Spring 2024 Kickoff

Sam Bieberich

Announcements

- Thanks for coming back
- QHack
    - https://qhack.ai/online-events/#streaming-sessions

What are we doing now?

We didn’t finish the book last semester, but instead of just going chapter by chapter, we will be targeting the prerequisites for the exam:

https://www.ibm.com/training/certification/ibm-certified-associate-developer-quantum-computation-using-qiskit-v02x-C0010300

On next slide too…

Number of questions: 60
Number of questions to pass: 44

Time allowed: 90 minutes
Status: Live

Section 1: Perform Operations on Quantum Circuits	47%
Section 2: Executing Experiments	3%
Section 3: Implement BasicAer: Python-based Simulators	3%
Section 4: Implement Qasm	1%
Section 5: Compare and Contrast Quantum Information	10%
Section 6: Return the Experiment Results	7%
Section 7: Use Qiskit Tools	1%
Section 8: Display and Use System Information	3%
Section 9: Construct Visualizations	19%
Section 10: Access Aer Provider	6%

Plans

Meetings 1-3 (Feb)

- Section 2-5
- Sections 7,8,10
- Section 1

Meetings 4-5 (Mar)

- Section 6
- Practice problems and exams

Exam

The exam is $200, but we think that scholarships would work out. To ensure that the people who are most dedicated to the exam (and passing it) get the scholarships, we will be taking attendance again, so please try to come every time.

We will also host a few practices in April, usually the exam is June/July online, so consider taking it if you can attend.

Section 2

Executing Experiments

Executing Experiments

To execute a quantum experiment/circuit, use the Qiskit execute_function command.

Ex:

```
from qiskit import QuantumCircuit, execute, BasicAer

backend = BasicAer.get_backend('qasm_simulator')

qc = QuantumCircuit(5, 5)
qc.h(0)
qc.cx(0, range(1, 5))
qc.measure_all()

job = execute(qc, backend, shots=4321)
```

Executing Experiments

qiskit.execute_function.execute(**experiments**, **backend**, basis_gates=None, coupling_map=None, backend_properties=None, initial_layout=None, seed_transpiler=None, optimization_level=None, pass_manager=None, **shots**=None, memory=None, seed_simulator=None, default_qubit_los=None, default_meas_los=None, qubit_lo_range=None, meas_lo_range=None, schedule_los=None, meas_level=None, meas_return=None, memory_slots=None, memory_slot_size=None, rep_time=None, rep_delay=None, parameter_binds=None, schedule_circuit=False, inst_map=None, meas_map=None, scheduling_method=None, **init_qubits**=None, **run_config)

Executing Experiments

So what does this do?

Definition: Executes a list of qiskit.circuit.QuantumCircuit or qiskit.pulse, then schedules on a backend.

The execution is asynchronous, and a handle to a job instance is returned.

Executing Experiments

Important Paramters:

1. Experiments: (QuantumCircuit or list or Schedule – Circuit(s) or pulse schedule(s) to execute
2. Backend: Backend to execute circuits on. Transpiler options are automatically grabbed from backend.configuration() and backend.properties(). If any other option is explicitly set (e.g. coupling_map), it will override the backend's.
3. Shots: Number of repetitions of each circuit, for sampling. Default: 1024

Executing Experiments

Returns

Returns job instance derived from Job

Return type

Job

Raises

QiskitError – if the execution cannot be interpreted as either circuits or schedules

Executing Experiments

Also need to know how to run a job in the circuit composer

Demo:
https://quantum.ibm.com/composer/files/2c3c0cf9b082b5ce7dd7d9fd755ce253

Section 3

Python-based Simulators (BasicAer)

Why Simulators?

Quantum computing, by its very nature, cannot be efficiently reproduced using classical computers. The resources required to simulate quantum hardware increases exponentially with the number of qubits. For the hardware developed in the next few years, even the world’s largest classical supercomputers won’t be enough.

Despite this, these near-term devices are still ‘intermediate scale’. Though large enough that their simulation will be intractable, they won’t yet be large enough to implement full-scale quantum error correction. This means that errors will occur during execution of any quantum computation. The longer our quantum program, the more these errors will accumulate.

These errors are unavoidable, and can take many forms depending on the physics of the devices. To develop quantum algorithms that are robust against their effects, we need to know our enemy. This requires us to have an accurate model of the errors that occur, as well as the ability to simulate their effects. Then we’ll be much better equipped to explore near-term quantum applications with noisy devices.

-IBM

Simulators

<u>BasicAer: Python-based Simulators</u>

qiskit.providers.basicaer

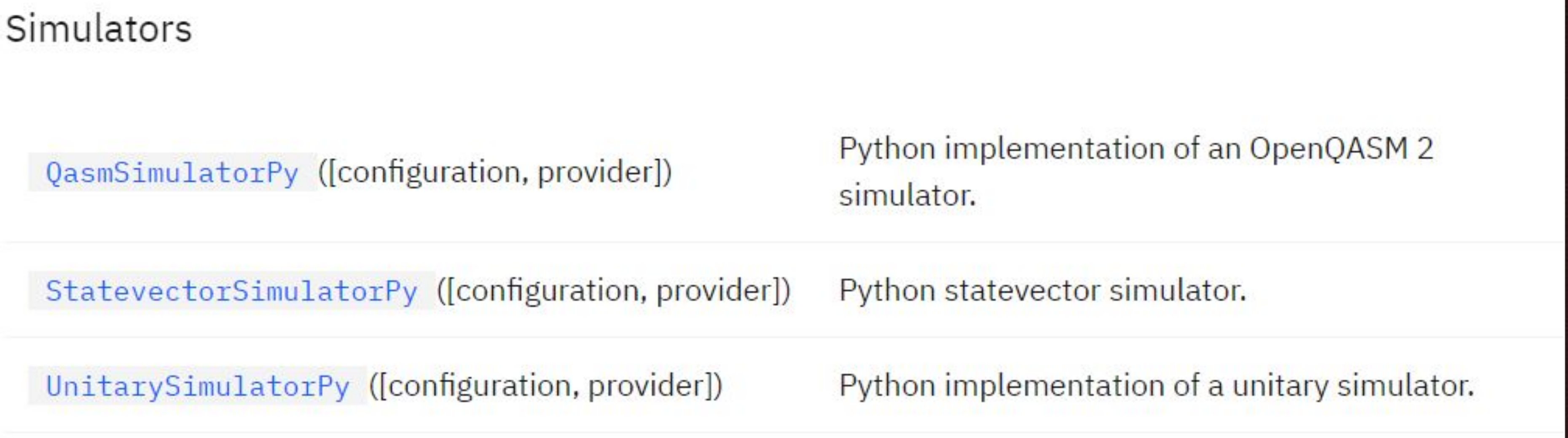
A module of Python-based quantum simulators. Simulators are accessed via the BasicAer provider:

```
from qiskit import BasicAer

backend = BasicAer.get_backend('qasm_simulator')
```

More info:

https://docs.quantum.ibm.com/api/qiskit/providers_basicaer

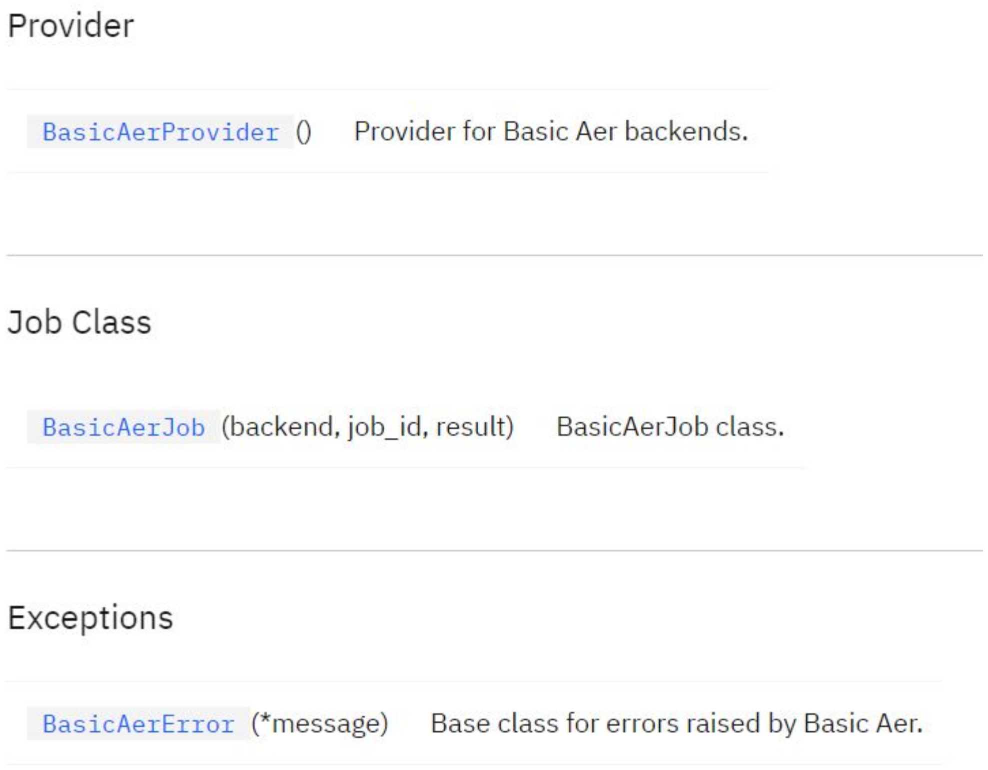
Simulators



Simulators

`QasmSimulatorPy` ([configuration, provider])	Python implementation of an OpenQASM 2 simulator.
`StatevectorSimulatorPy` ([configuration, provider])	Python statevector simulator.
`UnitarySimulatorPy` ([configuration, provider])	Python implementation of a unitary simulator.

Simulators



Provider

`BasicAerProvider` ()	Provider for Basic Aer backends.

Job Class

`BasicAerJob` (backend, job_id, result)	BasicAerJob class.

Exceptions

`BasicAerError` (*message)	Base class for errors raised by Basic Aer.

Simulators

Goal:

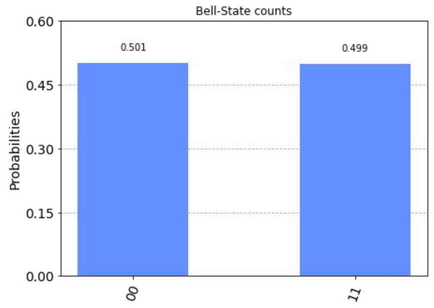
1. Returning the histogram data of an experiment
2. Returning the state vector of an experiment
3. Returning the unitary of an experiment
4. Available simulators
5. Accessing a statevector_simulator backend
6. Accessing a qasm_simulator backend
7. Accessing a unitary_simulator backend

Okay let’s do an example of each

Returning the histogram data of an experiment

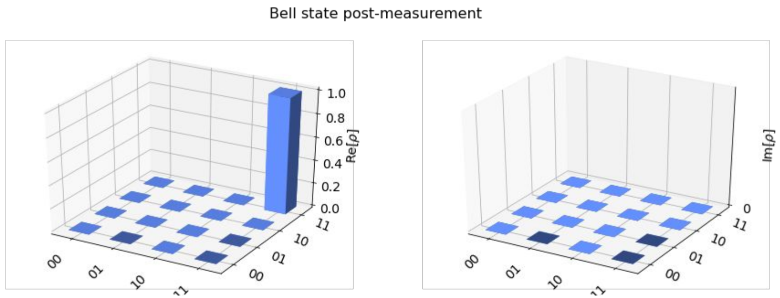
```
import numpy as np

# Import Qiskit
from qiskit import QuantumCircuit
from qiskit import Aer, transpile
from qiskit.tools.visualization import plot_histogram

Aer.backends()
[2]:

[AerSimulator('aer_simulator'),
 AerSimulator('aer_simulator_statevector'),
 AerSimulator('aer_simulator_density_matrix'),
 AerSimulator('aer_simulator_stabilizer'),
AerSimulator('aer_simulator_matrix_product_state'),
 AerSimulator('aer_simulator_extended_stabilizer'),
 AerSimulator('aer_simulator_unitary'),
 AerSimulator('aer_simulator_superop'),
 QasmSimulator('qasm_simulator'),
 StatevectorSimulator('statevector_simulator'),
 UnitarySimulator('unitary_simulator'),
```

```
# Create circuit
circ = QuantumCircuit(2)
circ.h(0)
circ.cx(0, 1)
circ.measure_all()

# Transpile for simulator
simulator = Aer.get_backend('aer_simulator')
circ = transpile(circ, simulator)

# Run and get counts
result = simulator.run(circ).result()
counts = result.get_counts(circ)
plot_histogram(counts, title='Bell-State counts')
```



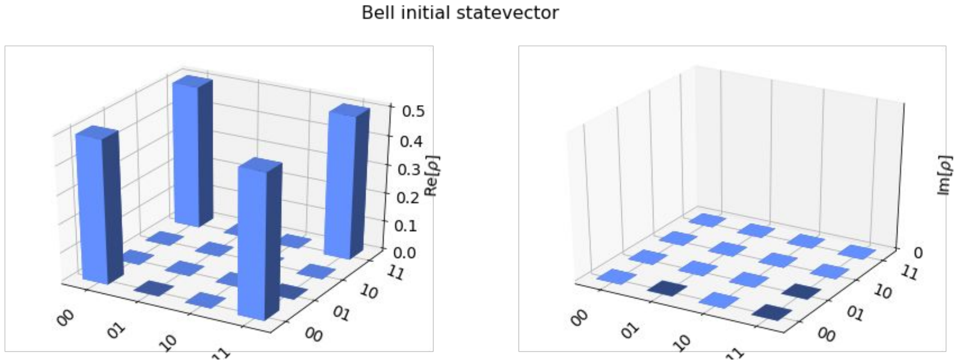
Returning the state vector of an experiment

```
# Construct quantum circuit with measure
circ = QuantumCircuit(2, 2)
circ.h(0)
circ.cx(0, 1)
circ.measure([0,1], [0,1])

# Select the StatevectorSimulator from the Aer provider
simulator = Aer.get_backend('statevector_simulator')

# Execute and get counts
result = execute(circ, simulator).result()
statevector = result.get_statevector(circ)
plot_state_city(statevector, title='Bell state post-measurement')
```



Returning the state vector of an experiment part 2

```
# Construct a quantum circuit that initialises qubits to a custom state
circ = QuantumCircuit(2)
circ.initialize([1, 0, 0, 1] / np.sqrt(2), [0, 1])

# Select the StatevectorSimulator from the Aer provider
simulator = Aer.get_backend('statevector_simulator')

# Execute and get counts
result = execute(circ, simulator).result()
statevector = result.get_statevector(circ)
plot_state_city(statevector, title="Bell initial statevector")
```



Returning the unitary of an experiment

The UnitarySimulator constructs the unitary matrix for a Qiskit Quantum Circuit by applying each gate matrix to an identity matrix. The circuit may only contain gates, if it contains resets or measure operations an exception will be raised.

Returning the unitary of an experiment part 2

```
# Construct an empty quantum circuit
circ = QuantumCircuit(2)
circ.h(0)
circ.cx(0, 1)

# Select the UnitarySimulator from the Aer provider
simulator = Aer.get_backend('unitary_simulator')

# Execute and get counts
result = execute(circ, simulator).result()
unitary = result.get_unitary(circ)
print("Circuit unitary:\n", unitary)
```

```
Circuit unitary:
 [[ 0.70710678+0.00000000e+00j  0.70710678-8.65956056e-17j
   0.        +0.00000000e+00j  0.        +0.00000000e+00j]
 [ 0.        +0.00000000e+00j  0.        +0.00000000e+00j
   0.70710678+0.00000000e+00j -0.70710678+8.65956056e-17j]
 [ 0.        +0.00000000e+00j  0.        +0.00000000e+00j
   0.70710678+0.00000000e+00j  0.70710678-8.65956056e-17j]
 [ 0.70710678+0.00000000e+00j -0.70710678+8.65956056e-17j
   0.        +0.00000000e+00j  0.        +0.00000000e+00j]]
```

Returning the unitary of an experiment part 3

```
# Construct an empty quantum circuit
circ = QuantumCircuit(2)
circ.id([0,1])

# Set the initial unitary
unitary1 = np.array([[ 1,  1,  0,  0],
                     [ 0,  0,  1, -1],
                     [ 0,  0,  1,  1],
                     [ 1, -1,  0,  0]] / np.sqrt(2))

# Select the UnitarySimulator from the Aer provider
simulator = Aer.get_backend('unitary_simulator')

# Execute and get counts
result = execute(circ, simulator, initial_unitary=unitary1).result()
unitary2 = result.get_unitary(circ)
print("Initial Unitary:\n", unitary2)
```

```
Initial Unitary:
 [[ 0.70710678+0.j  0.70710678+0.j  0.        +0.j  0.        +0.j]
 [ 0.        +0.j  0.        +0.j  0.70710678+0.j -0.70710678+0.j]
 [ 0.        +0.j  0.        +0.j  0.70710678+0.j  0.70710678+0.j]
 [ 0.70710678+0.j -0.70710678+0.j  0.        +0.j  0.        +0.j]]
```

Available simulators

```
# List Aer backends
Aer.backends()

[QasmSimulator(
 backend_name='qasm_simulator', provider=AerProvider()),
 StatevectorSimulator(
 backend_name='statevector_simulator', provider=AerProvider()),
 UnitarySimulator(
 backend_name='unitary_simulator', provider=AerProvider()),
 PulseSimulator(
 backend_name='pulse_simulator', provider=AerProvider())]
```

Accessing a statevector_simulator backend

```
# Select the StatevectorSimulator from the Aer provider
simulator = Aer.get_backend('statevector_simulator')
```

<u>Note: Remember to use</u>

```
from qiskit import Aer
```

Accessing a qasm_simulator backend

```
simulator = Aer.get_backend('qasm_simulator')
```

Accessing a unitary_simulator backend

```
simulator = Aer.get_backend('unitary_simulator')
```

Who’d have guessed

Simulators

1. The **QASM Simulator** is the **main** Qiskit Aer **backend**. This backend **emulates execution of a quantum circuits on a real device and returns measurement counts**. It includes highly configurable noise models and can even be loaded with automatically generated approximate noise models based on the calibration parameters of actual hardware devices.

2. The Statevector Simulator is an auxiliary backend for Qiskit Aer. **It simulates the ideal execution of a quantum circuit and returns the final quantum state vector of the device at the end of simulation**. This is useful for education, as well as the **theoretical study** and debugging of algorithms.

3. The Unitary Simulator is another auxiliary backend for Qiskit Aer. It allows simulation of the final unitary matrix implemented by an **ideal quantum circuit**. This is also useful for education and algorithm studies.

Section 4

Implement QASM

What’s QASM

QASM refers to Quantum Assembly Language or QASM. QASM is a simple text-format language that has elements of both the C and assembly languages which will be used for describing or specifying quantum circuits to be used as input into programming a quantum computer by building universal set of quantum gates.

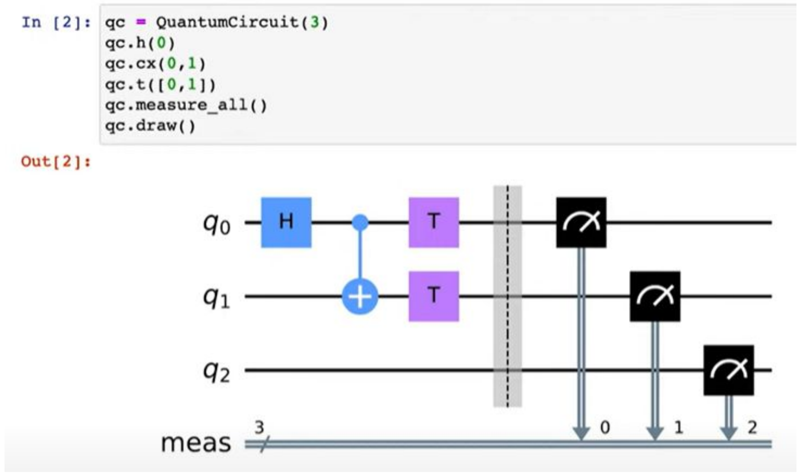
In the QASM program, classical bits and qubits are declared which will describes the operations or gates working on the qubits and the measurements required to obtain the classical results.

-https://so.ilitchbusiness.wayne.edu/knowledge/what-is-qasm

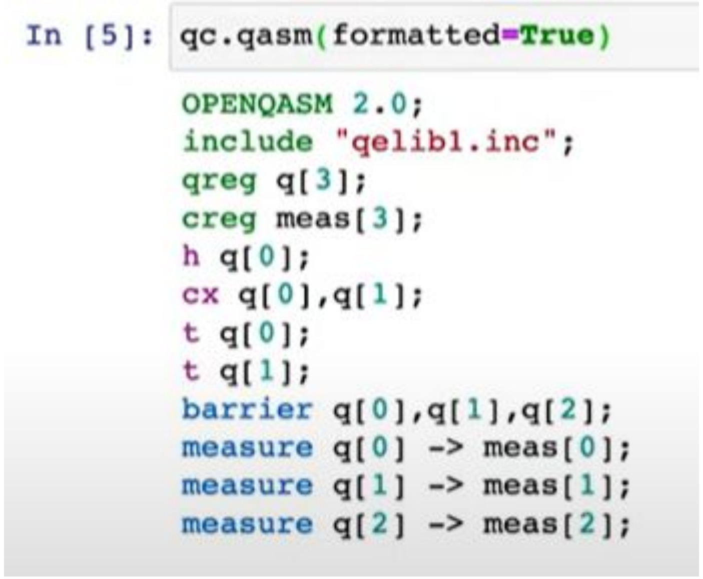
Goals

1. Returning the OpenQASM string for a circuit
2. Reading a QASM file

Returning the OpenQASM string for a circuit



```
In [2]: qc = QuantumCircuit(3)
        qc.h(0)
        qc.cx(0,1)
        qc.t([0,1])
        qc.measure_all()
        qc.draw()
Out[2]:
```



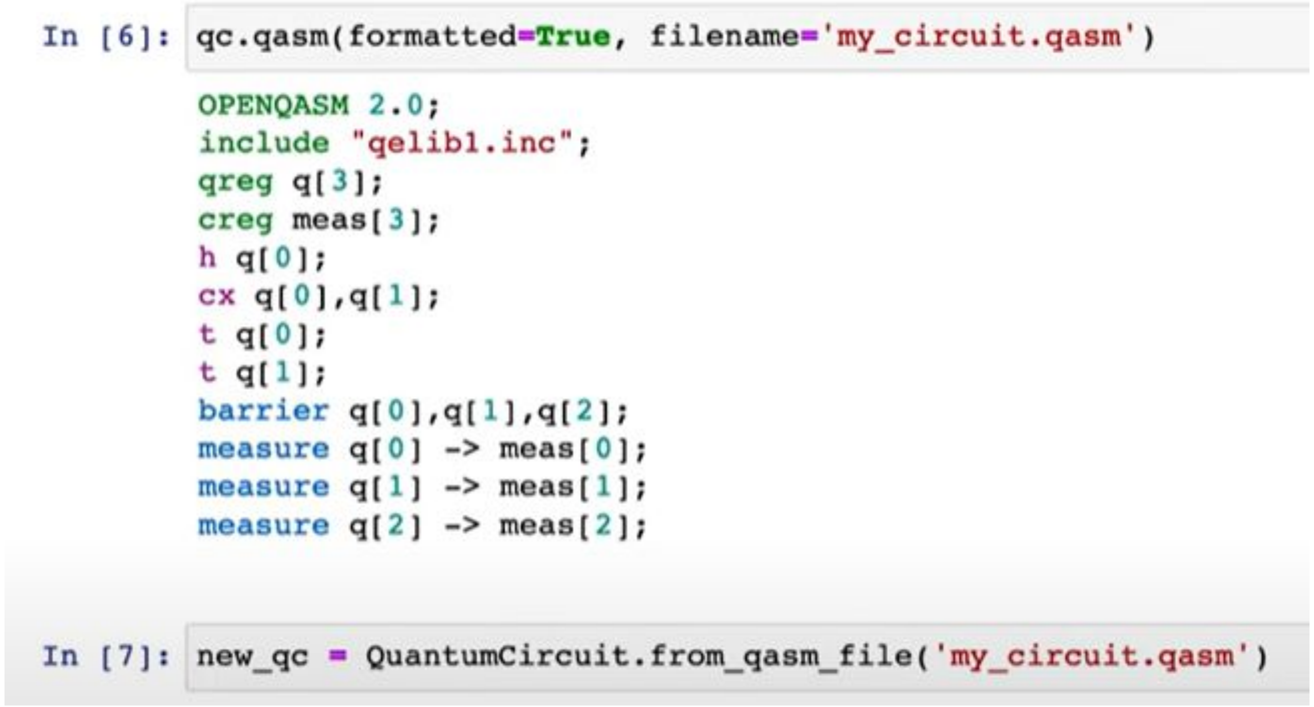
```
In [5]: qc.qasm(formatted=True)

        OPENQASM 2.0;
        include "qelib1.inc";
        qreg q[3];
        creg meas[3];
        h q[0];
        cx q[0],q[1];
        t q[0];
        t q[1];
        barrier q[0],q[1],q[2];
        measure q[0] -> meas[0];
        measure q[1] -> meas[1];
        measure q[2] -> meas[2];
```

Reading a QASM file

I am choosing to interpret this as reading data from a QASM file:

Note, files can also be copied into the circuit composer on the IBMQ site to make it easier to see a circuit with the block-based diagrams

Reading a QASM file



```
In [6]: qc.qasm(formatted=True, filename='my_circuit.qasm')

        OPENQASM 2.0;
        include "qelib1.inc";
        qreg q[3];
        creg meas[3];
        h q[0];
        cx q[0],q[1];
        t q[0];
        t q[1];
        barrier q[0],q[1],q[2];
        measure q[0] -> meas[0];
        measure q[1] -> meas[1];
        measure q[2] -> meas[2];

In [7]: new_qc = QuantumCircuit.from_qasm_file('my_circuit.qasm')
```

Wait, wait, wait, what’s OpenQASM?

Early QASM languages assumed a discrete set of quantum gates, but OpenQASM is flexible enough to describe universal computation with a continuous gate set.

This gate set was chosen for the convenience of defining new quantum gates and is **not** an enforced compilation target.

Usually when QASM is referred to in Qiskit, OpenQASM (at least) applies, assuming they didn’t mean it outright.

Section 5

Compare and Contrast Quantum Information

Compare and Contrast Quantum Information

Goals:

1. Use classical and quantum registers
2. Use operators
3. Measure fidelity

Making a Register

```
qiskit.circuit.QuantumRegister(size=None, name=None, bits=None)
```

Implement a quantum register.

Create a new generic register.

Either the `size` or the `bits` argument must be provided. If `size` is not None, the register will be pre-populated with bits of the correct type.

Parameters

- size (*int*
- (opens in a new tab)
-) – Optional. The number of bits to include in the register.
- name (*str*
- (opens in a new tab)
-) – Optional. The name of the register. If not provided, a unique name will be auto-generated from the register type.
- bits (*list*
- (opens in a new tab)
- *[Bit]*) – Optional. A list of Bit() instances to be used to populate the register.

Differences

QuantumRegister
- Collection of **qubits**
- Indexed to reference individual qubit: q[0]

ClassicalRegister
- Collection of **bits**
- Used as the **receiver of measurements** on qubits

Example

```
from qiskit import QuantumRegister, ClassicalRegister,
QuantumCircuit

qreg = QuantumRegister(3) # a 3-qubit register
creg = ClassicalRegister(3) # a 3-bit classical register

qc = QuantumCircuit(qreg,creg) # create a circuit

qc.measure(qreg,creg) # measure all qubits in qr, put results in cr
```

Bell State Example

```
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit

q = QuantumRegister(2)
c = ClassicalRegister(2)

qc = QuantumCircuit(q, c)

qc.h(q[0]) # Hadamard on first qubit
qc.cx(q[0],q[1]) # CNOT to entangle

# creates a Bell state
qc.measure(q,c)
```

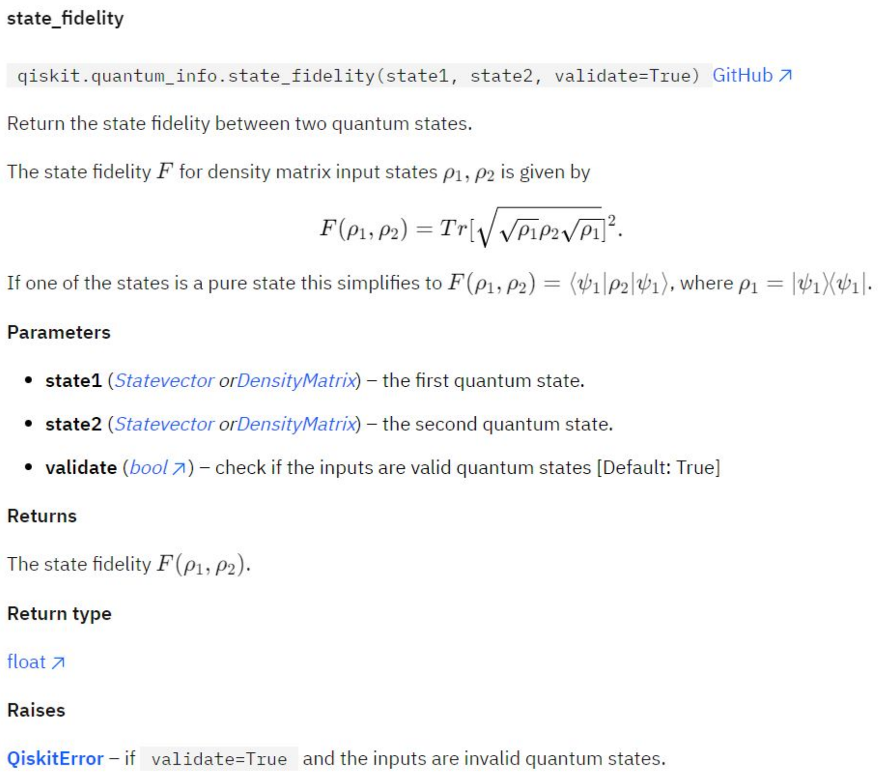
Operators

Operators module overview | IBM Quantum Documentation

This is kinda involved so above is the link. I encourage you to look up the plaintext version and read through the page in full when this is over, since we likely will not have time to cover everything.

Most quantum computing code defines an operator for the problem to be solved, so it is **one of the most important applications of Qiskit**.

Fidelity



state_fidelity

`qiskit.quantum_info.state_fidelity(state1, state2, validate=True)` GitHub ↗

Return the state fidelity between two quantum states.

The state fidelity F for density matrix input states ρ_1, ρ_2 is given by

$$F(\rho_1, \rho_2) = Tr[\sqrt{\sqrt{\rho_1}\rho_2\sqrt{\rho_1}}]^2.$$

If one of the states is a pure state this simplifies to $F(\rho_1, \rho_2) = \langle\psi_1|\rho_2|\psi_1\rangle$, where $\rho_1 = |\psi_1\rangle\langle\psi_1|$.

Parameters

- **state1** (*Statevector* or*DensityMatrix*) – the first quantum state.
- **state2** (*Statevector* or*DensityMatrix*) – the second quantum state.
- **validate** (*bool* ↗) – check if the inputs are valid quantum states [Default: True]

Returns

The state fidelity $F(\rho_1, \rho_2)$.

Return type

float ↗

Raises

QiskitError – if `validate=True` and the inputs are invalid quantum states.

The End

Samuel Bieberich