CirqProjectQ Documentation

Release 0

SZ

Nov 23, 2018

Contents:

CHAPTER 1

Reference

This package provdes a port between Cirq and ProejctQ.

Cirq is an open source quantum computing framework developed by Google and Projectq is a framework to compile and simulate quantum algorithms.

Cirqprojectq provides two main functionalities: Firstly, it provides methods and classes that can port algorithms from projectq to cirq and secondly it introduces Xmon gates and xmon gate compositions to projectq.

xmon_gates	This module provides Xmon gates for projectq.
xmon_decompositions	Provides decompositon rules to decompose common gates into Xmon gates.
xmon_rules	This module provides translation rules from Xmon gates in Projectq to Xmon gates in Cirq.
xmon_setup	Provides proejctq engines for simulation of xmon devices.
circ_engine	Provides a projectq engine that translates a projectq circuit to a cirq circuit.

CHAPTER 2

Examples

2.1 Single impurity Anderson model

The single impurity Anderson model is an important model in condensed matter physics which explains phenomena like charge transport properties of disordered materials. In the following example we demonstrate how Cirq can be used to simulate time evolution of the Anderson model on a gate based quantum computer.

```
# Copyright 2018 Heisenberg Quantum Simulations
# -*- coding: utf-8 -*-
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
from numbers import Number
import numpy as np
import cirq
from cirq.google import xmon_gates, xmon_qubit
from cirq.contrib.qcircuit_diagram import circuit_to_latex_using_qcircuit

def nearest_neighbor_hopping(amplitude, qubits):
    """
    Implement a nearest neighbor hopping step between two electrons using xmon
    gates only.
    """
    assert len(qubits)==2
    q0 = qubits[0]
```

(continues on next page)

(continued from previous page)

```
    q1 = qubits[1]
    amplitude /= np.pi
    yield xmon_gates.ExpWGate(half_turns=.5, axis_half_turns=0)(q0)
    yield xmon_gates.ExpWGate(half_turns=-.5, axis_half_turns=.5)(q1)
    yield xmon_gates.ExpZGate(half_turns=1)(q1)
    yield xmon_gates.Exp11Gate(half_turns=1.0)(q0, q1)
    yield xmon_gates.ExpWGate(half_turns=amplitude, axis_half_turns=0)(q0)
    yield xmon_gates.ExpWGate(half_turns=-amplitude, axis_half_turns=.5)(q1)
    yield xmon_gates.Exp11Gate(half_turns=1.0)(q0, q1)
    yield xmon_gates.ExpWGate(half_turns=-.5, axis_half_turns=0)(q0)
    yield xmon_gates.ExpWGate(half_turns=-.5, axis_half_turns=.5)(q1)
    yield xmon_gates.ExpZGate(half_turns=1)(q1)

def zz_interaction(amplitude, qubits):
    """
    Implement a density-density interaction between two fermionic orbitals
    using xmon gates only.
    """
    assert len(qubits)==2
    if isinstance(amplitude, Number):
        amplitude /= - np.pi
    q0 = qubits[0]
    q1 = qubits[1]
    yield xmon_gates.Exp11Gate(half_turns=amplitude)(q0, q1)

def onsite(amplitude, qubit):
    """
    Implement a local (onsite) fermionic term using xmon gates only.
    """
    assert len(qubit)==1
    yield xmon_gates.ExpZGate(half_turns = amplitude / np.pi)(qubit)

def trotter_step(t, U, qubits, impsite=None, order=2):
    """
    Implement a single Trotter step for an Anderson model with interaction
    strength U, and hopping amplitude t, and impurity location imploc.
    Trotter order can be 1 or 2.
    """
    sites = len(qubits)//2
    assert order in (1, 2)
    impsite = impsite or len(qubits)//4
    for i in range(sites - 1):
        for o in nearest_neighbor_hopping(t/order, (qubits[i], qubits[i+1])):
            yield o
        for o in nearest_neighbor_hopping(t/order, (qubits[i+sites],␣
↪qubits[i+1+sites])):
            yield o
    for o in zz_interaction(U, (qubits[impsite], qubits[impsite + sites])):
        yield o
    if order==2:
        for i in reversed(range(sites - 1)):
            for o in nearest_neighbor_hopping(t/2, (qubits[i+sites],␣
↪qubits[i+1+sites])):
                yield o
            for o in nearest_neighbor_hopping(t/2, (qubits[i], qubits[i+1])):
                yield o
```

(continues on next page)

(continued from previous page)

```
from cirq import Symbol
sites = 3
qubits = [xmon_qubit.XmonQubit(0, i) for i in range(sites)]\
            + [xmon_qubit.XmonQubit(1, i) for i in range(sites)]

def scale_H(amplitudes, allowed=np.pi):
    m = max(np.abs(amplitudes))
    return [a * allowed / m for a in amplitudes]

t = - 0.8 * np.pi
U = 0.5 * np.pi
#t, U = scale_H([t, U])

circuit = cirq.Circuit()
circuit.append(nearest_neighbor_hopping(t, (qubits[0], qubits[1])))
print(circuit)
with open('hopping.tex', 'w') as fl:
    print("\\documentclass{standalone}\n\\usepackage{amsmath}\n\\usepackage{qcircuit}
↪\n\\begin{document}\n$$", file=fl)
    print(circuit_to_latex_using_qcircuit(circuit), file=fl)
    print("$$\n\\end{document}", file=fl)


circuit = cirq.Circuit()
circuit.append(zz_interaction(U, (qubits[0], qubits[1])))
print(circuit)
with open('interaction.tex', 'w') as fl:
    print("\\documentclass{standalone}\n\\usepackage{amsmath}\n\\usepackage{qcircuit}
↪\n\\begin{document}\n$$", file=fl)
    print(circuit_to_latex_using_qcircuit(circuit), file=fl)
    print("$$\n\\end{document}", file=fl)
#
circuit = cirq.Circuit()
circuit.append(trotter_step(t, U, qubits, order=1), strategy=cirq.circuits.
↪InsertStrategy.EARLIEST)
print(circuit)
with open('trotter.tex', 'w') as fl:
    print("\\documentclass{standalone}\n\\usepackage{amsmath}\n\\usepackage{qcircuit}
↪\n\\begin{document}\n$$", file=fl)
    print(circuit_to_latex_using_qcircuit(circuit), file=fl)
    print("$$\n\\end{document}", file=fl)

from cirq.google import XmonSimulator
from openfermion.ops import FermionOperator
from openfermion.transforms import get_sparse_operator
from scipy import sparse
from itertools import product
from matplotlib import pyplot as plt

t = - 0.3 * np.pi
U = 0.6 * np.pi
t, U = scale_H([t, U])
Steps = list(range(1,16))
res = {1: [], 2: []}
for order, steps in product((1, 2), Steps):
    circuit = cirq.Circuit()
    init = []
```

(continues on next page)

(continued from previous page)

```
    for i in range(sites//2+sites%2):
        init.append(xmon_gates.ExpWGate(half_turns=1.0, axis_half_turns=0.
↪0)(qubits[i]))
    for i in range(sites//2, sites):
        init.append(xmon_gates.ExpWGate(half_turns=1.0, axis_half_turns=0.0)(qubits[i␣
↪+ sites]))
    circuit.append(init, strategy=cirq.circuits.InsertStrategy.EARLIEST)
    for j in range(steps):
        circuit.append(trotter_step(t/steps, U/steps, qubits, order=order),␣
↪strategy=cirq.circuits.InsertStrategy.EARLIEST)
    simulator = XmonSimulator()
    result = simulator.simulate(circuit)

    h = np.sum([FermionOperator(((i, 1), (i+1, 0)), t) +
                FermionOperator(((i+1, 1), (i, 0)), t) +
                FermionOperator(((i+sites, 1), (i+1+sites, 0)), t) +
                FermionOperator(((i+1+sites, 1), (i+sites, 0)), t) for i in␣
↪range(sites-1)])
    h += FermionOperator(((sites//2, 1), (sites//2, 0), (sites//2+sites, 1), (sites//
↪2+sites, 0)), U)
    init = FermionOperator(tuple([(i, 1) for i in range(sites//2+sites%2)] +␣
↪[(i+sites, 1) for i in range(sites//2, sites)]),
                           1.0)
    init = get_sparse_operator(init, sites*2)
    init = init.dot(np.array([1] + [0]*(2**(sites*2)-1)))
    h = get_sparse_operator(h, 2*sites)
    psi = sparse.linalg.expm_multiply(-1.0j * h, init)
    res[order].append(np.abs(psi.conj().T.dot(result.final_state))**2)

fig = plt.figure()
for k, r in res.items():
    plt.plot(Steps, r, '.-', label="Trotter order: {}".format(k))
plt.legend()
plt.xlabel("Trotter steps")
plt.ylabel("State fidelity")
plt.title("State fidelity for Anderson model using cirq simulator")

t = - 1.0
#U = 0.6 * np.pi
t, U = scale_H([t, U])
Ulist = np.linspace(0, 2)
resU = {1: [], 2: []}
for order, U in product((1, 2), Ulist):
    steps = 10
    circuit = cirq.Circuit()
    init = []
    for i in range(sites//2+sites%2):
        init.append(xmon_gates.ExpWGate(half_turns=1.0, axis_half_turns=0.
↪0)(qubits[i]))
    for i in range(sites//2, sites):
        init.append(xmon_gates.ExpWGate(half_turns=1.0, axis_half_turns=0.0)(qubits[i␣
↪+ sites]))
    circuit.append(init, strategy=cirq.circuits.InsertStrategy.EARLIEST)
    for j in range(steps):
        circuit.append(trotter_step(t/steps, U/steps, qubits, order=order),␣
↪strategy=cirq.circuits.InsertStrategy.EARLIEST)
    simulator = XmonSimulator()
```

(continues on next page)

(continued from previous page)

```
    result = simulator.simulate(circuit)

    h = np.sum([FermionOperator(((i, 1), (i+1, 0)), t) +
                FermionOperator(((i+1, 1), (i, 0)), t) +
                FermionOperator(((i+sites, 1), (i+1+sites, 0)), t) +
                FermionOperator(((i+1+sites, 1), (i+sites, 0)), t) for i in␣
↪range(sites-1)])
    h += FermionOperator(((sites//2, 1), (sites//2, 0), (sites//2+sites, 1), (sites//
↪2+sites, 0)), U)
    init = FermionOperator(tuple([(i, 1) for i in range(sites//2+sites%2)] +␣
↪[(i+sites, 1) for i in range(sites//2, sites)]),
                           1.0)
    init = get_sparse_operator(init, sites*2)
    init = init.dot(np.array([1] + [0]*(2**(sites*2)-1)))
    h = get_sparse_operator(h, 2*sites)
    psi = sparse.linalg.expm_multiply(-1.0j * h, init)
    resU[order].append(np.abs(psi.conj().T.dot(result.final_state))**2)
#    print("Overlap:", res[-1])
#    print(psi.conj().T.dot(h.dot(psi)).real)
#    print(result.final_state.conj().T.dot(h.dot(result.final_state)).real)

fig2 = plt.figure()
for k, r in resU.items():
    plt.plot(Ulist, r, '.-', label="Trotter order: {}".format(k))
plt.legend()
plt.xlabel("U")
plt.ylabel("State fidelity")
plt.title("State fidelity for Anderson model using cirq simulator")

plt.show()
```

CHAPTER 3

Convert to cirq

For example, we can translate a simple quantum algorithm to Cirq and print the final cirq.Circuit:

```
import cirq
import numpy as np
from cirqprojectq.circ_engine import CIRQ
import projectq
qubits = [cirq.google.XmonQubit(0, i) for i in range(2)]
CIRQ = CIRQ(qubits=qubits)
eng = projectq.MainEngine(backend=CIRQ)
qureg = eng.allocate_qureg(len(qubits))
eng.flush()
projectq.ops.Rx(0.25 * np.pi) | qureg[0]
projectq.ops.Ry(0.5 * np.pi) | qureg[1]
projectq.ops.Rz(0.5 * np.pi) | qureg[0]
projectq.ops.H | qureg[1]
projectq.ops.C(projectq.ops.X) | (qureg[0], qureg[1])
projectq.ops.Z | qureg[0]
projectq.ops.X | qureg[0]
projectq.ops.Y | qureg[0]
eng.flush()
print(CIRQ.circuit)
```

```
(0, 0): ───X^0.25───Z^0.5───────@───Z───X───Y───
                                │
(0, 1): ─────────────Y^0.5───H───X───────────────
```

3.1 The backend

Provides a projectq engine that translates a projectq circuit to a cirq circuit.

class cirqprojectq.circ_engine.**CIRQ**(*qubits=None*, *device=None*, *rules=None*, *strategy=InsertStrategy.EARLIEST*)

Bases: projectq.cengines._basics.BasicEngine

A projectq backend designated to translating to cirq.

Parameters

- **qubits** (list(`cirq.devices.grid_qubit`)) – the qubits
- **device** (`cirq.devices.Device`) – a device that provides the qubits.
- **rules** (`cirqprojectq._rules_pq_to_cirq.Ruleset_pq_to_cirq`) – rule set
- **strategy** (`cirq.circuits.InsertStrategy`) – Insert strategy in cirq.

circuit
`cirq.Circuit` – the circuit stored in the engine.

device
`cirq.devices.Device` – A device. Currently not used.

is_available(*cmd*)
Returns true if the command can be translated.

Parameters **cmd** (*Command*) – Command for which to check availability

qubits
list(`cirq.QubitID`) – The cirq qubits used in the circuit.

receive(*command_list*)
Receives a command list and, for each command, stores it until completion.

Parameters **command_list** – List of commands to execute

reset(*keep_map=True*)
Resets the engine.

3.2 Translation rules

This file provides classes to store rules for translating ProjectQ to Cirq operations.

class `cirqprojectq._rules_pq_to_cirq.`**Rule_pq_to_cirq**(*classes*, *translation*)
Bases: `object`

class `cirqprojectq._rules_pq_to_cirq.`**Ruleset_pq_to_cirq**(*rules=[]*)
Bases: `object`

add_rule(*rule*)
Add a single rule to the set of known rules.

Parameters **rule** (*Rule_pq_to_cirq*) – a rule that can be used for translations.

add_rules(*rules=[]*)
Add rules to the set of known rules.

Parameters **rules** (list of *Rule_pq_to_cirq*) – the rules that can be used for translations.

known_rules
Dictionary of known translations.

translate(*cmd*, *mapping*, *qubits*)
Translate a projectq operation into a Cirq operation.

Parameters

- **cmd** (`projectq.ops.Command`) – a projectq command instance

- **mapping** (dict) – a dictionary of qubit mappings
- **(list of** (*qubits*) – class:cirq.QubitID‘): cirq qubits

Returns cirq.Operation

3.2.1 Rules for common gates

This module provides translation rules from Projectq to Cirq for some common gates.

cirqprojectq.common_rules.**_gates_with_known_matrix**(*cmd*, *mapping*, *qubits*)

Translate a single qubit gate with known matrix into a Cirq gate.

Parameters

- **cmd** (projectq.ops.Command) – a projectq command instance
- **mapping** (dict) – a dictionary of qubit mappings
- **(list of** (*qubits*) – class:cirq.QubitID‘): cirq qubits

Returns cirq.Operation

cirqprojectq.common_rules.**_h_s_gate**(*cmd*, *mapping*, *qubits*)

Translate a Hadamard or S-gate into a Cirq gate.

Parameters

- **cmd** (projectq.ops.Command) – a projectq command instance
- **mapping** (dict) – a dictionary of qubit mappings
- **(list of** (*qubits*) – class:cirq.QubitID‘): cirq qubits

Returns cirq.Operation

cirqprojectq.common_rules.**_pauli_gates**(*cmd*, *mapping*, *qubits*)

Translate a Pauli (x, Y, Z) gate into a Cirq gate.

Parameters

- **cmd** (projectq.ops.Command) – a projectq command instance
- **mapping** (dict) – a dictionary of qubit mappings
- **(list of** (*qubits*) – class:cirq.QubitID‘): cirq qubits

Returns cirq.Operation

cirqprojectq.common_rules.**_rx_ry_rz**(*cmd*, *mapping*, *qubits*)

Translate a rotation gate into a Cirq roation (phase) gate.

Global phase difference betwee proejctq rotation gate and cirq phase gate is dropped.

Parameters

- **cmd** (projectq.ops.Command) – a projectq command instance
- **mapping** (dict) – a dictionary of qubit mappings
- **(list of** (*qubits*) – class:cirq.QubitID‘): cirq qubits

Returns cirq.Operation

3.2.2 Rules for xmon gates

This module provides translation rules from Xmon gates in Projectq to Xmon gates in Cirq.

`cirqprojectq.xmon_rules.`**`_exp11Gate`**(*cmd*, *mapping*, *qubits*)

Translate a ExpW gate into a Cirq gate.

Parameters

- **cmd** (–) –
- **mapping** (–) –
- **qubits (list of** (–) – class:cirq.QubitID‘) - cirq qubits

Returns

- `cirq.Operation`

`cirqprojectq.xmon_rules.`**`_expWGate`**(*cmd*, *mapping*, *qubits*)

Translate a ExpW gate into a Cirq gate.

Parameters

- **cmd** (–) –
- **mapping** (–) –
- **qubits (list of** (–) – class:cirq.QubitID‘) - cirq qubits

Returns

- `cirq.Operation`

`cirqprojectq.xmon_rules.`**`_expZGate`**(*cmd*, *mapping*, *qubits*)

Translate a ExpZ gate into a Cirq gate.

Parameters

- **cmd** (–) –
- **mapping** (–) –
- **qubits (list of** (–) – class:cirq.QubitID‘) - cirq qubits

Returns

- `cirq.Operation`

CHAPTER 4

Decompose to Xmon gates and simulate xmon devices

Cirqprojectq decomposes arbitrary circuits to xmon gates such that proejctq can be used to simulate xmon devices.

Important modules are

cirqprojectq.xmon_gates	This module provides Xmon gates for projectq.
cirqprojectq.xmon_decompositions	Provides decompositon rules to decompose common gates into Xmon gates.
cirqprojectq.xmon_setup	Provides proejctq engines for simulation of xmon devices.

In this example we show how to use projectq to decompose a circuit into Xmon native gates.

```
import cirqprojectq
from cirqprojectq.xmon_decompositions import all_defined_decomposition_rules as␣
↪xmondec

def is_supported(eng, cmd):
    if isinstance(cmd.gate, projectq.ops.ClassicalInstructionGate):
        # This is required to allow Measure, Allocate, Deallocate, Flush
        return True
    elif isinstance(cmd.gate, cirqprojectq.xmon_gates.XmonGate):
        return True
    else:
        return False

supported_gate_set_filter = InstructionFilter(is_supported)
ruleset = projectq.cengines.DecompositionRuleSet(xmondec)
replacer = projectq.cengines.AutoReplacer(ruleset)
engine_list = [replacer, supported_gate_set_filter]

eng = projectq.MainEngine(backend=projectq.backends.CommandPrinter(), engine_
↪list=engine_list)
qureg = eng.allocate_qureg(2)
```

(continues on next page)

(continued from previous page)

```
projectq.ops.H | qureg[0]
projectq.ops.H | qureg[1]
projectq.ops.C(projectq.ops.X) | (qureg[0], qureg[1])
eng.flush()
```

```
W(0.5, 0.5) | Qureg[0]
ExpZ(1.0) | Qureg[0]
W(0.5, 0.5) | Qureg[1]
ExpZ(1.0) | Qureg[1]
W(0.5, 0.5) | Qureg[1]
ExpZ(1.0) | Qureg[1]
Exp11(1.0) | ( Qureg[0], Qureg[1] )
W(0.5, 0.5) | Qureg[1]
ExpZ(1.0) | Qureg[1]
```

4.1 Decomposition rules

Provides decompositon rules to decompose common gates into Xmon gates.

The module *cirqprojectq.xmon_decompositions* provides decomposition rules for rotation gates (Rx, Ry, Rz), Pauli gates (X, Y, Z), the Hadamard gate and for CNOT gates into native Xmon gates. All defined rules can be imported as `cirqprojectq.xmon_decompositions.all_defined_decomposition_rules()`.

Note: The decompositions into xmon gates are correct up to global phases. If the module constant CORRECT_PHASES is set to True, ProjectQ global phase gates are applied to the affected qubits in order to keep track of the correct phase of thq qubit register. With `projectq.setups.decompositions.ph2r` these global phase gates can be translated to 1-qubit phase gates (R-gate) on control qubits if applicable. See example below

```
────  C ──────        ────  C ──── R(a) ──
      |          -->       |
  ---------             ---------
──|       | ──        ──|       | ────────
──| Ph(a) | ──        ──|       | ────────
──|       | ──        ──|       | ────────
  ---------             ---------
```

A full engine list can be generated with `cirqproejctq.xmon_setup.xmon_engines()`

`cirqprojectq.xmon_decompositions.`**`_check_phase`**(*angle*)

`cirqprojectq.xmon_decompositions.`**`_decompose_CNOT`**(*cmd*)

Decompose a CNOT gate into two Hadamards and an Exp11Gate.

Uses the following decomposition

```
── C ──          ───────     @      ───────
   |      -->                |
── X ──          ── H ── Exp11(1) ── H ──
```

This corresponds to the following map:

$$\text{CNOT} \rightarrow \text{CNOT}$$

Warning: The Hadamard gates are correct Hamard gates! They have no wrong phases. However, in a second decomposition step these gates will each yield a phase of $\exp(-i\pi/4)$ if CORRECT_PHASES is False! In this case, the final map will be $\mathrm{CNOT} \rightarrow \exp(-i\pi/4)\mathrm{CNOT}$

`cirqprojectq.xmon_decompositions.`**`_decompose_H`**(*cmd*)

Decompose a Hadamard gate into xmon gates.

If `CORRECT_PHASES()` is False the following decompositions will be used:

```
── H ──  -->  ── ExpW(1/2, 1/2) ── ExpZ(1) ──
```

This corresponds to the following map:

$$H \rightarrow e^{-i\pi/4}H$$

If `CORRECT_PHASES()` is True, the phases are countered by a projectq phase gate `ops.Ph`.

`cirqprojectq.xmon_decompositions.`**`_decompose_SWAP`**(*cmd*)

TODO

`cirqprojectq.xmon_decompositions.`**`_decompose_paulis`**(*cmd*)

Decompose a Pauli gate into xmon gates.

If `CORRECT_PHASES()` is False the following decompositions will be used:

```
Z  -->  ── ExpZ(1.0) ──
```

This corresponds to the following map:

$$Z \rightarrow \mathrm{ExpZ}(1) = e^{-i\pi/2}Z$$

```
── X ──  -->  ── ExpWGate(1, 0) ──
```

This corresponds to the following map:

$$X \rightarrow \mathrm{ExpW}(1, 0) = X$$

```
── Y ──  -->  ── ExpWGate(1, 1/2) ──
```

This corresponds to the following map:

$$Y \rightarrow \mathrm{ExpW}(1, 1/2) = Y$$

If `CORRECT_PHASES()` is True, the phases are countered by a projectq phase gate `ops.Ph`.

`cirqprojectq.xmon_decompositions.`**`_decompose_rotations`**(*cmd*)

Decompose a rotation gate into xmon gates.

We define a rotation gate as

$$R_i(\alpha) = \exp(-i\frac{\alpha}{2}\sigma_i)$$

If `CORRECT_PHASES()` is False the following decompositions will be used:

```
── Rz(a) ──  -->  ── ExpZ(a / pi) ──
```

This corresponds to the following map:

$$R_z(\alpha) \to \mathrm{ExpZ}(\alpha/\pi) = \begin{cases} -R_z(\alpha), & \pi < \alpha \leq 3\pi \\ R_z(\alpha), & \text{else} \end{cases}$$

```
── Rx(a) ──   -->   ── ExpWGate(a/pi, 1) ──
```

This corresponds to the following map:

$$R_x(\alpha) \to \mathrm{ExpW}(\alpha/\pi, 0) = e^{i\alpha/2} R_x(\alpha)$$

```
── Ry(a) ──   -->   ── ExpWGate(a/pi, 1/2) ──
```

This corresponds to the following map:

$$R_y(\alpha) \to \mathrm{ExpW}(\alpha/\pi, 1/2) = e^{i\alpha/2} R_y(\alpha)$$

If `CORRECT_PHASES()` is True, the phases are countered by a projectq phase gate `ops.Ph` such that the obtained pahse is correct.

`cirqprojectq.xmon_decompositions.`**`_recognize_CNOT`**(*cmd*)

`cirqprojectq.xmon_decompositions.`**`_recognize_H`**(*cmd*)

`cirqprojectq.xmon_decompositions.`**`_recognize_SWAP`**(*cmd*)

`cirqprojectq.xmon_decompositions.`**`_recognize_paulis`**(*cmd*)

`cirqprojectq.xmon_decompositions.`**`_recognize_rotations`**(*cmd*)

4.2 Xmon gates

This module provides Xmon gates for projectq.

Xmon qubits are a specific type of transmon qubit developed by Google. In the cirq framework xmon specific gates are defined. This file ports these gates to projectq and allows imulation of algorithms with xmon qubits.

class `cirqprojectq.xmon_gates.`**`Exp11Gate`**(*half_turns*)

Bases: *`cirqprojectq.xmon_gates.XmonGate`*

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

This gate implements $\exp(i\pi\varphi|11\rangle\langle 11|)$ where φ is half turns.

As a matrix it reads

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \exp(i\varphi\pi) \end{pmatrix}$$

> **Warning:** There is no (-) sign in the definition of the gate.

Note: The 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).

Parameters **half_turns** (*float*) – angle of rotation in units of π.

angle
Rotation angle in rad, $\in (-\pi, \pi]$

half_turns
Rotation angle in half turns, $\in (-1, 1]$

matrix
Gate matrix.

With $\varphi = \mathrm{half\_turns}$ the matrix implemented by the Exp11Gate reads as

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \exp(i\varphi\pi) \end{pmatrix}$$

tex_str()
Returns the class name and the angle as a subscript, i.e.

```
[CLASSNAME]$_[ANGLE]$
```

class cirqprojectq.xmon_gates.**ExpWGate**(*half_turns*, *axis_half_turns=0*)
Bases: *cirqprojectq.xmon_gates.XmonGate*

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

$$\exp(-i * \pi * W(\mathrm{axis\_half\_turn}) * \mathrm{half\_turn}/2)$$

where

$$W(\theta) = \cos(\pi\theta)X + \sin(\pi\theta)Y$$

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.'

Contrary to the docstring, cirq implements the gate ($\varphi = \mathrm{half\_turns}$ and $\theta = \mathrm{axis\_half\_turns}$):

$$\exp(i\pi\varphi/2)\exp(-i\pi W(\theta)\varphi/2)$$

which differs by a global phase $\exp(i\varphi/2)$.

This class mimics the cir implementation, i.e., the matrix represented by this gate reads as

$$\exp(i\varphi\pi/2)\begin{pmatrix}\cos(\varphi\pi/2) & -i\sin(\varphi\pi/2)exp(-i\theta\pi)\\ -i\sin(\varphi\pi/2)\exp(i\theta\pi) & \cos(\varphi\pi/2)\end{pmatrix}$$

whih is a rotation around the W-axis and an additional global phase.

Note: This gate corresponds to a phase gate in the basis defined by the W-axis, not to a rotation gate.

Note: Restricting half_turns to the rage (-1, 1] corresponds to the full range (0, 2pi] for a phase gate, which the ExpW gate in the cirq implementation actually is.

Note: Another convention is to change a positive rotation around a negative axis to a negative rotation around a postive axis, i.e., half_turns -> - half_turns and axis_half_turns -> axis_half_turns + 1 if axis_half_turns < 0.

Parameters

- **half_turns** (*float*) – angle of rotation in units of π.
- **axis_half_turns** (*float*) – axis between X and Y in units of π.

angle
Rotation angle in rad, $\in (-\pi, \pi]$

axis_angle
Axis angle in rad, $\in (-\pi, \pi]$

axis_half_turns
Axis angle in half turns, $\in (-1, 1]$

half_turns
Rotation angle in half turns, $\in (-1, 1]$

matrix
Rotation matrix.

With $\varphi = \text{half\_turns}$ and $\theta = \text{axis\_half\_turns}$ this gate implements the matrix

$$\exp(i\varphi\pi/2)\begin{pmatrix}\cos(\varphi\pi/2) & -i\sin(\varphi\pi/2)exp(-i\theta\pi)\\ -i\sin(\varphi\pi/2)\exp(i\theta\pi) & \cos(\varphi\pi/2)\end{pmatrix}$$

tex_str()

class cirqprojectq.xmon_gates.**ExpZGate**(*half_turns*)
Bases: *cirqprojectq.xmon_gates.XmonGate*

A rotation around the Z axis of the Bloch sphere.

This gate implements $\exp(-i\pi Z\varphi/2)$ where Z is the Z matrix

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

and φ are half turns.

The full matrix reads:

$$\begin{pmatrix}exp(-i\pi\varphi/2) & 0\\ 0 & exp(i\pi\varphi/2)\end{pmatrix}$$

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.

Half_turns are mapped to the range (-1, 1], i.e. to rotation angles in the range (-pi, pi].

Note: Restricting half_turns to the rage (-1, 1] corresponds to the full range (0, 2pi] for the phase difference between the Z eigenstates. However, we loose the global phase that comes with a full rotation gate in the range (0, 4pi]. Thus, the ExpZ gate is more like a phase then a rotation gate.

Parameters **half_turns** (*float*) – number of half turns on the Bloch sphere.

angle
Rotation angle in rad, $\in (-\pi, \pi]$

get_merged(*other*)
Return self merged with another gate. Default implementation handles rotation gate of the same type, where angles are simply added. :param other: Rotation gate of same type.

Raises `NotMergeable` – For non-rotation gates or rotation gates of different type.

Returns New object representing the merged gates.

half_turns
Rotation angle in half turns, $\in (-1, 1]$

matrix
Rotation matrix.

With $\varphi = \mathrm{half\_turns}$ this gate implements the matrix

$$\begin{pmatrix} \cos(\varphi\pi/2) - i\sin(\varphi\pi/2) & 0 \\ 0 & \cos(\varphi\pi/2) + i\sin(\varphi\pi/2) \end{pmatrix}$$

tex_str()
Latex representation of the gate.

class cirqprojectq.xmon_gates.**XmonGate**
Bases: `projectq.ops._basics.BasicGate`

cirqprojectq.xmon_gates.**drawer_settings**()

4.3 Xmon setup

Provides proejctq engines for simulation of xmon devices.

A full engine list can be generated with `cirqproejctq.xmon_setup.xmon_engines()`

cirqprojectq.xmon_setup.**replacer_xmon**()
Autoreplacer for decomposition into xmon gates.

cirqprojectq.xmon_setup.**xmon_engines**()
Full engine list for simulation with xmon gates.

cirqprojectq.xmon_setup.**xmon_rules**()
DecompositionRuleSet for decomposition into xmon gates.

cirqprojectq.xmon_setup.**xmon_supported_filter**()
InstructionFilter for xmon gates.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

Symbols

A

C

D

E

G

H