
pyQuil Documentation

Release 1.1.1

Rigetti Computing

Aug 09, 2017

1	Overview	1
2	Brief Introduction to Quantum Computing	3
2.1	From Bit to Qubit	3
2.1.1	Probabilistic Bits as Vector Spaces	3
2.1.2	Dirac Notation	5
2.1.3	Multiple Probabilistic Bits	5
2.1.4	Qubits	5
2.1.5	An Important Distinction	6
2.1.6	Some Code	7
2.2	Qubit Operations	8
2.2.1	Pauli Operators	9
2.2.2	Multi-Qubit Operations	10
2.3	The Quantum Abstract Machine	11
2.3.1	Qubit Measurements	11
2.3.2	Classical/Quantum Interaction	12
2.3.3	Classical Control	14
2.3.4	Example: The Probabilistic Halting Problem	15
3	Installation and Getting Started	17
3.1	Environment Setup	18
3.1.1	Prerequisites	18
3.1.2	Installation	18
3.1.3	Connecting to the Rigetti Forest	18
3.2	Running your first quantum program	19
3.3	Basic pyQuil Usage	19
3.3.1	Some Program Construction Features	21
3.3.2	Fixing a Mistaken Instruction	22
3.3.3	The Standard Gate Set	22
3.3.4	Defining New Gates	23
3.4	Advanced Usage	24
3.4.1	Quantum Fourier Transform (QFT)	24
3.4.2	Classical Control Flow	25
3.4.3	Parametric Depolarizing Noise	27
3.4.4	Parametric Programs	27
3.4.5	Pauli Operator Algebra	28
3.5	JobConnections	29

3.6	Exercises	30
3.6.1	Exercise 1 - Quantum Dice	30
3.6.2	Exercise 2 - Controlled Gates	30
3.6.3	Exercise 3 - Grover's Algorithm	30
4	Next Steps	31
5	The Rigetti QVM	33
6	Quantum Processor Unit	35
6.1	Rabi Experiments	36
6.2	T1 Experiments	37
6.3	Ramsey Experiments	39
6.4	pyquil.qpu	41
7	Examples of Quantum Programs on a QVM	43
7.1	Meyer-Penny Game	43
7.2	Exercises	45
7.2.1	Prisoner's Dilemma	45
8	Source Code Docs	47
8.1	pyquil.forest	47
8.2	pyquil.gates	47
8.3	pyquil.job_results	52
8.4	pyquil.paulis	53
8.5	pyquil.parametric	56
8.6	pyquil.plots	57
8.7	pyquil.quil	57
8.8	pyquil.quil_atom	58
8.9	pyquil.quilbase	59
8.10	pyquil.resource_manager	63
8.11	pyquil.slot	64
8.12	pyquil.wavefunction	65
9	Indices and Tables	67
	Python Module Index	69

Welcome to pyQuil!

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

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

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

pyQuil currently allows you to do quantum computing in two ways: 1. Quil programs can be executed on a local or cloud-based Quantum Virtual Machine. This is a classical simulation of a quantum processor that can simulate up to 36 qubits. The default access key allows you to run simulations of up to 25 qubits. These simulations can be run through either synchronous API calls, or through an asynchronous job queue for larger programs. More information about the QVM can be found at [Overview of the Quantum Virtual Machine](#). 2. pyQuil also runs some select single qubit experiments on Rigetti's superconducting quantum processors. This functionality will be expanded over time, and currently allows for Rabi, Ramsey, and T1 experiments to be run on a multi-qubit chip using the `QPUConnection` object. More information about these features are available at [Experimenting with a 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 [Brief Introduction to Quantum Computing](#), where the basics of quantum computing are introduced using Quil and the Quantum Abstract Machine on which it runs.

¹ <https://arxiv.org/abs/1608.03355>

Brief Introduction to Quantum Computing

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

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

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

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

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

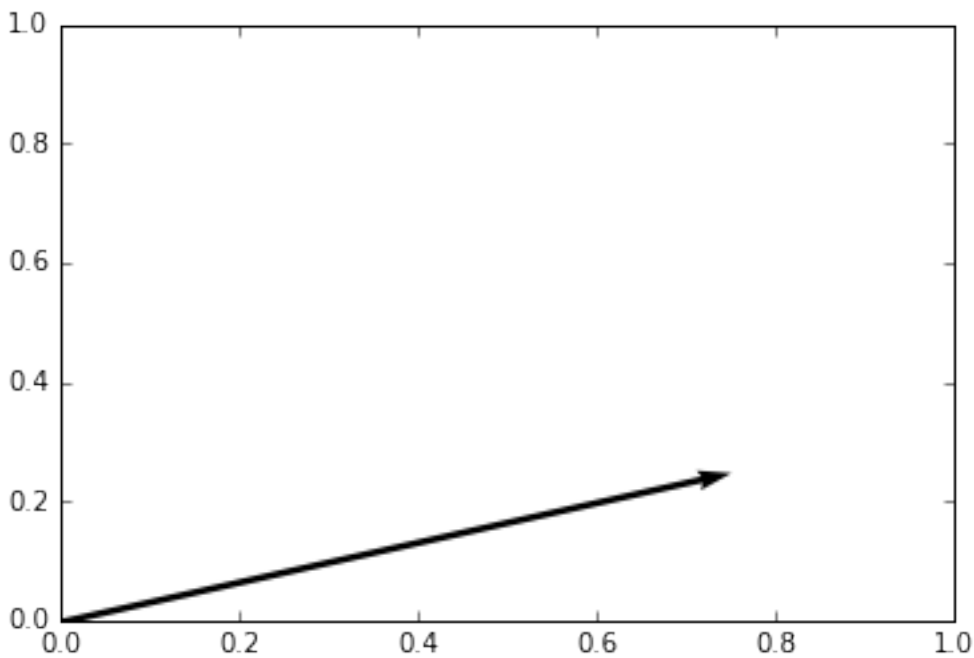
$$\vec{v} = a\vec{0} + b\vec{1},$$

where $\langle a \rangle$ represents the probability of the bit being 0 and $\langle b \rangle$ represents the probability of the bit being 1. This clearly also requires that $\langle a + b = 1 \rangle$. 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:

$$\Pr(0) = \vec{v}^T \cdot \vec{0} = a$$

$$\Pr(1) = \vec{v}^T \cdot \vec{1} = b,$$

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

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

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 $|1_u\rangle$ represents the 1 outcome on coin u . The **combined system** of the two coins has four possible outcomes $\{|0_u0_v\rangle, |0_u1_v\rangle, |1_u0_v\rangle, |1_u1_v\rangle\}$ 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 $\frac{1}{\sqrt{2}}(|0_u0_v\rangle + |1_u1_v\rangle)$.) 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

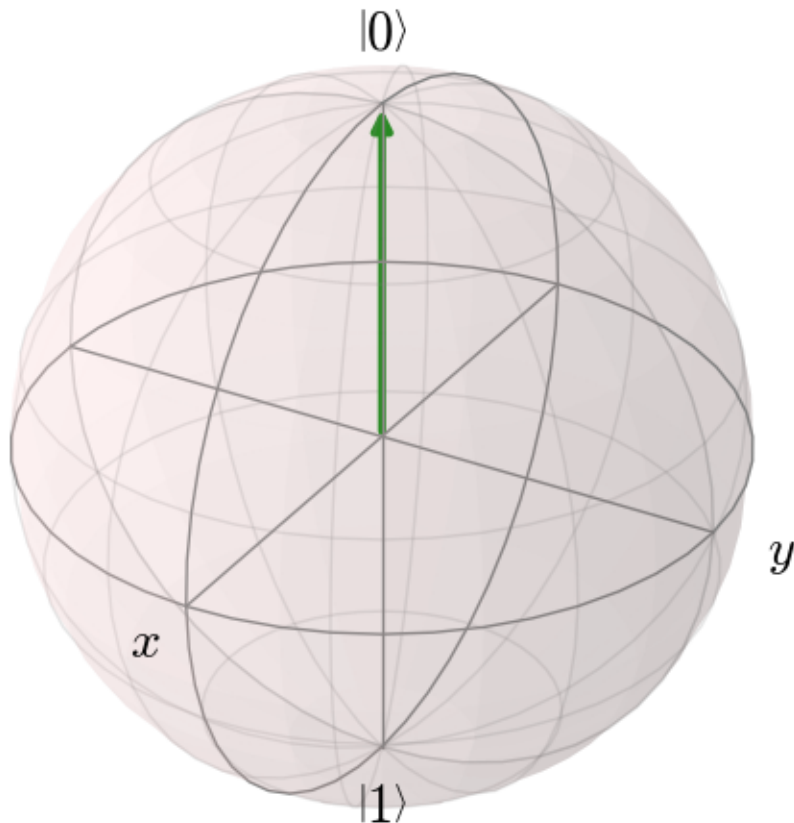
$$\alpha|0\rangle + \beta|1\rangle.$$

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
import pyquil.api as api
quantum_simulator = api.SyncConnection()

# pyQuil is based around operations (or gates) so we will start with the most
# basic one: the identity operation, called I. I takes one argument, the index
# of the qubit that it should be applied to.
from pyquil.gates import I

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

# Quantum states are called wavefunctions for historical reasons.
# We can run this basic program on our connection to the simulator.
# This call will return the state of our qubits after we run program p.
# This api call returns a tuple, but we'll ignore the second value for now.
wavefunc, _ = quantum_simulator.wavefunction(p)
# wavefunc is a Wavefunction object that stores a quantum state as a list of
↳ amplitudes
alpha, beta = wavefunc
print "Our qubit is in the state alpha={} and beta={}".format(alpha, beta)
print "The probability of measuring the qubit in outcome 0 is {}".
↳ format(abs(alpha)**2)
print "The probability of measuring the qubit in outcome 1 is {}".format(abs(beta)**2)
```

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

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

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

wavefunc, _ = quantum_simulator.wavefunction(p)
alpha, beta = wavefunc
print "Our qubit is in the state alpha={} and beta={}".format(alpha, beta)
print "The probability of measuring the qubit in outcome 0 is {}".
↳ format(abs(alpha)**2)
print "The probability of measuring the qubit in outcome 1 is {}".format(abs(beta)**2)
```

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

In this case we have flipped the probability of outcome 0 into the probability of outcome 1 for our qubit. We can also

investigate what happens to the state of multiple qubits. We'd expect the state of multiple qubits to grow exponentially in size, as their vectors are tensored together.

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

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

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

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

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

```
# wavefunction(Program) returns a coefficient array that corresponds to outcomes in_
→the following order
print quantum_simulator.bit_string_probabilities(Program(I(0), I(1)))
```

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

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) [0]
```

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

Pauli Operators

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

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

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

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

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

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

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

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

Quantum programs are built by applying successive gate operations:

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

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

Multi-Qubit Operations

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

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

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

```
from pyquil.gates import CNOT

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

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

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

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

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

```

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

print "SWAP|01> = ", quantum_simulator.wavefunction(p)[0]
print "With outcome probabilities\n", quantum_simulator.bit_string_probabilities(p)

```

```

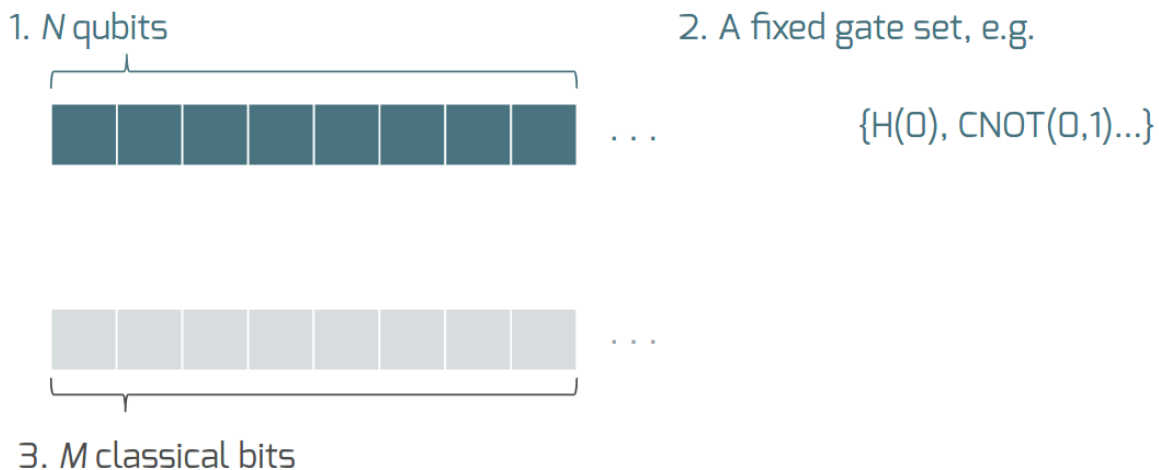
SWAP|01> = [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j]
With outcome probabilities
{'11': 0.0, '01': 0.0, '00': 0.0, '10': 1.0}

```

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

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]
quantum_simulator.run(p, classical_regs)
```

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

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

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

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

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

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

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

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

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

Classical/Quantum Interaction

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

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

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

```
from pyquil.gates import H

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



```

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

```

```

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

```

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

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

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

```

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

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

```

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

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

```

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

```

```

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

```

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

```

# This happens with bigger systems too
classical_reg = 0

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

```

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

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

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

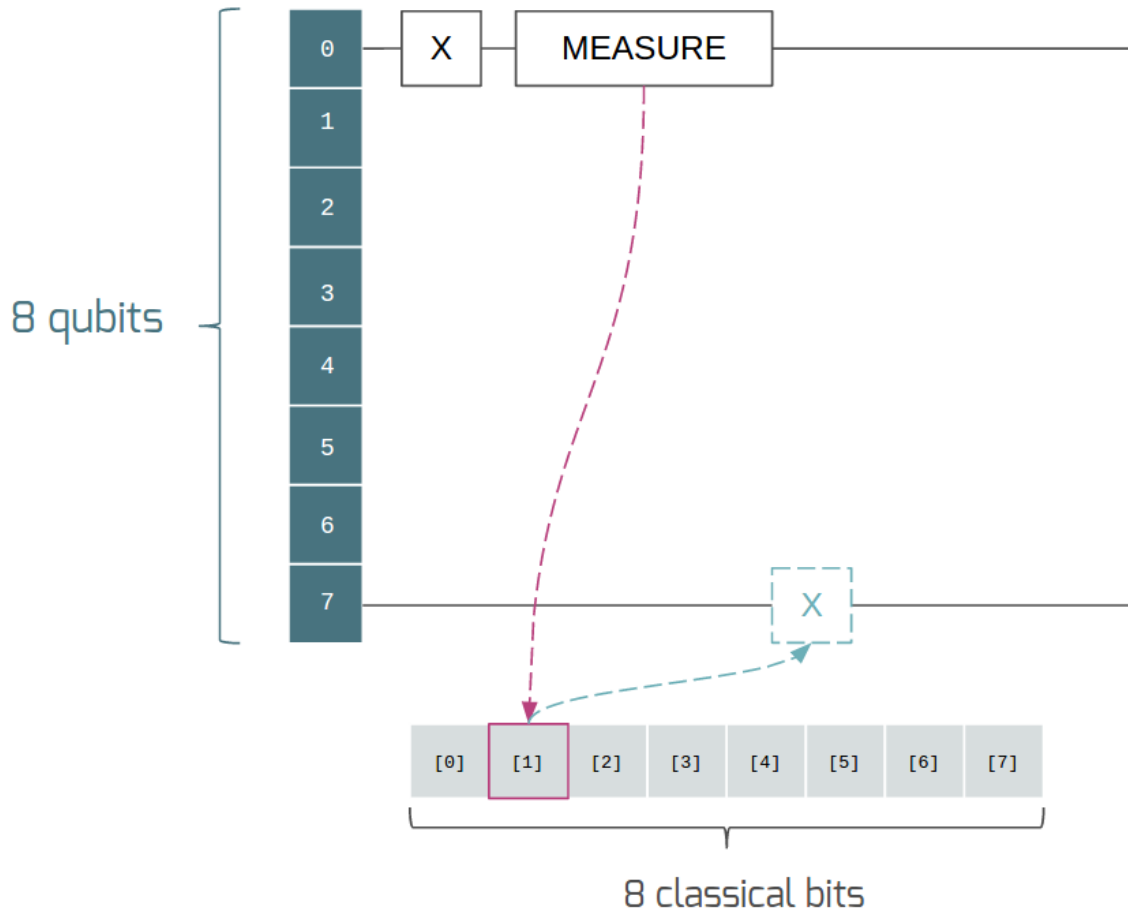
Classical Control

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

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

```
[[1]]
```

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



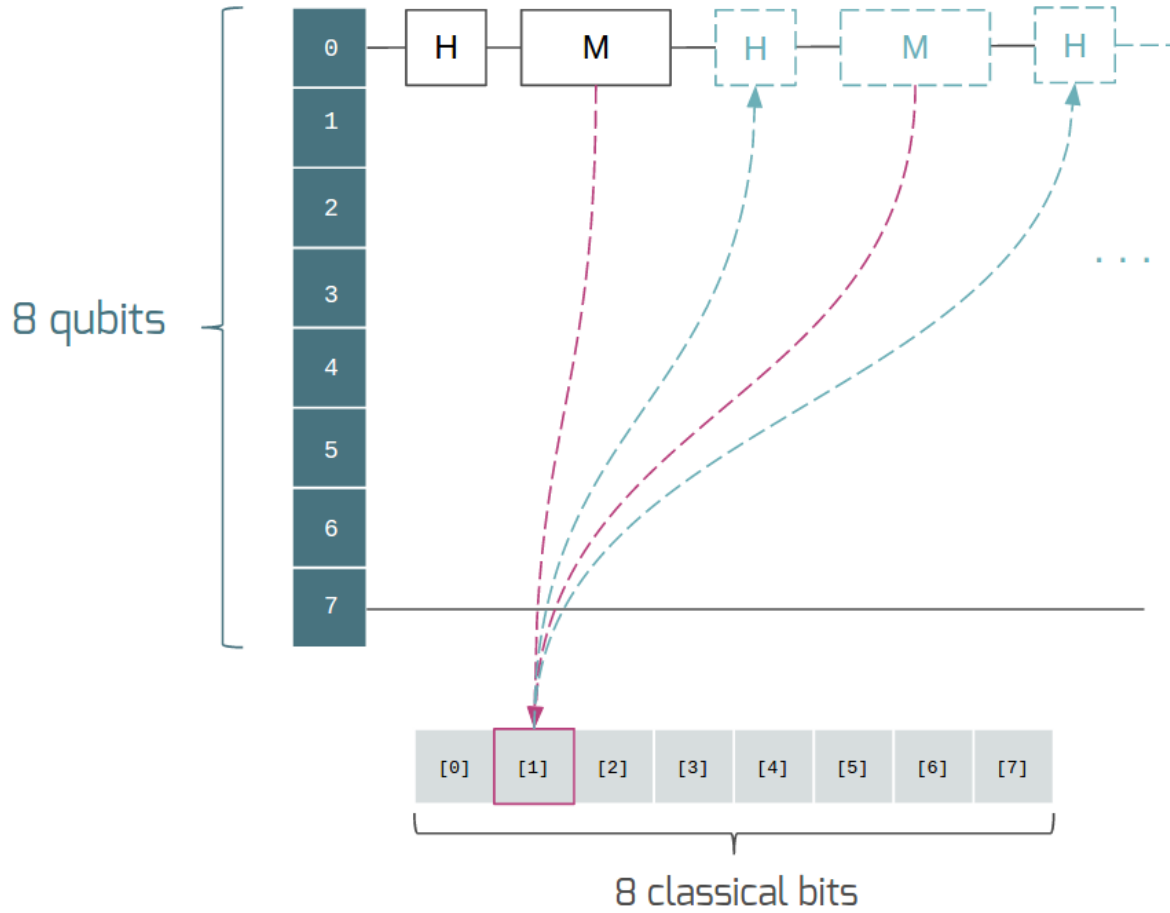
Example: The Probabilistic Halting Problem

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

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

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

```
[[0]]
```



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

Installation and Getting Started

This toolkit provides some simple libraries for writing quantum programs.

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

It comes with a few parts:

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

Environment Setup

Prerequisites

Before you can start writing quantum programs, you will need Python 2.7 (version 2.7.10 or greater) or Python 3.6 and the Python package manager `pip`. We recommend installing [Anaconda](#) for an all-in-one installation of Python (2.7). If you don't have `pip`, it can be installed with `easy_install pip`.

Installation

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

```
pip install pyquil
```

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

```
pip install -e .
```

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

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

This will also install `pyQuil`'s dependencies (`requests >= 2.4.2` and `NumPy >= 1.10`) if you do not already have them.

The library will now be available globally.

Connecting to the Rigetti Forest

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

Once you have your key, you need to set up configuration in the file `.pyquil_config` which `pyQuil` will attempt to find in your home directory by default. You can change this location by setting the environment variable `PYQUIL_CONFIG` to the path of the file.

Note: When setting the environment variable make sure to use the full path or include the `HOME` variable. E.g. `export PYQUIL_CONFIG=$HOME/<CONFIG_NAME>`

Loading the `pyquil.forest` module will print a warning if this is not found. The configuration file is in INI format and should contain all the information required to connect to Forest:

```
[Rigetti Forest]
url: <URL to Rigetti Forest or QVM endpoint>
key: <Rigetti Forest API key>
```

Look [here](#) to learn more about the Forest toolkit.

If `url` is not set, `pyQuil` will default to looking for a local endpoint at `127.0.0.1:5000`.

Alternatively, connection information can be provided in environment variables.

```
export QVM_URL=<URL to Rigetti Forest or QVM endpoint>
export QVM_API_KEY=<Rigetti Forest API key>
export QVM_USER_ID=<Rigetti User ID>
```

Endpoints

There are two important endpoints to keep in mind. You will use different ones for different types of jobs.

`https://api.rigetti.com/qvm` is used for making synchronous calls to the QVM. You should use this for most of the getting started materials unless otherwise instructed.

`https://job.rigetti.com/beta` is used for large async QVM jobs or for running jobs on a QPU.

Running your first quantum program

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

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

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

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

Basic pyQuil Usage

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

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

Next, we want to open a connection to the QVM. Forest supports two types of connections through pyQuil. The first is a synchronous connection that immediately runs requested jobs against the QVM. This will time out on longer jobs that run for more than 30 seconds. Synchronous connections are good for experimenting interactively as they give quick feedback.

```
# open a synchronous connection
qvm = api.SyncConnection()
```

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

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

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

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

```
print(p)
```

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

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

```
classical_regs = [0] # A list of which classical registers to return the values of.
qvm.run(p, classical_regs)
```

```
[[1]]
```

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

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

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

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

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

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

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

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

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

```
(<pyquil.wavefunction.Wavefunction at 0x1088a2c10>, [])
```

The first element in the returned tuple 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 = pq.Program().inst(H(0))
wvf, _ = qvm.wavefunction(coin_flip)
print(wvf)
```

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

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

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

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

The second element returned from a wavefunction call is an optional amount of classical memory to check:

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

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.SyncConnection(random_seed=17)
print(seeded_cxn.run(pq.Program(H(0)).measure(0, 0), [0], 20))

seeded_cxn = api.SyncConnection(random_seed=17)
# This will give identical output to the above
print(seeded_cxn.run(pq.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.

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(pq.Program().inst(X(0), Y(1), Z(0)).measure(0, 1))

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

print("A mix of chained inst and measures:")
print(pq.Program().inst(X(0)).measure(0, 1).inst(Y(1), X(0)).measure(0, 0))

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

Fixing a Mistaken Instruction

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

```
p = pq.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 += pq.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
```

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: 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.

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 = pq.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])
```

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

Quil in general supports defining parametric gates, though right now only static gates are supported by pyQuil. Below we show how we can define $X_0 \otimes \sqrt{X}_1$ as a single gate.

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

# Then we can use the new gate
p.inst(("X-SQRT-X", 0, 1))
wavf, _ = qvm.wavefunction(p)
print(wavf)
```

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

Advanced Usage

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

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

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

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

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

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

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

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

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 = pq.Program(TRUE([classical_flag_register]))
loop_body = pq.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 = pq.Program(X(0))
else_branch = pq.Program()

# Make a program that will put a 0 or 1 in test_register with 50% probability:
branching_prog = pq.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]]
```

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.SyncConnection(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 = pq.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]]
```

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 pq.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.

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.

JobConnections

Larger pyQuil programs can take longer than 30 seconds to run. These jobs can be posted into the cloud job queue using a different connection object. The mode of interaction with the API is asynchronous. This means that there is a separate query to post a job and to get the result.

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

job_qvm = JobConnection(endpoint="https://job.rigetti.com/beta")
res = job_qvm.run(Program(X(0)).measure(0, 0), [0])
```

The `res` is an instance of a `JobResult` object. It has an `id` and allows you to make queries to see if the job result is finished.

```
zz = res.get()
print type(zz), zz
```

```
<class 'pyquil.job_results.JobResult'> {u'status': u'Submitted', u'jobId': u'
↪ 'BLSLJCBGNP'}
```

`is_done` updates the `JobResult` object once, and returns `True` if the job has completed. Once the job is finished, then the results can be retrieved from the `JobResult` object:

```
import time

while not res.is_done():
    time.sleep(1)
print res
answer = res.decode()
print answer
```

```
{u'result': u'[[1]]', u'jobId': u'BLSLJCBGNP'}

<type 'list'> [[1]]
```

This same pattern applies to the `wavefunction`, `expectation`, and `run_and_measure` calls on the `JobConnection` object.

Exercises

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
```

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.

Exercise 3 - Grover's Algorithm

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

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

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

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

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

CHAPTER 4

Next Steps

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

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

The Rigetti QVM

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

Most API keys give access to the a QVM with 25 qubits, and the current QVM can currently run up to 36 qubits. If you would like access to more qubits or help running larger jobs, then contact us at softapps@rigetti.com.

¹ <https://arxiv.org/abs/1608.03355>

Quantum Processor Unit

pyQuil allows some basic single-qubit experiments to be run on a multi-qubit superconducting quantum processor. These three types of experiments are some of the basic building blocks for calibrating qubits. This documentation will cover the basics of each experiment, as well as show you how to run them using pyQuil.

Note: In order to run experiments on the QPU you will need an upgraded API key. If you are interested in running these experiments, then email us at support@rigetti.com.

For QPU plots, please install the `lmfit>=0.9.7` package.

Once your API key has been upgraded, update your environment to use the QPU endpoint:

```
export QVM_URL=https://job.rigetti.com/beta
```

Before running any experiments, we need to take a look at what devices are available on the platform.

```
from pyquil.gpu import QPUConnection, get_info
print get_info()
```

```
{u'devices': [{u'name': u'Z12-13-C4a2',
  u'qubits': [{u'num': 5,
    u'rabi_params': {u'start': 0.01,
      u'step': 0.02,
      u'stop': 0.5,
      u'time': 160.0},
    u'ramsey_params': {u'detuning': 0.5,
      u'start': 0.01,
      u'step': 0.2,
      u'stop': 20},
    u'ssr_fidelity': 0.923,
    u't1': 2e-05,
    u't1_params': {u'num_pts': 25, u'start': 0.01, u'stop': 40},
    u't2': 1.5e-05},
    {u'num': 6,
```

```
u'rabi_params': {u'start': 0.01,
                 u'step': 0.02,
                 u'stop': 0.5,
                 u'time': 100.0},
u'ramsey_params': {u'detuning': 0.5,
                  u'start': 0.01,
                  u'step': 0.2,
                  u'stop': 20},
u'ssr_fidelity': 0.923,
u't1': 2.1e-05,
u't1_params': {u'num_pts': 30, u'start': 0.01, u'stop': 40},
u't2': 1.5e-05}}}]}}
```

This JSON provides a list of available devices by their name. In this example we have one device, called Z12-13-C4a2 which has two qubits (indexed as qubit numbers 5 and 6) configured and available for experiments. This configuration information also returns details about the coherence times t_1 and t_2 and the measurement fidelity `ssr_fidelity` for each qubit. It also shows information about the parameters for each of the experiments on each qubit.

Next we open up a connection to the QPU for the available device.

```
qpu = QPUConnection("Z12-13-C4a2")
qpu.ping() # checks to make sure the connection is good
```

```
'ok'
```

Rabi Experiments

A Rabi experiment runs a series of Quil programs. Each program is parameterized by a rotation angle:

```
DEFCIRCUIT RABI(%angle) q:
  RX(%angle) q
  MEASURE q [q]
```

In our hardware, the angle in the RX gate is implemented by (roughly) scaling the amplitude of a microwave pulse. A larger amplitude corresponds to a larger rotation angle. A Rabi experiment, will run the RABI program for a series of different amplitudes for the RX pulse.

Here is how to run a Rabi experiment:

```
my_qubit = 5
res = qpu.rabi(my_qubit)
print type(res), res
```

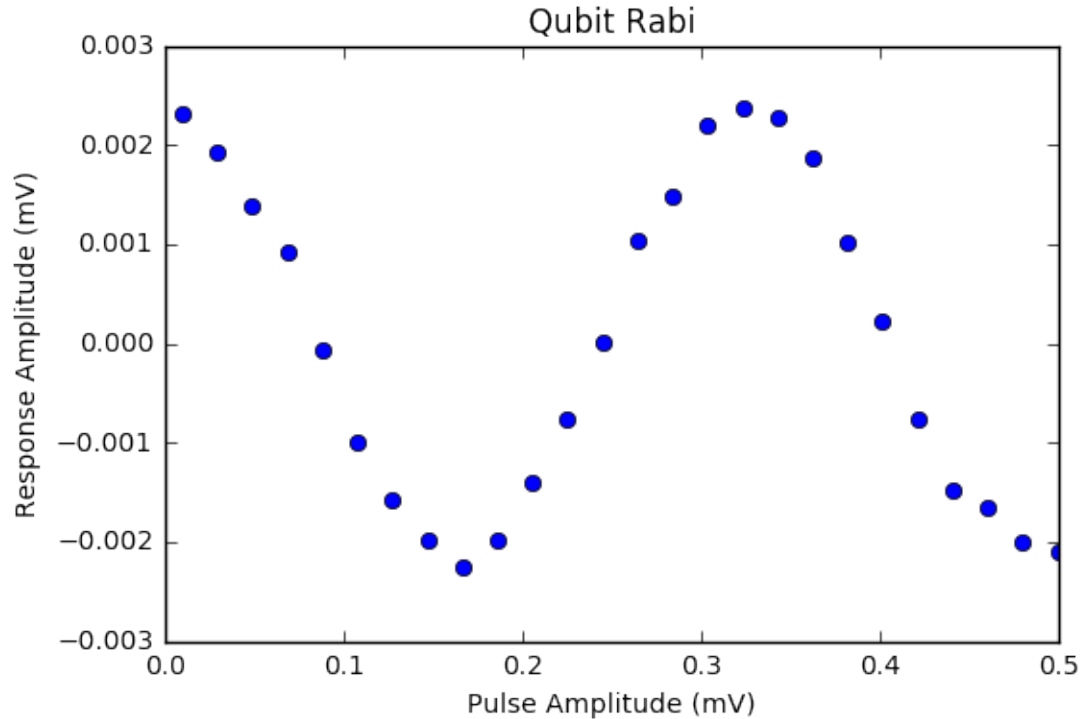
```
<class 'pyquil.job_results.RabiResult'> {u'status': u'Submitted', u'jobId': u
↪ 'BLSLJCBGNP' }
```

Just like in the JobConnection example for working with the QVM, experiments on a QPU work through the jobqueue pattern. When a job is completed, we can use a built in method to plot the results

```
from pyquil.job_results import wait_for_job
from pyquil.plots import analog_plot
wait_for_job(res) # blocks execution until the job is completed
print res.result
analog_plot(res)
```



```
{u'result': u'[[0.01, 0.0296, 0.0492, 0.0688, ... ,
-0.0007563137318034532, -0.0014812487258596086, -0.0016550719404520116, -0.
↪0020105023806951602,
-0.0021007405399624955]]', u'jobId': u'LZLQZTAVPK'}
```



This plot is showing real analog data from an experiment on a superconducting qubit. The x-axis is the amplitude of the RX pulse that is sent to the qubit, in millivolts. The y-axis is a response function from a measurement of that qubit that corresponds to the qubits state, as measured in the computational basis.

Since varying the amplitude of the RX pulse changes the rotation angle of the qubit around the x-axis on the Bloch sphere, we can expect to see a periodic change in the state of qubit. It starts in the ground state at $RX(0)$ and then moves to the excited state when $RX(\pi)$. In this example we see that an applied pulse amplitude of about 0.18mV corresponds to an $RX(\pi)$ rotation. This also tells us that a fitted response of about -0.0024 corresponds to the excited state. In this way we use experiments like this one, and the others given here, to figure out what physical signals corresponds to computational operations on our qubits.

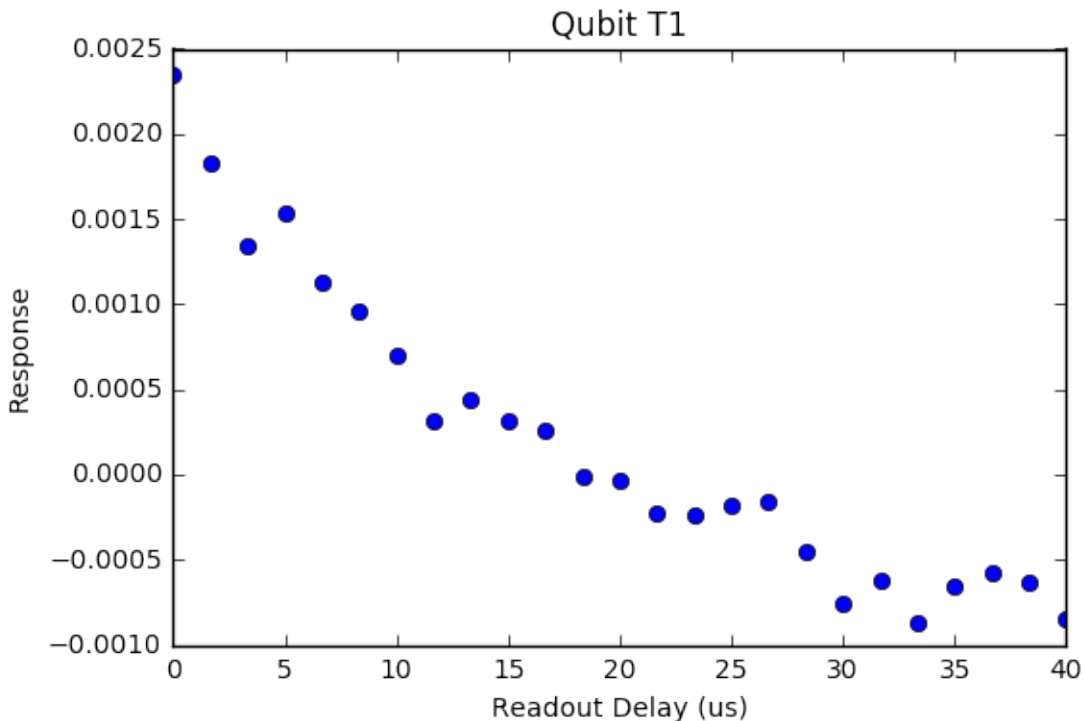
T1 Experiments

T1 experiments measure the t1 coherence time of qubits. A single run of a T1 experiment is an X gate followed by a wait time, followed by a measurement. Sweeping this wait time over many runs gives a T1 experiment. Since the X pulse puts the qubit in the excited state, sweeping over the wait time gives us a sense of how likely a qubit it to remain in the excited state over time. The likelihood of the qubit staying in the excited state typically decays exponentially, and the decay constant of this exponent is called the T1 coherence time.

You can run a T1 experiment on our qubits to check their coherence times.

```
my_qubit = 5
res_t1 = qpu.t1(my_qubit)
```

```
wait_for_job(res_t1)
analog_plot(res_t1)
```



In this example we can see an exponential decay of the excited qubit. If we fit this data to an exponential then we can extract what T1 decay we have measured:

```
from pyquil.plots import T1DecayModel
import numpy as np

model = T1DecayModel()
x, y = res_t1.decode()

fit_n_data = model.report_fit(np.asarray(x), np.asarray(y))
fit, data = (fit_n_data[0], fit_n_data[1:])
print fit.fit_report()
```

```
[[Model]]
Model(fn_T1_decay)
[[Fit Statistics]]
  # function evals  = 27
  # data points    = 25
  # variables      = 3
  chi-square       = 0.000
  reduced chi-square = 0.000
  Akaike info crit = -440.113
  Bayesian info crit = -436.456
[[Variables]]
  baseline:  -0.00106788 +/- 0.000145 (13.56%) (init=-0.0008674491)
  amplitude: 0.00326745 +/- 0.000129 (3.95%) (init= 0.00321364)
  T1:        1.6752e-05 +/- 1.97e-06 (11.76%) (init= 5.33208e-06)
[[Correlations]] (unreported correlations are < 0.100)
```

```
C(baseline, T1)           = -0.938
C(baseline, amplitude)   = -0.759
C(amplitude, T1)         = 0.555
```

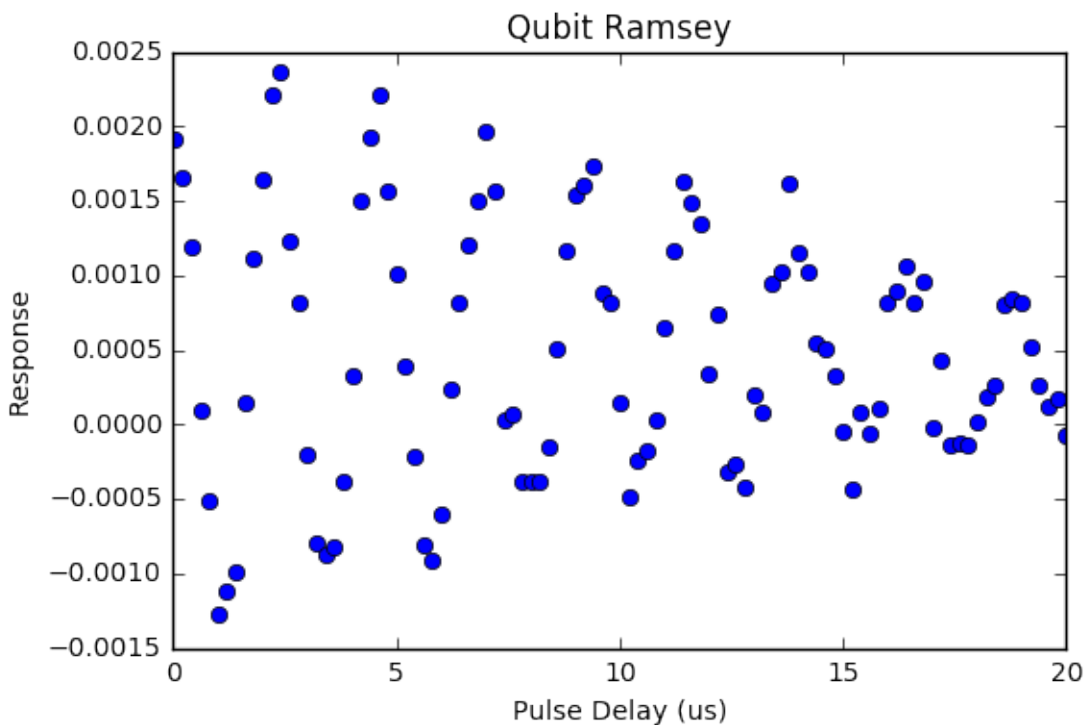
This example showed a T1 of about 16.7 microseconds.

Ramsey Experiments

Ramsey experiments measure a different kind of decoherence. While T1 experiments measure the loss of information along the north-south axis of the Bloch sphere, Ramsey experiments measure the loss of information around the axis. This type of decoherence is called dephasing and is captured in the T2 coherence time of qubits.

A single run of the experiment is a π -HALF pulse, followed by a wait time, followed by another π -HALF pulse and a measurement. Sweeping the wait time over many runs gives a Ramsey experiment. The first pulse puts the qubit into a state on the equator of the Bloch sphere. Waiting then allows the state to dephase and the second pulse rotates the state back towards the north-south Bloch sphere axis. If dephasing has occurred, then this will be represented by a decrease in amplitude in the resulting periodic function.

```
my_qubit = 5
ramsey_res = qpu.ramsey(my_qubit)
wait_for_job(ramsey_res)
analog_plot(ramsey_res)
```



If we fit this data to the a decaying periodic function, then we can extract the T2 decay constant.

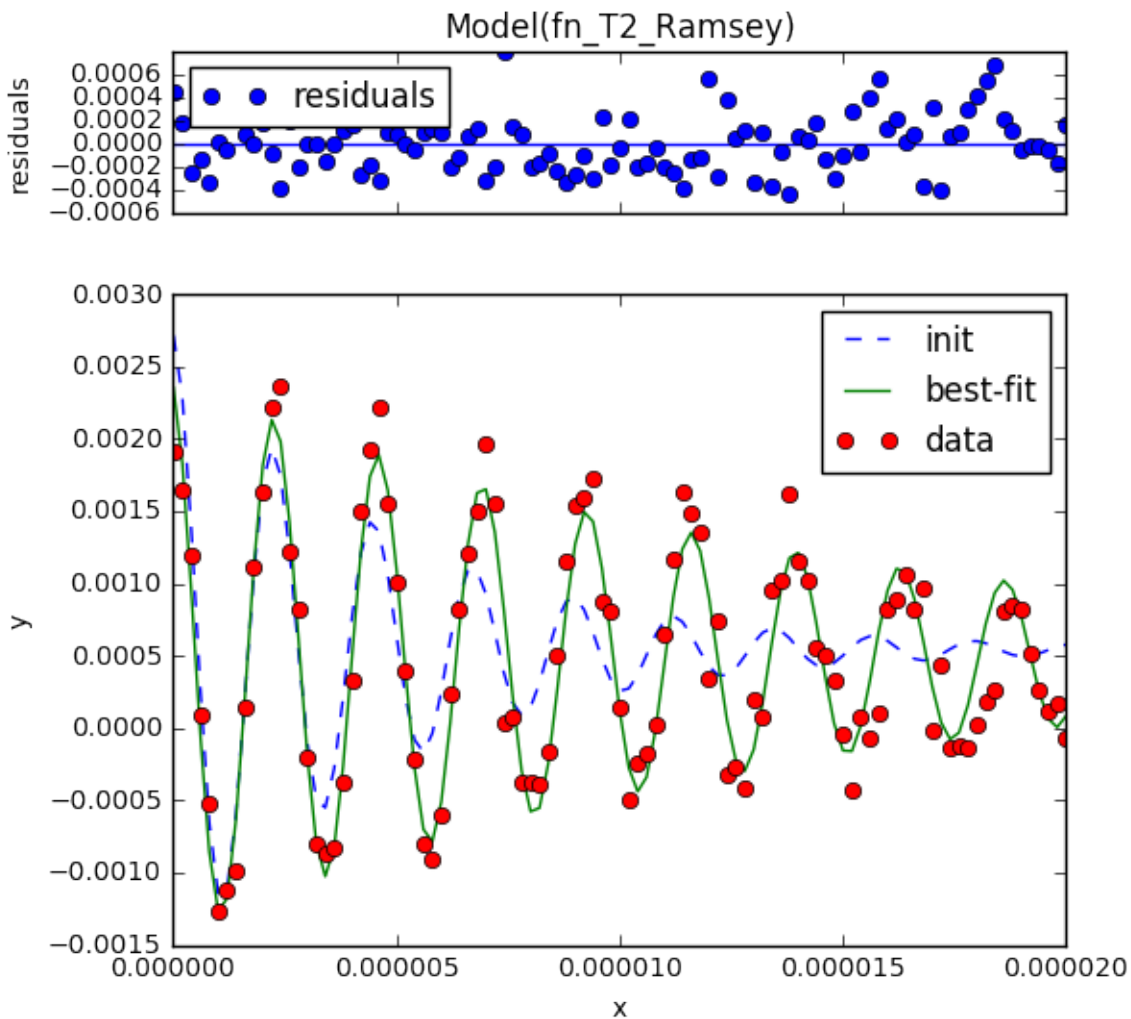
```
from pyquil.plots import T2RamseyModel

model = T2RamseyModel()
x, y = res_ramsey.decode()
```

```
fit_n_data = model.report_fit(np.asarray(x), np.asarray(y))
fit, data = (fit_n_data[0], fit_n_data[1:])
fit.plot()
plt.show()
print fit.fit_report()
```

Note: if you are using a python terminal instead of a notebook, then plot using the following instead:

```
fig = fit.plot()
fig[0].show()
print fit.fit_report()
```



```
[[Model]]
  Model(fn_T2_Ramsey)
[[Fit Statistics]]
  # function evals  = 52
  # data points    = 101
  # variables      = 5
  chi-square       = 0.000
  reduced chi-square = 0.000
  Akaike info crit = -1666.553
```

```

Bayesian info crit = -1653.477
[[Variables]]
baseline: 0.00049312 +/- 2.54e-05 (5.16%) (init= 0.0005435569)
amplitude: 0.00192550 +/- 9.55e-05 (4.96%) (init= 0.002179158)
T2: 1.4413e-05 +/- 1.35e-06 (9.36%) (init= 5e-06)
detuning: 4.2761e+05 +/- 1.05e+03 (0.25%) (init= 445767.4)
x0: -8.4993e-08 +/- 1.88e-08 (22.16%) (init= 0)
[[Correlations]] (unreported correlations are < 0.100)
C(detuning, x0) = 0.773
C(amplitude, T2) = -0.764

```

From this we can extract that the T2 decoherence for this qubit is about 14.4 microseconds.

pyquil.qpu

exception `pyquil.qpu.NoParametersFoundException`

Bases: `exceptions.Exception`

class `pyquil.qpu.QPUConnection` (*device_name*, *args)

Bases: `pyquil.api.JobConnection`

bit_string_probabilities (*quil_program*)

expectation (*prep_prog*, *operator_programs*=[<pyquil.kit.Program object>])

get_qubits ()

Returns A list of active qubit ids on this device.

Return type list

rabi (*qubit_id*)

Runs a Rabi experiment on the given qubit

Parameters `qubit_id` (*int*) –

Returns A RabiResult object

ramsey (*qubit_id*)

Runs a Ramsey experiment on the given qubit

Parameters `qubit_id` (*int*) –

Returns A RamseyResult object

run (*quil_program*, *classical_addresses*, *trials*=1)

run_and_measure (*quil_program*, *qubits*, *trials*=1)

t1 (*qubit_id*)

Runs a T1 experiment on the given qubit

Parameters `qubit_id` (*int*) –

Returns A T1Result object

wavefunction (*quil_program*, *classical_addresses*=[])

`pyquil.qpu.get_info` ()

Gets information about what devices are currently available through the Forest API. :return: A JSON dictionary with configuration information.

`pyquil.gpu.get_params` (*device_name, qcid, func*)

Get and parse the configuration information from the Forest API. :param str device_name: The device to get info for :param int qcid: The qubit number to look at :param func: A function to apply to the qubit specific JSON dictionary of config info :return:

`pyquil.gpu.get_rabi_params` (*device_name, qcid*)

Gets the current Rabi experiment parameters for a specific qubit on a specific device :param str device_name: :param int qcid: :return: A dictionary with the parameter info :rtype: dict

`pyquil.gpu.get_ramsey_params` (*device_name, qcid*)

Gets the current Ramsey experiment parameters for a specific qubit on a specific device :param str device_name: :param int qcid: :return: A dictionary with the parameter info :rtype: dict

`pyquil.gpu.get_t1_params` (*device_name, qcid*)

Gets the current T1 experiment parameters for a specific qubit on a specific device :param str device_name: :param int qcid: :return: A dictionary with the parameter info :rtype: dict

Examples of Quantum Programs on a QVM

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

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

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.

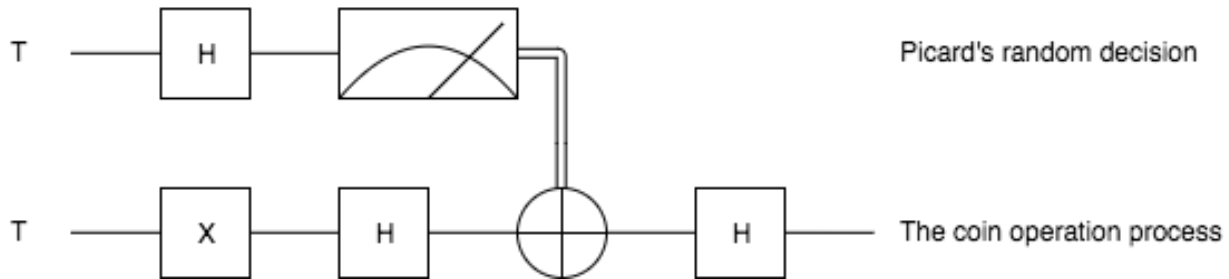
The rules: Picard is to place a penny Heads up into an opaque box. Then Picard and Q take turns to flip or not flip the penny without being able to see it; first Q then P then Q again. After this the penny is revealed; Q wins if it shows Heads (H), while Tails (T) makes Picard the winner.

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 show 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:

```
import pyquil.quil as pq
from pyquil import forest

from pyquil.gates import I, H, X
qvm = forest.Connection()
```

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 = pq.Program(X(0))
else_branch = pq.Program(I(0))
```

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

```
prog = (pq.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, 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?

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

pyquil.forest

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*)

Produces a CCNOT instruction. This gate applies to three qubit arguments to produce the controlled-controlled-not gate instruction.

Parameters

- **control-1** – The first control qubit.
- **control-2** – The second control qubit.
- **target** – The target qubit. The target qubit has an X-gate applied to it if both control qubits are in the excited state.

Returns A `Gate` object.

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

Produces a CNOT instruction. This gate applies to two qubit arguments to produce the controlled-not gate instruction.

Parameters

- **control** – The control qubit.
- **target** – The target qubit. The target qubit has an X-gate applied to it if the control qubit is in the excited state.

Returns A Gate object.

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

Produces a CPHASE00 instruction. This gate applies to two qubit arguments to produce one of the controlled phase instructions.

Parameters

- **angle** – The input phase angle to apply when both qubits are in the excited state.
- **q1** – Qubit 1.
- **q2** – Qubit 2.

Returns A Gate object.

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

Produces a CPHASE00 instruction. This gate applies to two qubit arguments to produce one of the controlled phase instructions.

Parameters

- **angle** – The input phase angle to apply when both qubits are in the ground state.
- **q1** – Qubit 1.
- **q2** – Qubit 2.

Returns A Gate object.

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

Produces a CPHASE01 instruction. This gate applies to two qubit arguments to produce one of the controlled phase instructions.

Parameters

- **angle** – The input phase angle to apply when q1 is in the excited state and q2 is in the ground state.
- **q1** – Qubit 1.
- **q2** – Qubit 2.

Returns A Gate object.

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

Produces a CPHASE10 instruction. This gate applies to two qubit arguments to produce one of the controlled phase instructions.

Parameters

- **angle** – The input phase angle to apply when q2 is in the excited state and q1 is in the ground state.
- **q1** – Qubit 1.

- **q2** – Qubit 2.

Returns A Gate object.

`pyquil.gates.CSWAP` (*qubits)

Produces a SWAP instruction. This gate swaps the state of two qubits.

Parameters

- **q1** – Qubit 1.
- **q2** – Qubit 2.

Returns A Gate object.

`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)

Produces the H instruction. This gate is a single qubit Hadamard gate.

Parameters **qubit** – The qubit apply the gate to.

Returns A Gate object.

`pyquil.gates.HALT` = <pyquil.quilbase.Halt object>

This instruction ends the program.

Returns A Halt object.

`pyquil.gates.I` (*qubits)

Produces the I instruction. This gate is a single qubit identity gate. Note that this gate is different that the NOP instruction as noise channels are typically still applied during the duration of identity gates. Identities will also block parallelization like any other gate.

Parameters **qubit** – The qubit apply the gate to.

Returns A Gate object.

`pyquil.gates.ISWAP` (*qubits)

Produces an ISWAP instruction. This gate swaps the state of two qubits, applying a -i phase to q1 when it is in the excited state and a -i phase to q2 when it is in the ground state.

Parameters

- **q1** – Qubit 1.
- **q2** – Qubit 2.

Returns A Gate object.

`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.NOP` = `<pyquil.quilbase.Nop object>`
This instruction applies no operation at that timestep. Typically these are ignored in error-models.

Returns A Nop object.

`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*)
Produces a PHASE instruction. This is the same as the RZ gate.

Parameters

- **angle** – The angle to rotate around the z-axis on the bloch sphere.
- **qubit** – The qubit apply the gate to.

Returns A Gate object.

`pyquil.gates.PSWAP` (**params*)
Produces a PSWAP instruction. This is a parameterized swap gate.

Parameters

- **angle** – The angle of the phase to apply to the swapped states. This phase is applied to q1 when it is in the excited state and to q2 when it is in the ground state.
- **q1** – Qubit 1.
- **q2** – Qubit 2.

Returns A Gate object.

`pyquil.gates.RESET` = `<pyquil.quilbase.Reset object>`
This instruction resets all the qubits to the ground state.

Returns A Reset object.

`pyquil.gates.RX(*params)`

Produces the RX instruction. This gate is a single qubit X-rotation.

Parameters

- **angle** – The angle to rotate around the x-axis on the bloch sphere.
- **qubit** – The qubit apply the gate to.

Returns A Gate object.

`pyquil.gates.RY(*params)`

Produces the RY instruction. This gate is a single qubit Y-rotation.

Parameters

- **angle** – The angle to rotate around the y-axis on the bloch sphere.
- **qubit** – The qubit apply the gate to.

Returns A Gate object.

`pyquil.gates.RZ(*params)`

Produces the RZ instruction. This gate is a single qubit Z-rotation.

Parameters

- **angle** – The angle to rotate around the z-axis on the bloch sphere.
- **qubit** – The qubit apply the gate to.

Returns A Gate object.

`pyquil.gates.S(*qubits)`

Produces the S instruction. This gate is a single qubit S-gate.

Parameters **qubit** – The qubit apply the gate to.

Returns A Gate object.

`pyquil.gates.STANDARD_GATES = {'CPHASE': <function gate_function>, 'CPHASE00': <function gate_function>, 'PSWAP': <function gate_function>}`

Dictionary of standard gates. Keys are gate names, values are gate functions.

`pyquil.gates.SWAP(*qubits)`

Produces a SWAP instruction. This gate swaps the state of two qubits.

Parameters

- **q1** – Qubit 1.
- **q2** – Qubit 2.

Returns A Gate object.

`pyquil.gates.T(*qubits)`

Produces the T instruction. This gate is a single qubit T-gate. It is the same as RZ(pi/4).

Parameters **qubit** – The qubit apply the gate to.

Returns A Gate object.

`pyquil.gates.TRUE(classical_reg)`

Produce a TRUE instruction.

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

Returns A ClassicalTrue instance.

`pyquil.gates.WAIT = <pyquil.quilbase.Wait object>`

This instruction tells the quantum computation to halt. Typically these is used while classical memory is being manipulated by a CPU in a hybrid classical/quantum algorithm.

Returns A Wait object.

`pyquil.gates.X(*qubits)`

Produces the X instruction. This gate is a single qubit X-gate.

Parameters `qubit` – The qubit apply the gate to.

Returns A Gate object.

`pyquil.gates.Y(*qubits)`

Produces the Y instruction. This gate is a single qubit Y-gate.

Parameters `qubit` – The qubit apply the gate to.

Returns A Gate object.

`pyquil.gates.Z(*qubits)`

Produces the Z instruction. This gate is a single qubit Z-gate.

Parameters `qubit` – The qubit apply the gate to.

Returns A Gate object.

`pyquil.gates.unpack_classical_reg(c)`

Get the address for a classical register.

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

Returns The address as an Addr.

`pyquil.gates.unpack_qubit(qubit)`

Get a qubit from an object.

Parameters `qubit` – An int or AbstractQubit.

Returns An AbstractQubit instance

pyquil.job_results

`class pyquil.job_results.JobResult(qpu, success, result=None, payload=None)`

Bases: `object`

`decode()`

Decodes the result of the job. :return: Depends on the type of job. A JSON object.

`get()`

Gets an update from the Forest API on the status of this job. :return: The JobResult object with fields updated. :rtype: JobResult

`is_done()`

Polls Forest API for status, then returns updated status.

Returns Returns True if the Job is completed and False otherwise.

Return type `bool`

`job_id()`

Returns Returns the id of this job

Return type str

classmethod `load_res` (*qpu, response*)

class `pyquil.job_results.RabiResult` (*qpu, success, result=None, payload=None*)
Bases: `pyquil.job_results.JobResult`

class `pyquil.job_results.RamseyResult` (*qpu, success, result=None, payload=None*)
Bases: `pyquil.job_results.JobResult`

class `pyquil.job_results.T1Result` (*qpu, success, result=None, payload=None*)
Bases: `pyquil.job_results.JobResult`

class `pyquil.job_results.WavefunctionResult` (*qpu, success, result=None, payload=None*)
Bases: `pyquil.job_results.JobResult`

decode ()

`pyquil.job_results.recover_complexes` (*coef_string, classical_addresses*)

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

`pyquil.job_results.wait_for_job` (*res, ping_time=0.5*)
Blocks execution and waits for an async Forest Job to complete.

Parameters

- **res** (`JobResult`) – The `JobResult` object to wait for.
- **ping_time** – The interval (in seconds) at which to ping the server.

Returns The completed `JobResult`

pyquil.paulis

Module for working with Pauli algebras.

`pyquil.paulis.ID = <pyquil.paulis.PauliTerm object>`
The identity Pauli operator on the 0th qubit.

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

A sum of one or more `PauliTerms`.

get_qubits ()

The support of all the operators in the `PauliSum` object.

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

Return type list

simplify ()

Simplifies the sum of Pauli operators according to Pauli algebra rules.

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

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

get_qubits ()

Gets all the qubits that this `PauliTerm` operates on.

`id()`

Returns the unique identifier string for the PauliTerm. Used in the simplify method of PauliSum.

Returns The unique identifier for this term.

Return type `string`

`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 for the check_commutation step in the Raesi, Wiebe, Sanders algorithm (arXiv:1108.4318, 2011).

Parameters

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

Returns True if pauli_two object commutes with pauli_list, False otherwise

Return type `bool`

`pyquil.paulis.commuting_sets` (*pauli_terms*, *nqubits*)

Gather the Pauli terms of pauli_terms variable into commuting sets

Uses algorithm defined in (Raesi, 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 $\alpha \rightarrow \exp(-1j*\alpha*term)$ represented as a Program.

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

Returns Program

Return type `Program`

`pyquil.paulis.exponentiate` (*term*)

Creates a pyQuil program that simulates the unitary evolution $\exp(-1j * term)$

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

Returns A Program object

Return type `Program`

`pyquil.paulis.is_identity` (*term*)

Check if Pauli Term is a scalar multiple of identity

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

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

Return type `bool`

`pyquil.paulis.is_zero` (*pauli_object*)

Tests to see if a PauliTerm or PauliSum is zero.

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

Returns True if PauliTerm is zero, False otherwise

Return type `bool`

`pyquil.paulis.sI(q)`

A function that returns the identity operator on a particular qubit.

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

Returns A `PauliTerm` object

Return type `PauliTerm`

`pyquil.paulis.sX(q)`

A function that returns the sigma_X operator on a particular qubit.

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

Returns A `PauliTerm` object

Return type `PauliTerm`

`pyquil.paulis.sY(q)`

A function that returns the sigma_Y operator on a particular qubit.

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

Returns A `PauliTerm` object

Return type `PauliTerm`

`pyquil.paulis.sZ(q)`

A function that returns the sigma_Z operator on a particular qubit.

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

Returns A `PauliTerm` object

Return type `PauliTerm`

`pyquil.paulis.suzuki_trotter(trotter_order, trotter_steps)`

Generate trotterization coefficients for a given number of Trotter steps.

$U = \exp(A + B)$ is approximated as $\exp(w_1 \cdot o_1) \exp(w_2 \cdot o_2) \dots$. This method returns a list $[(w_1, o_1), (w_2, o_2), \dots, (w_m, o_m)]$ 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 $[(0.5/trotter_steps, 0), (1/trotter_steps, 1), (0.5/trotter_steps, 0)] * trotter_steps$.

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` (*float*) – 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*

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*

pyquil.plots

pyquil.quil

Module for creating and defining Quil programs.

class `pyquil.quil.Program` (*instructions)

Bases: *pyquil.quilbase.InstructionGroup*

dagger (*inv_dict=None, suffix='-INV'*)

Creates the conjugate transpose of the Quil program. The program must not contain any irreversible actions (measurement, control flow, qubit allocation).

Returns The Quil program's inverse

Return type *Program*

defgate (*name, matrix*)

Define a new static gate.

Parameters

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

Returns The Program instance.

Return type *Program*

get_qubits ()

Returns a set of all the qubit indices allocated in this program, synthesizing freely allocated qubits if necessary. `.rtype`: 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`.

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*

measure (*qubit_index, classical_reg*)

Measures a qubit at `qubit_index` and puts the result in `classical_reg`

Parameters

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

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

Return type *Program*

`measure_all` (**qubit_reg_pairs*)

Measures many qubits into their specified classical bits, in the order they were entered.

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

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

Return type *Program*

`out` ()

Converts the Quil program to a readable string.

Returns String form of a program

Return type *string*

`synthesize` (*resource_manager=None*)

`while_do` (*classical_reg, q_program*)

While a classical register at index `classical_reg` is 1, loop `q_program`

Parameters

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

Returns The Quil Program with the loop instructions added.

Return type *Program*

`pyquil.quil.merge_programs` (*prog_list*)

Merges a list of pyQuil programs into a single one by appending them in sequence

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

Returns a single pyQuil program

Return type *Program*

pyquil.quil_atom

Contains the base class for Atomic Quil elements.

`class` `pyquil.quil_atom.QuilAtom`

Bases: `object`

Abstract class for atomic elements of Quil.

pyquil.quilbase

Contains the core pyQuil objects that correspond to Quil instructions.

```

class pyquil.quilbase.AbstractInstruction
    Bases: pyquil.quilbase.QuilAction

    Abstract class for representing single instructions.

    out ()

    synthesize (resource_manager=None)

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

    Representation of a classical bit address.

    Parameters value (int) – The classical address.

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

    The abstract class for binary classical instructions.

    out ()

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

    op = 'AND'

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

    op = 'EXCHANGE'

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

    op = 'FALSE'

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

    op = 'MOVE'

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

    op = 'NOT'

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

    op = 'OR'

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

    op = 'TRUE'

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

    A DEFGATE directive.

```

Parameters

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

This is the pyQuil object for a quantum gate instruction.

class `pyquil.quilbase.Halt`

Bases: `pyquil.quilbase.SimpleInstruction`

The HALT instruction.

op = 'HALT'

class `pyquil.quilbase.If` (*condition*)

Bases: `pyquil.quilbase.QuilAction`

Representation of an if construct. To use, initialize with an address to be branched on, and add instructions to `self`. Then and `self.Else` for the corresponding branches.

synthesize (*resource_manager=None*)

class `pyquil.quilbase.Instr` (*op, params, args*)

Bases: `pyquil.quilbase.AbstractInstruction`

Representation of an instruction represented by an operator, parameters, and arguments.

make_qubits_known (*rm*)

Make the qubits involved with this instruction known to a ResourceManager.

Parameters *rm* (`ResourceManager`) – A ResourceManager object.

out ()

qubits ()

The qubits this instruction affects.

Returns Set of qubit indexes.

Return type *set*

synthesize (*resource_manager=None*)

class `pyquil.quilbase.InstructionGroup` (*resource_manager=None*)

Bases: `pyquil.quilbase.QuilAction`

Representation of a sequence of instructions that can be synthesized into a Quil program.

alloc ()

Get a new qubit.

Returns A qubit.

Return type *Qubit*

extract_qubits ()

Return all qubit addresses involved in the instruction group. :return: Set of qubits. :rtype: set

free (*qubit*)

Free a qubit.

Parameters *q* (*AbstractQubit*) – An AbstractQubit instance.

inst (**instructions*)

Mutates the Program object by appending new instructions.

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

Returns self

out ()

pop ()

Pops off the last instruction.

Returns The (action, instruction) pair for the instruction that was popped.

Return type *tuple*

synthesize (*resource_manager=None*)

class `pyquil.quilbase.Jump` (*target*)

Bases: `pyquil.quilbase.AbstractInstruction`

Representation of an unconditional jump instruction (JUMP).

out ()

class `pyquil.quilbase.JumpConditional` (*target, condition*)

Bases: `pyquil.quilbase.AbstractInstruction`

Abstract representation of an conditional jump instruction.

out ()

class `pyquil.quilbase.JumpTarget` (*label*)

Bases: `pyquil.quilbase.AbstractInstruction`

Representation of a target that can be jumped to.

out ()

class `pyquil.quilbase.JumpUnless` (*target, condition*)

Bases: `pyquil.quilbase.JumpConditional`

The JUMP-UNLESS instruction.

op = 'JUMP-UNLESS'

class `pyquil.quilbase.JumpWhen` (*target, condition*)

Bases: `pyquil.quilbase.JumpConditional`

The JUMP-WHEN instruction.

op = 'JUMP-WHEN'

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

Representation of a label.

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

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

This is the pyQuil object for a Quil measurement instruction.

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

The RESET instruction.

op = 'NOP'

class `pyquil.quilbase.QuilAction`
 Bases: `object`

Representation of some executable code, i.e., something that can be synthesized into final Quil instructions.

synthesize (*resource_manager=None*)

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'

class `pyquil.quilbase.While` (*condition*)
 Bases: `pyquil.quilbase.QuilAction`

Representation of a while construct. To use, initialize with an address to branch on, and use `self.Body.inst()` to add instructions to the body of the loop.

synthesize (*resource_manager=None*)

`pyquil.quilbase.action` (*type, obj*)

`pyquil.quilbase.action_install` (*obj*)

`pyquil.quilbase.allow_raw_instructions = True`

Allow constructing programs containing raw instructions.

`pyquil.quilbase.format_matrix_element` (*element*)

Formats a parameterized matrix element.

Parameters *element* – {int, float, complex, str} The parameterized element to format.

`pyquil.quilbase.format_parameter` (*element*)

Formats a particular parameter.

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

`pyquil.quilbase.gen_label` (*prefix='L'*)

Generate a fresh label.

Parameters *prefix* (*string*) – An optional prefix for the label name.

Returns A new Label instance.

Return type *Label*

`pyquil.quilbase.issubinstance` (*x, cls*)

Checks if class *x* is an instance or subclass of *cls*.

`pyquil.quilbase.reset_label_counter` ()

pyquil.resource_manager

Contains objects and functions to manage allocation and de-allocation of quantum and classical resources in pyQuil.

class `pyquil.resource_manager.AbstractQubit`

Bases: `pyquil.quil_atom.QuilAtom`

Representation of a qubit.

index ()

class `pyquil.resource_manager.DirectQubit` (*index*)

Bases: `pyquil.resource_manager.AbstractQubit`

A reference to a particularly indexed qubit.

index ()

class `pyquil.resource_manager.Qubit` (*resource_manager*)

Bases: `pyquil.resource_manager.AbstractQubit`

A qubit whose index is determined by a ResourceManager.

index ()

Returns The index of this qubit.

Return type `int`

class `pyquil.resource_manager.ResourceManager`

Bases: `object`

allocate_qubit ()

Create a new Qubit object.

Returns A Qubit instance.

Return type *Qubit*

free_qubit (*q*)

Free the previously allocated qubit.

Parameters *q* (*Qubit*) – A live Qubit that was previously allocated.

instantiate (*qubit*)

Assign a number to a qubit.

Parameters *qubit* (*Qubit*) – A Qubit object.

reset ()

Frees all qubits, resetting this ResourceManager.

uninstantiate_index (*i*)

Free up an index that was previously instantiated.

Parameters *i* (*int*) – The index.

`pyquil.resource_manager.check_live_qubit` (*qubit*)

Ensure that an object is a live qubit. Raises errors if not.

Parameters *qubit* – A qubit, either a DirectQubit or Qubit object.

`pyquil.resource_manager.instantiated` (*qubit*)

Is the qubit instantiated?

Parameters *qubit* (*AbstractQubit*) – A AbstractQubit instance.

Returns A boolean.

Return type *bool*

`pyquil.resource_manager.merge_resource_managers` (*rm1*, *rm2*)

Merge two resource managers into a new resource manager. All qubits in the old managers will point to the new one. The `in_use` labels of the second resource manager have priority.

Parameters

- **rm1** (*ResourceManager*) – A ResourceManager.
- **rm2** (*ResourceManager*) – A ResourceManager.

Returns A merged ResourceManager.

Return type *ResourceManager*

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.

pyquil.wavefunction

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

class `pyquil.wavefunction.Wavefunction` (*amplitude_vector*)

Bases: `object`

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`

classmethod `ground` (*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`

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`

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

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

CHAPTER 9

Indices and Tables

- genindex
- modindex
- search

p

- `pyquil.gates`, 47
- `pyquil.job_results`, 52
- `pyquil.parametric`, 56
- `pyquil.paulis`, 53
- `pyquil.qpu`, 41
- `pyquil.quil`, 57
- `pyquil.quil_atom`, 58
- `pyquil.quilbase`, 59
- `pyquil.resource_manager`, 63
- `pyquil.slot`, 64
- `pyquil.wavefunction`, 65

A

AbstractInstruction (class in pyquil.quilbase), 59
 AbstractQubit (class in pyquil.resource_manager), 63
 action() (in module pyquil.quilbase), 63
 action_install() (in module pyquil.quilbase), 63
 Addr (class in pyquil.quilbase), 59
 alloc() (pyquil.quilbase.InstructionGroup method), 61
 allocate_qubit() (pyquil.resource_manager.ResourceManager method), 63
 allow_raw_instructions (in module pyquil.quilbase), 63
 AND() (in module pyquil.gates), 47
 argument_count() (in module pyquil.parametric), 56

B

BinaryClassicalInstruction (class in pyquil.quilbase), 59
 bit_string_probabilities() (pyquil.qpu.QPUConnection method), 41

C

CCNOT() (in module pyquil.gates), 47
 check_commutation() (in module pyquil.paulis), 54
 check_live_qubit() (in module pyquil.resource_manager), 64
 ClassicalAnd (class in pyquil.quilbase), 59
 ClassicalExchange (class in pyquil.quilbase), 59
 ClassicalFalse (class in pyquil.quilbase), 59
 ClassicalMove (class in pyquil.quilbase), 59
 ClassicalNot (class in pyquil.quilbase), 59
 ClassicalOr (class in pyquil.quilbase), 59
 ClassicalTrue (class in pyquil.quilbase), 59
 CNOT() (in module pyquil.gates), 47
 commuting_sets() (in module pyquil.paulis), 54
 CPHASE() (in module pyquil.gates), 48
 CPHASE00() (in module pyquil.gates), 48
 CPHASE01() (in module pyquil.gates), 48
 CPHASE10() (in module pyquil.gates), 48
 CSWAP() (in module pyquil.gates), 49

D

dagger() (pyquil.quil.Program method), 57

decode() (pyquil.job_results.JobResult method), 52
 decode() (pyquil.job_results.WavefunctionResult method), 53
 DefGate (class in pyquil.quilbase), 59
 defgate() (pyquil.quil.Program method), 57
 DirectQubit (class in pyquil.resource_manager), 63

E

EXCHANGE() (in module pyquil.gates), 49
 expectation() (pyquil.qpu.QPUConnection method), 41
 exponential_map() (in module pyquil.paulis), 54
 exponentiate() (in module pyquil.paulis), 54
 extract_qubits() (pyquil.quilbase.InstructionGroup method), 61

F

FALSE() (in module pyquil.gates), 49
 format_matrix_element() (in module pyquil.quilbase), 63
 format_parameter() (in module pyquil.quilbase), 63
 free() (pyquil.quilbase.InstructionGroup method), 61
 free_qubit() (pyquil.resource_manager.ResourceManager method), 64
 fuse() (pyquil.parametric.ParametricProgram method), 56

G

Gate (class in pyquil.quilbase), 60
 gen_label() (in module pyquil.quilbase), 63
 get() (pyquil.job_results.JobResult method), 52
 get_bitstring_from_index() (in module pyquil.wavefunction), 65
 get_constructor() (pyquil.quilbase.DefGate method), 60
 get_info() (in module pyquil.qpu), 41
 get_outcome_probs() (pyquil.wavefunction.Wavefunction method), 65
 get_params() (in module pyquil.qpu), 41
 get_qubits() (pyquil.paulis.PauliSum method), 53
 get_qubits() (pyquil.paulis.PauliTerm method), 53
 get_qubits() (pyquil.qpu.QPUConnection method), 41
 get_qubits() (pyquil.quil.Program method), 57

get_rabi_params() (in module pyquil.qpu), 42
 get_ramsey_params() (in module pyquil.qpu), 42
 get_t1_params() (in module pyquil.qpu), 42
 ground() (pyquil.wavefunction.Wavefunction class method), 65

H

H() (in module pyquil.gates), 49
 Halt (class in pyquil.quilbase), 60
 HALT (in module pyquil.gates), 49

I

I() (in module pyquil.gates), 49
 ID (in module pyquil.paulis), 53
 id() (pyquil.paulis.PauliTerm method), 53
 If (class in pyquil.quilbase), 60
 if_then() (pyquil.quil.Program method), 57
 index() (pyquil.resource_manager.AbstractQubit method), 63
 index() (pyquil.resource_manager.DirectQubit method), 63
 index() (pyquil.resource_manager.Qubit method), 63
 inst() (pyquil.quilbase.InstructionGroup method), 61
 instantiate() (pyquil.resource_manager.ResourceManager method), 64
 instantiated() (in module pyquil.resource_manager), 64
 Instr (class in pyquil.quilbase), 60
 InstructionGroup (class in pyquil.quilbase), 60
 is_done() (pyquil.job_results.JobResult method), 52
 is_identity() (in module pyquil.paulis), 54
 is_zero() (in module pyquil.paulis), 54
 issubinstance() (in module pyquil.quilbase), 63
 ISWAP() (in module pyquil.gates), 49

J

job_id() (pyquil.job_results.JobResult method), 52
 JobResult (class in pyquil.job_results), 52
 Jump (class in pyquil.quilbase), 61
 JumpConditional (class in pyquil.quilbase), 61
 JumpTarget (class in pyquil.quilbase), 61
 JumpUnless (class in pyquil.quilbase), 61
 JumpWhen (class in pyquil.quilbase), 61

L

Label (class in pyquil.quilbase), 62
 load_res() (pyquil.job_results.JobResult class method), 53

M

make_qubits_known() (pyquil.quilbase.Instr method), 60
 MEASURE() (in module pyquil.gates), 49
 measure() (pyquil.quil.Program method), 57
 measure_all() (pyquil.quil.Program method), 58

Measurement (class in pyquil.quilbase), 62
 merge_programs() (in module pyquil.quil), 58
 merge_resource_managers() (in module pyquil.resource_manager), 64
 MOVE() (in module pyquil.gates), 50

N

Nop (class in pyquil.quilbase), 62
 NOP (in module pyquil.gates), 50
 NoParametersFoundException, 41
 NOT() (in module pyquil.gates), 50
 num_args() (pyquil.quilbase.DefGate method), 60

O

op (pyquil.quilbase.ClassicalAnd attribute), 59
 op (pyquil.quilbase.ClassicalExchange attribute), 59
 op (pyquil.quilbase.ClassicalFalse attribute), 59
 op (pyquil.quilbase.ClassicalMove attribute), 59
 op (pyquil.quilbase.ClassicalNot attribute), 59
 op (pyquil.quilbase.ClassicalOr attribute), 59
 op (pyquil.quilbase.ClassicalTrue attribute), 59
 op (pyquil.quilbase.Halt attribute), 60
 op (pyquil.quilbase.JumpUnless attribute), 61
 op (pyquil.quilbase.JumpWhen attribute), 61
 op (pyquil.quilbase.Nop attribute), 62
 op (pyquil.quilbase.Reset attribute), 62
 op (pyquil.quilbase.Wait attribute), 62
 OR() (in module pyquil.gates), 50
 out() (pyquil.quil.Program method), 58
 out() (pyquil.quilbase.AbstractInstruction method), 59
 out() (pyquil.quilbase.BinaryClassicalInstruction method), 59
 out() (pyquil.quilbase.DefGate method), 60
 out() (pyquil.quilbase.Instr method), 60
 out() (pyquil.quilbase.InstructionGroup method), 61
 out() (pyquil.quilbase.Jump method), 61
 out() (pyquil.quilbase.JumpConditional method), 61
 out() (pyquil.quilbase.JumpTarget method), 61
 out() (pyquil.quilbase.RawInstr method), 62
 out() (pyquil.quilbase.SimpleInstruction method), 62
 out() (pyquil.quilbase.UnaryClassicalInstruction method), 62

P

parametric() (in module pyquil.parametric), 56
 ParametricProgram (class in pyquil.parametric), 56
 PauliSum (class in pyquil.paulis), 53
 PauliTerm (class in pyquil.paulis), 53
 PHASE() (in module pyquil.gates), 50
 pop() (pyquil.quilbase.InstructionGroup method), 61
 pretty_print() (pyquil.wavefunction.Wavefunction method), 65

- pretty_print_probabilities()
(pyquil.wavefunction.Wavefunction method), 65
- Program (class in pyquil.quil), 57
- PSWAP() (in module pyquil.gates), 50
- pyquil.gates (module), 47
- pyquil.job_results (module), 52
- pyquil.parametric (module), 56
- pyquil.paulis (module), 53
- pyquil.qpu (module), 41
- pyquil.quil (module), 57
- pyquil.quil_atom (module), 58
- pyquil.quilbase (module), 59
- pyquil.resource_manager (module), 63
- pyquil.slot (module), 64
- pyquil.wavefunction (module), 65
- ## Q
- QPUConnection (class in pyquil.qpu), 41
- Qubit (class in pyquil.resource_manager), 63
- qubits() (pyquil.quilbase.Instr method), 60
- QuilAction (class in pyquil.quilbase), 62
- QuilAtom (class in pyquil.quil_atom), 58
- ## R
- rabi() (pyquil.qpu.QPUConnection method), 41
- RabiResult (class in pyquil.job_results), 53
- ramsey() (pyquil.qpu.QPUConnection method), 41
- RamseyResult (class in pyquil.job_results), 53
- RawInstr (class in pyquil.quilbase), 62
- recover_complexes() (in module pyquil.job_results), 53
- Reset (class in pyquil.quilbase), 62
- RESET (in module pyquil.gates), 50
- reset() (pyquil.resource_manager.ResourceManager method), 64
- reset_label_counter() (in module pyquil.quilbase), 63
- ResourceManager (class in pyquil.resource_manager), 63
- run() (pyquil.qpu.QPUConnection method), 41
- run_and_measure() (pyquil.qpu.QPUConnection method), 41
- RX() (in module pyquil.gates), 51
- RY() (in module pyquil.gates), 51
- RZ() (in module pyquil.gates), 51
- ## S
- S() (in module pyquil.gates), 51
- sI() (in module pyquil.paulis), 55
- SimpleInstruction (class in pyquil.quilbase), 62
- simplify() (pyquil.paulis.PauliSum method), 53
- Slot (class in pyquil.slot), 64
- STANDARD_GATES (in module pyquil.gates), 51
- suzuki_trotter() (in module pyquil.paulis), 55
- SWAP() (in module pyquil.gates), 51
- sX() (in module pyquil.paulis), 55
- sY() (in module pyquil.paulis), 55
- synthesize() (pyquil.quil.Program method), 58
- synthesize() (pyquil.quilbase.AbstractInstruction method), 59
- synthesize() (pyquil.quilbase.If method), 60
- synthesize() (pyquil.quilbase.Instr method), 60
- synthesize() (pyquil.quilbase.InstructionGroup method), 61
- synthesize() (pyquil.quilbase.QuilAction method), 62
- synthesize() (pyquil.quilbase.While method), 62
- sZ() (in module pyquil.paulis), 55
- ## T
- T() (in module pyquil.gates), 51
- t1() (pyquil.qpu.QPUConnection method), 41
- T1Result (class in pyquil.job_results), 53
- term_with_coeff() (in module pyquil.paulis), 55
- trotterize() (in module pyquil.paulis), 55
- TRUE() (in module pyquil.gates), 51
- ## U
- UnaryClassicalInstruction (class in pyquil.quilbase), 62
- unstantiate_index() (pyquil.resource_manager.ResourceManager method), 64
- unpack_classical_reg() (in module pyquil.gates), 52
- unpack_qubit() (in module pyquil.gates), 52
- ## V
- value() (pyquil.slot.Slot method), 65
- ## W
- Wait (class in pyquil.quilbase), 62
- WAIT (in module pyquil.gates), 51
- wait_for_job() (in module pyquil.job_results), 53
- Wavefunction (class in pyquil.wavefunction), 65
- wavefunction() (pyquil.qpu.QPUConnection method), 41
- WavefunctionResult (class in pyquil.job_results), 53
- While (class in pyquil.quilbase), 62
- while_do() (pyquil.quil.Program method), 58
- ## X
- X() (in module pyquil.gates), 52
- ## Y
- Y() (in module pyquil.gates), 52
- ## Z
- Z() (in module pyquil.gates), 52