cirq.YYPowGate method), 118
qubit_index_to_equivalence_group_key() (cirq.ZZPowGate method), 121
qubit_list() (cirq.NeutralAtomDevice method), 404
qubit_map (cirq.CircuitDiagramInfoArgs attribute), 415
qubit_map (cirq.DensityMatrixStepResult attribute), 273
qubit_map() (cirq.DensityMatrixTrialResult property), 277
qubit_map() (cirq.SimulationTrialResult property), 298
qubit_map() (cirq.SparseSimulatorStep property), 309
qubit_map() (cirq.StateVectorSimulatorStep property), 312
qubit_map() (cirq.WaveFunctionTrialResult property), 322
QubitOrder (class in cirq), 257
QubitOrderOrList (in module cirq), 259
qubits() (cirq.ControlledOperation property), 162
qubits() (cirq.GateOperation property), 168
qubits() (cirq.Moment property), 253
qubits() (cirq.Operation property), 173
qubits() (cirq.ParallelGateOperation property), 257
qubits() (cirq.PauliString property), 192
query() (cirq.Schedule method), 261
Q
qasm() (in module cirq), 340
QasmArgs (class in cirq), 349
QasmOutput (class in cirq), 351
Qid (class in cirq), 71
qid_shape() (cirq.Circuit method), 211
qid_shape() (in module cirq), 340
random_circuit() (in module cirq.testing), 426
random_orthogonal() (in module cirq.testing), 427
random_special_orthogonal() (in module cirq.testing), 427
random_special_unitary() (in module cirq.testing), 427
random_superposition() (in module cirq.testing), 427
random_unitary() (in module cirq.testing), 427
range() (cirq.LineQubit static method), 69
range() (cirq.NamedQubit static method), 70
raw_picos() (cirq.Timestamp method), 394
reachable_frontier_from() (cirq.Circuit method), 211
rect() (cirq.GridQubit static method), 67
reflection_matrix_pow() (in module cirq), 387
relative_index() (cirq.Pauli method), 180
remove_edge() (cirq.CircuitDag method), 233
remove_edges_from() (cirq.CircuitDag method), 233
remove_job_labels() (cirq.google.Engine method), 413
remove_node() (cirq.CircuitDag method), 234
remove_nodes_from() (cirq.CircuitDag method), 234
R
Index 441
remove_program_labels()  (cirq.google.Engine method), 413
render()  (cirq.TextDiagramDrawer method), 392
repetitions()  (cirq.TrialResult property), 316
reset()  (in module cirq), 150
ResetChannel  (class in cirq), 150
resolve_parameters()  (in module cirq), 341
reverse()  (cirq.CircuitDag method), 234
right_axes()  (cirq.ApplyChannelArgs attribute), 344
run()  (cirq.DensityMatrixSimulator method), 268
run()  (cirq.google.Engine method), 413
run()  (cirq.Sampler method), 282
run()  (cirq.SimulatesSamples method), 295
run_async()  (cirq.DensityMatrixSimulator method), 268
run_async()  (cirq.Sampler method), 282
run_async()  (cirq.SimulatesSamples method), 295
run_async()  (cirq.Simulator method), 302
run_sweep()  (cirq.DensityMatrixSimulator method), 269
run_sweep()  (cirq.google.Engine method), 413
run_sweep()  (cirq.Sampler method), 283
run_sweep()  (cirq.SimulatesSamples method), 296
run_sweep()  (cirq.Simulator method), 302
run_sweep_async()  (cirq.DensityMatrixSimulator method), 269
run_sweep_async()  (cirq.Sampler method), 283
run_sweep_async()  (cirq.SimulatesSamples method), 296
run_sweep_async()  (cirq.Simulator method), 303
Rx()  (in module cirq), 82
Ry()  (in module cirq), 82
Rz()  (in module cirq), 82

S  (in module cirq), 82
sample()  (cirq.DensityMatrixSimulator method), 269
sample()  (cirq.DensityMatrixStepResult method), 274
sample()  (cirq.Sampler method), 283
sample()  (cirq.SimulatesSamples method), 296
sample()  (cirq.Simulator method), 303
sample()  (cirq.SparseSimulatorStep method), 308
sample()  (cirq.StepResult method), 312
sample()  (cirq.WaveFunctionStepResult method), 318
sample()  (in module cirq), 279
sample_density_matrix()  (in module cirq), 280
sample_measurement_ops()  (cirq.DensityMatrixStepResult method), 275
sample_measurement_ops()  (cirq.SparseSimulatorStep method), 308
sample_measurement_ops()  (cirq.StepResult method), 313
sample_measurement_ops()  (cirq.WaveFunctionStepResult method), 319
sample_state_vector()  (in module cirq), 280
sample_sweep()  (in module cirq), 281
Sampler  (class in cirq), 281
sampler()  (cirq.google.Engine method), 414
save()  (cirq.QasmOutput method), 351
save_qasm()  (cirq.Circuit method), 214
Schedule  (class in cirq), 260
schedule_from_proto_dicts()  (in module cirq.google), 418
schedule_to_proto_dicts()  (in module cirq.google), 418
scheduled_operations  (cirq.Schedule attribute), 260
ScheduledOperation  (class in cirq), 262
selfloop_edges()  (cirq.CircuitDag method), 235
set_density_matrix()  (cirq.DensityMatrixStepResult method), 275
set_job_labels()  (cirq.google.Engine method), 414
set_program_labels()  (cirq.google.Engine method), 414
set_state_vector()  (cirq.SparseSimulatorStep method), 309
setdefault()  (cirq.LinearCombinationOfGates method), 171
setdefault()  (cirq.LinearDict method), 385
simulate()  (cirq.DensityMatrixSimulator method), 271
simulate()  (cirq.SimulatesFinalState method), 285
simulate()  (cirq.SimulatesIntermediateState method), 287
simulate()  (cirq.SimulatesIntermediateWaveFunction method), 292
simulate()  (cirq.Simulator method), 304
simulateMomentSteps()  (cirq.DensityMatrixSimulator method), 271
simulateMomentSteps()  (cirq.SimulatesIntermediateState method), 288
simulateMomentSteps()  (cirq.SimulatesIntermediateWaveFunction method), 293
simulateMomentSteps()  (cirq.Simulator method), 305
simulate_sweep()  (cirq.DensityMatrixSimulator method), 272
simulate_sweep()  (cirq.SimulatesFinalState method), 286
simulate_sweep()  (cirq.SimulatesIntermediateState method), 289
simulate_sweep()  (cirq.SimulatesIntermediateWaveFunction method), 292
simulate_sweep() (cirq.Simulator method), 305
SimulatesFinalState (class in cirq), 284
SimulatesIntermediateState (class in cirq), 286
SimulatesIntermediateWaveFunction (class in cirq), 289
SimulatesSamples (class in cirq), 294
SimulationTrialResult (class in cirq), 297
Simulator (class in cirq), 298
single_qubit_matrix_to_gates() (in module cirq), 367
single_qubit_matrix_to_phased_x_z() (in module cirq), 367
single_qubit_matrix_to_pauli_rotations() (in module cirq), 367
single_qubit_op_to_framed_phase_form() (in module cirq), 367
single_qubit_operations_after (cirq.KakDecomposition attribute), 381
single_qubit_operations_before (cirq.KakDecomposition attribute), 381
SingleQubitCliffordGate (class in cirq), 193
SingleQubitGate (class in cirq), 173
SingleQubitMatrixGate (class in cirq), 83
size() (cirq.CircuitDag method), 235
slice_for_qubits_equal_to() (in module cirq), 388
so4_to_magic_su2s() (in module cirq), 388
sorted_by() (cirq.QubitOrder static method), 258
SparseSimulatorStep (class in cirq), 306
square() (cirq.GridQubit static method), 67
state_vector() (cirq.SparseSimulatorStep method), 309
state_vector() (cirq.StateVectorMixin method), 311
state_vector() (cirq.WaveFunctionTrialResult method), 321
StateVectorMixin (class in cirq), 309
StepResult (class in cirq), 312
subgraph() (cirq.CircuitDag method), 235
subspace_index() (cirq.ApplyUnitaryArgs method), 347
succ() (cirq.CircuitDag property), 250
successors() (cirq.CircuitDag method), 236
SupportsApplyChannel (class in cirq), 351
SupportsApproximateEquality (class in cirq), 352
SupportsChannel (class in cirq), 352
SupportsCircuitDiagramInfo (class in cirq), 352
SupportsConsistentApplyUnitary (class in cirq), 352
SupportsDecompose (class in cirq), 352
SupportsDecomposeWithQubits (class in cirq), 354
SupportsExplicitHasQubits (class in cirq), 354
SupportsExplicitNumQubits (class in cirq), 354
SupportsExplicitQidShape (class in cirq), 355
SupportsMixture (class in cirq), 355
SupportsParameterization (class in cirq), 355
SupportsPhase (class in cirq), 355
SupportsQasm (class in cirq), 356
SupportsQasmWithArgs (class in cirq), 356
SupportsQasmWithArgsAndQubits (class in cirq), 356
SupportsTraceDistanceBound (class in cirq), 357
SupportsUnitary (class in cirq), 357
SWAP (in module cirq), 110
SwapPowGate (class in cirq), 110
Sweep (class in cirq), 325
Sweepable (in module cirq), 326

T

T (in module cirq), 85
target_tensor (cirq.ApplyChannelArgs attribute), 344
target_tensor (cirq.ApplyUnitaryArgs attribute), 345
targeted_conjugate_about() (in module cirq), 389
targeted_left_multiply() (in module cirq), 390
TextDiagramDrawer (class in cirq), 391
third() (cirq.Pauli method), 181
ThreeQubitGate (class in cirq), 175
Timestamp (class in cirq), 393
to() (cirq.PauliTransform property), 192
to_circuit() (cirq.CircuitDag method), 237
to_circuit() (cirq.Schedule method), 261
to_directed() (cirq.CircuitDag method), 237
to_directed_class() (cirq.CircuitDag method), 237
to_qasm() (cirq.Circuit method), 214
to_resolvers() (in module cirq), 327
to_text_diagram() (cirq.Circuit method), 215
to_text_diagram_drawer() (cirq.Circuit method), 215
to_undirected() (cirq.CircuitDag method), 238
to_undirected_class() (cirq.CircuitDag method), 238
to_unitary_matrix() (cirq.Circuit method), 216
to_valid_density_matrix() (in module cirq), 316
to_valid_state_vector() (in module cirq), 317
to_z_basis_ops() (cirq.PauliString method), 190
TOFFOLI (in module cirq), 130
validate_args() (cirq.ZZPowGate method), 312
validate_circuit() (cirq.Device method), 64
validate_circuit() (cirq.IonDevice method), 399
validate_circuit() (cirq.NeutralAtomDevice method), 404
validate_dimension() (cirq.GridQubit static method), 68
validate_dimension() (cirq.LineQubit static method), 69
validate_dimension() (cirq.NamedQubit static method), 71
validate_dimension() (cirq.Qid static method), 72
validate_gate() (cirq.IonDevice method), 399
validate_gate() (cirq.NeutralAtomDevice method), 404
validate_mixture() (in module cirq), 342
validate_moment() (cirq.Device method), 64
validate_moment() (cirq.IonDevice method), 399
validate_moment() (cirq.NeutralAtomDevice method), 404
validate_normalized_state() (in module cirq), 317
validate_operation() (cirq.Device method), 65
validate_operation() (cirq.IonDevice method), 399
validate_operation() (cirq.NeutralAtomDevice method), 404
validate_probability() (in module cirq), 317
validate_schedule() (cirq.Device method), 65
validate_schedule() (cirq.IonDevice method), 399
validate_schedule() (cirq.NeutralAtomDevice method), 405
validate_scheduled_operation() (cirq.Device method), 65
validate_scheduled_operation() (cirq.IonDevice method), 400
validate_scheduled_operation() (cirq.NeutralAtomDevice method), 405
validate_version() (cirq.QasmArgs method), 351
value_equality() (in module cirq), 394
values() (cirq.ParamResolver method), 323
values() (cirq.LinearCombinationOfGates method), 171
values() (cirq.PauliString method), 385
values() (cirq.PauliString method), 191
vertical_line() (cirq.TextDiagramDrawer method), 393
vformat() (cirq.QasmArgs method), 351

WaveFunctionSimulatorState (class in cirq), 318
WaveFunctionStepResult (class in cirq), 318
WaveFunctionTrialResult (class in cirq), 319
width() (cirq.TextDiagramDrawer method), 393
with_args() (cirq.CircuitDiagramInfoArgs method), 349
with_axes_transposed_to_start() (cirq.ApplyUnitaryArgs method), 347
with_bits_flipped() (cirq.MeasurementGate method), 79
with_canonical_global_phase() (cirq.ZPowGate method), 90
with_canonical_global_phase() (cirq.YPowGate method), 94
with_canonical_global_phase() (cirq.ZPowGate method), 98
with_device() (cirq.Circuit method), 217
with_dimension() (cirq.GridQubit method), 68
with_dimension() (cirq.LineQubit method), 69
with_dimension() (cirq.NamedQubit method), 71
with_dimension() (cirq.Qid method), 72
with_gate() (cirq.GateOperation method), 168
with_gate() (cirq.ParallelGateOperation method), 256
with_noise() (cirq.Circuit method), 218
with_operation() (cirq.ControlledOperation method), 162
with_operation() (cirq.GateOperation method), 168
with_operation() (cirq.Operation method), 173
with_operation() (cirq.ParallelGateOperation method), 256
with_qubits() (cirq.PauliString method), 191
without_operations_touching() (cirq.Moment method), 253
wrap_in_linear_combination() (cirq.AmplitudeDampingChannel method), 137
wrap_in_linear_combination() (cirq.AsymmetricDepolarizingChannel method), 140
wrap_in_linear_combination() (cirq.BitFlipChannel method), 143
wrap_in_linear_combination() (cirq.CCXPowGate method), 125
wrap_in_linear_combination() (cirq.CCZPowGate method), 128
wrap_in_linear_combination() (cirq.CNotPowGate method), 102
wrap_in_linear_combination() (cirq.ControlledGate method), 160
wrap_in_linear_combination() (cirq.CSwapGate method), 130
wrap_in_linear_combination() (cirq.CZPowGate method), 105
Cirq Documentation, Release 0.6.1

wrap_in_linear_combination()
  (cirq.DepolarizingChannel method), 146
wrap_in_linear_combination()
  (cirq.EigenGate method), 164
wrap_in_linear_combination()
  (cirq.Gate method), 166
wrap_in_linear_combination()
  (cirq.GeneralizedAmplitudeDampingChannel method), 149
wrap_in_linear_combination()
  (cirq.HPowGate method), 76
wrap_in_linear_combination()
  (cirq.IdentityGate method), 133
wrap_in_linear_combination()
  (cirq.ISwapPowGate method), 109
wrap_in_linear_combination()
  (cirq.MeasurementGate method), 80
wrap_in_linear_combination()
  (cirq.Pauli method), 181
wrap_in_linear_combination()
  (cirq.PauliInteractionGate method), 183
wrap_in_linear_combination()
  (cirq.PhaseDampingChannel method), 155
wrap_in_linear_combination()
  (cirq.PhasedXPowGate method), 82
wrap_in_linear_combination()
  (cirq.PhaseFlipChannel method), 158
wrap_in_linear_combination()
  (cirqRESETChannel method), 152
wrap_in_linear_combination()
  (cirq.SingleQubitCliffordGate method), 198
wrap_in_linear_combination()
  (cirq.SingleQubitGate method), 175
wrap_in_linear_combination()
  (cirq.SingleQubitMatrixGate method), 85
wrap_in_linear_combination()
  (cirq.SingleQubitXPowGate method), 113
wrap_in_linear_combination()
  (cirq.ThreeQubitGate method), 177
wrap_in_linear_combination()
  (cirq.TwoQubitGate method), 178
wrap_in_linear_combination()
  (cirq.TwoQubitMatrixGate method), 87
wrap_in_linear_combination()
  (cirq.XPowGate method), 90
wrap_in_linear_combination()
  (cirq.XXPowGate method), 116
wrap_in_linear_combination()
  (cirq.XYPowGate method), 94
wrap_in_linear_combination()
  (cirq.YPowGate method), 98
wrap_in_linear_combination()
  (cirq.ZPowGate method), 122
write() (cirq.TextDiagramDrawer method), 393

X
  X (cirq.SingleQubitCliffordGate attribute), 198
  X (in module cirq), 87
  X_sqrt (cirq.SingleQubitCliffordGate attribute), 198
  xmon_op_from_proto_dict () (in module cirq.google), 418
  XPowGate (class in cirq), 87
  XX (in module cirq), 113
  XXPowGate (class in cirq), 113

Y
  Y (cirq.SingleQubitCliffordGate attribute), 199
  Y (in module cirq), 91
  Y_sqrt (cirq.SingleQubitCliffordGate attribute), 199
  YYPowGate (class in cirq), 91
  YY (in module cirq), 116
  YYPowGate (class in cirq), 116

Z
  Z (cirq.SingleQubitCliffordGate attribute), 199
  Z (in module cirq), 95
  Z_sqrt (cirq.SingleQubitCliffordGate attribute), 199
  ZPowGate (class in cirq), 95
  ZZ (in module cirq), 119
  ZZPowGate (class in cirq), 119