In [173…

```python
import math
import numpy as np

from qiskit import Aer, BasicAer, QuantumCircuit, transpile, execute

from qiskit.circuit import ClassicalRegister, QuantumRegister, QuantumCircuit, library
from qiskit.circuit.library.standard_gates import CPhaseGate

from qiskit.quantum_info import Statevector, DensityMatrix, Operator, Pauli, Kraus
from qiskit.quantum_info import average_gate_fidelity, process_fidelity,state_fidelity

from qiskit.visualization  import array_to_latex, plot_bloch_vector, plot_bloch_multivector, plot_histogram

from qiskit.providers.aer import AerSimulator

from qiskit.providers.aer.noise import NoiseModel, depolarizing_error

from qiskit.test.mock import FakeVigo
```

Quantum Circuits and Operations

Constructing Quantum Circuits

`QuantumCircuit` class

Drawing and barrier

Measuring

Obtaining info

Manipulating

Saving a state when running a ciruit on `AerSimulator`

`QuantumRegister` class`

`ClassicalRegister` class`

Instructions and Gates

`Instruction` class

`Gate` Class

`ControlledGate` class

Parameterized Quantum Circuits

Creating a `Parameter` Instance

Using the `ParameterVector` class

In [ ]:

```python

```

Running Quantum Circuits

The `qiskit.providers.basicaer` module contains a basic set of simulators implemented in Python, often referred to as **BasicAer simulators**.

The `qiskit.providers.aer` module contains a comprehensive set of high performance simulators, often referred to as **Aer simulators**.

The `qiskit.providers` module contains classes that support these **simulators as well as access to real quantum devices**.

Using the BasicAer simulators

In [9]:

```python
from qiskit import BasicAer

for i in range(len(BasicAer.backends())):
    print(BasicAer.backends()[i])
    i+=1
print(BasicAer.backends())
```

```
qasm_simulator
statevector_simulator
unitary_simulator
[<QasmSimulatorPy('qasm_simulator')>, <StatevectorSimulatorPy('statevector_simulator')>, <UnitarySimulatorPy('unitary_simulator')>]
```

BasicAer qasm_simulator

In [11]:

```
from qiskit import QuantumCircuit, BasicAer, transpile

##write circuit
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1)
qc.measure_all()

##run it
backend = BasicAer.get_backend('qasm_simulator')  #******
tqc = transpile(qc, backend)
job = backend.run(tqc, shots=1000)
result = job.result()
counts = result.get_counts(tqc) #******
print(counts)
```

```
{'00': 475, '11': 525}
```

Basic Aer statevector_simulator

In [17]:

```
from qiskit import QuantumCircuit, BasicAer, transpile

##write circuit
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1)
#qc.measure_all() ### don't want to measure it

##run it
backend = BasicAer.get_backend('statevector_simulator') #******
tqc = transpile(qc, backend)
job = backend.run(tqc, shots=1000)
result = job.result()
statevector = result.get_statevector(tqc)  #******
print(statevector)
```

```
[0.70710678+0.j 0.        +0.j 0.        +0.j 0.70710678+0.j]
```

Basic Aer unitary_simulator

In [18]:

```
from qiskit import QuantumCircuit, BasicAer, transpile

##write circuit
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1)
#qc.measure_all() ### don't want to measure it

##run it
backend = BasicAer.get_backend('unitary_simulator') #******
tqc = transpile(qc, backend)
job = backend.run(tqc, shots=1000)
result = job.result()
unitary = result.get_unitary(tqc)  #******
print(unitary)
```

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

Using the Aer simulators

In [25]:

```
from qiskit import Aer

for i in range(len(Aer.backends())):
    print(Aer.backends()[i])
    i+=1
print('\n',Aer.backends(),)
```

```
aer_simulator
aer_simulator_statevector
aer_simulator_density_matrix
aer_simulator_stabilizer
aer_simulator_matrix_product_state
aer_simulator_extended_stabilizer
aer_simulator_unitary
aer_simulator_superop
qasm_simulator
statevector_simulator
unitary_simulator
pulse_simulator

 [AerSimulator('aer_simulator'), AerSimulator('aer_simulator_statevector'), AerSimulator('aer_simulator_density_matrix'), AerSimulator('aer_si
mulator_stabilizer'), AerSimulator('aer_simulator_matrix_product_state'), AerSimulator('aer_simulator_extended_stabilizer'), AerSimulator('aer
_simulator_unitary'), AerSimulator('aer_simulator_superop'), QasmSimulator('qasm_simulator'), StatevectorSimulator('statevector_simulator'), U
nitarySimulator('unitary_simulator'), PulseSimulator('pulse_simulator')]
```

Using the Aer Legacy Simulators

`Aer` had enhanced functionality with `AerSimulator` and `PulseSimulator` classes.

Three `Aer` legacy simulators remain:

- `qasm_simulator`
- `statevector_simulator`
- `unitary_simulator`

Code in `Aer` is nerly identical to `BasicAer`, just changing this piece.

In [26]:

```
from qiskit import QuantumCircuit, Aer, transpile

##write circuit
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1)
#qc.measure_all() ### don't want to measure it

##run it
backend = Aer.get_backend('unitary_simulator') #******
tqc = transpile(qc, backend)
job = backend.run(tqc, shots=1000)
result = job.result()
unitary = result.get_unitary(tqc)  #******
print(unitary)
```

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

Using the `aer_simulator` Backend

`aer_simulator` : Main backend for `Aer`

- many types of *simulation methods*: default is `automatic`

Using the `aer_simulator` to hold measurement results

In [29]:

```
from qiskit import QuantumCircuit, BasicAer, transpile, Aer

##write circuit
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1)
qc.measure_all() ### don't want to measure it

##run it
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)
```

```
{'00': 497, '11': 503}
```

Using the `aer_simulator` to calculate and hold a statevector

In [35]:

```
from qiskit import QuantumCircuit, BasicAer, transpile, Aer

##write circuit
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1)
qc.save_statevector() ### this is NEW and different for Aer

##run it
backend = Aer.get_backend('aer_simulator') #******
tqc = transpile(qc, backend)
job = backend.run(tqc, shots=1000)
result = job.result()
statevector = result.get_statevector(tqc) #******
print(statevector)
```

```
[0.70710678+0.j 0.        +0.j 0.        +0.j 0.70710678+0.j]
```

The `save_statevector()` method saves the current simulator quantum state as a statevector. See "Saving state when running a circuit on AerSimulator" for other methods that save simulator state in a quantum circuit.

Saving state when running a circuit on AerSimulator

When running a circuit on an `AerSimulator` backend (see "Using the Aer Simulators"), simulator state may be saved in the circuit instance by using the `QuantumCircuit` methods in Table 1-5.

Table 1-5. Methods used to save simulator state in a circuit instance

Method name	Description
`save_state`	Saves the simulator state as appropriate for the simulation method
`save_density_matrix`	Saves the simulator state as a density matrix
`save_matrix_product_state`	Saves the simulator state as a matrix product state tensor
`save_stabilizer`	Saves the simulator state as a Clifford stabilizer
`save_statevector`	Saves the simulator state as a statevector
`save_superop`	Saves the simulator state as a superoperator matrix of the run circuit
`save_unitary`	Saves the simulator state as a unitary matrix of the run circuit

Using the `aer_simulator` to calculate and hold an unitary

In [36]:

```
from qiskit import QuantumCircuit, BasicAer, transpile, Aer

##write circuit
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1)
qc.save_unitary() ### this is NEW and different for Aer

##run it
backend = Aer.get_backend('aer_simulator') #******
tqc = transpile(qc, backend)
job = backend.run(tqc, shots=1000)
result = job.result()
unitary = result.get_unitary(tqc) #******
print(unitary)
```

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

Using the `aer_simulator` Bfor additional simulation methods

What if not `automatic` ? Simulation methods may be set explicitly

`set_options()`

In [37]:

```
backend = Aer.get_backend("aer_simulator")
backend.set_options(method="density_matrix")
```

pre-configured simulation method

In [38]:

```
backend = Aer.get_backend("aer_simulator_density_matrix")
```

passing into `run()`

In [ ]:

```
backend = Aer.get_backend("aer_simulator")
backend.run(tqc, method="density_matrix")
```

Table of **methods**:

Table 2-1. `AerSimulator` simulation methods

Name	Description
`automatic`	Default simulation method that selects the simulation method automatically based on the circuit and noise model.
`density_matrix`	Density matrix simulation that may sample measurement outcomes from noisy circuits with all measurements at end of the circuit.
`extended_stabilizer`	An approximate simulated for Clifford + T circuits based on a state decomposition into ranked-stabilizer state.
`matrix_product_state`	A tensor-network statevector simulator that uses a Matrix Product State (MPS) representation for the state.
`stabilizer`	An efficient Clifford stabilizer state simulator that can simulate noisy Clifford circuits if all errors in the noise model are also Clifford errors.
`statevector`	Statevector simulation that can sample measurement outcomes from ideal circuits with all measurements at end of the circuit. For noisy simulations each shot samples a randomly sampled noisy circuit from the noise model.
`superop`	Superoperator matrix simulation of an ideal or noisy circuit. This simulates the superoperator matrix of the circuit itself rather than the evolution of an initial quantum state.
`unitary`	Unitary matrix simulation of an ideal circuit. This simulates the unitary matrix of the circuit itself rather than the evolution of an initial quantum state.

Notice that some of the simulation method descriptions in Table 2-1 mention simulating noisy circuits. The `AerSimulator` supports this by allowing a noise model to be supplied that expresses error characteristics of a real or hypothetical quantum device.

Notice the `AerSimulator` backend allows for a noise model to be supplied.

Supplying a noise model to an `aer_simulator` backend

In [40]:

```
from qiskit import QuantumCircuit, Aer, transpile
from qiskit.providers.aer.noise import NoiseModel, depolarizing_error

err_1 = depolarizing_error(0.95, 1)  # build a noise model using qiskit.providers.aer.noise
err_2 = depolarizing_error(0.01, 2)
noise_model = NoiseModel()
noise_model.add_all_qubit_quantum_error(err_1, ['u1','u2','u3'])
noise_model.add_all_qubit_quantum_error(err_2, ['cx'])

##write circuit
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1)
qc.measure_all() ### don't want to measure it

##run it
backend = Aer.get_backend('aer_simulator') #******
backend.set_options(noise_model=noise_model)  #### set_options for noise model
tqc = transpile(qc, backend)
job = backend.run(tqc, shots=1000)
result = job.result()
counts = result.get_counts(tqc) #******
print(counts)
```

```
{'01': 3, '11': 467, '10': 3, '00': 527}
```

Creating an `AerSimulator` from a real device

Simulate a real device

In [42]:

```python
from qiskit import QuantumCircuit, transpile
from qiskit.providers.aer import AerSimulator
from qiskit.test.mock import FakeVigo

##write circuit
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1)
qc.measure_all() ### don't want to measure it

## creating a backend from FakeVigo
device_backend = FakeVigo() ##
backend = AerSimulator.from_backend(device_backend) #******
###

backend.set_options(noise_model=noise_model)  #### set_options for noise model
tqc = transpile(qc, backend)
job = backend.run(tqc, shots=1000)
result = job.result()
counts = result.get_counts(tqc) #******
print(counts)
```

```
{'10': 3, '11': 521, '00': 476}
```

device_backend = provider.get_backend('device')

where provider is a reference to the provider and device is the name of the device

In [ ]:

Monitoring Job Status and Obtaining Results

When running a quantum circuit, a reference to a job (currently a subclass of `qiskit.providers.JobV1`) is returned. This job reference may be used to monitor its status as well as to obtain a reference to a `qiskit.result.Result` instance. This Result reference may be used to obtain relevant results data from the experiment. Table 2-2, Table 2-3, and Table 2-4 describe some of the commonly used methods and attributes in these classes.

When running a quantum circuit, a reference to a job (currently a subclass of qiskit.providers.JobV1) is returned. This job reference may be used to monitor its status as well as to obtain a reference to a qiskit.result.Result instance. This Result reference may be used to obtain relevant results data from the experiment. Table 2-2, Table 2-3, and Table 2-4 describe some of the commonly used methods and attributes in these classes.

Table 2-2. Commonly used `qiskit.providers.JobV1` methods

Method name	Description
`job_id`	Returns a unique identifier for this job.
`backend`	Returns a reference to a subclass of `qiskit.providers.BackendV1` used for this job.
`status`	Returns the status of this job, for example `JobStatus.QUEUED`, `JobStatus.RUNNING`, or `JobStatus.DONE`.
`cancel`	Makes an attempt to cancel the job.
`cancelled`	Returns a boolean that indicates whether the job has been cancelled.
`running`	Returns a boolean that indicates whether the job is actively running on the quantum simulator or device.
`done`	Returns a boolean that indicates whether the job has successfully run.
`in_final_state`	Returns a boolean that indicates whether the job has finished. If so, it is in one of the final states: `JobStatus.CANCELLED`, `JobStatus.DONE`, or `JobStatus.ERROR`
`wait_for_final_state`	Polls the job status for a given duration at a given interval, calling an optional callback method. Returns when the job is in one of the final states, or the given duration has expired.
`result`	Returns an instance of `qiskit.result.Result` that holds relevant results data from the experiment.

NOTE

Methods in Table 2-2 could be leveraged to create a job monitoring facility. There is already a basic job monitoring facility in the `qiskit.tools` package, implemented in the `job_monitor` function.

Table 2-3. Commonly used `qiskit.result.Result` methods

Method name	Description
get_counts	Returns a dictionary containing the count of measurement outcomes per basis state, if available.
get_memory	Returns a list containing a basis state resulting from each shot, if available. Requires that the `memory` option is `True`.
get_statevector	Returns a list of complex probability amplitudes that express a saved statevector, if available.
get_unitary	Returns a square matrix of complex numbers that express a saved unitary, if available.
data	Returns the raw data for an experiment.
to_dict	Returns a dictionary representation of the `results` attribute (see Table 2-4).

Table 2-4. Commonly used `qiskit.result.Result` attributes

Attribute name	Description
`backend_name`	Holds the name of the backend quantum simulator or device.
`backend_version`	Holds the version of the backend quantum simulator or device.
`job_id`	Holds a unique identifier for the job that produced this result.
`results`	List containing results of experiments run. Note that all of our examples run just one circuit at a time.
`success`	Indicates whether experiments ran successfully.

Visualizing Quantum Measurement and States

Using the Transpiler

Quantum information: `quantum_info()`

`quantum_info` module: `qiskit.quantum_info()`

Table 5-1. Classes that represent states in the `qiskit.quantum_info` module

Class name	Description
`Statevector`	Represents a statevector
`DensityMatrix`	Represents a density matrix
`StabilizerState`	Simulation of stabilizer circuits

`Statevector` class

Represents a quantum sv, and constains functionality for initializing and operating on that statevector.

May be instatiated by passing in a `QuantumCircuit` instance.

1. Notice that instead of **running the circuit on a quantum simulator to get a statevector**, we simply create an instance of `Statevector` with the desired `QuantumCircuit`.

In [48]:
```
from qiskit import QuantumCircuit
from qiskit.quantum_info import Statevector

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

statevector = Statevector(qc)
print(statevector.data)
```

```
[0.70710678+0.j 0.        +0.j 0.        +0.j 0.70710678+0.j]
```

1. Another way to create a `Statevector` is to pass in a complex vector

In [50]:
```
import numpy as np
from qiskit.quantum_info import Statevector

statevector = Statevector([1,0,0,1]/np.sqrt(2))
print(statevector.data)
```

```
[0.70710678+0.j 0.        +0.j 0.        +0.j 0.70710678+0.j]
```

1. Yet another way: pass a string of eigenstate ket labels to the `from_label` method:

In [58]:
```
from qiskit.quantum_info import Statevector

statevector = Statevector.from_label('01-')
print(statevector.data)
```

```
[ 0.        +0.j  0.        +0.j  0.70710678+0.j -0.70710678+0.j
  0.        +0.j  0.        +0.j  0.        +0.j  0.        +0.j]
```

In [102…
```
from qiskit.visualization import array_to_latex

statevector = Statevector.from_label('1-')
display(array_to_latex(statevector.data))
statevector.draw('qsphere')
statevector.dim
```

$$\begin{bmatrix} 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}$$

Out[102… 4

Table 5.2 & 5.3

Methods and attributes in `Statevector` class

Table 5-2. Some `Statevector` methods

Method name	Description
`conjugate`	Returns the complex conjugate of the statevector.
`copy`	Creates and returns a copy of the statevector.
`dims`	Returns a tuple of dimensions.
`draw`	Returns a visualization of the `Statevector`, given the desired output method from the following: *text*, *latex*, *latex_source*, *qsphere*, *hinton*, *bloch*, *city*, or *paulivec*. Also see Chapter 3
`equiv`	Returns a boolean indicating whether a supplied `Statevector` is equivalent to this one, up to a global phase.
`evolve`	Returns a quantum state evolved by the supplied operator. Also see "Using Quantum Information Operators".
`expand`	Returns the reverse-order tensor product state of this statevector and a supplied `Statevector`.
`expectation_value`	Computes and returns the expectation value of a supplied operator.
`from_instruction`	Returns the `Statevector` output of a supplied `Instruction` or `QuantumCircuit` instance.
`from_label`	Instantiates a `Statevector` 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.
`inner`	Returns the inner product of this statevector and a supplied `Statevector`.
`is_valid`	Returns a boolean indicating whether this statevector has norm 1.
`measure`	Returns the measurement outcome as well as post-measure state.
`probabilities`	Returns the measurement probability vector.
`probabilities_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 statevector represents a pure quantum state.
`purity`	Returns a number from 0 to 1 indicating the purity of this quantum state. 1.0 indicates that this statevector represents a pure quantum state.
`reset`	Resets to the 0 state.
`reverse_qargs`	Returns a `Statevector` with reversed basis state ordering.
`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 statevector and a supplied `Statevector`.
`to_dict`	Returns the statevector as a dictionary.
`to_operator`	Returns a rank-1 projector operator by taking the outer product of the statevector with its complex conjugate.
`trace`	Returns the trace of the quantum state as if it was represented as a density matrix. Also see "Using the DensityMatrix Class"

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

In [106…

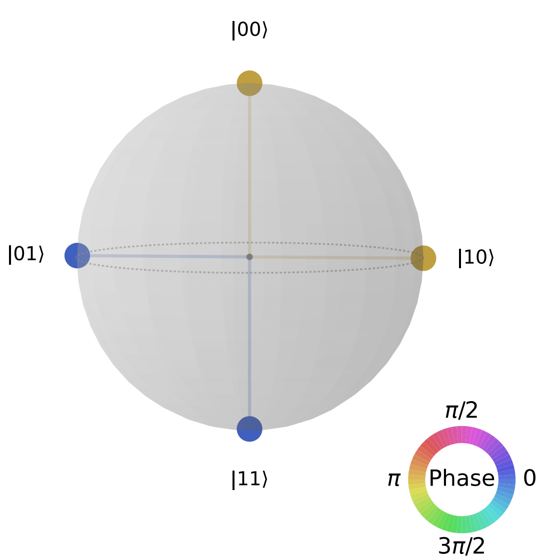
```
from qiskit.quantum_info import Statevector

statevector = Statevector.from_label('+-')
display(array_to_latex(statevector.data))
```

$$\begin{bmatrix} \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \end{bmatrix}$$

In [107…

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

Out[107…



In [108…

```
print(statevector.probabilities())
```

```
[0.25 0.25 0.25 0.25]
```

In [110…

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

```
{'00': 238, '01': 247, '10': 254, '11': 261}
```

`DensityMatrix` class

quantum density matrix. Functionality for initializing and operating on the density matrix.

Matrix $\in \mathbb{C}$: enables `DensityMatrix` to represent **mixed states**, ensemble of 2 or more quantum states.

In [119…

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

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

```
[[ 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]]
```

Table 5.4 & 5.5

Methods and attributes of `DensityMatrix`

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.
`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".
`expand`	Returns the reverse-order tensor product state of this density matrix and a supplied `DensityMatrix`.
`expectation_value`	Computes and returns the expectation value of a supplied operator.
`from_instruction`	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.
`probabilities_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.
`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 `DensityMatrix` methods

Mixed state, instatiated using `from_label`

In [138…

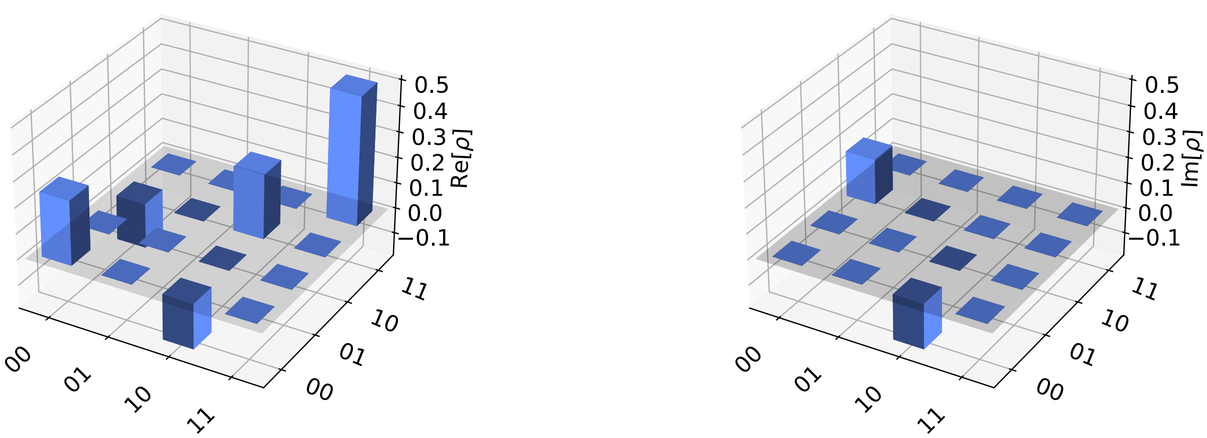
```
dens_mat = 0.5*DensityMatrix.from_label('11') + 0.5*DensityMatrix.from_label('+0')
print(dens_mat.data)
```

```
[[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]]
```

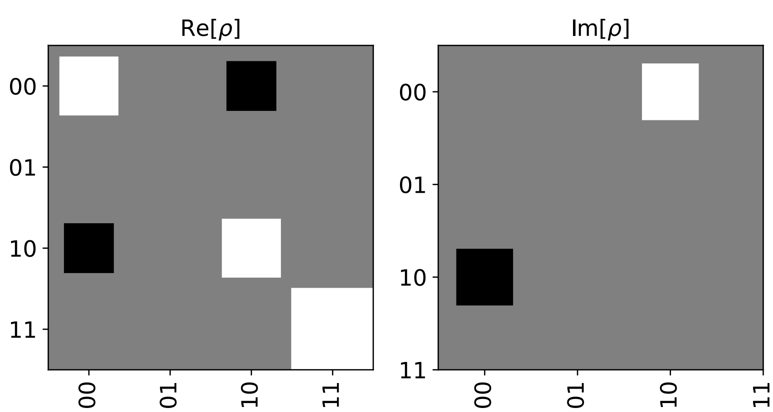
Use `evolve` to evolve the state with an operator and draw using city matrix

In [141…

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



Out[141…



`probabilities` and `sample_counts`

In [142…

```
print(dens_mat.probabilities())
```

```
[0.25 0.   0.25 0.5 ]
```

In [144…

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

```
{'00': 229, '10': 231, '11': 540}
```

Quantum Information Operators

Table 5.6

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

Focus on

- `Operator`
- `Pauli`

`Operator` class

Represents a quantum information operator, modeled by a matrix. Can be placed into a `QuantumCircuit` with the `append` method.

Can be instantiated in many ways:

1. passing in a `QunatumCircuit` instance
2. passing the desired complex vector
3. passing a `Pauli`
4. passing an `Instruction` or `Gate` object

Let's see these 4:

In [147…

```
#1 QuantumCircuit instance

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

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

```
array([[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 it is the **unitary** of the circuit. Can be used to obtain the unitary without running it on a quantum simulator!

In [149…

```
#2 desired complex vector

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

```
[[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]]
```

In [162…

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

```
[[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]]
```

In [165…

```
#4 Instruction or Gate

op_CP = Operator(CPhaseGate(np.pi/4))
display(array_to_latex(op_CP.data))
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{\sqrt{2}}(1+i) \end{bmatrix}$$

Table 5.7 & 5.8

Methods and attributes of `Operator`

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`.
`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.

`Pauli` class

multi-qubit Pauli operator in which each qubit is an X, Y, Z, or I Pauli.

May be instantiated in several ways:

1. passing in a string containing Pauli ops preceded by an optional phase coefficient
2. passing a `QuantumCircuit` containing only Paulis

In [168…
```
#1 string of Paulis and phase

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

```
[[ 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]]
```

In [172…
```
#2 quantum circuit

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

pauli_XZ = Pauli(qc)
print(pauli_XZ.equiv(pauli_piXZ))

## differ only by a global phase
```

```
True
```

Table 5-9 & 5.10

Methods and attributes of `Pauli`

Table 5-9. Some `Pauli` methods

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

Quantum Information Channels

Table 5.11

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

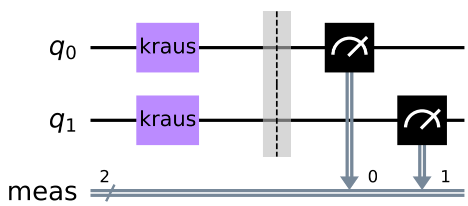
For instance:

`Kraus` class: model noise quantum channel whose qubits flip about 10% of the time. Instance creted with a matrix that model this bit-flip behavior and appended to `QuantumCircuit`

In [174…

```
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()
```

Out[174…



To see the results of the changing channel, run through simulator

In [177…

```
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)
```

```
{'01': 80, '11': 9, '10': 94, '00': 817}
```

In [ ]:

Quantum Information Measures

Table 5.12

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.

`state_fidelity`

`state_fidelity()` takes two `Statevector` or `DensityMatrix` instances and returns the state fidelity between them.

Ex: rotate $\pi/4$, 85% state fidelity: Type in the book? $\pi/8 \rightarrow \pi/4$??

In [179…

```
sv_a = Statevector.from_label('+')
sv_b = sv_a.evolve(Operator.from_label('T'))

print(state_fidelity(sv_a, sv_b))
```

```
0.8535533905932733
```

In [ ]:

In [178…

```
### `state_fidelity`
```

In [ ]:

Qiskit Circuit Library Standard Operations

In [ ]:

In [ ]:

In [ ]: