Intro to Qiskit

ECE 592 / CSC 591 - Fall 2019
Qiskit = IBM QC Dev Platform

- **Terra**: Composing programs using circuits and pulses
- **Aqua**: Building algorithms and applications
- **Aer**: Simulators, emulators, and debuggers
- **Ignis**: Addressing errors and noise
Qiskit Terra

• Build
  • Create circuit out of registers, gates

• Compile
  • Translate to QASM, then to backend instructions

• Execute
  • Backends = simulators (Aer), hardware
Building a Circuit

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

ClassicalRegister
  • Collection of bits
  • Used as the receiver of measurements on qubits

QuantumCircuit
  Starts with set of registers
  Add gates specifying registers/qubits as arguments
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit

defi qreg = QuantumRegister(3)  # a 3-qubit register
creg = ClassicalRegister(3)  # a 3-bit classical register
qc = QuantumCircuit(qreg,creg)  # create a circuit

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

<table>
<thead>
<tr>
<th>Quantum Gate</th>
<th>...on qubits</th>
<th>...on register</th>
</tr>
</thead>
<tbody>
<tr>
<td>X (NOT)</td>
<td><code>qc.x(qreg[0])</code></td>
<td><code>qc.x(qreg)</code></td>
</tr>
<tr>
<td>Hadamard</td>
<td><code>qc.h(qreg[0])</code></td>
<td><code>qc.h(qreg)</code></td>
</tr>
<tr>
<td>CNOT</td>
<td><code>qc.cx(qreg[0], qreg[1])</code></td>
<td>--</td>
</tr>
<tr>
<td>Toffoli</td>
<td><code>qc.ccx(qreg[0], qreg[1], qreg[2])</code></td>
<td>--</td>
</tr>
<tr>
<td>Phase shift</td>
<td><code>qc.u1(angle, qreg[0])</code></td>
<td><code>qc.u1(angle, qreg)</code></td>
</tr>
<tr>
<td>Swap</td>
<td><code>qc.swap(qreg[0], qreg[1])</code></td>
<td>--</td>
</tr>
<tr>
<td>Measure</td>
<td><code>qc.measure(qreg[0], creg[0])</code></td>
<td><code>qc.measure(qreg)</code></td>
</tr>
<tr>
<td>Reset</td>
<td><code>qc.reset(qreg[0])</code></td>
<td><code>qc.reset(qreg)</code></td>
</tr>
</tbody>
</table>

summary_of_quantum_operations.ipynb
### Other Circuit Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qc.barrier()</td>
<td>Completes operations before proceeding. Can specify registers, qubits.</td>
</tr>
<tr>
<td>qc.add(regs)</td>
<td>Add register(s) to circuit.</td>
</tr>
<tr>
<td>qc.combine(circuit)</td>
<td>Appends circuit (if compatible). Creates new circuit (qc + circuit) and returns it.</td>
</tr>
<tr>
<td>qc.qasm()</td>
<td>Returns a string containing the QASM representation of circuit.</td>
</tr>
</tbody>
</table>
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit

q = QuantumRegister(2)
c = ClassicalRegister(2)
qc = QuantumCircuit(q, c)

qc.h(q[0]) # Hadamard on first qubit
qc.cx(q[0], q[1]) # CNOT to entangle
# creates a Bell state
qc.measure(q, c)
Compiling and Running

- **Provider**
  - Facilitates access to a selection of backends
  - Aer Provider
    - simulators, running locally on your machine
  - IBM Q Provider
    - hardware, remote simulator

- **Backend**
  - Runs a compiled program (Qobj) and reports result

- **Job**
  - The result of an execution
  - Asynchronous -- query job to see status
  - Get result when complete
Backends

• To compile/execute a circuit, must specify a backend.

• Simulators:
  • Local (Aer):
    qasm_simulator -- emulates a machine with/without noise, multi-shot
    statevector_simulator -- single shot, returns state vector
    unitary_simulator -- returns unitary matrix represented by circuit
  • IBMQ: ibm_qasm_simulator

• Hardware:
  • IBMQ provider -- to be discussed later
# Job Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>job.status()</code></td>
<td>Returns current status.</td>
</tr>
<tr>
<td><code>job.done()</code></td>
<td>Returns True if done, False if not.</td>
</tr>
<tr>
<td><code>job.id()</code></td>
<td>Identifier (remote provider only)</td>
</tr>
<tr>
<td><code>job.result()</code></td>
<td>Results from completed job.</td>
</tr>
<tr>
<td><code>job.result().get_counts()</code></td>
<td>Instances of various measured states, e.g. '{'111': 512, '000': 512}'</td>
</tr>
<tr>
<td><code>job_monitor(job)</code></td>
<td>Loop that waits for job to complete, periodically printing the job status.</td>
</tr>
</tbody>
</table>
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
from qiskit import Aer, execute
from qiskit.tools.visualization import plot_histogram

# ... deleted circuit building commands...
qc.measure(q,c)

backend = Aer.get_backend('qasm_simulator')
job = execute(qc, backend, shots=512)  # shots default = 1024
result = job.result()
print(result.get_counts())
plot_histogram(result.get_counts())
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
from qiskit import Aer, execute
from qiskit.tools.visualization import plot_histogram

# ... deleted circuit building commands...
qc.measure(q,c)

backend = Aer.get_backend('qasm_simulator')
job = execute(qc, backend, shots=512)  # shots default = 1024
result = job.result()
print(result.get_counts())
plot_histogram(result.get_counts())
Example 2

What is the result of this measurement?

After the Toffoli gate, are the qubits entangled?
Example 2

What is the result of this measurement?

After the Toffoli gate, are the qubits entangled?
Example 3

Implement a quantum circuit that checks whether two qubits are equal (in the computational basis).

Use qiskit to demonstrate that your circuit works.
Qiskit Summary

- Create quantum and classical registers.
- Create quantum circuit, adding registers.
- Add gates and measurements to circuits.

- Choose backend from provider.
- Execute circuit -- compiles circuit to match specifics of backend.
- Get results from job.