Method name	Description
`trace`	Returns the trace of the quantum state as if it was represented as a density matrix. Also see "Using the DensityMatrix Class" on page 91

Table 5-3. Some `Statevector` attributes

Attribute name	Description
`data`	Contains the complex vector.
`dim`	Contains the number of basis states in the statevector.
`num_qubits`	Contains the number of qubits in the statevector, or None.

Example of using Statevector methods

As an example of using some of these `Statevector` methods and attributes, we'll first use the `from_label` method to create a `Statevector` whose basis states have equal probabilities of being the result of a measurement:

```
from qiskit.quantum_info import Statevector

statevector = Statevector.from_label('+-')
print(statevector.data)

output:
  [ 0.5+0.j -0.5+0.j  0.5+0.j -0.5+0.j]
```

We'll then use the `draw` method to visualize the statevector as the Q-sphere shown in Figure 5-1:

```
statevector.draw("qsphere")
```

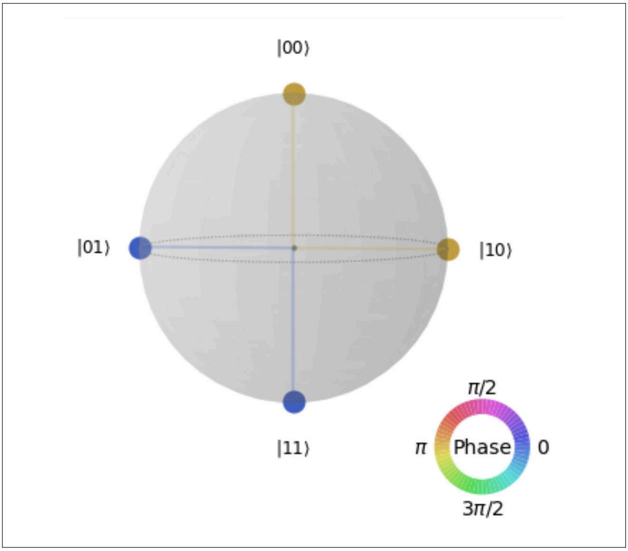


Figure 5-1. Example visualization produced with `Statevector draw("qsphere")`

Next, we'll use the `probabilities` method to show the probabilities of each basis state being the result of a measurement.

```
print(statevector.probabilities())

output:
  [0.25 0.25 0.25 0.25]
```

Finally, we'll use the `sample_counts` method to sample the probability distribution as if the circuit were being measured ~~1000~~ times.

```
print(statevector.sample_counts(1000))

output:
  {'00': 241, '01': 229, '10': 283, '11': 247}
```

Using the DensityMatrix Class

The `DensityMatrix` class represents a quantum density matrix, and contains functionality for initializing and operating on the density matrix. For example, as shown in the following code snippet, a `DensityMatrix` may be instantiated by passing in a `QuantumCircuit` instance.

```
from qiskit import QuantumCircuit
from qiskit.quantum_info import DensityMatrix

qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)
qc.z(1)

dens_mat = DensityMatrix(qc)
print(dens_mat.data)

output:
  [[ 0.5+0.j  0. +0.j  0. +0.j -0.5+0.j]
   [ 0. +0.j  0. +0.j  0. +0.j  0. +0.j]
   [ 0. +0.j  0. +0.j  0. +0.j  0. +0.j]
   [-0.5+0.j  0. +0.j  0. +0.j  0.5+0.j]]
```

Notice that the `DensityMatrix` contains a complex matrix, as opposed to the `Statevector` which contains a complex vector. This enables the `DensityMatrix` to represent *mixed states*, which are an ensemble of two or more quantum states.

Table~~ 5-4 and Table 5-5~~ describe some of the methods and attributes in the `DensityMatrix` class.

Table 5-4. Some `DensityMatrix` methods

Method name	Description
`conjugate`	Returns the complex conjugate of the density matrix.
`copy`	Creates and returns a copy of the density matrix.
`dims`	Returns a tuple of dimensions.

Method name	Description
`draw`	Returns a visualization of the `DensityMatrix`, given the desired output method from the following: *text*, *latex*, *latex_source*, *qsphere*, *hinton*, *bloch*, *city*, or *paulivec*. Also see Chapter 3
`evolve`	Returns a quantum state evolved by the supplied operator. Also see "Using Quantum Information Operators" on page 95.
`expand`	Returns the reverse-order tensor product state of this density matrix and a supplied `DensityMatrix`.
`expecta tion_value`	Computes and returns the expectation value of a supplied operator.
`from_instruc tion`	Returns the `DensityMatrix` output of a supplied `Instruction` or `QuantumCircuit` instance.
`from_label`	Instantiates a `DensityMatrix` given a string of eigenstate ket labels. Each ket label may be `0`, `1`, `+`, `-`, `r`, or `l`~~,~~ and correspond to the six states found on the X, Y and Z axes of a Bloch sphere.
`is_valid`	Returns a boolean indicating whether this density matrix has trace 1 and is positive semi-definite.
`measure`	Returns the measurement outcome as well as post-measure state.
`probabilities`	Returns the measurement probability vector.
`probabili ties_dict`	Returns the measurement probability dictionary.
`purity`	Returns a number from 0 to 1 indicating the purity of this quantum state. 1.0 indicates that this density matrix represents a pure quantum state.
`reset`	Resets to the 0 state.
`reverse_qargs`	Returns a `DensityMatrix` with reversed basis state ordering.

Method name	Description
`sample_counts`	Samples the probability distribution a supplied number of times, returning a dictionary of the counts.
`sample_memory`	Samples the probability distribution a supplied number of times, returning a list of the measurement results.
`seed`	Sets the seed for the quantum state random number generator.
`tensor`	Returns the tensor product state of this density matrix and a supplied `DensityMatrix`.
`to_dict`	Returns the density matrix as a dictionary.
`to_operator`	Returns an operator converted from the density matrix.
`to_statevector`	Returns a `Statevector` from a pure density matrix.
`trace`	Return the trace of the density matrix.

Table 5-5. Some `DensityMatrix` attributes

Attribute name	Description
`data`	Contains the complex matrix
`dim`	Contains the number of basis states in the density matrix
`num_qubits`	Contains the number of qubits in the density matrix, or None.

Example of using DensityMatrix methods

As an example of using some of these `DensityMatrix` methods and attributes we'll first create a mixed state by combining two density matrices, each of which is instantiated using the `from_label` method.

```
from qiskit.quantum_info import DensityMatrix, \
                                Operator

dens_mat = 0.5*DensityMatrix.from_label('11') + \
  0.5*DensityMatrix.from_label('+0')
print(dens_mat.data)
```

```
output:
  [[0.25+0.j  0.+0.j  0.25+0.j  0.+0.j]
   [0.+0.j    0.+0.j  0.+0.j    0.+0.j]
   [0.25+0.j  0.+0.j  0.25+0.j  0.+0.j]
   [0.+0.j    0.+0.j  0.+0.j    0.5+0.j]]
```

Next we'll use the `evolve` method to evolve the state with an operator (see "Using Quantum Information Operators" on page 95). We'll then use the `draw` method to visualize the density matrix as the city plot shown in Figure 5-2.

```
tt_op = Operator.from_label('TT')
dens_mat = dens_mat.evolve(tt_op)
dens_mat.draw('city')
```

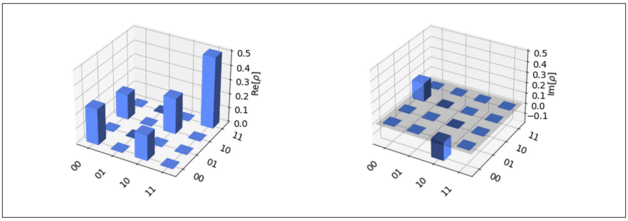


Figure 5-2. Example visualization produced by the `DensityMatrix` `draw` method

Next, we'll use the `probabilities` method to show the probabilities of each basis state being the result of a measurement.

```
print(dens_mat.probabilities())

output:
  [0.25 0.0 0.25 0.5]
```

Finally, we'll use the `sample_counts` method to sample the probability distribution as if the circuit were being measured ~~1000~~ times.

```
print(dens_mat.sample_counts(1000))

output:
  {'00': 240, '10': 256, '11': 504}
```

Using Quantum Information Operators

The `qiskit.quantum_info` module contains a few classes, shown in Table 5-6, that represent quantum information operators.

Table 5-6. Classes that represent operators in the `qiskit.quantum_info` module

Class name	Description
`Operator`	Operator class modeled with a complex matrix
`Pauli`	Multi-qubit Pauli operator
`Clifford`	Multi-qubit unitary operator from the Clifford group
`ScalarOp`	Scalar identity operator class
`SparsePauliOp`	Sparse multi-qubit operator in a Pauli basis representation
`CNOTDihedral`	Multi-qubit operator from the CNOT-Dihedral group
`PauliList`	List of multi-qubit Pauli operators

We'll focus on two most commonly used ~~of these~~, namely the `Operator` and `Pauli` classes.

Using the Operator Class

The `Operator` class represents a quantum information operator, modeled by a matrix. For example, an `Operator` instance was used to evolve the quantum state represented by a `DensityMatrix` in "Example of using DensityMatrix methods" on page 93. Operators are used in many ways, including by being placed into a `QuantumCircuit` with the `append` method discussed in "Using the append() method" on page 13.

An `Operator` may be instantiated in several ways, one of which is by passing in a `QuantumCircuit` instance as shown in the following code snippet~~:~~

```
from qiskit import QuantumCircuit
from qiskit.quantum_info import Operator
```

```
qc = QuantumCircuit(2)
qc.id(0)
qc.x(1)

op_XI = Operator(qc)
print(op_XI.data)

output:
  [[0.+0.j 0.+0.j 1.+0.j 0.+0.j]
   [0.+0.j 0.+0.j 0.+0.j 1.+0.j]
   [1.+0.j 0.+0.j 0.+0.j 0.+0.j]
   [0.+0.j 1.+0.j 0.+0.j 0.+0.j]]
```

Notice that matrix for the operator is the unitary for the circuit. This technique may be used to obtain a unitary for a circuit without running it on a quantum simulator (as shown in the code in "Using the AerSimulator to calculate and hold a unitary" on page 45).

Another way of creating an `Operator` is to pass in the desired complex matrix, as shown in the following listing.

```
from qiskit.quantum_info import Operator

op_XI = Operator([[0, 0, 1, 0],
                  [0, 0, 0, 1],
                  [1, 0, 0, 0],
                  [0, 1, 0, 0]])
print(op_XI.data)

output:
  [[0.+0.j 0.+0.j 1.+0.j 0.+0.j]
   [0.+0.j 0.+0.j 0.+0.j 1.+0.j]
   [1.+0.j 0.+0.j 0.+0.j 0.+0.j]
   [0.+0.j 1.+0.j 0.+0.j 0.+0.j]]
```

Yet another way of creating an `Operator` is to pass a `Pauli` instance (see "Using the Pauli Class" on page 98), as shown in the following listing.

```
from qiskit.quantum_info import Operator, Pauli
```

```
op_XI = Operator(Pauli('XI'))
print(op_XI.data)

output:
  [[0.+0.j 0.+0.j 1.+0.j 0.+0.j]
   [0.+0.j 0.+0.j 0.+0.j 1.+0.j]
   [1.+0.j 0.+0.j 0.+0.j 0.+0.j]
   [0.+0.j 1.+0.j 0.+0.j 0.+0.j]]
```

Notice that all of the three previous examples defined the same operator, as their underlying matrices are identical. An additional way of creating an `Operator` is to pass an `Instruction` or `Gate` object (see "Instructions and Gates" on page 26), as shown in the following code snippet.

```
from qiskit.quantum_info import Operator
from qiskit.circuit.library.standard_gates \
                                  import CPhaseGate

op_CP = Operator(CPhaseGate(np.pi / 4))
print(op_CP.data)

output:
  [[1.+0.j 0.+0.j 0.+0.j 0.+0.j]
   [0.+0.j 1.+0.j 0.+0.j 0.+0.j]
   [0.+0.j 0.+0.j 1.+0.j 0.+0.j]
   [0.+0.j 0.+0.j 0.+0.j 0.70710678+0.70710678j]]
```

The `CPhaseGate` used in the previous example may be seen in "CPhaseGate" on page 160.

Table ~~5-7 and Table 5-8~~ describe some of the methods and attributes in the `Operator` class.

Table 5-7. Some `Operator` methods

Method name	Description
`adjoint`	Returns the adjoint of the operator
`compose`	Returns the result of left-multiplying this operator with a supplied `Operator`~~.~~

Method name	Description
`conjugate`	Returns the complex conjugate of the operator
`copy`	Returns a copy of the `Operator`
`dot`	Returns the result of right-multiplying this operator with a supplied `Operator`
`equiv`	Returns a boolean indicating whether a supplied `Operator` is equivalent to this one, up to a global phase
`expand`	Returns the reverse-order tensor product with another `Operator`
`from_label`	Returns a tensor product of single-qubit operators among the following: ‘I’, ‘X’, ‘Y’, ‘Z’, ‘H’, ‘S’, ‘T’, ‘0’, ‘1’, ‘+’, ‘-‘, ‘r’, and ‘l’
`is_unitary`	Returns a boolean indicating whether this operator is a unitary matrix
`power`	Returns an `Operator` raised to the supplied power
`tensor`	Returns the tensor product with another `Operator`
`to_instruction`	Returns this operator converted to a `UnitaryGate`
`transpose`	Returns the transpose of the operator

Table 5-8. Some `Operator` attributes

Attribute name	Description
`data`	Contains the operator’s complex matrix
`dim`	Contains the dimensions of the operator’s complex matrix
`num_qubits`	Contains the number of qubits in the operator, or None.

Using the Pauli Class

The `Pauli` class represents a multi-qubit Pauli operator in which each qubit is an X, Y, Z, or I Pauli matrix. A `Pauli` may be instantiated in several ways, the most common of which is to pass in a string containing Pauli operators preceded by an optional phase coefficient.

```
from qiskit.quantum_info import Pauli

pauli_piXZ = Pauli('-XZ')
print(pauli_piXZ.to_matrix())

output:
  [[ 0.+0.j  0.+0.j -1.+0.j  0.+0.j]
   [ 0.+0.j  0.+0.j  0.+0.j  1.-0.j]
   [-1.+0.j  0.+0.j  0.+0.j  0.+0.j]
   [ 0.+0.j  1.-0.j  0.+0.j  0.+0.j]]
```

Another way of creating a `Pauli` is to pass in a `QuantumCircuit` instance that contains only Pauli gates (X, Y, Z, I) as shown in the following code snippet.

```
from qiskit import QuantumCircuit
from qiskit.quantum_info import Pauli

qc = QuantumCircuit(2)
qc.z(0)
qc.x(1)

pauli_XZ = Pauli(qc)
print(pauli_XZ.equiv(Pauli('-XZ')))

output:
  True
```

Notice that the previous two examples produced equivalent Pauli operators, as they differ only by a global phase.

Table ~~5-9 and Table 5-10~~ describe some of the methods and attributes in the `Pauli` class.

Table 5-9. Some `Pauli` methods

Method name	Description
`adjoint`	Returns the adjoint of the Pauli

Method name	Description
`commutes`	Returns a boolean indicating whether a supplied Pauli commutes with this one
`compose`	Returns the result of left-multiplying this Pauli with a supplied Pauli.
`conjugate`	Returns the complex conjugate of the Pauli
`copy`	Returns a copy of the Pauli
`dot`	Returns the result of right-multiplying this Pauli with a supplied Pauli
`equiv`	Returns a boolean indicating whether a supplied Pauli is equivalent to this one, up to a global phase
`expand`	Returns the reverse-order tensor product with another Pauli
`inverse`	Returns the inverse of the Pauli
`power`	Returns a Pauli raised to the supplied power
`tensor`	Returns the tensor product with another Pauli
`to_label`	Returns this Pauli converted to string label containing an optional phase, and Pauli gates X, Y, Z, I~~.~~
`to_matrix`	Returns this Pauli as a complex matrix
`transpose`	Returns the transpose of the Pauli

Table 5-10. Some `Pauli` attributes

Attribute name	Description
`dim`	Contains the dimensions of the Pauli's complex matrix
`num_qubits`	Contains the number of qubits in the Pauli, or None
`phase`	Contains an integer that represent the phase of the Pauli

Using Quantum Information Channels

The `qiskit.quantum_info` module contains a few classes, shown in ~~the~~ Table 5-11, that represent quantum information channels.

Table 5-11. Classes that represent channels in the `qiskit.quantum_info` *module*

Class name	Description
`Choi`	Choi-matrix representation of a quantum channel
`SuperOp`	Superoperator representation of a quantum channel
`Kraus`	Kraus representation of a quantum channel
`Stinespring`	Stinespring representation of a quantum channel
`Chi`	Pauli basis Chi-matrix representation of a quantum channel
`PTM`	Pauli Transfer Matrix (PTM) representation of a quantum channel

We'll focus on a representative of these, namely the `Kraus` class, to model a noisy quantum channel whose qubits flip about 10% of the time. In the following code snippet, a `Kraus` instance is created with matrix that models this bit-flip behavior, and appended to a quantum circuit.

```python
from qiskit import QuantumCircuit
from qiskit.quantum_info import Kraus

noise_ops = [np.sqrt(0.9) * np.array([[1, 0],
                                       [0, 1]]),
             np.sqrt(0.1) * np.array([[0, 1],
                                       [1, 0]])]
kraus = Kraus(noise_ops)

qc = QuantumCircuit(2)
qc.append(kraus, [0])
qc.append(kraus, [1])
qc.measure_all()
qc.draw()
```

The resulting circuit is shown in Figure 5-3.

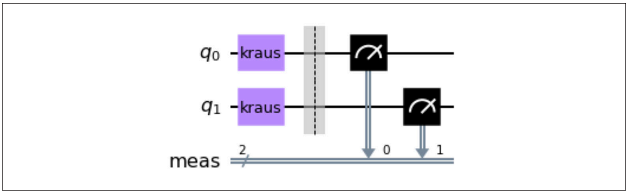


Figure 5-3. Example quantum circuit containing `Kraus` quantum channels

In the following code snippet we'll use an `aer_simulator` (see "Using the AerSimulator to hold measurement results" on page 42) to run the circuit, followed by printing the measurement results.

```
from qiskit import Aer, transpile

backend = Aer.get_backend("aer_simulator")
tqc = transpile(qc, backend)
job = backend.run(tqc, shots=1000)
result = job.result()
counts = result.get_counts(tqc)
print(counts)

output:
  {'11': 8, '01': 90, '00': 818, '10': 84}
```

Notice that the measurements are approximately what would be expected from each qubit having a .1 probability of flipping.

Using Quantum Information Measures

The `qiskit.quantum_info` module contains several functions, shown in ~~the~~ Table 5-12, that return various measurements values.

Table 5-12. Functions that return various measurements values in the `qiskit.quantum_info` module

Function name	Description
`average_gate_fidelity`	Returns the average gate fidelity of a noisy quantum channel
`process_fidelity`	Returns the process fidelity of a noisy quantum channel
`gate_error`	Returns the gate error of a noisy quantum channel
`diamond_norm`	Returns the diamond norm of the input quantum channel object
`state_fidelity`	Returns the state fidelity between two quantum states
`purity`	Returns the purity of a quantum state
`concurrence`	Returns the concurrence of a quantum state
`entropy`	Returns the von-Neumann entropy of a quantum state
`entanglement_of_formation`	Returns the entanglement of formation of quantum state
`mutual_information`	Returns the mutual information of a bipartite state

We'll focus on one representative of these, namely the `state_fidelity` function.

Using the state_fidelity Function

The `state_fidelity()` function takes two `Statevector` or `DensityMatrix` instances and returns the state fidelity between them. In the following code snippet, the state fidelity of a one-qubit statevector in equal superposition has an 85% state fidelity with a statevector whose phase is subsequently rotated by π/8 radians.

```
from qiskit.quantum_info import state_fidelity

sv_a = Statevector.from_label('+')
```

```
sv_b = sv_a.evolve(Operator.from_label('T'))
print(state_fidelity(sv_a, sv_b))

output:
  0.8535533905932733
```

CHAPTER 6

Operator Flow

The `qiskit.opflow` module contains classes for expressing and manipulating quantum states and operations. Some of the functionality is backed by classes in the `qiskit.quantum_info` module. One of the main purposes of this Operator Flow layer is to facilitate the development of quantum algorithms.

Creating Operator Flow Expressions

The `qiskit.opflow` module contains an immutable set of operators, shown in ~~the~~ Table 6-1, that are useful in creating expressions that contain quantum states and operators.

Table 6-1. Immutable operators in the `qiskit.opflow` module

Operator	Description
X	Pauli X
Y	Pauli Y
Z	Pauli Z
I	Pauli I
H	H gate

Operator	Description
`S`	S gate
`T`	T gate
`CX`	CX gate
`CZ`	CZ gate
`Swap`	Swap gate
`Zero`	Qubit 0 state
`One`	Qubit 1 state
`Plus`	Qubit + state
`Minus`	Qubit - state

In order to use these operators in expressions, we'll need algebraic operations and predicates such as shown in Table 6-2.

Table 6-2. Algebraic operations and predicates in the `qiskit.opflow` module

Algebraic operation	Description
`+`	Addition
`-`	Subtraction/negation
`*`	Scalar multiplication
`/`	Scalar division
`@`	Composition
`^`	Tensor product or tensor power
`**`	Composition power
`==`	Equality
`~`	Adjoint

These algebraic operation are syntactic sugar for underlying methods that perform functions such as computing tensor products and multiplying matrices, which allows representing complex formulas with a concise syntax.

Using operators and algebraic operations from ~~Table 6-1 and Table 6-2~~ we can create expressions such as the arbitrary state |10010⟩ in the following code snippet. As with other examples in this book, and in Qiskit as a whole, the least significant qubit is ~~the~~ represented by the rightmost binary digit.

```
from qiskit.opflow import Zero, One

state = One ^ Zero ^ One ^ Zero ^ Zero
print(state)

output:
  DictStateFn({'10100': 1})
```

Notice that the output reveals the class, `DictStateFn`, in which our state is held. We'll explore these classes soon.

We can also create expressions such as the arbitrary Pauli operator with a phase coefficient in the following listing.

```
from qiskit.opflow import X, Z

pauli_piXZ = -(X ^ Z)
print(pauli_piXZ)

output:
  -1.0 * XZ
```

You may recognize this Pauli operator as an example used in "Using the Pauli Class" on page 98. In that example the matrix that models the operator was printed, and we'll do so here to show that it is the same.

```
print(pauli_piXZ.to_matrix())

output:
  [[-0.+0.j -0.+0.j -1.+0.j -0.+0.j]
   [-0.+0.j -0.+0.j -0.+0.j  1.-0.j]
   [-1.+0.j -0.+0.j -0.+0.j -0.+0.j]
   [-0.+0.j  1.-0.j -0.+0.j -0.+0.j]]
```

Now we'll create an Operator Flow expression that represents a GHZ circuit.

```
from qiskit.opflow import I, X, H, CX

op = (CX ^ I) @ (I ^ CX) @ (I ^ I ^ H)
print(op)

output:
        ┌───┐
  q_0: ┤ H ├──■──────
        └───┘┌─┴─┐
  q_1: ──────┤ X ├──■──
             └───┘┌─┴─┐
  q_2: ───────────┤ X ├
                  └───┘
```

Notice that Operator Flow notation is little-endian. For example, the **H** gate in the expression is placed on the least significant qubit wire. Also notice that the control wire of a given **CX** gate is on the lesser index of its qubit wires.

For fun, let's convert the expression to a circuit~~,~~ and sample the probability distribution using a `Statevector` (see "Using the Statevector Class" on page 86).

```
from qiskit.quantum_info import Statevector

qc = op.to_circuit()
sv = Statevector(qc)
print(sv.sample_counts(1000))

output:
  {'000': 482, '111': 518}
```

Underlying this Operator Flow expression syntax is a rich set of classes for representing states, operators, and other constructs. Let's dive into how Operator Flow represents states, with the state function classes.

Using the Operator Flow State Function Classes

The `qiskit.opflow.state_fns` module contains a few classes, shown in ~~the~~ Table 6-3, that represent state functions.

Table 6-3. Some classes that represent state functions in the `qiskit.opflow.state_fns` module

Class name	Description
`StateFn`	Base class and factory for `StateFn` subclasses
`CircuitStateFn`	Represents a state function, backed by a `QuantumCircuit` instance that assumes all-zero qubit inputs. See "Constructing Quantum Circuits" on page 3
`DictStateFn`	Represents a state function, backed by a Python dictionary
`VectorStateFn`	Represents a state function, backed by a `Statevector` class. See "Using the Statevector Class" on page 86
`SparseVectorStateFn`	Contains a sparse representation of a state function
`OperatorStateFn`	Represents a state function, backed by a density operator.

We'll focus on the `StateFn` class, which is the base class and factory for the rest of these classes.

Using the StateFn Class

The `StateFn` class is the base class for all the state function classes in Operator Flow. This class also serves as a factory for these state function classes in Operator Flow. For example, as shown in the following listing, a `DictStateFn` may be instantiated by passing a bit string into a `StateFn`.

```python
from qiskit.opflow.state_fns import StateFn

statefn = StateFn('10100')
print(statefn)
```

```
output:
  DictStateFn({'10100': 1})
```

Notice that this creates an instance of the same class, `DictStateFn`, as when using an Operator Flow expression to create that state function in "Creating Operator Flow Expressions" on page 105.

For another example, as shown in the following listing, a `CircuitStateFn` may be instantiated by passing a `QuantumCircuit` into a `StateFn`.

```
from qiskit import QuantumCircuit

qc = QuantumCircuit(3)
qc.h(0)
qc.cx(0, 1)
qc.cx(1, 2)

statefn = StateFn(qc)
print(statefn)

output:
  CircuitStateFn(
       ┌───┐
  q_0: ┤ H ├──■──────────
       └───┘┌─┴─┐
  q_1: ─────┤ X ├──■─────
            └───┘┌─┴─┐
  q_2: ──────────┤ X ├
                 └───┘
  )
```

This creates an instance of the same class, `CircuitStateFn`, as when using an Operator Flow expression to create a GHZ state in "Creating Operator Flow Expressions" on page 105.

For yet another example, as shown in the following listing, a `VectorStateFn` may be instantiated by passing a list of amplitudes into a `StateFn`.

```
import numpy as np

statefn = StateFn([1, 0, 0, 1] / np.sqrt(2))
print(statefn)

output:
  VectorStateFn(Statevector([
                0.70710678+0.j,
                0.+0.j,
                0.+0.j,
                0.70710678+0.j],
            dims=(2, 2)))
```

Note that this creates an instance of `Statevector`, just as we did with a similar example in "Using the Statevector Class" on page 86.

Table 6-4, ~~Table~~ 6-5 and ~~Table~~ 6-6 describe some of the instantiation parameters, methods and attributes in the `StateFn` class.

Table 6-4. Some `StateFn` instantiation parameters

Parameter name	Description
`primitive`	Determines which of the `StateFn` classes will be created~~,~~ and sets its initial value. Can be either a str, dict, `Result`, list, ndarray, `Statevector`, `QuantumCircuit`, `Instruction`, `OperatorBase`, or None.
`coeff`	Coefficient of this state function
`is_measurement`	If `True`, this state function is to be a bra (row vector) rather than a ket (column vector).

Supplying a `True` `is_measurement` argument is related to using the ~ (adjoint) algebraic operator from Table 6-2. In the following example, the values of `statefn_a` and `statefn_b` are equivalent.

```
from qiskit.opflow.state_fns import StateFn
from qiskit.opflow import One, Zero
```

```
statefn_a = StateFn('100', is_measurement=True)
print('statefn_a:', statefn_a, statefn_a.is_measurement)

statefn_b = ~(One ^ Zero ^ Zero)
print('statefn_b:', statefn_b, statefn_b.is_measurement)

output:
  statefn_a: DictMeasurement({'100': 1}) True
  statefn_b: DictMeasurement({'100': 1}) True
```

Note that the `DictMeasurement` in the output indicates that `is_measurement` is `True`.

Table 6-5. Some `StateFn` methods

Method name	Description
`add`	Returns the addition of a supplied `StateFn` to this one. This is equivalent to using the + algebraic operator in Table 6-2.
`adjoint`	Returns the adjoint (complex conjugate) of this `StateFn`. This is equivalent to using the ~ algebraic operator in Table 6-2.
`equals`	Returns a boolean that indicates whether the supplied `StateFn` is equal to this one up to global phase. This is equivalent to using the == algebraic operator in Table 6-2.
`eval`	Evaluate underlying function of this `StateFn`.
`mul`	Returns the scalar multiplication of a supplied number to this `StateFn`. Number should be a valid `int`, `float`, complex, or `Parameter` instance. This is equivalent to using the * algebraic operator in Table 6-2.
`primitive_strings`	Return a set of strings describing the primitives contained in this `StateFn`.
`sample`	Samples the normalized probability distribution of this `StateFn` a supplied number of shots, and returns a dictionary with the results.

Method name	Description
`tensor`	Returns the tensor product of this `StateFn` with a supplied `StateFn`. This is equivalent to using the ^ algebraic operator in Table 6-2.
`tensorpower`	Returns the tensor product of this `StateFn` with itself a supplied number of times, represented as an `int`. This is equivalent to using the ^ algebraic operator in Table 6-2.
`to_circuit_op`	Returns a `CircuitOp` that is equivalent to this `StateFn`.
`to_den sity_matrix`	Return a matrix representing the product of `StateFn` evaluated on pairs of basis states.
`to_matrix`	Returns the NumPy representation of this `StateFn`.
`to_matrix_op`	Returns a `VectorStateFn` for this `StateFn`.

Table 6-6. Some `Statevector` attributes

Attribute name	Description
`coeff`	Coefficient of this state function
`is_measure ment`	If `True`, this state function represents a bra (row vector) rather than a ket (column vector).
`num_qubits`	Contains the number of qubits in the state function.
`primitive`	Which of the `StateFn` classes implements the behavior of this state function.

Let's turn our attention how Operator Flow represents operators, with the primitive operators classes.

Using the Operator Flow Primitive Operators classes

The `qiskit.opflow.primitive_ops` module contains a few classes, shown in Table 6-7, that represent primitive operators.

Table 6-7. Some classes that represent primitive operators in the `qiskit.opflow.primitive_ops` module

Class name	Description
`PrimitiveOp`	Base class and factory for `PrimitiveOp` subclasses.
`CircuitOp`	Represents a quantum operator, backed by a `QuantumCircuit` instance. See "Constructing Quantum Circuits" on page 3.
`MatrixOp`	Represents a quantum operator, backed by an `Operator` instance. See "Using the Operator Class" on page 95.
`PauliOp`	Represents a quantum operator, backed by a `Pauli` class. See "Using the Pauli Class" on page 98.

We'll focus on the `PrimitiveOp` class, which is the base class and factory for the rest of these classes.

Using the PrimitiveOp Class

The `PrimitiveOp` class is the base class for all of the primitive operator classes in Operator Flow. This class also serves as a factory for these classes. For example, as shown in the following listing, a `PauliOp` may be instantiated by passing `Pauli` instance into a `PrimitiveOp`.

```
from qiskit.opflow.primitive_ops import PrimitiveOp
from qiskit.quantum_info import Pauli

primop_piXZ = PrimitiveOp(Pauli('-XZ'))
print(primop_piXZ)
print(type(primop_piXZ))

output:
  -XZ
  <class '...PauliOp'>
```

Notice from the following code that this creates an instance of the same class, `PauliOp`, as when using an Operator Flow expression to create that primitive operator (see "Creating Operator Flow Expressions" on page 105). Also notice that the underlying primitives (the `Pauli` instances) are equivalent up to a global phase, but that they are not equal, given that they are of different types.

```
from qiskit.opflow import X, Z

pauli_piXZ = -(X ^ Z)
print(type(pauli_piXZ))
print(primop_piXZ.primitive
      .equiv(pauli_piXZ.primitive))

output:
  <class '...PauliOp'>
  True
```

For another example, as shown in the following listing, a `CircuitOp` may be instantiated by passing a `QuantumCircuit` into a `PrimitiveOp`.

```
from qiskit import QuantumCircuit

qc = QuantumCircuit(3)
qc.h([0,1,2])

h_primop = PrimitiveOp(qc)
print(h_primop)
print(type(h_primop))

output:
       ┌───┐
  q_0: ┤ H ├
       ├───┤
  q_1: ┤ H ├
       ├───┤
  q_2: ┤ H ├
       └───┘
  <class ...CircuitOp'>
```

This creates an instance of the same class, `CircuitOp`, as when using an Operator Flow expression to create the same circuit as shown in the following code.

```
from qiskit.opflow import H

hgates = H^3
print(hgates)
print(type(hgates))

output:
       ┌───┐
  q_0: ┤ H ├
       ├───┤
  q_1: ┤ H ├
       ├───┤
  q_2: ┤ H ├
       └───┘
  <class ...CircuitOp'>
```

Table 6-8, ~~Table~~ 6-9 and ~~Table~~ 6-10 describe some of the instantiation parameters, methods and attributes in the `PrimitiveOp` class.

Table 6-8. Some `PrimitiveOp` instantiation parameters

Parameter name	Description
`primitive`	Determines which of the `PrimitiveOp` classes will be created~~,~~ and sets its initial value. Can be either a `QuantumCircuit`, `Operator`, `Pauli`, `SparsePauliOp` or `OperatorBase`.
`coeff`	Coefficient of this primitive operator

Table 6-9. Some `PrimitiveOp` methods

Method name	Description
`add`	Returns the addition of a supplied `PrimitiveOp` to this one. This is equivalent to using the + algebraic operator in Table 6-2.
`adjoint`	Returns the adjoint (complex conjugate) of this `PrimitiveOp`. This is equivalent to using the ~ algebraic operator in Table 6-2.

Method name	Description
`compose`	Returns the operator composition of a supplied `PrimitiveOp` to this one. This is equivalent to using the `@` algebraic operator in Table 6-2.
`equals`	Returns a boolean that indicates whether the supplied `PrimitiveOp` is equal to this one. This is equivalent to using the == algebraic operator in Table 6-2.
`eval`	Evaluate underlying function of this `PrimitiveOp`.
`exp_i`	Returns the `PrimitiveOp` exponentiation.
`mul`	Returns the scalar multiplication of a supplied number to this `PrimitiveOp`. This is equivalent to using the `*` algebraic operator in Table 6-2.
`primitive_strings`	Returns a set of strings describing the primitives contained in this `PrimitiveOp`.
`tensor`	Returns the tensor product of this `PrimitiveOp` with a supplied `PrimitiveOp`. This is equivalent to using the ^ algebraic operator in Table 6-2.
`tensorpower`	Returns the tensor product of this `PrimitiveOp` with itself a supplied number of times. This is equivalent to using the ^ algebraic operator in Table 6-2.
`to_circuit_op`	Returns a `CircuitOp` that is equivalent to this `PrimitiveOp`.
`to_instruction`	Returns an `Instruction` equivalent to this `PrimitiveOp`.
`to_matrix`	Returns the NumPy representation of this `PrimitiveOp`.
`to_matrix_op`	Returns a `MatrixOp` for this `PrimitiveOp`.

Table 6-10. Some `PrimitiveOp` attributes

Attribute name	Description
`coeff`	Coefficient of this primitive operator~~.~~
`num_qubits`	Contains the number of qubits in the primitive operator~~.~~

Attribute name	Description
`primitive`	Which of the `PrimitiveOp` classes implements the behavior of this primitive operator.

Quantum Algorithms

Much like many sophisticated classical algorithms, in the future, we’d like to be able to run a quantum algorithm without knowing all the details about it~~’~~s implementation. Qiskit supports many popular quantum algorithms out of the box. You can simply specify a problem, then choose an algorithm to solve it. In this chapter, we’ll explore Qiskit’s algorithms module~~,~~ and the algorithms this module supports.

Background on Quantum Algorithms

Quantum algorithms are the motivation for most research and investment in quantum computing. The entire fields of quantum error correction, quantum hardware, and quantum software development (including Qiskit) ultimately work toward~~s~~ the common goal of running a useful algorithm on a quantum computer.

With this in mind, you might find it surprising that there are relatively few problems for which we think we could achieve ~~“~~quantum advantage~~”~~ (where a quantum computer outperforms modern classical computers). Finding new quantum algorithms, and new ways to apply known algorithms is a very active area of research. Maybe even more surprising is that, of these candidate quantum algorithms, we’re not actually sure some will have a speedup at all (never mind on huge, fault-tolerant computers). But why is this?

To guarantee a speedup over classical methods, we need to be able to directly compare the quantum algorithm to its classical counterpart, and to make a direct comparison, both algorithms must solve exactly the same problem. One consequence of this is that both algorithms must take classical data as an input~~,~~ and return classical data as an output. Some famous algorithms (e.g. Harrow, Hassidim, and Lloyd's algorithm, known as the ~~“~~HHL algorithm~~”~~) take/return quantum superpositions as inputs and/or outputs, ~~so~~ can't be directly compared to classical algorithms.

While still interesting in their own right, ~~we can view~~ algorithms with quantum inputs/outputs as building~~-~~blocks, used as subroutines in other algorithms that *do* use classical inputs and outputs. Examples include Brassard, Høyer and Tapp's quantum counting algorithm, which uses phase estimation as a subroutine, and Kerenidis and Prakash's recommendation algorithm, which they based on the HHL algorithm (more on this later).

If the quantum algorithm solves a classical problem, you can start to analy~~s~~e how it behaves~~,~~ and use this to compare it to the best-known classical algorithm. For example, we know Shor's algorithm grows significantly slower than its best-known classical competitor, the general number field sieve, and this result is the reassurance many people needed to invest time and money into building quantum computers.

So far, we have been talking about comparisons to the “best-known” classical algorithms. The final step in proving our quantum computer will *definitely* outperform a classical computer is to prove that no classical algorithm could possibly scale better than our quantum algorithm. As you might imagine, this is very hard to do.

Sometimes, the fact that many people have tried and failed to find an efficient classical algorithm counts as enough evidence that the quantum competitor is worth investing in, but the future sometimes surprises us. ~~Above~~, we mentioned Kerenidis and Prakash's recommendation algorithm, which was exponentially faster than the best-known classical algorithm for the same problem. Only three years later, Ewin Tang found a classical algorithm that was only polynomially slower than the quantum algorithm.

At the time of writing, we can already run simple quantum algorithms on real quantum hardware, and we are rapidly approaching the ability to test quantum algorithms empirically, instead of measuring their performance theoretically. As with classical computing, we will need to consider implementation details (not just the algorithm complexity) to ensure we get the best performance.

Using the Algorithms Module

All algorithm interfaces in Qiskit's algorithms module follow a consistent pattern. In this section, we'll learn about this general pattern~~,~~ and the rationale behind it.

Quickstart

First, let's see how we run a simple algorithm using Qiskit.

The code in the snippet ~~below~~ uses Qiskit's algorithms module to run Shor's algorithm on a simulator.

```
# choose a backend to use
from qiskit.providers.aer import AerSimulator
aer_sim = AerSimulator()

# create an instance of Shor's algorithm, using
# our backend
from qiskit.algorithms import Shor
shor = Shor(aer_sim)

# execute algorithm on specific problem and
# view the result
```

```
result = shor.factor(21)
result.factors  # Has value: [[3, 7]]
```

We've split the ~~above~~ code snippet into three steps:

1. First, we need to choose a backend to run the algorithm on. Here, we've chosen the `AerSimulator`.
2. Next, we create an instance of Shor's algorithm, using the backend.
3. We then run Shor's algorithm on the input `21`~~,~~ and view the results.

We can see that the `factors` attribute of `result` does contain the correct factors of 21.

The Algorithms Interface

When testing and researching different quantum algorithms, we want to be able to compare the performance of different algorithms (and variations on these algorithms) against the same problem.

The input to a factoring problem is always an integer, so Qiskit uses a Python `int` to represent a factoring problem. Other algorithms (e.g. amplitude amplification and amplitude estimation) have their own problem classes (e.g. `AmplificationProblem` and `EstimationProblem`) that the algorithms will try to solve.

We can then compare the performance of different algorithms on these problem objects. For example, if we create an `EstimationProblem` object, Qiskit offers four different quantum algorithms to solve it: `AmplitudeEstimation`, `FasterAmplitudeEstimation`, `IterativeAmplitudeEstimation`, and `MaximumLikelihoodAmplitudeEstimation`. For other problems, Qiskit even incorporates some classical algorithms, such as `NumPyEigensolver`, `NumPyLinearSolver`, and `NumPyMinimumEigensolver`, so we can compare their results and performance.

If we view a quantum algorithm as a method of solving real world problems, then we must also consider implementation details (e.g. the backend it runs on) as part of that method. When we construct the algorithm, we can specify the backend, as well as other device-specific implementation details, such as the transpiler `optimization_level`. These properties live inside a `QuantumInstance` object. For example, let's say we're using the default `AerSimulator`, which has no errors. This means we don't need to bother optimizing the circuits when we transpile them. In the code snippet ~~below~~, we create a `QuantumInstance` with the `optimization_level` set to `0`.

```
# choose a backend to use
from qiskit.providers.aer import AerSimulator
aer_sim = AerSimulator()

# construct the QuantumInstance with no
# optimization
from qiskit.utils import QuantumInstance
quantum_instance = QuantumInstance(
    aer_sim,
    optimization_level=0,
)
```

We can then use the code snippet ~~below~~ to construct and run Shor's algorithm using our `QuantumInstance` instead of a backend object.

```
from qiskit.algorithms import Shor
shor = Shor(quantum_instance)
result = shor.factor(15)
result.factors  # Has value: [[3, 5]]
```

As well as backend-specific parameters, we can also change algorithm-specific parameters. For example, the `FasterAmplitudeEstimation` algorithm class needs two parameters, one to specify the acceptable error, and another to specify the maximum number of iterations allowed.

Traditional Quantum Algorithms

In this section, we'll cover the more traditional quantum algorithms in Qiskit's algorithms module~~,~~ and give a short example of each algorithm in action.

Grover's Algorithm

Grover's algorithm is one of the most famous quantum algorithms. Grover's is one of the few quantum algorithms that we can prove scales better than any possible classical algorithm, and it's actually provably optimal for quantum algorithms too. Figure 7-1 shows a very high-level Grover circuit.

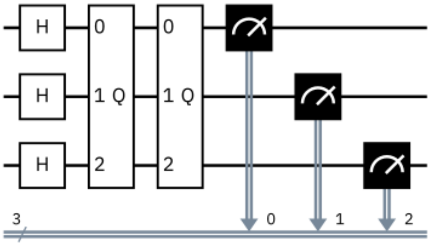


Figure 7-1. High-level example of Grover's algorithm~~. The 'Q' gate is the Grover operator (discussed later in this section)~~.

Grover's algorithm solves specific case of the *amplification problem*: ~~G~~iven two operators, A_θ and B_θ, that rotate around the states $|A\rangle$ and $|B\rangle$, create a circuit that transforms $|A\rangle$ into $|B\rangle$. Grover's specific case is where $|A\rangle$ is the superposition of all computational basis states~~,~~ and $|B\rangle$ is a specific computational basis state.

If we know how to create a program to check a solution to a problem, it's relatively straightforward to create a circuit that transforms around that solution's computational basis state (i.e. it's straightforward to create B_θ). This makes Grover's algorithm very widely applicable.

To use Grover's algorithm, we first need to specify the problem, we do ~~this~~ via the `AmplificationProblem` class. The `AmplificationProblem` constructor requires two arguments: the `oracle`, which is the `QuantumCircuit` that carries out the operator B_θ, and the `is_good_state` function, which takes a bit string and returns `True` if it's a solution.

In the code snippet ~~below~~, we use the `PhaseOracle` class from Qiskit's circuit library to create an oracle from a simple Boolean expression.

Let's imagine two parents, A and B, and their child, C, have two tickets for a play. Each person can either (0) not go to the play, or (1) go to the play. At least one adult needs to go, so we have the requirement `(A | B)`, where `|` is a Boolean `OR`. The other problem is that there are only two tickets, so we can't have all three going together. In our notation, this is `~(A & B & C)`, where `~` is the Boolean `NOT` and `&` is the Boolean `AND`. Finally, C particularly wants to go with B as C doesn't see B much during the week, so we have the added constraint of `(B & C)`. Can we satisfy all these constraints?

You may have already worked out the answer is *yes*, and that this problem only has one solution: C and B go, and A doesn't. If we convert this to bits, the solution is the string `110`, where A is the least significant bit.

```python
# create an oracle using a Boolean expression
from qiskit.circuit.library import PhaseOracle
oracle = PhaseOracle(
        '(A | B)'    # A must go if B doesn't
        '& ~(A & B & C)'  # can't all go
        '& (B & C)'  # C wants to go with B
        )
```

Figure 7-2 shows the result of `oracle.draw()`.

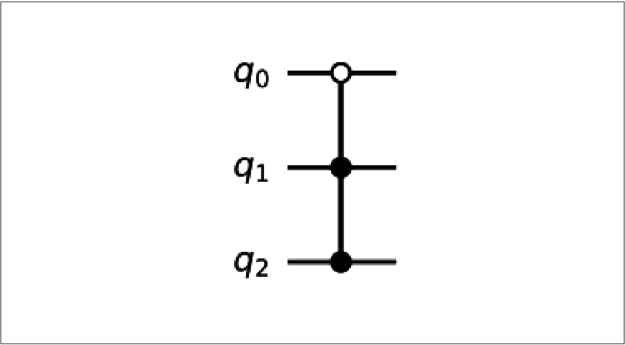


*Figure 7-2. Qiskit-generated Grover oracle for the example problem ~~above~~. The PhaseOracle constructor has compiled this to a simple diagonal gate that adds a phase of -1 to the state **110**. More difficult problems can still be compiled to oracles in polynomial time~~,~~ but won't be as easy to solve by inspection.*

In the code snippet ~~below~~, we create an `Ampli~~t~~ficationProblem` from our `PhaseOracle`. Conveniently, the `PhaseOracle` class has an `evaluate_bitstring` method, which `AmplificationProblem` knows to use as the `is_good_state` parameter, so we don't need to specify that.

```
from qiskit.algorithms import AmplificationProblem
problem = AmplificationProblem(oracle)
```

By default, the `AmplificationProblem` class defaults to a Grover's specific case, but we can set parameters to program other cases.

- The `state_preparation` argument takes a quantum circuit that prepares the state $|A\rangle$. If not specified, this defaults to a H-gate on each qubit.
- The `grover_operator` argument takes the circuit that performs $A_\theta B_\theta$. If not specified, Qiskit constructs this from the oracle and `state_preparation` circuit.

- The post_processing argument takes a callable Python function that Qiskit will apply to the top measured bit string before writing to the assignment (note this function is not called before passing bit strings to is_good_state).
- The objective_qubits argument takes a list of integers, which specifies the indexes of the qubits that contain the solution bit string. This is useful if your oracle uses auxiliary qubits that the diffuser and measurements should ignore.

Now we have our family dynamic problem encoded properly, we can then use Grover's algorithm to solve it. As with all algorithms, we first need to choose the backend to use. In the code snippet ~~below~~, we use the AerSimulator. Once we've constructed the Grover object, we can use it to solve the AmplificationProblem using the amplify() method, which returns a GroverResult object. From this GroverResult object, we can get the output bit string (plus any post-processing) via the assignment attribute.

Note that~~,~~ since we're happy with the default settings, we can skip creating a QuantumInstance and pass our backend straight to Grover, which will create this for us.

```
# choose backend to use
from qiskit.providers.aer import AerSimulator
aer_sim = AerSimulator()

# use Grover's algorithm to solve the oracle
from qiskit.algorithms import Grover
grover = Grover(quantum_instance=aer_sim)
result = grover.amplify(problem)
result.assignment  # Has value '110'
```

In the code snippet ~~above~~, the algorithm decided the most likely solution was 110, as we expected. Depending on the backend used, we can also access other data such as:

- `circuit_results`: The raw, unprocessed results of the circuit execution (can be a `Counts` dictionary~~,~~ or `Statevector`).
- `top_measurement`: The most frequently measured bit string.
- `max_probability`: The probability of measuring the most probable bit string.
- `iterations`: Since we might not know how many solutions there are beforehand, the algorithm tries out different powers of Grover iterations, checking the results using the `is_good_state` function. This value is a list of all the powers tried.

Figure 7-3 shows the Grover operator that Qiskit generates from `oracle`, ~~you~~ can access this via `problem.grover_operator`.

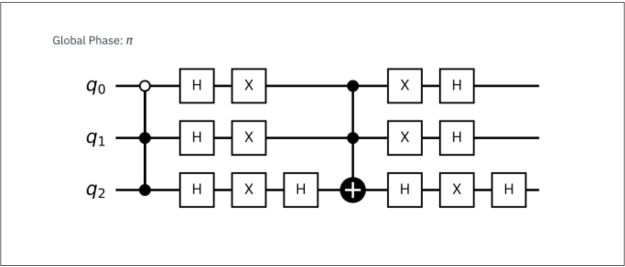


Figure 7-3. Qiskit-generated Grover operator

Phase Estimation Algorithms

Say we have a unitary circuit, Q, and a quantum state, $|\psi\rangle$. We're guaranteed that $|\psi\rangle$ is an eigenstate of Q, i.e.:

$$Q|\psi\rangle = e^{2\pi i\theta}|\psi\rangle$$

The phase estimation problem is to work out the value of θ.

Qiskit provides three different phase estimation algorithms `PhaseEstimation`, `HamiltonianPhaseEstimation`, and `IterativePhaseEstimation`.

`PhaseEstimation` is the classic textbook phase estimation algorithm. It uses two registers, one for the state $|\psi\rangle$, and another "evaluation" register to record the phase Q introduces. The more evaluation qubits, the higher the precision of the output (and the longer the circuit). The algorithm then uses the inverse quantum Fourier transform to read the evaluation register in the computational basis. Figure 7-4 shows an example of this algorithm estimating the phase the T-gate introduces to the state $|1\rangle$.

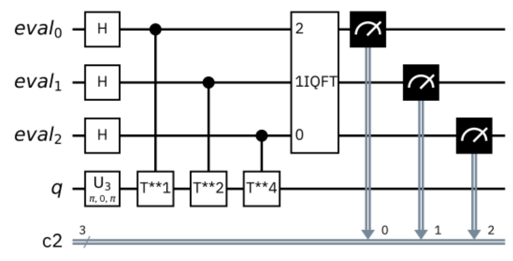


Figure 7-4. A simple phase estimation circuit that estimates the phase the T-gate introduces onto the state 1.

Each of Qiskit's phase estimation algorithms has an `estimate()` method, which takes a unitary circuit (or other operator), and a circuit that prepares an initial state. The code ~~below~~ shows a simple example for the T-gate, and the state $|1\rangle$.

```
from qiskit.algorithms import PhaseEstimation
from qiskit.test.mock import FakeSantiago
from qiskit import QuantumCircuit
santiago = FakeSantiago()

# We will first define the problem:
```

```
# Our unitary (Q) will be the T-gate
unitary = QuantumCircuit(1)
unitary.t(0)

# Our state (|psi>) will be |1>
state_prep = QuantumCircuit(1)
state_prep.x(0)

# Construct our algorithm instance. We will use
# a simulated Santiago device, and 3 evaluation
# qubits
phase_estimator = PhaseEstimation(3, santiago)

# Next, run this algorithm on our input problem
result = phase_estimator.estimate(unitary,
                                  state_prep)

# Finally, access the result
result.phase  # has value: 0.125
```

The `estimate()` method returns a `PhaseEstimationResult` object, which uses the circuit measurements to guess the most likely phase~~,~~ and returns a `float`. As with the other algorithms' `Result` objects, we can access more than just the most likely answer. The `PhaseEstimationResult` class has these attributes and methods:

- The `circuit_result` attribute contains the `Result` object from the job run on the backend.
- The `phases` attribute contains a dictionary where the keys are measured bit strings~~,~~ and the values are the probability of measuring those bit strings.
- The `filter_phases()` method returns the result of the `phases` attribute~~,~~ but with the keys converted from raw bit strings to decimal phases.

`HamiltonianPhaseEstimation` is essentially a wrapper for the `PhaseEstimation` class we explored ~~above~~. Instead of a unitary circuit, `HamiltonianPhaseEstimation.estimate()` takes a~~n~~ Hermitian operator (as well as a state preparation circuit). The algorithm then scales and exponentiates the operator, then runs `PhaseEstimation` on it. `HamiltonianPhaseEstimation.estimate()` has some other optional parameters:

`evolution`
: A convertor to transform the Hermitian operator to a unitary matrix. If unset, then the algorithm uses `PauliTrotterEvolution`.

`bound`
: This value limits the magnitude of the operator's eigenvalues, with tighter bounds resulting in better result precision.

`IterativePhaseEstimation` This algorithm is the same as `PhaseEstimation`~~,~~ but instead uses multiple circuits to reduce the evaluation register to just one qubit. You can use the constructor in the same way as the `PhaseEstimation` class. Here, the integer determines the number of iterations, instead of the number of evaluation qubits, but the end result is that both eventually determine the precision of the output phase.

Amplitude Estimation Algorithms

The *amplitude estimation* problem is very similar to amplitude amplification, but instead of trying to map one state to another, the amplitude estimation problem asks what the inner product of those two states ~~are~~. For example, given an operator that prepares the state $|a\rangle$, and an operator that rotates around $|b\rangle$, find the value of $\langle a|b\rangle$.

Also like Grover's algorithm, we can easily create an *amplitude estimation* problem from a *counting problem*: ~~G~~iven a Boolean function, f, that takes an n-bit string as input~~,~~ and returns a single bit as output, the counting problem asks us for the *number* of bit strings for which f will output 1. For this special case of amplitude estimation, the state $|a\rangle$ is the superposition of all computational basis states, and we can create the operator that rotates around $|b\rangle$ from f using phase kickback.

The EstimationProblem class defines an amplitude estimation problem. The only positional arguments are the state preparation circuit (state_preparation)~~,~~ and a list of the qubits to operate on (objective_qubits). We should also provide a grover_operator for our algorithm to perform phase estimation on. Figure 7-5 shows an example of a circuit that performs phase estimation on a Grover operator, ~~‘~~Q’.

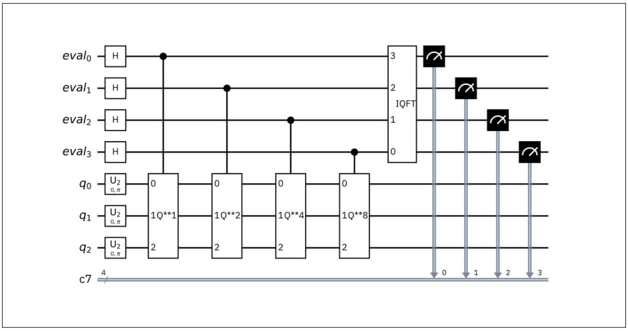


Figure 7-5. Example of a circuit for amplitude estimation~~. The ‘Q’ gate is the Grover operator for our problem.~~

In the code snippet ~~below~~, we’ll create an EstimationProblem from a Boolean expression we created in the Grover’s algorithm section.

```
from qiskit import QuantumCircuit
from qiskit.circuit.library import (PhaseOracle,
                                    GroverOperator)
from qiskit.algorithms import EstimationProblem
oracle = PhaseOracle('(A | B) & ~(A & B & C)'
                     '& (B & C)')

grover_op = GroverOperator(oracle)

# create state preparation operator
n = oracle.num_qubits
state_prep = QuantumCircuit(n)
state_prep.h(range(n))
```

```
    problem = EstimationProblem(state_prep,
                                [*range(n)],
                             grover_operator=grover_op)
```

Now we've defined our problem, let's use an algorithm to solve it. First we'll use Qiskit's `AmplitudeEstimation` algorithm. This is the original amplitude estimation algorithm that performs phase estimation on a Grover operator. In the code snippet ~~below~~, we create an `AmplitudeEstimation` instance with 9 counting qubits.

We happen to know already that this problem uses three bits (and so `oracle.num_qubits == 3`)~~,~~ and has one solution, so we expect the result to be $1/2^3 = 0.125$.

```
from qiskit.algorithms import AmplitudeEstimation
from qiskit.providers.aer import AerSimulator
aer_sim = AerSimulator()

# Create algorithm with 9 counting qubits
estimator = AmplitudeEstimation(9,
                        quantum_instance=aer_sim)
result = estimator.estimate(problem)
result.estimation  # has value: 0.1254318
```

We can see the value of `result.estimation` is what we expected.

Eigensolvers

An eigensolver is an algorithm that finds the eigenvalues (and/or eigenvectors) of a matrix. Since classical computers can solve eigenvalue problems in time polynomial with the size of the input matrix, the difficulty is when we want to solve polynomial sums of Pauli operators that result in exponentially large matrices.

For example, let's use Qiskit's `opflow` module to create a simple operator:

```
from qiskit.opflow import X, Y, Z, I

op = ( .5 * (X ^ Y ^ Z)
     + .2 * (Y ^ Y ^ I)
     - .3 * (Z ^ X ^ Z)
     + .2 * (I ^ X ^ Y))
op.to_matrix().size  # has value 64
```

We can see that the size of this operator's matrix is much larger than the number of operator terms

When most of the traditional quantum algorithms were developed, quantum computers were non-existent, and we didn't even know what they would look like. The only concern was asymptotic scaling, the specific gate count was unimportant. These early pioneers showed that investing in quantum computing would be worth the effort *eventually*, but now we have certain some small, working devices, another important question is "what can we do that might be useful *soon*?"

Qiskit implements a few near-term algorithms (algorithms with small numbers of qubits, and lower gate fidelities in mind). At the time of writing, these are all types of minimum eigensolvers, i.e., eigensolvers that ~~only~~ find the smallest eigenvalue.

NumPy Eigensolvers

At the time of writing, Qiskit ~~only~~ provides one algorithm to find all eigenvalues of an operator: The classical `NumPyEigensolver`. The code ~~below~~ shows how to use the `NumPyEigensolver` to find all the eigenvalues of `op`.

```
from qiskit.algorithms import NumPyEigensolver
np_solver = NumPyEigensolver(k=10)
result = np_solver.compute_eigenvalues(op)
print(result.eigenvalues.real)
```

Which prints the output:

```
[-0.89442719 -0.89442719 -0.2        -0.2
  0.2         0.2         0.89442719  0.89442719]
```

As with all algorithms, we start by creating an instance of the algorithm through the `NumPyEigensolver` constructor. This constructor takes two optional arguments:

- `k`: The number of eigenvalues to compute. This is `1` by default, which is also the minimum value (otherwise it wouldn't need to compute anything). In the example ~~above~~, we set this to 10, which is higher than the dimension of the matrix, so we got all ~~8~~ eigenvalues.
- `filter_criterion`: This is a callable object that takes three parameters (an eigenstate, that state's eigenvalue, and a tuple containing the mean ~~&~~ standard deviation (called `aux_values`), and returns `True` if we want to keep this eigenstate/value, or `False` to ignore it.

We can then use the `compute_eigenvalues` method to execute the algorithm, giving the operator as a positional parameter. This method returns an `EigensolverResult` object, which has three attributes:

- `eigenvalues`
- `eigenstates`
- `aux_operator_eigenvalues` (tuples of the mean ~~&~~ standard deviations for each eigenvalue, for algorithms with some uncertainty)

The code ~~below~~ shows a different instance of the `NumPyEigensolver` with different constructor arguments applied to the ~~same~~ problem ~~as above.~~

```
def ignore_negative(state, value, aux):
    return value >= 0  # bool

np_solver = NumPyEigensolver(k=3,
                             filter_criterion=ignore_negative)
result = np_solver.compute_eigenvalues(op)
result.eigenvalues.real  # [0.2, 0.2, 0.89442719]
```

Some quantum systems (e.g., molecules) are very difficult to simulate with classical computers, i.e., we don't have polynomial-time classical algorithms for simulating them. Despite this, we still find these systems in nature, so they must be solvable at least by a universal quantum computer. This quantum simulation problem was one of the earliest proposed applications of programmable quantum computers~~,~~ and is believed to be one of the more realistic near-term applications of quantum computing.

The problem of quantum simulation boils down to solving the Schrödinger equation for a specific Hamiltonian, which is a description of how the quantum system evolves with time. The eigenvalues of a Hamiltonian are the possible energies the system can have. We can convert a Hamiltonian into a matrix (which must be Hermitian, as the energy is a real number)~~,~~ and then use an eigensolver to find the allowed energies of the system. If we can write a Hamiltonian as a polynomially~~-~~sized sum of Pauli operators, then we can simulate this Hamiltonian efficiently on our quantum computer~~,~~ but not necessarily on a classical computer.

Since systems are usually more stable at their lower energy levels, the lowest possible energy of a system is often the most interesting. Qiskit's quantum eigensolvers are all minimum eigensolvers~~,~~ that ~~only~~ aim to find this smallest energy eigenvalue.

The next algorithm we'll look at is the `NumPyMinimumEigensolver`, which is the same algorithm as the `NumPyEigensolver`~~,~~ but ~~only~~ returns the lowest eigenvalue/vector. We can use this algorithm to check the accuracy of our quantum algorithms for relatively small matrices.

```
from qiskit.algorithms import NumPyMinimumEigensolver
np_min_solver = NumPyMinimumEigensolver()
result = np_min_solver.compute_minimum_eigenvalue(op)
result.eigenvalue.real  # -0.8944271909999164
```

As with the `NumPyEigensolver` shown in the ~~E~~igensolvers section, we can also provide an optional `filter_criterion` function to ignore certain eigenvalues/states. The returned result object also has `eigenstate`~~,~~ and `aux_operator_eigenvalues` attributes.

The Variational Quantum Eigensolver

Next, we'll look at the famous variational quantum eigensolver. This algorithm uses the fact that quantum computers can perform Hamiltonians efficiently, and uses this to calculate the expectation value of the Hamiltonian. The lowest possible expectation value we can measure will be the lowest eigenvalue of the Hamiltonian (when the state is it~~'~~s corresponding eigenstate).

The variational algorithms use a parameterized quantum circuit to prepare different quantum states. The algorithm measures the expectation values of these states, then uses a classical optimizer to try and find the lowest expectation value (that will hopefully also be the lowest eigenvalue).

We can create an instance of this algorithm using the `VQE` class. The constructor has two required arguments: ~~T~~he parameterized circuit~~,~~ and the backend we'll run the algorithm on. For this simple example, we'll use `EfficientSU2` from the circuit library~~,~~ and the `AerSimulator`'s `statevector` method~~.~~

```
from qiskit.providers.aer import AerSimulator
from qiskit.algorithms import VQE
from qiskit.circuit.library import EfficientSU2

circuit = EfficientSU2()
vqe = VQE(circuit,
          quantum_instance=AerSimulator(
                      method='statevector')
         )
result = vqe.compute_minimum_eigenvalue(op)
result.eigenvalue.real  # -0.8944268580187336
```

The `compute_minimum_eigenvalue` method returns a result object with an `eigenvalue` attribute. Comparing this with the `NumPyMinimumEigensolver` result, we can see the algorithm has found the correct minimum eigenvalue. The `MinimumEigensolverResult` object returned by the VQE algorithm also has some other attributes, ~~for example~~:

```
result.cost_function_evals  # has value: 477
```

Shows the number of times the algorithm measured the expectation value of the operator. The result object also contains the circuit parameters that create this best eigenstate (`optimal_parameters`, or `optimal_point`, depending on if you want a dictionary or a list)~~,~~ and the time taken by the algorithm (`optimizer_time`).

The `VQE` constructor also takes other optional arguments. One useful argument is the `callback` argument, which lets us call custom code at each step of the optimization. This argument takes a callable that has four positional arguments:

- Evaluation count: The number of steps taken so far in the optimization
- Parameters: The parameters of the parameterized circuit at this point in the optimization. If everything's going well, this will usually be the best-known parameters so far.
- Mean: This is the estimated expectation value at this point in the optimization.
- Standard deviation: The standard deviation of the distribution averaged to find the mean.

The code ~~below~~ creates a simple class with a method that accepts these values~~,~~ and stores some of them for analysis afterward~~s~~. You might also use a callback to print updates throughout the optimization~~.~~

```
class VQELog():
    def __init__(self):
        self.counts = []
        self.params = []
        self.means = []
    def callback(self, eval_count, params, mean, std_dev):
        self.counts.append(eval_count)
        self.params.append(params)
        self.means.append(mean)
```

In the code ~~below~~, we run VQE again with the callback~~,~~ and use this info to draw a graph showing how the algorithm progresses with each step.

```
log = VQELog()
vqe = VQE(circuit,
          callback=log.callback,
          quantum_instance=AerSimulator(
                    method='statevector')
          )
result = vqe.compute_minimum_eigenvalue(op)
result.eigenvalue.real  # -0.8944268580187336

import matplotlib.pyplot as plt
plt.plot(log.counts, log.means);
```

Figure 7-6 shows the plot created by the code ~~above~~.

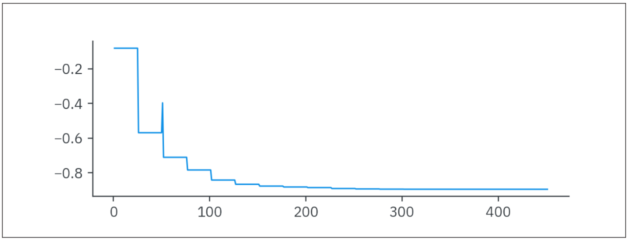


Figure 7-6. Graph of mean ~~vs~~ evaluation count

Another useful argument is the `initial_point` argument. By default, the VQE algorithm chooses a random set of numbers as starting circuit parameters, but if we have a good idea where the minimum might be, this argument allows us to start the algorithm from that point instead. For example, let's start our algorithm off closer to the minimum; the code ~~below~~ runs the VQE algorithm as before~~,~~ but starting with the parameters the algorithm discovered in the 200th optimization step in the results ~~displayed above.~~

```
initial_point = log.params[200]
log = VQELog()
vqe = VQE(circuit,
          callback=log.callback,
          initial_point=initial_point,
          quantum_instance=AerSimulator(
                      method='statevector')
         )
result = vqe.compute_minimum_eigenvalue(op)
result.eigenvalue.real  # -0.8944270665137739
plt.plot(log.counts, log.means);
```

Figure 7-7 shows the plot created by the code ~~above~~.

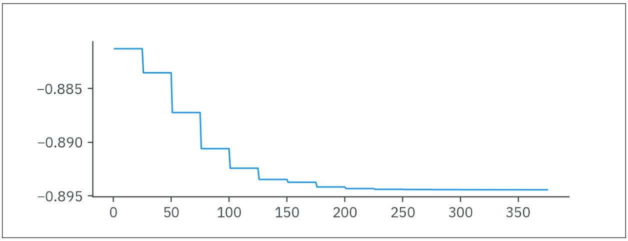


Figure 7-7. Graph of mean ~~vs~~ evaluation count for an algorithm starting at a point close to the optimal point~~.~~

We can see the algorithm found the minimum much faster.

Parameterized Circuits

We can also adjust the VQE algorithm by choosing a different form of parameterized circuit. In the previous section, we used the `EfficientSU2` circuit from Qiskit's library, but we could also use other circuits depending on the application. For example, the `TwoLocal` circuit has ~~less~~ parameters, so can converge much faster, but has the downside of not being able to create as many quantum states~~.~~

```
from qiskit.circuit.library import (EfficientSU2,
                                    TwoLocal)
len(EfficientSU2(3).parameters)  # 24
len(TwoLocal(3, 'ry', 'cx').parameters)  # 12
```

In the code cell ~~below~~, we use the `TwoLocal` circuit, with layers of `ry` and `cx` gates. The algorithm performs poorly~~,~~ and converges on a value close to `-0.5`. ~~This is because~~ this version of the `TwoLocal` gate can't create the lowest eigenstate of the operator.

```
log = VQELog()
vqe = VQE(TwoLocal(3, 'ry', 'cx'),
          callback=log.callback,
          quantum_instance=AerSimulator(
                method='statevector')
         )
result = vqe.compute_minimum_eigenvalue(op)
plt.plot(log.counts, log.means);
result.eigenvalue.real  # -0.49999960316294956
```

Figure 7-8 shows the plot created by the code ~~above~~.

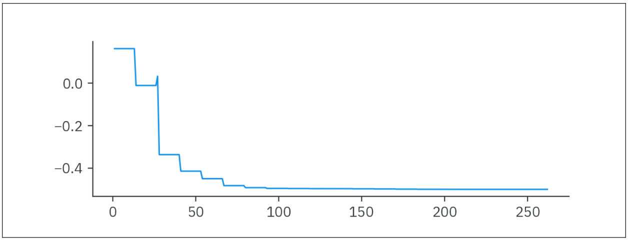


Figure 7-8. Graph of mean ~~vs~~ evaluation count for an algorithm using a poor parameterized circuit

If instead we use layers of `rx` and `cx` gates, we get much closer. The result is still not as close as with the `EfficientSU2` circuit, but this circuit converges much faster~~,~~ and gets within 1% of the correct value.

```
log = VQELog()
vqe = VQE(TwoLocal(3, 'rx', 'cx'),
                callback=log.callback,
                quantum_instance=AerSimulator(
                                  method='statevector')
              )
result = vqe.compute_minimum_eigenvalue(op)
result.eigenvalue.real  # -0.8890712131577212
plt.plot(log.counts, log.means);
```

Figure 7-9 shows the plot created by the code ~~above~~.

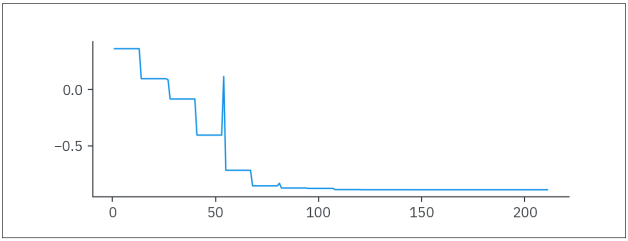


Figure 7-9. Graph of mean ~~vs~~ evaluation count for an algorithm using a good parameterized circuit

We can use any parameterized circuit with VQE, but some are more useful than others. We generally prefer circuits that can create many states (~~more~~ chance it can create our specific eigenstate), but that also scale efficiently enough to be useful on near term devices. Qiskit's circuit library contains some circuits designed for this use, known as "N-local" circuits.

Qiskit's N-local circuits have two layers: ~~A~~ *rotation* layer: ~~This~~ is a set of gates that ~~only~~ act on single qubits, or on small subsets of qubits. This layer is usually where the parameters are. ~~An *entangling* layer: This~~ is a set of multi-qubit (e.g. CCX) gates aimed to help us create entangled states.

The most general of these is the NLocal circuit. In the code snippet ~~below~~, we create an NLocal circuit with three qubits, using YGates in the rotation layers, CZGates in the entangling layers.

```
from qiskit.circuit.library import NLocal
from qiskit.circuit.library import RYGate, CZGate
from qiskit.circuit import Parameter
NLocal(3,  # number of qubits
       RYGate(Parameter('theta')),  # rotation layer
       CZGate(),  # gates in entangling layer
       entanglement='full',  # entangling gate pattern
       reps=3)
```

Figure 7-10 shows the circuit created in the code snippet ~~above~~, decomposed one layer.

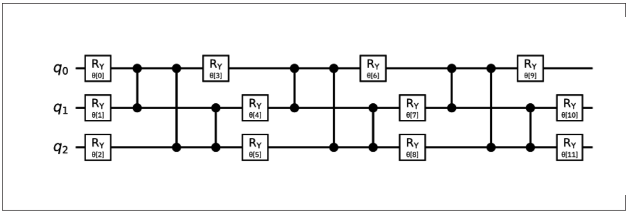


Figure 7-10. Example of an N-local circuit

With `entanglement='full'`, the entangling layers perform gates between each possible qubit pair, but the number of gates this introduces scales quadratically with the number of qubits. We can instead change this to `'linear'`, `'circular'`, or `'sca'` for different entangling schemes that each use around ~~1~~ entangling gate per qubit. We can also choose how many times the circuit repeats through the `reps` argument, which is `1` by default.

A specific case of the `NLocal` circuit is the `TwoLocal` circuit, which we've already seen in action as a~~n~~ parameterized circuit. This circuit template has layers of single-qubit gates, followed by layers of two-qubit entangling gates (e.g. CNOTs). As we saw ~~above~~, we can choose the gates this circuit uses in the rotation layers using strings, but we could also pass `Gate` or `QuantumCircuit` objects instead. Here, `reps` is `3` by default, so the line:

```
from qiskit.circuit.library import TwoLocal
TwoLocal(3, 'ry', 'cz')
```

creates the same circuit as the `NLocal` circuit we created ~~above~~ (shown decomposed in Figure 7-10).

Another example is the `RealAmplitudes` circuit, a special case of `TwoLocal`, in which the single-qubit gates are `ry` gates~~,~~ and the two-qubit gates are `cx` gates. This circuit ~~only~~ produces states with real amplitudes (i.e. phase = 0), hence the name.

Optimizers

The other key factor in variational algorithms is the classical program that decides how to twiddle the parameters to minimize/maximize the expectation value. Qiskit calls these programs *optimizers*~~,~~ and stores them under `qiskit.algorithms.optimizers`. In this guide, we will focus on *local* optimizers, which ~~only~~ aim to find local extrema~~,~~ and not necessarily the absolute lowest or highest possible energy. At the time of writing, Qiskit provides roughly 20 local optimizers.

Figure 7-11 shows a few different local optimizers finding the minimum of a very simple landscape, with only two parameters. At a high level, each of these optimizers evaluate the expectation value for a set of parameters (which we'll call a ~~‘~~point~~’~~)~~,~~ and then uses this information to guess which new points might have better expectation values. As we saw ~~above~~, we can specify a starting point if we have a good idea of where the optimal value might be, or VQE can choose a random starting point for us.

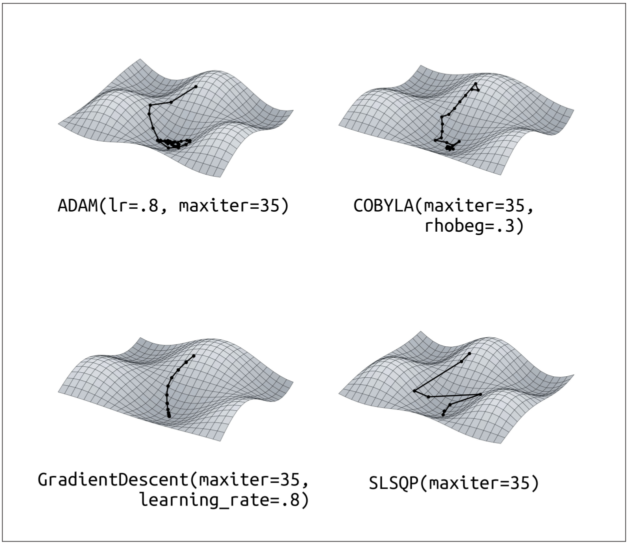


Figure 7-11. Image showing how different optimizers explore a simple 2D landscape. The x and y axes are the different possible values of the parameters, and the height of the surface shows the expectation value for those parameters. The black lines show the path the optimizers took, and the dots show the different 'points' at each step in the optimization process. Note that performance on this landscape with these arguments might not be indicative of performance in general.

For example, `GradientDescent` is a simple algorithm that estimates the gradient at its current point by measuring the difference in expectation values for small changes (perturbations) in the parameters. The algorithm then moves a step in the direction of steepest downward~~s~~ descent. We can tell the VQE algorithm to use this optimizer through the `optimizer` parameter, in the code snippet ~~below~~, we'll try this out with the default parameters.

```
from qiskit.algorithms.optimizers import GradientDes
log = VQELog()
vqe = VQE(EfficientSU2(),
          optimizer=GradientDescent(),
          callback=log.callback,
          quantum_instance=AerSimulator(
                      method='statevector')
         )
result = vqe.compute_minimum_eigenvalue(op)
result.eigenvalue.real  # -0.5997810307109372
```

The algorithm performed pretty poorly here. We know this parameterized circuit can achieve the correct value of ~ `-0.894`, so what happened? If we look at the log (Figure 7-12), we can see the algorithm used 2500 evaluations (the default maximum for `GradientDescent`), so the optimizer timed out before reaching the best value.

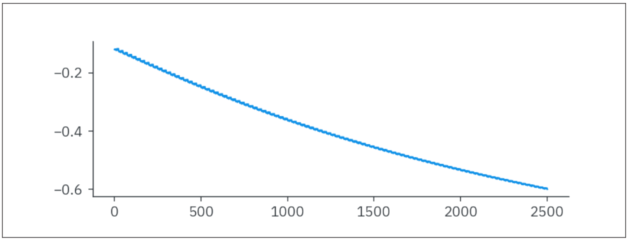


Figure 7-12. Graph of mean vs evaluation count for each point in the VQE search using gradient descent. The optimizer approaches the value slowly and reaches the maximum number of evaluations (2500) before converging on the minimum.

We can see the optimizer was heading towards the minimum correctly, but the steps were too small to get there in time. We could increase the number of iterations though the `GradientDescent`'s `'maxiter'` parameter, or better yet, change the size of steps through `'learning_rate'` parameter. The default is `0.01`, so in the code snippet ~~below~~, we set it to `0.2` to speed up convergence.

```
    log = VQELog()
    vqe = VQE(EfficientSU2(),
              optimizer=GradientDescent(
                  learning_rate=0.2
```

```
            ),
            callback=log.callback,
            quantum_instance=AerSimulator(
                        method='statevector')
          )
result = vqe.compute_minimum_eigenvalue(op)
result.eigenvalue.real  # -0.8938653211271332
```

Figure 7-13 shows how the `GradientDescent` algorithm converges as it progresses.

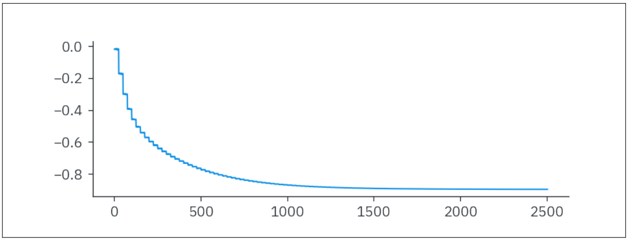


Figure 7-13. Graph of mean ~~vs~~ evaluation count for each point in the VQE search using gradient descent, with a larger learning rate. The optimizer approaches the value faster, but still hits the maximum evaluation count before converging.

This performs much better, but still doesn't converge before running out of evaluations.

For better performance, we can instead use the `COBYLA` (Constrained Optimization By Linear Approximation) algorithm (also known as Powell's method). This method doesn't need to estimate the gradient at each point, which saves circuit runs. Instead, the algorithm roughly calculates the gradient with a couple of evaluations (seen in the triangle-like path at the start of the optimization in Figure 7-11), then optimizes along a 1D line. Once it hits that constrained minimum, it chooses a new direction based on the approximated gradient and does another constrained optimization along this new line.

```
from qiskit.algorithms.optimizers import COBYLA
log = VQELog()
```

```
vqe = VQE(EfficientSU2(),
          optimizer=COBYLA(),
          callback=log.callback,
          quantum_instance=AerSimulator(
                      method='statevector')
         )
result = vqe.compute_minimum_eigenvalue(op)
result.eigenvalue.real  # -0.8944270576823009
```

Figure 7-14 plots the measured expectation values for each evaluation from the code ~~above~~.

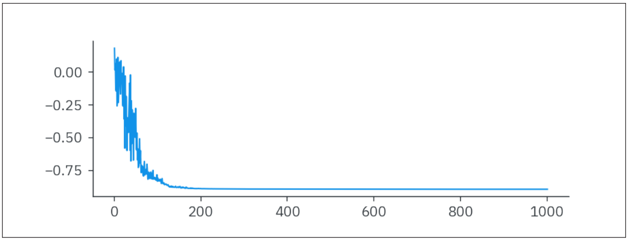


Figure 7-14. Graph of expectation value ~~vs~~ evaluation count for each point in the VQE search, using COBYLA.

This short example shows the choice of optimizer can make a big difference ~~to~~ the performance of our algorithms. Qiskit provides many optimizers, each with different arguments, behaviors, and performance on different tasks.

PART III

Additional Essential Functionality

In Part I we discussed fundamentals of quantum computing with Qiskit, and in Part II we demonstrated features of Qiskit that leverage the power of quantum information. Here in Part III we'll discuss various modules and features of Qiskit that are also essential for quantum application developers. First, Chapter 8, "Qiskit Circuit Library Standard Operations", serves as a reference for gates and instructions introduced in Part I.

Then in Chapter 9, "Working with Providers and Backends", we'll demonstrate features of Qiskit that abstract and facilitate working with various quantum computers and simulators. Finally in Chapter 10, "OpenQASM", we'll explore the quantum assembly language QASM 3.0.

CHAPTER 8

Qiskit Circuit Library Standard Operations

The Qiskit Circuit Library (module `qiskit.circuit.library`) contains many operations and circuits that may be used as building blocks for implementing quantum algorithms. Here are some standard operations categorized as instructions, single-qubit gates, and multi-qubit gates.

Standard Instructions

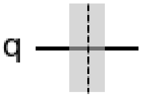
The standard instruction classes implement quantum operations that aren't necessarily unitary. They are subclasses of the `Instruction` class (see "The Instruction Class" on page 26).

Barrier

The `Barrier` class creates a barrier instruction (see "Creating a Barrier" on page 9) with a given number of qubits. A barrier provides both visual and functional separation between gates on a wire in a quantum circuit.

Signature:	Appearance:

```
Barrier(num_qubits)
```

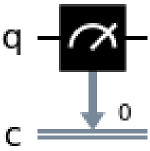


Measure

The `Measure` class creates a measurement instruction for measuring a quantum state in the computational basis, placing the binary result in a classical register (see "Measuring a quantum circuit" on page 10).

Signature: **Appearance:**

```
Measure()
```

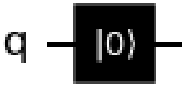


Reset

The `Reset` class creates a reset instruction that reset the qubit state to $|0\rangle$ (see "Using the reset() method" on page 19).

Signature: **Appearance:**

```
Reset()
```

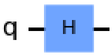


Standard Single-Qubit Gates

The standard single-qubit gates implement unitary quantum operations. They are subclasses of the `Gate` class (see "The Gate Class" on page 27). These gates may be created and applied to a circuit via the single-qubit gate methods of the `QuantumCircuit` class that appear in Table 1-1.

HGate

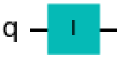
The `HGate` class creates a single-qubit **H** gate. It performs a π rotation around the X+Z axis. It also has the effect of changing the computational basis from $|0\rangle, |1\rangle$ to $|+\rangle, |-\rangle$ and vice-versa.

Signature:	Appearance:	Matrix:
`HGate(label=None)`		$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$

IGate

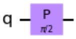
The `IGate` class creates a single-qubit **I** gate, which has no effect on the state of a qubit.

Signature:	Appearance:	Matrix:

`IGate(label=None)`		$\begin{pmatrix}1 & 0\\0 & 1\end{pmatrix}$

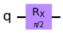
PhaseGate

The `PhaseGate` class creates a single-qubit **Phase** gate that performs a given phase rotation.

Signature:	Appearance:	Matrix:
`PhaseGate(theta, label=None)`		$\begin{pmatrix}1 & 0\\0 & e^{i\lambda}\end{pmatrix}$

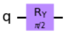
RXGate

The `RXGate` class creates a single-qubit **RX** gate that performs a given rotation around the X axis.

Signature:	Appearance:	Matrix:
`RXGate(theta, label=None)`		$\begin{pmatrix}\cos\frac{\theta}{2} & -i\sin\frac{\theta}{2}\\-i\sin\frac{\theta}{2} & \cos\frac{\theta}{2}\end{pmatrix}$

RYGate

The `RYGate` class creates a single-qubit **RY** gate that performs a given rotation around the Y axis.

Signature:	Appearance:	Matrix:
`RYGate(theta, label=None)`		$\begin{pmatrix}\cos\frac{\theta}{2} & -\sin\frac{\theta}{2}\\\sin\frac{\theta}{2} & \cos\frac{\theta}{2}\end{pmatrix}$

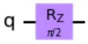
RZGate

The `RZGate` class creates a single-qubit **RZ** gate that performs a given rotation around the Z axis.

Signature:

```
RZGate(phi, label=None)
```

Appearance:



Matrix:

$$\begin{pmatrix} e^{-i\frac{\lambda}{2}} & 0 \\ 0 & e^{i\frac{\lambda}{2}} \end{pmatrix}$$

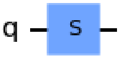
SGate

The `SGate` class creates a single-qubit **S** gate that performs a π/2 phase rotation.

Signature:

```
SGate(label=None)
```

Appearance:



Matrix:

$$\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

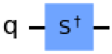
SdgGate

The `SdgGate` class creates a single-qubit **S†** gate that performs a -π/2 phase rotation.

Signature:

```
SdgGate(label=None)
```

Appearance:

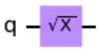


Matrix:

$$\begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}$$

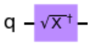
SXGate

The `SXGate` class creates a single-qubit square root of **X** gate that performs a π/2 rotation around the X axis while shifting the global phase by π/4.

Signature:	Appearance:	Matrix:
`SXGate(label=None)`		$\frac{1}{2}\begin{pmatrix}1+i & 1-i\\ 1-i & 1+i\end{pmatrix}$

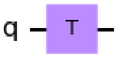
SXdgGate

The `SXdgGate` class creates a single-qubit inverse square root of **X** gate that performs a -π/2 rotation around the X axis while shifting the global phase by -π/4.

Signature:	Appearance:	Matrix:
`SXdgGate(label=None)`		$\frac{1}{2}\begin{pmatrix}1-i & 1+i\\ 1+i & 1-i\end{pmatrix}$

TGate

The `TGate` class creates a single-qubit **T** gate that performs a π/4 phase rotation.

Signature:	Appearance:	Matrix:
`TGate(label=None)`		$\begin{pmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{pmatrix}$

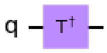
TdgGate

The `TdgGate` class creates a single-qubit **T†** gate that performs a -π/4 phase rotation.

Signature:

`TdgGate(label=None)`

Appearance:



Matrix:

$$\begin{pmatrix} 1 & 0 \\ 0 & e^{-i\pi/4} \end{pmatrix}$$

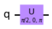
UGate

The `UGate` class creates a single-qubit **U** gate with 3 Euler angles.

Signature:

`UGate(theta, phi, lam, label=None)`

Appearance:



Matrix:

$$\begin{pmatrix} \cos(b) & -e^{i\lambda}\sin(b) \\ e^{i\varphi}\sin(b) & e^{i(\varphi+\lambda)}\cos(b) \end{pmatrix}$$

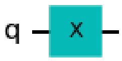
XGate

The `XGate` class creates a single-qubit **X** gate that performs a π rotation around the X axis.

Signature:

`XGate(label=None)`

Appearance:



Matrix:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

YGate

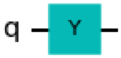
The `YGate` class creates a single-qubit **Y** gate that performs a π rotation around the Y axis.

Signature:

Appearance:

Matrix:

`YGate(label=None)`



$$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

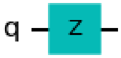
ZGate

The `ZGate` class creates a single-qubit **Z** gate that performs a π rotation around the Z axis.

Signature:

`ZGate(label=None)`

Appearance:



Matrix:

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Standard Multi-Qubit Gates

The standard multi-qubit gates implement unitary quantum operations. They are subclasses of the `ControlledGate` class (see "The ControlledGate Class" on page 28). Some of these gates may be created and applied to a circuit via the multi-qubit gate methods of the `QuantumCircuit` class, many of which appear in Table 1-2.

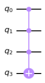
C3XGate

The `C3XGate` class creates a four-qubit gate that has an **X** gate and three control qubits.

Signature:

```
C3XGate(label=None,
ctrl_state=None)
```

Appearance:



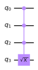
C3SXGate

The `C3SXGate` class creates a four-qubit gate that has a square root of **X** gate and three control qubits.

Signature:

```
C3SXGate(label=None, ctrl_state=None,
*, angle=None)
```

Appearance:



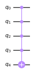
C4XGate

The `C4XGate` class creates a five-qubit gate that has an **X** gate and four control qubits.

Signature:

```
C4XGate(label=None,
ctrl_state=None)
```

Appearance:



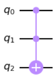
CCXGate

The `CCXGate` class creates a three-qubit gate that has an **X** gate and two control qubits. This is also known as a Toffoli gate.

Signature:

```
CCXGate(label=None,
ctrl_state=None)
```

Appearance:



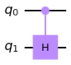
CHGate

The `CHGate` class creates a controlled-Hadamard gate, applying the Hadamard according to the control qubit state.

Signature:

Appearance:

```
CHGate(label=None, ctrl_state=None)
```



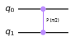
CPhaseGate

The `CPhaseGate` class creates a controlled Phase gate, applying the `PhaseGate` according to the control qubit's state.

Signature:

```
CPhaseGate(theta, label=None,
ctrl_state=None)
```

Appearance:



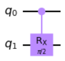
CRXGate

The `CRXGate` class creates a controlled-RX gate, applying the `RX` according to the control qubit state.

Signature:

```
CRXGate(theta, label=None,
ctrl_state=None)
```

Appearance:



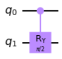
CRYGate

The `CRYGate` class creates a controlled-RY gate, applying the `RY` according to the control qubit state.

Signature:

```
CRYGate(theta, label=None,
ctrl_state=None)
```

Appearance:



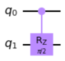
CRZGate

The `CRZGate` class creates a controlled-RZ gate, applying the `RZ` according to the control qubit state.

Signature:

```
CRZGate(theta, label=None,
ctrl_state=None)
```

Appearance:



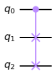
CSwapGate

The `CSwapGate` class creates a three-qubit gate whose **Swap** gate is applied according to the control qubit state.

Signature:

```
CSwapGate(label=None,
ctrl_state=None)
```

Appearance:



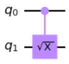
CSXGate

The `CSXGate` class creates a controlled-SX (square root of X) gate, applying the $\sqrt{X}$ gate according to the control qubit state.

Signature:

```
CSXGate(label=None,
ctrl_state=None)
```

Appearance:



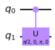
CUGate

The `CUGate` class creates a controlled-U gate, applying the **U** gate including a global phase argument, according to the control qubit state.

Signature:

```
CUGate(theta, phi, lam, gamma,
label=None, ctrl_state=None)
```

Appearance:



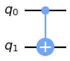
CXGate

The `CXGate` class creates a controlled-X gate, applying the **X** gate according to the control qubit state.

Signature:

```
CXGate(label=None, ctrl_state=None)
```

Appearance:



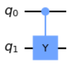
CYGate

The `CYGate` class creates a controlled-Y gate, applying the **Y** gate according to the control qubit state.

Signature:

```
CYGate(label=None, ctrl_state=None)
```

Appearance:



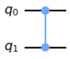
CZGate

The `CZGate` class creates a controlled-Z gate, applying the **Z** gate according to the control qubit state.

Signature:

```
CZGate(label=None, ctrl_state=None)
```

Appearance:

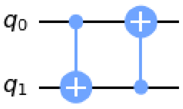


DCXGate

The `DCXGate` class creates a double-CNOT gate. This is a two-qubit gate that has two **CNOT** gates with their control-qubits on different wires.

Signature: **Appearance:**

`DCXGate()`



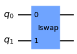
iSwapGate

The `iSwapGate` class swaps the qubit states of two quantum wires. It also changes the phase of $|01\rangle$ and $|10\rangle$ amplitudes by i.

Signature:

`iSwapGate(label=None, ctrl_state=None)`

Appearance:



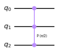
MCPhaseGate

The `MCPhaseGate` class creates a multi-controlled **Phase** gate with a given number of control qubits.

Signature:

`MCPhaseGate(lam, num_ctrl_qubits, label=None)`

Appearance:



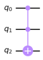
MCXGate

The `MCXGate` class creates a multi-controlled **X** gate with a given number of control qubits. This is a generalization of a Toffoli gate.

Signature:

```
MCXGate(num_ctrl_qubits=None,
label=None, ctrl_state=None)
```

Appearance:



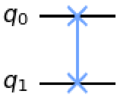
SwapGate

The `SwapGate` swaps the qubit states of two quantum wires.

Signature:

```
SwapGate(label=None)
```

Appearance:



CHAPTER 9

Working with Providers and Backends

There are lots of different ways to run quantum circuits and quantum algorithms. If we want to see how a noise-free quantum computer behaves, we could choose from a few different simulators (e.g., statevector or unitary), and we're not ~~just~~ limited to local simulators. Chances are, at some point you will want to run a circuit on a real quantum system, accessed remotely.

To manage all these options, and ensure all backends are compatible with the same data types, Qiskit defines a general `Backend` object. This makes it easier to interchange backends in our code (and also to remember what the right code is in the first place). You've already seen these `Backend` objects in action, as we use the `Backend.run()` method every time we run a circuit. Real quantum systems have unique properties that can change regularly. The `Backend` object also helps us access this information so we can investigate different systems~~,~~ and make decisions about which systems to use. We can access this information both programmatically~~,~~ and through graphical user interfaces.

These `Backend` objects are organised by *providers*. If you've read the rest of this guide, you will have used the `Aer`~~,~~ and

`BasicAer` providers, which manage simulator backends that run on your local machine (e.g. using `.get_backend()` to retrieve a backend object by name), but we can install other providers to use other backends (real or simulated). Examples are the IBM Quantum provider (from the `qiskit-ibmq-provider` Python package) through which we can access IBM Quantum's online backends, or the IonQ provider (from the `qiskit-ionq` package) to access IonQ's online backends.

Qiskit and its provider interface allow us to search through these backends, and provide some useful functionality for remote services that we'll explore later in this chapter. In this ~~guide~~, we'll use IBM Quantum's provider for the examples, but other providers are available, and you can even write your own. To set up a provider for remote backends, you'll usually need to make an account with them~~,~~ and save an API key to your environment. The code for IBM Quantum is shown ~~below~~, but other providers will have their own instructions~~:~~

```
from qiskit import IBMQ
IBMQ.save_account('API_TOKEN')
```

As it's an environment variable, you ~~only~~ need to save your account once. You can then access the open provider using the code ~~below~~.

```
from qiskit import IBMQ
provider = IBMQ.load_account()
```

Graphical Tools

For users working in Jupyter environments, Qiskit provides some graphical tools. You can install these using the command ~~below~~.

```
pip install qiskit[visualization]
```

Once installed, we can activate these tools with ~~the~~ line of code ~~below~~.

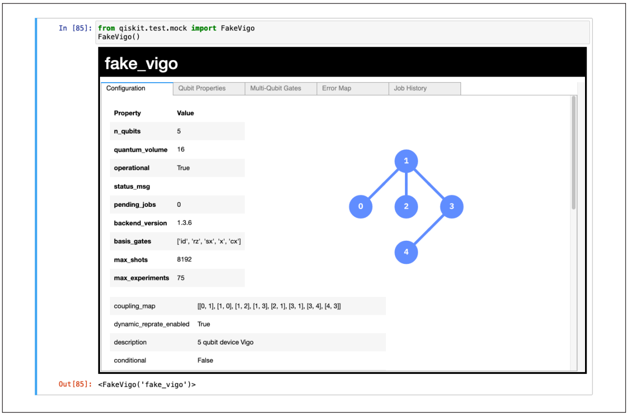
```
from qiskit.tools.jupyter import *
```

This will enable some Jupyter 'magics' (commands specific to this environment), and will patch some objects to enable richer representations.

We can see this in action ~~below~~. We import a mock backend, and ask Jupyter to display it by returning it as the last line in the cell.

```
from qiskit.test.mock import FakeVigo
FakeVigo()
```

Figure 9-1 shows the output of the code ~~above~~ in a Jupyter notebook.



```
In [85]: from qiskit.test.mock import FakeVigo
         FakeVigo()
```

fake_vigo

Property	Value
n_qubits	5
quantum_volume	16
operational	True
status_msg	
pending_jobs	0
backend_version	1.3.6
basis_gates	['id', 'rz', 'sx', 'x', 'cx']
max_shots	8192
max_experiments	75

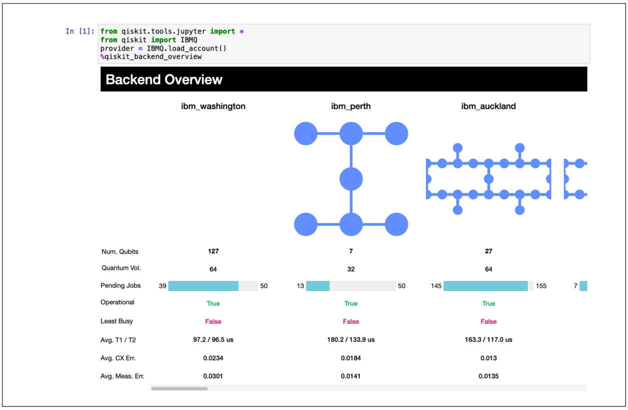
coupling_map	[[0, 1], [1, 0], [1, 2], [1, 3], [2, 1], [3, 1], [3, 4], [4, 3]]
dynamic_reprate_enabled	True
description	5 qubit device Vigo
conditional	False

```
Out[85]: <FakeVigo('fake_vigo')>
```

Figure 9-1. Screenshot of the interactive backend GUI in a Jupyter Notebook

This displays an interactive panel with images and switchable tabs. We can also now use magic commands, such as `%qiskit_version_table`, which displays the Qiskit version information as an HTML table. If we have the IBM Quantum provider set up, we can use the `%backend_overview` command to display an interface with information on its remote systems (such as queue times and system properties).

Figure 9-2 shows the %backend_overview GUI in a Jupyter notebook cell.



```
In [1]: from qiskit.tools.jupyter import *
        from qiskit import IBMQ
        provider = IBMQ.load_account()
        %qiskit_backend_overview
```

Backend Overview

	ibm_washington	ibm_perth	ibm_auckland
Num. Qubits	127	7	27
Quantum Vol.	64	32	64
Pending Jobs	39 / 50	13 / 50	145 / 155
Operational	True	True	True
Least Busy	False	False	False
Avg. T1 / T2	97.2 / 96.5 us	180.2 / 133.9 us	163.3 / 117.0 us
Avg. CX Err.	0.0234	0.0184	0.013
Avg. Meas. Err.	0.0301	0.0141	0.0135

Figure 9-2. Screenshot of the backend overview GUI in a Jupyter Notebook, showing details and status of available backends~~.~~

Text-~~B~~ased Tools

For non-Jupyter environments, some tools are also available as text-based tools. For example, there is a Python function with a similar job to the %backend_overview magic~~.~~

```
from qiskit.tools import backend_overview
backend_overview()
```

The code ~~above~~ prints output similar to the text ~~shown below~~. The output will depend on the backends available, and some text is omitted to save space~~.~~

```
ibm_washington                 ibm_perth
--------------                 ---------
Num. Qubits:  127              Num. Qubits:  7
Pending Jobs: 133              Pending Jobs: 1
Least busy:   False            Least busy:   False
Operational:  True             Operational:  True
Avg. T1:      97.2             Avg. T1:      173.6
Avg. T2:      96.5            Avg. T2:      133.9


ibm_cairo                     ibm_lagos
---------                     ---------
Num. Qubits:  27              Num. Qubits:  7
Pending Jobs: 7               Pending Jobs: 9
Least busy:   False           Least busy:   False
Operational:  True            Operational:  True
Avg. T1:      96.6            Avg. T1:      145.5
Avg. T2:      107.1           Avg. T2:      109.3
```

Similarly, we can print a textual summary of a backend using the `backend_monitor` function.

```
from qiskit import IBMQ
provider = IBMQ.load_account()
armonk = provider.get_backend('ibmq_armonk')

from qiskit.tools import backend_monitor
backend_monitor(armonk)
```

The code above will print output similar to ~~that shown below~~ (again, some text is omitted, and the values will change depending on the backend).

```
ibmq_armonk
===========
Configuration
-------------
    n_qubits: 1
    operational: True
    status_msg: active
    pending_jobs: 80
    backend_version: 2.4.29
    basis_gates: ['id', 'rz', 'sx', 'x']
    local: False
    simulator: False
    conditional_latency: []
    ...
```

Getting System Info Programmatically

We can also get a backend's information in a format that's easy to work with in Python. This is how the transpiler knows how to prepare and optimize a circuit for a specific system, given only the backend object. Qiskit splits information about a backend into three categories:

- The *configuration* is information about the system that doesn't change with time. Examples include the number of qubits~~,~~ and the coupling map. We access this through the `.configuration()` method.
- The *properties* is information that *can* change with time~~,~~ and that requires re~~-~~measuring and calibrating. Examples include the gate errors~~,~~ and decoherence times of the qubits. We access this through the `.properties()` method.
- The *options* are the default settings used when running jobs on the backend. We can override these when using the `.run()` method (e.g. `.run(qc, shots=2048)`), or change these defaults for an instance of a backend. We access these through the `.options` attribute.

We can pair these methods and attributes with the provider interface to automatically select backends that fit certain criteria. The last line in the code snippet ~~below~~ returns a list of backends that are *not* simulators~~,~~ and that have more than three qubits~~.~~

```python
def is_ok_backend(backend):
    return (
        not backend.configuration().simulator
        and backend.configuration().num_qubits > 3)

provider.backends(filters=is_ok_backend)
```

The data available from `.configuration()` and `.properties()` depends on the backend, but we can inspect this using Python's `vars()` function. In the code ~~below~~, we show the information available from the `FakeVigo` device (we've omitted some text to save space).

```
from qiskit.test.mock import FakeVigo
vars(FakeVigo().configuration())

{'_data': {'allow_q_object': True,
   'meas_map': [[0, 1, 2, 3, 4]],
   'multi_meas_enabled': False,
   'quantum_volume': 16,
   'url': 'None',
   'allow_object_storage': True},
  'backend_name': 'fake_vigo',
  'backend_version': '1.3.6',
  'n_qubits': 5,
  'basis_gates': ['id', 'rz', 'sx', 'x', 'cx'],
  ...
  'online_date': datetime.datetime(2019, 7, 3, 4, 0, tzinfo=t
  'description': '5 qubit device Vigo',
  'dt': 2.2222222222222221e-10,
  'dtm': 2.2222222222222221e-10}
```

The ability to get this information programmatically is particularly useful when finding specific properties across different devices~~,~~ or when collecting data. The code ~~below~~ finds the two qubits with the lowest CNOT error rate between them, out of all the devices from the provider.

```
def find_best_cx(provider):
    """Find the best (lowest error) CXGate
    across all qubits available in `provider`"""

    best_err, best_backend, best_pair = 1, None, Non
    for backend in provider.backends():
        config = backend.configuration()

        # skip simulators & single-qubit devices
        if config.simulator or config.num_qubits < 2:
            continue

        for pair in config.coupling_map:
            err = backend.properties().gate_error('cx', pair
            if err < best_err:
                best_err, best_backend, best_pair = err, bac

     return (best_backend, best_pair, best_err)
```

Finally, we can view and change a backend's default options through the .options attribute. The code ~~below~~ shows the default options for FakeVigo (again, with some information omitted).

```
vigo = FakeVigo()
vigo.options

Options(shots=1024, method=None, device='CPU', precis[illegible]'do
seed_simulator=None, ..., mps_omp_threads=1)
```

These are the default values vigo will use when running circuits. If we overwrite these (e.g. with vigo.options.shots = 2048), this will change the default for this instance of the object, i.e. vigo.run(qc) will use the new number of shots. Note that this will not overwrite the default for *other* instances of the object, so running vigo = FakeVigo() will reset vigo.options to its initial state.

Interacting with Quantum Systems on the Cloud

So far, we've treated remote backends in the same way as local backends, and all the functionality we've seen so far in the chapter applies to both. When using remote backends, however, we will often have to wait for network processes and device queues, which we don't experience with local backends. Qiskit includes some tools to help with this:

Convenience Tools

The job_monitor function regularly checks the status of a job and displays it to the user. If backend is a remote Backend object, and `qc` is a valid `QuantumCircuit` then the code ~~below~~ runs the circuit remotely~~,~~ and monitors the job~~.~~

```
from qiskit.tools import job_monitor
job = backend.run(qc)
job_monitor(job)
```

As the job progresses, we'll see updates printed, such as~~:~~

```
Job Status: job is being validated
Job Status: job is queued (1)
Job Status: job is actively running
Job Status: job has successfully run
```

Runtime Services

As we've seen, processing and queueing jobs can add a significant wait time to our experiments. While this is ~~ok~~ for one-off circuit executions, it makes carrying out any experiment that requires feedback loops between circuit creators and circuit results very difficult. Variational algorithms (such as the Variational Quantum Eigensolver, discussed in Chapter 7) are one such example.

As a solution, providers can offer runtime services that accept inputs for algorithms~~,~~ and that run these algorithms in full (not re~~-~~submitting jobs for each circuit execution). For example, the code ~~below~~ defines a ~~Variational Quantum Eigensolver (~~VQE~~)~~ algorithm as a dictionary. This ~~is~~ includes the problem (the variable `op` ~~below~~)~~,~~ and other algorithm parameters. Refer to Chapter 7 for more information on the VQE algorithm~~.~~

```
from qiskit.circuit.library import TwoLocal
from qiskit.algorithms.optimizers import COBYLA
from qiskit.opflow import X, Y, Z, I

op = ( .5 * (X ^ Y ^ Z)
     + .2 * (Y ^ Y ^ I)
     - .3 * (Z ^ X ^ Z)
     + .2 * (I ^ X ^ Y))

runtime_inputs = {
  'ansatz': TwoLocal(3, 'rx', 'cx'),
  'initial_parameters': 'random',
  'operator': op,
  'optimizer': COBYLA(maxiter=500)
}
```

With this procedure defined, we can then send this off to our provider to complete the full algorithm and return the results.

```
from qiskit import IBMQ
provider = IBMQ.load_account()

job = provider.runtime.run(
    program_id='vqe',
    options={'backend_name': 'ibmq_qasm_simulator'},
    inputs=runtime_inputs
)

result = job.result()
```

The runtime programs available will depend on your provider.

CHAPTER 10

OpenQASM

QASM is a low-level, imperative programming language, describing quantum programs in terms of the specific actions the quantum computer should take.

It is an intermediate representation between human-editable descriptions of quantum programs (such as Qiskit's `QuantumCircuit`)~~,~~ and quantum hardware controllers. To enable it to describe full quantum programs, QASM also supports some basic classical logic, similar to higher-level classical languages such as C.

In this ~~guide~~, we'll ~~only~~ cover gate-level operations, but QASM does also support some pulse-level quantum programming too.

Building Quantum Circuits in QASM

In this section, we'll cover the QASM syntax needed to create simple quantum circuits.

Comments

Before we start, we'll learn how to annotate the code we're writing.

You can use comments as messages to other humans reading your code (including your future self). Two slashes (//) will mark the rest of the line as a comment. You can also use the character sequences /* and */ to mark the start and end of comments respectively, over many lines. We'll use comments to describe the code examples in this chapter~~.~~

```
the compiler will read this // but not this
/* or any
of this */
```

Version strings

At the time of writing, there are three versions of QASM, so you might want to specify which version of QASM you're writing in. For this reason, the first non-comment line of a QASM file can be a *version string* (example shown in the code snippet ~~below~~). In this chapter, all code will be in QASM 3.0~~,~~ so will mostly ignore version strings. The version string starts with OPENQASM, followed by a space, the major version, a period, the minor version, and a semicolon. ~~Below~~ is the version string for OpenQASM 3.0, which we use in this ~~guide.~~

```
OPENQASM 3.0;
```

Basic Syntax

QASM is a rich language that can describe complex quantum programs, including classical and quantum routines. In this section, we'll ignore most of these features~~,~~ and ~~only~~ worry about creating individual quantum circuits. We'll see how to create more sophisticated programs later on in this chapter.

The quantum systems targeted by QASM can have any number of qubits~~,~~ and may support any kind of quantum gates. In this subsection, we'll imagine a system with three qubits~~,~~ and that supports CX gates and U gates. To start, let's look at the QASM code that describes the "hello, world!" of quantum circuits: ~~A~~ circuit that creates a GHZ state, in which our qubits are in an equal superposition of $|000\rangle$ and $|111\rangle$. We'll also add some measurements so our experimentalists can verify this. Figure 10-1 shows the circuit we'll create.

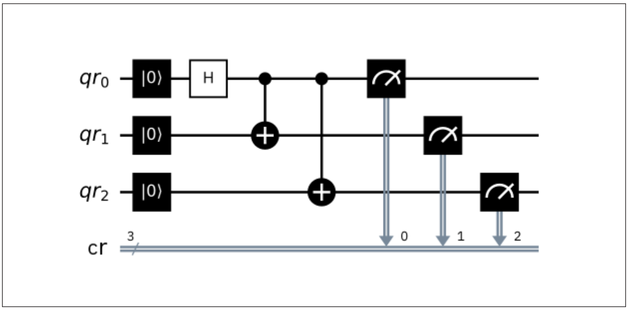


Figure 10-1. A simple quantum circuit

The following QASM code produces the circuit shown in Figure 10-1:

```
// Declare data
// type[number] name;
qubit[3] qr;
bit[3] cr;

// Initialize qubits to |0>
reset qr[0];
reset qr[1];
reset qr[2];

// Specify quantum gates
h qr[0];
CX qr[0], qr[1];
CX qr[0], qr[2];

// Measure to classical bits
cr[0] = measure qr[0];
cr[1] = measure qr[1];
cr[2] = measure qr[2];
```

We've split this code into four sections:

1. First, we declare and name the data (both quantum and classical) that we'll be manipulating. A declaration starts with the type, followed by the name, and ends with a semicolon. The two data types we use here are `bit`~~,~~ and `qubit`, two things you should already be familiar with. Like many other programming languages, the square bracket notation allows us to declare an array of a type. In the first non-code line, we're declaring that we'll manipulate three qubits, and in the next two we name the data. We called our array of qubits `qr` and our array of bits `cr`. Finally, we end the line with a semicolon.
2. Next, we initialize each qubit to the state $|0\rangle$ using the `reset` instruction. Here, we're using the square bracket notation to access the individual elements of the `qr` array.
3. We now tell the quantum computer how to manipulate our qubits. As with classical assembly languages, we first specify the operation, then the data we want to apply the operation to. For example, `h qr[0];` tells the quantum computer to perform a H-gate on the first qubit in the array `qr`. We end these lines with a semicolon.
4. Finally, we need to measure our qubits to see the results. The quantum gates we applied before had no "output" they just changed the state of the qubits. But the `measure` operation *does* return something (a `bit`), and we can assign its outputs to our classical bits.

Implicit Looping

When programming, we often find ourselves repeating the same operation on many items. QASM supports syntax to simplify this; if we apply an operation to an array, the compiler will try to repeat this operation on each item in that array. For example, the code ~~below~~:

```
qubit[3] qr;
h qr;
```

declares a three-qubit array named `qr`~~,~~ and applies a H-gate to each qubit in `qr`. This is ~~a~~ shorthand for the code ~~below~~:

```
qubit[3] qr;
h qr[0];
h qr[1];
h qr[2];
```

This behavior is the same for all operations that act on a single type. For operations that take two inputs, the operation will try to iterate through any arrays in unison. For example, in the code ~~below,~~ we iterate through two arrays of qubits (named `control` and `target`):

```
qubit[3] control;
qubit[3] target;

CX control, target;
```

This produces the circuit shown in Figure 10-2:

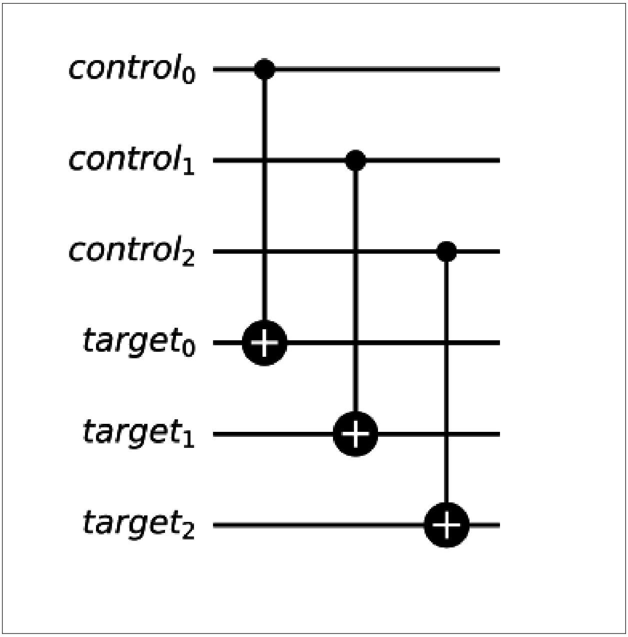


Figure 10-2. A two-register circuit where each qubit in one register is controlled by each qubit in the other~~.~~

And is equivalent to:

```
qubit[3] control;
qubit[3] target;

CX control[0], target[0];
CX control[1], target[1];
CX control[2], target[2];
```

This behavior ~~only~~ makes sense if the two arrays are the same size, so if we apply a multi-input operation on two arrays of different size, the compiler will raise an error. If any of the inputs are not arrays, then the compiler will not attempt to iterate through those inputs~~,~~ and will use the same input for each operation in the loop.

For example, in the code ~~below~~, `qr[0]` refers to a single `qubit` (not an array), but `qr[1:2]` does refer to an array (containing qubits `qr[1]` and `qr[2]`).

```
qubit[3] qr;
CX qr[0], qr[1:2];
```

The compiler will repeat the operation for each item in `qr[1:2]`, with the control qubit `qr[0]`. Figure 10-3 shows the circuit described by the code ~~immediately above~~.

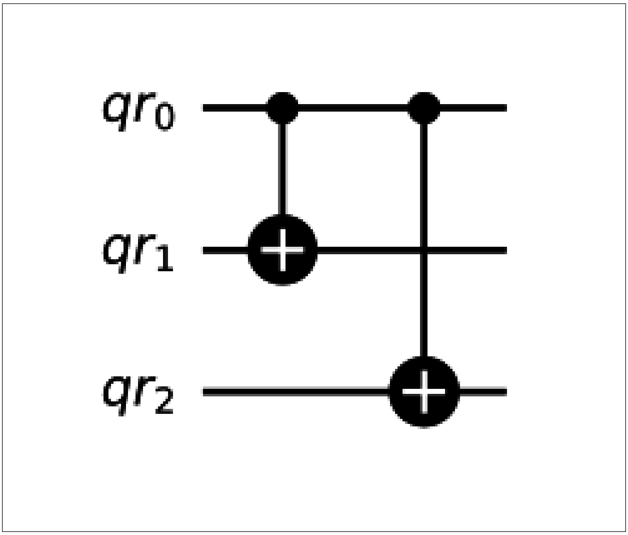


Figure 10-3. The quantum gates in the GHZ circuit

With this new syntax, we can create the GHZ circuit we saw in the previous section with the shorter, more readable, and more flexible code ~~below~~:

```
// Declare data
qubit[3] qr;
bit[3] cr;

// Initialize qubits to |0>
```

```
reset qr;

// Specify quantum gates
h qr[0];
CX qr[0], qr[1:2];

// Measure to classical bits
cr = measure qr;
```

Figure 10-4 shows the circuit produced by the code ~~above~~.

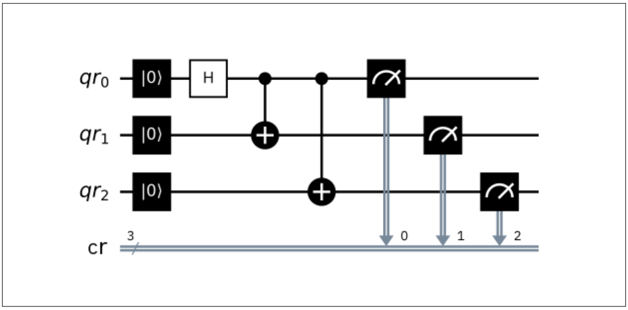


Figure 10-4. A simple quantum circuit

Quantum Gates and Instructions

In this section, we'll see how to describe some basic quantum operations through QASM.

Gates

QASM understands two quantum gates ~~out of the box~~: ~~T~~he single-qubit U-gate~~,~~ and the two-qubit CNOT. ~~These~~ two gates ~~together~~ are universal, meaning we can build any quantum gate from a combination of these two gates.

The U-gate is the most general single-qubit gate. It's a parameterized gate, and we specify the parameters using rounded brackets after the instruction. For example, the code ~~below~~ applies a U-gate to the third qubit of the quantum register qr:

```
U(0, 0, pi) qr[3];
```

The definition of the U-gate from the OpenQASM specification is the same as the U-gate implemented in Qiskit:

$$U(\theta, \phi, \lambda) : = \begin{pmatrix} \cos\left(\theta/2\right) & -e^{i\lambda}\sin\left(\theta/2\right) \\ e^{i\phi}\sin\left(\theta/2\right) & e^{i(\phi+\lambda)}\cos\left(\theta/2\right) \end{pmatrix}$$

This definition has three parameters, theta (θ), phi (ϕ), and lambda (λ), and as such, the `U` command requires these three parameters, i.e. `U(theta, phi, lambda)`.

The other built-in gate is the CNOT, which has the command `CX`.

Instructions

QASM also supports two built-in, non-unitary quantum instructions. The first is `reset`, which resets a qubit to $|0\rangle$.

```
qubit[4] qr;
reset qr;  // set all qubits in qr to |0>
```

The second is `measure`, which measures the state of a qubit in the computational basis (~~A.K.A.~~ the Z-basis), and writes the result to an output bit. These measurements project the state of a qubit to either $|0\rangle$ or $|1\rangle$, and we can immediately start manipulating this qubit after measurement.

```
bit[3] cr;
qubit[3] qr;
reset qr;

cr = measure qr;
```

Building Higher-Level Gates

In the last section, we saw the QASM's basic syntax, and built-in operations. In this section we'll look at how we can build custom operations from these built-in commands.

Modifying Existing Gates

One of the ways we can describe more complicated quantum gates is through *gate modifiers*. Wherever we use a quantum gate, we can prefix that gate with a modifier keyword and the `@` character to change that gate's behavior.

For example, the `ctrl` modifier controls a gate on the state of another qubit being |1⟩. For example, since `U(0, 0, pi)` is the Pauli Z gate, the code ~~below~~ performs the controlled-Z gate on the first two qubits of `qr`.

```
// controlled-Z gate
ctrl @ U(0, 0, pi) qr[0], qr[1];
```

Figure 10-5 shows the circuit described by the code ~~above~~.

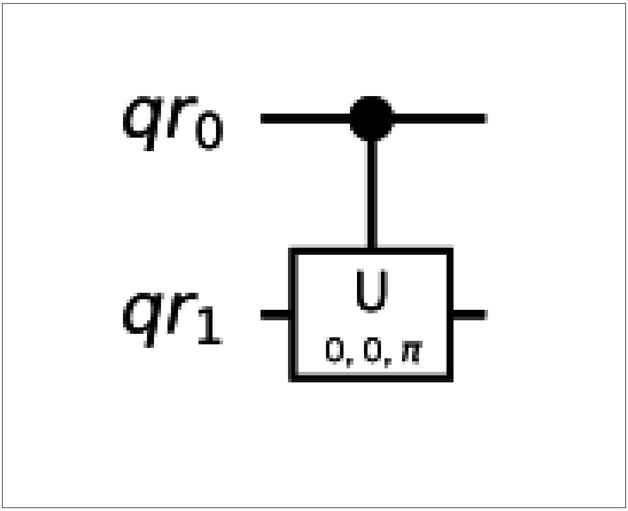


Figure 10-5. A controlled-U gate

Similarly, the `negctrl` modifier conditions the gate on the state of the control qubit being |0⟩.

The `inv` modifier *inverts* a gate. For example, in the code snippet ~~below~~, we apply a T-gate to the qubit `q`, followed by the inverse of the T-gate (i.e. the $T^\dagger$-gate). This sequence of gates is equivalent to doing nothing (the identity operation).

```
U(0, 0, pi/4) qr;  // T-gate
inv @ U(0, 0, pi/4) qr;  // T†-gate
```

The code produces the circuit shown below (note that ϕ and λ have the same effect when θ is 0). Figure 10-6 shows the circuit described by the code above.

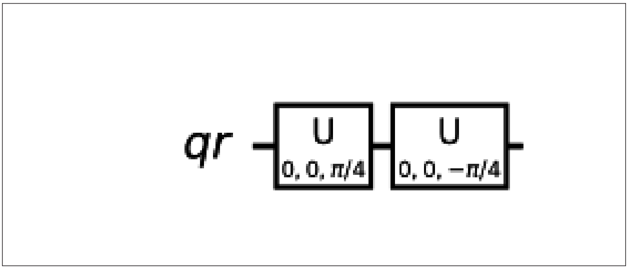


Figure 10-6. A U-gate, followed by its inverse

Finally, the `pow(n)` modifier repeats the gate `n` times.

```
pow(3) @ U(0, 0, pi/4) qr;
```

Figure 10-7 shows the circuit described by the code ~~above~~.

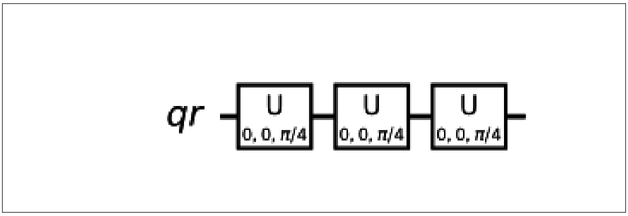


Figure 10-7. A U-gate repeated twice

We can also stack these modifiers as in the code ~~below~~.

```
inv @ pow(3) @ U(0, 0, pi/4) qr;
```

Figure 10-8 shows the circuit described by the code ~~above~~.

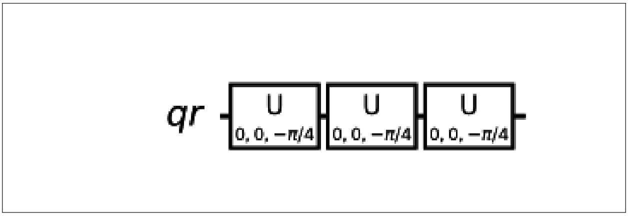


Figure 10-8. A U(0, 0, pi/4) gate, inverted and repeated twice

Defining New Gates

To declare a new quantum gate, we use the `gate` keyword, followed by:

1. The name ~~of the name~~ of the new gate~~.~~
2. Any parameters the gate takes (in parentheses)~~.~~
3. Names of qubits the gate acts on~~.~~
4. The gate's definition in terms of other operations (inside curly brackets)~~.~~

For example, the code ~~below~~ defines a controlled-RZ gate.

```
// call gate 'crz'
// gate takes one parameter (phi)
// gate acts on two qubits (q0 & q1)
gate crz(phi) q0, q1 {
    ctrl @ U(0, 0, phi) q0, q1;
}
```

The code ~~below~~ shows this gate definition in action; once we've defined a new gate type, we can use it just like any other gate.

```
// define controlled-RZ gate
gate crz(phi) q0, q1 {
    ctrl @ U(0, 0, phi) q0, q1;
}

// declare circuit data types
```

```
    qubit[2] qr;
    bit[2] cr;

    // construct simple circuit
    reset qr;
    crz(pi/4) qr[0], qr[1];
    cr = measure qr;
```

~~This~~ code ~~above~~ describes the quantum circuit shown in Figure 10-9.

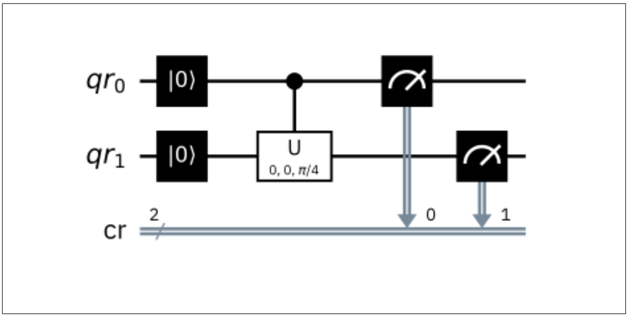


Figure 10-9. A circuit containing resets, a controlled-U operation, and measurements.

Classical Types and Instructions

In the previous sections, we used two of the data types that QASM supports. The first is the `qubit`, the smallest unit of quantum information, and the other is the `bit`, the smallest unit of classical information. We also saw that we could declare and operate on *arrays* of those types using the square bracket syntax.

```
    bit b0;      // declare a bit named 'b0'
    qubit q0;    // declare a qubit named 'q0'
    bit[2] cr;   // array of 2 bits named 'cr'
    qubit[3] qr; // array of 3 qubits named 'qr'
```

Due to the limited nature of near-term devices, the only quantum data type QASM supports is the `qubit`. However, quantum circuits are controlled by classical routines, and QASM does support many different classical types to make this easier. All classical types in QASM are arrays of `bits`, but it's much easier to abstract this out a level to types we're more familiar with in higher-level languages.

The syntax is the same as with declaring arrays, i.e. `type[size] name`. For example, the code ~~below~~ declares an integer (more on this ~~below~~) with 16 bits, named `my_int`~~.~~

```
int[16] my_int;
```

Table 10-1 lists some classical data types available in QASM 3.0.

Table 10-1. Classical data types supported by QASM

Type	Description
`bit`	We've already seen this type. This is the smallest unit of classical information~~,~~ and can have values of either `0` or `1`. All other classical types are built from `bits`.
`int`	This data type represents a signed integer (i.e. an integer that can be negative). One of the bits stores the sign (+ or `-`), and the rest store the integer in binary notation. This means `n` bits can store integers between -2^{n-1} and 2^{n-1}.
`uint`	This data type represents *unsigned* (positive) integers. `n` bits can store integers between 0 and 2^n.
`float`	This data type uses the IEEE 754 specification to represent floating-point numbers.
`angle`	You're less likely to have seen this is a built-in data type in other low-level languages, but due to the rotational nature of quantum gates, QASM allows us to specify an *angle* as a fraction of a full rotation. `n` bits allows us to specify angles between 0 and 2π within an rror of $\pm 2\pi/2^n$. Casting a different value to an `angle`