pyQuil Documentation

Release 1.9.1.dev0

Rigetti Computing

Jul 10, 2018

Contents

1 Overview **1**

2 Contents **3**

2.1 Introduction to Quantum Computing . . . 3
2.1.1 From Bit to Qubit . . . 3
2.1.2 Qubit Operations . . . 8
2.1.3 The Quantum Abstract Machine . . . 11
2.1.4 Next Steps . . . 16
2.2 Installation and Getting Started . . . 17
2.2.1 Connecting to Rigetti Forest . . . 17
2.2.2 Getting Started . . . 18
2.2.3 Your First Quantum Program . . . 18
2.3 The Basics: Programs and Gates . . . 19
2.3.1 Some Program Construction Features . . . 19
2.3.2 Fixing a Mistaken Instruction . . . 20
2.3.3 The Standard Gate Set . . . 21
2.3.4 Defining New Gates . . . 21
2.3.5 Defining Parametric Gates . . . 22
2.4 Advanced Usage . . . 22
2.4.1 Quantum Fourier Transform (QFT) . . . 22
2.4.2 Classical Control Flow . . . 24
2.4.3 Parametric Depolarizing Noise . . . 25
2.4.4 Parametric Programs . . . 26
2.4.5 Pauli Operator Algebra . . . 26
2.5 Exercises . . . 27
2.5.1 Exercise 1: Quantum Dice . . . 27
2.5.2 Exercise 2: Controlled Gates . . . 28
2.5.3 Exercise 3: Grover's Algorithm . . . 28
2.6 The Quantum Virtual Machine (QVM) . . . 28
2.6.1 Using the QVM . . . 29
2.6.2 Multi-Qubit Basis Enumeration . . . 31
2.6.3 Examples of Quantum Programs . . . 31
2.7 The Quantum Processing Unit (QPU) . . . 34
2.7.1 Using the QPU . . . 34
2.7.2 Agave QPU Properties . . . 38
2.7.3 Acorn QPU Properties – CURRENTLY UNAVAILABLE . . . 40

2.8 The Quil Compiler . . . 43
  2.8.1 Expectations for Program Contents . . . 43
  2.8.2 Interacting with the Compiler . . . 43
  2.8.3 Region-specific compiler features through PRAGMA . . . 44
  2.8.4 Common Error Messages . . . 46
2.9 Using Qubit Placeholders . . . 47
  2.9.1 Register . . . 48
2.10 Noise and Quantum Computation . . . 49
  2.10.1 Modeling Noisy Quantum Gates . . . 49
  2.10.2 Noisy Gates on the Rigetti QVM . . . 51
  2.10.3 Adding Decoherence Noise . . . 57
  2.10.4 Modeling Readout Noise . . . 60
  2.10.5 Working with Readout Noise . . . 63
2.11 Source Code Documentation . . . 68
  2.11.1 pyquil.api . . . 68
  2.11.2 pyquil.device . . . 75
  2.11.3 pyquil.gates . . . 79
  2.11.4 pyquil.noise . . . 80
  2.11.5 pyquil.parametric . . . 86
  2.11.6 pyquil.paulis . . . 87
  2.11.7 pyquil.quil . . . 91
  2.11.8 pyquil.quilbase . . . 95
  2.11.9 pyquil.slot . . . 98
  2.11.10 pyquil.wavefunction . . . 99
2.12 Changelog . . . 100
  2.12.1 v1.9 (June 6, 2018) . . . 100

3 Indices and Tables **103**

Bibliography **105**

Python Module Index **107**

CHAPTER 1

Overview

pyQuil is part of the Rigetti Forest toolkit for **quantum programming in the cloud**. 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 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 cloud-based **Quantum Virtual Machine** (QVM). This is a classical simulation of a quantum processor that can simulate various qubit operations. 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 in the *The Quantum Virtual Machine (QVM)*.

In addition to the QVM, we offer the ability to run programs on our superconducting quantum processors, or **Quantum Processing Units** (QPUs), at our lab in Berkeley, California. To request upgraded access to our 19Q QPU, please fill out the request form with a brief summary of what you hope to use it for. For more information on QPUs, check out *The Quantum Processing Unit (QPU)*.

If you are already familiar with quantum computing, then feel free to proceed to *Installation and Getting Started*. Otherwise, take a look at our *Introduction to Quantum Computing*, where we use Quil introduce the basics of quantum computing and the Quantum Abstract Machine on which it runs.

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

CHAPTER 2

Contents

2.1 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.

2.1.1 From Bit to Qubit

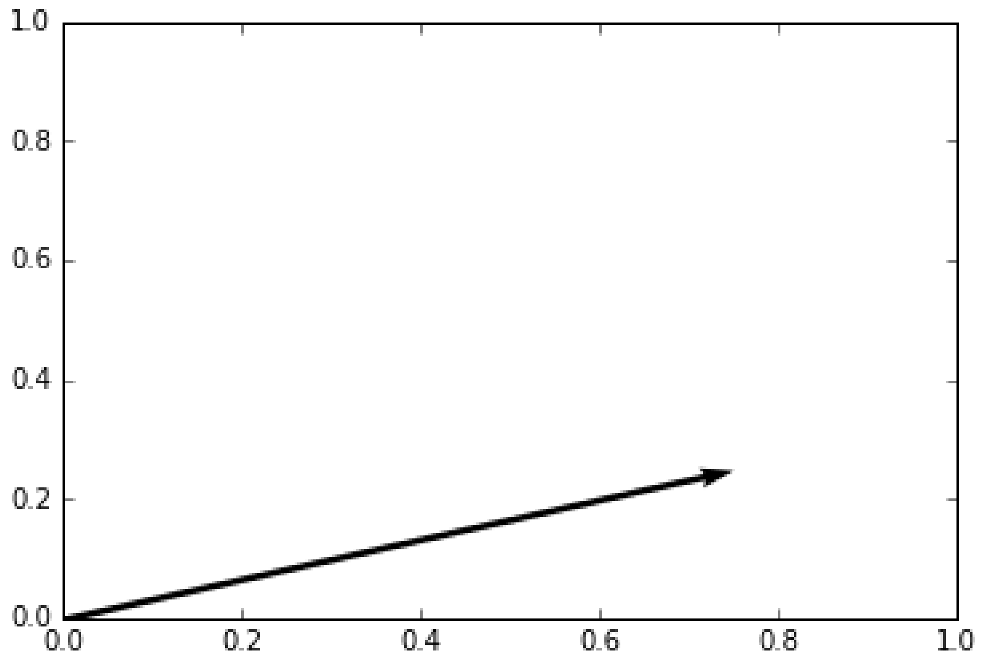
Probabilistic Bits as Vector Spaces

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.

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

$$\begin{aligned}\vec{v} &= |\,v\rangle \\ \vec{v}^T &= \langle v\,| \\ \vec{u}^T.\vec{v} &= \langle u\,|\,v\rangle.\end{aligned}$$

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

$$\begin{aligned}Pr(0) &= \langle v\,|\,0\rangle = a \\ Pr(1) &= \langle v\,|\,1\rangle = b.\end{aligned}$$

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

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

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

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 \(\bigotimes_{i=0}^{n-1}|\,v_i\rangle\).

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

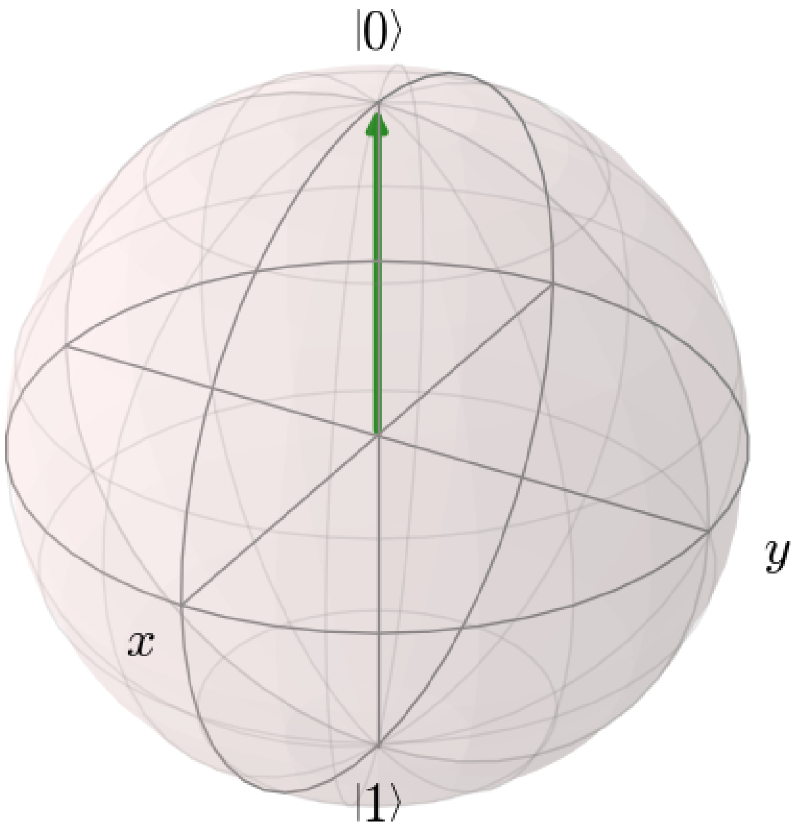
$$\alpha|\,0\rangle + \beta|\,1\rangle.$$

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:

$$\begin{aligned}\Pr(0) &= |\langle v\,|\,0\rangle|^2 = |\alpha|^2 \\ \Pr(1) &= |\langle v\,|\,1\rangle|^2 = |\beta|^2,\end{aligned}$$

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.

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.

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
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))
```

```
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))
wavefunction = quantum_simulator.wavefunction(p)
print("The quantum state is of dimension:", len(wavefunction.amplitudes))

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

p = Program()
for x in range(10):
    p += I(x)
wavefunction = quantum_simulator.wavefunction(p)
print("The quantum state is of dimension:", len(wavefunction.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
wavefunction = quantum_simulator.wavefunction(Program(I(0), I(1)))
print(wavefunction.get_outcome_probs())
```

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

2.1.2 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+0j)|0>
```

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))
wavefunction = quantum_simulator.wavefunction(p)
print("X|0> = ", wavefunction)
print("The outcome probabilities are", wavefunction.get_outcome_probs())
print("This looks like a bit flip.\n")

p = Program(Y(0))
wavefunction = quantum_simulator.wavefunction(p)
print("Y|0> = ", wavefunction)
print("The outcome probabilities are", wavefunction.get_outcome_probs())
print("This also looks like a bit flip.\n")

p = Program(Z(0))
wavefunction = quantum_simulator.wavefunction(p)
print("Z|0> = ", wavefunction)
print("The outcome probabilities are", wavefunction.get_outcome_probs())
print("This state looks unchanged.")
```

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

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

Z|0> =  (1+0j)|0>
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))
wavefunction = quantum_simulator.wavefunction(p)

print("ZYX|0> = ", wavefunction)
print("With outcome probabilities\n", wavefunction.get_outcome_probs())
```

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

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))
wavefunction = quantum_simulator.wavefunction(p)
print("CNOT|00> = ", wavefunction)
print("With outcome probabilities\n", wavefunction.get_outcome_probs())

p = Program(X(0), CNOT(0, 1))
wavefunction = quantum_simulator.wavefunction(p)
print("CNOT|01> = ", wavefunction)
print("With outcome probabilities\n", wavefunction.get_outcome_probs())

p = Program(X(1), CNOT(0, 1))
wavefunction = quantum_simulator.wavefunction(p)
print("CNOT|10> = ", wavefunction)
print("With outcome probabilities\n", wavefunction.get_outcome_probs())

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

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

CNOT|01> =  (1+0j)|11>
```

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

CNOT|10> =  (1+0j)|10>
With outcome probabilities
 {'00': 0.0, '01': 0.0, '10': 1.0, '11': 0.0}

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

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:

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

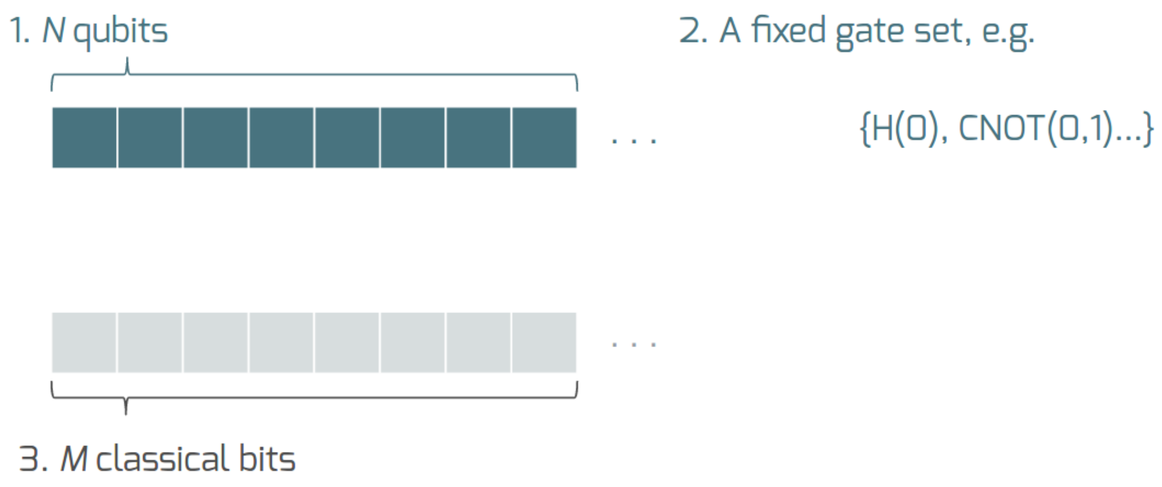
```
from pyquil.gates import SWAP
p = Program(X(0), SWAP(0,1))
wavefunction = quantum_simulator.wavefunction(p)

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

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

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.

2.1.3 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).

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]
print(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]
print(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]]
```

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))
wavefunction = quantum_simulator.wavefunction(coin_program)

print("H|0> = ", wavefunction)
print("With outcome probabilities\n", wavefunction.get_outcome_probs())
```

```
H|0> =  (0.7071067812+0j)|0> + (0.7071067812+0j)|1>
With outcome probabilities
{'0': 0.49999999999999989, '1': 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
print(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:  (1+0j)|1>
After measurement:  (1+0j)|0>
After measurement:  (1+0j)|0>
After measurement:  (1+0j)|1>
After measurement:  (1+0j)|1>
```

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))
wavefunction = quantum_simulator.wavefunction(bell_program)
print("Before measurement: Bell state = ", wavefunction)

bell_program.measure(0, classical_reg)
for x in range(5):
    wavefunction = quantum_simulator.wavefunction(bell_program)
    print("After measurement: ", wavefunction.get_outcome_probs())
```

```
Before measurement: Bell state =  (0.7071067812+0j)|00> + (0.7071067812+0j)|11>

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

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.

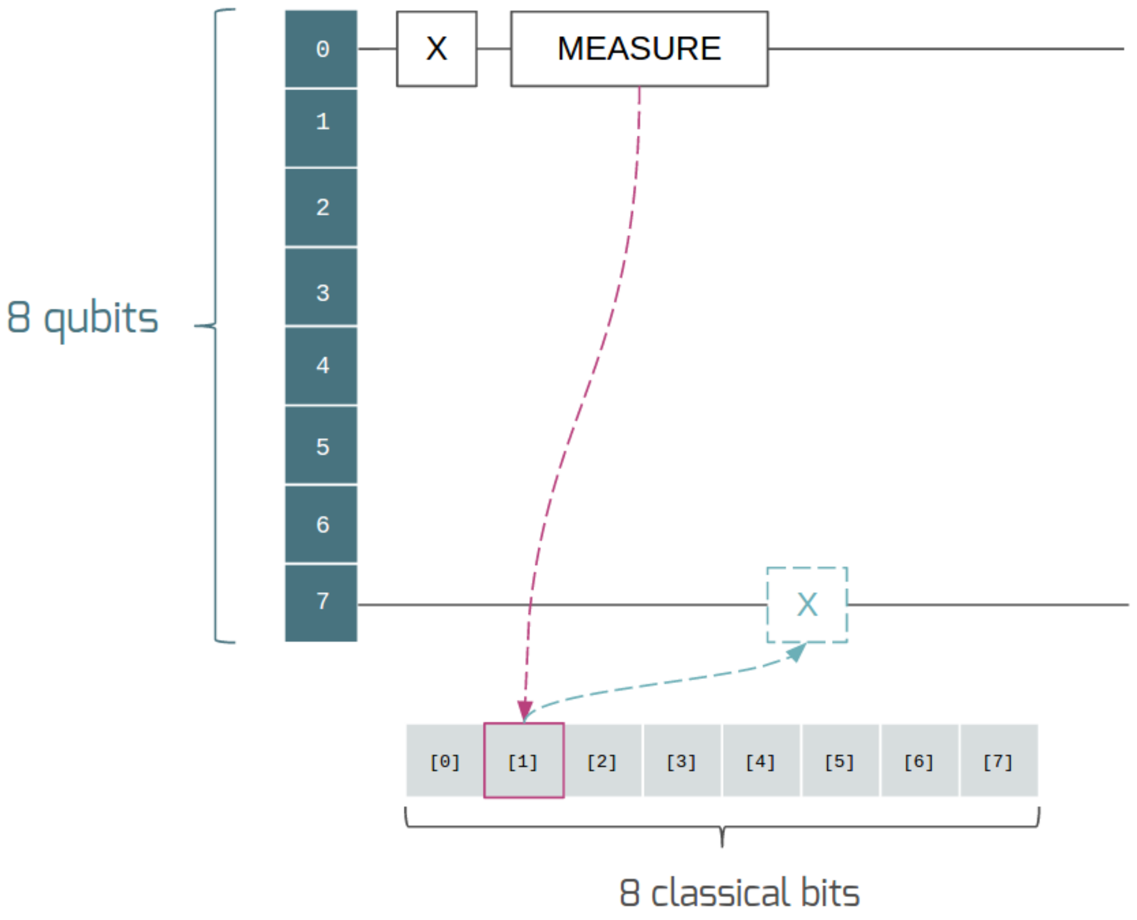
Classical Control

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

# Branch on classical reg [1]
p = Program(X(0)).measure(0, 1).if_then(1, true_branch, false_branch)

# Measure qubit #7 into classical register [7]
p.measure(7, 7)

# Run and check register [7]
print(quantum_simulator.run(p, [7]))
```

```
[[1]]
```

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

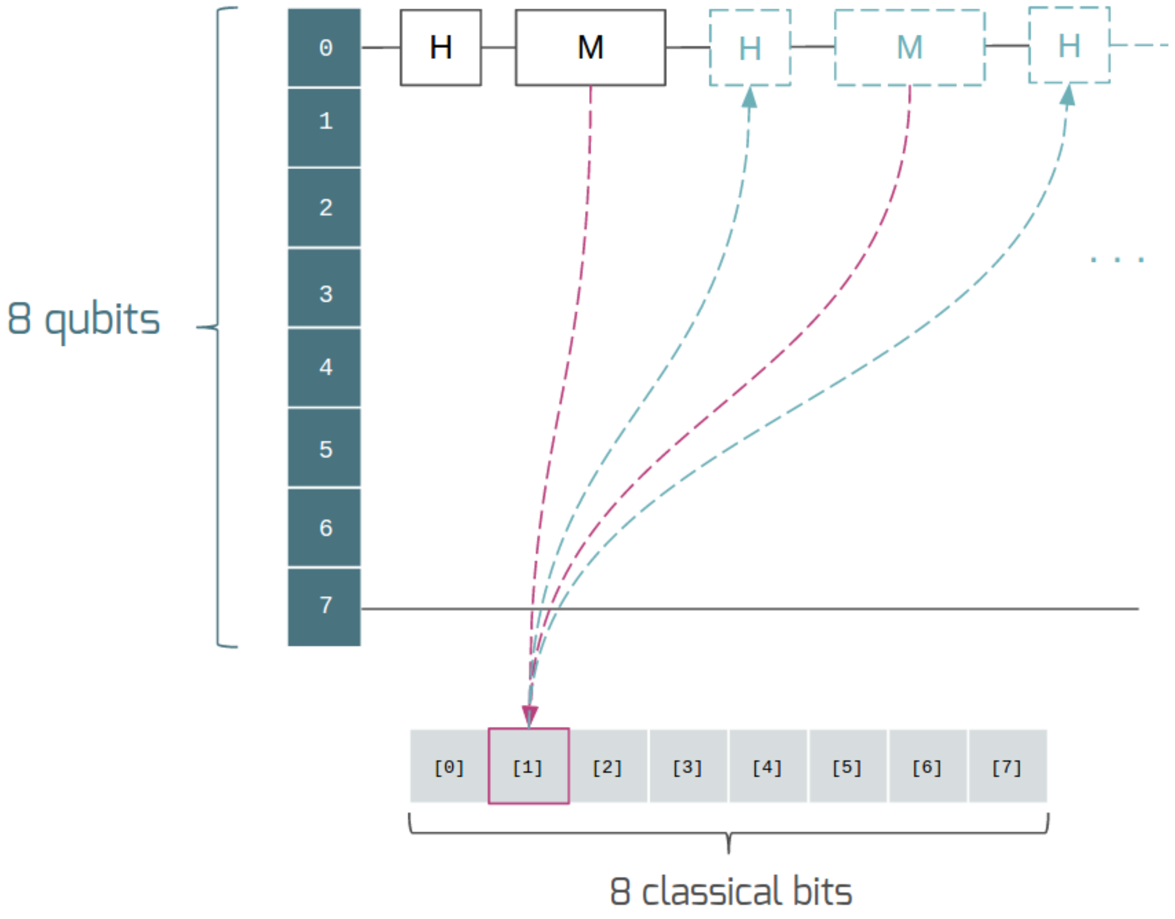


Example: The Probabilistic Halting Problem

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)

# Run and check register [1]
print(quantum_simulator.run(p, [1]))
```

```
[[0]]
```



2.1.4 Next Steps

We hope that you have enjoyed your whirlwind tour of quantum computing. You are now ready to check out the Installation and Getting Started guide!

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.

2.2 Installation and Getting Started

Make sure you have a current version of Python installed on your computer. We recommend installing the Anaconda Python distribution.

Then, install pyQuil with:

```
conda install -c rigetti pyquil
```

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.

Note: We also support installation via `pip` with `pip install pyquil`

2.2.1 Connecting to 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. We also highly recommend that you 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.

2.2.2 Getting Started

This toolkit provides some simple libraries for writing quantum programs.

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

qvm = QVMConnection()
p = Program(H(0), CNOT(0, 1))

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

```
(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. **pyQuil**: A Python library to help write and run Quil code and quantum programs.
3. **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.
4. **QPU**: pyQuil also includes some a special connection which lets you run experiments on Rigetti's prototype superconducting quantum processors over the cloud.
5. **Quilc**: In addition to running on the QVM or the QPU, users can directly use the Quil compiler, to investigate how arbitrary quantum programs can be compiled to target specific physical instruction set architectures (ISAs).

2.2.3 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 successfully interacted with 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.

2.3 The Basics: Programs and Gates

Quantum programs are written in Forest using the `Program` object from the `quil` module.

```
from pyquil.quil import Program
p = Program()
```

Programs are then constructed from quantum gates, which can be found in the `gates` module. We can add quantum gates to programs in numerous ways, including using the `.inst(...)` method. We use the `.measure(...)` method to measure qubits into classical registers:

```
from pyquil.gates import X
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 Quantum Virtual Machine, or QVM:

```
from pyquil.api import QVMConnection
qvm = QVMConnection()

print(qvm.run(p, [0]))
```

Congratulations! You just ran a program on the QVM. The returned value should be:

```
[[1]]
```

For more information on what the above result means, and on executing quantum programs on the QVM in general, see *The Quantum Virtual Machine (QVM)*. Feel free to skip ahead and read about executing programs on the QVM (and the QPU for that matter), but don't forget to come back. The remainder of this section of the docs will be dedicated to constructing programs in detail, an essential part of becoming fluent in quantum programming.

2.3.1 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
```

2.3.2 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
```

2.3.3 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.

2.3.4 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>
```

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))
```

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

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

2.3.5 Defining Parametric Gates

It is also possible to define parametric gates using pyQuil. Let's say we want to have a controlled RX gate. Since RX is a parametric gate, we need a slightly different way of defining it than in the previous section.

```
from pyquil.parameters import Parameter, quil_sin, quil_cos
from pyquil.quilbase import DefGate
import numpy as np

theta = Parameter('theta')
crx = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, quil_cos(theta / 2), -1j * quil_
↪sin(theta / 2)], [0, 0, -1j * quil_sin(theta / 2), quil_cos(theta / 2)]])

dg = DefGate('CRX', crx, [theta])
CRX = dg.get_constructor()

p = Program()
p.inst(dg)
p.inst(H(0))
p.inst(CRX(np.pi/2)(0, 1))

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

```
(0.7071067812+0j)|00> + (0.5+0j)|01> + -0.5j|11>
```

`quil_sin` and `quil_cos` work as the regular sinus and cosinus, but they support the parametrization. Parametrized functions you can use with pyQuil are: `quil_sin`, `quil_cos`, `quil_sqrt`, `quil_exp`, and `quil_cis`.

2.4 Advanced Usage

2.4.1 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))
wavefunction = qvm.wavefunction(state_prep + add_dummy_qubits)
print(wavefunction)
```

```
(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.

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

2.4.2 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]]
```

2.4.3 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]]
```

2.4.4 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.

2.4.5 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.

2.5 Exercises

2.5.1 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
```

2.5.2 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.

2.5.3 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:

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

2.6 The Quantum Virtual Machine (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 30 qubits. If you would like access to more qubits or help running larger jobs, then contact us at support@rigetti.com. On request we may also provide access to a QVM that allows persistent wavefunction memory between different programs as well as direct access to the wavefunction memory (wrapped as a `numpy` array) from python.

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

2.6.1 Using the QVM

The QVM is available in pyQuil via the `api` module.

```
from pyquil.api import QVMConnection
qvm = QVMConnection()
```

One executes quantum programs on the QVM using two paradigms: the `.run(...)` method, and the `.wavefunction(...)` method. The former closely mirrors how one will execute programs on a real QPU (see *Using the QPU*), while the latter takes advantage of the virtual machine, and allows direct access to the wavefunction. These two methods are described in the following two sections. (For information on constructing quantum programs, please refer back to *The Basics: Programs and Gates*.)

The `.run(...)` method

```
program = Program(X(0), MEASURE(0, 0))
results = qvm.run(program, trials=1)
# results = [[1]]
```

The `.run(...)` method takes numerous arguments, several of which are optional. The most important are

1. the `program` to be executed on the QVM,
2. the `classical_addresses` which to be returned from the QVM (not included above; by default, these are set to the addresses used in the program's `MEASURE` instructions), and
3. the number of `trials` to be executed on the machine.

The results returned are a *list of lists of integers*. In the above case, that's

```
[[1]]
```

Let's unpack this. The *outer* list is an enumeration over the trials; if you set `trials=1` then `len(results)` should equal `1`.

The *inner* list, on the other hand, is an enumeration over the results stored in the classical addresses. 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` 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.

The `.wavefunction(...)` method

The QVM is a virtual machine. As such, we can directly inspect the wavefunction of a program, even without measurements, using the `.wavefunction(...)` method:

```
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))
wavefunction = qvm.wavefunction(coin_flip)
print(wavefunction)
```

```
(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 wavefunction[0] == 1 / np.sqrt(2)
# The amplitudes are stored as a numpy array on the Wavefunction object
print(wavefunction.amplitudes)
prob_dict = wavefunction.get_outcome_probs() # extracts the probabilities of outcomes␣
↪as a dict
print(prob_dict)
prob_dict.keys() # these stores the bitstring outcomes
assert len(wavefunction) == 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)
wavefunction = qvm.wavefunction(coin_flip, classical_addresses=range(9))
classical_mem = wavefunction.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.

2.6.2 Multi-Qubit Basis Enumeration

The Rigetti QVM enumerates bitstrings such that qubit *0* is the least significant bit (LSB) and therefore on the right end of a bitstring as shown in the table below which contains some examples.

bitstring	qubit_(n-1)	…	qubit_2	qubit_1	qubit_0
1…101	1	…	1	0	1
0…110	0	…	1	1	0

This convention is counter to that often found in the quantum computing literature where bitstrings are often ordered such that the lowest-index qubit is on the left. The vector representation of a wavefunction assumes the "canonical" ordering of basis elements. I.e., for two qubits this order is `00, 01, 10, 11`. In the typical Dirac notation for quantum states, the tensor product of two different degrees of freedom is not always explicitly understood as having a fixed order of those degrees of freedom. This is in contrast to the kronecker product between matrices which uses the same mathematical symbol and is clearly not commutative. This, however, becomes important when writing things down as coefficient vectors or matrices:

$$0_0 \otimes 1_1 = 1_1 \otimes 0_0 = 10_{1,0} \equiv \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

As a consequence there arise some subtle but important differences in the ordering of wavefunction and multi-qubit gate matrix coefficients. According to our conventions the matrix

$$U_{\mathrm{CNOT}(1,0)} \equiv \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

corresponds to the Quil instruction `CNOT(1, 0)` which is counter to how most other people in the field order their tensor product factors (or more specifically their kronecker products). In this convention `CNOT(0, 1)` is given by

$$U_{\mathrm{CNOT}(0,1)} \equiv \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

For additional information why we decided on this basis ordering check out our note *Someone shouts, "|01000>!" Who is Excited?*[2].

2.6.3 Examples of Quantum Programs

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.

[2] https://arxiv.org/abs/1711.02086

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.

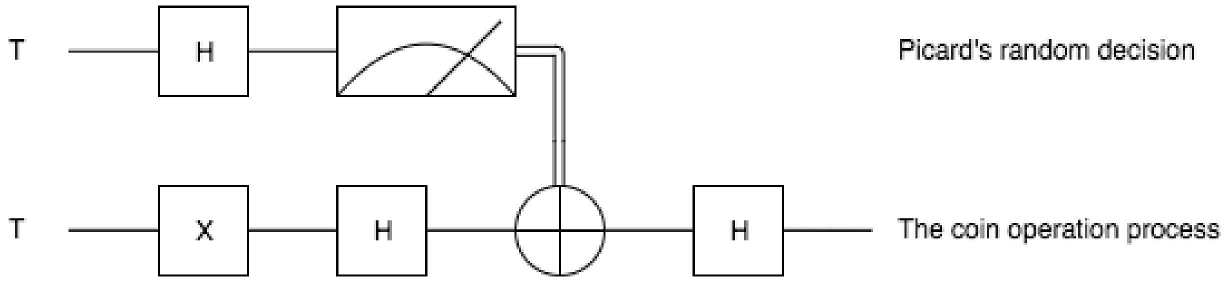
The rules: Picard is to place a penny Heads up into an opaque box. Then Picard and Q take turns to flip or not flip the penny without being able to see it; first Q then P then Q again. After this the penny is revealed;Q wins if it shows Heads (H), while Tails (T) makes Picard the winner.

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!

Exercises

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?

2.7 The Quantum Processing Unit (QPU)

A quantum processing unit (QPU), also referred to as a *quantum chip*, is a physical (fabricated) chip that contains a number of interconnected qubits. It is the foundational component of a full quantum computer, which includes the housing environment for the QPU, the control electronics, and many other components.

This page describes how to use the Forest API for interacting with Rigetti QPUs, and provides technical details and average performance of **Agave**, the 8Q QPU currently available, that has been designed, fabricated and packaged by Rigetti.

2.7.1 Using the QPU

Note: User permissions for QPU access must be enabled by a Forest administrator. `QPUConnection` calls will automatically fail without these user permissions. Speak to a Forest administrator for information about upgrading your access plan.

One establishes a connection to a Rigetti QPU in the same manner as a QVM:

```
from pyquil.api import QPUConnection
qpu = QPUConnection() # NOTE: This raises a UserWarning!
```

There is one caveat, however, as shown in the `UserWarning` that is raised by the above command: You must specify a `device` as an argument. This is described in the following section.

Accessing available `devices` with `get_devices()`

The initialization function for a `QPUConnection` object must be provided a speciffic Rigetti QPU as an argument, so that Forest knows on which quantum computer you want to execute your programs. The available QPUs, synonymously referred to as `devices` in Forest, can be inspected via the function `get_devices` in the `api` module:

```
from pyquil.api import get_devices
for device in get_devices():
    if device.is_online():
        print('Device {} is online'.format(device.name))
```

Note: The `Device` objects returned by `get_devices` captures other characteristics about the associated QPU, such as its connectivity, coherence times, single- and two-qubit gate fidelities. For more information on the `Device` class, see *Getting QPU Information from the Device Class*.

Devices are typically named according to the convention `[n]Q-[name]`, where `n` is the number of active qubits on the device and `name` is a human-readable name that designates the device.

Execution on the QPU

One may execute Quil programs on the QPU (nearly) identically to the QVM, via the `.run(...)` method (obviously, since the QPU is a real quantum computer, the `.wavefunction(...)` method is not available). We may fix the above example then by providing a device to the `QPUConnection`:

```
from pyquil.api import get_devices, QPUConnection

agave = get_devices(as_dict=True)['8Q-Agave']
qpu = QPUConnection(agave)
# The device name as a string is also acceptable
# qpu = QPUConnection('8Q-Agave')
```

You have now established a connection to the `8Q-Agave` QPU. Executing programs is then identical to the QVM (we may ommit the `classical_addresses` and `trials` arguments to use their defaults):

```
from pyquil.quil import Program
from pyquil.gates import X, MEASURE

program = Program(X(0), MEASURE(0, 0))
qpu.run(program, [0])
```

In addition to the `.run(...)` method, a `QPUConnection` object provides the following methods:

- `.run(quil_program, classical_addresses, trials=1)`: This method sends the `Program` object `quil_program` to the QPU for execution, which runs the program `trials` many times. After each run on the QPU, all the qubits in the QPU are simultaneously measured and their results are stored in classical registers according to the MEASURE instructions provided. Then, a list of registers listed in `classical_addresses` is returned to the user for each trial. This call is blocking: it will wait until the QPU returns its results for inspection.
- `.run_async(quil_program, classical_addresses, trials=1)`: This method has identical behavior to `.run` except that it is **nonblocking**, and it instead returns a job ID string.
- `.run_and_measure(quil_program, qubits, trials=1)`: This method sends the `Program` object `quil_program` to the QPU for execution, which runs the program `trials` many times. After each run on the QPU, the all the qubits in the QPU are simultaneously measured, and the results from those listed in `qubits` are returned to the user for each trial. This call is blocking: it will wait until the QPU returns its results for inspection.
- `.run_and_measure_async(quil_program, qubits, trials=1)`: This method has identical behavior to `.run_and_measure` except that it is **nonblocking**, and it instead returns a job ID string.

Note: The QPU's `run` functionality matches that of the QVM's `run` functionality, but the behavior of `run_and_measure` **does not match** its `QVMConnection` counterpart (cf. Optimized Calls). The `QVMConnection` version of `run` repeats the execution of a program many times, producing a (potentially) different outcome each time, whereas `run_and_measure` executes a program only once and uses the QVM's unique ability to perform wavefunction inspection to multiply sample the same distribution. The QPU **does not** have this ability, and thus its `run_and_measure` call behaves as the QVM's `run`.

For example, the following Python snippet demonstrates the execution of a small job on the QPU identified as "8Q-Agave":

```
from pyquil.quil import Program
import pyquil.api as api
from pyquil.gates import *
qpu = api.QPUConnection('8Q-Agave')
p = Program(H(0), CNOT(0, 1), MEASURE(0, 0), MEASURE(1, 1))
qpu.run(p, [0, 1], 1000)
```

When the QPU execution time is expected to be long and there is classical computation that the program would like to accomplish in the meantime, the `QPUConnection` object allows for an asynchronous `run_async` call to be placed instead. By storing the resulting job ID, the state of the job and be queried later and its results obtained then. The mechanism for querying the state of a job is also through the `QPUConnection` object: a job ID string can be transformed to a `pyquil.api.Job` object via the method `.get_job(job_id)`; the state of a `Job` object (taken at its creation time) can then be inspected by the method `.is_done()`; and when this returns `True` the output of the QPU can be retrieved via the method `.result()`.

For example, consider the following Python snippet:

```
from pyquil.quil import Program
import pyquil.api as api
from pyquil.gates import *
qpu = api.QPUConnection('8Q-Agave')
p = Program(H(0), CNOT(0, 1), MEASURE(0, 0), MEASURE(1, 1))
job_id = qpu.run_async(p, [0, 1], 1000)
while not qpu.get_job(job_id).is_done():
    ## get some other work done while we wait
    ...
    ## and eventually yield to recheck the job result
## now the job is guaranteed to be finished, so pull the QPU results
job_result = qpu.get_job(job_id).result()
```

Getting QPU Information from the Device Class

The pyQuil `Device` class provides useful information for learning about, and working with, Rigetti's available QPUs. One may query for available devices using the `get_devices` function:

```
from pyquil.api import get_devices

devices = get_devices(as_dict=True)
# E.g. {'8Q-Agave': <Device 8Q-Agave online>, '19Q-Acorn': <Device 19Q-Acorn offline>}

agave = devices['8Q-Agave']
```

The variable `agave` points to a `Device` object that holds useful information regarding the QPU, including:

1. Connectivity via its instruction set architecture (`agave.isa` of class `ISA`).
2. Hardware specifications such as coherence times and fidelities (`agave.specs` of class `Specs`).
3. Noise model information (`agave.noise_model` of class `NoiseModel`).

These 3 attributes are accessed in the following ways (note that the specs shown below may be out of date):

```
  print(agave.specs)
  # Specs(qubits_specs=..., edges_specs=...)

  print(agave.specs.qubits_specs)
  """
  [_QubitSpecs(id=0, fRO=0.7841, f1QRB=0.9575, T1=1.07e-05, T2=1.06e-05),
   _QubitSpecs(id=1, fRO=0.9099, f1QRB=0.9513, T1=1e-05, T2=9.2e-06),
   _QubitSpecs(id=2, fRO=0.9427, f1QRB=0.9825, T1=1.55e-05, T2=1.25e-05),
   _QubitSpecs(id=3, fRO=0.9122, f1QRB=0.9703, T1=1.42e-05, T2=1.85e-05),
   _QubitSpecs(id=4, fRO=0.6777, f1QRB=0.9693, T1=1.46e-05, T2=2.62e-05),
   _QubitSpecs(id=5, fRO=0.8319, f1QRB=0.9624, T1=1.49e-05, T2=1.28e-05),
   _QubitSpecs(id=6, fRO=0.7526, f1QRB=0.969, T1=1.42e-05, T2=1.29e-05),
   _QubitSpecs(id=7, fRO=0.8954, f1QRB=0.9322, T1=1.32e-05, T2=1.77e-05)]
   """
  print(agave.isa)
  # ISA(qubits=..., edges=...)

  print(agave.isa.edges)
  """
[Edge(targets=[0, 1], type='CZ', dead=False),
 Edge(targets=[0, 7], type='CZ', dead=False),
 Edge(targets=[1, 2], type='CZ', dead=False),
 Edge(targets=[2, 3], type='CZ', dead=False),
 Edge(targets=[3, 4], type='CZ', dead=False),
 Edge(targets=[4, 5], type='CZ', dead=False),
 Edge(targets=[5, 6], type='CZ', dead=False),
 Edge(targets=[6, 7], type='CZ', dead=False)]
  """

  print(agave.noise_model)
  # NoiseModel(gates=[KrausModel(...) ...] ...)
```

Additionally, the `Specs` class provides methods for access specs info across the chip in a more succinct manner:

```
agave.specs.T1s()
# {0: 1.07e-05, 1: 1e-05, 2: 1.55e-05, 3: 1.42e-05, 4: 1.46e-05, 5: 1.49e-05, 6: 1.
↪42e-05, 7: 1.32e-05}

agave.specs.fCZs()
# {(0, 1): 0.9176, (0, 7): 0.9095, (1, 2): 0.9065, (2, 3): 0.8191, (3, 4): 0.87, (4,␣
↪5): 0.67, (5, 6): 0.9302, (6, 7): 0.9295}
```

With these tools provided by the `Device` class, users may learn more about Rigetti hardware, and construct programs tailored specifically to that hardware. The `Device` class can also be used to seed a QVM with characteristics of the device, supporting noisy simulation. For more information on this, see the next section.

Simulating the QPU using the QVM

The QVM is a powerful tool for testing quantum programs before executing them on the QPU. In addition to the `noise.py` module for generating custom noise models for simulating noise on the QVM, pyQuil provides a simple interface for loading the QVM with noise models tailored to Rigetti's available QPUs, in just one modified line of code. This is made possible via the `Device` class, which holds hardware specification information, noise model information, and instruction set architecture (ISA) information regarding connectivity. This information is held in the `Specs`, `ISA` and `NoiseModel` attributes of the `Device` class, respectively.

Specifically, to load a QVM with the `NoiseModel` information from a `Device`, all that is required is to provide a `Device` object to the QVM during initialization:

```
from pyquil.api import get_devices, QVMConnection

agave = get_devices(as_dict=True)['8Q-Agave']
qvm = QVMConnection(agave)
```

By simply providing a device during QVM initialization, all programs executed on this QVM will, by default, have noise applied that is characteristic of the corresponding Rigetti QPU (in the case above, the `agave` device). One may then efficiently test realistic quantum algorithms on the QVM, in advance of running those programs on the QPU.

Retune Interruptions

Because the QPU is a physical device, it is occasionally taken offline for recalibration. This offline period typically lasts 10-40 minutes, depending upon QPU characteristics and other external factors. During this period, the QPU will be listed as offline, and it will reject new jobs (but pending jobs will remain queued). When the QPU resumes activity, its performance characteristics may be slightly different (in that different gates may enjoy different process fidelities).

2.7.2 Agave QPU Properties

The quantum processor consists of 8 superconducting transmon qubits with fixed capacitive coupling in the planar lattice design shown in Fig. 1.

The resonant frequencies of qubits 0, 2, 4, and 6 are tunable while qubits 1, 3, 5, and 7 are fixed. The former set has two Josephson junctions in an asymmetric SQUID geometry to provide roughly 1 GHz of frequency tunability, and flux-insensitive "sweet spots" near

$\omega^{\rm max}_{01}/2\pi \approx 4.8\,{\rm GHz}$

and

$\omega^{\rm min}_{01}/2\pi \approx 3.3\,{\rm GHz}$.

These tunable devices are coupled to bias lines for AC and DC flux delivery. Each qubit is capacitively coupled to a quasi-lumped element resonator for dispersive readout of the qubit state. Single-qubit control is effected by applying microwave drives at the resonator ports. Two-qubit gates are activated via RF drives on the flux bias lines.

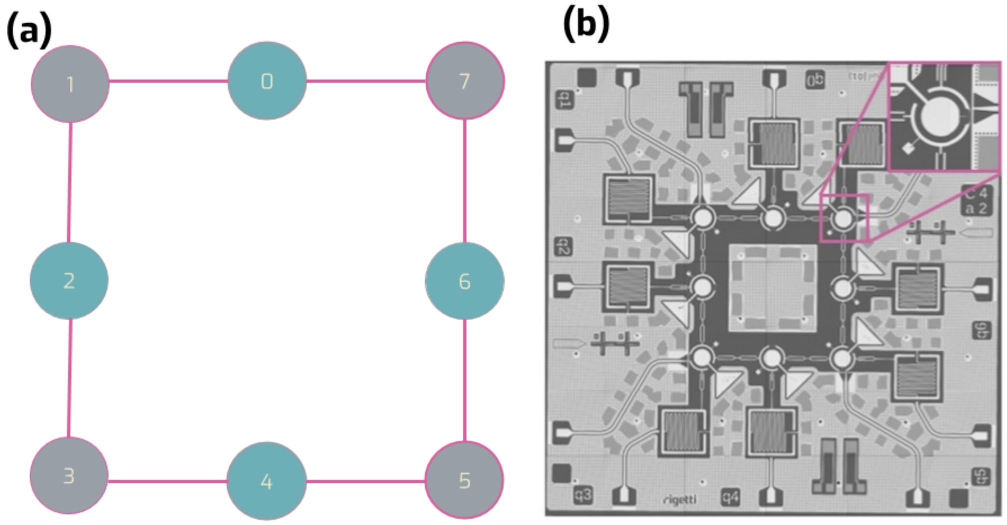


Fig. 2.1: **Figure 1 | Connectivity of Rigetti 8Q. a,** Chip schematic showing tunable transmons (green circles) capacitively coupled to fixed-frequency transmons (blue circles). **b**, Optical image of an 8Q chip, representative of Agave.

1-Qubit Gate Performance

The device is characterized by several parameters:

- $\omega_{01}/2\pi$ is the qubit transition frequency
- $\omega_{\rm r}/2\pi$ is the resonator frequency

- $\eta/2\pi$ is the anharmonicity of the qubit
- $g/2\pi$ is the coupling strength between a qubit and a resonator
- $\lambda/2\pi$ is the coupling strength between two neighboring qubits

In Rigetti 8Q, each tunable qubit is capacitively coupled to two fixed-frequency qubits. We use a parametric flux modulation to activate a controlled Z gate between tunable and fixed qubits. The typical time-scale of these entangling gates is in the range 100–250 ns.

Table 1 summarizes the main performance parameters of Rigetti 8Q. The resonator and qubit frequencies are measured with standard spectroscopic techniques. The relaxation time T_1 is extracted from repeated inversion recovery experiments. Similarly, the coherence time T_2^* is measured with repeated Ramsey fringe experiments. Single-qubit gate fidelities are estimated with randomized benchmarking protocols in which a sequence of m Clifford gates is applied to the qubit followed by a measurement on the computational basis. The sequence of Clifford gates are such that the first $m-1$ gates are chosen uniformly at random from the Clifford group, while the last Clifford gate is chosen to bring the state of the system back to the initial state. This protocol is repeated for different values of $m \in \{2, 4, 8, 16, 32, 64, 128\}$. The reported single-qubit gate fidelity is related to the randomized benchmarking decay constant p in the following way: $\mathsf{F}_{1q} = p + (1-p)/2$. Finally, the readout assignment fidelities are extracted with dispersive readouts combined with a linear classifier trained on $|0\rangle$ and $|1\rangle$ state preparation for each qubit. The reported readout assignment fidelity is given by expression $\mathsf{F}_{RO} = [p(0|0) + p(1|1)]/2$, where $p(b|a)$ is the probability of measuring the qubit in state b when prepared in state a.

Table 2.1: **Table 1 | Rigetti 8Q performance**

	$\omega_r^{\max}/2\pi$	$\omega_{01}^{\max}/2\pi$	T_1	T_2^*	F_{1q}	F_{RO}
	MHz	MHz	μs	μs		
0	5863	4586	10.72	10.6	0.957	0.784
1	5293	3909	10.04	9.2	0.951	0.910
2	5713	4524	15.52	12.5	0.982	0.943
3	5411	4054	14.17	18.5	0.970	0.912
4	5620	4660	14.58	26.2	0.969	0.678
5	5171	4081	14.86	12.8	0.962	0.832
6	5751	4760	14.17	12.9	0.969	0.753
7	5454	4110	13.19	17.7	0.932	0.895

Qubit-Qubit Coupling

The coupling strength between two qubits can be extracted from a precise measurement of the shift in qubit frequency after the neighboring qubit is in the excited state. This protocol consists of two steps: a π pulse is applied to the first qubit, followed by a Ramsey fringe experiment on the second qubit which precisely determines its transition frequency (see Fig. 2a). The effective shift is denoted by χ_{qq} and typical values are in the range $\approx 100\,\text{kHz}$. The coupling strength λ between the two qubits can be calculated in the following way:

$$\lambda^{(1,2)} = \sqrt{\left|\frac{\chi_{qq}^{(1,2)}\left[f_{01}^{(1)} - f_{12}^{(2)}\right]\left[f_{12}^{(1)} - f_{01}^{(2)}\right]}{2(\eta_1 + \eta_2)}\right|}$$

Figure 2b shows the coupling strength for our device. This quantity is crucial to predict the gate time of our parametric entangling gates.

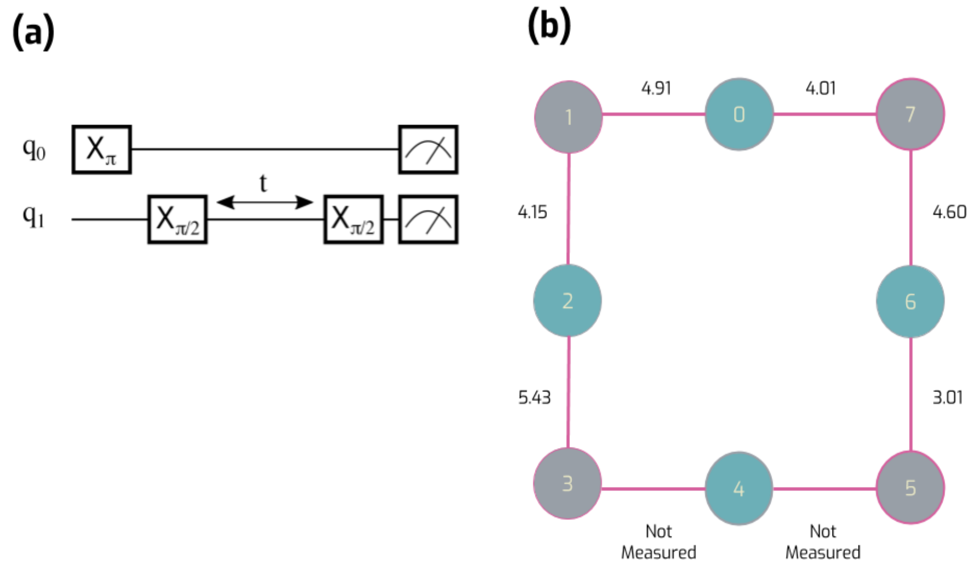


Fig. 2.2: **Figure 2 | Coupling strength. a,** Quantum circuit implemented to measure the qubit-qubit effective frequency shift. **b,** Capacitive coupling between neighboring qubits expressed in MHz.

2-Qubit Gate Performance

Table 2 shows the two-qubit gate performance of Rigetti 8Q. These parameters refer to parametric CZ gates performed on one pair at a time. We analyze these CZ gates through quantum process tomography (QPT). This procedure starts by applying local rotations to the two qubits taken from the set $\{I, R_x(\pi/2), R_y(\pi/2), R_x(\pi)\}$, followed by a CZ gate and post-rotations that bring the qubit states back to the computational basis. QPT involves the analysis of $16 \times 16 = 256$ different experiments, each of which we repeat 500 times. The reported fidelity $\mathsf{F}^{\mathrm{cptp}}_{\mathrm{PT}}$ is the *average gate fidelity [Nielsen2002]* of the ideal process and the process matrix inferred via maximum likelihood tomography under complete positivity (cp) and trace preservation (tp) constraints (cf. supplementary material of *[Reagor2018]*).

Table 2.2: **Table 2 | Rigetti 8Q two-qubit gate performance**

	f_{m}	t_{CZ}	$\mathsf{F}^{\mathrm{cptp}}_{\mathrm{PT}}$
	MHz	ns	
0 - 1	226	195	0.92
1 - 2	153	198	0.91
2 - 3	138	132	0.82
3 - 4	163	160	0.87
4 - 5	168	163	0.67
5 - 6	107	186	0.93
6 - 7	123	162	0.93
7 - 0	298	118	0.91

2.7.3 Acorn QPU Properties – CURRENTLY UNAVAILABLE

This quantum processor consists of 20 superconducting transmon qubits with fixed capacitive coupling in the planar lattice design shown in Fig. 3.

The resonant frequencies of qubits 0–4 and 10–14 are tunable while qubits 5–9 and 15–19 are fixed. The former have two Josephson junctions in an asymmetric SQUID geometry to provide roughly 1 GHz of frequency tunability, and flux-insensitive “sweet spots” near

$\omega_{01}^{\max}/2\pi \approx 4.5\,\mathrm{GHz}$

and

$\omega_{01}^{\min}/2\pi \approx 3.0\,\mathrm{GHz}$.

These tunable devices are coupled to bias lines for AC and DC flux delivery. Each qubit is capacitively coupled to a quasi-lumped element resonator for dispersive readout of the qubit state. Single-qubit control is effected by applying microwave drives at the resonator ports. Two-qubit gates are activated via RF drives on the flux bias lines.

Due to a fabrication defect, qubit 3 is not tunable, which prohibits operation of the two-qubit parametric gate described below between qubit 3 and its neighbors. Consequently, we will treat this as a 19-qubit processor. **However, this chip is currently unavailable.**

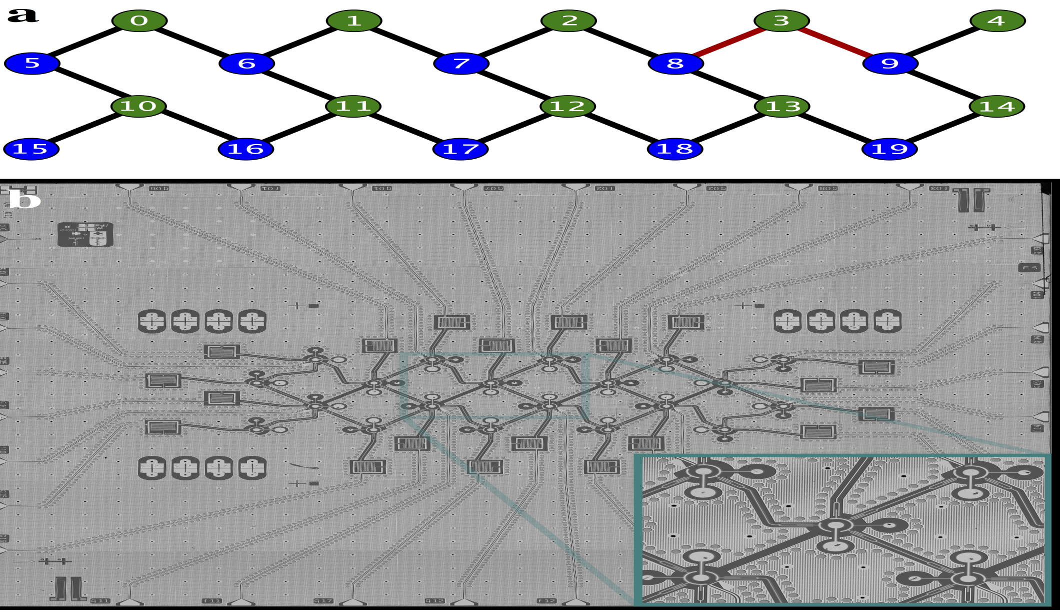


Fig. 2.3: **Figure 3** | **Connectivity of Rigetti 19Q. a,** Chip schematic showing tunable transmons (green circles) capacitively coupled to fixed-frequency transmons (blue circles). **b**, Optical chip image. Note that some couplers have been dropped to produce a lattice with three-fold, rather than four-fold connectivity.

Table 3 summarizes the main performance parameters of Rigetti 19Q.

Table 2.3: **Table 3 | Rigetti 19Q performance**

	$\omega_r^{max}/2\pi$	$\omega_{01}^{max}/2\pi$	$\eta/2\pi$	T_1	T_2^*	F_{1q}	F_{RO}
	MHz	MHz	MHz	μs	μs		
0	5592	4386	-208	**15.2** $\pm$ 2.5	**7.2** $\pm$ 0.7	0.9815	0.938
1	5703	4292	-210	**17.6** $\pm$ 1.7	**7.7** $\pm$ 1.4	0.9907	0.958
2	5599	4221	-142	**18.2** $\pm$ 1.1	**10.8** $\pm$ 0.6	0.9813	0.97
3	5708	3829	-224	**31.0** $\pm$ 2.6	**16.8** $\pm$ 0.8	0.9908	0.886
4	5633	4372	-220	**23.0** $\pm$ 0.5	**5.2** $\pm$ 0.2	0.9887	0.953
5	5178	3690	-224	**22.2** $\pm$ 2.1	**11.1** $\pm$ 1.0	0.9645	0.965
6	5356	3809	-208	**26.8** $\pm$ 2.5	**26.8** $\pm$ 2.5	0.9905	0.84
7	5164	3531	-216	**29.4** $\pm$ 3.8	**13.0** $\pm$ 1.2	0.9916	0.925
8	5367	3707	-208	**24.5** $\pm$ 2.8	**13.8** $\pm$ 0.4	0.9869	0.947
9	5201	3690	-214	**20.8** $\pm$ 6.2	**11.1** $\pm$ 0.7	0.9934	0.927
10	5801	4595	-194	**17.1** $\pm$ 1.2	**10.6** $\pm$ 0.5	0.9916	0.942
11	5511	4275	-204	**16.9** $\pm$ 2.0	**4.9** $\pm$ 1.0	0.9901	0.900
12	5825	4600	-194	**8.2** $\pm$ 0.9	**10.9** $\pm$ 1.4	0.9902	0.942
13	5523	4434	-196	**18.7** $\pm$ 2.0	**12.7** $\pm$ 0.4	0.9933	0.921
14	5848	4552	-204	**13.9** $\pm$ 2.2	**9.4** $\pm$ 0.7	0.9916	0.947
15	5093	3733	-230	**20.8** $\pm$ 3.1	**7.3** $\pm$ 0.4	0.9852	0.970
16	5298	3854	-218	**16.7** $\pm$ 1.2	**7.5** $\pm$ 0.5	0.9906	0.948
17	5097	3574	-226	**24.0** $\pm$ 4.2	**8.4** $\pm$ 0.4	0.9895	0.921
18	5301	3877	-216	**16.9** $\pm$ 2.9	**12.9** $\pm$ 1.3	0.9496	0.930
19	5108	3574	-228	**24.7** $\pm$ 2.8	**9.8** $\pm$ 0.8	0.9942	0.930

Table 4 shows the two-qubit gate performance of Rigetti 19Q.

Table 2.4: **Table 4 | Rigetti 19Q two-qubit gate performance**

	A_0	f_m	t_{CZ}	F_{PT}^{cptp}
	Φ/Φ_0	MHz	ns	
0 - 5	0.27	94.5	168	0.936
0 - 6	0.36	123.9	197	0.889
1 - 6	0.37	137.1	173	0.888
1 - 7	0.59	137.9	179	0.919
2 - 7	0.62	87.4	160	0.817
2 - 8	0.23	55.6	189	0.906
4 - 9	0.43	183.6	122	0.854
5 - 10	0.60	152.9	145	0.870
6 - 11	0.38	142.4	180	0.838
7 - 12	0.60	241.9	214	0.87
8 - 13	0.40	152.0	185	0.881
9 - 14	0.62	130.8	139	0.872
10 - 15	0.53	142.1	154	0.854
10 - 16	0.43	170.3	180	0.838
11 - 16	0.38	160.6	155	0.891
11 - 17	0.29	85.7	207	0.844
12 - 17	0.36	177.1	184	0.876
12 - 18	0.28	113.9	203	0.886
13 - 18	0.24	66.2	152	0.936
13 - 19	0.62	109.6	181	0.921
14 - 19	0.59	188.1	142	0.797

2.8 The Quil Compiler

2.8.1 Expectations for Program Contents

The QPUs have much more limited natural gate sets than the standard gate set offered by pyQuil: the gate operators are constrained to lie in `RZ(θ)`, `RX(kπ/2)`, and `CZ`; and the gates are required to act on physically available hardware (for single-qubit gates, this means acting only on live qubits, and for qubit-pair gates, this means acting on neighboring qubits).

To ameliorate these limitations, the QPU execution stack contains an optimizing compiler that translates arbitrary ProtoQuil to QPU-executable Quil. The compiler is designed to avoid changing even non-semantic details of input Quil code, except to make it shorter when possible. For instance, it will not readdress Quil code that is already appropriately addressed to physically realizable hardware objects on the QPU. The following figure illustrates the layout and addressing of the Rigetti 8Q-Agave QPU.

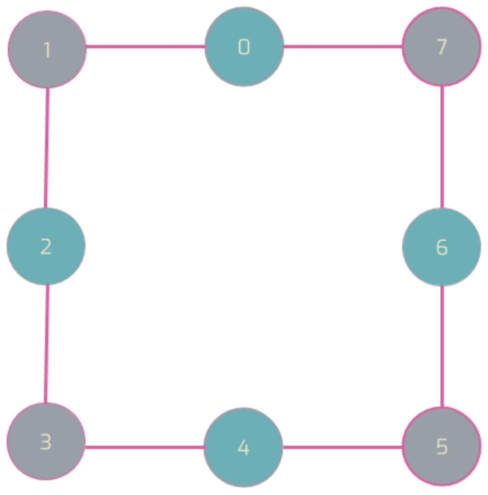


Fig. 2.4: Qubit adjacency schematic for the Rigetti 8Q-Agave QPU.

2.8.2 Interacting with the Compiler

The `QVMConnection` and `QPUConnection` classes in pyQuil offer indirect support for interacting with the compiler: they are both capable of submitting jobs to the compiler for preprocessing before the job is forwarded to the execution target. This behavior is disabled by default for the QVM and enabled by default for the QPU. PyQuil also offers the `CompilerConnection` class for direct access to the compiler, which returns compiled Program jobs to the user without executing them. `CompilerConnection` can be used to learn about the properties of the program, like gate volume, single qubit gate depth, topological swaps, program fidelity and multiqubit gate depth. In all cases, the user's Forest plan must have compiler access enabled to use these features.

Here's an example of using `CompilerConnection` to compile a program that targets the 8Q-Agave QPU, separately from sending a program to the QPU/QVM.

```python
from pyquil.api import CompilerConnection, get_devices
from pyquil.quil import Pragma, Program
from pyquil.gates import CNOT, H
```

```
devices = get_devices(as_dict=True)
agave = devices['8Q-Agave']
compiler = CompilerConnection(agave)

job_id = compiler.compile_async(Program(H(0), CNOT(0,1), CNOT(1,2)))
job = compiler.wait_for_job(job_id)

print('compiled quil', job.compiled_quil())
print('gate volume', job.gate_volume())
print('gate depth', job.gate_depth())
print('topological swaps', job.topological_swaps())
print('program fidelity', job.program_fidelity())
print('multiqubit gate depth', job.multiqubit_gate_depth())
```

Here's what you should see:

```
PRAGMA EXPECTED_REWIRING "#(0 1 2 3 4 5 6 7)"
RZ(pi/2) 0
RX(pi/2) 0
RZ(-pi/2) 1
RX(pi/2) 1
CZ 1 0
RX(-pi/2) 1
RZ(-pi/2) 2
RX(pi/2) 2
CZ 2 1
RZ(-pi/2) 0
RZ(-pi/2) 1
RX(-pi/2) 2
RZ(pi/2) 2
PRAGMA CURRENT_REWIRING "#(0 1 2 3 4 5 6 7)"
PRAGMA EXPECTED_REWIRING "#(0 1 2 3 4 5 6 7)"
PRAGMA CURRENT_REWIRING "#(0 1 2 3 4 5 6 7)"

gate volume 13
gate depth 7
topological swaps 0
program fidelity 0.8981559276580811
multiqubit gate depth 2
```

The `QVMConnection` and `QPUConnection` objects have their compiler interactions set up in the same way: the `.run` and `.run_and_measure` methods take the optional arguments `needs_compilation` and `isa` that respectively toggle the compilation preprocessing step and provide the compiler with a target instruction set architecture, specified as a pyQuil `ISA` object. The compiler can be bypassed by passing the method parameter `needs_compilation=False`. If the `isa` named argument is not set, then the `default_isa` property on the connection object is used instead. The compiled program can be accessed after a job has been submitted to the QPU by using the `.compiled_quil()` accessor method on the resulting `Job` object instance.

2.8.3 Region-specific compiler features through PRAGMA

The Quil compiler can also be communicated with through `PRAGMA` commands embedded in the Quil program.

Preserved regions

The compiler can be circumvented in user-specified regions. The start of such a region is denoted by `PRAGMA PRESERVE_BLOCK`, and the end is denoted by `PRAGMA END_PRESERVE_BLOCK`. The Quil compiler promises not to modify any instructions contained in such a region.

The following is an example of a program that prepares a Bell state on qubits 0 and 1, then performs a time delay to invite noisy system interaction before measuring the qubits. The time delay region is marked by `PRAGMA PRESERVE_BLOCK` and `PRAGMA END_PRESERVE_BLOCK`; without these delimiters, the compiler will remove the identity gates that serve to provide the time delay. However, the regions outside of the `PRAGMA` region will still be compiled, converting the Bell state preparation to the native gate set.

```
#   prepare a Bell state
H 0
CNOT 0 1
#   wait a while
PRAGMA PRESERVE_BLOCK
I 0
I 1
I 0
I 1
# ...
I 0
I 1
PRAGMA END_PRESERVE_BLOCK
#   and read out the results
MEASURE 0 [0]
MEASURE 1 [1]
```

Parallelizable regions

The compiler can sometimes arrange gate sequences more cleverly if the user gives it hints about sequences of gates that commute. A region containing commuting sequences is bookended by `PRAGMA COMMUTING_BLOCKS` and `PRAGMA END_COMMUTING_BLOCKS`; within such a region, a given commuting sequence is bookended by `PRAGMA BLOCK` and `PRAGMA END_BLOCK`.

The following snippet demonstrates this hinting syntax in a context typical of VQE-type algorithms: after a first stage of performing some state preparation on individual qubits, there is a second stage of "mixing operations" that both re-use qubit resources and mutually commute, followed by a final rotation and measurement. The following program is naturally laid out on a ring with vertices (read either clockwise or counterclockwise) as 0, 1, 2, 3. After scheduling the first round of preparation gates, the compiler will use the hinting to schedule the first and third blocks (which utilize qubit pairs 0-1 and 2-3) before the second and fourth blocks (which utilize qubit pairs 1-2 and 0-3), resulting in a reduction in circuit depth by one half. Without hinting, the compiler will instead execute the blocks in their written order.

```
# Stage one
H 0
H 1
H 2
H 3
# Stage two
PRAGMA COMMUTING_BLOCKS
PRAGMA BLOCK
CNOT 0 1
RZ(0.4) 1
CNOT 0 1
```

```
PRAGMA END_BLOCK
PRAGMA BLOCK
CNOT 1 2
RZ(0.6) 2
CNOT 1 2
PRAGMA END_BLOCK
PRAGMA BLOCK
CNOT 2 3
RZ(0.8) 3
CNOT 2 3
PRAGMA END_BLOCK
PRAGMA BLOCK
CNOT 0 3
RZ(0.9) 3
CNOT 0 3
PRAGMA END_BLOCK
PRAGMA END_COMMUTING_BLOCKS
# Stage three
H 0
H 1
H 2
H 3
MEASURE 0 [0]
MEASURE 1 [1]
MEASURE 2 [2]
MEASURE 3 [3]
```

Rewirings

When a Quil program contains multi-qubit instructions that do not name qubit-qubit links present on a target device, the compiler will rearrange the qubits so that execution becomes possible. In order to help the user understand what rearrangement may have been done, the compiler emits two forms of `PRAGMA`: `PRAGMA EXPECTED_REWIRING` and `PRAGMA CURRENT_REWIRING`. From the perspective of the user, both `PRAGMA` instructions serve the same purpose: `PRAGMA ..._REWIRING "#(n0 n1 ... nk)"` indicates that the logical qubit labeled `j` in the program has been assigned to lie on the physical qubit labeled `nj` on the device. This is strictly for human-readability: user-supplied instructions of the form `PRAGMA [EXPECTED|CURRENT]_REWIRING` are discarded and have no effect.

In addition, you have some control over how the compiler constructs its rewiring. If you include a `PRAGMA INITIAL_REWIRING "[NAIVE|RANDOM|PARTIAL|GREEDY]"` instruction before any non-pragmas, the compiler will alter its rewiring behavior.

- *PARTIAL* (default): the compiler will start with nothing assigned to each physical qubit. Then, it will fill in the logical-to-physical mapping as it encounters new qubits in the program, making its best guess for where they should be placed.
- *NAIVE*: the compiler will start with an identity mapping as the initial rewiring
- *RANDOM*: the compiler will start with a random permutation
- *GREEDY*: the compiler will make a guess for the initial rewiring based on a quick initial scan of the entire program.

2.8.4 Common Error Messages

The compiler itself is subject to some limitations, and some of the more commonly observed errors follow:

- `! ! ! Error: Failed to select a SWAP instruction. Perhaps the qubit graph is disconnected?` This error indicates a readdressing failure: some non-native Quil could not be reassigned to lie on native devices. Two common reasons for this failure are:
  - It is possible for the readdressing problem to be too difficult for the compiler to sort out, causing deadlock.
  - If a qubit-qubit gate is requested to act on two qubit resources that lie on disconnected regions of the qubit graph, the addresser will fail.
- `! ! ! Error: Matrices do not lie in the same projective class.` The compiler attempted to decompose an operator as native Quil instructions, and the resulting instructions do not match the original operator. This can happen when the original operator is not a unitary matrix, and could indicate an invalid `DEFGATE` block.
- `! ! ! Error: Addresser loop only supports pure quantum instructions.` The compiler inspected an instruction that it does not understand. The most common cause of this error is the inclusion of classical control in a program submission, which is legal Quil but falls outside of the domain of ProtoQuil.

2.9 Using Qubit Placeholders

In PyQuil, we typically use integers to identify qubits

```
from pyquil.quil import Program
from pyquil.gates import CNOT, H
print(Program(H(0), CNOT(0, 1)))
```

```
H 0
CNOT 0 1
```

However, when running on real, near-term QPUs we care about what particular physical qubits our program will run on. In fact, we may want to run the same program on an assortment of different qubits. This is where using `QubitPlaceholders` comes in.

```
from pyquil.quilatom import QubitPlaceholder
q0 = QubitPlaceholder()
q1 = QubitPlaceholder()
prog = Program(H(q0), CNOT(q0, q1))
print(prog)
```

```
H {q4402789176}
CNOT {q4402789176} {q4402789120}
```

If you try to use this program directly, it will not work

```
print(prog.out())
```

```
---------------------------------------------------------------------------

RuntimeError                              Traceback (most recent call last)

<ipython-input-3-da474d3af403> in <module>()
----> 1 print(prog.out())

...
```

```
pyquil/pyquil/quilatom.py in out(self)
     53 class QubitPlaceholder(QuilAtom):
     54     def out(self):
---> 55         raise RuntimeError("Qubit {} has not been assigned an index".
↪format(self))
     56
     57     def __str__(self):


RuntimeError: Qubit q4402789176 has not been assigned an index
```

Instead, you must explicitly map the placeholders to physical qubits. By default, the function `address_qubits` will address qubits from 0 to N.

```
from pyquil.quil import address_qubits
print(address_qubits(prog))
```

```
H 0
CNOT 0 1
```

The real power comes into play when you provide an explicit mapping

```
print(address_qubits(prog, qubit_mapping={
    q0: 14,
    q1: 19,
}))
```

```
H 14
CNOT 14 19
```

2.9.1 Register

Usually, your algorithm will use an assortment of qubits. You can use the convenience function `QubitPlaceholder.register()` to request a list of qubits to build your program.

```
qbyte = QubitPlaceholder.register(8)
prog2 = Program(H(q) for q in qbyte)
print(address_qubits(prog2, {q: i*2 for i, q in enumerate(qbyte)}))
```

```
H 0
H 2
H 4
H 6
H 8
H 10
H 12
H 14
```

2.10 Noise and Quantum Computation

2.10.1 Modeling Noisy Quantum Gates

Pure States vs. Mixed States

Errors in quantum computing can introduce classical uncertainty in what the underlying state is. When this happens we sometimes need to consider not only wavefunctions but also probabilistic sums of wavefunctions when we are uncertain as to which one we have. For example, if we think that an X gate was accidentally applied to a qubit with a 50-50 chance then we would say that there is a 50% chance we have the 0 state and a 50% chance that we have a 1 state. This is called an "impure" or "mixed"state in that it isn't just a wavefunction (which is pure) but instead a distribution over wavefunctions. We describe this with something called a density matrix, which is generally an operator. Pure states have very simple density matrices that we can write as an outer product of a ket vector ψ with its own bra version $\psi = \psi^\dagger$. For a pure state the density matrix is simply

$$\rho_\psi = \psi\psi.$$

The expectation value of an operator for a mixed state is given by

$$\langle X \rangle_\rho = X\rho$$

where $\cdot$ is the trace of an operator, which is the sum of its diagonal elements which is independent of choice of basis. Pure state density matrices satisfy

$$\rho \text{ is pure } \Leftrightarrow \rho^2 = \rho$$

which you can easily verify for ρ_ψ assuming that the state is normalized. If we want to describe a situation with classical uncertainty between states ρ_1 and ρ_2, then we can take their weighted sum

$$\rho = p\rho_1 + (1-p)\rho_2$$

where $p \in [0, 1]$ gives the classical probability that the state is ρ_1.

Note that classical uncertainty in the wavefunction is markedly different from superpositions. We can represent superpositions using wavefunctions, but use density matrices to describe distributions over wavefunctions. You can read more about density matrices here *[DensityMatrix]*.

Quantum Gate Errors

For a quantum gate given by its unitary operator U, a "quantum gate error" describes the scenario in which the actually induces transformation deviates from $\psi \mapsto U\psi$. There are two basic types of quantum gate errors:

1. **coherent errors** are those that preserve the purity of the input state, i.e., instead of the above mapping we carry out a perturbed, but unitary operation $\psi \mapsto \tilde{U}\psi$, where $\tilde{U} \neq U$.
2. **incoherent errors** are those that do not preserve the purity of the input state, in this case we must actually represent the evolution in terms of density matrices. The state $\rho := \psi\psi$ is then mapped as

$$\rho \mapsto \sum_{j=1}^{n} K_j \rho K_j^\dagger,$$

   where the operators $\{K_1, K_2, \dots, K_m\}$ are called Kraus operators and must obey $\sum_{j=1}^{m} K_j^\dagger K_j = I$ to conserve the trace of ρ. Maps expressed in the above form are called Kraus maps. It can be shown that every physical map on a finite dimensional quantum system can be represented as a Kraus map, though this representation is not generally unique. You can find more information about quantum operations here

In a way, coherent errors are *in principle* amendable by more precisely calibrated control. Incoherent errors are more tricky.

Why Do Incoherent Errors Happen?

When a quantum system (e.g., the qubits on a quantum processor) is not perfectly isolated from its environment it generally co-evolves with the degrees of freedom it couples to. The implication is that while the total time evolution of system and environment can be assumed to be unitary, restriction to the system state generally is not.

Let's throw some math at this for clarity: Let our total Hilbert space be given by the tensor product of system and environment Hilbert spaces: $\mathcal{H} = \mathcal{H}_S \otimes \mathcal{H}_E$. Our system "not being perfectly isolated" must be translated to the statement that the global Hamiltonian contains a contribution that couples the system and environment:

$$H = H_S \otimes I + I \otimes H_E + V$$

where V non-trivally acts on both the system and the environment. Consequently, even if we started in an initial state that factorized over system and environment $\psi_{S,0} \otimes \psi_{E,0}$ if everything evolves by the Schrödinger equation

$$\psi_t = e^{-i\frac{Ht}{\hbar}} \left(\psi_{S,0} \otimes \psi_{E,0}\right)$$

the final state will generally not admit such a factorization.

A Toy Model

In this (somewhat technical) section we show how environment interaction can corrupt an identity gate and derive its Kraus map. For simplicity, let us assume that we are in a reference frame in which both the system and environment Hamiltonian's vanish $H_S = 0, H_E = 0$ and where the cross-coupling is small even when multiplied by the duration of the time evolution $\|\frac{tV}{\hbar}\|^2 \sim \epsilon \ll 1$ (any operator norm $\|\cdot\|$ will do here). Let us further assume that $V = \sqrt{\epsilon} V_S \otimes V_E$ (the more general case is given by a sum of such terms) and that the initial environment state satisfies $\psi_{E,0} V_E \psi_{E,0} = 0$. This turns out to be a very reasonable assumption in practice but a more thorough discussion exceeds our scope.

Then the joint system + environment state $\rho = \rho_{S,0} \otimes \rho_{E,0}$ (now written as a density matrix) evolves as

$$\rho \mapsto \rho' := e^{-i\frac{Vt}{\hbar}} \rho e^{+i\frac{Vt}{\hbar}}$$

Using the Baker-Campbell-Hausdorff theorem we can expand this to second order in ϵ

$$\rho' = \rho - \frac{it}{\hbar}[V, \rho] - \frac{t^2}{2\hbar^2}[V, [V, \rho]] + O(\epsilon^{3/2})$$

We can insert the initially factorizable state $\rho = \rho_{S,0} \otimes \rho_{E,0}$ and trace over the environmental degrees of freedom to obtain

$$\rho'_S := \rho'_E = \rho_{S,0} \underbrace{\rho_{E,0}}_{1} - \frac{i\sqrt{\epsilon}t}{\hbar} \underbrace{\left[V_S \rho_{S,0} \underbrace{V_E \rho_{E,0}}_{\psi_{E,0} V_E \psi_{E,0} = 0} - \rho_{S,0} V_S \underbrace{\rho_{E,0} V_E}_{\psi_{E,0} V_E \psi_{E,0} = 0} \right]}_{0} \tag{2.1}$$

$$- \frac{\epsilon t^2}{2\hbar^2} \left[V_S^2 \rho_{S,0} V_E^2 \rho_{E,0} + \rho_{S,0} V_S^2 \rho_{E,0} V_E^2 - 2 V_S \rho_{S,0} V_S V_E \rho_{E,0} V_E \right] \tag{2.2}$$

$$= \rho_{S,0} - \frac{\gamma}{2} \left[V_S^2 \rho_{S,0} + \rho_{S,0} V_S^2 - 2 V_S \rho_{S,0} V_S \right] \tag{2.3}$$

where the coefficient in front of the second part is by our initial assumption very small $\gamma := \frac{\epsilon t^2}{2\hbar^2} V_E^2 \rho_{E,0} \ll 1$. This evolution happens to be approximately equal to a Kraus map with operators $K_1 := I - \frac{\gamma}{2} V_S^2, K_2 := \sqrt{\gamma} V_S$:

$$\rho_S \to \rho'_S = K_1 \rho K_1^\dagger + K_2 \rho K_2^\dagger = \rho - \frac{\gamma}{2} \left[V_S^2 \rho + \rho V_S^2 \right] + \gamma V_S \rho_S V_S + O(\gamma^2) \tag{2.4}$$

This agrees to $O(\epsilon^{3/2})$ with the result of our derivation above. This type of derivation can be extended to many other cases with little complication and a very similar argument is used to derive the Lindblad master equation.

2.10.2 Noisy Gates on the Rigetti QVM

As of today, users of our Forest API can annotate their QUIL programs by certain pragma statements that inform the QVM that a particular gate on specific target qubits should be replaced by an imperfect realization given by a Kraus map.

The QVM propagates **pure states** — so how does it simulate noisy gates? It does so by yielding the correct outcomes **in the average over many executions of the QUIL program**: When the noisy version of a gate should be applied the QVM makes a random choice which Kraus operator is applied to the current state with a probability that ensures that the average over many executions is equivalent to the Kraus map. In particular, a particular Kraus operator K_j is applied to ψ_S

$$\psi'_S = \frac{1}{\sqrt{p_j}} K_j \psi_S$$

with probability $p_j := \psi_S K_j^\dagger K_j \psi_S$. In the average over many execution $N \gg 1$ we therefore find that

$$\overline{\rho'_S} = \frac{1}{N}\sum_{n=1}^{N} \psi'_{nS}\psi'_{nS} \tag{2.5}$$

$$= \frac{1}{N}\sum_{n=1}^{N} p_{j_n}^{-1} K_{j_n} \psi'_S \psi'_S K_{j_n}^\dagger \tag{2.6}$$

where j_n is the chosen Kraus operator label in the n-th trial. This is clearly a Kraus map itself! And we can group identical terms and rewrite it as

$$\overline{\rho'_S} = \sum_{\ell=1}^{n} \frac{N_\ell}{N} p_\ell^{-1} K_\ell \psi'_S \psi'_S K_\ell^\dagger \tag{2.7}$$

where N_ℓ is the number of times that Kraus operator label ℓ was selected. For large enough N we know that $N_\ell \approx N p_\ell$ and therefore

$$\overline{\rho'_S} \approx \sum_{\ell=1}^{n} K_\ell \psi'_S \psi'_S K_\ell^\dagger \tag{2.8}$$

which proves our claim. **The consequence is that noisy gate simulations must generally be repeated many times to obtain representative results**.

Getting Started

1. Come up with a good model for your noise. We will provide some examples below and may add more such examples to our public repositories over time. Alternatively, you can characterize the gate under consideration using Quantum Process Tomography or Gate Set Tomography and use the resulting process matrices to obtain a very accurate noise model for a particular QPU.
2. Define your Kraus operators as a list of numpy arrays `kraus_ops = [K1, K2, ..., Km]`.
3. For your QUIL program `p`, call:

```
p.define_noisy_gate("MY_NOISY_GATE", [q1, q2], kraus_ops)
```

   where you should replace `MY_NOISY_GATE` with the gate of interest and `q1, q2` the indices of the qubits.

Scroll down for some examples!

```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import binom
import matplotlib.colors as colors
%matplotlib inline
```

```
from pyquil.quil import Program, MEASURE
from pyquil.api.qvm import QVMConnection
from pyquil.job_results import wait_for_job
from pyquil.gates import CZ, H, I, X
from scipy.linalg import expm
```

```
cxn = QVMConnection()
```

Example 1: Amplitude Damping

Amplitude damping channels are imperfect identity maps with Kraus operators

$$K_1 = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-p} \end{pmatrix}$$
$$K_2 = \begin{pmatrix} 0 & \sqrt{p} \\ 0 & 0 \end{pmatrix}$$

where p is the probability that a qubit in the 1 state decays to the 0 state.

```
def damping_channel(damp_prob=.1):
    """
    Generate the Kraus operators corresponding to an amplitude damping
    noise channel.

    :params float damp_prob: The one-step damping probability.
    :return: A list [k1, k2] of the Kraus operators that parametrize the map.
    :rtype: list
    """
    damping_op = np.sqrt(damp_prob) * np.array([[0, 1],
                                                [0, 0]])

    residual_kraus = np.diag([1, np.sqrt(1-damp_prob)])
    return [residual_kraus, damping_op]

def append_kraus_to_gate(kraus_ops, g):
    """
    Follow a gate `g` by a Kraus map described by `kraus_ops`.

    :param list kraus_ops: The Kraus operators.
    :param numpy.ndarray g: The unitary gate.
    :return: A list of transformed Kraus operators.
    """
    return [kj.dot(g) for kj in kraus_ops]


def append_damping_to_gate(gate, damp_prob=.1):
    """
    Generate the Kraus operators corresponding to a given unitary
    single qubit gate followed by an amplitude damping noise channel.

    :params np.ndarray|list gate: The 2x2 unitary gate matrix.
    :params float damp_prob: The one-step damping probability.
    :return: A list [k1, k2] of the Kraus operators that parametrize the map.
    :rtype: list
    """
    return append_kraus_to_gate(damping_channel(damp_prob), gate)
```

```
%%time

# single step damping probability
damping_per_I = 0.02

# number of program executions
trials = 200

results = []
outcomes = []
lengths = np.arange(0, 201, 10, dtype=int)
for jj, num_I in enumerate(lengths):

    print("{}/{}, ".format(jj, len(lengths)), end="")


    p = Program(X(0))
    # want increasing number of I-gates
    p.inst([I(0) for _ in range(num_I)])
    p.inst(MEASURE(0, [0]))

    # overload identity I on qc 0
    p.define_noisy_gate("I", [0], append_damping_to_gate(np.eye(2), damping_per_I))
    cxn.random_seed = int(num_I)
    res = cxn.run(p, [0], trials=trials)
    results.append([np.mean(res), np.std(res) / np.sqrt(trials)])

results = np.array(results)
```

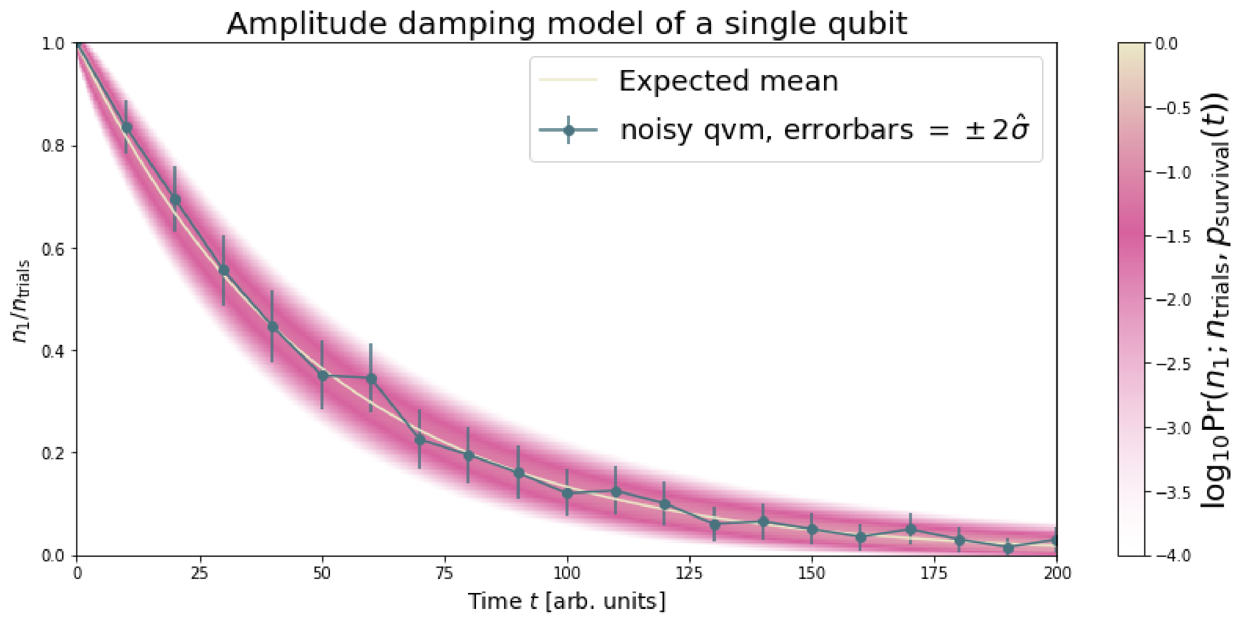
```
0/21, 1/21, 2/21, 3/21, 4/21, 5/21, 6/21, 7/21, 8/21, 9/21, 10/21, 11/21, 12/21, 13/
↪21, 14/21, 15/21, 16/21, 17/21, 18/21, 19/21, 20/21, CPU times: user 138 ms, sys:␣
↪19.2 ms, total: 157 ms
Wall time: 6.4 s
```

```
dense_lengths = np.arange(0, lengths.max()+1, .2)
survival_probs = (1-damping_per_I)**dense_lengths
logpmf = binom.logpmf(np.arange(trials+1)[np.newaxis, :], trials, survival_probs[:,␣
↪np.newaxis])/np.log(10)
```

```
DARK_TEAL = '#48737F'
FUSCHIA = "#D6619E"
BEIGE = '#EAE8C6'
cm = colors.LinearSegmentedColormap.from_list('anglemap', ["white", FUSCHIA, BEIGE],␣
↪N=256, gamma=1.5)
```

```
plt.figure(figsize=(14, 6))
plt.pcolor(dense_lengths, np.arange(trials+1)/trials, logpmf.T, cmap=cm, vmin=-4,␣
↪vmax=logpmf.max())
```

```
plt.plot(dense_lengths, survival_probs, c=BEIGE, label="Expected mean")
plt.errorbar(lengths, results[:,0], yerr=2*results[:,1], c=DARK_TEAL,
             label=r"noisy qvm, errorbars $ = \pm 2\hat{\sigma}$", marker="o")
cb = plt.colorbar()
cb.set_label(r"$\log_{10} \mathrm{Pr}(n_1; n_{\rm trials}, p_{\rm survival}(t))$",␣
↪size=20)

plt.title("Amplitude damping model of a single qubit", size=20)
plt.xlabel(r"Time $t$ [arb. units]", size=14)
plt.ylabel(r"$n_1/n_{\rm trials}$", size=14)
plt.legend(loc="best", fontsize=18)
plt.xlim(*lengths[[0, -1]])
plt.ylim(0, 1)
```



Example 2: Dephased CZ-gate

Dephasing is usually characterized through a qubit's T_2 time. For a single qubit the dephasing Kraus operators are

$$K_1(p) = \sqrt{1-p}I_2$$
$$K_2(p) = \sqrt{p}\sigma_Z$$

where $p = 1 - \exp(-T_2/T_{\rm gate})$ is the probability that the qubit is dephased over the time interval of interest, I_2 is the 2×2-identity matrix and σ_Z is the Pauli-Z operator.

For two qubits, we must construct a Kraus map that has *four* different outcomes:

1. No dephasing
2. Qubit 1 dephases
3. Qubit 2 dephases
4. Both dephase

The Kraus operators for this are given by

$$K'_1(p,q) = K_1(p) \otimes K_1(q) \tag{2.9}$$
$$K'_2(p,q) = K_2(p) \otimes K_1(q) \tag{2.10}$$
$$K'_3(p,q) = K_1(p) \otimes K_2(q) \tag{2.11}$$
$$K'_4(p,q) = K_2(p) \otimes K_2(q) \tag{2.12}$$

where we assumed a dephasing probability p for the first qubit and q for the second.

Dephasing is a *diagonal* error channel and the CZ gate is also diagonal, therefore we can get the combined map of dephasing and the CZ gate simply by composing $U_{\rm CZ}$ the unitary representation of CZ with each Kraus operator

$$K^{\rm CZ}_1(p,q) = K_1(p) \otimes K_1(q) U_{\rm CZ} \tag{2.13}$$
$$K^{\rm CZ}_2(p,q) = K_2(p) \otimes K_1(q) U_{\rm CZ} \tag{2.14}$$
$$K^{\rm CZ}_3(p,q) = K_1(p) \otimes K_2(q) U_{\rm CZ} \tag{2.15}$$
$$K^{\rm CZ}_4(p,q) = K_2(p) \otimes K_2(q) U_{\rm CZ} \tag{2.16}$$

Note that this is not always accurate, because a CZ gate is often achieved through non-diagonal interaction Hamiltonians! However, for sufficiently small dephasing probabilities it should always provide a good starting point.

```
def dephasing_kraus_map(p=.1):
    """
    Generate the Kraus operators corresponding to a dephasing channel.

    :params float p: The one-step dephasing probability.
    :return: A list [k1, k2] of the Kraus operators that parametrize the map.
    :rtype: list
    """
    return [np.sqrt(1-p)*np.eye(2), np.sqrt(p)*np.diag([1, -1])]

def tensor_kraus_maps(k1, k2):
    """
    Generate the Kraus map corresponding to the composition
    of two maps on different qubits.

    :param list k1: The Kraus operators for the first qubit.
    :param list k2: The Kraus operators for the second qubit.
    :return: A list of tensored Kraus operators.
    """
    return [np.kron(k1j, k2l) for k1j in k1 for k2l in k2]


def append_kraus_to_gate(kraus_ops, g):
    """
    Follow a gate `g` by a Kraus map described by `kraus_ops`.

    :param list kraus_ops: The Kraus operators.
    :param numpy.ndarray g: The unitary gate.
    :return: A list of transformed Kraus operators.
    """
    return [kj.dot(g) for kj in kraus_ops]
```

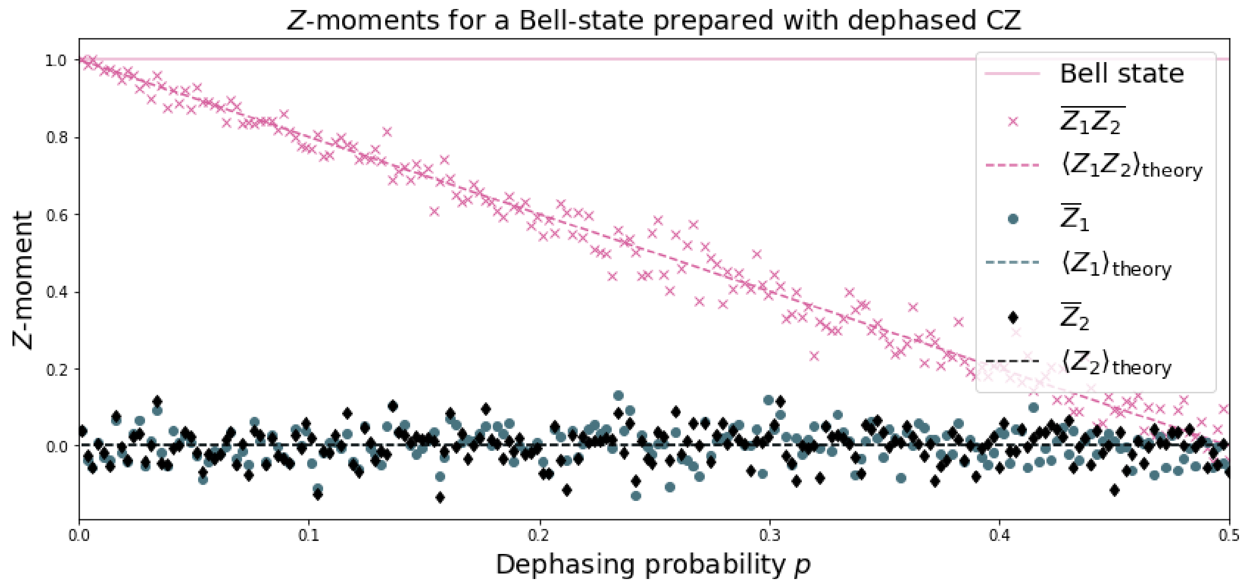
```
%%time
# single step damping probabilities
ps = np.linspace(.001, .5, 200)
```

```
# number of program executions
trials = 500

results = []

for jj, p in enumerate(ps):

    corrupted_CZ = append_kraus_to_gate(
    tensor_kraus_maps(
        dephasing_kraus_map(p),
        dephasing_kraus_map(p)
    ),
    np.diag([1, 1, 1, -1]))


    print("{}/{}, ".format(jj, len(ps)), end="")

    # make Bell-state
    p = Program(H(0), H(1), CZ(0,1), H(1))

    p.inst(MEASURE(0, [0]))
    p.inst(MEASURE(1, [1]))

    # overload identity I on qc 0
    p.define_noisy_gate("CZ", [0, 1], corrupted_CZ)
    cxn.random_seed = jj
    res = cxn.run(p, [0, 1], trials=trials)
    results.append(res)

results = np.array(results)
```

```
0/200, 1/200, 2/200, 3/200, 4/200, 5/200, 6/200, 7/200, 8/200, 9/200, 10/200, 11/200,␣
↪12/200, 13/200, 14/200, 15/200, 16/200, 17/200, 18/200, 19/200, 20/200, 21/200, 22/
↪200, 23/200, 24/200, 25/200, 26/200, 27/200, 28/200, 29/200, 30/200, 31/200, 32/200,
↪ 33/200, 34/200, 35/200, 36/200, 37/200, 38/200, 39/200, 40/200, 41/200, 42/200, 43/
↪200, 44/200, 45/200, 46/200, 47/200, 48/200, 49/200, 50/200, 51/200, 52/200, 53/200,
↪ 54/200, 55/200, 56/200, 57/200, 58/200, 59/200, 60/200, 61/200, 62/200, 63/200, 64/
↪200, 65/200, 66/200, 67/200, 68/200, 69/200, 70/200, 71/200, 72/200, 73/200, 74/200,
↪ 75/200, 76/200, 77/200, 78/200, 79/200, 80/200, 81/200, 82/200, 83/200, 84/200, 85/
↪200, 86/200, 87/200, 88/200, 89/200, 90/200, 91/200, 92/200, 93/200, 94/200, 95/200,
↪ 96/200, 97/200, 98/200, 99/200, 100/200, 101/200, 102/200, 103/200, 104/200, 105/
↪200, 106/200, 107/200, 108/200, 109/200, 110/200, 111/200, 112/200, 113/200, 114/
↪200, 115/200, 116/200, 117/200, 118/200, 119/200, 120/200, 121/200, 122/200, 123/
↪200, 124/200, 125/200, 126/200, 127/200, 128/200, 129/200, 130/200, 131/200, 132/
↪200, 133/200, 134/200, 135/200, 136/200, 137/200, 138/200, 139/200, 140/200, 141/
↪200, 142/200, 143/200, 144/200, 145/200, 146/200, 147/200, 148/200, 149/200, 150/
↪200, 151/200, 152/200, 153/200, 154/200, 155/200, 156/200, 157/200, 158/200, 159/
↪200, 160/200, 161/200, 162/200, 163/200, 164/200, 165/200, 166/200, 167/200, 168/
↪200, 169/200, 170/200, 171/200, 172/200, 173/200, 174/200, 175/200, 176/200, 177/
↪200, 178/200, 179/200, 180/200, 181/200, 182/200, 183/200, 184/200, 185/200, 186/
↪200, 187/200, 188/200, 189/200, 190/200, 191/200, 192/200, 193/200, 194/200, 195/
↪200, 196/200, 197/200, 198/200, 199/200, CPU times: user 1.17 s, sys: 166 ms,␣
↪total: 1.34 s
Wall time: 1min 49s
```

```
Z1s = (2*results[:,:,0]-1.)
Z2s = (2*results[:,:,1]-1.)
Z1Z2s = Z1s * Z2s

Z1m = np.mean(Z1s, axis=1)
Z2m = np.mean(Z2s, axis=1)
Z1Z2m = np.mean(Z1Z2s, axis=1)
```

```
plt.figure(figsize=(14, 6))
plt.axhline(y=1.0, color=FUSCHIA, alpha=.5, label="Bell state")

plt.plot(ps, Z1Z2m, "x", c=FUSCHIA, label=r"$\overline{Z_1 Z_2}$")
plt.plot(ps, 1-2*ps, "--", c=FUSCHIA, label=r"$\langle Z_1 Z_2\rangle_{\rm theory}$")

plt.plot(ps, Z1m, "o", c=DARK_TEAL, label=r"$\overline{Z}_1$")
plt.plot(ps, 0*ps, "--", c=DARK_TEAL, label=r"$\langle Z_1\rangle_{\rm theory}$")

plt.plot(ps, Z2m, "d", c="k", label=r"$\overline{Z}_2$")
plt.plot(ps, 0*ps, "--", c="k", label=r"$\langle Z_2\rangle_{\rm theory}$")

plt.xlabel(r"Dephasing probability $p$", size=18)
plt.ylabel(r"$Z$-moment", size=18)
plt.title(r"$Z$-moments for a Bell-state prepared with dephased CZ", size=18)
plt.xlim(0, .5)
plt.legend(fontsize=18)
```



2.10.3 Adding Decoherence Noise

In this example, we investigate how a program might behave on a near-term device that is subject to *T1*- and *T2*-type noise using the convenience function `pyquil.noise.add_decoherence_noise()`. The same module also contains some other useful functions to define your own types of noise models, e.g., `pyquil.noise.tensor_kraus_maps()` for generating multi-qubit noise processes, `pyquil.noise.combine_kraus_maps()` for describing the succession of two noise processes and `pyquil.noise.append_kraus_to_gate()` which allows appending a noise process to a unitary gate.

```
from pyquil.quil import Program
from pyquil.paulis import PauliSum, PauliTerm, exponentiate, exponential_map,␣
↪trotterize
from pyquil.gates import MEASURE, H, Z, RX, RZ, CZ
import numpy as np
```

The Task

We want to prepare $e^{i\theta XY}$ and measure it in the Z basis.

```
from numpy import pi
theta = pi/3
xy = PauliTerm('X', 0) * PauliTerm('Y', 1)
```

The Idiomatic PyQuil Program

```
prog = exponential_map(xy)(theta)
print(prog)
```

```
H 0
RX(pi/2) 1
CNOT 0 1
RZ(2*pi/3) 1
CNOT 0 1
H 0
RX(-pi/2) 1
```

The Compiled Program

To run on a real device, we must compile each program to the native gate set for the device. The high-level noise model is similarly constrained to use a small, native gate set. In particular, we can use

- I
- $RZ(\theta)$
- $RX(\pm\pi/2)$
- CZ

For simplicity, the compiled program is given below but generally you will want to use a compiler to do this step for you.

```
def get_compiled_prog(theta):
    return Program([
        RZ(-pi/2, 0),
        RX(-pi/2, 0),
        RZ(-pi/2, 1),
        RX( pi/2, 1),
        CZ(1, 0),
        RZ(-pi/2, 1),
        RX(-pi/2, 1),
        RZ(theta, 1),
        RX( pi/2, 1),
        CZ(1, 0),
        RX( pi/2, 0),
        RZ( pi/2, 0),
        RZ(-pi/2, 1),
        RX( pi/2, 1),
        RZ(-pi/2, 1),
    ])
```

Scan Over Noise Parameters

We perform a scan over three levels of noise each at 20 theta points.

Specifically, we investigate T1 values of 1, 3, and 10 us. By default, T2 = T1 / 2, 1 qubit gates take 50 ns, and 2 qubit gates take 150 ns.

In alignment with the device, I and parametric RZ are noiseless while RX and CZ gates experience 1q and 2q gate noise, respectively.

```
from pyquil.api import QVMConnection
cxn = QVMConnection()
```

```
t1s = np.logspace(-6, -5, num=3)
thetas = np.linspace(-pi, pi, num=20)
t1s * 1e6 # us
```

```
array([  1.        ,   3.16227766,  10.        ])
```

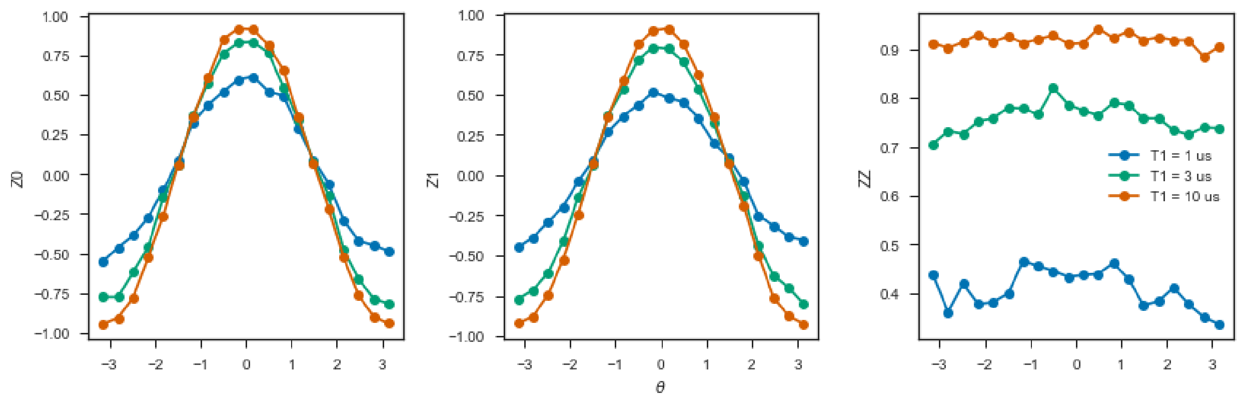
```
from pyquil.noise import add_decoherence_noise
records = []
for theta in thetas:
    for t1 in t1s:
        prog = get_compiled_prog(theta)
        noisy = add_decoherence_noise(prog, T1=t1).inst([
            MEASURE(0, 0),
            MEASURE(1, 1),
        ])
        bitstrings = np.array(cxn.run(noisy, [0,1], 1000))

        # Expectation of Z0 and Z1
        z0, z1 = 1 - 2*np.mean(bitstrings, axis=0)

        # Expectation of ZZ by computing the parity of each pair
        zz = 1 - (np.sum(bitstrings, axis=1) % 2).mean() * 2

        record = {
            'z0': z0,
            'z1': z1,
            'zz': zz,
            'theta': theta,
            't1': t1,
        }
        records += [record]
```

Plot the Results

Note that to run the code below you will need to install the *pandas* and *seaborn* packages.

```
%matplotlib inline
from matplotlib import pyplot as plt
import seaborn as sns
sns.set(style='ticks', palette='colorblind')
```

```
import pandas as pd
df_all = pd.DataFrame(records)
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12,4))

for t1 in t1s:
    df = df_all.query('t1 == @t1')

    ax1.plot(df['theta'], df['z0'], 'o-')
    ax2.plot(df['theta'], df['z1'], 'o-')
    ax3.plot(df['theta'], df['zz'], 'o-', label='T1 = {:.0f} us'.format(t1*1e6))

ax3.legend(loc='best')

ax1.set_ylabel('Z0')
ax2.set_ylabel('Z1')
ax3.set_ylabel('ZZ')
ax2.set_xlabel(r'$\theta$')
fig.tight_layout()
```



2.10.4 Modeling Readout Noise

Qubit-Readout can be corrupted in a variety of ways. The two most relevant error mechanisms on the Rigetti QPU right now are:

1. Transmission line noise that makes a 0-state look like a 1-state or vice versa. We call this **classical readout bit-flip error**. This type of readout noise can be reduced by tailoring optimal readout pulses and using superconducting, quantum limited amplifiers to amplify the readout signal before it is corrupted by classical noise at the higher temperature stages of our cryostats.
2. T1 qubit decay during readout (our readout operations can take more than a µsecond unless they have been specially optimized), which leads to readout signals that initially behave like 1-states but then collapse to something resembling a 0-state. We will call this **T1-readout error**. This type of readout error can be reduced by achieving

shorter readout pulses relative to the T1 time, i.e., one can try to reduce the readout pulse length, or increase the T1 time or both.

Qubit Measurements

This section provides the necessary theoretical foundation for accurately modeling noisy quantum measurements on superconducting quantum processors. It relies on some of the abstractions (density matrices, Kraus maps) introduced in our notebook on gate noise models.

The most general type of measurement performed on a single qubit at a single time can be characterized by some set $\mathcal{O}$ of measurement outcomes, e.g., in the simplest case $\mathcal{O} = \{0, 1\}$, and some unnormalized quantum channels (see notebook on gate noise models) that encapsulate 1. the probability of that outcome 2. how the qubit state is affected conditional on the measurement outcome.

Here the *outcome* is understood as classical information that has been extracted from the quantum system.

Projective, Ideal Measurement

The simplest case that is usually taught in introductory quantum mechanics and quantum information courses are Born's rule and the projection postulate which state that there exist a complete set of orthogonal projection operators

$$P_{\mathcal{O}} := \{\Pi_x \text{ Projector } \mid x \in \mathcal{O}\},$$

i.e., one for each measurement outcome. Any projection operator must satisfy $\Pi_x^\dagger = \Pi_x = \Pi_x^2$ and for an *orthogonal* set of projectors any two members satisfy

$$\Pi_x \Pi_y = \delta_{xy} \Pi_x = \begin{cases} 0 & \text{if } x \neq y \\ \Pi_x & \text{if } x = y \end{cases}$$

and for a *complete* set we additionally demand that $\sum_{x\in\mathcal{O}} \Pi_x = 1$. Following our introduction to gate noise, we write quantum states as density matrices as this is more general and in closer correspondence with classical probability theory.

With these the probability of outcome x is given by $p(x) = \Pi_x \rho \Pi_x = \Pi_x^2 \rho = \Pi_x \rho$ and the post measurement state is

$$\rho_x = \frac{1}{p(x)} \Pi_x \rho \Pi_x,$$

which is the projection postulate applied to mixed states.

If we were a sloppy quantum programmer and accidentally erased the measurement outcome then our best guess for the post measurement state would be given by something that looks an awful lot like a Kraus map:

$$\rho_{\text{post measurement}} = \sum_{x\in\mathcal{O}} p(x)\rho_x = \sum_{x\in\mathcal{O}} \Pi_x \rho \Pi_x.$$

The completeness of the projector set ensures that the trace of the post measurement is still 1 and the Kraus map form of this expression ensures that $\rho_{\text{post measurement}}$ is a positive (semi-)definite operator.

Classical Readout Bit-Flip Error

Consider now the ideal measurement as above, but where the outcome x is transmitted across a noisy classical channel that produces a final outcome $x' \in \mathcal{O}' = \{0', 1'\}$ according to some conditional probabilities $p(x'|x)$ that can be recorded in the *assignment probability matrix*

$$P_{x'|x} = \begin{pmatrix} p(0|0) & p(0|1) \\ p(1|0) & p(1|1) \end{pmatrix}$$

Note that this matrix has only two independent parameters as each column must be a valid probability distribution, i.e. all elements are non-negative and each column sums to 1.

This matrix allows us to obtain the probabilities $\mathbf{p}' := (p(x'=0), p(x'=1))^T$ from the original outcome probabilities $\mathbf{p} := (p(x=0), p(x=1))^T$ via $\mathbf{p}' = P_{x'|x}\mathbf{p}$. The difference relative to the ideal case above is that now an outcome $x' = 0$ does not necessarily imply that the post measurement state is truly $\Pi_0 \rho \Pi_0 / p(x=0)$. Instead, the post measurement state given a noisy outcome x' must be

$$\begin{aligned}\rho_{x'} &= \sum_{x\in\mathcal{O}} p(x|x')\rho_x \\ &= \sum_{x\in\mathcal{O}} p(x'|x)\frac{p(x)}{p(x')}\rho_x \\ &= \frac{1}{p(x')}\sum_{x\in\mathcal{O}} p(x'|x)\Pi_x\rho\Pi_x\end{aligned}$$

where

$$\begin{aligned}p(x') &= \sum_{x\in\mathcal{O}} p(x'|x)p(x) \\ &= \sum_{x\in\mathcal{O}} p(x'|x)\Pi_x\rho\Pi_x \\ &= \rho\sum_{x\in\mathcal{O}} p(x'|x)\Pi_x \\ &= \rho E'_x.\end{aligned}$$

where we have exploited the cyclical property of the trace $ABC = BCA$ and the projection property $\Pi_x^2 = \Pi_x$. This has allowed us to derive the noisy outcome probabilities from a set of positive operators

$$E_{x'} := \sum_{x\in\mathcal{O}} p(x'|x)\Pi_x \geq 0$$

that must sum to 1:

$$\sum_{x'\in\mathcal{O}'} E'_x = \sum_{x\in\mathcal{O}} \underbrace{\left[\sum_{x'\in\mathcal{O}'} p(x'|x)\right]}_{1}\Pi_x = \sum_{x\in\mathcal{O}}\Pi_x = 1.$$

The above result is a type of generalized **Bayes' theorem** that is extremely useful for this type of (slightly) generalized measurement and the family of operators $\{E_{x'}|x' \in \mathcal{O}'\}$ whose expectations give the probabilities is called a **positive operator valued measure** (POVM). These operators are not generally orthogonal nor valid projection operators but they naturally arise in this scenario. This is not yet the most general type of measurement, but it will get us pretty far.

How to Model T_1 Error

T1 type errors fall outside our framework so far as they involve a scenario in which the *quantum state itself* is corrupted during the measurement process in a way that potentially erases the pre-measurement information as opposed to a loss of purely classical information. The most appropriate framework for describing this is given by that of measurement instruments, but for the practical purpose of arriving at a relatively simple description, we propose describing this by a T1 damping Kraus map followed by the noisy readout process as described above.

Further Reading

Chapter 3 of John Preskill's lecture notes http://www.theory.caltech.edu/people/preskill/ph229/notes/chap3.pdf

2.10.5 Working with Readout Noise

1. Come up with a good guess for your readout noise parameters $p(0|0)$ and $p(1|1)$, the off-diagonals then follow from the normalization of $P_{x'|x}$. If your assignment fidelity F is given, and you assume that the classical bit flip noise is roughly symmetric, then a good approximation is to set $p(0|0) = p(1|1) = F$.
2. For your QUIL program `p`, and a qubit index `q` call:

   ```
   p.define_noisy_readout(q, p00, p11)
   ```

   where you should replace `p00` and `p11` with the assumed probabilities.

Scroll down for some examples!

```
from __future__ import print_function, division
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

from pyquil.quil import Program, MEASURE, Pragma
from pyquil.api.qvm import QVMConnection
from pyquil.gates import I, X, RX, H, CNOT
from pyquil.noise import (estimate_bitstring_probs, correct_bitstring_probs,
                          bitstring_probs_to_z_moments, estimate_assignment_probs)

DARK_TEAL = '#48737F'
FUSCHIA = '#D6619E'
BEIGE = '#EAE8C6'

cxn = QVMConnection()
```

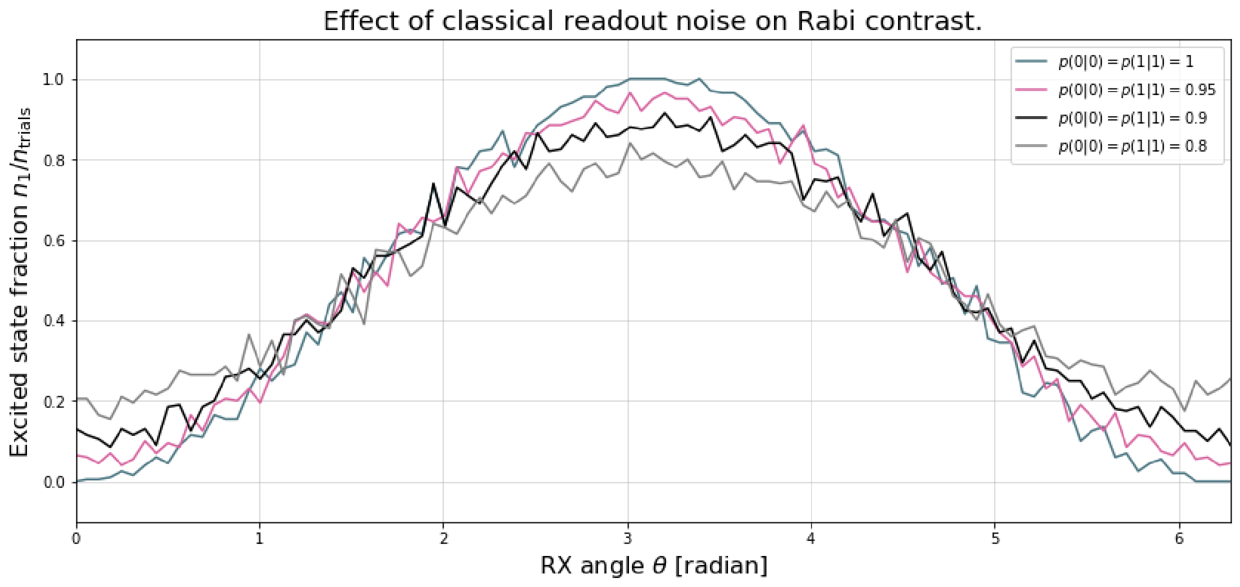
Example 1: Rabi Sequence with Noisy Readout

```
%%time

# number of angles
num_theta = 101

# number of program executions
trials = 200

thetas = np.linspace(0, 2*np.pi, num_theta)

p00s = [1., 0.95, 0.9, 0.8]

results_rabi = np.zeros((num_theta, len(p00s)))

for jj, theta in enumerate(thetas):
    for kk, p00 in enumerate(p00s):
        cxn.random_seed = hash((jj, kk))
        p = Program(RX(theta)(0))
        # assume symmetric noise p11 = p00
        p.define_noisy_readout(0, p00=p00, p11=p00)
        p.measure(0, 0)
        res = cxn.run(p, [0], trials=trials)
        results_rabi[jj, kk] = np.sum(res)
```

```
CPU times: user 1.2 s, sys: 73.6 ms, total: 1.27 s
Wall time: 3.97 s
```

```
plt.figure(figsize=(14, 6))
for jj, (p00, c) in enumerate(zip(p00s, [DARK_TEAL, FUSCHIA, "k", "gray"])):
    plt.plot(thetas, results_rabi[:, jj]/trials, c=c, label=r"$p(0|0)=p(1|1)={:g}$".
↪format(p00))
plt.legend(loc="best")
plt.xlim(*thetas[[0,-1]])
plt.ylim(-.1, 1.1)
plt.grid(alpha=.5)
plt.xlabel(r"RX angle $\theta$ [radian]", size=16)
plt.ylabel(r"Excited state fraction $n_1/n_{\rm trials}$", size=16)
plt.title("Effect of classical readout noise on Rabi contrast.", size=18)
```

```
<matplotlib.text.Text at 0x104314250>
```



Example 2: Estimate the Assignment Probabilities

Here we will estimate $P_{x'|x}$ ourselves! You can run some simple experiments to estimate the assignment probability matrix directly from a QPU.

On a perfect quantum computer

```
estimate_assignment_probs(0, 1000, cxn, Program())
```

```
array([[ 1.,  0.],
       [ 0.,  1.]])
```

On an imperfect quantum computer

```
cxn.seed = None
header0 = Program().define_noisy_readout(0, .85, .95)
header1 = Program().define_noisy_readout(1, .8, .9)
header2 = Program().define_noisy_readout(2, .9, .85)
```

```
ap0 = estimate_assignment_probs(0, 100000, cxn, header0)
ap1 = estimate_assignment_probs(1, 100000, cxn, header1)
ap2 = estimate_assignment_probs(2, 100000, cxn, header2)
```

```
print(ap0, ap1, ap2, sep="\n")
```

```
[[ 0.84967  0.04941]
 [ 0.15033  0.95059]]
[[ 0.80058  0.09993]
 [ 0.19942  0.90007]]
[[ 0.90048  0.14988]
 [ 0.09952  0.85012]]
```

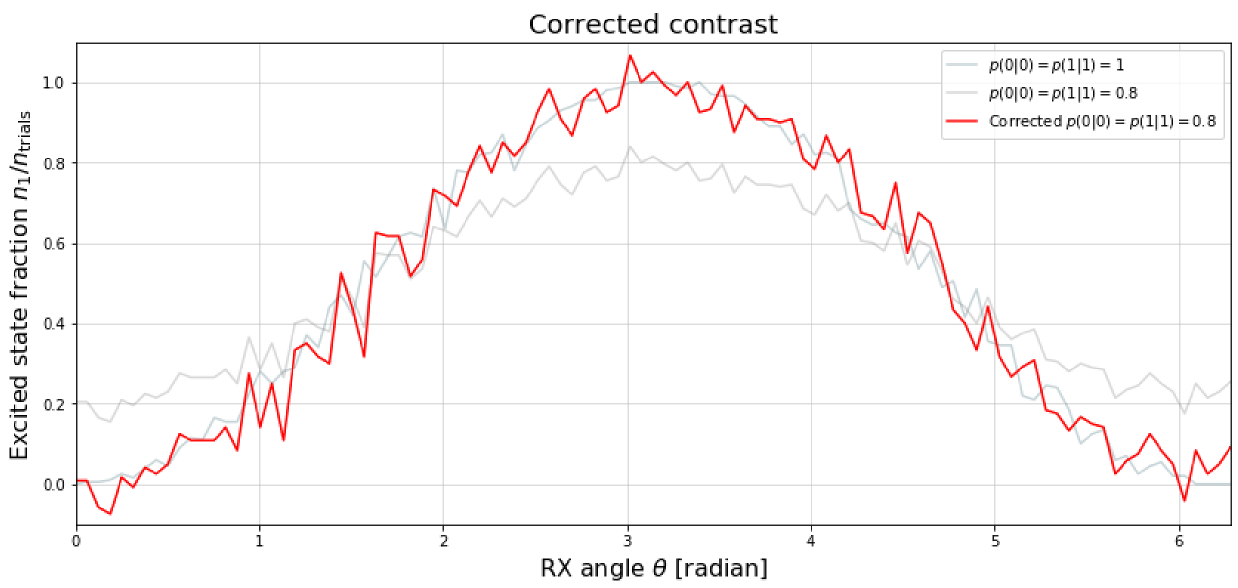
Example 3: Correct for Noisy Readout

3a) Correcting the Rabi Signal from Above

```
ap_last = np.array([[p00s[-1], 1 - p00s[-1]],
                    [1 - p00s[-1], p00s[-1]]])
corrected_last_result = [correct_bitstring_probs([1-p, p], [ap_last])[1] for p in␣
↪results_rabi[:, -1] / trials]
```

```
plt.figure(figsize=(14, 6))
for jj, (p00, c) in enumerate(zip(p00s, [DARK_TEAL, FUSCHIA, "k", "gray"])):
    if jj not in [0, 3]:
        continue
    plt.plot(thetas, results_rabi[:, jj]/trials, c=c, label=r"$p(0|0)=p(1|1)={:g}$".
↪format(p00), alpha=.3)
plt.plot(thetas, corrected_last_result, c="red", label=r"Corrected $p(0|0)=p(1|1)={:g}
↪$".format(p00s[-1]))
plt.legend(loc="best")
plt.xlim(*thetas[[0,-1]])
plt.ylim(-.1, 1.1)
plt.grid(alpha=.5)
plt.xlabel(r"RX angle $\theta$ [radian]", size=16)
plt.ylabel(r"Excited state fraction $n_1/n_{\rm trials}$", size=16)
plt.title("Corrected contrast", size=18)
```

```
<matplotlib.text.Text at 0x1055e7310>
```



We find that the corrected signal is fairly noisy (and sometimes exceeds the allowed interval $[0, 1]$) due to the overall very small number of samples $n = 200$.

3b) Corrupting and Correcting GHZ State Correlations

In this example we will create a GHZ state $\frac{1}{\sqrt{2}}[|000\rangle + |111\rangle]$ and measure its outcome probabilities with and without the above noise model. We will then see how the Pauli-Z moments that indicate the qubit correlations are corrupted (and corrected) using our API.

```
ghz_prog = Program(H(0), CNOT(0, 1), CNOT(1, 2),
                   MEASURE(0, 0), MEASURE(1, 1), MEASURE(2, 2))
print(ghz_prog)
results = cxn.run(ghz_prog, [0, 1, 2], trials=10000)
```

```
H 0
CNOT 0 1
CNOT 1 2
MEASURE 0 [0]
MEASURE 1 [1]
MEASURE 2 [2]
```

```
header = header0 + header1 + header2
noisy_ghz = header + ghz_prog
print(noisy_ghz)
noisy_results = cxn.run(noisy_ghz, [0, 1, 2], trials=10000)
```

```
PRAGMA READOUT-POVM 0 "(0.85 0.050000000000000044 0.15000000000000002 0.95)"
PRAGMA READOUT-POVM 1 "(0.8 0.09999999999999998 0.19999999999999996 0.9)"
PRAGMA READOUT-POVM 2 "(0.9 0.15000000000000002 0.09999999999999998 0.85)"
H 0
CNOT 0 1
CNOT 1 2
MEASURE 0 [0]
MEASURE 1 [1]
MEASURE 2 [2]
```

Uncorrupted probability for $|000\rangle$ and $|111\rangle$

```
probs = estimate_bitstring_probs(results)
probs[0, 0, 0], probs[1, 1, 1]
```

```
(0.50419999999999998, 0.49580000000000002)
```

As expected the outcomes `000` and `111` each have roughly probability $1/2$.

Corrupted probability for $|011\rangle$ and $|100\rangle$

```
noisy_probs = estimate_bitstring_probs(noisy_results)
noisy_probs[0, 0, 0], noisy_probs[1, 1, 1]
```

```
(0.30869999999999997, 0.3644)
```

The noise-corrupted outcome probabilities deviate significantly from their ideal values!

Corrected probability for $|011\rangle$ and $|100\rangle$

```
corrected_probs = correct_bitstring_probs(noisy_probs, [ap0, ap1, ap2])
corrected_probs[0, 0, 0], corrected_probs[1, 1, 1]
```

```
(0.50397601453064977, 0.49866843912900716)
```

The corrected outcome probabilities are much closer to the ideal value.

Estimate $\langle Z_0^j Z_1^k Z_2^\ell \rangle$ for $jkl = 100, 010, 001$ from non-noisy data

We expect these to all be very small

```
zmoments = bitstring_probs_to_z_moments(probs)
zmoments[1, 0, 0], zmoments[0, 1, 0], zmoments[0, 0, 1]
```

```
(0.0083999999999999631, 0.0083999999999999631, 0.0083999999999999631)
```

Estimate $\langle Z_0^j Z_1^k Z_2^\ell \rangle$ for $jkl = 110, 011, 101$ from non-noisy data

We expect these to all be close to 1.

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

```
(1.0, 1.0, 1.0)
```

Estimate $\langle Z_0^j Z_1^k Z_2^\ell \rangle$ for $jkl = 100, 010, 001$ from noise-corrected data

```
zmoments_corr = bitstring_probs_to_z_moments(corrected_probs)
zmoments_corr[1, 0, 0], zmoments_corr[0, 1, 0], zmoments_corr[0, 0, 1]
```

```
(0.0071476770049732075, -0.0078641261685578612, 0.0088462563282706852)
```

Estimate $\langle Z_0^j Z_1^k Z_2^\ell \rangle$ for $jkl = 110, 011, 101$ from noise-corrected data

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

```
(0.99477496902638118, 1.0008376440216553, 1.0149652015905912)
```

Overall the correction can restore the contrast in our multi-qubit observables, though we also see that the correction can lead to slightly non-physical expectations. This effect is reduced the more samples we take.

2.11 Source Code Documentation

2.11.1 pyquil.api

Module for facilitating connections to the QVM / QPU.

class pyquil.api.**QVMConnection**(*device=None*, *sync_endpoint='https://api.rigetti.com'*, *async_endpoint='https://job.rigetti.com/beta'*, *api_key=None*, *user_id=None*, *use_queue=False*, *ping_time=0.1*, *status_time=2*, *gate_noise=None*, *measurement_noise=None*, *random_seed=None*)

Bases: object

Represents a connection to the QVM.

expectation(*prep_prog*, *operator_programs=None*, *needs_compilation=False*, *isa=None*)

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*.

To measure the expectation of a PauliSum, you probably want to do something like this:

```
progs, coefs = hamiltonian.get_programs()
expect_coeffs = np.array(cxn.expectation(prep_program, operator_
↪programs=progs))
return np.real_if_close(np.dot(coefs, expect_coeffs))
```

Parameters

- **prep_prog** (Program) – Quil program for state preparation.
- **operator_programs** (*list*) – A list of Programs, each specifying an operator whose expectation to compute. Default is a list containing only the empty Program.
- **needs_compilation** (*bool*) – If True, preprocesses the job with the compiler.

- **isa** (ISA) – If set, compiles to this target ISA.

Returns Expectation values of the operators.

Return type List[float]

expectation_async (*prep_prog*, *operator_programs=None*, *needs_compilation=False*, *isa=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.

get_job (*job_id*)
Given a job id, return information about the status of the job

Parameters **job_id** (*str*) – job id

Returns Job object with the status and potentially results of the job

Return type *Job*

pauli_expectation (*prep_prog*, *pauli_terms*)
Calculate the expectation value of Pauli operators given a state prepared by prep_program.

If `pauli_terms` is a `PauliSum` then the returned value is a single `float`, otherwise the returned value is a list of `float``s, one for each ``PauliTerm` in the list.

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.
- **pauli_terms** (*Sequence[PauliTerm]|PauliSum*) – A list of PauliTerms or a PauliSum.

Returns If `pauli_terms` is a PauliSum return its expectation value. Otherwise return a list of expectation values.

Return type float|List[float]

ping ()

run (*quil_program*, *classical_addresses=None*, *trials=1*, *needs_compilation=False*, *isa=None*)
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|range*) – A list of addresses.
- **trials** (*int*) – Number of shots to collect.
- **needs_compilation** (*bool*) – If True, preprocesses the job with the compiler.
- **isa** (ISA) – If set, compiles to this target ISA.

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*, *needs_compilation=False*, *isa=None*)
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` | `range`*) – A list of qubits.
- **`trials`** (*`int`*) – Number of shots to collect.
- **`needs_compilation`** (*`bool`*) – If True, preprocesses the job with the compiler.
- **`isa`** (`ISA`) – If set, compiles to this target ISA.

Returns A list of a list of bits.

Return type list

`run_and_measure_async`(*quil_program*, *qubits*, *trials=1*, *needs_compilation=False*, *isa=None*)
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=None*, *trials=1*, *needs_compilation=False*, *isa=None*)
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.

`wait_for_job`(*job_id*, *ping_time=None*, *status_time=None*)
Wait for the results of a job and periodically print status

Parameters

- **`job_id`** – Job id
- **`ping_time`** – How often to poll the server. Defaults to the value specified in the constructor. (0.1 seconds)
- **`status_time`** – How often to print status, set to False to never print status. Defaults to the value specified in the constructor (2 seconds)

Returns Completed Job

`wavefunction`(*quil_program*, *classical_addresses=None*, *needs_compilation=False*, *isa=None*)
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` | `range`*) – An optional list of classical addresses.
- **`needs_compilation`** – If True, preprocesses the job with the compiler.
- **`isa`** – If set, compiles to this target ISA.

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*, *needs_compilation=False*, *isa=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**(*device=None*, *async_endpoint='https://job.rigetti.com/beta'*, *api_key=None*, *user_id=None*, *ping_time=0.1*, *status_time=2*, *device_name=None*)

Bases: `object`

Represents a connection to the QPU (Quantum Processing Unit)

get_job(*job_id*)

Given a job id, return information about the status of the job

Parameters **job_id** (*str*) – job id

Returns Job object with the status and potentially results of the job

Return type *Job*

run(*quil_program*, *classical_addresses=None*, *trials=1*, *needs_compilation=True*, *isa=None*)

Run a pyQuil program on the QPU and return the values stored in the classical registers designated by the classical_addresses parameter. The program is repeated according to the number of trials provided to the run method. This functionality is in beta.

It is important to note that our QPUs currently only allow a single set of simultaneous readout pulses on all qubits in the QPU at the end of the program. This means that missing or duplicate MEASURE instructions do not change the pulse program, but instead only contribute to making a less rich or richer mapping, respectively, between classical and qubit addresses.

Parameters

- **quil_program** (`Program`) – Pyquil program to run on the QPU
- **classical_addresses** (*list | range*) – Classical register addresses to return
- **trials** (*int*) – Number of times to run the program (a.k.a. number of shots)
- **needs_compilation** (*bool*) – If True, preprocesses the job with the compiler.
- **isa** (`ISA`) – If set, specifies a custom ISA to compile to. If left unset, Forest uses the default ISA associated to this QPU device.

Returns A list of a list of classical registers (each register contains a bit)

Return type list

run_and_measure(*quil_program*, *qubits*, *trials=1*, *needs_compilation=True*, *isa=None*)

Similar to run, except for how MEASURE operations are dealt with. With run, users are expected to include MEASURE operations in the program if they want results back. With run_and_measure, users provide a pyquil program that does not have MEASURE instructions, and also provide a list of qubits to measure. All qubits in this list will be measured at the end of the program, and their results stored in corresponding classical registers.

Parameters

- **quil_program** (`Program`) – Pyquil program to run on the QPU
- **qubits** (*list | range*) – The list of qubits to measure
- **trials** (*int*) – Number of times to run the program (a.k.a. number of shots)
- **needs_compilation** (*bool*) – If True, preprocesses the job with the compiler.

- **isa** (ISA) – If set, specifies a custom ISA to compile to. If left unset, Forest uses the default ISA associated to this QPU device.

Returns A list of a list of classical registers (each register contains a bit)

Return type list

run_and_measure_async (*quil_program*, *qubits*, *trials*, *needs_compilation=True*, *isa=None*)
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=None*, *trials=1*, *needs_compilation=True*, *isa=None*)
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.

wait_for_job (*job_id*, *ping_time=None*, *status_time=None*)
Wait for the results of a job and periodically print status

Parameters

- **job_id** – Job id
- **ping_time** – How often to poll the server. Defaults to the value specified in the constructor. (0.1 seconds)
- **status_time** – How often to print status, set to False to never print status. Defaults to the value specified in the constructor (2 seconds)

Returns Completed Job

class pyquil.api.**CompilerConnection** (*device=None*, *sync_endpoint='https://api.rigetti.com'*, *async_endpoint='https://job.rigetti.com/beta'*, *api_key=None*, *user_id=None*, *use_queue=False*, *ping_time=0.1*, *status_time=2*, *isa_source=None*, *specs_source=None*)

Bases: `object`

Represents a connection to the Quil compiler.

apply_clifford_to_pauli (*clifford*, *pauli_in*)
Given a circuit that consists only of elements of the Clifford group, return its action on a PauliTerm.

In particular, for Clifford C, and Pauli P, this returns the PauliTerm representing PCP^{dagger}.

Parameters

- **clifford** (`Program`) – A Program that consists only of Clifford operations.
- **pauli_in** (`PauliTerm`) – A PauliTerm to be acted on by clifford via conjugation.

Returns A PauliTerm corresponding to pauli_in * clifford * pauli_in^{dagger}

compile (*quil_program*, *isa=None*)
Sends a Quil program to the Forest compiler and returns the resulting compiled Program.

Parameters

- **quil_program** (`Program`) – Quil program to be compiled.
- **isa** (`ISA`) – An optional ISA to target. This takes precedence over the `device` or `isa_source` arguments to this object's constructor. If this is not specified, you must have provided one of the aforementioned constructor arguments.

Returns The compiled Program object.

Return type *Program*

compile_async(*quil_program*, *isa=None*)

Similar to compile 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.

generate_rb_sequence(*depth*, *gateset*)

Construct a randomized benchmarking experiment on the given qubits, decomposing into gateset.

The JSON payload that is parsed is a list of lists of indices, or Nones. In the former case, they are the index of the gate in the gateset.

Parameters

- **depth** (*int*) – The number of Clifford gates to include in the randomized benchmarking experiment. This is different than the number of gates in the resulting experiment.
- **gateset** (*list*) – A list of pyquil gates to decompose the Clifford elements into. These must generate the clifford group on the qubits of interest. e.g. for one qubit [RZ(np.pi/2), RX(np.pi/2)].

Returns A list of pyquil programs. Each pyquil program is a circuit that represents an element of the Clifford group. When these programs are composed, the resulting Program will be the randomized benchmarking experiment of the desired depth. e.g. if the return programs are called cliffords then *sum(cliffords, Program())* will give the randomized benchmarking experiment, which will compose to the identity program.

get_job(*job_id*)

Given a job id, return information about the status of the job

Parameters **job_id** (*str*) – job id

Returns Job object with the status and potentially results of the job

Return type *Job*

wait_for_job(*job_id*, *ping_time=None*, *status_time=None*)

Wait for the results of a job and periodically print status

Parameters

- **job_id** – Job id
- **ping_time** – How often to poll the server. Defaults to the value specified in the constructor. (0.1 seconds)
- **status_time** – How often to print status, set to False to never print status. Defaults to the value specified in the constructor (2 seconds)

Returns Completed Job

class pyquil.api.**Job**(*raw*, *machine*)

Bases: object

Represents the current status of a Job in the Forest queue.

Job statuses are initially QUEUED when QVM/QPU resources are not available They transition to RUNNING when they have been started Finally they are marked as FINISHED, ERROR, or CANCELLED once completed

compiled_quil()

If the Quil program associated with the Job was compiled (e.g., to translate it to the QPU's natural gateset) return this compiled program.

Return type Optional[*Program*]

decode()

estimated_time_left_in_queue()
If the job is queued, this will return how much time left (in seconds) is estimated before execution.

gate_depth()
If the job has metadata and this contains the gate depth, return this, otherwise None. The gate depth is a measure of how long a quantum program takes. On a non-fault-tolerant QPU programs with a low gate depth have a higher chance of succeeding.

Return type Optional[int]

gate_volume()
If the job has metadata and this contains the gate volume, return this, otherwise None. On a non-fault-tolerant QPU programs with a low gate volume have a higher chance of succeeding. This is a less sensitive measure than gate depth.

Return type Optional[int]

get()

is_compiling()
Is the job actively compiling?

is_done()
Has the job completed yet?

is_queued()
Is the job still in the Forest queue?

is_queued_for_compilation()
Is the job still in the Forest compilation queue?

is_running()
Is the job currently running?

job_id
Job id :rtype: str

multiqubit_gate_depth()
If the job has metadata and this contains the multiqubit gate depth, return this, otherwise None. The multiqubit gate depth is a measure of how inaccurately a quantum program will behave on nonideal hardware. On a non-fault-tolerant QPU programs with a low gate depth have a higher chance of succeeding.

Return type Optional[int]

position_in_queue()
If the job is queued, this will return how many other jobs are ahead of it. If the job is not queued, this will return None

program_fidelity()
If the job has metadata and this contains a job program fidelity estimate, return this, otherwise None. This is a number between 0 and 1; a higher value means more likely odds of a meaningful answer.

Return type Optional[float]

result()
The result of the job if available throws ValueError is result is not available yet throws ApiError if server returned an error indicating program execution was not successful or if the job was cancelled

topological_swaps()
If the program could not be mapped directly to the QPU because of missing links in the two-qubit gate connectivity graph, the compiler must insert topological swap gates. Return the number of such topological swaps.

Return type Optional[int]

`pyquil.api.`**`get_devices`**(*async_endpoint='https://job.rigetti.com/beta'*, *api_key=None*, *user_id=None*, *as_dict=False*)

Get a list of currently available devices. The arguments for this method are the same as those for QPUConnection. Note that this method will only work for accounts that have QPU access.

Returns Set or Dictionary (keyed by device name) of all available devices.

Return type Set|Dict

2.11.2 pyquil.device

class `pyquil.device.`**`Device`**(*name*, *raw*)

Bases: `object`

A device (quantum chip) that can accept programs. Only devices that are online will actively be accepting new programs. In addition to the `self._raw` attribute, two other attributes are optionally constructed from the entries in `self._raw` – `isa` and `noise_model` – which should conform to the dictionary format required by the `.from_dict()` methods for `ISA` and `NoiseModel`, respectively.

Variables

- **`_raw`** (*`dict`*) – Raw JSON response from the server with additional information about the device.
- **`isa`** (`ISA`) – The instruction set architecture (ISA) for the device.
- **`noise_model`** (`NoiseModel`) – The noise model for the device.

`is_online`()

Whether or not the device is online and accepting new programs.

Return type bool

`is_retuning`()

Whether or not the device is currently retuning.

Return type bool

`status`

Returns a string describing the device's status

- **online**: The device is online and ready for use
- **retuning** : The device is not accepting new jobs because it is re-calibrating
- **offline**: The device is not available for use, potentially because you don't have the right permissions.

class `pyquil.device.`**`Edge`**(*targets*, *type*, *dead*)

Bases: `tuple`

`dead`

Alias for field number 2

`targets`

Alias for field number 0

`type`

Alias for field number 1

`pyquil.device.`**`EdgeSpecs`**

alias of `_QubitQubitSpecs`

class `pyquil.device.`**`ISA`**

Bases: `pyquil.device._ISA`

Basic Instruction Set Architecture specification.

Variables

- **`qubits`** (*`Sequence[Qubit]`*) – The qubits associated with the ISA.
- **`edges`** (*`Sequence[Edge]`*) – The multi-qubit gates.

static from_dict(*d*)

Re-create the ISA from a dictionary representation.

Parameters **`d`** (*`Dict[str, Any]`*) – The dictionary representation.

Returns The restored ISA.

Return type *ISA*

to_dict()

Create a JSON-serializable representation of the ISA.

The dictionary representation is of the form:

```
{
    "1Q": {
        "0": {
            "type": "Xhalves"
        },
        "1": {
            "type": "Xhalves",
            "dead": True
        },
        ...
    },
    "2Q": {
        "1-4": {
            "type": "CZ"
        },
        "1-5": {
            "type": "CZ"
        },
        ...
    },
    ...
}
```

Returns A dictionary representation of self.

Return type Dict[str, Any]

class `pyquil.device.`**`Qubit`**(*id*, *type*, *dead*)

Bases: `tuple`

dead

Alias for field number 2

id

Alias for field number 0

type
Alias for field number 1

pyquil.device.**QubitSpecs**
alias of `_QubitSpecs`

class pyquil.device.**Specs**
Bases: `pyquil.device._Specs`

Basic specifications for the device, such as gate fidelities and coherence times.

Variables

- **qubits_specs** (*List[QubitSpecs]*) – The specs associated with individual qubits.
- **edges_specs** (*List[EdgesSpecs]*) – The specs associated with edges, or qubit-qubit pairs.

T1s()
Get a dictionary of T1s (in seconds) from the specs, keyed by qubit index.

Returns A dictionary of T1s, in seconds.

Return type Dict[int, float]

T2s()
Get a dictionary of T2s (in seconds) from the specs, keyed by qubit index.

Returns A dictionary of T2s, in seconds.

Return type Dict[int, float]

f1QRBs()
Get a dictionary of single-qubit randomized benchmarking fidelities (normalized to unity) from the specs, keyed by qubit index.

Returns A dictionary of 1QRBs, normalized to unity.

Return type Dict[int, float]

fBellStates()
Get a dictionary of two-qubit Bell state fidelities (normalized to unity) from the specs, keyed by targets (qubit-qubit pairs).

Returns A dictionary of Bell state fidelities, normalized to unity.

Return type Dict[tuple(int, int), float]

fCPHASEs()
Get a dictionary of CPHASE fidelities (normalized to unity) from the specs, keyed by targets (qubit-qubit pairs).

Returns A dictionary of CPHASE fidelities, normalized to unity.

Return type Dict[tuple(int, int), float]

fCZs()
Get a dictionary of CZ fidelities (normalized to unity) from the specs, keyed by targets (qubit-qubit pairs).

Returns A dictionary of CZ fidelities, normalized to unity.

Return type Dict[tuple(int, int), float]

fROs()
Get a dictionary of single-qubit readout fidelities (normalized to unity) from the specs, keyed by qubit index.

Returns A dictionary of RO fidelities, normalized to unity.

Return type Dict[int, float]

static from_dict(*d*)

Re-create the Specs from a dictionary representation.

Parameters **Any] d** (*Dict[str,*) – The dictionary representation.

Returns The restored Specs.

Return type *Specs*

to_dict()

Create a JSON-serializable representation of the device Specs.

The dictionary representation is of the form:

```
{
    '1Q': {
        "0": {
            "f1QRB": 0.99,
            "T1": 20e-6,
            ...
        },
        "1": {
            "f1QRB": 0.989,
            "T1": 19e-6,
            ...
        },
        ...
    },
    '2Q': {
        "1-4": {
            "fBellState": 0.93,
            "fCZ": 0.92,
            "fCPHASE": 0.91
        },
        "1-5": {
            "fBellState": 0.9,
            "fCZ": 0.89,
            "fCPHASE": 0.88
        },
        ...
    },
    ...
}
```

Returns A dctionary representation of self.

Return type Dict[str, Any]

pyquil.device.**gates_in_isa**(*isa*)

Generate the full gateset associated with an ISA.

Parameters **isa** (ISA) – The instruction set architecture for a QPU.

Returns A sequence of Gate objects encapsulating all gates compatible with the ISA.

Return type Sequence[*Gate*]

2.11.3 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*`)`

2.11.4 pyquil.noise

Module for creating and verifying noisy gate and readout definitions.

`class` `pyquil.noise.`**`KrausModel`**

Bases: `pyquil.noise._KrausModel`

Encapsulate a single gate's noise model.

Variables

- ***`gate`*** (`str`) – The name of the gate.

- **params** (*Sequence[float]*) – Optional parameters for the gate.
- ***targets*** (*Sequence[int]*) – The target qubit ids.
- **kraus_ops** (*Sequence[np.array]*) – The Kraus operators (must be square complex numpy arrays).
- **fidelity** (*float*) – The average gate fidelity associated with the Kraus map relative to the ideal operation.

static from_dict(*d*)

Recreate a KrausModel from the dictionary representation.

Parameters **d** (*dict*) – The dictionary representing the KrausModel. See *to_dict* for an example.

Returns The deserialized KrausModel.

Return type *KrausModel*

to_dict()

Create a dictionary representation of a KrausModel.

For example:

```
{
    "gate": "RX",
    "params": np.pi,
    "targets": [0],
    "kraus_ops": [            # In this example single Kraus op = ideal␣
↪RX(pi) gate
        [[[0,    0],           # element-wise real part of matrix
          [0,    0]],
          [[0, -1],            # element-wise imaginary part of matrix
          [-1, 0]]]
    ],
    "fidelity": 1.0
}
```

Returns A JSON compatible dictionary representation.

Return type Dict[str,Any]

static unpack_kraus_matrix(*m*)

Helper to optionally unpack a JSON compatible representation of a complex Kraus matrix.

Parameters **m** (*Union[list, np.array]*) – The representation of a Kraus operator. Either a complex square matrix (as numpy array or nested lists) or a JSON-able pair of real matrices (as nested lists) representing the element-wise real and imaginary part of m.

Returns A complex square numpy array representing the Kraus operator.

Return type np.array

class pyquil.noise.**NoiseModel**

Bases: `pyquil.noise._NoiseModel`

Encapsulate the QPU noise model containing information about the noisy gates.

Variables

- ***gates*** (*Sequence[KrausModel]*) – The tomographic estimates of all gates.

- **assignment_probs** (*Dict[int, np.array]*) – The single qubit readout assignment probability matrices keyed by qubit id.

static from_dict (*d*)
Re-create the noise model from a dictionary representation.

Parameters d (*Dict[str, Any]*) – The dictionary representation.

Returns The restored noise model.

Return type *NoiseModel*

gates_by_name (*name*)
Return all defined noisy gates of a particular gate name.

Parameters name (*str*) – The gate name.

Returns A list of noise models representing that gate.

Return type Sequence[*KrausModel*]

to_dict ()
Create a JSON serializable representation of the noise model.

For example:

```
{
    "gates": [
        # list of embedded dictionary representations of KrausModels here [...
↪]
    ]
    "assignment_probs": {
        "0": [[.8, .1],
              [.2, .9]],
        "1": [[.9, .4],
              [.1, .6]],
    }
}
```

Returns A dictionary representation of self.

Return type Dict[str,Any]

exception pyquil.noise.**NoisyGateUndefined**
Bases: Exception

Raise when user attempts to use noisy gate outside of currently supported set.

pyquil.noise.**add_decoherence_noise** (*prog*, *T1=3e-05*, *T2=3e-05*, *gate_time_1q=5e-08*, *gate_time_2q=1.5e-07*, *ro_fidelity=0.95*)
Add generic damping and dephasing noise to a program.

This high-level function is provided as a convenience to investigate the effects of a generic noise model on a program. For more fine-grained control, please investigate the other methods available in the `pyquil.noise` module.

In an attempt to closely model the QPU, noisy versions of RX(+-pi/2) and CZ are provided; I and parametric RZ are noiseless, and other gates are not allowed. To use this function, you need to compile your program to this native gate set.

The default noise parameters

- T1 = 30 us

- T2 = 30 us
- 1q gate time = 50 ns
- 2q gate time = 150 ns

are currently typical for near-term devices.

This function will define new gates and add Kraus noise to these gates. It will translate the input program to use the noisy version of the gates.

Parameters

- **prog** – A pyquil program consisting of I, RZ, CZ, and RX(+-pi/2) instructions
- **T1** (*Union[Dict[int, float], float]*) – The T1 amplitude damping time either globally or in a dictionary indexed by qubit id. By default, this is 30 us.
- **T2** (*Union[Dict[int, float], float]*) – The T2 dephasing time either globally or in a dictionary indexed by qubit id. By default, this is also 30 us.
- **gate_time_1q** (*float*) – The duration of the one-qubit gates, namely RX(+pi/2) and RX(-pi/2). By default, this is 50 ns.
- **gate_time_2q** (*float*) – The duration of the two-qubit gates, namely CZ. By default, this is 150 ns.
- **ro_fidelity** (*Union[Dict[int, float], float]*) – The readout assignment fidelity $F = (p(0|0) + p(1|1))/2$ either globally or in a dictionary indexed by qubit id.

Returns A new program with noisy operators.

`pyquil.noise.`**`append_kraus_to_gate`**(*kraus_ops*, *gate_matrix*)

Follow a gate `gate_matrix` by a Kraus map described by `kraus_ops`.

Parameters

- **kraus_ops** (*list*) – The Kraus operators.
- **gate_matrix** (*numpy.ndarray*) – The unitary gate.

Returns A list of transformed Kraus operators.

`pyquil.noise.`**`apply_noise_model`**(*prog*, *noise_model*)

Apply a noise model to a program and generated a 'noisy-fied' version of the program.

Parameters

- **prog** (`Program`) – A Quil Program object.
- **noise_model** (`NoiseModel`) – A NoiseModel, either generated from an ISA or from a simple decoherence model.

Returns A new program translated to a noisy gateset and with noisy readout as described by the noisemodel.

Return type *Program*

`pyquil.noise.`**`bitstring_probs_to_z_moments`**(*p*)

Convert between bitstring probabilities and joint Z moment expectations.

Parameters **p** (*np.array*) – An array that enumerates bitstring probabilities. When flattened out `p = [p_00...0, p_00...1, ...,p_11...1]`. The total number of elements must therefore be a power of 2. The canonical shape has a separate axis for each qubit, such that `p[i,j,...,k]` gives the estimated probability of bitstring `ij...k`.

Returns

`z_moments`, an np.array with one length-2 axis per qubit which contains the expectations of all monomials in `{I, Z_0, Z_1, ..., Z_{n-1}}`. The expectations of each monomial can be accessed via:

```
<Z_0^j_0 Z_1^j_1 ... Z_m^j_m> = z_moments[j_0,j_1,...,j_m]
```

Return type np.array

`pyquil.noise.`**`combine_kraus_maps`**(*k1*, *k2*)

Generate the Kraus map corresponding to the composition of two maps on the same qubits with k1 being applied to the state after k2.

Parameters

- **k1** (*list*) – The list of Kraus operators that are applied second.
- **k2** (*list*) – The list of Kraus operators that are applied first.

Returns A combinatorially generated list of composed Kraus operators.

`pyquil.noise.`**`correct_bitstring_probs`**(*p*, *assignment_probabilities*)

Given a 2d array of corrupted bitstring probabilities (outer axis iterates over shots, inner axis over bits) and a list of assignment probability matrices (one for each bit in the readout) compute the corrected probabilities.

Parameters

- **p** (*np.array*) – An array that enumerates bitstring probabilities. When flattened out `p = [p_00...0, p_00...1, ...,p_11...1]`. The total number of elements must therefore be a power of 2. The canonical shape has a separate axis for each qubit, such that `p[i,j,...,k]` gives the estimated probability of bitstring `ij...k`.
- **assignment_probabilities** (*List[np.array]*) – A list of assignment probability matrices per qubit. Each assignment probability matrix is expected to be of the form:

```
[[p00 p01]
 [p10 p11]]
```

Returns `p_corrected` an array with as many dimensions as there are qubits that contains the noisy-readout-corrected estimated probabilities for each measured bitstring, i.e., `p[i,j,...,k]` gives the estimated probability of bitstring `ij...k`.

Return type np.array

`pyquil.noise.`**`corrupt_bitstring_probs`**(*p*, *assignment_probabilities*)

Given a 2d array of true bitstring probabilities (outer axis iterates over shots, inner axis over bits) and a list of assignment probability matrices (one for each bit in the readout, ordered like the inner axis of results) compute the corrupted probabilities.

Parameters

- **p** (*np.array*) – An array that enumerates bitstring probabilities. When flattened out `p = [p_00...0, p_00...1, ...,p_11...1]`. The total number of elements must therefore be a power of 2. The canonical shape has a separate axis for each qubit, such that `p[i,j,...,k]` gives the estimated probability of bitstring `ij...k`.
- **assignment_probabilities** (*List[np.array]*) – A list of assignment probability matrices per qubit. Each assignment probability matrix is expected to be of the form:

```
[[p00 p01]
 [p10 p11]]
```

Returns `p_corrected` an array with as many dimensions as there are qubits that contains the noisy-readout-corrected estimated probabilities for each measured bitstring, i.e., `p[i, j, ..., k]` gives the estimated probability of bitstring `ij...k`.

Return type np.array

`pyquil.noise.`**`damping_after_dephasing`**(*T1*, *T2*, *gate_time*)

Generate the Kraus map corresponding to the composition of a dephasing channel followed by an amplitude damping channel.

Parameters

- **`T1`** (*`float`*) – The amplitude damping time
- **`T2`** (*`float`*) – The dephasing time
- **`gate_time`** (*`float`*) – The gate duration.

Returns A list of Kraus operators.

`pyquil.noise.`**`damping_kraus_map`**(*p=0.1*)

Generate the Kraus operators corresponding to an amplitude damping noise channel.

Parameters **`p`** (*`float`*) – The one-step damping probability.

Returns A list [k1, k2] of the Kraus operators that parametrize the map.

Return type list

`pyquil.noise.`**`dephasing_kraus_map`**(*p=0.1*)

Generate the Kraus operators corresponding to a dephasing channel.

Params float p The one-step dephasing probability.

Returns A list [k1, k2] of the Kraus operators that parametrize the map.

Return type list

`pyquil.noise.`**`estimate_assignment_probs`**(*q*, *trials*, *cxn*, *p0=None*)

Estimate the readout assignment probabilities for a given qubit `q`. The returned matrix is of the form:

```
[[p00 p01]
 [p10 p11]]
```

Parameters

- **`q`** (*`int`*) – The index of the qubit.
- **`trials`** (*`int`*) – The number of samples for each state preparation.
- **`cxn`** (*`Union[QVMConnection, QPUConnection]`*) – The quantum abstract machine to sample from.
- **`p0`** (`Program`) – A header program to prepend to the state preparation programs.

Returns The assignment probability matrix

Return type np.array

`pyquil.noise.`**`estimate_bitstring_probs`**(*results*)

Given an array of single shot results estimate the probability distribution over all bitstrings.

Parameters **`results`** (*`np.array`*) – A 2d array where the outer axis iterates over shots and the inner axis over bits.

Returns An array with as many axes as there are qubit and normalized such that it sums to one. `p[i,j,...,k]` gives the estimated probability of bitstring `ij...k`.

Return type np.array

`pyquil.noise.`**`get_noisy_gate`**(*gate_name*, *params*)

Look up the numerical gate representation and a proposed 'noisy' name.

Parameters

- **`gate_name`** (*`str`*) – The Quil gate name
- **`params`** (*`Tuple[float]`*) – The gate parameters.

Returns A tuple (matrix, noisy_name) with the representation of the ideal gate matrix and a proposed name for the noisy version.

Return type Tuple[np.array, str]

`pyquil.noise.`**`tensor_kraus_maps`**(*k1*, *k2*)

Generate the Kraus map corresponding to the composition of two maps on different qubits.

Parameters

- **`k1`** (*`list`*) – The Kraus operators for the first qubit.
- **`k2`** (*`list`*) – The Kraus operators for the second qubit.

Returns A list of tensored Kraus operators.

2.11.5 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*

2.11.6 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_programs``()`

Get a Pyquil Program corresponding to each term in the PauliSum and a coefficient for each program

Returns (programs, coefficients)

`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(*sort_ops=True*)
Returns an identifier string for the PauliTerm (ignoring the coefficient).

Don't use this to compare terms. This function will not work with qubits that aren't sortable.

Parameters **sort_ops** – Whether to sort operations by qubit. This is True by default for backwards compatibility but will change in pyQuil 2.0. Callers should never rely on comparing id's for testing equality. See `operations_as_set` instead.

Returns A string representation of this term's operations.

Return type string

operations_as_set()
Return a frozenset of operations in this term.

Use this in place of *id()* if the order of operations in the term does not matter.

Returns frozenset of strings representing Pauli operations

pauli_string(*qubits=None*)
Return a string representation of this PauliTerm mod its phase, as a concatenation of the string representation of the >>> p = PauliTerm("X", 0) * PauliTerm("Y", 1, 1.j) >>> p.pauli_string() "XY" >>> p.pauli_string([0]) "X" >>> p.pauli_string([0, 2]) "XI"

Parameters **qubits** (*list*) – The list of qubits to represent, given as ints. If None, defaults to all qubits in this PauliTerm.

Returns The string representation of this PauliTerm, modulo its phase.

Return type String

program

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

`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.`**`exponentiate_commuting_pauli_sum`**`(`*pauli_sum*`)`
Returns a function that maps all substituent PauliTerms and sums them into a program. NOTE: Use this function with care. Substituent PauliTerms should commute.

Parameters **`pauli_sum`** (`PauliSum`) – PauliSum to exponentiate.

Returns A function that parametrizes the exponential.

Return type function

`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.`**`simplify_pauli_sum`**(*pauli_sum*)

`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*

2.11.7 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*, *parameters=None*)

Define a new static gate.

Note: The matrix elements along each axis are ordered by bitstring. For two qubits the order is `00, 01, 10, 11`, where the the bits **are ordered in reverse** by the qubit index, i.e., for qubits 0 and 1 the bitstring `01` indicates that qubit 0 is in the state 1. See also *the related documentation section in the QVM Overview*.

Parameters

- **`name`** (*`string`*) – The name of the gate.
- **`matrix`** (*`array-like`*) – List of lists or Numpy 2d array.
- **`parameters`** (*`list`*) – list of parameters that are used in this gate

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.

Note: The matrix elements along each axis are ordered by bitstring. For two qubits the order is `00, 01, 10, 11`, where the the bits **are ordered in reverse** by the qubit index, i.e., for qubits 0 and 1 the bitstring `01` indicates that qubit 0 is in the state 1. See also *the related documentation section in the QVM Overview*.

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*

define_noisy_readout (*qubit*, *p00*, *p11*)

For this program define a classical bit flip readout error channel parametrized by `p00` and `p11`. This models the effect of thermal noise that corrupts the readout signal **after** it has interrogated the qubit.

Parameters

- **qubit** (*int | QubitPlaceholder*) – The qubit with noisy readout.
- **p00** (*float*) – The probability of obtaining the measurement result 0 given that the qubit is in state 0.
- **p11** (*float*) – The probability of obtaining the measurement result 1 given that the qubit is in state 1.

Returns The Program with an appended READOUT-POVM Pragma.

Return type *Program*

defined_gates

A list of defined gates on the program.

gate (*name*, *params*, *qubits*)

Add a gate to the program.

Note: The matrix elements along each axis are ordered by bitstring. For two qubits the order is `00, 01, 10, 11`, where the the bits **are ordered in reverse** by the qubit index, i.e., for qubits 0 and 1 the bitstring `01` indicates that qubit 0 is in the state 1. See also *the related documentation section in the QVM Overview*.

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 (*indices=True*)

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
```

Parameters **indices** – Return qubit indices as integers intead of the wrapping `Qubit` object

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(i) for i in range(4)) # A generator 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. If no qubit/register pairs are provided, measure all qubits present in the program into classical addresses of the same index.

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 ()
Serializes the Quil program to a string suitable for submitting to the QVM or QPU.

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.`**`address_qubits`**(*program*, *qubit_mapping=None*)

Takes a program which contains placeholders and assigns them all defined values.

Either all qubits must be defined or all undefined. If qubits are undefined, you may provide a qubit mapping to specify how placeholders get mapped to actual qubits. If a mapping is not provided, integers 0 through N are used.

This function will also instantiate any label placeholders.

Parameters

- **program** – The program.
- **qubit_mapping** – A dictionary-like object that maps from `QubitPlaceholder` to `Qubit` or `int` (but not both).

Returns A new Program with all qubit and label placeholders assigned to real qubits and labels.

`pyquil.quil.`**`get_classical_addresses_from_program`**(*program*)

Returns a sorted list of classical addresses found in the MEASURE instructions in the program.

Parameters **program** (Program) – The program from which to get the classical addresses.

Returns A list of integer classical addresses.

Return type list

`pyquil.quil.`**`get_default_qubit_mapping`**(*program*)

Takes a program which contains qubit placeholders and provides a mapping to the integers 0 through N-1.

The output of this function is suitable for input to *`address_qubits()`*.

Parameters **program** – A program containing qubit placeholders

Returns A dictionary mapping qubit placeholder to an addressed qubit from 0 through N-1.

`pyquil.quil.`**`instantiate_labels`**(*instructions*)

Takes an iterable of instructions which may contain label placeholders and assigns them all defined values.

Returns list of instructions with all label placeholders assigned to real labels.

`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*

2.11.8 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.**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*, *parameters=None*)
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.
- **parameters** (*list*) – list of parameters that are used in this gate

get_constructor()

Returns A function that constructs this gate on variable qubit indices. E.g. *my-gate.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.

`get_qubits`(*indices=True*)

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

This is the pyQuil object for a Quil measurement instruction.

`get_qubits`(*indices=True*)

`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.**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'

2.11.9 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.

2.11.10 pyquil.wavefunction

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

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

Encapsulate a wavefunction representing a quantum state as returned by the QVM.

Note: The elements of the wavefunction are ordered by bitstring. E.g., for two qubits the order is `00, 01, 10, 11`, where the the bits **are ordered in reverse** by the qubit index, i.e., for qubits 0 and 1 the bitstring `01` indicates that qubit 0 is in the state 1. See also *the related documentation section in the QVM Overview*.

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

probabilities()
Returns an array of probabilities in lexicographical order

sample_bitstrings(*n_samples*)
Sample bitstrings from the distribution defined by the wavefunction.

Parameters **n_samples** – The number of bitstrings to sample

Returns An array of shape (n_samples, n_qubits)

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

2.12 Changelog

2.12.1 v1.9 (June 6, 2018)

We're happy to announce the release of Pyquil 1.9. Pyquil is Rigetti's toolkit for constructing and running quantum programs. This release is the latest in our series of regular releases, and it's filled with convenience features, enhancements, bug fixes, and documentation improvements.

Special thanks to community members sethuiyer, vtomole, rht, akarazeev, ejdanderson, markf94, playadust, and kadora626 for contributing to this release!

Qubit placeholders

One of the focuses of this release is a re-worked concept of "Qubit Placeholders". These are logical qubits that can be used to construct programs. Now, a program containing qubit placeholders must be "addressed" prior to running on a QPU or QVM. The addressing stage involves mapping each qubit placeholder to a physical qubit (represented as an integer). For example, if you have a 3 qubit circuit that you want to run on different sections of the Agave chip, you now can prepare one Program and address it to many different subgraphs of the chip topology. Check out the `QubitPlaceholder` example notebook for more.

To support this idea, we've refactored parts of Pyquil to remove the assumption that qubits can be "sorted". While true for integer qubit labels, this probably isn't true in general. A notable change can be found in the construction of a `PauliSum`: now terms will stay in the order they were constructed.

- `PauliTerm` now remembers the order of its operations. `sX(1)*sZ(2)` will compile to different Quil code than `sZ(2)*sX(1)`, although the terms will still be equal according to the __eq__ method. During `PauliSum` combination of like terms, a warning will be emitted if two terms are combined that have different orders of operation.
- `PauliTerm.id()` takes an optional argument `sort_ops` which defaults to True for backwards compatibility. However, this function should not be used for comparing term-type like it has been used previously. Use `PauliTerm.operations_as_set()` instead. In the future, `sort_ops` will default to False and will eventually be removed.

- `Program.alloc()` has been deprecated. Please instantiate `QubitPlaceholder()` directly or request a "register" (list) of `n` placeholders by using the class constructor `QubitPlaceholder.register(n)()`.
- Programs must contain either (1) all instantiated qubits with integer indexes or (2) all placeholder qubits of type `QubitPlaceholder`. We have found that most users use (1) but (2) will become useful with larger and more diverse devices.
- Programs that contain qubit placeholders must be **explicitly addressed** prior to execution. Previously, qubits would be assigned "under the hood" to integers 0. . . N. Now, you must use `address_qubits()` which returns a new program with all qubits indexed depending on the `qubit_mapping` argument. The original program is unaffected and can be "readdressed" multiple times.
- `PauliTerm` can now accept `QubitPlaceholder` in addition to integers.
- `QubitPlaceholder` is no longer a subclass of `Qubit`. `LabelPlaceholder` is no longer a subclass of `Label`.
- `QuilAtom` subclasses' hash functions have changed.

Randomized benchmarking sequence generation

Pyquil now includes support for performing a simple benchmarking routine - randomized benchmarking. There is a new method in the `CompilerConnection` that will return sequences of pyquil programs, corresponding to elements of the Clifford group. These programs are uniformly randomly sampled, and have the property that they compose to the identity. When concatenated and run as one program, these programs can be used in a procedure called randomized benchmarking to gain insight about the fidelity of operations on a QPU.

In addition, the `CompilerConnection` has another new method, `apply_clifford_to_pauli()`, which conjugates `PauliTerms` by :py:class:'Program's that are composed of Clifford gates. That is to say, given a circuit C, that contains only gates corresponding to elements of the Clifford group, and a tensor product of elements P, from the Pauli group, this method will compute $PCP^{dagger}$. Such a procedure can be used in various ways. An example is predicting the effect a Clifford circuit will have on an input state modeled as a density matrix, which can be written as a sum of Pauli matrices.

Ease of Use

This release includes some quality-of-life improvements such as the ability to initialize programs with generator expressions, sensible defaults for `Program.measure_all()`, and sensible defaults for `classical_addresses` in `run()` methods.

- `Program` can be initiated with a generator expression.
- `Program.measure_all()` (with no arguments) will measure all qubits in a program.
- `classical_addresses` is now optional in QVM and QPU `run()` methods. By default, any classical addresses targeted by `MEASURE` will be returned.
- `QVMConnection.pauli_expectation()` accepts `PauliSum` as arguments. This offers a more sensible API compared to `QVMConnection.expectation()`.
- pyQuil will now retry jobs every 10 seconds if the QPU is re-tuning.
- `CompilerConnection.compile()` now takes an optional argument `isa` that allows per-compilation specification of the target ISA.
- An empty program will trigger an exception if you try to run it.

Supported versions of Python

We strongly support using Python 3 with Pyquil. Although this release works with Python 2, we are dropping official support for this legacy language and moving to community support for Python 2. The next major release of Pyquil will introduce Python 3.5+ only features and will no longer work without modification for Python 2.

Bug fixes

- `shift_quantum_gates` has been removed. Users who relied on this functionality should use `QubitPlaceholder` and `address_qubits()` to achieve the same result. Users should also double-check data resulting from use of this function as there were several edge cases which would cause the shift to be applied incorrectly resulting in badly-addressed qubits.
- Slightly perturbed angles when performing RX gates under a Kraus noise model could result in incorrect behavior.
- The quantum die example returned incorrect values when `n = 2^m`.

CHAPTER 3

Indices and Tables

- genindex
- modindex
- search

Bibliography

[Nielsen2002] Nielsen, M. A. (2002) ‘A simple formula for the average gate fidelity of a quantum dynamical operation’, http://arxiv.org/abs/quant-ph/0205035

[Reagor2018] Reagor, M. et al. (2018) ‘Demonstration of universal parametric entangling gates on a multi-qubit lattice’, http://advances.sciencemag.org/lookup/doi/10.1126/sciadv.aao3603

[DensityMatrix] https://en.wikipedia.org/wiki/Density_matrix

Python Module Index

p

pyquil.api, 68
pyquil.device, 75
pyquil.gates, 79
pyquil.noise, 80
pyquil.parametric, 86
pyquil.paulis, 87
pyquil.quil, 91
pyquil.quilbase, 95
pyquil.slot, 98
pyquil.wavefunction, 99

Index

A

AbstractInstruction (class in pyquil.quilbase), 95
add_decoherence_noise() (in module pyquil.noise), 82
address_qubits() (in module pyquil.quil), 95
alloc() (pyquil.quil.Program method), 91
AND() (in module pyquil.gates), 79
append_kraus_to_gate() (in module pyquil.noise), 83
apply_clifford_to_pauli() (pyquil.api.CompilerConnection method), 72
apply_noise_model() (in module pyquil.noise), 83
argument_count() (in module pyquil.parametric), 86

B

BinaryClassicalInstruction (class in pyquil.quilbase), 96
bitstring_probs_to_z_moments() (in module pyquil.noise), 83

C

CCNOT() (in module pyquil.gates), 79
check_commutation() (in module pyquil.paulis), 88
ClassicalAnd (class in pyquil.quilbase), 96
ClassicalExchange (class in pyquil.quilbase), 96
ClassicalFalse (class in pyquil.quilbase), 96
ClassicalMove (class in pyquil.quilbase), 96
ClassicalNot (class in pyquil.quilbase), 96
ClassicalOr (class in pyquil.quilbase), 96
ClassicalTrue (class in pyquil.quilbase), 96
CNOT() (in module pyquil.gates), 79
combine_kraus_maps() (in module pyquil.noise), 84
commuting_sets() (in module pyquil.paulis), 88
compile() (pyquil.api.CompilerConnection method), 72
compile_async() (pyquil.api.CompilerConnection method), 72
compiled_quil() (pyquil.api.Job method), 73
CompilerConnection (class in pyquil.api), 72
copy() (pyquil.paulis.PauliTerm method), 87
correct_bitstring_probs() (in module pyquil.noise), 84
corrupt_bitstring_probs() (in module pyquil.noise), 84
CPHASE() (in module pyquil.gates), 79
CPHASE00() (in module pyquil.gates), 79
CPHASE01() (in module pyquil.gates), 79
CPHASE10() (in module pyquil.gates), 79
CSWAP() (in module pyquil.gates), 79
CZ() (in module pyquil.gates), 79

D

dagger() (pyquil.quil.Program method), 91
damping_after_dephasing() (in module pyquil.noise), 85
damping_kraus_map() (in module pyquil.noise), 85
dead (pyquil.device.Edge attribute), 75
dead (pyquil.device.Qubit attribute), 76
decode() (pyquil.api.Job method), 73
DefGate (class in pyquil.quilbase), 96
defgate() (pyquil.quil.Program method), 91
define_noisy_gate() (pyquil.quil.Program method), 91
define_noisy_readout() (pyquil.quil.Program method), 92
defined_gates (pyquil.quil.Program attribute), 92
dephasing_kraus_map() (in module pyquil.noise), 85
Device (class in pyquil.device), 75

E

Edge (class in pyquil.device), 75
EdgeSpecs (in module pyquil.device), 75
estimate_assignment_probs() (in module pyquil.noise), 85
estimate_bitstring_probs() (in module pyquil.noise), 85
estimated_time_left_in_queue() (pyquil.api.Job method), 73
EXCHANGE() (in module pyquil.gates), 79
expectation() (pyquil.api.QVMConnection method), 68
expectation_async() (pyquil.api.QVMConnection method), 69
exponential_map() (in module pyquil.paulis), 88
exponentiate() (in module pyquil.paulis), 89
exponentiate_commuting_pauli_sum() (in module pyquil.paulis), 89

F

f1QRBs() (pyquil.device.Specs method), 77
FALSE() (in module pyquil.gates), 79
fBellStates() (pyquil.device.Specs method), 77
fCPHASEs() (pyquil.device.Specs method), 77
fCZs() (pyquil.device.Specs method), 77
from_bit_packed_string() (pyquil.wavefunction.Wavefunction static method), 99
from_dict() (pyquil.device.ISA static method), 76
from_dict() (pyquil.device.Specs static method), 78
from_dict() (pyquil.noise.KrausModel static method), 81
from_dict() (pyquil.noise.NoiseModel static method), 82
from_list() (pyquil.paulis.PauliTerm class method), 87
fROs() (pyquil.device.Specs method), 77
fuse() (pyquil.parametric.ParametricProgram method), 86

G

Gate (class in pyquil.quilbase), 97
gate() (pyquil.quil.Program method), 92
gate_depth() (pyquil.api.Job method), 74
gate_volume() (pyquil.api.Job method), 74
gates_by_name() (pyquil.noise.NoiseModel method), 82
gates_in_isa() (in module pyquil.device), 78
generate_rb_sequence() (pyquil.api.CompilerConnection method), 73
get() (pyquil.api.Job method), 74
get_bitstring_from_index() (in module pyquil.wavefunction), 100
get_classical_addresses_from_program() (in module pyquil.quil), 95
get_constructor() (pyquil.quilbase.DefGate method), 96
get_default_qubit_mapping() (in module pyquil.quil), 95
get_devices() (in module pyquil.api), 75
get_job() (pyquil.api.CompilerConnection method), 73
get_job() (pyquil.api.QPUConnection method), 71
get_job() (pyquil.api.QVMConnection method), 69
get_noisy_gate() (in module pyquil.noise), 86
get_outcome_probs() (pyquil.wavefunction.Wavefunction method), 99
get_programs() (pyquil.paulis.PauliSum method), 87
get_qubits() (pyquil.paulis.PauliSum method), 87
get_qubits() (pyquil.paulis.PauliTerm method), 87
get_qubits() (pyquil.quil.Program method), 92
get_qubits() (pyquil.quilbase.Gate method), 97
get_qubits() (pyquil.quilbase.Measurement method), 97
ground() (pyquil.wavefunction.Wavefunction static method), 99

H

H() (in module pyquil.gates), 79
Halt (class in pyquil.quilbase), 97

I

I() (in module pyquil.gates), 79
id (pyquil.device.Qubit attribute), 76
ID() (in module pyquil.paulis), 87
id() (pyquil.paulis.PauliTerm method), 87
if_then() (pyquil.quil.Program method), 92
inst() (pyquil.quil.Program method), 93
instantiate_labels() (in module pyquil.quil), 95
instructions (pyquil.quil.Program attribute), 93
is_compiling() (pyquil.api.Job method), 74
is_done() (pyquil.api.Job method), 74
is_identity() (in module pyquil.paulis), 89
is_online() (pyquil.device.Device method), 75
is_protoquil() (pyquil.quil.Program method), 93
is_queued() (pyquil.api.Job method), 74
is_queued_for_compilation() (pyquil.api.Job method), 74
is_retuning() (pyquil.device.Device method), 75
is_running() (pyquil.api.Job method), 74
is_zero() (in module pyquil.paulis), 89
ISA (class in pyquil.device), 75
ISWAP() (in module pyquil.gates), 79

J

Job (class in pyquil.api), 73
job_id (pyquil.api.Job attribute), 74
Jump (class in pyquil.quilbase), 97
JumpConditional (class in pyquil.quilbase), 97
JumpTarget (class in pyquil.quilbase), 97
JumpUnless (class in pyquil.quilbase), 97
JumpWhen (class in pyquil.quilbase), 97

K

KrausModel (class in pyquil.noise), 80

M

MEASURE() (in module pyquil.gates), 79
measure() (pyquil.quil.Program method), 94
measure_all() (pyquil.quil.Program method), 94
Measurement (class in pyquil.quilbase), 97
merge_programs() (in module pyquil.quil), 95
MOVE() (in module pyquil.gates), 79
multiqubit_gate_depth() (pyquil.api.Job method), 74

N

no_noise() (pyquil.quil.Program method), 94
NoiseModel (class in pyquil.noise), 81
NoisyGateUndefined, 82
Nop (class in pyquil.quilbase), 97
NOT() (in module pyquil.gates), 80
num_args() (pyquil.quilbase.DefGate method), 96

O

op (pyquil.quilbase.ClassicalAnd attribute), 96

op (pyquil.quilbase.ClassicalExchange attribute), 96
op (pyquil.quilbase.ClassicalFalse attribute), 96
op (pyquil.quilbase.ClassicalMove attribute), 96
op (pyquil.quilbase.ClassicalNot attribute), 96
op (pyquil.quilbase.ClassicalOr attribute), 96
op (pyquil.quilbase.ClassicalTrue attribute), 96
op (pyquil.quilbase.Halt attribute), 97
op (pyquil.quilbase.JumpUnless attribute), 97
op (pyquil.quilbase.JumpWhen attribute), 97
op (pyquil.quilbase.Nop attribute), 97
op (pyquil.quilbase.Reset attribute), 98
op (pyquil.quilbase.Wait attribute), 98
operations_as_set() (pyquil.paulis.PauliTerm method), 88
OR() (in module pyquil.gates), 80
out() (pyquil.quil.Program method), 94
out() (pyquil.quilbase.AbstractInstruction method), 95
out() (pyquil.quilbase.BinaryClassicalInstruction method), 96
out() (pyquil.quilbase.DefGate method), 96
out() (pyquil.quilbase.Gate method), 97
out() (pyquil.quilbase.Jump method), 97
out() (pyquil.quilbase.JumpConditional method), 97
out() (pyquil.quilbase.JumpTarget method), 97
out() (pyquil.quilbase.Measurement method), 97
out() (pyquil.quilbase.Pragma method), 98
out() (pyquil.quilbase.RawInstr method), 98
out() (pyquil.quilbase.SimpleInstruction method), 98
out() (pyquil.quilbase.UnaryClassicalInstruction method), 98

P

parametric() (in module pyquil.parametric), 87
ParametricProgram (class in pyquil.parametric), 86
pauli_expectation() (pyquil.api.QVMConnection method), 69
pauli_string() (pyquil.paulis.PauliTerm method), 88
PauliSum (class in pyquil.paulis), 87
PauliTerm (class in pyquil.paulis), 87
PHASE() (in module pyquil.gates), 80
ping() (pyquil.api.QVMConnection method), 69
plot() (pyquil.wavefunction.Wavefunction method), 99
pop() (pyquil.quil.Program method), 94
position_in_queue() (pyquil.api.Job method), 74
Pragma (class in pyquil.quilbase), 98
pretty_print() (pyquil.wavefunction.Wavefunction method), 99
pretty_print_probabilities() (pyquil.wavefunction.Wavefunction method), 99
probabilities() (pyquil.wavefunction.Wavefunction method), 99
Program (class in pyquil.quil), 91
program (pyquil.paulis.PauliTerm attribute), 88
program_fidelity() (pyquil.api.Job method), 74
PSWAP() (in module pyquil.gates), 80
pyquil.api (module), 68
pyquil.device (module), 75
pyquil.gates (module), 79
pyquil.noise (module), 80
pyquil.parametric (module), 86
pyquil.paulis (module), 87
pyquil.quil (module), 91
pyquil.quilbase (module), 95
pyquil.slot (module), 98
pyquil.wavefunction (module), 99

Q

QPUConnection (class in pyquil.api), 71
Qubit (class in pyquil.device), 76
QubitSpecs (in module pyquil.device), 77
QVMConnection (class in pyquil.api), 68

R

RawInstr (class in pyquil.quilbase), 98
Reset (class in pyquil.quilbase), 98
result() (pyquil.api.Job method), 74
run() (pyquil.api.QPUConnection method), 71
run() (pyquil.api.QVMConnection method), 69
run_and_measure() (pyquil.api.QPUConnection method), 71
run_and_measure() (pyquil.api.QVMConnection method), 69
run_and_measure_async() (pyquil.api.QPUConnection method), 72
run_and_measure_async() (pyquil.api.QVMConnection method), 70
run_async() (pyquil.api.QPUConnection method), 72
run_async() (pyquil.api.QVMConnection method), 70
RX() (in module pyquil.gates), 80
RY() (in module pyquil.gates), 80
RZ() (in module pyquil.gates), 80

S

S() (in module pyquil.gates), 80
sample_bitstrings() (pyquil.wavefunction.Wavefunction method), 100
sI() (in module pyquil.paulis), 89
SimpleInstruction (class in pyquil.quilbase), 98
simplify() (pyquil.paulis.PauliSum method), 87
simplify_pauli_sum() (in module pyquil.paulis), 90
Slot (class in pyquil.slot), 98
Specs (class in pyquil.device), 77
status (pyquil.device.Device attribute), 75
suzuki_trotter() (in module pyquil.paulis), 90
SWAP() (in module pyquil.gates), 80
sX() (in module pyquil.paulis), 89
sY() (in module pyquil.paulis), 89

sZ() (in module pyquil.paulis), 90

T

T() (in module pyquil.gates), 80
T1s() (pyquil.device.Specs method), 77
T2s() (pyquil.device.Specs method), 77
targets (pyquil.device.Edge attribute), 75
tensor_kraus_maps() (in module pyquil.noise), 86
term_with_coeff() (in module pyquil.paulis), 90
to_dict() (pyquil.device.ISA method), 76
to_dict() (pyquil.device.Specs method), 78
to_dict() (pyquil.noise.KrausModel method), 81
to_dict() (pyquil.noise.NoiseModel method), 82
topological_swaps() (pyquil.api.Job method), 74
trotterize() (in module pyquil.paulis), 90
TRUE() (in module pyquil.gates), 80
type (pyquil.device.Edge attribute), 75
type (pyquil.device.Qubit attribute), 76

U

UnaryClassicalInstruction (class in pyquil.quilbase), 98
UnequalLengthWarning, 88
unpack_kraus_matrix() (pyquil.noise.KrausModel static method), 81

V

value() (pyquil.slot.Slot method), 99

W

Wait (class in pyquil.quilbase), 98
wait_for_job() (pyquil.api.CompilerConnection method), 73
wait_for_job() (pyquil.api.QPUConnection method), 72
wait_for_job() (pyquil.api.QVMConnection method), 70
Wavefunction (class in pyquil.wavefunction), 99
wavefunction() (pyquil.api.QVMConnection method), 70
wavefunction_async() (pyquil.api.QVMConnection method), 71
while_do() (pyquil.quil.Program method), 94

X

X() (in module pyquil.gates), 80

Y

Y() (in module pyquil.gates), 80

Z

Z() (in module pyquil.gates), 80
ZERO() (in module pyquil.paulis), 88
zeros() (pyquil.wavefunction.Wavefunction static method), 100