Intro to Qiskit

ECE 592/CSC 591 – Fall 2018

Qiskit = IBM QC 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

• Compiler
  • Translate to QASM, then to backend instructions

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

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

### 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>qc.x(qreg[0])</td>
<td>Yes</td>
</tr>
<tr>
<td>Hadamard</td>
<td>qc.h(qreg[0])</td>
<td>Yes</td>
</tr>
<tr>
<td>CNOT</td>
<td>qc.cx(qreg[0],qreg[1])</td>
<td>--</td>
</tr>
<tr>
<td>Toffoli</td>
<td>qc.ccz(qreg[0], qreg[1], qreg[2])</td>
<td>--</td>
</tr>
<tr>
<td>Unitary gates</td>
<td>qc.u3(pi/2,pi/2,pi/2,qreg[0])</td>
<td>Yes</td>
</tr>
<tr>
<td>Swap</td>
<td>qc.swap(qreg[0],qreg[1])</td>
<td>--</td>
</tr>
<tr>
<td>Measure (not a gate)</td>
<td>qc.measure(qreg[0],creg[0])</td>
<td>Yes</td>
</tr>
<tr>
<td>Reset (not a gate)</td>
<td>qc.reset(qreg[0])</td>
<td>Yes</td>
</tr>
</tbody>
</table>
## 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>

```python
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
q = QuantumRegister(2)
c = ClassicalRegister(2)
qc = QuantumCircuit(q, c)
qc.h(q[1])        # Hadamard on first qubit
qc.cx(q[1],q[0])  # CNOT to entangle
qc.measure(q,c)
```
Compiling and Running

• Provider
  • Facilitates access to a selection of backends
  • LocalProvider by default
    • simulators, running locally on your machine
  • Use register command to connect to 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_qasm_simulator
  local_unitary_simulator (no measurement)
  local_statevector_simulator (no measurement)

• Hardware:
  ibmq_20_tokyo
Getting info about Backends

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>available_backends()</td>
<td>Complete list</td>
</tr>
<tr>
<td>available_backends({'local': False})</td>
<td>Non-local</td>
</tr>
<tr>
<td>available_backends({'simulator': False})</td>
<td>Hardware only</td>
</tr>
<tr>
<td>least_busy(available_backends({'simulator': False}))</td>
<td>Least-loaded hardware</td>
</tr>
</tbody>
</table>

Also, commands to find out lots of detailed information about each backend.

- Number of qubits
- Interconnection
- Calibration information

working_with_backends.ipynb

Compling and Running

compile
- Create a Qobj for a specific backend

backend.run
- Executes on specific backend
- Creates a Job

eexecute
- Compile and run in one step
- Creates a Job
Job Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>job.status()</td>
<td>Returns current status.</td>
</tr>
<tr>
<td>job.done()</td>
<td>Returns True if done, False if not.</td>
</tr>
<tr>
<td>job.id()</td>
<td>Identifier (remote provider only)</td>
</tr>
<tr>
<td>job.cancel()</td>
<td>Cancel job – only on IBM Q premium devices.</td>
</tr>
<tr>
<td>job.result()</td>
<td>Results from completed job.</td>
</tr>
<tr>
<td>job.result().get_counts()</td>
<td>Instances of various measured states, e.g.</td>
</tr>
<tr>
<td></td>
<td>{'111': 512, '000': 512}</td>
</tr>
</tbody>
</table>

```python
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
from qiskit import execute, register, available_backends, least_busy

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

backend = 'local_qasm_simulator'
job = execute(qc, backend, shots=512)  # shots default = 1024
result = job.result()
print(result.get_counts())```
Using Hardware

- Qconfig.py
  - Keep API token and other information needed to register
- Register
  - Use register command to connect to IBM Q provider
- Specify hardware backend

```python
# Qconfig.py
APItoken = '2ca0267e68e032d873e68a1a4dc04727facef15c75c97cb3028de35b0baa33bf942822708029a180c04ddd773713769deedd4476f753fe365f3775f5bd734334'

cfg = {
    'url': 'https://q-console-api.mybluemix.net/api',
    'hub': 'ibm-q-ncsu',
    'group': 'nc-state',
    'project': 'on-boarding'
}

if 'APItoken' not in locals():
    raise Exception('Please set up your access token. See Qconfig.py.')
```
```python
import sys
try:
    sys.path.append("../") # go to parent dir (if needed)
    import Qconfig
    qx_config = {
        "APItoken": Qconfig.APItoken,  
        "url": Qconfig.config['url'],
        "hub": Qconfig.config['hub'],
        "group": Qconfig.config['group'],
        "project": Qconfig.config['project']
    }
    print('Qconfig loaded from %s.' % Qconfig.__file__)
except:
    print('Unable to load Qconfig')

register(qx_config['APItoken'], qx_config['url'], qx_config['hub'], qx_config['group'], qx_config['project'])

device_name = least_busy(available_backends({'simulator': False}))
my_backend = get_backend(device_name)
job = execute(qc, backend=my_backend, shots=1024)
lapse = 0
interval = 10
while not job.done:
    print('Status @ {} seconds'.format(interval * lapse))
    print(job.status)
    time.sleep(interval)
    lapse += 1
    print(job.status)
    result = job.result()
    print(result.get_counts())
```

If you're using public machines, only need token and url.