

---

# **pyQuil Documentation**

***Release 2.0.0***

**Rigetti Computing**

**Nov 01, 2018**



<b>1</b>	<b>Contents</b>	<b>3</b>
1.1	Installation and Getting Started	3
1.1.1	Upgrading or Installing pyQuil	3
1.1.2	Downloading the QVM and Compiler	4
1.1.3	Getting Started	5
1.2	Forest 2.0: Migration Guide	7
1.2.1	What's changed	7
1.2.2	Overview of Updates to Quil and pyQuil	7
1.2.3	Parametric programs	8
1.2.4	Details of updates to Quil	9
1.2.5	Backwards compatibility	9
1.2.6	Details of pyQuil and Forest updates	9
1.2.7	Backwards compatibility and migration	10
1.2.8	Miscellanea	16
1.2.9	Error reporting	16
1.3	Programs and Gates	18
1.3.1	Introduction	18
1.3.2	The Standard Gate Set	19
1.3.3	Declaring Memory	20
1.3.4	Measurement	20
1.3.5	Parametric Compilation	20
1.3.6	Defining New Gates	20
1.3.7	Defining Parametric Gates	21
1.3.8	Pragmas	22
1.3.9	Ways to Construct Programs	22
1.3.10	QPU-allowable Quil	23
1.4	The Quantum Virtual Machine (QVM)	23
1.4.1	Using the QVM	23
1.4.2	Multi-Qubit Basis Enumeration	25
1.4.3	Simulating the QPU using the QVM	25
1.4.4	Examples of Quantum Programs	26
1.5	The Wavefunction Simulator	28
1.5.1	Meyer-Penny Game	29
1.6	The Quil Compiler	30
1.6.1	Expectations for Program Contents	30
1.6.2	Interacting with the Compiler	31

1.6.3	Legal compiler input . . . . .	32
1.6.4	Region-specific compiler features through PRAGMA . . . . .	32
1.6.5	Common Error Messages . . . . .	34
1.7	Noise and Quantum Computation . . . . .	35
1.7.1	Modeling Noisy Quantum Gates . . . . .	35
1.7.2	Noisy Gates on the Rigetti QVM . . . . .	37
1.7.3	Adding Decoherence Noise . . . . .	43
1.7.4	Modeling Readout Noise . . . . .	46
1.7.5	Working with Readout Noise . . . . .	49
1.8	Advanced Usage . . . . .	54
1.8.1	Using Qubit Placeholders . . . . .	54
1.8.2	Quantum Fourier Transform (QFT) . . . . .	56
1.8.3	Classical Control Flow . . . . .	57
1.8.4	Parametric Depolarizing Noise . . . . .	59
1.8.5	Parametric Programs . . . . .	59
1.8.6	Pauli Operator Algebra . . . . .	60
1.9	Exercises . . . . .	61
1.9.1	Exercise 1: Quantum Dice . . . . .	61
1.9.2	Exercise 2: Controlled Gates . . . . .	61
1.9.3	Exercise 3: Grover’s Algorithm . . . . .	61
1.10	Source Code Documentation . . . . .	62
1.10.1	pyquil.api . . . . .	62
1.10.2	pyquil.device . . . . .	73
1.10.3	pyquil.gates . . . . .	79
1.10.4	pyquil.noise . . . . .	88
1.10.5	pyquil.parser . . . . .	94
1.10.6	pyquil.paulis . . . . .	94
1.10.7	pyquil.quil . . . . .	98
1.10.8	pyquil.quilbase . . . . .	105
1.10.9	pyquil.wavefunction . . . . .	110
1.11	Changelog . . . . .	111
1.11.1	v2.0.0 (November 1, 2018) . . . . .	111
1.11.2	v1.9 (June 6, 2018) . . . . .	112
1.12	Introduction to Quantum Computing . . . . .	114
1.12.1	From Bit to Qubit . . . . .	114
1.12.2	Qubit Operations . . . . .	119
1.12.3	The Quantum Abstract Machine . . . . .	122
1.12.4	Next Steps . . . . .	127
<b>2</b>	<b>Indices and Tables</b>	<b>129</b>
	<b>Bibliography</b>	<b>131</b>
	<b>Python Module Index</b>	<b>133</b>

The Rigetti Forest [Software Development Kit](#) includes pyQuil, the Rigetti Quil Compiler (quilc), and the Quantum Virtual Machine (qvm).

**Longtime users of Rigetti Forest will notice a few changes.** First, the SDK now contains a downloadable compiler and a QVM. Second, the SDK contains pyQuil 2.0, with significant updates to previous versions. As a result, programs written using previous versions of the Forest toolkit will need to be updated to pyQuil 2.0 to be compatible with the QVM or compiler.

After installing the SDK and updating pyQuil in [Installation and Getting Started](#), see [Forest 2.0: Migration Guide](#) to get caught up on what's new!

Quantum Cloud Services will provide users with a dedicated Quantum Machine Image, which will come prepackaged with the Forest SDK. We're releasing a Preview to the Forest SDK now, so current users can begin migrating code (and share feedback with us early and often!). Longtime Forest users should start with the Migration Guide which outlines key changes in this SDK Preview release.

If you're new to Forest, we hope this documentation will provide everything you need to get up and running with the toolkit. Once you've oriented yourself here, proceed to the section [Installation and Getting Started](#) to get started. If you're new to quantum computing, you also go to our section on [Introduction to Quantum Computing](#). There, you'll learn the basic concepts needed to write quantum software. You can also work through an introduction to quantum computing in a jupyter notebook; launch the notebook from the source folder in pyquil's docs:

```
cd pyquil/docs/source
jupyter notebook intro_to_qc.ipynb
```

#### A few terms to orient you as you get started with Forest:

1. **pyQuil:** An open source Python library to help you write and run quantum programs. The source is hosted on [github](#).
2. **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, [A Practical Quantum Instruction Set Architecture](#).
3. **QVM:** The [Quantum Virtual Machine](#) is an implementation of a quantum abstract machine on classical hardware. The QVM lets you use a regular computer to simulate a small quantum computer and execute Quil programs.
4. **QPU:** Quantum processing unit. This refers to the physical hardware chip which we run quantum programs on.
5. **Quil Compiler:** The compiler, `quilc`, compiles Quil written for one quantum abstract machine (QAM) to another. Our compiler will take arbitrary Quil and compile it for the given QAM, according to its supported instruction set architecture.
6. **Forest SDK:** Our software development kit, optimized for near-term quantum computers that operate as co-processors, working in concert with traditional processors to run hybrid quantum-classical algorithms. For references on problems addressable with near-term quantum computers, see [Quantum Computing in the NISQ era and beyond](#).

Our flagship product [Quantum Cloud Services](#) offers users an on-premise, dedicated access point to our quantum computers. This access point is a fully-configured VM, which we call a Quantum Machine Image. A QMI is bundled with the same downloadable SDK mentioned above, and a command line interface (CLI), which is used for scheduling compute time on our quantum computers. To sign up for our waitlist, please click the link above. If you'd like to access to our quantum computers for research, please email [support@rigetti.com](mailto:support@rigetti.com).

---

**Note:** To join our user community, connect to the Rigetti Slack workspace at <https://rigetti-forest.slack.com>.

---



## 1.1 Installation and Getting Started

To make full use of the Rigetti Forest SDK, you will need pyQuil, the QVM, and the Quil Compiler. On this page, we will take you through the process of installing all three of these. We also step you through *running a basic pyQuil program*.

---

**Note:** If you're running from a Quantum Machine Image, installation has been completed for you. Continue to *Getting Started*.

---

### 1.1.1 Upgrading or Installing pyQuil

PyQuil 2.0 is our library for generating and executing Quil programs on the Rigetti Forest platform.

Before you install, we recommend that you activate a Python 3.6+ virtual environment. Then, install pyQuil using `pip`:

```
pip install --pre pyquil
```

For those of you that already have pyQuil, you can upgrade with:

```
pip install --upgrade --pre pyquil
```

If you would like to stay up to date with the latest changes and bug fixes, you can also opt to install pyQuil from the source [here](#).

---

**Note:** PyQuil requires Python 3.6 or later.

---

## 1.1.2 Downloading the QVM and Compiler

The Forest 2.0 Downloadable SDK Preview currently contains:

- The Rigetti Quantum Virtual Machine (qvm) which allows high-performance simulation of Quil programs
- The Rigetti Quil Compiler (quilk) which allows compilation and optimization of Quil programs to native gate sets

The QVM and the compiler are packed as program binaries that are accessed through the command line. Both of them provide support for direct command-line interaction, as well as a server mode. The *server mode* is required for use with pyQuil.

Request the Forest SDK [here](#). You'll receive an email right away with the download links for macOS, Linux (.deb), Linux (.rpm), and Linux (bare-bones).

All installation mechanisms, except the bare-bones package, require administrative privileges to install. To use the QVM and Quil Compiler from the bare-bones package, you will have to install the prerequisite dependencies on your own.

### Installing on macOS

Mount the file `forest-sdk.dmg` by double clicking on it in your email. From there, open `forest-sdk.pkg` by double-clicking on it. Follow the installation instructions.

Upon successful installation, one should be able to open a new terminal window and run the following two commands:

```
qvm --version
quilk --version
```

To uninstall, delete the following files:

```
/usr/local/bin/qvm
/usr/local/bin/quilk
/usr/local/share/man/man1/qvm.1
/usr/local/share/man/man1/quilk.1
```

### Installing the QVM and Compiler on Linux (deb)

Download the Debian distribution by clicking on the link in your email. Unpack the tarball and change to that directory by doing:

```
tar -xf forest-sdk-linux-deb.tar.bz2
cd forest-sdk-2.0rc2-linux-deb
```

From here, run the following command:

```
sudo ./forest-sdk-2.0rc2-linux-deb.run
```

Upon successful installation, one should be able to run the following two commands:

```
qvm --version
quilk --version
```

To uninstall, type:

```
sudo apt remove forest-sdk
```

### Installing the QVM and Compiler on Linux (rpm)

Download the RPM-based distribution by clicking on the link in your email. Unpack the tarball and change to that directory by doing:

```
tar -xf forest-sdk-linux-rpm.tar.bz2
cd forest-sdk-2.0rc2-linux-rpm
```

From here, run the following command:

```
sudo ./forest-sdk-2.0rc2-linux-rpm.run
```

Upon successful installation, one should be able to run the following two commands:

```
qvm --version
quirc --version
```

To uninstall, type:

```
sudo rpm -e forest-sdk
# or
sudo yum uninstall forest-sdk
```

### Installing the QVM and Compiler on Linux (bare-bones)

The bare-bones installation only contains the executable binaries and manual pages, and doesn't contain any of the requisite dynamic libraries. As such, installation doesn't require administrative or `sudo` privileges.

First, unpack the tarball and change to that directory by doing:

```
tar -xf forest-sdk-linux-barebones.tar.bz2
cd forest-sdk-2.0rc2-linux-barebones
```

From here, run the following command:

```
./forest-sdk-2.0rc2-linux-barebones.run
```

Upon successful installation, this will have created a new directory `rigetti` in your home directory that contains all of the binary and documentation artifacts.

This method of installation requires one, through whatever means, to install shared libraries for BLAS, LAPACK, and libffi. On a Debian-derivative system, this could be accomplished with

```
sudo apt-get install liblapack-dev libblas-dev libffi-dev
```

To uninstall, remove the directory `~/rigetti`.

## 1.1.3 Getting Started

To get started using the SDK, you can either interact with the QVM and the compiler directly from the command line, or you can run them in server mode and use them with pyQuil. In this section, we're going to explain how to do the latter.

For more information about directly interacting with the QVM and the compiler, refer to their respective manual pages. After *installation*, you can read the manual pages by opening a new terminal window and typing `man qvm` (for the QVM) or `man quilc` (for the compiler). Quit out of the manual page by typing `q`.

### Setting Up Server Mode for PyQuil

---

**Note:** This set up is only necessary to run pyQuil locally. If you’re running in a QMI, this has already been done for you.

---

It’s easy to start up local servers for the QVM and quilc on your laptop. You should have two terminal windows open to run in the background. We recommend using a resource such as `tmux` for running and managing multiple programs in one terminal.

```
### CONSOLE 1
$ qvm -S

Welcome to the Rigetti QVM
(Configured with 10240 MiB of workspace and 8 workers.)
[2018-09-20 15:39:50] Starting server on port 5000.

### CONSOLE 2
$ quilc -S

Welcome to the Rigetti Quil Compiler
[2018-09-19 11:22:37] Starting server: 0.0.0.0 : 6000.
```

That’s it! You’re all set up to run pyQuil locally. Your programs will make requests to these server endpoints to compile your Quil programs to native Quil, and to simulate those programs on the QVM.

### Run Your First Program

Now that our local endpoints are up and running, we can start running pyQuil programs! We will run a simple program on the Quantum Virtual Machine (QVM).

The program we will create prepares a fully entangled state between two qubits, called a Bell State. This state is in an equal superposition between `|00` and `|11`, meaning that it is equally likely that a measurement will result in measuring both qubits in the ground state or both qubits in the excited state. For more details about the physics behind these concepts, see *Introduction to Quantum Computing*.

To begin, start up python however you like. You can open a jupyter notebook (type `jupyter notebook` in your terminal), open an interactive python notebook in your terminal (with `ipython3`), or simply launch python in your terminal (type `python3`). Recall that you need Python 3.6+ to use pyQuil.

Import a few things from pyQuil:

```
from pyquil import Program, get_qc
from pyquil.gates import *
```

The `Program` object allows us to build up a Quil program. `get-qc` connects us to a `QuantumComputer` object, which specifies what our program should run on (see: *The Quantum Virtual Machine (QVM)*). We’ve also imported all (\*) gates from the `pyquil.gates` module, which allows us to add operations to our program (*Programs and Gates*).

Next, let’s construct our Bell State.

```
# construct a Bell State program
p = Program(H(0), CNOT(0, 1))
```

We’ve accomplished this by driving qubit 0 into a superposition state (that’s what the “H” gate does), and then creating an entangled state between qubits 0 and 1 (that’s what the “CNOT” gate does). Finally, we’ll want to run our program:

```
# run the program on a QVM
qc = get_qc('9q-square-qvm')
result = qc.run_and_measure(p, trials=10)
print(result[0])
print(result[1])
```

Compare the two arrays of measurement results. The results will be correlated between the qubits and random from shot to shot.

The `qc` is a simulated quantum computer. By specifying we want to `.run_and_measure`, we’ve told our QVM to run the program specified above, collapse the state with a measurement, and return the results to us. `trials` refers to the number of times we run the whole program.

The call to `run_and_measure` will make a request to the two servers we started up in the previous section: first, to the `quilc` server instance to compile the Quil program into native Quil, and then to the `qvm` server instance to simulate and return measurement results of the program 10 times. If you open up the terminal windows where your servers are running, you should see output printed to the console regarding the requests you just made.

In the following sections, we’ll cover gates, program construction & execution, and go into detail about our Quantum Virtual Machine, our QPUs, noise models and more. If you’ve used pyQuil before, continue on to our [Forest 2.0: Migration Guide](#). Once you’re set with that, jump to [Programs and Gates](#) to continue.

## 1.2 Forest 2.0: Migration Guide

The goals of this guide are to cover changes to the Forest SDK (containing pyquil 2.0, new Quil, Quil Compiler, and QVM), and to go through an example of migrating a VQE program from Forest 1.3 (pyQuil 1.9, Quil 1.0) to be able to run on the new Forest SDK.

---

**Note:** For installation & setup, follow the download instructions in the section [Installation and Getting Started](#) at the top of the page.

---

### 1.2.1 What’s changed

With the new Forest SDK, users will be able to run pyQuil programs on a downloadable QVM and Quil Compiler!

In the following section, we’ll cover the main changes to pyQuil, Quil, the Quil Compiler, and the QVM.

### 1.2.2 Overview of Updates to Quil and pyQuil

The primary differences in the programming language Quil 1.0 (as appearing in pyQuil 1.3) and Quil 2 (as appearing in 2.0) amount to an enhanced memory model. Whereas the classical memory model in Quil 1.0 amounted to an flat bit array of indefinite size, the memory model in Quil 2 is segmented into typed, sized, named regions.

In terms of compatibility with Quil 1.0, this primarily changes how `MEASURE` instructions are formulated, since their classical address targets must be modified to fit the new framework. In terms of new functionality, this allows angle values to be read in from classical memory.

Quil 2 also introduces easier ways to manipulate gates by using gate modifiers. Two gate modifiers are supported currently, *DAGGER* and *CONTROLLED*.

*DAGGER* can be written before a gate to refer to its inverse. For instance

```
DAGGER RX(pi/3) 0
```

would have the same effect as

```
RX(-pi/3) 0
```

*DAGGER* can be applied to any gate, but also circuits defined with *DEFCIRCUIT*. This allows for easy reversal of unitary circuits:

```
DEFCIRCUIT BELL:
    H 0
    CNOT 0 1

# construct a Bell state
BELL
# disentangle, bringing us back to identity
DAGGER BELL
```

### 1.2.3 Parametric programs

The main benefit for users of declared memory regions in Quil is that angle values for parametric gates can be loaded at execution time on the QPU. Consider the following simple QAOA instance:

```
DECLARE ro BIT[2]
DECLARE beta REAL
DECLARE gamma REAL

H 0
RZ(beta) 0
H 0
H 1
RZ(beta) 1
H 1

CNOT 0 1
RZ(gamma) 1
CNOT 0 1

MEASURE 0 ro[0]
MEASURE 1 ro[1]
```

To generate a “landscape” plot as `beta` and `gamma` range, it was previously required to generate a different program for each possible pair of values, substitute that pair in, send it to the compiler, and send the resulting compiled program to the QPU for execution (and hence generate the expectation values). With Quil 2, this exact program can be sent to the compiler, which returns a nativized Quil program that still has parametric gates with parameters referencing the classical memory regions `beta` and `gamma`. This program can then be loaded onto the QPU for repeated execution with different values of `beta` and `gamma`, without recompilation in between.

### 1.2.4 Details of updates to Quil

Classical memory regions must be explicitly requested and named by a Quil program using `DECLARE` directive. A generic `DECLARE` directive has the following syntax:

```
DECLARE region-name type([count])? (SHARING parent-region-name (OFFSET
(offset-count offset-type)+))?
```

The non-keyword items have the following allowable values:

- `region-name`: any non-keyword formal name.
- `type`: one of `REAL`, `BIT`, `OCTET`, or `INTEGER`
- `parent-region-name`: any non-keyword formal name previously used as `region-name` in a different `DECLARE` statement.
- `offset-count`: a nonnegative integer.
- `offset-type`: the same allowable values as `type`.

Here are some examples:

```
DECLARE beta REAL[32]
DECLARE ro BIT[128]
DECLARE beta-bits BIT[1436] SHARING beta
DECLARE fourth-bit-in-beta1 BIT SHARING beta OFFSET 1 REAL 4 BIT
```

In order, the intention of these `DECLARE` statements is:

- Allocate an array called `beta` of length 32, each entry of which is a `REAL` number.
- Allocate an array called `ro` of length 128, each entry of which is a `BIT`.
- Name an array called `beta-bits`, which is an overlay onto the existing array `beta`, so that the bit representations of elements of `beta` can be directly examined and manipulated.
- Name a single `BIT` called `fourth-bit-in-beta1` which overlays the fourth bit of the bit representation of the `REAL` value `beta[1]`.

### 1.2.5 Backwards compatibility

Quil 1.0 is not compatible with Quil 2 in the following ways:

- The unnamed memory references `[n]` and `[n-m]` have no direct equivalent in Quil 2 and must be replaced by named memory references. (This primarily affects `MEASURE` instructions.)
- The classical memory manipulation instructions have been modified: the operands of `AND` have been reversed (so that in Quil 2, the left operand is the target address) and `OR` has been replaced by `IOR` and its operands reversed (so that, again, in Quil 2 the left operand is the target address).

In all other instances, Quil 1.0 will operate identically with Quil 2.

When confronted with program text conforming to Quil 1.0, pyQuil 2.0 will automatically rewrite `MEASURE q [n]` to `MEASURE q ro[n]` and insert a `DECLARE` statement which allocates a `BIT`-array of the appropriate size named `ro`.

### 1.2.6 Details of pyQuil and Forest updates

#### Updates to Forest

- In Forest 1.3, job submission to the QPU was done from your workstation and the ability was gated by on user ID. In Forest 2.0, job submission to the QPU must be done from your remote virtual machine, called a QMI (*Quantum Machine Image*).
- In Forest 1.3, user data persisted indefinitely in cloud storage and could be accessed using the assigned job ID. In Forest 2.0, user data is stored only transiently, and it is the user’s responsibility to handle long-term data storage on their QMI.

### Updates to pyQuil

- In pyQuil 1.9, API calls were organized by endpoint (e.g., all simulation calls were passed to a `QVMConnection` object). In pyQuil 2.0, API calls are organized by type (e.g., `run` calls are sent to a `QuantumComputer` but `wavefunction` calls are sent to a `WavefunctionSimulator`).
- In pyQuil 1.9, quantum program evaluation was asynchronous on the QPU and a mix of synchronuous or asynchronous on the QVM. In pyQuil 2.0, all quantum program evaluation is synchronous.
- In pyQuil 1.9, each quantum program execution call started from scratch. In pyQuil 2.0, compiled program objects can be reused.

## 1.2.7 Backwards compatibility and migration

PyQuil 2.0 is not backwards compatible with pyQuil 1.9. However, the new API objects available in pyQuil 2.0 have compatibility methods that make migration to pyQuil 2.0 easier.

---

**Note:** Users writing new programs from scratch are encouraged to use the bare pyQuil 2.0 programming model over the compatibility methods. It is not possible to use the fanciest new features of Forest 2.0 (e.g., parametric execution of parametric programs) from within the compatibility model.

---

Whereas pyQuil 1.9 organized API calls around “connection objects” (viz., `CompilerConnection`, `QPUConnection`, and `QVMConnection`), pyQuil 2.0 organizes API calls around function, so that QVM- and QPU-based objects can be more easily swapped. These API objects fall into two groups:

- **QuantumComputer:** This wrapper object houses the typical ingredients for execution of a hybrid classical-quantum algorithm: an interface to a compiler, an interface to a quantum computational device, and some optional wrapper routines. `QuantumComputer` objects themselves can be manually initialized with these ingredients, or they can be requested by name from the Forest 2.0 service, which will populate these subfields with the appropriate objects for execution on a particular quantum device, real or simulated.
- **AbstractCompiler:** An interface to a compiler service. Compilers are responsible for two tasks: converting arbitrary Quil to “native” (or “device-specific”) Quil, and converting native Quil to control system binaries.
- **QAM:** An interface to a quantum computational device. This can be populated by a connection to an actual QPU, or it can be populated by a connection to a QVM (**Quantum Virtual Machine**).
- **Wrapper routines:** Execution of programs in pyQuil 1.9 was typically done with a single API call (e.g., `.run()`). `QuantumComputer` exposes a near-identical interface for single runs of quantum programs, which wraps and hides the more low-level pyQuil 2.0 infrastructure.
- **WavefunctionSimulator:** This wrapper object houses the typical ingredients used for the debug process of wavefunction inspection. This is inherently **not** a procedure natively available on a quantum computational device, and so this wrapper either calls out to a QVM or functions as a repeated sampling wrapper from a physical quantum computational device.

**Example: Computing the bond energy of molecular hydrogen, pyQuil 1.9 vs 2.0**

By way of example, let's consider the following pyQuil 1.9 program, which computes the natural bond distance in molecular hydrogen using a VQE-type algorithm:

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

def setup_forest_objects():
    qvm = QVMConnection()
    return qvm

def build_wf_ansatz_prep(theta):
    program = Program(f"""
# set up initial state
X 0
X 1

# build the exponentiated operator
RX(pi/2) 0
H 1
H 2
H 3

CNOT 0 1
CNOT 1 2
CNOT 2 3
RZ({theta}) 3
CNOT 2 3
CNOT 1 2
CNOT 0 1

RX(-pi/2) 0
H 1
H 2
H 3

# measure out the results
MEASURE 0 [0]
MEASURE 1 [1]
MEASURE 2 [2]
MEASURE 3 [3]""")
    return program

# some constants
bond_step, bond_min, bond_max = 0.05, 0, 200
angle_step, angle_min, angle_max = 0.1, 0, 63
convolution_coefficients = [0.1698845197777728, 0.16988451977777283, -0.
↪2188630663199042,
                           -0.2188630663199042]

shots = 1000

# set up the Forest object
qvm = setup_forest_objects()
```

(continues on next page)

(continued from previous page)

```

# get all the unweighted expectations for all the sample wavefunctions
occupations = list(range(angle_min, angle_max))
indices = list(range(4))
for offset in occupations:
    # set up the Program object, each time we have a new parameter
    program = build_wf_ansatz_prep(angle_min + offset * angle_step)
    bitstrings = qvm.run(program, indices, trials=shots)

    totals = [0, 0, 0, 0]
    for bitstring in bitstrings:
        for index in indices:
            totals[index] += bitstring[index]
    occupations[offset] = [t / shots for t in totals]

# compute minimum energy as a function of bond length
min_energies = list(range(bond_min, bond_max))
for bond_length in min_energies:
    energies = []
    for offset in range(angle_min, angle_max):
        energy = 0
        for j in range(4):
            energy += occupations[offset][j] * convolution_coefficients[j]
        energies.append(energy)

    min_energies[bond_length] = min(energies)

min_index = min_energies.index(min(min_energies))
min_energy, relaxed_length = min_energies[min_index], min_index * bond_step

```

In order to port this code to pyQuil 2.0, we need change only one thing: the part referencing `QVMConnection` should be replaced by an equivalent part referencing a `QuantumComputer` connected to a QVM. Specifically, the following snippet

```

from pyquil.api import QVMConnection

def setup_forest_objects():
    qvm = QVMConnection()
    return qvm

```

can be changed to

```

from pyquil.api import get_qc

def setup_forest_objects():
    qc = get_qc("9q-square-qvm")
    return qc

```

and the references to `qvm` in the main body are changed to `qc` instead. Since the `QuantumComputer` object also exposes a `run` routine and pyQuil itself automatically rewrites 1.9-style MEASURE instructions into 2.0-style instructions, this is all we need to do.

If we are willing to be more intrusive, we can also take advantage of pyQuil 2.0's classical memory and parametric programs. The first piece to change is the Quil program itself: we remove the argument `theta` from the Python function `build_wf_ansatz_prep`, with the intention of letting the QPU fill it in later. In turn, we modify the Quil program itself to have a REAL memory parameter named `theta`. We also declare a few BITS for our MEASURE instructions to target.

```

def build_wf_ansatz_prep():
    program = Program("""
# set up memory
DECLARE ro BIT[4]
DECLARE theta REAL

# set up initial state
X 0
X 1

# build the exponentiated operator
RX(pi/2) 0
H 1
H 2
H 3

CNOT 0 1
CNOT 1 2
CNOT 2 3
RZ(theta) 3
CNOT 2 3
CNOT 1 2
CNOT 0 1

RX(-pi/2) 0
H 1
H 2
H 3

# measure out the results
MEASURE 0 ro[0]
MEASURE 1 ro[1]
MEASURE 2 ro[2]
MEASURE 3 ro[3]""")
    return program

```

Next, we modify the execution loop. Rather than reformulating the `Program` object each time, we build and compile it once, then use the `.load()` method to transfer the parametric program to the (simulated) quantum device. We then set only the angle value within the inner loop, and we change to using `.run()` and `.wait()` methods to manage control between us and the quantum device.

More specifically, the old execution loop

```

# get all the unweighted expectations for all the sample wavefunctions
occupations = list(range(angle_min, angle_max))
indices = list(range(4))
for offset in occupations:
    # set up the Program object, each time we have a new parameter
    program = build_wf_ansatz_prep(angle_min + offset * angle_step)
    bitstrings = qvm.run(program, indices, trials=shots)

    totals = [0, 0, 0, 0]
    for bitstring in bitstrings:
        for index in indices:
            totals[index] += bitstring[index]
    occupations[offset] = [t / shots for t in totals]

```

becomes

```
# set up the Program object, ONLY ONCE
program = build_wf_ansatz_prep()
program.wrap_in_numshots_loop(shots=shots)
nq_program = qc.compiler.quil_to_native_quil(program)
binary = qc.compiler.native_quil_to_executable(nq_program)
qc.qam.load(binary)

# get all the unweighted expectations for all the sample wavefunctions
occupations = list(range(angle_min, angle_max))
indices = list(range(4))
for offset in occupations:
    qc.qam.write_memory(region_name='theta', value=angle_min + offset * angle_step)
    qc.qam.run()
    qc.qam.wait()
    bitstrings = qc.qam.read_memory(region_name="ro")

    totals = [0, 0, 0, 0]
    for bitstring in bitstrings:
        for index in indices:
            totals[index] += bitstring[index]
    occupations[offset] = [t / shots for t in totals]
```

Overall, the resulting program looks like this:

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

def setup_forest_objects():
    qc = get_qc("9q-square-qvm")
    return qc

def build_wf_ansatz_prep():
    program = Program("""
# set up memory
DECLARE ro BIT[4]
DECLARE theta REAL

# set up initial state
X 0
X 1

# build the exponentiated operator
RX(pi/2) 0
H 1
H 2
H 3

CNOT 0 1
CNOT 1 2
CNOT 2 3
RZ(theta) 3
CNOT 2 3
CNOT 1 2
CNOT 0 1
```

(continues on next page)

(continued from previous page)

```

RX(-pi/2) 0
H 1
H 2
H 3

# measure out the results
MEASURE 0 ro[0]
MEASURE 1 ro[1]
MEASURE 2 ro[2]
MEASURE 3 ro[3]"""
    return program

# some constants
bond_step, bond_min, bond_max = 0.05, 0, 200
angle_step, angle_min, angle_max = 0.1, 0, 63
convolution_coefficients = [0.1698845197777728, 0.16988451977777283, -0.
↪2188630663199042,
                           -0.2188630663199042]

shots = 1000

# set up the Forest object
qc = setup_forest_objects()

# set up the Program object, ONLY ONCE
program = build_wf_ansatz_prep()
program.wrap_in_numshots_loop(shots=shots)
nq_program = qc.compiler.quil_to_native_quil(program)
binary = qc.compiler.native_quil_to_executable(nq_program)
qc.qam.load(binary)

# get all the unweighted expectations for all the sample wavefunctions
occupations = list(range(angle_min, angle_max))
indices = list(range(4))
for offset in occupations:
    qc.qam.write_memory(region_name='theta', value=angle_min + offset * angle_step)
    qc.qam.run()
    qc.qam.wait()
    bitstrings = qc.qam.read_memory(region_name="ro")

    totals = [0, 0, 0, 0]
    for bitstring in bitstrings:
        for index in indices:
            totals[index] += bitstring[index]
    occupations[offset] = [t / shots for t in totals]

# compute minimum energy as a function of bond length
min_energies = list(range(bond_min, bond_max))
for bond_length in min_energies:
    energies = []
    for offset in range(angle_min, angle_max):
        energy = 0
        for j in range(4):
            energy += occupations[offset][j] * convolution_coefficients[j]
        energies.append(energy)

    min_energies[bond_length] = min(energies)

```

(continues on next page)

(continued from previous page)

```
min_index = min_energies.index(min(min_energies))
min_energy, relaxed_length = min_energies[min_index], min_index * bond_step
```

## 1.2.8 Miscellanea

Quil promises that a BIT is 1 bit and that an OCTET is 8 bits. Quil does not promise, however, the size or layout of INTEGER or REAL. These are implementation-dependent.

On the QPU, INTEGER refers to an unsigned integer stored in a 48-bit wide little-endian word, and REAL refers to a 48-bit wide little-endian fixed point number of type <0.48>. In general, these datatypes are implementation-dependent. OCTET always refers to an 8-bit wide unsigned integer and is independent of implementation.

Memory regions are all “global”: DECLARE directives cannot appear in the body of a DEFCIRCUIT.

On the QVM, INTEGER is a two’s complement signed 64-bit integer. REAL is an IEEE-754 double-precision floating-point number.

## 1.2.9 Error reporting

Because the Forest 2.0 execution model is no longer asynchronous, our error reporting model has also changed. Rather than writing to technical support with a job ID, users will need to provide all pertinent details to how they produced an error.

PyQuil 2.0 makes this task easy with the function decorator @pyquil\_protect, found in the module pyquil.api. By decorating a failing function (or a function that has the potential to fail), any unhandled exceptions will cause an error log to be written to disk (at a user-specifiable location). For example, the nonsense code block

```
from pyquil.api import pyquil_protect

...

@pyquil_protect
def my_function():
    ...
    qc.qam.load(qc)
    ...

my_function()
```

causes the following error to be printed:

```
>>> PYQUIL_PROTECT <<<
An uncaught exception was raised in a function wrapped in pyquil_protect. We are_
↪writing out a
log file to "/Users/your_name/Documents/pyquil/pyquil_error.log".

Along with a description of what you were doing when the error occurred, send this_
↪file to Rigetti Computing
support by email at support@rigetti.com for assistance.
>>> PYQUIL_PROTECT <<<
```

as well as the following log file to be written to disk at the indicated location:

```

{
  "stack_trace": [
    {
      "name": "pyquil_protect_wrapper",
      "filename": "/Users/your_name/Documents/pyquil/pyquil/error_reporting.py",
      "line_number": 197,
      "locals": {
        "e": "TypeError('quil_binary argument must be a QVMExecutableResponse. This_
↪error is typically triggered by
        forgetting to pass (nativized) Quil to native_quil_to_executable or by_
↪using a compiler meant to be used
        for jobs bound for a QPU.',)",
        "old_filename": "'pyquil_error.log'",
        "kwargs": "{}",
        "args": "()",
        "log_filename": "'pyquil_error.log'",
        "func": "<function my_function at 0x106dc4510>"
      }
    },
    {
      "name": "my_function",
      "filename": "<stdin>",
      "line_number": 10,
      "locals": {
        "offset": "0",
        "occupations": "[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
↪ 18, 19, 20, 21, 22, 23, 24,
        25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
↪43, 44, 45, 46, 47, 48, 49, 50,
        51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62]"
      }
    },
    {
      "name": "wrapper",
      "filename": "/Users/your_name/Documents/pyquil/pyquil/error_reporting.py",
      "line_number": 228,
      "locals": {
        "pre_entry": "CallLogValue(timestamp_in=datetime.datetime(2018, 9, 11, 18, 40,
↪ 19, 65538),
        timestamp_out=None, return_value=None)",
        "key": "run('<pyquil.api._qvm.QVM object at 0x1027e3940>', )",
        "kwargs": "{}",
        "args": "(<pyquil.api._qvm.QVM object at 0x1027e3940>)",
        "func": "<function QVM.run at 0x106db4e18>"
      }
    },
    {
      "name": "run",
      "filename": "/Users/your_name/Documents/pyquil/pyquil/api/_qvm.py",
      "line_number": 376,
      "locals": {
        "self": "<pyquil.api._qvm.QVM object at 0x1027e3940>",
        "__class__": "<class 'pyquil.api._qvm.QVM'>"
      }
    }
  ],
  "timestamp": "2018-09-11T18:40:19.253286",

```

(continues on next page)

(continued from previous page)

```

"call_log": {
  "__init__('<pyquil.api._qvm.QVM object at 0x1027e3940>', '<pyquil.api._base_
↳connection.ForestConnection object at
    0x1027e3588>', )": {
    "timestamp_in": "2018-09-11T18:40:18.967750",
    "timestamp_out": "2018-09-11T18:40:18.968170",
    "return_value": "None"
  },
  "run('<pyquil.api._qvm.QVM object at 0x1027e3940>', )": {
    "timestamp_in": "2018-09-11T18:40:19.065538",
    "timestamp_out": null,
    "return_value": null
  }
},
"exception": "TypeError('quil_binary argument must be a QVMExecutableResponse. This
↳error is typically triggered
  by forgetting to pass (nativized) Quil to native_quil_to_executable or by using a
↳compiler meant to be used for
  jobs bound for a QPU.',)",
"system_info": {
  "python_version": "3.6.3 (default, Jan 25 2018, 13:55:02) \n[GCC 4.2.1 Compatible
↳Apple LLVM 9.0.0
    (clang-900.0.39.2)]",
  "pyquil_version": "2.0.0-internal.1"
}
}

```

Please attach such a logfile to any request for support.

## 1.3 Programs and Gates

**Note:** If you're running locally, remember set up the QVM and quilc in server mode before trying to use them: [Setting Up Server Mode for PyQuil](#).

### 1.3.1 Introduction

Quantum programs are written in Forest using the Program object. This Program abstraction will help us compose Quil programs.

```
from pyquil import Program
```

Programs are constructed by adding quantum gates to it, which are defined in the `gates` module. We can import all standard gates with the following:

```
from pyquil.gates import *
```

Let's instantiate a Program and add an operation to it. We will act an X gate on qubit 0.

```
p = Program()
p += X(0)
```

All qubits begin in the ground state. This means that if we measure a qubit without applying operations on it, we expect to receive a 0 result. The X gate will rotate qubit 0 from the ground state to the excited state, so a measurement immediately after should return a 1 result. More details about gate operations are explained in [Introduction to Quantum Computing](#).

We can print our pyQuil program (`print(p)`) to see the equivalent Quil representation:

```
X 0
```

This isn't going to be very useful to us without measurements. In pyQuil 2.0, we have to `DECLARE` a memory space to read measurement results, which we call “readout results” and abbreviate as `ro`. With measurement, our whole program looks like this:

```
from pyquil import Program
from pyquil.gates import *

p = Program()
ro = p.declare('ro', 'BIT', 1)
p += X(0)
p += MEASURE(0, ro[0])

print(p)
```

```
DECLARE ro BIT[1]
X 0
MEASURE 0 ro[0]
```

We've instantiated a program, declared a memory space named `ro` with one single bit of memory, applied an X gate on qubit 0, and finally measured qubit 0 into the zeroth index of the memory space named `ro`.

Awesome! That's all we need to get results back. Now we can actually see what happens if we run this program on the Quantum Virtual Machine (QVM). We just have to add a few lines to do this.

```
from pyquil import get_qc

...

qc = get_qc('1q-qvm') # You can make any 'nq-qvm' this way for any reasonable 'n'
compiled_program = qc.compile(p)
result = qc.run(compiled_program)
print(result)
```

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

```
[[1]]
```

For more information on what the above result means, and on executing quantum programs on the QVM in general, see [The Quantum Virtual Machine \(QVM\)](#). The remainder of this section of the docs will be dedicated to constructing programs in detail, an essential part of becoming fluent in quantum programming.

### 1.3.2 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(theta)`, `S`, `T`
- Controlled phase gates: `CZ`, `CPHASE00(alpha)`, `CPHASE01(alpha)`, `CPHASE10(alpha)`, `CPHASE(alpha)`
- Cartesian rotation gates: `RX(theta)`, `RY(theta)`, `RZ(theta)`
- Controlled  $X$  gates: `CNOT`, `CCNOT`
- Swap gates: `SWAP`, `CSWAP`, `ISWAP`, `PSWAP(alpha)`

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

### 1.3.3 Declaring Memory

*Coming soon*

### 1.3.4 Measurement

*Coming soon*

### Specifying the number of trials

*Coming soon*

### 1.3.5 Parametric Compilation

*Coming soon*

### 1.3.6 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

from pyquil import Program
from pyquil.quil import DefGate

# First we define the new gate from a matrix
sqrt_x = np.array([[ 0.5+0.5j,  0.5-0.5j],
                   [ 0.5-0.5j,  0.5+0.5j]])

# Get the Quil definition for the new gate
sqrt_x_definition = DefGate("SQRT-X", sqrt_x)
# Get the gate constructor
SQRT_X = sqrt_x_definition.get_constructor()

# Then we can use the new gate
p = Program()
p += sqrt_x_definition
p += 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
```

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)
```

Now we can use this gate in the same way that we used SQRT\_X, but we will pass it two arguments rather than one, since it operates on two qubits.

```
x_sqrt_x_definition = DefGate("X-SQRT-X", x_sqrt_x)
X_SQRT_X = x_sqrt_x_definition.get_constructor()

# Then we can use the new gate
p = Program(x_sqrt_x_definition, X_SQRT_X(0, 1))
```

**Tip:** To inspect the wavefunction that will result from applying your new gate, you can use the [Wavefunction Simulator](#) (e.g. `print (WavefunctionSimulator().wavefunction(p))`).

### 1.3.7 Defining Parametric Gates

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

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

# Define the new gate from a matrix
theta = Parameter('theta')
crx = np.array([
    [1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, quil_cos(theta / 2), -1j * quil_sin(theta / 2)],
    [0, 0, -1j * quil_sin(theta / 2), quil_cos(theta / 2)]
])

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

# Create our program and use the new parametric gate
p = Program()
p += gate_definition
p += H(0)
p += CRX(np.pi/2)(0, 1)
```

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

---

**Tip:** To inspect the wavefunction that will result from applying your new gate, you can use the *Wavefunction Simulator* (e.g. `print(WavefunctionSimulator().wavefunction(p))`).

---

## 1.3.8 Pragmas

### Specifying A Qubit Rewiring Scheme

*Coming soon*

### Asking for a Delay

*Coming soon* (Note: time limit)

## 1.3.9 Ways to Construct Programs

PyQuil supports a variety of methods for constructing programs however you prefer. Multiple instructions can be applied at once, and programs can be added together. PyQuil can also produce a `Program` by interpreting raw Quil text. You can still use the more pyQuil 1.X style of using the `.inst` method to add instruction gates. Thus, the following are all valid programs:

```
# Preferred method
p = Program()
p += X(0)
p += Y(1)
print(p)

# Multiple instructions in declaration
print(Program(X(0), Y(1)))

# A composition of two programs
print(Program(X(0)) + Program(Y(1)))

# Raw Quil with newlines
print(Program("X 0\nY 1"))

# Raw Quil comma separated
print(Program("X 0", "Y 1"))

# Chained inst; less preferred
print(Program().inst(X(0)).inst(Y(1)))
```

All of the above methods will produce the same output:

```
X 0
Y 1
```

The `pyquil.parser` submodule provides a front-end to other similar parser functionality.

## Fixing a Mistaken Instruction

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

```
p = Program(X(0), Y(1))
print(p)

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

```
X 0
Y 1

We can fix by popping:
X 0
```

### 1.3.10 QPU-allowable Quil

Apart from DECLARE and PRAGMA directives, a program must break into the following three regions, each optional:

1. A RESET command.
2. A sequence of quantum gate applications.
3. A sequence of MEASURE commands.

The only memory that is writeable is the region named `ro`, and only through MEASURE instructions. All other memory is read-only.

The keyword SHARING is disallowed.

Compilation is unavailable for invocations of DEFGATES with parameters read from classical memory.

## 1.4 The Quantum Virtual Machine (QVM)

The Rigetti Quantum Virtual Machine is an implementation of the Quantum Abstract Machine from *A Practical Quantum Instruction Set Architecture*.<sup>1</sup> 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.

### 1.4.1 Using the QVM

After [downloading the SDK](#), the QVM is available on your local machine. You can initialize a local QVM instance by doing the following:

```
### CONSOLE 1
$ qvm -S
Configured with 2048 MiB of workspace and 8 workers.)
[2018-09-20 15:39:50] Starting server on port 5000.
```

<sup>1</sup> <https://arxiv.org/abs/1608.03355>

```
from pyquil import get_qc, Program
from pyquil.gates import *
qvm = get_qc('9q-square-qvm')
```

One executes quantum programs on the QVM using a `.run(...)` method, intended to closely mirror how one will execute programs on a real QPU (check out [our website to see current and legacy QPUs](#)). We also offer a Wavefunction Simulator (formerly a part of the QVM object), which allows users to construct and inspect wavefunctions of quantum programs. Learn more about *The Wavefunction Simulator*.

(For information on constructing quantum programs, please refer back to *Programs and Gates*.)

## The `.run(...)` method

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

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

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

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

```
[[1]]
```

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

The *inner* list, on the other hand, is an enumeration over the results stored in the classical addresses. We see that the result of this program is that the classical register `[0]` now stores the state of qubit 0, which should be 1 after an `X`-gate. We can of course ask for more classical registers:

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

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

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

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

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

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

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

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

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

## 1.4.2 Multi-Qubit Basis Enumeration

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

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

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

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

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

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

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

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

For additional information why we decided on this basis ordering check out our note *Someone shouts, “|01000>!” Who is Excited?*<sup>2</sup>.

## 1.4.3 Simulating the QPU using the QVM

The QVM is a powerful tool for testing quantum programs before executing them on the QPU. In addition to the `noise.py` module for generating custom noise models for simulating noise on the QVM, pyQuil provides a simple interface for loading the QVM with noise models tailored to Rigetti’s available QPUs, in just one modified line of

<sup>2</sup> <https://arxiv.org/abs/1711.02086>

code. This is made possible via the `Device` class, which holds hardware specification information, noise model information, and instruction set architecture (ISA) information regarding connectivity. This information is held in the `Specs`, `ISA` and `NoiseModel` attributes of the `Device` class, respectively.

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

---

**Note:** This feature is currently deprecated, in advance of a new QPU (with new noise models). For users interested in creating noise models for the QVM, you can do so by following the instructions in [Noise and Quantum Computation](#).

---

```
from pyquil.api import get_devices, QVMConnection

device_name = get_device('quantum_device_name')
qvm = QVMConnection(device_name)
```

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

## 1.4.4 Examples of Quantum Programs

To create intuition for a new class of algorithms, that will run on the Quantum Virtual Machine (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.

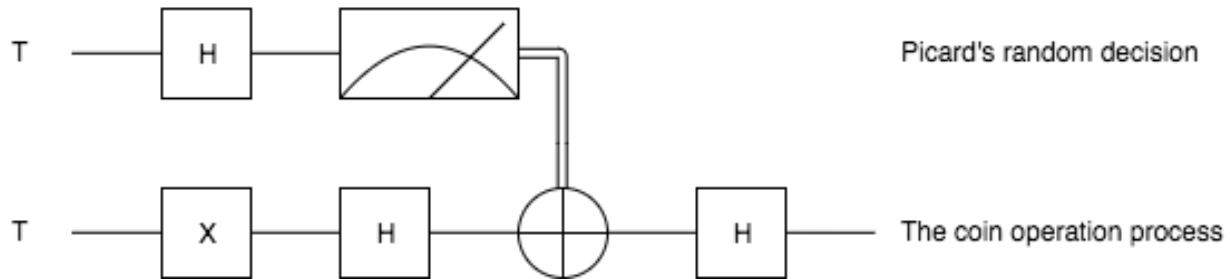
The rules: Picard is to place a penny Heads up into an opaque box. Then Picard and Q take turns to flip or not flip the penny without being able to see it; first Q then P then Q again. After this the penny is revealed; Q wins if it shows Heads (H), while Tails (T) makes Picard the winner.

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:

```
from pyquil import get_qc
from pyquil.gates import I, H, X
from pyquil import Program
from pyquil.api import WavefunctionSimulator
prog = Program()
ro = prog.declare('classical_register', 'BIT', 2)
qvm = get_qc('9q-generic-qvm')
```

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.

```
prog = Program()
ro = prog.declare('classical_register', 'BIT', 2)
picard_register = ["1"]
answer_register = ["0"]
```

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

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

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

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

Finally we play the game several times

```
qvm.run_and_measure(prog, [0, 1], 10)
```

and record the register outputs as

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

Remember that the first number is the outcome of the game (value of the *answer\_register*) whereas the second number is the outcome of Picard's decision (value of the *picard\_register*).

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

## Exercises

### Prisoner's Dilemma

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

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

Can you design a program that simulates this game?

## 1.5 The Wavefunction Simulator

Formerly a part of the QVM object in pyQuil, the Wavefunction Simulator allows you to directly inspect the wavefunction of a quantum state prepared by your program. Because of the probabilistic nature of quantum information, the programs you'll be running on the QPU can give a distribution of outputs. When running on the QPU or QVM, you would aggregate results (anywhere from tens of trials to 100k+!) that you can sample to get back a distribution.

With the Wavefunction Simulator, you can look at the distribution without having to collect samples from your program. This can save a lot of time for small programs. Let's walk through a basic example of using WavefunctionSimulator:

```
from pyquil import Program  
from pyquil.gates import *  
from pyquil.api import WavefunctionSimulator  
wf_sim = WavefunctionSimulator()  
coin_flip = Program(H(0))  
wf_sim.wavefunction(coin_flip)
```

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

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

```
coin_flip = Program(H(0))
wavefunction = wf_sim.wavefunction(coin_flip)
print(wavefunction)
```

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

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

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

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

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

## 1.5.1 Meyer-Penny Game

A conceptually simple example that falls into this class is the Meyer-Penny Game. The game goes as follows:

The Starship Enterprise, during one of its deep-space missions, is facing an immediate calamity, when a powerful alien suddenly appears on the bridge. The alien, named Q, offers to help Picard, the captain of the Enterprise, under the condition that Picard beats Q in a simple game of penny flips.

### The rules:

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

### Picard vs. Q

Picard 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.

Let's see how this works!

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.

We first import the necessary tools

```
from pyquil import Program
from pyquil.api import WavefunctionSimulator
from pyquil.gates import *

wf_sim = WavefunctionSimulator()
prog = Program()
ro = prog.declare('ro', 'BIT', 2)
```

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 = ro[1]
answer_register = ro[0]
```

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

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

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

```
prog.inst(X(0), H(1))
prog.inst(H(0))
prog.measure(1, picard_register)
prog.if_then(picard_register, then_branch, else_branch)
prog.inst(H(0))
prog.measure(0, answer_register)
print(prog)
```

Finally we play the game several times

```
wf_sim.run_and_measure(prog, [0, 1], 10)
```

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`).

No matter what Picard does, Q will always win!

## 1.6 The Quil Compiler

### 1.6.1 Expectations for Program Contents

The QPUs have much more limited natural gate sets than the standard gate set offered by pyQuil: the gate operators are constrained to lie in  $RZ(\theta)$ ,  $RX(k\pi/2)$ , and  $CZ$ ; and the gates are required to act on physically available hardware

(for single-qubit gates, this means acting only on live qubits, and for qubit-pair gates, this means acting on neighboring qubits). To ameliorate these limitations, the Rigetti software toolkit contains an optimizing compiler that translates arbitrary Quil to native Quil and native ProtoQuil to executables suitable for Rigetti hardware.

## 1.6.2 Interacting with the Compiler

A `QuantumComputer` object supplied by the function `pyquil.api.get_qc()` comes equipped with a connection to a Rigetti quantum compiler. If the object is called `qc`, then this can be accessed using the instance method `.compile()`, as in the following:

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

qc = get_qc("9q-square-qvm")

ep = qc.compile(Program(H(0), CNOT(0,1), CNOT(1,2)))

print(ep.program) # here ep is of type PyquilExecutableResponse, which is not always_
↪inspectable
```

with output

```
PRAGMA EXPECTED_REWIRING "#(7 8 5 0 1 2 3 4 6)"
RZ(pi/2) 7
RX(pi/2) 7
RZ(-pi/2) 8
RX(pi/2) 8
CZ 8 7
RZ(-pi/2) 5
RX(pi/2) 5
RX(-pi/2) 8
CZ 5 8
RX(-pi/2) 5
RZ(pi/2) 5
RZ(-pi/2) 7
RZ(-pi/2) 8
PRAGMA CURRENT_REWIRING "#(7 8 5 0 1 2 3 4 6)"
PRAGMA EXPECTED_REWIRING "#(7 8 5 0 1 2 3 4 6)"
PRAGMA CURRENT_REWIRING "#(7 8 5 0 1 2 3 4 6)"
```

The compiler connection is also available directly via the property `qc.compiler`. The precise class of this object changes based on context (e.g., `QPUCompiler`, `QVMCompiler`, or `LocalQVMCompiler`), but it always conforms to the interface laid out by `pyquil.api._qac`:

- `compiler.quil_to_native_quil(program)`: This method converts a Quil program into native Quil, according to the ISA that the compiler is initialized with. The input parameter is specified as a `Program` object, and the output is given as a new `Program` object, equipped with a `.metadata` property that gives extraneous information about the compilation output (e.g., gate depth, as well as many others). This call blocks until Quil compilation finishes.
- `compiler.native_quil_to_executable(nq_program)`: This method converts a ProtoQuil program, which is promised to consist only of native gates for a given ISA, into an executable suitable for submission to one of a QVM or a QPU. This call blocks until the executable is generated.

The instance method `qc.compile` described above is a combination of these two methods: first the incoming Quil is nativized, and then that is immediately turned into an executable. Accordingly, the previous example snippet is

identical to the following:

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

qc = get_qc("9q-square-qvm")

p = Program(H(0), CNOT(0,1), CNOT(1,2))
np = qc.compiler.quil_to_native_quil(p)
ep = qc.compiler.native_quil_to_executable(np)

print(ep.program) # here ep is of type PyquilExecutableResponse, which is not always_
↪inspectable
```

### 1.6.3 Legal compiler input

The QPU is not able to execute all possible Quil programs, and so Quil bound for execution on a QPU must conform to the “ProtoQuil” standard. At present, a Quil program qualifies as ProtoQuil if it has the following form:

- The program may or may not begin with a RESET instruction. (If provided, the QPU will actively reset the state of the quantum device to the ground state before program execution. If omitted, the QPU will wait for a relaxation period to pass before program execution instead.)
- This is then followed by a block of native quantum gates. A gate is native if it is of the form RZ ( $\theta$ ) for any value  $\theta$ , RX ( $k\pi/2$ ) for an integer  $k$ , or CZ  $q_0\ q_1$  for  $q_0, q_1$  a pair of qubits participating in a qubit-qubit interaction.
- This is then followed by a block of MEASURE instructions.

### 1.6.4 Region-specific compiler features through PRAGMA

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

---

**Note:** The pyQuil compiler interface is under construction, and some of the PRAGMA directives will soon be replaced by finer-grained method calls.

---

#### Preserved regions

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

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

```
DECLARE ro BIT[2]

#   prepare a Bell state
```

(continues on next page)

(continued from previous page)

```

H 0
CNOT 0 1

#   wait a while
PRAGMA PRESERVE_BLOCK
I 0
I 1
I 0
I 1
#   ...
I 0
I 1
PRAGMA END_PRESERVE_BLOCK

#   and read out the results
MEASURE 0 ro[0]
MEASURE 1 ro[1]

```

## Parallelizable regions

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

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

```

DECLARE ro BIT[4]

# Stage one
H 0
H 1
H 2
H 3

# Stage two
PRAGMA COMMUTING_BLOCKS
PRAGMA BLOCK
CNOT 0 1
RZ(0.4) 1
CNOT 0 1
PRAGMA END_BLOCK
PRAGMA BLOCK
CNOT 1 2
RZ(0.6) 2
CNOT 1 2
PRAGMA END_BLOCK
PRAGMA BLOCK

```

(continues on next page)

(continued from previous page)

```
CNOT 2 3
RZ(0.8) 3
CNOT 2 3
PRAGMA END_BLOCK
PRAGMA BLOCK
CNOT 0 3
RZ(0.9) 3
CNOT 0 3
PRAGMA END_BLOCK
PRAGMA END_COMMUTING_BLOCKS

# Stage three
H 0
H 1
H 2
H 3

MEASURE 0 ro[0]
MEASURE 1 ro[1]
MEASURE 2 ro[2]
MEASURE 3 ro[3]
```

## Rewirings

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

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

- *PARTIAL* (default): The compiler will start with nothing assigned to each physical qubit. Then, it will fill in the logical-to-physical mapping as it encounters new qubits in the program, making its best guess for where they should be placed.
- *NAIVE*: The compiler will start with an identity mapping as the initial rewiring. In particular, qubits will **not** be rewired unless the program requests a qubit-qubit interaction not natively available on the QPU.
- *RANDOM*: the compiler will start with a random permutation
- *GREEDY*: the compiler will make a guess for the initial rewiring based on a quick initial scan of the entire program.

### 1.6.5 Common Error Messages

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

- `!!! Error: Matrices do not lie in the same projective class. The compiler attempted to decompose an operator as native Quil instructions, and the resulting instructions do not match`

the original operator. This can happen when the original operator is not a unitary matrix, and could indicate an invalid `DEFGATE` block.

## 1.7 Noise and Quantum Computation

### 1.7.1 Modeling Noisy Quantum Gates

#### Pure States vs. Mixed States

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

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

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

$$\langle X \rangle_\rho = \text{Tr}(X\rho)$$

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

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

which you can easily verify for  $\rho_\psi$  assuming that the state is normalized. If we want to describe a situation with classical uncertainty between states  $\rho_1$  and  $\rho_2$ , then we can take their weighted sum

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

where  $p \in [0, 1]$  gives the classical probability that the state is  $\rho_1$ .

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

#### Quantum Gate Errors

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

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

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

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

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

## Why Do Incoherent Errors Happen?

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

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

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

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

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

the final state will generally not admit such a factorization.

## A Toy Model

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

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

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

Using the Baker-Campbell-Hausdorff theorem we can expand this to second order in  $\epsilon$

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

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

$$\begin{aligned} \rho'_S := \rho'_E = \rho_{S,0} \underbrace{\rho_{E,0}}_{\psi_{E,0} V_E \psi_{E,0} = 0} - \frac{i\sqrt{\epsilon}t}{\hbar} \underbrace{\left[ V_S \rho_{S,0} \underbrace{V_E \rho_{E,0}}_{\psi_{E,0} V_E \psi_{E,0} = 0} - \rho_{S,0} V_S \underbrace{\rho_{E,0} V_E}_{\psi_{E,0} V_E \psi_{E,0} = 0} \right]}_0 \\ - \frac{\epsilon t^2}{2\hbar^2} [V_S^2 \rho_{S,0} V_E^2 \rho_{E,0} + \rho_{S,0} V_S^2 \rho_{E,0} V_E^2 - 2V_S \rho_{S,0} V_S V_E \rho_{E,0} V_E] \\ = \rho_{S,0} - \frac{\gamma}{2} [V_S^2 \rho_{S,0} + \rho_{S,0} V_S^2 - 2V_S \rho_{S,0} V_S] \end{aligned} \quad (1.1)$$

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

$$\rho_S \rightarrow \rho'_S = K_1 \rho K_1^\dagger + K_2 \rho K_2^\dagger = \rho - \frac{\gamma}{2} [V_S^2 \rho + \rho V_S^2] + \gamma V_S \rho V_S + O(\gamma^2) \quad (1.4)$$

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

### 1.7.2 Noisy Gates on the Rigetti QVM

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

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

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

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

$$\begin{aligned} \overline{\rho'_S} &= \frac{1}{N} \sum_{n=1}^N \psi'_n \psi_n^\dagger \\ &= \frac{1}{N} \sum_{n=1}^N p_{j_n}^{-1} K_{j_n} \psi'_n \psi_n^\dagger K_{j_n}^\dagger \end{aligned} \quad (1.5)$$

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

$$\overline{\rho'_S} = \sum_{\ell=1}^n \frac{N_\ell}{N} p_\ell^{-1} K_\ell \psi'_S \psi_S^\dagger K_\ell^\dagger \quad (1.7)$$

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

$$\overline{\rho'_S} \approx \sum_{\ell=1}^n K_\ell \psi'_S \psi_S^\dagger K_\ell^\dagger \quad (1.8)$$

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

### Getting Started

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

3. For your QUIL program `p`, call:

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

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

**Scroll down for some examples!**

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

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

```
cxn = QVMConnection()
```

### Example 1: Amplitude Damping

Amplitude damping channels are imperfect identity maps with Kraus operators

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

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

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

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

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

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

    :param list kraus_ops: The Kraus operators.
    :param numpy.ndarray g: The unitary gate.
    :return: A list of transformed Kraus operators.
    """
```

(continues on next page)

(continued from previous page)

```

    return [kj.dot(g) for kj in kraus_ops]

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

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

%%time

# single step damping probability
damping_per_I = 0.02

# number of program executions
trials = 200

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

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

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

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

results = np.array(results)

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

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

DARK_TEAL = '#48737F'

```

(continues on next page)

(continued from previous page)

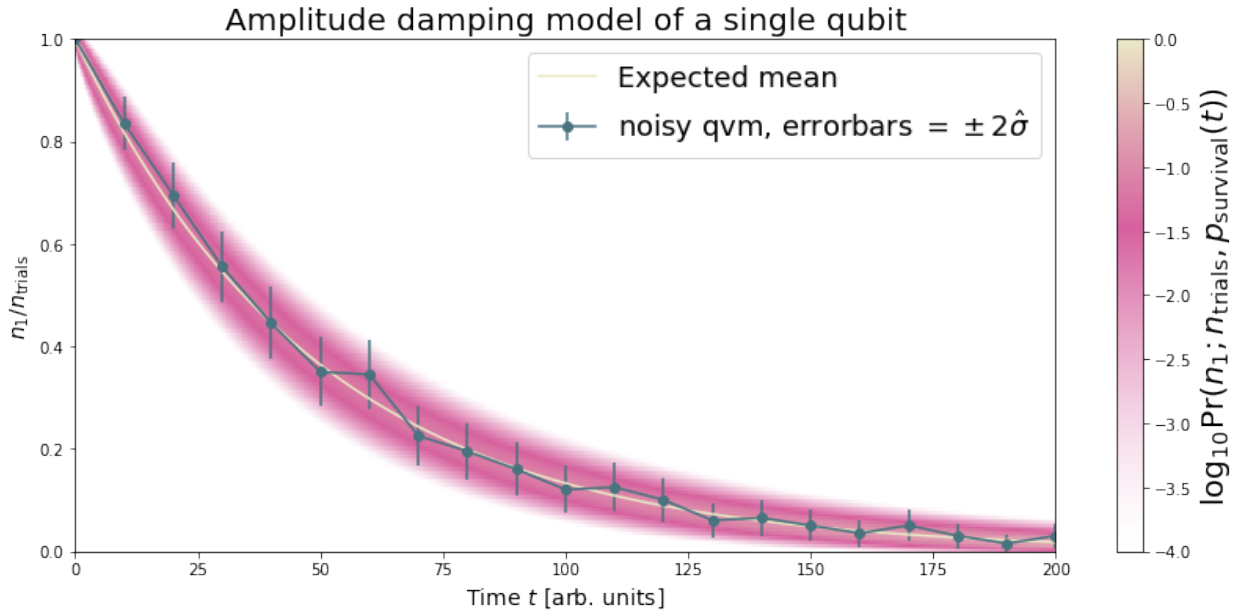
```

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

plt.figure(figsize=(14, 6))
plt.pcolor(dense_lengths, np.arange(trials+1)/trials, logpmf.T, cmap=cm, vmin=-4,
↪vmax=logpmf.max())
plt.plot(dense_lengths, survival_probs, c=BEIGE, label="Expected mean")
plt.errorbar(lengths, results[:,0], yerr=2*results[:,1], c=DARK_TEAL,
            label=r"noisy qvm, errorbars $ = \pm 2\hat{\sigma}$", marker="o")
cb = plt.colorbar()
cb.set_label(r"$\log_{10} \mathrm{Pr}(n_1; n_{\mathrm{trials}}, p_{\mathrm{survival}}(t))$",
↪size=20)

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

```



### Example 2: Dephased CZ-gate

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

$$\begin{aligned}
 K_1(p) &= \sqrt{1-p}I_2 \\
 K_2(p) &= \sqrt{p}\sigma_Z
 \end{aligned}$$

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

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

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

The Kraus operators for this are given by

$$K'_1(p, q) = K_1(p) \otimes K_1(q) \quad (1.9)$$

$$K'_2(p, q) = K_2(p) \otimes K_1(q) \quad (1.10)$$

$$K'_3(p, q) = K_1(p) \otimes K_2(q) \quad (1.11)$$

$$K'_4(p, q) = K_2(p) \otimes K_2(q) \quad (1.12)$$

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

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

$$K_1^{CZ}(p, q) = K_1(p) \otimes K_1(q) U_{CZ} \quad (1.13)$$

$$K_2^{CZ}(p, q) = K_2(p) \otimes K_1(q) U_{CZ}$$

$$K_3^{CZ}(p, q) = K_1(p) \otimes K_2(q) U_{CZ}$$

$$K_4^{CZ}(p, q) = K_2(p) \otimes K_2(q) U_{CZ}$$

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```

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

```

```

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

# number of program executions
trials = 500

results = []

for jj, p in enumerate(ps):

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

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

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

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

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

results = np.array(results)

```

```

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

```

(continues on next page)

(continued from previous page)

Wall time: 1min 49s

```

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

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

```

```

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

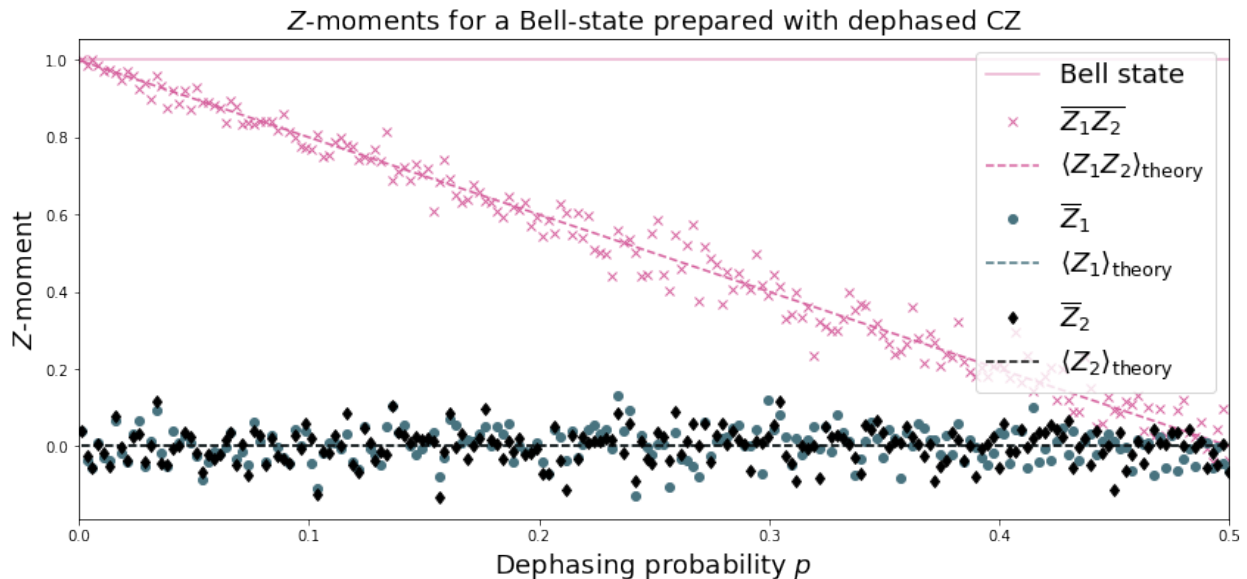
plt.plot(ps, Z1Z2m, "x", c=FUSCHIA, label=r"$\overline{Z_1 Z_2}$")
plt.plot(ps, 1-2*ps, "--", c=FUSCHIA, label=r"$\langle Z_1 Z_2 \rangle_{\rm theory}$")

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

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

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

```



### 1.7.3 Adding Decoherence Noise

In this example, we investigate how a program might behave on a near-term device that is subject to  $T1$ - and  $T2$ -type noise using the convenience function `pyquil.noise.add_decoherence_noise()`. The same module also contains some other useful functions to define your own types of noise models, e.g.,

`pyquil.noise.tensor_kraus_maps()` for generating multi-qubit noise processes, `pyquil.noise.combine_kraus_maps()` for describing the succession of two noise processes and `pyquil.noise.append_kraus_to_gate()` which allows appending a noise process to a unitary gate.

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

## The Task

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

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

## The Idiomatic PyQuil Program

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

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

## The Compiled Program

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

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

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

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

(continues on next page)

(continued from previous page)

```

    RZ(-pi/2, 1),
    RX(-pi/2, 1),
    RZ(theta, 1),
    RX(pi/2, 1),
    CZ(1, 0),
    RX(pi/2, 0),
    RZ(pi/2, 0),
    RZ(-pi/2, 1),
    RX(pi/2, 1),
    RZ(-pi/2, 1),
    1)

```

## Scan Over Noise Parameters

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

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

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

```

from pyquil.api import QVMConnection
cxn = QVMConnection()

```

```

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

```

```

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

```

```

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

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

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

        record = {
            'z0': z0,
            'z1': z1,
            'zz': zz,
            'theta': theta,
            't1': t1,

```

(continues on next page)

(continued from previous page)

```

    }
    records += [record]

```

## Plot the Results

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

```

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

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

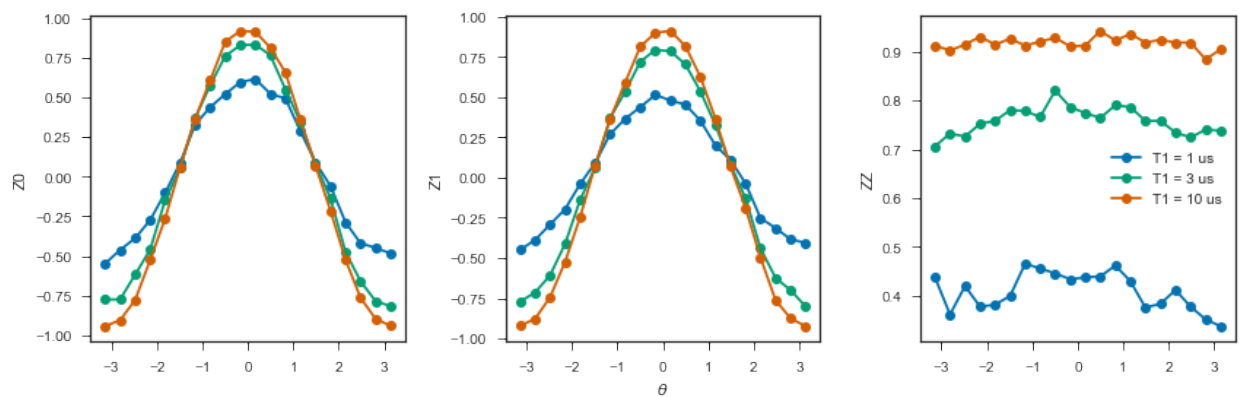
for t1 in t1s:
    df = df_all.query('t1 == @t1')

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

ax3.legend(loc='best')

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

```



## 1.7.4 Modeling Readout Noise

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

1. Transmission line noise that makes a 0-state look like a 1-state or vice versa. We call this **classical readout bit-flip error**. This type of readout noise can be reduced by tailoring optimal readout pulses and using super-

conducting, quantum limited amplifiers to amplify the readout signal before it is corrupted by classical noise at the higher temperature stages of our cryostats.

2. T1 qubit decay during readout (our readout operations can take more than a  $\mu$ second unless they have been specially optimized), which leads to readout signals that initially behave like 1-states but then collapse to something resembling a 0-state. We will call this **T1-readout error**. This type of readout error can be reduced by achieving shorter readout pulses relative to the T1 time, i.e., one can try to reduce the readout pulse length, or increase the T1 time or both.

## Qubit Measurements

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

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

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

## Projective, Ideal Measurement

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

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

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

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

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

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

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

which is the projection postulate applied to mixed states.

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

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

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

## Classical Readout Bit-Flip Error

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

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

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

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

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

where

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

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

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

that must sum to 1:

$$\sum_{x' \in \mathcal{O}'} E_{x'} = \sum_{x \in \mathcal{O}} \underbrace{\left[ \sum_{x' \in \mathcal{O}'} p(x'|x) \right]}_1 \Pi_x = \sum_{x \in \mathcal{O}} \Pi_x = 1.$$

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

## How to Model $T_1$ Error

$T_1$  type errors fall outside our framework so far as they involve a scenario in which the *quantum state itself* is corrupted during the measurement process in a way that potentially erases the pre-measurement information as opposed to a loss

of purely classical information. The most appropriate framework for describing this is given by that of measurement instruments, but for the practical purpose of arriving at a relatively simple description, we propose describing this by a T1 damping Kraus map followed by the noisy readout process as described above.

## Further Reading

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

## 1.7.5 Working with Readout Noise

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

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

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

Scroll down for some examples!

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

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

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

cxn = QVMConnection()
```

### Example 1: Rabi Sequence with Noisy Readout

```
%%time

# number of angles
num_theta = 101

# number of program executions
trials = 200

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

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

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

(continues on next page)

(continued from previous page)

```

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

```

```

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

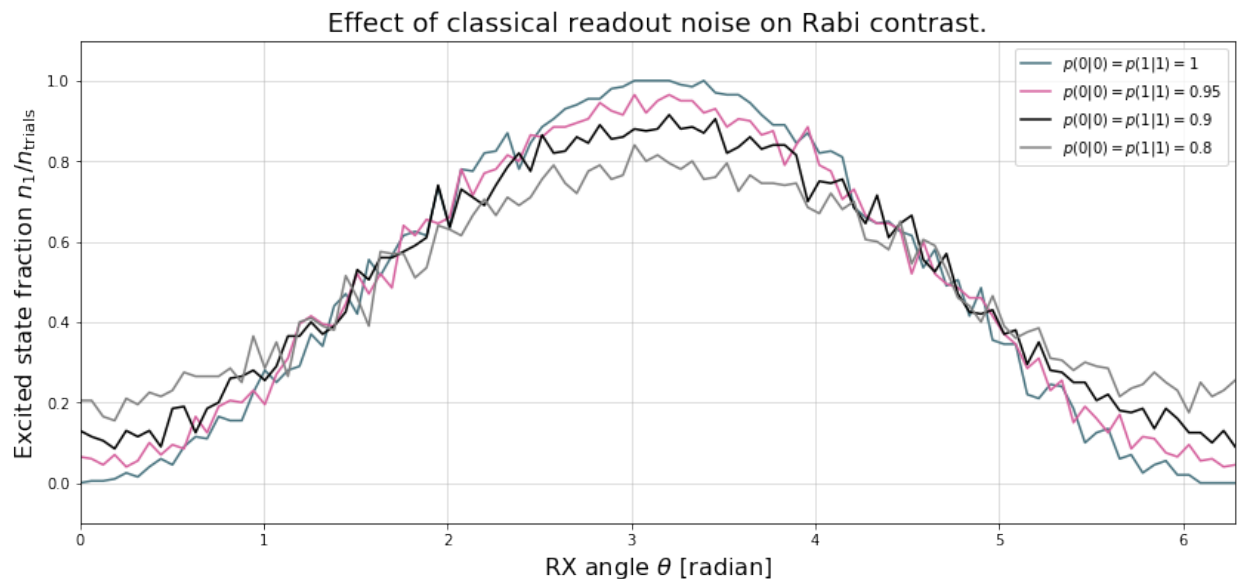
```

```

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

```

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



## Example 2: Estimate the Assignment Probabilities

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

### On a perfect quantum computer

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

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

### On an imperfect quantum computer

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

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

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

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

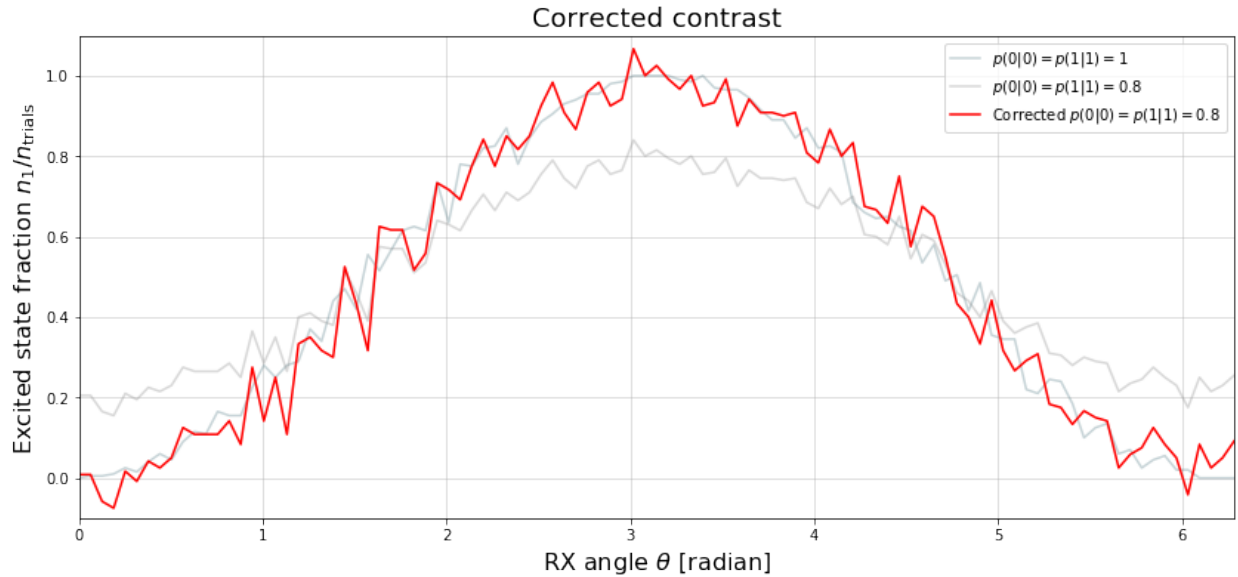
## Example 3: Correct for Noisy Readout

### 3a) Correcting the Rabi Signal from Above

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

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

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



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

### 3b) Corrupting and Correcting GHZ State Correlations

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
MEASURE 1 [1]
MEASURE 2 [2]
```

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

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

```
(0.5041999999999998, 0.49580000000000002)
```

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

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

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

```
(0.3086999999999997, 0.3644)
```

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

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

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

```
(0.50397601453064977, 0.49866843912900716)
```

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

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

*We expect these to all be very small*

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

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

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

*We expect these to all be close to 1.*

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

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

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

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

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

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

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

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

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

## 1.8 Advanced Usage

First, initialize a localQVM instance on your laptop. You should have two consoles open in your terminal to run in the background.

```
### CONSOLE 1
$ quilc -S
port triggered: 6000.
[2018-09-19 11:22:37] Starting server: 0.0.0.0 : 6000.

### CONSOLE 2
$ qvm -S
Welcome to the Rigetti QVM
(Configured with 2048 MiB of workspace and 8 workers.)
[2018-09-20 15:39:50] Starting server on port 5000.
```

```
from pyquil import Program, get_qc
from pyquil.gates import *
qvm = get_qc('9q-square-qvm')
```

Now that our local endpoints are up and running, we can start running pyQuil programs! Open a jupyter notebook (type `..code::jupyter notebook` in your terminal), or launch python in your terminal (type `..code::python3`).

### 1.8.1 Using Qubit Placeholders

In PyQuil, we typically use integers to identify qubits

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

```
H 0
CNOT 0 1
```

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

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

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

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

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

```
-----
RuntimeError                                Traceback (most recent call last)

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

...

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

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

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

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

```
H 0
CNOT 0 1
```

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

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

```
H 14
CNOT 14 19
```

## Register

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

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

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

### 1.8.2 Quantum Fourier Transform (QFT)

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

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

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

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

```
from math import pi

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

### 1.8.3 Classical Control Flow

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

1. We first initialize this register to 1 with the `init_register` program so our while loop will execute. This is often called the *loop preamble* or *loop initialization*.

2. Next, we write body of the loop in a program itself. This will be a program that computes an  $X$  followed by an  $H$  on our qubit.
3. Lastly, we put it all together using the `while_do` method.

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

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

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

print(loop_prog)
```

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

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

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

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

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

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

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

# Add the conditional branching:
```

(continues on next page)

(continued from previous page)

```
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]]
```

## 1.8.4 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 = qvm(gate_noise=gate_noise_probs, measurement_noise=meas_noise_probs)
```

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

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

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

## 1.8.5 Parametric Programs

In PyQuil 1.x, there was an object named `ParametricProgram`:

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

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

This object has been removed from PyQuil 2. Please consider simply using a Python function for the above functionality:

```
par_p = rotator
```

Or using declared classical memory:

```
p = Program()
angle = p.declare('angle', 'REAL')
p += RX(angle, 0)
```

## 1.8.6 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  $\beta$ .

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
```

(continues on next page)

(continued from previous page)

```

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

```

`exponential_map` returns a function allowing you to fill in a multiplicative constant later. 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  $\alpha$ .

```

expH = pl.exponential_map(H)
print(expH(0.0))
print(expH(1.0))
print(expH(2.0))

```

## 1.9 Exercises

### 1.9.1 Exercise 1: Quantum Dice

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

```

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

```

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

```

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

```

### 1.9.2 Exercise 2: Controlled Gates

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

1. Write a function `controlled` which takes a  $2 \times 2$  matrix  $U$  representing a single qubit operator, and makes a  $4 \times 4$  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.

### 1.9.3 Exercise 3: Grover's Algorithm

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

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

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

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

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

## 1.10 Source Code Documentation

### 1.10.1 pyquil.api

Module for facilitating connections to the QVM / QPU.

```
class pyquil.api.QVMConnection(device=None, endpoint=None, gate_noise=None, mea-
                                surement_noise=None, random_seed=None, com-
                                piler_endpoint=None)
```

Bases: `object`

Represents a connection to the QVM.

**expectation** (*prep\_prog*, *operator\_programs=None*)

Calculate the expectation value of operators given a state prepared by *prep\_program*.

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

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

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

#### Parameters

- **prep\_prog** (*Program*) – Quil program for state preparation.
- **operator\_programs** (*list*) – A list of Programs, each specifying an operator whose expectation to compute. Default is a list containing only the empty Program.

**Returns** Expectation values of the operators.

**Return type** List[float]

**get\_version\_info** ()

Return version information for the QVM.

**Returns** Dictionary with version information

**pauli\_expectation** (*prep\_prog*, *pauli\_terms*)

Calculate the expectation value of Pauli operators given a state prepared by *prep\_program*.

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

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

#### Parameters

- **prep\_prog** (`Program`) – Quil program for state preparation.
- **pauli\_terms** (`Sequence[PauliTerm] | PauliSum`) – A list of `PauliTerms` or a `PauliSum`.

**Returns** If *pauli\_terms* is a `PauliSum` return its expectation value. Otherwise return a list of expectation values.

**Return type** `float|List[float]`

**run** (*quil\_program*, *classical\_addresses*: `List[int] = None`, *trials*=1)

Run a Quil program multiple times, accumulating the values deposited in a list of classical addresses.

#### Parameters

- **quil\_program** (`Program`) – A Quil program.
- **classical\_addresses** – The classical memory to retrieve. Specified as a list of integers that index into a readout register named `ro`. This function—and particularly this argument—are included for backwards compatibility and will be removed in the future.
- **trials** (`int`) – Number of shots to collect.

**Returns** A list of dictionaries of bits. Each dictionary corresponds to the values in *classical\_addresses*.

**Return type** `list`

**run\_and\_measure** (*quil\_program*, *qubits*, *trials*=1)

Run a Quil program once to determine the final wavefunction, and measure multiple times.

**Note** If the execution of *quil\_program* is **non-deterministic**, i.e., if it includes measurements and/or noisy quantum gates, then the final wavefunction from which the returned bit-strings are sampled itself only represents a stochastically generated sample and the outcomes sampled from *different* *run\_and\_measure* calls *generally sample different bitstring distributions*.

#### Parameters

- **quil\_program** (`Program`) – A Quil program.
- **qubits** (`list | range`) – A list of qubits.
- **trials** (`int`) – Number of shots to collect.

**Returns** A list of a list of bits.

**Return type** `list`

**wavefunction** (*quil\_program*)

Simulate a Quil program and get the wavefunction back.

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

**Parameters** `quil_program` (`Program`) – A Quil program.

**Returns** A Wavefunction object representing the state of the QVM.

**Return type** *Wavefunction*

```
class pyquil.api.LocalQVMCompiler(endpoint: str, device: pyquil.device.AbstractDevice)
    Bases: pyquil.api._qac.AbstractCompiler
```

```
get_version_info() → dict
```

Return version information for this compiler and its dependencies.

**Returns** Dictionary of version information.

```
native_quil_to_executable(nq_program: pyquil.quil.Program)
```

Compile a native quil program to a binary executable.

**Parameters** `nq_program` – Native quil to compile

**Returns** An (opaque) binary executable

```
quil_to_native_quil(program: pyquil.quil.Program) → pyquil.quil.Program
```

Compile an arbitrary quil program according to the ISA of `target_device`.

**Parameters** `program` – Arbitrary quil to compile

**Returns** Native quil and compiler metadata

```
class pyquil.api.QVMCompiler(endpoint: str, device: pyquil.device.AbstractDevice)
    Bases: pyquil.api._qac.AbstractCompiler
```

```
get_version_info() → dict
```

Return version information for this compiler and its dependencies.

**Returns** Dictionary of version information.

```
native_quil_to_executable(nq_program: pyquil.quil.Program) →
    rpcq.messages.PyQuilExecutableResponse
```

Compile a native quil program to a binary executable.

**Parameters** `nq_program` – Native quil to compile

**Returns** An (opaque) binary executable

```
quil_to_native_quil(program: pyquil.quil.Program) → pyquil.quil.Program
```

Compile an arbitrary quil program according to the ISA of `target_device`.

**Parameters** `program` – Arbitrary quil to compile

**Returns** Native quil and compiler metadata

```
class pyquil.api.QPUCompiler(endpoint: str, device: pyquil.device.AbstractDevice)
    Bases: pyquil.api._qac.AbstractCompiler
```

```
get_version_info() → dict
```

Return version information for this compiler and its dependencies.

**Returns** Dictionary of version information.

```
native_quil_to_executable(nq_program: pyquil.quil.Program) →
    rpcq.messages.BinaryExecutableResponse
```

Compile a native quil program to a binary executable.

**Parameters** `nq_program` – Native quil to compile

**Returns** An (opaque) binary executable

**quil\_to\_native\_quil** (*program: pyquil.quil.Program*) → `pyquil.quil.Program`

Compile an arbitrary quil program according to the ISA of `target_device`.

**Parameters** `program` – Arbitrary quil to compile

**Returns** Native quil and compiler metadata

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

Bases: `object`

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

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

**compiled\_quil** ()

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

**Return type** Optional[*Program*]

**decode** ()

**estimated\_time\_left\_in\_queue** ()

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

**gate\_depth** ()

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

**Return type** Optional[int]

**gate\_volume** ()

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

**Return type** Optional[int]

**get** ()

**is\_compiling** ()

Is the job actively compiling?

**is\_done** ()

Has the job completed yet?

**is\_queued** ()

Is the job still in the Forest queue?

**is\_queued\_for\_compilation** ()

Is the job still in the Forest compilation queue?

**is\_running** ()

Is the job currently running?

**job\_id**

Job id :rtype: str

**multiqubit\_gate\_depth()**

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

**Return type** Optional[int]

**position\_in\_queue()**

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

**program\_fidelity()**

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

**Return type** Optional[float]

**result()**

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

**running\_time()**

For how long was the job running? :return: Running time, seconds :rtype: Optional[float]

**time\_in\_queue()**

For how long was the job in the Forest queue? :return: Time in queue, seconds :rtype: Optional[float]

**topological\_swaps()**

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

**Return type** Optional[int]

**class pyquil.api.Device(name, raw)**

Bases: `pyquil.device.AbstractDevice`

A device (quantum chip) that can accept programs.

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

**Variables**

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

**get\_isa(oneq\_type='Xhalves', twoq\_type='CZ') → pyquil.device.ISA**

Construct an ISA suitable for targeting by compilation.

This will raise an exception if the requested ISA is not supported by the device.

**Parameters**

- **oneq\_type** – The family of one-qubit gates to target
- **twoq\_type** – The family of two-qubit gates to target

```

get_specs()
    Construct a Specs object required by compilation

isa

qubit_topology() → networkx.classes.graph.Graph
    The connectivity of qubits in this device given as a NetworkX graph.

qubits()
    A sorted list of qubits in the device topology.

class pyquil.api.ForestConnection(sync_endpoint=None, compiler_endpoint=None, forest_cloud_endpoint=None)
    Bases: object

pyquil.api.pyquil_protect(func, log_filename='pyquil_error.log')
    A decorator that sets up an error context, captures errors, and tears down the context.

class pyquil.api.WavefunctionSimulator(connection: pyquil.api._base_connection.ForestConnection = None, random_seed: Optional[int] = None)
    Bases: object

expectation(prep_prog: pyquil.quil.Program, pauli_terms: Union[pyquil.paulis.PauliSum, List[pyquil.paulis.PauliTerm]]) → Union[float, numpy.ndarray]
    Calculate the expectation value of Pauli operators given a state prepared by prep_program.

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

```

---

**Note:** If your program contains measurements or noisy gates, this method may not do what you want. If the execution of `quil_program` is **non-deterministic** then the final wavefunction from which the expectation value is calculated only represents a stochastically generated sample and the wavefunctions returned by *different* wavefunction calls will generally be different.

---

### Parameters

- **prep\_prog** – A program that prepares the state on which we measure the expectation.
- **pauli\_terms** – A Pauli representation of a quantum operator.

**Returns** Either a float or array floats depending on `pauli_terms`.

```

run_and_measure(quil_program: pyquil.quil.Program, qubits: List[int] = None, trials: int = 1) → numpy.ndarray

```

Run a Quil program once to determine the final wavefunction, and measure multiple times.

Alternatively, consider using `wavefunction` and calling `sample_bitstrings` on the resulting object.

For a large wavefunction and a low-medium number of trials, use this function. On the other hand, if you're sampling a small system many times you might want to use `Wavefunction.sample_bitstrings`.

---

**Note:** If your program contains measurements or noisy gates, this method may not do what you want. If the execution of `quil_program` is **non-deterministic** then the final wavefunction from which the returned bitstrings are sampled itself only represents a stochastically generated sample and the outcomes sampled from *different* `run_and_measure` calls generally sample different bitstring distributions.

---

### Parameters

- **quil\_program** – The program to run and measure
- **qubits** – An optional list of qubits to measure. The order of this list is respected in the returned bitstrings. If not provided, all qubits used in the program will be measured and returned in their sorted order.
- **trials** (*int*) – Number of times to sample from the prepared wavefunction.

**Returns** An array of measurement results (0 or 1) of shape (trials, len(qubits))

**wavefunction** (*quil\_program: pyquil.quil.Program*) → *pyquil.wavefunction.Wavefunction*  
Simulate a Quil program and return the wavefunction.

---

**Note:** If your program contains measurements or noisy gates, this method may not do what you want. If the execution of `quil_program` is **non-deterministic** then the final wavefunction only represents a stochastically generated sample and the wavefunctions returned by *different* `wavefunction` calls will *generally be different*.

---

**Parameters** `quil_program` – A Quil program.

**Returns** A Wavefunction object representing the state of the QVM.

```
class pyquil.api.QuantumComputer(*, name: str, qam: pyquil.api._qam.QAM, de-
                                vice: pyquil.device.AbstractDevice, compiler:
                                pyquil.api._qac.AbstractCompiler, symmetrize_readout:
                                bool = False)
```

Bases: `object`

```
compile(program: pyquil.quil.Program, to_native_gates: bool = True, optimize: bool = True) →
    rpcq_base.Message
```

```
get_isa(oneq_type: str = 'Xhalves', twoq_type: str = 'CZ') → pyquil.device.ISA
```

```
qubit_topology() → <module 'networkx.classes.graph' from
    '/home/docs/checkouts/readthedocs.org/user_builds/pyquil/envs/v2.0.0/lib/python3.6/site-
    packages/networkx/classes/graph.py'>
```

```
qubits() → List[int]
```

```
run(executable: Union[rpcq.messages.BinaryExecutableResponse, rpcq.messages.PyQuilExecutableResponse])
    → numpy.ndarray
    Run a quil executable.
```

**Parameters** `executable` – The program to run. You are responsible for compiling this first.

**Returns** A numpy array of shape (trials, len(ro-register)) that contains 0s and 1s

```
run_and_measure(program: pyquil.quil.Program, trials: int) → Dict[int, numpy.ndarray]
    Run the provided state preparation program and measure all qubits.
```

This will measure all the qubits on this QuantumComputer, not just qubits that are used in the program.

The returned data is a dictionary keyed by qubit index because qubits for a given QuantumComputer may be non-contiguous and non-zero-indexed. To turn this dictionary into a 2d numpy array of bitstrings, consider:

```
bitstrings = qc.run_and_measure(...)
bitstring_array = np.vstack(bitstrings[q] for q in sorted(qc.qubits()))
bitstring_array.shape # (trials, len(qc.qubits()))
```

---

**Note:** In contrast to `QVMConnection.run_and_measure`, this method simulates noise correctly for noisy QVMs. However, this method is slower for `trials > 1`. For faster noise-free simulation, consider `WavefunctionSimulator.run_and_measure`.

---

### Parameters

- **program** – The state preparation program to run and then measure.
- **trials** – The number of times to run the program.

**Returns** A dictionary keyed by qubit index where the corresponding value is a 1D array of measured bits.

**run\_symmetrized\_readout** (*program: pyquil.quil.Program, trials: int*) → `numpy.ndarray`

Run a quil program in such a way that the readout error is made collectively symmetric

This means the probability of a bitstring `b` being mistaken for a bitstring `c` is the same as the probability of `not(b)` being mistaken for `not(c)`

A more general symmetrization would guarantee that the probability of `b` being mistaken for `c` depends only on which bit of `c` are different from `b`. This would require choosing random subsets of bits to flip.

In a noisy device, the probability of accurately reading the 0 state might be higher than that of the 1 state. This makes correcting for readout more difficult. This function runs the program normally (`trials//2`) times. The other half of the time, it will insert an X gate prior to any MEASURE instruction and then flip the measured classical bit back.

See `run()` for this function’s parameter descriptions.

`pyquil.api.list_quantum_computers` (*connection: pyquil.api.\_base\_connection.ForestConnection = None, qpus: bool = True, qvms: bool = True*) → `List[str]`

List the names of available quantum computers

### Parameters

- **connection** – An optional `py:class:ForestConnection` object. If not specified, the default values for URL endpoints will be used, and your API key will be read from `~/pyquil_config`. If you deign to change any of these parameters, pass your own `ForestConnection` object.
- **qpus** – Whether to include QPU’s in the list.
- **qvms** – Whether to include QVM’s in the list.

`pyquil.api.get_qc` (*name: str, \*, as\_qvm: bool = None, noisy: bool = None, connection: pyquil.api.\_base\_connection.ForestConnection = None*) → `pyquil.api._quantum_computer.QuantumComputer`

Get a quantum computer.

A quantum computer is an object of type `QuantumComputer` and can be backed either by a QVM simulator (“Quantum/Quil Virtual Machine”) or a physical Rigetti QPU (“Quantum Processing Unit”) made of superconducting qubits.

You can choose the quantum computer to target through a combination of its name and optional flags. There are multiple ways to get the same quantum computer. The following are equivalent:

```
>>> qc = get_qc("Aspen-0-12Q-A-noisy-qvm")
>>> qc = get_qc("Aspen-0-12Q-A", as_qvm=True, noisy=True)
```

and will construct a simulator of the 8q-agave chip with a noise model based on device characteristics. We also provide a means for constructing generic quantum simulators that are not related to a given piece of Rigetti hardware:

```
>>> qc = get_qc("9q-square-qvm")
>>> qc = get_qc("9q-square", as_qvm=True)
```

Finally, you can get request a QVM with “no” topology of a given number of qubits (technically, it’s a fully connected graph among the given number of qubits) with:

```
>>> qc = get_qc("5q-qvm") # or "6q-qvm", or "34q-qvm", ...
```

These less-realistic, fully-connected QVMs will also be more lenient on what types of programs they will run. Specifically, you do not need to do any compilation. For the other, realistic QVMs you must use `qc.compile()` or `qc.compiler.native_quil_to_executable()` prior to `qc.run()`.

Redundant flags are acceptable, but conflicting flags will raise an exception:

```
>>> qc = get_qc("9q-square-qvm") # qc is fully specified by its name
>>> qc = get_qc("9q-square-qvm", as_qvm=True) # redundant, but ok
>>> qc = get_qc("9q-square-qvm", as_qvm=False) # Error!
```

Use `list_quantum_computers()` to retrieve a list of known qc names.

This method is provided as a convenience to quickly construct and use QVM’s and QPU’s. Power users may wish to have more control over the specification of a quantum computer (e.g. custom noise models, bespoke topologies, etc.). This is possible by constructing a `QuantumComputer` object by hand. Please refer to the documentation on `QuantumComputer` for more information.

### Parameters

- **name** – The name of the desired quantum computer. This should correspond to a name returned by `list_quantum_computers()`. Names ending in “-qvm” will return a QVM. Names ending in “-noisy-qvm” will return a QVM with a noise model. Otherwise, we will return a QPU with the given name.
- **as\_qvm** – An optional flag to force construction of a QVM (instead of a QPU). If specified and set to `True`, a QVM-backed quantum computer will be returned regardless of the name’s suffix
- **noisy** – An optional flag to force inclusion of a noise model. If specified and set to `True`, a quantum computer with a noise model will be returned regardless of the name’s suffix. The noise model for QVM’s based on a real QPU is an empirically parameterized model based on real device noise characteristics. The generic QVM noise model is simple T1 and T2 noise plus readout error. See `decoherence_noise_with_asymmetric_ro()`.
- **connection** – An optional `:py:class:ForestConnection` object. If not specified, the default values for URL endpoints, ping time, and status time will be used. Your user id and API key will be read from `~/pyquil_config`. If you deign to change any of these parameters, pass your own `ForestConnection` object.

### Returns

```
class pyquil.api.QAM
    Bases: abc.ABC
```

The platonic ideal of this class is as a generic interface describing how a classical computer interacts with a live quantum computer. Eventually, it will turn into a thin layer over the QPU and QVM’s “QPI” interfaces.

The reality is that neither the QPU nor the QVM currently support a full-on QPI interface, and so the undignified job of this class is to collect enough state that it can convincingly pretend to be a QPI-compliant quantum computer.

**load** (*executable*)

Initialize a QAM into a fresh state.

**Parameters** **executable** – Load a compiled executable onto the QAM.

**read\_from\_memory\_region** (\*, *region\_name: str*)

Reads from a memory region named *region\_name* on the QAM.

This is a shim over the eventual API and only can return memory from a region named “ro” of type BIT.

**Parameters** **region\_name** – The string naming the declared memory region.

**Returns** A list of values of the appropriate type.

**read\_memory** (\*, *region\_name: str*)

Reads from a memory region named *region\_name* on the QAM.

This is a shim over the eventual API and only can return memory from a region named “ro” of type BIT.

**Parameters** **region\_name** – The string naming the declared memory region.

**Returns** A list of values of the appropriate type.

**run** ()

Reset the program counter on a QAM and run its loaded Quil program.

**wait** ()

Blocks until the QPU enters the halted state.

**write\_memory** (\*, *region\_name: str, offset: int = 0, value=None*)

Writes a value into a memory region on the QAM at a specified offset.

**Parameters**

- **region\_name** – Name of the declared memory region on the QAM.
- **offset** – Integer offset into the memory region to write to.
- **value** – Value to store at the indicated location.

```
class pyquil.api.QVM(connection: pyquil.api._base_connection.ForestConnection,
                        noise_model=None, gate_noise=None, measurement_noise=None, ran-
                        dom_seed=None, requires_executable=False)
```

Bases: `pyquil.api._qam.QAM`

**augment\_program\_with\_memory\_values** (*quil\_program*)

**get\_version\_info** ()

Return version information for the QVM.

**Returns** Dictionary with version information

**load** (*executable*)

Initialize a QAM and load a program to be executed with a call to `run()`.

If `QVM.requires_executable` is set to `True`, this function will only load `PyQuilExecutableResponse` executables. This more closely follows the behavior of `QPU`. However, the quantum simulator doesn’t *actually* need a compiled binary executable, so if this flag is set to `False` we also accept `Program` objects.

**Parameters** **executable** – An executable. See the above note for acceptable types.

**run()**

Run a Quil program on the QVM multiple times and return the values stored in the classical registers designated by the `classical_addresses` parameter.

**Returns** An array of bitstrings of shape `(trials, len(classical_addresses))`

**class** `pyquil.api.QPU(endpoint: str, user: str = 'pyquil-user')`

Bases: `pyquil.api._qam.QAM`

**get\_version\_info()** → dict

Return version information for this QPU's execution engine and its dependencies.

**Returns** Dictionary of version information.

**run()**

Run a pyquil program on the QPU.

This formats the classified data from the QPU server by stacking measured bits into an array of shape `(trials, classical_addresses)`. The mapping of qubit to classical address is backed out from MEASURE instructions in the program, so only do measurements where there is a 1-to-1 mapping between qubits and classical addresses.

**Returns** The QPU object itself.

**class** `pyquil.api.BenchmarkConnection(endpoint=None)`

Bases: `pyquil.api._qac.AbstractBenchmarker`

Represents a connection to a server that generates benchmarking data.

**apply\_clifford\_to\_pauli** (*clifford, pauli\_in*)

Given a circuit that consists only of elements of the Clifford group, return its action on a PauliTerm.

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

**Parameters**

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

**Returns** A PauliTerm corresponding to `pauli_in * clifford * pauli_in†`

**generate\_rb\_sequence** (*depth, gateset, seed=None*)

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

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

**Parameters**

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

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

**class** pyquil.api.LocalBenchmarkConnection (endpoint=None)

Bases: pyquil.api.\_qac.AbstractBenchmarker

Represents a connection to a locally-running server that generates randomized benchmarking data.

**apply\_clifford\_to\_pauli** (clifford, pauli\_in)

Given a circuit that consists only of elements of the Clifford group, return its action on a PauliTerm.

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

#### Parameters

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

**Returns** A PauliTerm corresponding to  $pauli\_in * clifford * pauli\_in^{\dagger}$

**generate\_rb\_sequence** (depth, gateset, seed=None)

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

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

#### Parameters

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

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

pyquil.api.get\_benchmarker (endpoint: str = None)

Retrieve an instance of the appropriate AbstractBenchmarker subclass for a given endpoint.

**Parameters** **endpoint** – Benchmarking sequence server address. Defaults to the setting in the user's pyQuil config.

**Returns** Instance of an AbstractBenchmarker subclass, connected to the given endpoint.

## 1.10.2 pyquil.device

**class** pyquil.device.AbstractDevice

Bases: abc.ABC

**get\_isa** (oneq\_type='Xhalves', twoq\_type='CZ') → pyquil.device.ISA

Construct an ISA suitable for targeting by compilation.

This will raise an exception if the requested ISA is not supported by the device.

#### Parameters

- **oneq\_type** – The family of one-qubit gates to target
- **twoq\_type** – The family of two-qubit gates to target

**get\_specs()** → pyquil.device.Specs

Construct a Specs object required by compilation

**qubit\_topology()** → networkx.classes.graph.Graph

The connectivity of qubits in this device given as a NetworkX graph.

**qubits()**

A sorted list of qubits in the device topology.

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

Bases: `pyquil.device.AbstractDevice`

A device (quantum chip) that can accept programs.

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

#### Variables

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

**get\_isa** (*oneq\_type='Xhalves', twoq\_type='CZ'*) → pyquil.device.ISA

Construct an ISA suitable for targeting by compilation.

This will raise an exception if the requested ISA is not supported by the device.

#### Parameters

- **oneq\_type** – The family of one-qubit gates to target
- **twoq\_type** – The family of two-qubit gates to target

**get\_specs()**

Construct a Specs object required by compilation

**isa**

**qubit\_topology()** → networkx.classes.graph.Graph

The connectivity of qubits in this device given as a NetworkX graph.

**qubits()**

A sorted list of qubits in the device topology.

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

Bases: `tuple`

**dead**

Alias for field number 2

**targets**

Alias for field number 0

**type**

Alias for field number 1

pyquil.device.EdgeSpecs

alias of pyquil.device.\_QubitQubitSpecs

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

Bases: pyquil.device.\_ISA

Basic Instruction Set Architecture specification.

#### Variables

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

**static from\_dict** (*d*)

Re-create the ISA from a dictionary representation.

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

**Returns** The restored ISA.

**Return type** *ISA*

**to\_dict** ()

Create a JSON-serializable representation of the ISA.

The dictionary representation is of the form:

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

**Returns** A dictionary representation of self.

**Return type** *Dict*[*str*, *Any*]

**class** pyquil.device.**NxDevice** (*topology: networkx.classes.graph.Graph*)

Bases: *pyquil.device.AbstractDevice*

A shim over the AbstractDevice API backed by a NetworkX graph.

A Device holds information about the physical device. Specifically, you might want to know about connectivity, available gates, performance specs, and more. This class implements the AbstractDevice API for devices not available via `get_devices()`. Instead, the user is responsible for constructing a NetworkX graph which represents a chip topology.

**edges** () → List[Tuple[int, int]]

**get\_isa** (oneq\_type='Xhalves', twoq\_type='CZ')

Construct an ISA suitable for targeting by compilation.

This will raise an exception if the requested ISA is not supported by the device.

#### Parameters

- **oneq\_type** – The family of one-qubit gates to target
- **twoq\_type** – The family of two-qubit gates to target

**get\_specs** ()

Construct a Specs object required by compilation

**qubit\_topology** ()

The connectivity of qubits in this device given as a NetworkX graph.

**qubits** () → List[int]

A sorted list of qubits in the device topology.

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

Bases: tuple

**dead**

Alias for field number 2

**id**

Alias for field number 0

**type**

Alias for field number 1

pyquil.device.QubitSpecs

alias of pyquil.device.\_QubitSpecs

**class** pyquil.device.Specs

Bases: pyquil.device.\_Specs

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

#### Variables

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

**T1s** ()

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

**Returns** A dictionary of T1s, in seconds.

**Return type** Dict[int, float]

**T2s** ()

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

**Returns** A dictionary of T2s, in seconds.

**Return type** Dict[int, float]

**f1QRBs** ()

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

**Returns** A dictionary of 1QRBs, normalized to unity.

**Return type** Dict[int, float]

**fActiveResets** ()

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

**Returns** A dictionary of reset fidelities, normalized to unity.

**fBellStates** ()

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

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

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

**fCPHASEs** ()

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

**Returns** A dictionary of CPHASE fidelities, normalized to unity.

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

**fCZs** ()

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

**Returns** A dictionary of CZ fidelities, normalized to unity.

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

**fROs** ()

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

**Returns** A dictionary of RO fidelities, normalized to unity.

**Return type** Dict[int, float]

**static from\_dict** (d)

Re-create the Specs from a dictionary representation.

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

**Returns** The restored Specs.

**Return type** Specs

**to\_dict** ()

Create a JSON-serializable representation of the device Specs.

The dictionary representation is of the form:

```
{
  '1Q': {
    '0': {
      'f1QRB': 0.99,
      'T1': 20e-6,
      ...
    },
    '1': {
      'f1QRB': 0.989,
      'T1': 19e-6,
```

(continues on next page)

(continued from previous page)

```

        ...
    },
    ...
},
'2Q': {
    "1-4": {
        "fBellState": 0.93,
        "fCZ": 0.92,
        "fCPHASE": 0.91
    },
    "1-5": {
        "fBellState": 0.9,
        "fCZ": 0.89,
        "fCPHASE": 0.88
    },
    ...
},
...
}

```

**Returns** A dictionary representation of self.

**Return type** Dict[str, Any]

`pyquil.device.THETA = Parameter('theta')`

Used as the symbolic parameter in RZ, CPHASE gates.

`pyquil.device.gates_in_isa(isa)`

Generate the full gateset associated with an ISA.

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

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

**Return type** Sequence[Gate]

`pyquil.device.isa_from_graph(graph: networkx.classes.graph.Graph, oneq_type='Xhalves', twoq_type='CZ') → pyquil.device.ISA`

Generate an ISA object from a NetworkX graph.

**Parameters**

- **graph** – The graph
- **oneq\_type** – The type of 1-qubit gate. Currently 'Xhalves'
- **twoq\_type** – The type of 2-qubit gate. One of 'CZ' or 'CPHASE'.

`pyquil.device.isa_to_graph(isa: pyquil.device.ISA) → networkx.classes.graph.Graph`

Construct a NetworkX qubit topology from an ISA object.

This discards information about supported gates.

**Parameters** `isa` – The ISA.

`pyquil.device.specs_from_graph(graph: networkx.classes.graph.Graph)`

Generate a Specs object from a NetworkX graph with placeholder values for the actual specs.

**Parameters** `graph` – The graph

### 1.10.3 pyquil.gates

`pyquil.gates.I(qubit)`

Produces the I instruction.

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

This gate is a single qubit identity gate. Note that this gate is different than 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.X(qubit)`

Produces the X instruction.

$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

This gate is a single qubit X-gate.

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

**Returns** A Gate object.

`pyquil.gates.Y(qubit)`

Produces the Y instruction.

$Y = \begin{bmatrix} 0 & -1j \\ 1j & 0 \end{bmatrix}$

This gate is a single qubit Y-gate.

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

**Returns** A Gate object.

`pyquil.gates.Z(qubit)`

Produces the Z instruction.

$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

This gate is a single qubit Z-gate.

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

**Returns** A Gate object.

`pyquil.gates.H(qubit)`

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

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.S(qubit)`

Produces the S instruction.

$S = \begin{bmatrix} 1 & 0 \\ 0 & 1j \end{bmatrix}$

This gate is a single qubit S-gate.

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

**Returns** A Gate object.

`pyquil.gates.T(qubit)`

Produces the T instruction.

**T** =  $[[1, 0], [0, \exp(1j * \pi / 4)]]$

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.PHASE(angle, qubit)`

Produces the PHASE instruction.

**PHASE(phi)** =  $[[1, 0], [0, \exp(1j * \phi)]]$

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.RX(angle, qubit)`

Produces the RX instruction.

**RX(phi)** =  $[[\cos(\phi / 2), -1j * \sin(\phi / 2)], [-1j * \sin(\phi / 2), \cos(\phi / 2)]]$

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(angle, qubit)`

Produces the RY instruction.

**RY(phi)** =  $[[\cos(\phi / 2), -\sin(\phi / 2)], [\sin(\phi / 2), \cos(\phi / 2)]]$

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(angle, qubit)`

Produces the RZ instruction.

**RZ(phi)** =  $[[\cos(\phi / 2) - 1j * \sin(\phi / 2), 0], [0, \cos(\phi / 2) + 1j * \sin(\phi / 2)]]$

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.CZ` (*control, target*)

Produces a CZ instruction.

**CZ** = `[[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, -1]]`

This gate applies to two qubit arguments to produce the controlled-Z gate instruction.

#### Parameters

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

**Returns** A Gate object.

`pyquil.gates.CNOT` (*control, target*)

Produces a CNOT instruction.

**CNOT** = `[[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 1], [0, 0, 1, 0]]`

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.CCNOT` (*control1, control2, target*)

Produces a CCNOT instruction.

**CCNOT** = `[[1, 0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 0, 1]]`

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

#### Parameters

- **control1** – The first control qubit.
- **control2** – 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.CPHASE00` (*angle, control, target*)

Produces a CPHASE00 instruction.

**CPHASE00**(phi) = `diag([exp(1j * phi), 1, 1, 1])`

This gate applies to two qubit arguments to produce the variant of the controlled phase instruction that affects the state 00.

#### Parameters

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

**Returns** A Gate object.

`pyquil.gates.CPHASE01` (*angle, control, target*)

Produces a CPHASE01 instruction.

$\text{CPHASE01}(\phi) = \text{diag}([1.0, \exp(1j * \phi), 1.0, 1.0])$

This gate applies to two qubit arguments to produce the variant of the controlled phase instruction that affects the state 01.

**Parameters**

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

**Returns** A Gate object.

`pyquil.gates.CPHASE10` (*angle, control, target*)

Produces a CPHASE10 instruction.

$\text{CPHASE10}(\phi) = \text{diag}([1, 1, \exp(1j * \phi), 1])$

This gate applies to two qubit arguments to produce the variant of the controlled phase instruction that affects the state 10.

**Parameters**

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

**Returns** A Gate object.

`pyquil.gates.CPHASE` (*angle, control, target*)

Produces a CPHASE instruction, which is a synonym for CPHASE11.

$\text{CPHASE}(\phi) = \text{diag}([1, 1, 1, \exp(1j * \phi)])$

This gate applies to two qubit arguments to produce the variant of the controlled phase instruction that affects the state 11.

**Parameters**

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

**Returns** A Gate object.

`pyquil.gates.SWAP` (*q1, q2*)

Produces a SWAP instruction.

$\text{SWAP} = [[1, 0, 0, 0],$   
 $[0, 0, 1, 0], [0, 1, 0, 0], [0, 0, 0, 1]]$

This gate swaps the state of two qubits.

**Parameters**

- **q1** – Qubit 1.

- **q2** – Qubit 2.

**Returns** A Gate object.

`pyquil.gates.CSWAP` (*control*, *target\_1*, *target\_2*)

**CSWAP** = `[[1, 0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 0, 1]]`

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

#### Parameters

- **control** – The control qubit.
- **target-1** – The first target qubit.
- **target-2** – The second target qubit. The two target states are swapped if the control is in the excited state.

`pyquil.gates.ISWAP` (*q1*, *q2*)

Produces an ISWAP instruction.

**ISWAP** = `[[1, 0, 0, 0], [0, 0, 1j, 0], [0, 1j, 0, 0], [0, 0, 0, 1]]`

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.PSWAP` (*angle*, *q1*, *q2*)

Produces a PSWAP instruction.

**PSWAP(phi)** = `[[1, 0, 0, 0], [0, 0, exp(1j * phi), 0], [0, exp(1j * phi), 0, 0], [0, 0, 0, 1]]`

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.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.RESET` (*qubit\_index=None*)

Reset all qubits or just a specific qubit at *qubit\_index*.

**Parameters** **qubit\_index** (*Optional[int]*) – The address of the qubit to reset. If None, reset all qubits.

**Returns** A Reset or ResetQubit Quil AST expression corresponding to a global or targeted reset, respectively.

**Return type** Union[*Reset*, *ResetQubit*]

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

This instruction ends the program.

**Returns** A Halt 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.TRUE (classical_reg)`

Produce a TRUE instruction.

**Parameters** **classical\_reg** – A classical register to modify.

**Returns** An instruction object representing the equivalent MOVE.

`pyquil.gates.FALSE (classical_reg)`

Produce a FALSE instruction.

**Parameters** **classical\_reg** – A classical register to modify.

**Returns** An instruction object representing the equivalent MOVE.

`pyquil.gates.NOT (classical_reg)`

Produce a NOT instruction.

**Parameters** **classical\_reg** – A classical register to modify.

**Returns** A ClassicalNot instance.

`pyquil.gates.AND (classical_reg1, classical_reg2)`

Produce an AND instruction.

NOTE: The order of operands was reversed in pyQuil <=1.9 .

**Parameters**

- **classical\_reg1** – The first classical register, which gets modified.
- **classical\_reg2** – The second classical register or immediate value.

**Returns** A ClassicalAnd instance.

`pyquil.gates.OR (classical_reg1, classical_reg2)`

Produce an OR instruction.

NOTE: Deprecated. Use IOR instead.

**Parameters**

- **classical\_reg1** – The first classical register.

- **classical\_reg2** – The second classical register, which gets modified.

**Returns** A ClassicalOr instance.

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

**Parameters**

- **classical\_reg1** – The first classical register, which gets modified.
- **classical\_reg2** – The second classical register or immediate value.

**Returns** A ClassicalMove instance.

`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.IOR(classical_reg1, classical_reg2)`  
Produce an inclusive OR instruction.

**Parameters**

- **classical\_reg1** – The first classical register, which gets modified.
- **classical\_reg2** – The second classical register or immediate value.

**Returns** A ClassicalOr instance.

`pyquil.gates.XOR(classical_reg1, classical_reg2)`  
Produce an exclusive OR instruction.

**Parameters**

- **classical\_reg1** – The first classical register, which gets modified.
- **classical\_reg2** – The second classical register or immediate value.

**Returns** A ClassicalOr instance.

`pyquil.gates.NEG(classical_reg)`  
Produce a NEG instruction.

**Parameters** **classical\_reg** – A classical memory address to modify.

**Returns** A ClassicalNeg instance.

`pyquil.gates.ADD(classical_reg, right)`  
Produce an ADD instruction.

**Parameters**

- **classical\_reg** – Left operand for the arithmetic operation. Also serves as the store target.
- **right** – Right operand for the arithmetic operation.

**Returns** A ClassicalAdd instance.

`pyquil.gates.SUB(classical_reg, right)`  
Produce a SUB instruction.

**Parameters**

- **classical\_reg** – Left operand for the arithmetic operation. Also serves as the store target.
- **right** – Right operand for the arithmetic operation.

**Returns** A ClassicalSub instance.

`pyquil.gates.MUL(classical_reg, right)`  
Produce a MUL instruction.

**Parameters**

- **classical\_reg** – Left operand for the arithmetic operation. Also serves as the store target.
- **right** – Right operand for the arithmetic operation.

**Returns** A ClassicalMul instance.

`pyquil.gates.DIV(classical_reg, right)`  
Produce an DIV instruction.

**Parameters**

- **classical\_reg** – Left operand for the arithmetic operation. Also serves as the store target.
- **right** – Right operand for the arithmetic operation.

**Returns** A ClassicalDiv instance.

`pyquil.gates.EQ(classical_reg1, classical_reg2, classical_reg3)`  
Produce an EQ instruction.

**Parameters**

- **classical\_reg1** – Memory address to which to store the comparison result.
- **classical\_reg2** – Left comparison operand.
- **classical\_reg3** – Right comparison operand.

**Returns** A ClassicalEqual instance.

`pyquil.gates.GT(classical_reg1, classical_reg2, classical_reg3)`  
Produce an GT instruction.

**Parameters**

- **classical\_reg1** – Memory address to which to store the comparison result.
- **classical\_reg2** – Left comparison operand.
- **classical\_reg3** – Right comparison operand.

**Returns** A ClassicalGreaterThan instance.

`pyquil.gates.GE(classical_reg1, classical_reg2, classical_reg3)`  
Produce an GE instruction.

**Parameters**

- **classical\_reg1** – Memory address to which to store the comparison result.
- **classical\_reg2** – Left comparison operand.
- **classical\_reg3** – Right comparison operand.

**Returns** A ClassicalGreaterEqual instance.

`pyquil.gates.LE(classical_reg1, classical_reg2, classical_reg3)`  
Produce an LE instruction.

**Parameters**

- **classical\_reg1** – Memory address to which to store the comparison result.
- **classical\_reg2** – Left comparison operand.
- **classical\_reg3** – Right comparison operand.

**Returns** A ClassicalLessEqual instance.

`pyquil.gates.LT(classical_reg1, classical_reg2, classical_reg3)`  
Produce an LT instruction.

**Parameters**

- **classical\_reg1** – Memory address to which to store the comparison result.
- **classical\_reg2** – Left comparison operand.
- **classical\_reg3** – Right comparison operand.

**Returns** A ClassicalLessThan instance.

`pyquil.gates.LOAD(target_reg, region_name, offset_reg)`  
Produce a LOAD instruction.

**Parameters**

- **target\_reg** – LOAD storage target.
- **region\_name** – Named region of memory to load from.
- **offset\_reg** – Offset into region of memory to load from. Must be a MemoryReference.

**Returns** A ClassicalLoad instance.

`pyquil.gates.STORE(region_name, offset_reg, source)`  
Produce a STORE instruction.

**Parameters**

- **region\_name** – Named region of memory to store to.
- **offset\_reg** – Offset into memory region. Must be a MemoryReference.
- **source** – Source data. Can be either a MemoryReference or a constant.

**Returns** A ClassicalStore instance.

`pyquil.gates.CONVERT(classical_reg1, classical_reg2)`  
Produce a CONVERT instruction.

**Parameters**

- **classical\_reg1** – MemoryReference to store to.
- **classical\_reg2** – MemoryReference to read from.

**Returns** A ClassicalCONVERT instance.

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

This is the pyQuil object for a quantum gate instruction.

```
get_qubits (indices=True)
out ()
```

### 1.10.4 pyquil.noise

Module for creating and verifying noisy gate and readout definitions.

`pyquil.noise.INFINITY = inf`  
Used for infinite coherence times.

**class** `pyquil.noise.KrausModel`  
Bases: `pyquil.noise._KrausModel`  
Encapsulate a single gate's noise model.

#### Variables

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

**static** `from_dict(d)`  
Recreate a KrausModel from the dictionary representation.

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

**Returns** The deserialized KrausModel.

**Return type** *KrausModel*

`to_dict()`  
Create a dictionary representation of a KrausModel.

For example:

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

**Returns** A JSON compatible dictionary representation.

**Return type** `Dict[str, Any]`

**static unpack\_kraus\_matrix** (*m*)

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

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

**Returns** A complex square numpy array representing the Kraus operator.

**Return type** *np.array*

**class** *pyquil.noise.NoiseModel*

Bases: *pyquil.noise.\_NoiseModel*

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

**Variables**

- **gates** (*Sequence[KrausModel]*) – The tomographic estimates of all gates.
- **assignment\_probs** (*Dict[int, np.array]*) – The single qubit readout assignment probability matrices keyed by qubit id.

**static from\_dict** (*d*)

Re-create the noise model from a dictionary representation.

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

**Returns** The restored noise model.

**Return type** *NoiseModel*

**gates\_by\_name** (*name*)

Return all defined noisy gates of a particular gate name.

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

**Returns** A list of noise models representing that gate.

**Return type** *Sequence[KrausModel]*

**to\_dict** ()

Create a JSON serializable representation of the noise model.

For example:

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

**Returns** A dictionary representation of self.

**Return type** *Dict[str, Any]*

**exception** `pyquil.noise.NoisyGateUndefined`

Bases: `Exception`

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

`pyquil.noise.add_decoherence_noise` (*prog*, *T1*= $3e-05$ , *T2*= $3e-05$ , *gate\_time\_1q*= $5e-08$ ,  
*gate\_time\_2q*= $1.5e-07$ , *ro\_fidelity*=0.95)

Add generic damping and dephasing noise to a program.

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

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

The default noise parameters

- $T1 = 30$  us
- $T2 = 30$  us
- 1q gate time = 50 ns
- 2q gate time = 150 ns

are currently typical for near-term devices.

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

#### Parameters

- **prog** – A pyquil program consisting of  $I$ ,  $RZ$ ,  $CZ$ , and  $RX(+\pi/2)$  instructions
- **T1** (`Union[Dict[int, float], float]`) – The  $T1$  amplitude damping time either globally or in a dictionary indexed by qubit id. By default, this is 30 us.
- **T2** (`Union[Dict[int, float], float]`) – The  $T2$  dephasing time either globally or in a dictionary indexed by qubit id. By default, this is also 30 us.
- **gate\_time\_1q** (`float`) – The duration of the one-qubit gates, namely  $RX(+\pi/2)$  and  $RX(-\pi/2)$ . By default, this is 50 ns.
- **gate\_time\_2q** (`float`) – The duration of the two-qubit gates, namely  $CZ$ . By default, this is 150 ns.
- **ro\_fidelity** (`Union[Dict[int, float], float]`) – The readout assignment fidelity  $F = (p(0|0) + p(1|1))/2$  either globally or in a dictionary indexed by qubit id.

**Returns** A new program with noisy operators.

`pyquil.noise.append_kraus_to_gate` (*kraus\_ops*, *gate\_matrix*)

Follow a gate *gate\_matrix* by a Kraus map described by *kraus\_ops*.

#### Parameters

- **kraus\_ops** (`list`) – The Kraus operators.
- **gate\_matrix** (`numpy.ndarray`) – The unitary gate.

**Returns** A list of transformed Kraus operators.

`pyquil.noise.apply_noise_model` (*prog*, *noise\_model*)

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

**Parameters**

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

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

**Return type** `Program`

`pyquil.noise.bitstring_probs_to_z_moments(p)`

Convert between bitstring probabilities and joint Z moment expectations.

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

**Returns**

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

$$\langle Z_0^{j_0} Z_1^{j_1} \dots Z_m^{j_m} \rangle = z\_moments[j\_0, j\_1, \dots, j\_m]$$

**Return type** `np.array`

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

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

**Parameters**

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

**Returns** A combinatorially generated list of composed Kraus operators.

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

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

**Parameters**

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

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

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

**Return type** `np.array`

`pyquil.noise.corrupt_bitstring_probs` (*p*, *assignment\_probabilities*)

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

**Parameters**

- **p** (*np.array*) – An array that enumerates bitstring probabilities. When flattened out  $p = [p_{00} \dots 0, p_{00} \dots 1, \dots, p_{11} \dots 1]$ . The total number of elements must therefore be a power of 2. The canonical shape has a separate axis for each qubit, such that  $p[i, j, \dots, k]$  gives the estimated probability of bitstring  $ij \dots k$ .
- **assignment\_probabilities** (*List[np.array]*) – A list of assignment probability matrices per qubit. Each assignment probability matrix is expected to be of the form:

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

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

**Return type** `np.array`

`pyquil.noise.damping_after_dephasing` (*T1*, *T2*, *gate\_time*)

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

**Parameters**

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

**Returns** A list of Kraus operators.

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

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

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

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

**Return type** `list`

`pyquil.noise.decoherence_noise_with_asymmetric_ro` (*gates:* Se-  
*quence*[*pyquil.quilbase.Gate*],  
*p00=0.975*, *p11=0.911*)

Similar to `:py:func`_decoherence_noise_model``, but with asymmetric readout.

For simplicity, we use the default values for T1, T2, gate times, et al. and only allow the specification of readout fidelities.

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

Generate the Kraus operators corresponding to a dephasing channel.

**Params** **float p** The one-step dephasing probability.

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

**Return type** `list`

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

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

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

#### Parameters

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

**Returns** The assignment probability matrix

**Return type** `np.array`

`pyquil.noise.estimate_bitstring_probs(results)`

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

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

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

**Return type** `np.array`

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

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

#### Parameters

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

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

**Return type** `Tuple[np.array, str]`

`pyquil.noise.pauli_kraus_map(probabilities)`

Generate the Kraus operators corresponding to a pauli channel.

**Params** `listfloats probabilities` The  $4^{\text{num\_qubits}}$  list of probabilities specifying the desired pauli channel.

There should be either 4 or 16 probabilities specified in the order I, X, Y, Z for 1 qubit or II, IX, IY, IZ, XI, XX, XY, etc for 2 qubits.

For example:

```
The d-dimensional depolarizing channel \Delta parameterized as
\Delta(\rho) = p \rho + [(1-p)/d] I
is specified by the list of probabilities
[p + (1-p)/d, (1-p)/d, (1-p)/d, ... , (1-p)/d]
```

**Returns** A list of the  $4^{\text{num\_qubits}}$  Kraus operators that parametrize the map.

**Return type** `list`

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

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

**Parameters**

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

**Returns** A list of tensored Kraus operators.

## 1.10.5 pyquil.parser

Module for parsing Quil programs from text into PyQuil objects

`pyquil.parser.parse(quil)`

Parse a raw Quil program and return a corresponding list of PyQuil objects.

**Parameters** **quil** (`str`) – a single or multiline Quil program

**Returns** list of instructions

`pyquil.parser.parse_program(quil)`

Parse a raw Quil program and return a PyQuil program.

**Parameters** **quil** (`str`) – a single or multiline Quil program

**Returns** PyQuil Program object

## 1.10.6 pyquil.paulis

Module for working with Pauli algebras.

`pyquil.paulis.HASH_PRECISION = 1000000.0`

The precision used when hashing terms to check equality. The `simplify()` method uses `np.isclose()` for coefficient comparisons to 0 which has its own default precision. We can't use `np.isclose()` for hashing terms though.

`pyquil.paulis.ID()`

The identity Pauli Term.

**class** `pyquil.paulis.PauliSum(terms)`

Bases: `object`

A sum of one or more PauliTerms.

**get\_programs()**

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

**Returns** (programs, coefficients)

**get\_qubits()**

The support of all the operators in the PauliSum object.

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

**Return type** `list`

**simplify()**

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

```
class pyquil.paulis.PauliTerm(op, index, coefficient=1.0)
```

Bases: `object`

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

```
copy()
```

Properly creates a new PauliTerm, with a completely new dictionary of operators

```
classmethod from_list(terms_list, coefficient=1.0)
```

Allocates a Pauli Term from a list of operators and indices. This is more efficient than multiplying together individual terms.

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

**Returns** PauliTerm

```
get_qubits()
```

Gets all the qubits that this PauliTerm operates on.

```
id(sort_ops=True)
```

Returns an identifier string for the PauliTerm (ignoring the coefficient).

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

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

**Returns** A string representation of this term’s operations.

**Return type** string

```
operations_as_set()
```

Return a frozenset of operations in this term.

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

**Returns** frozenset of strings representing Pauli operations

```
pauli_string(qubits=None)
```

Return a string representation of this PauliTerm mod its phase, as a concatenation of the string representation of the >>> p = PauliTerm(“X”, 0) \* PauliTerm(“Y”, 1, 1.j) >>> p.pauli\_string() “XY” >>> p.pauli\_string([0]) “X” >>> p.pauli\_string([0, 2]) “XI”

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

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

**Return type** String

```
program
```

```
exception pyquil.paulis.UnequalLengthWarning(*args, **kwargs)
```

Bases: `Warning`

```
pyquil.paulis.ZERO()
```

The zero Pauli Term.

```
pyquil.paulis.check_commutation(pauli_list, pauli_two)
```

Check if commuting a PauliTerm commutes with a list of other terms by natural calculation. Derivation similar to arXiv:1405.5749v2 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*)

Gather the Pauli terms of pauli\_terms variable into commuting sets

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

**Parameters** **pauli\_terms** (*PauliSum*) – A PauliSum object

**Returns** List of lists where each list contains a commuting set

**Return type** *list*

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

Creates map  $\alpha \rightarrow \exp(-1j*\alpha*term)$  represented as a Program.

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

**Returns** Program

**Return type** Function

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

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

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

**Returns** A Program object

**Return type** *Program*

`pyquil.paulis.exponentiate_commuting_pauli_sum` (*pauli\_sum*)

Returns a function that maps all substituent PauliTerms and sums them into a program. NOTE: Use this function with care. Substituent PauliTerms should commute.

**Parameters** **pauli\_sum** (*PauliSum*) – PauliSum to exponentiate.

**Returns** A function that parametrizes the exponential.

**Return type** function

`pyquil.paulis.integer_types` = (`<class 'int'>`, `<class 'numpy.int64'>`, `<class 'numpy.int32'>`),

Explicitly include numpy integer dtypes (for python 3).

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

Check if Pauli Term is a scalar multiple of identity

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

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

**Return type** *bool*

`pyquil.paulis.is_zero` (*pauli\_object*)

Tests to see if a PauliTerm or PauliSum is zero.

**Parameters** **pauli\_object** – Either a PauliTerm or PauliSum

**Returns** True if PauliTerm is zero, False otherwise

**Return type** *bool*

`pyquil.paulis.SI(q)`

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

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

**Returns** A PauliTerm object

**Return type** *PauliTerm*

`pyquil.paulis.SX(q)`

A function that returns the sigma\_X operator on a particular qubit.

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

**Returns** A PauliTerm object

**Return type** *PauliTerm*

`pyquil.paulis.SY(q)`

A function that returns the sigma\_Y operator on a particular qubit.

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

**Returns** A PauliTerm object

**Return type** *PauliTerm*

`pyquil.paulis.SZ(q)`

A function that returns the sigma\_Z operator on a particular qubit.

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

**Returns** A PauliTerm object

**Return type** *PauliTerm*

`pyquil.paulis.simplify_pauli_sum(pauli_sum)`

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

Generate trotterization coefficients for a given number of Trotter steps.

$U = \exp(A + B)$  is approximated as  $\exp(w1*o1)\exp(w2*o2)\dots$ . This method returns a list [(w1, o1), (w2, o2), ... , (wm, om)] of tuples where o=0 corresponds to the A operator, o=1 corresponds to the B operator, and w is the coefficient in the exponential. For example, a second order Suzuki-Trotter approximation to  $\exp(A + B)$  results in the following [(0.5/trotter\_steps, 0), (1/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` (*Number*) – The coefficient to set on the PauliTerm

**Returns** A new PauliTerm that duplicates term but sets coeff

**Return type** *PauliTerm*

`pyquil.paulis.trotterize` (*first\_pauli\_term*, *second\_pauli\_term*, *trotter\_order=1*, *trotter\_steps=1*)

Create a Quil program that approximates  $\exp((A + B)t)$  where *A* and *B* are `PauliTerm` operators.

#### Parameters

- **first\_pauli\_term** (`PauliTerm`) – `PauliTerm` denoted *A*
- **second\_pauli\_term** (`PauliTerm`) – `PauliTerm` denoted *B*
- **trotter\_order** (*int*) – Optional argument indicating the Suzuki-Trotter approximation order—only accepts orders 1, 2, 3, 4.
- **trotter\_steps** (*int*) – Optional argument indicating the number of products to decompose the exponential into.

**Returns** Quil program

**Return type** `Program`

### 1.10.7 pyquil.quil

Module for creating and defining Quil programs.

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

Bases: `object`

**alloc** ()

Get a new qubit.

**Returns** A qubit.

**Return type** `Qubit`

**copy** ()

Perform a shallow copy of this program.

`QuilAtom` and `AbstractInstruction` objects should be treated as immutable to avoid strange behavior when performing a copy.

**Returns** a new `Program`

**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`

**declare** (*name*, *memory\_type='BIT'*, *memory\_size=1*, *shared\_region=None*, *offsets=None*)

DECLARE a quil variable

This adds the declaration to the current program and returns a `MemoryReference` to the base (offset = 0) of the declared memory.

---

**Note:** This function returns a `MemoryReference` and cannot be chained like some of the other `Program` methods. Consider using `inst(DECLARE(...))` if you would like to chain methods, but please be aware that you must create your own `MemoryReferences` later on.

---

#### Parameters

- **name** – Name of the declared variable

- **memory\_type** – Type of the declared variable
- **memory\_size** – Number of array elements in the declared memory.
- **shared\_region** – You can declare a variable that shares its underlying memory with another region. This allows aliasing. For example, you can interpret an array of measured bits as an integer.
- **offsets** – If you are using `shared_region`, this allows you to share only a part of the parent region. The offset is given by an array type and the number of elements of that type. For example, `DECLARE target-bit BIT SHARING real-region OFFSET 1 REAL 4 BIT` will let you use `target-bit` to poke into the fourth bit of the second real from the leading edge of `real-region`.

**Returns** a `MemoryReference` to the start of the declared memory region, ie a memory reference to `name[0]`.

**defgate** (*name*, *matrix*, *parameters=None*)

Define a new static gate.

---

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

---

#### Parameters

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

**Returns** The Program instance.

**Return type** *Program*

**define\_noisy\_gate** (*name*, *qubit\_indices*, *kraus\_ops*)

Overload a static ideal gate with a noisy one defined in terms of a Kraus map.

---

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

---

#### Parameters

- **name** (*str*) – The name of the gate.
- **qubit\_indices** (*tuple/list*) – The qubits it acts on.
- **kraus\_ops** (*tuple/list*) – The Kraus operators.

**Returns** The Program instance

**Return type** *Program*

**define\_noisy\_readout** (*qubit*, *p00*, *p11*)

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

**Parameters**

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

**Returns** The Program with an appended READOUT-POVM Pragma.

**Return type** *Program*

**defined\_gates**

A list of defined gates on the program.

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

Add a gate to the program.

---

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

---

**Parameters**

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

**Returns** The Program instance

**Return type** *Program*

**get\_qubits** (*indices=True*)

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

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

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

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

**Return type** *set*

**if\_then** (*classical\_reg*, *if\_program*, *else\_program=None*)

If the classical register at index *classical\_reg* is 1, run *if\_program*, else run *else\_program*.

Equivalent to the following construction:

```
IF [c]:
    instrA...
ELSE:
    instrB...
=>
    JUMP-WHEN @THEN [c]
    instrB...
    JUMP @END
    LABEL @THEN
    instrA...
    LABEL @END
```

#### Parameters

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

**Returns** The Quil Program with the branching instructions added.

**Return type** *Program*

**inst** (*\*instructions*)

Mutates the Program object by appending new instructions.

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

```
>>> p = Program()
>>> p.inst(H(0)) # A single instruction
>>> p.inst(H(0), H(1)) # Multiple instructions
>>> p.inst([H(0), H(1)]) # A list of instructions
>>> p.inst(H(i) for i in range(4)) # A generator of instructions
>>> p.inst(("H", 1)) # A tuple representing an instruction
>>> p.inst("H 0") # A string representing an instruction
>>> q = Program()
>>> p.inst(q) # Another program
```

**It can also be chained:**

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

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

**Returns** self for method chaining

**instructions**

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

**is\_protoquil** ()

Protoquil programs may only contain gates, Pragmas, and final global RESETs. It may not contain classical instructions or jumps.

**Returns** True if the Program is Protoquil, False otherwise

**measure** (*qubit\_index*, *classical\_reg=None*)

Measures a qubit at *qubit\_index* and puts the result in *classical\_reg*

**Parameters**

- **qubit\_index** (*int*) – The address of the qubit to measure.
- **classical\_reg** (*int*) – The address of the classical bit to store the result.

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

**Return type** *Program*

**measure\_all** (*\*qubit\_reg\_pairs*)

Measures many qubits into their specified classical bits, in the order they were entered. If no qubit/register pairs are provided, measure all qubits present in the program into classical addresses of the same index.

**Parameters** **qubit\_reg\_pairs** (*Tuple*) – Tuples of qubit indices paired with classical bits.

**Returns** The Quil Program with the appropriate measure instructions appended, e.g.

```
MEASURE 0 [1]
MEASURE 1 [2]
MEASURE 2 [3]
```

**Return type** *Program*

**no\_noise** ()

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

**Returns** Program

**out** ()

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

**pop** ()

Pops off the last instruction.

**Returns** The instruction that was popped.

**Return type** *tuple*

**reset** (*qubit\_index=None*)

Reset all qubits or just a specific qubit at *qubit\_index*.

**Parameters** **qubit\_index** (*Optional[int]*) – The address of the qubit to reset. If None, reset all qubits.

**Returns** The Quil Program with the appropriate reset instruction appended, e.g. RESET 0

**Return type** *Program*

**while\_do** (*classical\_reg*, *q\_program*)

While a classical register at index *classical\_reg* is 1, loop *q\_program*

Equivalent to the following construction:

```

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

```

### Parameters

- **classical\_reg** (*int*) – The classical register to check
- **q\_program** (*Program*) – The Quil program to loop.

**Returns** The Quil Program with the loop instructions added.

**Return type** *Program*

### **wrap\_in\_numshots\_loop** (*shots: int*)

Wraps a Quil program in a loop that re-runs the same program many times.

Note: this function is a prototype of what will exist in the future when users will be responsible for writing this loop instead of having it happen automatically.

**Parameters** **shots** – Number of iterations to loop through.

### **pyquil.quil.address\_qubits** (*program, qubit\_mapping=None*)

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

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

This function will also instantiate any label placeholders.

### Parameters

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

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

### **pyquil.quil.get\_classical\_addresses\_from\_program** (*program*) → Dict[str, List[int]]

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

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

**Returns** A mapping from memory region names to lists of offsets appearing in the program.

### **pyquil.quil.get\_default\_qubit\_mapping** (*program*)

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

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

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

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

### **pyquil.quil.implicitly\_declare\_ro** (*instructions: List[pyquil.quilbase.AbstractInstruction]*)

Implicitly declare a register named `ro` for backwards compatibility with Quil 1.

There used to be one un-named hunk of classical memory. Now there are variables with declarations. Instead of:

```
MEASURE 0 [0]
```

You must now measure into a named register, idiomatically:

```
MEASURE 0 ro[0]
```

The `MEASURE` instruction will emit this (with a deprecation warning) if you're still using bare integers for classical addresses. However, you must also declare memory in the new scheme:

```
DECLARE ro BIT[8]
MEASURE 0 ro[0]
```

This method will determine if you are in “backwards compatibility mode” and will declare a read-out `ro` register for you. If your program contains any `DECLARE` commands or if it does not have any `MEASURE x ro[x]`, this will not do anything.

This behavior is included for backwards compatibility and will be removed in future releases of PyQuil. Please `DECLARE` all memory including `ro`.

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

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

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

`pyquil.quil.merge_programs` (*prog\_list*)

Merges a list of pyQuil programs into a single one by appending them in sequence. If multiple programs in the list contain the same gate and/or noisy gate definition with identical name, this definition will only be applied once. If different definitions with the same name appear multiple times in the program list, each will be applied once in the order of last occurrence.

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

**Returns** a single pyQuil program

**Return type** *Program*

`pyquil.quil.merge_with_pauli_noise` (*prog\_list: Iterable, probabilities: List, qubits: List*)

Insert pauli noise channels between each item in the list of programs. This noise channel is implemented as a single noisy identity gate acting on the provided qubits. This method does not rely on `merge_programs` and so avoids the inclusion of redundant Kraus Pragmas that would occur if `merge_programs` was called directly on programs with distinct noisy gate definitions.

**Parameters**

- **prog\_list** – an iterable such as a program or a list of programs. If a program is provided, a single noise gate will be applied after each gate in the program. If a list of programs is provided, the noise gate will be applied after each program.
- **probabilities** – The  $4^{\text{num\_qubits}}$  list of probabilities specifying the desired pauli channel. There should be either 4 or 16 probabilities specified in the order I, X, Y, Z or II, IX, IY, IZ, XI, XX, XY, etc respectively.
- **qubits** – a list of the qubits that the noisy gate should act on.

**Returns** A single program with noisy gates inserted between each element of the program list.

**Return type** *Program*

`pyquil.quil.percolate_declares` (*program: pyquil.quil.Program*) → `pyquil.quil.Program`  
 Move all the DECLARE statements to the top of the program. Return a fresh object.

**Parameters** `program` – Perhaps jumbled program.

**Returns** Program with DECLAREs all at the top and otherwise the same sorted contents.

`pyquil.quil.validate_protoquil` (*program: pyquil.quil.Program*) → `None`

Ensure that a program is valid ProtoQuil, otherwise raise a `ValueError`. Protoquil allows a global RESET before any gates, and MEASUREs on each qubit after any gates on that qubit. Pragmas are always allowed, and a final Halt instruction is allowed.

**Parameters** `program` – The Quil program to validate.

## 1.10.8 pyquil.quilbase

Contains the core pyQuil objects that correspond to Quil instructions.

**class** `pyquil.quilbase.AbstractInstruction`

Bases: `object`

Abstract class for representing single instructions.

`out()`

**class** `pyquil.quilbase.ArithmeticBinaryOp` (*left, right*)

Bases: `pyquil.quilbase.AbstractInstruction`

The abstract class for binary arithmetic classical instructions.

`out()`

**class** `pyquil.quilbase.ClassicalAdd` (*left, right*)

Bases: `pyquil.quilbase.ArithmeticBinaryOp`

The ADD instruction.

`op = 'ADD'`

**class** `pyquil.quilbase.ClassicalAnd` (*left, right*)

Bases: `pyquil.quilbase.LogicalBinaryOp`

WARNING: The operand order for ClassicalAnd has changed. In pyQuil versions <= 1.9, AND had signature

AND %source %target

Now, AND has signature

AND %target %source

`op = 'AND'`

**class** `pyquil.quilbase.ClassicalComparison` (*target, left, right*)

Bases: `pyquil.quilbase.AbstractInstruction`

Abstract class for ternary comparison instructions.

`out()`

**class** `pyquil.quilbase.ClassicalConvert` (*left, right*)

Bases: `pyquil.quilbase.AbstractInstruction`

The CONVERT instruction.

`op = 'CONVERT'`

`out ()`

**class** `pyquil.quilbase.ClassicalDiv (left, right)`  
Bases: `pyquil.quilbase.ArithmeticBinaryOp`

The DIV instruction.

`op = 'DIV'`

**class** `pyquil.quilbase.ClassicalEqual (target, left, right)`  
Bases: `pyquil.quilbase.ClassicalComparison`

The EQ comparison instruction.

`op = 'EQ'`

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

The EXCHANGE instruction.

`op = 'EXCHANGE'`

`out ()`

**class** `pyquil.quilbase.ClassicalExclusiveOr (left, right)`  
Bases: `pyquil.quilbase.LogicalBinaryOp`

The XOR instruction.

`op = 'XOR'`

**class** `pyquil.quilbase.ClassicalFalse (target)`  
Bases: `pyquil.quilbase.ClassicalMove`

Deprecated class.

**class** `pyquil.quilbase.ClassicalGreaterEqual (target, left, right)`  
Bases: `pyquil.quilbase.ClassicalComparison`

The GE comparison instruction.

`op = 'GE'`

**class** `pyquil.quilbase.ClassicalGreaterThan (target, left, right)`  
Bases: `pyquil.quilbase.ClassicalComparison`

The GT comparison instruction.

`op = 'GT'`

**class** `pyquil.quilbase.ClassicalInclusiveOr (left, right)`  
Bases: `pyquil.quilbase.LogicalBinaryOp`

The IOR instruction.

`op = 'IOR'`

**class** `pyquil.quilbase.ClassicalLessEqual (target, left, right)`  
Bases: `pyquil.quilbase.ClassicalComparison`

The LE comparison instruction.

`op = 'LE'`

**class** `pyquil.quilbase.ClassicalLessThan (target, left, right)`  
Bases: `pyquil.quilbase.ClassicalComparison`

The LT comparison instruction.

```
op = 'LT'
```

```
class pyquil.quilbase.ClassicalLoad(target, left, right)
```

Bases: *pyquil.quilbase.AbstractInstruction*

The LOAD instruction.

```
op = 'LOAD'
```

```
out()
```

```
class pyquil.quilbase.ClassicalMove(left, right)
```

Bases: *pyquil.quilbase.AbstractInstruction*

The MOVE instruction.

**WARNING: In pyQuil 2.0, the order of operands is as MOVE <target> <source>.** In pyQuil 1.9, the order of operands was MOVE <source> <target>. These have reversed.

```
op = 'MOVE'
```

```
out()
```

```
class pyquil.quilbase.ClassicalMul(left, right)
```

Bases: *pyquil.quilbase.ArithmeticBinaryOp*

The MUL instruction.

```
op = 'MUL'
```

```
class pyquil.quilbase.ClassicalNeg(target)
```

Bases: *pyquil.quilbase.UnaryClassicalInstruction*

The NEG instruction.

```
op = 'NEG'
```

```
class pyquil.quilbase.ClassicalNot(target)
```

Bases: *pyquil.quilbase.UnaryClassicalInstruction*

The NOT instruction.

```
op = 'NOT'
```

```
class pyquil.quilbase.ClassicalOr(left, right)
```

Bases: *pyquil.quilbase.ClassicalInclusiveOr*

Deprecated class.

```
class pyquil.quilbase.ClassicalStore(target, left, right)
```

Bases: *pyquil.quilbase.AbstractInstruction*

The STORE instruction.

```
op = 'STORE'
```

```
out()
```

```
class pyquil.quilbase.ClassicalSub(left, right)
```

Bases: *pyquil.quilbase.ArithmeticBinaryOp*

The SUB instruction.

```
op = 'SUB'
```

**class** pyquil.quilbase.**ClassicalTrue** (*target*)

Bases: [pyquil.quilbase.ClassicalMove](#)

Deprecated class.

**class** pyquil.quilbase.**Declare** (*name, memory\_type, memory\_size=1, shared\_region=None, offsetsets=None*)

Bases: [pyquil.quilbase.AbstractInstruction](#)

A DECLARE directive.

This is printed in Quil as:

```
DECLARE <name> <memory-type> (SHARING <other-name> (OFFSET <amount> <type>)* )?
```

**asdict** ()

**out** ()

**class** pyquil.quilbase.**DefGate** (*name, matrix, parameters=None*)

Bases: [pyquil.quilbase.AbstractInstruction](#)

A DEFGATE directive.

#### Parameters

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

**get\_constructor** ()

**Returns** A function that constructs this gate on variable qubit indices. E.g. *my-gate.get\_constructor()(1)* applies the gate to qubit 1.

**num\_args** ()

**Returns** The number of qubit arguments the gate takes.

**Return type** `int`

**out** ()

Prints a readable Quil string representation of this gate.

**Returns** String representation of a gate

**Return type** `string`

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

Bases: [pyquil.quilbase.AbstractInstruction](#)

This is the pyQuil object for a quantum gate instruction.

**get\_qubits** (*indices=True*)

**out** ()

**class** pyquil.quilbase.**Halt**

Bases: [pyquil.quilbase.SimpleInstruction](#)

The HALT instruction.

**op** = `'HALT'`

```

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

    Representation of an unconditional jump instruction (JUMP).

    out ()

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

    Abstract representation of an conditional jump instruction.

    out ()

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

    Representation of a target that can be jumped to.

    out ()

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

    The JUMP-UNLESS instruction.

    op = 'JUMP-UNLESS'

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

    The JUMP-WHEN instruction.

    op = 'JUMP-WHEN'

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

    The abstract class for binary logical classical instructions.

    out ()

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

    This is the pyQuil object for a Quil measurement instruction.

    get_qubits(indices=True)

    out ()

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

    The NOP instruction.

    op = 'NOP'

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

    A PRAGMA instruction.

    This is printed in QUIL as:

```

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

```

    out ()

```

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

    A raw instruction represented as a string.

    out ()

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

    The RESET instruction.

    op = 'RESET'

class pyquil.quilbase.ResetQubit (qubit)
    Bases: pyquil.quilbase.AbstractInstruction

    This is the pyQuil object for a Quil targeted reset instruction.

    get_qubits (indices=True)

    out ()

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'
```

### 1.10.9 pyquil.wavefunction

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

```
class pyquil.wavefunction.Wavefunction (amplitude_vector)
    Bases: object

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

---

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

---

```
static from_bit_packed_string (coef_string)
    From a bit packed string, unpacks to get the wavefunction :param bytes coef_string: :return:

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

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

**Return type** `dict`

**static** `ground(qubit_num)`

**plot** (*qubit\_subset=None*)

Plots a bar chart with bitstring on the x axis and probability on the y axis.

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

**pretty\_print** (*decimal\_digits=2*)

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

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

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

**Return type** `str`

**pretty\_print\_probabilities** (*decimal\_digits=2*)

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

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

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

**Return type** `dict`

**probabilities** ()

Returns an array of probabilities in lexicographical order

**sample\_bitstrings** (*n\_samples*)

Sample bitstrings from the distribution defined by the wavefunction.

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

**Returns** An array of shape (n\_samples, n\_qubits)

**static** `zeros(qubit_num)`

Constructs the groundstate wavefunction for a given number of qubits.

**Parameters** `qubit_num` (*int*) –

**Returns** A Wavefunction in the ground state

**Return type** `Wavefunction`

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

Returns the bitstring in lexical order that corresponds to the given index in 0 to  $2^{(qubit\_num)}$  :param int index:  
:param int qubit\_num: :return: the bitstring :rtype: str

## 1.11 Changelog

### 1.11.1 v2.0.0 (November 1, 2018)

PyQuil 2.0 is a major release of pyQuil, Rigetti's toolkit for constructing and running quantum programs. This release contains many major changes including:

1. The introduction of [Quantum Cloud Services](#). Access Rigetti’s QPUs from co-located classical compute resources for minimal latency. The web API for running QVM and QPU jobs has been deprecated and cannot be accessed with pyQuil 2.0
2. Advances in classical control systems and compilation allowing the pre-compilation of parametric binary executables for rapid hybrid algorithm iteration.
3. Changes to Quil—our quantum instruction language—to provide easier ways of interacting with classical memory.

The new QCS access model and features will allow you to execute hybrid quantum algorithms several orders of magnitude (!) faster than the previous web endpoint. However, to fully exploit these speed increases you must update your programs to use the latest pyQuil features and APIs. Please read [Forest 2.0: Migration Guide](#) for a comprehensive migration guide.

An incomplete list of significant changes:

- Python 2 is no longer supported. Please use Python 3.6+
- Parametric gates are now normal functions. You can no longer write `RX(pi/2)(0)` to get a Quil `RX(pi/2) 0` instruction. Just use `RX(pi/2, 0)`.
- Gates support keyword arguments, so you can write `RX(angle=pi/2, qubit=0)`.
- All `async` methods have been removed from `QVMConnection` and `QVMConnection` is deprecated. `QPUConnection` has been removed in accordance with the QCS access model. Use `pyquil.get_qc()` as the primary means of interacting with the QVM or QPU.
- `WavefunctionSimulator` allows unfettered access to wavefunction properties and routines. These methods and properties previously lived on `QVMConnection` and have been deprecated there.
- Classical memory in Quil must be declared with a name and type. Please read [Forest 2.0: Migration Guide](#) for more.
- Compilation has changed. There are now different `Compiler` objects that target either the QPU or QVM. You **must** explicitly compile your programs to run on a QPU or a realistic QVM.

### 1.11.2 v1.9 (June 6, 2018)

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

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

#### Qubit placeholders

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

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

- `PauliTerm` now remembers the order of its operations. `sX(1)*sZ(2)` will compile to different Quil code than `sZ(2)*sX(1)`, although the terms will still be equal according to the `__eq__` method. During `PauliSum` combination of like terms, a warning will be emitted if two terms are combined that have different orders of operation.
- `PauliTerm.id()` takes an optional argument `sort_ops` which defaults to `True` for backwards compatibility. However, this function should not be used for comparing term-type like it has been used previously. Use `PauliTerm.operations_as_set()` instead. In the future, `sort_ops` will default to `False` and will eventually be removed.
- `Program.alloc()` has been deprecated. Please instantiate `QubitPlaceholder()` directly or request a “register” (list) of `n` placeholders by using the class constructor `QubitPlaceholder.register(n)()`.
- Programs must contain either (1) all instantiated qubits with integer indexes or (2) all placeholder qubits of type `QubitPlaceholder`. We have found that most users use (1) but (2) will become useful with larger and more diverse devices.
- Programs that contain qubit placeholders must be **explicitly addressed** prior to execution. Previously, qubits would be assigned “under the hood” to integers  $0 \dots N$ . Now, you must use `address_qubits()` which returns a new program with all qubits indexed depending on the `qubit_mapping` argument. The original program is unaffected and can be “readdressed” multiple times.
- `PauliTerm` can now accept `QubitPlaceholder` in addition to integers.
- `QubitPlaceholder` is no longer a subclass of `Qubit`. `LabelPlaceholder` is no longer a subclass of `Label`.
- `QuilAtom` subclasses’ hash functions have changed.

## Randomized benchmarking sequence generation

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

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

## Ease of Use

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

- `Program` can be initiated with a generator expression.
- `Program.measure_all()` (with no arguments) will measure all qubits in a program.
- `classical_addresses` is now optional in QVM and QPU `run()` methods. By default, any classical addresses targeted by `MEASURE` will be returned.
- `QVMConnection.pauli_expectation()` accepts `PauliSum` as arguments. This offers a more sensible API compared to `QVMConnection.expectation()`.

- pyQuil will now retry jobs every 10 seconds if the QPU is re-tuning.
- `CompilerConnection.compile()` now takes an optional argument `isa` that allows per-compilation specification of the target ISA.
- An empty program will trigger an exception if you try to run it.

## Supported versions of Python

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

## Bug fixes

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

## 1.12 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.

### 1.12.1 From Bit to Qubit

#### Probabilistic Bits as Vector Spaces

From an operational perspective, a bit is described by the results of measurements performed on it. Let the possible results of measuring a bit (0 or 1) be represented by orthonormal basis vectors  $\vec{0}$  and  $\vec{1}$ . We will

call these vectors **outcomes**. These outcomes span a two-dimensional vector space that represents a probabilistic bit. A probabilistic bit can be represented as a vector

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

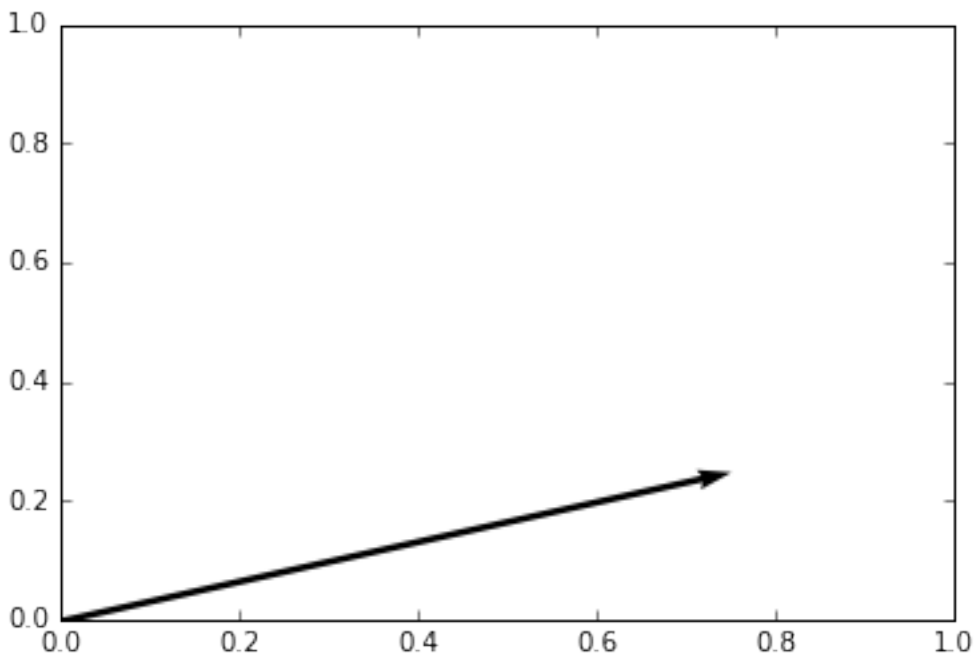
where  $a$  represents the probability of the bit being 0 and  $b$  represents the probability of the bit being 1. This clearly also requires that  $a+b=1$ . In this picture the **system** (the probabilistic bit) is a two-dimensional real vector space and a **state** of a system is a particular vector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

```
import numpy as np
import matplotlib.pyplot as plt

outcome_0 = np.array([1.0, 0.0])
outcome_1 = np.array([0.0, 1.0])
a = 0.75
b = 0.25

prob_bit = a*outcome_0 + b*outcome_1

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



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

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

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

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

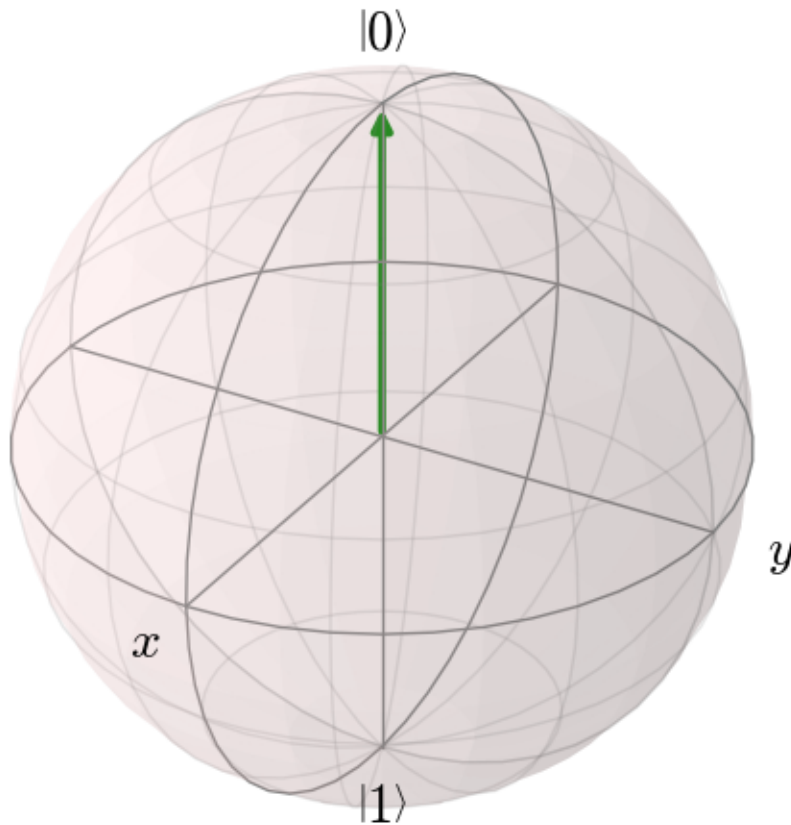
$$\alpha|0\rangle + \beta|1\rangle.$$

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
from pyquil.api import QVMConnection
quantum_simulator = QVMConnection()

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

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

# Quantum states are called wavefunctions for historical reasons.
# We can run this basic program on our connection to the simulator.
# This call will return the state of our qubits after we run program p.
# This api call returns a tuple, but we'll ignore the second value for now.
wavefunction = quantum_simulator.wavefunction(p)

# wavefunction is a Wavefunction object that stores a quantum state as a list of
↳ amplitudes
alpha, beta = wavefunction

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

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

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

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

p = Program(X(0))

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

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

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

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

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

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

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

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

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

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

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

### 1.12.2 Qubit Operations

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

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

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

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

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

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

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

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

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

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

## Pauli Operators

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

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

```
from pyquil.gates import X, Y, Z

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

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

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

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

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

Z|0> = (1+0j)|0>
The outcome probabilities are {'0': 1.0, '1': 0.0}
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  $\hat{x}$  axis. This explains the results of our code above: for a state vector initially in the  $+\hat{z}$  direction, both X and Y gates will rotate it to  $-\hat{z}$ , and the Z gate will leave it unchanged.

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

Quantum programs are built by applying successive gate operations:

```
# Composing qubit operations is the same as multiplying matrices sequentially
p = Program(X(0), Y(0), Z(0))
wavefunction = quantum_simulator.wavefunction(p)

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

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

## Multi-Qubit Operations

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

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

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

```
from pyquil.gates import CNOT

p = Program(CNOT(0, 1))
wavefunction = quantum_simulator.wavefunction(p)
print("CNOT|00> = ", wavefunction)
print("With outcome probabilities\n", wavefunction.get_outcome_probs())

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

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

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

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

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

(continues on next page)

(continued from previous page)

```

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

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

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

```

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

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

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

```

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

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

```

```

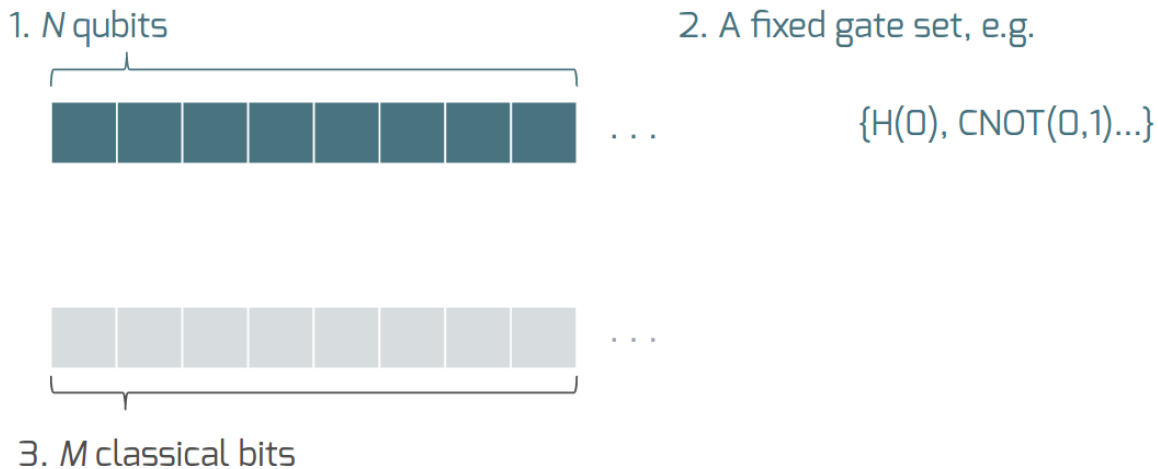
SWAP|01> = (1+0j)|10>
With outcome probabilities
{'00': 0.0, '01': 0.0, '10': 1.0, '11': 0.0}

```

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

### 1.12.3 The Quantum Abstract Machine

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



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

Full details on the Quantum Abstract Machine and Quil can be found in the [Quil whitepaper](#).

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

## Qubit Measurements

Measurements have two effects:

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

Here's a simple example:

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

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

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

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

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

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

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

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

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

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

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

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

## Classical/Quantum Interaction

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

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

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

```
from pyquil.gates import H

# The Hadamard produces what is called a superposition state
coin_program = Program(H(0))
wavefunction = quantum_simulator.wavefunction(coin_program)

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

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

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

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

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
print(quantum_simulator.run(coin_program, [0], trials))
```

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

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

```
classical_reg = 0
coin_program = Program(H(0))
print("Before measurement: H|0> = ", quantum_simulator.wavefunction(coin_program))

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

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

After measurement: (1+0j)|1>
After measurement: (1+0j)|0>
After measurement: (1+0j)|0>
After measurement: (1+0j)|1>
After measurement: (1+0j)|1>
```

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

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

# This program prepares something called a Bell state (a special kind of "entangled_
↪state")
bell_program = Program(H(0), CNOT(0, 1))
wavefunction = quantum_simulator.wavefunction(bell_program)
print("Before measurement: Bell state = ", wavefunction)

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

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

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

The above program prepares **entanglement** because, even though there are random outcomes, after every measurement

both qubits are in the same state. They are either both  $|0\rangle$  or both  $|1\rangle$ . This special kind of correlation is part of what makes quantum mechanics so unique and powerful.

## Classical Control

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

```

true_branch = Program(X(7)) # if branch
false_branch = Program(I(7)) # else branch

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

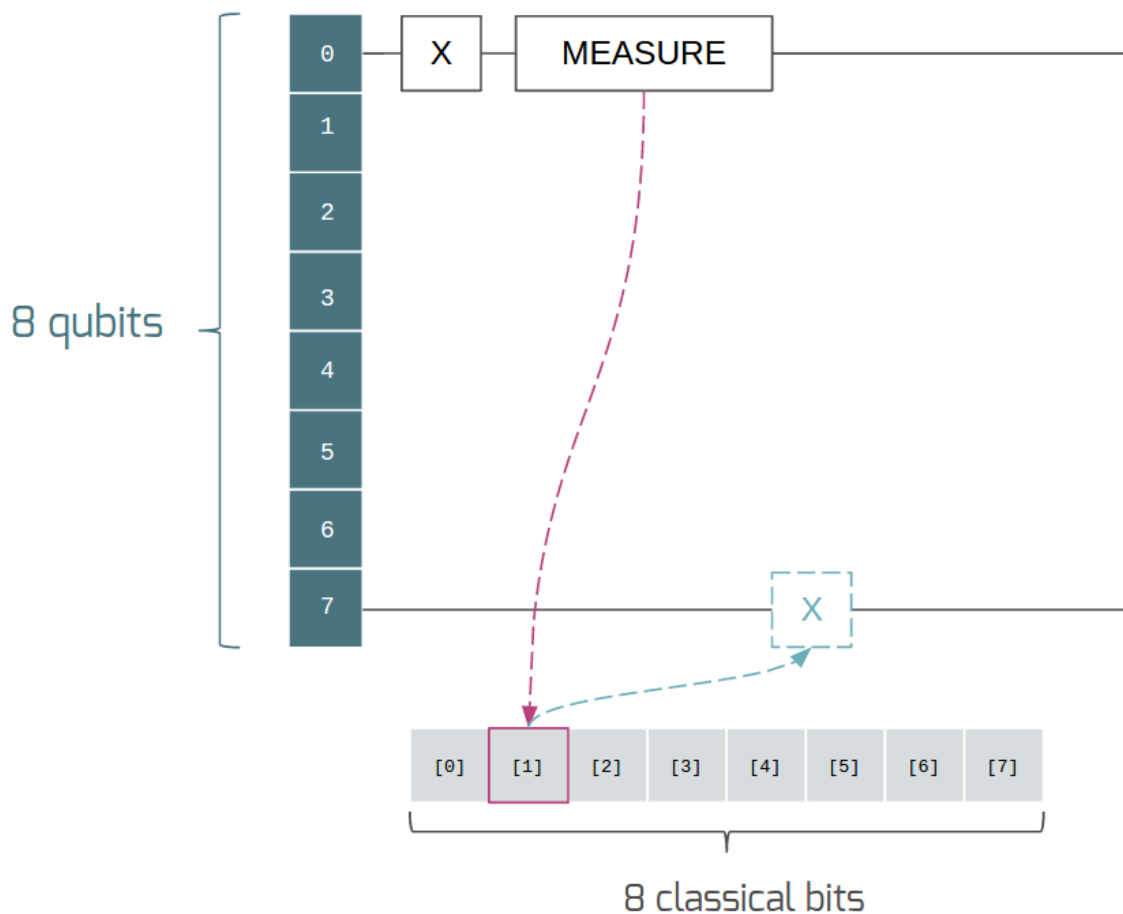
# Measure qubit #7 into classical register [7]
p.measure(7, 7)

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

```

```
[[1]]
```

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



### Example: The Probabilistic Halting Problem

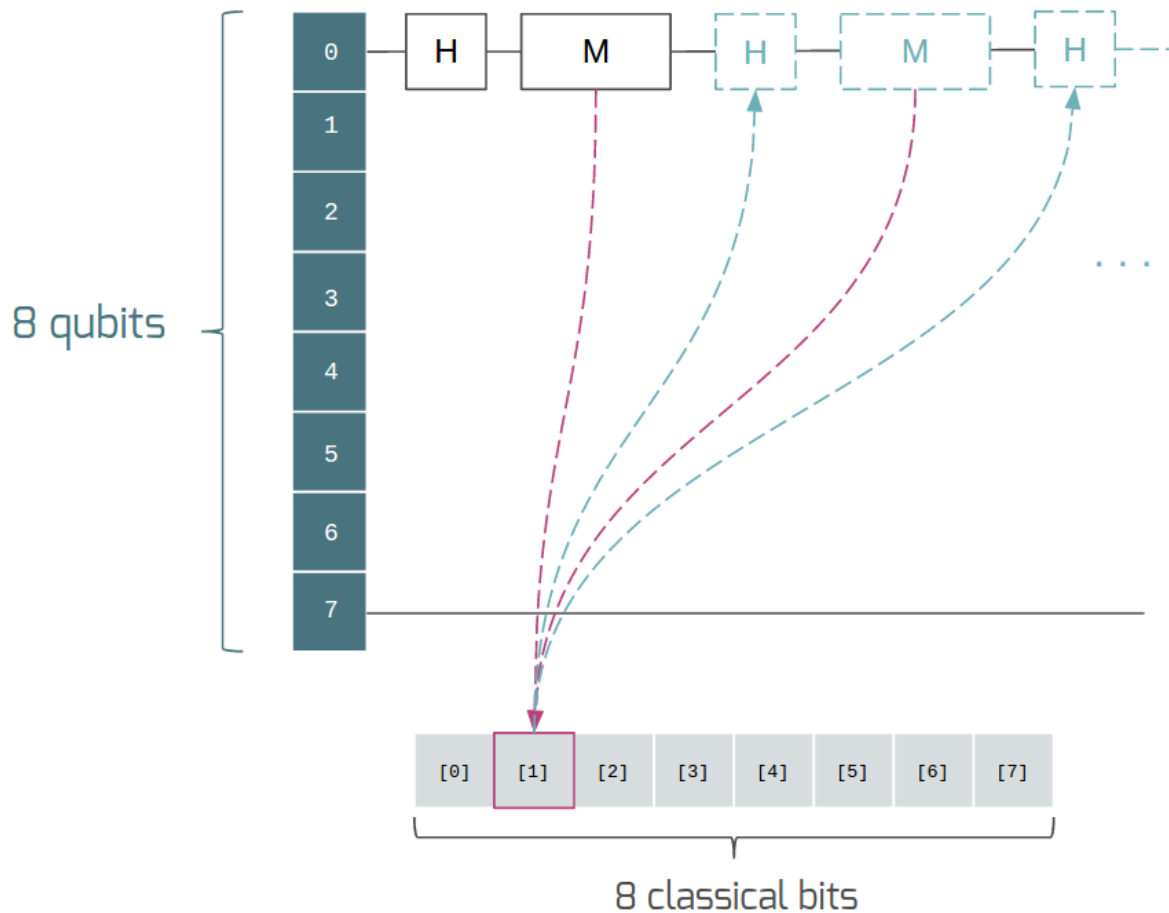
A fun example is to create a program that has an exponentially increasing chance of halting, but that may run forever!

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

p = Program().inst(X(0)).while_do(1, inside_loop)

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

```
[[0]]
```



#### 1.12.4 Next Steps

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

If you would like to learn more, Nielsen and Chuang's *Quantum Computation and Quantum Information* is a particularly excellent resource for newcomers to the field.

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



## CHAPTER 2

---

### Indices and Tables

---

- `genindex`
- `modindex`
- `search`



---

## Bibliography

---

[DensityMatrix] [https://en.wikipedia.org/wiki/Density\\_matrix](https://en.wikipedia.org/wiki/Density_matrix)



### p

- `pyquil.api`, 62
- `pyquil.device`, 73
- `pyquil.gates`, 79
- `pyquil.noise`, 88
- `pyquil.parser`, 94
- `pyquil.paulis`, 94
- `pyquil.quil`, 98
- `pyquil.quilbase`, 105
- `pyquil.wavefunction`, 110



## A

AbstractDevice (class in pyquil.device), 73  
 AbstractInstruction (class in pyquil.qubitbase), 105  
 ADD() (in module pyquil.gates), 85  
 add\_decoherence\_noise() (in module pyquil.noise), 90  
 address\_qubits() (in module pyquil.qubit), 103  
 alloc() (pyquil.qubit.Program method), 98  
 AND() (in module pyquil.gates), 84  
 append\_kraus\_to\_gate() (in module pyquil.noise), 90  
 apply\_clifford\_to\_pauli()  
     (pyquil.api.BenchmarkConnection method), 72  
 apply\_clifford\_to\_pauli()  
     (pyquil.api.LocalBenchmarkConnection  
     method), 73  
 apply\_noise\_model() (in module pyquil.noise), 90  
 ArithmeticBinaryOp (class in pyquil.qubitbase), 105  
 asdict() (pyquil.qubitbase.Declare method), 108  
 augment\_program\_with\_memory\_values()  
     (pyquil.api.QVM method), 71

## B

BenchmarkConnection (class in pyquil.api), 72  
 bitstring\_probs\_to\_z\_moments() (in module  
     pyquil.noise), 91

## C

CCNOT() (in module pyquil.gates), 81  
 check\_commutation() (in module pyquil.paulis), 95  
 ClassicalAdd (class in pyquil.qubitbase), 105  
 ClassicalAnd (class in pyquil.qubitbase), 105  
 ClassicalComparison (class in pyquil.qubitbase), 105  
 ClassicalConvert (class in pyquil.qubitbase), 105  
 ClassicalDiv (class in pyquil.qubitbase), 106  
 ClassicalEqual (class in pyquil.qubitbase), 106  
 ClassicalExchange (class in pyquil.qubitbase), 106  
 ClassicalExclusiveOr (class in pyquil.qubitbase), 106  
 ClassicalFalse (class in pyquil.qubitbase), 106  
 ClassicalGreaterEqual (class in pyquil.qubitbase), 106  
 ClassicalGreaterThan (class in pyquil.qubitbase), 106

ClassicalInclusiveOr (class in pyquil.qubitbase), 106  
 ClassicalLessEqual (class in pyquil.qubitbase), 106  
 ClassicalLessThan (class in pyquil.qubitbase), 106  
 ClassicalLoad (class in pyquil.qubitbase), 107  
 ClassicalMove (class in pyquil.qubitbase), 107  
 ClassicalMul (class in pyquil.qubitbase), 107  
 ClassicalNeg (class in pyquil.qubitbase), 107  
 ClassicalNot (class in pyquil.qubitbase), 107  
 ClassicalOr (class in pyquil.qubitbase), 107  
 ClassicalStore (class in pyquil.qubitbase), 107  
 ClassicalSub (class in pyquil.qubitbase), 107  
 ClassicalTrue (class in pyquil.qubitbase), 107  
 CNOT() (in module pyquil.gates), 81  
 combine\_kraus\_maps() (in module pyquil.noise), 91  
 commuting\_sets() (in module pyquil.paulis), 96  
 compile() (pyquil.api.QuantumComputer method), 68  
 compiled\_quil() (pyquil.api.Job method), 65  
 CONVERT() (in module pyquil.gates), 87  
 copy() (pyquil.paulis.PauliTerm method), 95  
 copy() (pyquil.qubit.Program method), 98  
 correct\_bitstring\_probs() (in module pyquil.noise), 91  
 corrupt\_bitstring\_probs() (in module pyquil.noise), 92  
 CPHASE() (in module pyquil.gates), 82  
 CPHASE00() (in module pyquil.gates), 81  
 CPHASE01() (in module pyquil.gates), 81  
 CPHASE10() (in module pyquil.gates), 82  
 CSWAP() (in module pyquil.gates), 83  
 CZ() (in module pyquil.gates), 80

## D

dagger() (pyquil.qubit.Program method), 98  
 damping\_after\_dephasing() (in module pyquil.noise), 92  
 damping\_kraus\_map() (in module pyquil.noise), 92  
 dead (pyquil.device.Edge attribute), 74  
 dead (pyquil.device.Qubit attribute), 76  
 Declare (class in pyquil.qubitbase), 108  
 declare() (pyquil.qubit.Program method), 98  
 decode() (pyquil.api.Job method), 65  
 decoherence\_noise\_with\_asymmetric\_ro() (in module  
     pyquil.noise), 92

DefGate (class in pyquil.quilbase), 108  
defgate() (pyquil.quil.Program method), 99  
define\_noisy\_gate() (pyquil.quil.Program method), 99  
define\_noisy\_readout() (pyquil.quil.Program method), 99  
defined\_gates (pyquil.quil.Program attribute), 100  
dephasing\_kraus\_map() (in module pyquil.noise), 92  
Device (class in pyquil.api), 66  
Device (class in pyquil.device), 74  
DIV() (in module pyquil.gates), 86

## E

Edge (class in pyquil.device), 74  
edges() (pyquil.device.NxDevice method), 75  
EdgeSpecs (in module pyquil.device), 74  
EQ() (in module pyquil.gates), 86  
estimate\_assignment\_probs() (in module pyquil.noise), 92  
estimate\_bitstring\_probs() (in module pyquil.noise), 93  
estimated\_time\_left\_in\_queue() (pyquil.api.Job method), 65  
EXCHANGE() (in module pyquil.gates), 85  
expectation() (pyquil.api.QVMConnection method), 62  
expectation() (pyquil.api.WavefunctionSimulator method), 67  
exponential\_map() (in module pyquil.paulis), 96  
exponentiate() (in module pyquil.paulis), 96  
exponentiate\_commuting\_pauli\_sum() (in module pyquil.paulis), 96

## F

f1QRBs() (pyquil.device.Specs method), 76  
fActiveResets() (pyquil.device.Specs method), 77  
FALSE() (in module pyquil.gates), 84  
fBellStates() (pyquil.device.Specs method), 77  
fCPHASEs() (pyquil.device.Specs method), 77  
fCZs() (pyquil.device.Specs method), 77  
ForestConnection (class in pyquil.api), 67  
from\_bit\_packed\_string() (pyquil.wavefunction.Wavefunction static method), 110  
from\_dict() (pyquil.device.ISA static method), 75  
from\_dict() (pyquil.device.Specs static method), 77  
from\_dict() (pyquil.noise.KrausModel static method), 88  
from\_dict() (pyquil.noise.NoiseModel static method), 89  
from\_list() (pyquil.paulis.PauliTerm class method), 95  
fROs() (pyquil.device.Specs method), 77

## G

Gate (class in pyquil.gates), 87  
Gate (class in pyquil.quilbase), 108  
gate() (pyquil.quil.Program method), 100  
gate\_depth() (pyquil.api.Job method), 65  
gate\_volume() (pyquil.api.Job method), 65  
gates\_by\_name() (pyquil.noise.NoiseModel method), 89

gates\_in\_isa() (in module pyquil.device), 78  
GE() (in module pyquil.gates), 86  
generate\_rb\_sequence() (pyquil.api.BenchmarkConnection method), 72  
generate\_rb\_sequence() (pyquil.api.LocalBenchmarkConnection method), 73  
get() (pyquil.api.Job method), 65  
get\_benchmark() (in module pyquil.api), 73  
get\_bitstring\_from\_index() (in module pyquil.wavefunction), 111  
get\_classical\_addresses\_from\_program() (in module pyquil.quil), 103  
get\_constructor() (pyquil.quilbase.DefGate method), 108  
get\_default\_qubit\_mapping() (in module pyquil.quil), 103  
get\_isa() (pyquil.api.Device method), 66  
get\_isa() (pyquil.api.QuantumComputer method), 68  
get\_isa() (pyquil.device.AbstractDevice method), 73  
get\_isa() (pyquil.device.Device method), 74  
get\_isa() (pyquil.device.NxDevice method), 76  
get\_noisy\_gate() (in module pyquil.noise), 93  
get\_outcome\_probs() (pyquil.wavefunction.Wavefunction method), 110  
get\_programs() (pyquil.paulis.PauliSum method), 94  
get\_qc() (in module pyquil.api), 69  
get\_qubits() (pyquil.gates.Gate method), 87  
get\_qubits() (pyquil.paulis.PauliSum method), 94  
get\_qubits() (pyquil.paulis.PauliTerm method), 95  
get\_qubits() (pyquil.quil.Program method), 100  
get\_qubits() (pyquil.quilbase.Gate method), 108  
get\_qubits() (pyquil.quilbase.Measurement method), 109  
get\_qubits() (pyquil.quilbase.ResetQubit method), 110  
get\_specs() (pyquil.api.Device method), 66  
get\_specs() (pyquil.device.AbstractDevice method), 73  
get\_specs() (pyquil.device.Device method), 74  
get\_specs() (pyquil.device.NxDevice method), 76  
get\_version\_info() (pyquil.api.LocalQVMCompiler method), 64  
get\_version\_info() (pyquil.api.QPU method), 72  
get\_version\_info() (pyquil.api.QPUCompiler method), 64  
get\_version\_info() (pyquil.api.QVM method), 71  
get\_version\_info() (pyquil.api.QVMCompiler method), 64  
get\_version\_info() (pyquil.api.QVMConnection method), 62  
ground() (pyquil.wavefunction.Wavefunction static method), 111  
GT() (in module pyquil.gates), 86

## H

H() (in module pyquil.gates), 79  
Halt (class in pyquil.quilbase), 108  
HALT (in module pyquil.gates), 84  
HASH\_PRECISION (in module pyquil.paulis), 94

## I

I() (in module pyquil.gates), 79  
 id (pyquil.device.Qubit attribute), 76  
 ID() (in module pyquil.paulis), 94  
 id() (pyquil.paulis.PauliTerm method), 95  
 if\_then() (pyquil.quil.Program method), 100  
 implicitly\_declare\_ro() (in module pyquil.quil), 103  
 INFINITY (in module pyquil.noise), 88  
 inst() (pyquil.quil.Program method), 101  
 instantiate\_labels() (in module pyquil.quil), 104  
 instructions (pyquil.quil.Program attribute), 101  
 integer\_types (in module pyquil.paulis), 96  
 IOR() (in module pyquil.gates), 85  
 is\_compiling() (pyquil.api.Job method), 65  
 is\_done() (pyquil.api.Job method), 65  
 is\_identity() (in module pyquil.paulis), 96  
 is\_protoquil() (pyquil.quil.Program method), 101  
 is\_queued() (pyquil.api.Job method), 65  
 is\_queued\_for\_compilation() (pyquil.api.Job method), 65  
 is\_running() (pyquil.api.Job method), 65  
 is\_zero() (in module pyquil.paulis), 96  
 ISA (class in pyquil.device), 74  
 isa (pyquil.api.Device attribute), 67  
 isa (pyquil.device.Device attribute), 74  
 isa\_from\_graph() (in module pyquil.device), 78  
 isa\_to\_graph() (in module pyquil.device), 78  
 ISWAP() (in module pyquil.gates), 83

## J

Job (class in pyquil.api), 65  
 job\_id (pyquil.api.Job attribute), 65  
 Jump (class in pyquil.quilbase), 108  
 JumpConditional (class in pyquil.quilbase), 109  
 JumpTarget (class in pyquil.quilbase), 109  
 JumpUnless (class in pyquil.quilbase), 109  
 JumpWhen (class in pyquil.quilbase), 109

## K

KrausModel (class in pyquil.noise), 88

## L

LE() (in module pyquil.gates), 87  
 list\_quantum\_computers() (in module pyquil.api), 69  
 LOAD() (in module pyquil.gates), 87  
 load() (pyquil.api.QAM method), 71  
 load() (pyquil.api.QVM method), 71  
 LocalBenchmarkConnection (class in pyquil.api), 72  
 LocalQVMCompiler (class in pyquil.api), 64  
 LogicalBinaryOp (class in pyquil.quilbase), 109  
 LT() (in module pyquil.gates), 87

## M

MEASURE() (in module pyquil.gates), 84

measure() (pyquil.quil.Program method), 102  
 measure\_all() (pyquil.quil.Program method), 102  
 Measurement (class in pyquil.quilbase), 109  
 merge\_programs() (in module pyquil.quil), 104  
 merge\_with\_pauli\_noise() (in module pyquil.quil), 104  
 MOVE() (in module pyquil.gates), 85  
 MUL() (in module pyquil.gates), 86  
 multiqubit\_gate\_depth() (pyquil.api.Job method), 65

## N

native\_quil\_to\_executable()  
     (pyquil.api.LocalQVMCompiler method),  
     64  
 native\_quil\_to\_executable() (pyquil.api.QPUCompiler  
     method), 64  
 native\_quil\_to\_executable() (pyquil.api.QVMCompiler  
     method), 64  
 NEG() (in module pyquil.gates), 85  
 no\_noise() (pyquil.quil.Program method), 102  
 NoiseModel (class in pyquil.noise), 89  
 NoisyGateUndefined, 89  
 Nop (class in pyquil.quilbase), 109  
 NOP (in module pyquil.gates), 84  
 NOT() (in module pyquil.gates), 84  
 num\_args() (pyquil.quilbase.DefGate method), 108  
 NxDevice (class in pyquil.device), 75

## O

op (pyquil.quilbase.ClassicalAdd attribute), 105  
 op (pyquil.quilbase.ClassicalAnd attribute), 105  
 op (pyquil.quilbase.ClassicalConvert attribute), 105  
 op (pyquil.quilbase.ClassicalDiv attribute), 106  
 op (pyquil.quilbase.ClassicalEqual attribute), 106  
 op (pyquil.quilbase.ClassicalExchange attribute), 106  
 op (pyquil.quilbase.ClassicalExclusiveOr attribute), 106  
 op (pyquil.quilbase.ClassicalGreaterEqual attribute), 106  
 op (pyquil.quilbase.ClassicalGreaterThan attribute), 106  
 op (pyquil.quilbase.ClassicalInclusiveOr attribute), 106  
 op (pyquil.quilbase.ClassicalLessEqual attribute), 106  
 op (pyquil.quilbase.ClassicalLessThan attribute), 107  
 op (pyquil.quilbase.ClassicalLoad attribute), 107  
 op (pyquil.quilbase.ClassicalMove attribute), 107  
 op (pyquil.quilbase.ClassicalMul attribute), 107  
 op (pyquil.quilbase.ClassicalNeg attribute), 107  
 op (pyquil.quilbase.ClassicalNot attribute), 107  
 op (pyquil.quilbase.ClassicalStore attribute), 107  
 op (pyquil.quilbase.ClassicalSub attribute), 107  
 op (pyquil.quilbase.Halt attribute), 108  
 op (pyquil.quilbase.JumpUnless attribute), 109  
 op (pyquil.quilbase.JumpWhen attribute), 109  
 op (pyquil.quilbase.Nop attribute), 109  
 op (pyquil.quilbase.Reset attribute), 110  
 op (pyquil.quilbase.Wait attribute), 110  
 operations\_as\_set() (pyquil.paulis.PauliTerm method), 95

[OR\(\)](#) (in module `pyquil.gates`), 84  
[out\(\)](#) (`pyquil.gates.Gate` method), 88  
[out\(\)](#) (`pyquil.quil.Program` method), 102  
[out\(\)](#) (`pyquil.quilbase.AbstractInstruction` method), 105  
[out\(\)](#) (`pyquil.quilbase.ArithmeticBinaryOp` method), 105  
[out\(\)](#) (`pyquil.quilbase.ClassicalComparison` method), 105  
[out\(\)](#) (`pyquil.quilbase.ClassicalConvert` method), 105  
[out\(\)](#) (`pyquil.quilbase.ClassicalExchange` method), 106  
[out\(\)](#) (`pyquil.quilbase.ClassicalLoad` method), 107  
[out\(\)](#) (`pyquil.quilbase.ClassicalMove` method), 107  
[out\(\)](#) (`pyquil.quilbase.ClassicalStore` method), 107  
[out\(\)](#) (`pyquil.quilbase.Declare` method), 108  
[out\(\)](#) (`pyquil.quilbase.DefGate` method), 108  
[out\(\)](#) (`pyquil.quilbase.Gate` method), 108  
[out\(\)](#) (`pyquil.quilbase.Jump` method), 109  
[out\(\)](#) (`pyquil.quilbase.JumpConditional` method), 109  
[out\(\)](#) (`pyquil.quilbase.JumpTarget` method), 109  
[out\(\)](#) (`pyquil.quilbase.LogicalBinaryOp` method), 109  
[out\(\)](#) (`pyquil.quilbase.Measurement` method), 109  
[out\(\)](#) (`pyquil.quilbase.Pragma` method), 109  
[out\(\)](#) (`pyquil.quilbase.RawInstr` method), 110  
[out\(\)](#) (`pyquil.quilbase.ResetQubit` method), 110  
[out\(\)](#) (`pyquil.quilbase.SimpleInstruction` method), 110  
[out\(\)](#) (`pyquil.quilbase.UnaryClassicalInstruction` method), 110

## P

[parse\(\)](#) (in module `pyquil.parser`), 94  
[parse\\_program\(\)](#) (in module `pyquil.parser`), 94  
[pauli\\_expectation\(\)](#) (`pyquil.api.QVMConnection` method), 62  
[pauli\\_kraus\\_map\(\)](#) (in module `pyquil.noise`), 93  
[pauli\\_string\(\)](#) (`pyquil.paulis.PauliTerm` method), 95  
[PauliSum](#) (class in `pyquil.paulis`), 94  
[PauliTerm](#) (class in `pyquil.paulis`), 94  
[percolate\\_declares\(\)](#) (in module `pyquil.quil`), 104  
[PHASE\(\)](#) (in module `pyquil.gates`), 80  
[plot\(\)](#) (`pyquil.wavefunction.Wavefunction` method), 111  
[pop\(\)](#) (`pyquil.quil.Program` method), 102  
[position\\_in\\_queue\(\)](#) (`pyquil.api.Job` method), 66  
[Pragma](#) (class in `pyquil.quilbase`), 109  
[pretty\\_print\(\)](#) (`pyquil.wavefunction.Wavefunction` method), 111  
[pretty\\_print\\_probabilities\(\)](#) (`pyquil.wavefunction.Wavefunction` method), 111  
[probabilities\(\)](#) (`pyquil.wavefunction.Wavefunction` method), 111  
[Program](#) (class in `pyquil.quil`), 98  
[program](#) (`pyquil.paulis.PauliTerm` attribute), 95  
[program\\_fidelity\(\)](#) (`pyquil.api.Job` method), 66  
[PSWAP\(\)](#) (in module `pyquil.gates`), 83  
[pyquil.api](#) (module), 62  
[pyquil.device](#) (module), 73

[pyquil.gates](#) (module), 79  
[pyquil.noise](#) (module), 88  
[pyquil.parser](#) (module), 94  
[pyquil.paulis](#) (module), 94  
[pyquil.quil](#) (module), 98  
[pyquil.quilbase](#) (module), 105  
[pyquil.wavefunction](#) (module), 110  
[pyquil\\_protect\(\)](#) (in module `pyquil.api`), 67

## Q

[QAM](#) (class in `pyquil.api`), 70  
[QPU](#) (class in `pyquil.api`), 72  
[QPUCompiler](#) (class in `pyquil.api`), 64  
[QuantumComputer](#) (class in `pyquil.api`), 68  
[Qubit](#) (class in `pyquil.device`), 76  
[qubit\\_topology\(\)](#) (`pyquil.api.Device` method), 67  
[qubit\\_topology\(\)](#) (`pyquil.api.QuantumComputer` method), 68  
[qubit\\_topology\(\)](#) (`pyquil.device.AbstractDevice` method), 74  
[qubit\\_topology\(\)](#) (`pyquil.device.Device` method), 74  
[qubit\\_topology\(\)](#) (`pyquil.device.NxDevice` method), 76  
[qubits\(\)](#) (`pyquil.api.Device` method), 67  
[qubits\(\)](#) (`pyquil.api.QuantumComputer` method), 68  
[qubits\(\)](#) (`pyquil.device.AbstractDevice` method), 74  
[qubits\(\)](#) (`pyquil.device.Device` method), 74  
[qubits\(\)](#) (`pyquil.device.NxDevice` method), 76  
[QubitSpecs](#) (in module `pyquil.device`), 76  
[quil\\_to\\_native\\_quil\(\)](#) (`pyquil.api.LocalQVMCompiler` method), 64  
[quil\\_to\\_native\\_quil\(\)](#) (`pyquil.api.QPUCompiler` method), 65  
[quil\\_to\\_native\\_quil\(\)](#) (`pyquil.api.QVMCompiler` method), 64  
[QVM](#) (class in `pyquil.api`), 71  
[QVMCompiler](#) (class in `pyquil.api`), 64  
[QVMConnection](#) (class in `pyquil.api`), 62

## R

[RawInstr](#) (class in `pyquil.quilbase`), 109  
[read\\_from\\_memory\\_region\(\)](#) (`pyquil.api.QAM` method), 71  
[read\\_memory\(\)](#) (`pyquil.api.QAM` method), 71  
[Reset](#) (class in `pyquil.quilbase`), 110  
[RESET\(\)](#) (in module `pyquil.gates`), 83  
[reset\(\)](#) (`pyquil.quil.Program` method), 102  
[ResetQubit](#) (class in `pyquil.quilbase`), 110  
[result\(\)](#) (`pyquil.api.Job` method), 66  
[run\(\)](#) (`pyquil.api.QAM` method), 71  
[run\(\)](#) (`pyquil.api.QPU` method), 72  
[run\(\)](#) (`pyquil.api.QuantumComputer` method), 68  
[run\(\)](#) (`pyquil.api.QVM` method), 71  
[run\(\)](#) (`pyquil.api.QVMConnection` method), 63

run\_and\_measure() (pyquil.api.QuantumComputer method), 68  
 run\_and\_measure() (pyquil.api.QVMConnection method), 63  
 run\_and\_measure() (pyquil.api.WavefunctionSimulator method), 67  
 run\_symmetrized\_readout() (pyquil.api.QuantumComputer method), 69  
 running\_time() (pyquil.api.Job method), 66  
 RX() (in module pyquil.gates), 80  
 RY() (in module pyquil.gates), 80  
 RZ() (in module pyquil.gates), 80

## S

S() (in module pyquil.gates), 79  
 sample\_bitstrings() (pyquil.wavefunction.Wavefunction method), 111  
 sI() (in module pyquil.paulis), 96  
 SimpleInstruction (class in pyquil.quilbase), 110  
 simplify() (pyquil.paulis.PauliSum method), 94  
 simplify\_pauli\_sum() (in module pyquil.paulis), 97  
 Specs (class in pyquil.device), 76  
 specs\_from\_graph() (in module pyquil.device), 78  
 STORE() (in module pyquil.gates), 87  
 SUB() (in module pyquil.gates), 85  
 suzuki\_trotter() (in module pyquil.paulis), 97  
 SWAP() (in module pyquil.gates), 82  
 sX() (in module pyquil.paulis), 97  
 sY() (in module pyquil.paulis), 97  
 sZ() (in module pyquil.paulis), 97

## T

T() (in module pyquil.gates), 79  
 T1s() (pyquil.device.Specs method), 76  
 T2s() (pyquil.device.Specs method), 76  
 targets (pyquil.device.Edge attribute), 74  
 tensor\_kraus\_maps() (in module pyquil.noise), 94  
 term\_with\_coeff() (in module pyquil.paulis), 97  
 THETA (in module pyquil.device), 78  
 time\_in\_queue() (pyquil.api.Job method), 66  
 to\_dict() (pyquil.device.ISA method), 75  
 to\_dict() (pyquil.device.Specs method), 77  
 to\_dict() (pyquil.noise.KrausModel method), 88  
 to\_dict() (pyquil.noise.NoiseModel method), 89  
 topological\_swaps() (pyquil.api.Job method), 66  
 trotterize() (in module pyquil.paulis), 97  
 TRUE() (in module pyquil.gates), 84  
 type (pyquil.device.Edge attribute), 74  
 type (pyquil.device.Qubit attribute), 76

## U

UnaryClassicalInstruction (class in pyquil.quilbase), 110  
 UnequalLengthWarning, 95

unpack\_kraus\_matrix() (pyquil.noise.KrausModel static method), 88

## V

validate\_protoquil() (in module pyquil.quil), 105

## W

Wait (class in pyquil.quilbase), 110  
 WAIT (in module pyquil.gates), 83  
 wait() (pyquil.api.QAM method), 71  
 Wavefunction (class in pyquil.wavefunction), 110  
 wavefunction() (pyquil.api.QVMConnection method), 63  
 wavefunction() (pyquil.api.WavefunctionSimulator method), 68  
 WavefunctionSimulator (class in pyquil.api), 67  
 while\_do() (pyquil.quil.Program method), 102  
 wrap\_in\_numshots\_loop() (pyquil.quil.Program method), 103  
 write\_memory() (pyquil.api.QAM method), 71

## X

X() (in module pyquil.gates), 79  
 XOR() (in module pyquil.gates), 85

## Y

Y() (in module pyquil.gates), 79

## Z

Z() (in module pyquil.gates), 79  
 ZERO() (in module pyquil.paulis), 95  
 zeros() (pyquil.wavefunction.Wavefunction static method), 111