pyQuil Documentation

Release 1.4.0

Rigetti Computing

Nov 22, 2017

Contents

CHAPTER 1

Overview

Welcome to pyQuil!

pyQuil is part of the Rigetti Forest toolkit for **quantum programming in the cloud**, which is currently in public beta. If you are interested in obtaining an API key for the beta, please reach out by signing up here. We look forward to hearing from you.

pyQuil is an open source Python library developed at Rigetti Quantum Computing that constructs programs for quantum computers. The source is hosted on GitHub.

More concretely, pyQuil produces programs in the Quantum Instruction Language (Quil). For a full description of Quil, please refer to the whitepaper *A Practical Quantum Instruction Set Architecture*.[1] Quil is an opinionated quantum instruction language: its basic belief is that in the near term quantum computers will operate as coprocessors, working in concert with traditional CPUs. This means that Quil is designed to execute on a Quantum Abstract Machine that has a shared classical/quantum architecture at its core.

Quil programs can be executed on a local or cloud-based Quantum Virtual Machine. This is a classical simulation of a quantum processor that can simulate up to 36 qubits. The default access key allows you to run simulations of up to 26 qubits. These simulations can be run through either synchronous API calls, or through an asynchronous job queue for larger programs. More information about the QVM can be found at Overview of the Quantum Virtual Machine.

If you are already familiar with quantum computing, then feel free to proceed to Installation and Getting Started.

Otherwise, take a look at our Brief Introduction to Quantum Computing, where the basics of quantum computing are introduced using Quil and the Quantum Abstract Machine on which it runs.

[1] https://arxiv.org/abs/1608.03355

CHAPTER 2

Brief Introduction to Quantum Computing

With every breakthrough in science there is the potential for new technology. For over twenty years, researchers have done inspiring work in quantum mechanics, transforming it from a theory for understanding nature into a fundamentally new way to engineer computing technology. This field, quantum computing, is beautifully interdisciplinary, and impactful in two major ways:

1. It reorients the relationship between physics and computer science. Physics does not just place restrictions on what computers we can design, it also grants new power and inspiration.
2. It can simulate nature at its most fundamental level, allowing us to solve deep problems in quantum chemistry, materials discovery, and more.

Quantum computing has come a long way, and in the next few years there will be significant breakthroughs in the field. To get here, however, we have needed to change our intuition for computation in many ways. As with other paradigms - such as object-oriented programming, functional programming, distributed programming, or any of the other marvelous ways of thinking that have been expressed in code over the years - even the basic tenants of quantum computing opens up vast new potential for computation.

However, unlike other paradigms, quantum computing goes further. It requires an extension of classical probability theory. This extension, and the core of quantum computing, can be formulated in terms of linear algebra. Therefore, we begin our investigation into quantum computing with linear algebra and probability.

2.1 From Bit to Qubit

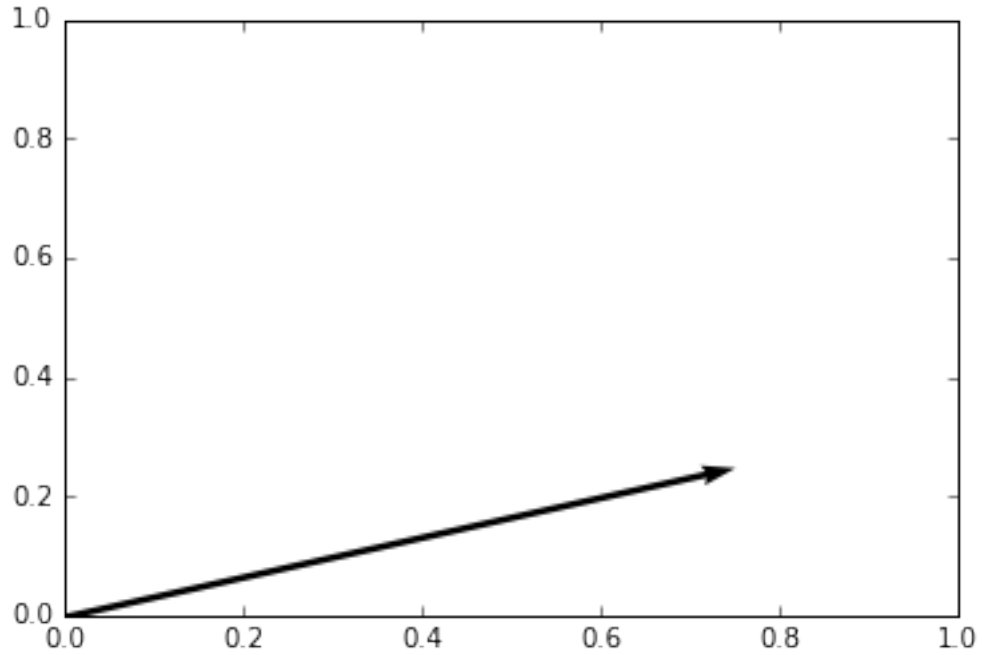
2.1.1 Probabilistic Bits as Vector Spaces

From an operational perspective, a bit is described by the results of measurements performed on it. Let the possible results of measuring a bit (0 or 1) be represented by orthonormal basis vectors \(\vec{0}\) and \(\vec{1}\). We will call these vectors **outcomes**. These outcomes span a two-dimensional vector space that represents a probabilistic bit. A probabilistic bit can be represented as a vector

$$\vec{v} = a\,\vec{0} + b\,\vec{1},$$

where \(a\) represents the probability of the bit being 0 and \(b\) represents the probability of the bit being 1. This clearly also requires that \(a+b=1\). In this picture the **system** (the probabilistic bit) is a two-dimensional real vector space and a **state** of a system is a particular vector in that vector space.

```
import numpy as np
import matplotlib.pyplot as plt
outcome_0 = np.array([1.0, 0.0])
outcome_1 = np.array([0.0, 1.0])
a = 0.75
b = 0.25

prob_bit = a*outcome_0 + b*outcome_1

X,Y = prob_bit
plt.figure()
ax = plt.gca()
ax.quiver(X,Y,angles='xy',scale_units='xy',scale=1)
ax.set_xlim([0,1])
ax.set_ylim([0,1])
plt.draw()
plt.show()
```



Given some state vector, like the one plotted above, we can find the probabilities associated with each outcome by projecting the vector onto the basis outcomes. This gives us the following rule:

$$\Pr(0) = \vec{v}^T.\vec{0} = a$$
$$\Pr(1) = \vec{v}^T.\vec{1} = b,$$

where Pr(0) and Pr(1) are the probabilities of the 0 and 1 outcomes respectively.

2.1.2 Dirac Notation

Physicists have introduced a convenient notation for the vector transposes and dot products we used in the previous example. This notation, called Dirac notation in honor of the great theoretical physicist Paul Dirac, allows us to define

$$\vec{v} = |\, v\rangle$$
$$\vec{v}^T = \langle v\,|$$
$$\vec{u}^T.\vec{v} = \langle u\,|\, v\rangle.$$

Thus, we can rewrite our "measurement rule" in this notation as

$$Pr(0) = \langle v\,|\, 0\rangle = a$$
$$Pr(1) = \langle v\,|\, 1\rangle = b.$$

We will use this notation throughout the rest of this introduction.

2.1.3 Multiple Probabilistic Bits

This vector space interpretation of a single probabilistic bit can be straightforwardly extended to multiple bits. Let us take two coins as an example (labelled 0 and 1 instead of H and T since we are programmers). Their states can be represented as

$$|\, u\rangle = \frac{1}{2}|\, 0_u\rangle + \frac{1}{2}|\, 1_u\rangle$$
$$|\, v\rangle = \frac{1}{2}|\, 0_v\rangle + \frac{1}{2}|\, 1_v\rangle,$$

where \(1_u\) represents the 1 outcome on coin \(u\). The **combined system** of the two coins has four possible outcomes \(\{ 0_u0_v,\;0_u1_v,\;1_u0_v,\;1_u1_v \}\) that are the basis states of a larger four-dimensional vector space. The rule for constructing a **combined state** is to take the tensor product of individual states, e.g.

$$|\, u\rangle \otimes |\, v\rangle = \frac{1}{4}|\, 0_u0_v\rangle + \frac{1}{4}|\, 0_u1_v\rangle + \frac{1}{4}|\, 1_u0_v\rangle + \frac{1}{4}|\, 1_u1_v\rangle.$$

Then, the combined space is simply the space spanned by the tensor products of all pairs of basis vectors of the two smaller spaces.

We will talk more about these larger spaces in the quantum case, but it is important to note that not all composite states can be written as tensor products of sub-states. (Consider the state \(\frac{1}{2}|\,0_u0_v\rangle + \frac{1}{2}|\,1_u1_v\rangle\).) In general, the combined state for \(n\) probabilistic bits is a vector of size \(2^n\) and is given by \(\bigotimes_{i=0}^{n-1}|\,v_i\rangle\).

2.1.4 Qubits

Quantum mechanics rewrites these rules to some extent. A quantum bit, called a qubit, is the quantum analog of a bit in that it has two outcomes when it is measured. Similar to the previous section, a qubit can also be represented in a vector space, but with complex coefficients instead of real ones. A qubit **system** is a two-dimensional complex vector space, and the **state** of a qubit is a complex vector in that space. Again we will define a basis of outcomes \(\{|\,0\rangle, |\,1\rangle\}\) and let a generic qubit state be written as

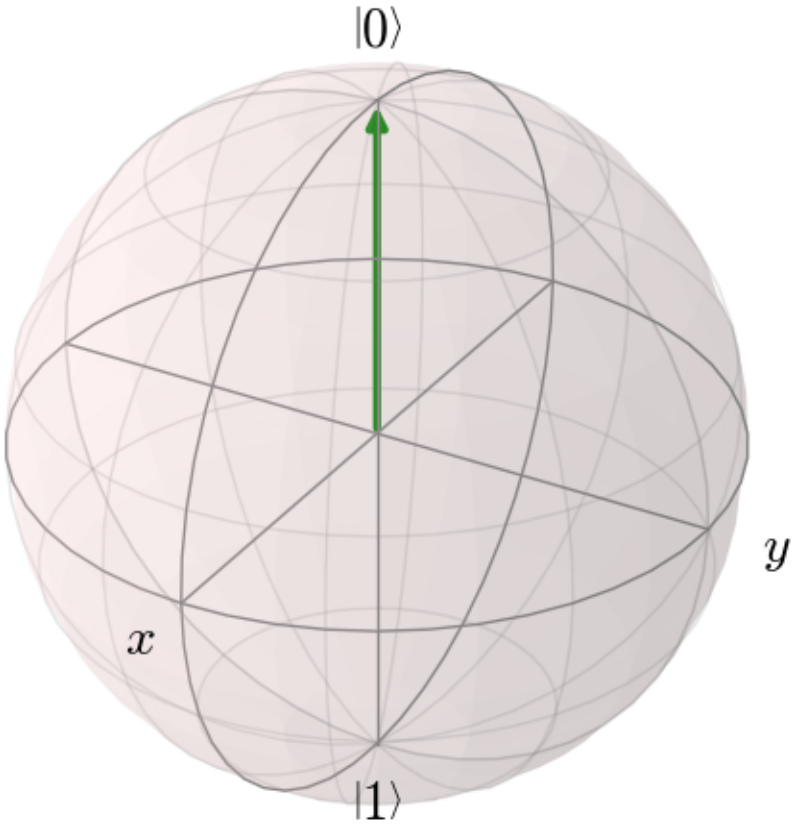
$$\alpha|\, 0\rangle + \beta|\, 1\rangle.$$

Since these coefficients can be imaginary, they cannot be simply interpreted as probabilities of their associated outcomes. Instead we rewrite the rule for outcomes in the following manner:

$$\Pr(0) = |\langle v\,|\, 0\rangle|^2 = |\alpha|^2$$
$$\Pr(1) = |\langle v\,|\, 1\rangle|^2 = |\beta|^2,$$

and as long as \(|\alpha|^2 + |\beta|^2 = 1\) we are able to recover acceptable probabilities for outcomes based on our new complex vector.

This switch to complex vectors means that rather than representing a state vector in a plane, we instead to represent the vector on a sphere (called the Bloch sphere in quantum mechanics literature). From this perspective the quantum state corresponding to an outcome of 0 is represented by:



Notice that the two axes in the horizontal plane have been labeled x and y, implying that z is the vertical axis (not labeled). Physicists use the convention that a qubit's \(\{|\,0\rangle, |\,1\rangle\}\) states are the positive and negative unit vectors along the z axis, respectively. These axes will be useful later in this document.

Multiple qubits are represented in precisely the same way, but taking tensor products of the spaces and states. Thus \(n\) qubits have \(2^n\) possible states.

2.1.5 An Important Distinction

An important distinction between the probabilistic case described above and the quantum case is that probabilistic states may just mask out ignorance. For example a coin is physically only 0 or 1 and the probabilistic view merely represents our ignorance about which it actually is. **This is not the case in quantum mechanics**. Assuming events cannot instantaneously influence one another, the quantum states - as far as we know - cannot mask any underlying state. This is what people mean when they say that there is no local hidden variable theory for quantum mechanics. These probabilistic quantum states are as real as it gets: they don't describe our knowledge of the quantum system, they describe the physical reality of the system.

2.1.6 Some Code

Let us take a look at some code in pyQuil to see how these quantum states play out. We will dive deeper into quantum operations and pyQuil in the following sections. Note that in order to run these examples you will need to install pyQuil and set up a connection to the Forest API. Each of the code snippets below will be immediately followed by its output.

```
# Imports for pyQuil (ignore for now)
import numpy as np
from pyquil.quil import Program
import pyquil.api as api
quantum_simulator = api.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.
wavefunc = quantum_simulator.wavefunction(p)
# wavefunc is a Wavefunction object that stores a quantum state as a list of␣
↪amplitudes
alpha, beta = wavefunc
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))
```

```
Our qubit is in the state alpha=(1+0j) and beta=0j
The probability of measuring the qubit in outcome 0 is 1.0
The probability of measuring the qubit in outcome 1 is 0.0
```

Applying an operation to our qubit affects the probability of each outcome.

```
# We can import the qubit "flip" operation, called X, and see what it does.
# We will learn more about this operation in the next section.
from pyquil.gates import X
p = Program(X(0))

wavefunc = quantum_simulator.wavefunction(p)
alpha, beta = wavefunc
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))
```

```
Our qubit is in the state alpha=0j and beta=(1+0j)
The probability of measuring the qubit in outcome 0 is 0.0
The probability of measuring the qubit in outcome 1 is 1.0
```

In this case we have flipped the probability of outcome 0 into the probability of outcome 1 for our qubit. We can also investigate what happens to the state of multiple qubits. We'd expect the state of multiple qubits to grow exponentially in size, as their vectors are tensored together.

```
# Multiple qubits also produce the expected scaling of the state.
p = Program(I(0), I(1))
wvf = quantum_simulator.wavefunction(p)
print("The quantum state is of dimension:", len(wvf.amplitudes))

p = Program(I(0), I(1), I(2), I(3))
wvf = quantum_simulator.wavefunction(p)
print("The quantum state is of dimension:", len(wvf.amplitudes))

p = Program()
for x in range(10):
    p.inst(I(x))
wvf = quantum_simulator.wavefunction(p)
print("The quantum state is of dimension:", len(wvf.amplitudes)  )
```

```
The quantum state is of dimension: 4
The quantum state is of dimension: 16
The quantum state is of dimension: 1024
```

Let's look at the actual value for the state of two qubits combined. The resulting dictionary of this method contains outcomes as keys and the probabilities of those outcomes as values.

```
# wavefunction(Program) returns a coefficient array that corresponds to outcomes in␣
↪the following order
wvf = quantum_simulator.wavefunction(Program(I(0), I(1)))
print(wvf.get_outcome_probs())
```

```
{'11': 0.0, '10': 0.0, '00': 1.0, '01': 0.0}
```

2.2 Qubit Operations

In the previous section we introduced our first two **operations**: the I (or identity) operation and the X operation. In this section we will get into some more details on what these operations are.

Quantum states are complex vectors on the Bloch sphere, and quantum operations are matrices with two properties:

1. They are reversible.
2. When applied to a state vector on the Bloch sphere, the resulting vector is also on the Bloch sphere.

Matrices that satisfy these two properties are called unitary matrices. Applying an operation to a quantum state is the same as multiplying a vector by one of these matrices. Such an operation is called a **gate**.

Since individual qubits are two-dimensional vectors, operations on individual qubits are 2x2 matrices. The identity matrix leaves the state vector unchanged:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

so the program that applies this operation to the zero state is just

$$I\,|0\rangle = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle$$

```
p = Program(I(0))
print(quantum_simulator.wavefunction(p))
```

```
[ 1.+0.j  0.+0.j]
```

2.2.1 Pauli Operators

Let's revisit the X gate introduced above. It is one of three important single-qubit gates, called the Pauli operators:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

```
from pyquil.gates import X, Y, Z
p = Program(X(0))
print("X|0> = ", quantum_simulator.wavefunction(p))
print("The outcome probabilities are", quantum_simulator.bit_string_probabilities(p))
print("This looks like a bit flip.\n")
p = Program(Y(0))
print("Y|0> = ", quantum_simulator.wavefunction(p))
print("The outcome probabilities are", quantum_simulator.bit_string_probabilities(p))
print("This also looks like a bit flip.\n")
p = Program(Z(0))
print("Z|0> = ", quantum_simulator.wavefunction(p))
print("The outcome probabilities are", quantum_simulator.bit_string_probabilities(p))
print("This state looks unchanged.")
```

```
X|0> =  [ 0.+0.j  1.+0.j]
The outcome probabilities are {'1': 1.0, '0': 0.0}
This looks like a bit flip.

Y|0> =  [ 0.+0.j  0.+1.j]
The outcome probabilities are {'1': 1.0, '0': 0.0}
This also looks like a bit flip.

Z|0> =  [ 1.+0.j  0.+0.j]
The outcome probabilities are {'1': 0.0, '0': 1.0}
This state looks unchanged.
```

The Pauli matrices have a visual interpretation: they perform 180 degree rotations of qubit state vectors on the Bloch sphere. They operate about their respective axes as shown in the Bloch sphere depicted above. For example, the X gate performs a 180 degree rotation *about* the x axis. This explains the results of our code above: for a state vector initially in the +z direction, both X and Y gates will rotate it to -z, and the Z gate will leave it unchanged.

However, notice that while the X and Y gates produce the same outcome probabilities, they actually produce different states. These states are not distinguished if they are measured immediately, but they produce different results in larger programs.

Quantum programs are built by applying successive gate operations:

```
# Composing qubit operations is the same as multiplying matrices sequentially
p = Program(X(0), Y(0), Z(0))
print("ZYX|0> = ", quantum_simulator.wavefunction(p))
print("With outcome probabilities\n", quantum_simulator.bit_string_probabilities(p))
```

```
ZYX|0> =  [ 0.-1.j  0.+0.j]
With outcome probabilities
{'1': 0.0, '0': 1.0}
```

2.2.2 Multi-Qubit Operations

Operations can also be applied to composite states of multiple qubits. One common example is the controlled-not or CNOT gate that works on two qubits. Its matrix form is:

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Let's take a look at how we could use a CNOT gate in pyQuil.

```
from pyquil.gates import CNOT

p = Program(CNOT(0, 1))
print("CNOT|00> = ", quantum_simulator.wavefunction(p))
print("With outcome probabilities\n", quantum_simulator.bit_string_probabilities(p))
p = Program(X(0), CNOT(0, 1))
print("CNOT|01> = ", quantum_simulator.wavefunction(p))
print("With outcome probabilities\n", quantum_simulator.bit_string_probabilities(p))
p = Program(X(1), CNOT(0, 1))
print("CNOT|10> = ", quantum_simulator.wavefunction(p))
print("With outcome probabilities\n", quantum_simulator.bit_string_probabilities(p))
p = Program(X(0), X(1), CNOT(0, 1))
print("CNOT|11> = ", quantum_simulator.wavefunction(p))
print("With outcome probabilities\n", quantum_simulator.bit_string_probabilities(p))
```

```
CNOT|00> =  [ 1.+0.j  0.+0.j  0.+0.j  0.+0.j]
With outcome probabilities
{'11': 0.0, '10': 0.0, '00': 1.0, '01': 0.0}
CNOT|01> =  [ 0.+0.j  0.+0.j  0.+0.j  1.+0.j]
With outcome probabilities
{'11': 1.0, '10': 0.0, '00': 0.0, '01': 0.0}
CNOT|10> =  [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j]
With outcome probabilities
{'11': 0.0, '10': 1.0, '00': 0.0, '01': 0.0}
CNOT|11> =  [ 0.+0.j  1.+0.j  0.+0.j  0.+0.j]
With outcome probabilities
{'11': 0.0, '10': 0.0, '00': 0.0, '01': 1.0}
```

The CNOT gate does what its name implies: the state of the second qubit is flipped (negated) if and only if the state of the first qubit is 1 (true).

Another two-qubit gate example is the SWAP gate, which swaps the \(|01\rangle \) and \(|10\rangle \) states:

$$SWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

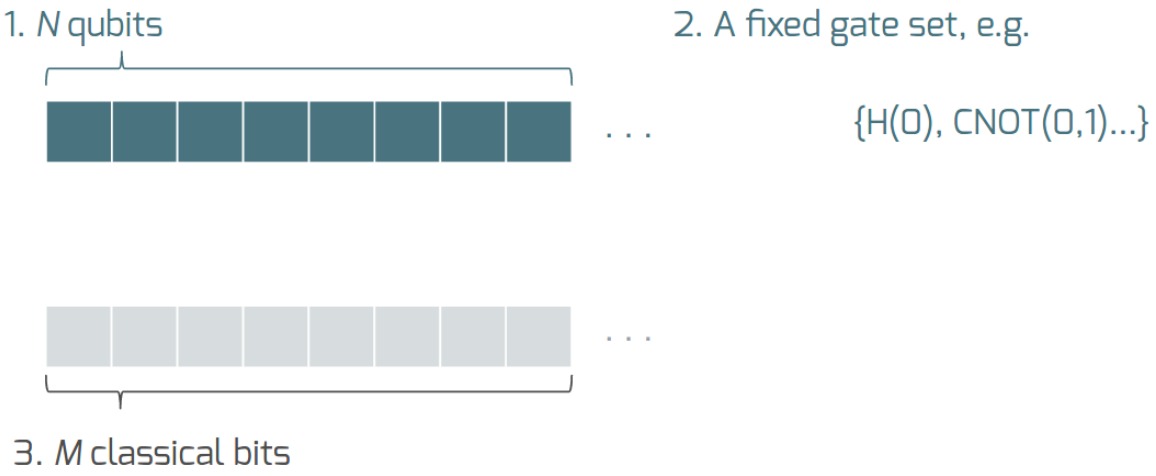
```
from pyquil.gates import SWAP
p = Program(X(0), SWAP(0,1))

print("SWAP|01> = ", quantum_simulator.wavefunction(p))
print("With outcome probabilities\n", quantum_simulator.bit_string_probabilities(p))
```

```
SWAP|01> =  [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j]
With outcome probabilities
{'11': 0.0, '01': 0.0, '00': 0.0, '10': 1.0}
```

In summary, quantum computing operations are composed of a series of complex matrices applied to complex vectors. These matrices must be unitary (meaning that their complex conjugate transpose is equal to their inverse) because the overall probability of all outcomes must always sum to one.

2.3 The Quantum Abstract Machine

We now have enough background to introduce the programming model that underlies Quil. This is a hybrid quantum-classical model in which \(N\) qubits interact with \(M\) classical bits:



These qubits and classical bits come with a defined gate set, e.g. which gate operations can be applied to which qubits. Different kinds of quantum computing hardware place different limitations on what gates can be applied, and the fixed gate set represents these limitations.

Full details on the Quantum Abstract Machine and Quil can be found in the Quil whitepaper.

The next section on measurements will describe the interaction between the classical and quantum parts of a Quantum Abstract Machine (QAM).

2.3.1 Qubit Measurements

Measurements have two effects:

1. They project the state vector onto one of the basic outcomes
2. (*optional*) They store the outcome of the measurement in a classical bit.

Here's a simple example:

```
# Create a program that stores the outcome of measuring qubit #0 into classical␣
↪register [0]
classical_register_index = 0
p = Program(I(0)).measure(0, classical_register_index)
```

Up until this point we have used the quantum simulator to cheat a little bit - we have actually looked at the wavefunction that comes back. However, on real quantum hardware, we are unable to directly look at the wavefunction. Instead we only have access to the classical bits that are affected by measurements. This functionality is emulated by the `run` command.

```
# Choose which classical registers to look in at the end of the computation
classical_regs = [0, 1]
quantum_simulator.run(p, classical_regs)
```

```
[[0, 0]]
```

We see that both registers are zero. However, if we had flipped the qubit before measurement then we obtain:

```
classical_register_index = 0
p = Program(X(0)) # flip the qubit
p.measure(0, classical_register_index) # measure the qubit

classical_regs = [0, 1]
quantum_simulator.run(p, classical_regs)
```

```
[[1, 0]]
```

These measurements are deterministic, e.g. if we make them multiple times then we always get the same outcome:

```
classical_register_index = 0
p = Program(X(0)) # Flip the qubit
p.measure(0, classical_register_index) # Measure the qubit

classical_regs = [0]
trials = 10
print(quantum_simulator.run(p, classical_regs, trials))
```

```
[[1], [1], [1], [1], [1], [1], [1], [1], [1], [1]]
```

2.3.2 Classical/Quantum Interaction

However this is not the case in general - measurements can affect the quantum state as well. In fact, measurements act like projections onto the outcome basis states. To show how this works, we first introduce a new single-qubit gate, the Hadamard gate. The matrix form of the Hadamard gate is:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

The following pyQuil code shows how we can use the Hadamard gate:

```
from pyquil.gates import H

# The Hadamard produces what is called a superposition state
```

```
coin_program = Program(H(0))
print("H|0> = ", quantum_simulator.wavefunction(coin_program))
print("With outcome probabilities\n", quantum_simulator.bit_string_probabilities(coin_
↪program))
```

```
H|0> =  [ 0.70710678+0.j  0.70710678+0.j]
With outcome probabilities
{'1': 0.49999999999999989, '0': 0.49999999999999989}
```

A qubit in this state will be measured half of the time in the \(|0\rangle \) state, and half of the time in the \(|1\rangle \) state. In a sense, this qubit truly is a random variable representing a coin. In fact, there are many wavefunctions that will give this same operational outcome. There is a continuous family of states of the form

$$\frac{1}{\sqrt{2}}\left(|\,0\rangle + e^{i\theta}|\,1\rangle\right)$$

that represent the outcomes of an unbiased coin. Being able to work with all of these different new states is part of what gives quantum computing extra power over regular bits.

```
# Introduce measurement
classical_reg = 0
coin_program = Program(H(0)).measure(0, classical_reg)
trials = 10

# We see probabilistic results of about half 1's and half 0's
quantum_simulator.run(coin_program, [0], trials)
```

```
[[0], [1], [1], [0], [1], [0], [0], [1], [0], [0]]
```

pyQuil allows us to look at the wavefunction **after** a measurement as well:

```
classical_reg = 0
coin_program = Program(H(0))
print("Before measurement: H|0> = ", quantum_simulator.wavefunction(coin_program))
coin_program.measure(0, classical_reg)
for x in range(5):
    print("After measurement: ", quantum_simulator.wavefunction(coin_program))
```

```
Before measurement: H|0> =  [ 0.70710678+0.j  0.70710678+0.j]
After measurement:  [ 0.+0.j  1.+0.j]
After measurement:  [ 1.+0.j  0.+0.j]
After measurement:  [ 1.+0.j  0.+0.j]
After measurement:  [ 0.+0.j  1.+0.j]
After measurement:  [ 0.+0.j  1.+0.j]
```

We can clearly see that measurement has an effect on the quantum state independent of what is stored classically. We begin in a state that has a 50-50 probability of being \(|0\rangle \) or \(|1\rangle \). After measurement, the state changes into being entirely in \(|0\rangle \) or entirely in \(|1\rangle \) according to which outcome was obtained. This is the phenomenon referred to as the **collapse** of the wavefunction. Mathematically, the wavefunction is being projected onto the vector of the obtained outcome and subsequently rescaled to unit norm.

```
# This happens with bigger systems too
classical_reg = 0

# This program prepares something called a Bell state (a special kind of "entangled␣
↪state")
bell_program = Program(H(0), CNOT(0, 1))
```

```
print("Before measurement: H|0> = ", quantum_simulator.wavefunction(bell_program))
bell_program.measure(0, classical_reg)
for x in range(5):
    print("After measurement: ", quantum_simulator.bit_string_probabilities(bell_
↪program))
```

```
Before measurement: H|0> =  [ 0.70710678+0.j  0.00000000+0.j  0.00000000+0.j  0.
↪70710678+0.j]
After measurement:  {'11': 1.0, '10': 0.0, '00': 0.0, '01': 0.0}
After measurement:  {'11': 0.0, '10': 0.0, '00': 1.0, '01': 0.0}
After measurement:  {'11': 0.0, '10': 0.0, '00': 1.0, '01': 0.0}
After measurement:  {'11': 1.0, '10': 0.0, '00': 0.0, '01': 0.0}
After measurement:  {'11': 1.0, '10': 0.0, '00': 0.0, '01': 0.0}
```

The above program prepares **entanglement** because, even though there are random outcomes, after every measurement both qubits are in the same state. They are either both \(|0\rangle \) or both \(|1\rangle \). This special kind of correlation is part of what makes quantum mechanics so unique and powerful.

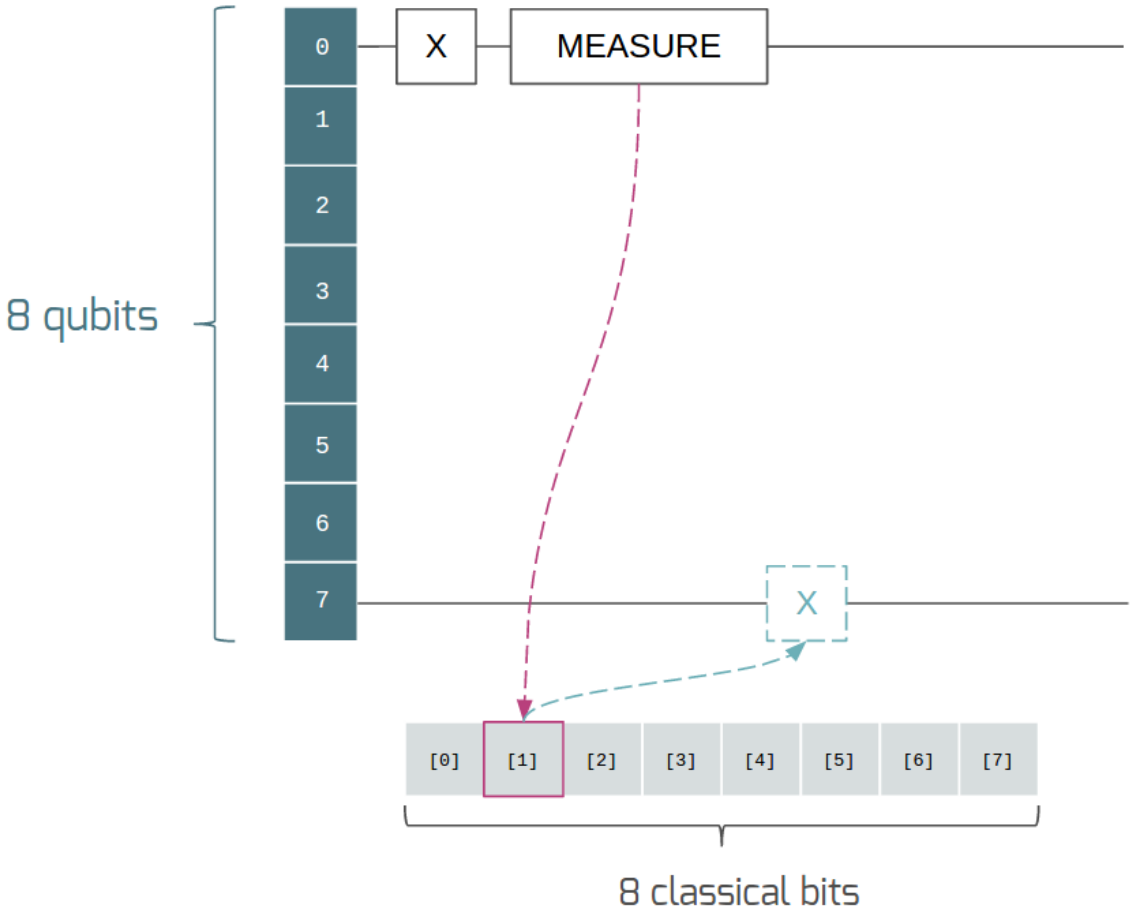
2.3.3 Classical Control

There are also ways of introducing classical control of quantum programs. For example, we can use the state of classical bits to determine what quantum operations to run.

```
true_branch = Program(X(7)) # if branch
false_branch = Program(I(7)) # else branch
p = Program(X(0)).measure(0, 1).if_then(1, true_branch, false_branch) # Branch on␣
↪classical reg [1]
p.measure(7, 7) # Measure qubit #7 into classical register [7]
print(quantum_simulator.run(p, [7])) # Run and check register [7]
```

```
[[1]]
```

A [1] here means that qubit 7 was indeed flipped.

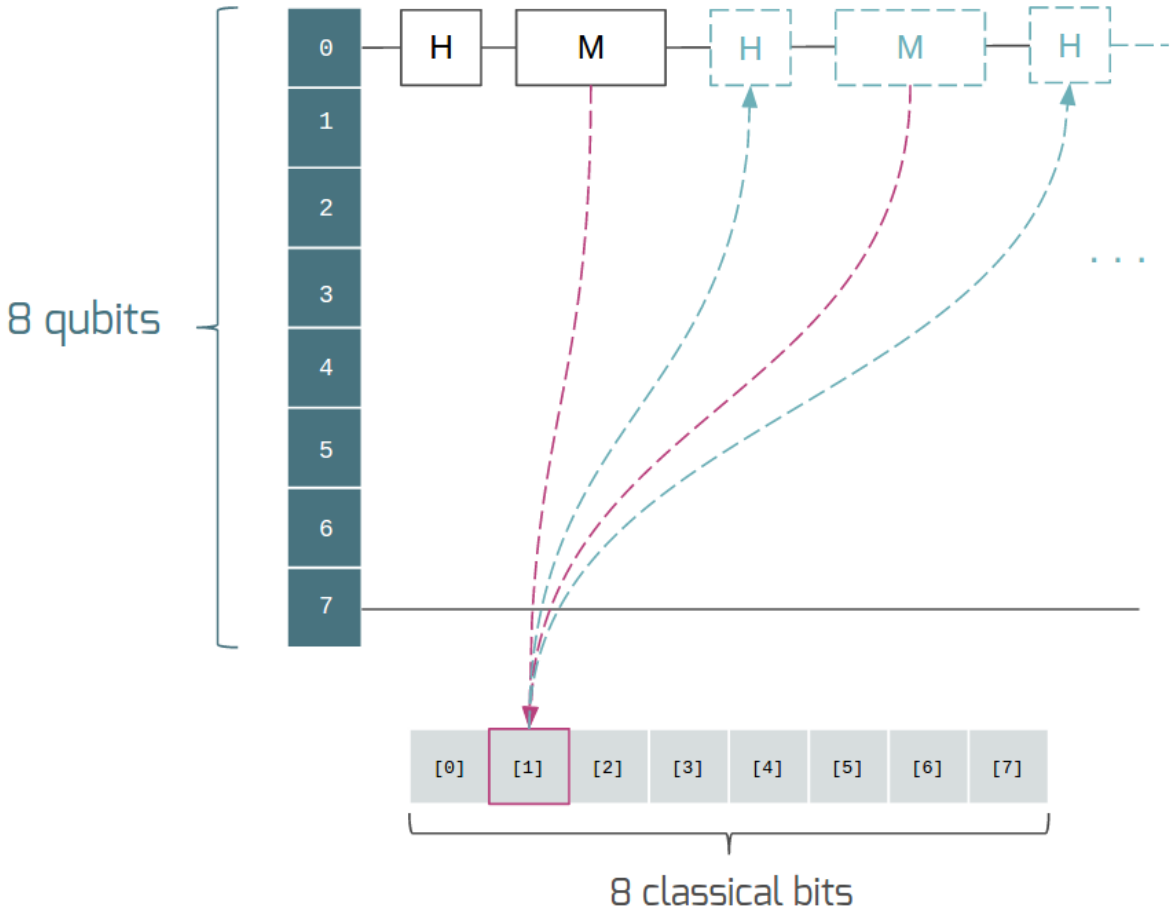


2.3.4 Example: The Probabilistic Halting Problem

A fun example is to create a program that has an exponentially increasing chance of halting, but that may run forever!

```
inside_loop = Program(H(0)).measure(0, 1)

p = Program().inst(X(0)).while_do(1, inside_loop)
print(quantum_simulator.run(p, [1])) # Run and check register [1]
```

```
[[0]]
```



You are now ready to check out the Installation and Getting Started guide! Feel free to look at Next Steps for further information and references on quantum computing.

CHAPTER 3

Installation and Getting Started

This toolkit provides some simple libraries for writing quantum programs.

```
from pyquil.quil import Program
import pyquil.api as api
from pyquil.gates import *
qvm = api.QVMConnection()
p = Program()
p.inst(H(0), CNOT(0, 1))
    <pyquil.pyquil.Program object at 0x101ebfb50>
wvf = qvm.wavefunction(p)
print(wvf)
    (0.7071067812+0j)|00> + (0.7071067812+0j)|11>
```

It comes with a few parts:

1. **Quil**: The Quantum Instruction Language standard. Instructions written in Quil can be executed on any implementation of a quantum abstract machine, such as the quantum virtual machine (QVM), or on a real quantum processing unit (QPU). More details regarding Quil can be found in the whitepaper.
2. **QVM**: A Quantum Virtual Machine, which is an implementation of the quantum abstract machine on classical hardware. The QVM lets you use a regular computer to simulate a small quantum computer. You can access the Rigetti QVM running in the cloud with your API key. Sign up here to get your key.
3. **pyQuil**: A Python library to help write and run Quil code and quantum programs.
4. **QPUConnection**: pyQuil also includes some a special connection which lets you run experiments on Rigetti's prototype superconducting quantum processors over the cloud.

3.1 Environment Setup

3.1.1 Prerequisites

Before you can start writing quantum programs, you will need Python 2.7 (version 2.7.10 or greater) or Python 3.6 and the Python package manager pip.

Note: PyQuil works on both Python 2 and 3. However, Rigetti **strongly** recommends using Python 3 if possible. Future feature developments in PyQuil may support Python 3 only.

3.1.2 Installation

You can install pyQuil directly from the Python package manager pip using:

```
pip install pyquil
```

To instead install the bleeding-edge version from source, clone the pyquil GitHub repository, navigate into its directory in a terminal, and run:

```
pip install -e .
```

On Mac/Linux, if this command does not succeed because of permissions errors, then instead run:

```
sudo pip install -e .
```

This will also install pyQuil's dependencies (numpy, requests, etc.) if you do not already have them.

The library will now be available globally.

3.1.3 Connecting to the Rigetti Forest

pyQuil can be used to build and manipulate Quil programs without restriction. However, to run programs (e.g., to get wavefunctions, get multishot experiment data), you will need an API key for Rigetti Forest. This will allow you to run your programs on the Rigetti QVM or QPU.

Sign up here to get a Forest API key, it's free and only takes a few seconds.

It's also highly recommended to join our public slack channel where you can connect with other users and Rigetti members for support.

Run the following command to automatically set up the config. This will prompt you for the required information (URL, key, and user id). It will then create a file in the proper location (the user's root directory):

```
pyquil-config-setup
```

If the setup completed successfully then you can skip to the next section.

You can also create the configuration file manually if you'd like and place it at `~/.pyquil_config`. The configuration file is in INI format and should contain all the information required to connect to Forest:

```
[Rigetti Forest]
key: <Rigetti Forest API key>
user_id: <Rigetti User ID>
```

Alternatively, you can place the file at your own chosen location and then set the `PYQUIL_CONFIG` environment variable to the path of the file.

Note: You may specify an absolute path or use the ~ to indicate your home directory. On Linux, this points to `/users/username`. On Mac, this points to `/Users/Username`. On Windows, this points to `C:\Users\Username`

Note: Windows users may find it easier to name the file `pyquil.ini` and open it using notepad. Then, set the `PYQUIL_CONFIG` environment variable by opening up a command prompt and running: `setenv PYQUIL_CONFIG=C:\Users\Username\pyquil.ini`

As a last resort, connection information can be provided via environment variables.

```
export QVM_API_KEY=<Rigetti Forest API key>
export QVM_USER_ID=<Rigetti User ID>
```

If you are still seeing errors or warnings then file a bug using Github Issues.

3.2 Running your first quantum program

pyQuil is a Python library that helps you write programs in the Quantum Instruction Language (Quil). It also ships with a simple script `examples/run_quil.py` that runs Quil code directly. You can test your connection to Forest using this script by executing the following on your command line

```
cd examples/
python run_quil.py hello_world.quil
```

You should see the following output array `[[1, 0, 0, 0, 0, 0, 0, 0]]`. This indicates that you have a good connection to our API.

You can continue to write more Quil code in files and run them using the `run_quil.py` script. The following sections describe how to use the pyQuil library directly to build quantum programs in Python.

3.3 Basic pyQuil Usage

To ensure that your installation is working correctly, try running the following Python commands interactively. First, import the `quil` module (which constructs quantum programs) and the `api` module (which allows connections to the Rigetti QVM). We will also import some basic gates for pyQuil as well as numpy.

```
from pyquil.quil import Program
import pyquil.api as api
from pyquil.gates import *
import numpy as np
```

Next, we want to open a connection to the QVM.

```
qvm = api.QVMConnection()
```

Now we can make a program by adding some Quil instruction using the `inst` method on a `Program` object.

```
p = Program()
p.inst(X(0)).measure(0, 0)
```

```
<pyquil.quil.Program at 0x101d45a90>
```

This program simply applies the X-gate to the zeroth qubit, measures that qubit, and stores the measurement result in the zeroth classical register. We can look at the Quil code that makes up this program simply by printing it.

```
print(p)
```

```
X 0
MEASURE 0 [0]
```

Most importantly, of course, we can see what happens if we run this program on the QVM:

```
classical_regs = [0] # A list of which classical registers to return the values of.

qvm.run(p, classical_regs)
```

```
[[1]]
```

We see that the result of this program is that the classical register `[0]` now stores the state of qubit 0, which should be $|1\rangle$ after an X-gate. We can of course ask for more classical registers:

```
qvm.run(p, [0, 1, 2])
```

```
[[1, 0, 0]]
```

The classical registers are initialized to zero, so registers `[1]` and `[2]` come out as zero. If we stored the measurement in a different classical register we would obtain:

```
p = Program()    # clear the old program
p.inst(X(0)).measure(0, 1)
qvm.run(p, [0, 1, 2])
```

```
[[0, 1, 0]]
```

We can also run programs multiple times and accumulate all the results in a single list.

```
coin_flip = Program().inst(H(0)).measure(0, 0)
num_flips = 5
qvm.run(coin_flip, [0], num_flips)
```

```
[[0], [1], [0], [1], [0]]
```

Try running the above code several times. You will see that you will, with very high probability, get different results each time.

As the QVM is a virtual machine, we can also inspect the wavefunction of a program directly, even without measurements:

```
coin_flip = Program().inst(H(0))
qvm.wavefunction(coin_flip)
```

```
<pyquil.wavefunction.Wavefunction at 0x1088a2c10>
```

The return value is a Wavefunction object that stores the amplitudes of the quantum state at the conclusion of the program. We can print this object

```
coin_flip = Program().inst(H(0))
wvf = qvm.wavefunction(coin_flip)
print(wvf)
```

```
(0.7071067812+0j)|0> + (0.7071067812+0j)|1>
```

To see the amplitudes listed as a sum of computational basis states. We can index into those amplitudes directly or look at a dictionary of associated outcome probabilities.

```
assert wvf[0] == 1 / np.sqrt(2)
# The amplitudes are stored as a numpy array on the Wavefunction object
print(wvf.amplitudes)
prob_dict = wvf.get_outcome_probs() # extracts the probabilities of outcomes as a dict
print(prob_dict)
prob_dict.keys() # these stores the bitstring outcomes
assert len(wvf) == 1 # gives the number of qubits
```

```
[ 0.70710678+0.j  0.70710678+0.j]
{'1': 0.49999999999999989, '0': 0.49999999999999989}
```

The result from a wavefunction call also contains an optional amount of classical memory to check:

```
coin_flip = Program().inst(H(0)).measure(0,0)
wavf = qvm.wavefunction(coin_flip, classical_addresses=range(9))
classical_mem = wavf.classical_memory
```

Additionally, we can pass a random seed to the Connection object. This allows us to reliably reproduce measurement results for the purpose of testing:

```
seeded_cxn = api.QVMConnection(random_seed=17)
print(seeded_cxn.run(Program(H(0)).measure(0, 0), [0], 20))

seeded_cxn = api.QVMConnection(random_seed=17)
# This will give identical output to the above
print(seeded_cxn.run(Program(H(0)).measure(0, 0), [0], 20))
```

It is important to remember that this `wavefunction` method is just a useful debugging tool for small quantum systems, and it cannot be feasibly obtained on a quantum processor.

3.3.1 Some Program Construction Features

Multiple instructions can be applied at once or chained together. The following are all valid programs:

```
print("Multiple inst arguments with final measurement:")
print(Program().inst(X(0), Y(1), Z(0)).measure(0, 1))

print("Chained inst with explicit MEASURE instruction:")
print(Program().inst(X(0)).inst(Y(1)).measure(0, 1).inst(MEASURE(1, 2)))

print("A mix of chained inst and measures:")
```

```
print(Program().inst(X(0)).measure(0, 1).inst(Y(1), X(0)).measure(0, 0))

print("A composition of two programs:")
print(Program(X(0)) + Program(Y(0)))
```

```
Multiple inst arguments with final measurement:
X 0
Y 1
Z 0
MEASURE 0 [1]

Chained inst with explicit MEASURE instruction:
X 0
Y 1
MEASURE 0 [1]
MEASURE 1 [2]

A mix of chained inst and measures:
X 0
MEASURE 0 [1]
Y 1
X 0
MEASURE 0 [0]

A composition of two programs:
X 0
Y 0
```

3.3.2 Fixing a Mistaken Instruction

If an instruction was appended to a program incorrectly, one can pop it off.

```
p = Program().inst(X(0))
p.inst(Y(1))
print("Oops! We have added Y 1 by accident:")
print(p)

print("We can fix by popping:")
p.pop()
print(p)

print("And then add it back:")
p += Program(Y(1))
print(p)
```

```
Oops! We have added Y 1 by accident:
X 0
Y 1

We can fix by popping:
X 0

And then add it back:
X 0
Y 1
```

3.3.3 The Standard Gate Set

The following gates methods come standard with Quil and `gates.py`:

- Pauli gates `I`, `X`, `Y`, `Z`
- Hadamard gate: `H`
- Phase gates: `PHASE(`θ`)`, `S`, `T`
- Controlled phase gates: `CZ`, `CPHASE00(`α`)`, `CPHASE01(`α`)`, `CPHASE10(`α`)`, `CPHASE(`α`)`
- Cartesian rotation gates: `RX(`θ`)`, `RY(`θ`)`, `RZ(`θ`)`
- Controlled X gates: `CNOT`, `CCNOT`
- Swap gates: `SWAP`, `CSWAP`, `ISWAP`, `PSWAP(`α`)`

The parameterized gates take a real or complex floating point number as an argument.

3.3.4 Defining New Gates

New gates can be easily added inline to Quil programs. All you need is a matrix representation of the gate. For example, below we define a $\sqrt{X}$ gate.

```
import numpy as np

# First we define the new gate from a matrix
x_gate_matrix = np.array(([0.0, 1.0], [1.0, 0.0]))
sqrt_x = np.array([[ 0.5+0.5j,  0.5-0.5j],
                   [ 0.5-0.5j,  0.5+0.5j]])
p = Program().defgate("SQRT-X", sqrt_x)

# Then we can use the new gate,
p.inst(("SQRT-X", 0))
print(p)
```

```
DEFGATE SQRT-X:
    0.5+0.5i, 0.5-0.5i
    0.5-0.5i, 0.5+0.5i

SQRT-X 0
```

```
print(qvm.wavefunction(p))
```

```
(0.5+0.5j)|0> + (0.5-0.5j)|1>
```

Quil in general supports defining parametric gates, though right now only static gates are supported by pyQuil. Below we show how we can define $X_0 \otimes \sqrt{X_1}$ as a single gate.

```
# A multi-qubit defgate example
x_gate_matrix = np.array(([0.0, 1.0], [1.0, 0.0]))
sqrt_x = np.array([[ 0.5+0.5j,  0.5-0.5j],
                   [ 0.5-0.5j,  0.5+0.5j]])
x_sqrt_x = np.kron(x_gate_matrix, sqrt_x)
p = Program().defgate("X-SQRT-X", x_sqrt_x)

# Then we can use the new gate
p.inst(("X-SQRT-X", 0, 1))
```

```
wavf = qvm.wavefunction(p)
print(wavf)
```

```
(0.5+0.5j)|01> + (0.5-0.5j)|11>
```

3.4 Advanced Usage

3.4.1 Quantum Fourier Transform (QFT)

Let us do an example that includes multi-qubit parameterized gates.

Here we wish to compute the discrete Fourier transform of `[0, 1, 0, 0, 0, 0, 0, 0]`. We do this in three steps:

1. Write a function called `qft3` to make a 3-qubit QFT quantum program.
2. Write a state preparation quantum program.
3. Execute state preparation followed by the QFT on the QVM.

First we define a function to make a 3-qubit QFT quantum program. This is a mix of Hadamard and CPHASE gates, with a final bit reversal correction at the end consisting of a single SWAP gate.

```
from math import pi

def qft3(q0, q1, q2):
    p = Program()
    p.inst( H(q2),
            CPHASE(pi/2.0, q1, q2),
            H(q1),
            CPHASE(pi/4.0, q0, q2),
            CPHASE(pi/2.0, q0, q1),
            H(q0),
            SWAP(q0, q2) )
    return p
```

There is a very important detail to recognize here: The function `qft3` doesn't *compute* the QFT, but rather it *makes a quantum program* to compute the QFT on qubits `q0`, `q1`, and `q2`.

We can see what this program looks like in Quil notation by doing the following:

```
print(qft3(0, 1, 2))
```

```
H 2
CPHASE(1.5707963267948966) 1 2
H 1
CPHASE(0.7853981633974483) 0 2
CPHASE(1.5707963267948966) 0 1
H 0
SWAP 0 2
```

Next, we want to prepare a state that corresponds to the sequence we want to compute the discrete Fourier transform of. Fortunately, this is easy, we just apply an X-gate to the zeroth qubit.

```
state_prep = Program().inst(X(0))
```

We can verify that this works by computing its wavefunction. However, we need to add some "dummy" qubits, because otherwise `wavefunction` would return a two-element vector.

```
add_dummy_qubits = Program().inst(I(1), I(2))
wavf = qvm.wavefunction(state_prep + add_dummy_qubits)
print(wavf)
```

```
(1+0j)|001>
```

If we have two quantum programs `a` and `b`, we can concatenate them by doing `a + b`. Using this, all we need to do is compute the QFT after state preparation to get our final result.

```
wavf = qvm.wavefunction(state_prep + qft3(0, 1, 2))
print(wavf.amplitudes)
```

```
array([  3.53553391e-01+0.j        ,   2.50000000e-01+0.25j      ,
         2.16489014e-17+0.35355339j,  -2.50000000e-01+0.25j      ,
        -3.53553391e-01+0.j        ,  -2.50000000e-01-0.25j      ,
        -2.16489014e-17-0.35355339j,   2.50000000e-01-0.25j      ])
```

We can verify this works by computing the (inverse) FFT from NumPy.

```
from numpy.fft import ifft
ifft([0,1,0,0,0,0,0,0], norm="ortho")
```

```
array([ 0.35355339+0.j        ,  0.25000000+0.25j      ,
        0.00000000+0.35355339j, -0.25000000+0.25j      ,
       -0.35355339+0.j        , -0.25000000-0.25j      ,
        0.00000000-0.35355339j,  0.25000000-0.25j      ])
```

3.4.2 Classical Control Flow

Here are a couple quick examples that show how much richer the classical control of a Quil program can be. In this first example, we have a register called `classical_flag_register` which we use for looping. Then we construct the loop in the following steps:

1. We first initialize this register to `1` with the `init_register` program so our while loop will execute. This is often called the *loop preamble* or *loop initialization*.
2. Next, we write body of the loop in a program itself. This will be a program that computes an X followed by an H on our qubit.
3. Lastly, we put it all together using the `while_do` method.

```
# Name our classical registers:
classical_flag_register = 2

# Write out the loop initialization and body programs:
init_register = Program(TRUE([classical_flag_register]))
loop_body = Program(X(0), H(0)).measure(0, classical_flag_register)

# Put it all together in a loop program:
loop_prog = init_register.while_do(classical_flag_register, loop_body)

print(loop_prog)
```

```
TRUE [2]
LABEL @START1
JUMP-UNLESS @END2 [2]
X 0
H 0
MEASURE 0 [2]
JUMP @START1
LABEL @END2
```

Notice that the `init_register` program applied a Quil instruction directly to a classical register. There are several classical commands that can be used in this fashion:

- `TRUE` which sets a single classical bit to be 1
- `FALSE` which sets a single classical bit to be 0
- `NOT` which flips a classical bit
- `AND` which operates on two classical bits
- `OR` which operates on two classical bits
- `MOVE` which moves the value of a classical bit at one classical address into another
- `EXCHANGE` which swaps the value of two classical bits

In this next example, we show how to do conditional branching in the form of the traditional `if` construct as in many programming languages. Much like the last example, we construct programs for each branch of the `if`, and put it all together by using the `if_then` method.

```
# Name our classical registers:
test_register = 1
answer_register = 0

# Construct each branch of our if-statement. We can have empty branches
# simply by having empty programs.
then_branch = Program(X(0))
else_branch = Program()

# Make a program that will put a 0 or 1 in test_register with 50% probability:
branching_prog = Program(H(1)).measure(1, test_register)

# Add the conditional branching:
branching_prog.if_then(test_register, then_branch, else_branch)

# Measure qubit 0 into our answer register:
branching_prog.measure(0, answer_register)

print(branching_prog)
```

```
H 1
MEASURE 1 [1]
JUMP-WHEN @THEN3 [1]
JUMP @END4
LABEL @THEN3
X 0
LABEL @END4
MEASURE 0 [0]
```

We can run this program a few times to see what we get in the `answer_register`.

```
qvm.run(branching_prog, [answer_register], 10)
```

```
[[1], [1], [1], [0], [1], [0], [0], [1], [1], [0]]
```

3.4.3 Parametric Depolarizing Noise

The Rigetti QVM has support for emulating certain types of noise models. One such model is *parametric Pauli noise*, which is defined by a set of 6 probabilities:

- The probabilities P_X, P_Y, and P_Z which define respectively the probability of a Pauli X, Y, or Z gate getting applied to *each* qubit after *every* gate application. These probabilities are called the *gate noise probabilities*.
- The probabilities P'_X, P'_Y, and P'_Z which define respectively the probability of a Pauli X, Y, or Z gate getting applied to the qubit being measured *before* it is measured. These probabilities are called the *measurement noise probabilities*.

We can instantiate a noisy QVM by creating a new connection with these probabilities specified.

```
# 20% chance of a X gate being applied after gate applications and before␣
↪measurements.
gate_noise_probs = [0.2, 0.0, 0.0]
meas_noise_probs = [0.2, 0.0, 0.0]
noisy_qvm = api.QVMConnection(gate_noise=gate_noise_probs, measurement_noise=meas_
↪noise_probs)
```

We can test this by applying an X-gate and measuring. Nominally, we should always measure `1`.

```
p = Program().inst(X(0)).measure(0, 0)
print("Without Noise: {}".format(qvm.run(p, [0], 10)))
print("With Noise   : {}".format(noisy_qvm.run(p, [0], 10)))
```

```
Without Noise: [[1], [1], [1], [1], [1], [1], [1], [1], [1], [1]]
With Noise   : [[0], [0], [0], [0], [0], [1], [1], [1], [1], [0]]
```

3.4.4 Parametric Programs

A big advantage of working in pyQuil is that you are able to leverage all the functionality of Python to generate Quil programs. In quantum/classical hybrid algorithms this often leads to situations where complex classical functions are used to generate Quil programs. pyQuil provides a convenient construction to allow you to use Python functions to generate templates of Quil programs, called `ParametricPrograms`:

```
# This function returns a quantum circuit with different rotation angles on a gate on␣
↪qubit 0
def rotator(angle):
    return Program(RX(angle, 0))

from pyquil.parametric import ParametricProgram
par_p = ParametricProgram(rotator) # This produces a new type of parameterized␣
↪program object
```

The parametric program `par_p` now takes the same arguments as `rotator`:

```
print(par_p(0.5))
```

```
RX(0.5) 0
```

We can think of `ParametricPrograms` as a sort of template for Quil programs. They cache computations that happen in Python functions so that templates in Quil can be efficiently substituted.

3.4.5 Pauli Operator Algebra

Many algorithms require manipulating sums of Pauli combinations, such as $\sigma = \frac{1}{2}I - \frac{3}{4}X_0Y_1Z_3 + (5 - 2i)Z_1X_2$, where G_n indicates the gate G acting on qubit n. We can represent such sums by constructing `PauliTerm` and `PauliSum`. The above sum can be constructed as follows:

```
from pyquil.paulis import ID, sX, sY, sZ

# Pauli term takes an operator "X", "Y", "Z", or "I"; a qubit to act on, and
# an optional coefficient.
a = 0.5 * ID
b = -0.75 * sX(0) * sY(1) * sZ(3)
c = (5-2j) * sZ(1) * sX(2)

# Construct a sum of Pauli terms.
sigma = a + b + c
print("sigma = {}".format(sigma))
```

```
sigma = 0.5*I + -0.75*X0*Y1*Z3 + (5-2j)*Z1*X2
```

Right now, the primary thing one can do with Pauli terms and sums is to construct the exponential of the Pauli term, i.e., $\exp[-i\beta\sigma]$. This is accomplished by constructing a parameterized Quil program that is evaluated when passed values for the coefficients of the angle β.

Related to exponentiating Pauli sums we provide utility functions for finding the commuting subgroups of a Pauli sum and approximating the exponential with the Suzuki-Trotter approximation through fourth order.

When arithmetic is done with Pauli sums, simplification is automatically done.

The following shows an instructive example of all three.

```
import pyquil.paulis as pl

# Simplification
sigma_cubed = sigma * sigma * sigma
print("Simplified  : {}".format(sigma_cubed))
print()

#Produce Quil code to compute exp[iX]
H = -1.0 * sX(0)
print("Quil to compute exp[iX] on qubit 0:")
print(pl.exponential_map(H)(1.0))
```

```
Simplified  : (32.46875-30j)*I + (-16.734375+15j)*X0*Y1*Z3 + (71.5625-144.625j)*Z1*X2

Quil to compute exp[iX] on qubit 0:
H 0
RZ(-2.0) 0
H 0
```

A more sophisticated feature of pyQuil is that it can create templates of Quil programs in ParametricProgram objects. An example use of these templates is in exponentiating a Hamiltonian that is parametrized by a constant. This commonly occurs in variational algorithms. The function `exponential_map` is used to compute exp[i * alpha * H] without explicitly filling in a value for alpha.

```
parametric_prog = pl.exponential_map(H)
print(parametric_prog(0.0))
print(parametric_prog(1.0))
print(parametric_prog(2.0))
```

This ParametricProgram now acts as a template, caching the result of the `exponential_map` calculation so that it can be used later with new values.

3.5 Connections

Larger pyQuil programs can involve more qubits and take a longer time to run. Instead of running the program immediately, you can insert your programs into a queue. This is done with the `use_queue` parameter to QVMConnection functions. By default, this parameter is set to False which means it skips the queue and runs it immediately. However, the QVM will reject programs that are more than 16 qubits or take longer than 5 seconds to run. Therefore, to run programs of this size you must set the `use_queue` parameter to True which has more overhead.

```
from pyquil.quil import Program
from pyquil.api import QVMConnection

qvm = QVMConnection()
qvm.run(Program(X(0).measure(0, 0), [0], use_queue=True)
```

The Forest queue also allows an asynchronous mode of interaction with methods postfixed with *_async*. This means that there is a seperate query to post a job and to get the result.

```
from pyquil.quil import Program
from pyquil.gates import X, H, I
from pyquil.api import QVMConnection

qvm = QVMConnection()
job_id = qvm.run_async(Program(X(0)).measure(0, 0), [0])
```

The *job_id* is a string that uniquely identifies the job in Forest. You can use the *.get_job* method on QVMConnection to get the current status.

```
job = qvm.get_job(job_id)
if not job.is_done():
  time.sleep(1)
  job = qvm.get_job(job_id)
print(job.result())
```

```
[[1]]
```

The *wait_for_job* method periodically checks for updates and prints the job's position in the queue, similar to the above code.

```
job = qvm.wait_for_job(job_id)
print(job.result())
```

```
[[1]]
```

3.6 Optimized Calls

This same pattern as above applies to the *wavefunction()*, *expectation()* and *run_and_measure()*. These are very useful if used appropriately: They all execute a given program *once and only once* and then either return the final wavefunction or use it to generate expectation values or a specified number of random bitstring samples.

Warning: This behavior can have unexpected consequences if the program that prepares the final state is non-deterministic, e.g., if it contains measurements and/or noisy gate applications. In this case, the final state after the program execution is itself a random variable and a single call to these functions therefore **cannot** sample the full space of outcomes. Therefore, if the program is non-deterministic and sampling the full program output distribution is important for the application at hand, we recommend using the basic *run()* API function as this re-runs the full program for every requested trial.

3.7 Exercises

3.7.1 Exercise 1 - Quantum Dice

Write a quantum program to simulate throwing an 8-sided die. The Python function you should produce is:

```
def throw_octahedral_die():
    # return the result of throwing an 8 sided die, an int between 1 and 8, by␣
↪running a quantum program
```

Next, extend the program to work for any kind of fair die:

```
def throw_polyhedral_die(num_sides):
    # return the result of throwing a num_sides sided die by running a quantum program
```

3.7.2 Exercise 2 - Controlled Gates

We can use the full generality of NumPy to construct new gate matrices.

1. Write a function `controlled` which takes a 2×2 matrix U representing a single qubit operator, and makes a 4×4 matrix which is a controlled variant of U, with the first argument being the *control qubit*.
2. Write a Quil program to define a controlled-Y gate in this manner. Find the wavefunction when applying this gate to qubit 1 controlled by qubit 0.

3.7.3 Exercise 3 - Grover's Algorithm

Write a quantum program for the single-shot Grover's algorithm. The Python function you should produce is:

```
# data is an array of 0's and 1's such that there are exactly three times as many
# 0's as 1's
def single_shot_grovers(data):
    # return an index that contains the value 1
```

As an example: `single_shot_grovers([0,0,1,0])` should return 2.

HINT - Remember that the Grover's diffusion operator is:

$$\begin{pmatrix} 2/N-1 & 2/N & \cdots & 2/N \\ 2/N & & & \\ \vdots & & \ddots & \\ 2/N & & & 2/N-1 \end{pmatrix}$$

CHAPTER 4

Next Steps

We hope that you have enjoyed your whirlwind tour of quantum computing. If you would like to learn more, Nielsen and Chuang's *Quantum Computation and Quantum Information* is a particularly excellent resource for newcomers to the field.

If you're interested in learning about the software behind quantum computing, take a look at our blog posts on The Quantum Software Challenge.

CHAPTER 5

The Rigetti QVM

The Rigetti Quantum Virtual Machine is an implementation of the Quantum Abstract Machine from *A Practical Quantum Instruction Set Architecture*.[1] It is implemented in ANSI Common LISP and executes programs specified in the Quantum Instruction Language (Quil). Quil is an opinionated quantum instruction language: its basic belief is that in the near term quantum computers will operate as coprocessors, working in concert with traditional CPUs. This means that Quil is designed to execute on a Quantum Abstract Machine that has a shared classical/quantum architecture at its core. The QVM is a wavefunction simulation of unitary evolution with classical control flow and shared quantum classical memory.

Most API keys give access to the QVM with up to 26 qubits. If you would like access to more qubits or help running larger jobs, then contact us at support@rigetti.com.

[1] https://arxiv.org/abs/1608.03355

CHAPTER 6

Examples of Quantum Programs on a QVM

To create intuition for a new class of algorithms, that will run on Quantum Virtual Machines (QVM), it is useful (and fun) to play with the abstraction that the software provides.

A broad class of programs that can easily be implemented on a QVM are generalizations of Game Theory to incorporate Quantum Strategies.

6.1 Meyer-Penny Game

A conceptually simple example that falls into this class is the Meyer-Penny Game. The game goes as follows: The Starship Enterprise, during one of its deep-space missions, is facing an immediate calamity, when a powerful alien suddenly appears on the bridge. The alien, named Q, offers to help Picard, the captain of the Enterprise, under the condition that Picard beats Q in a simple game of penny flips.

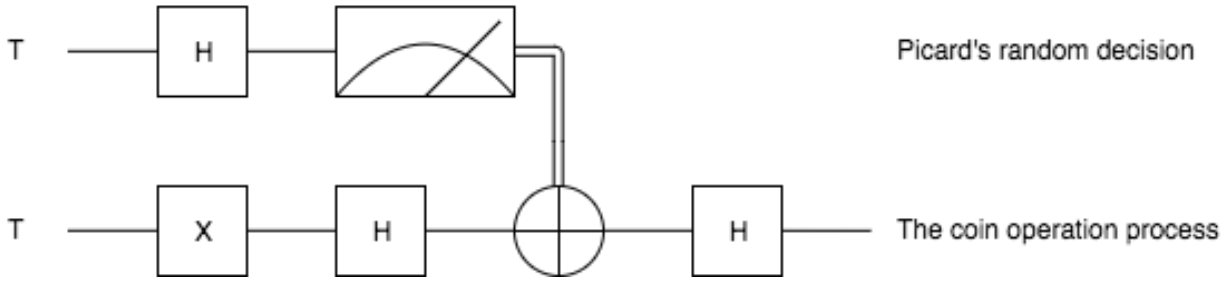
The rules: Picard is to place a penny Heads up into an opaque box. Then Picard and Q take turns to flip or not flip the penny without being able to see it; first Q then P then Q again. After this the penny is revealed;Q wins if it shows Heads (H), while Tails (T) makes Picard the winner.

Picard quickly estimates that his chance of winning is 50% and agrees to play the game. He loses the first round and insists on playing again. To his surprise Q agrees, and they continue playing several rounds more, each of which Picard loses. How is that possible?

What Picard did not anticipate is that Q has access to quantum tools. Instead of flipping the penny, Q puts the penny into a superposition of Heads and Tails proportional to the quantum state $|H\rangle + |T\rangle$. Then no matter whether Picard flips the penny or not, it will stay in a superposition (though the relative sign might change). In the third step Q undoes the superposition and always finds the penny to shows Heads.

To simulate the game we first construct the corresponding quantum circuit, which takes two qubits – one to simulate Picard's choice whether or not to flip the penny and the other to represent the penny. The initial state for all Qubits is $|0\rangle (= |T\rangle)$. To simulate Picard's decision, we assume that he chooses randomly whether or not to flip the coin, in agreement with the optimal strategy for the classic penny-flip game. This random choice can be created by putting one qubit into an equal superposition, e.g. with the Hadamard gate H, and then measure its state. The measurement will show Heads or Tails with equal probability p=0.5.

To simulate the penny flip game we take the second qubit and put it into its excited state $|1\rangle(= |H\rangle)$ by applying the X (or NOT) gate. Q's first move is to apply the Hadamard gate H. Picard's decision about the flip is simulated as a CNOT operation where the control bit is the outcome of the random number generator described above. Finally Q applies a Hadamard gate again, before we measure the outcome. The full circuit is shown in the figure below.



First we import all the necessary tools:

```
from pyquil.quil import Program
import pyquil.api as api

from pyquil.gates import I, H, X
qvm = api.QVMConnection()
```

Then we need to define two registers that will be used for the measurement of Picard's decision bit and the final answer of the penny tossing game.

```
picard_register = 1
answer_register = 0
```

Moreover we need to encode the two different actions of Picard, which conceptually is equivalent to an *if-else* control flow as:

```
then_branch = Program(X(0))
else_branch = Program(I(0))
```

and then wire it all up into the overall measurement circuit:

```
prog = (Program()
    # Prepare Qubits in Heads state or superposition, respectively
    .inst(X(0), H(1))
    # Q puts the penny into a superposition
    .inst(H(0))
    # Picard makes a decision and acts accordingly
    .measure(1, picard_register)
    .if_then(picard_register, then_branch, else_branch)
    # Q undoes his superposition operation
    .inst(H(0))
    # The outcome is recorded into the answer register
    .measure(0, answer_register))
```

Finally we play the game several times

```
qvm.run(prog, [0, 1], trials=10)
```

and record the register outputs as

```
[[1, 1],
 [1, 1],
 [1, 0],
 [1, 0],
 [1, 0],
 [1, 0],
 [1, 1],
 [1, 1],
 [1, 0],
 [1, 0]]
```

Remember that the first number is the outcome of the game (value of the *answer_register*) whereas the second number is the outcome of Picard's decision (value of the *picard_register*).

Indeed, no matter what Picard does, Q will always win!

6.2 Exercises

6.2.1 Prisoner's Dilemma

A classic strategy game is the prisoner's dilemma where two prisoners get the minimal penalty if they collaborate and stay silent, get zero penalty if one of them defects and the other collaborates (incurring maximum penalty) and get intermediate penalty if they both defect. This game has an equilibrium where both defect and incur intermediate penalty.

However, things change dramatically when we allow for quantum strategies leading to the Quantum Prisoner's Dilemma.

Can you design a program that simulates this game?

CHAPTER 7

Source Code Docs

Here you can find documentation for the different submodules in pyQuil.

7.1 pyquil.api

Module for facilitating connections to the QVM / QPU.

class pyquil.api.**QVMConnection**(*sync_endpoint=None*, *async_endpoint=None*, *api_key=None*, *user_id=None*, *gate_noise=None*, *measurement_noise=None*, *random_seed=None*)

Bases: pyquil.api._base_connection.BaseConnection

Represents a connection to the QVM.

expectation(*prep_prog*, *operator_programs=None*, *use_queue=False*)

Calculate the expectation value of operators given a state prepared by prep_program.

Note If the execution of quil_program is **non-deterministic**, i.e., if it includes measurements and/or noisy quantum gates, then the final wavefunction from which the expectation values are computed itself only represents a stochastically generated sample. The expectations returned from *different* expectation calls *will then generally be different*.

Parameters

- **prep_prog** (Program) – Quil program for state preparation.
- **operator_programs** (*list*) – A list of PauliTerms. Default is Identity operator.

Returns Expectation value of the operators.

Return type float

expectation_async(*prep_prog*, *operator_programs=None*)

Similar to expectation except that it returns a job id and doesn't wait for the program to be executed. See https://go.rigetti.com/connections for reasons to use this method.

ping()

run(*quil_program*, *classical_addresses*, *trials=1*, *use_queue=False*)
Run a Quil program multiple times, accumulating the values deposited in a list of classical addresses.

Parameters

- **quil_program** (Program) – A Quil program.
- **classical_addresses** (*list*) – A list of addresses.
- **trials** (*int*) – Number of shots to collect.
- **use_queue** (*bool*) – Disabling this parameter may improve performance for small, quick programs. To support larger programs, set it to True. (default: False)

Returns A list of lists of bits. Each sublist corresponds to the values in *classical_addresses*.

Return type list

run_and_measure(*quil_program*, *qubits*, *trials=1*, *use_queue=False*)
Run a Quil program once to determine the final wavefunction, and measure multiple times.

Note If the execution of `quil_program` is **non-deterministic**, i.e., if it includes measurements and/or noisy quantum gates, then the final wavefunction from which the returned bitstrings are sampled itself only represents a stochastically generated sample and the outcomes sampled from *different* `run_and_measure` calls *generally sample different bitstring distributions*.

Parameters

- **quil_program** (Program) – A Quil program.
- **qubits** (*list*) – A list of qubits.
- **trials** (*int*) – Number of shots to collect.

Returns A list of a list of bits.

Return type list

run_and_measure_async(*quil_program*, *qubits*, *trials=1*)
Similar to run_and_measure except that it returns a job id and doesn't wait for the program to be executed. See https://go.rigetti.com/connections for reasons to use this method.

run_async(*quil_program*, *classical_addresses*, *trials=1*)
Similar to run except that it returns a job id and doesn't wait for the program to be executed. See https://go.rigetti.com/connections for reasons to use this method.

wavefunction(*quil_program*, *classical_addresses=None*, *use_queue=False*)
Simulate a Quil program and get the wavefunction back.

Note If the execution of `quil_program` is **non-deterministic**, i.e., if it includes measurements and/or noisy quantum gates, then the final wavefunction from which the returned bitstrings are sampled itself only represents a stochastically generated sample and the wavefunctions returned by *different* `wavefunction` calls *will generally be different*.

Parameters

- **quil_program** (Program) – A Quil program.
- **classical_addresses** (*list*) – An optional list of classical addresses.

Returns A tuple whose first element is a Wavefunction object, and whose second element is the list of classical bits corresponding to the classical addresses.

Return type *Wavefunction*

wavefunction_async(*quil_program*, *classical_addresses=None*)
Similar to wavefunction except that it returns a job id and doesn't wait for the program to be executed. See https://go.rigetti.com/connections for reasons to use this method.

class pyquil.api.**QPUConnection**(*async_endpoint='https://job.rigetti.com/beta'*, *api_key=None*, *user_id=None*)
Bases: pyquil.api._base_connection.BaseConnection

Represents a connection to the QPU (Quantum Processing Unit)

run(*quil_program*, *classical_addresses*, *trials=1*)
Run a pyQuil program on the QPU. This functionality is in beta.

Parameters

- **quil_program** (Program) – Quil program to run on the QPU
- **classical_addresses** (*list*) – Currently unused
- **trials** (*int*) – Number of shots to take

Returns A list of lists of bits. Each sublist corresponds to the values in *classical_addresses*.

Return type list

run_and_measure(*quil_program*, *qubits*, *trials=1*)
Run a pyQuil program on the QPU multiple times, measuring all the qubits in the QPU simultaneously at the end of the program each time. This functionality is in beta.

Parameters

- **quil_program** (Program) – A Quil program.
- **qubits** (*list*) – The list of qubits to measure
- **trials** (*int*) – Number of shots to collect.

Returns A list of a list of bits.

Return type list

run_and_measure_async(*quil_program*, *qubits*, *trials*)
Similar to run_and_measure except that it returns a job id and doesn't wait for the program to be executed. See https://go.rigetti.com/connections for reasons to use this method.

run_async(*quil_program*, *classical_addresses*, *trials=1*)
Similar to run except that it returns a job id and doesn't wait for the program to be executed. See https://go.rigetti.com/connections for reasons to use this method.

7.2 pyquil.gates

A lovely bunch of gates and instructions for programming with. This module is used to provide Pythonic sugar for Quil instructions.

pyquil.gates.**AND**(*classical_reg1*, *classical_reg2*)
Produce an AND instruction.

Parameters

- **classical_reg1** – The first classical register.
- **classical_reg2** – The second classical register, which gets modified.

Returns A ClassicalAnd instance.

pyquil.gates.**CCNOT**(**qubits*)

pyquil.gates.**CNOT**(**qubits*)

pyquil.gates.**CPHASE**(**params*)

pyquil.gates.**CPHASE00**(**params*)

pyquil.gates.**CPHASE01**(**params*)

pyquil.gates.**CPHASE10**(**params*)

pyquil.gates.**CSWAP**(**qubits*)

pyquil.gates.**CZ**(**qubits*)

pyquil.gates.**EXCHANGE**(*classical_reg1*, *classical_reg2*)
Produce an EXCHANGE instruction.

Parameters

- **classical_reg1** – The first classical register, which gets modified.
- **classical_reg2** – The second classical register, which gets modified.

Returns A ClassicalExchange instance.

pyquil.gates.**FALSE**(*classical_reg*)
Produce a FALSE instruction.

Parameters **classical_reg** – A classical register to modify.

Returns A ClassicalFalse instance.

pyquil.gates.**H**(**qubits*)

pyquil.gates.**I**(**qubits*)

pyquil.gates.**ISWAP**(**qubits*)

pyquil.gates.**MEASURE**(*qubit*, *classical_reg=None*)
Produce a MEASURE instruction.

Parameters

- **qubit** – The qubit to measure.
- **classical_reg** – The classical register to measure into, or None.

Returns A Measurement instance.

pyquil.gates.**MOVE**(*classical_reg1*, *classical_reg2*)
Produce a MOVE instruction.

Parameters

- **classical_reg1** – The first classical register.
- **classical_reg2** – The second classical register, which gets modified.

Returns A ClassicalMove instance.

pyquil.gates.**NOT**(*classical_reg*)
Produce a NOT instruction.

Parameters **classical_reg** – A classical register to modify.

Returns A ClassicalNot instance.

pyquil.gates.**OR**(*classical_reg1*, *classical_reg2*)
Produce an OR instruction.

Parameters

- **classical_reg1** – The first classical register.
- **classical_reg2** – The second classical register, which gets modified.

Returns A ClassicalOr instance.

pyquil.gates.**PHASE**(**params*)

pyquil.gates.**PSWAP**(**params*)

pyquil.gates.**RX**(**params*)

pyquil.gates.**RY**(**params*)

pyquil.gates.**RZ**(**params*)

pyquil.gates.**S**(**qubits*)

pyquil.gates.**SWAP**(**qubits*)

pyquil.gates.**T**(**qubits*)

pyquil.gates.**TRUE**(*classical_reg*)
Produce a TRUE instruction.

Parameters **classical_reg** – A classical register to modify.

Returns A ClassicalTrue instance.

pyquil.gates.**X**(**qubits*)

pyquil.gates.**Y**(**qubits*)

pyquil.gates.**Z**(**qubits*)

pyquil.gates.**unpack_classical_reg**(*c*)
Get the address for a classical register.

Parameters **c** – A list of length 1 or an int or an Addr.

Returns The address as an Addr.

7.3 pyquil.paulis

Module for working with Pauli algebras.

pyquil.paulis.**ID**()
The identity Pauli Term.

class pyquil.paulis.**PauliSum**(*terms*)
Bases: object

A sum of one or more PauliTerms.

get_qubits()
The support of all the operators in the PauliSum object.

Returns A list of all the qubits in the sum of terms.

Return type list

simplify()
Simplifies the sum of Pauli operators according to Pauli algebra rules.

class pyquil.paulis.**PauliTerm**(*op*, *index*, *coefficient=1.0*)
Bases: object

A term is a product of Pauli operators operating on different qubits.

copy()
Properly creates a new PauliTerm, with a completely new dictionary of operators

classmethod from_list(*terms_list*, *coefficient=1.0*)
Allocates a Pauli Term from a list of operators and indices. This is more efficient than multiplying together individual terms.

Parameters **terms_list** (*list*) – A list of tuples, e.g. [("X", 0), ("Y", 1)]

Returns PauliTerm

get_qubits()
Gets all the qubits that this PauliTerm operates on.

id()

Returns the unique identifier string for the PauliTerm (ignoring the coefficient). Used in the simplify method of PauliSum.

Returns The unique identifier for this term.

Return type string

exception pyquil.paulis.**UnequalLengthWarning**(**args*, ***kwargs*)
Bases: exceptions.Warning

pyquil.paulis.**ZERO**()
The zero Pauli Term.

pyquil.paulis.**check_commutation**(*pauli_list*, *pauli_two*)
Check if commuting a PauliTerm commutes with a list of other terms by natural calculation. Derivation similar to arXiv:1405.5749v2 fo the check_commutation step in the Raesi, Wiebe, Sanders algorithm (arXiv:1108.4318, 2011).

Parameters

- **pauli_list** (*list*) – A list of PauliTerm objects
- **pauli_two_term** (PauliTerm) – A PauliTerm object

Returns True if pauli_two object commutes with pauli_list, False otherwise

Return type bool

pyquil.paulis.**commuting_sets**(*pauli_terms*, *nqubits*)
Gather the Pauli terms of pauli_terms variable into commuting sets

Uses algorithm defined in (Raeisi, Wiebe, Sanders, arXiv:1108.4318, 2011) to find commuting sets. Except uses commutation check from arXiv:1405.5749v2

Parameters **pauli_terms** (PauliSum) – A PauliSum object

Returns List of lists where each list contains a commuting set

Return type list

pyquil.paulis.**exponential_map**(*term*)
Creates map alpha -> exp(-1j*alpha*term) represented as a Program.

Parameters **term** (PauliTerm) – Tests is a PauliTerm is the identity operator

Returns Program

Return type *Program*

pyquil.paulis.**exponentiate**(*term*)
Creates a pyQuil program that simulates the unitary evolution exp(-1j * term)

Parameters **term** (PauliTerm) – Tests is a PauliTerm is the identity operator

Returns A Program object

Return type *Program*

pyquil.paulis.**is_identity**(*term*)
Check if Pauli Term is a scalar multiple of identity

Parameters **term** (PauliTerm) – A PauliTerm object

Returns True if the PauliTerm is a scalar multiple of identity, false otherwise

Return type bool

pyquil.paulis.**is_zero**(*pauli_object*)
Tests to see if a PauliTerm or PauliSum is zero.

Parameters **pauli_object** – Either a PauliTerm or PauliSum

Returns True if PauliTerm is zero, False otherwise

Return type bool

pyquil.paulis.**sI**(*q*)
A function that returns the identity operator on a particular qubit.

Parameters **qubit_index** (*int*) – The index of the qubit

Returns A PauliTerm object

Return type *PauliTerm*

pyquil.paulis.**sX**(*q*)
A function that returns the sigma_X operator on a particular qubit.

Parameters **qubit_index** (*int*) – The index of the qubit

Returns A PauliTerm object

Return type *PauliTerm*

pyquil.paulis.**sY**(*q*)
A function that returns the sigma_Y operator on a particular qubit.

Parameters **qubit_index** (*int*) – The index of the qubit

Returns A PauliTerm object

Return type *PauliTerm*

pyquil.paulis.**sZ**(*q*)
A function that returns the sigma_Z operator on a particular qubit.

Parameters **qubit_index** (*int*) – The index of the qubit

Returns A PauliTerm object

Return type *PauliTerm*

pyquil.paulis.**suzuki_trotter**(*trotter_order*, *trotter_steps*)
Generate trotterization coefficients for a given number of Trotter steps.

U = exp(A + B) is approximated as exp(w1*o1)exp(w2*o2)... This method returns a list [(w1, o1), (w2, o2), ... , (wm, om)] of tuples where o=0 corresponds to the A operator, o=1 corresponds to the B operator, and w is the coefficient in the exponential. For example, a second order Suzuki-Trotter approximation to exp(A + B) results in the following [(0.5/trotter_steps, 0), (1/trotteri_steps, 1), (0.5/trotter_steps, 0)] * trotter_steps.

Parameters

- **trotter_order** (*int*) – order of Suzuki-Trotter approximation
- **trotter_steps** (*int*) – number of steps in the approximation

Returns List of tuples corresponding to the coefficient and operator type: o=0 is A and o=1 is B.

Return type list

pyquil.paulis.**term_with_coeff**(*term*, *coeff*)
Change the coefficient of a PauliTerm.

Parameters

- **term** (PauliTerm) – A PauliTerm object
- **coeff** (*Number*) – The coefficient to set on the PauliTerm

Returns A new PauliTerm that duplicates term but sets coeff

Return type *PauliTerm*

pyquil.paulis.**trotterize**(*first_pauli_term*, *second_pauli_term*, *trotter_order=1*, *trotter_steps=1*)
Create a Quil program that approximates exp((A + B)t) where A and B are PauliTerm operators.

Parameters

- **first_pauli_term** (PauliTerm) – PauliTerm denoted *A*
- **second_pauli_term** (PauliTerm) – PauliTerm denoted *B*
- **trotter_order** (*int*) – Optional argument indicating the Suzuki-Trotter approximation order–only accepts orders 1, 2, 3, 4.
- **trotter_steps** (*int*) – Optional argument indicating the number of products to decompose the exponential into.

Returns Quil program

Return type *Program*

7.4 pyquil.parametric

Module for creating and defining parametric programs.

class pyquil.parametric.**ParametricProgram**(*program_constructor*)
Bases: object

Note: Experimental

A class representing Programs with changeable gate parameters.

fuse(*other*)

Note: Experimental

Fuse another program to this one.

Parameters **other** – A Program or ParametricProgram.

Returns A new ParametricProgram.

Return type *ParametricProgram*

pyquil.parametric.**argument_count**(*thing*)
Get the number of arguments a callable has.

Parameters **thing** – A callable.

Returns The number of arguments it takes.

Return type int

pyquil.parametric.**parametric**(*decorated_function*)

Note: Experimental

A decorator to change a function into a ParametricProgram.

Parameters **decorated_function** – The function taking parameters producing a Program object.

Returns a callable ParametricProgram

Return type *ParametricProgram*

7.5 pyquil.quil

Module for creating and defining Quil programs.

class pyquil.quil.**Program**(**instructions*)
Bases: object

alloc()
Get a new qubit.

Returns A qubit.

Return type *Qubit*

dagger(*inv_dict=None*, *suffix='-INV'*)
Creates the conjugate transpose of the Quil program. The program must not contain any irreversible actions (measurement, control flow, qubit allocation).

Returns The Quil program's inverse

Return type *Program*

defgate(*name*, *matrix*)
Define a new static gate.

Parameters

- **name** (*string*) – The name of the gate.
- **matrix** (*array-like*) – List of lists or Numpy 2d array.

Returns The Program instance.

Return type *Program*

define_noisy_gate(*name*, *qubit_indices*, *kraus_ops*)
Overload a static ideal gate with a noisy one defined in terms of a Kraus map.

Parameters

- **name** (*str*) – The name of the gate.
- **qubit_indices** (*tuple|list*) – The qubits it acts on.
- **kraus_ops** (*tuple|list*) – The Kraus operators.

Returns The Program instance

Return type *Program*

defined_gates
A list of defined gates on the program.

gate(*name*, *params*, *qubits*)
Add a gate to the program.

Parameters

- **name** (*string*) – The name of the gate.
- **params** (*list*) – Parameters to send to the gate.
- **qubits** (*list*) – Qubits that the gate operates on.

Returns The Program instance

Return type *Program*

get_qubits()
Returns all of the qubit indices used in this program, including gate applications and allocated qubits. e.g.

```
>>> p = Program()
>>> p.inst(("H", 1))
>>> p.get_qubits()
{1}
>>> q = p.alloc()
>>> p.inst(H(q))
>>> len(p.get_qubits())
2
```

Returns A set of all the qubit indices used in this program

Return type set

if_then(*classical_reg*, *if_program*, *else_program=None*)
If the classical register at index classical reg is 1, run if_program, else run else_program.

Equivalent to the following construction: IF [c]:

instrA...

ELSE: instrB...

=> JUMP-WHEN @THEN [c] instrB... JUMP @END LABEL @THEN instrA... LABEL @END

Parameters

- **classical_reg** (*int*) – The classical register to check as the condition
- **if_program** (Program) – A Quil program to execute if classical_reg is 1
- **else_program** (Program) – A Quil program to execute if classical_reg is 0. This argument is optional and defaults to an empty Program.

Returns The Quil Program with the branching instructions added.

Return type *Program*

inst (**instructions*)

Mutates the Program object by appending new instructions.

This function accepts a number of different valid forms, e.g.

```
>>> p = Program()
>>> p.inst(H(0)) # A single instruction
>>> p.inst(H(0), H(1)) # Multiple instructions
>>> p.inst([H(0), H(1)]) # A list of instructions
>>> p.inst(("H", 1)) # A tuple representing an instruction
>>> p.inst("H 0") # A string representing an instruction
>>> q = Program()
>>> p.inst(q) # Another program
```

It can also be chained:

```
>>> p = Program()
>>> p.inst(H(0)).inst(H(1))
```

Parameters **instructions** – A list of Instruction objects, e.g. Gates

Returns self for method chaining

instructions

Fill in any placeholders and return a list of quil AbstractInstructions.

is_protoquil()

Protoquil programs may only contain gates, no classical instructions and no jumps.

Returns True if the Program is Protoquil, False otherwise

measure (*qubit_index*, *classical_reg=None*)

Measures a qubit at qubit_index and puts the result in classical_reg

Parameters

- **qubit_index** (*int*) – The address of the qubit to measure.
- **classical_reg** (*int*) – The address of the classical bit to store the result.

Returns The Quil Program with the appropriate measure instruction appended, e.g. MEASURE 0 [1]

Return type *Program*

measure_all(**qubit_reg_pairs*)
Measures many qubits into their specified classical bits, in the order they were entered.

Parameters **qubit_reg_pairs** (*Tuple*) – Tuples of qubit indices paired with classical bits.

Returns The Quil Program with the appropriate measure instructions appended, e.g. MEASURE 0 [1] MEASURE 1 [2] MEASURE 2 [3]

Return type *Program*

no_noise()
Prevent a noisy gate definition from being applied to the immediately following Gate instruction.

Returns Program

out()
Converts the Quil program to a readable string.

Returns String form of a program

Return type string

pop()
Pops off the last instruction.

Returns The instruction that was popped.

Return type tuple

while_do(*classical_reg*, *q_program*)
While a classical register at index classical_reg is 1, loop q_program

Equivalent to the following construction:

WHILE [c]: instr...

=> LABEL @START JUMP-UNLESS @END [c] instr... JUMP @START LABEL @END

Parameters

- **classical_reg** (*int*) – The classical register to check
- **q_program** (Program) – The Quil program to loop.

Returns The Quil Program with the loop instructions added.

Return type *Program*

pyquil.quil.**merge_programs**(*prog_list*)
Merges a list of pyQuil programs into a single one by appending them in sequence

Parameters **prog_list** (*list*) – A list of pyquil programs

Returns a single pyQuil program

Return type *Program*

7.6 pyquil.quilbase

Contains the core pyQuil objects that correspond to Quil instructions.

class pyquil.quilbase.**AbstractInstruction**
Bases: object

Abstract class for representing single instructions.

out()

class pyquil.quilbase.**Addr**(*value*)
Bases: *pyquil.quilbase.QuilAtom*

Representation of a classical bit address.

Parameters **value** (*int*) – The classical address.

out()

class pyquil.quilbase.**BinaryClassicalInstruction**(*left*, *right*)
Bases: *pyquil.quilbase.AbstractInstruction*

The abstract class for binary classical instructions.

out()

class pyquil.quilbase.**ClassicalAnd**(*left*, *right*)
Bases: *pyquil.quilbase.BinaryClassicalInstruction*

op = 'AND'

class pyquil.quilbase.**ClassicalExchange**(*left*, *right*)
Bases: *pyquil.quilbase.BinaryClassicalInstruction*

op = 'EXCHANGE'

class pyquil.quilbase.**ClassicalFalse**(*target*)
Bases: *pyquil.quilbase.UnaryClassicalInstruction*

op = 'FALSE'

class pyquil.quilbase.**ClassicalMove**(*left*, *right*)
Bases: *pyquil.quilbase.BinaryClassicalInstruction*

op = 'MOVE'

class pyquil.quilbase.**ClassicalNot**(*target*)
Bases: *pyquil.quilbase.UnaryClassicalInstruction*

op = 'NOT'

class pyquil.quilbase.**ClassicalOr**(*left*, *right*)
Bases: *pyquil.quilbase.BinaryClassicalInstruction*

op = 'OR'

class pyquil.quilbase.**ClassicalTrue**(*target*)
Bases: *pyquil.quilbase.UnaryClassicalInstruction*

op = 'TRUE'

class pyquil.quilbase.**DefGate**(*name*, *matrix*)
Bases: *pyquil.quilbase.AbstractInstruction*

A DEFGATE directive.

Parameters

- **name** (*string*) – The name of the newly defined gate.
- **matrix** (*array-like*) – {list, nparray, np.matrix} The matrix defining this gate.

get_constructor()

Returns A function that constructs this gate on variable qubit indices. E.g. *mygate.get_constructor()(1) applies the gate to qubit 1.*

num_args()

Returns The number of qubit arguments the gate takes.

Return type int

out()
Prints a readable Quil string representation of this gate.

Returns String representation of a gate

Return type string

class pyquil.quilbase.**Gate**(*name*, *params*, *qubits*)
Bases: *pyquil.quilbase.AbstractInstruction*

This is the pyQuil object for a quantum gate instruction.

out()

class pyquil.quilbase.**Halt**
Bases: *pyquil.quilbase.SimpleInstruction*

The HALT instruction.

op = 'HALT'

class pyquil.quilbase.**Jump**(*target*)
Bases: *pyquil.quilbase.AbstractInstruction*

Representation of an unconditional jump instruction (JUMP).

out()

class pyquil.quilbase.**JumpConditional**(*target*, *condition*)
Bases: *pyquil.quilbase.AbstractInstruction*

Abstract representation of an conditional jump instruction.

out()

class pyquil.quilbase.**JumpTarget**(*label*)
Bases: *pyquil.quilbase.AbstractInstruction*

Representation of a target that can be jumped to.

out()

class pyquil.quilbase.**JumpUnless**(*target*, *condition*)
Bases: *pyquil.quilbase.JumpConditional*

The JUMP-UNLESS instruction.

op = 'JUMP-UNLESS'

class pyquil.quilbase.**JumpWhen**(*target*, *condition*)
Bases: *pyquil.quilbase.JumpConditional*

The JUMP-WHEN instruction.

op = 'JUMP-WHEN'

class pyquil.quilbase.**Label**(*label_name*)
Bases: *pyquil.quilbase.QuilAtom*

Representation of a label.

Parameters **label_name** (*string*) – The label name.

out()

class pyquil.quilbase.**LabelPlaceholder**(*prefix='L'*)
Bases: *pyquil.quilbase.Label*

name

class pyquil.quilbase.**Measurement**(*qubit*, *classical_reg=None*)
Bases: *pyquil.quilbase.AbstractInstruction*

This is the pyQuil object for a Quil measurement instruction.

out()

class pyquil.quilbase.**Nop**
Bases: *pyquil.quilbase.SimpleInstruction*

The RESET instruction.

op = 'NOP'

class pyquil.quilbase.**Pragma**(*command*, *args=()*, *freeform_string=''*)
Bases: *pyquil.quilbase.AbstractInstruction*

A PRAGMA instruction.

This is printed in QUIL as:

```
PRAGMA <command> <arg1> <arg2> ... <argn> "<freeform_string>"
```

out()

class pyquil.quilbase.**Qubit**(*index*)
Bases: *pyquil.quilbase.QuilAtom*

Representation of a qubit.

Parameters **index** (*int*) – Index of the qubit.

out()

class pyquil.quilbase.**QubitPlaceholder**
Bases: *pyquil.quilbase.Qubit*

index

class pyquil.quilbase.**QuilAtom**
Bases: object

Abstract class for atomic elements of Quil.

out()

class pyquil.quilbase.**RawInstr**(*instr_str*)
Bases: *pyquil.quilbase.AbstractInstruction*

A raw instruction represented as a string.

out()

class pyquil.quilbase.**Reset**
Bases: *pyquil.quilbase.SimpleInstruction*

The RESET instruction.

op = 'RESET'

class pyquil.quilbase.**SimpleInstruction**
Bases: *pyquil.quilbase.AbstractInstruction*

Abstract class for simple instructions with no arguments.

out()

class pyquil.quilbase.**UnaryClassicalInstruction**(*target*)
Bases: *pyquil.quilbase.AbstractInstruction*

The abstract class for unary classical instructions.

out()

class pyquil.quilbase.**Wait**
Bases: *pyquil.quilbase.SimpleInstruction*

The WAIT instruction.

op = 'WAIT'

pyquil.quilbase.**check_for_pi**(*element*)
Check to see if there exists a rational number r = p/q in reduced form for which the difference between element/np.pi and r is small and q <= 8.

Parameters **element** – float

Return element pretty print string if true, else standard representation.

pyquil.quilbase.**format_parameter**(*element*)
Formats a particular parameter. Essentially the same as built-in formatting except using 'i' instead of 'j' for the imaginary number.

Parameters **element** – {int, float, long, complex, Slot} Formats a parameter for Quil output.

pyquil.quilbase.**unpack_qubit**(*qubit*)
Get a qubit from an object.

Parameters **qubit** – An int or Qubit.

Returns A Qubit instance

7.7 pyquil.slot

Contains Slot pyQuil placeholders for constructing Quil template programs.

class pyquil.slot.**Slot**(*value=0.0*, *func=None*)
Bases: object

A placeholder for a parameter value.

Arithmetic operations: +−∗/
Logical: abs, max, <, >, <=, >=, !=, ==
Arbitrary functions are not supported

Parameters

- **value** (*float*) – A value to initialize to. Defaults to 0.0
- **func** (*function*) – An initial function to determine the final parameterized value.

value()
Computes the value of this Slot parameter.

7.8 pyquil.wavefunction

Module containing the Wavefunction object and methods for working with wavefunctions.

class pyquil.wavefunction.**Wavefunction**(*amplitude_vector*, *classical_memory=None*)
Bases: object

static from_bit_packed_string(*coef_string*, *classical_addresses*)
From a bit packed string, unpacks to get the wavefunction and classical measurement results :param coef_string: :param classical_addresses: :return:

get_outcome_probs()
Parses a wavefunction (array of complex amplitudes) and returns a dictionary of outcomes and associated probabilities.

Returns A dict with outcomes as keys and probabilities as values.

Return type dict

static ground(*qubit_num*)

plot(*qubit_subset=None*)
Plots a bar chart with bitstring on the x axis and probability on the y axis.

Parameters **qubit_subset** (*list*) – Optional parameter used for plotting a subset of the Hilbert space.

pretty_print(*decimal_digits=2*)
Returns a string repr of the wavefunction, ignoring all outcomes with approximately zero amplitude (up to a certain number of decimal digits) and rounding the amplitudes to decimal_digits.

Parameters **decimal_digits** (*int*) – The number of digits to truncate to.

Returns A dict with outcomes as keys and complex amplitudes as values.

Return type str

pretty_print_probabilities(*decimal_digits=2*)
Prints outcome probabilities, ignoring all outcomes with approximately zero probabilities (up to a certain number of decimal digits) and rounding the probabilities to decimal_digits.

Parameters **decimal_digits** (*int*) – The number of digits to truncate to.

Returns A dict with outcomes as keys and probabilities as values.

Return type dict

static zeros(*qubit_num*)
Constructs the groundstate wavefunction for a given number of qubits.

Parameters **qubit_num** (*int*) –

Returns A Wavefunction in the ground state

Return type *Wavefunction*

pyquil.wavefunction.**get_bitstring_from_index**(*index*, *qubit_num*)

Returns the bitstring in lexical order that corresponds to the given index in 0 to 2^(qubit_num) :param int index: :param int qubit_num: :return: the bitstring :rtype: str

CHAPTER 8

Indices and Tables

- genindex
- modindex
- search

Python Module Index

p

Index

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z