## Contents

1 Alpha Disclaimer ......................................................... 3

2 User Documentation .................................................. 5
   2.1 Installing Cirq .................................................. 5
   2.2 Tutorial ......................................................... 7
   2.3 Circuits ........................................................ 17
   2.4 Gates ............................................................ 23
   2.5 Simulation ....................................................... 26
   2.6 Schedules and Devices .......................................... 30
   2.7 Development .................................................... 33
   2.8 Examples ......................................................... 38

3 API Reference .......................................................... 59
   3.1 API Reference .................................................... 59

Python Module Index .................................................... 321
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 3.5.2 or greater (or else python 2.7).
   
   See [Installing Python 3 on Linux](https://docs.python-guide.org/motivation/virtualenv/) at 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 system dependencies that pip can’t handle.

```bash
sudo apt-get install python3-tk texlive-latex-base latexmk
```

- Without `python3-tk`, plotting functionality won’t work.
- Without `texlive-latex-base` and `latexmk`, pdf writing functionality will not work.

5. Check that it works!

```python
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)
```

### 2.1.3 Installing on Mac OS X

1. Make sure you have python 3.5 or greater (or else python 2.7).
   
   See Installing Python 3 on Mac OS X @ the hitchhiker’s guide to python.

2. Consider using a virtual environment.

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

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

4. Check that it works!

   ```python
   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 (or else python 2.7.9+).
   
   See Installing Python 3 on Windows @ the hitchhiker’s guide to python.

3. Use `pip` to install `cirq`:
python -m pip install --upgrade pip
python -m pip install cirq

4. Check that it works!

```python
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)
```

### 2.1.5 Installing on Docker

This will create a Docker image containing Cirq’s source code. It will isolate a Cirq installation from the rest of the system.

1. Install Docker on your host system.

2. Build the docker image for your system:

   ```bash
git clone https://github.com/quantumlib/Cirq
cd Cirq
docker build -t cirq/cirq . # This builds the actual image based on latest Ubuntu, and installs Cirq with the needed dependencies.
```

3. Check that it works!

   ```bash
docker run -it cirq/cirq python3 -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)
```

4. You can use the created image as:

   ```bash
docker run -it cirq/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 \((\text{ISING})\). This problem is \( \text{NP} \)-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

\[
E = \sum_{i,j} J_{i,j} s_i s_j + \sum_i h_i s_i
\]

where here each \( s_i, J_{i,j}, \text{and } h_i \) 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 \( J_{i,j}, \text{and } h_i \), find an assignment of \( s_i \) 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

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

where \( p_i \) 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 `Circuits` 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. A `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]

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

```python
import cirq

# define 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)
```

(continues on next page)
Here we see that we’ve created a bunch of GridQubits. GridQubits implement the QubitId class, which just means that they are equatable and hashable. QubitId 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 \( H \) to every qubit whose row index plus column index is even and an \( X \) gate to every qubit whose row index plus column index is odd. To do this we write

```python
# 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)]
```

```python
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)
```

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. InsertStrategies describe 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:

```python
circuit = cirq.Circuit()
circuit.append([cirq.H(q) for q in qubits if (q.row + q.col) % 2 == 0], strategy=cirq.InsertStrategy.NEW_THEN_INLINE)
```
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.""
    rot = cirq.XPowGate(exponent=half_turns)
    for i in range(length):
        circuit.append(rot, strategy=cirq.InsertStrategy.NEW_THEN_INLINE)
```

(continues on next page)
for j in range(length):
    yield rot(cirq.GridQubit(i, j))

circuit = cirq.Circuit()
circuit.append(rot_x_layer(2, 0.1))
print(circuit)
# prints
# (0, 0): X^0.1
# (0, 1): X^0.1
# (1, 0): X^0.1
# (1, 1): X^0.1

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

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(rows)] for _ in range(cols)]

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

```python
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))
            if jr_ij == -1:
                yield cirq.X(cirq.GridQubit(i, 0))
                yield cirq.X(cirq.GridQubit(i + 1, 0))
    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))
            if jc_ij == -1:
                yield cirq.X(cirq.GridQubit(i, 0))
                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):

```python
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)
    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)
```

1.2. Tutorial
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.google.XmonSimulator()
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 {}'.format(obj_func(results)))
# prints something like
# Value of the objective function 6.2
```

### 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 value 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
circuit = cirq.Circuit()
alpha = cirq.Symbol('alpha')
beta = cirq.Symbol('beta')
```
Note now that the circuit’s gates are parameterized.

Parameters are specified at runtime using a `ParamResolver` which is 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 `LinSpace`.

```python
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', 0.7000000000000001)]) 6.58
# OrderedDict([('alpha', 0.1), ('beta', 0.1), ('gamma', 0.9)]) 6.58
... # OrderedDict([('alpha', 0.9), ('beta', 0.9), ('gamma', 0.7000000000000001)]) 0.76
# OrderedDict([('alpha', 0.9), ('beta', 0.9), ('gamma', 0.9)]) 0.94```

### 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.

```python
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.
```

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: Circuit and 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 some effect that operates on a specific subset of Qubits, the most common type of Operation is a GateOperation.

Let’s unpack this.

At the base of this construction is the notion of a qubit. In Cirq, qubits are represented by subclasses of the QubitId base class. Different subclasses of QubitId 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 QubitId. For example, we can create a 3 by 3 grid of qubits using

```
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.

```
# 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)
```

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)
```

2.3. Circuits 19
This appended an entire new moment to the qubit, which we can continue to do,

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

```plaintext
# prints
# (0, 0): @
# (1, 0): @
# (2, 0): @
```

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

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

```plaintext
# prints
# (0, 0): @
# (1, 0): @
# (2, 0): @
```

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:

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

```plaintext
# prints
# (0, 0): @
# (1, 0): @
```

(continues on next page)
# (2, 0): H

After creating the first moment with a CZ gate, the second append uses the InsertStrategy.EARLIEST strategy. The H on q0 cannot slide back, while the H on q2 can and so ends up in the first Moment.

Contrast this with the InsertStrategy.NEW InsertStrategy:

```
 InsertStrategy.NEW: Every operation that is inserted is created in a new moment.
```

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

```
# prints
# (0, 0): H
# (1, 0): H
# (2, 0): H
```

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:

```
 InsertStrategy.INLINE: Attempts to add the operation to insert into the moment just before the desired insert location. But, if there’s already an existing operation affecting any of the qubits touched by the operation to insert, a new moment is created instead.
```

```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)
```

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

After two initial CZ between the second and third qubit, we try to insert 3 H Operations. We see that the H on the first qubit is inserted into the previous Moment, but the 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:

```
 InsertStrategy.NEW_THEN_INLINE: Creates a new moment at the desired insert location for the first operation, but then switches to inserting operations according to InsertStrategy.INLINE.
```

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

```
# prints
```

(continues on next page)
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.

```
# (0, 0): ———H———
# # (1, 0): ———O——
# # (2, 0): ———O——
```

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)
    # prints
# CZ((0, 0), (1, 0))
# [cirq.H.on(cirq.GridQubit(0, 0)), cirq.H.on(cirq.GridQubit(1, 0)), cirq.H.on(cirq.
  →GridQubit(2, 0))]
# [cirq.CZ.on(cirq.GridQubit(1, 0), cirq.GridQubit(2, 0))]
# [cirq.H.on(cirq.GridQubit(0, 0)), [cirq.CZ.on(cirq.GridQubit(1, 0), cirq.
  →GridQubit(2, 0))]]

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

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 an 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`.  

22 Chapter 2. User Documentation
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 via the static method Circuit.from_ops (which takes an insertion strategy as a parameter):

```python
circuit = cirq.Circuit.from_ops(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.from_ops(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.from_ops(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].

### 2.4 Gates

A Gate is an operation that can be applied to a collection of qubits (objects with a QubitId). 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 qubit and classes that implement Gate must supply the num_qubits method. For convenience one can use the SingleQubitGate, TwoQubitGate, and ThreeQubitGate classes for these common gate sizes. Boiler plate code can also be reduced by using MultiQubitGate.
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.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 *invertable* 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
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 $Z**0.5 \ S \ \text{np.diag}(1, i), Z**-0.5 \ S**-1, X**0.5 \ H\ S\ H$, and the square root of $Y$ is inferred via the right hand rule.

**_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(θ) is $\exp(-i Z \theta /2)$. 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> 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

TODO: describe these in more detail.

2.5 Simulation

Cirq comes with built in Python simulators for testing out small circuits. One of these simulators works for generic gates that implement their unitary matrix, cirq.Simulator. The other simulator is customized for the native gate set of Google’s Xmon hardware cirq.google.XmonSimulator. This later simulator can shard its simulation across different processes/threads and so take advantage of multiple cores/CPUs. Depending on your local computer architecture one or the other of these may be faster, but we recommend starting with cirq.Simulator.

Here is a simple circuit

```python
import cirq

q0 = cirq.GridQubit(0, 0)
q1 = cirq.GridQubit(1, 0)

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:
        (continues on next page)
```
yield cirq.measure(q0, key='q0'), cirq.measure(q1, key='q1')

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

print(circuit)
# prints
# (0, 0): ───X^0.5──@──X^0.5──M('q0')──
# (1, 0): ───X^0.5──@──X^0.5──M('q1')──

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 a 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's 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))
```

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.

### 2.5.1 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).

2.5. Simulation
The simplest `qubit_order` value you can provide is a list of the qubits in the desired order. 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.from_ops(cirq.X(q_flip))

# first qubit in order flipped
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.]
```

### 2.5.2 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
circuit = cirq.Circuit()
circuit.append(basic_circuit())
for i, step in enumerate(simulator.simulate_moment_steps(circuit)):
    print('state at step %d: %s' % (i, np.around(step.state_vector(), 3)))
(continues on next page)```
# 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 lie entirely in the computation basis state for the binary expansion of the passed integer.

### 2.5.3 XmonSimulator

In addition to `cirq.Simulator` there is also a simulator which is specialized to the Google native gate set. In particular this simulator is specialized to use the `CZPowGate`, `MeasurementGate`, `PhasedXPowGate`, `XPowGate`, `YPowGate`, and the `ZPowGate`. This simulator can be configured to use processes or threads, and depending on your local computing architecture may sometimes be faster or slower than `cirq.Simulator`.

### 2.5.4 Gate sets

The `XmonSimulator` is designed to work with operations that are either a `GateOperation` applying a supported gate (such as `cirq.CZ`), a composite operation that implements `decompose`, or a 1-qubit or 2-qubit operation that returns a unitary matrix from its `unitary` method.

So if you are implementing a custom gate, there are two options for getting it to work with the simulator:

- Implement a `decompose` method that returns supported gates (or gates that decompose into supported gates).
- If the operation applies to two or fewer qubits, implement a `unitary` method that returns the operation’s matrix.

### 2.5.5 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
rot_w_gate = cirq.X**cirq.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 -0.35j -0.35j -0.15+0.j]
# [0.5 +0.j -0.5j -0.5j -0.5 +0.j]
# [0.15+0.j -0.35j -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.

### 2.5.6 XmonSimulator Configurations and Options

The xmon simulator also contain some extra configuration on the simulate commands. One of these is initial_state. This can be passed the full wave function as a numpy array, or the initial state as the binary expansion of a supplied integer (following the order supplied by the qubits list).

A simulator itself can also be passed Options in it’s constructor. These options define some configuration for how the simulator runs. For the xmon simulator, these include

- **num_shards**: The simulator works by sharding the wave function over this many shards. If this is not a power of two, the smallest power of two less than or equal to this number will be used. The sharding shards on the first log base 2 of this number qubit’s state. When this is not set the simulator will use the number of cpus, which tends to max out the benefit of multi-processing.

- **min_qubits_before_shard**: Sharding and multiprocessing does not really help for very few number of qubits, and in fact can hurt because processes have a fixed (large) cost in Python. This is the minimum number of qubits that are needed before the simulator starts to do sharding. By default this is 10.

### 2.6 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 QubitIds upon which the gate acts.)

2.6.1 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: {}'.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:
    (continues on next page)
print(e)
# prints something like
# ValueError: Non-local interaction: Operation(cirq.CZ, (GridQubit(0, 0), GridQubit(2, 0)))

2.6.2 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
import cirq

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

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

```python
schedule = cirq.moment_by_moment_schedule(circuit.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(circuit.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.
2.7 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.

2.7.1 Cloning the repository

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

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

To do your development in a Docker virtual machine, you can use dev_tools/Dockerfile:

```
git clone https://github.com/quantumlib/Cirq
cd Cirq/dev_tools
docker build -t cirq/dev .
# This builds the actual image based on latest Ubuntu, cloning the Cirq tree into it with the needed dependencies.
docker run -it cirq/dev python3 -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.

2.7.2 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.

```
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.

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

2.7.3 Setting up an environment

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

1. 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
   ```

   If you change protocol buffers you will need to regenerate the proto files, so you should install the protocol buffer compiler. Instructions for this can be found here.

2. 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 -r requirements.txt
   python -m pip install -r dev_tools/conf/pip-list-dev-tools.txt
   python -m pip install -r cirq/contrib/contrib-requirements.txt
   ```

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

3. Check that the tests pass.

   ```bash
   pytest
   ```

4. (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 ./
   ```

2.7.4 Running continuous integration checks locally

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:

```bash
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:

```bash
# 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]

# Transpile to python 2 and run tests.
./check/pytest2  # Note: you must be in a python 2 virtual env to run this.

# 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]
```

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`. In order to run a check that is significantly more likely to agree with the Travis builds, you can use the `continuous-integration/check.sh` script:

```
./continuous-integration/check.sh
```

This script will create (temporary) virtual environments, do a fresh install of all relevant dependencies, transpile the python 2 code, and run all relevant checks within those clean environments. Note that creating the virtual environments takes time, and prevents some caching mechanisms from working, so `continuous-integration/check.sh` is significantly slower than the simpler check scripts. When using this script, you can run a subset of the checks using the `--only` flag. This flag can be `pylint`, `typecheck`, `pytest`, `pytest2`, or `incremental-coverage`.

### 2.7.5 Producing the Python 2.7 code

Run `dev_tools/python2.7-generate.sh` to transpile Cirq’s python 3 code into python 2.7 code:

```
./dev_tools/python2.7-generate.sh [output_dir] [input_dir] [virtual_env_with_3to2]
```

If you don’t specify any arguments then the input directory will be the current working directory, the output directory will be `python2.7-output` within the current directory, and `3to2` will be invoked in the current environment. The script fails with no effects if the output directory already exists.

### 2.7.6 Writing docstrings and generating documentation

Cirq uses Google style docstrings 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
```

(continues on next page)
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:

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

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

### 2.7.7 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.

   Once the package has uploaded, verify that it works.
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. It will do this for both python 2 and python 3. 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__)"
   ```
# 2.8 Examples

## 2.8.1 BCS Mean Field

```python
# coding=utf-8

# 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

\[ |\text{BCS} = \prod_k (u_k + v_k c_{k,\uparrow}^\dagger c_{k,\downarrow}^\dagger)|\text{vac}\]

where \(|\text{vac}\) is the vacuum state and \(u_k^2 = \frac{\xi_k}{\xi_k^2 + \Delta_k^2}^{1/2}\) and \(v_k^2 = \frac{1 - \xi_k}{\xi_k^2 + \Delta_k^2}^{1/2}\).

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


### 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)

\(W(0.25)\)

\(i\text{Swap}\)

\(i\text{Swap}^{-1.83}\)

\(W(0.625)\)

\(W(0.25)\)

\(i\text{Swap}\)

\(i\text{Swap}^{-1.67}\)

(continues on next page)
Circuit for the inverse fermionic Fourier transformation on the spin-up states:
(0, 0) (0, 1) (0, 2) (0, 3)
\[ S^{-1} \text{iSwap} \text{iSwap} \]
\[ S^{-1} 2 \]
\[ \text{iSwap} \text{iSwap}^{0.5} \]
\[ 2 \text{iSwap} \text{iSwap}^{0.5} \]
\[ S^{-1} \text{iSwap} \text{iSwap} \]
\[ S^{-1} S^{-1} \]
\[ \text{iSwap} \text{iSwap}^{0.5} \]
\[ S^{-1} \text{iSwap} \text{iSwap}^{0.5} \]
\[ S^{-1} \text{iSwap} \text{iSwap} \]
\[ S^{-1} S^{-1} \]

Circuit for the inverse fermionic Fourier transformation on the spin-down states:
(1, 0) (1, 1) (1, 2) (1, 3)
\[ S^{-1} \text{iSwap} \text{iSwap} \]
\[ S^{-1} 2 \]
\[ \text{iSwap} \text{iSwap}^{0.5} \]
```python
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.

    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.from_ops(
        bsgoliubov_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.from_ops(
    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.from_ops(
    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):
    """The 2-mode Bogoliubov transformation is mapped to two-qubit operations. We use
    the identity X S^\dag X X = Y S^\dag Y S X = X to transform the Hamiltonian
    XY+YX 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)

2.8. Examples
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.
    """
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]) ** 3
yield fermi_fourier_trans_2(qubits[2], qubits[3])
yield fswap(qubits[1], qubits[2])
```

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, 100000. * 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)
```python
return delta, bog_theta

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

## 2.8.2 Bell Inequality

"""Creates and simulates a circuit equivalent to a Bell inequality test.

=== EXAMPLE OUTPUT ===

Circuit:

(0, 0): H @ X^-0.25 X^0.5 M
(0, 1): H @ M
(1, 0): X X^0.5 M
(1, 1): H @ M

Simulating 75 repetitions...

Results

(a XOR b) == (x AND y):

Win rate: 84.0%

"""

```python
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.google.XmonSimulator().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])
```

(continues on next page)
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)
2.8.3 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:
\( f(a) = a \cdot <0, 1, 1, 1, 0, 0, 1, 0> + 1 \) (mod 2)
Circuit:
\[
\begin{align*}
(0, 0) & : H \quad H \quad M \\
(1, 0) & : H \quad M \\
(2, 0) & : H \quad M \\
(3, 0) & : H \quad M \\
(4, 0) & : H \quad M \\
(5, 0) & : H \quad M \\
(6, 0) & : H \quad M \\
(7, 0) & : H \quad M \\
(8, 0) & : X \quad X \quad X \quad X \quad X \\
\end{align*}
\]

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:
  \[ f(a) = a \cdot \langle \text{secret_factor_bits} \rangle + \text{secret_bias_bit} \ (\text{mod } 2) \].

# 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.google.XmonSimulator()
result = simulator.run(circuit, repetitions=circuit_sample_count)
frequencies = result.histogram(key='result', fold_func=bitstring)
print('Sampled results: ')
print(frequencies)

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

def make_oracle(input_qubits, output_qubit, secret_factor_bits, secret_bias_bit):
    """Gates implementing the function \[ f(a) = a \cdot \text{secret_factor_bits} + \text{secret_bias_bit} \ (\text{mod } 2) \].""
    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 \[ f(a) = a \cdot \text{secret_factor_bits} + \text{secret_bias_bit} \ (\text{mod } 2) \] 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()

## 2.8.4 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

### EXAMPLE OUTPUT ###
Secret bit sequence: [1, 0]
Circuit:
(0, 0): ───H───S─────H───S─────H───M
(1, 0): ───H───X─────H───X─────H───X───H───M
(2, 0): ───X───H───X
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.""
    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 \( x\neq x' \).""
    # 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.

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

# Make oracle (black box)

```python
oracle = make_oracle(input_qubits, output_qubit, x_bits)
```

# Embed the oracle into a quantum circuit implementing Grover's algorithm.

```python
circuit = make_grover_circuit(input_qubits, output_qubit, oracle)
print('Circuit: ')
print(circuit)
```

# Sample from the circuit a couple times.

```python
simulator = cirq.google.XmonSimulator()
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.

```python
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)))
```

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

---

## 2.8.5 Hello Qubit

```python
"""Creates and simulates a simple circuit.

=== EXAMPLE OUTPUT ===
Circuit:
(0, 0): ───X^0.5──M('m')──
Results:
m=110001111101100100
""

import cirq
def main():
    # Pick a qubit.
    qubit = cirq.GridQubit(0, 0)

    # Create a circuit
circuit = cirq.Circuit.from_ops(
        cirq.X(qubit)**0.5,  # Square root of NOT.
        cirq.measure(qubit, key='m')  # Measurement.
    )
    print("Circuit:")
    print(circuit)
```

(continues on next page)
# Simulate the circuit several times.
simulator = cirq.google.XmonSimulator()
result = simulator.run(circuit, repetitions=20)
print("Results:")
print(result)

if __name__ == '__main__':
    main()

## 2.8.6 Phase Estimator

```python
"""Creates and simulates a phase estimator circuit.

--- EXAMPLE OUTPUT ---
Estimation with 2 qubits.
Actual, Estimation (Raw binary)
0.0000, 0.0000 (00)
0.1000, 0.0000 (00)
0.2000, 0.2500 (01)
0.3000, 0.2500 (01)
0.4000, 0.5000 (10)
0.5000, 0.5000 (10)
0.6000, 0.5000 (10)
0.7000, 0.7500 (11)
0.8000, 0.7500 (11)
0.9000, 0.0000 (00)
RMS Error: 0.2915

Estimation with 4 qubits.
Actual, Estimation (Raw binary)
0.0000, 0.0000 (0000)
0.1000, 0.1250 (0010)
0.2000, 0.1875 (0011)
0.3000, 0.3125 (0101)
0.4000, 0.3750 (0110)
0.5000, 0.5000 (1000)
0.6000, 0.6250 (1010)
0.7000, 0.6875 (1011)
0.8000, 0.8125 (1101)
0.9000, 0.8750 (1110)
RMS Error: 0.0177

Estimation with 8 qubits.
Actual, Estimation (Raw binary)
0.0000, 0.0000 (00000000)
0.1000, 0.1016 (00011010)
0.2000, 0.1992 (00110011)
0.3000, 0.3008 (01001101)
0.4000, 0.3984 (01100110)
0.5000, 0.5000 (10000000)
0.6000, 0.6016 (10011010)
0.7000, 0.6992 (10110011)
0.8000, 0.8008 (11001101)
```

(continues on next page)
import numpy as np
import cirq

class QftInverse(cirq.MultiQubitGate):
    """Quantum gate for the inverse Quantum Fourier Transformation
    """
    def __init__(self, num_qubits):
        super(QftInverse, self).__init__(num_qubits)

    def _decompose_(self, qubits):
        """A quantum circuit (QFT_inv) with the following structure.
        ---H--@-------@--------@----------------------------------------------
        |  |                             |  |
        ---@^-0.5--+--------+---------H--@-------@-------------------------
        |  |                             |  |
        ---------------@^-0.25--+------------@^-0.5--+---------H--@------------
        |  |                             |  |
        -----------------------@^-0.125-------------@^-0.25------@^-0.5---H---
        The number of qubits can be arbitrary.
        """
        qubits = list(qubits)
        while len(qubits) > 0:
            q_head = qubits.pop(0)
            yield cirq.H(q_head)
            for i, qubit in enumerate(qubits):
                yield (cirq.CZ**(-1/2.0**(i+1)))(qubit, q_head)

    def run_estimate(unknown_gate, qnum, repeats):
        """Construct the following phase estimator circuit and execute simulations.
        ---------
        ---H---------------------@------| |---M--- [m4]:lowest bit
        | |                             |  |
        ---H---------------@-----+------| |---M--- [m3]
        | | | QFT_inv |
        ---H---------@-----+-----+------| |---M--- [m2]
        | | | | |
        ---H---@-----+-----+-----+------| |---M--- [m1]:highest bit
        | | | | |
        -------U-----U^2---U^4---U^8----------------------
        The measurement results M=[m1, m2,...] are translated to the estimated
        phase with the following formula:
        \[ \phi = m_1 \cdot \left( \frac{1}{2} \right) + m_2 \cdot \left( \frac{1}{2} \right)^2 + m_3 \cdot \left( \frac{1}{2} \right)^3 + \ldots \]
        """
qubits = [None] * qnum
for i in range(len(qubits)):
    qubits[i] = cirq.GridQubit(0, i)
ancilla = cirq.GridQubit(0, len(qubits))

circuit = cirq.Circuit.from_ops(
    cirq.H.on_each(qubits),
    [cirq.ControlledGate(unknown_gate**(2**i)).on(qubits[qnum-i-1], ancilla)
     for i in range(qnum)],
    QftInverse(qnum)(*qubits),
    cirq.measure(*qubits, key='phase'))
simulator = cirq.google.XmonSimulator()
result = simulator.run(circuit, repetitions=repeats)
return result

def experiment(qnum, repeats=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('Estimation with {}qubits.'.format(qnum))
print('Actual, Estimation (Raw binary)')[
errors = []
fold_func = lambda ms: ''.join(np.flip(ms, 0).astype(int).astype(str))
for phi in np.arange(0, 1, 0.1):
    result = run_estimate(example_gate(phi), qnum, repeats)
    hist = result.histogram(key='phase', fold_func=fold_func)
    estimate_bin = hist.most_common(1)[0][0]
    estimate = (sum([float(s)*0.5**(order+1)
                      for order, s in enumerate(estimate_bin)]))
    print('{:0.4f}, {:0.4f} ({})'.format(phi, estimate, estimate_bin))
    errors.append((phi-estimate)**2)
print('RMS Error: {:0.4f}

if __name__ == '__main__':
    main()}
2.8.7 Place on Bristlecone

```python
# 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)
    (6, 1) (6, 2)
        (7, 2) (7, 3)
            (8, 3) (8, 4)
                (9, 4) (9, 5)

Initial circuit:
0: _______ M('all')
    |
1: ___ x x M
    |
2: ___ x x M
    |
3: ___ x x M
    |
4: ___ x x M
    |
5: ___ x x M
    |
6: ___ x M
    |
7: ___ x M
    |
8: ___ x M
    |
9: ___ x M

Xmon circuit:
(5, 0): _______ Y^-0.5 @ Y^0.5 @ Y^-0.
    |<---------------------- M('all')-------------------->
    |__________
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |__________
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |        
    |__________
    |        
    |        
    |        
    |        
    |        
    |        
    |        
(continues on next page)
```
```python
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.from_ops(
        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()
```

2.8. Examples 55
### 2.8.8 Quantum Fourier Transform

```python
import numpy as np
import cirq

def main():
    # Create circuit
    qft_circuit = generate_2x2_grid_qft_circuit()
    print('Circuit:')
    print(qft_circuit)
    # Simulate and collect final_state
    simulator = cirq.google.XmonSimulator()
    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.

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, 0), cirq.GridQubit(1, 1)]

    circuit = cirq.Circuit.from_ops(
        cirq.H(a),
        _cz_and_swap(a, b, 0.5),
        (continues on next page)
_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()
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>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>Hardware constraints for validating circuits and schedules.</td>
</tr>
<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>QubitId</td>
<td>Identifies a qubit.</td>
</tr>
<tr>
<td>UnconstrainedDevice</td>
<td>A device that allows everything.</td>
</tr>
</tbody>
</table>

```
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.
```

Continued on next page
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>decompose_operation</code></td>
<td>Returns a device-valid decomposition for the given operation.</td>
</tr>
<tr>
<td><code>duration_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_moment</code></td>
<td>Raises an exception if a moment is not valid.</td>
</tr>
<tr>
<td><code>validate_operation</code></td>
<td>Raises an exception if an operation is not valid.</td>
</tr>
<tr>
<td><code>validate_schedule</code></td>
<td>Raises an exception if a schedule is not valid.</td>
</tr>
<tr>
<td><code>validate_scheduled_operation</code></td>
<td>Raises an exception if the scheduled operation is not valid.</td>
</tr>
</tbody>
</table>

**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**

```
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

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

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

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>

__init__ (row: int, col: int)
Initialize self. See help(type(self)) for accurate signature.
Methods

<table>
<thead>
<tr>
<th>from_proto_dict(proto_dict)</th>
<th>Proto dict must have ‘row’ and ‘col’ keys.</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_adjacent(other)</td>
<td>Determines if two qubits are adjacent qubits.</td>
</tr>
<tr>
<td>to_proto_dict()</td>
<td>Return the proto in dictionary form.</td>
</tr>
</tbody>
</table>

`cirq.GridQubit.from_proto_dict`

```python
static GridQubit.from_proto_dict(proto_dict: Dict) -> cirq.devices.grid_qubit.GridQubit
Proto dict must have ‘row’ and ‘col’ keys.
```

`cirq.GridQubit.is_adjacent`

```python
GridQubit.is_adjacent(other: cirq.ops.raw_types.QubitId) -> bool
Determines if two qubits are adjacent qubits.
```

`cirq.GridQubit.to_proto_dict`

```python
GridQubit.to_proto_dict() -> Dict
Return the proto in dictionary form.
```

`cirq.LineQubit`

```python
class cirq.LineQubit(x: int)
A qubit on a 1d lattice with nearest-neighbor connectivity.
__init__(x: int) -> None
    Initializes a line qubit at the given x coordinate.
```

Methods

<table>
<thead>
<tr>
<th>is_adjacent(other)</th>
<th>Determines if two qubits are adjacent line qubits.</th>
</tr>
</thead>
<tbody>
<tr>
<td>range(*range_args)</td>
<td>Returns a range of line qubits.</td>
</tr>
</tbody>
</table>

`cirq.LineQubit.is_adjacent`

```python
LineQubit.is_adjacent(other: cirq.ops.raw_types.QubitId) -> bool
Determines if two qubits are adjacent line qubits.
```

`cirq.LineQubit.range`

```python
static LineQubit.range(*range_args) -> List[cirq.line.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.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.

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.QubitId

class cirq.QubitId
   Identifies a qubit. Child classes implement specific types of qubits.

   The main criteria that a "qubit id" 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 \texttt{x}. \texttt{QubitId} implements all equality, comparison, and hashing methods via \texttt{._comparison_key}.

\begin{verbatim}
__init__(
    Initialize self. See help(type(self)) for accurate signature.
\end{verbatim}

cirq.UnconstrainedDevice

cirq.UnconstrainedDevice = cirq.UnconstrainedDevice
A device that allows everything.

\subsection*{3.1.2 Single Qubit Gates}

Unitary operations you can apply to a single qubit. Also measurement.

\begin{table}[h]
\begin{tabular}{|l|p{0.5\textwidth}|}
\hline
\texttt{H} & A Gate that performs a rotation around the X+Z axis of the Bloch sphere. \\
\texttt{HPowGate(*, exponent, float\[1.0, global\_shift\] = \texttt{1.0}, \texttt{global\_shift})} & A Gate that performs a rotation around the X+Z axis of the Bloch sphere. \\
\texttt{measure(*qubits, key, invert\_mask,...] = ()} & Returns a single MeasurementGate applied to all the given qubits. \\
\texttt{measure\_each(*qubits, key\_func, str\[\texttt{class}] = \texttt{<class>})} & Returns a list of operations individually measuring the given qubits. \\
\texttt{MeasurementGate(num\_qubits, key,...]} & A gate that measures qubits in the computational basis. \\
\texttt{PhasedXPowGate(*, phase\_exponent,...]} & A gate equivalent to the circuit \[Z^\texttt{--X}^\texttt{--Z}^\texttt{--p}\] . \\
\texttt{Rx(rads)} & Returns a gate with the matrix \[e^{-i X \text{rads} / 2} \] . \\
\texttt{Ry(rads)} & Returns a gate with the matrix \[e^{-i Y \text{rads} / 2} \] . \\
\texttt{Rz(rads)} & Returns a gate with the matrix \[e^{-i Z \text{rads} / 2} \] . \\
\texttt{S} & A gate that rotates around the Z axis of the Bloch sphere. \\
\texttt{SingleQubitMatrixGate(matrix)} & A 1-qubit gate defined by its matrix. \\
\texttt{T} & A gate that rotates around the Z axis of the Bloch sphere. \\
\texttt{TwoQubitMatrixGate(matrix)} & A 2-qubit gate defined only by its matrix. \\
\texttt{X} & \\
\texttt{XPowGate(*, exponent, float\[1.0, global\_shift\] = \texttt{1.0}, \texttt{global\_shift})} & A gate that rotates around the X axis of the Bloch sphere. \\
\texttt{YPowGate(*, exponent, float\[1.0, global\_shift\] = \texttt{1.0}, \texttt{global\_shift})} & A gate that rotates around the Y axis of the Bloch sphere. \\
\texttt{Z} & A gate that rotates around the Z axis of the Bloch sphere. \\
\texttt{ZPowGate(*, exponent, float\[1.0, global\_shift\] = \texttt{1.0}, \texttt{global\_shift})} & A gate that rotates around the Z axis of the Bloch sphere. \\
\end{tabular}
\end{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 \texttt{HPowGate(exponent=t)} is:

\begin{verbatim}

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

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.

cirq.HPowGate

class cirq.HPowGate(*, exponent: Union[cirq.value.symbol.Symbol, float] = 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 HPowGate(exponent=t) is:
\[
\begin{bmatrix}
g \cdot \left(c - i \cdot s / \sqrt{2}\right), & -i \cdot g \cdot s / \sqrt{2} \\
-i \cdot g \cdot s / \sqrt{2}, & g \cdot \left(c + i \cdot s / \sqrt{2}\right)
\end{bmatrix}
\]

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

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[cirq.value.symbol.Symbol, float] = 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:
   \[
   \theta
   \]

2. Shifting the angle by global_shift:
   \[
   \theta + s
   \]

3. Scaling the angle by exponent:
   \[
   (\theta + s) \times e
   \]

4. Converting from half turns to a complex number on the unit circle:
   \[
   \exp(i \times \pi \times (\theta + s) \times e)
   \]

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 apply this gate to each of the 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>
</tbody>
</table>

#### `cirq.HPowGate.num_qubits`

HPowGate.num_qubits() \(\rightarrow\) int

The number of qubits this gate acts on.

#### `cirq.HPowGate.on`

HPowGate.on(*qubits) \(\rightarrow\) gate_operation.GateOperation

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: Iterable[cirq.ops.raw_types.QubitId]) \(\rightarrow\) Union[cirq.ops.raw_types.Operation, Iterable[Any]]

Returns a list of operations apply this gate to each of the targets.

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

- **Returns** Operations applying this gate to the target qubits.

#### `cirq.HPowGate.validate_args`

HPowGate.validate_args(qubits)

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

Does no checks by default. Child classes can override.

- **Parameters** qubits – The collection of qubits to potentially apply the gate to.
**Throws**: ValueError: The gate can’t be applied to the qubits.

**Attributes**

```
exponent
```

```
cirq.HPowGate.exponent
```

```
cirq.measure
```

```py

cirq.measure(*qubits, key: Optional[str] = None, invert_mask: Tuple[bool, ...] = ()) ->
cirq.ops.gate_operation.GateOperation
```

Returns a single MeasurementGate applied to all the given qubits.

The qubits are measured in the computational basis.

**Parameters**

- `*qubits` – 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 QubitId.

```
cirq.measure_each
```

```py

cirq.measure_each(*qubits, key_func: Callable[cirq.ops.raw_types.QubitId, str] = <class 'str'>) ->
List[cirq.ops.gate_operation.GateOperation]
```

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
```

```py

class cirq.MeasurementGate(num_qubits: int, key: str = '', invert_mask: Tuple[bool, ...] = ())
```

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: int, key: str = "", invert_mask: Tuple[bool, ...] = ()) → 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.

**Raises** ValueError if the length of `invert_mask` is greater than `num_qubits`.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_measurement(op, cirq.ops.raw_types.Operation)</code></td>
<td>The number of qubits this gate acts on.</td>
</tr>
<tr>
<td><code>num_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>
</tbody>
</table>

**cirquit.MeasurementGate.is_measurement**

```python
static MeasurementGate.is_measurement(op: Union[cirq.ops.raw_types.Gate, cirq.ops.raw_types.Operation]) → bool
```

**cirquit.MeasurementGate.num_qubits**

```python
MeasurementGate.num_qubits() → int
```

The number of qubits this gate acts on.

**cirquit.MeasurementGate.on**

```python
MeasurementGate.on(*qubits) → gate_operation.GateOperation
```

Returns an application of this gate to the given qubits.

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

**cirquit.MeasurementGate.validate_args**

```python
MeasurementGate.validate_args(qubits)
```

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

Does no checks by default. Child classes can override.
**Parameters qubits** – The collection of qubits to potentially apply the gate to.

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

cirq.MeasurementGate.with_bits_flipped

`MeasurementGate.with_bits_flipped(*bit_positions) → cirq.ops.common_gates.MeasurementGate`

Toggles whether or not the measurement inverts various outputs.

cirq.PhasedXPowGate

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

A gate equivalent to the circuit \(-Z^{-p}X^tZ^p\).

___init___(*, phase_exponent: Union[float, cirq.value.symbol.Symbol], exponent: Union[float, cirq.value.symbol.Symbol] = 1.0, global_shift: float = 0.0) → None

**Parameters**

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

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 apply this gate to each of the 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>
</tbody>
</table>

cirq.PhasedXPowGate.num_qubits

PhasedXPowGate.num_qubits() → int

The number of qubits this gate acts on.

cirq.PhasedXPowGate.on

PhasedXPowGate.on(*qubits) → gate_operation.GateOperation

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.\texttt{on\_each}(\texttt{targets: Iterable[cirq.ops.raw\_types.QubitId]} \rightarrow \texttt{Union[cirq.ops.raw\_types.Operation, Iterable[Any]]})

Returns a list of operations apply this gate to each of the targets.

- **Parameters** \texttt{targets} – The qubits to apply this gate to.
- **Returns** Operations applying this gate to the target qubits.

cirq.PhasedXPowGate.validate_args

PhasedXPowGate.\texttt{validate\_args}(\texttt{qubits})

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

Does no checks by default. Child classes can override.

- **Parameters** \texttt{qubits} – The collection of qubits to potentially apply the gate to.
- **Throws**: ValueError: The gate can’t be applied to the qubits.

**Attributes**

- \texttt{exponent}  
The exponent on the central X gate conjugated by the Z gates.
- \texttt{phase\_exponent}  
The exponent on the Z gates conjugating the X gate.

cirq.PhasedXPowGate.exponent

PhasedXPowGate.\texttt{exponent}

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

cirq.PhasedXPowGate.phase_exponent

PhasedXPowGate.\texttt{phase\_exponent}

The exponent on the Z gates conjugating the X gate.

cirq.Rx

cirq.\texttt{Rx}(\texttt{rads: float}) \rightarrow \texttt{cirq.ops.common\_gates.XPowGate}

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

cirq.Ry

cirq.\texttt{Ry}(\texttt{rads: float}) \rightarrow \texttt{cirq.ops.common\_gates.YPowGate}

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

cirq.Rz(rads: float) → cirq.ops.common_gates.ZPowGate
Returns a gate with the matrix $e^{-iZ \text{rads} / 2}$.

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:

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

where:

$$
g = \exp(i \cdot \pi \cdot t).
$$

Note in particular that this gate has a global phase factor of $e^{i \cdot \pi \cdot t/2}$ 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 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

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

- **on(*qubits)**
  Returns an application of this gate to the given qubits.
Table 12 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>on_each</code> (targets)</td>
<td>Returns a list of operations apply this gate to each of the targets.</td>
</tr>
<tr>
<td><code>validate_args</code> (qubits)</td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
</tbody>
</table>

**`cirq.SingleQubitMatrixGate.num_qubits`**

`SingleQubitMatrixGate.num_qubits() → int`

The number of qubits this gate acts on.

**`cirq.SingleQubitMatrixGate.on`**

`SingleQubitMatrixGate.on (*qubits) → gate_operation.GateOperation`

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: Iterable[cirq.ops.raw_types.QubitId]) → Union[cirq.ops.raw_types.Operation, Iterable[Any]]`

Returns a list of operations apply this gate to each of the targets.

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

**Returns** Operations applying this gate to the target qubits.

**`cirq.SingleQubitMatrixGate.validate_args`**

`SingleQubitMatrixGate.validate_args (qubits)`

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

Does no checks by default. Child classes can override.

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

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

**`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:

```
[[1, 0],
 [0, g]]
```

where:

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

#### `cirq.TwoQubitMatrixGate.num_qubits`

```
TwoQubitMatrixGate.num_qubits() → int
```

The number of qubits this gate acts on.

#### `cirq.TwoQubitMatrixGate.on`

```
TwoQubitMatrixGate.on(*qubits) → gate_operation.GateOperation
```

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.
Does no checks by default. Child classes can override.

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

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

cirq.X
cirq.X = cirq.X
cirq.XPowGate
class cirq.XPowGate(*, exponent: Union[cirq.value.symbol.Symbol, float] = 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:

\[
\begin{bmatrix}
g \cdot c, & -i \cdot g \cdot s, \\
-i \cdot g \cdot s, & g \cdot c
\end{bmatrix}
\]

where:

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

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

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

__init__(*, exponent: Union[cirq.value.symbol.Symbol, float] = 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:

   \[ \theta \]

2. Shifting the angle by global_shift:

   \[ \theta + s \]

3. Scaling the angle by exponent:

   \[(\theta + s) \cdot e\]
4. Converting from half turns to a complex number on the unit circle:

\[ \exp(i \times \pi \times (\theta + s) \times e) \]

**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**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 apply this gate to each of the 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>
</tbody>
</table>

**cirq.XPowGate.num_qubits**

`XPowGate.num_qubits() \rightarrow int`

The number of qubits this gate acts on.

**cirq.XPowGate.on**

`XPowGate.on(*qubits) \rightarrow gate_operation.GateOperation`

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: Iterable[cirq.ops.raw_types.QubitId]) \rightarrow Union[cirq.ops.raw_types.Operation, Iterable[Any]]`

Returns a list of operations apply this gate to each of the targets.

**Parameters**

- `targets` – The qubits to apply this gate to.

**Returns** Operations applying this gate to the target qubits.
**cirq.XPowGate.validate_args**

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

Does no checks by default. Child classes can override.

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

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

**Attributes**

- **exponent**
- **phase_exponent**

**cirq.XPowGate.exponent**

XPowGate.exponent

**cirq.XPowGate.phase_exponent**

XPowGate.phase_exponent

**cirq.Y**

cirq.Y = cirq.Y

**cirq.YPowGate**

class cirq.YPowGate(*, exponent: Union[cirq.value.symbol.Symbol, float] = 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:

$$[[g \cdot c, g \cdot s],
[-g \cdot s, g \cdot c]]$$

where:

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

Note in particular that this gate has a global phase factor of \(e^{i \cdot \pi \cdot t/2}\) 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.
cirq.Y, the Pauli Y gate, is an instance of this gate at exponent=1.

```python
@cirq.value.deprecated("cirq.Y is deprecated. Use cirq.PauliY instead.")
@cirq.value.deprecated("Use cirq.pauli_gate("Y") instead.")
cirq.Y = cirq.PauliY
```

__init__(*, exponent: Union[cirq.value.symbol.Symbol, float] = 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:
   \[ \theta \]

2. Shifting the angle by global_shift:
   \[ \theta + s \]

3. Scaling the angle by exponent:
   \[ (\theta + s) \times e \]

4. Converting from half turns to a complex number on the unit circle:
   \[ \exp(i \times \pi \times (\theta + s) \times e) \]

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 \(e^{i \pi \times \text{exponent}}\) 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

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

- **on_qubits**
  Returns an application of this gate to the given qubits.

- **on_each**
  Returns a list of operations apply this gate to each of the targets.

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

```
cirq.YPowGate.num_qubits
```

YPowGate.num_qubits() → int

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

YPowGate.on(*qubits) → gate_operation.GateOperation
Returns an application of this gate to the given qubits.

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

cirq.YPowGate.on_each

YPowGate.on_each(targets: Iterable[cirq.ops.raw_types.QubitId]) → Union[cirq.ops.raw_types.Operation, Iterable[Any]]
Returns a list of operations apply this gate to each of the targets.

Parameters targets – The qubits to apply this gate to.

Returns Operations applying this gate to the target qubits.

cirq.YPowGate.validate_args

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

Does no checks by default. Child classes can override.

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

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

Attributes

---

| exponent
| phase_exponent

---

cirq.YPowGate.exponent

YPowGate.exponent

cirq.YPowGate.phase_exponent

YPowGate.phase_exponent

cirq.Z
cirq.Z = cirq.Z

cirq.ZPowGate

class cirq.ZPowGate(*, exponent: Union[cirq.value.symbol.Symbol, float] = 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:

$$
g = \exp(i \cdot \pi \cdot t).
$$

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 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.

```python
__init__(*, exponent: Union[cirq.value.symbol.Symbol, float] = 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:

$$
\theta
$$

2. Shifting the angle by `global_shift`:

$$
\theta + s
$$

3. Scaling the angle by `exponent`:

$$
(\theta + s) \times e
$$

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

$$
\exp(i \times \pi \times (\theta + s) \times e)
$$

**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 $e^{i \pi \cdot exponent}$ 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:

$$
\exp(i \times \pi \times global_shift \times exponent)
$$

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>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 apply this gate to each of the 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>
</tbody>
</table>

\[\text{cirq.ZPowGate.num_qubits} \]

\[\text{ZPowGate.num_qubits} \rightarrow \text{int}\]

The number of qubits this gate acts on.

\[\text{cirq.ZPowGate.on} \]

\[\text{ZPowGate.on}(*\text{qubits}) \rightarrow \text{gate_operation.GateOperation}\]

Returns an application of this gate to the given qubits.

Parameters

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

\[\text{cirq.ZPowGate.on_each} \]

\[\text{ZPowGate.on_each}(targets: \text{Iterable[cirq.ops.raw_types.QubitId]} \rightarrow \text{Union[cirq.ops.raw_types.Operation, Iterable[Any]]}\]

Returns a list of operations apply this gate to each of the targets.

Parameters

\[\text{targets} – \text{The qubits to apply this gate to.} \]

Returns

Operations applying this gate to the target qubits.

\[\text{cirq.ZPowGate.validate_args} \]

\[\text{ZPowGate.validate_args}(qubits)\]

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

Does no checks by default. Child classes can override.

Parameters

\[\text{qubits} – \text{The collection of qubits to potentially apply the gate to.} \]

Throws

\[\text{ValueError: The gate can’t be applied to the qubits.} \]

Attributes

\[\text{exponent} \]

\[\text{cirq.ZPowGate.exponent} \]

\[\text{ZPowGate.exponent} \]
3.1.3 Two Qubit 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, float] = 1.0,...)</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, float] = 1.0,...)</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, float] = 1.0,...)</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, float] = 1.0,...)</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, float] = 1.0,...)</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, float] = 1.0,...)</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, float] = 1.0,...)</td>
<td>The Z-parity gate, possibly raised to a power.</td>
</tr>
</tbody>
</table>

**cirq.CNOT**

```python
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:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & g·c & -i·g·s \\
0 & 0 & -i·g·s & g·c
\end{bmatrix}
\]

where:

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

**cirq.CNOT**, the controlled NOT gate, is an instance of this gate at exponent=1.
cirq.CNotPowGate

class cirq.CNotPowGate(*, exponent: Union[cirq.value.symbol.Symbol, float] = 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:

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

cirq.CNOT, the controlled NOT gate, is an instance of this gate at exponent=1.

__init__(*, exponent: Union[cirq.value.symbol.Symbol, float] = 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:

   \[\theta\]

2. Shifting the angle by global_shift:

   \[\theta + s\]

3. Scaling the angle by exponent:

   \[(\theta + s) \cdot e\]

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

   \[\exp(i \cdot \pi \cdot (\theta + s) \cdot e)\]
• **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:

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

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>num_qubits()</code></td>
<td>The number of qubits this gate acts on.</td>
</tr>
<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>
</tbody>
</table>

#### `cirq.CNotPowGate.num_qubits`

CNotPowGate.`num_qubits()` → int

The number of qubits this gate acts on.

#### `cirq.CNotPowGate.on`

CNotPowGate.`on(*args, **kwargs)` → cirq.ops.gate_operation.GateOperation

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`

CNotPowGate.`validate_args(qubits)`

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

Does no checks by default. Child classes can override.

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

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

### Attributes

- **exponent**

#### `cirq.CNotPowGate.exponent`

CNotPowGate.`exponent`
cirq.CZ

\[ cirq.CZ = cirq.CZ \]

A gate that applies a phase to the \|11\rangle state of two qubits.

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

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

where:

\[ g = \exp(i \cdot \frac{\pi \cdot t}{2}). \]

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

cirq.CZPowGate

\textbf{class} \hspace{1em} cirq.CZPowGate(*, exponent: \text{Union[cirq.value.symbol.Symbol, float]} = 1.0, global_shift: \text{float} = 0.0)

A gate that applies a phase to the \|11\rangle state of two qubits.

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

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

where:

\[ g = \exp(i \cdot \frac{\pi \cdot t}{2}). \]

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

\textbf{__init__}(*, exponent: \text{Union[cirq.value.symbol.Symbol, float]} = 1.0, global_shift: \text{float} = 0.0) \rightarrow \text{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:

\[ \theta \]

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

\[
(\theta + s) \times e
\]

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

\[
\exp(i \times \pi \times (\theta + s) \times e)
\]

**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**

- **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: int)**
  Returns a key that differs between non-interchangeable qubits.

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

```python
CZPowGate.num_qubits()
CZPowGate.on(*qubits)
CZPowGate.qubit_index_to_equivalence_group_key(index: int)
CZPowGate.validate_args(qubits)
```
cirq.CZPowGate.validate_args

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

Does no checks by default. Child classes can override.

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

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

Attributes

exponent
cirq.CZPowGate.exponent

CZPowGate.exponent
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:

\[
\text{ISWAP}^\ast \exp\left(\frac{i \pi t (XX + YY)}{4}\right)
\]

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:

\[
c = \cos\left(\frac{\pi t}{2}\right) \\
s = \sin\left(\frac{\pi t}{2}\right)
\]

cirq.ISWAP, the swap gate that applies -i to the |01> and |10> states,
is an instance of this gate at exponent=1.
cirq.ISwapPowGate

class cirq.ISwapPowGate(*, exponent: Union[cirq.value.symbol.Symbol, float] = 1.0, global_shift: 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:

\[
\text{ISWAP}^t \exp(+i \frac{\pi}{2} (XX + YY) / 4)
\]

which is given by the matrix:

\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & c & i\cdot s & 0 \\
0 & i\cdot s & c & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

where:

\[
c = \cos(\pi \cdot t / 2) \\
s = \sin(\pi \cdot t / 2)
\]

cirq.ISWAP, the swap gate that applies -i to the |01> and |10> states, is an instance of this gate at exponent=1.

__init__(*, exponent: Union[cirq.value.symbol.Symbol, float] = 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:
   \[
   \theta
   \]

2. Shifting the angle by global_shift:
   \[
   \theta + s
   \]

3. Scaling the angle by exponent:
   \[
   (\theta + s) \ast e
   \]

4. Converting from half turns to a complex number on the unit circle:
   \[
   \exp(i \ast \pi \ast (\theta + s) \ast e)
   \]
• **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**\text{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:

\[ \exp(i \pi \text{global}_\text{shift} \times \text{exponent}) \]

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

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

**cirq.ISwapPowGate.num_qubits**

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

**cirq.ISwapPowGate.on**

`ISwapPowGate.on(*qubits) → gate_operation.GateOperation`  
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**

`ISwapPowGate.qubit_index_to_equivalence_group_key(index: int) → int`  
Returns a key that differs between non-interchangeable qubits.

**cirq.ISwapPowGate.validate_args**

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

Does no checks by default. Child classes can override.

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

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

**exponent**

cirq.ISwapPowGate.exponent

**cirq.MS**

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:

\[
\begin{bmatrix}
\cos(t) & 0 & 0 & -i \sin(t) \\
0 & \cos(t) & -i \sin(t) & 0 \\
0 & -i \sin(t) & \cos(t) & 0 \\
-i \sin(t) & 0 & 0 & \cos(t)
\end{bmatrix}
\]

**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.

SwapPowGate()**t = SwapPowGate(exponent=t) 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:

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

**cirq.SWAP**, the swap gate, is an instance of this gate at exponent=1.
cirq.SwapPowGate

class cirq.SwapPowGate(*, exponent: Union[cirq.value.symbol.Symbol, float] = 1.0, global_shift: float = 0.0)
The SWAP gate, possibly raised to a power. Exchanges qubits.

SwapPowGate()**t = SwapPowGate(exponent=t) 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:

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

cirq.SWAP, the swap gate, is an instance of this gate at exponent=1.

__init__(*, exponent: Union[cirq.value.symbol.Symbol, float] = 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:
   \[
   \theta
   \]

2. Shifting the angle by global_shift:
   \[
   \theta + s
   \]

3. Scaling the angle by exponent:
   \[
   (\theta + s) \cdot e
   \]

4. Converting from half turns to a complex number on the unit circle:
   \[
   \exp(i \cdot \pi \cdot (\theta + s) \cdot e)
   \]

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**l is applied will gain a relative phase of e^{i \pi \cdot exponent} when gate**exponent is applied (relative to eigenvectors unaffected by gate**l).

- **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, `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>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>
</tbody>
</table>

#### `cirq.SwapPowGate.num_qubits`

`SwapPowGate.num_qubits() → int`

The number of qubits this gate acts on.

#### `cirq.SwapPowGate.on`

`SwapPowGate.on(*qubits) → gate_operation.GateOperation`

Returns an application of this gate to the given qubits.

**Parameters**

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

#### `cirq.SwapPowGate.qubit_index_to_equivalence_group_key`

`SwapPowGate.qubit_index_to_equivalence_group_key(index: int) → int`

Returns a key that differs between non-interchangeable qubits.

#### `cirq.SwapPowGate.validate_args`

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

Does no checks by default. Child classes can override.

**Parameters**

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

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

#### Attributes

**exponent**

`cirq.SwapPowGate.exponent`

`SwapPowGate.exponent`
cirq.XX

cirq.XX = cirq.XX

The X-parity gate, possibly raised to a power.

At exponent=1, this gate implements the following unitary:

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

See also: cirq.MS (the Mølmer–Sørensen gate), which is implemented via this class.

cirq.XXPowGate

class cirq.XXPowGate(*, exponent: Union[cirq.value.symbol.Symbol, float] = 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:

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

See also: cirq.MS (the Mølmer–Sørensen gate), which is implemented via this class.

__init__(*, exponent: Union[cirq.value.symbol.Symbol, float] = 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:

   \[ \theta \]

2. Shifting the angle by global_shift:

   \[ \theta + s \]

3. Scaling the angle by exponent:

   \[(\theta + s) \times e\]

4. Converting from half turns to a complex number on the unit circle:
\[ \exp(i \cdot \pi \cdot (\theta + s) \cdot e) \]

**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 \( e^{i \cdot \pi \cdot \text{exponent}} \) when \( gate^{**\text{exponent}} \) 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:
  \[ \exp(i \cdot \pi \cdot \text{global_shift} \cdot \text{exponent}) \]
  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><strong>num_qubits()</strong></td>
<td>The number of qubits this gate acts on.</td>
</tr>
<tr>
<td>*<em>on(<em>qubits)</em></em></td>
<td>Returns an application of this gate to the given qubits.</td>
</tr>
<tr>
<td><strong>qubit_index_to_equivalence_group_key</strong></td>
<td>Returns a key that differs between non-interchangeable qubits.</td>
</tr>
<tr>
<td><strong>validate_args(qubits)</strong></td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
</tbody>
</table>

**Methods**

- **cirq.XXPowGate.num_qubits**

  \(XXPowGate.num_qubits() \to \text{int} \)
  The number of qubits this gate acts on.

- **cirq.XXPowGate.on**

  \(XXPowGate.on(*qubits) \to \text{gate_operation.GateOperation} \)
  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**

  \(XXPowGate.qubit_index_to_equivalence_group_key(index: \text{int}) \to \text{int} \)
  Returns a key that differs between non-interchangeable qubits.

- **cirq.XXPowGate.validate_args**

  \(XXPowGate.validate_args(qubits) \)
  Checks if this gate can be applied to the given qubits.
  Does no checks by default. Child classes can override.

  **Parameters** qubits – The collection of qubits to potentially apply the gate to.
Throws: ValueError: The gate can’t be applied to the qubits.

Attributes

```
cirq.XXPowGate.exponent
```

```
x = cirq.XXPowGate().exponent
```

```
cirq.YYPowGate
```

```
cirq.YYPowGate(\*, exponent: Union[cirq.value.symbol.Symbol, float] = 1.0, global_shift: float = 0.0)
```

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:
   
   \[
   \theta
   \]

2. Shifting the angle by `global_shift`:
   
   \[
   \theta + s
   \]

3. Scaling the angle by `exponent`:
   
   \[
   (\theta + s) \times e
   \]

4. Converting from half turns to a complex number on the unit circle:
   
   \[
   \exp(i \times \pi \times (\theta + s) \times e)
   \]

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**I is applied will gain a relative phase of e^{i pi 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 * 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**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

**cirq.YYPowGate.num_qubits**

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

**cirq.YYPowGate.on**

YYPowGate.`on(*qubits)` → gate_operation.GateOperation
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)`
Checks if this gate can be applied to the given qubits.

Does no checks by default. Child classes can override.

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

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

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>exponent</code></td>
<td></td>
</tr>
</tbody>
</table>

3.1. API Reference
cirq.YYPowGate.exponent

YYPowGate.exponent

cirq.ZZ

cirq.ZZ = cirq.ZZ
The Z-parity gate, possibly raised to a power.

The ZZ**t gate implements the following unitary:

(\text{ZZ})^t = \begin{bmatrix}
1 & \ldots \\
. & w & . \\
. & . & w \\
. & . & . & 1
\end{bmatrix}

where w = e^{i \pi t} and '.' means '0'.

cirq.ZZPowGate

class cirq.ZZPowGate(*, exponent: Union[cirq.value.symbol.Symbol, float] = 1.0, global_shift: float = 0.0)
The Z-parity gate, possibly raised to a power.

The ZZ**t gate implements the following unitary:

(\text{ZZ})^t = \begin{bmatrix}
1 & \ldots \\
. & w & . \\
. & . & w \\
. & . & . & 1
\end{bmatrix}

where w = e^{i \pi t} and '.' means '0'.

__init__(*\text{, exponent: Union[cirq.value.symbol.Symbol, float] = 1.0, global_shift: float = 0.0}) → None

Initialize 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:

   $\theta$

2. Shifting the angle by global_shift:

   $\theta + s$

3. Scaling the angle by exponent:

   $(\theta + s) * e$

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

   $\exp(i * \pi * (\theta + s) * e)$
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:
  \[ \exp(i \cdot \pi \cdot \text{global_shift} \cdot \text{exponent}) \]
  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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

**cirq.ZZPowGate.num_qubits**

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

  The number of qubits this gate acts on.

**cirq.ZZPowGate.on**

ZZPowGate.on(*qubits) \rightarrow \text{gate_operation.GateOperation}

  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**

ZZPowGate.qubit_index_to_equivalence_group_key(index: int) \rightarrow \text{int}

  Returns a key that differs between non-interchangeable qubits.

**cirq.ZZPowGate.validate_args**

ZZPowGate.validate_args(qubits)

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

  **Does no checks by default. Child classes can override.**

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

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

.. attribute:: exponent

   .. attribute:: cirq.ZZPowGate.exponent

### 3.1.4 Three Qubit 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**

```python
@U(cirq.CCX)  # A Toffoli (doubly-controlled-NOT) that can be raised to a power.
```

The matrix of \(\text{CCX}^t\) is an 8x8 identity except the bottom right 2x2 area is the matrix of \(\text{X}^t\).

**cirq.CCXPowGate**

```python
class cirq.CCXPowGate(*, exponent: Union[cirq.value.symbol.Symbol, float] = 1.0, global_shift: float = 0.0)  # A Toffoli (doubly-controlled-NOT) that can be raised to a power.
```

The matrix of \(\text{CCX}^t\) is an 8x8 identity except the bottom right 2x2 area is the matrix of \(\text{X}^t\).

```python
__init__(*, exponent: Union[cirq.value.symbol.Symbol, float] = 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:
\[ \theta \]

2. Shifting the angle by global\_shift:
\[ \theta + s \]

3. Scaling the angle by exponent:
\[ (\theta + s) \times e \]

4. Converting from half turns to a complex number on the unit circle:
\[ \exp(i \times \pi \times (\theta + s) \times e) \]

**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, \(\text{cirq.X**t}\) uses a global\_shift of 0 but \(\text{cirq.Rx(t)}\) uses a 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><strong>num_qubits()</strong></td>
<td>The number of qubits this gate acts on.</td>
</tr>
<tr>
<td>*<em>on(<em>qubits)</em></em></td>
<td>Returns an application of this gate to the given qubits.</td>
</tr>
<tr>
<td><strong>qubit_index_to_equivalence_group_key(index)</strong></td>
<td>Returns a key that differs between non-interchangeable qubits.</td>
</tr>
<tr>
<td><strong>validate_args(qubits)</strong></td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
</tbody>
</table>

**cirq.CCXPowGate.num\_qubits**

CCXPowGate.num\_qubits() → int
The number of qubits this gate acts on.

**cirq.CCXPowGate.on**

CCXPowGate.on(*qubits) → gate\_operation.GateOperation
Returns an application of this gate to the given qubits.

**Parameters**

- ***qubits** – The collection of qubits to potentially apply the gate to.
```python
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)
Checks if this gate can be applied to the given qubits.
Does no checks by default. Child classes can override.

Parameters qubits -- The collection of qubits to potentially apply the gate to.

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

Attributes

exponent
```
cirq.CCXPowGate.exponent

CCXPowGate.exponent

cirq.CCZ

cirq.CCZ = cirq.CCZ
A doubly-controlled-Z that can be raised to a power.
The matrix of \( \text{CCZ}^{*t} \) is \( \text{diag}(1, 1, 1, 1, 1, 1, 1, \exp(i \pi t)) \).

cirq.CCZPowGate

class cirq.CCZPowGate(*, exponent: Union[cirq.value.symbol.Symbol, float] = 1.0, global_shift: float = 0.0)
A doubly-controlled-Z that can be raised to a power.
The matrix of \( \text{CCZ}^{*t} \) is \( \text{diag}(1, 1, 1, 1, 1, 1, 1, \exp(i \pi t)) \).

__init__(*, exponent: Union[cirq.value.symbol.Symbol, float] = 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:

\[
\theta
\]

2. Shifting the angle by global_shift:
3. Scaling the angle by exponent:

\[(\theta + s) \times e\]

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

\[\exp(i \pi \times (\theta + s) \times e)\]

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 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:

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

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

- **num_qubits()**
  Returns 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.

\texttt{cirq.CCZPowGate.num_qubits}

CCZPowGate.num_qubits() \to int

The number of qubits this gate acts on.

\texttt{cirq.CCZPowGate.on}

CCZPowGate.on(*qubits) \to gate_operation.GateOperation

Returns an application of this gate to the given qubits.

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

\texttt{cirq.CCZPowGate.qubit_index_to_equivalence_group_key}

CCZPowGate.qubit_index_to_equivalence_group_key(index: int) \to int

Returns a key that differs between non-interchangeable qubits.
cirq.CCZPowGate.validate_args

```
CCZPowGate.validate_args(qubits)
Checks if this gate can be applied to the given qubits.
Does no checks by default. Child classes can override.

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

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

Attributes

```

```


cirq.CCZPowGate.exponent

```

```
CCZPowGate.exponent
```

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

```
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.
```

cirq.CSwapGate.num_qubits

cSwapGate.num_qubits() \rightarrow \text{int}
The number of qubits this gate acts on.
### `cirq.CSwapGate.on`

CSwapGate.on(*qubits) → gate_operation.GateOperation

Returns an application of this gate to the given qubits.

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

### `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)

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

Does no checks by default. Child classes can override.

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

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

---

### `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 \(\text{CCX}^t\) is an 8x8 identity except the bottom right 2x2 area is the matrix of \(X^t\).

---

#### 3.1.5 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>ControlledGate(sub_gate)</td>
<td>Augments existing gates with a control qubit.</td>
</tr>
<tr>
<td>EigenGate(*, exponent, float = 1.0, ...)</td>
<td>A gate with a known eigendecomposition.</td>
</tr>
<tr>
<td>Gate</td>
<td>An operation type that can be applied to a collection of qubits.</td>
</tr>
<tr>
<td>GateOperation(gate, qubits)</td>
<td>An application of a gate to a sequence of qubits.</td>
</tr>
<tr>
<td>InterchangeableQubitsGate</td>
<td>Indicates operations should be equal under some qubit permutations.</td>
</tr>
<tr>
<td>Operation</td>
<td>An effect applied to a collection of qubits.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 41 – continued from previous page

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReversibleCompositeGate</td>
<td>A composite gate that gets decomposed into reversible gates.</td>
</tr>
<tr>
<td>SingleQubitGate</td>
<td>A gate that must be applied to exactly one qubit.</td>
</tr>
<tr>
<td>ThreeQubitGate</td>
<td>A gate that must be applied to exactly three qubits.</td>
</tr>
<tr>
<td>TwoQubitGate</td>
<td>A gate that must be applied to exactly two qubits.</td>
</tr>
</tbody>
</table>

**cirq.ControlledGate**

```python
class cirq.ControlledGate(sub_gate: cirq.ops.raw_types.Gate)
    Augments existing gates with a control qubit.

    __init__(sub_gate: cirq.ops.raw_types.Gate) → None
        Initializes the controlled gate.

        Parameters sub_gate – The gate to add a control qubit to.
```

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</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>validate_args(qubits)</td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
</tbody>
</table>

**cirq.ControlledGate.num_qubits**

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

**cirq.ControlledGate.on**

```
calledGate.on(*qubits) → gate_operation.GateOperation
    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**

```
calledGate.validate_args(qubits) → None
    Checks if this gate can be applied to the given qubits.

    Does no checks by default. Child classes can override.

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

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

class cirq.EigenGate(*, exponent: Union[cirq.value.symbol.Symbol, float] = 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.

__init__(*, exponent: Union[cirq.value.symbol.Symbol, float] = 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:

   \[ \theta \]

2. Shifting the angle by global_shift:

   \[ \theta + s \]

3. Scaling the angle by exponent:

   \[ (\theta + s) \times e \]

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

   \[ \exp(i \times \pi \times (\theta + s) \times e) \]

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**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**l**).

- **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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

```python
cirq.EigenGate.num_qubits
```

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

```python
cirq.EigenGate.on
```

EigenGate.on(*qubits) → gate_operation.GateOperation
Returns an application of this gate to the given qubits.

Parameters

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

```python
cirq.EigenGate.validate_args
```

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

Does no checks by default. Child classes can override.

Parameters

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

Throws:

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

Attributes

```python
cirq.EigenGate.exponent
```

EigenGate.exponent

```python
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.

```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>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>
</tbody>
</table>

`cirq.Gate.num_qubits`

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

`cirq.Gate.on`

Gate.on(*qubits) \(\rightarrow\) gate_operation.GateOperation
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.QubitId]) \(\rightarrow\) None
Checks if this gate can be applied to the given qubits.

Does no checks by default. Child classes can override.

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

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

`cirq.GateOperation`

class cirq.GateOperation(gate: cirq.ops.raw_types.Gate, qubits: Sequence[cirq.ops.raw_types.QubitId])
An application of a gate to a sequence of qubits.

__init__(gate: cirq.ops.raw_types.Gate, qubits: Sequence[cirq.ops.raw_types.QubitId]) \(\rightarrow\) None

Parameters
- `gate` – The gate to apply.
- `qubits` – The qubits to operate on.
Methods

- `transform_qubits(func, ...)` Returns the same operation, but with different qubits.
- `with_gate(new_gate)`
- `with_qubits(*new_qubits)`

`cirq.GateOperation.transform_qubits`

GateOperation.transform_qubits(func: Callable[cirq.ops.raw_types.QubitId, cirq.ops.raw_types.QubitId]) → TSelf.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.with_gate`

GateOperation.with_gate(new_gate: cirq.ops.raw_types.Gate) → cirq.ops.gate_operation.GateOperation

`cirq.GateOperation.with_qubits`

GateOperation.with_qubits(*new_qubits) → cirq.ops.gate_operation.GateOperation

Attributes

- `gate` The gate applied by the operation.
- `qubits` The qubits targeted by the operation.

`cirq.GateOperation.gate`

GateOperation.gate

The gate applied by the operation.

`cirq.GateOperation.qubits`

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

qubit_index_to_equivalence_group_key(index)
Returns a key that differs between non-interchangeable qubits.

cirq.InterchangeableQubitsGate.qubit_index_to_equivalence_group_key
InterchangeableQubitsGate.qubit_index_to_equivalence_group_key(index: int) → int
Returns a key that differs between non-interchangeable qubits.

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

transform_qubits(func, ...)
Returns the same operation, but with different qubits.

with_qubits(*new_qubits)

cirq.Operation.transform_qubits

Operation.transform_qubits(func: Callable[cirq.ops.raw_types.QubitId, cirq.ops.raw_types.QubitId]) → TSelf_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.with_qubits

Operation.with_qubits(*new_qubits) → TSelf_Operation
Attributes

qubits

cirq.Operation.qubits

Operation.qubits

cirq.ReversibleCompositeGate
class cirq.ReversibleCompositeGate
A composite gate that gets decomposed into reversible gates.
__init__()
Initialize self. See help(type(self)) for accurate signature.

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

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 apply this gate to each of the targets.
validate_args(qubits) Checks if this gate can be applied to the given qubits.

cirq.SingleQubitGate.num_qubits

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

cirq.SingleQubitGate.on

SingleQubitGate.on(*qubits) → gate_operation.GateOperation
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: Iterable[cirq.ops.raw_types.QubitId]) → Union[cirq.ops.raw_types.Operation, Iterable[Any]]
Returns a list of operations applying this gate to each of the targets.

Parameters

  targets – The qubits to apply this gate to.

  Returns 
    Operations applying this gate to the target qubits.

cirq.SingleQubitGate.validate_args

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

Does no checks by default. Child classes can override.

Parameters

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

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

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

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.

cirq.ThreeQubitGate.num_qubits

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

cirq.ThreeQubitGate.on

ThreeQubitGate.on(*qubits) → gate_operation.GateOperation
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)
    Checks if this gate can be applied to the given qubits.
    Does no checks by default. Child classes can override.
    
    Parameters qubits – The collection of qubits to potentially apply the gate to.
    
    Throws: ValueError: The gate can’t be applied to the qubits.

cirq.TwoQubitGate

class cirq.TwoQubitGate
    A gate that must be applied to exactly two qubits.
    
    __init__()
        Initialize self. See help(type(self)) for accurate signature.

    Methods

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

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

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

    cirq.TwoQubitGate.num_qubits

    TwoQubitGate.num_qubits() -> int
        The number of qubits this gate acts on.

    cirq.TwoQubitGate.on

    TwoQubitGate.on(*qubits) -> gate_operation.GateOperation
        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)
        Checks if this gate can be applied to the given qubits.
        Does no checks by default. Child classes can override.

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

        Throws: ValueError: The gate can’t be applied to the qubits.
### 3.1.6 Circuits and Schedules

Utilities for representing and manipulating quantum computations.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Circuit(moments, device)</code></td>
<td>A mutable list of groups of operations to apply to some qubits.</td>
</tr>
<tr>
<td><code>flatten_op_tree(root, Iterable[Any], ...)</code></td>
<td>Performs an in-order iteration of the operations (leaves) in an OP_TREE.</td>
</tr>
<tr>
<td><code>freeze_op_tree(root, Iterable[Any])</code></td>
<td>Replaces all iterables in the OP_TREE with tuples.</td>
</tr>
<tr>
<td><code>InsertStrategy(name, doc)</code></td>
<td>Indicates preferences on how to add multiple operations to a circuit.</td>
</tr>
<tr>
<td><code>Moment(operations)</code></td>
<td>A simplified time-slice of operations within a sequenced circuit.</td>
</tr>
<tr>
<td><code>moment_by_moment_schedule(device, circuit)</code></td>
<td>Returns a schedule aligned with the moment structure of the Circuit.</td>
</tr>
<tr>
<td><code>OP_TREE</code></td>
<td>Union type; Union[X, Y] means either X or Y.</td>
</tr>
<tr>
<td><code>QubitOrder(explicit_func, ...)</code></td>
<td>Defines the kronecker product order of qubits.</td>
</tr>
<tr>
<td><code>QubitOrderOrList</code></td>
<td>Union type; Union[X, Y] means either X or Y.</td>
</tr>
<tr>
<td><code>Schedule(device, scheduled_operations)</code></td>
<td>A quantum program with operations happening at specific times.</td>
</tr>
<tr>
<td><code>ScheduledOperation(time, duration, operation)</code></td>
<td>An operation that happens over a specified time interval.</td>
</tr>
<tr>
<td><code>transform_op_tree(root, Iterable[Any], ...)</code></td>
<td>Maps transformation functions onto the nodes of an OP_TREE.</td>
</tr>
</tbody>
</table>

**cirq.Circuit**

```python
```

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_with_gate_type`
- `are_all_measurements_terminal`
- `to_unitary_matrix`
- `apply_unitary_effect_to_state`
- `to_text_diagram`
- `to_text_diagram_drawer`

Methods for mutation:
- `insert`
- `append`
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**

- **moments** – The initial list of moments defining the circuit.
- **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>all_operations()</code></td>
<td>Iterates over the operations applied by this circuit.</td>
</tr>
<tr>
<td><code>all_qubits()</code></td>
<td>Returns the qubits acted upon by Operations in this circuit.</td>
</tr>
<tr>
<td><code>append(moment_or_operation_tree, ...)</code></td>
<td>Appends operations onto the end of the circuit.</td>
</tr>
<tr>
<td><code>apply_unitary_effect_to_state(initial_state, ...)</code></td>
<td>Left-multiplies a state vector by the circuit’s unitary effect.</td>
</tr>
<tr>
<td><code>are_all_measurements_terminal()</code></td>
<td></td>
</tr>
<tr>
<td><code>batch_insert(insertions, ...)</code></td>
<td>Applies a batched insert operation to the circuit.</td>
</tr>
<tr>
<td><code>batch_insert_into(insert_intos, ...)</code></td>
<td>Inserts operations into empty spaces in existing moments.</td>
</tr>
<tr>
<td><code>batch_remove(removals, ...)</code></td>
<td>Removes several operations from a circuit.</td>
</tr>
<tr>
<td><code>clear_operations_touching(qubits, moment_indices)</code></td>
<td>Clears operations that are touching given qubits at given moments.</td>
</tr>
<tr>
<td><code>copy()</code></td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
### findall_operations(predicate, bool)
Find the locations of all operations that satisfy a given condition.

### findall_operations_between(start_frontier, ...)
Finds operations between the two given frontiers.

### findall_operations_with_gate_type(gate_type)
The locations of all gate operations of a given type.

### from_ops(*operations, strategy, device)
Creates an empty circuit and appends the given operations.

### insert(index, moment_or_operation_tree, ...)
Inserts operations into the circuit.

### insert_at_frontier(operations, ...)
Inserts operations inline at frontier.

### insert_into_range(operations, ...)
Writes operations inline into an area of the circuit.

### next_moment_operating_on(qubits, ...)
Finds the index of the next moment that touches the given qubits.

### next_moments_operating_on(qubits, ...)
Finds the index of the next moment that touches each qubit.

### operation_at(qubit, moment_index)
Finds the operation on a qubit within a moment, if any.

### prev_moment_operating_on(qubits, ...)
Finds the index of the next moment that touches the given qubits.

### reachable_frontier_from(start_frontier, ...)
Determines how far can be reached into a circuit under certain rules.

### save_qasm(file_path, bytes, int], header, ...)
Save a QASM file equivalent to the circuit.

### to_qasm(header, precision, qubit_order, ...)
Returns QASM equivalent to the circuit.

### to_text_diagram(*, use_unicode_characters, ...)
Returns text containing a diagram describing the circuit.

### to_text_diagram_drawer(*, ...)
Returns a TextDiagramDrawer with the circuit drawn into it.

### to_unitary_matrix(qubit_order, ...)
Converts the circuit into a unitary matrix, if possible.

### with_device(new_device, qubit_mapping, ...)
Maps the current circuit onto a new device, and validates.

---

**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.

**cirq.Circuit.all_qubits**

Circuit.all_qubits() → FrozenSet[cirq.ops.raw_types.QubitId]
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


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.are_all_measurements_terminal

Circuit.are_all_measurements_terminal()

cirq.Circuit.batch_insert

Circuit.batch_insert(insertions: Iterable[Tuple[int, Union[cirq.ops.raw_types.Operation, Iterable[Any]]]]) → None

Applies a batched insert operation to the circuit.

Transparency 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

**cirq.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

**cirq.Circuit.batch_remove** *(removals: Iterable[Tuple[int, cirq.ops.raw_types.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

**cirq.Circuit.clear_operations_touching** *(qubits: Iterable[cirq.ops.raw_types.QubitId], moment_indices: Iterable[int])*

Clears operations that are touching given qubits at given moments.

**Parameters**

- **qubits** – The qubits to check for operations on.
- **moment_indices** – The indices of moments to check for operations within.

cirq.Circuit.copy

**cirq.Circuit.copy** () → circq.circuits.circuit.Circuit

cirq.Circuit.findall_operations


Find the locations of all operations that satisfy a given condition.
This returns an iterator of (index, operation) tuples where each operation satisfies \( \text{op-cond(operation)} \) 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

```python
Circuit.findall_operations_between(start_frontier: Dict[cirq.ops.raw_types.QubitId, int], end_frontier: Dict[cirq.ops.raw_types.QubitId, 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_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

**static** Circuit.from_ops(\*operations, **strategy**: cirq.circuits.insert_strategy.InsertStrategy = cirq.InsertStrategy.EARLIEST, **device**: cirq.devices.device.Device = cirq.UnconstrainedDevice) → cirq.circuits.circuit.Circuit

Creates an empty circuit and appends the given operations.

**Parameters**

- **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.insert


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

Circuit.insert_at_frontier(\*operations: \*Union[cirq.ops.raw_types.Operation, \*Iterable[\*Any]], \*start**: \*int, \*frontier**: \*Dict[cirq.ops.raw_types.QubitId, \*int] = None) → \*Dict[cirq.ops.raw_types.QubitId, \*int]

Inserts operations inline at frontier.

**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.
cirq.Circuit.insert_into_range

Circuit.insert_into_range(operations: Union[cirq.ops.raw_types.Operation, Iterable[Any]], 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.

cirq.Circuit.next_moment_operating_on

Circuit.next_moment_operating_on(qubits: Iterable[cirq.ops.raw_types.QubitId], 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.

cirq.Circuit.next_moments_operating_on

Circuit.next_moments_operating_on(qubits: Iterable[cirq.ops.raw_types.QubitId], start_moment_index: int = 0) → Dict[cirq.ops.raw_types.QubitId, 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.QubitId, moment_index: int) \rightarrow \text{Optional}[\text{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: \text{Sequence}[\text{cirq.ops.raw_types.QubitId}], end_moment_index: \text{Optional}[\text{int}] = \text{None}, max_distance: \text{Optional}[\text{int}] = \text{None}) \rightarrow \text{Optional}[\text{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.reachable_frontier_from

Circuit.reachable_frontier_from(start_frontier: \text{Dict}[\text{cirq.ops.raw_types.QubitId}, \text{int}], *, is_blocker: \text{Callable}[\text{cirq.ops.raw_types.Operation}, \text{bool}] = <function Circuit.<lambda>>, \text{Dict}[\text{cirq.ops.raw_types.QubitId}, \text{int}]) \rightarrow \text{Dict}[\text{cirq.ops.raw_types.QubitId}, \text{int}]

Determines how far can be reached into a circuit under certain rules.

The location \(L = (\text{qubit}, \text{moment}._\text{index})\) is reachable if and only if:

a) \(L\) is one of the items in `'start_frontier'`.

OR

b) There is no operation at \(L\) and \(\text{prev}(L) = (\text{qubit}, \text{moment}._\text{index}-1)\)
   is reachable and \(L\) is within the bounds of the circuit.

OR

c) There is an operation \(P\) covering \(L\) and, for every location \(M = (q', \text{moment}_\text{index})\) that \(P\) covers, the location

(continues on next page)
prev(M) = (q', moment_index-1) is reachable. Also, P must not be classified as a blocker by the given `is_blocker` argument.

In other words, the reachable region extends forward through time along each qubit until it hits a blocked operation or an operation that crosses into the set of not-involved-at-the-moment qubits.

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 end_q. The result of this method is a dictionary, and that dictionary maps each qubit q to its end_q.

**Examples**

If start_frontier is 
```
{ 
  cirq.LineQubit(0): 6, 
  cirq.LineQubit(1): 2, 
  cirq.LineQubit(2): 2, 
}
```
then the reachable wire locations in the following circuit are highlighted with ‘‘ characters:

```
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---@-------------------
```

And the computed end_frontier is 
```
{ 
  cirq.LineQubit(0): 11, 
  cirq.LineQubit(1): 9, 
  cirq.LineQubit(2): 6, 
}
```

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-----H-----
2: H-----H-----H-----
3: H-----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**

```python
```

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: Union[cirq.ops.qubit_order.QubitOrder, Iterable[cirq.ops.raw_types.QubitId]] = <cirq.ops.qubit_order.QubitOrder object>, qubits_that_should_be_present: Iterable[cirq.ops.raw_types.QubitId] = (), ignore_terminal_measurements: bool = True, dtype: Type[numpy.number] = <class 'numpy.complex128'>) → numpy.ndarray

Converts the circuit into a unitary matrix, if possible.

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.

### Attributes

```python
circuit.device
```

### `cirq.flatten_op_tree`

```python
```

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`

```python
```

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.

```python
__init__ (name, doc)
Initialize self. See help(type(self)) for accurate signature.
```

### Methods

---

### Attributes

- **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.

**operations**
A tuple of the Operations for this Moment.

**qubits**
A set of the qubits acted upon by this Moment.

```python
__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><strong>operates_on</strong> (qubits)</td>
<td>Determines if the moment has operations touching the given qubits.</td>
</tr>
<tr>
<td><strong>transform_qubits</strong> (func, ...)</td>
<td>Returns an equal moment, but with the given op added.</td>
</tr>
<tr>
<td><strong>with_operation</strong> (operation)</td>
<td>Returns an equal moment, but without ops on the given qubits.</td>
</tr>
</tbody>
</table>

**cirq.Moment.operates_on**

```python
Moment.operates_on (qubits: Iterable[cirq.ops.raw_types.QubitId]) → 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.transform_qubits**

```python
Moment.transform_qubits (func: Callable[cirq.ops.raw_types.QubitId,
                                       cirq.ops.raw_types.QubitId]) → TSelf_Moment
```

**cirq.Moment.with_operation**

```python
Moment.with_operation (operation: cirq.ops.raw_types.Operation)
```
Returns an equal moment, but with the given op added.

**Parameters**
- **operation** – The operation to append.
**Returns** The new moment.

**cirq.Moment.without_operations_touching**

```
cirq.Moment.without_operations_touching(qubits: Iterable[cirq.ops.raw_types.QubitId])
```

Returns an equal moment, but without ops on the given qubits.

**Parameters**

- **qubits** – Operations that touch these will be removed.

**Returns** The new moment.

**cirq.moment_by_moment_schedule**

```
```

Returns a schedule aligned with the moment structure of the Circuit.

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_TREE**

```
```

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.:

```
Union[int, str] == Union[str, int]
```

• When two arguments have a subclass relationship, the least derived argument is kept, e.g.:

```
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:

```
Union[int, object] == object
```

• You cannot subclass or instantiate a union.

• You can use Optional[X] as a shorthand for Union[X, None].

---

cirq.QubitOrder

class cirq.QubitOrder(explicit_func: Callable[Iterable[cirq.ops.raw_types.QubitId], Tuple[cirq.ops.raw_types.QubitId, ...]])
Defines the kronecker product order of qubits.

```
__init__(explicit_func: Callable[Iterable[cirq.ops.raw_types.QubitId], Tuple[cirq.ops.raw_types.QubitId, ...]]) -> 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>as_qubit_order(val)</td>
<td>Converts a value into a basis.</td>
</tr>
<tr>
<td>explicit(fixed_qubits, fallback)</td>
<td>A basis that contains exactly the given qubits in the given order.</td>
</tr>
<tr>
<td>map(internalize, TInternalQubit[...])</td>
<td>Transforms the Basis so that it applies to wrapped qubits.</td>
</tr>
<tr>
<td>order_for(qubits)</td>
<td>Returns a qubit tuple ordered corresponding to the basis.</td>
</tr>
<tr>
<td>sorted_by(key, Any})</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

**static** `QubitOrder.explicit` *(fixed_qubits: Iterable[cirq.ops.raw_types.QubitId], 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

**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

**QubitOrder.order_for** *(qubits: Iterable[cirq.ops.raw_types.QubitId]) → Tuple[cirq.ops.raw_types.QubitId, ...]*

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

**static** `QubitOrder.sorted_by` *(key: Callable[cirq.ops.raw_types.QubitId, 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

**DEFAULT**

A basis that orders qubits in the same way that calling `sorted` does.

`cirq.QubitOrder.DEFAULT`  

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`  

cirq.QubitOrderOrList = typing.Union[cirq.ops.qubit_order.QubitOrder, typing.Iterable[cirq.raw_types.QubitId]]

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.:

  Union[Union[int, str], float] == Union[int, str, float]

- Unions of a single argument vanish, e.g.:

  Union[int] == int  
  # The constructor actually returns int

- Redundant arguments are skipped, e.g.:

  Union[int, str, int] == Union[int, str]

- When comparing unions, the argument order is ignored, e.g.:

  Union[int, str] == Union[str, int]

- When two arguments have a subclass relationship, the least derived argument is kept, e.g.:

  ```
  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:

  ```
  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

```python
    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.

```python
```

Initialize 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

```python
exclude(scheduled_operation) Omits a scheduled operation from the schedule, if present.
include(scheduled_operation) Adds a scheduled operation to the schedule.
opinations_happening_at_same_time_as(. . . ) Finds operations happening at the same time as the
query(*, time, duration, qubits[. . . ]) Finds operations by time and qubit.
to_circuit() Convert the schedule to a circuit.
```

cirq.Schedule.exclude

```python
Schedule.exclude(scheduled_operation: cirq.schedules.scheduled_operation.ScheduledOperation) → bool
```

Omits a scheduled operation from the schedule, if present.
Cirq Documentation, Release 0.5.0.dev

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)  
Add 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.value.duration.Duration = cirq.Duration(picos=0), **qubits**: Iterable[cirq.ops.raw_types.QubitId] = 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


An operation that happens over a specified time interval.

__init__ (time: cirq.value.timestamp.Timestamp, duration: cirq.value.duration.Duration, 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

**op_at_on**(operation, time, device) Creates a scheduled operation with a device-determined duration.

cirq.ScheduledOperation.op_at_on


Creates a scheduled operation with a device-determined duration.

cirq.transform_op_tree


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.

### 3.1.7 Trials and Simulations

Classes for parameterized circuits.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bloch_vector_from_state_vector(state, index)</code></td>
<td>Returns the bloch vector of a qubit.</td>
</tr>
<tr>
<td><code>density_matrix_from_state_vector(state, indices)</code></td>
<td>Returns the density matrix of the wavefunction.</td>
</tr>
<tr>
<td><code>dirac_notation(state, decimals)</code></td>
<td>Returns the wavefunction as a string in Dirac notation.</td>
</tr>
<tr>
<td><code>Linspace(key, start, stop, length)</code></td>
<td>A simple sweep over linearly-spaced values.</td>
</tr>
<tr>
<td><code>measure_state_vector(state, indices, out)</code></td>
<td>Performs a measurement of the state in the computational basis.</td>
</tr>
<tr>
<td><code>ParamResolver(param_dict, float)</code></td>
<td>Resolves Symbols to actual values.</td>
</tr>
<tr>
<td><code>plot_state_histogram(result)</code></td>
<td>Plot the state histogram from a single result with repetitions.</td>
</tr>
<tr>
<td><code>Points(key, points)</code></td>
<td>A simple sweep with explicitly supplied values.</td>
</tr>
<tr>
<td><code>sample_state_vector(state, indices, repetitions)</code></td>
<td>Samples repeatedly from measurements in the computational basis.</td>
</tr>
<tr>
<td><code>SimulatesSamples</code></td>
<td>Simulator that mimics running on quantum hardware.</td>
</tr>
<tr>
<td><code>SimulationTrialResult(params, measurements, ...)</code></td>
<td>Results of a simulation by a SimulatesFinalWaveFunction.</td>
</tr>
<tr>
<td><code>Simulator([dtype])</code></td>
<td>A sparse matrix wave function simulator that uses numpy.</td>
</tr>
<tr>
<td><code>SimulatorStep(state, measurements, ...)</code></td>
<td>Results of a step of a SimulatesIntermediateWaveFunction.</td>
</tr>
<tr>
<td><code>StepResult(qubit_map, int] = None, ...)</code></td>
<td>A sweep is an iterator over ParamResolvers.</td>
</tr>
<tr>
<td><code>Sweepable Union type; Union[X, Y] means either X or Y.</code></td>
<td>Union type; Union[X, Y] means either X or Y.</td>
</tr>
<tr>
<td><code>to_valid_state_vector(state_rep, ...)</code></td>
<td>Verifies the initial_state is valid and converts it to ndarray form.</td>
</tr>
<tr>
<td><code>validate_normalized_state(state, num_qubits, ...)</code></td>
<td>Validates that the given state is a valid wave function.</td>
</tr>
<tr>
<td><code>to_resolvers(sweepable, ...)</code></td>
<td>Convert a Sweepable to a list of ParamResolvers.</td>
</tr>
<tr>
<td><code>TrialResult(*, params, measurements, ...)</code></td>
<td>The results of multiple executions of a circuit with fixed parameters.</td>
</tr>
<tr>
<td><code>UnitSweep</code></td>
<td>A sweep with a single element that assigns no parameter values.</td>
</tr>
</tbody>
</table>
cirq.bloch_vector_from_state_vector

\( \text{cirq.bloch_vector_from_state_vector(state: Sequence, index: int)} \rightarrow \text{numpy.ndarray} \)

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.

**Returns** A length 3 numpy array representing the qubit’s bloch vector.

**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 index is out of range for the number of qubits corresponding to the state.

---

cirq.density_matrix_from_state_vector

\( \text{cirq.density_matrix_from_state_vector(state: Sequence, indices: Iterable[int] = None)} \rightarrow \text{numpy.ndarray} \)

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:

```python
::

    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.dirac_notation

**cirq.dirac_notation**(state: Sequence, decimals: int = 2) → 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.Linspace

**class cirq.Linspace**(key: str, start: float, stop: float, length: int)

A simple sweep over linearly-spaced values.

**__init__**(key: str, 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**

```python
Linspace.param_tuples() → Iterator[Tuple[Tuple[str, float], ...]]
```

An iterator over (key, value) pairs assigning Symbol key to value.

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys</td>
<td>The keys for the all of the Symbols that are resolved.</td>
</tr>
</tbody>
</table>

**cirq.Linspace.keys**

```python
Linspace.keys
```

The keys for the all of the Symbols that are resolved.

**cirq.measure_state_vector**

```python
cirq.measure_state_vector(state: numpy.ndarray, indices: List[int], out: numpy.ndarray = None) → Tuple[List[bool], 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 ** integer`. The state can be of shape `(2 ** integer)` or `(2, 2, ..., 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.
- `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 `out`.

**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.ParamResolver**

```python
class cirq.ParamResolver (param_dict: Dict[str, float])
```

Resolves 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.

```python
__init__(param_dict: Dict[str, float]) → 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><strong>value_of</strong>(value, float, str)</td>
<td>Attempt to resolve a Symbol or name or float to its assigned value.</td>
</tr>
</tbody>
</table>

**cirq.ParamResolver.value_of**

```python
ParamResolver.value_of(value: Union[cirq.value.symbol.Symbol, float, str]) → Union[cirq.value.symbol.Symbol, float]
```

Try to resolve a Symbol or name or float to its assigned value.

- If unable to resolve a Symbol, returns it unchanged.
- If unable to resolve a name, returns a Symbol with that name.

**Parameters**

- **value** -- The 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**

```python
cirq.plot_state_histogram(result: cirq.study.trial_result.TrialResult) → 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`(key: str, points: Sequence[float])

A simple sweep with explicitly supplied values.

```python
__init__(key: str, points: Sequence[float]) → None
```

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

**Methods**

```python
param_tuples()
```

An iterator over (key, value) pairs assigning Symbol key to value.

```python
cirq.Points.param_tuples
```

Points.param_tuples() → Iterator[Tuple[Tuple[str, float], ...]]

An iterator over (key, value) pairs assigning Symbol key to value.

**Attributes**

```python
keys
```

The keys for the all of the Symbols that are resolved.

```python
cirq.Points.keys
```

Points.keys

The keys for the all of the Symbols that are resolved.

**cirq.sample_state_vector**

```python
cirq.sample_state_vector(state: numpy.ndarray, indices: List[int], repetitions: int = 1) → 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 the largest values that the 0th index.

- `repetitions` – The number of times to sample the state.

**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 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.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

circuit.SimulatesSamples.compute_samples_displays(program, ...)
Computes SamplesDisplays in the supplied Circuit or Schedule.

circuit.SimulatesSamples.compute_samples_displays_sweep(program, ...
Computes SamplesDisplays in the supplied Circuit or Schedule.

circuit.SimulatesSamples.run(program, ...)
Runs the supplied Circuit or Schedule, mimicking quantum hardware.

circuit.SimulatesSamples.run_sweep(program, ...)
Runs the supplied Circuit or Schedule, mimicking quantum hardware.

3.1. API Reference 143
SimulatesSamples\texttt{.compute_samples_displays_sweep} (\texttt{program:} \texttt{Union[cirq.circuits.circuit.Circuit, cirq.schedules.schedule.Schedule]}, \texttt{params:} \texttt{Union[cirq.study.resolver.ParamResolver, Iterable[cirq.study.resolver.ParamResolver], cirq.study.sweeps.Sweep, Iterable[cirq.study.sweeps.Sweep], None] = None}) \rightarrow \texttt{List[cirq.study.compute_displays_result.ComputeDisplaysResult]}

Computes \textit{SamplesDisplays} in the supplied Circuit or Schedule.

In contrast to \texttt{compute_displays}, this allows for sweeping over different parameter values.

\textbf{Parameters}

\begin{itemize}
\item \texttt{program} – The circuit or schedule to simulate.
\item \texttt{params} – Parameters to run with the program.
\end{itemize}

\textbf{Returns} List of \texttt{ComputeDisplaysResults} for this run, one for each possible parameter resolver.

SimulatesSamples\texttt{.run} (\texttt{program:} \texttt{Union[cirq.circuits.circuit.Circuit, cirq.schedules.schedule.Schedule]}, \texttt{param_resolver:} \texttt{Optional[cirq.study.resolver.ParamResolver] = None}, \texttt{repetitions:} \texttt{int = 1}) \rightarrow \texttt{cirq.study.trial_result.TrialResult}

Runs the supplied Circuit or Schedule, mimicking quantum hardware.

\textbf{Parameters}

\begin{itemize}
\item \texttt{program} – The circuit or schedule to simulate.
\item \texttt{param_resolver} – Parameters to run with the program.
\item \texttt{repetitions} – The number of repetitions to simulate.
\end{itemize}

\textbf{Returns} \texttt{TrialResult} for a run.

SimulatesSamples\texttt{.run_sweep} (\texttt{program:} \texttt{Union[cirq.circuits.circuit.Circuit, cirq.schedules.schedule.Schedule]}, \texttt{params:} \texttt{Union[cirq.study.resolver.ParamResolver, Iterable[cirq.study.resolver.ParamResolver], cirq.study.sweeps.Sweep, Iterable[cirq.study.sweeps.Sweep]], repetitions:} \texttt{int = 1}) \rightarrow \texttt{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.SimulationTrialResult**

```python
class cirq.SimulationTrialResult (params: cirq.study.resolver.ParamResolver, measurements: Dict[str, numpy.ndarray], final_state: numpy.ndarray)
```

Results of a simulation by a SimulatesFinalWaveFunction.

Unlike TrialResult these results contain the final state (wave function) 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.)

**final_state**

The final state (wave function) of the system after the trial finishes. The state is returned in the computational basis with these basis states defined by the qubit ordering of the simulation. In particular the qubit ordering can be used to produce a list of qubits, and these qubits can the be associated with their index in the list. This mapping of qubit to index is then 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 ordering: [QubitA, QubitB, QubitC]
```

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>0</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>

`__init__(params: cirq.study.resolver.ParamResolver, measurements: Dict[str, numpy.ndarray], final_state: numpy.ndarray) → 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>bloch_vector(index)</code></td>
<td>Returns the bloch vector of a qubit.</td>
</tr>
<tr>
<td><code>density_matrix(indices)</code></td>
<td>Returns the density matrix of the wavefunction.</td>
</tr>
<tr>
<td><code>dirac_notation(decimals)</code></td>
<td>Returns the wavefunction as a string in Dirac notation.</td>
</tr>
</tbody>
</table>

`cirq.SimulationTrialResult.bloch_vector`

`SimulationTrialResult.bloch_vector(index: int) → numpy.ndarray`

Returns the bloch vector of a qubit.

Calculates the bloch vector of the qubit at index in the wavefunction given by self.state. Given that self.state follows the standard Kronecker convention of numpy.kron.

**Parameters** `index` – index of 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.SimulationTrialResult.density_matrix`

`SimulationTrialResult.density_matrix(indices: Iterable[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 self.final_state traced out. If indices is `None` the full density matrix for self.final_state is returned, given self.final_state

(continues on next page)
follows the standard Kronecker convention of numpy.kron.

For example:

```python
def final_state(self):
    self.final_state = np.array([1/np.sqrt(2), 1/np.sqrt(2)],
dtype=np.complex64)
indices = None
gives us

ho = \begin{bmatrix}
0.5 & 0.5 \\
0.5 & 0.5 \\
\end{bmatrix}
```

Args:
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.

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.SimulationTrialResult.dirac_notation

SimulationTrialResult.dirac_notation(decimals: int = 2) → str
Returns the wavefunction 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.Simulator

class cirq.Simulator(dtype=<class 'numpy.complex64'>)
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_ methods, or else a _decompose_ method that returns operations satisfying these same conditions. That is to say, the operations should follow the cirq.SupportsApplyUnitary protocol, the cirq.SupportsUnitary protocol, or the cirq.CompositeOperation protocol. (It is also permitted for the circuit
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 `TrialResult` objects 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 `SimulationTrialResult` 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:

```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 `SimulationTrialResult` which contains in addition to measurement results and information about the parameters that were used in the simulation access to the state via the `final_state` method.

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.
simulate_moment_steps(circuit, param Resolver, qubit_order, initial_state)

One can iterate over the moments via

for step_result in simulate_moments(circuit): # do something with the wave function via
step_result.state

Finally, one can compute the values of displays (instances of SamplesDisplay or WaveFunctionDisplay) in the circuit:

compute_displays(circuit, param Resolver, qubit_order, initial_state)
compu te_displays_sweep(circuit, params, qubit_order, initial_state)

The result of computing display values is stored in a ComputeDisplaysResult.

See Simulator for the definitions of the supported methods.

__init__(dtype=<class 'numpy.complex64'>)
A sparse matrix simulator.

Parameters

• dtype – The numpy.dtype used by the simulation. One of

• or numpy.complex128 (numpy.complex64) –

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute_displays(program, ...)</td>
<td>Computes displays in the supplied Circuit or Schedule.</td>
</tr>
<tr>
<td>compute_displays_sweep(program, ...)</td>
<td>Computes displays in the supplied Circuit or Schedule.</td>
</tr>
<tr>
<td>compute_samples_displays(program, ...)</td>
<td>Computes SamplesDisplays in the supplied Circuit or Schedule.</td>
</tr>
<tr>
<td>compute_samples_displays_sweep(program, ...)</td>
<td>Computes SamplesDisplays in the supplied Circuit or Schedule.</td>
</tr>
<tr>
<td>run(program, ...)</td>
<td>Runs the supplied Circuit or Schedule, mimicking quantum hardware.</td>
</tr>
<tr>
<td>run_sweep(program, ...)</td>
<td>Runs the supplied Circuit or Schedule, mimicking quantum hardware.</td>
</tr>
<tr>
<td>simulate(program, ...)</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, ...)</td>
<td>Simulates the supplied Circuit or Schedule.</td>
</tr>
</tbody>
</table>

cirq.Simulator.compute_displays


Computes displays in the supplied Circuit or Schedule.
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 used to define the order of amplitudes in the wave function.

• **initial_state** – If an int, the state is set to the computational basis state corresponding to this state. Otherwise if this is a np.ndarray it is the full initial state. In this case it must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an appropriate dtype for the simulator.

Returns  ComputeDisplaysResult for the simulation.

cirq.Simulator.compute_displays_sweep


Computes displays in the supplied Circuit or Schedule.

In contrast to `compute_displays`, 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 used to define the order of amplitudes in the wave function.

• **initial_state** – If an int, the state is set to the computational basis state corresponding to this state. Otherwise if this is a np.ndarray it is the full initial state. In this case it must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an appropriate dtype for the simulator.

Returns  List of ComputeDisplaysResults for this run, one for each possible parameter resolver.
cirq.Simulator.compute_samples_displays


Computes SamplesDisplays in the supplied Circuit or Schedule.

Parameters

• program – The circuit or schedule to simulate.
• paramresolver – Parameters to run with the program.

Returns ComputeDisplaysResult for the simulation.

cirq.Simulator.compute_samples_displays_sweep


Computes SamplesDisplays in the supplied Circuit or Schedule.

In contrast to compute_displays, this allows for sweeping over different parameter values.

Parameters

• program – The circuit or schedule to simulate.
• params – Parameters to run with the program.

Returns List of ComputeDisplaysResults for this run, one for each possible parameter resolver.

cirq.Simulator.run


Runs the supplied Circuit or Schedule, mimicking quantum hardware.

Parameters

• program – The circuit or schedule to simulate.
• param_resolver – Parameters to run with the program.
• repetitions – The number of repetitions to simulate.

Returns TrialResult for a run.
cirq.Simulator.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.Simulator.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 used to define the order of amplitudes in the wave function.
- **initial_state** – If an int, the state is set to the computational basis state corresponding to this state. Otherwise if this is a np.ndarray it is the full initial state. In this case it must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an appropriate dtype for the simulator.

**Returns** `SimulateTrialResults` for the simulation. Includes the final wave function.
cirq.Simulator.simulate_moment_steps


Returns an iterator of StepResults for each moment simulated.

Parameters

• circuit – The Circuit to simulate.
• param_resolver – A ParamResolver for determining values of Symbols.
• qubit_order – Determines the canonical ordering of the qubits used to define the order of amplitudes in the wave function.
• initial_state – If an int, the state is set to the computational basis state corresponding to this state. Otherwise if this is a np.ndarray it is the full initial state. In this case it must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an appropriate dtype for the simulator.

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 used to define the order of amplitudes in the wave function.
• **initial_state** – If an int, the state is set to the computational basis state corresponding to this state. Otherwise if this is a np.ndarray it is the full initial state. In this case it must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an appropriate dtype for the simulator.

**Returns** List of SimulatorTrialResults for this run, one for each possible parameter resolver.

cirq.SimulatorStep

class cirq.SimulatorStep(state, measurements, qubit_map, dtype)

__init__(state, measurements, qubit_map, dtype)

Results of a step of the simulator.

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).

measurements

A dictionary from measurement gate key to measurement results, ordered by the qubits that the measurement operates on.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bloch_vector_of(qubit)</td>
<td>Returns the bloch vector of a qubit in the state.</td>
</tr>
<tr>
<td>density_matrix_of(qubits)</td>
<td>Returns the density matrix of the state.</td>
</tr>
<tr>
<td>dirac_notation(decimals)</td>
<td>Returns the state as a string in Dirac notation.</td>
</tr>
<tr>
<td>sample(qubits, repetitions)</td>
<td>Samples from the wave function at this point in the computation.</td>
</tr>
<tr>
<td>sample_measurement_ops(measurement_ops, ...)</td>
<td>Samples from the wave function at this point in the computation.</td>
</tr>
<tr>
<td>set_state(state, numpy.ndarray)</td>
<td>Updates the state of the simulator to the given new state.</td>
</tr>
<tr>
<td>state_vector()</td>
<td>Return the state (wave function).</td>
</tr>
</tbody>
</table>

cirq.SimulatorStep.bloch_vector_of

SimulatorStep.bloch_vector_of(qubit: cirq.ops.raw_types.QubitId) -> 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.SimulatorStep.density_matrix_of

SimulatorStep.density_matrix_of(qubits: List[cirq.ops.raw_types.QubitId] = 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:

```python
self.state_vector() = np.array([1/np.sqrt(2), 1/np.sqrt(2)],
   dtype=np.complex64)
qubits = None
gives us
ho = \begin{bmatrix}
0.5 & 0.5 \\
0.5 & 0.5
\end{bmatrix}
```

**Args:**

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.SimulatorStep.dirac_notation

SimulatorStep.dirac_notation(\textit{decimals}: \textit{int} = 2) \rightarrow \text{str}

Returns the state as a string in Dirac notation.

\textbf{Parameters} \textit{decimals} – How many decimals to include in the pretty print.

\textbf{Returns} A pretty string consisting of a sum of computational basis kets and non-zero floats of the specified accuracy.

cirq.SimulatorStep.sample

SimulatorStep.sample(\textit{qubits}: \text{List}[cirq.ops.raw_types.QubitId], \textit{repetitions}: \text{int} = 1) \rightarrow \text{numpy.ndarray}

Samples from the wave function at this point in the computation.

Note that this does not collapse the wave function.

\textbf{Parameters}

\begin{itemize}
\item \textit{qubits} – The qubits to be sampled in an order that influence the returned measurement results.
\item \textit{repetitions} – The number of samples to take.
\end{itemize}

\textbf{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.SimulatorStep.sample_measurement_ops

SimulatorStep.sample_measurement_ops(\textit{measurement_ops}: \text{List}[cirq.ops.gate_operation.GateOperation], \textit{repetitions}: \text{int} = 1) \rightarrow \text{Dict}[\text{str}, \text{numpy.ndarray}]

Samples from the wave function at this point in the computation.

Note that this does not collapse the wave function.

In contrast to \texttt{sample} which samples qubits, this takes a list of \texttt{cirq.GateOperation} instances whose gates are \texttt{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.

\textbf{Parameters}

\begin{itemize}
\item \texttt{measurement_ops} – \texttt{GateOperation} instances whose gates are \texttt{MeasurementGate} instances to be sampled form.
\item \texttt{repetitions} – The number of samples to take.
\end{itemize}

\textbf{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.SimulatorStep.set_state`

**SimulatorStep.set_state**(*state*: `Union[int, numpy.ndarray]`)

Updates the state of the simulator to the given new state.

**Parameters**

- **state** – If this is an int, then this is the state to reset  
  - *stepper to, expressed as an integer of the computational basis.* *(the)*—  
  - *to bitwise indices is little endian. Otherwise if this is (Integer)*—  
  - *np.ndarray this must be the correct size and have dtype of (a)*—  
  - *np.complex64.*—  

**Raises**

- `ValueError` if the state is incorrectly sized or not of the correct  
  - *dtype.*

### `cirq.SimulatorStep.state_vector`

**SimulatorStep.state_vector**() → `numpy.ndarray`

Return the state (wave function).

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>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

---

3.1. API Reference 157
class cirq.StepResult(qubit_map: Optional[Dict[cirq.ops.raw_types.QubitId, int]] = None, measurements: Optional[Dict[str, List[bool]]] = None)

Results of a step of a SimulatesIntermediateWaveFunction.

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).

measurements

A dictionary from measurement gate key to measurement
results, ordered by the qubits that the measurement operates on.

__init__(qubit_map: Optional[Dict[cirq.ops.raw_types.QubitId, int]] = None, measurements: Optional[Dict[str, List[bool]]] = None) → 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>bloch_vector_of(qubit)</td>
<td>Returns the bloch vector of a qubit in the state.</td>
</tr>
<tr>
<td>density_matrix_of(qubits)</td>
<td>Returns the density matrix of the state.</td>
</tr>
<tr>
<td>dirac_notation(decimals)</td>
<td>Returns the state as a string in Dirac notation.</td>
</tr>
<tr>
<td>sample(qubits, repetitions)</td>
<td>Samples from the wave function at this point in the computation.</td>
</tr>
<tr>
<td>sample_measurement_ops(measurement_ops, ...)</td>
<td>Samples from the wave function at this point in the computation.</td>
</tr>
<tr>
<td>set_state(state, numpy.ndarray)</td>
<td>Updates the state of the simulator to the given new state.</td>
</tr>
<tr>
<td>state_vector()</td>
<td>Return the state (wave function).</td>
</tr>
</tbody>
</table>
**cirq.StepResult.bloch_vector_of**

```
cirq.StepResult.bloch_vector_of(qubit: cirq.ops.raw_types.QubitId) \rightarrow \text{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.StepResult.density_matrix_of**

```
cirq.StepResult.density_matrix_of(qubits: List[cirq.ops.raw_types.QubitId] = None) \rightarrow \text{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)
quubits = None
gives us
```

```
ho = \begin{bmatrix}
0.5 & 0.5 \\
0.5 & 0.5 
\end{bmatrix}
```

- **Args**:
  - `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**:

(continues on next page)
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.StepResult.dirac_notation

```python
StepResult.dirac_notation(decimals: int = 2) \rightarrow str
```

Returns the state 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.StepResult.sample

```python
StepResult.sample(qubits: List[cirq.ops.raw_types.QubitId], repetitions: int = 1) \rightarrow numpy.ndarray
```

Samples from the wave function 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.

**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.StepResult.sample_measurement_ops

```python
StepResult.sample_measurement_ops(measurement_ops: List[cirq.ops.gate_operation.GateOperation], repetitions: int = 1) \rightarrow Dict[str, numpy.ndarray]
```

Samples from the wave function 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.

**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.StepResult.set_state**

```python
StepResult.set_state(state: Union[int, numpy.ndarray]) -> None
```

Updates the state of the simulator to the given new state.

**Parameters**

- **state** – If this is an int, then this is the state to reset
- **stepper to, expressed as an integer of the computational basis.** (the)
- **to bitwise indices is little endian. Otherwise if this is** (Integer)
- **np.ndarray this must be the correct size and have dtype of** (a)
- **np.complex64.**

**Raises**

- Value error if the state is incorrectly sized or not of the correct
- dtype.

**cirq.StepResult.state_vector**

```python
StepResult.state_vector() -> numpy.ndarray
```

Return the state (wave function).

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**

```python
qubit_map: {QubitA: 0, QubitB: 1, QubitC: 2}
```

Then the returned vector will have indices mapped to qubit basis

---

**3.1. API Reference**

161
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>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</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>

cirq.SimulatesFinalWaveFunction

class cirq.SimulatesFinalWaveFunction
    Simulator that allows access to a quantum computer’s wavefunction.

    Implementors of this interface should implement the simulate_sweep method. This simulator only returns the wave function for the final step of a simulation. For simulators that also allow stepping through a circuit see SimulatesIntermediateWaveFunction.

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

    Methods

    simulate(program,...) Simulates the supplied Circuit or Schedule.
    simulate_sweep(program,...) Simulates the supplied Circuit or Schedule.
cirq.SimulatesFinalWaveFunction.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 used to define the order of amplitudes in the wave function.
- **initial_state** – If an int, the state is set to the computational basis state corresponding to this state. Otherwise if this is a np.ndarray it is the full initial state. In this case it must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an appropriate dtype for the simulator.

**Returns** SimulateTrialResults for the simulation. Includes the final wave function.

cirq.SimulatesFinalWaveFunction.simulate_sweep

SimulatesFinalWaveFunction.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 used to define the order of amplitudes in the wave function.
- **initial_state** – If an int, the state is set to the computational basis state corresponding to this state. Otherwise if this is a np.ndarray it is the full initial state. In this case it must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an appropriate dtype for the simulator.

**Returns** List of SimulatorTrialResults for this run, one for each possible parameter resolver.

cirq.SimulatesIntermediateWaveFunction

class cirq.SimulatesIntermediateWaveFunction

A SimulatesFinalWaveFunction that simulates a circuit by moments.

Whereas a general SimulatesFinalWaveFunction may return the entire wave function at the end of a circuit, a SimulatesIntermediateWaveFunction 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>compute_displays(program,...)</td>
<td>Computes displays in the supplied Circuit or Schedule.</td>
</tr>
<tr>
<td>compute_displays_sweep(program,...)</td>
<td>Computes displays in the supplied Circuit or Schedule.</td>
</tr>
<tr>
<td>simulate(program,...)</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,...)</td>
<td>Simulates the supplied Circuit or Schedule.</td>
</tr>
</tbody>
</table>
cirq.SimulatesIntermediateWaveFunction.compute_displays

SimulatesIntermediateWaveFunction.compute_displays(program:
Union[cirq.circuits.circuit.Circuit,
cirq.schedules.schedule.Schedule],
param_resolver:
cirq.study.resolver.ParamResolver
= cirq.ParamResolver({}),
qubit_order:
Union[cirq.ops.qubit_order.QubitOrder,
Iterable[cirq.ops.raw_types.QubitId]]
= <cirq.ops.qubit_order.QubitOrder
object>,
initial_state: Union[int,
numpy.ndarray] = 0) →
cirq.study.compute_displays_result.ComputeDisplaysResult

Computes displays in the supplied Circuit or Schedule.

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 used to define the order of amplitudes in the wave function.
- **initial_state** – If an int, the state is set to the computational basis state corresponding to this state. Otherwise if this is a np.ndarray it is the full initial state. In this case it must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an appropriate dtype for the simulator.

Returns ComputeDisplaysResult for the simulation.
SimulatesIntermediateWaveFunction.compute_displays_sweep

SimulatesIntermediateWaveFunction.compute_displays_sweep

\[
\text{compute_displays_sweep}(\text{program}: \quad \text{Union[cirq.circuits.circuit.Circuit, cirq.schedules.schedule.Schedule]}, \\
\text{params}: \quad \text{Union[cirq.study.resolver.ParamResolver, Iterable[cirq.study.resolver.ParamResolver], cirq.study.sweeps.Sweep, Iterable[cirq.study.sweeps.Sweep], None]} = \text{None}, \\
\text{qubit_order}: \quad \text{Union[cirq.ops.qubit_order.QubitOrder, Iterable[cirq.ops.raw_types.QubitId]]} \\
= \quad <\text{cirq.ops.qubit_order.QubitOrder object}, \text{initial_state}: \quad \text{Union[int, numpy.ndarray]} \\
\text{initial_state:} \quad \text{int, numpy.ndarray} \\
\text{initial_state:} \quad 0) \quad \rightarrow \quad \text{List[cirq.study.compute_displays_result.ComputeDisplaysResult]}
\]

Computes displays in the supplied Circuit or Schedule.

In contrast to compute_displays, 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 used to define the order of amplitudes in the wave function.
- **initial_state** – If an int, the state is set to the computational basis state corresponding to this state. Otherwise if this is a np.ndarray it is the full initial state. In this case it must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an appropriate dtype for the simulator.

Returns List of ComputeDisplaysResults for this run, one for each possible parameter resolver.
cirq.SimulatesIntermediateWaveFunction.simulate

SimulatesIntermediateWaveFunction.simulate (program:
Union[cirq.circuits.circuit.Circuit,
cirq.schedules.schedule.Schedule],
param_resolver: Optional[cirq.study.resolver.ParamResolver] = None,
qubit_order: Union[cirq.ops.qubit_order.QubitOrder,
Iterable[cirq.ops.raw_types.QubitId]] = <cirq.ops.qubit_order.QubitOrder object>,
initial_state: Union[int,
numpy.ndarray] = 0) → 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 used to define the order
of amplitudes in the wave function.
• initial_state – If an int, the state is set to the computational basis state correspond-
ing to this state. Otherwise if this is a np.ndarray it is the full initial state. In this case it
must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an
appropriate dtype for the simulator.

Returns SimulateTrialResults for the simulation. Includes the final wave function.

cirq.SimulatesIntermediateWaveFunction.simulate_moment_steps

SimulatesIntermediateWaveFunction.simulate_moment_steps (circuit:
cirq.circuits.circuit.Circuit,
param_resolver: Optional[cirq.study.resolver.ParamResolver] = None,
qubit_order: Union[cirq.ops.qubit_order.QubitOrder,
Iterable[cirq.ops.raw_types.QubitId]] =
<cirq.ops.qubit_order.QubitOrder object>,
initial_state: Union[int,
numpy.ndarray] = 0) → Iterator[cirq.sim.simulator.StepResult]

Returns an iterator of StepResults for each moment simulated.
Parameters

- **circuit** – The Circuit to simulate.
- **param_resolver** – A ParamResolver for determining values of Symbols.
- **qubit_order** – Determines the canonical ordering of the qubits used to define the order of amplitudes in the wave function.
- **initial_state** – If an int, the state is set to the computational basis state corresponding to this state. Otherwise if this is a np.ndarray it is the full initial state. In this case it must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an appropriate dtype for the simulator.

Returns

Iterator that steps through the simulation, simulating each moment and returning a StepResult for each moment.

cirq.SimulatesIntermediateWaveFunction.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 used to define the order of amplitudes in the wave function.
- **initial_state** – If an int, the state is set to the computational basis state corresponding to this state. Otherwise if this is a np.ndarray it is the full initial state. In this case it
Cirq Documentation, Release 0.5.0.dev

must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an appropriate dtype for the simulator.

Returns List of SimulatorTrialResults for this run, one for each possible parameter resolver.

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’ 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.

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

Methods

param_tuples() An iterator over (key, value) pairs assigning Symbol key to value.

cirq.Sweep.param_tuples

Sweep.param_tuples() → Iterator[Tuple[Tuple[str, float], ...]]
An iterator over (key, value) pairs assigning Symbol key to value.

Attributes

keys The keys for the all of the Symbols that are resolved.
cirq.Sweep.keys

Sweep.keys
The keys for the all of the 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.:

  Union[Union[int, str], float] == Union[int, str, float]

- Unions of a single argument vanish, e.g.:

  Union[int] == int  # The constructor actually returns int

- Redundant arguments are skipped, e.g.:

  Union[int, str, int] == Union[int, str]

- When comparing unions, the argument order is ignored, e.g.:

  Union[int, str] == Union[str, int]

- When two arguments have a subclass relationship, the least derived argument is kept, e.g.:

  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:

  Union[int, object] == object

- You cannot subclass or instantiate a union.
- You can use Optional[X] as a shorthand for Union[X, None].

cirq.to_valid_state_vector

cirq.to_valid_state_vector(state_rep: Union[int, numpy.ndarray], num_qubits: int, dtype: Type[numpy.number] = <class 'numpy.complex64'>) → numpy.ndarray

Verifies the initial_state 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.

- **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.

**Returns** A numpy ndarray corresponding to the state on the given number of qubits.

### cirq.validate_normalized_state

```python
cirq.validate_normalized_state(state: numpy.ndarray, num_qubits: int, dtype: Type[numpy.number] = <class 'numpy.complex64'>) → None
```
Validates that the given state is a valid wave function.

### cirq.to_resolvers

```python
```
Convert a Sweepable to a list of ParamResolvers.

### cirq.TrialResult

```python
class cirq.TrialResult(*, params: cirq.study.resolver.ParamResolver, measurements: Dict[str, numpy.ndarray], repetitions: int)
```
The results of multiple executions of a circuit with fixed parameters.

**params**
A ParamResolver of settings used when sampling result.

**measurements**
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.)

**repetitions**
The number of times a circuit was sampled to get these results.

```python
__init__(*, params: cirq.study.resolver.ParamResolver, measurements: Dict[str, numpy.ndarray], repetitions: int) -> 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.
- **repetitions** – The number of times the circuit was sampled.

**Methods**

```python
histogram(*, key, fold_func, ...) Counts the number of times a measurement result occurred.
```

```python
multi_measurement_histogram(*, keys, ...) Counts the number of times combined measurement results occurred.
```

**cirq.TrialResult.histogram**

```python
TrialResult.histogram(*, key: str, fold_func: Callable[numpy.ndarray, T] = <function big_endian_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**
- **positional_args** – Never specified. Forces keyword arguments.
- **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.

```python
cirq.TrialResult.multi_measurement_histogram

TrialResult.multi_measurement_histogram(*, keys: Iterable[str], fold_func: Callable[Tuple[numpy.ndarray, ...], T] = <function _tuple_of_big_endian_int>) \rightarrow collections.Counter
```

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:

```python
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.

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.

### 3.1.8 Magic Method Protocols

Utility methods for accessing generic functionality exposed by some gates, operations, and other types.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>apply_unitary</strong></td>
<td>High performance left-multiplication of a unitary effect onto a tensor.</td>
</tr>
<tr>
<td><strong>circuit_diagram_info</strong></td>
<td>Requests information on drawing an operation in a circuit diagram.</td>
</tr>
<tr>
<td><strong>decompose</strong></td>
<td>Recursively decomposes a value into cirq. Operation s meeting a criteria.</td>
</tr>
<tr>
<td><strong>decompose_once</strong></td>
<td>Decomposes a value into operations, if possible.</td>
</tr>
<tr>
<td><strong>decompose_once_with_qubits</strong></td>
<td>Decomposes a value into operations on the given qubits.</td>
</tr>
<tr>
<td><strong>inverse</strong></td>
<td>Returns the inverse ( \text{val}^{-1} ) of the given value, if defined.</td>
</tr>
<tr>
<td><strong>mul</strong></td>
<td>Returns ( \text{lhs} \times \text{rhs} ), or else a default if the operator is not implemented.</td>
</tr>
<tr>
<td><strong>pow</strong></td>
<td>Returns ( \text{val}^{\text{factor}} ) of the given value, if defined.</td>
</tr>
<tr>
<td><strong>qasm</strong></td>
<td>Returns QASM code for the given value, if possible.</td>
</tr>
<tr>
<td><strong>is_parameterized</strong></td>
<td>Returns whether the object is parameterized with any Symbols.</td>
</tr>
<tr>
<td><strong>resolve_parameters</strong></td>
<td>Resolves symbol parameters in the effect using the param resolver.</td>
</tr>
<tr>
<td><strong>has_unitary</strong></td>
<td>Returns whether the value has a unitary matrix representation.</td>
</tr>
<tr>
<td><strong>unitary</strong></td>
<td>Returns a unitary matrix describing the given value.</td>
</tr>
<tr>
<td><strong>trace_distance_bound</strong></td>
<td>Returns a maximum on the trace distance between this effect’s input</td>
</tr>
<tr>
<td><strong>phase_by</strong></td>
<td>Returns a phased version of the effect.</td>
</tr>
</tbody>
</table>

cirq.apply_unitary

cirq.**apply_unitary** (**unitary_value**: Any, **args**: cirq.protocols.apply_unitary.ApplyUnitaryArgs, **default**: TDefault = array([], dtype=float64)) ➔ Union[numpy.ndarray, TDefault]

High performance left-multiplication of a unitary effect onto a tensor.
If `unitary_value` defines an `_apply_unitary_` method, that method will be used to apply `unitary_value`'s unitary effect to the target tensor. Otherwise, if `unitary_value` defines a `_unitary_` method, its unitary matrix will be retrieved and applied using a generic method. Otherwise the application fails, and either an exception is raised or the specified default value is returned.

**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 is not able to apply its unitary effect, the specified default value is returned (or a TypeError is raised). If this occurs, then `target_tensor` should not have been mutated.

If the receiving object was able to work inline, directly mutating `target_tensor` it will return `target_tensor`. The caller is responsible for checking if the result is `target_tensor`.

If the receiving object wrote its output over `available_buffer`, the result will be `available_buffer`. The caller is responsible for checking if the result is `available_buffer` (and e.g. swapping the buffer for the target tensor before the next call).

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` – `unitary_value` doesn’t have a unitary effect and `default` wasn’t specified.

cirq.circuit_diagram_info


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.
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.
- `TypeError` – 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=([],), **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.
• kwargs – Arguments to forward into the _decompose_ method of val. For example, this is used to tell gates what qubits they are being applied to.

Returns The result of val._decompose_(**kwargs), 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.QubitId], 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=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=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.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.mul

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.pow

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`

```
```

Returns QASM code for the given value, if possible.

**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. `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.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 \texttt{_is_parameterized_} method and
that method returns a truthy value.

**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 \texttt{NotImplemented}, this will default to
False.

cirq.resolve_parameters

cirq.resolve_parameters(val: Any, 
param_resolver: cirq.study.resolver.ParamResolver) → Any

Resolves symbol parameters in the effect using the param resolver.

This function will use the \texttt{_resolve_parameters_} magic method
of \texttt{val} to resolve any Symbols with concrete values from the given
parameter resolver.

**Parameters**

- \texttt{val} – The object to resolve (e.g. the gate, operation, etc)
- \texttt{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 \texttt{val} has no \texttt{_resolve_parameters_} method or if it returns \texttt{NotImple-
mented}, \texttt{val} itself is returned.

cirq.has_unitary

cirq.has_unitary(val: Any) → bool

Returns whether the value has a unitary matrix representation.

**Returns** If \texttt{val} has a \texttt{_has_unitary_} method and its result is not \texttt{NotImplemented}, that result is re-
turned. Otherwise, if the value has a \texttt{_unitary_} method return if that has a non-default value.
Returns False if neither function exists.

cirq.unitary

cirq.unitary(val: Any, 
default: TDefault = array([], dtype=float64)) → Union[numpy.ndarray, TDe-
fault]

Returns a unitary matrix describing the given value.

**Parameters**

- \texttt{val} – The value to describe with a unitary matrix.
- \texttt{default} – Determines the fallback behavior when \texttt{val} doesn’t have a unitary matrix. If
\texttt{default} is not set, a TypeError is raised. If default is set to a value, that value is returned.

**Returns** If \texttt{val} has a \texttt{_unitary_} method and its result is not \texttt{NotImplemented}, that result is returned.
Otherwise, if a default value was specified, the default value is returned.

**Raises** \texttt{TypeError} – \texttt{val} doesn’t have a \texttt{_unitary_} method (or that method returned \texttt{NotImple-
mented}) and also no default value was specified.
cirq.trace_distance_bound

\[
\text{cirq.\text{trace\_distance\_bound}}(\text{val}: \text{Any}) \rightarrow \text{float}
\]

Returns a maximum on the trace distance between this effect’s input and output. This method makes use of the effect’s \_trace_distance_bound_ method to determine the maximum bound on the trace difference between before and after the effect.

**Parameters**  
\text{val} – The effect of which the bound should be calculated

**Returns**  
If \text{val} has a \_trace_distance_bound_ method and its result is not NotImplemented, that result is returned. Otherwise, 1 is returned. Result is capped at a maximum of 1, even if the underlying function produces a result greater than 1.

cirq.phase_by

\[
\text{cirq.\text{phase\_by}}(\text{val}: \text{Any}, \text{phase\_turns}: \text{float}, \text{qubit\_index}: \text{int}, \text{default}: \text{TDefault} = (\text{[]}, ))
\]

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 \text{val}'s \text{phase\_by} method and returning the result.

**Parameters**

- \text{val} – The value to describe with a unitary matrix.
- \text{phase\_turns} – The amount to phase the gate, in fractions of a whole turn. Divide by 2pi to get radians.
- \text{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 \text{on} method.
- \text{default} – The default value to return if \text{val} can’t be phased. If not specified, an error is raised when \text{val} can’t be phased.

**Returns**  
If \text{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 – \text{val} doesn’t have a \_phase\_by_ method (or that method returned NotImplemented) and no \text{default} was specified.

### 3.1.9 Magic Method Protocol Types

Classes defining and used by the magic method protocols.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CircuitDiagramInfo</td>
<td>Describes how to draw an operation in a circuit diagram.</td>
</tr>
<tr>
<td>CircuitDiagramInfoArgs</td>
<td>A request for information on drawing an operation in a circuit diagram.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 81 – continued from previous page

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QasmArgs(precision, version, qubit_id_map, ...)</td>
<td>An object that can be efficiently left-multiplied into tensors.</td>
</tr>
<tr>
<td>QasmOutput(operations, Iterable[Any], ...)</td>
<td>A diagrammable operation on qubits.</td>
</tr>
<tr>
<td>SupportsApplyUnitary(*args, **kwargs)</td>
<td>An object that can be decomposed into simpler operations.</td>
</tr>
<tr>
<td>SupportsCircuitDiagramInfo(*args, **kwargs)</td>
<td>An object that can be decomposed into operations on given qubits.</td>
</tr>
<tr>
<td>SupportsDecompose(*args, **kwargs)</td>
<td>An object that can be parameterized by Symbols and resolved</td>
</tr>
<tr>
<td>SupportsDecomposeWithQubits(*args, **kwargs)</td>
<td>An effect that can be phased around the Z axis of target qubits.</td>
</tr>
<tr>
<td>SupportsQasm(*args, **kwargs)</td>
<td>An object that can be turned into QASM code.</td>
</tr>
<tr>
<td>SupportsQasmWithArgs(*args, **kwargs)</td>
<td>An object that can be turned into QASM code.</td>
</tr>
<tr>
<td>SupportsQasmWithArgsAndQubits(*args, **kwargs)</td>
<td>An object that can be turned into QASM code if it knows its qubits.</td>
</tr>
<tr>
<td>SupportsTraceDistanceBound(*args, **kwargs)</td>
<td>An effect with known bounds on how easy it is to detect.</td>
</tr>
<tr>
<td>SupportsUnitary(*args, **kwargs)</td>
<td>An object that may be describable by a unitary matrix.</td>
</tr>
</tbody>
</table>

**cirq.CircuitDiagramInfo**

```python
cirq.CircuitDiagramInfo
```

**class cirq.CircuitDiagramInfo** (wire_symbols: Tuple[str, ...], exponent: Any = 1, connected: bool = True)

Describes how to draw an operation in a circuit diagram.

```python
__init__(wire_symbols: Tuple[str, ...], exponent: Any = 1, connected: 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.

**Methods**

```python
```
**cirq.CircuitDiagramInfoArgs**

```python
class cirq.CircuitDiagramInfoArgs(known_qubits: Optional[Iterable[cirq.QubitId]],
                                   known_qubit_count: Optional[int],
                                   use_unicode_characters: bool, precision: Optional[int],
                                   qubit_map: Optional[Dict[cirq.QubitId, 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.

```python
__init__(known_qubits: Optional[Iterable[cirq.QubitId]],
         known_qubit_count: Optional[int],
         use_unicode_characters: bool, precision: Optional[int],
         qubit_map: Optional[Dict[cirq.QubitId, int]]) → None
```

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

**Methods**
`copy()`

`with_args(**kwargs)`

cirq.CircuitDiagramInfoArgs.copy

CircuitDiagramInfoArgs.copy()

cirq.CircuitDiagramInfoArgs.with_args

CircuitDiagramInfoArgs.with_args(**kwargs)

Attributes

`UNINFORMED_DEFAULT`

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.QubitId, str] = None, meas_key_id_map: Dict[str, str] = None)

__init__ (precision: int = 10, version: str = '2.0', qubit_id_map: Dict[cirq.QubitId, 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 creg QASM identifiers.

Methods

check_unused_args(used_args, args, kwargs)

convert_field(value, conversion)

format(**kwargs)

format_field(value, spec) Method of string.Formatter that specifies the output of format().

get_field(field_name, args, kwargs)

Continued on next page
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_value(key, args, kwargs)</code></td>
<td></td>
</tr>
<tr>
<td><code>parse(format_string)</code></td>
<td></td>
</tr>
<tr>
<td><code>validate_version(*supported_versions)</code></td>
<td></td>
</tr>
<tr>
<td><code>vformat(format_string, args, kwargs)</code></td>
<td></td>
</tr>
</tbody>
</table>

**circ.QasmArgs.check_unused_args**

QasmArgs.check_unused_args(used_args, args, kwargs)

**circ.QasmArgs.convert_field**

QasmArgs.convert_field(value, conversion)

**circ.QasmArgs.format**

QasmArgs.format(**kwargs)

**circ.QasmArgs.format_field**

QasmArgs.format_field(value: Any, spec: str) → str

  Method of string.Formatter that specifies the output of format().

**circ.QasmArgs.get_field**

QasmArgs.get_field(field_name, args, kwargs)

**circ.QasmArgs.get_value**

QasmArgs.get_value(key, args, kwargs)

**circ.QasmArgs.parse**

QasmArgs.parse(format_string)

**circ.QasmArgs.validate_version**

QasmArgs.validate_version(*supported_versions) → None

**circ.QasmArgs.vformat**

QasmArgs.vformat(format_string, args, kwargs)
cirq.QasmOutput


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, bytes, int]) 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.QasmOutput.valid_id_re

QasmOutput .valid_id_re = re.compile('[a-z][a-zA-Z0-9]*\s\s\s\s')

cirq.SupportsApplyUnitary

class cirq.SupportsApplyUnitary (*args, **kwargs)
An object that can be efficiently left-multiplied into tensors.

__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.

```python
__init__(*args, **kwargs)
```

**Methods**

---

### `cirq.SupportsDecomposeWithQubits`

**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.SupportsParameterization`

**class** `cirq.SupportsParameterization(*args, **kwargs)`

An object that can be parameterized by Symbols and resolved via a `ParamResolver`
Methods

---

cirq.SupportsPhase

class cirq.SupportsPhase(*args, **kwargs)
An effect that can be phased around the Z axis of target qubits.

Methods

---

cirq.SupportsQasm

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.

Methods

---

cirq.SupportsQasmWithArgs

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.

Methods
**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.

```
__init__(*args, **kwargs)
```

**Methods**

**class** cirq.SupportsTraceDistanceBound(*args, **kwargs)

An effect with known bounds on how easy it is to detect.

Used when deciding whether or not an operation is negligible. For example, the trace distance between the states before and after a `Z**0.00000001` operation is very close to 0, so it would typically be considered negligible.

```
__init__(*args, **kwargs)
```

**Methods**

**class** cirq.SupportsUnitary(*args, **kwargs)

An object that may be describable by a unitary matrix.

```
__init__(*args, **kwargs)
```

**Methods**
3.1.10 Optimization

Classes and methods for rewriting circuits.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ConvertToCzAndSingleGates</code>&lt;br&gt;<code>ignore_failures</code>, ...</td>
<td>Attempts to convert strange multi-qubit gates into CZ and single qubit gates.</td>
</tr>
<tr>
<td><code>DropEmptyMoments</code></td>
<td>Removes empty moments from a circuit.</td>
</tr>
<tr>
<td><code>DropNegligible</code>(tolerance)</td>
<td>An optimization pass that removes operations with tiny effects.</td>
</tr>
<tr>
<td><code>EjectPhasedPaulis</code>(tolerance)</td>
<td>Pushes X, Y, and PhasedX gates towards the end of the circuit.</td>
</tr>
<tr>
<td><code>EjectZ</code>(tolerance)</td>
<td>Pushes Z gates towards the end of the circuit.</td>
</tr>
<tr>
<td><code>ExpandComposite</code>(no_decomp,...)</td>
<td>An optimizer that expands composite operations via <code>cirq.decompose</code>.</td>
</tr>
<tr>
<td><code>google.optimized_for_xmon</code>(circuit,...)</td>
<td>Optimizes a circuit with XmonDevice in mind.</td>
</tr>
<tr>
<td><code>merge_single_qubit_gates_into_phased_x_z</code></td>
<td>Canonicalizes runs of single-qubit operations in a circuit.</td>
</tr>
<tr>
<td><code>MergeInteractions</code>(tolerance,...)</td>
<td>Combines series of adjacent one and two-qubit gates operating on a</td>
</tr>
<tr>
<td><code>MergeSingleQubitGates</code>(*, rewriter,...)</td>
<td>Optimizes runs of adjacent unitary 1-qubit operations.</td>
</tr>
<tr>
<td><code>PointOptimizationSummary</code>(clear_span,...)</td>
<td>A description of a local optimization to perform.</td>
</tr>
<tr>
<td><code>PointOptimizer</code>(post_clean_up,...)</td>
<td>Makes circuit improvements focused on a specific location.</td>
</tr>
<tr>
<td><code>single_qubit_matrix_to_gates</code>(mat, tolerance)</td>
<td>Implements a single-qubit operation with few gates.</td>
</tr>
<tr>
<td><code>single_qubit_matrix_to_pauli_rotations</code>(mat,...)</td>
<td>Implements a single-qubit operation with few rotations.</td>
</tr>
<tr>
<td><code>single_qubit_matrix_to_phased_x_z</code>(mat, atol)</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</code>(mat)</td>
<td>Decomposes a 2x2 unitary M into $U^{-1} \cdot \text{diag}(1, r) \cdot U \cdot \text{diag}(g, g)$.</td>
</tr>
<tr>
<td><code>two_qubit_matrix_to_operations</code>(q0, q1, mat,...)</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) → Optional[cirq.circuits.optimization_pass.PointOptimizationSummary]
```

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.

```
optimize_circuit(circuit)
```

Describes how to change operations near the given location.
cirq.DropEmptyMoments

class cirq.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) → None

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)
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) → None
Parameters tolerance – Maximum absolute error tolerance. The optimization is permit-
ted to simply drop negligible combinations of Z gates, with a threshold determined by this
tolerance.
Methods

```python
optimize_circuit(circuit)
```

cirq.EjectPhasedPaulis.optimize_circuit

```python
```

cirq.EjectZ

```python
cirq.EjectZ(tolerance: float = 0.0)

    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.

```python
__init__(tolerance: float = 0.0) → 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.
```

Methods

```python
optimize_circuit(circuit)
```

cirq.EjectZ.optimize_circuit

```python
```

cirq.ExpandComposite

```python
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.

```python
__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**

<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.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.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

class cirq.MergeInteractions(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, Iterable[Any]]] = <function MergeInteractions.<lambda>>)

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, Iterable[Any]]] = <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
optimization_at(circuit, index, op) Describes how to change operations near the given location.

optimize_circuit(circuit)

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

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

class cirq.MergeSingleQubitGates(*, rewriter: Optional[Callable[List[cirq.ops.raw_types.Operation], Union[cirq.ops.raw_types.Operation, Iterable[Any], None]] = None, synthesizer: Optional[Callable[[cirq.ops.raw_types.QubitId, numpy.ndarray], Union[cirq.ops.raw_types.Operation, Iterable[Any], None]] = None]

Optimizes runs of adjacent unitary 1-qubit operations.

__init__(*, rewriter: Optional[Callable[List[cirq.ops.raw_types.Operation], Union[cirq.ops.raw_types.Operation, Iterable[Any], None]] = None, synthesizer: Optional[Callable[[cirq.ops.raw_types.QubitId, numpy.ndarray], Union[cirq.ops.raw_types.Operation, Iterable[Any], None]] = None)

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”.

---

3.1. API Reference
• **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”.

### 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**

• **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:

```python
class cirq.PointOptimizer:
    post_clean_up: Callable[Sequence[cirq.ops.raw_types.Operation],
                                Union[cirq.ops.raw_types.Operation, Iterable[Any]]] = <function PointOptimizer.<lambda>>

Makes circuit improvements focused on a specific location.

__init__(post_clean_up: Callable[Sequence[cirq.ops.raw_types.Operation],
                                   Union[cirq.ops.raw_types.Operation,Iterable[Any]]] = <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**

---

```python
optimization_at(circuit, index, op) → Optional[cirq.circuits.optimization_pass.PointOptimizationSummary]
```

Describes how to change operations near the given location.

```python
optimize_circuit(circuit)
```

**cirq.PointOptimizer.optimization_at**

```python
PointOptimizer.optimization_at(circuit: cirq.circuits.circuit.Circuit, index: int,
                               op: cirq.ops.raw_types.Operation) → Optional[cirq.circuits.optimization_pass.PointOptimizationSummary]
```

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.

```python
cirq.PointOptimizer.optimize_circuit
circuit: cirq.circuits.circuit.Circuit
```

**cirq.single_qubit_matrix_to_gates**

```python
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**

```python
cirq.single_qubit_matrix_to_pauli_rotations(mat: numpy.ndarray, tolerance: 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.
- **tolerance** – A limit on the amount of 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**

```python
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 $U^{-1} \cdot \text{diag}(1, r) \cdot U \cdot \text{diag}(g, g)$.

$U$ 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** *(q0: cirq.ops.raw_types.QubitId, q1: cirq.ops.raw_types.QubitId, mat: numpy.ndarray, allow_partial_czs: bool, tolerance: float = 1e-08, clean_operations: bool = True) → List[cirq.ops.raw_types.Operation]*)  
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.  
• **tolerance** – A limit on the amount of 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.

### 3.1.11 Utilities

General utility methods, mostly related to performing relevant linear algebra operations and decompositions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>allclose_up_to_global_phase(a, b, rtol, ...)</code></td>
<td>Determines if $a \sim b \cdot \exp(it)$ for some $t$.</td>
</tr>
<tr>
<td><code>apply_matrix_to_slices(target, matrix, ...)</code></td>
<td>Left-multiplies an N$x$N matrix onto N slices of a numpy array.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bidiagonalize_real_matrix_pair_with_symmetric_products</code></td>
<td>Finds orthogonal matrices that diagonalize both mat1 and mat2.</td>
</tr>
<tr>
<td><code>bidiagonalize_unitary_with_special_orthogonals</code></td>
<td>Finds orthogonal matrices L, R such that L @ matrix @ R is diagonal.</td>
</tr>
<tr>
<td><code>canonicalize_half_turns(half_turns, float)</code></td>
<td>Wraps the input into the range (-1, +1].</td>
</tr>
<tr>
<td><code>chosen_angle_to_canonical_half_turns(...)</code></td>
<td>Returns a canonicalized half_turns based on the given arguments.</td>
</tr>
<tr>
<td><code>chosen_angle_to_half_turns(half_turns, ...)</code></td>
<td>Returns a half_turns value based on the given arguments.</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>block_diag(*blocks)</code></td>
<td>Concatenates blocks into a block diagonal 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>commutes(m1, m2, tolerance[, atol, equal_nan])</code></td>
<td>Determines if two matrices approximately commute.</td>
</tr>
<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(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(*, picos, float] = 0, nanos, float] = 0)</code></td>
<td>A time delta that supports picosecond accuracy.</td>
</tr>
<tr>
<td><code>is_diagonal(matrix, tolerance[, atol, equal_nan])</code></td>
<td>Determines if a matrix is a approximately diagonal.</td>
</tr>
<tr>
<td><code>is_hermitian(matrix, tolerance[, 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 Hermitian.</td>
</tr>
<tr>
<td><code>is_orthogonal(matrix, tolerance[, atol, . . .])</code></td>
<td>Determines if a matrix is approximately orthogonal.</td>
</tr>
<tr>
<td><code>is_special_orthogonal(matrix, tolerance[, . . .])</code></td>
<td>Determines if a matrix is approximately special orthogonal.</td>
</tr>
<tr>
<td><code>is_special_unitary(matrix, tolerance[, . . .])</code></td>
<td>Determines if a matrix is approximately unitary with unit determinant.</td>
</tr>
<tr>
<td><code>is_unitary(matrix, tolerance[, atol, equal_nan])</code></td>
<td>Determines if a matrix is approximately unitary.</td>
</tr>
<tr>
<td><code>kak_canonicalize_vector(x, y, z)</code></td>
<td>Canonicalizes an XX/YY/ZZ interaction by swap/negate/shift-ing axes.</td>
</tr>
<tr>
<td><code>kak_decomposition(mat, tolerance[, atol, . . .])</code></td>
<td>Decomposes a 2-qubit unitary into 1-qubit ops and XX/YY/ZZ interactions.</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(*matrices)</code></td>
<td>Computes the kronecker product of a sequence of matrices.</td>
</tr>
<tr>
<td><code>kron_factor_4x4_to_2x2s(matrix, tolerance[, . . .])</code></td>
<td>Splits a 4x4 matrix U = kron(A, B) into A, B, and a global factor.</td>
</tr>
<tr>
<td><code>kron_with_controls(*matrices)</code></td>
<td>Computes the kronecker product of a sequence of matrices and controls.</td>
</tr>
<tr>
<td><code>map_eigenvalues(matrix, func, complex[, . . .])</code></td>
<td>Applies a function to the eigenvalues of a matrix.</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>so4_to_magic_su2s(mat, tolerance[, atol, . . .])</code></td>
<td>Finds 2x2 special-unitaries A, B where mat = Mag.H @ kron(A, B) @ Mag.</td>
</tr>
<tr>
<td><code>Symbol(name)</code></td>
<td>A constant plus the runtime value of a parameter with a given key.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 110 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>targeted_left_multiply(left_matrix, ...)</td>
<td>Left-multiplies the given axes of the target tensor by the given matrix.</td>
</tr>
<tr>
<td>TextDiagramDrawer()</td>
<td>A utility class for creating simple text diagrams.</td>
</tr>
<tr>
<td>Timestamp(*, picos, float] = 0, nanos, ...)</td>
<td>A location in time with picosecond accuracy.</td>
</tr>
<tr>
<td>Tolerance(rtol, atol, equal_nan)</td>
<td>Specifies thresholds for doing approximate accuracy.</td>
</tr>
<tr>
<td>value_equality(cls, *, unhashable, ...)</td>
<td>Implements eq/ne/hash via a value_equality_values 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 \( a \approx b \exp(it) \) 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: List[Union[int, slice, ellipsis, Sequence[Union[int, slice, ellipsis]]]], *, out: Optional[numpy.ndarray] = None) → numpy.ndarray

Left-multiplies an NxN matrix onto N slices of a numpy array.

**Example**

The 4x4 matrix of a fractional SWAP gate can be expressed as

\[
\begin{bmatrix}
1 & \exp(it) & 1 \\
\exp(it) & 1 & 1 \\
1 & 1 & 1
\end{bmatrix}
\]

Where \( X \) is the 2x2 Pauli X gate and \( t \) is the power of the swap with \( t=1 \) being a full swap. \( \exp(it) \) 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:

```python
def fractional_swap(target):
    assert target.shape == (4,)
    return apply_matrix_to_slices(
```

(continues on next page)
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.

```python
cirq.bidiagonalize_real_matrix_pair_with_symmetric_products(mat1: numpy.ndarray, mat2: numpy.ndarray, tolerance: cirq.linalg.tolerance.Tolerance = Tolerance(rtol=1e-05, atol=1e-08, equal_nan=False)) → 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.
- `tolerance` – Numeric error thresholds.

Returns A tuple (L, R) of two orthogonal matrices, such that both L @ mat1 @ R and L @ mat2 @ R are diagonal matrices.

Raises ValueError – Matrices don’t meet preconditions (e.g. not real).
cirq.bidiagonalize_unitary_with_special_orthogonals

cirq.bidiagonalize_unitary_with_special_orthogonals(mat: numpy.ndarray, tolerance:
cirq.linalg.tolerance.Tolerance =
Tolerance(rtol=1e-05, atol=1e-
08, equal_nan=False) ) → Tu-
ple[numpy.ndarray, numpy.array,
numpy.ndarray]

Finds orthogonal matrices L, R such that L @ matrix @ R is diagonal.

Parameters

• mat – A unitary matrix.
• tolerance – Numeric error thresholds.

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.canonicalize_half_turns

cirq.canonicalize_half_turns(half_turns: Union[cirq.value.symbol.Symbol, float]) →
Union[cirq.value.symbol.Symbol, float]

Wraps the input into the range (-1, +1].

cirq.chosen_angle_to_canonical_half_turns

cirq.chosen_angle_to_canonical_half_turns(half_turns: Union[cirq.value.symbol.Symbol,
float, None] = None, rads: Optional[float] = None, degs: Optional[float]
= None, default: float = 1.0) →
Union[cirq.value.symbol.Symbol, float]

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[cirq.value.symbol.Symbol, float, None] = None, rads: Optional[float] = None, degs: Optional[float] = None, default: float = 1.0) → Union[cirq.value.symbol.Symbol, float]
```

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.slice_for_qubits_equal_to

```python
cirq.slice_for_qubits_equal_to(target_qubit_axes: Sequence[int], little_endian_qureg_value: int) → 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).

**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)).

**Returns** An index object that will slice out a mutable view of the desired subset of a tensor.

cirq.block_diag
cirq.block_diag(*blocks) \(\to\) 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.match_global_phase
cirq.match_global_phase(a: numpy.ndarray, b: numpy.ndarray) \(\to\) 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 \(a == b*\exp(i \cdot t)\) for some \(t\).

cirq.commutes
cirq.commutes(m1: numpy.ndarray, m2: numpy.ndarray, tolerance: cirq.linalg.tolerance.Tolerance =
\(\text{Tolerance}(\text{rtol} = 1e-05, \text{atol} = 1e-08, \text{equal_nan} = False)) \(\to\) 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.
- **tolerance** – The per-matrix-entry tolerance on equality.

**Returns** Whether the two matrices have compatible sizes and a commutator equal to zero within tolerance.
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, tolerance: cirq.linalg.tolerance.Tolerance = Tolerance(rtol=1e-05, atol=1e-08, equal_nan=False)) → 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.

• tolerance – Numeric error thresholds.

Returns An orthogonal matrix P such that P.T @ symmetric_matrix @ P is diagonal and P.T @ diagonal_matrix @ P = 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, tolerance: cirq.linalg.tolerance.Tolerance = Tolerance(rtol=1e-05, atol=1e-08, equal_nan=False)) → numpy.ndarray

Returns an orthogonal matrix that diagonalizes the given matrix.

Parameters

• matrix – A real symmetric matrix to diagonalize.

• tolerance – Numeric error thresholds.

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) \rightarrow \text{Union[\text{float, complex, numpy.ndarray}]} 
Computes the dot/matrix product of a sequence of values.

A *args version of np.linalg.multi_dot.

**Parameters**
- *values* – The values to combine with the dot/matrix product.

**Returns**
The resulting value or matrix.

cirq.Duration

class cirq.Duration(*, picos: Union[int, float] = 0, nanos: Union[int, float] = 0)
A time delta that supports picosecond accuracy.

__init__(*picos: Union[int, float] = 0, nanos: Union[int, float] = 0*) \rightarrow \text{None} 
If both picos and nanos are specified, their contributions are added.

**Parameters**
- **picos** – A number of picoseconds to add to the time delta.
- **nanos** – A number of nanoseconds to add to the time delta.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>total_nanos()</td>
<td>Returns the number of nanoseconds that the duration spans.</td>
</tr>
<tr>
<td>total_picos()</td>
<td>Returns the number of picoseconds that the duration spans.</td>
</tr>
</tbody>
</table>

**cirq.Duration.total_nanos**

Duration.total_nanos() \rightarrow float 
Returns the number of nanoseconds that the duration spans.

**cirq.Duration.total_picos**

Duration.total_picos() \rightarrow float 
Returns the number of picoseconds that the duration spans.

cirq.is_diagonal

cirq.is_diagonal(matrix: numpy.ndarray, tolerance: cirq.linalg.tolerance.Tolerance = 
Tolerance(rtol=1e-05, atol=1e-08, equal_nan=False)) \rightarrow \text{bool} 
Determines if a matrix is a approximately diagonal.

A matrix is diagonal if i\neq j implies m[i,j]==0.

**Parameters**
- **matrix** – The matrix to check.
- **tolerance** – The per-matrix-entry tolerance on equality.

**Returns** Whether the matrix is diagonal within the given tolerance.

### `cirq.is_hermitian`

```python
cirq.is_hermitian(matrix: numpy.ndarray, tolerance: cirq.linalg.tolerance.Tolerance = Tolerance(rtol=1e-05, atol=1e-08, equal_nan=False)) → 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.
- **tolerance** – The per-matrix-entry tolerance on equality.

**Returns** Whether the matrix is Hermitian within the given tolerance.

### `cirq.is_negligible_turn`

```python
cirq.is_negligible_turn(turns: float, tolerance: float) → bool
```

### `cirq.is_orthogonal`

```python
cirq.is_orthogonal(matrix: numpy.ndarray, tolerance: cirq.linalg.tolerance.Tolerance = Tolerance(rtol=1e-05, atol=1e-08, equal_nan=False)) → 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.
- **tolerance** – The per-matrix-entry tolerance on equality.

**Returns** Whether the matrix is orthogonal within the given tolerance.

### `cirq.is_special_orthogonal`

```python
cirq.is_special_orthogonal(matrix: numpy.ndarray, tolerance: cirq.linalg.tolerance.Tolerance = Tolerance(rtol=1e-05, atol=1e-08, equal_nan=False)) → 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.
tolerance – The per-matrix-entry 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, tolerance: cirq.linalg.tolerance.Tolerance = Tolerance(rtol=1e-05, atol=1e-08, equal_nan=False)) → 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.
• tolerance – The per-matrix-entry 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, tolerance: cirq.linalg.tolerance.Tolerance = Tolerance(rtol=1e-05, atol=1e-08, equal_nan=False)) → 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.
• tolerance – The per-matrix-entry 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) → 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.

Returns

The canonicalized decomposition, with vector coefficients (x2, y2, z2) satisfying:

\[
\begin{align*}
0 & \quad \text{abs}(z2) \quad y2 \quad x2 \quad \pi/4 \quad z2 \quad -\pi/4 \\
\end{align*}
\]

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(mat: numpy.ndarray, tolerance: cirq.linalg.tolerance.Tolerance = Tolerance(rtol=1e-05, atol=1e-08, equal_nan=False)) \rightarrow cirq.linalg.decompositions.KakDecomposition

Decomposes a 2-qubit unitary into 1-qubit ops and XX/YY/ZZ interactions.

Parameters

- **mat** – The 4x4 unitary matrix to decompose.
- **tolerance** – Per-matrix-entry tolerance on equality.

Returns

A `cirq.KakDecomposition` canonicalized such that the interaction coefficients x, y, z satisfy:

\[ 0 \leq \text{abs}(z) \leq y \leq x \leq \pi/4 \]

z ≤ -\pi/4

Raises

- `ValueError` – Bad matrix.
- `ArithmeticError` – Failed to perform the decomposition.

References


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

\[ U = g \cdot (a_1 \ a_0) \cdot \exp(i \cdot (x \cdot XX + y \cdot YY + z \cdot ZZ)) \cdot (b_1 \ b_0) \]

This class stores \( g \), \( (b_0, b_1) \), \( (x, y, z) \), and \( (a_0, a_1) \).

**global_phase**

g from the above equation.

**single_qubit_operations_before**

\( b_0, b_1 \) from the above equation.

**interaction_coefficients**

\( x, y, z \) from the above equation.

**single_qubit_operations_after**

\( a_0, a_1 \) 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.

\[
U = g \cdot (a_1 \ a_0) \cdot \exp(i \cdot (x \cdot XX + y \cdot YY + z \cdot ZZ)) \cdot (b_1 \ b_0)
\]

**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**

```python
cirq.kron
```

`cirq.kron(*matrices) → numpy.ndarray`

Computes the kronecker product of a sequence of matrices.

A *args version of lambda args: functools.reduce(np.kron, args).

**Parameters** *matrices – The matrices and controls to combine with the kronecker product.

**Returns** The resulting matrix.

```python
cirq.kron_factor_4x4_to_2x2s
```

`cirq.kron_factor_4x4_to_2x2s(matrix: numpy.ndarray, tolerance: cirq.linalg.tolerance.Tolerance = Tolerance( atol=1e-08, equal_nan=False)) → 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.

**Parameters**

- `matrix` – The 4x4 unitary matrix to factor.
- `tolerance` – Acceptable numeric error thresholds.

**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(*matrices) → numpy.ndarray
Computes the kronecker product of a sequence of matrices and controls.

Use linalg.CONTROL_TAG to represent controls. Any entry of the output matrix 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.

Parameters
• *matrices – The matrices and controls to combine with the kronecker product.

Returns The resulting matrix.

cirq.map_eigenvalues

cirq.map_eigenvalues(matrix: numpy.ndarray, func: Callable[complex, complex], tolerance: cirq.linalg.tolerance.Tolerance = Tolerance(rtol=1e-05, atol=1e-08, equal_nan=False)) → 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.
• tolerance – Thresholds used when separating eigenspaces.

Returns The transformed matrix.

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.so4_to_magic_su2s

cirq.so4_to_magic_su2s(mat: numpy.ndarray, tolerance: cirq.linalg.tolerance.Tolerance = Tolerance(rtol=1e-05, atol=1e-08, equal_nan=False)) → 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:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>i</td>
</tr>
<tr>
<td>0</td>
<td>i</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>i</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-i</td>
</tr>
</tbody>
</table>

**(times sqrt(0.5) to normalize)**

**Parameters**

• **mat** – A real 4x4 orthogonal matrix.

• **tolerance** – Per-matrix-entry tolerance on equality.

**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.Symbol

class cirq.Symbol(name: str)

A constant plus the runtime value of a parameter with a given key.

**name**

The non-empty name of a parameter to lookup at runtime and add to the constant offset.

**__init__**(name: str) → None

Initializes a Symbol with the given name.

**Parameters** name – The name of a parameter.

**Methods**

cirq.targeted_left_multiply

cirq.targeted_left_multiply(left_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, 2)`, and a 2-qubit unitary operation `op` with shape `(2, 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:

```python
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:

```python
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
__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.

Continued on next page
Table 114 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>force_vertical_padding_after(index, padding)</code></td>
<td>Change the padding after the given row.</td>
</tr>
<tr>
<td><code>grid_line(x1, y1, x2, y2, emphasize)</code></td>
<td>Adds a vertical or horizontal line from (x1, y1) to (x2, y2).</td>
</tr>
<tr>
<td><code>height()</code></td>
<td>Determines how many entry rows are in the diagram.</td>
</tr>
<tr>
<td><code>horizontal_line(y, x1, x2, emphasize)</code></td>
<td>Adds a line from (x1, y) to (x2, y).</td>
</tr>
<tr>
<td><code>insert_empty_columns(x, amount)</code></td>
<td>Insert a number of columns after the given column.</td>
</tr>
<tr>
<td><code>insert_empty_rows(y, amount)</code></td>
<td>Insert a number of rows after the given row.</td>
</tr>
<tr>
<td><code>render(horizontal_spacing, vertical_spacing,...)</code></td>
<td>Outputs text containing the diagram.</td>
</tr>
<tr>
<td><code>transpose()</code></td>
<td>Returns the same diagram, but mirrored across its diagonal.</td>
</tr>
<tr>
<td><code>vertical_line(x, y1, y2, emphasize)</code></td>
<td>Adds a line from (x, y1) to (x, y2).</td>
</tr>
<tr>
<td><code>width()</code></td>
<td>Determines how many entry columns are in the diagram.</td>
</tr>
<tr>
<td><code>write(x, y, text, transposed_text)</code></td>
<td>Adds text to the given location.</td>
</tr>
</tbody>
</table>

**cirq.TextDiagramDrawer.content_present**

`TextDiagramDrawer.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**

`TextDiagramDrawer.force_horizontal_padding_after(index: int, padding: int) → None`  
Change the padding after the given column.

**cirq.TextDiagramDrawer.force_vertical_padding_after**

`TextDiagramDrawer.force_vertical_padding_after(index: int, padding: int) → None`  
Change the padding after the given row.

**cirq.TextDiagramDrawer.grid_line**

`TextDiagramDrawer.grid_line(x1: int, y1: int, x2: int, y2: int, emphasize: bool = False)`  
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.

**Raises** `ValueError` – If line is neither horizontal nor vertical.

**cirq.TextDiagramDrawer.height**

`TextDiagramDrawer.height() → int`  
Determines how many entry rows are in the diagram.
cirq.TextDiagramDrawer.horizontal_line

TextDiagramDrawer.horizontal_line(y: int, x1: int, x2: int, emphasize: bool = False) → None

Adds a line from (x1, y) to (x2, y).

cirq.TextDiagramDrawer.insert_empty_columns

TextDiagramDrawer.insert_empty_columns(x: int, amount: int = 1) → None

Insert a number of columns after the given column.

cirq.TextDiagramDrawer.insert_empty_rows

TextDiagramDrawer.insert_empty_rows(y: int, amount: int = 1) → None

Insert a number of rows after the given row.

cirq.TextDiagramDrawer.render

TextDiagramDrawer.render(horizontal_spacing: int = 1, vertical_spacing: int = 1, crossing_char: str = None, use_unicode_characters: bool = True) → str

Outputs text containing the diagram.

cirq.TextDiagramDrawer.transpose

TextDiagramDrawer.transpose() → cirq.circuits.text_diagram_drawer.TextDiagramDrawer

Returns the same diagram, but mirrored across its diagonal.

cirq.TextDiagramDrawer.vertical_line

TextDiagramDrawer.vertical_line(x: int, y1: int, y2: int, emphasize: bool = False) → None

Adds a line from (x, y1) to (x, y2).

cirq.TextDiagramDrawer.width

TextDiagramDrawer.width() → int

Determines how many entry columns are in the diagram.

cirq.TextDiagramDrawer.write

TextDiagramDrawer.write(x: int, y: int, text: str, transposed_text: str = None) → 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).
• **transposted_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**

```
raw_picos() \rightarrow float
```

The timestamp’s location in picoseconds from arbitrary time zero.

### cirq.Timestamp.raw_picos

```
Timestamp.raw_picos() \rightarrow float
```

The timestamp’s location in picoseconds from arbitrary time zero.

### cirq.Tolerance

**class** cirq.Tolerance(rtol: float = 1e-05, atol: float = 1e-08, equal_nan: bool = False)**

Specifies thresholds for doing approximate equality.

```
__init__(rtol: float = 1e-05, atol: float = 1e-08, equal_nan: bool = False) \rightarrow None
```

Initializes a Tolerance instance with the specified parameters.

**Notes**

Matrix Comparisons (methods beginning with “all_”) are done by numpy.allclose, which considers x and y to be close when abs(x - y) <= atol + rtol * abs(y). See numpy.allclose’s documentation for more details. The scalar methods perform the same calculations without the numpy matrix construction.
Parameters

- `rtol` – Relative tolerance.
- `atol` – Absolute tolerance.
- `equal_nan` – Whether NaNs are equal to each other.

Methods

- `all_close(a, b)`
- `all_near_zero(a)`
- `all_near_zero_mod(a, period)`
- `close(a, b)`
- `near_zero(a)`
- `near_zero_mod(a, period)`

`cirq.Tolerance.all_close`

Tolerance.all_close(a, b)

`cirq.Tolerance.all_near_zero`

Tolerance.all_near_zero(a)

`cirq.Tolerance.all_near_zero_mod`

Tolerance.all_near_zero_mod(a, period)

`cirq.Tolerance.close`

Tolerance.close(a, b)

`cirq.Tolerance.near_zero`

Tolerance.near_zero(a)

`cirq.Tolerance.near_zero_mod`

Tolerance.near_zero_mod(a, period)

Attributes

- `DEFAULT`
- `ZERO`
cirq.Tolerance.DEFAULT

Tolerance.DEFAULT = Tolerance(rtol=1e-05, atol=1e-08, equal_nan=False)

cirq.Tolerance.ZERO

Tolerance.ZERO = Tolerance(rtol=0, atol=0, equal_nan=False)

cirq.value_equality

cirq.value_equality (cls: type = None, *, unhashable: bool = False, distinct_child_types: bool = False, approximate: bool = False) → 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.

• approximate – When set, the decorated class will be enhanced with _approx_eq_ implementation and thus start to support the SupportsApproximateEquality protocol.
3.1.12 Experiments

Utilities for running experiments on hardware, or producing things required to run experiments.

```python
generate_supremacy_circuit_google_v2(qubits, ...) Generates Google Random Circuits v2 as in github.com/sboixo/GRCS
generate_supremacy_circuit_google_v2_bristlecone(...) Generates Google Random Circuits v2 in Bristlecone.
generate_supremacy_circuit_google_v2_grid(...) Generates Google Random Circuits v2 as in github.com/sboixo/GRCS
```

cirq.generate_supremacy_circuit_google_v2

cirq.generate_supremacy_circuit_google_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 cz_v2.
See also https://arxiv.org/abs/1807.10749

Parameters

- **qubits** – qubit grid in which to generate the circuit.
- **cz_depth** – number of layers with CZ gates.
- **seed** – seed for the random instance.

Returns A circuit corresponding to instance \( \text{inst}_{\text{n\_rows}\times\text{n\_cols}\times\text{cz\_depth+1}}\{\text{seed}\} \)

The mapping of qubits is cirq.GridQubit(j,k) -> q[j*n_cols+k] (as in the QASM mapping)

cirq.generate_supremacy_circuit_google_v2_bristlecone

cirq.generate_supremacy_circuit_google_v2_bristlecone (n_rows: int, cz_depth: int, seed: int) → cirq.circuits.circuit.Circuit

Generates Google Random Circuits v2 in Bristlecone.
See also https://arxiv.org/abs/1807.10749

Parameters

- **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.
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\times n_cols\times cz_depth+1}_{seed}

The mapping of qubits is cirq.GridQubit(j,k) -> q[j\times n_cols+k] (as in the QASM mapping)

3.1.13 Google

Functionality specific to quantum hardware and services from Google.

- **google.AnnealSequenceSearchStrategy(...)** Linearized sequence search using simulated annealing method.
- **google.GreedySequenceSearchStrategy(algorithm)** Greedy search method for linear sequence of qubits on a chip.
- **google.Bristlecone**
- **google.ConvertToXmonGates(...)** Attempts to convert strange gates into XmonGates.
- **google.Engine(api_key, api, version,...)** Runs programs via the Quantum Engine API.
- **google.engine_from_environment()** Returns an Engine instance configured using environment variables.
- **google.Foxtail**
- **google.gate_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(project_id, program_id,...)** Configuration for a program and job to run on the Quantum Engine API.
- **google.LinePlacementStrategy(choice and options for the line placement calculation method).**
- **google.LineOnDevice(device, length, method)** Searches for linear sequence of qubits on device.
- **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.
- **google.schedule_to_proto_dicts(schedule)** Convert a schedule into an iterable of proto dictionaries.
- **google.unpack_results(data, repetitions,...)** Unpack data from a bitstring into individual measurement results.
- **google.xmon_op_from_proto_dict(proto_dict)** Convert the proto dictionary to the corresponding operation.

Continued on next page
Table 119 – continued from previous page

<table>
<thead>
<tr>
<th>google.XmonDevice</th>
<th>A device with qubits placed in a grid.</th>
</tr>
</thead>
<tbody>
<tr>
<td>google.XmonOptions</td>
<td>XmonOptions for the XmonSimulator.</td>
</tr>
<tr>
<td>google.XmonSimulator</td>
<td>XmonSimulator for Xmon class quantum circuits.</td>
</tr>
<tr>
<td>google.XmonStepResult</td>
<td>Results of a step of the simulator.</td>
</tr>
</tbody>
</table>

**cirq.google.AnnealSequenceSearchStrategy**

```python
class cirq.google.AnnealSequenceSearchStrategy:

class cirq.google.AnnealSequenceSearchStrategy:
    def __init__(self, trace_func: Callable[[List[List[cirq.devices.grid_qubit.GridQubit]], float, float, float, bool], None] = None, seed: int = None):
        Linearized sequence search using simulated annealing method.

        TODO: This line search strategy is still work in progress and requires efficiency improvements.

        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(self, device, length):
    Runs line sequence search.
```

**cirq.google.AnnealSequenceSearchStrategy.place_line**

```
cirq.google.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.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.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.

```python
__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**

```python
ConvertToXmonGates.convert(op: cirq.ops.raw_types.Operation) → List[cirq.ops.raw_types.Operation]
```

**cirq.google.ConvertToXmonGates.optimization_at**

```python
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

class cirq.google.Engine(api_key: str, api: str = 'quantum', version: str = 'v1alpha1',
                        default_project_id: Optional[str] = None, discovery_url: Optional[str] = None,
                        default_gcs_prefix: Optional[str] = None, **kwargs)
    Runs programs via the Quantum Engine API.

This class has methods for creating programs and jobs that execute on Quantum Engine:
run
run_sweep

Another set of methods return information about programs and jobs that have been previously created on the Quantum Engine:
get_program
get_job
get_job_results

Finally, the engine has methods to update existing programs and jobs:
cancel_job
set_program_labels
add_program_labels
remove_program_labels
set_job_labels
add_job_labels
remove_job_labels

__init__(api_key: str, api: str = 'quantum', version: str = 'v1alpha1',
         default_project_id: Optional[str] = None, discovery_url: Optional[str] = None,
         default_gcs_prefix: Optional[str] = None, **kwargs) → None
    Engine service client.

Parameters

* api_key – API key to use to retrieve discovery doc.
* api – API name.
* version – API version.
* default_project_id – A fallback project_id to use when one isn’t specified in the JobConfig given to ‘run’ methods. See JobConfig for more information on project_id.
* discovery_url – Discovery url for the API. If not supplied, uses Google’s default api.googleapis.com endpoint.
• **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.

## Methods

**add_job_labels**(job_resource_name, labels, str)

**add_program_labels**(program_resource_name, ...

**cancel_job**(job_resource_name) Cancels the given job.

**get_job**(job_resource_name) Returns metadata about a previously created job.

**get_job_results**(job_resource_name) Returns the actual results (not metadata) of a completed job.

**get_program**(program_resource_name) Returns the previously created quantum program.

**implied_job_config**(job_config)

**program_as_schedule**(program, ...) 

**remove_job_labels**(job_resource_name, label_keys)

**remove_program_labels**(program_resource_name, ...

**run**(*, program, ...) Runs the supplied Circuit or Schedule via Quantum Engine.

**run_sweep**(*, program, ...) Runs the supplied Circuit or Schedule via Quantum Engine.

**set_job_labels**(job_resource_name, labels, str)

**set_program_labels**(program_resource_name, ...

## 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_resource_name: 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.
circq.google.Engine.get_job

Engine.get_job(job_resource_name: str) \rightarrow 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.

circq.google.Engine.get_job_results

Engine.get_job_results(job_resource_name: str) \rightarrow 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.

circq.google.Engine.get_program

Engine.get_program(program_resource_name: str) \rightarrow Dict

Returns the previously created quantum program.

Params:
program_resource_name: A string of the form projects/project_id/programs/program_id.

Returns A dictionary containing the metadata and the program.

circq.google.Engine.implied_job_config

Engine.implied_job_config(job_config: Optional[cirq.google.engine.engine.JobConfig]) \rightarrow
cirq.google.engine.engine.JobConfig
**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_resource_name: 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.
- **job_config** – Configures the names of programs and jobs.
- **param_resolver** – Parameters to run with the program.
- **repetitions** – The number of repetitions to simulate.
- **priority** – The priority to run at, 0-100.
- **target_route** – The engine route to run against.

**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.
- `job_config` – Configures the names of programs and jobs.
- `params` – Parameters to run with the program.
- `repetitions` – The number of circuit repetitions to run.
- `priority` – The priority to run at, 0-100.
- `target_route` – The engine route to run against.

**Returns** An EngineJob. If this is iterated over it returns a list of TrialResults, one for each parameter sweep.

```python
cirq.google.Engine.set_job_labels
```

```python
Engine.set_job_labels(job_resource_name: str, labels: Dict[str, str])
```

```python
cirq.google.Engine.set_program_labels
```

```python
Engine.set_program_labels(program_resource_name: str, labels: Dict[str, str])
```

```python
cirq.google.engine_from_environment
```

```python
cirq.google.engine_from_environment() \rightarrow 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.

QUANTUM_ENGINE_API_KEY: An API key for the google cloud project named by QUANTUM_ENGINE_PROJECT.

**Raises** EnvironmentError – The environment variables are not set.

```python
cirq.google.Foxtail
```

```
cirq.google.Foxtail = cirq.google.Foxtail
```

```python
cirq.google.gate_to_proto_dict
```

```python
cirq.google.gate_to_proto_dict(gate: cirq.ops.raw_types.Gate, qubits: Tuple[cirq.ops.raw_types.QubitId, ...]) \rightarrow Dict
```
cirq.google.is_native_xmon_op

cirq.google.is_native_xmon_op(op: cirq.ops.raw_types.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 program and job to run on the Quantum Engine API.

Quantum engine has two resources: programs and jobs. Programs live under cloud projects. Every program may have many jobs, which represent scheduled or terminated programs executions. Program and job resources have string names. This object contains the information necessary to create a program and then create a job on Quantum Engine, hence running the program.

Program ids are of the form
projects/project_id/programs/program_id
while job ids are of the form
projects/project_id/programs/program_id/jobs/job_id

Configuration for a job that is run on Quantum Engine.

Requires project_id.

Parameters

• project_id – The project id string of the Google Cloud Project to use. Programs and Jobs will be created under this project id. If this is set to None, the engine’s default project id will be used instead. If that also isn’t set, calls will fail.

• program_id – Id of the program to create, defaults to a random version of ‘prog-ABCD’.

• 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_program – Explicit override for the program storage location.

• gcs_results – Explicit override for the results storage location.
Methods

.. _Methods:

.. _copy:

.. automethod:: cirq.google.JobConfig.copy()

.. _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:

.. 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.

.. _init__:

.. _Methods:

.. _place_line:

.. automethod:: cirq.google.LinePlacementStrategy.place_line(device: cirq.google.XmonDevice, length: int)

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(device: xmon_device.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(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(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>... 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 (proto_dict: Dict) → cirq.ops.raw_types.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**

- ValueError if the dictionary does not contain required values corresponding to the proto.

---

cirq.google.XmonDevice

```
class cirq.google.XmonDevice (measurement_duration: cirq.value.duration.Duration, 
                            exp_w_duration: cirq.value.duration.Duration, 
                            exp_11_duration: cirq.value.duration.Duration, 
                            qubits: Iterable[cirq.devices.grid_qubit.GridQubit]) 
```

A device with qubits placed in a grid. Neighboring qubits can interact.

```
__init__ (measurement_duration: cirq.value.duration.Duration, 
         exp_w_duration: cirq.value.duration.Duration, 
         exp_11_duration: cirq.value.duration.Duration, 
         qubits: Iterable[cirq.devices.grid_qubit.GridQubit]) → None
```

Initializes the description of an xmon device.

**Parameters**

- **measurement_duration** – The maximum duration of a measurement.
- **exp_w_duration** – The maximum duration of an ExpW operation.
- **exp_11_duration** – The maximum duration of an ExpZ operation.
- **qubits** – Qubits on the device, identified by their x, y location.

**Methods**

- **at**(row, col) Returns the qubit at the given position, if there is one, else None.
- **can_add_operation_into_moment**(operation, moment) Determines if it’s possible to add an operation into a moment.
- **col**(col) Returns the qubits in the given column, in ascending order.
- **decompose_operation**(operation) Returns a device-valid decomposition for the given operation.
- **duration_of**(operation) Returns the qubits that the given qubit can interact with.
- **row**(row) Returns the qubits in the given row, in ascending order.

Continued on next page
Table 126 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>validate_circuit(circuit)</code></td>
<td>Raises an exception if a circuit is not valid.</td>
</tr>
<tr>
<td><code>validate_gate(gate)</code></td>
<td>Raises an error if the given gate isn’t allowed.</td>
</tr>
<tr>
<td><code>validate_moment(moment)</code></td>
<td>Raises an exception if a moment is not valid.</td>
</tr>
<tr>
<td><code>validate_operation(operation)</code></td>
<td>Raises an exception if an operation is not valid.</td>
</tr>
<tr>
<td><code>validate_schedule(schedule)</code></td>
<td>Raises an exception if a schedule is not valid.</td>
</tr>
<tr>
<td><code>validate_scheduled_operation(schedule, ...)</code></td>
<td>Raises an exception if the scheduled operation is not valid.</td>
</tr>
</tbody>
</table>

**`cirq.google.XmonDevice.at`**

```python
XmonDevice.at(row: int, col: int) → Optional[cirq.devices.grid_qubit.GridQubit]

Returns the qubit at the given position, if there is one, else None.
```

**`cirq.google.XmonDevice.can_add_operation_into_moment`**

```python
XmonDevice.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.google.XmonDevice.col`**

```python
XmonDevice.col(col: int) → List[cirq.devices.grid_qubit.GridQubit]

Returns the qubits in the given column, in ascending order.
```

**`cirq.google.XmonDevice.decompose_operation`**

```python
XmonDevice.decompose_operation(operation: cirq.ops.raw_types.Operation) →
  Union[cirq.ops.raw_types.Operation, Iterable[Any]]

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.google.XmonDevice.duration_of

XmonDevice.duration_of(operation)

cirq.google.XmonDevice.neighbors_of

XmonDevice.neighbors_of(qubit: cirq.devices.grid_qubit.GridQubit)
Returns the qubits that the given qubit can interact with.

cirq.google.XmonDevice.row

XmonDevice.row(row: int) \rightarrow \text{List}[\text{cirq.devices.grid_qubit.GridQubit}]
Returns the qubits in the given row, in ascending order.

cirq.google.XmonDevice.validate_circuit

XmonDevice.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.

cirq.google.XmonDevice.validate_gate

XmonDevice.validate_gate(gate: cirq.ops.raw_types.Gate)
Raises an error if the given gate isn’t allowed.

Raises ValueError – Unsupported gate.

cirq.google.XmonDevice.validate_moment

XmonDevice.validate_moment(moment: cirq.ops.moment.Moment)
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.google.XmonDevice.validate_operation

XmonDevice.validate_operation(operation: cirq.ops.raw_types.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.
**cirq.google.XmonDevice.validate_schedule**

XmonDevice.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.google.XmonDevice.validate_scheduled_operation**

XmonDevice.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.google.XmonOptions**

```python
class cirq.google.XmonOptions(num_shards: int = None, min_qubits_before_shard: int = 18, use_processes: bool = False):

XmonOptions for the XmonSimulator.
```

**num_prefix_qubits**

Sharding of the wave function is performed over \(2^n\) number of qubits.

**min_qubits_before_shard**

Sharding will be done only for this number of qubits or more. The default is 18.

**use_processes**

Whether or not to use processes instead of threads. Processes can improve the performance slightly (varies by machine but on the order of 10 percent faster). However this varies significantly by architecture, and processes should not be used for interactive use on Windows.

```python
__init__(num_shards: int = None, min_qubits_before_shard: int = 18, use_processes: bool = False)
→ None

XmonSimulator options constructor.
```
Parameters

- **num_shards** – sharding will be done for the greatest value of a power of two less than this value. If None, the default will be used which is the smallest power of two less than or equal to the number of CPUs.

- **min_qubits_before_shard** – Sharding will be done only for this number of qubits or more. The default is 18.

- **use_processes** – Whether or not to use processes instead of threads. Processes can improve the performance slightly (varies by machine but on the order of 10 percent faster). However this varies significantly by architecture, and processes should not be used for interactive python use on Windows.

Methods

**cirq.google.XmonSimulator**

class cirq.google.XmonSimulator(\[options: cirq.google.sim.xmon_simulator.XmonOptions = None\])

XmonSimulator for Xmon class quantum circuits.

This simulator has different methods for different types of simulations.

For simulations that mimic the quantum hardware, the run methods are defined in the SimulatesSamples interface:

- run
- run_sweep

These methods do not return or give access to the full wave function.

To get access to the wave function during a simulation, including being able to set the wave function, the simulate methods are defined in the SimulatesFinalWaveFunction interface:

- simulate
- simulate_sweep
- simulate_moment_steps (for stepping through a circuit moment by moment)

```
__init__(options: cirq.google.sim.xmon_simulator.XmonOptions = None) → None

Construct a XmonSimulator.

Parameters options – XmonOptions configuring the simulation.
```

Methods

```
compute_displays(program, …) Computes displays in the supplied Circuit or Schedule.
```
Table 128 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute_displays_sweep</td>
<td>Computes displays in the supplied Circuit or Schedule.</td>
</tr>
<tr>
<td>compute_samples_displays</td>
<td>Computes SamplesDisplays in the supplied Circuit or Schedule.</td>
</tr>
<tr>
<td>compute_samples_displays_sweep</td>
<td>Computes SamplesDisplays in the supplied Circuit or Schedule.</td>
</tr>
<tr>
<td>run</td>
<td>Runs the supplied Circuit or Schedule, mimicking quantum hardware.</td>
</tr>
<tr>
<td>run_sweep</td>
<td>Runs the supplied Circuit or Schedule, mimicking quantum hardware.</td>
</tr>
<tr>
<td>simulate</td>
<td>Simulates the supplied Circuit or Schedule.</td>
</tr>
<tr>
<td>simulate_moment_steps</td>
<td>Returns an iterator of StepResults for each moment simulated.</td>
</tr>
<tr>
<td>simulate_sweep</td>
<td>Simulates the supplied Circuit or Schedule.</td>
</tr>
</tbody>
</table>

**cirq.google.XmonSimulator.compute_displays**


Computes displays in the supplied Circuit or Schedule.

**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 used to define the order of amplitudes in the wave function.
- **initial_state** – If an int, the state is set to the computational basis state corresponding to this state. Otherwise if this is a np.ndarray it is the full initial state. In this case it must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an appropriate dtype for the simulator.

**Returns** ComputeDisplaysResult for the simulation.
### `cirq.google.XmonSimulator.compute_displays_sweep`

The method `compute_displays_sweep` accepts a variety of parameters that can be defined as:

- `program`: The circuit or schedule to simulate.
- `params`: Parameters to run with the program.
- `qubit_order`: Determines the canonical ordering of the qubits used to define the order of amplitudes in the wave function.
- `initial_state`: If an int, the state is set to the computational basis state corresponding to this state. Otherwise, if this is a numpy.ndarray it is the full initial state. In this case it must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an appropriate dtype for the simulator.

This method computes displays in the supplied Circuit or Schedule and 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 used to define the order of amplitudes in the wave function.
- **initial_state** – If an int, the state is set to the computational basis state corresponding to this state. Otherwise, if this is a numpy.ndarray it is the full initial state. In this case it must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an appropriate dtype for the simulator.

### Returns

List of `cirq.study.compute_displays_result.ComputeDisplaysResult` for this run, one for each possible parameter resolver.

### `cirq.google.XmonSimulator.compute_samples_displays`

The method `compute_samples_displays` is similar in its parameters and functionality to `compute_displays_sweep`, but it specifically computes a `cirq.study.compute_displays_result.ComputeDisplaysResult` for the simulation.

### Parameters

- **program** – The circuit or schedule to simulate.
- **param_resolver** – Parameters to run with the program.

### Returns

`cirq.study.compute_displays_result.ComputeDisplaysResult` for the simulation.
**cirq.google.XmonSimulator.compute_samples_displays_sweep**

\[
\text{XmonSimulator.} \text{compute_samples_displays_sweep}(\text{program:}) = \text{Union[cirq.circuits.circuit.Circuit, cirq.schedules.schedule.Schedule], params:} \text{Union[cirq.study.resolver.ParamResolver, cirq.study.sweeps.Sweep], List[cirq.study.compute_displays_result.ComputeDisplaysResult]}
\]

Computes SamplesDisplays in the supplied Circuit or Schedule.

In contrast to `compute_displays`, this allows for sweeping over different parameter values.

**Parameters**

- **program** – The circuit or schedule to simulate.
- **params** – Parameters to run with the program.

**Returns** List of ComputeDisplaysResults for this run, one for each possible parameter resolver.

**cirq.google.XmonSimulator.run**

\[
\text{XmonSimulator.} \text{run}(\text{program:}) = \text{Union[cirq.circuits.circuit.Circuit, cirq.schedules.schedule.Schedule], param_resolver: Optional[cirq.study.resolver.ParamResolver] = None, repetitions: int = 1) = \text{cirq.study.trial_result.TrialResult}
\]

Runs the supplied Circuit or Schedule, mimicking quantum hardware.

**Parameters**

- **program** – The circuit or schedule to simulate.
- **param_resolver** – Parameters to run with the program.
- **repetitions** – The number of repetitions to simulate.

**Returns** TrialResult for a run.

**cirq.google.XmonSimulator.run_sweep**

\[
\text{XmonSimulator.} \text{run_sweep}(\text{program:}) = \text{Union[cirq.circuits.circuit.Circuit, cirq.schedules.schedule.Schedule], params:} \text{Union[cirq.study.resolver.ParamResolver, cirq.study.sweeps.Sweep], List[cirq.study.trial_result.TrialResult]}
\]

Runs the supplied Circuit or Schedule, mimicking quantum hardware.

In contrast to `run`, this allows for sweeping over different parameter values.
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.

```python
XmonSimulator.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 used to define the order of amplitudes in the wave function.
- **initial_state** – If an int, the state is set to the computational basis state corresponding to this state. Otherwise if this is a np.ndarray it is the full initial state. In this case it must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an appropriate dtype for the simulator.

**Returns** SimulateTrialResults for the simulation. Includes the final wave function.

```python
XmonSimulator.simulate_moment_steps
```

Returns an iterator of StepResults for each moment simulated.
Parameters

- **circuit** – The Circuit to simulate.
- **param_resolver** – A ParamResolver for determining values of Symbols.
- **qubit_order** – Determines the canonical ordering of the qubits used to define the order of amplitudes in the wave function.
- **initial_state** – If an int, the state is set to the computational basis state corresponding to this state. Otherwise if this is a np.ndarray it is the full initial state. In this case it must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an appropriate dtype for the simulator.

Returns

Iterator that steps through the simulation, simulating each moment and returning a StepResult for each moment.

cirq.google.XmonSimulator.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 used to define the order of amplitudes in the wave function.
- **initial_state** – If an int, the state is set to the computational basis state corresponding to this state. Otherwise if this is a np.ndarray it is the full initial state. In this case it must be the correct size, be normalized (an L2 norm of 1), and be safely castable to an appropriate dtype for the simulator.

Returns

List of SimulatorTrialResults for this run, one for each possible parameter resolver.

cirq.google.XmonStepResult

class cirq.google.XmonStepResult (stepper: cirq.google.sim.xmon_stepper.Stepper, qubit_map: Dict, measurements: Dict[str, numpy.ndarray])

Results of a step of the simulator.
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).

measurements

A dictionary from measurement gate key to measurement results, ordered by the qubits that the measurement operates on.

__init__(stepper: cirq.google.sim.xmon_stepper.Stepper, qubit_map: Dict, measurements: Dict[str, numpy.ndarray]) → 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>bloch_vector_of(qubit)</td>
<td>Returns the bloch vector of a qubit in the state.</td>
</tr>
<tr>
<td>density_matrix_of(qubits)</td>
<td>Returns the density matrix of the state.</td>
</tr>
<tr>
<td>dirac_notation(decimals)</td>
<td>Returns the state as a string in Dirac notation.</td>
</tr>
<tr>
<td>sample(qubits, repetitions)</td>
<td>Samples from the wave function at this point in the computation.</td>
</tr>
<tr>
<td>sample_measurement_ops(measurement_ops, ...)</td>
<td>Samples from the wave function at this point in the computation.</td>
</tr>
<tr>
<td>set_state(state, numpy.ndarray)</td>
<td>Updates the state of the simulator to the given new state.</td>
</tr>
<tr>
<td>state_vector()</td>
<td>Return the state (wave function) at this point in the computation.</td>
</tr>
</tbody>
</table>

cirq.google.XmonStepResult.bloch_vector_of

XmonStepResult.bloch_vector_of(qubit: cirq.ops.raw_types.QubitId) → 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.google.XmonStepResult.density_matrix_of

`XmonStepResult.density_matrix_of(qubits: List[cirq.ops.raw_types.QubitId] = 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:

```python
self.state_vector() = np.array([1/np.sqrt(2), 1/np.sqrt(2)],
dtype=np.complex64)
qubits = None
gives us
ho = \begin{bmatrix}
0.5 & 0.5 \\
0.5 & 0.5
\end{bmatrix}
```

Args:
- `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.google.XmonStepResult.dirac_notation

`XmonStepResult.dirac_notation(decimals: int = 2) → str`

Returns the state 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.google.XmonStepResult.sample**

`XmonStepResult.sample(qubits: List[cirq.ops.raw_types.QubitId], repetitions: int = 1)`

Samples from the wave function at this point in the computation.

Note that this does not collapse the wave function.

**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.

**cirq.google.XmonStepResult.sample_measurement_ops**

`XmonStepResult.sample_measurement_ops(measurement_ops: List[cirq.ops.gate_operation.GateOperation], repetitions: int = 1) → Dict[str, numpy.ndarray]`

Samples from the wave function 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 from.
- `repetitions` – The number of samples to take.

**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.google.XmonStepResult.set_state**

`XmonStepResult.set_state(state: Union[int, numpy.ndarray])`

Updates the state of the simulator to the given new state.

**Parameters**

- `state` – If this is an int, then this is the state to reset
- `stepper to, expressed as an integer of the computational basis. (the)` –
- `to bitwise indices is little endian. Otherwise if this is (Integer)` –
- `np.ndarray` this must be the correct size and have dtype of `(a)`
- `np.complex64`

**Raises**
- `ValueError` if the state is incorrectly sized or not of the correct dtype.

```python
cirq.google.XmonStepResult.state_vector
```

Return the state (wave function) at this point in 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**

```python
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>[0:0]</td>
<td>[0:0]</td>
<td>[0:0]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</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>

### 3.1.14 Testing

Functionality for writing unit tests involving objects from Cirq, and also some general testing utilities.
testing.assert_allclose_up_to_global_phase

cirq.testing.assert_allclose_up_to_global_phase(actual: numpy.ndarray, desired: numpy.ndarray, *, rtol: float = 1e-07, atol: float = 0.0, equal_nan: bool = True, err_msg: Optional[str] = None, verbose: bool = True) \rightarrow None

Checks if a \approx b \times \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` – The matrices aren’t nearly equal up to global phase.

cirq.testing.assert_circuits_with_terminal_measurements_are_equivalent


Determines 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

**cirq.testing.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_eigen_gate_has_consistent_apply_unitary

**cirq.testing.assert_eigen_gate_has_consistent_apply_unitary**(eigen_gate_type: Type[cirq.ops.eigen_gate.EigenGate], *, exponents=(0, 1, -1, 0.5, 0.25, -0.5, 0.1, cirq.Symbol('s')), global_shifts=(0, 0.5, -0.5, 0.1), qubit_count: Optional[int] = None) → None

Tests whether an EigenGate type's apply_unitary is correct.

Contrasts the effects of the gate’s _apply_unitary_ with the
matrix returned by the gate’s _unitary_ method, trying various values for
the gate exponent and global shift.

**Parameters**

- **eigen_gate_type** – The type of gate to test. The type must have an __init__ method
  that takes an exponent and a global_shift.
- **exponents** – The exponents to try. Defaults to a variety of special and arbitrary angles,
  as well as a parameterized angle (a symbol).
- **global_shifts** – The global shifts to try. Defaults to a variety of special angles.
- **qubit_count** – The qubit count to use for the gate. This argument isn’t needed if the gate has a unitary matrix or implements
cirq.SingleQubitGate/cirq.TwoQubitGate/cirq.ThreeQubitGate; it will be inferred.


cirq.testing.assert_equivalent_repr

**cirq.testing.assert_equivalent_repr**(value: Any, *, setup_code: str = 'import cirq
import numpy as np', global_vals: Optional[Dict[str, Any]] = None, local_vals: Optional[Dict[str, Any]] = None) → None

Checks that eval(repr(v)) == v.

**Parameters**

- **value** – A value whose repr should be evaluatable 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.
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.
- **qubit_count** – Usually inferred. The number of qubits the value acts on. This argument isn’t needed if the gate has a unitary matrix or implements `cirq.SingleQubitGate`/`cirq.TwoQubitGate`/`cirq.ThreeQubitGate`.

---

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.

- **qubit_count** – A minimum qubit count for the test system. This argument isn’t needed if the gate has a unitary matrix or implements `cirq.SingleQubitGate/cirq.TwoQubitGate/cirq.ThreeQubitGate`; it will be inferred.

**cirq.testing.assert_has_diagram**

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.
- **kwargs** – Keyword arguments to be passed to actual.to_text_diagram().

**cirq.testing.assert_phase_by_is_consistent_with_unitary**

cirq.testing.assert_phase_by_is_consistent_with_unitary(val: Any) Uses val._unitary_ to check val._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**
add_equality_group(*group_items)  Tries to add a disjoint equivalence group to the equality tester.

make_equality_group(*factories)  Tries to add a disjoint equivalence group to the equality tester.

cirq.testing.EqualsTester.add_equality_group

EqualsTester.add_equality_group(*group_items)
    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)
    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.QubitId, q1: cirq.ops.raw_types.QubitId, q2: cirq.ops.raw_types.QubitId, device: cirq.devices.device.Device = cirq.UnconstrainedDevice)  →  cirq.circuits.circuit.Circuit
cirq.testing.only_test_in_python3

cirq.testing.only_test_in_python3(func)
A decorator that indicates a test should not execute in python 2.

For example, in python 2 repr('a') is “u’a’” instead of “’a’” when from future import unicode is present (which it will be, since 3to2 inserts it for us). This is annoying to work around when testing repr methods, so instead you can just tag the test with this decorator.

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

add_ascending(*items) Tries to add a sequence of ascending items to the order tester.

add_ascending_equivalence_group(*group_items) To add an ascending equivalence group to the order tester.

cirq.testing.OrderTester.add_ascending

OrderTester.add_ascending(*items)
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 been already
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)
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.

Adds the objects as a group.

**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.QubitId], int], n_moments: int, op_density: float, gate_domain: Optional[Dict[cirq.ops.raw_types.Gate, int]] = None) → cirq.circuits.circuit.Circuit

Generates a random circuit.

**Parameters**

- `qubits` – the qubits that the circuit acts on. Because the qubits on which an operation acts are chosen randomly, not all given qubits may be acted upon.
- `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.

**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

---

cirq.testing.random_special_orthogonal

cirq.testing.random_special_orthogonal(dim: int) → numpy.ndarray

---

cirq.testing.random_special_unitary

cirq.testing.random_special_unitary(dim: int) → numpy.ndarray

---

cirq.testing.random_unitary

cirq.testing.random_unitary(dim: int) → numpy.ndarray

Returns a random unitary matrix distributed with Haar measure.

**Parameters**

- `dim` – The width and height of the matrix.

**Returns**

The sampled unitary matrix.
References


cirq.testing.TempDirectoryPath

class cirq.testing.TempDirectoryPath

A context manager that provides a temporary directory for use within a ‘with’ statement.

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

cirq.testing.TempFilePath

class cirq.testing.TempFilePath

A context manager that provides a temporary file path for use within a ‘with’ statement.

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

3.1.15 Work in Progress - Noisy Channels

Imperfect operations.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amplitude_damp(gamma)</td>
<td>Returns an AmplitudeDampingChannel with the given probability gamma.</td>
</tr>
<tr>
<td>AmplitudeDampingChannel(gamma)</td>
<td>Dampen qubit amplitudes through dissipation.</td>
</tr>
<tr>
<td>asymmetric_depolarize(p_x, p_y, p_z)</td>
<td>Returns an AsymmetricDepolarizingChannel with given parameter.</td>
</tr>
<tr>
<td>AsymmetricDepolarizingChannel(p_x, p_y, p_z)</td>
<td>A channel that depolarizes asymmetrically along different directions.</td>
</tr>
<tr>
<td>bit_flip(p)</td>
<td>Construct a BitFlipChannel that flips a qubit state</td>
</tr>
<tr>
<td>BitFlipChannel(p)</td>
<td>Probabilistically flip a qubit from 1 to 0 state or vice versa.</td>
</tr>
<tr>
<td>channel(val, default[, dtype])</td>
<td>Returns a list of matrices describing the channel for the given value.</td>
</tr>
<tr>
<td>depolarize(p)</td>
<td>Returns a DepolarizingChannel with given probability of error.</td>
</tr>
<tr>
<td>DepolarizingChannel(p)</td>
<td>A channel that depolarizes a qubit.</td>
</tr>
<tr>
<td>generalized_amplitude_damp(p, gamma)</td>
<td>Returns a GeneralizedAmplitudeDampingChannel with the given</td>
</tr>
</tbody>
</table>

Continued on next page
Table 133 – continued from previous page

<table>
<thead>
<tr>
<th>GeneralizedAmplitudeDampingChannel (p, \gamma)</th>
<th>Dampen qubit amplitudes through non ideal dissipation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>phase_damp (\gamma)</td>
<td>Creates a PhaseDampingChannel with damping constant (\gamma).</td>
</tr>
<tr>
<td>PhaseDampingChannel (\gamma)</td>
<td>Dampen qubit phase.</td>
</tr>
<tr>
<td>phase_flip (p)</td>
<td>Returns a PhaseFlipChannel that flips a qubit’s phase with probability</td>
</tr>
<tr>
<td>PhaseFlipChannel (p)</td>
<td>Probabilistically flip the sign of the phase of a qubit.</td>
</tr>
<tr>
<td>rotation_error</td>
<td></td>
</tr>
<tr>
<td>RotationErrorChannel</td>
<td></td>
</tr>
<tr>
<td>SupportsChannel (*args, **kwargs)</td>
<td>An object that may be describable as a quantum channel.</td>
</tr>
</tbody>
</table>

**cirq.amplitude_damp**

cirq.amplitude_damp \(\gamma: float\) \(\rightarrow\) cirq.ops.common_channels.AmplitudeDampingChannel

Returns an AmplitudeDampingChannel with the given probability \(\gamma\).

This channel evolves a density matrix via:

\[
\rho \rightarrow M_0 \rho M_0^\dagger + M_1 \rho M_1^\dagger
\]

With:

\[
M_0 = \begin{bmatrix}
1 & 0 \\
0 & \sqrt{1 - \gamma}
\end{bmatrix}
\]

\[
M_1 = \begin{bmatrix}
0 & \sqrt{\gamma} \\
0 & 0
\end{bmatrix}
\]

**Parameters** \(\gamma\) – the probability of the interaction being dissipative.

**Raises** ValueError – if \(\gamma\) is not a valid probability.

**cirq.AmplitudeDampingChannel**

class cirq.AmplitudeDampingChannel \(\gamma\)

Dampen qubit amplitudes through dissipation.

This channel models the effect of energy dissipation to the surrounding environment.

```
__init__(\gamma) \rightarrow 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:

$$M_0 = \begin{bmatrix}
1 & 0 \\
0 & \sqrt{1 - \gamma}
\end{bmatrix}$$

$$M_1 = \begin{bmatrix}
0 & \sqrt{\gamma} \\
0 & 0
\end{bmatrix}$$

**Parameters**

- gamma – the probability of the interaction being dissipative.

**Raises**

- ValueError – if gamma is not a valid probability.

**Methods**

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

**cirq.AmplitudeDampingChannel.num_qubits**

AmplitudeDampingChannel.num_qubits() \(\rightarrow\) int

The number of qubits this gate acts on.

**cirq.AmplitudeDampingChannel.on**

AmplitudeDampingChannel.on(*qubits) \(\rightarrow\) gate_operation.GateOperation

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: Iterable[cirq.ops.raw_types.QubitId]) \(\rightarrow\)

Union[cirq.ops.raw_types.Operation, Iterable[Any]]

Returns a list of operations apply this gate to each of the targets.

**Parameters**

- targets – The qubits to apply this gate to.
**Returns** Operations applying this gate to the target qubits.

**cirq.AmplitudeDampingChannel.validate_args**

AmplitudeDampingChannel.validate_args(qubits)

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

Does no checks by default. Child classes can override.

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

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

**cirq.asymmetric_depolarize**

cirq.asymmetric_depolarize(p_x: float, p_y: float, p_z: float) ➔ 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**

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) ➔ 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**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 apply this gate to each of the 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>
</tbody>
</table>

**cirq.AsymmetricDepolarizingChannel.num_qubits**

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

**cirq.AsymmetricDepolarizingChannel.on**

AsymmetricDepolarizingChannel.on(*qubits) \(\rightarrow\) gate_operation.GateOperation
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: Iterable[cirq.ops.raw_types.QubitId]) \(\rightarrow\) Union[cirq.ops.raw_types.Operation, Iterable[Any]]
Returns a list of operations apply this gate to each of the targets.

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

Returns  Operations applying this gate to the target qubits.

**cirq.AsymmetricDepolarizingChannel.validate_args**

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

Does no checks by default. Child classes can override.
### Parameters `qubits` – The collection of qubits to potentially apply the gate to.

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

#### `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

\[
M_0 = \sqrt{p} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
\]

\[
M_1 = \sqrt{1-p} \begin{bmatrix} 0 & 1 \\ 1 & -0 \end{bmatrix}
\]

**Parameters** `p` – the probability of a bit flip.

**Raises** `ValueError` – if `p` is not a valid probability.

#### `cirq.BitFlipChannel`

**class** `cirq.BitFlipChannel(p)`

Probabilistically flip a qubit from 1 to 0 state or vice versa.

```
__init__(p) → 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:

\[
M_0 = \sqrt{p} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
\]

\[
M_1 = \sqrt{1-p} \begin{bmatrix} 0 & 1 \\ 1 & -0 \end{bmatrix}
\]

(continues on next page)
### Parameters

**p** – the probability of a bit flip.

### Raises

`ValueError` – if `p` is not a valid probability.

## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 apply this gate to each of the 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>
</tbody>
</table>

### `cirq.BitFlipChannel.num_qubits`

BitFlipChannel:num_qubits(): → int

The number of qubits this gate acts on.

### `cirq.BitFlipChannel.on`

BitFlipChannel:on(*qubits): → gate_operation.GateOperation

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`

BitFlipChannel:on_each(targets: Iterable[cirq.ops.raw_types.QubitId]): → Union[gate_operation.Operation, Iterable[Any]]

Returns a list of operations apply this gate to each of the targets.

**Parameters**

targets – The qubits to apply this gate to.

**Returns**

Operations applying this gate to the target qubits.

### `cirq.BitFlipChannel.validate_args`

BitFlipChannel:validate_args(qubits)

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

Does no checks by default. Child classes can override.

**Parameters**

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

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

### cirq.channel

**Syntax**
```python
cirq.channel(val: Any, default: Iterable[TDefault] = (array([], dtype=float64), )) -> Union[Tuple[numpy.ndarray], Iterable[TDefault]]
```

**Returns**
A list of matrices describing the channel for the given value.

**Description**
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:

\[
\text{ho} \rightarrow \sum_{k=0}^{r-1} A_0 \text{ho} A_0^\dagger
\]

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.

**Args**
- `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 _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.depolarize

**Syntax**
```python
cirq.depolarize(p: float) -> cirq.ops.common_channels.DepolarizingChannel
```

**Returns**
A DepolarizingChannel with given probability of error.

**Description**
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
\[ \rho \to (1 - p) \rho + (p / 3) X \rho X + (p / 3) Y \rho Y + (p / 3) Z \rho Z \]

**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.DependorizingChannel`

class `cirq.DependorizingChannel(p)`

A channel that depolarizes a qubit.

```
__init__(p) \to 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
\[ \rho \to (1 - p) \rho + (p / 3) X \rho X + (p / 3) Y \rho Y + (p / 3) Z \rho Z \]

**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

<table>
<thead>
<tr>
<th>Method</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<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 apply this gate to each of the 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>
</tbody>
</table>

### `cirq.DependorizingChannel.num_qubits`

```
DepolarizingChannel.num_qubits() \to int
```

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

DepolarizingChannel.on(*qubits) → gate_operation.GateOperation
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: Iterable[cirq.ops.raw_types.QubitId]) →
Union[cirq.ops.raw_types.Operation, Iterable[Any]]
Returns a list of operations apply this gate to each of the targets.

Parameters targets – The qubits to apply this gate to.

Returns Operations applying this gate to the target qubits.

cirq.DepolarizingChannel.validate_args

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

Does no checks by default. Child classes can override.

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

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

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:

$$M_0 = \sqrt{p} \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1 - \gamma} \end{bmatrix}$$

$$M_1 = \sqrt{p} \begin{bmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{bmatrix}$$

$$M_2 = \sqrt{1-p} \begin{bmatrix} \sqrt{1-\gamma} & 0 \\ 0 & 0 \end{bmatrix}$$

(continues on next page)
\begin{bmatrix}
0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
0 & 0 \\
\sqrt{\gamma} & 0 \\
\end{bmatrix}
\begin{bmatrix}
0 & 0 \\
\sqrt{\gamma} & 0 \\
\end{bmatrix}
\begin{bmatrix}
0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
0 & 0 \\
\sqrt{\gamma} & 0 \\
\end{bmatrix}
\begin{bmatrix}
0 & 1 \\
\end{bmatrix}

Parameters

- \textbf{gamma} – the probability of the interaction being dissipative.
- \textbf{p} – the probability of the qubit and environment exchanging energy.

\textbf{Raises} \texttt{ValueError} – gamma or p is not a valid probability.

cirq.GeneralizedAmplitudeDampingChannel

class \texttt{cirq.GeneralizedAmplitudeDampingChannel} \(p: \text{float}, \text{gamma}: \text{float}\)
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.

\texttt{\_\_init\_\_\(p: \text{float}, \text{gamma}: \text{float}\) \rightarrow \text{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 \(|0\) and probability \(1-p\) of being \(|1\).

This channel evolves a density matrix via

\begin{align}
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{align}
Parameters

- **gamma** – the probability of the interaction being dissipative.
- **p** – the probability of the qubit and environment exchanging energy.

Raises **ValueError** – if gamma or p is not a valid probability.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</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>on_each(targets)</td>
<td>Returns a list of operations apply this gate to each of the targets.</td>
</tr>
<tr>
<td>validate_args(qubits)</td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
</tbody>
</table>

**cirq.GeneralizedAmplitudeDampingChannel.num_qubits**

GeneralizedAmplitudeDampingChannel.num_qubits() → int

The number of qubits this gate acts on.

**cirq.GeneralizedAmplitudeDampingChannel.on**

GeneralizedAmplitudeDampingChannel.on(*qubits) → gate_operation.GateOperation

Returns an application of this gate to the given qubits.

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

**cirq.GeneralizedAmplitudeDampingChannel.on_each**

GeneralizedAmplitudeDampingChannel.on_each(targets: Iterable[cirq.ops.raw_types.QubitId]) → Union[cirq.ops.raw_types.Operation, Iterable[Any]]

Returns a list of operations apply this gate to each of the targets.

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

Returns Operations applying this gate to the target qubits.

**cirq.GeneralizedAmplitudeDampingChannel.validate_args**

GeneralizedAmplitudeDampingChannel.validate_args(qubits)

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

Does no checks by default. Child classes can override.

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

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

**Raises**

ValueError – is gamma is not a valid probability.

---

cirq.PhaseDampingChannel

class cirq.PhaseDampingChannel(gamma)

Dampen qubit phase.

This channel models phase damping which is the loss of quantum
information without the loss of energy.

__init__(gamma) → 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:

\[
M_0 = \begin{bmatrix}
1 & 0 \\
0 & \sqrt{1 - \gamma}
\end{bmatrix}
\]
\[
M_1 = \begin{bmatrix}
0 & 0 \\
0 & \sqrt{\gamma}
\end{bmatrix}
\]
Parameters `gamma` – The damping constant.

Raises `ValueError` – if gamma is not a valid probability.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 apply this gate to each of the 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>
</tbody>
</table>

`cirq.PhaseDampingChannel.num_qubits`

```python
PhaseDampingChannel.num_qubits() → int
The number of qubits this gate acts on.
```

`cirq.PhaseDampingChannel.on`

```python
PhaseDampingChannel.on(*qubits) → gate_operation.GateOperation
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`

```python
PhaseDampingChannel.on_each(targets: Iterable[cirq.ops.raw_types.QubitId]) → Union[cirq.ops.raw_types.Operation, Iterable[Any]]
Returns a list of operations apply this gate to each of the targets.

Parameters targets – The qubits to apply this gate to.

Returns Operations applying this gate to the target qubits.
```

`cirq.PhaseDampingChannel.validate_args`

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

Does no checks by default. Child classes can override.

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

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

`cirq.phase_flip`

```python
cirq.phase_flip(p: Optional[float] = None) → Union[cirq.ops.common_gates.ZPowGate,
cirq.ops.common_channels.PhaseFlipChannel]
```

Chapter 3. API Reference
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:

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

Parameters  

- **p** – the probability of a phase flip.

Raises  

- ValueError – if p is not a valid probability.

```
cirq.PhaseFlipChannel

class cirq.PhaseFlipChannel(p)

Probabilistically flip the sign of the phase of a qubit.

__init__(p) → 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:

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

Parameters  

- **p** – the probability of a phase flip.

Raises  

- ValueError – if p is not a valid probability.

Methods

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

Continued on next page
Table 140 – 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>on_each(targets)</code></td>
<td>Returns a list of operations apply this gate to each of the 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>
</tbody>
</table>

### `cirq.PhaseFlipChannel.num_qubits`

PhaseFlipChannel.num_qubits() → int

The number of qubits this gate acts on.

### `cirq.PhaseFlipChannel.on`

PhaseFlipChannel.on(*qubits) → gate_operation.GateOperation

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: Iterable[cirq.ops.raw_types.QubitId]) → Union[cirq.ops.raw_types.Operation, Iterable[Any]]

Returns a list of operations apply this gate to each of the targets.

**Parameters**

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

**Returns**

Operations applying this gate to the target qubits.

### `cirq.PhaseFlipChannel.validate_args`

PhaseFlipChannel.validate_args(qubits)

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

Does no checks by default. Child classes can override.

**Parameters**

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

**Throws**

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

### `cirq.SupportsChannel`

#### `class cirq.SupportsChannel(*args, **kwargs)`

An object that may be describable as a quantum channel.

**Methods**

___
3.1.16 Work in Progress - Stabilizers

Tools for working with the well-behaved operations from the Clifford+Measurement set.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CircuitDag(can_reorder, ...)</td>
<td>A representation of a Circuit as a directed acyclic graph.</td>
</tr>
<tr>
<td>SingleQubitCliffordGate(*, _rotation_map,</td>
<td>Any single qubit Clifford rotation.</td>
</tr>
<tr>
<td>...)</td>
<td></td>
</tr>
<tr>
<td>Pauli(index, name)</td>
<td>Represents the Pauli gates.</td>
</tr>
<tr>
<td>PauliInteractionGate(pauli0, invert0, ...)</td>
<td></td>
</tr>
<tr>
<td>PauliString(qubit_pauli_map, ...)</td>
<td></td>
</tr>
<tr>
<td>PauliTransform(to, flip)</td>
<td></td>
</tr>
<tr>
<td>Unique(val)</td>
<td>A wrapper for a value that doesn’t compare equal to other instances.</td>
</tr>
</tbody>
</table>

### `cirq.CircuitDag`

**class cirq.CircuitDag**

```python
```

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.ops.raw_types.Operation, cirq.ops.raw_types.Operation], bool] = _disjoint_qubits, incoming_graph_data: Any = None, device: cirq.devices.device.Device = cirq.UnconstrainedDevice) -> 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

- `add_cycle(nodes, **attr)`: Add an edge between u and v.
- `add_edge(u_of_edge, v_of_edge, **attr)`: Add an edge between u and v.
- `add_edges_from(ebunch_to_add, **attr)`: Add all the edges in ebunch_to_add.
- `add_node(node_for_adding, **attr)`: Add a single node `node_for_adding` and update node attributes.
- `add_nodes_from(nodes_for_adding, **attr)`: Add multiple nodes.
- `add_path(nodes, **attr)`: Add a single path.
- `add_star(nodes, **attr)`: Add a star graph.
- `add_weighted_edges_from(ebunch_to_add[, weight])`: Add weighted edges in `ebunch_to_add` with specified weight attr.
- `adjacency()`: Return an iterator over (node, adjacency dict) tuples for all nodes.
- `all_operations()`: Return all operations.
- `append(op)`: Append an operation.
- `clear()`: Remove all nodes and edges from the graph.
- `copy([as_view])`: Return a copy of the graph.
- `disjoint_qubits(op1, op2)`: Returns true only if the operations have qubits in common.
- `edge_subgraph(edges)`: Returns the subgraph induced by the specified edges.
- `fresh_copy()`: Return a fresh copy of the graph.
- `from_circuit(circuit, can_reorder, ...)`: From circuit.
- `from_ops(*operations, can_reorder, ...)`: From operations.
- `get_edge_data(u, v[, default])`: Return the attribute dictionary associated with edge (u, v).
- `has_edge(u, v)`: Return True if the edge (u, v) is in the graph.
- `has_node(n)`: Return True if the graph contains the node n.
- `has_predecessor(u, v)`: Return True if node u has predecessor v.
- `has_successor(u, v)`: Return True if node u has successor v.
- `is_directed()`: Return True if graph is directed, False otherwise.
- `is_multigraph()`: Return True if graph is a multigraph, False otherwise.
- `make_node(op)`: Make a node.
- `nbunch_iter([nbunch])`: Return an iterator over nodes contained in nbunch that are also in the graph.
- `neighbors(n)`: Return an iterator over successor nodes of n.
- `nodes_with_selfloops()`: Return an iterator over nodes with self-loops.
- `number_of_edges([u, v])`: Return the number of edges between two nodes.
- `number_of_nodes()`: Return the number of nodes in the graph.
- `number_of_selfloops()`: Return the number of self-loops in the graph.
- `order()`: Return the number of nodes in the graph.
- `ordered_nodes()`: Return an iterator over nodes.
- `predecessors(n)`: Return an iterator over predecessor nodes of n.
- `remove_edge(u, v)`: Remove the edge between u and v.
- `remove_edges_from(ebunch)`: Remove all edges specified in ebunch.
- `remove_node(n)`: Remove node n.
- `remove_nodes_from(nodes)`: Remove multiple nodes.
- `reverse([copy])`: Return the reverse of the graph.
- `selfloop_edges([data, keys, default])`: Return an iterator over self-loops.
Table 143 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size([weight])</td>
<td>Return the number of edges or total of all edge weights.</td>
</tr>
<tr>
<td>subgraph(nodes)</td>
<td>Return a SubGraph view of the subgraph induced on nodes.</td>
</tr>
<tr>
<td>successors(n)</td>
<td>Return an iterator over successor nodes of n.</td>
</tr>
<tr>
<td>to_circuit()</td>
<td></td>
</tr>
<tr>
<td>to_directed()</td>
<td>Return 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()</td>
<td>Return 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
```
Cirq Documentation, Release 0.5.0.dev

>>> 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:

>>> 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.

>>> 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

>>> 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

>>> G.add_edges_from([(1, 2), (2, 3)], weight=3)
>>> G.add_edges_from([(3, 4), (1, 4)], label='WN2898')
**cirq.CircuitDag.add_node**

```python
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`.

**See also:**

`add_nodes_from()`

**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.

**cirq.CircuitDag.add_nodes_from**

```python
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
```

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
```

```
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 Multi-
Graph/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()

Return 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, nbrdict) for n, nbrdict in G.adjacency()]
[(0, {1: {}}), (1, {0: {}, 2: {}}), (2, {1: {}, 3: {}}), (3, {2: {}})]
```

cirq.CircuitDag.all_operations

CircuitDag.all_operations() → Iterator[cirq.ops.raw_types.Operation]

cirq.CircuitDag.append

CircuitDag.append(op: cirq.ops.raw_types.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
>>> 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)
Return 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)
>>> H = nx.Graph(G)
>>> H = G.__class__(G)
```

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__()
>>> H.add_nodes_from(G)
>>> H.add_edges_from(G.edges)
```

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 `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.

Return type 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
circuitDag.disjoint_qubits(op1: cirq.ops.raw_types.Operation, op2: cirq.ops.raw_types.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 `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]
```
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)

Return the attribute dictionary associated with edge (u, v).

This is identical to $G[u][v]$ except the default is returned instead of an exception is 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

>>> 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

>>> 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.\texttt{has\_edge}\,(u, v)

Return True if the edge \((u, v)\) is in the graph.

This is the same as \(v \in G[u]\) without KeyError exceptions.

\textbf{Parameters} \(v (u, )\) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.

\textbf{Returns} \texttt{edge\_ind} – True if edge is in the graph, False otherwise.

\textbf{Return type} \ \texttt{bool}

\textbf{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 \texttt{in} G[0]  # though this gives KeyError if 0 not in G
True
```

cirq.CircuitDag.has_node

CircuitDag.\texttt{has\_node}\,(n)

Return 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
```

```python

cirq.CircuitDag.has_predecessor

CircuitDag.has_predecessor(u, v)
Return 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)
Return True if node u has successor v.
This is true if graph has the edge u->v.


cirq.CircuitDag.is_directed

CircuitDag.is_directed()
Return True if graph is directed, False otherwise.


cirq.CircuitDag.is_multigraph

CircuitDag.is_multigraph()
Return True if graph is a multigraph, False otherwise.


cirq.CircuitDag.make_node

```cirq.ops.raw_types.Operation``
→
cirq.circuits.circuit_dag.Unique


cirq.CircuitDag.nbunch_iter

CircuitDag.nbunch_iter(nbunch=None)
Return 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)*

Return 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

cirq.CircuitDag.number_of_edges

CircuitDag.number_of_edges *(u=None, v=None)*

Return 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

**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.number_of_nodes()  
Return the number of nodes in the graph.

**Returns nnodes** – The number of nodes in the graph.

**Return type** int

**Examples**

```python
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> len(G)
3
```

**cirq.CircuitDag.number_of_selfloops**

CircuitDag.number_of_selfloops()
cirq.CircuitDag.order

CircuitDag.order()  
Return the number of nodes in the graph.  

Returns nnodes – The number of nodes in the graph.  

Return type int  

See also: number_of_nodes(), __len__()  


cirq.CircuitDag.ordered_nodes

CircuitDag.ordered_nodes() → Iterator[cirq.circuits.circuit_dag.Unique[cirq.ops.raw_types.Operation]]


cirq.CircuitDag.predecessors

CircuitDag.predecessors(n)  
Return an iterator over predecessor nodes of n.  

A predecessor of n is a node m such that there exists a directed edge from m to n.  

Parameters n (node) – A node in the graph  

Raises NetworkXError – If n is not in the graph.  

See also: 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

>>> 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)
Return 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.

**cirq.CircuitDag.selfloop_edges**

CircuitDag.selfloop_edges(data=False, keys=False, default=None)

**cirq.CircuitDag.size**

CircuitDag.size(weight=None)
Return 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
```

cirq.CircuitDag.subgraph

CircuitDag.subgraph(nodes)

Return 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) for n, nbrs in G.adj.items() for nbr in nbrs)
```

(continues on next page)
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**

```python
CircuitDag.successors(n)
```

Return 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**

```python
CircuitDag.to_circuit() \to cirq.circuits.circuit.Circuit
```

**cirq.CircuitDag.to_directed**

```python
CircuitDag.to_directed(as_view=False)
```

Return 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 \( D=\text{DiGraph}(G) \) 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 \texttt{Graph} to use dict-like objects in the data structure, those changes do not transfer to the \texttt{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)]
```

\texttt{cirq.CircuitDag.to\_directed\_class}

\texttt{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 \texttt{to\_directed()} copies.

\texttt{cirq.CircuitDag.to\_undirected}

\texttt{CircuitDag.to\_undirected}(\texttt{reciprocal=False}, \texttt{as\_view=False})

Return an undirected representation of the digraph.

**Parameters**

- \texttt{reciprocal} (\texttt{bool (optional)}) – If True only keep edges that appear in both directions in the original digraph.
- \texttt{as\_view} (\texttt{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** \texttt{Graph}

See also:

\texttt{Graph()}, \texttt{copy()}, \texttt{add\_edge()}, \texttt{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 \(G=\text{DiGraph}(D)\) 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 \text{DiGraph} to use dict-like objects in the data structure, those changes do not transfer to the Graph created by this method.

Examples

```python
>>> 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)]
```

cirq.CircuitDag.to_undirected_class

CircuitDag\textbackslash{}.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 \texttt{to\_directed()} copies.

cirq.CircuitDag.update

CircuitDag\textbackslash{}.update\ (edges=\textit{None}, nodes=\textit{None})

Update the graph using nodes/edges/graphs as input.

Like 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 edges or nodes. 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 \((u, v, \text{datadict})\).

Parameters

- **edges** (\textit{Graph object, collection of edges, or None}) – The first parameter can be a graph or some edges. If it has attributes \textit{nodes} and \textit{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 None, no edges are added.
• **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 None. If *edges is None* and *nodes is None* an exception is raised. If the first parameter is a Graph, then *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, nodes)
```

### Notes

If 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)

>>> 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, 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}}}
>>> e = [(u, v, {'weight': d}) for u, nbrs in adj.items() for v, d in nbrs.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, nbrs in pred.items() for v in nbrs]

>>> MDG = nx.MultiDiGraph()
>>> # MultiGraph dict-of-dict-of-dict-of-attribute
>>> adj = {1: {2: {0: {'weight': 1.3}, 1: {'weight': 1.2}}},
... 3: {2: {0: {'weight': 0.7})}}
>>> e = [(u, v, ekey, d) for u, nbrs in adj.items() for v, keydict in nbrs.items() for ekey, d in keydict.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>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adj</td>
<td>Graph adjacency object holding the neighbors of each node.</td>
</tr>
<tr>
<td>degree</td>
<td>A DegreeView for the Graph as G.degree or G.degree().</td>
</tr>
<tr>
<td>edges</td>
<td>An OutEdgeView of the DiGraph as G.edges or G.edges().</td>
</tr>
<tr>
<td>in_degree</td>
<td>An InDegreeView for (node, in_degree) or in_degree for single node.</td>
</tr>
<tr>
<td>in_edges</td>
<td>An InEdgeView of the Graph as G.in_edges or G.in_edges().</td>
</tr>
<tr>
<td>name</td>
<td>String identifier of the graph.</td>
</tr>
<tr>
<td>node</td>
<td>A NodeView of the Graph as G.nodes or G.nodes().</td>
</tr>
<tr>
<td>nodes</td>
<td>A NodeView of the Graph as G.nodes or G.nodes().</td>
</tr>
<tr>
<td>out_degree</td>
<td>An OutDegreeView for (node, out_degree)</td>
</tr>
<tr>
<td>out_edges</td>
<td>An OutEdgeView of the DiGraph as G.edges or G.edges().</td>
</tr>
<tr>
<td>pred</td>
<td>Graph adjacency object holding the predecessors of each node.</td>
</tr>
<tr>
<td>succ</td>
<td>Graph adjacency object holding the successors of each node.</td>
</tr>
</tbody>
</table>

#### Cirq.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 `for nbr, foo_value in G[node].data('foo', default=1):` works.

For directed graphs, `G.adj` holds outgoing (successor) info.
cirq.CircuitDag.degree

CircuitDag.\texttt{degree}
A DegreeView for the Graph as \(G.\texttt{degree}\) or \(G.\texttt{degree()}\).

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 of 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:*

\texttt{in}\_\texttt{degree}, \texttt{out}\_\texttt{degree}

*Examples*

```python
g = nx.DiGraph()  # or MultiDiGraph
g.add_path([0, 1, 2, 3])
g.degree(0)  # node 0 with degree 1
1
g.degree([0, 1, 2])
[(0, 1), (1, 2), (2, 2)]
```

cirq.CircuitDag.edges

CircuitDag.\texttt{edges}
An OutEdgeView of the DiGraph as \(G.\texttt{edges}\) or \(G.\texttt{edges()}\).

\begin{verbatim}
edges(self, nbunch=None, data=False, default=None)
\end{verbatim}

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
elements, 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
g.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), (1, 2, 1), (2, 3, 5)])
>>> G.edges([0, 2])  # only edges incident to these nodes
OutEdgeDataView([(0, 1), (2, 3)])
>>> G.edges(0)  # only edges incident to a single node (use G.adj[0]?)
OutEdgeDataView([(0, 1)])
```
cirq.CircuitDag.in_degree

CircuitDag.in_degree
An InDegreeView for (node, in_degree) or in_degree for single node.

The node in_degree is the number of edges pointing to the node. The weighted node degree is the sum of the edge weights for edges incident to that node.

This object provides an iteration over (node, in_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) – In-degree of the node
- OR if multiple nodes are requested
- **nd_iter** (iterator) – The iterator returns two-tuples of (node, in-degree).

See also:

degree, out_degree

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

CircuitDag.in_edges
An InEdgeView of the Graph as G.in_edges or G.in_edges().
in_edges(self, nbunch=None, data=False, default=None):

Parameters
In_edges

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

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

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]
>>> 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, {})]
```

```python
>>> list(G.nodes.data('foo'))
[(0, 'bar'), (1, None), (2, None)]
>>> list(G.nodes.data('time'))
[(0, None), (1, '5pm'), (2, None)]
```

```python
>>> 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**

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:
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, {})]
```

```python
>>> list(G.nodes.data())
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
```

```python
>>> list(G.nodes(data='foo'))
[(0, 'bar'), (1, None), (2, None)]
```

```python
>>> list(G.nodes.data('foo'))
[(0, 'bar'), (1, None), (2, None)]
```

```python
>>> list(G.nodes(data='time'))
[(0, None), (1, '5pm'), (2, None)]
```

```python
>>> list(G.nodes.data('time'))
[(0, None), (1, '5pm'), (2, None)]
```

```python
>>> list(G.nodes.data('time', default='Not Available'))
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
```

```python
>>> 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

`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

**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})])
>>> G.edges.data('weight', default=1)
OutEdgeDataView([(0, 1, 1), (1, 2, 1), (2, 3, 5)])
>>> G.edges([0, 2])  # only edges incident to these nodes
OutEdgeDataView([(0, 1), (2, 3)])
>>> G.edges(0)  # only edges incident to a single node (use G.adj[0]?)
OutEdgeDataView([(0, 1)])
```

cirq.CircuitDag.pred

**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
for nbr, datadict in G.pred[n].items(): A data-view not provided by dicts also exists: 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

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 for nbr, datadict in G.succ[n].items(): A data-view not provided by dicts also exists: 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.

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:

Continued on next page

3.1. API Reference 305
Table 145 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>decompose_rotation()</code></td>
<td>Returns ((first_rotation_axis, first_rotation_quarter_turns), ...)</td>
</tr>
<tr>
<td><code>equivalent_gate_before(after)</code></td>
<td>Returns a SingleQubitCliffordGate such that the circuits</td>
</tr>
<tr>
<td><code>from_double_map(pauli_map_to, ...)</code></td>
<td>Returns a SingleQubitCliffordGate for the</td>
</tr>
<tr>
<td><code>from_pauli(pauli, sqrt)</code></td>
<td></td>
</tr>
<tr>
<td><code>from_quarter_turns(pauli, quarter_turns)</code></td>
<td>Returns a SingleQubitCliffordGate for the</td>
</tr>
<tr>
<td><code>from_single_map(pauli_map_to, ...)</code></td>
<td>Returns a SingleQubitCliffordGate for the</td>
</tr>
<tr>
<td><code>from_xz_map(x_to, bool], z_to, bool])</code></td>
<td>Returns a SingleQubitCliffordGate for the specified transforms.</td>
</tr>
<tr>
<td><code>merged_with(second)</code></td>
<td>Returns a SingleQubitCliffordGate such that the circuits</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 apply this gate to each of the targets.</td>
</tr>
<tr>
<td><code>transform(pauli)</code></td>
<td></td>
</tr>
<tr>
<td><code>validate_args(qubits)</code></td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
</tbody>
</table>

### Cirq.SingleQubitCliffordGate.commutes_with

Cirq.SingleQubitCliffordGate.commutes_with

```
SingleQubitCliffordGate.commutes_with(gate_or_pauli: Union[cirq.ops.pauli_gates.Pauli, cirq.ops.pauli_gates.Pauli]) → bool
```

### Cirq.SingleQubitCliffordGate.commutes_with_pauli

Cirq.SingleQubitCliffordGate.commutes_with_pauli

```
SingleQubitCliffordGate.commutes_with_pauli(pauli: cirq.ops.pauli_gates.Pauli) → bool
```

### Cirq.SingleQubitCliffordGate.commutes_with_single_qubit_gate

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:

**cirq.SingleQubitCliffordGate.decompose_rotation**

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**

SingleQubitCliffordGate.equivalent_gate_before(after: cirq.ops.clifford_gate.SingleQubitCliffordGate) → cirq.ops.clifford_gate.SingleQubitCliffordGate

Returns a SingleQubitCliffordGate such that the circuits –output–self– and –self–gate–
are equivalent up to global phase.

**cirq.SingleQubitCliffordGate.from_double_map**

static SingleQubitCliffordGate.from_double_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.

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**

static SingleQubitCliffordGate.from_pauli(pauli: cirq.ops.pauli_gates.Pauli, sqrt: bool = False) → cirq.ops.clifford_gate.SingleQubitCliffordGate

Returns a SingleQubitCliffordGate for the specified transform with a 90 or 180 degree rotation.

Numeric transform from pauli: x, y or z (both Pauli and its square root) are supported.
cirq.SingleQubitCliffordGate.from_quarter_turns

```
static SingleQubitCliffordGate.from_quarter_turns(pauli: cirq.ops.pauli_gates.Pauli, 
        quarter_turns: int) → 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 \( \text{cirq.X} \)
- **y_to** – The transform from \( \text{cirq.Y} \)
- **z_to** – The transform from \( \text{cirq.Z} \)

---

**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 \( \text{Y} \) transform is derived from the \( \text{X} \) and \( \text{Z} \).

**Parameters**

- **x_to** – Which Pauli to transform \( \text{X} \) to and if it should negate.
- **z_to** – Which Pauli to transform \( \text{Z} \) 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) → gate_operation.GateOperation

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: Iterable[cirq.ops.raw_types.QubitId]) → Union[cirq.ops.raw_types.Operation, Iterable[Any]]

Returns a list of operations apply this gate to each of the targets.

Parameters targets – The qubits to apply this gate to.

Returns Operations applying this gate to the target qubits.

cirq.SingleQubitCliffordGate.transform

SingleQubitCliffordGate.transform(pauli: cirq.ops.pauli_gates.Pauli) → cirq.ops.clifford_gate.PauliTransform

cirq.SingleQubitCliffordGate.validate_args

SingleQubitCliffordGate.validate_args(qubits)

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

Does no checks by default. Child classes can override.

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

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

Attributes
<table>
<thead>
<tr>
<th>Gate</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H$</td>
<td>$\begin{cases} X: +Z, &amp; Y: -Y, \ Z: +X \end{cases}$</td>
</tr>
<tr>
<td>$I$</td>
<td>$\begin{cases} X: +X, &amp; Y: +Y, \ Z: +Z \end{cases}$</td>
</tr>
<tr>
<td>$X$</td>
<td>$\begin{cases} X: +X, &amp; Y: -Y, \ Z: -Z \end{cases}$</td>
</tr>
<tr>
<td>$X_{\sqrt{n}}$</td>
<td>$\begin{cases} X: +X, &amp; Y: -Z, \ Z: +Y \end{cases}$</td>
</tr>
<tr>
<td>$X_{\sqrt{2}}$</td>
<td>$\begin{cases} X: +X, &amp; Y: +Z, \ Z: -Y \end{cases}$</td>
</tr>
<tr>
<td>$Y$</td>
<td>$\begin{cases} X: -X, &amp; Y: +Y, \ Z: -Z \end{cases}$</td>
</tr>
<tr>
<td>$Y_{\sqrt{n}}$</td>
<td>$\begin{cases} X: +Z, &amp; Y: +Y, \ Z: -X \end{cases}$</td>
</tr>
<tr>
<td>$Y_{\sqrt{2}}$</td>
<td>$\begin{cases} X: -Z, &amp; Y: +Y, \ Z: +X \end{cases}$</td>
</tr>
</tbody>
</table>
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


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

    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:

       \[ \theta \]

    2. Shifting the angle by global_shift:

       \[ \theta + s \]

    3. Scaling the angle by exponent:

       \[ (\theta + s) \times e \]

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

       \[ \exp(i \times \pi \times (\theta + s) \times e) \]

    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**I is applied will gain a rel-
      ative phase of \[e^{i \times \pi \times \text{exponent}}\] 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 con-
      trols 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>by_index(index)</code></td>
<td>The number of qubits this gate acts on.</td>
</tr>
<tr>
<td><code>by_relative_index(p, relative_index)</code></td>
<td>Returns an application of this gate to the given qubits.</td>
</tr>
<tr>
<td><code>commutes_with(other)</code></td>
<td>Relative index of self w.r.t. third.</td>
</tr>
<tr>
<td><code>num_qubits()</code></td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
<tr>
<td><code>on(*qubits)</code></td>
<td>Checks if this gate can be applied to the given qubits.</td>
</tr>
<tr>
<td><code>relative_index(second)</code></td>
<td>Checks if this gate can be applied 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>
</tbody>
</table>

```python
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(other: cirq.ops.pauli_gates.Pauli) -> bool

cirq.Pauli.num_qubits

Pauli.num_qubits() -> int

    The number of qubits this gate acts on.

cirq.Pauli.on

Pauli.on(*qubits) -> gate_operation.GateOperation

    Returns an application of this gate to the given qubits.

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


cirq.Pauli.relative_index

Pauli.relative_index(second: cirq.ops.pauli_gates.Pauli) -> 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.QubitId]) → None
Checks if this gate can be applied to the given qubits.

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

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

Attributes

exponent

cirq.Pauli.exponent

Pauli.exponent

cirq.PauliInteractionGate


__init__(pauli0: cirq.ops.pauli_gates.Pauli, invert0: bool, pauli1: cirq.ops.pauli_gates.Pauli, invert1: bool, *, exponent: Union[cirq.value.symbol.Symbol, float] = 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

num_qubits() The number of qubits this gate acts on.
Table 149 – 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>
</tbody>
</table>

### `cirq.PauliInteractionGate.num_qubits`

```python
PauliInteractionGate.num_qubits() \rightarrow int
```

The number of qubits this gate acts on.

### `cirq.PauliInteractionGate.on`

```python
PauliInteractionGate.on(*qubits) \rightarrow gate_operation.GateOperation
```

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`

```python
PauliInteractionGate.qubit_index_to_equivalence_group_key(index: int) \rightarrow int
```

Returns a key that differs between non-interchangeable qubits.

### `cirq.PauliInteractionGate.validate_args`

```python
PauliInteractionGate.validate_args(qubits)
```

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

Does no checks by default. Child classes can override.

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

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

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CNOT</code></td>
<td></td>
</tr>
<tr>
<td><code>CZ</code></td>
<td></td>
</tr>
<tr>
<td><code>exponent</code></td>
<td></td>
</tr>
</tbody>
</table>

### `cirq.PauliInteractionGate.CNOT`

```python
PauliInteractionGate.CNOT = cirq.PauliInteractionGate(cirq.Z, False, cirq.X, False)
```

### `cirq.PauliInteractionGate.CZ`

```python
```
**cirq.PauliInteractionGate.exponent**

PauliInteractionGate.exponent

**cirq.PauliString**

class cirq.PauliString(
quubit_pauli_map: Mapping[cirq.ops.raw_types.QubitId, cirq.ops.pauli_gates.Pauli], negated: bool = False)

    __init__ (qubit_pauli_map: Mapping[cirq.ops.raw_types.QubitId, cirq.ops.pauli_gates.Pauli],
               negated: bool = False) → None
    Initialize self. See help(type(self)) for accurate signature.

**Methods**

- commutes_with (other)
- equal_up_to_sign (other)
- from_single (qubit, pauli) Creates a PauliString with a single qubit.
- get (key[, default])
- items ()
- keys ()
- map_qubits (qubit_map,...)
- negate ()
- pass_operations_over (ops, after_to_before) Determines how the Pauli string changes when conjugated by Cliffords.
- to_z_basis_ops () Returns operations to convert the qubits to the computational basis.
- transform_qubits (func,...) Returns the same operation, but with different qubits.
- values ()
- with_qubits (*new_qubits)
- zip_items (other)
- zip_paulis (other)

**cirq.PauliString.commutes_with**

PauliString .commutes_with (other: cirq.ops.pauli_string.PauliString) → bool

**cirq.PauliString.equal_up_to_sign**

PauliString .equal_up_to_sign (other: cirq.ops.pauli_string.PauliString) → bool

**cirq.PauliString.from_single**

static PauliString .from_single (qubit:
cirq.ops.raw_types.QubitId,
pauli: cirq.ops.pauli_gates.Pauli) →
cirq.ops.pauli_string.PauliString

Creates a PauliString with a single qubit.
cirq.PauliString.get

PauliString.get(key: cirq.ops.raw_types.QubitId, default=None)

cirq.PauliString.items

PauliString.items() → ItemsView

cirq.PauliString.keys

PauliString.keys() → KeysView[cirq.ops.raw_types.QubitId]

cirq.PauliString.map_qubits

PauliString.map_qubits(qubit_map: Dict[cirq.ops.raw_types.QubitId, cirq.ops.raw_types.QubitId]) → cirq.ops.pauli_string.PauliString

cirq.PauliString.negate

PauliString.negate() → cirq.ops.pauli_string.PauliString

cirq.PauliString.pass_operations_over

PauliString.pass_operations_over(ops: Iterable[cirq.ops.raw_types.Operation], after_to_before: bool = False) → cirq.ops.pauli_string.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() → Union[cirq.ops.raw_types.Operation, Iterable[Any]]

Returns operations to convert the qubits to the computational basis.

cirq.PauliString.transform_qubits

PauliString.transform_qubits(func: Callable[cirq.ops.raw_types.QubitId, cirq.ops.raw_types.QubitId]) → TSelf.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.values

PauliString.values() → ValuesView[cirq.ops.pauli_gates.Pauli]

cirq.PauliString.with_qubits

PauliString.with_qubits(*new_qubits) → cirq.ops.pauli_string.PauliString

cirq.PauliString.zip_items


cirq.PauliString.zip_paulis


Attributes
qubits

cirq.PauliString.qubits

PauliString.qubits

cirq.PauliTransform

class cirq.PauliTransform(to, flip)

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

    Methods

    count(value)
        PauliTransform.count(value) → integer – return number of occurrences of value

    index(value, [start, [stop]])
        PauliTransform.index(value[, start[, stop]]) → 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

        PauliTransform.flip
            Alias for field number 1

    cirq.PauliTransform.to

        PauliTransform.to
            Alias for field number 0
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

3.1.17 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.

| contrib.acquaintance | Tools for creating and using acquaintance strategies. |
| contrib.jobs         | Package for handling a full quantum job.            |
| contrib.paulistring |                                                                 |
| contrib.qcircuit    | Converts cirq circuits into latex using qcircuit.   |
| contrib.quirk       | Converts cirq circuits into quirk circuits.         |
| contrib.tpu         |                                                                 |

cirq.contrib.acquaintance

Tools for creating and using acquaintance strategies.

cirq.contrib.jobs

Package for handling a full quantum job.

Types and methods related to transforming circuits in preparation of sending
them to Quantum Engine. Contains classes to help with adding parameter
sweeps and error simulation.

cirq.contrib.paulistring
cirq.contrib.qcircuit

Converts cirq circuits into latex using qcircuit.

cirq.contrib.quirk

Converts cirq circuits into quirk circuits.
Python Module Index

C

cirq.contrib.acquaintance, 319

cirq.contrib.jobs, 319

cirq.contrib.paulistring, 319

cirq.contrib.qcircuit, 320

cirq.contrib.quirk, 320
Symbols

__init__() (cirq.AmplitudeDampingChannel method), 258
__init__() (cirq.AsymmetricDepolarizingChannel method), 260
__init__() (cirq.BitFlipChannel method), 262
__init__() (cirq.CCXGate method), 98
__init__() (cirq.CCZGate method), 100
__init__() (cirq.CNotGate method), 82
__init__() (cirq.CSwapGate method), 102
__init__() (cirq.CZGate method), 84
__init__() (cirq.Circuit method), 114
__init__() (cirq.CircuitDag method), 273
__init__() (cirq.CircuitDiagramInfo method), 182
__init__() (cirq.CircuitDiagramInfoArgs method), 183
__init__() (cirq.ControlledGate method), 104
__init__() (cirq.ConvertToCzAndSingleGates method), 192
__init__() (cirq.Circuit method), 114
__init__() (cirq.Device method), 59
__init__() (cirq.DropEmptyMoments method), 193
__init__() (cirq.DropNegligible method), 193
__init__() (cirq.Duration method), 209
__init__() (cirq.EigenGate method), 105
__init__() (cirq.EjectPhasedPaulis method), 193
__init__() (cirq.EjectZ method), 194
__init__() (cirq.ExpandComposite method), 194
__init__() (cirq.Gate method), 107
__init__() (cirq.GateOperation method), 107
__init__() (cirq.GeneralizedAmplitudeDampingChannel method), 267
__init__() (cirq.GridQubit method), 61
__init__() (cirq.HGate method), 65
__init__() (cirq.ISwapGate method), 87
__init__() (cirq.InsertStrategy method), 128
__init__() (cirq.InterchangeableQubitsGate method), 108
__init__() (cirq.KakDecomposition method), 213
__init__() (cirq.LineQubit method), 62
__init__() (cirq.Linspace method), 139
__init__() (cirq.MeasurementGate method), 68
__init__() (cirq.MergeInteractions method), 196
__init__() (cirq.MergeSingleQubitGates method), 197
__init__() (cirq.Moment method), 129
__init__() (cirq.NamedQubit method), 63
__init__() (cirq.Operation method), 109
__init__() (cirq.ParamResolver method), 141
__init__() (cirq.Pauli method), 311
__init__() (cirq.PauliInteractionGate method), 313
__init__() (cirq.PauliString method), 315
__init__() (cirq.PauliTransform method), 318
__init__() (cirq.PhaseDampingChannel method), 269
__init__() (cirq.PhaseFlipChannel method), 271
__init__() (cirq.PhasedXPowGate method), 69
__init__() (cirq.PointOptimizationSummary method), 198
__init__() (cirq.PointOptimizer method), 199
__init__() (cirq.Points method), 142
__init__() (cirq.QasmArgs method), 184
__init__() (cirq.QasmOutput method), 186
__init__() (cirq.QubitId method), 64
__init__() (cirq.QubitOrder method), 131
__init__() (cirq.ReversibleCompositeGate method), 110
__init__() (cirq.Schedule method), 134
__init__() (cirq.ScheduledOperation method), 136
__init__() (cirq.SimulatesFinalWaveFunction method), 162
__init__() (cirq.SimulatesIntermediateWaveFunction method), 164
__init__() (cirq.SimulatesSamples method), 143
__init__() (cirq.SimulationTrialResult method), 146
__init__() (cirq.Simulator method), 149
__init__() (cirq.SimulatorStep method), 154
__init__() (cirq.SingleQubitCliffordGate method), 305
__init__() (cirq.SingleQubitGate method), 110
__init__() (cirq.SingleQubitMatrixGate method), 71
__init__() (cirq.StepResult method), 158
__init__() (cirq.SupportsApplyUnitary method), 186
__init__() (cirq.SupportsChannel method), 272
Cirq Documentation, Release 0.5.0.dev

A
add_ascending() (cirq.testing.OrderTester method), 255
add_ascending_equivalence_group() (cirq.testing.OrderTester method), 255
add_cycle() (cirq.CircuitDag method), 275
add_edge() (cirq.CircuitDag method), 275
add_edges_from() (cirq.CircuitDag method), 276
add_equality_group() (cirq.testing.EqualsTester method), 254
add_job_labels() (cirq.google.Engine method), 228
add_node() (cirq.CircuitDag method), 277
add_nodes_from() (cirq.CircuitDag method), 277
add_path() (cirq.CircuitDag method), 278
add_program_labels() (cirq.google.Engine method), 228
add_star() (cirq.CircuitDag method), 278
add_weighted_edges_from() (cirq.CircuitDag method), 278
adj (cirq.CircuitDag attribute), 295
adjacency() (cirq.CircuitDag method), 279
all_close() (cirq.Tolerance method), 220
all_near_zero() (cirq.Tolerance method), 220
all_near_zero_mod() (cirq.Tolerance method), 220
all_operations() (cirq.Circuit method), 115
all_operations() (cirq.CircuitDag method), 279
all_qubits() (cirq.Circuit method), 115
allclose_up_to_global_phase() (in module cirq), 203
amplitude_damp() (in module cirq), 258
AmplitudeDampingChannel (class in cirq), 258
AnnealSequenceSearchStrategy (class in cirq.google), 224
append() (cirq.Circuit method), 116
append() (cirq.CircuitDag method), 279
apply_matrix_to_slices() (in module cirq), 203
apply_unitary() (in module cirq), 174
apply_unitary_effect_to_state() (cirq.Circuit method), 116
are_all_measurements_terminal() (cirq.Circuit method), 117
as_qubit_order() (cirq.QubitOrder static method), 131
assert_allclose_up_to_global_phase() (in module cirq.testing), 249
assert_circuits_with_terminal_measurements_are_equivalent() (in module cirq.testing), 250
assert_decompose_is_consistent_with_unitary() (in module cirq.testing), 251
assert_eigen_gate_has_consistent_apply_unitary() (in module cirq.testing), 251
assert_equivalent_repr() (in module cirq.testing), 251
assert_has_consistent_apply_unitary() (in module cirq.testing), 252
assert_has_consistent_apply_unitary_for_various_exponents() (in module cirq.testing), 252
assert_has_diagram() (in module cirq.testing), 253
assert_phase_by_is_consistent_with_unitary() (in module cirq.testing), 253
assert_qasm_is_consistent_with_unitary() (in module cirq.testing), 253
assert_same_circuits() (in module cirq.testing), 253
asymmetric_depolarize() (in module cirq), 260
AsymmetricDepolarizingChannel (class in cirq), 260
at() (cirq.google.XmonDevice method), 236

B
batch_insert() (cirq.Circuit method), 117
batch_insert_into() (cirq.Circuit method), 118
batch_remove() (cirq.Circuit method), 118
bidirectionalize_real_matrix_pair_with_symmetric_products() (in module cirq), 204
bidirectionalize_unitary_with_special_orthogonals() (in module cirq), 205
bit_flip() (in module cirq), 262
BitFlipChannel (class in cirq), 262
bloch_vector() (cirq.SimulationTrialResult method), 146
bloch_vector_from_state_vector() (in module cirq), 138
bloch_vector_of() (cirq.google.XmonStepResult method), 245
bloch_vector_of() (cirq.SimulatorStep method), 154
bloch_vector_of() (cirq.StepResult method), 159
block_diag() (in module cirq), 207
Bristlecone (in module cirq.google), 225
by_index() (cirq.Pauli static method), 312
by_relative_index() (cirq.Pauli static method), 312

C
can_add_operation_into_moment() (cirq.Device method), 60
can_add_operation_into_moment() (cirq.google.XmonDevice method), 236
cancel_job() (cirq.google.Engine method), 228
canonicalize_half_turns() (in module cirq), 205
CCX (in module cirq), 98
CCXPowGate (class in cirq), 98
CCZ (in module cirq), 100
CCZPowGate (class in cirq), 100
channel() (in module cirq), 264
check_unused_args() (cirq.QuasmArgs method), 185
closed_angle_to_canonical_half_turns() (in module cirq), 205
closed_angle_to_half_turns() (in module cirq), 206
Circuit (class in cirq), 113
circuit_diagram_info() (in module cirq), 175
CircuitDag (class in cirq), 273
CircuitDiagramInfo (class in cirq), 182
CircuitDiagramInfoArgs (class in cirq), 183
cirq.contrib.acquaintance (module), 319
cirq.contrib.jobs (module), 319
cirq.contrib.paulistring (module), 319
cirq.contrib.qcircuit (module), 320
cirq.contrib.quirk (module), 320
clear() (cirq.CircuitDag method), 279
clear_operations_touching() (cirq.Circuit method), 118
close() (cirq.Tolerance method), 220
CNOT (cirq.PauliInteractionGate attribute), 314
CNOT (in module cirq), 81
CNotPowGate (class in cirq), 82
col() (cirq.google.XmonDevice method), 236
commutes() (in module cirq), 207
commutes_with() (cirq.Pauli method), 312
commutes_with() (cirq.PauliString method), 315
commutes_with() (cirq.SingleQubitCliffordGate method), 306
commutes_with_pauli() (cirq.SingleQubitCliffordGate method), 306
commutes_with_single_qubit_gate() (cirq.SingleQubitCliffordGate method), 306
compute_displays() (cirq.google.XmonSimulator method), 240
compute_displays() (cirq.SimulatesIntermediateWaveFunction method), 165
compute_displays() (cirq.Simulator method), 149
compute_displays_sweep() (cirq.google.XmonSimulator method), 241
compute_displays_sweep() (cirq.SimulatesIntermediateWaveFunction method), 166
compute_displays_sweep() (cirq.Simulator method), 150
compute_samples_displays() (cirq.google.XmonSimulator method), 241
compute_samples_displays() (cirq.SimulatesSamples method), 143
compute_samples_displays() (cirq.Simulator method), 151
compute_samples_displays_sweep() (cirq.google.XmonSimulator method), 242
compute_samples_displays_sweep() (cirq.SimulatesSamples method), 144
compute_samples_displays_sweep() (cirq.Simulator method), 151
content_present() (cirq.TextDiagramDrawer method), 217
CONTROL_TAG (in module cirq), 208
ControlledGate (class in cirq), 104
circuit_diagram() (cirq.google.ConvertToXmonGates method), 226
circuit_diagram() (cirq.QuasmArgs method), 185
ConvertToCzAndSingleGates (class in cirq), 191
ConvertToXmonGates (class in cirq.google), 225
copy() (cirq.Circuit method), 118
copy() (cirq.CircuitDag method), 280
copy() (cirq.CircuitDiagramInfoArgs method), 184
copy() (cirq.google.JobConfig method), 233
count() (cirq.PauliTransform method), 318
CSWAP (in module cirq), 102
CSwapGate (class in cirq), 102
CZ (cirq.PauliInteractionGate attribute), 314
CZ (in module cirq), 84

Index 325
Cirq Documentation, Release 0.5.0.dev

CZPowGate (class in cirq), 84

decompose() (in module cirq), 176
decompose_once() (in module cirq), 177
decompose_once_with_qubits() (in module cirq), 177
decompose_operation() (cirq.Device method), 60
decompose_rotation() (cirq.SingleQubitCliffordGate method), 236
decompose_rotation() (cirq.google.XmonDevice method), 307
DEFAUL (cirq.QubitOrder attribute), 133
DEFAUL (cirq.Tolerance attribute), 221
degree (cirq.CircuitDag attribute), 296
density_matrix() (cirq.SimulationTrialResult method), 146
density_matrix_from_state_vector() (in module cirq), 138
density_matrix_of() (cirq.google.XmonStepResult method), 246
density_matrix_of() (cirq.SimulatorStep method), 155
density_matrix_of() (cirq.StepResult method), 159
depolarize() (in module cirq), 264
DepolarizingChannel (class in cirq), 265
device (cirq.Circuit attribute), 127
device (cirq.Schedule attribute), 134
Device (class in cirq), 59
diagonalize_real_symmetric_and_sorted_diagonal_matrices (in module cirq), 208
diagonalize_real_symmetric_matrix() (in module cirq), 208
dirac_notation() (cirq.google.XmonStepResult method), 246
dirac_notation() (cirq.SimulationTrialResult method), 147
dirac_notation() (cirq.SimulatorStep method), 156
dirac_notation() (cirq.StepResult method), 160
dirac_notation() (in module cirq), 139
disjoint_qubits() (cirq.CircuitDag static method), 281
dot() (in module cirq), 209
DropEmptyMoments (class in cirq), 193
DropNegligible (class in cirq), 193
Duration (class in cirq), 209
duration_of() (cirq.Device method), 60
duration_of() (cirq.google.XmonDevice method), 237

E

EARLIEST (cirq.InsertStrategy attribute), 128
diagram_subgraph() (cirq.CircuitDag method), 281
diagrams (cirq.CircuitDag attribute), 296
EjectPhasedPaulis (class in cirq), 193
EjectZ (class in cirq), 194
Engine (class in cirq.google), 227
engine_from_environment() (in module cirq.google), 231
equal_up_to_sign() (cirq.PauliString method), 315
EqualsTester (class in cirq.testing), 253
equivalent_gate_before() (cirq.SingleQubitCliffordGate method), 307
exclude() (cirq.Schedule method), 134
ExpandComposite (class in cirq), 194
explicit() (cirq.QubitOrder static method), 132
exponent (cirq.CCXPowGate attribute), 100
exponent (cirq.CCZPowGate attribute), 102
exponent (cirq.CNotPowGate attribute), 83
exponent (cirq.CZPowGate attribute), 86
exponent (cirq.EigenGate attribute), 106
exponent (cirq.HPowGate attribute), 67
exponent (cirq.ISwapPowGate attribute), 90
exponent (cirq.Pauli attribute), 313
exponent (cirq.PauliInteractionGate attribute), 315
exponent (cirq.PhasedXPowGate attribute), 70
exponent (cirq.SwapPowGate attribute), 91
exponent (cirq.XPowGate attribute), 76
exponent (cirq.XXPowGate attribute), 94
exponent (cirq.YPowGate attribute), 78
exponent (cirq.YYPowGate attribute), 96
exponent (cirq.ZPowGate attribute), 80
exponent (cirq.ZZPowGate attribute), 98

F

final_state (cirq.SimulationTrialResult attribute), 145
findall_operations() (cirq.Circuit method), 118
findall_operations_between() (cirq.Circuit method), 119
findall_operations_with_gate_type() (cirq.Circuit method), 119
flatten_op_tree() (in module cirq), 127
flip (cirq.PauliTransform attribute), 318
force_horizontal_padding_after() (cirq.TextDiagramDrawer method), 217
force_vertical_padding_after() (cirq.TextDiagramDrawer method), 217
format() (cirq.QasmArgs method), 185
format_field() (cirq.QasmArgs method), 185
Foxtail (in module cirq.google), 231
FREDKIN (in module cirq), 103
freeze_op_tree() (in module cirq), 127
fresh_copy() (cirq.CircuitDag method), 282
from_circuit() (cirq.CircuitDag static method), 282
from_double_map() (cirq.SingleQubitCliffordGate static method), 307
from_ops() (cirq.Circuit static method), 120
from_ops() (cirq.CircuitDag static method), 282
from_pauli() (cirq.SingleQubitCliffordGate static method), 307
from_proto_dict() (cirq.GridQubit static method), 62
from_quarter_turns() (cirq.SingleQubitCliffordGate static method), 308
from_single() (cirq.PauliString static method), 315

326
LineQubit (class in cirq), 62
Linspace (class in cirq), 139

make_equality_group()  (cirq.testing.EqualsTester method), 254
make_node() (cirq.CircuitDag static method), 284
map() (cirq.QubitOrder method), 132
map_eigenvalues() (in module cirq), 214
map_qubits() (cirq.PauliString method), 316
match_global_phase() (in module cirq), 207
measure() (in module cirq), 67
measure_each() (in module cirq), 67
measure_state_vector() (in module cirq), 140
MeasurementGate (class in cirq), 67
measurements (cirq.google.XmonStepResult attribute), 245
measurements (cirq.SimulationTrialResult attribute), 145
measurements (cirq.SimulatorStep attribute), 154
measurements (cirq.StepResult attribute), 158
measurements (cirq.TrialResult attribute), 171
merge_single_qubit_gates_into_phased_x_z() (in module cirq), 196
merged_with() (cirq.SingleQubitCliffordGate method), 309
MergeInteractions (class in cirq), 196
MergeSingleQubitGates (class in cirq), 197
min_qubits_before_shard (cirq.google.XmonOptions attribute), 238
Moment (class in cirq), 128
moment_by_moment_schedule() (in module cirq), 130
MS() (in module cirq), 89
mul() (in module cirq), 178
multi_measurement_histogram() (cirq.TrialResult method), 173

N
name (cirq.CircuitDag attribute), 299
name (cirq.Symbol attribute), 215
NamedQubit (class in cirq), 63
nbunch_iter() (cirq.CircuitDag method), 284
near_zero() (cirq.Tolerance method), 220
near_zero_mod() (cirq.Tolerance method), 220
negate() (cirq.PauliString method), 316
neighbors() (cirq.CircuitDag method), 285
neighbors_of() (cirq.google.XmonDevice method), 237
NEW (cirq.InsertStrategy attribute), 128
NEW_THENINLINE (cirq.InsertStrategy attribute), 128
next_moment_operating_on() (cirq.Circuit method), 121
next_moments_operating_on() (cirq.Circuit method), 121
node (cirq.CircuitDag attribute), 299
nodes (cirq.CircuitDag attribute), 301
nodes_with_selfloops() (cirq.CircuitDag method), 285
nonoptimal_toffoli_circuit() (in module cirq.testing), 254
num_prefix_qubits (cirq.google.XmonOptions attribute), 238
num_qubits() (cirq.AmplitudeDampingChannel method), 259
num_qubits() (cirq.AsymmetricDepolarizingChannel method), 261
num_qubits() (cirq.BitFlipChannel method), 263
num_qubits() (cirq.CCXPowGate method), 99
num_qubits() (cirq.CCZPowGate method), 101
num_qubits() (cirq.CNotPowGate method), 83
num_qubits() (cirq.ControlledGate method), 104
num_qubits() (cirq.CSwapGate method), 102
num_qubits() (cirq.CZPowGate method), 85
num_qubits() (cirq.DepolarizingChannel method), 265
num_qubits() (cirq.EigenGate method), 106
num_qubits() (cirq.Gate method), 107
num_qubits() (cirq.GeneralizedAmplitudeDampingChannel method), 268
num_qubits() (cirq.HPowGate method), 66
num_qubits() (cirq.ISwapPowGate method), 88
num_qubits() (cirq.MeasurementGate method), 68
num_qubits() (cirq.Pauli method), 312
num_qubits() (cirq.PauliInteractionGate method), 314
num_qubits() (cirq.PhaseDampingChannel method), 270
num_qubits() (cirq.PhasedXPowGate method), 69
num_qubits() (cirq.PhaseFlipChannel method), 272
num_qubits() (cirq.SingleQubitCliffordGate method), 309
num_qubits() (cirq.SingleQubitGate method), 110
num_qubits() (cirq.SingleQubitMatrixGate method), 72
num_qubits() (cirq.SwapPowGate method), 91
num_qubits() (cirq.ThreeQubitGate method), 111
num_qubits() (cirq.TwoQubitGate method), 112
num_qubits() (cirq.TwoQubitMatrixGate method), 73
num_qubits() (cirq.XPowGate method), 75
num_qubits() (cirq.XXPowGate method), 93
num_qubits() (cirq.YPowGate method), 77
num_qubits() (cirq.YYPowGate method), 95
num_qubits() (cirq.ZPowGate method), 80
num_qubits() (cirq.ZZPowGate method), 97
number_of_edges() (cirq.CircuitDag method), 285
number_of_nodes() (cirq.CircuitDag method), 286
number_of_selfloops() (cirq.CircuitDag method), 286

O
on() (cirq.AmplitudeDampingChannel method), 259
on() (cirq.AsymmetricDepolarizingChannel method), 261
on() (cirq.BitFlipChannel method), 263
on() (cirq.CCXPowGate method), 99
on() (cirq.CCZPowGate method), 101
on() (cirq.CNotPowGate method), 83
on() (cirq.ControlledGate method), 104
on() (cirq.CSwapGate method), 103
on() (cirq.CZPowGate method), 85
on() (cirq.DepolarizingChannel method), 266
on() (cirq.EigenGate method), 106
on() (cirq.Gate method), 107
on() (cirq.GeneralizedAmplitudeDampingChannel method), 268
on() (cirq.HPowGate method), 66
on() (cirq.ISwapPowGate method), 88
on() (cirq.MeasurementGate method), 68
on() (cirq.Pauli method), 312
on() (cirq.PauliInteractionGate method), 314
on() (cirq.PhaseDampingChannel method), 270
on() (cirq.PhasedXPowGate method), 69
on() (cirq.PhaseFlipChannel method), 272
on() (cirq.SingleQubitCliffordGate method), 309
on() (cirq.SingleQubitGate method), 110
on() (cirq.SingleQubitMatrixGate method), 72
on() (cirq.SwapGate method), 91
on() (cirq.ThreeQubitGate method), 111
on() (cirq.TwoQubitGate method), 112
on() (cirq.TwoQubitMatrixGate method), 73
on() (cirq.XPowGate method), 75
on() (cirq.XXPowGate method), 93
on() (cirq.YPowGate method), 78
on() (cirq.YPowGate method), 95
on() (cirq.ZPowGate method), 80
on() (cirq.ZZPowGate method), 97
on_each() (cirq.AmplitudeDampingChannel method), 259
on_each() (cirq.AsymmetricDepolarizingChannel method), 261
on_each() (cirq.BitFlipChannel method), 263
on_each() (cirq.DepolarizingChannel method), 266
on_each() (cirq.GeneralizedAmplitudeDampingChannel method), 268
on_each() (cirq.HPowGate method), 66
on_each() (cirq.PhaseDampingChannel method), 270
on_each() (cirq.PhasedXPowGate method), 70
on_each() (cirq.PhaseFlipChannel method), 272
on_each() (cirq.SingleQubitCliffordGate method), 309
on_each() (cirq.SingleQubitGate method), 111
on_each() (cirq.SingleQubitMatrixGate method), 72
on_each() (cirq.XPowGate method), 75
on_each() (cirq.YPowGate method), 80
on_each() (cirq.ZPowGate method), 97
out_degree (cirq.CircuitDag attribute), 302
out_edges (cirq.CircuitDag attribute), 303
Optimization_at() (cirq.ConvertToCzAndSingleGates method), 192
optimize() (cirq.ContratToCzAndSingleGates method), 192
optimize() (cirq.TransformComposite method), 195
optimize() (cirq.MergeInteractions method), 197
optimize() (cirq.MergeSingleQubitGates method), 198
optimize_circuit() (cirq.PauliString method), 272
optimize_circuit() (cirq.MergeInteractions method), 197
optimize_circuit() (cirq.MergeSingleQubitGates method), 198
optimize_circuit() (cirq.PointOptimizer method), 200
optimized_for_xmon() (in module cirq.google), 195
operator() (cirq.CircuitDag method), 287
order() (cirq.CircuitDag method), 132
order() (cirq.CircuitDag method), 287
order() (cirq.CircuitDag method), 287
order() (cirq.CircuitDag method), 287
order() (cirq.CircuitDag method), 287
order() (cirq.CircuitDag method), 287
order() (cirq.CircuitDag method), 287
order() (cirq.CircuitDag method), 287
order() (cirq.CircuitDag method), 287
order() (cirq.CircuitDag method), 287
order() (cirq.CircuitDag method), 287
order() (cirq.CircuitDag method), 287
order() (cirq.CircuitDag method), 287
order() (cirq.CircuitDag method), 287
order() (cirq.CircuitDag method), 287
order() (cirq.CircuitDag method), 287
ordered_nodes() (cirq.CircuitDag method), 287
out_degree (cirq.CircuitDag attribute), 302
out_edges (cirq.CircuitDag attribute), 303
pack_results() (in module cirq.google), 234
param_dict (cirq.ParamResolver attribute), 141
param_tuples() (cirq.ParamResolver method), 140
param_tuples() (cirq.Points method), 142
param_tuples() (cirq.Sweep method), 169
ParamResolver (class in cirq), 140
params (cirq.SimulationTrialResult attribute), 145
params (cirq.TrialResult attribute), 171
parse() (cirq.QasmArgs method), 185
parse_operations_over() (cirq.PauliString method), 316
Pauli (class in cirq), 311
PauliInteractionGate (class in cirq), 313
PauliString (class in cirq), 315
PauliTransform (class in cirq), 318
phase_by() (in module cirq), 181
phase_damp() (in module cirq), 269
phase_exponent (cirq.PhasedXPowGate attribute), 70
phase_exponent (cirq.XPowGate attribute), 76
phase_exponent (cirq.YPowGate attribute), 78
phase_rotate() (in module cirq), 270
PhaseDampingChannel (class in cirq), 269
PhaseFlipChannel (class in cirq), 69
PhasedXPowGate (class in cirq), 69
sample_state_vector() (in module cirq), 142
save() (cirq.QasmOutput method), 186
save_qasm() (cirq.Circuit method), 125
Schedule (class in cirq), 134
schedule_from_proto_dicts() (in module cirq.google), 234
schedule_to_proto_dicts() (in module cirq.google), 234
scheduled_operations (cirq.Schedule attribute), 134
ScheduledOperation (class in cirq.google), 234
selfloop_edges() (cirq.CircuitDag method), 289
set_job_labels() (cirq.google.Engine method), 231
set_program_labels() (cirq.google.Engine method), 231
set_state() (cirq.google.XmonStepResult method), 247
set_state() (cirq.StepResult method), 161
SimulatesFinalWaveFunction (class in cirq), 162
SimulatesIntermediateWaveFunction (class in cirq), 164
SimulatesSamples (class in cirq), 143
SimulationTrialResult (class in cirq), 145
Simulator (class in cirq), 147
SimulatorStep (class in cirq), 154
single_qubit_matrix_to_gates() (in module cirq), 200
single_qubit_matrix_to_pauli_rotations() (in module cirq), 200
single_qubit_matrix_to_phased_x_z() (in module cirq), 200
single_qubit_operations_after (cirq.KakDecomposition attribute), 212
single_qubit_operations_before (cirq.KakDecomposition attribute), 212
SingleQubitCliffordGate (class in cirq), 305
SingleQubitGate (class in cirq), 110
SingleQubitMatrixGate (class in cirq), 71
to_z_basis_ops() (cirq.PauliString method), 317
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOFFOLI (in module cirq)</td>
<td>103</td>
</tr>
<tr>
<td>Tolerance (class in cirq)</td>
<td>219</td>
</tr>
<tr>
<td>total_nanos() (cirq.Duration method)</td>
<td>209</td>
</tr>
<tr>
<td>total_picos() (cirq.Duration method)</td>
<td>209</td>
</tr>
<tr>
<td>trace_distance_bound() (in module cirq)</td>
<td>181</td>
</tr>
<tr>
<td>transform() (cirq.SingleQubitCliffordGate method)</td>
<td>309</td>
</tr>
<tr>
<td>transform_op_tree() (in module cirq)</td>
<td>136</td>
</tr>
<tr>
<td>transform_qubits() (cirq.GateOperation method)</td>
<td>108</td>
</tr>
<tr>
<td>transform_qubits() (cirq.Operation method)</td>
<td>109</td>
</tr>
<tr>
<td>transform_qubits() (cirq.PauliString method)</td>
<td>317</td>
</tr>
<tr>
<td>transpose() (cirq.TextDiagramDrawer method)</td>
<td>218</td>
</tr>
<tr>
<td>TrialResult (class in cirq)</td>
<td>171</td>
</tr>
<tr>
<td>two_qubit_matrix_to_operations() (in module cirq)</td>
<td>201</td>
</tr>
<tr>
<td>TwoQubitGate (class in cirq)</td>
<td>112</td>
</tr>
<tr>
<td>TwoQubitMatrixGate (class in cirq)</td>
<td>73</td>
</tr>
</tbody>
</table>

**U**

UnconstrainedDevice (in module cirq) | 64
UNINFORMED_DEFAULT (cirq.CircuitDiagramInfoArgs attribute) | 184
Unique (class in cirq) | 319
unitary() (in module cirq) | 180
UnitSweep (in module cirq) | 174
unpack_results() (in module cirq.google) | 234
update() (cirq.CircuitDag method) | 293
use_processes (cirq.google.XmonOptions attribute) | 238
use_unicode_characters (cirq.CircuitDiagramInfoArgs attribute) | 183

**V**

valid_id_re (cirq.QasmOutput attribute) | 186
validate_args() (cirq.AmplitudeDampingChannel method) | 260
validate_args() (cirq.AsymmetricDepolarizingChannel method) | 261
validate_args() (cirq.BitFlipChannel method) | 263
validate_args() (cirq.CCXPowGate method) | 100
validate_args() (cirq.CCZPowGate method) | 102
validate_args() (cirq.CNotPowGate method) | 83
validate_args() (cirq.ControlledGate method) | 104
validate_args() (cirq.CSwapGate method) | 103
validate_args() (cirq.CZPowGate method) | 86
validate_args() (cirq.DependorizingChannel method) | 266
validate_args() (cirq.EigenGate method) | 106
validate_args() (cirq.Gate method) | 107
validate_args() (cirq.GenerlizedAmplitudeDampingChannel method) | 268
validate_args() (cirq.HPowGate method) | 66
validate_args() (cirq.ISwapPowGate method) | 88
validate_args() (cirq.MeasurementGate method) | 68
validate_args() (cirq.Pauli method) | 313
validate_args() (cirq.PauliInteractionGate method) | 314
validate_args() (cirq.PhaseDampingChannel method) | 270
validate_args() (cirq.PhasedXPowGate method) | 70
validate_args() (cirq.PhaseFlipChannel method) | 272
validate_args() (cirq.SingleQubitCliffordGate method) | 309
validate_args() (cirq.SingleQubitGate method) | 111
validate_args() (cirq.SingleQubitMatrixGate method) | 72
validate_args() (cirq.SwapPowGate method) | 91
validate_args() (cirq.ThreeQubitGate method) | 112
validate_args() (cirq.TwoQubitGate method) | 112
validate_args() (cirq.TwoQubitMatrixGate method) | 73
validate_args() (cirq.XPowGate method) | 76
validate_args() (cirq.XXPowGate method) | 93
validate_args() (cirq.YPowGate method) | 78
validate_args() (cirq.YYPowGate method) | 95
validate_args() (cirq.ZPowGate method) | 80
validate_args() (cirq.ZZPowGate method) | 97
validate_circuit() (cirq.Device method) | 60
validate_circuit() (cirq.google.XmonDevice method) | 237
validate_gate() (cirq.google.XmonDevice method) | 237
validate_op() (cirq.CircuitDag method) | 293
validate_schedule() (cirq.Device method) | 61
validate_schedule() (cirq.google.XmonDevice method) | 238
validate_scheduled_operation() (cirq.Device method) | 61
validate_scheduled_operation() (cirq.google.XmonDevice method) | 238
validate_version() (cirq.QasmArgs method) | 238
value_equality() (in module cirq) | 221
value_of() (cirq.ParamResolver method) | 141
values() (cirq.PauliString method) | 317
vertical_line() (cirq.TextDiagramDrawer method) | 218
vformat() (cirq.QasmArgs method) | 185

**W**

width() (cirq.TextDiagramDrawer method) | 218
with_args() (cirq.CircuitDiagramInfoArgs method) | 184
with_bits_flipped() (cirq.MeasurementGate method) | 69
with_device() (cirq.Circuit method) | 127
with_gate() (cirq.GateOperation method) | 108
with_operation() (cirq.Moment method) | 129
with_qubits() (cirq.GateOperation method) | 108
with_quibs() (cirq.Operation method) | 109
with_quibs() (cirq.PauliString method) | 317
without_operations_touching() (cirq.Moment method) | 130
write() (cirq.TextDiagramDrawer method) | 218
Cirq Documentation, Release 0.5.0.dev

Index 333