Cirq Documentation

Release 0.1

The Cirq Developers

Jul 19, 2018

Contents

CHAPTER 1

Installing Cirq

Choose your operating system:

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

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

1.1 Installing on Linux

1. Make sure you have python 3.5 or greater (or else python 2.7).

   See Installing Python 3 on Linux @ the hitchhiker's guide to python.

2. Consider using a virtual environment.

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

   ```
   pip install --upgrade pip
   pip install cirq
   ```

4. (Optional) install system dependencies that pip can't handle.

   ```
   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 -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)
```

1.2 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`:

```
pip install --upgrade pip
pip install cirq
```

4. Check that it works!

```
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)
```

1.3 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 -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)
```

CHAPTER 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 aboutthe level of the textbook "Quantum Computation and Quantum Information" by Nielsen and Chuang. For a more conceptual overview see the conceptual documentation.

To begin, please follow the instructions for installing Cirq.

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 anstaz 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 variataional algorithms will succeed, and exploring this class of algorithms is a key part of research for noisy intermediate scale quantum computers.

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

Consider the energy function

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

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

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

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

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

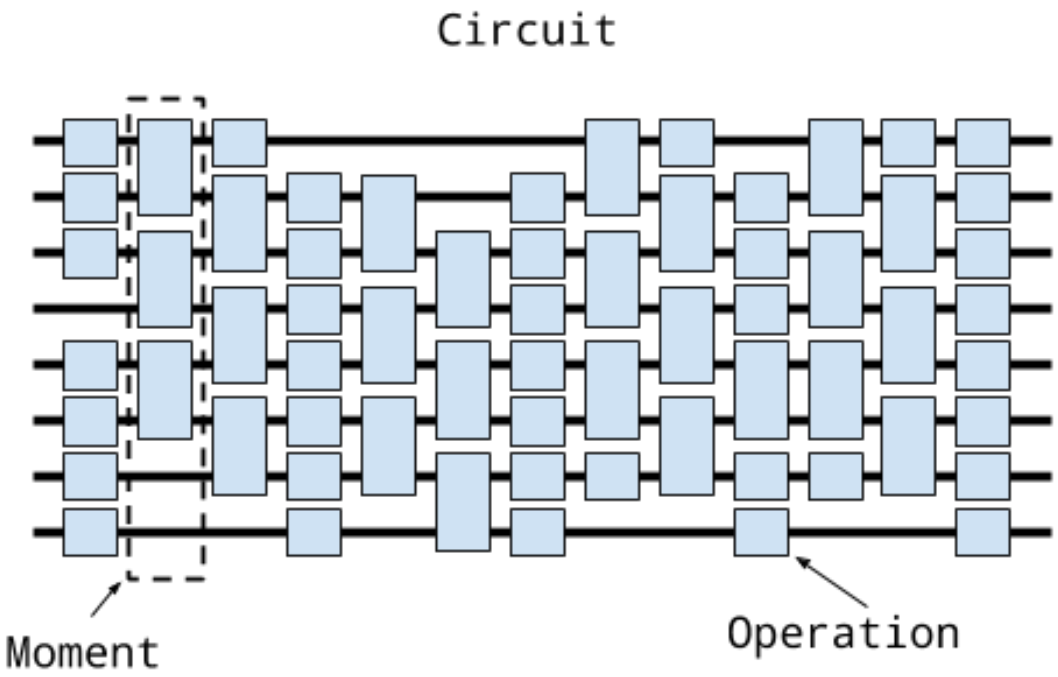
Note that one cannot always measure H directly (without the use of quantum phase estimation), so one often relies on the linearity of expectation values to measure parts of H in step 2. One always needs to repeat the measurements to obtain an estimate of the expectation value. How many measurements needed 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 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. An `Operation` is a 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.



See the conceptual documentation for more details on these classes. 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

```
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)
# prints
# [GridQubit(0, 0), GridQubit(0, 1), GridQubit(0, 2), GridQubit(1, 0), GridQubit(1,␣
↪1), GridQubit(1, 2), GridQubit(2, 0), GridQubit(2, 1), GridQubit(2, 2)]
```

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. `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

```
circuit = cirq.Circuit()
circuit.append(cirq.H.on(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)
```

(continues on next page)

(continued from previous page)

```
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———————
```

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 qubits, but now a sequence of qubits should be supplied as parameters.

Another thing one notices about the above circuit is that the circuit has staggered gates. This is because the way in which we have applied the gates has created 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`. The reason that two `Moments` were created was that the `append` method uses an `InsertStrategy` of `NEW_THEN_INLINE`. `InsertStrategys` describe how new insertions into `Circuits` place their gates. Details of these strategies can be found in the circuit documentation. If we wanted to insert the gates so that they form one `Moment`, we could instead use the `EARLIEST` insertion strategy:

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

(continues on next page)

(continued from previous page)

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

We now see that we have only one moment, as the `X` gates have been slid over to act at the earliest `Moment` they can.

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 `Operationss`.
- 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:

```
def rot_x_layer(length, half_turns):
    """Yields X rotations by half_turns on a square grid of given length."""
    rot = cirq.RotXGate(half_turns=half_turns)
    for i in range(length):
        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(half_turns * pi) I + i sin(half_turns * pi) X.

There is a lot of freedom defining a variational ansatz. Here we will do a variation on a QOAO strategy and define an anstaz related to he 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

```
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 steps of a circuit made up of

1. Apply a RotXGate for the same parameter for all qubits. This is the method we have written above.
2. Apply a RotZGate for the same parameter for all qubits where the transverse field term h is +1.

```
def rot_z_layer(h, half_turns):
    """Yields Z rotations by half_turns conditioned on the field h."""
    gate = cirq.RotZGate(half_turns=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 Rot11Gate for the same parameter between all qubits where the coupling field term J is +1. If the field is -1 apply Rot11Gate conjugated by X gates on all qubits.

```
def rot_11_layer(jr, jc, half_turns):
    """Yeilds rotations about |11> conditioned on the jr and jc fields."""
    gate = cirq.Rot11Gate(half_turns=half_turns)
    for i, jr_row in enumerate(jr):
        for j, jr_ij in enumerate(jr_row):
            if jr_ij == -1:
                yield cirq.X(cirq.GridQubit(i, j))
                yield cirq.X(cirq.GridQubit(i + 1, j))
            yield gate(cirq.GridQubit(i, j),
                       cirq.GridQubit(i + 1, j))
            if jr_ij == -1:
```

(continues on next page)

(continued from previous page)

```
                yield cirq.X(cirq.GridQubit(i, j))
                yield cirq.X(cirq.GridQubit(i + 1, j))

    for i, jc_row in enumerate(jc):
        for j, jc_ij in enumerate(jc_row):
            if jc_ij == 1:
                yield gate(cirq.GridQubit(i, j),
                           cirq.GridQubit(i, j + 1))
```

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

```
def one_step(h, jr, jc, x_half_turns, h_half_turns, j_half_turns):
    length = len(h)
    yield rot_x_layer(length, x_half_turns)
    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))
print(circuit)
# prints something like
# (0, 0): ───X^0.
↪1───X───────────@────────X───────────────────────────────────────────────────────@────────────────
#                                            │                                                     ␣
↪               │
# (0, 1): ───X^0.1───Z^0.
↪2───────────────X───@────────X──────────────────────────────────────────@^0.
↪3───────────
#                                            │            │
# (0, 2): ───X^0.
↪1────────────────────────────────────@──────────────────────────────────────────────────────
#                                            │            │            │
# (1, 0): ───X^0.1───Z^0.2───X───@^0.
↪3───X───────────X───────────@────────X────────────────────────@────────────────
#                                                         │            │        │          ␣
↪               │
# (1, 1): ───X^0.1───────────────────────X───@^0.
↪3───X─────────────────────X───@────────X────────────@^0.3───@────────
#                                                                      │        │            │  ␣
↪                       │
# (1, 2): ───X^0.1───Z^0.2────────────────────────────────@^0.
↪3─────────────────────X───@────────X────────@^0.3───
#                                                                               │            │  ␣
↪    │
# (2, 0): ───X^0.1──────────────────────────────────X────────────@^0.
↪3───X────────────────────────────────────────
#                                                                                            │  ␣
↪    │
# (2, 1): ───X^0.1───Z^0.2──────────────────────────────────────────────────X───@^0.
↪3───X───────────────────────────
#                                                                                               ␣
↪    │
```

(continues on next page)

(continued from previous page)

```
# (2, 2): ———X^0.1———Z^0.
↪2———————————————————————————————————————————————————————————X———@^0.
↪3———X———————————————
```

Where here we see that we have chosen particular parameter values (0.1, 0.2, 0.3).

2.4 Simulation

Now lets see how 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.

```
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, qubit_order=qubits)
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.

```
import numpy as np

def energy_func(length, h, jr, jc):
    def energy(measurements):
        meas_list_of_lists = [measurements[i:i + length] for i in range(length)]
        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]
```

(continues on next page)

(continued from previous page)

```
        return tot_energy
    return energy
print(results.histogram(key='x', fold_func=energy_func(3, h, jr, jc)))
# prints something like
# Counter({-3: 94, -1: 3, 7: 2, 1: 1})
```

One can then calculate the expectation value over all repetitions

```
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 -2.7
```

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.

```
circuit = cirq.Circuit()
alpha = cirq.Symbol('alpha')
beta = cirq.Symbol('beta')
gamma = cirq.Symbol('gamma')
circuit.append(one_step(h, jr, jc, alpha, beta, gamma))
circuit.append(cirq.measure(*qubits, key='x'))
print(circuit)
# prints something like
# (0, 0): ───X^
↪alpha───X──────────────@──────────X──────────────────────────────────────────────────────────────@─
#                                    │                                                      ␣
↪                            │                │
# (0, 1): ───X^alpha───Z^
↪beta──────────────────X───@──────────X──────────────────────────────────────────────────@^
↪gamma────────────────M───
#                                    │              │                                     ␣
↪                                            │
# (0, 2): ───X^
↪alpha───────────────────────────────────────────@──────────────────────────────────────────
#                                    │              │              │                      ␣
↪                                            │
# (1, 0): ───X^alpha───Z^beta───X───@^
↪gamma───X──────────────X─────────────@──────────X───────────────────────────@──────────────────M───
#                                                   │              │         │            ␣
↪                            │                │
# (1, 1): ───X^alpha──────────────────────────X───@^
↪gamma───X──────────────────────X───@──────────X──────────────@^gamma───@───M
```

(continues on next page)

(continued from previous page)

```
#                                                                    │         │
↪   │                                  │         │
# (1, 2): ───X^alpha───Z^beta───────────────────────────────────@^
↪gamma─────────────────────────X───@─────────X─────────@^gamma───M───
#                                                                              │
↪   │             │                              │
# (2, 0): ───X^alpha─────────────────────────────────────X─────────────@^
↪gamma───X─────────────────────────────────────────────M───
#
↪   │             │                              │
# (2, 1): ───X^alpha───Z^
↪beta──────────────────────────────────────────────────────X───@^
↪gamma───X────────────────────────────────M───
#
↪             │                              │
# (2, 2): ───X^alpha───Z^
↪beta────────────────────────────────────────────────────────────────────X───@^
↪gamma───X───────────────────M───
```

Note now that the circuit's gates are parameterized.

Parameters are specified at run time using a `ParamResolver` which is which is just a dictionary from `Symbol` keys to runtime values. For example,

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

```
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)]) -4.7
# OrderedDict([('alpha', 0.1), ('beta', 0.1), ('gamma', 0.30000000000000004)]) -4.86
# OrderedDict([('alpha', 0.1), ('beta', 0.1), ('gamma', 0.5)]) -4.56
# OrderedDict([('alpha', 0.1), ('beta', 0.1), ('gamma', 0.7000000000000001)]) -4.86
...
# OrderedDict([('alpha', 0.9), ('beta', 0.9), ('gamma', 0.7000000000000001)]) 4.54
# OrderedDict([('alpha', 0.9), ('beta', 0.9), ('gamma', 0.9)]) 4.64
```

2.6 Finiding 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.

```
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 -5.06.
```

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. Finally a good place to learn more about features of Cirq is to read through the conceptual documentation.

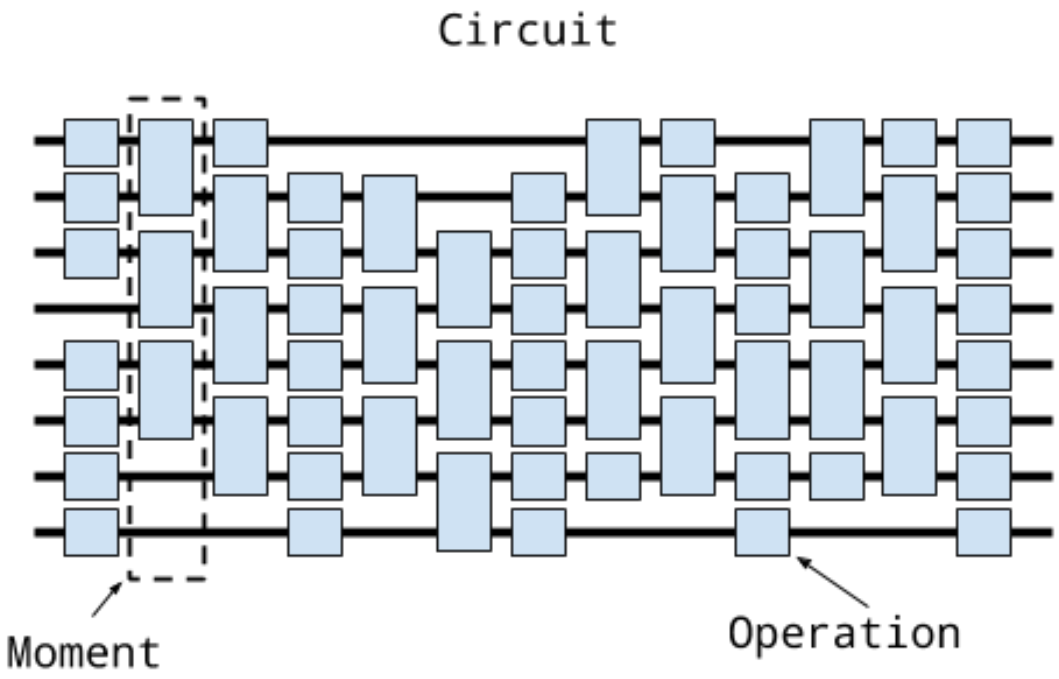
CHAPTER 3

Circuits

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 a 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 an `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:

```
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

```
cz01 = cirq.CZ(qubits[0], qubits[1])
x2 = cirq.X(qubits[2])
cz12 = cirq.CZ(qubits[1], qubits[2])
moment0 = cirq.Moment([cz01, x2])
moment1 = cirq.Moment([cz12])
circuit = cirq.Circuit((moment0, moment1))

print(circuit)
# prints the text diagram for the circuit:
# (0, 0): ———@———————
#            |
# (0, 1): ———@———@———
#                  |
# (0, 2): ———X———@———
```

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

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.

```
from cirq.ops import CZ, H
q0, q1, q2 = [cirq.GridQubit(i, 0) for i in range(3)]
circuit = cirq.Circuit()
circuit.append([CZ(q0, q1), H(q2)])

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

This appended an entire new moment to the qubit, which we can continue to do,

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

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

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

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

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

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

3.3 InsertStrategies

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

`InsertStrategy.EARLIEST` is defined as

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

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

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

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

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.

```
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.

```
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`.

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

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

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.

TODO(dabacon): diagram.

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:

```
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))
# [GateOperation(H, (GridQubit(0, 0),)), GateOperation(H, (GridQubit(1, 0),)),␣
↪GateOperation(H, (GridQubit(2, 0),))]
# [GateOperation(CZ, (GridQubit(1, 0), GridQubit(2, 0)))]
# [GateOperation(H, (GridQubit(0, 0),)), [GateOperation(CZ, (GridQubit(1, 0),␣
↪GridQubit(2, 0)))]]

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

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.

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):

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

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:

```
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:

```
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]`.

CHAPTER 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`.

```
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))
```

4.1 Gate Features

The raw `Gate` class itself simply describes that a `Gate` can be applied to qubits to produce an `Operation`. We then use marker classes for `Gates` indicated what additional features a `Gate` has.

For example, one feature is `ReversibleEffect`. A `Gate` that inherits this class is required to implement the method `inverse` which returns the inverse gate. Algorithms that operate on gates can use `isinstance(gate, ReversibleEffect)` to determine whether gates implements `inverse` method, and then use it. (Note that, even if the gate is not reversible, the algorithm may have been given an `Extension` with a cast from the gate to `ReversibleEffect`. See the extensions documentation for more information.)

We describe some gate features below.

4.1.1 ReversibleEffect, SelfInverseGate

As described above, a `ReversibleEffect` implements the `inverse` method (returns a gatethat is the inverse of the receiving gate). `SelfInverseGate` is a `Gate` for which the `inverse` is simply the `Gate` itself (so the feature `SelfInverseGate` doesn't need to implement `inverse`, it already just returns `self`).

4.1.2 ExtrapolatableEffect

Represents an effect which can be scaled continuously up or down, or negated. Implementing gates and operations implement the `extrapolate_effect` method, which takes a single float parameter `factor`. This factor is the amount to scale the gate by. Roughly, one can think about this as applying the effect `factor` times. There is some subtlety in this definition since, for example, there are often two ways to define the square root of a gate. It is up to the implementation to define which root is chosen.

The primary use of `ExtrapolatableEffect` is to allow easy *powering* of gates. That is one can define for these gates a power

```
import numpy as np
from cirq.ops import X
print(np.around(X.matrix()))
# prints
# [[0.+0.j 1.+0.j]
#  [1.+0.j 0.+0.j]]

sqrt_x = X**0.5
print(sqrt_x.matrix())
# prints
# [[0.5+0.5j 0.5-0.5j]
#  [0.5-0.5j 0.5+0.5j]]
```

The Pauli gates included in Cirq use the convention `Z**0.5  S  np.diag(1, i)`, `Z**-0.5  S**-1`, `X**0.5  H*S*H`, and the square root of `Y` is inferred via the right hand rule. Note that it is often the case that `(g**a)**b != g**(a * b)`, due to the intermediate values normalizing rotation angles into a canonical range.

4.1.3 KnownMatrix

We've seen this above. These are `Gate` or `Operation` instances which implement the `matrix` method. This returns a numpy `ndarray` matrix which is the unitary gate for the gate/operation.

4.1.4 CompositeGate and CompositeOperation

A `CompositeGate` is a gate which consists of multiple gates that can be applied to a given set of qubits. This is a manner in which one can decompose one gate into multiple gates. In particular `CompositeGates` implement the method `default_decompose` which acts on a sequence of qubits, and returns a list of the operations acting on these qubits for the constituents gates.

One thing about `CompositeGates` is that sometimes you want to modify the decomposition. Algorithms that allow this can take an `Extension` which allows for overriding the `CompositeGate`. An example of this is for in `Simulators` where an optional extension can be supplied that can be used to override the CompositeGate.

A `CompositeOperation` is just like a `CompositeGate`, except it already knows the qubits it should be applied to.

4.1.5 TextDiagrammable

Text diagrams of `Circuits` are actually quite useful for visualizing the moment structure of a `Circuit`. Gates that implement this feature can specify compact representations to use in the diagram (e.g. '×' instead of 'SWAP').

4.2 XmonGates

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 `XmonGates` are

ExpWGate This gate is a rotation about a combination of a Pauli `X` and Pauli `Y` gates. The `ExpWGate` takes two parameters, `half_turns` and `axis_half_turns`. The later describes the angle of the operator that is being rotated about in the `XY` plane. In particular if we define `W(theta) = cos(pi theta) X + sin (pi theta) Y` then `axis_half_turns` is `theta`. And the full gate is `exp(-i pi half_turns W(axis_half_turns) / 2)`.

ExpZGate This gate is a rotation about the Pauli `Z` axis. The gate is `exp(-i pi Z half_turns / 2)` where `half_turns` is the supplied parameter. Note that in quantum computing hardware, this gate is often compiled out of the circuit (TODO: explain this in more detail)

Exp11Gate This is a two qubit gate and is a rotation about the `|11><11|` projector. It takes a single parameter `half_turns` and is the gate `exp(i pi |11><11| half_turns)`.

XmonMeasurementGate This is a single qubit measurement in the computational basis.

4.3 CommonGates

`XmonGates` are hardware specific. In addition Cirq has a number of more commonly named gates that are then implemented as `XmonGates` via an extension or composite gates. Some of these are our old friends:

RotXGate, **RotYGate**, **RotZGate**, **Rot11Gate**. These are gates corresponding to the Pauli rotations or (in the case of `Rot11Gate` a two qubit rotation).

Our old friends the Paulis: **X**, **Y**, and **Z**. Some other two qubit fiends, **CZ** the controlled-Z gate, **CNOT** the controlled-X gate, and **SWAP** the swap gate. As well as some other Clifford friends, **H** and **S**, and our error correcting friend **T**.

TODO: describe these in more detail.

4.4 Extensions

TODO

CHAPTER 5

Simulation

Cirq comes with a built in Python simulator for testing out small circuits. This simulator can shard its simulation across different processes/threads and so take advantage of multiple cores/CPUs.

Currently the simulator is tailored to the gate set from Google's Xmon architecture, but the simulator can be used to run general gate sets, assuming that you provide an implementation in terms of this basic gate set.

Here is a simple circuit

```
import cirq
from cirq import Circuit
from cirq.devices import GridQubit
from cirq.google import ExpWGate, Exp11Gate, XmonMeasurementGate

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

def basic_circuit(meas=True):
    sqrt_x = ExpWGate(half_turns=0.5, axis_half_turns=0.0)
    cz = Exp11Gate()
    yield sqrt_x(q0), sqrt_x(q1)
    yield cz(q0, q1)
    yield sqrt_x(q0), sqrt_x(q1)
    if meas:
        yield XmonMeasurementGate(key='q0')(q0), XmonMeasurementGate(key='q1')(q1)

circuit = 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.google.Simulator` and passing the circuit into its `run` method:

```
from cirq.google import XmonSimulator
simulator = XmonSimulator()
result = simulator.run(circuit)

print(result)
# prints something like
# q0=1 q1=1
```

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

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

```
result = simulator.run(circuit)

print(result)
# prints something like
# q0=1 q1=0
```

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:

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

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

Note that the simulator uses numpy's `float32` precision (which is `complex64` for complex numbers).

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).

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

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

```
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 outer-most loop, the last qubit's computational basis values are looped over in the inner-most loop, etc:

```
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:

```
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.]
```

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`:

```
circuit = 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(), 3)))
# prints something like
# state at step 0: [ 0.5+0.j   0.0+0.5j  0.0+0.5j -0.5+0.j ]
# state at step 1: [ 0.5+0.j   0.0+0.5j  0.0+0.5j  0.5+0.j ]
# state at step 2: [-0.5-0.j  -0.0+0.5j -0.0+0.5j -0.5+0.j ]
# state at step 3: [ 0.+0.j  0.+0.j -0.+1.j  0.+0.j]
```

The object returned by the `moment_steps` iterator is a `XmonStepResult`. 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`XmonStepResult` 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.

5.3 Gate sets

The xmon simulator is designed to work with operations that are either a `GateOperation` applying an `XmonGate`, a `CompositeOperation` that decomposes (recursively) to `XmonGates`, or a 1-qubit or 2-qubit operation with a `KnownMatrix`. By default the xmon simulator uses an `Extension` defined in `xgate_gate_extensions` to try to resolve gates that are not `XmonGates` to `XmonGates`.

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

- Define it directly as an `XmonGate`.
- Provide a `CompositeGate` made up of `XmonGates`.
- Supply an `Exentension` to the simulator which converts the gate to an `XmonGate` or to a `CompositeGate` which itself can be decomposed in `XmonGates`.

5.4 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.

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

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

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

```
resolvers = [ParamResolver({'x': y / 2.0}) for y in range(3)]
circuit = Circuit()
circuit.append([rot_w_gate(q0), rot_w_gate(q1)])
circuit.append([XmonMeasurementGate(key='q0')(q0), XmonMeasurementGate(key='q1')(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
```

(continues on next page)

(continued from previous page)

```
# 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 `Symbol`s.

TODO(dabacon): Describe the iterable of parameterized resolvers supported by Google's API.

5.5 Simulation 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.

CHAPTER 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.)

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:

```
import cirq
from cirq.devices import GridQubit
from cirq.google import XmonGate
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) or
              not isinstance(operation.gate, XmonGate)):
```

(continues on next page)

(continued from previous page)

```
            raise ValueError('{!r} is not an XmonGate'.format(operation))
        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:

```
from cirq.google.xmon_gates import Exp11Gate
device = Xmon10Device()
circuit = cirq.Circuit()
CZ = Exp11Gate(half_turns=1.0)
circuit.append([CZ(device.qubits[0], device.qubits[2])])
try:
  device.validate_circuit(circuit)
except ValueError as e:
  print(e)
# prints something like
# ValueError: Non-local interaction: Operation(Exp11Gate(half_turns=1.0),␣
↪(GridQubit(0, 0), GridQubit(2, 0)))
```

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

```
from cirq.google.xmon_gates import ExpWGate
circuit = cirq.Circuit()
CZ = Exp11Gate(half_turns=1.0)
X = ExpWGate(half_turns=1.0)
circuit.append([CZ(device.qubits[0], device.qubits[1]), X(device.qubits[0])])
print(circuit)
# prints:
# (0, 0): ———Z———X———
```

(continues on next page)

(continued from previous page)

```
#              |
# (1, 0): ———Z————————
```

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

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

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

```
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

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

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

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

CHAPTER 7

API Reference

7.1 Circuits

Circuit(moments, device)	A mutable list of groups of operations to apply to some qubits.
Moment(operations)	A simplified time-slice of operations within a sequenced circuit.
InsertStrategy(name, doc)	Indicates preferences on how to add multiple operations to a circuit.
OP_TREE	Union type; Union[X, Y] means either X or Y.

7.1.1 cirq.Circuit

class cirq.**Circuit**(*moments: Iterable[cirq.circuits.moment.Moment] = ()*, *device: cirq.devices.device.Device = UnconstrainedDevice*)

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 operation_at qubits findall_operations to_unitary_matrix apply_unitary_effect_to_state to_text_diagram to_text_diagram_drawer

Methods for mutation: insert append insert_into_range clear_operations_touching

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(...)]

__init__(*moments: Iterable[cirq.circuits.moment.Moment] = ()*, *device: cirq.devices.device.Device = UnconstrainedDevice*) → None
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

all_operations()	Iterates over the operations applied by this circuit.
all_qubits()	Returns the qubits acted upon by Operations in this circuit.
append(moment_or_operation_tree, ...)	Appends operations onto the end of the circuit.
apply_unitary_effect_to_state(initial_state, ...)	Left-multiplies a state vector by the circuit's unitary effect.
are_all_measurements_terminal()	
batch_insert(insertions, ...)	Applies a batched insert operation to the circuit.
batch_insert_into(insert_intos, ...)	Inserts operations into empty spaces in existing moments.
batch_remove(removals, ...)	Removes several operations from a circuit.
clear_operations_touching(qubits, moment_indices)	Clears operations that are touching given qubits at given moments.
copy()	
findall_operations(predicate, bool])	Find the locations of all operations that satisfy a given condition.
findall_operations_with_gate_type(gate_type)	Find 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 middle of the circuit.
insert_at_frontier(operations, ...)	Inserts operations inline at frontier.
insert_into_range(operations, ...)	Writes operations inline into an area of the circuit.
is_parameterized(ext)	Whether the effect is parameterized.
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.
save_qasm(file_path, bytes, int], header, ...)	Save a QASM file equivalent to the circuit.

Continued on next page

Table 2 – continued from previous page

to_qasm(header, precision, qubit_order, ...)	Returns QASM equivalent to the circuit.
to_text_diagram(ext, use_unicode_characters, ...)	Returns text containing a diagram describing the circuit.
to_text_diagram_drawer(ext, ...)	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.
with_parameters_resolved_by(param_resolver, ext)	Resolve the parameters in the effect.

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

Circuit.**append**(*moment_or_operation_tree: Union[cirq.circuits.moment.Moment, cirq.ops.raw_types.Operation, typing.Iterable[typing.Any]], strategy: cirq.circuits.insert_strategy.InsertStrategy = InsertStrategy.NEW_THEN_INLINE*)

Appends operations onto the end of the circuit.

Parameters

- **moment_or_operation_tree** – An operation or tree of operations.
- **strategy** – How to pick/create the moment to put operations into.

cirq.Circuit.apply_unitary_effect_to_state

Circuit.**apply_unitary_effect_to_state**(*initial_state: Union[int, numpy.ndarray] = 0, qubit_order: Union[cirq.ops.qubit_order.QubitOrder, typing.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, ext: cirq.extension.extensions.Extensions = None*) → numpy.ndarray

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 circuit.to_unitary_matrix(...), but computed in a more efficient way.

Parameters

- **qubit_order** – Determines how qubits are ordered when passing matrices into np.kron.
- **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.
- **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.
- **ext** – The extensions to use when attempting to cast operations into KnownMatrix instances.

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, typing.Iterable[typing.Any]]]]*`)` → None

Applies a batched insert operation to the circuit.

Transparently handles the fact that earlier insertions may shift the index that later insertions should occur at. For example, if you insert an operation at index 2 and at index 4, but the insert at index 2 causes a new moment to be created, then the insert at "4" will actually occur at index 5 to account for the shift from the new moment.

All insertions are done with the strategy 'EARLIEST'.

Parameters **insertions** – A sequence of (insert_index, operations) pairs indicating operations to add into the circuit at specific places.

cirq.Circuit.batch_insert_into

`Circuit.`**`batch_insert_into`**`(`*insert_intos: Iterable[Tuple[int, cirq.ops.raw_types.Operation]]*`)` → None

Inserts operations into empty spaces in existing moments.

If any of the insertions fails (due to colliding with an existing operation), this method fails without making any changes to the circuit.

Parameters **insert_intos** – A sequence of (moment_index, new_operation) pairs indicating a moment to add a new operation into.

ValueError: One of the insertions collided with an existing operation.

IndexError: Inserted into a moment index that doesn't exist.

cirq.Circuit.batch_remove

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

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

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

cirq.Circuit.findall_operations

Circuit.**findall_operations**(*predicate: Callable[cirq.ops.raw_types.Operation, bool]*) → Iterable[Tuple[int, cirq.ops.raw_types.Operation]]

Find the locations of all operations that satisfy a given condition.

This returns an iterator of (index, operation) tuples where each operation satisfies op_cond(operation) is truthy. The indices are in order of the moments and then order of the ops within that moment.

Parameters **predicate** – A method that takes an Operation and returns a Truthy value indicating the operation meets the find condition.

Returns An iterator (index, operation)'s that satisfy the op_condition.

cirq.Circuit.findall_operations_with_gate_type

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. RotXGate 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 = InsertStrategy.NEW_THEN_INLINE, device: cirq.devices.device.Device = 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

Circuit.**insert**(*index: int, moment_or_operation_tree: Union[cirq.circuits.moment.Moment, cirq.ops.raw_types.Operation, typing.Iterable[typing.Any]], strategy: cirq.circuits.insert_strategy.InsertStrategy = InsertStrategy.NEW_THEN_INLINE*) → int

Inserts operations into the middle of the circuit.

Parameters

- **index** – The index to insert all of the operations at.
- **moment_or_operation_tree** – An operation or tree of operations.
- **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

- IndexError – Bad insertion index.
- ValueError – Bad insertion strategy.

cirq.Circuit.insert_at_frontier

Circuit.**insert_at_frontier**(*operations: Union[cirq.ops.raw_types.Operation, typing.Iterable[typing.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, typing.Iterable[typing.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.is_parameterized

Circuit.**is_parameterized**(*ext: cirq.extension.extensions.Extensions = None*) → bool

Whether the effect is parameterized.

Returns True if the gate has any unresolved Symbols and False otherwise.

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*) → Union[int, NoneType]

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*) → Union[cirq.ops.raw_types.Operation, NoneType]

Finds the operation on a qubit within a moment, if any.

Parameters

- **`qubit`** – The qubit to check for an operation on.
- **`moment_index`** – The index of the moment to check for an operation within. Allowed to be beyond the end of the circuit.

Returns None if there is no operation on the qubit at the given moment, or else the operation.

cirq.Circuit.prev_moment_operating_on

`Circuit.`**`prev_moment_operating_on`**(*qubits: Sequence[cirq.ops.raw_types.QubitId], end_moment_index: Union[int, NoneType] = None, max_distance: Union[int, NoneType] = None*) → Union[int, NoneType]

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.save_qasm

Circuit.**save_qasm**(*file_path: Union[str, bytes, int], header: str = 'Generated from Cirq', precision: int = 10, qubit_order: Union[cirq.ops.qubit_order.QubitOrder, typing.Iterable[cirq.ops.raw_types.QubitId]] = <cirq.ops.qubit_order.QubitOrder object>, ext: cirq.extension.extensions.Extensions = None*) → None

Save a QASM file equivalent to the circuit.

Parameters

- **header** – A multi-line string that is placed in a comment at the top of the QASM.
- **precision** – Number of digits to use when representing numbers.
- **qubit_order** – Determines how qubits are ordered in the QASM register.
- **ext** – For extending operations/gates to implement QasmConvertableOperation/QasmConvertableGate.

cirq.Circuit.to_qasm

Circuit.**to_qasm**(*header: str = 'Generated from Cirq', precision: int = 10, qubit_order: Union[cirq.ops.qubit_order.QubitOrder, typing.Iterable[cirq.ops.raw_types.QubitId]] = <cirq.ops.qubit_order.QubitOrder object>, ext: cirq.extension.extensions.Extensions = None*) → str

Returns QASM equivalent to the circuit.

Parameters

- **header** – A multi-line string that is placed in a comment at the top of the QASM.
- **precision** – Number of digits to use when representing numbers.
- **qubit_order** – Determines how qubits are ordered in the QASM register.
- **ext** – For extending operations/gates to implement QasmConvertableOperation/QasmConvertableGate.

cirq.Circuit.to_text_diagram

Circuit.**to_text_diagram**(*ext: cirq.extension.extensions.Extensions = None, use_unicode_characters: bool = True, transpose: bool = False, precision: Union[int, NoneType] = 3, qubit_order: Union[cirq.ops.qubit_order.QubitOrder, typing.Iterable[cirq.ops.raw_types.QubitId]] = <cirq.ops.qubit_order.QubitOrder object>*) → str

Returns text containing a diagram describing the circuit.

Parameters

- **ext** – For extending operations/gates to implement TextDiagrammable.
- **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

Circuit.**to_text_diagram_drawer**(*ext: cirq.extension.extensions.Extensions = None, use_unicode_characters: bool = True, qubit_name_suffix: str = '', precision: Union[int, NoneType] = 3, qubit_order: Union[cirq.ops.qubit_order.QubitOrder, typing.Iterable[cirq.ops.raw_types.QubitId]] = <cirq.ops.qubit_order.QubitOrder object>*) → cirq.circuits.text_diagram_drawer.TextDiagramDrawer

Returns a TextDiagramDrawer with the circuit drawn into it.

Parameters

- **ext** – For extending operations/gates to implement TextDiagrammable.
- **use_unicode_characters** – Determines if unicode characters are allowed (as opposed to ascii-only diagrams).
- **qubit_name_suffix** – Appended to qubit names in the diagram.
- **precision** – Number of digits to use when representing numbers.
- **qubit_order** – Determines how qubits are ordered in the diagram.

Returns The TextDiagramDrawer instance.

cirq.Circuit.to_unitary_matrix

Circuit.**to_unitary_matrix**(*qubit_order: Union[cirq.ops.qubit_order.QubitOrder, typing.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, ext: cirq.extension.extensions.Extensions = None*) → 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.
- **ext** – The extensions to use when attempting to cast operations into KnownMatrix instances.
- **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.

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

Circuit.**with_device**(*new_device: cirq.devices.device.Device*, *qubit_mapping: Callable[cirq.ops.raw_types.QubitId, cirq.ops.raw_types.QubitId] = <function Circuit.<lambda>>*) → cirq.circuits.circuit.Circuit

Maps the current circuit onto a new device, and validates.

Parameters

- **new_device** – The new device that the circuit should be on.
- **qubit_mapping** – How to translate qubits from the old device into qubits on the new device.

Returns The translated circuit.

cirq.Circuit.with_parameters_resolved_by

Circuit.**with_parameters_resolved_by**(*param_resolver: cirq.study.resolver.ParamResolver*, *ext: cirq.extension.extensions.Extensions = None*) → cirq.circuits.circuit.Circuit

Resolve the parameters in the effect.

Returns a gate or operation of the same type, but with all Symbols replaced with floats according to the given ParamResolver.

Attributes

device

cirq.Circuit.device

Circuit.**device**

7.1.2 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.

__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

operates_on(qubits)	Determines if the moment has operations touching the given qubits.
with_operation(operation)	Returns an equal moment, but with the given op added.
without_operations_touching(qubits)	Returns an equal moment, but without ops on the given qubits.

cirq.Moment.operates_on

`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.with_operation

`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

`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.

7.1.3 cirq.InsertStrategy

class `cirq`.**InsertStrategy**(*name*, *doc*)
Indicates preferences on how to add multiple operations to a circuit.

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

Methods

—

Attributes

EARLIEST
INLINE
NEW
NEW_THEN_INLINE

cirq.InsertStrategy.EARLIEST

InsertStrategy.**EARLIEST = InsertStrategy.EARLIEST**

cirq.InsertStrategy.INLINE

InsertStrategy.**INLINE = InsertStrategy.INLINE**

cirq.InsertStrategy.NEW

InsertStrategy.**NEW = InsertStrategy.NEW**

cirq.InsertStrategy.NEW_THEN_INLINE

InsertStrategy.**NEW_THEN_INLINE = InsertStrategy.NEW_THEN_INLINE**

7.1.4 cirq.OP_TREE

cirq.**OP_TREE = typing.Union[cirq.ops.raw_types.Operation, typing.Iterable[typing.Any]]**

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].

7.2 Operations

Operation	An effect applied to a collection of qubits.
GateOperation(gate, qubits)	An application of a gate to a collection of qubits.
CompositeOperation	An operation with a known decomposition into simpler operations.
QasmConvertableOperation	An operation that knows its representation in QASM.

7.2.1 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**

7.2.2 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 collection of qubits.

gate
The applied gate.

qubits
A sequence of the qubits on which the gate is applied.

__init__(*gate: cirq.ops.raw_types.Gate, qubits: Sequence[cirq.ops.raw_types.QubitId]*) → None
Initialize self. See help(type(self)) for accurate signature.

Methods

default_decompose()	
extrapolate_effect(factor)	
inverse()	
is_parameterized()	
known_qasm_output(args)	
matrix()	
phase_by(phase_turns, qubit_index)	
text_diagram_info(args)	
trace_distance_bound()	
transform_qubits(func, ...)	Returns the same operation, but with different qubits.
try_cast_to(desired_type, extensions)	Turns this value into the desired type, if possible.
with_gate(new_gate)	
with_parameters_resolved_by(param_resolver)	
with_qubits(*new_qubits)	

cirq.GateOperation.default_decompose

GateOperation.**default_decompose**()

cirq.GateOperation.extrapolate_effect

GateOperation.**extrapolate_effect**(*factor: float*) → cirq.ops.gate_operation.GateOperation

cirq.GateOperation.inverse

GateOperation.**inverse**() → cirq.ops.gate_operation.GateOperation

cirq.GateOperation.is_parameterized

GateOperation.**is_parameterized**() → bool

cirq.GateOperation.known_qasm_output

GateOperation.**known_qasm_output**(*args: cirq.ops.gate_features.QasmOutputArgs*) → Union[str, NoneType]

cirq.GateOperation.matrix

GateOperation.**matrix**() → numpy.ndarray

cirq.GateOperation.phase_by

GateOperation.**phase_by**(*phase_turns: float*, *qubit_index: int*) → cirq.ops.gate_operation.GateOperation

cirq.GateOperation.text_diagram_info

GateOperation.**text_diagram_info**(*args: cirq.ops.gate_features.TextDiagramInfoArgs*) → cirq.ops.gate_features.TextDiagramInfo

cirq.GateOperation.trace_distance_bound

GateOperation.**trace_distance_bound**() → float

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.try_cast_to

GateOperation.**try_cast_to**(*desired_type*, *extensions*)
Turns this value into the desired type, if possible.

Correct implementations should delegate to super() after failing to cast, instead of returning None.

Parameters

- **desired_type** – The type of thing that the caller wants to use.
- **extensions** – The extensions instance that is asking us to try to cast ourselves into something as part of its try_cast method. If we need to recursively cast some of our fields in order to cast ourselves, this is the extensions instance we should use.

Returns

None if the receiving instance doesn't recognize or can't implement the desired type. Otherwise a value that meets the interface.

cirq.GateOperation.with_gate

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

cirq.GateOperation.with_parameters_resolved_by

GateOperation.**with_parameters_resolved_by**(*param_resolver: study.ParamResolver*) → GateOperation

cirq.GateOperation.with_qubits

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

Attributes

gate
qubits

cirq.GateOperation.gate

GateOperation.**gate**

cirq.GateOperation.qubits

GateOperation.**qubits**

7.2.3 cirq.CompositeOperation

class cirq.**CompositeOperation**
An operation with a known decomposition into simpler operations.

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

Methods

default_decompose()	Yields simpler operations for performing the receiving operation.

cirq.CompositeOperation.default_decompose

CompositeOperation.**default_decompose**() → Union[cirq.ops.raw_types.Operation, typing.Iterable[typing.Any]]
Yields simpler operations for performing the receiving operation.

7.2.4 cirq.QasmConvertableOperation

class cirq.**QasmConvertableOperation**
An operation that knows its representation in QASM.

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

Methods

known_qasm_output(args)	Returns lines of QASM output representing the operation or None if a simple conversion is not possible.

cirq.QasmConvertableOperation.known_qasm_output

QasmConvertableOperation.**known_qasm_output**(*args: cirq.ops.gate_features.QasmOutputArgs*) → Union[str, NoneType]
Returns lines of QASM output representing the operation or None if a simple conversion is not possible.

7.3 Schedules

Schedule(device, scheduled_operations)	A quantum program with operations happening at specific times.
ScheduledOperation(time, duration, operation)	An operation that happens over a specified time interval.
Duration(*, picos, float] = 0, nanos, float] = 0)	A time delta that supports picosecond accuracy.
Timestamp(*, picos, float] = 0, nanos, ...)	A location in time with picosecond accuracy.

7.3.1 cirq.Schedule

class cirq.**Schedule**(*device: cirq.devices.device.Device*, *scheduled_operations: Iterable[cirq.schedules.scheduled_operation.ScheduledOperation] = ()*)
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.

__init__(*device: cirq.devices.device.Device*, *scheduled_operations: Iterable[cirq.schedules.scheduled_operation.ScheduledOperation] = ()*) → None
Initializes a new schedule.

Parameters

- **device** – The hardware this schedule will run on.
- **scheduled_operations** – Initial list of operations to apply. These will be moved into a sorted list, with a key equal to each operation's start time.

Methods

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

cirq.Schedule.exclude

Schedule.**exclude**(*scheduled_operation: cirq.schedules.scheduled_operation.ScheduledOperation*) → bool
Omits a scheduled operation from the schedule, if present.

Parameters **scheduled_operation** – The operation to try to remove.

Returns True if the operation was present and is now removed, False if it was already not present.

cirq.Schedule.include

Schedule.**include**(*scheduled_operation: cirq.schedules.scheduled_operation.ScheduledOperation*)
Adds a scheduled operation to the schedule.

Parameters **scheduled_operation** – The operation to add.

Raises ValueError – The operation collided with something already in the schedule.

cirq.Schedule.operations_happening_at_same_time_as

Schedule.**operations_happening_at_same_time_as**(*scheduled_operation: cirq.schedules.scheduled_operation.ScheduledOperation*) → List[cirq.schedules.scheduled_operation.ScheduledOpera

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 = 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.

7.3.2 cirq.ScheduledOperation

class cirq.**ScheduledOperation**(*time: cirq.value.timestamp.Timestamp*, *duration: cirq.value.duration.Duration*, *operation: cirq.ops.raw_types.Operation*)

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

static ScheduledOperation.**op_at_on**(*operation: cirq.ops.raw_types.Operation*, *time: cirq.value.timestamp.Timestamp*, *device: cirq.devices.device.Device*)

Creates a scheduled operation with a device-determined duration.

7.3.3 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*) → None

Initializes a Duration with a time specified in ns and/or ps.

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

total_nanos()	Returns the number of nanoseconds that the duration spans.
total_picos()	Returns the number of picoseconds that the duration spans.

cirq.Duration.total_nanos

Duration.**total_nanos**() → float

Returns the number of nanoseconds that the duration spans.

cirq.Duration.total_picos

Duration.**total_picos**() → float

Returns the number of picoseconds that the duration spans.

7.3.4 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*) → 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()	The timestamp's location in picoseconds from arbitrary time zero.

cirq.Timestamp.raw_picos

Timestamp.**raw_picos**() → float
The timestamp's location in picoseconds from arbitrary time zero.

7.3.5 Gates

Gate	An operation type that can be applied to a collection of qubits.
MeasurementGate(key, invert_mask, . . .] = ())	Indicates that qubits should be measured plus a key to identify results.

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).

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

Methods

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.Gate.on

Gate.**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.Gate.validate_args

Gate.**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.

cirq.MeasurementGate

class cirq.**MeasurementGate**(*key: str = ''*, *invert_mask: Tuple[bool, ...] = ()*)
Indicates that qubits should be measured plus a key to identify results.

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 shorted. Qubits with indices past the end of the mask are not flipped.

__init__(*key: str = ''*, *invert_mask: Tuple[bool, ...] = ()*) → None
Initialize self. See help(type(self)) for accurate signature.

Methods

is_measurement(op, cirq.ops.raw_types.Operation])	
known_qasm_output(qubits, ...], args)	Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.
on(*qubits)	Returns an application of this gate to the given qubits.
text_diagram_info(args)	Describes how to draw something in a text diagram.
validate_args(qubits)	Checks if this gate can be applied to the given qubits.
with_bits_flipped(*bit_positions)	Toggles whether or not the measurement inverts various outputs.

cirq.MeasurementGate.is_measurement

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

cirq.MeasurementGate.known_qasm_output

MeasurementGate.**known_qasm_output**(*qubits: Tuple[cirq.ops.raw_types.QubitId, ...], args: cirq.ops.gate_features.QasmOutputArgs*) → Union[str, NoneType]

Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.

cirq.MeasurementGate.on

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.

cirq.MeasurementGate.text_diagram_info

MeasurementGate.**text_diagram_info**(*args: cirq.ops.gate_features.TextDiagramInfoArgs*) → cirq.ops.gate_features.TextDiagramInfo

Describes how to draw something in a text diagram.

Parameters **args** – A TextDiagramInfoArgs instance encapsulating various pieces of information (e.g. how many qubits are we being applied to) as well as user options (e.g. whether to avoid unicode characters).

Returns A TextDiagramInfo instance describing what to print.

cirq.MeasurementGate.validate_args

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.

Gate Features and Effects

KnownMatrix	An effect that can be described by a matrix.
ParameterizableEffect	An effect that can be parameterized by Symbols.
CompositeGate	A gate with a known decomposition into simpler gates.
ExtrapolatableEffect	A gate whose effect can be continuously scaled up/down/negated.
ReversibleEffect	A gate whose effect can be undone in a known way.
InterchangeableQubitsGate	Indicates operations should be equal under some qubit permutations.
PhaseableEffect	An effect that can be phased around the Z axis of target qubits.
TextDiagrammable	A thing which can be printed in a text diagram.
BoundedEffect	An effect with known bounds on how easy it is to detect.
SingleQubitGate	A gate that must be applied to exactly one qubit.
TwoQubitGate	A gate that must be applied to exactly two qubits.
QasmConvertableGate	A gate that knows its representation in QASM.
EigenGate(*, exponent, float] = 1.0)	A gate with a known eigendecomposition.

cirq.KnownMatrix

class cirq.**KnownMatrix**

An effect that can be described by a matrix.

__init__()

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

Methods

matrix()	The unitary matrix of the gate/operation.

cirq.KnownMatrix.matrix

KnownMatrix.**matrix**() → numpy.ndarray

The unitary matrix of the gate/operation.

The matrix order is implicit for both gates and operations. For a gate, the matrix must be in order with respect to the list of qubits that the gate is applied to. For an operation, the order must be with respect to its qubits attribute. The qubit-to-amplitude order mapping matches the ordering of numpy.kron(A, B), where A is a qubit earlier in the list than the qubit B.

For example, when applying a CNOT gate the control qubit goes first and so the CNOT gate's matrix is:

1 _ _ _ _ 1 _ _ _ _ _ 1 _ _ 1 _

cirq.ParameterizableEffect

class cirq.**ParameterizableEffect**

An effect that can be parameterized by Symbols.

__init__()

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

Methods

is_parameterized()	Whether the effect is parameterized.
with_parameters_resolved_by(param_resolver)	Resolve the parameters in the effect.

cirq.ParameterizableEffect.is_parameterized

ParameterizableEffect.**is_parameterized**() → bool

Whether the effect is parameterized.

Returns True if the gate has any unresolved Symbols and False otherwise.

cirq.ParameterizableEffect.with_parameters_resolved_by

ParameterizableEffect.**with_parameters_resolved_by**(*param_resolver: cirq.study.resolver.ParamResolver*) → TSelf_ParameterizableEffect

Resolve the parameters in the effect.

Returns a gate or operation of the same type, but with all Symbols replaced with floats according to the given ParamResolver.

cirq.CompositeGate

class cirq.**CompositeGate**

A gate with a known decomposition into simpler gates.

__init__()

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

Methods

default_decompose(qubits)	Yields operations for performing this gate on the given qubits.

cirq.CompositeGate.default_decompose

CompositeGate.**default_decompose**(*qubits: Sequence[cirq.ops.raw_types.QubitId]*) → Union[cirq.ops.raw_types.Operation, typing.Iterable[typing.Any]]

Yields operations for performing this gate on the given qubits.

Parameters **qubits** – The qubits the gate should be applied to.

cirq.ExtrapolatableEffect

class cirq.**ExtrapolatableEffect**

A gate whose effect can be continuously scaled up/down/negated.

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

Methods

extrapolate_effect(factor)	Augments, diminishes, or reverses the effect of the receiving gate.
inverse()	Returns a gate with an exactly opposite effect.

cirq.ExtrapolatableEffect.extrapolate_effect

ExtrapolatableEffect.**extrapolate_effect**(*factor: float*) → TSelf_ExtrapolatableEffect
Augments, diminishes, or reverses the effect of the receiving gate.

Parameters **factor** – The amount to scale the gate's effect by.

Returns A gate equivalent to applying the receiving gate 'factor' times.

cirq.ExtrapolatableEffect.inverse

ExtrapolatableEffect.**inverse**() → TSelf_ExtrapolatableEffect
Returns a gate with an exactly opposite effect.

cirq.ReversibleEffect

class cirq.**ReversibleEffect**
A gate whose effect can be undone in a known way.

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

Methods

inverse()	Returns a gate with an exactly opposite effect.

cirq.ReversibleEffect.inverse

ReversibleEffect.**inverse**() → cirq.ops.gate_features.ReversibleEffect
Returns a gate with an exactly opposite effect.

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.PhaseableEffect

class cirq.**PhaseableEffect**

An effect that can be phased around the Z axis of target qubits.

__init__()

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

Methods

phase_by(phase_turns, qubit_index)	Returns a phased version of the effect.

cirq.PhaseableEffect.phase_by

PhaseableEffect.**phase_by**(*phase_turns: float*, *qubit_index: int*) → TSelf_PhaseableEffect

Returns a phased version of the effect.

For example, an X gate phased by 90 degrees would be a Y gate.

Parameters

- **phase_turns** – The amount to phase the gate, in fractions of a whole turn.
- **qubit_index** – The index of the target qubit the phasing applies to.

Returns The phased gate or operation.

cirq.TextDiagrammable

class cirq.**TextDiagrammable**

A thing which can be printed in a text diagram.

__init__()

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

Methods

text_diagram_info(args)	Describes how to draw something in a text diagram.

cirq.TextDiagrammable.text_diagram_info

TextDiagrammable.**text_diagram_info**(*args: cirq.ops.gate_features.TextDiagramInfoArgs*) → cirq.ops.gate_features.TextDiagramInfo

Describes how to draw something in a text diagram.

Parameters **args** – A TextDiagramInfoArgs instance encapsulating various pieces of information (e.g. how many qubits are we being applied to) as well as user options (e.g. whether to avoid unicode characters).

Returns A TextDiagramInfo instance describing what to print.

cirq.BoundedEffect

class cirq.**BoundedEffect**

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__()

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

Methods

trace_distance_bound()	A maximum on the trace distance between this effect's input/output.

cirq.BoundedEffect.trace_distance_bound

BoundedEffect.**trace_distance_bound**() → float

A maximum on the trace distance between this effect's input/output.

Generally this method is used when deciding whether to keep gates, so only the behavior near 0 is important. Approximations that overestimate the maximum trace distance are permitted. Even ones that exceed 1. Underestimates are not permitted.

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

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.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, typing.Iterable[typing.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.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.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

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.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.

cirq.QasmConvertableGate

class cirq.**QasmConvertableGate**
A gate that knows its representation in QASM.

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

Methods

known_qasm_output(qubits, ...], args)	Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.

cirq.QasmConvertableGate.known_qasm_output

QasmConvertableGate.**known_qasm_output**(*qubits: Tuple[cirq.ops.raw_types.QubitId, ...], args: cirq.ops.gate_features.QasmOutputArgs*)
→ Union[str, NoneType]
Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.

cirq.EigenGate

class cirq.**EigenGate**(**, exponent: Union[cirq.value.symbol.Symbol, float] = 1.0*)
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*) → None

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

Methods

extrapolate_effect(factor)	
inverse()	
is_parameterized()	Whether the effect is parameterized.
matrix()	
on(*qubits)	Returns an application of this gate to the given qubits.
trace_distance_bound()	A maximum on the trace distance between this effect's input/output.
try_cast_to(desired_type, ext)	Turns this value into the desired type, if possible.
validate_args(qubits)	Checks if this gate can be applied to the given qubits.
with_parameters_resolved_by(param_resolver)	Resolve the parameters in the effect.

cirq.EigenGate.extrapolate_effect

EigenGate.**extrapolate_effect**(*factor: float*) → TSelf

cirq.EigenGate.inverse

EigenGate.**inverse**() → TSelf

cirq.EigenGate.is_parameterized

EigenGate.**is_parameterized**() → bool
Whether the effect is parameterized.

Returns True if the gate has any unresolved Symbols and False otherwise.

cirq.EigenGate.matrix

EigenGate.**matrix**() → numpy.ndarray

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.

cirq.EigenGate.trace_distance_bound

EigenGate.**trace_distance_bound**()
A maximum on the trace distance between this effect's input/output.

Generally this method is used when deciding whether to keep gates, so only the behavior near 0 is important. Approximations that overestimate the maximum trace distance are permitted. Even ones that exceed 1. Underestimates are not permitted.

cirq.EigenGate.try_cast_to

EigenGate.**try_cast_to**(*desired_type*, *ext*)

Turns this value into the desired type, if possible.

Correct implementations should delegate to super() after failing to cast, instead of returning None.

Parameters

- **desired_type** – The type of thing that the caller wants to use.
- **extensions** – The extensions instance that is asking us to try to cast ourselves into something as part of its try_cast method. If we need to recursively cast some of our fields in order to cast ourselves, this is the extensions instance we should use.

Returns

None if the receiving instance doesn't recognize or can't implement the desired type. Otherwise a value that meets the interface.

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.

cirq.EigenGate.with_parameters_resolved_by

EigenGate.**with_parameters_resolved_by**(*param_resolver*) → TSelf

Resolve the parameters in the effect.

Returns a gate or operation of the same type, but with all Symbols replaced with floats according to the given ParamResolver.

Single Qubit Gates

RotXGate(*, half_turns, float, ...)	Fixed rotation around the X axis of the Bloch sphere.
RotYGate(*, half_turns, float, ...)	Fixed rotation around the Y axis of the Bloch sphere.
RotZGate(*, half_turns, float, ...)	Fixed rotation around the Z axis of the Bloch sphere.
HGate	180 degree rotation around the X+Z axis of the Bloch sphere.
X	Fixed rotation around the X axis of the Bloch sphere.
Y	Fixed rotation around the Y axis of the Bloch sphere.
Z	Fixed rotation around the Z axis of the Bloch sphere.

Continued on next page

Table 36 – continued from previous page

H	180 degree rotation around the X+Z axis of the Bloch sphere.
S	Fixed rotation around the Z axis of the Bloch sphere.
T	Fixed rotation around the Z axis of the Bloch sphere.

cirq.RotXGate

class cirq.**RotXGate**(*, *half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *rads: Union[float, NoneType] = None*, *degs: Union[float, NoneType] = None*)

Fixed rotation around the X axis of the Bloch sphere.

__init__(*, *half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *rads: Union[float, NoneType] = None*, *degs: Union[float, NoneType] = None*) → None

Initializes the gate.

At most one angle argument may be specified. If more are specified, the result is considered ambiguous and an error is thrown. If no angle argument is given, the default value of one half turn is used.

Parameters

- **half_turns** – The relative phasing of X's eigenstates, in half_turns.
- **rads** – The relative phasing of X's eigenstates, in radians.
- **degs** – The relative phasing of X's eigenstates, in degrees.

Methods

extrapolate_effect(factor)	
inverse()	
is_parameterized()	Whether the effect is parameterized.
known_qasm_output(qubits, ...], args)	Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.
matrix()	
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.
text_diagram_info(args)	Describes how to draw something in a text diagram.
trace_distance_bound()	A maximum on the trace distance between this effect's input/output.
try_cast_to(desired_type, ext)	Turns this value into the desired type, if possible.
validate_args(qubits)	Checks if this gate can be applied to the given qubits.
with_parameters_resolved_by(param_resolver)	Resolve the parameters in the effect.

cirq.RotXGate.extrapolate_effect

RotXGate.**extrapolate_effect**(*factor: float*) → TSelf

cirq.RotXGate.inverse

RotXGate.**inverse**() → TSelf

cirq.RotXGate.is_parameterized

RotXGate.**is_parameterized**() → bool
Whether the effect is parameterized.

Returns True if the gate has any unresolved Symbols and False otherwise.

cirq.RotXGate.known_qasm_output

RotXGate.**known_qasm_output**(*qubits: Tuple[cirq.ops.raw_types.QubitId, ...], args: cirq.ops.gate_features.QasmOutputArgs*) → Union[str, NoneType]
Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.

cirq.RotXGate.matrix

RotXGate.**matrix**() → numpy.ndarray

cirq.RotXGate.on

RotXGate.**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.RotXGate.on_each

RotXGate.**on_each**(*targets: Iterable[cirq.ops.raw_types.QubitId]*) → Union[cirq.ops.raw_types.Operation, typing.Iterable[typing.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.RotXGate.text_diagram_info

RotXGate.**text_diagram_info**(*args: cirq.ops.gate_features.TextDiagramInfoArgs*) → cirq.ops.gate_features.TextDiagramInfo
Describes how to draw something in a text diagram.

Parameters **args** – A TextDiagramInfoArgs instance encapsulating various pieces of information (e.g. how many qubits are we being applied to) as well as user options (e.g. whether to avoid unicode characters).

Returns A TextDiagramInfo instance describing what to print.

cirq.RotXGate.trace_distance_bound

RotXGate.**trace_distance_bound**()
A maximum on the trace distance between this effect's input/output.

Generally this method is used when deciding whether to keep gates, so only the behavior near 0 is important. Approximations that overestimate the maximum trace distance are permitted. Even ones that exceed 1. Underestimates are not permitted.

cirq.RotXGate.try_cast_to

RotXGate.**try_cast_to**(*desired_type*, *ext*)
Turns this value into the desired type, if possible.

Correct implementations should delegate to super() after failing to cast, instead of returning None.

Parameters

- **desired_type** – The type of thing that the caller wants to use.
- **extensions** – The extensions instance that is asking us to try to cast ourselves into something as part of its try_cast method. If we need to recursively cast some of our fields in order to cast ourselves, this is the extensions instance we should use.

Returns

None if the receiving instance doesn't recognize or can't implement the desired type. Otherwise a value that meets the interface.

cirq.RotXGate.validate_args

RotXGate.**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.RotXGate.with_parameters_resolved_by

RotXGate.**with_parameters_resolved_by**(*param_resolver*) → TSelf
Resolve the parameters in the effect.

Returns a gate or operation of the same type, but with all Symbols replaced with floats according to the given ParamResolver.

Attributes

half_turns

cirq.RotXGate.half_turns

RotXGate.**half_turns**

cirq.RotYGate

class cirq.**RotYGate**(*, *half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *rads: Union[float, NoneType] = None*, *degs: Union[float, NoneType] = None*)

Fixed rotation around the Y axis of the Bloch sphere.

__init__(*, *half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *rads: Union[float, NoneType] = None*, *degs: Union[float, NoneType] = None*) → None

Initializes the gate.

At most one angle argument may be specified. If more are specified, the result is considered ambiguous and an error is thrown. If no angle argument is given, the default value of one half turn is used.

Parameters

- **half_turns** – The relative phasing of Y's eigenstates, in half_turns.
- **rads** – The relative phasing of Y's eigenstates, in radians.
- **degs** – The relative phasing of Y's eigenstates, in degrees.

Methods

extrapolate_effect(factor)	
inverse()	
is_parameterized()	Whether the effect is parameterized.
known_qasm_output(qubits, ...], args)	Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.
matrix()	
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.
text_diagram_info(args)	Describes how to draw something in a text diagram.
trace_distance_bound()	A maximum on the trace distance between this effect's input/output.
try_cast_to(desired_type, ext)	Turns this value into the desired type, if possible.
validate_args(qubits)	Checks if this gate can be applied to the given qubits.
with_parameters_resolved_by(param_resolver)	Resolve the parameters in the effect.

cirq.RotYGate.extrapolate_effect

RotYGate.**extrapolate_effect**(*factor: float*) → TSelf

cirq.RotYGate.inverse

RotYGate.**inverse**() → TSelf

cirq.RotYGate.is_parameterized

RotYGate.**is_parameterized**() → bool
Whether the effect is parameterized.

Returns True if the gate has any unresolved Symbols and False otherwise.

cirq.RotYGate.known_qasm_output

RotYGate.**known_qasm_output**(*qubits: Tuple[cirq.ops.raw_types.QubitId, ...], args: cirq.ops.gate_features.QasmOutputArgs*) → Union[str, NoneType]
Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.

cirq.RotYGate.matrix

RotYGate.**matrix**() → numpy.ndarray

cirq.RotYGate.on

RotYGate.**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.RotYGate.on_each

RotYGate.**on_each**(*targets: Iterable[cirq.ops.raw_types.QubitId]*) → Union[cirq.ops.raw_types.Operation, typing.Iterable[typing.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.RotYGate.text_diagram_info

RotYGate.**text_diagram_info**(*args: cirq.ops.gate_features.TextDiagramInfoArgs*) → cirq.ops.gate_features.TextDiagramInfo
Describes how to draw something in a text diagram.

Parameters **args** – A TextDiagramInfoArgs instance encapsulating various pieces of information (e.g. how many qubits are we being applied to) as well as user options (e.g. whether to avoid unicode characters).

Returns A TextDiagramInfo instance describing what to print.

cirq.RotYGate.trace_distance_bound

RotYGate.**trace_distance_bound**()
A maximum on the trace distance between this effect's input/output.

Generally this method is used when deciding whether to keep gates, so only the behavior near 0 is important. Approximations that overestimate the maximum trace distance are permitted. Even ones that exceed 1. Underestimates are not permitted.

cirq.RotYGate.try_cast_to

RotYGate.**try_cast_to**(*desired_type*, *ext*)
Turns this value into the desired type, if possible.

Correct implementations should delegate to super() after failing to cast, instead of returning None.

Parameters

- **desired_type** – The type of thing that the caller wants to use.
- **extensions** – The extensions instance that is asking us to try to cast ourselves into something as part of its try_cast method. If we need to recursively cast some of our fields in order to cast ourselves, this is the extensions instance we should use.

Returns

None if the receiving instance doesn't recognize or can't implement the desired type. Otherwise a value that meets the interface.

cirq.RotYGate.validate_args

RotYGate.**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.RotYGate.with_parameters_resolved_by

RotYGate.**with_parameters_resolved_by**(*param_resolver*) → TSelf
Resolve the parameters in the effect.

Returns a gate or operation of the same type, but with all Symbols replaced with floats according to the given ParamResolver.

Attributes

half_turns

cirq.RotYGate.half_turns

RotYGate.**half_turns**

cirq.RotZGate

class cirq.**RotZGate**(*, *half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *rads: Union[float, NoneType] = None*, *degs: Union[float, NoneType] = None*)

Fixed rotation around the Z axis of the Bloch sphere.

__init__(*, *half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *rads: Union[float, NoneType] = None*, *degs: Union[float, NoneType] = None*) → None

Initializes the gate.

At most one angle argument may be specified. If more are specified, the result is considered ambiguous and an error is thrown. If no angle argument is given, the default value of one half turn is used.

Parameters

- **half_turns** – The relative phasing of Z's eigenstates, in half_turns.
- **rads** – The relative phasing of Z's eigenstates, in radians.
- **degs** – The relative phasing of Z's eigenstates, in degrees.

Methods

extrapolate_effect(factor)	
inverse()	
is_parameterized()	Whether the effect is parameterized.
known_qasm_output(qubits, ...], args)	Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.
matrix()	
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.
phase_by(phase_turns, qubit_index)	Returns a phased version of the effect.
text_diagram_info(args)	Describes how to draw something in a text diagram.
trace_distance_bound()	A maximum on the trace distance between this effect's input/output.
try_cast_to(desired_type, ext)	Turns this value into the desired type, if possible.
validate_args(qubits)	Checks if this gate can be applied to the given qubits.
with_parameters_resolved_by(param_resolver)	Resolve the parameters in the effect.

cirq.RotZGate.extrapolate_effect

RotZGate.**extrapolate_effect**(*factor: float*) → TSelf

cirq.RotZGate.inverse

RotZGate.**inverse**() → TSelf

cirq.RotZGate.is_parameterized

RotZGate.**is_parameterized**() → bool
Whether the effect is parameterized.

Returns True if the gate has any unresolved Symbols and False otherwise.

cirq.RotZGate.known_qasm_output

RotZGate.**known_qasm_output**(*qubits: Tuple[cirq.ops.raw_types.QubitId, ...], args: cirq.ops.gate_features.QasmOutputArgs*) → Union[str, NoneType]
Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.

cirq.RotZGate.matrix

RotZGate.**matrix**() → numpy.ndarray

cirq.RotZGate.on

RotZGate.**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.RotZGate.on_each

RotZGate.**on_each**(*targets: Iterable[cirq.ops.raw_types.QubitId]*) → Union[cirq.ops.raw_types.Operation, typing.Iterable[typing.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.RotZGate.phase_by

RotZGate.**phase_by**(*phase_turns: float*, *qubit_index: int*)
Returns a phased version of the effect.

For example, an X gate phased by 90 degrees would be a Y gate.

Parameters

- **phase_turns** – The amount to phase the gate, in fractions of a whole turn.
- **qubit_index** – The index of the target qubit the phasing applies to.

Returns The phased gate or operation.

cirq.RotZGate.text_diagram_info

RotZGate.**text_diagram_info**(*args: cirq.ops.gate_features.TextDiagramInfoArgs*) → cirq.ops.gate_features.TextDiagramInfo

Describes how to draw something in a text diagram.

Parameters **args** – A TextDiagramInfoArgs instance encapsulating various pieces of information (e.g. how many qubits are we being applied to) as well as user options (e.g. whether to avoid unicode characters).

Returns A TextDiagramInfo instance describing what to print.

cirq.RotZGate.trace_distance_bound

RotZGate.**trace_distance_bound**()

A maximum on the trace distance between this effect's input/output.

Generally this method is used when deciding whether to keep gates, so only the behavior near 0 is important. Approximations that overestimate the maximum trace distance are permitted. Even ones that exceed 1. Underestimates are not permitted.

cirq.RotZGate.try_cast_to

RotZGate.**try_cast_to**(*desired_type*, *ext*)

Turns this value into the desired type, if possible.

Correct implementations should delegate to super() after failing to cast, instead of returning None.

Parameters

- **desired_type** – The type of thing that the caller wants to use.
- **extensions** – The extensions instance that is asking us to try to cast ourselves into something as part of its try_cast method. If we need to recursively cast some of our fields in order to cast ourselves, this is the extensions instance we should use.

Returns

None if the receiving instance doesn't recognize or can't implement the desired type. Otherwise a value that meets the interface.

cirq.RotZGate.validate_args

RotZGate.**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.RotZGate.with_parameters_resolved_by

RotZGate.**with_parameters_resolved_by**(*param_resolver*) → TSelf
Resolve the parameters in the effect.

Returns a gate or operation of the same type, but with all Symbols replaced with floats according to the given ParamResolver.

Attributes

half_turns

cirq.RotZGate.half_turns

RotZGate.**half_turns**

cirq.HGate

class cirq.**HGate**
180 degree rotation around the X+Z axis of the Bloch sphere.

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

Methods

default_decompose(qubits)	Yields operations for performing this gate on the given qubits.
inverse()	Returns a gate with an exactly opposite effect.
known_qasm_output(qubits, ...], args)	Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.
matrix()	See base class.
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.
text_diagram_info(args)	Describes how to draw something in a text diagram.
validate_args(qubits)	Checks if this gate can be applied to the given qubits.

cirq.HGate.default_decompose

HGate.**default_decompose**(*qubits*)
Yields operations for performing this gate on the given qubits.

Parameters **qubits** – The qubits the gate should be applied to.

cirq.HGate.inverse

HGate.**inverse**()
Returns a gate with an exactly opposite effect.

cirq.HGate.known_qasm_output

HGate.**known_qasm_output**(*qubits: Tuple[cirq.ops.raw_types.QubitId, ...], args: cirq.ops.gate_features.QasmOutputArgs*) → Union[str, NoneType]
Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.

cirq.HGate.matrix

HGate.**matrix**()
See base class.

cirq.HGate.on

HGate.**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.HGate.on_each

HGate.**on_each**(*targets: Iterable[cirq.ops.raw_types.QubitId]*) → Union[cirq.ops.raw_types.Operation, typing.Iterable[typing.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.HGate.text_diagram_info

HGate.**text_diagram_info**(*args: cirq.ops.gate_features.TextDiagramInfoArgs*) → cirq.ops.gate_features.TextDiagramInfo
Describes how to draw something in a text diagram.

Parameters **args** – A TextDiagramInfoArgs instance encapsulating various pieces of information (e.g. how many qubits are we being applied to) as well as user options (e.g. whether to avoid unicode characters).

Returns A TextDiagramInfo instance describing what to print.

cirq.HGate.validate_args

HGate.**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 = X**
Fixed rotation around the X axis of the Bloch sphere.

cirq.Y

cirq.**Y = Y**
Fixed rotation around the Y axis of the Bloch sphere.

cirq.Z

cirq.**Z = Z**
Fixed rotation around the Z axis of the Bloch sphere.

cirq.H

cirq.**H = H**
180 degree rotation around the X+Z axis of the Bloch sphere.

cirq.S

cirq.**S = S**
Fixed rotation around the Z axis of the Bloch sphere.

cirq.T

cirq.**T = T**
Fixed rotation around the Z axis of the Bloch sphere.

Two Qubit Gates

Rot11Gate(*, half_turns, float, ...)	Phases the |11> state of two adjacent qubits by a fixed amount.
CNotGate(*, half_turns, float, ...)	A controlled-NOT.
SwapGate(*, half_turns, float] = 1.0)	Swaps two qubits.
ISwapGate(*, exponent, float] = 1.0)	Rotates the |01-vs-|10 subspace of two qubits around its Bloch X-axis.
CZ	Phases the |11> state of two adjacent qubits by a fixed amount.

Continued on next page

Table 44 – continued from previous page

CNOT	A controlled-NOT.
ISWAP	Rotates the |01-vs-|10 subspace of two qubits around its Bloch X-axis.

cirq.Rot11Gate

class cirq.**Rot11Gate**(*, *half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *rads: Union[float, NoneType] = None*, *degs: Union[float, NoneType] = None*)

Phases the |11> state of two adjacent qubits by a fixed amount.

A ParameterizedCZGate guaranteed to not be using the parameter key field.

__init__(*, *half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *rads: Union[float, NoneType] = None*, *degs: Union[float, NoneType] = None*) → None

Initializes the gate.

At most one angle argument may be specified. If more are specified, the result is considered ambiguous and an error is thrown. If no angle argument is given, the default value of one half turn is used.

Parameters

- **half_turns** – Relative phasing of CZ's eigenstates, in half_turns.
- **rads** – Relative phasing of CZ's eigenstates, in radians.
- **degs** – Relative phasing of CZ's eigenstates, in degrees.

Methods

extrapolate_effect(factor)	
inverse()	
is_parameterized()	Whether the effect is parameterized.
known_qasm_output(qubits, ...], args)	Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.
matrix()	
on(*qubits)	Returns an application of this gate to the given qubits.
phase_by(phase_turns, qubit_index)	Returns a phased version of the effect.
qubit_index_to_equivalence_group_key(index)	Returns a key that differs between non-interchangeable qubits.
text_diagram_info(args)	Describes how to draw something in a text diagram.
trace_distance_bound()	A maximum on the trace distance between this effect's input/output.
try_cast_to(desired_type, ext)	Turns this value into the desired type, if possible.
validate_args(qubits)	Checks if this gate can be applied to the given qubits.
with_parameters_resolved_by(param_resolver)	Resolve the parameters in the effect.

cirq.Rot11Gate.extrapolate_effect

Rot11Gate.**extrapolate_effect**(*factor: float*) → TSelf

cirq.Rot11Gate.inverse

Rot11Gate.**inverse**() → TSelf

cirq.Rot11Gate.is_parameterized

Rot11Gate.**is_parameterized**() → bool
Whether the effect is parameterized.

Returns True if the gate has any unresolved Symbols and False otherwise.

cirq.Rot11Gate.known_qasm_output

Rot11Gate.**known_qasm_output**(*qubits: Tuple[cirq.ops.raw_types.QubitId, ...], args: cirq.ops.gate_features.QasmOutputArgs*) → Union[str, NoneType]
Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.

cirq.Rot11Gate.matrix

Rot11Gate.**matrix**() → numpy.ndarray

cirq.Rot11Gate.on

Rot11Gate.**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.Rot11Gate.phase_by

Rot11Gate.**phase_by**(*phase_turns*, *qubit_index*)
Returns a phased version of the effect.

For example, an X gate phased by 90 degrees would be a Y gate.

Parameters

- **phase_turns** – The amount to phase the gate, in fractions of a whole turn.
- **qubit_index** – The index of the target qubit the phasing applies to.

Returns The phased gate or operation.

cirq.Rot11Gate.qubit_index_to_equivalence_group_key

Rot11Gate.**qubit_index_to_equivalence_group_key**(*index: int*) → int
Returns a key that differs between non-interchangeable qubits.

cirq.Rot11Gate.text_diagram_info

Rot11Gate.**text_diagram_info**(*args: cirq.ops.gate_features.TextDiagramInfoArgs*) → cirq.ops.gate_features.TextDiagramInfo

Describes how to draw something in a text diagram.

Parameters **args** – A TextDiagramInfoArgs instance encapsulating various pieces of information (e.g. how many qubits are we being applied to) as well as user options (e.g. whether to avoid unicode characters).

Returns A TextDiagramInfo instance describing what to print.

cirq.Rot11Gate.trace_distance_bound

Rot11Gate.**trace_distance_bound**()

A maximum on the trace distance between this effect's input/output.

Generally this method is used when deciding whether to keep gates, so only the behavior near 0 is important. Approximations that overestimate the maximum trace distance are permitted. Even ones that exceed 1. Underestimates are not permitted.

cirq.Rot11Gate.try_cast_to

Rot11Gate.**try_cast_to**(*desired_type*, *ext*)

Turns this value into the desired type, if possible.

Correct implementations should delegate to super() after failing to cast, instead of returning None.

Parameters

- **desired_type** – The type of thing that the caller wants to use.
- **extensions** – The extensions instance that is asking us to try to cast ourselves into something as part of its try_cast method. If we need to recursively cast some of our fields in order to cast ourselves, this is the extensions instance we should use.

Returns

None if the receiving instance doesn't recognize or can't implement the desired type. Otherwise a value that meets the interface.

cirq.Rot11Gate.validate_args

Rot11Gate.**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.Rot11Gate.with_parameters_resolved_by

Rot11Gate.**with_parameters_resolved_by**(*param_resolver*) → TSelf

Resolve the parameters in the effect.

Returns a gate or operation of the same type, but with all Symbols replaced with floats according to the given ParamResolver.

Attributes

half_turns

cirq.Rot11Gate.half_turns

Rot11Gate.**half_turns**

cirq.CNotGate

class cirq.**CNotGate**(*, *half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *rads: Union[float, NoneType] = None*, *degs: Union[float, NoneType] = None*)

A controlled-NOT. Toggle the second qubit when the first qubit is on.

__init__(*, *half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *rads: Union[float, NoneType] = None*, *degs: Union[float, NoneType] = None*) → None

Initializes the gate.

At most one angle argument may be specified. If more are specified, the result is considered ambiguous and an error is thrown. If no angle argument is given, the default value of one half turn is used.

Parameters

- **half_turns** – Relative phasing of CNOT's eigenstates, in half_turns.
- **rads** – Relative phasing of CNOT's eigenstates, in radians.
- **degs** – Relative phasing of CNOT's eigenstates, in degrees.

Methods

default_decompose(qubits)	Yields operations for performing this gate on the given qubits.
extrapolate_effect(factor)	
inverse()	
is_parameterized()	Whether the effect is parameterized.
known_qasm_output(qubits, ...], args)	Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.
matrix()	
on(*qubits)	Returns an application of this gate to the given qubits.
text_diagram_info(args)	Describes how to draw something in a text diagram.

Continued on next page

Table 47 – continued from previous page

trace_distance_bound()	A maximum on the trace distance between this effect's input/output.
try_cast_to(desired_type, ext)	Turns this value into the desired type, if possible.
validate_args(qubits)	Checks if this gate can be applied to the given qubits.
with_parameters_resolved_by(param_resolver)	Resolve the parameters in the effect.

cirq.CNotGate.default_decompose

CNotGate.**default_decompose**(*qubits*)

Yields operations for performing this gate on the given qubits.

Parameters **qubits** – The qubits the gate should be applied to.

cirq.CNotGate.extrapolate_effect

CNotGate.**extrapolate_effect**(*factor: float*) → TSelf

cirq.CNotGate.inverse

CNotGate.**inverse**() → TSelf

cirq.CNotGate.is_parameterized

CNotGate.**is_parameterized**() → bool

Whether the effect is parameterized.

Returns True if the gate has any unresolved Symbols and False otherwise.

cirq.CNotGate.known_qasm_output

CNotGate.**known_qasm_output**(*qubits: Tuple[cirq.ops.raw_types.QubitId, ...], args: cirq.ops.gate_features.QasmOutputArgs*) → Union[str, NoneType]

Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.

cirq.CNotGate.matrix

CNotGate.**matrix**() → numpy.ndarray

cirq.CNotGate.on

CNotGate.**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.CNotGate.text_diagram_info

CNotGate.**text_diagram_info**(*args: cirq.ops.gate_features.TextDiagramInfoArgs*) → cirq.ops.gate_features.TextDiagramInfo

Describes how to draw something in a text diagram.

Parameters **args** – A TextDiagramInfoArgs instance encapsulating various pieces of information (e.g. how many qubits are we being applied to) as well as user options (e.g. whether to avoid unicode characters).

Returns A TextDiagramInfo instance describing what to print.

cirq.CNotGate.trace_distance_bound

CNotGate.**trace_distance_bound**()

A maximum on the trace distance between this effect's input/output.

Generally this method is used when deciding whether to keep gates, so only the behavior near 0 is important. Approximations that overestimate the maximum trace distance are permitted. Even ones that exceed 1. Underestimates are not permitted.

cirq.CNotGate.try_cast_to

CNotGate.**try_cast_to**(*desired_type*, *ext*)

Turns this value into the desired type, if possible.

Correct implementations should delegate to super() after failing to cast, instead of returning None.

Parameters

- **desired_type** – The type of thing that the caller wants to use.
- **extensions** – The extensions instance that is asking us to try to cast ourselves into something as part of its try_cast method. If we need to recursively cast some of our fields in order to cast ourselves, this is the extensions instance we should use.

Returns

None if the receiving instance doesn't recognize or can't implement the desired type. Otherwise a value that meets the interface.

cirq.CNotGate.validate_args

CNotGate.**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.CNotGate.with_parameters_resolved_by

CNotGate.**with_parameters_resolved_by**(*param_resolver*) → TSelf

Resolve the parameters in the effect.

Returns a gate or operation of the same type, but with all Symbols replaced with floats according to the given ParamResolver.

Attributes

half_turns

cirq.CNotGate.half_turns

CNotGate.**half_turns**

cirq.SwapGate

class cirq.**SwapGate**(*, *half_turns: Union[cirq.value.symbol.Symbol, float] = 1.0*)

Swaps two qubits.

__init__(*, *half_turns: Union[cirq.value.symbol.Symbol, float] = 1.0*) → None

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

Methods

default_decompose(qubits)	See base class.
extrapolate_effect(factor)	
inverse()	
is_parameterized()	Whether the effect is parameterized.
known_qasm_output(qubits, ...], args)	Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.
matrix()	
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.
text_diagram_info(args)	Describes how to draw something in a text diagram.
trace_distance_bound()	A maximum on the trace distance between this effect's input/output.
try_cast_to(desired_type, ext)	Turns this value into the desired type, if possible.
validate_args(qubits)	Checks if this gate can be applied to the given qubits.
with_parameters_resolved_by(param_resolver)	Resolve the parameters in the effect.

cirq.SwapGate.default_decompose

SwapGate.**default_decompose**(*qubits*)
See base class.

cirq.SwapGate.extrapolate_effect

SwapGate.**extrapolate_effect**(*factor: float*) → TSelf

cirq.SwapGate.inverse

SwapGate.**inverse**() → TSelf

cirq.SwapGate.is_parameterized

SwapGate.**is_parameterized**() → bool
Whether the effect is parameterized.

Returns True if the gate has any unresolved Symbols and False otherwise.

cirq.SwapGate.known_qasm_output

SwapGate.**known_qasm_output**(*qubits: Tuple[cirq.ops.raw_types.QubitId, ...], args: cirq.ops.gate_features.QasmOutputArgs*) → Union[str, None-Type]
Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.

cirq.SwapGate.matrix

SwapGate.**matrix**() → numpy.ndarray

cirq.SwapGate.on

SwapGate.**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.SwapGate.qubit_index_to_equivalence_group_key

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

cirq.SwapGate.text_diagram_info

SwapGate.**text_diagram_info**(*args: cirq.ops.gate_features.TextDiagramInfoArgs*) → cirq.ops.gate_features.TextDiagramInfo

Describes how to draw something in a text diagram.

Parameters **args** – A TextDiagramInfoArgs instance encapsulating various pieces of information (e.g. how many qubits are we being applied to) as well as user options (e.g. whether to avoid unicode characters).

Returns A TextDiagramInfo instance describing what to print.

cirq.SwapGate.trace_distance_bound

SwapGate.**trace_distance_bound**()

A maximum on the trace distance between this effect's input/output.

Generally this method is used when deciding whether to keep gates, so only the behavior near 0 is important. Approximations that overestimate the maximum trace distance are permitted. Even ones that exceed 1. Underestimates are not permitted.

cirq.SwapGate.try_cast_to

SwapGate.**try_cast_to**(*desired_type*, *ext*)

Turns this value into the desired type, if possible.

Correct implementations should delegate to super() after failing to cast, instead of returning None.

Parameters

- **desired_type** – The type of thing that the caller wants to use.
- **extensions** – The extensions instance that is asking us to try to cast ourselves into something as part of its try_cast method. If we need to recursively cast some of our fields in order to cast ourselves, this is the extensions instance we should use.

Returns

None if the receiving instance doesn't recognize or can't implement the desired type. Otherwise a value that meets the interface.

cirq.SwapGate.validate_args

SwapGate.**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.SwapGate.with_parameters_resolved_by

SwapGate.**with_parameters_resolved_by**(*param_resolver*) → TSelf

Resolve the parameters in the effect.

Returns a gate or operation of the same type, but with all Symbols replaced with floats according to the given ParamResolver.

Attributes

half_turns

cirq.SwapGate.half_turns

SwapGate.**half_turns**

cirq.ISwapGate

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

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:

ISWAPt exp(+i π t (XX + YY) / 4)**

[1 0 0 0] [0 cos(π·t/2) i·sin(π·t/2) 0] [0 i·sin(π·t/2) cos(π·t/2) 0] [0 0 0 1]

__init__(*, *exponent: Union[cirq.value.symbol.Symbol, float] = 1.0*) → None

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

Methods

default_decompose(qubits)	Yields operations for performing this gate on the given qubits.
extrapolate_effect(factor)	
inverse()	
is_parameterized()	Whether the effect is parameterized.
matrix()	
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.
text_diagram_info(args)	Describes how to draw something in a text diagram.
trace_distance_bound()	A maximum on the trace distance between this effect's input/output.
try_cast_to(desired_type, ext)	Turns this value into the desired type, if possible.
validate_args(qubits)	Checks if this gate can be applied to the given qubits.
with_parameters_resolved_by(param_resolver)	Resolve the parameters in the effect.

cirq.ISwapGate.default_decompose

ISwapGate.**default_decompose**(*qubits*)
Yields operations for performing this gate on the given qubits.

Parameters **qubits** – The qubits the gate should be applied to.

cirq.ISwapGate.extrapolate_effect

ISwapGate.**extrapolate_effect**(*factor: float*) → TSelf

cirq.ISwapGate.inverse

ISwapGate.**inverse**() → TSelf

cirq.ISwapGate.is_parameterized

ISwapGate.**is_parameterized**() → bool
Whether the effect is parameterized.

Returns True if the gate has any unresolved Symbols and False otherwise.

cirq.ISwapGate.matrix

ISwapGate.**matrix**() → numpy.ndarray

cirq.ISwapGate.on

ISwapGate.**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.ISwapGate.qubit_index_to_equivalence_group_key

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

cirq.ISwapGate.text_diagram_info

ISwapGate.**text_diagram_info**(*args: cirq.ops.gate_features.TextDiagramInfoArgs*) → cirq.ops.gate_features.TextDiagramInfo
Describes how to draw something in a text diagram.

Parameters **args** – A TextDiagramInfoArgs instance encapsulating various pieces of information (e.g. how many qubits are we being applied to) as well as user options (e.g. whether to avoid unicode characters).

Returns A TextDiagramInfo instance describing what to print.

cirq.ISwapGate.trace_distance_bound

ISwapGate.**trace_distance_bound**()
A maximum on the trace distance between this effect's input/output.

Generally this method is used when deciding whether to keep gates, so only the behavior near 0 is important. Approximations that overestimate the maximum trace distance are permitted. Even ones that exceed 1. Underestimates are not permitted.

cirq.ISwapGate.try_cast_to

ISwapGate.**try_cast_to**(*desired_type*, *ext*)
Turns this value into the desired type, if possible.

Correct implementations should delegate to super() after failing to cast, instead of returning None.

Parameters

- **desired_type** – The type of thing that the caller wants to use.
- **extensions** – The extensions instance that is asking us to try to cast ourselves into something as part of its try_cast method. If we need to recursively cast some of our fields in order to cast ourselves, this is the extensions instance we should use.

Returns

None if the receiving instance doesn't recognize or can't implement the desired type. Otherwise a value that meets the interface.

cirq.ISwapGate.validate_args

ISwapGate.**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.ISwapGate.with_parameters_resolved_by

ISwapGate.**with_parameters_resolved_by**(*param_resolver*) → TSelf
Resolve the parameters in the effect.

Returns a gate or operation of the same type, but with all Symbols replaced with floats according to the given ParamResolver.

Attributes

exponent

cirq.ISwapGate.exponent

ISwapGate.**exponent**

cirq.CZ

cirq.**CZ = CZ**

Phases the |11> state of two adjacent qubits by a fixed amount.

A ParameterizedCZGate guaranteed to not be using the parameter key field.

cirq.CNOT

cirq.**CNOT = CNOT**

A controlled-NOT. Toggle the second qubit when the first qubit is on.

cirq.ISWAP

cirq.**ISWAP = 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:

ISWAP**t exp(+i π t (XX + YY) / 4)**

[1 0 0 0] [0 cos(π·t/2) i·sin(π·t/2) 0] [0 i·sin(π·t/2) cos(π·t/2) 0] [0 0 0 1]

Three Qubit Gates

CCZ	A doubly-controlled-Z.
CCX	A doubly-controlled-NOT.
CSWAP	A controlled swap gate.
TOFFOLI	A doubly-controlled-NOT.
FREDKIN	A controlled swap gate.

cirq.CCZ

cirq.**CCZ = CCZ**

A doubly-controlled-Z.

cirq.CCX

cirq.**CCX = TOFFOLI**

A doubly-controlled-NOT. The Toffoli gate.

cirq.CSWAP

cirq.**CSWAP = FREDKIN**
A controlled swap gate. The Fredkin gate.

cirq.TOFFOLI

cirq.**TOFFOLI = TOFFOLI**
A doubly-controlled-NOT. The Toffoli gate.

cirq.FREDKIN

cirq.**FREDKIN = FREDKIN**
A controlled swap gate. The Fredkin gate.

7.4 Qubits

General classes for qubits and related concepts.

QubitId	Identifies a qubit.
NamedQubit(name)	A qubit identified by name.
LineQubit(x)	A qubit on a 1d lattice with nearest-neighbor connectivity.
GridQubit(row, col)	A qubit on a 2d square lattice.
QubitOrder(explicit_func, ...)	Defines the kronecker product order of qubits.
QubitOrderOrList	Union type; Union[X, Y] means either X or Y.
QubitOrder.DEFAULT	A basis that orders qubits based on their names.

7.4.1 cirq.QubitId

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

Child classes must be equatable and hashable.

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

7.4.2 cirq.NamedQubit

class cirq.**NamedQubit**(*name: str*)
A qubit identified by name.

__init__(*name: str*) → None
Initialize self. See help(type(self)) for accurate signature.

Methods

—

7.4.3 cirq.LineQubit

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

is_adjacent(other)	Determines if two qubits are adjacent line qubits.
range(*range_args)	Returns a range of line qubits.

cirq.LineQubit.is_adjacent

LineQubit.**is_adjacent**(*other: cirq.ops.raw_types.QubitId*) → bool

Determines if two qubits are adjacent line qubits.

cirq.LineQubit.range

static LineQubit.**range**(**range_args*) → List[_ForwardRef('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.

7.4.4 cirq.GridQubit

class cirq.**GridQubit**(*row*, *col*)

A qubit on a 2d square lattice.

GridQubits use row-major ordering:

GridQubit(0, 0) < GridQubit(0, 1) < GridQubit(1, 0) < GridQubit(1, 1)

__init__(*row*, *col*)

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

Methods

from_proto(q)	
is_adjacent(other)	Determines if two qubits are adjacent qubits.
to_proto(out)	Return the proto form, mutating supplied form if supplied.

cirq.GridQubit.from_proto

static GridQubit.**from_proto**(*q: cirq.api.google.v1.operations_pb2.Qubit*) → cirq.devices.grid_qubit.GridQubit

cirq.GridQubit.is_adjacent

GridQubit.**is_adjacent**(*other: cirq.ops.raw_types.QubitId*) → bool

Determines if two qubits are adjacent qubits.

cirq.GridQubit.to_proto

GridQubit.**to_proto**(*out: cirq.api.google.v1.operations_pb2.Qubit = None*) → cirq.api.google.v1.operations_pb2.Qubit

Return the proto form, mutating supplied form if supplied.

7.4.5 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

as_qubit_order(val)	Converts a value into a basis.
explicit(fixed_qubits, fallback, ...)	A basis that contains exactly the given qubits in the given order.
map(internalize, TInternalQubit], ...)	Transforms the Basis so that it applies to wrapped qubits.
order_for(qubits)	Returns a qubit tuple ordered corresponding to the basis.
sorted_by(key, Any])	A basis that orders qubits ascending based on a key function.

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: Union[_ForwardRef('QubitOrder'), NoneType] = 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[Any, 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 based on their names.

cirq.QubitOrder.DEFAULT

QubitOrder.**DEFAULT = <cirq.ops.qubit_order.QubitOrder object>**
A basis that orders qubits based on their names.

7.4.6 cirq.QubitOrderOrList

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

7.5 Devices

Classes characterizing constraints of hardware.

Device	Hardware constraints for validating circuits and schedules.
UnconstrainedDevice	A device that allows everything.

7.5.1 cirq.Device

class cirq.**Device**

Hardware constraints for validating circuits and schedules.

__init__()

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

Methods

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

cirq.Device.can_add_operation_into_moment

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

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

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

Parameters

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

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

cirq.Device.decompose_operation

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

Returns a device-valid decomposition for the given operation.

This method is used when adding operations into circuits with a device specified, to avoid spurious failures due to e.g. using a Hadamard gate instead of ExpWGate.

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.

7.5.2 cirq.UnconstrainedDevice

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

7.6 Placement

Classes for placing circuits onto circuits.

LinePlacementStrategy	Choice and options for the line placement calculation method.
GreedySequenceSearchStrategy(algorithm)	Greedy search method for linear sequence of qubits on a chip.
AnnealSequenceSearchStrategy(trace_func, ...)	Linearized sequence search using simulated annealing method.
line_on_device(device, length, method)	Searches for linear sequence of qubits on device.

7.6.1 cirq.LinePlacementStrategy

class cirq.**LinePlacementStrategy**
Choice and options for the line placement calculation method.

Currently two methods are available: cirq.line.GreedySequenceSearchMethod and cirq.line.AnnealSequenceSearchMethod.

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

Methods

place_line(device, length)	Runs line sequence search.

cirq.LinePlacementStrategy.place_line

LinePlacementStrategy.**place_line**(*device: cirq.google.xmon_device.XmonDevice*, *length: int*) → cirq.line.placement.sequence.GridQubitLineTuple
Runs line sequence search.

Parameters

- **device** – Chip description.
- **length** – Required line length.

Returns Linear sequences found on the chip.

7.6.2 cirq.GreedySequenceSearchStrategy

class cirq.**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.GreedySequenceSearchStrategy.place_line

GreedySequenceSearchStrategy.**place_line**(*device: cirq.google.xmon_device.XmonDevice*, *length: int*) → cirq.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.

Attributes

BEST

cirq.GreedySequenceSearchStrategy.BEST

GreedySequenceSearchStrategy.**BEST = 'best'**

7.6.3 cirq.AnnealSequenceSearchStrategy

class cirq.**AnnealSequenceSearchStrategy**(*trace_func: Callable[[List[List[cirq.devices.grid_qubit.GridQubit]], float, float, float, bool], NoneType] = 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.

__init__(*trace_func: Callable[[List[List[cirq.devices.grid_qubit.GridQubit]], float, float, float, bool], NoneType] = None, seed: int = None*) → None
Linearized sequence search using simulated annealing method.

Parameters

- **trace_func** – Optional callable which will be called for each simulated annealing step with arguments: solution candidate (list of linear sequences on the chip), current temperature (float), candidate cost (float), probability of accepting candidate (float), and acceptance decision (boolean).
- **seed** – Optional seed value for random number generator.

Returns List of linear sequences on the chip found by simulated annealing method.

Methods

place_line(device, length)	Runs line sequence search.

cirq.AnnealSequenceSearchStrategy.place_line

AnnealSequenceSearchStrategy.**place_line**(*device: cirq.google.xmon_device.XmonDevice, length: int*) → cirq.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.

7.6.4 cirq.line_on_device

cirq.**line_on_device**(*device: cirq.google.xmon_device.XmonDevice, length: int, method: cirq.line.placement.place_strategy.LinePlacementStrategy = <cirq.line.placement.greedy.GreedySequenceSearchStrategy object>*) → cirq.line.placement.sequence.GridQubitLineTuple
Searches for linear sequence of qubits on device.

Parameters

- **device** – Google Xmon device instance.
- **length** – Desired number of qubits making up the line.
- **method** – Line placement method. Defaults to cirq.greedy.GreedySequenceSearchMethod.

Returns Line sequences search results.

7.7 Parameterization

Classes for parameterized circuits.

Symbol(name)	A constant plus the runtime value of a parameter with a given key.
ParamResolver(param_dict, float])	Resolves Symbols to actual values.
Sweep	A sweep is an iterator over ParamResolvers.
Points(key, points)	A simple sweep with explicitly supplied values.
Linspace(key, start, stop, length)	A simple sweep over linearly-spaced values.
Sweepable	Union type; Union[X, Y] means either X or Y.

7.7.1 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

—

7.7.2 cirq.ParamResolver

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.

__init__(*param_dict: Dict[str, float]*) → None

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

Methods

value_of(value, float, str])	Attempt to resolve a Symbol or name or float to its assigned value.

cirq.ParamResolver.value_of

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

Attempt 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.

7.7.3 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.

7.7.4 cirq.Points

class cirq.**Points**(*key: str, points: Sequence[float]*)
A simple sweep with explicitly supplied values.

__init__(*key: str, points: Sequence[float]*) → None
Initialize self. See help(type(self)) for accurate signature.

Methods

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

cirq.Points.param_tuples

Points.**param_tuples**() → 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.Points.keys

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

7.7.5 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

Linspace.**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.Linspace.keys

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

7.7.6 cirq.Sweepable

cirq.**Sweepable = typing.Union[cirq.study.resolver.ParamResolver, typing.Iterable[cirq.study**
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].

7.8 Optimization

Classes for compiling.

OptimizationPass	Rewrites a circuit's operations in place to make them better.
PointOptimizer	Makes circuit improvements focused on a specific location.
PointOptimizationSummary(clear_span, ...)	A description of a local optimization to perform.
ExpandComposite(composite_gate_extension, ...)	An optimization pass that expands CompositeOperation instances.
DropEmptyMoments	Removes empty moments from a circuit.
DropNegligible(tolerance, extensions)	An optimization pass that removes operations with tiny effects.

7.8.1 cirq.OptimizationPass

class cirq.**OptimizationPass**

Rewrites a circuit's operations in place to make them better.

__init__()

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

Methods

optimize_circuit(circuit)	Rewrites the given circuit to make it better.

cirq.OptimizationPass.optimize_circuit

OptimizationPass.**optimize_circuit**(*circuit: cirq.circuits.circuit.Circuit*)

Rewrites the given circuit to make it better.

Note that this performs an in place optimization.

Parameters **circuit** – The circuit to improve.

7.8.2 cirq.PointOptimizer

class cirq.**PointOptimizer**

Makes circuit improvements focused on a specific location.

__init__()

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

Methods

optimization_at(circuit, index, op)	Describes how to change operations near the given location.

Continued on next page

Table 78 – continued from previous page

optimize_circuit(circuit)	Rewrites the given circuit to make it better.

cirq.PointOptimizer.optimization_at

PointOptimizer.**optimization_at**(*circuit: cirq.circuits.circuit.Circuit*, *index: int*, *op: cirq.ops.raw_types.Operation*) → Union[cirq.circuits.optimization_pass.PointOptimizationSummary, NoneType]

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.PointOptimizer.optimize_circuit

PointOptimizer.**optimize_circuit**(*circuit: cirq.circuits.circuit.Circuit*)

Rewrites the given circuit to make it better.

Note that this performs an in place optimization.

Parameters **circuit** – The circuit to improve.

7.8.3 cirq.PointOptimizationSummary

class cirq.**PointOptimizationSummary**(*clear_span: int*, *clear_qubits: Iterable[cirq.ops.raw_types.QubitId]*, *new_operations: Union[cirq.ops.raw_types.Operation, typing.Iterable[typing.Any]]*)

A description of a local optimization to perform.

__init__(*clear_span: int*, *clear_qubits: Iterable[cirq.ops.raw_types.QubitId]*, *new_operations: Union[cirq.ops.raw_types.Operation, typing.Iterable[typing.Any]]*) → None

Parameters

- **clear_span** – Defines the range of moments to affect. Specifically, refers to the indices in range(start, start+clear_span) where start is an index known from surrounding context.
- **clear_qubits** – Defines the set of qubits that should be cleared with each affected moment.
- **new_operations** – The operations to replace the cleared out operations with.

Methods

—

7.8.4 cirq.ExpandComposite

class cirq.**ExpandComposite**(*composite_gate_extension: cirq.extension.extensions.Extensions = None, no_decomp: Callable[cirq.ops.raw_types.Operation, bool] = <function ExpandComposite.<lambda>>*)

An optimization pass that expands CompositeOperation instances.

For each operation in the circuit, this pass examines if the operation is a CompositeOperation, or is composite according to a supplied Extension, and if it is, clears the operation and replaces it with its decomposition using a fixed insertion strategy.

__init__(*composite_gate_extension: cirq.extension.extensions.Extensions = None, no_decomp: Callable[cirq.ops.raw_types.Operation, bool] = <function ExpandComposite.<lambda>>*) → None

Construct the optimization pass.

Parameters

- **composite_gate_extension** – An extension that that can be used to supply or override a CompositeOperation decomposition.
- **no_decomp** – A predicate that determines whether an operation should be decomposed or not. Defaults to decomposing everything.

Methods

optimization_at(circuit, index, op)	Describes how to change operations near the given location.
optimize_circuit(circuit)	Rewrites the given circuit to make it better.

cirq.ExpandComposite.optimization_at

ExpandComposite.**optimization_at**(*circuit*, *index*, *op*)

Describes how to change operations near the given location.

For example, this method could realize that the given operation is an X gate and that in the very next moment there is a Z gate. It would indicate that they should be combined into a Y gate by returning PointOptimizationSummary(clear_span=2,

clear_qubits=op.qubits, new_operations=cirq.Y(op.qubits[0]))

Parameters

- **circuit** – The circuit to improve.
- **index** – The index of the moment with the operation to focus on.
- **op** – The operation to focus improvements upon.

Returns A description of the optimization to perform, or else None if no change should be made.

cirq.ExpandComposite.optimize_circuit

ExpandComposite.**optimize_circuit**(*circuit: cirq.circuits.circuit.Circuit*)
Rewrites the given circuit to make it better.

Note that this performs an in place optimization.

Parameters **circuit** – The circuit to improve.

7.8.5 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)	Rewrites the given circuit to make it better.

cirq.DropEmptyMoments.optimize_circuit

DropEmptyMoments.**optimize_circuit**(*circuit: cirq.circuits.circuit.Circuit*)
Rewrites the given circuit to make it better.

Note that this performs an in place optimization.

Parameters **circuit** – The circuit to improve.

7.8.6 cirq.DropNegligible

class cirq.**DropNegligible**(*tolerance: float = 1e-08*, *extensions: cirq.extension.extensions.Extensions = None*)
An optimization pass that removes operations with tiny effects.

__init__(*tolerance: float = 1e-08*, *extensions: cirq.extension.extensions.Extensions = None*) → None
Initialize self. See help(type(self)) for accurate signature.

Methods

optimize_circuit(circuit)	Rewrites the given circuit to make it better.

cirq.DropNegligible.optimize_circuit

DropNegligible.**optimize_circuit**(*circuit: cirq.circuits.circuit.Circuit*) → None
Rewrites the given circuit to make it better.

Note that this performs an in place optimization.

Parameters **circuit** – The circuit to improve.

7.9 Implementations

Packages to use specific hardware implementations.

7.9.1 Google

Quantum hardware from Google.

Gates

google.XmonGate	A gate with a known mechanism for encoding into google API protos.
google.Exp11Gate(*, half_turns, float, ...)	A two-qubit interaction that phases the amplitude of the 11 state.
google.ExpWGate(*, axis_half_turns, float, ...)	A rotation around an axis in the XY plane of the Bloch sphere.
google.ExpZGate(*, half_turns, float, ...)	A rotation around the Z axis of the Bloch sphere.
google.XmonMeasurementGate(key, invert_mask, ...)	Indicates that qubits should be measured, and where the result goes.
google.single_qubit_matrix_to_native_gates(...)	Implements a single-qubit operation with few native gates.
google.two_qubit_matrix_to_native_gates(qc, ...)	Decomposes a two-qubit operation into Z/XY/CZ gates.
google.ConvertToXmonGates(extensions[, ...])	Attempts to convert strange gates into XmonGates.

cirq.google.XmonGate

class cirq.google.**XmonGate**

A gate with a known mechanism for encoding into google API protos.

__init__()

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

Methods

from_proto(op)	
is_xmon_op(op)	
on(*qubits)	Returns an application of this gate to the given qubits.
parameterized_value_from_proto(message)	
parameterized_value_to_proto(param, float], out)	
to_proto(*qubits)	
try_get_xmon_gate(op)	
validate_args(qubits)	Checks if this gate can be applied to the given qubits.

cirq.google.XmonGate.from_proto

static XmonGate.**from_proto**(*op: cirq.api.google.v1.operations_pb2.Operation*) → cirq.ops.raw_types.Operation

cirq.google.XmonGate.is_xmon_op

static XmonGate.**is_xmon_op**(*op: cirq.ops.raw_types.Operation*) → bool

cirq.google.XmonGate.on

XmonGate.**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.google.XmonGate.parameterized_value_from_proto

static XmonGate.**parameterized_value_from_proto**(*message: cirq.api.google.v1.operations_pb2.ParameterizedFloat*) → Union[cirq.value.symbol.Symbol, float]

cirq.google.XmonGate.parameterized_value_to_proto

static XmonGate.**parameterized_value_to_proto**(*param: Union[cirq.value.symbol.Symbol, float], out: cirq.api.google.v1.operations_pb2.ParameterizedFloat = None*) → cirq.api.google.v1.operations_pb2.ParameterizedFloat

cirq.google.XmonGate.to_proto

XmonGate.**to_proto**(**qubits*) → cirq.api.google.v1.operations_pb2.Operation

cirq.google.XmonGate.try_get_xmon_gate

static XmonGate.**try_get_xmon_gate**(*op: cirq.ops.raw_types.Operation*) → Union[_ForwardRef('XmonGate'), NoneType]

cirq.google.XmonGate.validate_args

XmonGate.**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.

cirq.google.Exp11Gate

class cirq.google.**Exp11Gate**(*, *half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *rads: Union[float, NoneType] = None*, *degs: Union[float, NoneType] = None*)

A two-qubit interaction that phases the amplitude of the 11 state.

This gate is exp(i * pi * |11><11| * half_turn).

Note that this half_turn parameter is such that a full turn is the identity matrix, in contrast to the single qubit gates, where a full turn is minus identity. The single qubit half-turn gates are defined so that a full turn corresponds to a rotation on the Bloch sphere of a 360 degree rotation. For two qubit gates, there isn't a Bloch sphere, so the half_turn corresponds to half of a full rotation in U(4).

__init__(*, *half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *rads: Union[float, NoneType] = None*, *degs: Union[float, NoneType] = None*) → None

Initializes the gate.

At most one angle argument may be specified. If more are specified, the result is considered ambiguous and an error is thrown. If no angle argument is given, the default value of one half turn is used.

Parameters

- **half_turns** – The amount of phasing of the 11 state, in half_turns.
- **rads** – The amount of phasing of the 11 state, in radians.
- **degs** – The amount of phasing of the 11 state, in degrees.

Methods

from_proto(op)	
has_matrix()	
is_parameterized()	Whether the effect is parameterized.
is_xmon_op(op)	
known_qasm_output(qubits, ...], args)	Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.
matrix()	
on(*qubits)	Returns an application of this gate to the given qubits.
parameterized_value_from_proto(message)	
parameterized_value_to_proto(param, float], out)	
phase_by(phase_turns, qubit_index)	Returns a phased version of the effect.
qubit_index_to_equivalence_group_key(index)	Returns a key that differs between non-interchangeable qubits.
text_diagram_info(args)	Describes how to draw something in a text diagram.
to_proto(*qubits)	
try_cast_to(desired_type, ext)	Turns this value into the desired type, if possible.

Continued on next page

Table 85 – continued from previous page

try_get_xmon_gate(op)	
validate_args(qubits)	Checks if this gate can be applied to the given qubits.
with_parameters_resolved_by(param_resolver)	Resolve the parameters in the effect.

cirq.google.Exp11Gate.from_proto

static Exp11Gate.**from_proto**(*op: cirq.api.google.v1.operations_pb2.Operation*) → cirq.ops.raw_types.Operation

cirq.google.Exp11Gate.has_matrix

Exp11Gate.**has_matrix**()

cirq.google.Exp11Gate.is_parameterized

Exp11Gate.**is_parameterized**() → bool

Whether the effect is parameterized.

Returns True if the gate has any unresolved Symbols and False otherwise.

cirq.google.Exp11Gate.is_xmon_op

static Exp11Gate.**is_xmon_op**(*op: cirq.ops.raw_types.Operation*) → bool

cirq.google.Exp11Gate.known_qasm_output

Exp11Gate.**known_qasm_output**(*qubits: Tuple[cirq.ops.raw_types.QubitId, ...], args: cirq.ops.gate_features.QasmOutputArgs*) → Union[str, NoneType]

Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.

cirq.google.Exp11Gate.matrix

Exp11Gate.**matrix**()

cirq.google.Exp11Gate.on

Exp11Gate.**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.google.Exp11Gate.parameterized_value_from_proto

static Exp11Gate.**parameterized_value_from_proto**(*message: cirq.api.google.v1.operations_pb2.ParameterizedFloat*) → Union[cirq.value.symbol.Symbol, float]

cirq.google.Exp11Gate.parameterized_value_to_proto

static Exp11Gate.**parameterized_value_to_proto**(*param: Union[cirq.value.symbol.Symbol, float], out: cirq.api.google.v1.operations_pb2.ParameterizedFloat = None*) → cirq.api.google.v1.operations_pb2.ParameterizedFloat

cirq.google.Exp11Gate.phase_by

Exp11Gate.**phase_by**(*phase_turns*, *qubit_index*)

Returns a phased version of the effect.

For example, an X gate phased by 90 degrees would be a Y gate.

Parameters

- **phase_turns** – The amount to phase the gate, in fractions of a whole turn.
- **qubit_index** – The index of the target qubit the phasing applies to.

Returns The phased gate or operation.

cirq.google.Exp11Gate.qubit_index_to_equivalence_group_key

Exp11Gate.**qubit_index_to_equivalence_group_key**(*index: int*) → int

Returns a key that differs between non-interchangeable qubits.

cirq.google.Exp11Gate.text_diagram_info

Exp11Gate.**text_diagram_info**(*args: cirq.ops.gate_features.TextDiagramInfoArgs*) → cirq.ops.gate_features.TextDiagramInfo

Describes how to draw something in a text diagram.

Parameters **args** – A TextDiagramInfoArgs instance encapsulating various pieces of information (e.g. how many qubits are we being applied to) as well as user options (e.g. whether to avoid unicode characters).

Returns A TextDiagramInfo instance describing what to print.

cirq.google.Exp11Gate.to_proto

Exp11Gate.**to_proto**(**qubits*)

cirq.google.Exp11Gate.try_cast_to

Exp11Gate.**try_cast_to**(*desired_type*, *ext*)

Turns this value into the desired type, if possible.

Correct implementations should delegate to super() after failing to cast, instead of returning None.

Parameters

- **desired_type** – The type of thing that the caller wants to use.
- **extensions** – The extensions instance that is asking us to try to cast ourselves into something as part of its try_cast method. If we need to recursively cast some of our fields in order to cast ourselves, this is the extensions instance we should use.

Returns

None if the receiving instance doesn't recognize or can't implement the desired type. Otherwise a value that meets the interface.

cirq.google.Exp11Gate.try_get_xmon_gate

static Exp11Gate.**try_get_xmon_gate**(*op: cirq.ops.raw_types.Operation*) → Union[_ForwardRef('XmonGate'), NoneType]

cirq.google.Exp11Gate.validate_args

Exp11Gate.**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.

cirq.google.Exp11Gate.with_parameters_resolved_by

Exp11Gate.**with_parameters_resolved_by**(*param_resolver*) → cirq.google.xmon_gates.Exp11Gate

Resolve the parameters in the effect.

Returns a gate or operation of the same type, but with all Symbols replaced with floats according to the given ParamResolver.

cirq.google.ExpWGate

class cirq.google.**ExpWGate**(*, *axis_half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *axis_rads: Union[float, NoneType] = None*, *axis_degs: Union[float, NoneType] = None*, *half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *rads: Union[float, NoneType] = None*, *degs: Union[float, NoneType] = None*)

A rotation around an axis in the XY plane of the Bloch sphere.

This gate is a "phased X rotation". Specifically: ——W(axis)^t—— = ——Z^-axis——X^t——Z^axis——

This gate is exp(-i * pi * W(axis_half_turn) * half_turn / 2) where

W(theta) = cos(pi * theta) X + sin(pi * theta) Y

or in matrix form

W(theta) = [[0, cos(pi * theta) - i sin(pi * theta)], [cos(pi * theta) + i sin(pi * theta), 0]]

Note the half_turn nomenclature here comes from viewing this as a rotation on the Bloch sphere. Two half_turns correspond to a rotation in the bloch sphere of 360 degrees. Note that this is minus identity, not just identity. Similarly the axis_half_turns refers thinking of rotating the Bloch operator, starting with the operator pointing along the X direction. An axis_half_turn of 1 corresponds to the operator pointing along the -X direction while an axis_half_turn of 0.5 correspond to an operator pointing along the Y direction.

__init__(*, *axis_half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *axis_rads: Union[float, NoneType] = None*, *axis_degs: Union[float, NoneType] = None*, *half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *rads: Union[float, NoneType] = None*, *degs: Union[float, NoneType] = None*) → None

Initializes the gate.

At most one rotation angle argument may be specified. At most one axis angle argument may be specified. If more are specified, the result is considered ambiguous and an error is thrown. If no angle argument is given, the default value of one half turn is used.

The axis angle determines the rotation axis in the XY plane, with 0 being positive-ward along X and 90 degrees being positive-ward along Y.

Parameters

- **axis_half_turns** – The axis angle in the XY plane, in half_turns.
- **axis_rads** – The axis angle in the XY plane, in radians.
- **axis_degs** – The axis angle in the XY plane, in degrees.
- **half_turns** – The amount to rotate, in half_turns.
- **rads** – The amount to rotate, in radians.
- **degs** – The amount to rotate, in degrees.

Methods

from_proto(op)	
has_inverse()	
has_matrix()	
inverse()	
is_parameterized()	Whether the effect is parameterized.
is_xmon_op(op)	
matrix()	
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.
parameterized_value_from_proto(message)	

Continued on next page

Table 86 – continued from previous page

parameterized_value_to_proto(param, float], out)	
phase_by(phase_turns, qubit_index)	Returns a phased version of the effect.
text_diagram_info(args)	Describes how to draw something in a text diagram.
to_proto(*qubits)	
trace_distance_bound()	A maximum on the trace distance between this effect's input/output.
try_cast_to(desired_type, ext)	Turns this value into the desired type, if possible.
try_get_xmon_gate(op)	
validate_args(qubits)	Checks if this gate can be applied to the given qubits.
with_parameters_resolved_by(param_resolver)	Resolve the parameters in the effect.

cirq.google.ExpWGate.from_proto

static ExpWGate.**from_proto**(*op: cirq.api.google.v1.operations_pb2.Operation*) → cirq.ops.raw_types.Operation

cirq.google.ExpWGate.has_inverse

ExpWGate.**has_inverse**()

cirq.google.ExpWGate.has_matrix

ExpWGate.**has_matrix**()

cirq.google.ExpWGate.inverse

ExpWGate.**inverse**()

cirq.google.ExpWGate.is_parameterized

ExpWGate.**is_parameterized**() → bool

Whether the effect is parameterized.

Returns True if the gate has any unresolved Symbols and False otherwise.

cirq.google.ExpWGate.is_xmon_op

static ExpWGate.**is_xmon_op**(*op: cirq.ops.raw_types.Operation*) → bool

cirq.google.ExpWGate.matrix

ExpWGate.**matrix**()

cirq.google.ExpWGate.on

ExpWGate.**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.google.ExpWGate.on_each

ExpWGate.**on_each**(*targets: Iterable[cirq.ops.raw_types.QubitId]*) → Union[cirq.ops.raw_types.Operation, typing.Iterable[typing.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.google.ExpWGate.parameterized_value_from_proto

static ExpWGate.**parameterized_value_from_proto**(*message: cirq.api.google.v1.operations_pb2.ParameterizedFloat*) → Union[cirq.value.symbol.Symbol, float]

cirq.google.ExpWGate.parameterized_value_to_proto

static ExpWGate.**parameterized_value_to_proto**(*param: Union[cirq.value.symbol.Symbol, float], out: cirq.api.google.v1.operations_pb2.ParameterizedFloat = None*) → cirq.api.google.v1.operations_pb2.ParameterizedFloat

cirq.google.ExpWGate.phase_by

ExpWGate.**phase_by**(*phase_turns*, *qubit_index*)
Returns a phased version of the effect.

For example, an X gate phased by 90 degrees would be a Y gate.

Parameters

- **phase_turns** – The amount to phase the gate, in fractions of a whole turn.
- **qubit_index** – The index of the target qubit the phasing applies to.

Returns The phased gate or operation.

cirq.google.ExpWGate.text_diagram_info

ExpWGate.**text_diagram_info**(*args: cirq.ops.gate_features.TextDiagramInfoArgs*) → cirq.ops.gate_features.TextDiagramInfo
Describes how to draw something in a text diagram.

Parameters **args** – A TextDiagramInfoArgs instance encapsulating various pieces of information (e.g. how many qubits are we being applied to) as well as user options (e.g. whether to avoid unicode characters).

Returns A TextDiagramInfo instance describing what to print.

cirq.google.ExpWGate.to_proto

ExpWGate.**to_proto**(**qubits*)

cirq.google.ExpWGate.trace_distance_bound

ExpWGate.**trace_distance_bound**()

A maximum on the trace distance between this effect's input/output.

Generally this method is used when deciding whether to keep gates, so only the behavior near 0 is important. Approximations that overestimate the maximum trace distance are permitted. Even ones that exceed 1. Underestimates are not permitted.

cirq.google.ExpWGate.try_cast_to

ExpWGate.**try_cast_to**(*desired_type*, *ext*)

Turns this value into the desired type, if possible.

Correct implementations should delegate to super() after failing to cast, instead of returning None.

Parameters

- **desired_type** – The type of thing that the caller wants to use.
- **extensions** – The extensions instance that is asking us to try to cast ourselves into something as part of its try_cast method. If we need to recursively cast some of our fields in order to cast ourselves, this is the extensions instance we should use.

Returns

None if the receiving instance doesn't recognize or can't implement the desired type. Otherwise a value that meets the interface.

cirq.google.ExpWGate.try_get_xmon_gate

static ExpWGate.**try_get_xmon_gate**(*op: cirq.ops.raw_types.Operation*) → Union[_ForwardRef('XmonGate'), NoneType]

cirq.google.ExpWGate.validate_args

ExpWGate.**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.google.ExpWGate.with_parameters_resolved_by

ExpWGate.**with_parameters_resolved_by**(*param_resolver*) → cirq.google.xmon_gates.ExpWGate

Resolve the parameters in the effect.

Returns a gate or operation of the same type, but with all Symbols replaced with floats according to the given ParamResolver.

cirq.google.ExpZGate

class cirq.google.**ExpZGate**(*, *half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *rads: Union[float, NoneType] = None*, *degs: Union[float, NoneType] = None*)

A rotation around the Z axis of the Bloch sphere.

This gate is exp(-i * pi * Z * half_turns / 2) where Z is the Z matrix

Z = [[1, 0], [0, -1]]

Note the half_turn nomenclature here comes from viewing this as a rotation on the Bloch sphere. Two half_turns correspond to a rotation in the bloch sphere of 360 degrees.

__init__(*, *half_turns: Union[cirq.value.symbol.Symbol, float, NoneType] = None*, *rads: Union[float, NoneType] = None*, *degs: Union[float, NoneType] = None*) → None

Initializes the gate.

At most one angle argument may be specified. If more are specified, the result is considered ambiguous and an error is thrown. If no angle argument is given, the default value of one half turn is used.

Parameters

- **half_turns** – The relative phasing of Z's eigenstates, in half_turns.
- **rads** – The relative phasing of Z's eigenstates, in radians.
- **degs** – The relative phasing of Z's eigenstates, in degrees.

Methods

from_proto(op)	
has_inverse()	
has_matrix()	
inverse()	
is_parameterized()	Whether the effect is parameterized.
is_xmon_op(op)	
known_qasm_output(qubits, ...], args)	Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.
matrix()	
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.
parameterized_value_from_proto(message)	

Continued on next page

Table 87 – continued from previous page

parameterized_value_to_proto(param, float], out)	
phase_by(phase_turns, qubit_index)	Returns a phased version of the effect.
text_diagram_info(args)	Describes how to draw something in a text diagram.
to_proto(*qubits)	
trace_distance_bound()	A maximum on the trace distance between this effect's input/output.
try_cast_to(desired_type, ext)	Turns this value into the desired type, if possible.
try_get_xmon_gate(op)	
validate_args(qubits)	Checks if this gate can be applied to the given qubits.
with_parameters_resolved_by(param_resolver)	Resolve the parameters in the effect.

cirq.google.ExpZGate.from_proto

static ExpZGate.**from_proto**(*op: cirq.api.google.v1.operations_pb2.Operation*) → cirq.ops.raw_types.Operation

cirq.google.ExpZGate.has_inverse

ExpZGate.**has_inverse**()

cirq.google.ExpZGate.has_matrix

ExpZGate.**has_matrix**()

cirq.google.ExpZGate.inverse

ExpZGate.**inverse**()

cirq.google.ExpZGate.is_parameterized

ExpZGate.**is_parameterized**() → bool

Whether the effect is parameterized.

Returns True if the gate has any unresolved Symbols and False otherwise.

cirq.google.ExpZGate.is_xmon_op

static ExpZGate.**is_xmon_op**(*op: cirq.ops.raw_types.Operation*) → bool

cirq.google.ExpZGate.known_qasm_output

ExpZGate.**known_qasm_output**(*qubits: Tuple[cirq.ops.raw_types.QubitId, ...], args: cirq.ops.gate_features.QasmOutputArgs*) → Union[str, NoneType]

Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.

cirq.google.ExpZGate.matrix

ExpZGate.**matrix**()

cirq.google.ExpZGate.on

ExpZGate.**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.google.ExpZGate.on_each

ExpZGate.**on_each**(*targets: Iterable[cirq.ops.raw_types.QubitId]*) → Union[cirq.ops.raw_types.Operation, typing.Iterable[typing.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.google.ExpZGate.parameterized_value_from_proto

static ExpZGate.**parameterized_value_from_proto**(*message: cirq.api.google.v1.operations_pb2.ParameterizedFloat*) → Union[cirq.value.symbol.Symbol, float]

cirq.google.ExpZGate.parameterized_value_to_proto

static ExpZGate.**parameterized_value_to_proto**(*param: Union[cirq.value.symbol.Symbol, float], out: cirq.api.google.v1.operations_pb2.ParameterizedFloat = None*) → cirq.api.google.v1.operations_pb2.ParameterizedFloat

cirq.google.ExpZGate.phase_by

ExpZGate.**phase_by**(*phase_turns: float*, *qubit_index: int*)
Returns a phased version of the effect.

For example, an X gate phased by 90 degrees would be a Y gate.

Parameters

- **phase_turns** – The amount to phase the gate, in fractions of a whole turn.
- **qubit_index** – The index of the target qubit the phasing applies to.

Returns The phased gate or operation.

cirq.google.ExpZGate.text_diagram_info

ExpZGate.**text_diagram_info**(*args: cirq.ops.gate_features.TextDiagramInfoArgs*) → cirq.ops.gate_features.TextDiagramInfo

Describes how to draw something in a text diagram.

Parameters **args** – A TextDiagramInfoArgs instance encapsulating various pieces of information (e.g. how many qubits are we being applied to) as well as user options (e.g. whether to avoid unicode characters).

Returns A TextDiagramInfo instance describing what to print.

cirq.google.ExpZGate.to_proto

ExpZGate.**to_proto**(**qubits*)

cirq.google.ExpZGate.trace_distance_bound

ExpZGate.**trace_distance_bound**()

A maximum on the trace distance between this effect's input/output.

Generally this method is used when deciding whether to keep gates, so only the behavior near 0 is important. Approximations that overestimate the maximum trace distance are permitted. Even ones that exceed 1. Underestimates are not permitted.

cirq.google.ExpZGate.try_cast_to

ExpZGate.**try_cast_to**(*desired_type*, *ext*)

Turns this value into the desired type, if possible.

Correct implementations should delegate to super() after failing to cast, instead of returning None.

Parameters

- **desired_type** – The type of thing that the caller wants to use.
- **extensions** – The extensions instance that is asking us to try to cast ourselves into something as part of its try_cast method. If we need to recursively cast some of our fields in order to cast ourselves, this is the extensions instance we should use.

Returns

None if the receiving instance doesn't recognize or can't implement the desired type. Otherwise a value that meets the interface.

cirq.google.ExpZGate.try_get_xmon_gate

static ExpZGate.**try_get_xmon_gate**(*op: cirq.ops.raw_types.Operation*) → Union[_ForwardRef('XmonGate'), NoneType]

cirq.google.ExpZGate.validate_args

ExpZGate.**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.google.ExpZGate.with_parameters_resolved_by

ExpZGate.**with_parameters_resolved_by**(*param_resolver*) → cirq.google.xmon_gates.ExpZGate
Resolve the parameters in the effect.

Returns a gate or operation of the same type, but with all Symbols replaced with floats according to the given ParamResolver.

cirq.google.XmonMeasurementGate

class cirq.google.**XmonMeasurementGate**(*key: str = ''*, *invert_mask: Tuple[bool, ...] = ()*)
Indicates that qubits should be measured, and where the result goes.

This measurement is done in the computational basis.

__init__(*key: str = ''*, *invert_mask: Tuple[bool, ...] = ()*) → None
Initialize self. See help(type(self)) for accurate signature.

Methods

from_proto(op)	
is_measurement(op, cirq.ops.raw_types.Operation])	
is_xmon_op(op)	
known_qasm_output(qubits, ...], args)	Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.
on(*qubits)	Returns an application of this gate to the given qubits.
parameterized_value_from_proto(message)	
parameterized_value_to_proto(param, float], out)	
text_diagram_info(args)	Describes how to draw something in a text diagram.
to_proto(*qubits)	
try_get_xmon_gate(op)	
validate_args(qubits)	Checks if this gate can be applied to the given qubits.
with_bits_flipped(*bit_positions)	Toggles whether or not the measurement inverts various outputs.

cirq.google.XmonMeasurementGate.from_proto

static XmonMeasurementGate.**from_proto**(*op: cirq.api.google.v1.operations_pb2.Operation*) → cirq.ops.raw_types.Operation

cirq.google.XmonMeasurementGate.is_measurement

static XmonMeasurementGate.**is_measurement**(*op: Union[cirq.ops.raw_types.Gate, cirq.ops.raw_types.Operation]*) → bool

cirq.google.XmonMeasurementGate.is_xmon_op

static XmonMeasurementGate.**is_xmon_op**(*op: cirq.ops.raw_types.Operation*) → bool

cirq.google.XmonMeasurementGate.known_qasm_output

XmonMeasurementGate.**known_qasm_output**(*qubits: Tuple[cirq.ops.raw_types.QubitId, ...], args: cirq.ops.gate_features.QasmOutputArgs*) → Union[str, NoneType]

Returns lines of QASM output representing the gate on the given qubits or None if a simple conversion is not possible.

cirq.google.XmonMeasurementGate.on

XmonMeasurementGate.**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.google.XmonMeasurementGate.parameterized_value_from_proto

static XmonMeasurementGate.**parameterized_value_from_proto**(*message: cirq.api.google.v1.operations_pb2.Parame* → Union[cirq.value.symbol.Symbol, float]

cirq.google.XmonMeasurementGate.parameterized_value_to_proto

static XmonMeasurementGate.**parameterized_value_to_proto**(*param: Union[cirq.value.symbol.Symbol, float], out: cirq.api.google.v1.operations_pb2.Parameter = None*) → cirq.api.google.v1.operations_pb2.Parameteri

cirq.google.XmonMeasurementGate.text_diagram_info

XmonMeasurementGate.**text_diagram_info**(*args: cirq.ops.gate_features.TextDiagramInfoArgs*) → cirq.ops.gate_features.TextDiagramInfo

Describes how to draw something in a text diagram.

Parameters **args** – A TextDiagramInfoArgs instance encapsulating various pieces of information (e.g. how many qubits are we being applied to) as well as user options (e.g. whether to avoid unicode characters).

Returns A TextDiagramInfo instance describing what to print.

cirq.google.XmonMeasurementGate.to_proto

XmonMeasurementGate.**to_proto**(**qubits*)

cirq.google.XmonMeasurementGate.try_get_xmon_gate

static XmonMeasurementGate.**try_get_xmon_gate**(*op: cirq.ops.raw_types.Operation*) → Union[_ForwardRef('XmonGate'), NoneType]

cirq.google.XmonMeasurementGate.validate_args

XmonMeasurementGate.**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.google.XmonMeasurementGate.with_bits_flipped

XmonMeasurementGate.**with_bits_flipped**(**bit_positions*) → cirq.google.xmon_gates.XmonMeasurementGate

Toggles whether or not the measurement inverts various outputs.

cirq.google.single_qubit_matrix_to_native_gates

cirq.google.**single_qubit_matrix_to_native_gates**(*mat: numpy.ndarray*, *tolerance: float = 0*) → List[cirq.ops.gate_features.SingleQubitGate]

Implements a single-qubit operation with few native 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.google.two_qubit_matrix_to_native_gates

cirq.google.**two_qubit_matrix_to_native_gates**(*q0: cirq.ops.raw_types.QubitId*, *q1: cirq.ops.raw_types.QubitId*, *mat: numpy.ndarray*, *allow_partial_czs: bool*, *tolerance: float = 1e-08*) → 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.

Returns A list of operations implementing the matrix.

cirq.google.ConvertToXmonGates

class cirq.google.**ConvertToXmonGates**(*extensions: cirq.extension.extensions.Extensions = None*, *ignore_failures=False*)

Attempts to convert strange gates into XmonGates.

First, checks if the given extensions are able to cast the gate into an XmonGate instance.

Second, checks if the given extensions are able to cast the operation into a KnownMatrix. If so, and the gate is a 1-qubit or 2-qubit gate, then performs circuit synthesis of the operation.

Third, checks if the given extensions are able to cast the operation into a CompositeOperation. If so, recurses on the decomposition.

Fourth, if ignore_failures is set, gives up and returns the gate unchanged. Otherwise raises a TypeError.

__init__(*extensions: cirq.extension.extensions.Extensions = None*, *ignore_failures=False*) → None

Parameters

- **extensions** – The extensions instance to use when trying to cast gates to known types. Defaults to the standard xmon gate extension.
- **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)	Rewrites the given circuit to make it better.

cirq.google.ConvertToXmonGates.convert

ConvertToXmonGates.**convert**(*op: cirq.ops.raw_types.Operation*) → Union[cirq.ops.raw_types.Operation, typing.Iterable[typing.Any]]

cirq.google.ConvertToXmonGates.optimization_at

ConvertToXmonGates.**optimization_at**(*circuit*, *index*, *op*)

Describes how to change operations near the given location.

For example, this method could realize that the given operation is an X gate and that in the very next moment there is a Z gate. It would indicate that they should be combined into a Y gate by returning PointOptimizationSummary(clear_span=2,

clear_qubits=op.qubits, new_operations=cirq.Y(op.qubits[0]))

Parameters

- **circuit** – The circuit to improve.
- **index** – The index of the moment with the operation to focus on.
- **op** – The operation to focus improvements upon.

Returns A description of the optimization to perform, or else None if no change should be made.

cirq.google.ConvertToXmonGates.optimize_circuit

ConvertToXmonGates.**optimize_circuit**(*circuit: cirq.circuits.circuit.Circuit*)

Rewrites the given circuit to make it better.

Note that this performs an in place optimization.

Parameters **circuit** – The circuit to improve.

Devices

google.Bristlecone	A device with qubits placed in a grid.
google.Foxtail	A device with qubits placed in a grid.
google.XmonDevice(measurement_duration, ...)	A device with qubits placed in a grid.

cirq.google.Bristlecone

cirq.google.**Bristlecone = <cirq.google.xmon_device.XmonDevice object>**

A device with qubits placed in a grid. Neighboring qubits can interact.

cirq.google.Foxtail

cirq.google.**Foxtail = <cirq.google.xmon_device.XmonDevice object>**

A device with qubits placed in a grid. Neighboring qubits can interact.

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)	
neighbors_of(qubit)	Returns the qubits that the given qubit can interact with.
row(row)	Returns the qubits in the given row, in ascending order.
validate_circuit(circuit)	Raises an exception if a circuit is not valid.
validate_gate(gate)	Raises an error if the given gate isn't allowed.
validate_moment(moment)	Raises an exception if a moment is not valid.
validate_operation(operation)	Raises an exception if an operation is not valid.
validate_schedule(schedule)	Raises an exception if a schedule is not valid.
validate_scheduled_operation(schedule, ...)	Raises an exception if the scheduled operation is not valid.

cirq.google.XmonDevice.at

XmonDevice.**at**(*row: int*, *col: int*) → Union[cirq.devices.grid_qubit.GridQubit, NoneType]

Returns the qubit at the given position, if there is one, else None.

cirq.google.XmonDevice.can_add_operation_into_moment

XmonDevice.**can_add_operation_into_moment**(*operation: cirq.ops.raw_types.Operation*, *moment: cirq.circuits.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

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

XmonDevice.**decompose_operation**(*operation: cirq.ops.raw_types.Operation*) → Union[cirq.ops.raw_types.Operation, typing.Iterable[typing.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 instead of ExpWGate.

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*) → List[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.circuits.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.

Simulator

google.XmonOptions(num_shards, ...)	XmonOptions for the XmonSimulator.
google.XmonSimulator(options)	XmonSimulator for Xmon class quantum circuits.
google.XmonStepResult(stepper, qubit_map, ...)	Results of a step of the simulator.
google.XmonSimulateTrialResult(params, ...)	Results of a simulation of the XmonSimulator.

cirq.google.XmonOptions

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 raised to this value 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.

__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 provided:

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 provided:

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

run(circuit, param_resolver, repetitions, ...)	Runs the entire supplied Circuit, mimicking the quantum hardware.
run_sweep(program, ...)	Runs the entire supplied Circuit, mimicking the quantum hardware.
simulate(circuit, param_resolver, ...)	Simulates the entire supplied Circuit.
simulate_moment_steps(program, options, ...)	Returns an iterator of XmonStepResults for each moment simulated.
simulate_sweep(program, ...)	Simulates the entire supplied Circuit.

cirq.google.XmonSimulator.run

XmonSimulator.**run**(*circuit: cirq.circuits.circuit.Circuit, param_resolver: cirq.study.resolver.ParamResolver = ParamResolver({}), repetitions: int = 1, qubit_order: Union[cirq.ops.qubit_order.QubitOrder, typing.Iterable[cirq.ops.raw_types.QubitId]] = <cirq.ops.qubit_order.QubitOrder object>, extensions: cirq.extension.extensions.Extensions = None*) → cirq.study.trial_result.TrialResult
Runs the entire supplied Circuit, mimicking the quantum hardware.

If one wants access to the wave function (both setting and getting), the "simulate" methods should be used.

The initial state of the run methods is the all zeros state in the computational basis.

Parameters

- **circuit** – The circuit to simulate.
- **param_resolver** – Parameters to run with the program.
- **repetitions** – The number of repetitions to simulate.
- **qubit_order** – Determines the canonical ordering of the qubits used to define the order of amplitudes in the wave function.
- **extensions** – Extensions that will be applied while trying to decompose the circuit's gates into XmonGates. If None, this uses the default of xmon_gate_ext.

Returns TrialResult for a run.

cirq.google.XmonSimulator.run_sweep

XmonSimulator.**run_sweep**(*program: Union[cirq.circuits.circuit.Circuit, cirq.schedules.schedule.Schedule], params: Union[cirq.study.resolver.ParamResolver, typing.Iterable[cirq.study.resolver.ParamResolver], cirq.study.sweeps.Sweep, typing.Iterable[cirq.study.sweeps.Sweep]] = ParamResolver({}), repetitions: int = 1, qubit_order: Union[cirq.ops.qubit_order.QubitOrder, typing.Iterable[cirq.ops.raw_types.QubitId]] = <cirq.ops.qubit_order.QubitOrder object>, extensions: cirq.extension.extensions.Extensions = None*) → List[cirq.study.trial_result.TrialResult]

Runs the entire supplied Circuit, mimicking the quantum hardware.

If one wants access to the wave function (both setting and getting), the "simulate" methods should be used.

The initial state of the run methods is the all zeros state in the computational basis.

Parameters

- **program** – The circuit or schedule to simulate.
- **params** – Parameters to run with the program.
- **repetitions** – The number of repetitions to simulate.
- **qubit_order** – Determines the canonical ordering of the qubits used to define the order of amplitudes in the wave function.
- **extensions** – Extensions that will be applied while trying to decompose the circuit's gates into XmonGates. If None, this uses the default of xmon_gate_ext.

Returns TrialResult list for this run; one for each possible parameter resolver.

cirq.google.XmonSimulator.simulate

XmonSimulator.**simulate**(*circuit: cirq.circuits.circuit.Circuit, param_resolver: cirq.study.resolver.ParamResolver = ParamResolver({}), qubit_order: Union[cirq.ops.qubit_order.QubitOrder, typing.Iterable[cirq.ops.raw_types.QubitId]] = <cirq.ops.qubit_order.QubitOrder object>, initial_state: Union[int, numpy.ndarray] = 0, extensions: cirq.extension.extensions.Extensions = None*) → cirq.google.sim.xmon_simulator.XmonSimulateTrialResult

Simulates the entire supplied Circuit.

This method returns the final wave function.

Parameters

- **circuit** – The circuit 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 a np.complex64.

- **extensions** – Extensions that will be applied while trying to decompose the circuit's gates into XmonGates. If None, this uses the default of xmon_gate_ext.

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

cirq.google.XmonSimulator.simulate_moment_steps

XmonSimulator.**simulate_moment_steps**(*program: cirq.circuits.circuit.Circuit*, *options: Union[cirq.google.sim.xmon_simulator.XmonOptions, NoneType] = None*, *qubit_order: Union[cirq.ops.qubit_order.QubitOrder, typing.Iterable[cirq.ops.raw_types.QubitId]] = <cirq.ops.qubit_order.QubitOrder object>*, *initial_state: Union[int, numpy.ndarray] = 0*, *param_resolver: cirq.study.resolver.ParamResolver = None*, *extensions: cirq.extension.extensions.Extensions = None*) → Iterator[_ForwardRef('XmonStepResult')]

Returns an iterator of XmonStepResults for each moment simulated.

Parameters

- **program** – The Circuit to simulate.
- **options** – XmonOptions configuring the simulation.
- **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 a np.complex64.
- **param_resolver** – A ParamResolver for determining values of Symbols.
- **extensions** – Extensions that will be applied while trying to decompose the circuit's gates into XmonGates. If None, this uses the default of xmon_gate_ext.

Returns SimulatorIterator that steps through the simulation, simulating each moment and returning a XmonStepResult for each moment.

cirq.google.XmonSimulator.simulate_sweep

XmonSimulator.**simulate_sweep**(*program: Union[cirq.circuits.circuit.Circuit, cirq.schedules.schedule.Schedule], params: Union[cirq.study.resolver.ParamResolver, typing.Iterable[cirq.study.resolver.ParamResolver], cirq.study.sweeps.Sweep, typing.Iterable[cirq.study.sweeps.Sweep]] = ParamResolver({}), qubit_order: Union[cirq.ops.qubit_order.QubitOrder, typing.Iterable[cirq.ops.raw_types.QubitId]] = <cirq.ops.qubit_order.QubitOrder object>, initial_state: Union[int, numpy.ndarray] = 0, extensions: cirq.extension.extensions.Extensions = None*) → List[cirq.google.sim.xmon_simulator.XmonSimulateTrialResult]

Simulates the entire supplied Circuit.

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 a np.complex64.
- **extensions** – Extensions that will be applied while trying to decompose the circuit's gates into XmonGates. If None, this uses the default of xmon_gate_ext.

Returns List of XmonSimulatorTrialResults 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, List[bool]]*)

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() 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, List[bool]]*) → None

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

Methods

sample(qubits, repetitions)	Samples from the wave function at this point in the computation.
set_state(state, numpy.ndarray])	Updates the state of the simulator to the given new state.
state()	Return the state (wave function) at this point in the computation.

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> state. The outer list is for repetitions, and the inner corresponds to measurements ordered by the supplied qubits.

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.

cirq.google.XmonStepResult.state

XmonStepResult.**state**() → numpy.ndarray

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

qubit_map: {QubitA: 0, QubitB: 1, QubitC: 2} Then the returned vector will have indices mapped to qubit basis states like the following table

| QubitA | QubitB | QubitC |

0	1	2	3	4	5	6	7
0	0	0	0	1	1	1	1
0	0	1	1	0	0	1	1
0	1	0	1	0	1	0	1

cirq.google.XmonSimulateTrialResult

class cirq.google.**XmonSimulateTrialResult**(*params: cirq.study.resolver.ParamResolver, measurements: Dict[str, numpy.ndarray], final_state: numpy.ndarray*)

Results of a simulation of the XmonSimulator.

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.

__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

—

Optimizers

google.optimized_for_xmon(circuit, ...)	Optimizes a circuit with XmonDevice in mind.
google.EjectZ(tolerance, ext)	Pushes Z gates towards the end of the circuit.
google.EjectFullW(tolerance, ext)	Pushes ExpW gates with half_turns=1 towards the end of the circuit.

cirq.google.optimized_for_xmon

cirq.google.**optimized_for_xmon**(*circuit: cirq.circuits.circuit.Circuit, new_device: Union[cirq.google.xmon_device.XmonDevice, NoneType] = None, qubit_map: Callable[cirq.ops.raw_types.QubitId, cirq.devices.grid_qubit.GridQubit] = <function <lambda>>*) → cirq.circuits.circuit.Circuit

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).

Returns The optimized circuit.

cirq.google.EjectZ

class cirq.google.**EjectZ**(*tolerance: float = 0.0*, *ext: cirq.extension.extensions.Extensions = None*)

Pushes Z gates towards the end of the circuit.

As the Z gates get pushed they may absorb other Z gates, get absorbed into measurements, cross CZ gates, cross W gates (by phasing them), etc.

__init__(*tolerance: float = 0.0*, *ext: cirq.extension.extensions.Extensions = None*) → 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.
- **ext** – Extensions object used for determining if gates are phaseable (i.e. if Z gates can pass through them).

Methods

optimize_circuit(circuit)	Rewrites the given circuit to make it better.

cirq.google.EjectZ.optimize_circuit

EjectZ.**optimize_circuit**(*circuit: cirq.circuits.circuit.Circuit*)

Rewrites the given circuit to make it better.

Note that this performs an in place optimization.

Parameters **circuit** – The circuit to improve.

cirq.google.EjectFullW

class cirq.google.**EjectFullW**(*tolerance: float = 1e-08*, *ext: cirq.extension.extensions.Extensions = None*)

Pushes ExpW gates with half_turns=1 towards the end of the circuit.

As the gates get pushed, they may absorb Z gates, cancel against other ExpW gates with half_turns=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*, *ext: cirq.extension.extensions.Extensions = None*) → 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.
- **ext** – Extensions object used for determining if gates are phaseable (i.e. if Z gates can pass through them).

Methods

optimize_circuit(circuit)	Rewrites the given circuit to make it better.

cirq.google.EjectFullW.optimize_circuit

EjectFullW.**optimize_circuit**(*circuit: cirq.circuits.circuit.Circuit*)

Rewrites the given circuit to make it better.

Note that this performs an in place optimization.

Parameters **circuit** – The circuit to improve.

CHAPTER 8

Advanced topics

8.1 Optimization Passes

TODO

8.2 Extensions

The extension mechanism in cirq is designed to solve a specific kind of compatibility problem, which is best explained by an example.

Suppose library A defines a kind of thing called a "ReactiveIterable" and provides many utility methods for reactive iterables. Separately, library B has implemented something called an "EventTrampoline". It just so happens that an "EventTrampoline" is a "ReactiveIterable" in all but name. If there was just some way to translate what library B provides into what library A wants, then one could apply all the great utility methods from library A on the value from library B. Alas, there isn't.

The goal of the extension mechanism is to perform the translatation between what B provides and what A wants. Essentially, when library A was written, instead of having code like this:

```
def some_method_in_A(reactive_iterable):
    ...
```

It would have code like this:

```
def some_method_in_A(value, extensions):
    reactive_iterable = extensions.cast(value, ReactiveIterable)
    ...
```

And then, when you wanted library A to understand a type from library B, you would invoke the method like this:

```
def your_code():
    ext = cirq.Extensions()
```

(continues on next page)

(continued from previous page)

```
    ext.add_cast(desired_type=ReactiveIterable,
                 actual_type=EventTrampoline,
                 conversion=lambda trampoline: ...)

    some_method(trampoline, ext)
```

And EventTrampolines would be automatically converted into ReactiveIterables even though the people writing libraries A and B never knew about the other library.

8.2.1 PotentialImplementation

An important part of the extension mechanism is the ability for classes to say *at runtime* whether or not they support a specific kind of functionality. Classes that implement `PotentialImplementation` have a `try_cast(desired_type, extensions)` method that they use to tell a caller whether or not they support some desired functionality.

For example, the `RotZGate` doesn't have a well-defined matrix when the amount of rotation is a symbol instead of a number. So it has a `try_cast` method that, when `desired_type` is set to `KnownMatrixgate`, returns `None` when the rotation angle is a symbol. Otherwise it returns itself (indicating that its `matrix` method will not raise an error when called).

The `extensions` parameter given to `try_cast` is useful functionality in the callee depends on functionality being present in a dependency. For example, `ControlledGate` only has a matrix if the gate it is controlling has a matrix.

CHAPTER 9

Reference

- genindex
- modindex
- search

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z