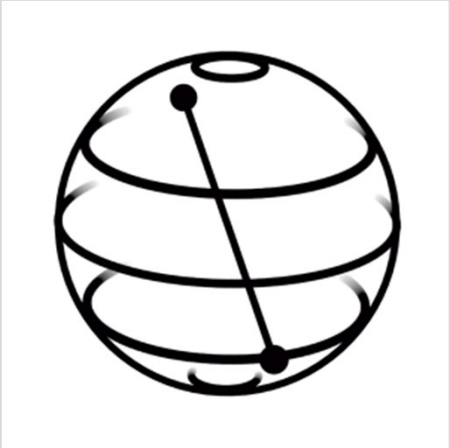


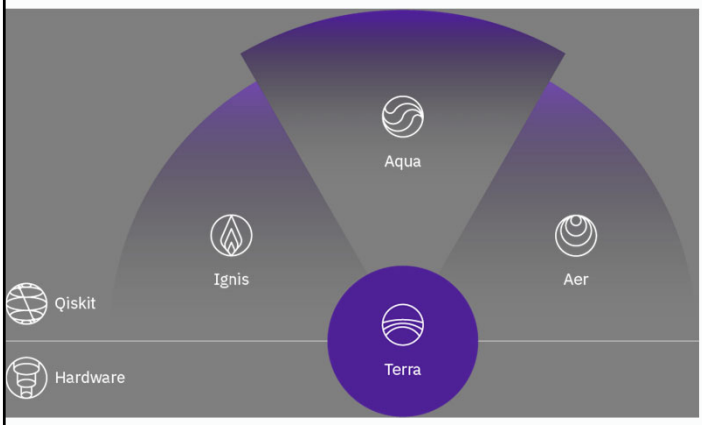
Intro to Qiskit

ECE 592/CSC 591 – Fall 2018

NC STATE UNIVERSITY Electrical & Computer Engineering

@NCStateECE

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

Quantum Gate	...on qubits	...on register?
X (NOT)	`qc.x(qreg[0])`	Yes
Hadamard	`qc.h(qreg[0])`	Yes
CNOT	`qc.cx(qreg[0],qreq[1])`	--
Toffoli	`qc.ccz(qreg[0], qreg[1], qreg[2])`	--
Unitary gates	`qc.u3(pi/2,pi/2,pi/2,qreq[0])`	Yes
Swap	`qc.swap(qreg[0],qreg[1])`	--
Measure (not a gate)	`qc.measure(qreg[0],creg[0])`	Yes
Reset (not a gate)	`qc.reset(qreg[0])`	Yes

quantum_gates_and_linear_algebra.ipynb

Other Circuit Operations

Operation	Description
`qc.barrier()`	Completes operations before proceeding. Can specify registers, qubits.
`qc.add(`*regs*`)`	Add register(s) to circuit.
`qc.combine(`*circuit*`)`	Appends circuit (if compatible). Creates new circuit (qc + circuit) and returns it.
`qc.extend(`*circuit*`)`	Appends circuit (if compatible). Modifies qc.
`qc.qasm()`	Returns a string containing the QASM representation of circuit.

```
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

Operation	Description
`available_backends()`	Complete list
`available_backends({'local': False})`	Non-local
`available_backends({'simulator': False})`	Hardware only
`least_busy(available_backends({'simulator': False}))`	Least-loaded hardware

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

execute

- Compile and run in one step
- Creates a Job

Job Operations

Operation	Description
`job.status()`	Returns current status.
`job.done()`	Returns True if done, False if not.
`job.id()`	Identifier (remote provider only)
`job.cancel()`	Cancel job -- only on IBM Q premium devices.
`job.result()`	Results from completed job.
`job.result().get_counts()`	Instances of various measured states, e.g. `{'111': 512, '000': 512}`

```
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

```
# Qconfig.py
APItoken =
'2ca0267e68e032d873e68a1a4dc04727facef15c75c97cb3028de35bdbaa33bf942822
708029a180c04ddd773713769deedd4476f753fe365f3775f5bd734334'


config = {
    '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.')
```

```
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')
```

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

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

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

```
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())
```