**Rigetti vs. IBM Q, A Comparative Analysis**

For those who only even have the slightest interest in the budding field of quantum computing have probably heard of IBM’s advancements and efforts in IBM Q. Those newer to the field may have minimal, or even no knowledge of another company making strides, a start-up from Berkeley called Rigetti.

Both of these companies have developed their own quantum processors. As of April 2018, Rigetti’s quantum processor is 20 qubits while IBM’s is also commercially offering a quantum processor of 20 qubits. Rigetti provides access to their quantum chip (and quantum simulators) by providing an API called Forest while IBM gives their QISkit SDK which also lets users access their quantum chip through an API interface. IBM Q works with the quantum instruction language QASM to configure quantum circuits on their quantum processor or in simulators. On the other hand, Rigetti uses their own quantum instruction language called Quil to build quantum circuits.

**Overview Comparison:**

**Rigetti:**

Forest comes with Quil, the quantum instruction language, pyQuil, a python library to help write and run Quil code, access to a QVM (Quantum Virtual Machine), and a special connection to QPU (Rigetti’s 20 qubit Quantum Processor). The QPU is available for customers with upgraded access. With this access, it is possible to run “analog” experiments to understand the performance of their qubits and characterize them based on their performance in the presence of noise. The resonant frequencies of qubits 0-4 and 10-14 are tunable to the users need, while everything else is fixed. Normal users are required to run their programs on a simulator called Rigetti’s QVM (quantum virtual machine) with up to 30 virtual qubits.

To write quantum programs, Rigetti provides their pyQuil sdk which allows customers to write their quantum algorithms in Python. Rigetti also provides a separate repo of quantum algorithms built using pyQuil called Grover. On the client side, quantum algorithms written by the user are trans compiled into Quil code. This Quil code is sent over the cloud for which it is used to compile into quantum hardware and execute on their quantum processor or on their simulator.

Quil, as a quantum instruction language, is portable, foundational, and hybrid in nature. With Quil, quantum and classical processors have a “symbiotic” relationship, which makes the Forest API good for leveraging classical/quantum hybrid algorithms on near-term quantum devices (devices with anywhere between 3-100 noisy qubits) instead of running “pure” quantum algorithms such as Shor’s algorithm. PyQuil also grants easy access to importing the most important gates, such as the Pauli operators. It is also very easy to implement a new program by simply initializing a Program() object with a set of quantum operations on a base state. In addition to running programs on the QVM or QPU, users can also directly access the Quil compiler for further investigation into how quantum programs can be compiled to target specific instruction set architectures.

Rigetti's compiler is now accessible through a dedicated API, this lets you experiment compiling a quantum program into different hardware architectures. Rigetti has also introduced built-in tools for quantum state and process tomography. This will make it easier to debug the programs created. In addition, Rigetti also provides jsQuil, if there is a preference to use JS over Python. PyQuil is also open source. This feature is nice for debugging purposes.

**IBM Q:**

Much of IBM Q is similar to Rigetti Forest. IBM’s QISKit project includes the QISKit SDK, the QISKit API, and a collection of Jupyter notebooks using QISKit as examples. At a high level, we generate a quantum program using the python interface of QISKit. Upon compilation, the python is used to generate QASM code describing quantum circuits. Although IBM also uses both classical and quantum processors in QASM to define their quantum circuits, they place much less emphasis on hybrid computation within their python interface relative to Rigetti.

**Python Interface:**

**Rigetti:**

Defining a quantum program using Rigetti’s pyQuil SDK is as easy as importing a few modules provided by the SDK, instantiating a quantum program object with the number of qubits you plan to use, and using imported operators or “gates” to apply and operator to the system or to take a measurement. A measurement returns a wavefunction describing the state of the qubits used in the program.

For example, I pulled this example from their documents:

# Imports for pyQuil (ignore for now)

import numpy as np

from pyquil.quil import Program

from pyquil.api import QVMConnection

quantum\_simulator = QVMConnection()

# pyQuil is based around operations (or gates) so we will start with the most

# basic one: the identity operation, called I. I takes one argument, the index

# of the qubit that it should be applied to.

from pyquil.gates import I

# Make a quantum program that allocates one qubit (qubit #0) and does nothing to it

p = Program(I(0))

# Quantum states are called wavefunctions for historical reasons.

# We can run this basic program on our connection to the simulator.

# This call will return the state of our qubits after we run program p.

# This api call returns a tuple, but we'll ignore the second value for now.

wavefunction = quantum\_simulator.wavefunction(p)

# wavefunction is a Wavefunction object that stores a quantum state as a list of amplitudes

alpha, beta = wavefunction

print("Our qubit is in the state alpha={} and beta={}".format(alpha, beta))

print("The probability of measuring the qubit in outcome 0 is {}".format(abs(alpha)\*\*2))

print("The probability of measuring the qubit in outcome 1 is {}".format(abs(beta)\*\*2))

This code describes a quantum program that uses one qubit and applies the Identity operator to it. As you can see, we import the Program and QVMConnection classes to both create a quantum program and test it. We also import the Identity operator from the pyQuil.gates module.

PyQuil has several provided operators for us to use, such as the Identity operator shown previously, they include:

* The Identity operator
* The Pauli Operators
* Hadamard operator
* Phase gates
* Controlled phase gates
* Cartesian rotation gates
* Controlled gates
* SWAP gates

In addition to this, new gates are easily defined by creating your own numpy arrays and using the defgate() function to add the gate to your program. To define parametric gates, the only difference is that you would use import the DefGate class and create an instantiation using a numpy array with a defined parameter.

Measurements in pyQuil project the state vector onto one of the basic outcomes, while optionally storing the outcome of the measurement in a classical bit. It’s important to note that while optional, we cannot actually check the wavefunction, instead, we can only check the classical bits that are affected by the measurements, which we can do with the optional measurement feature. This gives us the flexibility to also introduce “classical control” of our quantum programs, like using the state of classical bits to determine what operations to run on our qubits.

Some features of quantum programs constructed using pyQuil’s python interface include:

1. Multiple instructions can be chained together in any order
2. It is easy to remove or add extra quantum instructions to quantum programs
3. The Rigetti QVM also has support for emulating noise models.
4. Parametric programs are a convenient way to allow you to use Python functions to generate templates of Quil programs. The great thing about Parametric programs is that they cache computations that happen in their “wrapped” python programs so that the templates in Quil can be efficiently substituted.
5. Programs that use more than 19 qubits (or take longer than 10 seconds to run) are rejected by the QVM. To get around this, you must instantiate a program with the use\_queue parameter set to True. There is also an asynchronous mode of interaction in these queues.
6. To get the current status of a job you can use the get\_job() method on the QVM.

Public Consultation:

I have asked a couple of people involved in the quantum computing community, and from their experience, Rigetti Forest is easier to use, provides much more readable code (with both its python interface and its quantum processing language). In terms of debugging, pyQUIL allows the user to print the wavefunction of a system.