pyQuil Documentation

Release 2.6.0

Rigetti Computing

Apr 19, 2019

Contents

1 Contents **3**

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 . . . 6
1.2 New in Forest 2 - Running on the QVM . . . 8
1.2.1 Program construction is largely the same . . . 8
1.2.2 Unphysical methods deprecated in `QVMConnection` . . . 8
1.2.3 Async methods removed . . . 10
1.3 New in Forest 2 - QuantumComputer . . . 11
1.3.1 Running a program . . . 11
1.3.2 Debugging with `WavefunctionSimulator` . . . 12
1.3.3 `get_qc` . . . 12
1.3.4 Sampling with `run_and_measure` . . . 13
1.3.5 Change alert: `run_and_measure` will return a dictionary of 1d bitstrings. . . . 13
1.3.6 Change alert: All qubits are measured . . . 13
1.3.7 Change alert: `run_and_measure` works with noise models now. . . . 14
1.3.8 `list_quantum_computers` . . . 15
1.3.9 `QuantumComputer`s have a topology . . . 15
1.3.10 Inspecting the topology . . . 15
1.3.11 What If I don't want a topology? . . . 16
1.3.12 Heirarchy of realism . . . 16
1.3.13 "What is a `QuantumComputer`?" Advanced Edition . . . 16
1.4 New in Forest 2 - Parametric Programs . . . 18
1.4.1 Scan over the parameter (the old way) . . . 19
1.4.2 Do an optimization (the old way) . . . 20
1.4.3 Compilation . . . 20
1.4.4 Parametric compilation . . . 21
1.4.5 Measuring . . . 22
1.4.6 Number of shots . . . 22
1.4.7 Using `qc.run()` . . . 23
1.4.8 Scan over the parameter (the new way) . . . 24
1.4.9 Using `memory_map` . . . 24
1.4.10 Do an optimization (the new way) . . . 25
1.5 New in Forest 2 - Other . . . 26
1.5.1 Updates to the Quil language . . . 26

1.5.2 Backwards compatibility . . . 27
1.5.3 Updates to Forest . . . 28
1.5.4 Miscellanea . . . 33
1.5.5 Error reporting . . . 33
1.5.6 Parametric Programs . . . 35
1.6 Programs and Gates . . . 36
1.6.1 Introduction . . . 36
1.6.2 The Standard Gate Set . . . 37
1.6.3 Declaring Memory . . . 37
1.6.4 Measurement . . . 38
1.6.5 Parametric Compilation . . . 40
1.6.6 Gate Modifiers . . . 41
1.6.7 Defining New Gates . . . 41
1.6.8 Defining Parametric Gates . . . 42
1.6.9 Pragmas . . . 43
1.6.10 Ways to Construct Programs . . . 44
1.6.11 QPU-allowable Quil . . . 45
1.7 The Quantum Computer . . . 46
1.7.1 The Quantum Virtual Machine (QVM) . . . 46
1.7.2 The Quantum Processing Unit (QPU) . . . 47
1.7.3 The `QuantumComputer` . . . 47
1.8 The Wavefunction Simulator . . . 51
1.8.1 Multi-Qubit Basis Enumeration . . . 52
1.9 The Quil Compiler . . . 53
1.9.1 Expectations for Program Contents . . . 53
1.9.2 Interacting with the Compiler . . . 53
1.9.3 Legal compiler input . . . 55
1.9.4 Region-specific compiler features through PRAGMA . . . 55
1.9.5 Common Error Messages . . . 58
1.10 Noise and Quantum Computation . . . 58
1.10.1 Modeling Noisy Quantum Gates . . . 58
1.10.2 Noisy Gates on the Rigetti QVM . . . 60
1.10.3 Adding Decoherence Noise . . . 67
1.10.4 Modeling Readout Noise . . . 70
1.10.5 Working with Readout Noise . . . 72
1.11 Advanced Usage . . . 79
1.11.1 PyQuil Configuration Files . . . 79
1.11.2 Using Qubit Placeholders . . . 80
1.11.3 Classical Control Flow . . . 81
1.11.4 Pauli Operator Algebra . . . 83
1.12 Exercises . . . 85
1.12.1 Exercise 1: Quantum Dice . . . 85
1.12.2 Exercise 2: Controlled Gates . . . 85
1.12.3 Exercise 3: Grover's Algorithm . . . 85
1.12.4 Exercise 4: Prisoner's Dilemma . . . 85
1.12.5 Exercise 5: Quantum Fourier Transform . . . 86
1.12.6 Example: The Meyer-Penny Game . . . 88
1.13 Changelog . . . 90
1.13.1 v2.6 (March 29, 2019) . . . 90
1.13.2 v2.5 (March 6, 2019) . . . 90
1.13.3 v2.4 (February 14, 2019) . . . 90
1.13.4 v2.3 (January 28, 2019) . . . 91
1.13.5 v2.2 (January 4, 2019) . . . 92
1.13.6 v2.1 (November 30, 2018) . . . 93

1.13.7 v2.0 (November 1, 2018) . . . 93
1.13.8 v1.9 (June 6, 2018) . . . 94
1.14 Introduction to Quantum Computing . . . 96
1.14.1 From Bit to Qubit . . . 96
1.14.2 Qubit Operations . . . 101
1.14.3 The Quantum Abstract Machine . . . 104
1.14.4 Next Steps . . . 111
1.15 Program . . . 111
1.15.1 Program.instructions . . . 111
1.15.2 Program.defined_gates . . . 111
1.15.3 Program.out . . . 111
1.15.4 Program.get_qubits . . . 112
1.15.5 Program.is_protoquil . . . 112
1.15.6 Program.__iadd__ . . . 113
1.15.7 Program.__add__ . . . 113
1.15.8 Program.inst . . . 113
1.15.9 Program.gate . . . 114
1.15.10 Program.defgate . . . 114
1.15.11 Program.define_noisy_gate . . . 114
1.15.12 Program.define_noisy_readout . . . 115
1.15.13 Program.no_noise . . . 115
1.15.14 Program.measure . . . 115
1.15.15 Program.reset . . . 116
1.15.16 Program.measure_all . . . 116
1.15.17 Program.alloc . . . 116
1.15.18 Program.declare . . . 116
1.15.19 Program.wrap_in_numshots_loop . . . 117
1.15.20 Program.while_do . . . 117
1.15.21 Program.if_then . . . 117
1.15.22 Program.copy . . . 118
1.15.23 Program.pop . . . 118
1.15.24 Program.dagger . . . 118
1.15.25 Program.__getitem__ . . . 118
1.15.26 Utility Functions . . . 119
1.16 Gates and Instructions . . . 121
1.16.1 Native gates for Rigetti QPUs . . . 121
1.16.2 All gates and instructions . . . 122
1.17 Pauli Operators . . . 135
1.17.1 Construction functions . . . 135
1.17.2 Working with operators . . . 137
1.17.3 Classes . . . 139
1.18 Quantum Computer . . . 142
1.18.1 QuantumComputer.run . . . 144
1.18.2 QuantumComputer.run_and_measure . . . 144
1.18.3 QuantumComputer.run_symmetrized_readout . . . 145
1.18.4 QuantumComputer.qubits . . . 145
1.18.5 QuantumComputer.qubit_topology . . . 146
1.18.6 QuantumComputer.get_isa . . . 146
1.18.7 QuantumComputer.compile . . . 146
1.19 Compilers . . . 146
1.19.1 pyquil.api._qac.AbstractCompiler . . . 147
1.19.2 pyquil.api.QVMCompiler . . . 147
1.19.3 pyquil.api.QPUCompiler . . . 148
1.20 QAMs . . . 148

1.20.1 QAM . . . 149
1.20.2 QPU . . . 149
1.20.3 QVM . . . 149
1.20.4 PyQVM . . . 150
1.21 Devices . . . 150
1.21.1 pyquil.device.AbstractDevice . . . 150
1.21.2 pyquil.device.Device . . . 151
1.21.3 pyquil.device.NxDevice . . . 152
1.21.4 pyquil.device.ISA . . . 152
1.21.5 pyquil.device.Specs . . . 153
1.21.6 Utility functions . . . 154
1.22 Simulators . . . 155
1.22.1 WavefunctionSimulator . . . 155
1.22.2 ReferenceWavefunctionSimulator . . . 155
1.22.3 ReferenceDensitySimulator . . . 155
1.22.4 NumpyWavefunctionSimulator . . . 156
1.22.5 Reference Utilities . . . 156
1.22.6 Numpy Utilities . . . 157
1.23 Noise Models . . . 158
1.23.1 Functions . . . 158
1.23.2 Classes . . . 164
1.24 Operator Estimation . . . 167
1.24.1 Data structures . . . 167
1.24.2 Methods . . . 168
1.24.3 Utilities . . . 170

2 Indices and Tables **171**

Bibliography **173**

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 the *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:

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 open source 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. Find QVM on GitHub.
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 open source compiler will take arbitrary Quil and compile it for the given QAM, according to its supported instruction set architecture. Find quilc on GitHub.
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.

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

CHAPTER 1

Contents

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 pyquil
```

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

```
pip install --upgrade 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 (`quilc`) 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.

Note: You can also find the open source code for quilc and qvm on GitHub, where you can find instructions for compiling, installing, and contributing to the compiler and QVM.

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
quilc --version
```

To uninstall, delete the following files:

```
/usr/local/bin/qvm
/usr/local/bin/quilc
/usr/local/share/man/man1/qvm.1
/usr/local/share/man/man1/quilc.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
quilc --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
quilc --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-2.1-linux-barebones
```

From here, run the following command:

```
./forest-sdk-2.1-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 libzmq3-dev
```

Or on any rhel-derivative systems (e.g. Amazon Linux) with

```
sudo yum install -y lapack-devel blas-devel epel-release
sudo yum install -y zeromq3-devel
```

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

... - Launching quilc.
... - Spawning server at (tcp://*:5555) .
```

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.

NOTE: We are transitioning from using an HTTP `quilc` server to an RPCQ one. In the near term, `-S` will spawn an HTTP server at port 6000 and an RPCQ server at port 5555 (accessible via `tcp://localhost:5555`).

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 $|00\rangle$ and $|11\rangle$, 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 Computer*). We've also imported all (`*`) gates from the `pyquil.gates` module, which allows us to add operations to our program (*Programs and Gates*).

Note: PyQuil also provides a handy function for you to ensure that a local qvm and quilc are currently running in your environment. To make sure both are available you import *from pyquil.api import local_qvm* and then run *local_qvm()*. This will start a qvm and quilc instances using subprocesses if they have not already been started. You can also use it as a context manager as in the following example:

```
from pyquil import get_qc, Program
from pyquil.gates import CNOT, Z
from pyquil.api import local_qvm

qvm = get_qc('9q-square-qvm')
prog = Program(Z(0), CNOT(0, 1))

with local_qvm():
    results = qvm.run_and_measure(prog, trials=10)
```

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 *New in Forest 2 - Other*. Once you're set with that, jump to *Programs and Gates* to continue.

1.2 New in Forest 2 - Running on the QVM

PyQuil is for constructing and running quantum programs on real quantum computers. With the release of pyQuil 2, we have changed parts of the API (compared to pyQuil 1.x) to better reflect that focus.

1.2.1 Program construction is largely the same

To construct run-and-measure-style programs, there are no changes in program construction. When using explicit `MEASURE` instructions or using parametric programs for massive speed increases, please read about the *new quil memory model*.

Below, we build a program that constructs a Bell state.

[1]:
```
from pyquil import Program
from pyquil.gates import *
program = Program(
    H(0),
    CNOT(0, 1),
)
print(program)
```

```
H 0
CNOT 0 1
```

1.2.2 Unphysical methods deprecated in `QVMConnection`

In pyQuil 1.x, you would use the `QVMConnection` to run quantum programs on a web-hosted quantum simulator (the "Quantum Virtual Machine"). To run on a real quantum processor (QPU), you would switch all instances of `QVMConnection` to `QPUConnection`. However, `QVMConnection` let you do many unphysical operations that are unsupported on a real QPU. These methods are detailed below and have been deprecated in favor of a new API that clearly delineats the physical from the unphysical.

Wavefunction

When simulating a quantum program, we can inspect its wavefunction. This is very useful! It is–however–impossible to inspect the wavefunction of a real quantum processor. This is one of the central tenets of quantum information! Attempting to switch instances of `QVMConnection` to `QPUConnection` results in an error. Additionally, using `QVMConnection` with a noise model will cause the `wavefunction` call to behave badly.

The old way

[2]:
```
from pyquil.api import QVMConnection
wfn_old = QVMConnection().wavefunction(program)
print(wfn_old)
```

```
(0.7071067812+0j)|00> + (0.7071067812+0j)|11>
```

The new way

`WavefunctionSimulator` encapsulates all functionality that requires peering into a wavefunction. This also opens the door for different types of simulators other than those backed by a wavefunction. For example, you can simulate a quantum circuit with a density matrix simulation or a path integral simulation.

[3]:
```
from pyquil.api import WavefunctionSimulator
wfn = WavefunctionSimulator().wavefunction(program)
print(wfn)
```

```
(0.7071067812+0j)|00> + (0.7071067812+0j)|11>
```

Expectation

Many quantum algorithms involve calculating expectation values of quantum observables. We can represent quantum observables with `PauliSum`s. When using a simulator, it is possible to exactly calculate expectation values (a consequence of having access to the full wavefunction) whereas when running on a QPU you must estimate expectation values by sampling from many runs of a program.

Let's use our program defined above to construct a bell state.

[4]:
```
print(program)
```

```
H 0
CNOT 0 1
```

And we wish to measure the following quantum observables O - the expected value of the 0th bit $O = (1 - Z_0)/2$ - the expected value of the 1st bit $O = (1 - Z_1)/2$ - the expected value of the exclusive or (XOR) between the two qubits $O = (1 - ZZ)/2$

Exercise for the reader: convince yourself that $(1 - ZZ)/2$ is the XOR function

[5]:
```
from pyquil.paulis import sZ

z0 = (1-sZ(0))*0.5
z1 = (1-sZ(1))*0.5
xor = (1-sZ(0)*sZ(1))*0.5
```

A Bell state is entangled, so each of the 1-qubit operators should give an expectation value of 0.5 (as each qubit taken individually has a 50% change of being measured in the 0 or 1 state) whereas the two-qubit operator should give an expectation value of 0 (as the two qubits considered together will always have even parity)

The old way

Because of technical details, `QVMConnection.expectation` requires transforming your observable represented as a `PauliSum` into a program, and then remembering to multiply back in the coefficients. A more usable API was introduced in Pyquil 1.9 and was given the name `pauli_expectation`.

[6]:
```
for observable in [z0, z1, xor]:
    expectation = QVMConnection().pauli_expectation(prep_prog=program, pauli_
↪terms=observable)
    print(observable, '\t', expectation)
```

```
(0.5+0j)*I + (-0.5+0j)*Z0         (0.4999999999999999+0j)
(0.5+0j)*I + (-0.5+0j)*Z1         (0.4999999999999999+0j)
(0.5+0j)*I + (-0.5+0j)*Z0*Z1      0j
```

The new way

If you want analytical expectation values, the solution is to use `WavefunctionSimulator.expectation`

Note that the method is not named `pauli_expectation` as we do not support the pyQuil<1.9 way of using `Programs` to represent `PauliSums`

[7]:
```
for observable in [z0, z1, xor]:
    expectation = WavefunctionSimulator().expectation(prep_prog=program, pauli_
↪terms=observable)
    print(observable, '\t', expectation)
```

```
(0.5+0j)*I + (-0.5+0j)*Z0         (0.4999999999999999+0j)
(0.5+0j)*I + (-0.5+0j)*Z1         (0.4999999999999999+0j)
(0.5+0j)*I + (-0.5+0j)*Z0*Z1      0j
```

Sampling expectations on a QPU.

Estimating expectation values via *sampling* from a QPU is often something we would like to do! Please look forward for helper functions for sampling observables in the near future.

1.2.3 Async methods removed

pyQuil 2 is tailored for Rigetti's Quantum Cloud Services (QCS). In prior releases, executing programs on either the QVM or QPU involved communicating with a cloud-hosted endpoint. Now, you are empowered with

1. A pre-configured quantum machine image (QMI) with your own dedicated qvm and quilc instance.
2. low-latency QPU access all to yourself during a scheduled time window
3. A downloadable local version of the qvm and quilc

Taken together, there is no longer any motivation for supporting asynchronous access to either the QVM or QPU.

When running many programs it was often possible to reduce runtime by batching jobs and exploiting the async queue. The following example does not work in pyQuil 2 but gives a sketch about how this would have worked.

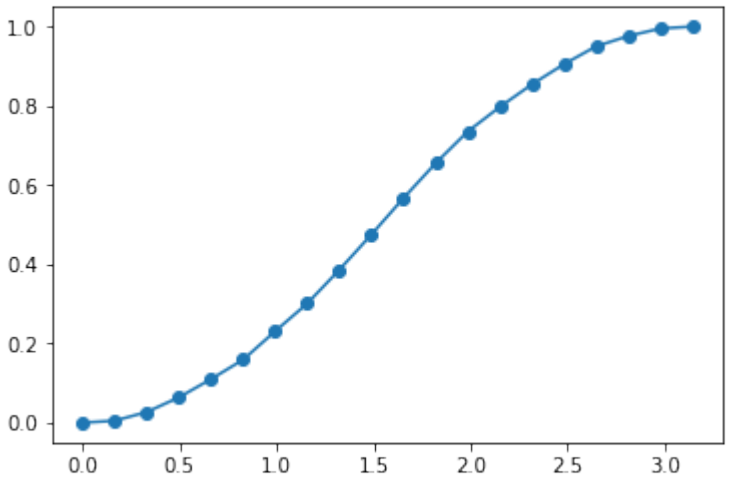
```
import numpy as np
cxn = QVMConnection()
thetas = np.linspace(0, 2*np.pi, 20)
jobs = [Program(RY(theta, 0)) for theta in thetas]
job_ids = [cxn.run_and_measure_async(job, qubits=[0], trials=1000) for job in jobs]
bitstrings = [np.asarray(cxn.wait_for_job(job)) for job in jobs]
```

Since this is such an important use case, there have been many changes to support running many programs as quickly as possible. We demonstrate an equivalent, synchronous version of the example given above. **To idiomatically run this set of jobs, there are additional features you should use that are not covered in this document**. Please continue reading the documentation, especially the page covering *parametric programs*.

[8]:
```
import numpy as np
cxn = QVMConnection()
thetas = np.linspace(0, np.pi, 20)
bitstrings = [np.asarray(cxn.run_and_measure(Program(RY(theta, 0)), qubits=[0],␣
↪trials=1000))
              for theta in thetas]
```

[9]:
```
%matplotlib inline
from matplotlib import pyplot as plt

averages = [np.mean(bs[:,0]) for bs in bitstrings]
_ = plt.plot(thetas, averages, 'o-')
```



1.3 New in Forest 2 - QuantumComputer

PyQuil is for constructing and running quantum programs on real quantum computers. With the release of pyQuil 2, we have changed parts of the API to better reflect that focus. Instead of swapping between a `QVMConnection` and a `QPUConnection`, you will primarily deal with a `QuantumComputer` with consistent API and behavior regardless of

- QVM / QPU
- Presence of noise model
- Device topology

1.3.1 Running a program

Let's show how you can run a simple program on a `QuantumComputer` first we start with the relevant imports.

[1]:
```
from pyquil import Program
from pyquil.gates import *
```

We'll write a function that takes a list of qubits and returns a pyQuil `Program` that constructs an entangled "GHZ" state. This is a generalization of the two-qubit Bell state.

[2]:
```
def ghz_state(qubits):
    """Create a GHZ state on the given list of qubits by applying
    a Hadamard gate to the first qubit followed by a chain of CNOTs
    """
    program = Program()
    program += H(qubits[0])
    for q1, q2 in zip(qubits, qubits[1:]):
        program += CNOT(q1, q2)
    return program
```

For example, creating a GHZ state on qubits 1, 2, and 3 would look like:

[3]:
```
program = ghz_state(qubits=[0, 1, 2])
print(program)
```

```
H 0
CNOT 0 1
CNOT 1 2
```

1.3.2 Debugging with `WavefunctionSimulator`

We can check that this program gives us the desired wavefunction by using WavefunctionSimulator.wavefunction()

[4]:
```
from pyquil.api import WavefunctionSimulator
wfn = WavefunctionSimulator().wavefunction(program)
print(wfn)
```

```
(0.7071067812+0j)|000> + (0.7071067812+0j)|111>
```

We can't get the wavefunction from a real quantum computer though, so instead we'll sample bitstrings. We expect to always measure the bitstring 000 or the bitstring 111 based on the definition of a GHZ state and confirmed by our wavefunction simulation.

1.3.3 `get_qc`

We'll construct a `QuantumComputer` via the helper method get_qc. You may be tempted to use the `QuantumComputer` constructor directly. Please refer to the *advanced documentation* to see how to do that. Our program uses 3 qubits, so we'll ask for a 3-qubit QVM.

[5]:
```
from pyquil import get_qc
qc = get_qc('3q-qvm')
qc
```

[5]:
```
QuantumComputer[name="3q-qvm"]
```

We can do a quick check to make sure it has 3 qubits

[6]:
```
qc.qubits()
```

[6]:
```
[0, 1, 2]
```

1.3.4 Sampling with `run_and_measure`

`QuantumComputer.run_and_measure` will run a given program (that does not have explicit `MEASURE` instructions) and then measure *all qubits present in the quantum computer*.

[7]:
```
bitstrings = qc.run_and_measure(program, trials=10)
bitstrings
```

[7]:
```
{0: array([1, 0, 0, 1, 1, 1, 1, 0, 1, 0]),
 1: array([1, 0, 0, 1, 1, 1, 1, 0, 1, 0]),
 2: array([1, 0, 0, 1, 1, 1, 1, 0, 1, 0])}
```

Let's programatically verify that we always measure 000 or 111 by "summing" each bitstring and checking if it's eather 0 (for 000) or 3 (for 111)

[8]:
```
import numpy as np
bitstring_array = np.vstack(bitstrings[q] for q in qc.qubits()).T
sums = np.sum(bitstring_array, axis=1)
sums
```

[8]:
```
array([3, 0, 0, 3, 3, 3, 3, 0, 3, 0])
```

[9]:
```
sample_is_ghz = np.logical_or(sums == 0, sums == 3)
sample_is_ghz
```

[9]:
```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True])
```

[10]:
```
np.all(sample_is_ghz)
```

[10]:
```
True
```

1.3.5 Change alert: `run_and_measure` will return a dictionary of 1d bitstrings.

Not a 2d array. To demonstrate why, consider a lattice whose qubits are not contiguously indexed from 0.

[11]:
```
# TODO: we need a lattice that is not zero-indexed
# qc = get_qc('Aspen-0-3Q-B')
# qc.run_and_measure(ghz_state(qubits=[1,2,3]))
```

1.3.6 Change alert: All qubits are measured

PyQuil 1.x's `run_and_measure` would only measure qubits used in the given program. Now all qubits (per `qc.qubits()`) are measured. This is easier to reason about and reflects the reality of running on a QPU. When accounting for noise or when running QCVV tasks, you may be interested in the measurement results of qubits that weren't even used in your program!

[12]:
```
qc = get_qc('4q-qvm')
bitstrings = qc.run_and_measure(Program(X(0), X(1), X(2)), trials=10)
bitstrings
```

[12]:
```
{0: array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1]),
 1: array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1]),
 2: array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1]),
 3: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])}
```

You can drop qubits you're not interested in by indexing into the returned dictionary

[13]:
```
# Stacking everything
np.vstack(bitstrings[q] for q in qc.qubits()).T
```

[13]:
```
array([[1, 1, 1, 0],
       [1, 1, 1, 0],
       [1, 1, 1, 0],
       [1, 1, 1, 0],
       [1, 1, 1, 0],
       [1, 1, 1, 0],
       [1, 1, 1, 0],
       [1, 1, 1, 0],
       [1, 1, 1, 0],
       [1, 1, 1, 0]])
```

[14]:
```
# Stacking what you want (contrast with above)
qubits = [0, 1, 2]
np.vstack(bitstrings[q] for q in qubits).T
```

[14]:
```
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1],
       [1, 1, 1],
       [1, 1, 1],
       [1, 1, 1],
       [1, 1, 1],
       [1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
```

1.3.7 Change alert: `run_and_measure` works with noise models now.

In pyQuil 1.x, `run_and_measure` would not work with noise models. Now noise models are supported. Pre-configured noise models can be used via `get_qc('xxxx-noisy-qvm')`.

As a consequence, `run_and_measure` for large numbers of trials will be slower in Pyquil 2.

[15]:
```
qc = get_qc('3q-noisy-qvm')
bitstrings = qc.run_and_measure(program, trials=10)
bitstrings
```

[15]:
```
{0: array([0, 1, 1, 0, 0, 1, 1, 0, 1, 1]),
 1: array([0, 1, 1, 0, 0, 1, 1, 0, 1, 1]),
 2: array([0, 1, 1, 0, 0, 1, 1, 0, 1, 1])}
```

[16]:
```
bitstring_array = np.vstack(bitstrings[q] for q in qc.qubits()).T
sums = np.sum(bitstring_array, axis=1)
sums
```

[16]:
```
array([0, 3, 3, 0, 0, 3, 3, 0, 3, 3])
```

[17]:
```
# Noise means now we measure things other than 000 or 111
np.all(np.logical_or(sums == 0, sums == 3))
```

[17]:
```
True
```

1.3.8 `list_quantum_computers`

You can find all possible arguments to `get_qc` with `list_quantum_computers`

[18]:
```
from pyquil import list_quantum_computers
# TODO: unauthenticated endpoint
# list_quantum_computers()
```

1.3.9 `QuantumComputer`s have a topology

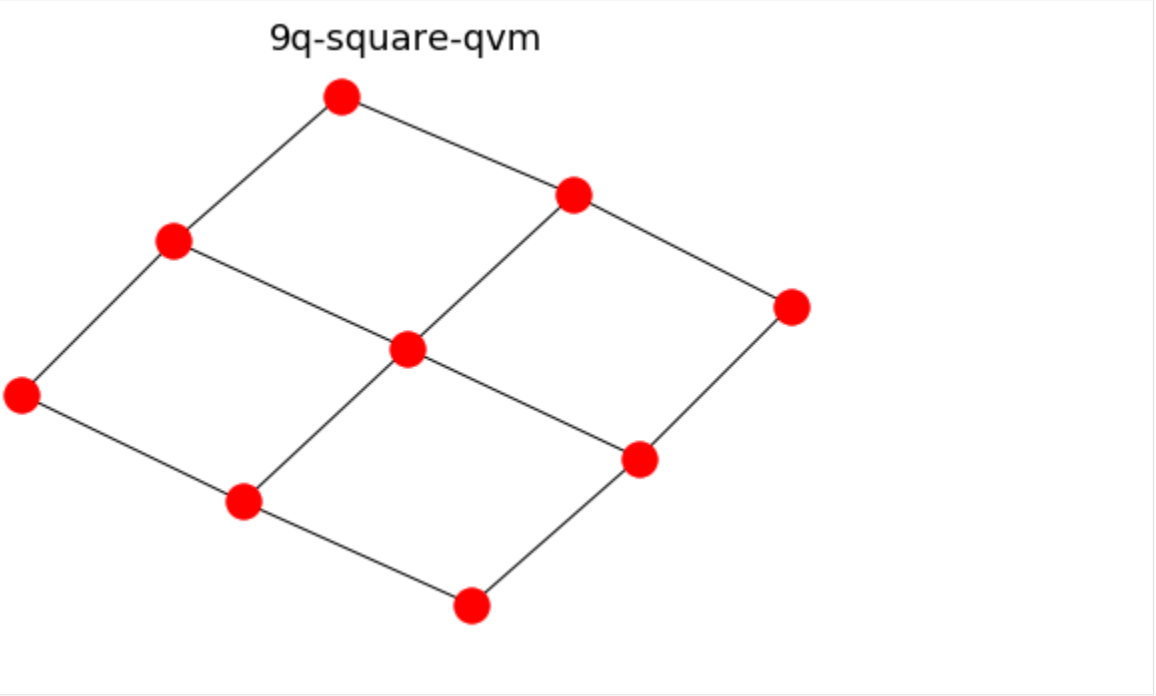
An important restriction when running on a real quantum computer is the mapping of qubits to the supported two-qubit gates. The QVM is designed to provide increasing levels of "realism" to guarantee that if your program executes successfully on `get_qc("Aspen-xxx-noisy-qvm")` then it will execute successfully on `get_qc("Aspen-xxx")`*

* guarantee not currently guaranteed. This is a work in progress.

1.3.10 Inspecting the topology

You can access a topology by `qc.qubit_topology()`, which will return a NetworkX representation of qubit connectivity. You can access the full set of supported instructions by `qc.get_isa()`. For example, we include a generic QVM named `"9q-square-qvm"` that has a square topology.

[19]:
```
qc = get_qc('9q-square-qvm')
%matplotlib inline
import networkx as nx
nx.draw(qc.qubit_topology())
from matplotlib import pyplot as plt
_ = plt.title('9q-square-qvm', fontsize=18)
```

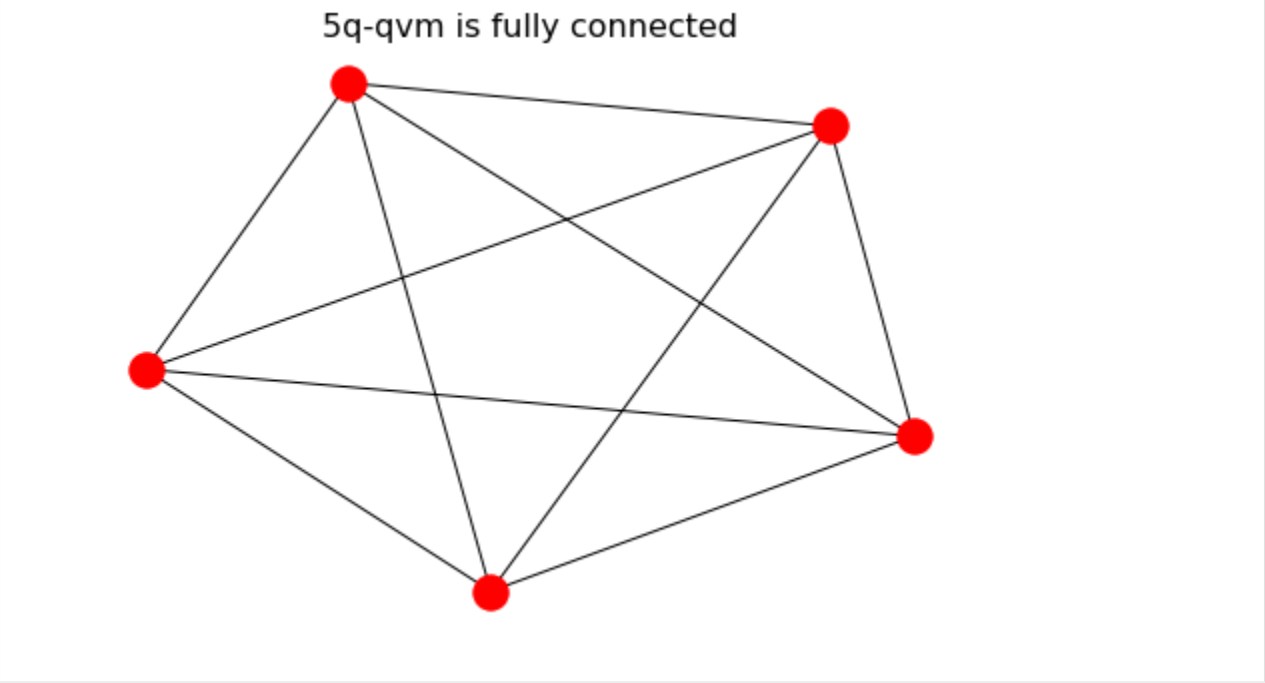


1.3.11 What If I don't want a topology?

`WavefunctionSimulator` still has no notion of qubit connectivity, so feel free to use that for simulating quantum algorithms that you aren't concerned about running on an actual QPU.

Above we used `get_qc("3q-qvm")`, `"4q-qvm"`, and indeed you can do any `"{n}q-qvm"` (subject to computational resource constraints). These QVM's are constructed with a topology! It just happens to be fully connected

[20]:
```
nx.draw(get_qc('5q-qvm').qubit_topology())
_ = plt.title('5q-qvm is fully connected', fontsize=16)
```



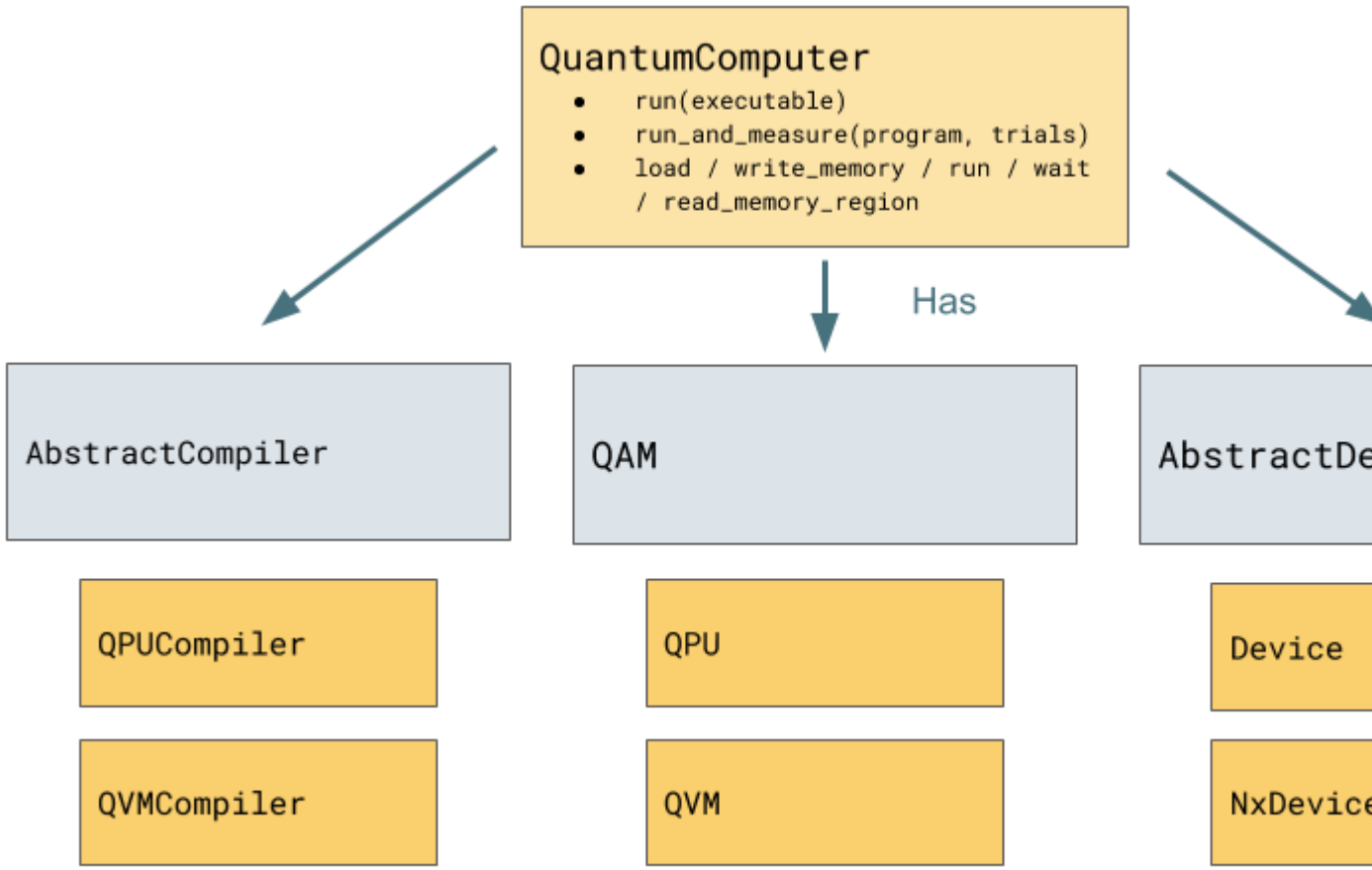
1.3.12 Heirarchy of realism

- `WavefunctionSimulator` to debug algorithm
- `get_qc("5q-qvm")` to debug sampling
- `get_qc("9q-square-qvm")` to debug mapping to a lattice
- `get_qc("9q-square-noisy-qvm")` to debug generic noise characteristics
- `get_qc("Aspen-0-16Q-A-qvm")` to debug mapping to a real lattice
- `get_qc("Aspen-0-16Q-A-noisy-qvm")` to debug noise characteristics of a real device
- `get_qc("Aspen-0-16Q-A")` to run on a real device

1.3.13 "What is a `QuantumComputer`?" Advanced Edition

A `QuantumComputer` is a wrapper around three constituent parts, each of which has a programatic interface that must be respected by all classes that implement the interface. By having clear interfaces we can write backend-agnostic methods on `QuantumComputer` and mix-and-match backing objects.

The following diagram shows the three objects that must be provided when constructing a `QuantumComputer` "by hand". The abstract classes are backed in grey with example implementing classes listed below. Please consult the *api reference* for details on each interface.



As an example, let's construct a 5-qubit QVM with one central node and only even numbered qubits.

[21]:
```
topology = nx.from_edgelist([
    (10, 2),
    (10, 4),
    (10, 6),
    (10, 8),
])
from pyquil.device import NxDevice
device = NxDevice(topology)

from pyquil.api._qac import AbstractCompiler
class MyLazyCompiler(AbstractCompiler):
    def quil_to_native_quil(self, program):
        return program

    def native_quil_to_executable(self, nq_program):
        return nq_program

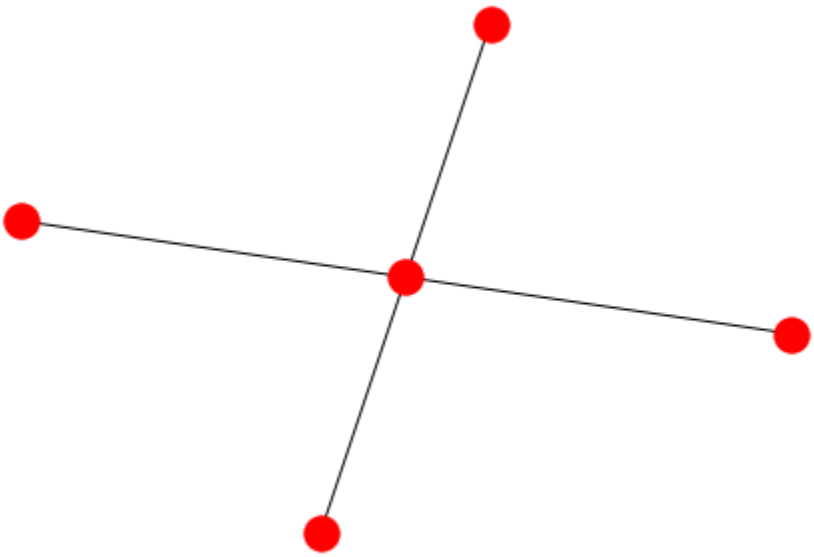

from pyquil.api import QuantumComputer, QVM, ForestConnection
```

(continues on next page)

(continued from previous page)

```
my_qc = QuantumComputer(
    name='my-qvm',
    qam=QVM(connection=ForestConnection()),
    device=device,
    compiler=MyLazyCompiler(),
)

nx.draw(my_qc.qubit_topology())
```



[22]:
```
my_qc.run_and_measure(Program(X(10)), trials=5)
```

[22]:
```
{2: array([0, 0, 0, 0, 0]),
 4: array([0, 0, 0, 0, 0]),
 6: array([0, 0, 0, 0, 0]),
 8: array([0, 0, 0, 0, 0]),
 10: array([1, 1, 1, 1, 1])}
```

1.4 New in Forest 2 - Parametric Programs

pyQuil is for constructing and running hybrid quantum/classical algorithms on real quantum computers. With the release of pyQuil 2, we have changed parts of the API to take advantage of some exciting new features available on QCS.

A hybrid algorithm involves using the quantum computer to create a quantum state that would be difficult to prepare classically; measure it in a way particular to your problem; and then update your procedure for creating the state so that the measurements are closer to the correct answer. A *real* hybrid algorithm involves structured ansatze like QAOA for optimization or a UCC ansatz for chemistry. Here, we'll do a much simpler parameterized program

[1]:
```
from pyquil import Program, get_qc
from pyquil.gates import *

def ansatz(theta):
```

(continues on next page)

(continued from previous page)

```
    program = Program()
    program += RY(theta, 0)
    return program

print(ansatz(theta=0.2))
```

```
RY(0.2) 0
```

1.4.1 Scan over the parameter (the old way)

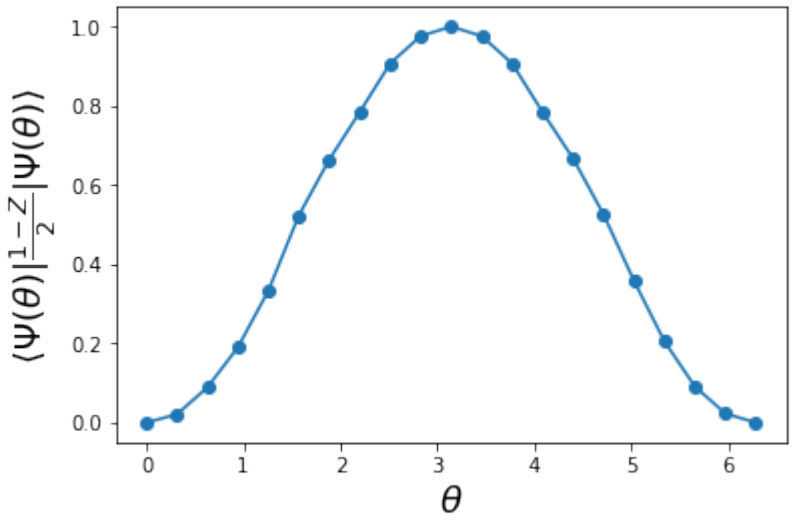
For this extrordinarily simple ansatz, we can discretize the parameter theta and try all possible values. As the number of parameters increases, the number of combinations increases exponentially so doing a full grid search will become intractable for anything more than ~two parameters.

[2]:
```
import numpy as np
qc = get_qc("9q-square-qvm")

thetas = np.linspace(0, 2*np.pi, 21)
results = []
for theta in thetas:
    program = ansatz(theta)
    bitstrings = qc.run_and_measure(program, trials=1000)
    results.append(np.mean(bitstrings[0]))
```

[3]:
```
%matplotlib inline
from matplotlib import pyplot as plt

plt.plot(thetas, results, 'o-')
plt.xlabel(r'$\theta$', fontsize=18)
_ = plt.ylabel(r'$\langle \Psi(\theta) | \frac{1 - Z}{2} | \Psi(\theta) \rangle$',
               fontsize=18)
```



1.4.2 Do an optimization (the old way)

Instead of doing a full grid search, we will employ a classical optimizer to find the best parameter values. Here we use scipy to find the theta that results in sampling the most `1`s in our resultant bitstrings.

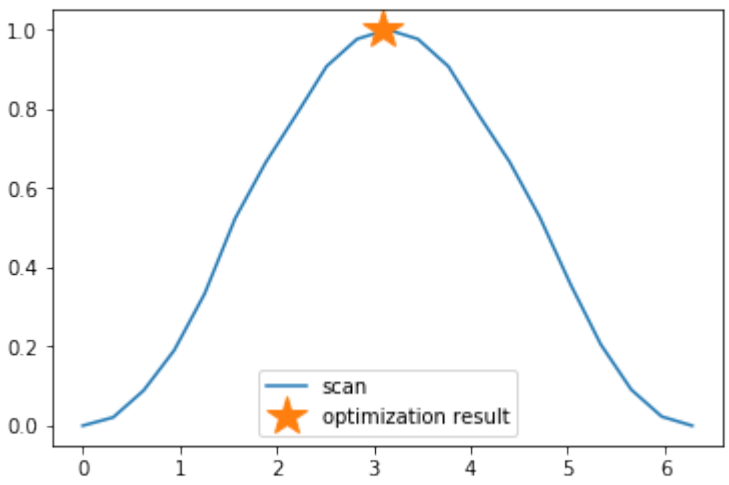
[4]:
```
def objective_function(theta):
    program = ansatz(theta[0])
    bitstrings = qc.run_and_measure(program, trials=1000)
    result = np.mean(bitstrings[0])
    return -result

import scipy.optimize
res = scipy.optimize.minimize(objective_function, x0=[0.1], method='COBYLA')
res
```

[4]:
```
     fun: -1.0
   maxcv: 0.0
 message: 'Optimization terminated successfully.'
    nfev: 13
  status: 1
 success: True
       x: array([3.1])
```

[5]:
```
plt.plot(thetas, results, label='scan')
plt.plot([res.x], [-res.fun], '*', ms=20, label='optimization result')
plt.legend()
```

[5]:
```
<matplotlib.legend.Legend at 0x1015dedf28>
```



1.4.3 Compilation

Prior to QCS, a QPU job would be routed via a series of cloud-based queues and compilation steps. With Forest 2, you are in control of the two stages of compilation so you can amortize the cost of compiling. Your QMI and all classical infrastructure is hosted on the Rigetti premises, so network latency is minimal.

Quil to native quil

The first step of compilation converts gates to their hardware-supported equivalent. For example, our parametric RY is converted into RX's and RZ's because these are physically realizable on a Rigetti QPU

[6]:
```
nq_program = qc.compiler.quil_to_native_quil(ansatz(theta=0.5))
print(nq_program)
```

```
PRAGMA EXPECTED_REWIRING "#(0 1 2 3 4 5 6 7 8)"
RX(pi/2) 0
RZ(0.5) 0
RX(-pi/2) 0
PRAGMA CURRENT_REWIRING "#(0 1 2 3 4 5 6 7 8)"
PRAGMA EXPECTED_REWIRING "#(0 1 2 3 4 5 6 7 8)"
PRAGMA CURRENT_REWIRING "#(0 1 2 3 4 5 6 7 8)"
```

Native quil to executable

The second step of compilation will turn named gates into calibrated pulses stored in a binary format suitable for consumption by the control electronics. This means that you can fully compile a given program and run it many times with minimal classical overhead.

Note: since we're using a QVM, for which there is no binary format, this stage is mocked out and you can see the original Quil inside the `PyQuilExecutableResponse` that is returned. When running on the QPU, this will return a `BinaryExecutableResponse` whose contents are opaque.

TODO: obscure the contents of `PyQuilExecutableResponse`: https://github.com/rigetti/pyquil/issues/700

[7]:
```
qc.compiler.native_quil_to_executable(nq_program)
```

[7]:
```
PyQuilExecutableResponse(attributes={'native_quil_metadata': {'final-rewiring': [0, 1,
↪ 2, 3, 4, 5, 6, 7, 8], 'topological_swaps': 0, 'gate_depth': 3, 'gate_volume': 3,
↪'program_duration': 18.01, 'program_fidelity': 1.0, 'multiqubit_gate_depth': 0},
↪'num_shots': 1}, program='PRAGMA EXPECTED_REWIRING "#(0 1 2 3 4 5 6 7 8)"\nRX(pi/2)␣
↪0\nRZ(0.5) 0\nRX(-pi/2) 0\nPRAGMA CURRENT_REWIRING "#(0 1 2 3 4 5 6 7 8)"\nPRAGMA␣
↪EXPECTED_REWIRING "#(0 1 2 3 4 5 6 7 8)"\nPRAGMA CURRENT_REWIRING "#(0 1 2 3 4 5 6␣
↪7 8)"\n')
```

1.4.4 Parametric compilation

This doesn't buy us much if we have to know exactly what circuit we want to run before compiling it and amortizing the compilation cost. Maybe you could get away with it when you're doing a parameter scan, but for hybrid algorithms, the circuit parameter (here: `theta`) depends on the results of a circuit before. This is the essence of hybrid programming! Therefore, all compilation steps have been upgraded to support named, symbolic parameters that will be updated at runtime with minimal overhead.

With this feature, you can compile a parametric program once and run it many times will different parameter values *and* you need not know the parameter values at compilation time.

There are a couple of prerequisites to use this feature effectively from PyQuil, which we address in this document.

First, you must declare a parameter when constructing your quil program. When declaring a named classical variable, you must specify at least a name and a type. It is conventional to make sure the Python variable name of the reference memory matches the Quil variable name. In our case, we name both things `theta`. Our circuit above would be modified in this way:

[8]:
```
program = Program()
theta = program.declare('theta', memory_type='REAL')
program += RY(theta, 0)
print(program)
```

```
DECLARE theta REAL[1]
RY(theta) 0
```

1.4.5 Measuring

In the documentation so far, we've been using the `run_and_measure` functionality of `QuantumComputer`. It's time to get our hands dirty and introduce explicit measure instructions.

Above, we declared a classical piece of memory, we've given it a name (`theta`), and we've given it a type (`REAL`). The bits that we measure (or "read out" – `ro` for short) must now also be declared, given a name, and a type. Additionally, we'll usually be measuring more than one qubit so we can give this register a size.

The index of the readout register need not match the qubit index. For example below, we create a bell state on qubits 5 and 6 and measure the readout results into `ro[0]` and `ro[1]`.

Note: The readout register must be named "ro" (for now)

[9]:
```
program = Program()
ro = program.declare('ro', memory_type='BIT', memory_size=2)
program += H(5)
program += CNOT(5, 6)
program += MEASURE(5, ro[0])
program += MEASURE(6, ro[1])
print(program)
```

```
DECLARE ro BIT[2]
H 5
CNOT 5 6
MEASURE 5 ro[0]
MEASURE 6 ro[1]
```

Our very simple ansatz only has one qubit, so the measurement is quite simple.

[10]:
```
program = Program()
theta = program.declare('theta', memory_type='REAL')
ro = program.declare('ro', memory_type='BIT', memory_size=1)
program += RY(theta, 0)
program += MEASURE(0, ro[0])
print(program)
```

```
DECLARE theta REAL[1]
DECLARE ro BIT[1]
RY(theta) 0
MEASURE 0 ro[0]
```

1.4.6 Number of shots

The number of trials is compiled into the executable binary, so we must specify this number *prior* to compilation.

TODO: add to str / repr https://github.com/rigetti/pyquil/issues/701

[11]:
```
program = Program()
theta = program.declare('theta', memory_type='REAL')
ro = program.declare('ro', memory_type='BIT', memory_size=1)
program += RY(theta, 0)
program += MEASURE(0, ro[0])
program.wrap_in_numshots_loop(shots=1000)
print(program)
```

```
DECLARE theta REAL[1]
DECLARE ro BIT[1]
RY(theta) 0
MEASURE 0 ro[0]
```

1.4.7 Using `qc.run()`

To use the lower-level but more powerful `qc.run` interface, we have had to take control of these three things

1. We decalred a read-out register named `ro` of type `BIT` and included explicit `MEASURE` instructions. Since this sets up a (potentially sprase) mapping from qubits to classical addresses, we can expect `qc.run()` to return the classic 2d ndarray of yore instead of the dictionary returned by `run_and_measure`
2. We have called `program.wrap_in_numshots_loop()` prior to compilation so the number of shots can be encoded in an efficient binary representation of the program
3. We have taken control of compilation; either by calling `qc.compile(program)` or by using the lower-level functions: `nq_program = qc.compiler.quil_to_native_quil(program) executable = qc.compiler.native_quil_to_executable(nq_program)`

[12]:
```
def ansatz(theta):
    program = Program()
    ro = program.declare('ro', memory_type='BIT', memory_size=1)
    program += RY(theta, 0)
    program += MEASURE(0, ro[0])
    return program

print(ansatz(theta=np.pi))
```

```
DECLARE ro BIT[1]
RY(pi) 0
MEASURE 0 ro[0]
```

We can run the program with a pre-set angle (here, `theta = np.pi`).

[13]:
```
program = ansatz(theta=np.pi)
program.wrap_in_numshots_loop(shots=5)
executable = qc.compile(program)
bitstrings = qc.run(executable)
print(bitstrings.shape)
bitstrings
```

```
(5, 1)
```

[13]:
```
array([[1],
       [1],
       [1],
```

(continues on next page)

(continued from previous page)

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

1.4.8 Scan over the parameter (the new way)

Finally, all the pieces are in place to compile and run parameterized executable binaries. We declare parameters that will be compiled symbolically into the binary allowing us to amortize the cost of compilation when running hybrid algorithms.

[14]:
```
def ansatz():
    program = Program()
    theta = program.declare('theta', memory_type='REAL')
    ro = program.declare('ro', memory_type='BIT', memory_size=1)
    program += RY(theta, 0)
    program += MEASURE(0, ro[0])
    return program

print(ansatz())
```

```
DECLARE theta REAL[1]
DECLARE ro BIT[1]
RY(theta) 0
MEASURE 0 ro[0]
```

1.4.9 Using `memory_map`

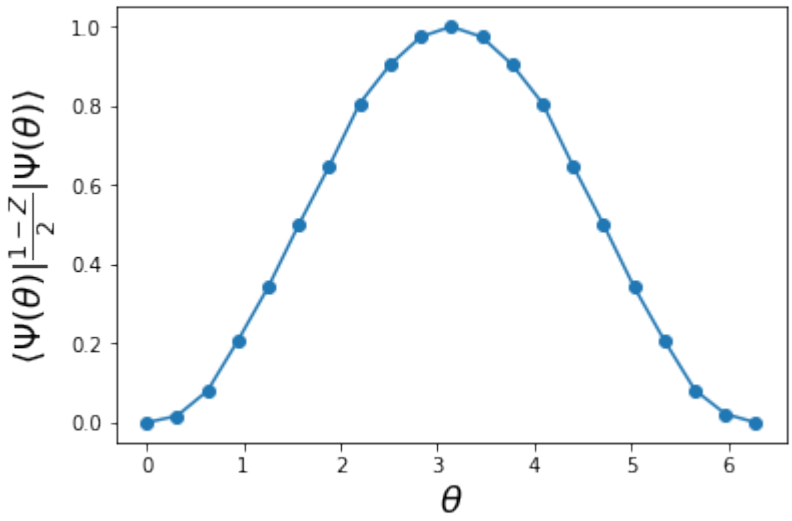
Now, when we call `qc.run` we provide a `memory_map` argument which will substitute in values for previously-declared Quil variables in a pre-compiled executable.

[15]:
```
program = ansatz()  # look ma, no arguments!
program.wrap_in_numshots_loop(shots=1000)
executable = qc.compile(program)

thetas = np.linspace(0, 2*np.pi, 21)
results = []
for theta in thetas:
    bitstrings = qc.run(executable, memory_map={'theta': [theta]})
    results.append(np.mean(bitstrings[:, 0]))

%matplotlib inline
from matplotlib import pyplot as plt

plt.plot(thetas, results, 'o-')
plt.xlabel(r'$\theta$', fontsize=18)
_ = plt.ylabel(r'$\langle \Psi(\theta) | \frac{1 - Z}{2} | \Psi(\theta) \rangle$',␣
↪fontsize=18)
```



1.4.10 Do an optimization (the new way)

Since parameters are compiled symbolically, we can do hybrid algorithms just as fast as parameter scans.

[16]:
```
program = ansatz()  # look ma, no arguments!
program.wrap_in_numshots_loop(shots=1000)
executable = qc.compile(program)

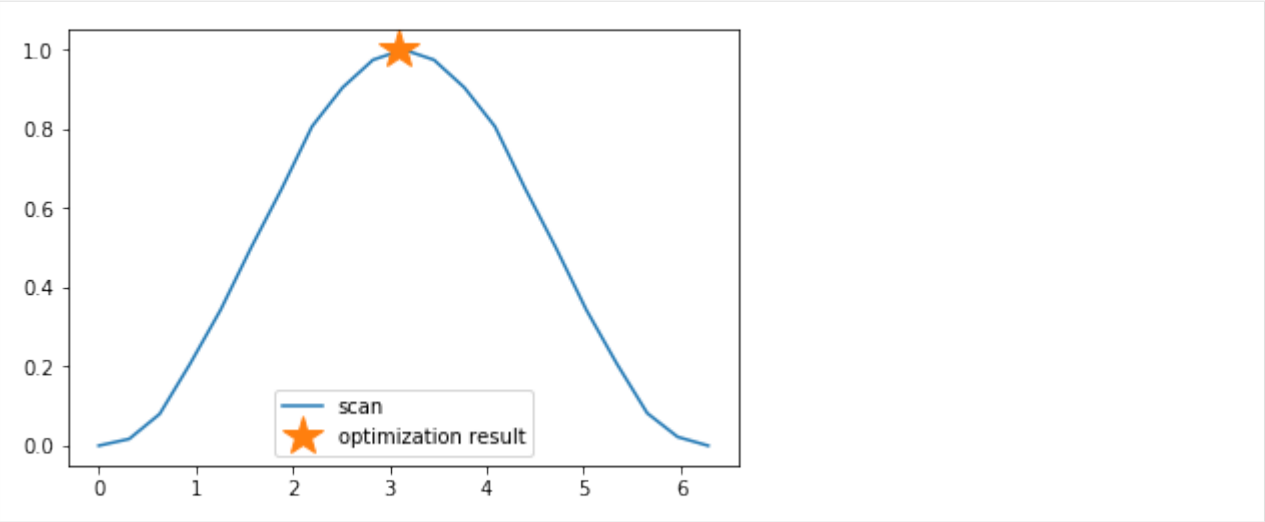
def objective_function(thetas):
    bitstrings = qc.run(executable, memory_map={'theta': thetas})
    result = np.mean(bitstrings[:, 0])
    return -result

res = scipy.optimize.minimize(objective_function, x0=[0.1], method='COBYLA')
res
```

[16]:
```
     fun: -1.0
   maxcv: 0.0
 message: 'Optimization terminated successfully.'
    nfev: 12
  status: 1
 success: True
       x: array([3.1])
```

[17]:
```
plt.plot(thetas, results, label='scan')
plt.plot([res.x], [-res.fun], '*', ms=20, label='optimization result')
plt.legend()
```

[17]:
```
<matplotlib.legend.Legend at 0x1015f13898>
```



1.5 New in Forest 2 - Other

There are many other changes to the Forest SDK (comprising pyQuil, Quil, the Quil Compiler, and the QVM).

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

1.5.1 Updates to the Quil language

The primary differences in the programming language Quil 1.0 (as appearing in pyQuil 1.x) and Quil 2 (as appearing in pyQuil 2) 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
```

(continues on next page)

(continued from previous page)

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

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]`.

1.5.2 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 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`.

1.5.3 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, 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, user data is stored only transiently, and it is the user's responsibility to handle long-term data storage on their QMI.
- Forest 1.3 refered to the software developer kit (pyQuil, QVM, Quilc) and the cloud platform for submitting jobs. Forest 2 is the SDK which you can install on your own computer or use pre-installed on a QMI. The entire platform is called Quantum Cloud Services (QCS).

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]
```

(continues on next page)

(continued from previous page)

```
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()

# 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 `BIT`s 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().wrap_in_numshots_loop(shots=shots)
binary = qc.compile(program)

# 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:
    bitstrings = qc.run(binary, {'theta': [angle_min + offset * angle_step]})

    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
```

(continues on next page)

(continued from previous page)

```
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


# 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().wrap_in_numshots_loop(shots=shots)
binary = qc.compile(program)

# 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:
    bitstrings = qc.run(binary, {'theta': [angle_min + offset * angle_step]})

    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):
```

(continues on next page)

(continued from previous page)

```
            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
```

1.5.4 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.

1.5.5 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 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'>"
      }
    }
  ],
```


(continues on next page)

(continued from previous page)

```
  "timestamp": "2018-09-11T18:40:19.253286",
  "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.

1.5.6 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)
```

1.6 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*.

1.6.1 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'
executable = qc.compile(p)
result = qc.run(executable)
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 Computer*. 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.

1.6.2 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.

1.6.3 Declaring Memory

Classical memory regions must be explicitly requested and named by a Quil program using the `DECLARE` directive. Details about the Quil directive can be found in declare.

In pyQuil, we declare memory with the `.declare` method on a `Program`. Let's inspect the function signature

```
# pyquil.quil.Program

def declare(self, name, memory_type='BIT', memory_size=1, shared_region=None,␣
↪offsets=None):
```

and break down each argument:

- `name` is any name you want to give this memory region.
- `memory_type` is one of `'REAL'`, `'BIT'`, `'OCTET'`, or `'INTEGER'` (given as a string). Only `BIT` and `OCTET` always have a determined size, which is 1 bit and 8 bits respectively.
- `memory_size` is the number of elements of that type to reserve.

- `shared_region` and `offsets` allow you to alias memory regions. For example, you might want to name the third bit in your readout array as `q3_ro`. `SHARING` is currently disallowed for our QPUs, so we won't focus on this here.

Now we can get into an example.

```
from pyquil import Program

p = Program()
ro = p.declare('ro', 'BIT', 16)
theta = p.declare('theta', 'REAL')
```

Warning: `.declare` cannot be chained, since it doesn't return a modified `Program` object.

Notice that the `.declare` method returns a reference to the memory we've just declared. We will need this reference to make use of these memory spaces again. Let's see how the Quil is looking so far:

```
DECLARE ro BIT[16]
DECLARE theta REAL[1]
```

That's all we have to do to declare the memory. Continue to the next section on *Measurement* to learn more about using `ro` to store measured readout results. Check out *Parametric Compilation* to see how you might use `theta` to compile gate parameters dynamically.

1.6.4 Measurement

There are several ways you can handle measurements in your program. We will start with the simplest method – letting the `QuantumComputer` abstraction do it for us.

```
from pyquil import Program, get_qc
from pyquil.gates import H, CNOT

# Get our QuantumComputer instance, with a Quantum Virutal Machine (QVM) backend
qc = get_qc("8q-qvm")

# Construct a simple Bell State
p = Program(H(0), CNOT(0, 1))

results = qc.run_and_measure(p, trials=10)
print(results)
```

```
{0: array([1, 1, 0, 1, 0, 0, 1, 1, 0, 1]),
 1: array([1, 1, 0, 1, 0, 0, 1, 1, 0, 1]),
 2: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
 3: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
 4: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
 5: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
 6: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
 7: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])}
```

The method `.run_and_measure` will handle declaring memory for readout results, adding `MEASURE` instructions for each qubit in the QVM, telling the QVM how many trials to run, running and returning the measurement results.

You might sometimes want finer grained control. In this case, we're probably only interested in the results on qubits 0 and 1, but `.run_and_measure` returns the results for all eight qubits in the QVM. We can change our program to

be more particular about what we want.

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

p = Program()
ro = p.declare('ro', 'BIT', 2)
p += H(0)
p += CNOT(0, 1)
p += MEASURE(0, ro[0])
p += MEASURE(1, ro[1])
```

In the last two lines, we've added our `MEASURE` instructions, saying that we want to store the result of qubit 0 into the 0th bit of `ro`, and the result of qubit 1 into the 1st bit of `ro`. The following snippet could be a useful way to measure many qubits, in particular, on a lattice that doesn't start at qubit 0 (although you can use the compiler to *re-index* your qubits):

```
qubits = [5, 6, 7]
# ...
for i, q in enumerate(qubits):
    p += MEASURE(q, ro[i])
```

Note: The QPU can only handle `MEASURE` final programs. You can't operate gates after measurements.

Specifying the number of trials

Quantum computing is inherently probabilistic. We often have to repeat the same experiment many times to get the results we need. Sometimes we expect the results to all be the same, such as when we apply no gates, or only an `X` gate. When we prepare a superposition state, we expect probabilistic outcomes, such as a 50% probability measuring 0 or 1.

The number of *shots* (also called *trials*) is the number of times to execute a program at once. This determines the length of the results that are returned.

As we saw above, the `.run_and_measure` method of the `QuantumComputer` object can handle multiple executions of a program. If you would like more explicit control for representing multi-shot execution, another way to do this is with `.wrap_in_numshots_loop`. This puts the number of shots to be run in the representation of the program itself, as opposed to in the arguments list of the execution method itself. Below, we specify that our program should be executed 1000 times.

```
p = Program()
...   # build up your program here...
p.wrap_in_numshots_loop(1000)
```

Note: Did You Know?

The word "shot" comes from experimental physics where an experiment is performed many times, and each result is called a shot.

1.6.5 Parametric Compilation

Modern quantum algorithms are often parametric, following a hybrid model. In this hybrid model, the program ansatz (template of gates) is fixed, and iteratively updated with new parameters. These new parameters are often determined by an update given by a classical optimizer. Depending on the complexity of the algorithm, problem of interest, and capabilities of the classical optimizer, this loop may need to run many times. In order to efficiently operate within this hybrid model, parametric compilation can be used.

Parametric compilation allows one to compile the program ansatz just once. Making use of declared memory regions, we can load values to the parametric gates at execution time, after compilation. Taking the compiler out of the execution loop for programs like this offers a huge performance improvement compared to compiling the program each time a parameter update is required. (Some more details about this and an example are found here.)

The first step is to build our parametric program, which functions like a template for all the precise programs we will run. Below we create a simple example program to illustrate, which puts the qubit onto the equator of the Bloch Sphere and then rotates it around the Z axis for some variable angle theta before applying another X pulse and measuring.

```
import numpy as np

from pyquil import Program
from pyquil.gates import RX, RZ, MEASURE

qubit = 0

p = Program()
ro = p.declare("ro", "BIT", 1)
theta_ref = p.declare("theta", "REAL")

p += RX(np.pi / 2, qubit)
p += RZ(theta_ref, qubit)
p += RX(-np.pi / 2, qubit)

p += MEASURE(qubit, ro[0])
```

Note: The example program, although simple, is actually more than just a toy example. It is similar to an experiment which measures the qubit frequency.

Notice how `theta` hasn't been specified yet. The next steps will have to involve a `QuantumComputer` or a compiler implementation. For simplicity, we will demonstrate with a `QuantumComputer` instance.

```
from pyquil import get_qc

# Get a Quantum Virtual Machine to simulate execution
qc = get_qc("1q-qvm")
executable = qc.compile(p)
```

We are able to compile our program, even with `theta` still not specified. Now we want to run our program with `theta` filled in for, say, 200 values between 0 and 2π. We demonstrate this below.

```
# Somewhere to store each list of results
parametric_measurements = []

for theta in np.linspace(0, 2 * np.pi, 200):
    # Get the results of the run with the value we want to execute with
    bitstrings = qc.run(executable, {'theta': [theta]})
```

(continues on next page)

(continued from previous page)

```
    # Store our results
    parametric_measurements.append(bitstrings)
```

In the example here, if you called `qc.run(executable)` and didn't specify `'theta'`, the program would apply `RZ(0, qubit)` for every execution.

Note: Classical memory defaults to zero. If you don't specify a value for a declared memory region, it will be zero.

1.6.6 Gate Modifiers

Gate applications in Quil can be preceded by a *gate modifier*. There are two supported modifiers: `DAGGER` and `CONTROLLED`. The `DAGGER` modifier represents the dagger of the gate. For instance,

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

would have an equivalent effect to `RX(-pi/3) 0`.

The `CONTROLLED` modifier takes a gate and makes it a controlled gate. For instance, one could write the Toffoli gate in any of the three following ways:

```
CCNOT 0 1 2
CONTROLLED CNOT 0 1 2
CONTROLLED CONTROLLED X 0 1 2
```

Note: The letter `C` in the gate name has no semantic significance in Quil. To make a controlled `Y` gate, one *cannot* write `CY`, but rather one has to write `CONTROLLED Y`.

All gates (objects deriving from the `Gate` class) provide the methods `Gate.controlled(control_qubit)` and `Gate.dagger()` that can be used to programmatically apply the `CONTROLLED` and `DAGGER` modifiers.

For example, to produce the controlled-NOT gate (`CNOT`) with control qubit `0` and target qubit `1`

```
prog = Program(X(1).controlled(0))
```

You can achieve the oft-used *control-off* gate (flip the target qubit `1` if the control qubit `0` is zero) with

```
prog = Program(X(0), X(1).controlled(0), X(0))
```

1.6.7 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],
```

(continues on next page)

(continued from previous page)

```
                    [ 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))`).

1.6.8 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([
```

(continues on next page)

(continued from previous page)

```
    [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))`).

1.6.9 Pragmas

`PRAGMA` directives give users more control over how Quil programs are processed or simulated but generally do not change the semantics of the Quil program itself. As a general rule of thumb, deleting all `PRAGMA` directives in a Quil program should leave a valid and semantically equivalent program.

In pyQuil, `PRAGMA` directives play many roles, such as controlling the behavior of gates in noisy simulations, or commanding the Quil compiler to perform actions in a certain way. Here, we will cover the basics of two very common use cases for including a `PRAGMA` in your program: qubit rewiring and delays. For a more comprehensive review of what pragmas are and what the compiler supports, check out *The Quil Compiler*. For more information about `PRAGMA` in Quil, see A Practical Quantum ISA, and Simulating Quantum Processor Errors.

Specifying A Qubit Rewiring Scheme

Qubit rewiring is one of the most powerful features of the Quil compiler. We are able to write Quil programs which are agnostic to the topology of the chip, and the compiler will intelligently relabel our qubits to give better performance.

When we intend to run a program on the QPU, sometimes we write programs which use specific qubits targeting a specific device topology, perhaps to achieve a high-performance program. Other times, we write programs that are agnostic to the underlying topology, thereby making the programs more portable. Qubit rewiring accommodates both use cases in an automatic way.

Consider the following program.

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

p = Program(X(3))
```

We've tested this on the QVM, and we've reserved a lattice on the QPU which has qubits 4, 5, and 6, but not qubit 3. Rather than rewrite our program for each reservation, we modify our program to tell the compiler to do this for us.

```
from pyquil.quil import Pragma

p = Program(Pragma('INITIAL_REWIRING', ['"GREEDY"']))
p += X(3)
```

Now, when we pass our program through the compiler (such as with `QuantumComputer.compile()`) we will get native Quil with the qubit reindexed to one of 4, 5, or 6. If qubit 3 is available, and we don't want that pulse to be applied to any other qubit, we would instead use `Pragma('INITIAL_REWIRING', ['"NAIVE"'])`. Detailed information about the available options is *here*.

Note: In general, we assume that the qubits you're supplying as input are also the ones which you prefer to operate on, and so NAIVE rewiring is the default.

Asking for a Delay

At times, we may want to add a delay in our program. Usually this is associated with qubit characterization. Delays are not regular gate operations, and they do not affect the abstract semantics of the Quil program, so they're implemented with a `PRAGMA` directive.

```
#  ...
# qubit index and time in seconds must be defined and provided
# the time argument accepts exponential notation e.g. 200e-9
p += Pragma('DELAY', [qubit], str(time))
```

Warning: These delays currently have effects on the real QPU. They have no effect on QVM's even when those QVM's have noise models applied.

Warning: Keep in mind, the program duration is currently capped at 15 seconds, and the length of the program is multiplied by the number of shots. If you have a 1000 shot program, where each shot contains a 100ms delay, you won't be able to execute it.

1.6.10 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
```

(continues on next page)

(continued from previous page)

```
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
```

1.6.11 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 `DEFGATE`s with parameters read from classical memory.

1.7 The Quantum Computer

PyQuil is used to build Quil (Quantum Instruction Language) programs and execute them on simulated or real quantum devices. 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 (QAM) that has a shared classical/quantum architecture at its core.

A QAM must, therefore, implement certain abstract methods to manipulate classical and quantum states, such as loading programs, writing to shared classical memory, and executing programs.

The program execution itself is sent from pyQuil to quantum computer endpoints, which will be one of two options:

- A Rigetti Quantum Virtual Machine (QVM)
- A Rigetti Quantum Processing Unit (QPU)

Within pyQuil, there is a *QVM* object and a *QPU* object which use the exposed APIs of the QVM and QPU servers, respectively.

On this page, we'll learn a bit about the *QVM* and *QPU*. Then we will show you how to use them from pyQuil with a *The QuantumComputer*.

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

1.7.1 The Quantum Virtual Machine (QVM)

The Rigetti Quantum Virtual Machine is an implementation of the Quantum Abstract Machine from *A Practical Quantum Instruction Set Architecture*.[1] It is implemented in ANSI Common LISP and executes programs specified in Quil.

The QVM is a wavefunction simulation of unitary evolution with classical control flow and shared quantum classical memory.

The QVM is part of the Forest SDK, and it's available for you to use on your local machine. After *downloading and installing the SDK*, you can initialize a local QVM server by typing `qvm -S` into your terminal. You should see the following message.

```
$ qvm -S
*******************************
* Welcome to the Rigetti QVM *
*******************************
Copyright (c) 2018 Rigetti Computing.

(Configured with 2048 MiB of workspace and 8 workers.)

[2018-11-06 18:18:18] Starting server on port 5000.
```

By default, the server is started on port 5000 on your local machine. Consequently, the endpoint which the pyQuil *QVM* will default to for the QVM address is `http://127.0.0.1:5000`. When you run your program, a pyQuil client will send a Quil program to the QVM server and wait for a response back.

It's also possible to use the QVM from the command line. You can write a Quil program in its own file:

```
# example.quil

DECLARE ro BIT[1]
```

(continues on next page)

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

(continued from previous page)

```
RX(pi/2) 0
CZ 0 1
```

and then execute it with the QVM directly from the command line:

```
$ qvm -e < example.quil

[2018-11-30 11:13:58] Reading program.
[2018-11-30 11:13:58] Allocating memory for QVM of 2 qubits.
[2018-11-30 11:13:58] Allocation completed in 4 ms.
[2018-11-30 11:13:58] Loading quantum program.
[2018-11-30 11:13:58] Executing quantum program.
[2018-11-30 11:13:58] Execution completed in 6 ms.
[2018-11-30 11:13:58] Printing 2-qubit state.
[2018-11-30 11:13:58] Amplitudes:
[2018-11-30 11:13:58]   |00>: 0.0, P=  0.0%
[2018-11-30 11:13:58]   |01>: 0.0-1.0i, P=100.0%
[2018-11-30 11:13:58]   |10>: 0.0, P=  0.0%
[2018-11-30 11:13:58]   |11>: 0.0, P=  0.0%
[2018-11-30 11:13:58] Classical memory (low -> high indexes):
[2018-11-30 11:13:58]     ro:  1 0
```

For a detailed description of how to use the `qvm` from the command line, see the QVM README or type `man qvm` in your terminal.

We also offer a Wavefunction Simulator (formerly a part of the *QVM* object), which allows users to contruct and inspect wavefunctions of quantum programs. Learn more about the Wavefunction Simulator *here*.

1.7.2 The Quantum Processing Unit (QPU)

To access a QPU endpoint, you will have to sign up for Quantum Cloud Services (QCS). Documentation for getting started with your Quantum Machine Image (QMI) is found here. Using QCS, you will `ssh` into your QMI, and reserve a QPU lattice for a particular time block.

When your reservation begins, you will be authorized to access the QPU. A configuration file will be automatically populated for you with the proper QPU endpoint for your reservation. Both your QMI and the QPU are located on premises, giving you low latency access to the QPU server. That server accepts jobs in the form of a `BinaryExecutableRequest` object, which is precisely what you get back when you compile your program in pyQuil and target the QPU (more on this soon). This request contains all the information necessary to run your program on the control rack which sends and receives waveforms from the QPU, so that you can receive classical binary readout results.

For information on available lattices, you can check out your dashboard at https://qcs.rigetti.com/dashboard after you've been invited to QCS.

1.7.3 The `QuantumComputer`

The *`QuantumComputer`* abstraction offered by pyQuil provides an easy access point to the most critical objects used in pyQuil for building and executing your quantum programs. We will cover the main methods and attributes on this page. The QuantumComputer API Reference provides a reference for all of its methods and options.

At a high level, the *`QuantumComputer`* wraps around our favorite quantum computing tools:

- **A quantum abstract machine** `.qam` : this is our general purpose quantum computing device, which implements the required abstract methods described *above*. It is implemented as a *`QVM`* or *`QPU`* object in pyQuil.

- **A compiler** `.compiler` : this determines how we manipulate the Quil input to something more efficient when possible, and then into a form which our QAM can accept as input.
- **A device** `.device` : this specifies the topology and Instruction Set Architecture (ISA) of the targeted device by listing the supported 1Q and 2Q gates.

When you instantiate a *QuantumComputer* instance, these subcomponents will be compatible with each other. So, if you get a `QPU` implementation for the `.qam`, you will have a `QPUCompiler` for the `.compiler`, and your `.device` will match the device used by the `.compiler`.

The *QuantumComputer* instance makes methods available which are built on the above objects. If you need more fine grained controls for your work, you might try exploring what is offered by these objects.

For more information on each of the above, check out the following pages:

- Compiler API Reference
- *Quil Compiler docs*
- Device API Reference
- *Providing Your Own Device Topology*
- Quantum abstract machine (QAM) API Reference
- The Quil Whitepaper which describes the QAM

Instantiation

A decent amount of information needs to be provided to initialize the `compiler`, `device`, and `qam` attributes, much of which is already in your config files (or provided reasonable defaults when running locally). Typically, you will want a *QuantumComputer* which either:

- pertains to a real, available QPU device
- is a QVM but mimics the topology of a QPU
- is some generic QVM

All of this can be accomplished with `get_qc()`.

```
def get_qc(name: str, *, as_qvm: bool = None, noisy: bool = None,
           connection: ForestConnection = None) -> QuantumComputer:
```

```
from pyquil import get_qc

# Get a QPU
qc = get_qc(QPU_LATTICE_NAME)  # QPU_LATTICE_NAME is just a string naming the device

# Get a QVM with the same topology as the QPU lattice
qc = get_qc(QPU_LATTICE_NAME, as_qvm=True)
# or, equivalently
qc = get_qc(f"{QPU_LATTICE_NAME}-qvm")

# A fully connected QVM
number_of_qubits = 10
qc = get_qc(f"{number_of_qubits}q-qvm")
```

For now, you will have to join QCS to get `QPU_LATTICE_NAME` by running the `qcs lattices` command from your QMI. Access to the QPU is only possible from a QMI, during a booked reservation. If this sounds unfamiliar, check out our documentation for QCS and join the waitlist.

For more information about creating and adding your own noise models, check out *Noise and Quantum Computation*.

Note: When connecting to a QVM locally (such as with `get_qc(..., as_qvm=True)`) you'll have to set up the QVM in *server mode*.

Methods

Now that you have your `qc`, there's a lot you can do with it. Most users will want to use `compile`, `run` or `run_and_measure`, and `qubits` very regularly. The general flow of use would look like this:

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

qc = get_qc('9q-square-qvm')            # not general to any number of qubits, 9q-
↪square-qvm is special

qubits = qc.qubits()                    # this information comes from qc.device
p = Program()
# ... build program, potentially making use of the qubits list

compiled_program = qc.compile(p)        # this makes multiple calls to qc.compiler

results = qc.run(compiled_program)      # this makes multiple calls to qc.qam
```

Note: In addition to a running QVM server, you will need a running `quilc` server to compile your program. Setting up both of these is very easy, as explained *here*.

The `.run_and_measure(...)` method

This is the most high level way to run your program. With this method, you are **not** responsible for compiling your program before running it, nor do you have to specify any `MEASURE` instructions; all qubits will get measured.

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

qc = get_qc("8q-qvm")

p = Program(X(0))

results = qc.run_and_measure(p, trials=5)
print(results)
```

`trials` specifies how many times to run this program. Let's see our results:

```
{0: array([1, 1, 1, 1, 1]),
 1: array([0, 0, 0, 0, 0]),
 2: array([0, 0, 0, 0, 0]),
 3: array([0, 0, 0, 0, 0]),
 4: array([0, 0, 0, 0, 0]),
 5: array([0, 0, 0, 0, 0]),
 6: array([0, 0, 0, 0, 0]),
 7: array([0, 0, 0, 0, 0])}
```

The return value is a dictionary from qubit index to results for all trials. Every qubit in the lattice is measured for you, and as expected, qubit 0 has been flipped to the excited state for each trial.

The `.run(...)` method

The lower-level `.run(...)` method gives you more control over how you want to build and compile your program than `.run_and_measure` does. **You are responsible for compiling your program before running it.** The above program would be written in this way to execute with `run`:

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

qc = get_qc("8q-qvm")

p = Program()
ro = p.declare('ro', 'BIT', 1)
p += X(0)
p += MEASURE(0, ro[0])
p += MEASURE(1, ro[1])
p.wrap_in_numshots_loop(5)

executable = qc.compile(p)
bitstrings = qc.run(executable)  # .run takes in a compiled program, unlike .run_and_
↪measure
print(bitstrings)
```

By specifying `MEASURE` ourselves, we will only get the results that we are interested in. To be completely equivalent to the previous example, we would have to measure all eight qubits.

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

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

Let's unpack this. The *outer* list is an enumeration over the trials; the argument given to `wrap_in_numshots_loop` will match the length of `results`.

The *inner* list, on the other hand, is an enumeration over the results stored in the memory region named `ro`, which we use as our readout register. We see that the result of this program is that the memory region `ro[0]` now stores the state of qubit 0, which should be `1` after an X-gate. See *Declaring Memory* and *Measurement* for more details about declaring and accessing classical memory regions.

Tip: Get the results for qubit 0 with `numpy.array(bitstrings)[:,0]`.

Providing Your Own Device Topology

It is simple to provide your own device topology as long as you can give your qubits each a number, and specify which edges exist. Here is an example, using the topology of our 16Q chip (two octagons connected by a square):

```
import networkx as nx

from pyquil.device import NxDevice, gates_in_isa
from pyquil.noise import decoherence_noise_with_asymmetric_ro
```

(continues on next page)

(continued from previous page)

```
qubits = [0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17]  # qubits are␣
↪numbered by octagon
edges = [(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 0),  # first␣
↪octagon
         (1, 16), (2, 15),  # connections across the square
         (10, 11), (11, 12), (13, 14), (14, 15), (15, 16), (16, 17), (10, 17)] #␣
↪second octagon

# Build the NX graph
topo = nx.from_edgelist(edges)
# You would uncomment the next line if you have disconnected qubits
# topo.add_nodes_from(qubits)
device = NxDevice(topo)
device.noise_model = decoherence_noise_with_asymmetric_ro(gates_in_isa(device.get_
↪isa()))  # Optional
```

Now that you have your device, you could set `qc.device` and `qc.compiler.device` to point to your new device, or use it to make new objects.

Simulating the QPU using the QVM

The `QAM` methods are intended to be used in the same way, whether a QVM or QPU is being targeted. Everywhere on this page, you can swap out the type of the QAM (QVM <=> QPU) and you will still get reasonable results back. As long as the topology of the devices are the same, programs compiled and ran on the QVM will be able to run on the QPU and visa-versa. Since *`QuantumComputer`* is built on the `QAM` abstract class, its methods will also work for both QAM implementations.

This makes the QVM a powerful tool for testing quantum programs before executing them on the QPU.

```
qpu = get_qc(QPU_LATTICE_NAME)
qvm = get_qc(QPU_LATTICE_NAME, as_qvm=True)
```

By simply providing `as_qvm=True`, we get a QVM which will have the same topology as the named QPU. It's a good idea to run your programs against the QVM before booking QPU time to iron out bugs. To learn more about how to add noise models to your virtual `QuantumComputer` instance, check out *Noise and Quantum Computation*.

In the next section, we will see how to use the Wavefunction Simulator aspect of the Rigetti QVM to inspect the full wavefunction set up by a Quil program.

1.8 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
```

(continues on next page)

(continued from previous page)

```
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.

1.8.1 Multi-Qubit Basis Enumeration

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

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

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

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

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

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

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

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

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

1.9 The Quil Compiler

1.9.1 Expectations for Program Contents

The QPUs have much more limited natural gate sets than the standard gate set offered by pyQuil: on Rigetti QPUs, the gate operators are constrained to lie in `RZ(`θ`)`, `RX(k*`π`/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). However, as a programmer, it is often (though not always) desirable to to be able to write programs which don't take these details into account. These generally leads to more portable code if one isn't tied to a specific set of gates or QPU architecture. To ameliorate these limitations, the Rigetti software toolkit contains an optimizing compiler that translates arbitrary Quil to native Quil and native Quil to executables suitable for Rigetti hardware.

1.9.2 Interacting with the Compiler

After *downloading the SDK*, the Quil Compiler, `quilc` is available on your local machine. You can initialize a local `quilc` server by typing `quilc -S` into your terminal. You should see the following message.

```
$ quilc -S
+-----------------+
|  W E L C O M E  |
|   T O   T H E   |
|  R I G E T T I  |
|     Q U I L     |
| C O M P I L E R |
+-----------------+
Copyright (c) 2018 Rigetti Computing.

... - Launching quilc.
... - Spawning server at (tcp://*:5555) .
```

To get a description of `quilc`, and options and examples of its command line use, see quilc_man.

A `QuantumComputer` object supplied by the function `pyquil.api.get_qc()` comes equipped with a connection to your local Rigetti Quil compiler. 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 "#(0 1 2 3 4 5 6 7)"
RZ(pi/2) 0
RX(pi/2) 0
RZ(-pi/2) 1
RX(pi/2) 1
CZ 1 0
RX(-pi/2) 1
RZ(-pi/2) 2
RX(pi/2) 2
CZ 2 1
RZ(-pi/2) 0
RZ(-pi/2) 1
RX(-pi/2) 2
RZ(pi/2) 2
PRAGMA CURRENT_REWIRING "#(0 1 2 3 4 5 6 7)"
```

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`), 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 native Quil 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)
```

(continues on next page)

(continued from previous page)

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

1.9.3 Legal compiler input

The QPU is not able to execute all possible Quil programs. At present, a Quil program qualifies for execution if 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(`θ`)` for any value θ, `RX(k*`π`/2)` for an integer `k`, or `CZ q0 q1` for `q0`, `q1` a pair of qubits participating in a qubit-qubit interaction.
- This is then followed by a block of `MEASURE` instructions.

1.9.4 Region-specific compiler features through PRAGMA

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

Note: The interface to the Quil compiler from pyQuil 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.

Warning: If a preserved block is not legal QPU input, then it is not guaranteed to execute or it may produced unexpected results.

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
H 0
CNOT 0 1

#   wait a while
PRAGMA PRESERVE_BLOCK
I 0
```

(continues on next page)

(continued from previous page)

```
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`.

Warning: Lying to the compiler about what blocks can commute can cause incorrect results.

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
CNOT 2 3
```

(continues on next page)

(continued from previous page)

```
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 `nj` on the device. This is strictly for human-readability: user-supplied instructions of the form `PRAGMA [EXPECTED|CURRENT]_REWIRING` are discarded and have no effect.

In addition, you have some control over how the compiler constructs its rewiring, which is controlled by `PRAGMA INITIAL_REWIRING`. The syntax is as follows.

```
# <type> can be NAIVE, RANDOM, PARTIAL, or GREEDY
#
# The double quotes are required.
PRAGMA INITIAL_REWIRING "<type>"
```

Including this *before any non-pragmas* will allow the compiler to alter its rewiring behavior. The possible options are:

- `NAIVE` (default): 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.
- `PARTIAL`: 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.
- `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.

Note: `NAIVE` rewiring is the default, and for the most part, it follows the "Do What I Mean" (DWIM) principle. It is the least sophisticated, but attempts to follow what the user has constructed with their program. Choosing another

rewiring, such as `PARTIAL`, may lead to higher-performing programs because the compiler has more freedom to optimize the layout of the gates on the qubits.

1.9.5 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. In some rare circumstances, it can also happen due to floating point precision issues.

1.10 Noise and Quantum Computation

1.10.1 Modeling Noisy Quantum Gates

Pure States vs. Mixed States

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

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

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

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

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

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

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

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

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

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

Quantum Gate Errors

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

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

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

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

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

Why Do Incoherent Errors Happen?

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

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

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

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

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

the final state will generally not admit such a factorization.

A Toy Model

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

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

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

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

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

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

$$\begin{aligned}\rho'_S := \rho'_E &= \rho_{S,0}\underbrace{\rho_{E,0}}_{1} - \frac{i\sqrt{\epsilon}t}{\hbar}\underbrace{\left[V_S\rho_{S,0} \quad \underbrace{V_E\rho_{E,0}}_{\psi_{E,0}V_E\psi_{E,0}=0} \quad -\rho_{S,0}V_S \quad \underbrace{\rho_{E,0}V_E}_{\psi_{E,0}V_E\psi_{E,0}=0}\right]}_{0} \\ &\quad - \frac{\epsilon t^2}{2\hbar^2}\left[V_S^2\rho_{S,0}V_E^2\rho_{E,0} + \rho_{S,0}V_S^2\rho_{E,0}V_E^2 - 2V_S\rho_{S,0}V_SV_E\rho_{E,0}V_E\right] \\ &= \rho_{S,0} - \frac{\gamma}{2}\left[V_S^2\rho_{S,0} + \rho_{S,0}V_S^2 - 2V_S\rho_{S,0}V_S\right]\end{aligned} \tag{1.1}$$

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 $K_1 := I - \frac{\gamma}{2}V_S^2, K_2 := \sqrt{\gamma}V_S$:

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

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.

1.10.2 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 ψ_S

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

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

$$\begin{aligned}\overline{\rho'_S} &= \frac{1}{N}\sum_{n=1}^{N}\psi'_{nS}\psi'_{nS} \\ &= \frac{1}{N}\sum_{n=1}^{N}p_{j_n}^{-1}K_{j_n}\psi'_S\psi'_S K_{j_n}^\dagger\end{aligned} \tag{1.5}$$

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

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

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

$$\overline{\rho'_S} \approx \sum_{\ell=1}^{n} K_\ell \psi'_S \psi'_S K_\ell^\dagger \tag{1.8}$$

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!

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

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

```
# We could ask for "2q-noisy-qvm" but we will be specifying
# our noise model as PRAGMAs on the Program itself.
qc = get_qc('2q-qvm')
```

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
```

(continues on next page)

(continued from previous page)

```
    noise channel.

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

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

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

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


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

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

```
%%time

# single step damping probability
damping_per_I = 0.02

# number of program executions
trials = 200

results_damping = []
lengths = np.arange(0, 201, 10, dtype=int)
for jj, num_I in enumerate(lengths):
    print("\r{}/{}, ".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))
```

(continues on next page)

(continued from previous page)

```
    p.wrap_in_numshots_loop(trials)
    qc.qam.random_seed = int(num_I)
    res = qc.run(p)
    results_damping.append([np.mean(res), np.std(res) / np.sqrt(trials)])

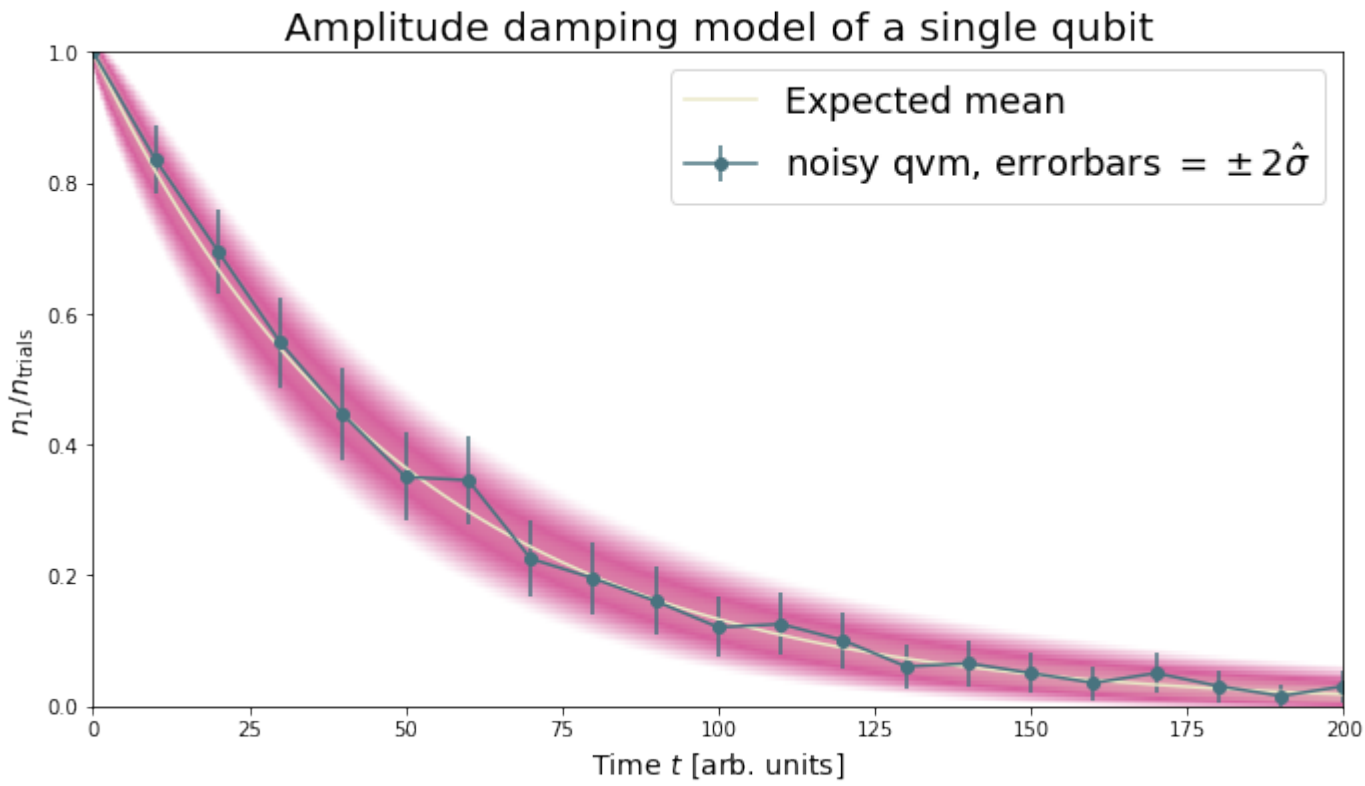
results_damping = np.array(results_damping)
```

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

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

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

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



Example 2: Dephased CZ-gate

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

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

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

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

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

The Kraus operators for this are given by

$$K_1'(p,q) = K_1(p) \otimes K_1(q) \tag{1.9}$$
$$K_2'(p,q) = K_2(p) \otimes K_1(q) \tag{1.10}$$
$$K_3'(p,q) = K_1(p) \otimes K_2(q) \tag{1.11}$$
$$K_4'(p,q) = K_2(p) \otimes K_2(q) \tag{1.12}$$

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

$$K_1^{\mathrm{CZ}}(p,q) = K_1(p) \otimes K_1(q) U_{\mathrm{CZ}} \tag{1.13}$$
$$K_2^{\mathrm{CZ}}(p,q) = K_2(p) \otimes K_1(q) U_{\mathrm{CZ}} \tag{1.14}$$
$$K_3^{\mathrm{CZ}}(p,q) = K_1(p) \otimes K_2(q) U_{\mathrm{CZ}} \tag{1.15}$$
$$K_4^{\mathrm{CZ}}(p,q) = K_2(p) \otimes K_2(q) U_{\mathrm{CZ}} \tag{1.16}$$

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

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

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

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

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

```
%%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("\r{}/{}, ".format(jj, len(ps)), end="")

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

(continues on next page)

(continued from previous page)

```
    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)
    p.wrap_in_numshots_loop(trials)
    qc.qam.random_seed = jj
    res = qc.run(p)
    results.append(res)

results = np.array(results)
```

```
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)
```

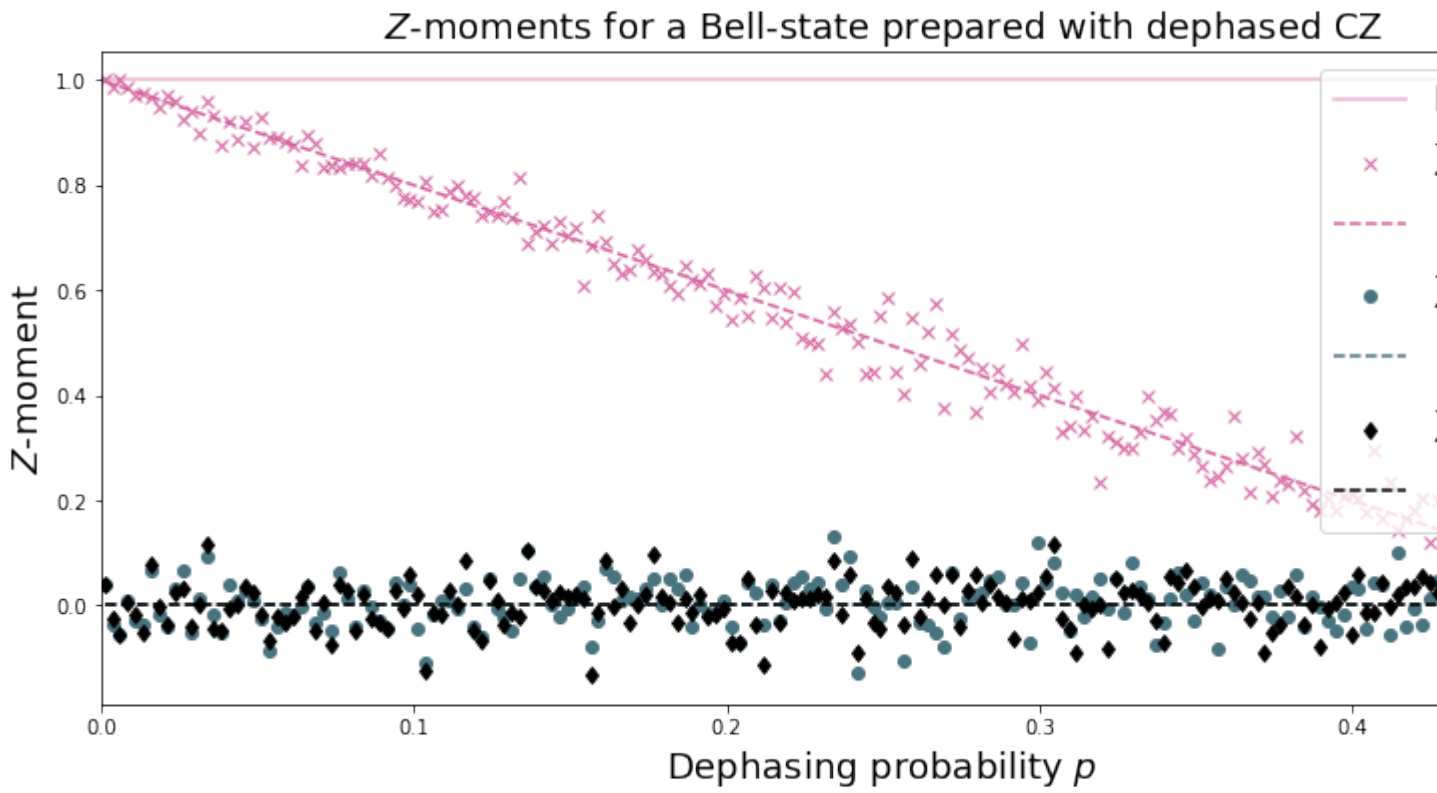
```
plt.figure(figsize=(14, 6))
plt.axhline(y=1.0, color=FUSCHIA, alpha=.5, label="Bell state")

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)
```



1.10.3 Adding Decoherence Noise

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

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

The Task

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

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

The Idiomatic PyQuil Program

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

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

The Compiled Program

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

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

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

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

Scan Over Noise Parameters

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

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

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

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

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

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

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

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

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

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

Plot the Results

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

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

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

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

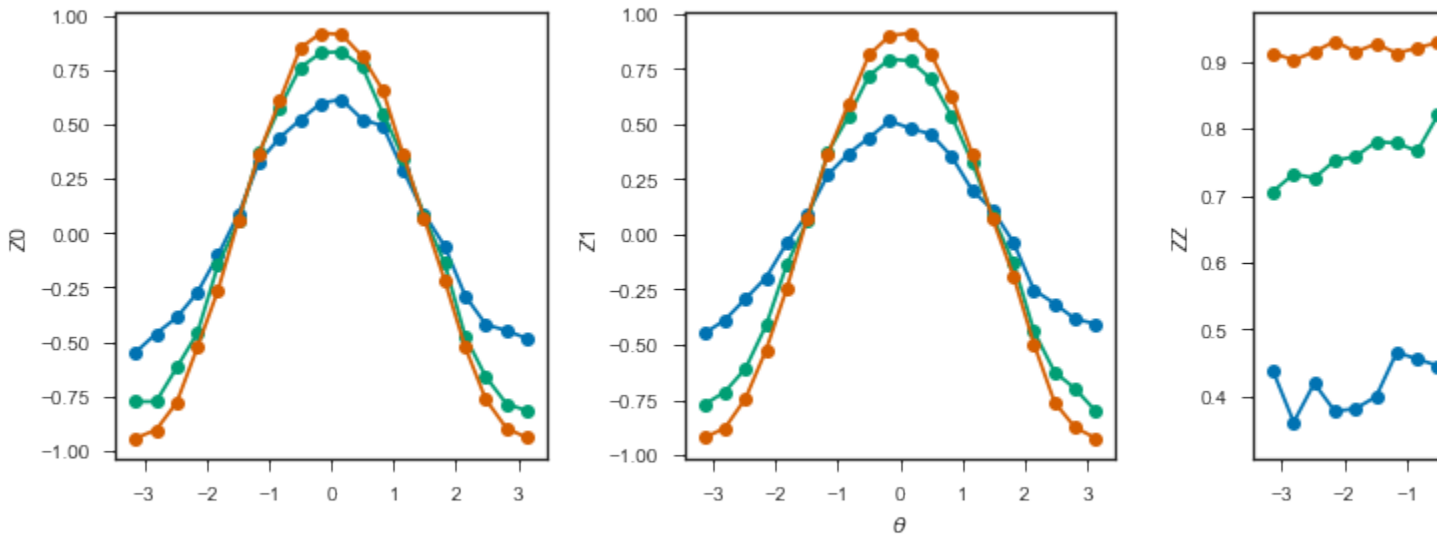
    ax1.plot(df['theta'], df['z0'], 'o-')
    ax2.plot(df['theta'], df['z1'], 'o-')
    ax3.plot(df['theta'], df['zz'], 'o-', label='T1 = {:.0f} us'.format(t1*1e6))
```

(continues on next page)

(continued from previous page)

```
ax3.legend(loc='best')

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



1.10.4 Modeling Readout Noise

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

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

Qubit Measurements

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

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

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

Projective, Ideal Measurement

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

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

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

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

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

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

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

which is the projection postulate applied to mixed states.

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

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

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

Classical Readout Bit-Flip Error

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

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

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

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

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

where

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

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

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

that must sum to 1:

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

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

How to Model T_1 Error

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

Further Reading

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

1.10.5 Working with Readout Noise

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

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

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

Scroll down for some examples!

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

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

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

cxn = QVMConnection()
```

Example 1: Rabi Sequence with Noisy Readout

```
%%time

# number of angles
num_theta = 101

# number of program executions
trials = 200

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

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

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

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

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

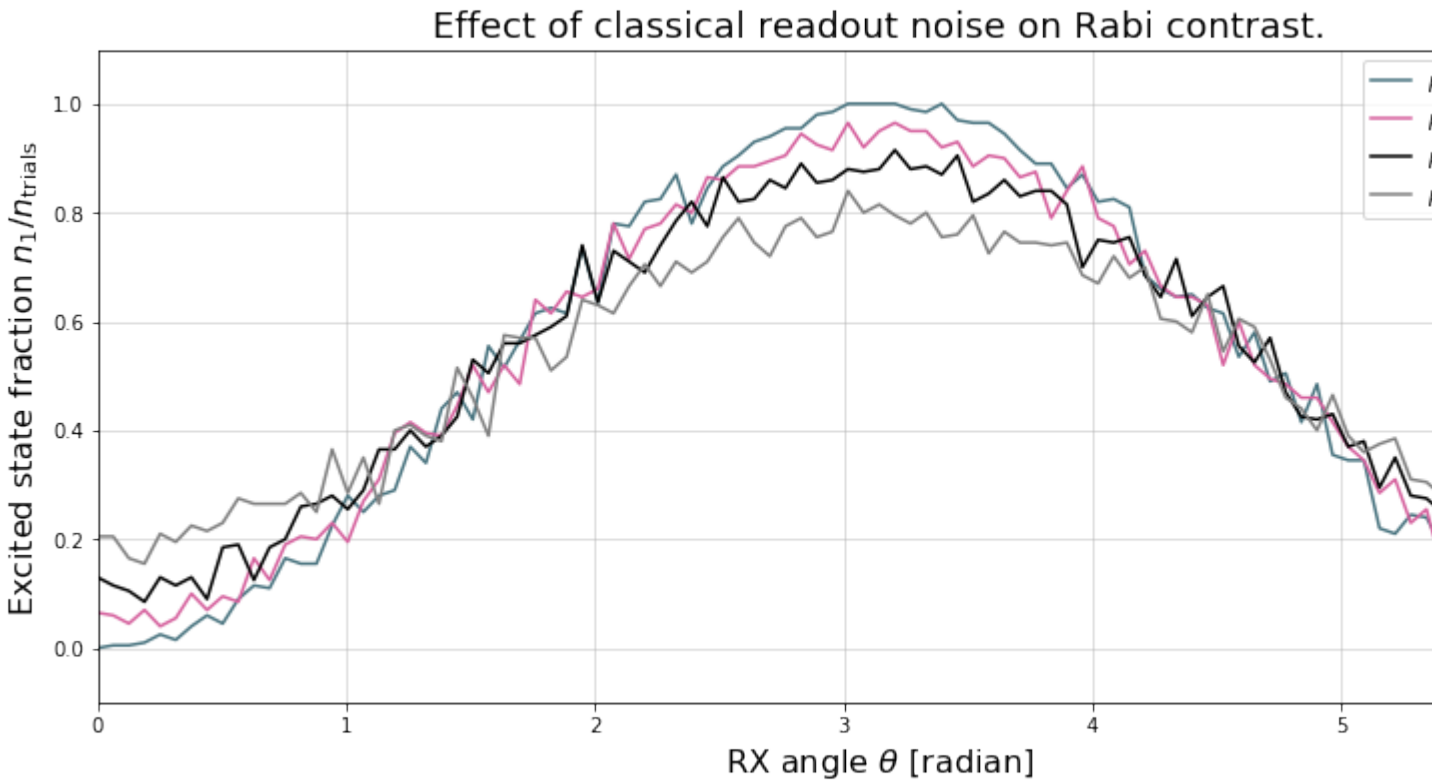
```
plt.figure(figsize=(14, 6))
for jj, (p00, c) in enumerate(zip(p00s, [DARK_TEAL, FUSCHIA, "k", "gray"])):
    plt.plot(thetas, results_rabi[:, jj]/trials, c=c, label=r"$p(0|0)=p(1|1)={:g}$".
↪format(p00))
plt.legend(loc="best")
plt.xlim(*thetas[[0,-1]])
plt.ylim(-.1, 1.1)
plt.grid(alpha=.5)
plt.xlabel(r"RX angle $\theta$ [radian]", size=16)
```

(continues on next page)

(continued from previous page)

```
plt.ylabel(r"Excited state fraction $n_1/n_{\rm trials}$", size=16)
plt.title("Effect of classical readout noise on Rabi contrast.", size=18)
```

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



Example 2: Estimate the Assignment Probabilities

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

On a perfect quantum computer

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

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

On an imperfect quantum computer

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

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

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

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

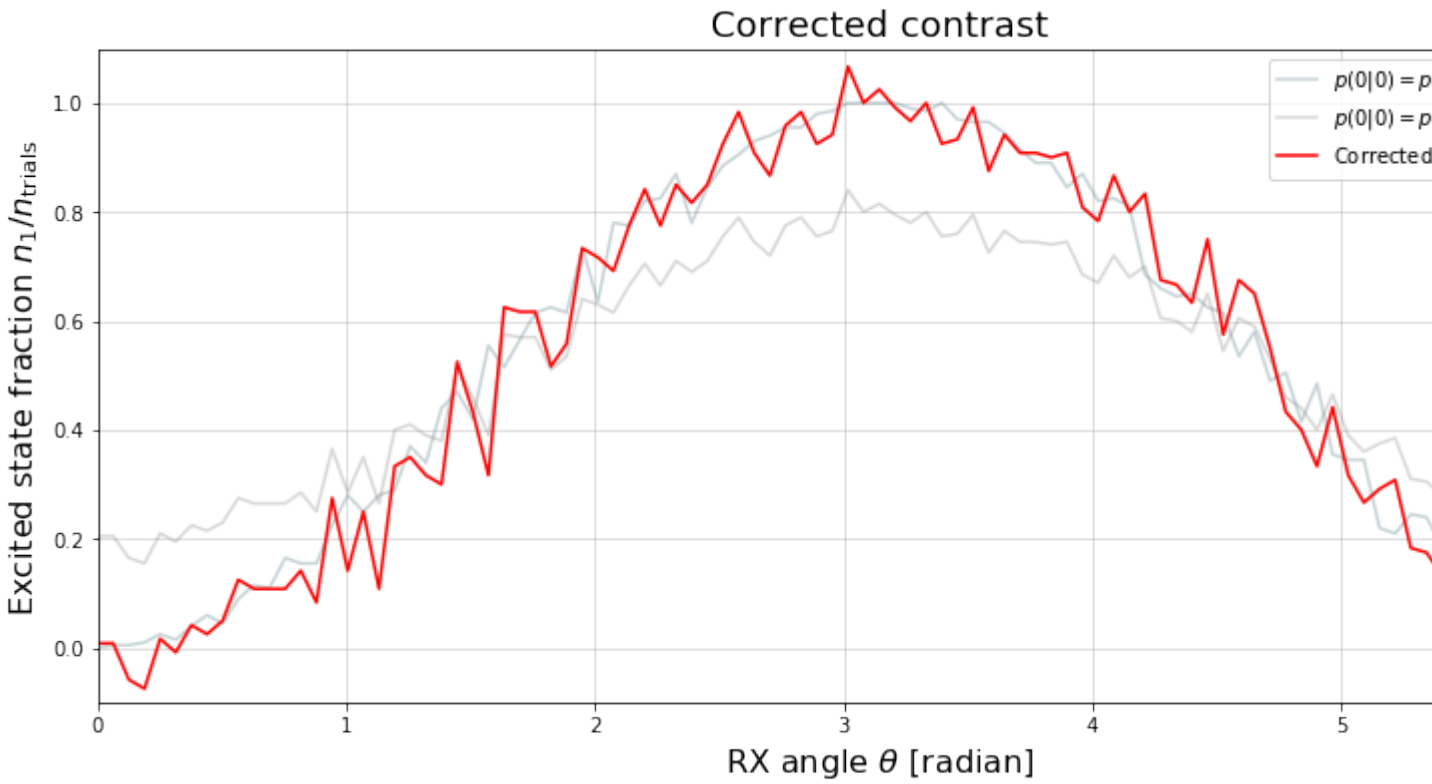
Example 3: Correct for Noisy Readout

3a) Correcting the Rabi Signal from Above

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

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

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



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

3b) Corrupting and Correcting GHZ State Correlations

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

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

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

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

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

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

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

```
(0.50419999999999998, 0.49580000000000002)
```

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

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

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

```
(0.30869999999999997, 0.3644)
```

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

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

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

```
(0.50397601453064977, 0.49866843912900716)
```

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

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

We expect these to all be very small

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

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

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

We expect these to all be close to 1.

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

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

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

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

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

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

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

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

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

Alternative: A global Pauli error model

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]]
```

1.11 Advanced Usage

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*.

1.11.1 PyQuil Configuration Files

Network endpoints for the Rigetti Forest infrastructure and information pertaining to QPU access are stored in a pair of configuration files. These files are located by default at `~/.qcs_config` and `~/.forest_config`. The location can be changed by setting the environment variables `QCS_CONFIG` or `FOREST_CONFIG` to point to the new location.

When running on a QMI, the values in these configuration files are automatically managed so as to point to the correct endpoints. When running locally, configuration files are not necessary. Thus, the average user will not have to do any work to get their configuration files set up.

If for some reason you want to use an atypical configuration, you may need to modify these files.

The default QCS config file on any QMI looks similar to the following:

```
# .qcs_config
[Rigetti Forest]
url = https://forest-server.qcs.rigetti.com
key = 4fd12391-11eb-52ec-35c2-262765ae4c4f
user_id = 4fd12391-11eb-52ec-35c2-262765ae4c4f

[QPU]
exec_on_engage = bash exec_on_engage.sh
```

where

- `url` is the endpoint that pyQuil hits for device information and for the 2.0 endpoints,
- `key` stores the Forest 1.X API key,
- `user_id` stores a Forest 2.0 user ID, and
- `exec_on_engage` specifies the shell command that the QMI will launch when the QMI becomes QPU-engaged. It would have no effect if you are running locally, but is important if you are running on the QMI. By default, it runs the `exec_on_engage.sh` shell script. It's best to leave the configuration as is, and edit that script. More documentation about `exec_on_engage.sh` can be found in the QCS docs here.

The Forest config file on any QMI has these contents, with specific IP addresses filled in:

```
# .forest_config
[Rigetti Forest]
qpu_endpoint_address = None
qvm_address = http://10.1.165.XX:5000
compiler_server_address = tcp://10.1.165.XX:5555
```

where

- `qpu_endpoint_address` is the endpoint where pyQuil will try to communicate with the QPU orchestrating service during QPU-engagement. It may not appear until your QMI engages, and furthermore will have no effect if you are running locally. It's best to leave this alone. If you obtain access to one of our QPUs, we will fill it in for you.
- `qvm_address` is the endpoint where pyQuil will try to communicate with the Rigetti Quantum Virtual Machine. On a QMI, this points to the provided QVM instance. On a local installation, this should be set to the server endpoint for a locally running QVM instance. However, pyQuil will use the default value `http://localhost:5000` if this file isn't found, which is the correct endpoint when you run the QVM locally with `qvm -S`.
- `compiler_server_address`: This is the endpoint where pyQuil will try to communicate with the compiler server. On a QMI, this points to a provided compiler server instance. On a local installation, this should be set to the server endpoint for a locally running `quilc` instance. However, pyQuil will use the default value `tcp://localhost:5555` if this isn't set, which is the correct endpoint when you run `quilc` locally with `quilc -S`.

Note: PyQuil itself reads these values out using the helper class `pyquil._config.PyquilConfig`. PyQuil users should not ever need to touch this class directly.

1.11.2 Using Qubit Placeholders

Note: The functionality provided inline by `QubitPlaceholders` is similar to writing a function which returns a `Program`, with qubit indices taken as arguments to the function.

In pyQuil, we typically use integers to identify qubits

```
from pyquil 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()
p = Program(H(q0), CNOT(q0, q1))
print(p)
```

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

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

```
print(p.out())
```

```
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(p))
```

```
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)
p_evens = Program(H(q) for q in qbyte)
print(address_qubits(p_evens, {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
```

1.11.3 Classical Control Flow

Note: Classical control flow is not yet supported on the QPU.

Here are a couple quick examples that show how much richer a Quil program can be with classical control flow. In this first example, we create a while loop by following these steps:

1. Declare a register called `flag_register` to use as a boolean test for looping.
2. Initialize this register to `1` program so our while loop will execute. This is often called the *loop preamble* or *loop initialization*.
3. Write the body of the loop in its own *Program*. This will be a program that applies an X gate followed by a H gate on our qubit.

4. Using the *while_do()* method to add control flow.

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

# Initialize the Program and declare a 1 bit memory space for our boolean flag
outer_loop = Program()
flag_register = outer_loop.declare('flag_register', 'BIT')

# Set the initial flag value to 1
outer_loop += MOVE(flag_register, 1)

# Define the body of the loop with a new Program
inner_loop = Program()
inner_loop += Program(X(0), H(0))
inner_loop += MEASURE(0, flag_register)

# Run inner_loop in a loop until flag_register is 0
outer_loop.while_do(flag_register, inner_loop)

print(outer_loop)
```

```
DECLARE flag_register BIT[1]
MOVE flag_register 1
LABEL @START1
JUMP-UNLESS @END2 flag_register
X 0
H 0
MEASURE 0 flag_register
JUMP @START1
LABEL @END2
```

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

- `NOT` which flips a classical bit
- `AND` which operates on two classical bits
- `IOR` 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.

```
# Declare our memory spaces
branching_prog = Program()
test_register = branching_prog.declare('test_register', 'BIT')
ro = branching_prog.declare('ro', 'BIT')

# 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()
```

(continues on next page)

(continued from previous page)

```
# Construct our program so that the result in test_register is equally likely to be a␣
↪0 or 1
branching_prog += H(1)
branching_prog += MEASURE(1, test_register)

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

# Measure qubit 0 into our readout register
branching_prog += MEASURE(0, ro)

print(branching_prog)
```

```
DECLARE test_register BIT[1]
DECLARE ro BIT[1]
H 1
MEASURE 1 test_register
JUMP-WHEN @THEN1 test_register
JUMP @END2
LABEL @THEN1
X 0
LABEL @END2
MEASURE 0 ro
```

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

```
from pyquil import get_qc

qc = get_qc("2q-qvm")
branching_prog.wrap_in_numshots_loop(10)
qc.run(branching_prog)
```

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

1.11.4 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(f"sigma = {sigma}")
```

```
sigma = (0.5+0j)*I + (-0.75+0j)*X0*Y1*Z3 + (5-2j)*Z1*X2
```

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

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

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

The following shows an instructive example of all three.

```
from pyquil.paulis import exponential_map

sigma_cubed = sigma * sigma * sigma
print(f"Simplified: {sigma_cubed}\n")

# Produce Quil code to compute exp[iX]
H = -1.0 * sX(0)
print(f"Quil to compute exp[iX] on qubit 0:\n"
       f"{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 α.

```
expH = exponential_map(H)
print(f"0:\n{expH(0.0)}\n")
print(f"1:\n{expH(1.0)}\n")
print(f"2:\n{expH(2.0)}")
```

```
0:
H 0
RZ(0) 0
H 0

1:
H 0
RZ(-2.0) 0
H 0

2:
H 0
RZ(-4.0) 0
H 0
```

To take it one step further, you can use *Parametric Compilation* with `exponential_map`. For instance:

```
ham = sZ(0) * sZ(1)
prog = Program()
theta = prog.declare('theta', 'REAL')
prog += exponential_map(ham)(theta)
```

1.12 Exercises

1.12.1 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
```

1.12.2 Exercise 2: Controlled Gates

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

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

1.12.3 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:

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

1.12.4 Exercise 4: 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?

1.12.5 Exercise 5: Quantum Fourier Transform

The quantum Fourier transform (QFT) is a quantum implementation of the discrete Fourier transform. The Fourier transform can be used to transform a function from the time domain into the frequency domain.

Compute the discrete Fourier transform of `[0, 1, 0, 0, 0, 0, 0, 0]`, using pyQuil:

a. Write a state preparation quantum program.

b. Write a function to make a 3-qubit QFT program, taking qubit indices as arguments.

c. Combine your solutions to part a and b into one program and use the `WavefunctionSimulator` to get the solution.

Note: For a more challenging initial state, try `01100100`.

Solution

Part a: Prepare the initial state

We are going to apply the QFT on the *amplitudes* of the states.

We want to prepare a state that corresponds to the sequence for which we want to compute the discrete Fourier transform. As the exercise hinted in part b, we need 3 qubits to transform an 8 bit sequence. It is simplest to understand if we think of the qubits as three digits in a binary string (aka bitstring). There are 8 possible values the bitstring can have, and in our quantum state, each of these possibilities has an amplitude. Our 8 indices in the QFT sequence label each of these states. For clarity:

$|000\rangle$ => `10000000`

$|001\rangle$ => `01000000`

...

$|111\rangle$ -> `00000001`

The sequence we want to compute is `01000000`, so our initial state is simply $|001\rangle$. For a bitstring with more than one `1`, we would want an equal superposition over all the selected states. (E.g. `01100000` would be an equal superposition of $|001\rangle$ and $|010\rangle$).

To set up the $|001\rangle$ state, we only have to apply one X-gate to the zeroth qubit.

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

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

We can verify that this works by computing its wavefunction with the *Wavefunction Simulator*. However, we need to add some "dummy" qubits, because otherwise `wavefunction` would return a two-element vector for only qubit 0.

```
from pyquil.api import WavefunctionSimulator

add_dummy_qubits = Program(I(1), I(2))  # The identity gate I has no affect

wf_sim = WavefunctionSimulator()
wavefunction = wf_sim.wavefunction(state_prep + add_dummy_qubits)
print(wavefunction)
```

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

We'll need `wf_sim` for part c, too.

Part b: Three qubit QFT program

In this part, we define a function, `qft3`, to make a 3-qubit QFT quantum program. The algorithm is nicely described on this page. It is a mix of Hadamard and CPHASE gates, with a SWAP gate for bit reversal correction.

```
from math import pi

def qft3(q0, q1, q2):
    p = Program()
    p += [SWAP(q0, q2),
          H(q0),
          CPHASE(-pi / 2.0, q0, q1),
          H(q1),
          CPHASE(-pi / 4.0, q0, q2),
          CPHASE(-pi / 2.0, q1, q2),
          H(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 with `print(qft(0, 1, 2))`.

```
SWAP 0 2
H 0
CPHASE(-pi/2) 0 1
H 1
CPHASE(-pi/4) 0 2
CPHASE(-pi/2) 1 2
H 2
```

Part c: Execute the QFT

Combining parts a and b:

```
compute_qft_prog = state_prep + qft3(0, 1, 2)
wavefunction = wf_sim.wavefunction(compute_qft_prog)
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 on the output with NumPy and seeing that we get back our input (with some floating point error).

```
from numpy.fft import ifft
ifft(wavefunction.amplitudes, norm="ortho")
```

```
array([0.+0.00000000e+00j, 1.+9.38127079e-17j, 0.+0.00000000e+00j,
       0.-1.53080850e-17j, 0.+0.00000000e+00j, 0.-6.31965379e-17j,
       0.+0.00000000e+00j, 0.-1.53080850e-17j])
```

After ignoring the terms that are on the order of `1e-17`, we get `[0, 1, 0, 0, 0, 0, 0, 0]`, which was our input!

1.12.6 Example: The Meyer-Penny Game

To create intuition for quantum algorithms, it is useful (and fun) to play with the abstraction that the software provides.

The Meyer-Penny Game[1] is a simple example we'll use from quantum game theory. The interested reader may want to read more about quantum game theory in the article *Toward a general theory of quantum games*[2]. The Meyer-Penny Game goes as follows:

The Starship Enterprise, during one of its deep-space missions, is facing an immediate calamity at the edge of a wormhole, when a powerful alien suddenly appears. 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 heads or tails.

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 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?

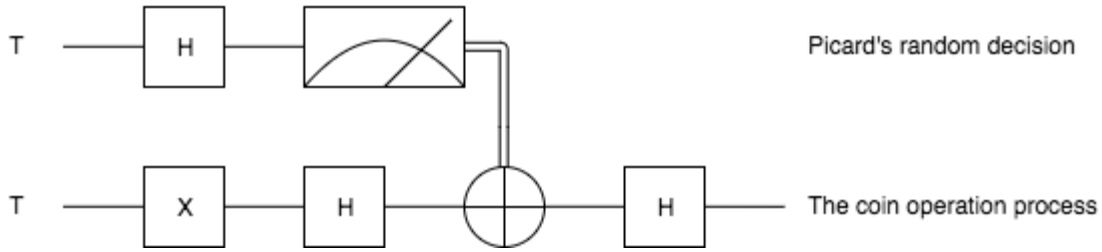
What Picard did not anticipate is that Q has access to quantum tools. Instead of flipping the penny, Q puts the penny into a superposition of heads and tails proportional to the quantum state $|H\rangle + |T\rangle$. Then no matter whether Picard flips the penny or not, it will stay in a superposition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

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 $|0\rangle$ (which is mapped to $|T\rangle$, tails). 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 $p_h = p_t = 0.5$.

To simulate the penny flip game we take the second qubit and put it into its excited state $|1\rangle$ (which is mapped to $|H\rangle$, heads) 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.

[1] https://link.aps.org/doi/10.1103/PhysRevLett.82.1052
[2] https://arxiv.org/abs/quant-ph/0611234



In pyQuil

We first import and initialize the necessary tools[3]

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

wf_sim = WavefunctionSimulator()
p = Program()
```

and then wire it all up into the overall measurement circuit; remember that qubit 0 is the penny, and qubit 1 represents Picard's choice.

```
p += X(0)
p += H(0)
p += H(1)
p += CNOT(1, 0)
p += H(0)
```

We use the quantum mechanics principle of deferred measurement to keep all the measurement logic separate from the gates. Our method call to the `WavefunctionSimulator` will handle measuring for us[4].

Finally, we play the game several times. (Remember to run your *qvm server*.)

```
wf_sim.run_and_measure(p, trials=10)
```

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

In each trial, the first number is the outcome of the game, whereas the second number represents Picard's choice to flip or not flip the penny.

Inspecting the results, we see that no matter what Picard does, Q will always win!

[3] See more: *Programs and Gates*
[4] More about measurements and `run_and_measure`: *Measurement*

1.13 Changelog

1.13.1 v2.6 (March 29, 2019)

Improvements and Changes:

- Added a CODEOWNERS file for default reviewers (gh-855).
- Bifurcated the `QPUCompiler` endpoint parameter into two – `quilc_endpoint` and `qpu_compiler_endpoint` – to reflect changes in Quantum Cloud Services (gh-856).
- Clarified documentation around the DELAY pragma (gh-862).
- Added information about the `local_qvm` context manager to the getting started documentation (gh-851).

Bugfixes:

- Added a non-None default timeout to the `QVMCompiler` object (gh-850) and the `get_benchmarker` function (gh-854).
- Fixed the docstring for the `apply_clifford_to_pauli` function (gh-836).
- Allowed the `apply_clifford_to_pauli` function to now work with the Identity as input (gh-849).
- Updated a stale link to the Rigetti Forest Slack workspace (gh-860).
- Fixed a notation typo in the documentation for noise (gh-861).

Special thanks to willzeng for all the contributions this release!

1.13.2 v2.5 (March 6, 2019)

Improvements and Changes:

- PyQuil's Gate objects now expose `.controlled(q)` and `.dagger()` modifiers, which turn a gate respectively into its controlled variant, conditional on the qubit `q`, or into its inverse.
- The operator estimation suite's `measure_observables` method now exposes a `readout_symmetrize` argument, which helps mitigate a machine's fidelity asymmetry between recognizing a qubit in the ground state versus the excited state.
- The `MEASURE` instruction in pyQuil now has a *mandatory* second argument. Previously, the second argument could be omitted to induce "measurement for effect", without storing the readout result to a classical register, but users found this to be a common source of accidental error and a generally rude surprise. To ensure the user really intends to measure only for effect, we now require that they supply an explicit `None` as the second argument.

Bugfixes:

- Some stale tests have been brought into the modern era.

1.13.3 v2.4 (February 14, 2019)

Announcements:

- The Quil Compiler (quilc) and the Quantum Virtual Machine (QVM), which are part of the Forest SDK, have been open sourced! In addition to downloading the binaries, you can now build these applications locally from source, or run them via the Docker images rigetti/quilc and rigetti/qvm. These Docker images are now used as the `services` in the GitLab CI build plan YAML (gh-792, gh-794, gh-795).

Improvements and Changes:

- The `WavefunctionSimulator` now supports the use of parametric Quil programs, via the `memory_map` parameter for its various methods (gh-787).
- Operator estimation data structures introduced in **v2.2** have changed. Previously, `ExperimentSettings` had two members: `in_operator` and `out_operator`. The `out_operator` is unchanged, but `in_operator` has been renamed to `in_state` and its data type is now `TensorProductState` instead of `PauliTerm`. It was always an abuse of notation to interpret pauli operators as defining initial states. Analogous to the Pauli helper functions sI, sX, sY, and sZ, `TensorProductState` objects are constructed by multiplying together terms generated by the helper functions plusX, minusX, plusY, minusY, plusZ, and minusZ. This functionality enables process tomography and process DFE (gh-770).
- Operator estimation now offers a "greedy" method for grouping tomography-like experiments that share a natural tensor product basis (ntpb), as an alternative to the clique cover version (gh-754).
- The `quilc` endpoint for rewriting Quil parameter arithmetic has been changed from `resolve_gate_parameter_arithmetic` to `rewrite_arithmetic` (gh-802).
- The difference between ProtoQuil and QPU-supported Quil is now better defined (gh-798).

Bugfixes:

- Resolved an issue with post-gate noise in the pyQVM (gh-801).
- A `TypeError` with a useful error message is now raised when a `Program` object is run on a QPU-backed `QuantumComputer`, rather than a confusing `AttributeError` (gh-799).

1.13.4 v2.3 (January 28, 2019)

PyQuil 2.3 is the latest release of pyQuil, Rigetti's toolkit for constructing and running quantum programs. A major new feature is the release of a new suite of simulators:

- We're proud to introduce the first iteration of a Python-based quantum virtual machine (QVM) called PyQVM. This QVM is completely contained within pyQuil and does not need any external dependencies. Try using it with `get_qc("9q-square-pyqvm")` or explore the `pyquil.pyqvm.PyQVM` object directly. Under-the-hood, there are three quantum simulator backends:
  - `ReferenceWavefunctionSimulator` uses standard matrix-vector multiplication to evolve a statevector. This includes a suite of tools in `pyquil.unitary_tools` for dealing with unitary matrices.
  - `NumpyWavefunctionSimulator` uses numpy's tensordot functionality to efficiently evolve a statevector. For most simulations, performance is quite good.
  - `ReferenceDensitySimulator` uses matrix-matrix multiplication to evolve a density matrix.
- Matrix representations of Quil standard gates are included in `pyquil.gate_matrices` (gh-552).
- The density simulator has extremely limited support for Kraus-operator based noise models. Let us know if you're interested in contributing more robust noise-model support.
- This functionality should be considered experimental and may undergo minor API changes.

Important changes to note:

- Quil math functions (like COS, SIN, ...) used to be ambiguous with respect to case sensitivity. They are now case-sensitive and should be uppercase (gh-774).
- In the next release of pyQuil, communication with quilc will happen exclusively via the rpcq protocol. `LocalQVMCompiler` and `LocalBenchmarkConnection` will be removed in favor of a unified `QVMCompiler` and `BenchmarkConnection`. This change should be transparent if you use `get_qc` and

`get_benchmarker`, respectively. In anticipation of this change we recommend that you upgrade your version of quilc to 1.3, released Jan 30, 2019 (gh-730).

- When using a paramaterized gate, the QPU control electronics only allowed multiplying parameters by powers of two. If you only ever multiply a parameter by the same constant, this isn't too much of a problem because you can fold the multiplicative constant into the definition of the parameter. However, if you are multiplying the same variable (e.g. `gamma` in QAOA) by different constants (e.g. weighted maxcut edge weights) it doesn't work. PyQuil will now transparently handle the latter case by expanding to a vector of parameters with the constants folded in, allowing you to multiply variables by whatever you want (gh-707).

As always, this release contains bug fixes and improvements:

- The CZ gate fidelity metric available in the Specs object now has its associated standard error, which is accessible from the method `Specs.fCZ_std_errs` (gh-751).
- Operator estimation code now correctly handles identity terms with coefficients. Previously, it would always estimate these terms as 1.0 (gh-758).
- Operator estimation results include the total number of counts (shots) taken.
- Operator estimation JSON serialization uses utf-8. Please let us know if this causes problems (gh-769).
- The example quantum die program now can roll dice that are not powers of two (gh-749).
- The teleportation and Meyer penny game examples had a syntax error (gh-778, gh-772).
- When running on the QPU, you could get into trouble if the QPU name passed to `get_qc` did not match the lattice you booked. This is now validated (gh-771).

We extend thanks to community member estamm12 for their contribution to this release.

1.13.5 v2.2 (January 4, 2019)

PyQuil 2.2 is the latest release of pyQuil, Rigetti's toolkit for constructing and running quantum programs. Bug fixes and improvements include:

- `pauli.is_zero` and `paulis.is_identity` would sometimes return erroneous answers (gh-710).
- Parameter expressions involving addition and subtraction are now converted to Quil with spaces around the operators, e.g. `theta + 2` instead of `theta+2`. This disambiguates subtracting two parameters, e.g. `alpha - beta` is not one variable named `alpha-beta` (gh-743).
- T1 is accounted for in T2 noise models (gh-745).
- Documentation improvements (gh-723, gh-719, gh-720, gh-728, gh-732, gh-742).
- Support for PNG generation of circuit diagrams via LaTeX (gh-745).
- We've started transitioning to using Gitlab as our continuous integration provider for pyQuil (gh-741, gh-752).

This release includes a new module for facilitating the estimation of quantum observables/operators (gh-682). First-class support for estimating observables should make it easier to express near-term algorithms. This release includes:

- data structures for expressing tomography-like experiments and their results
- grouping of experiment settings that can be simultaneously estimated
- functionality to executing a tomography-like experiment on a quantum computer

Please look forward to more features and polish in future releases. Don't hesitate to submit feedback or suggestions as GitHub issues.

We extend thanks to community member petterwittek for their contribution to this release.

Bugfix release 2.2.1 was released January 11 to maintain compatibility with the latest version of the quilc compiler (gh-759).

1.13.6 v2.1 (November 30, 2018)

PyQuil 2.1 is an incremental release of pyQuil, Rigetti's toolkit for constructing and running quantum programs. Changes include:

- Major documentation improvements.
- `QuantumComputer.run()` accepts an optional `memory_map` parameter to facilitate running parametric executables (gh-657).
- `QuantumComputer.reset()` will reset the state of a QAM to recover from an error condition (gh-703).
- Bug fixes (gh-674, gh-696).
- Quil parser improvements (gh-689, gh-685).
- Optional interleaver argument when generating RB sequences (gh-673).
- Our GitHub organization name has changed from `rigetticomputing` to `rigetti` (gh-713).

1.13.7 v2.0 (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 *New in Forest 2 - Other* 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 *New in Forest 2 - Other* 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.

Version 2.0.1 was released on November 9, 2018 and includes documentation changes only. This release is only available as a git tag. We have not pushed a new package to PyPI.

1.13.8 v1.9 (June 6, 2018)

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

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

Qubit placeholders

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

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

- `PauliTerm` now remembers the order of its operations. `sX(1)*sZ(2)` will compile to different Quil code than `sZ(2)*sX(1)`, although the terms will still be equal according to the `__eq__` method. During `PauliSum` combination of like terms, a warning will be emitted if two terms are combined that have different orders of operation.
- `PauliTerm.id()` takes an optional argument `sort_ops` which defaults to True for backwards compatibility. However, this function should not be used for comparing term-type like it has been used previously. Use `PauliTerm.operations_as_set()` instead. In the future, `sort_ops` will default to False and will eventually be removed.
- `Program.alloc()` has been deprecated. Please instantiate `QubitPlaceholder()` directly or request a "register" (list) of `n` placeholders by using the class constructor `QubitPlaceholder.register(n)()`.
- Programs must contain either (1) all instantiated qubits with integer indexes or (2) all placeholder qubits of type `QubitPlaceholder`. We have found that most users use (1) but (2) will become useful with larger and more diverse devices.
- Programs that contain qubit placeholders must be **explicitly addressed** prior to execution. Previously, qubits would be assigned "under the hood" to integers 0...N. Now, you must use `address_qubits()` which returns a new program with all qubits indexed depending on the `qubit_mapping` argument. The original program is unaffected and can be "readdressed" multiple times.
- `PauliTerm` can now accept `QubitPlaceholder` in addition to integers.
- `QubitPlaceholder` is no longer a subclass of `Qubit`. `LabelPlaceholder` is no longer a subclass of `Label`.
- `QuilAtom` subclasses' hash functions have changed.

Randomized benchmarking sequence generation

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

In addition, the `CompilerConnection` has another new method, `apply_clifford_to_pauli()` which conjugates `PauliTerms` by `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`.

1.14 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.

1.14.1 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

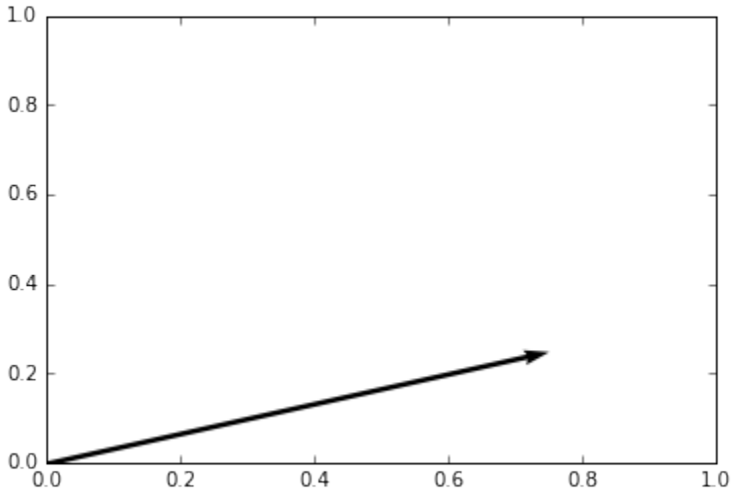
prob_bit = a * outcome_0 + b * outcome_1

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])
```

(continues on next page)

(continued from previous page)

```
plt.draw()
plt.show()
```



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

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

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

Dirac Notation

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

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

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

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

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

Multiple Probabilistic Bits

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

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

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

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

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

Similarly, the combined state for \(n\) such probabilistic bits is a vector of size \(2^n\) and is given by \(\bigotimes_{i=0}^{n-1}|\,v_i\rangle\). 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 (e.g. consider the state \(\frac{1}{2}|\,0_u0_v\rangle + \frac{1}{2}|\,1_u1_v\rangle\)). The most general composite state of \(n\) probabilistic bits can be written as \(\sum_{j=0}^{2^n - 1} a_{j} (\bigotimes_{i=0}^{n-1}|\,b_{ij}\rangle\)) where each \(b_{ij} \in \{0, 1\}\) and \(a_j \in \mathbb{R}\), i.e. as a linear combination (with real coefficients) of tensor products of basis states. Note that this still gives us \(2^n\) possible states.

Qubits

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

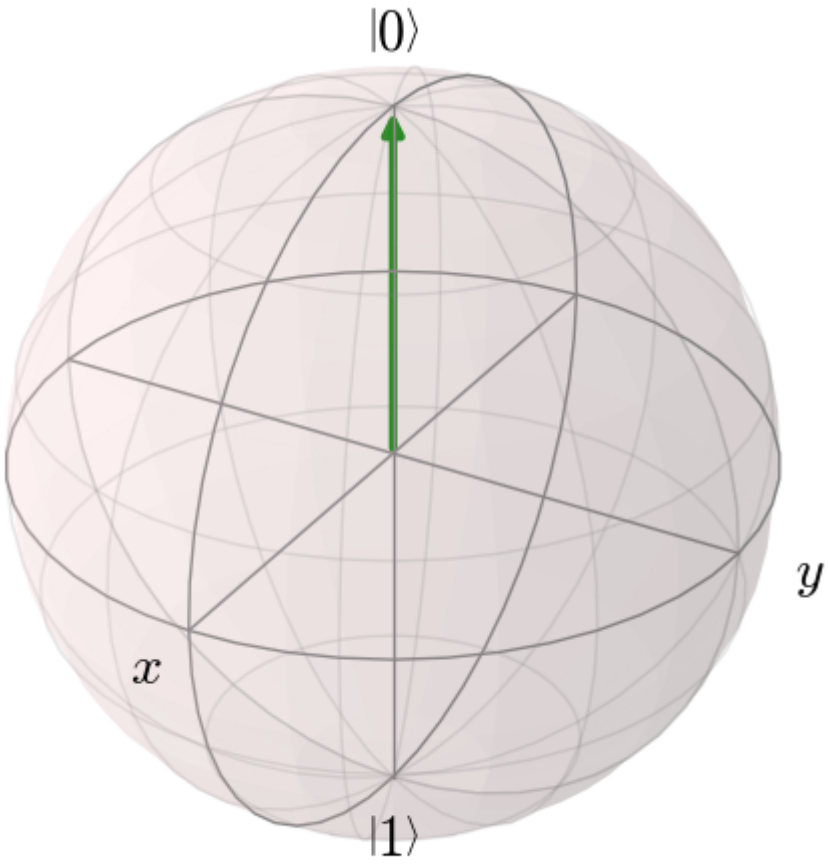
$$\alpha|\,0\rangle + \beta|\,1\rangle.$$

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

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

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

This switch to complex vectors means that rather than representing a state vector in a plane, we instead 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, by taking linear combinations (with complex coefficients, now) of tensor products of basis states. Thus \(n\) qubits have \(2^n\) possible states.

An Important Distinction

The probabilistic states described above represent ignorance of an underlying state, like 0 or 1 for probabilistic bits. **This is not true for quantum states.** The nature of quantum states is a deep topic with no full scientific consensus. However, no-go theorems like Bell's Theorem have ruled out the option of local hidden variable theories for quantum mechanics. Effectively, these say that quantum states can't be interpreted as purely representing ignorance of an underlying local objective state. In practice this means that a pure quantum state simply *is* the complex vector described in the last section, and we consider it just as "real" as a heads-up coin. This distinction between quantum and classical states is foundational for understanding quantum computing.

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 download the QVM and Compiler. 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 WavefunctionSimulator


# create a WavefunctionSimulator object
wavefunction_simulator = WavefunctionSimulator()

# 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 = wavefunction_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 = wavefunction_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 = wavefunction_simulator.wavefunction(p)
print("The quantum state is of dimension:", len(wavefunction.amplitudes))

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

p = Program()
for x in range(10):
    p += I(x)
wavefunction = wavefunction_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 = wavefunction_simulator.wavefunction(Program(I(0), I(1)))
print(wavefunction.get_outcome_probs())
```

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

1.14.2 Qubit Operations

In the previous section we introduced our first two **operations**: the `I` (or Identity) operation and the `X` (or NOT) 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. Such matrices have the characteristic property that their complex conjugate transpose is equal to their inverse, a property directly linked to the requirement that the probabilities of measuring qubits in any of the allowed states must sum to 1. 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(wavefunction_simulator.wavefunction(p))
```

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

Pauli Operators

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

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

```
from pyquil.gates import X, Y, Z


p = Program(X(0))
wavefunction = wavefunction_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 = wavefunction_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 = wavefunction_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 \(x\) axis. This explains the results of our code above: for a state vector initially in the +\(z\) direction, both `X` and `Y` gates will rotate it to -\(z\), and the `Z` gate will leave it unchanged.

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

Quantum programs are built by applying successive gate operations:

```
# Composing qubit operations is the same as multiplying matrices sequentially
p = Program(X(0), Y(0), Z(0))
wavefunction = wavefunction_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 = wavefunction_simulator.wavefunction(p)
print("CNOT|00> = ", wavefunction)
print("With outcome probabilities\n", wavefunction.get_outcome_probs(), "\n")

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

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

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

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

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

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

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

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

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

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

```
from pyquil.gates import SWAP


p = Program(X(0), SWAP(0,1))
wavefunction = wavefunction_simulator.wavefunction(p)

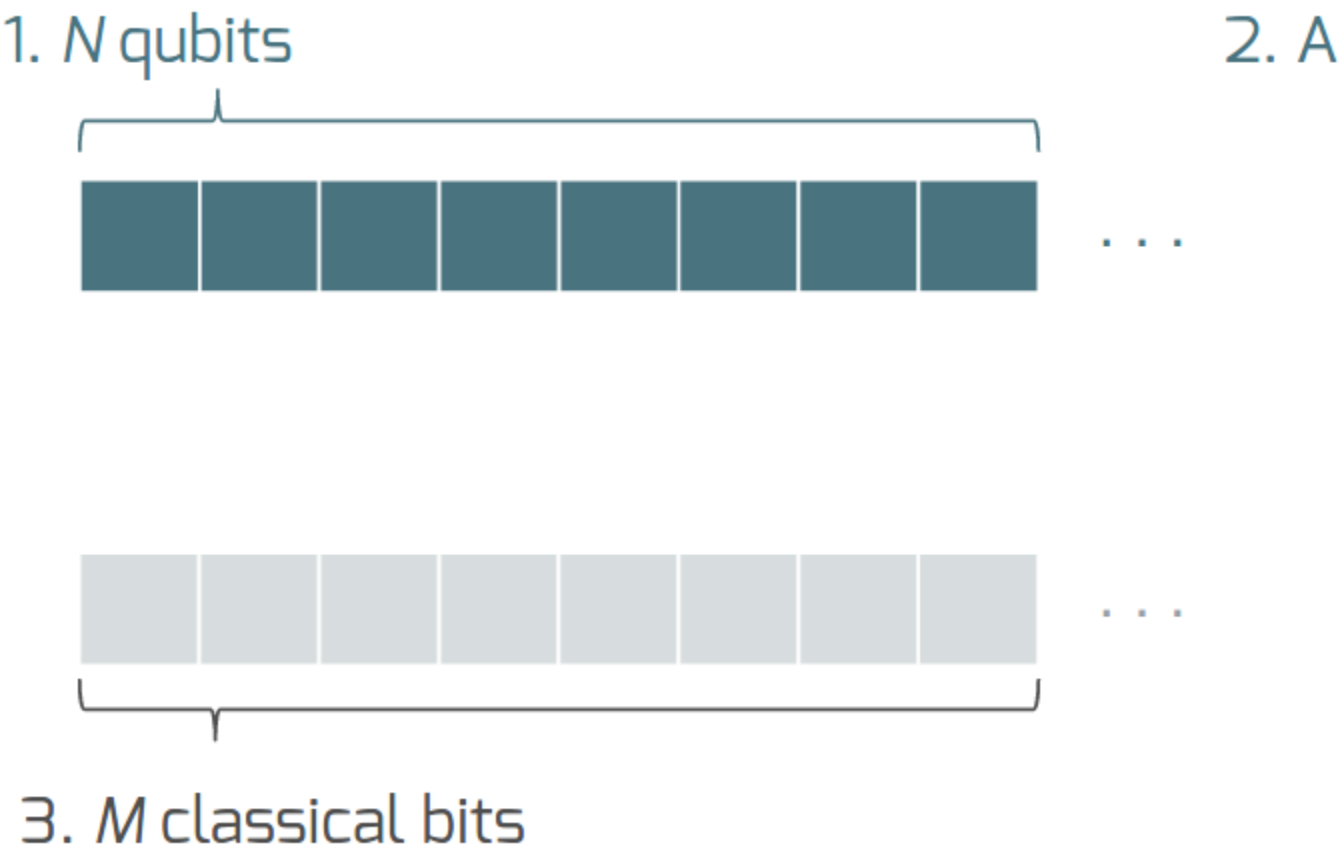
print("SWAP|01> = ", wavefunction)
print("With outcome probabilities\n", wavefunction.get_outcome_probs())
```

```
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.

1.14.3 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]
p = Program()
classical_register = p.declare('ro', 'BIT', 1)
p += Program(I(0)).measure(0, classical_register[0])
```

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 `QuantumComputer.run()`. Note that the `run` command is to be applied on the compiled version of the program.

```
from pyquil import get_qc


qc = get_qc('9q-square-qvm')
print (qc.run(qc.compile(p)))
```

```
[[0]]
```

We see that the classical register reports a value of zero. However, if we had flipped the qubit before measurement then we obtain:

```
p = Program()
classical_register = p.declare('ro', 'BIT', 1)
p += Program(X(0))    # Flip the qubit
p.measure(0, classical_register[0])    # Measure the qubit

print (qc.run(qc.compile(p)))
```

```
[[1]]
```

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

```
p = Program()
classical_register = p.declare('ro', 'BIT', 1)
p += Program(X(0))    # Flip the qubit
p.measure(0, classical_register[0])    # Measure the qubit

trials = 10
p.wrap_in_numshots_loop(shots=trials)

print (qc.run(qc.compile(p)))
```

```
[[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))
```

(continues on next page)

(continued from previous page)

```
wavefunction = wavefunction_simulator.wavefunction(coin_program)

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

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

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

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

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

```
p = Program()
ro = p.declare('ro', 'BIT', 1)

p += Program(H(0)).measure(0, ro[0])

# Measure qubit #0 a number of times
p.wrap_in_numshots_loop(shots=10)

# We see probabilistic results of about half 1's and half 0's
print (qc.run(qc.compile(p)))
```

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

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

```
coin_program = Program(H(0))
print ("Before measurement: H|0> = ", wavefunction_simulator.wavefunction(coin_
↪program), "\n")

ro = coin_program.declare('ro', 'BIT', 1)
coin_program.measure(0, ro[0])
for _ in range(5):
    print ("After measurement: ", wavefunction_simulator.wavefunction(coin_program))
```

```
Before measurement: H|0> =  (0.7071067812+0j)|0> + (0.7071067812+0j)|1>

After measurement:  (1+0j)|1>
After measurement:  (1+0j)|1>
After measurement:  (1+0j)|1>
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, as can be seen with this program,
# which prepares something called a Bell state (a special kind of "entangled state")
bell_program = Program(H(0), CNOT(0, 1))
wavefunction = wavefunction_simulator.wavefunction(bell_program)
print("Before measurement: Bell state = ", wavefunction, "\n")

classical_regs = bell_program.declare('ro', 'BIT', 2)
bell_program.measure(0, classical_regs[0]).measure(1, classical_regs[1])

for _ in range(5):
    wavefunction = wavefunction_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': 0.0, '01': 0.0, '10': 0.0, '11': 1.0}
After measurement:  {'00': 0.0, '01': 0.0, '10': 0.0, '11': 1.0}
After measurement:  {'00': 0.0, '01': 0.0, '10': 0.0, '11': 1.0}
After measurement:  {'00': 0.0, '01': 0.0, '10': 0.0, '11': 1.0}
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

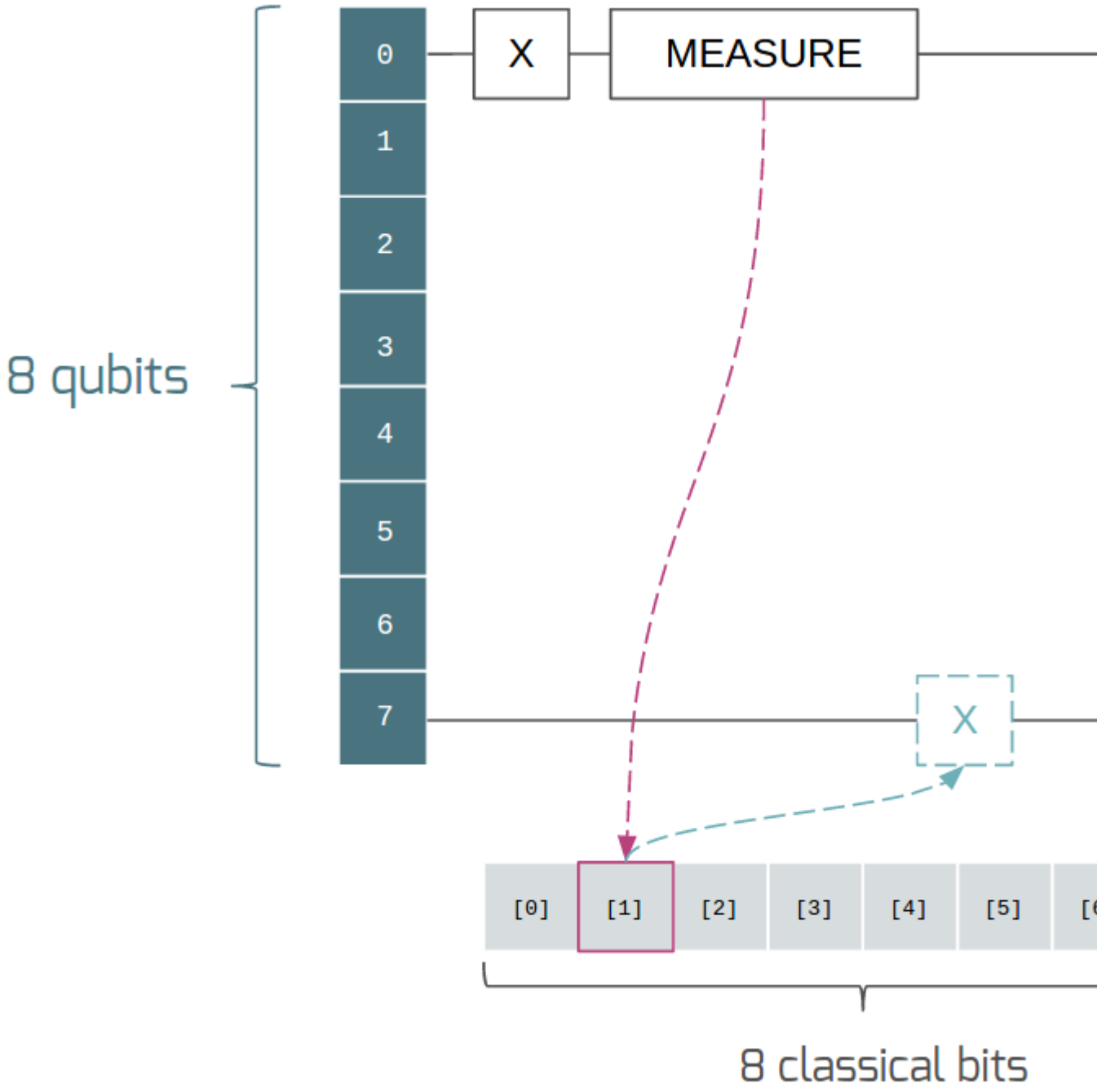
# Branch on ro[1]
p = Program()
ro = p.declare('ro', 'BIT', 8)
p += Program(X(0)).measure(0, ro[1]).if_then(ro[1], true_branch, false_branch)

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

# Run and check register [7]
print (qc.run(qc.compile(p)))
```

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

The second [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!

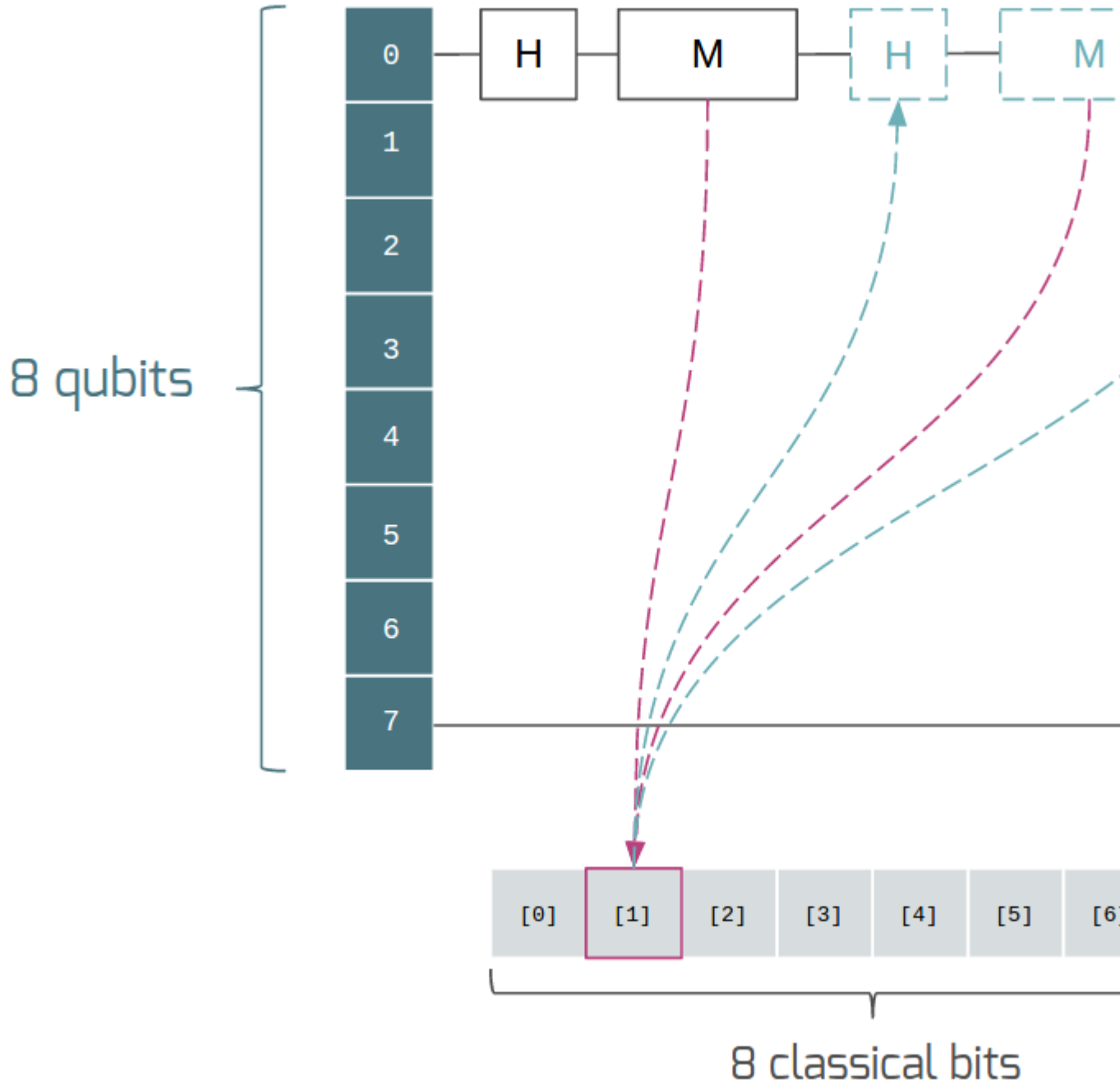
```
p = Program()
ro = p.declare('ro', 'BIT', 1)
inside_loop = Program(H(0)).measure(0, ro[0])
p.inst(X(0)).while_do(ro[0], inside_loop)
```

(continues on next page)

(continued from previous page)

```
qc = get_qc('9q-square-qvm')
print (qc.run(qc.compile(p)))
```

```
[[0]]
```



1.14.4 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.

1.15 Program

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

A list of pyQuil instructions that comprise a quantum program.

```
>>> from pyquil import Program
>>> from pyquil.gates import *
>>> p = Program()
>>> p += H(0)
>>> p += CNOT(0, 1)
```

Attributes

instructions	Fill in any placeholders and return a list of quil AbstractInstructions.
defined_gates	A list of defined gates on the program.
out()	Serializes the Quil program to a string suitable for submitting to the QVM or QPU.
get_qubits([indices])	Returns all of the qubit indices used in this program, including gate applications and allocated qubits.
is_protoquil()	Protoquil programs may only contain gates, Pragmas, and RESET.

1.15.1 Program.instructions

Program.**instructions**

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

1.15.2 Program.defined_gates

Program.**defined_gates**

A list of defined gates on the program.

1.15.3 Program.out

Program.**out**()

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

1.15.4 Program.get_qubits

Program.**get_qubits**(*indices=True*)

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

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

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

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

Return type set

1.15.5 Program.is_protoquil

Program.**is_protoquil**()

Protoquil programs may only contain gates, Pragmas, and RESET. It may not contain classical instructions or jumps.

Returns True if the Program is Protoquil, False otherwise

Program Construction

__iadd__(other)	Concatenate two programs together using +=, returning a new one.
__add__(other)	Concatenate two programs together, returning a new one.
inst(*instructions)	Mutates the Program object by appending new instructions.
gate(name, params, qubits)	Add a gate to the program.
defgate(name, matrix[, parameters])	Define a new static gate.
define_noisy_gate(name, qubit_indices, kraus_ops)	Overload a static ideal gate with a noisy one defined in terms of a Kraus map.
define_noisy_readout(qubit, p00, p11)	For this program define a classical bit flip readout error channel parametrized by `p00` and `p11`.
no_noise()	Prevent a noisy gate definition from being applied to the immediately following Gate instruction.
measure(qubit_index, classical_reg)	Measures a qubit at qubit_index and puts the result in classical_reg
reset([qubit_index])	Reset all qubits or just a specific qubit at qubit_index.
measure_all(*qubit_reg_pairs)	Measures many qubits into their specified classical bits, in the order they were entered.
alloc()	Get a new qubit.
declare(name[, memory_type, memory_size, ...])	DECLARE a quil variable

Continued on next page

Table 2 – continued from previous page

wrap_in_numshots_loop(shots)	Wraps a Quil program in a loop that re-runs the same program many times.

1.15.6 Program.__iadd__

Program.**__iadd__**(*other*)

Concatenate two programs together using +=, returning a new one.

Parameters **other** (Program) – Another program or instruction to concatenate to this one.

Returns A newly concatenated program.

Return type *Program*

1.15.7 Program.__add__

Program.**__add__**(*other*)

Concatenate two programs together, returning a new one.

Parameters **other** (Program) – Another program or instruction to concatenate to this one.

Returns A newly concatenated program.

Return type *Program*

1.15.8 Program.inst

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

1.15.9 Program.gate

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

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

Parameters

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

Returns The Program instance

Return type *Program*

1.15.10 Program.defgate

Program.**defgate**(*name*, *matrix*, *parameters=None*)
Define a new static gate.

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

Parameters

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

Returns The Program instance.

Return type *Program*

1.15.11 Program.define_noisy_gate

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

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

Parameters

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

Returns The Program instance

Return type *Program*

1.15.12 Program.define_noisy_readout

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*

1.15.13 Program.no_noise

Program.**no_noise**()

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

Returns Program

1.15.14 Program.measure

Program.**measure**(*qubit_index*, *classical_reg*)

Measures a qubit at qubit_index and puts the result in classical_reg

Parameters

- **qubit_index** (*int*) – The address of the qubit to measure.
- **classical_reg** (*int*) – The address of the classical bit to store the result. Set to None to measure for effect (discard result).

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

Return type *Program*

1.15.15 Program.reset

Program.**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*

1.15.16 Program.measure_all

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.

Return type *Program*

1.15.17 Program.alloc

Program.**alloc**()
Get a new qubit.

Returns A qubit.

Return type Qubit

1.15.18 Program.declare

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 memory: 'BIT', 'REAL', 'OCTET' or 'INTEGER'
- **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]`.

1.15.19 Program.wrap_in_numshots_loop

`Program.`**`wrap_in_numshots_loop`**(*shots*)

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** (*int*) – Number of iterations to loop through.

Control Flow

while_do(classical_reg, q_program)	While a classical register at index classical_reg is 1, loop q_program
if_then(classical_reg, if_program[, ...])	If the classical register at index classical reg is 1, run if_program, else run else_program.

1.15.20 Program.while_do

`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:

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*

1.15.21 Program.if_then

`Program.`**`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:

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*

Utility Methods

copy()	Perform a shallow copy of this program.
pop()	Pops off the last instruction.
dagger([inv_dict, suffix])	Creates the conjugate transpose of the Quil program.
__getitem__(index)	Allows indexing into the program to get an action.

1.15.22 Program.copy

Program.**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

1.15.23 Program.pop

Program.**pop**()

Pops off the last instruction.

Returns The instruction that was popped.

Return type tuple

1.15.24 Program.dagger

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*

1.15.25 Program.__getitem__

Program.**__getitem__**(*index*)

Allows indexing into the program to get an action.

Parameters **index** – The action at the specified index.

Returns

1.15.26 Utility Functions

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.**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.**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.**implicitly_declare_ro**(*instructions*)
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 you 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.**merge_with_pauli_noise**(*prog_list*, *probabilities*, *qubits*)
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** (Iterable[+T_co]) – 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** (List[~T]) – The 4^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** (List[~T]) – 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.**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.**get_classical_addresses_from_program**(*program*)

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

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

Return type Dict[str, List[int]]

Returns A mapping from memory region names to lists of offsets appearing in the program.

pyquil.quil.**percolate_declares**(*program*)

Move all the DECLARE statements to the top of the program. Return a fresh obejct.

Parameters **program** (*Program*) – Perhaps jumbled program.

Return type *Program*

Returns Program with DECLAREs all at the top and otherwise the same sorted contents.

pyquil.quil.**validate_protoquil**(*program*)

Ensure that a program is valid ProtoQuil, otherwise raise a ValueError. Protoquil is a subset of Quil which excludes control flow and classical instructions.

Parameters **program** (*Program*) – The Quil program to validate.

Return type None

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

1.16 Gates and Instructions

A *Program* is effectively a list of gates and instructions which can be created using the function documented in this section:

```
>>> p = Program()
>>> p += H(qubit=0)
>>> p += RY(angle=pi/3, qubit=1)
>>> p += CNOT(0, 1)
    ...
```

1.16.1 Native gates for Rigetti QPUs

Physical quantum processors can enact a subset of all named gates. Luckily, a small set of gates is universal for quantum computation, so all named gates can be enacted by suitable combinations of physically realizable gates. Rigetti's superconducting quantum processors can perform *RX()* with `angle=+-pi/2` or `angle=+-pi`, *RZ()* with an arbitrary angle, and *CZ()* interactions between neighboring qubits. Rigetti QPUs can natively measure in the computational (Z) basis.

RX(angle, qubit)	Produces the RX gate.
RZ(angle, qubit)	Produces the RZ gate.
CZ(control, target)	Produces a controlled-Z gate.
MEASURE(qubit, classical_reg)	Produce a MEASURE instruction.

RX

pyquil.gates.**RX**(*angle*, *qubit*)

Produces the RX gate:

```
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.

RZ

pyquil.gates.**RZ**(*angle*, *qubit*)

Produces the RZ gate:

```
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.

CZ

pyquil.gates.**CZ**(*control*, *target*)

Produces a controlled-Z gate:

```
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.

MEASURE

pyquil.gates.**MEASURE**(*qubit*, *classical_reg*)

Produce a MEASURE instruction.

Parameters

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

Returns A Measurement instance.

1.16.2 All gates and instructions

In general, you will write a quantum program using the full suite of Quil gates and instructions and use *the Quil compiler* to compile programs into the target instruction set (ISA). The full list of quantum gates and classical Quil instructions is enumerated here.

Single-qubit gates

I(qubit)	Produces the I identity gate.
X(qubit)	Produces the X ("NOT") gate.
Y(qubit)	Produces the Y gate.
Z(qubit)	Produces the Z gate.
H(qubit)	Produces the Hadamard gate.
S(qubit)	Produces the S gate.
T(qubit)	Produces the T gate.
RX(angle, qubit)	Produces the RX gate.
RY(angle, qubit)	Produces the RY gate.
RZ(angle, qubit)	Produces the RZ gate.
PHASE(angle, qubit)	Produces the PHASE gate.

I

pyquil.gates.**I**(*qubit*)

Produces the I identity gate:

```
I = [1, 0]
    [0, 1]
```

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

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

Returns A Gate object.

X

pyquil.gates.**X**(*qubit*)

Produces the X ("NOT") gate:

```
X = [[0, 1],
     [1, 0]]
```

This gate is a single qubit X-gate.

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

Returns A Gate object.

Y

pyquil.gates.**Y**(*qubit*)

Produces the Y gate:

```
Y = [[0, 0 - 1j],
     [0 + 1j, 0]]
```

This gate is a single qubit Y-gate.

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

Returns A Gate object.

Z

pyquil.gates.**Z**(*qubit*)
Produces the Z gate:

```
Z = [[1,  0],
     [0, -1]]
```

This gate is a single qubit Z-gate.

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

Returns A Gate object.

H

pyquil.gates.**H**(*qubit*)
Produces the Hadamard gate:

```
H = (1 / sqrt(2)) * [[1,  1],
                     [1, -1]]
```

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

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

Returns A Gate object.

S

pyquil.gates.**S**(*qubit*)
Produces the S gate:

```
S = [[1, 0],
     [0, 1j]]
```

This gate is a single qubit S-gate.

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

Returns A Gate object.

T

pyquil.gates.**T**(*qubit*)
Produces the T gate:

```
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.

RY

pyquil.gates.**RY**(*angle*, *qubit*)

Produces the RY gate:

```
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.

PHASE

pyquil.gates.**PHASE**(*angle*, *qubit*)

Produces the PHASE gate:

```
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.

Multi-qubit gates

CZ(control, target)	Produces a controlled-Z gate.
CNOT(control, target)	Produces a controlled-NOT (controlled-X) gate.
CCNOT(control1, control2, target)	Produces a doubly-controlled NOT gate.
CPHASE00(angle, control, target)	Produces a controlled-phase gate that phases the |00> state.
CPHASE01(angle, control, target)	Produces a controlled-phase gate that phases the |01> state.
CPHASE10(angle, control, target)	Produces a controlled-phase gate that phases the |10> state.
CPHASE(angle, control, target)	Produces a controlled-phase instruction.
SWAP(q1, q2)	Produces a SWAP gate which swaps the state of two qubits.
CSWAP(control, target_1, target_2)	Produces a controlled-SWAP gate.
ISWAP(q1, q2)	Produces an ISWAP gate.
PSWAP(angle, q1, q2)	Produces a parameterized SWAP gate.

CNOT

pyquil.gates.**CNOT**(*control*, *target*)

Produces a controlled-NOT (controlled-X) gate:

```
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 |1> state.

Returns A Gate object.

CCNOT

pyquil.gates.**CCNOT**(*control1*, *control2*, *target*)

Produces a doubly-controlled NOT gate:

```
CCNOT = [[1, 0, 0, 0, 0, 0, 0, 0],
         [0, 1, 0, 0, 0, 0, 0, 0],
         [0, 0, 1, 0, 0, 0, 0, 0],
         [0, 0, 0, 1, 0, 0, 0, 0],
         [0, 0, 0, 0, 1, 0, 0, 0],
         [0, 0, 0, 0, 0, 1, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 1],
         [0, 0, 0, 0, 0, 0, 1, 0]]
```

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.

CPHASE00

pyquil.gates.**CPHASE00**(*angle*, *control*, *target*)

Produces a controlled-phase gate that phases the |00> state:

```
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 |0> state.
- **control** – Qubit 1.
- **target** – Qubit 2.

Returns A Gate object.

CPHASE01

pyquil.gates.**CPHASE01**(*angle*, *control*, *target*)

Produces a controlled-phase gate that phases the |01> state:

```
CPHASE01(phi) = 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 |1> state and q2 is in the |0> state.
- **control** – Qubit 1.
- **target** – Qubit 2.

Returns A Gate object.

CPHASE10

pyquil.gates.**CPHASE10**(*angle*, *control*, *target*)

Produces a controlled-phase gate that phases the |10> state:

```
CPHASE10(phi) = 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 |1> state and q1 is in the |0> state.
- **control** – Qubit 1.
- **target** – Qubit 2.

Returns A Gate object.

CPHASE

pyquil.gates.**CPHASE**(*angle*, *control*, *target*)

Produces a controlled-phase instruction:

```
CPHASE(phi) = 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.

Compare with the `CPHASExx()` variants. This variant is the most common and does not have a suffix, although you can think of it as ‘‘CPHASE11‘.

Parameters

- **angle** – The input phase angle to apply when both qubits are in the `|1>` state.
- **control** – Qubit 1.
- **target** – Qubit 2.

Returns A Gate object.

SWAP

pyquil.gates.**SWAP**(*q1*, *q2*)

Produces a SWAP gate which swaps the state of two qubits:

```
SWAP = [[1, 0, 0, 0],
        [0, 0, 1, 0],
        [0, 1, 0, 0],
        [0, 0, 0, 1]]
```

Parameters

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

Returns A Gate object.

CSWAP

pyquil.gates.**CSWAP**(*control*, *target_1*, *target_2*)

Produces a controlled-SWAP gate. This gate conditionally swaps the state of two qubits:

```
CSWAP = [[1, 0, 0, 0, 0, 0, 0, 0],
         [0, 1, 0, 0, 0, 0, 0, 0],
         [0, 0, 1, 0, 0, 0, 0, 0],
         [0, 0, 0, 1, 0, 0, 0, 0],
         [0, 0, 0, 0, 1, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 1, 0],
         [0, 0, 0, 0, 0, 1, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 1]]
```

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 `|1>` state.

ISWAP

pyquil.gates.**ISWAP**(*q1*, *q2*)

Produces an ISWAP gate:

```
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 1 state and a -i phase to q2 when it is in the 0 state.

Parameters

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

Returns A Gate object.

PSWAP

pyquil.gates.**PSWAP**(*angle*, *q1*, *q2*)

Produces a parameterized SWAP gate:

```
PSWAP(phi) = [[1, 0,              0,              0],
              [0, 0,              exp(1j * phi), 0],
              [0, exp(1j * phi), 0,              0],
              [0, 0,              0,              1]]
```

Parameters

- **angle** – The angle of the phase to apply to the swapped states. This phase is applied to q1 when it is in the 1 state and to q2 when it is in the 0 state.
- **q1** – Qubit 1.
- **q2** – Qubit 2.

Returns A Gate object.

Classical instructions

WAIT	This instruction tells the quantum computation to halt.
RESET([qubit_index])	Reset all qubits or just a specific qubit at qubit_index.
NOP	This instruction applies no operation at that timestep.
HALT	This instruction ends the program.
MEASURE(qubit, classical_reg)	Produce a MEASURE instruction.
NEG(classical_reg)	Produce a NEG instruction.
NOT(classical_reg)	Produce a NOT instruction.
AND(classical_reg1, classical_reg2)	Produce an AND instruction.
OR(classical_reg1, classical_reg2)	Produce an OR instruction.
IOR(classical_reg1, classical_reg2)	Produce an inclusive OR instruction.
XOR(classical_reg1, classical_reg2)	Produce an exclusive OR instruction.

Continued on next page

Table 8 – continued from previous page

MOVE(classical_reg1, classical_reg2)	Produce a MOVE instruction.
EXCHANGE(classical_reg1, classical_reg2)	Produce an EXCHANGE instruction.
LOAD(target_reg, region_name, offset_reg)	Produce a LOAD instruction.
STORE(region_name, offset_reg, source)	Produce a STORE instruction.
CONVERT(classical_reg1, classical_reg2)	Produce a CONVERT instruction.
ADD(classical_reg, right)	Produce an ADD instruction.
SUB(classical_reg, right)	Produce a SUB instruction.
MUL(classical_reg, right)	Produce a MUL instruction.
DIV(classical_reg, right)	Produce an DIV instruction.
EQ(classical_reg1, classical_reg2, ...)	Produce an EQ instruction.
LT(classical_reg1, classical_reg2, ...)	Produce an LT instruction.
LE(classical_reg1, classical_reg2, ...)	Produce an LE instruction.
GT(classical_reg1, classical_reg2, ...)	Produce an GT instruction.
GE(classical_reg1, classical_reg2, ...)	Produce an GE instruction.

WAIT

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.

RESET

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]

NOP

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.

HALT

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

This instruction ends the program.

Returns A Halt object.

NEG

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

Parameters **classical_reg** – A classical memory address to modify.

Returns A ClassicalNeg instance.

NOT

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

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

Returns A ClassicalNot instance.

AND

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.

OR

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.

IOR

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.

XOR

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.

MOVE

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.

EXCHANGE

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.

LOAD

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.

STORE

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.

CONVERT

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.

ADD

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.

SUB

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.

MUL

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.

DIV

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.

EQ

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.

LT

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.

LE

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.

GT

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.

GE

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.

Collections

QUANTUM_GATES	Dictionary of quantum gate functions keyed by gate names.
STANDARD_INSTRUCTIONS	Dictionary of standard instruction functions keyed by instruction names.

1.17 Pauli Operators

Quantum operators can be expressed as combinations of Pauli operators I, X, Y, Z:

```
>>> operator = sZ(0)*sZ(1) + sX(2)*sY(3)
>>> print(operator)
(1+0j)*Z0*Z1 + (1+0j)*X2*Y3
```

1.17.1 Construction functions

sX(q)	A function that returns the sigma_X operator on a particular qubit.
sY(q)	A function that returns the sigma_Y operator on a particular qubit.
sZ(q)	A function that returns the sigma_Z operator on a particular qubit.

Continued on next page

Table 10 – continued from previous page

sI([q])	A function that returns the identity operator, optionally on a particular qubit.
ID()	The identity operator.
ZERO()	The zero operator.

sX

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*

sY

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*

sZ

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*

sI

pyquil.paulis.**sI**(*q=None*)
A function that returns the identity operator, optionally on a particular qubit.

This can be specified without a qubit.

Parameters **qubit_index** (*int*) – The optional index of a qubit.

Returns A PauliTerm object

Return type *PauliTerm*

ID

pyquil.paulis.**ID**()
The identity operator.

ZERO

pyquil.paulis.**ZERO**()
The zero operator.

1.17.2 Working with operators

simplify_pauli_sum(pauli_sum)	Simplify the sum of Pauli operators according to Pauli algebra rules.
check_commutation(pauli_list, pauli_two)	Check if commuting a PauliTerm commutes with a list of other terms by natural calculation.
commuting_sets(pauli_terms)	Gather the Pauli terms of pauli_terms variable into commuting sets
is_identity(term)	Tests to see if a PauliTerm or PauliSum is a scalar multiple of identity
is_zero(pauli_object)	Tests to see if a PauliTerm or PauliSum is zero.
exponentiate(term)	Creates a pyQuil program that simulates the unitary evolution exp(-1j * term)
exponential_map(term)	Returns a function f(alpha) that constructs the Program corresponding to exp(-1j*alpha*term).
exponentiate_commuting_pauli_sum(pauli_sum)	Returns a function that maps all substituent PauliTerms and sums them into a program.
suzuki_trotter(trotter_order, trotter_steps)	Generate trotterization coefficients for a given number of Trotter steps.
trotterize(first_pauli_term, second_pauli_term)	Create a Quil program that approximates exp((A + B)t) where A and B are PauliTerm operators.

simplify_pauli_sum

pyquil.paulis.**simplify_pauli_sum**(*pauli_sum*)
Simplify the sum of Pauli operators according to Pauli algebra rules.

check_commutation

pyquil.paulis.**check_commutation**(*pauli_list*, *pauli_two*)
Check if commuting a PauliTerm commutes with a list of other terms by natural calculation. Uses the result in Section 3 of arXiv:1405.5749v2, modified slightly here to check for the number of anti-coincidences (which must always be even for commuting PauliTerms) instead of the no. of coincidences, as in the paper.

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

commuting_sets

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

is_identity

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

Tests to see if a PauliTerm or PauliSum is a scalar multiple of identity

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

Returns True if the PauliTerm or PauliSum is a scalar multiple of identity, False otherwise

Return type bool

is_zero

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

exponentiate

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

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

Parameters **term** (*PauliTerm*) – A pauli term to exponentiate

Returns A Program object

Return type *Program*

exponential_map

pyquil.paulis.**exponential_map**(*term*)

Returns a function f(alpha) that constructs the Program corresponding to exp(-1j*alpha*term).

Parameters **term** – A pauli term to exponentiate

Returns A function that takes an angle parameter and returns a program.

Return type Function

exponentiate_commuting_pauli_sum

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

suzuki_trotter

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

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

Parameters

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

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

Return type list

trotterize

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*

1.17.3 Classes

class pyquil.paulis.**PauliSum**(*terms*)
A sum of one or more PauliTerms.

Parameters **terms** (*Sequence*) – A Sequence of PauliTerms.

Methods

get_qubits()	The support of all the operators in the PauliSum object.
simplify()	Simplifies the sum of Pauli operators according to Pauli algebra rules.
get_programs()	Get a Pyquil Program corresponding to each term in the PauliSum and a coefficient for each program

PauliSum.get_qubits

PauliSum.**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

PauliSum.simplify

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

PauliSum.get_programs

PauliSum.**get_programs**()
Get a Pyquil Program corresponding to each term in the PauliSum and a coefficient for each program

Returns (programs, coefficients)

class pyquil.paulis.**PauliTerm**(*op*, *index*, *coefficient=1.0*)
A term is a product of Pauli operators operating on different qubits.

Create a new Pauli Term with a Pauli operator at a particular index and a leading coefficient.

Parameters

- **op** (str) – The Pauli operator as a string "X", "Y", "Z", or "I"
- **index** (int) – The qubit index that that operator is applied to.
- **coefficient** (Union[int, float, complex]) – The coefficient multiplying the operator, e.g. 1.5 * Z_1

Methods

id([sort_ops])	Returns an identifier string for the PauliTerm (ignoring the coefficient).
operations_as_set()	Return a frozenset of operations in this term.
copy()	Properly creates a new PauliTerm, with a completely new dictionary of operators
program	

Continued on next page

Table 13 – continued from previous page

from_list(terms_list[, coefficient])	Allocates a Pauli Term from a list of operators and indices.
pauli_string([qubits])	Return a string representation of this PauliTerm without its coefficient and with implicit qubit indices.

PauliTerm.id

PauliTerm.**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 a future version. 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

PauliTerm.operations_as_set

PauliTerm.**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 (op_str, coefficient) representing Pauli operations

PauliTerm.copy

PauliTerm.**copy**()

Properly creates a new PauliTerm, with a completely new dictionary of operators

PauliTerm.program

PauliTerm.**program**

PauliTerm.from_list

classmethod PauliTerm.**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

PauliTerm.pauli_string

PauliTerm.**pauli_string**(*qubits=None*)

Return a string representation of this PauliTerm without its coefficient and with implicit qubit indices.

If a list of qubits is provided, each character in the resulting string represents a Pauli operator on the corresponding qubit. If qubit indices are not provided as input, the returned string will be all non-identity operators in the order. This doesn't make much sense, so please provide a list of qubits. Not providing a list of qubits is deprecated.

```
>>> p = PauliTerm("X", 0) * PauliTerm("Y", 1, 1.j)
>>> p.pauli_string()
"XY"
>>> p.pauli_string(qubits=[0])
"X"
>>> p.pauli_string(qubits=[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, sans coefficient

1.18 Quantum Computer

pyquil.**get_qc**(*name*, *, *as_qvm=None*, *noisy=None*, *connection=None*)

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-1-16Q-A-noisy-qvm")
>>> qc = get_qc("Aspen-1-16Q-A", as_qvm=True, noisy=True)
```

and will construct a simulator of an Aspen-1 lattice 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()`.

The Rigetti QVM must be downloaded from https://www.rigetti.com/forest and run as a server alongside your python program. To use pyQuil's built-in QVM, replace all `"-qvm"` suffixes with `"-pyqvm"`:

```
>>> qc = get_qc("5q-pyqvm")
```

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** (str) – 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 "-pyqvm" will return a PyQVM. 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** (Optional[bool]) – 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** (Optional[bool]) – 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 QVMs 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** (Optional[ForestConnection]) – An optional ForestConnection object. If not specified, the default values for URL endpoints will be used. If you deign to change any of these parameters, pass your own ForestConnection object.

Return type QuantumComputer

Returns A pre-configured QuantumComputer

pyquil.**list_quantum_computers**(*connection=None*, *qpus=True*, *qvms=True*)

List the names of available quantum computers

Parameters

- **connection** (Optional[ForestConnection]) – 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** (bool) – Whether to include QPU's in the list.
- **qvms** (bool) – Whether to include QVM's in the list.

Return type List[str]

class pyquil.api.**QuantumComputer**(*, *name*, *qam*, *device*, *compiler*, *symmetrize_readout=False*)

A quantum computer for running quantum programs.

A quantum computer has various characteristics like supported gates, qubits, qubit topologies, gate fidelities, and more. A quantum computer also has the ability to run quantum programs.

A quantum computer can be a real Rigetti QPU that uses superconducting transmon qubits to run quantum programs, or it can be an emulator like the Rigetti QVM with noise models and mimicked topologies.

Parameters

- **name** (*str*) – A string identifying this particular quantum computer.
- **qam** (*QAM*) – A quantum abstract machine which handles executing quantum programs. This dispatches to a QVM or QPU.
- **device** (*AbstractDevice*) – A collection of connected qubits and associated specs and topology.
- **symmetrize_readout** (*bool*) – Whether to apply readout error symmetrization. See *run_symmetrized_readout()* for a complete description.

Methods

run(executable[, memory_map])	Run a quil executable.
run_and_measure(program, trials)	Run the provided state preparation program and measure all qubits.
run_symmetrized_readout(program, trials)	Run a quil program in such a way that the readout error is made collectively symmetric
qubits()	Return a sorted list of this QuantumComputer's device's qubits
qubit_topology()	Return a NetworkX graph representation of this QuantumComputer's device's qubit connectivity.
get_isa([oneq_type, twoq_type])	Return a target ISA for this QuantumComputer's device.
compile(program[, to_native_gates, optimize])	A high-level interface to program compilation.

1.18.1 QuantumComputer.run

QuantumComputer.**run**(*executable*, *memory_map=None*)

Run a quil executable. If the executable contains declared parameters, then a memory map must be provided, which defines the runtime values of these parameters.

Parameters

- **executable** (Union[BinaryExecutableResponse, PyQuilExecutableResponse]) – The program to run. You are responsible for compiling this first.
- **memory_map** (Optional[Dict[str, List[Union[int, float]]]]) – The mapping of declared parameters to their values. The values are a list of floats or integers.

Return type ndarray

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

1.18.2 QuantumComputer.run_and_measure

QuantumComputer.**run_and_measure**(*program*, *trials*)

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 qc.qubits()).T
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** (*Program*) – The state preparation program to run and then measure.
- **trials** (`int`) – The number of times to run the program.

Return type `Dict[int, ndarray]`

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

1.18.3 QuantumComputer.run_symmetrized_readout

`QuantumComputer.`**`run_symmetrized_readout`**(*program*, *trials*)

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.

Return type `ndarray`

1.18.4 QuantumComputer.qubits

`QuantumComputer.`**`qubits`**`()`

Return a sorted list of this QuantumComputer's device's qubits

See `AbstractDevice.qubits()` for more.

Return type `List[int]`

1.18.5 QuantumComputer.qubit_topology

QuantumComputer.**qubit_topology**()
Return a NetworkX graph representation of this QuantumComputer's device's qubit connectivity.

See AbstractDevice.qubit_topology() for more.

Return type <module 'networkx.classes.graph' from '/home/docs/checkouts/readthedocs.org/user_builds/pyquil/envs/
packages/networkx/classes/graph.py'>

1.18.6 QuantumComputer.get_isa

QuantumComputer.**get_isa**(*oneq_type='Xhalves'*, *twoq_type='CZ'*)
Return a target ISA for this QuantumComputer's device.

See AbstractDevice.get_isa() for more.

Parameters

- **oneq_type** (str) – The family of one-qubit gates to target
- **twoq_type** (str) – The family of two-qubit gates to target

Return type *ISA*

1.18.7 QuantumComputer.compile

QuantumComputer.**compile**(*program*, *to_native_gates=True*, *optimize=True*)
A high-level interface to program compilation.

Compilation currently consists of two stages. Please see the AbstractCompiler docs for more information. This function does all stages of compilation.

Right now both to_native_gates and optimize must be either both set or both unset. More modular compilation passes may be available in the future.

Parameters

- **program** (*Program*) – A Program
- **to_native_gates** (bool) – Whether to compile non-native gates to native gates.
- **optimize** (bool) – Whether to optimize programs to reduce the number of operations.

Return type Union[BinaryExecutableResponse, PyQuilExecutableResponse]

Returns An executable binary suitable for passing to *QuantumComputer.run()*.

1.19 Compilers

An appropriate compiler is automatically created when using *get_qc()* and it is stored on the *QuantumComputer* object as the compiler attribute.

The exact process for compilation depends on whether you're targeting a QPU or a QVM, and you can conceive of other compilation strategies than those included with pyQuil by default. Therefore, we define an abstract interface that all compilers must follow. See *AbstractCompiler* for more, or use one of the listed compilers below.

_qac.AbstractCompiler	The abstract interface for a compiler.
QVMCompiler(endpoint, device[, timeout])	Client to communicate with the Compiler Server.
QPUCompiler(quilc_endpoint, ... [, timeout, name])	Client to communicate with the Compiler Server.

1.19.1 pyquil.api._qac.AbstractCompiler

class pyquil.api._qac.**AbstractCompiler**

The abstract interface for a compiler.

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

get_version_info()	Return version information for this compiler and its dependencies.
native_quil_to_executable(nq_program)	Compile a native quil program to a binary executable.
quil_to_native_quil(program)	Compile an arbitrary quil program according to the ISA of target_device.

1.19.2 pyquil.api.QVMCompiler

class pyquil.api.**QVMCompiler**(*endpoint*, *device*, *timeout=10*)

Client to communicate with the Compiler Server.

Parameters

- **endpoint** (str) – TCP or IPC endpoint of the Compiler Server
- **device** (*AbstractDevice*) – PyQuil Device object to use as compilation target

__init__(*endpoint*, *device*, *timeout=10*)

Client to communicate with the Compiler Server.

Parameters

- **endpoint** (str) – TCP or IPC endpoint of the Compiler Server
- **device** (*AbstractDevice*) – PyQuil Device object to use as compilation target

Return type None

Methods

__init__(endpoint, device[, timeout])	Client to communicate with the Compiler Server.
get_version_info()	Return version information for this compiler and its dependencies.
native_quil_to_executable(nq_program)	Compile a native quil program to a binary executable.
quil_to_native_quil(program)	Compile an arbitrary quil program according to the ISA of target_device.

1.19.3 pyquil.api.QPUCompiler

class pyquil.api.**QPUCompiler**(*quilc_endpoint*, *qpu_compiler_endpoint*, *device*, *timeout=10*, *name=None*)

Client to communicate with the Compiler Server.

Parameters

- **quilc_endpoint** (str) – TCP or IPC endpoint of the Quil Compiler (quilc)
- **qpu_compiler_endpoint** (Optional[str]) – TCP or IPC endpoint of the QPU Compiler
- **device** (*AbstractDevice*) – PyQuil Device object to use as compilation target
- **timeout** (int) – Number of seconds to wait for a response from the client.
- **name** (Optional[str]) – Name of the lattice being targeted

__init__(*quilc_endpoint*, *qpu_compiler_endpoint*, *device*, *timeout=10*, *name=None*)

Client to communicate with the Compiler Server.

Parameters

- **quilc_endpoint** (str) – TCP or IPC endpoint of the Quil Compiler (quilc)
- **qpu_compiler_endpoint** (Optional[str]) – TCP or IPC endpoint of the QPU Compiler
- **device** (*AbstractDevice*) – PyQuil Device object to use as compilation target
- **timeout** (int) – Number of seconds to wait for a response from the client.
- **name** (Optional[str]) – Name of the lattice being targeted

Return type None

Methods

__init__(quilc_endpoint, ... [, timeout, name])	Client to communicate with the Compiler Server.
get_version_info()	Return version information for this compiler and its dependencies.
native_quil_to_executable(nq_program)	Compile a native quil program to a binary executable.
quil_to_native_quil(program)	Compile an arbitrary quil program according to the ISA of target_device.

1.20 QAMs

An appropriate QAM is automatically created when using *get_qc()* and it is stored on the *QuantumComputer* object as the qam attribute.

The Quantum Abstract Machine (QAM) provides an abstract interface for running hybrid quantum/classical quil programs on either a Quantum Virtual Machine (QVM, a classical simulator) or a Quantum Processor Unit (QPU, a real quantum device).

QAM()	The platonic ideal of this class is as a generic interface describing how a classical computer interacts with a live quantum computer.
QPU(endpoint[, user])	A connection to the QPU.
QVM(connection[, noise_model, gate_noise, ...])	A virtual machine that classically emulates the execution of Quil programs.
PyQVM(n_qubits[, quantum_simulator_type, ...])	PyQuil's built-in Quil virtual machine.

1.20.1 QAM

class pyquil.api._qam.**QAM**

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.

1.20.2 QPU

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

A connection to the QPU.

Parameters

- **endpoint** (str) – Address to connect to the QPU server.
- **user** (str) – A string identifying who's running jobs.

1.20.3 QVM

class pyquil.api.**QVM**(*connection*, *noise_model=None*, *gate_noise=None*, *measurement_noise=None*, *random_seed=None*, *requires_executable=False*)

A virtual machine that classically emulates the execution of Quil programs.

Parameters

- **connection** (ForestConnection) – A connection to the Forest web API.
- **noise_model** – A noise model that describes noise to apply when emulating a program's execution.
- **gate_noise** – A list of three numbers [Px, Py, Pz] indicating the probability of an X, Y, or Z gate getting applied to each qubit after a gate application or reset. The default value of None indicates no noise.
- **measurement_noise** – A list of three numbers [Px, Py, Pz] indicating the probability of an X, Y, or Z gate getting applied before a measurement. The default value of None indicates no noise.
- **random_seed** – A seed for the QVM's random number generators. Either None (for an automatically generated seed) or a non-negative integer.
- **requires_executable** – Whether this QVM will refuse to run a Program and only accept the result of compiler.native_quil_to_executable(). Setting this to True better emulates the behavior of a QPU.

1.20.4 PyQVM

class pyquil.pyqvm.**PyQVM**(*n_qubits*, *quantum_simulator_type=None*, *seed=None*, *post_gate_noise_probabilities=None*)

PyQuil's built-in Quil virtual machine.

This class implements common control flow and plumbing and dispatches the "actual" work to quantum simulators like ReferenceWavefunctionSimulator, ReferenceDensitySimulator, and NumpyWavefunctionSimulator

Parameters

- **n_qubits** – The number of qubits. Typically this results in the allocation of a large ndarray, so be judicious.
- **quantum_simulator_type** (Optional[Type[AbstractQuantumSimulator]]) – A class that can be instantiated to handle the quantum aspects of this QVM. If not specified, the default will be either NumpyWavefunctionSimulator (no noise) or ReferenceDensitySimulator (noise)
- **post_gate_noise_probabilities** (Optional[Dict[str, float]]) – A specification of noise model given by probabilities of certain types of noise. The dictionary keys are from "relaxation", "dephasing", "depolarizing", "phase_flip", "bit_flip", and "bitphase_flip". WARNING: experimental. This interface will likely change.
- **seed** – An optional random seed for performing stochastic aspects of the QVM.

1.21 Devices

An appropriate Device is automatically created when using *get_qc()* and it is stored on the *QuantumComputer* object as the `device` attribute.

There are properties of real quantum computers that go beyond the quantum abstract machine (QAM) abstraction. Real devices have performance specs, limited ISAs, and restricted topologies. *AbstractDevice* provides an abstract interface for accessing properties of a real quantum device or for mocking out relevant properties for a more realistic QVM.

AbstractDevice	
Device(name, raw)	A device (quantum chip) that can accept programs.
NxDevice(topology)	A shim over the AbstractDevice API backed by a NetworkX graph.

1.21.1 pyquil.device.AbstractDevice

class pyquil.device.**AbstractDevice**

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

get_isa([oneq_type, twoq_type])	Construct an ISA suitable for targeting by compilation.
get_specs()	Construct a Specs object required by compilation
qubit_topology()	The connectivity of qubits in this device given as a NetworkX graph.
qubits()	A sorted list of qubits in the device topology.

1.21.2 pyquil.device.Device

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

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.

Parameters

- **name** – name of the device
- **raw** – raw JSON response from the server with additional information about this device.

__init__(*name*, *raw*)

Parameters

- **name** – name of the device
- **raw** – raw JSON response from the server with additional information about this device.

Methods

__init__(name, raw)	**param name** name of the device
get_isa([oneq_type, twoq_type])	Construct an ISA suitable for targeting by compilation.
get_specs()	Construct a Specs object required by compilation
qubit_topology()	The connectivity of qubits in this device given as a NetworkX graph.
qubits()	A sorted list of qubits in the device topology.

Attributes

isa

1.21.3 pyquil.device.NxDevice

class pyquil.device.**NxDevice**(*topology*)

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.

__init__(*topology*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(topology)	Initialize self.
edges()	**rtype** List[Tuple[int, int]]
get_isa([oneq_type, twoq_type])	Construct an ISA suitable for targeting by compilation.
get_specs()	Construct a Specs object required by compilation
qubit_topology()	The connectivity of qubits in this device given as a NetworkX graph.
qubits()	A sorted list of qubits in the device topology.

The data structures used are documented here

ISA	Basic Instruction Set Architecture specification.
Specs	Basic specifications for the device, such as gate fidelities and coherence times.

1.21.4 pyquil.device.ISA

class 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.

Create new instance of _ISA(qubits, edges)

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

count(value)	
from_dict(d)	Re-create the ISA from a dictionary representation.
index(value, [start, [stop]])	Raises ValueError if the value is not present.
to_dict()	Create a JSON-serializable representation of the ISA.

Attributes

edges	Alias for field number 1
qubits	Alias for field number 0

1.21.5 pyquil.device.Specs

class 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.

Create new instance of _Specs(qubits_specs, edges_specs)

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

T1s()	Get a dictionary of T1s (in seconds) from the specs, keyed by qubit index.
T2s()	Get a dictionary of T2s (in seconds) from the specs, keyed by qubit index.
count(value)	
f1QRBs()	Get a dictionary of single-qubit randomized benchmarking fidelities (normalized to unity) from the specs, keyed by qubit index.
fActiveResets()	Get a dictionary of single-qubit active reset fidelities (normalized to unity) from the specs, keyed by qubit index.
fBellStates()	Get a dictionary of two-qubit Bell state fidelities (normalized to unity) from the specs, keyed by targets (qubit-qubit pairs).
fCPHASEs()	Get a dictionary of CPHASE fidelities (normalized to unity) from the specs, keyed by targets (qubit-qubit pairs).

Continued on next page

Table 28 – continued from previous page

fCZ_std_errs()	Get a dictionary of the standard errors of the CZ fidelities from the specs, keyed by targets (qubit-qubit pairs).
fCZs()	Get a dictionary of CZ fidelities (normalized to unity) from the specs, keyed by targets (qubit-qubit pairs).
fROs()	Get a dictionary of single-qubit readout fidelities (normalized to unity) from the specs, keyed by qubit index.
from_dict(d)	Re-create the Specs from a dictionary representation.
index(value, [start, [stop]])	Raises ValueError if the value is not present.
to_dict()	Create a JSON-serializable representation of the device Specs.

Attributes

edges_specs	Alias for field number 1
qubits_specs	Alias for field number 0

1.21.6 Utility functions

pyquil.device.**isa_from_graph**(*graph*, *oneq_type='Xhalves'*, *twoq_type='CZ'*)

Generate an ISA object from a NetworkX graph.

Parameters

- **graph** (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'.

Return type *ISA*

pyquil.device.**specs_from_graph**(*graph*)

Generate a Specs object from a NetworkX graph with placeholder values for the actual specs.

Parameters **graph** (Graph) – The graph

pyquil.device.**isa_to_graph**(*isa*)

Construct a NetworkX qubit topology from an ISA object.

This discards information about supported gates.

Parameters **isa** (*ISA*) – The ISA.

Return type Graph

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]

1.22 Simulators

QVMs promise to behave like a real QPU. However, under-the-hood there is usually a simulation that has fewer constraints than a physical device. For example, in a wavefunction (or statevector) simulation, you can directly inspect amplitudes and probabilities.

`WavefunctionSimulator`([connection, random_seed])	A simulator that propagates a wavefunction representation of a quantum state.
`ReferenceWavefunctionSimulator`(n_qubits[, rs])	A wavefunction simulator that prioritizes readability over performance.
`ReferenceDensitySimulator`(n_qubits[, rs])	A density matrix simulator that prioritizes readability over performance.
`NumpyWavefunctionSimulator`(n_qubits[, rs])	A wavefunction simulator that uses numpy's tensordot or einsum to update a state vector

1.22.1 WavefunctionSimulator

class pyquil.api.**WavefunctionSimulator**(*connection=None*, *random_seed=None*)

A simulator that propagates a wavefunction representation of a quantum state.

Parameters

- **connection** (Optional[ForestConnection]) – A connection to the Forest web API.
- **random_seed** (Optional[int]) – A seed for the simulator's random number generators. Either None (for an automatically generated seed) or a non-negative integer.

1.22.2 ReferenceWavefunctionSimulator

class pyquil.reference_simulator.**ReferenceWavefunctionSimulator**(*n_qubits*, *rs=None*)

A wavefunction simulator that prioritizes readability over performance.

Please consider using `PyQVM(..., wf_simulator_type=ReferenceWavefunctionSimulator)` rather than using this class directly.

This class uses a flat state-vector of length 2^n_qubits to store wavefunction amplitudes. The basis is taken to be bitstrings ordered lexicographically with qubit 0 as the rightmost bit. This is the same as the Rigetti Lisp QVM.

Parameters

- **n_qubits** (int) – Number of qubits to simulate.
- **rs** (Optional[RandomState]) – a RandomState (should be shared with the owning PyQVM) for doing anything stochastic. A value of None disallows doing anything stochastic.

1.22.3 ReferenceDensitySimulator

class pyquil.reference_simulator.**ReferenceDensitySimulator**(*n_qubits*, *rs=None*)

A density matrix simulator that prioritizes readability over performance.

Please consider using `PyQVM(..., wf_simulator_type=ReferenceDensitySimulator)` rather than using this class directly.

This class uses a dense matrix of shape `(2^n_qubits, 2^n_qubits)` to store the density matrix.

Parameters

- **n_qubits** (`int`) – Number of qubits to simulate.
- **rs** (`Optional[RandomState]`) – a RandomState (should be shared with the owning `PyQVM`) for doing anything stochastic. A value of `None` disallows doing anything stochastic.

1.22.4 NumpyWavefunctionSimulator

class pyquil.numpy_simulator.**NumpyWavefunctionSimulator**(*n_qubits*, *rs=None*)

A wavefunction simulator that uses numpy's tensordot or einsum to update a state vector

Please consider using `PyQVM(..., quantum_simulator_type=NumpyWavefunctionSimulator)` rather than using this class directly.

This class uses a n_qubit-dim ndarray to store wavefunction amplitudes. The array is indexed into with a tuple of n_qubits 1's and 0's, with qubit 0 as the leftmost bit. This is the opposite convention of the Rigetti Lisp QVM.

Parameters

- **n_qubits** – Number of qubits to simulate.
- **rs** (`Optional[RandomState]`) – a RandomState (should be shared with the owning `PyQVM`) for doing anything stochastic. A value of `None` disallows doing anything stochastic.

1.22.5 Reference Utilities

lifted_pauli(pauli_sum, qubits)	Takes a PauliSum object along with a list of qubits and returns a matrix corresponding the tensor representation of the object.
lifted_gate(gate, n_qubits)	Lift a pyquil `Gate` in a full `n_qubits`-qubit Hilbert space.
program_unitary(program, n_qubits)	Return the unitary of a pyQuil program.
all_bitstrings(n_bits)	All bitstrings in lexicographical order as a 2d np.ndarray.

lifted_pauli

pyquil.unitary_tools.**lifted_pauli**(*pauli_sum*, *qubits*)

Takes a PauliSum object along with a list of qubits and returns a matrix corresponding the tensor representation of the object.

Useful for generating the full Hamiltonian after a particular fermion to pauli transformation. For example:

Converting a PauliSum X0Y1 + Y1X0 into the matrix

```
[[ 0.+0.j,  0.+0.j,  0.+0.j,  0.-2.j],
 [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
 [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
 [ 0.+2.j,  0.+0.j,  0.+0.j,  0.+0.j]]
```

Developer note: Quil and the QVM like qubits to be ordered such that qubit 0 is on the right. Therefore,

in `qubit_adjacent_lifted_gate`, `lifted_pauli`, and `lifted_state_operator`, we build up the lifted matrix by performing the kronecker product from right to left.

Parameters

- **pauli_sum** (`Union[PauliSum, PauliTerm]`) – Pauli representation of an operator
- **qubits** (`List[int]`) – list of qubits in the order they will be represented in the resultant matrix.

Returns matrix representation of the pauli_sum operator

lifted_gate

`pyquil.unitary_tools.`**lifted_gate**(*gate*, *n_qubits*)

Lift a pyquil `Gate` in a full `n_qubits`-qubit Hilbert space.

This function looks up the matrix form of the gate and then dispatches to `lifted_gate_matrix()` with the target qubits.

Parameters

- **gate** (`Gate`) – A gate
- **n_qubits** (`int`) – The total number of qubits.

Returns A 2^n by 2^n lifted version of the gate acting on its specified qubits.

program_unitary

`pyquil.unitary_tools.`**program_unitary**(*program*, *n_qubits*)

Return the unitary of a pyQuil program.

Parameters **program** – A program consisting only of `Gate`.:

Returns a unitary corresponding to the composition of the program's gates.

all_bitstrings

`pyquil.unitary_tools.`**all_bitstrings**(*n_bits*)

All bitstrings in lexicographical order as a 2d np.ndarray.

This should be the same as `np.array(list(itertools.product([0,1], repeat=n_bits)))` but faster.

1.22.6 Numpy Utilities

targeted_einsum(gate, wf, wf_target_inds)	Left-multiplies the given axes of the wf tensor by the given gate matrix.
targeted_tensordot(gate, wf, wf_target_inds)	Left-multiplies the given axes of the wf tensor by the given gate matrix.

targeted_einsum

`pyquil.numpy_simulator.`**targeted_einsum**(*gate*, *wf*, *wf_target_inds*)

Left-multiplies the given axes of the wf tensor by the given gate matrix.

Note that the matrix must have a compatible tensor structure. For example, if you have an 6-qubit state vector `wf` with shape (2, 2, 2, 2, 2, 2), and a 2-qubit unitary operation `op` with shape (2, 2, 2, 2), and you want to apply `op` to the 5th and 3rd qubits within `input_state`, then the output state vector is computed as follows:

```
output_state = targeted_einsum(op, input_state, [5, 3])
```

This method also works when the right hand side is a matrix instead of a vector. If a unitary circuit's matrix is `old_effect`, and you append a CNOT(q1, q4) operation onto the circuit, where the control q1 is the qubit at offset 1 and the target q4 is the qubit at offset 4, then the appended circuit's unitary matrix is computed as follows:

```
new_effect = targeted_left_multiply(CNOT.reshape((2, 2, 2, 2)), old_effect, [1,␣
↪4])
```

Parameters

- **gate** (ndarray) – What to left-multiply the target tensor by.
- **wf** (ndarray) – A tensor to carefully broadcast a left-multiply over.
- **wf_target_inds** (List[int]) – Which axes of the target are being operated on.

Return type ndarray

Returns The output tensor.

targeted_tensordot

pyquil.numpy_simulator.**targeted_tensordot**(*gate*, *wf*, *wf_target_inds*)

Left-multiplies the given axes of the wf tensor by the given gate matrix.

Compare with *targeted_einsum()*. The semantics of these two functions should be identical, except this uses `np.tensordot` instead of `np.einsum`.

Parameters

- **gate** (ndarray) – What to left-multiply the target tensor by.
- **wf** (ndarray) – A tensor to carefully broadcast a left-multiply over.
- **wf_target_inds** (Sequence[int]) – Which axes of the target are being operated on.

Return type ndarray

Returns The output tensor.

1.23 Noise Models

1.23.1 Functions

append_kraus_to_gate(kraus_ops, gate_matrix)	Follow a gate `gate_matrix` by a Kraus map described by `kraus_ops`.
pauli_kraus_map(probabilities)	Generate the Kraus operators corresponding to a pauli channel.
damping_kraus_map([p])	Generate the Kraus operators corresponding to an amplitude damping noise channel.

Continued on next page

Table 33 – continued from previous page

dephasing_kraus_map([p])	Generate the Kraus operators corresponding to a dephasing channel.
tensor_kraus_maps(k1, k2)	Generate the Kraus map corresponding to the composition of two maps on different qubits.
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.
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.
get_noisy_gate(gate_name, params)	Look up the numerical gate representation and a proposed 'noisy' name.
_decoherence_noise_model(gates[, T1, T2, ...])	The default noise parameters
decoherence_noise_with_asymmetric_ro(gates)	Similar to *_decoherence_noise_model()*, but with asymmetric readout.
apply_noise_model(prog, noise_model)	Apply a noise model to a program and generated a 'noisy-fied' version of the program.
add_decoherence_noise(prog[, T1, T2, ...])	Add generic damping and dephasing noise to a program.
estimate_bitstring_probs(results)	Given an array of single shot results estimate the probability distribution over all bitstrings.
corrupt_bitstring_probs(p, ...)	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.
bitstring_probs_to_z_moments(p)	Convert between bitstring probabilities and joint Z moment expectations.
estimate_assignment_probs(q, trials, cxn[, p0])	Estimate the readout assignment probabilities for a given qubit `q`.

append_kraus_to_gate

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.

pauli_kraus_map

pyquil.noise.**pauli_kraus_map**(*probabilities*)

Generate the Kraus operators corresponding to a pauli channel.

Params list|floats probabilities The 4^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^num_qubits Kraus operators that parametrize the map.

Return type list

damping_kraus_map

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

dephasing_kraus_map

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

tensor_kraus_maps

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.

combine_kraus_maps

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.

damping_after_dephasing

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.

get_noisy_gate

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]

_decoherence_noise_model

pyquil.noise.**_decoherence_noise_model**(*gates*, *T1=3e-05*, *T2=3e-05*, *gate_time_1q=5e-08*, *gate_time_2q=1.5e-07*, *ro_fidelity=0.95*)

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

- **gates** (*Sequence[Gate]*) – The gates to provide the noise model for.
- **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 NoiseModel with the appropriate Kraus operators defined.

decoherence_noise_with_asymmetric_ro

pyquil.noise.**decoherence_noise_with_asymmetric_ro**(*gates*, *p00=0.975*, *p11=0.911*)

Similar to *_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.

apply_noise_model

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*

add_decoherence_noise

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.

estimate_bitstring_probs

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

corrupt_bitstring_probs

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

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

Parameters

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

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

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

Return type np.array

bitstring_probs_to_z_moments

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

Convert between bitstring probabilities and joint Z moment expectations.

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

Returns

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

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

Return type np.array

estimate_assignment_probs

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

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

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

Parameters

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

Returns The assignment probability matrix

Return type np.array

1.23.2 Classes

class 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.

Create new instance of _NoiseModel(gates, assignment_probs)

Methods

gates_by_name(name)	Return all defined noisy gates of a particular gate name.
to_dict()	Create a JSON serializable representation of the noise model.
from_dict(d)	Re-create the noise model from a dictionary representation.

NoiseModel.gates_by_name

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*]

NoiseModel.to_dict

NoiseModel.**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]

NoiseModel.from_dict

static NoiseModel.**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*

class 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.

Create new instance of _KrausModel(gate, params, targets, kraus_ops, fidelity)

Methods

unpack_kraus_matrix(m)	Helper to optionally unpack a JSON compatible representation of a complex Kraus matrix.
unpack_kraus_matrix(m)	Helper to optionally unpack a JSON compatible representation of a complex Kraus matrix.
to_dict()	Create a dictionary representation of a KrausModel.
from_dict(d)	Recreate a KrausModel from the dictionary representation.

KrausModel.unpack_kraus_matrix

static KrausModel.**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

KrausModel.to_dict

KrausModel.**to_dict**()

Create a dictionary representation of a KrausModel.

For example:

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

(continues on next page)

(continued from previous page)

```
            [[0, -1],           # element-wise imaginary part of matrix
            [-1, 0]]]
    ],
    "fidelity": 1.0
}
```

Returns A JSON compatible dictionary representation.

Return type Dict[str,Any]

KrausModel.from_dict

static KrausModel.**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*

1.24 Operator Estimation

1.24.1 Data structures

ExperimentSetting(in_state, out_operator)	Input and output settings for a tomography-like experiment.
TomographyExperiment(settings, program, qubits)	A tomography-like experiment.
ExperimentResult(setting, expectation, ...)	An expectation and standard deviation for the measurement of one experiment setting in a tomographic experiment.

ExperimentSetting

class pyquil.operator_estimation.**ExperimentSetting**(*in_state*, *out_operator*)

Input and output settings for a tomography-like experiment.

Many near-term quantum algorithms take the following form:

- Start in a pauli state
- Prepare some ansatz
- Measure it w.r.t. pauli operators

Where we typically use a large number of (start, measure) pairs but keep the ansatz preparation program consistent. This class represents the (start, measure) pairs. Typically a large number of these *ExperimentSetting* objects will be created and grouped into a *TomographyExperiment*.

TomographyExperiment

class pyquil.operator_estimation.**TomographyExperiment**(*settings*, *program*, *qubits*)
A tomography-like experiment.

Many near-term quantum algorithms involve:

- some limited state preparation
- enacting a quantum process (like in tomography) or preparing a variational ansatz state (like in VQE)
- measuring observables of the state.

Where we typically use a large number of (state_prep, measure) pairs but keep the ansatz program consistent. This class stores the ansatz program as a `Program` and maintains a list of *ExperimentSetting* objects which each represent a (state_prep, measure) pair.

Settings diagonalized by a shared tensor product basis (TPB) can (optionally) be estimated simultaneously. Therefore, this class is backed by a list of list of ExperimentSettings. Settings sharing an inner list will be estimated simultaneously. If you don't want this, provide a list of length-1-lists. As a convenience, if you pass a 1D list to the constructor will expand it to a list of length-1-lists.

This class will not group settings for you. Please see *group_experiments()* for a function that will automatically process a TomographyExperiment to group Experiments sharing a TPB.

ExperimentResult

class pyquil.operator_estimation.**ExperimentResult**(*setting*, *expectation*, *stddev*, *total_counts*, *raw_expectation=None*, *raw_stddev=None*, *calibration_expectation=None*, *calibration_stddev=None*, *calibration_counts=None*)
An expectation and standard deviation for the measurement of one experiment setting in a tomographic experiment.

In the case of readout error calibration, we also include expectation, standard deviation and count for the calibration results, as well as the expectation and standard deviation for the corrected results.

1.24.2 Methods

group_experiments(experiments[, method])	Group experiments that are diagonal in a shared tensor product basis (TPB) to minimize number of QPU runs.
measure_observables(qc, tomo_experiment[, ...])	Measure all the observables in a TomographyExperiment.

group_experiments

pyquil.operator_estimation.**group_experiments**(*experiments*, *method='greedy'*)
Group experiments that are diagonal in a shared tensor product basis (TPB) to minimize number of QPU runs.

Given some PauliTerm operator, the 'natural' tensor product basis to diagonalize this term is the one which diagonalizes each Pauli operator in the product term-by-term.

For example, X(1) * Z(0) would be diagonal in the 'natural' tensor product basis {(|0> +/- |1>)/Sqrt[2]} * {|0>, |1>}, whereas Z(1) * X(0) would be diagonal in the 'natural' tpb {|0>, |1>} * {(|0> +/- |1>)/Sqrt[2]}. The two

operators commute but are not diagonal in each others 'natural' tpb (in fact, they are anti-diagonal in each others 'natural' tpb). This function tests whether two operators given as PauliTerms are both diagonal in each others 'natural' tpb.

Note that for the given example of X(1) * Z(0) and Z(1) * X(0), we can construct the following basis which simultaneously diagonalizes both operators:

– |0>' = |0> (|+>) + |1> (|->) – |1>' = |0> (|+>) - |1> (|->) – |2>' = |0> (|->) + |1> (|+>) – |3>' = |0> (-|->) + |1> (|+>)

In this basis, X Z looks like diag(1, -1, 1, -1), and Z X looks like diag(1, 1, -1, -1). Notice however that this basis cannot be constructed with single-qubit operations, as each of the basis vectors are entangled states.

The "greedy" method will keep a running set of 'buckets' into which grouped ExperimentSettings will be placed. Each new ExperimentSetting considered is assigned to the first applicable bucket and a new bucket is created if there are no applicable buckets.

The "clique-removal" method maps the term grouping problem onto Max Clique graph problem. This method constructs a NetworkX graph where an edge exists between two settings that share an nTPB and then uses networkx's algorithm for clique removal. This method can give you marginally better groupings in certain circumstances, but constructing the graph is pretty slow so "greedy" is the default.

Parameters

- **experiments** (*TomographyExperiment*) – a tomography experiment
- **method** (str) – method used for grouping; the allowed methods are one of ['greedy', 'clique-removal']

Return type *TomographyExperiment*

Returns a tomography experiment with all the same settings, just grouped according to shared TPBs.

measure_observables

pyquil.operator_estimation.**measure_observables**(*qc*, *tomo_experiment*, *n_shots=10000*, *progress_callback=None*, *active_reset=False*, *readout_symmetrize='exhaustive'*, *calibrate_readout='plus-eig'*)

Measure all the observables in a TomographyExperiment.

Parameters

- **qc** (QuantumComputer) – A QuantumComputer which can run quantum programs
- **tomo_experiment** (*TomographyExperiment*) – A suite of tomographic observables to measure
- **n_shots** (int) – The number of shots to take per ExperimentSetting
- **progress_callback** – If not None, this function is called each time a group of settings is run with arguments `f(i, len(tomo_experiment)` such that the progress is `i / len(tomo_experiment)`.
- **active_reset** – Whether to actively reset qubits instead of waiting several times the coherence length for qubits to decay to |0> naturally. Setting this to True is much faster but there is a ~1% error per qubit in the reset operation. Thermal noise from "traditional" reset is not routinely characterized but is of the same order.
- **readout_symmetrize** (str) – Method used to symmetrize the readout errors, i.e. set p(0|1) = p(1|0). For uncorrelated readout errors, this can be achieved by randomly selecting

between the POVMs {X.D1.X, X.D0.X} and {D0, D1} (where both D0 and D1 are diagonal). However, here we currently support exhaustive symmetrization and loop through all possible 2^n POVMs {X/I . POVM . X/I}^n, and obtain symmetrization more generally, i.e. set p(00|00) = p(01|01) = .. = p(11|11), as well as p(00|01) = p(01|00) etc. If this is None, no symmetrization is performed. The exhaustive method can be specified by setting this variable to 'exhaustive' (default value). Set to *None* if no symmetrization is desired.

- **calibrate_readout** (str) – Method used to calibrate the readout results. Currently, the only method supported is normalizing against the operator's expectation value in its +1 eigenstate, which can be specified by setting this variable to 'plus-eig' (default value). The preceding symmetrization and this step together yield a more accurate estimation of the observable. Set to *None* if no calibration is desired.

1.24.3 Utilities

to_json(fn, obj)	Convenience method to save pyquil.operator_estimation objects as a JSON file.
read_json(fn)	Convenience method to read pyquil.operator_estimation objects from a JSON file.

to_json

pyquil.operator_estimation.**to_json**(*fn*, *obj*)

Convenience method to save pyquil.operator_estimation objects as a JSON file.

See *read_json()*.

read_json

pyquil.operator_estimation.**read_json**(*fn*)

Convenience method to read pyquil.operator_estimation objects from a JSON file.

See *to_json()*.

CHAPTER 2

Indices and Tables

- genindex
- modindex
- search

Bibliography

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

Index

Symbols

__add__() (*pyquil.quil.Program method*), 113
__getitem__() (*pyquil.quil.Program method*), 118
__iadd__() (*pyquil.quil.Program method*), 113
__init__() (*pyquil.api.QPUCompiler method*), 148
__init__() (*pyquil.api.QVMCompiler method*), 147
__init__() (*pyquil.api._qac.AbstractCompiler method*), 147
__init__() (*pyquil.device.AbstractDevice method*), 150
__init__() (*pyquil.device.Device method*), 151
__init__() (*pyquil.device.ISA method*), 152
__init__() (*pyquil.device.NxDevice method*), 152
__init__() (*pyquil.device.Specs method*), 153
_decoherence_noise_model() (*in module pyquil.noise*), 161

A

AbstractCompiler (*class in pyquil.api._qac*), 147
AbstractDevice (*class in pyquil.device*), 150
ADD() (*in module pyquil.gates*), 133
add_decoherence_noise() (*in module pyquil.noise*), 162
address_qubits() (*in module pyquil.quil*), 119
all_bitstrings() (*in module pyquil.unitary_tools*), 157
alloc() (*pyquil.quil.Program method*), 116
AND() (*in module pyquil.gates*), 131
append_kraus_to_gate() (*in module pyquil.noise*), 159
apply_noise_model() (*in module pyquil.noise*), 162

B

bitstring_probs_to_z_moments() (*in module pyquil.noise*), 164

C

CCNOT() (*in module pyquil.gates*), 126
check_commutation() (*in module pyquil.paulis*), 137
CNOT() (*in module pyquil.gates*), 126
combine_kraus_maps() (*in module pyquil.noise*), 160
commuting_sets() (*in module pyquil.paulis*), 137
compile() (*pyquil.api.QuantumComputer method*), 146
CONVERT() (*in module pyquil.gates*), 133
copy() (*pyquil.paulis.PauliTerm method*), 141
copy() (*pyquil.quil.Program method*), 118
corrupt_bitstring_probs() (*in module pyquil.noise*), 163
CPHASE() (*in module pyquil.gates*), 127
CPHASE00() (*in module pyquil.gates*), 126
CPHASE01() (*in module pyquil.gates*), 127
CPHASE10() (*in module pyquil.gates*), 127
CSWAP() (*in module pyquil.gates*), 128
CZ() (*in module pyquil.gates*), 122

D

dagger() (*pyquil.quil.Program method*), 118
damping_after_dephasing() (*in module pyquil.noise*), 161
damping_kraus_map() (*in module pyquil.noise*), 160
declare() (*pyquil.quil.Program method*), 116
decoherence_noise_with_asymmetric_ro() (*in module pyquil.noise*), 162
defgate() (*pyquil.quil.Program method*), 114
define_noisy_gate() (*pyquil.quil.Program method*), 114
define_noisy_readout() (*pyquil.quil.Program method*), 115
defined_gates (*pyquil.quil.Program attribute*), 111
dephasing_kraus_map() (*in module pyquil.noise*), 160
Device (*class in pyquil.device*), 151
DIV() (*in module pyquil.gates*), 134

E

EQ() (*in module pyquil.gates*), 134
estimate_assignment_probs() (*in module pyquil.noise*), 164
estimate_bitstring_probs() (*in module pyquil.noise*), 163
EXCHANGE() (*in module pyquil.gates*), 132
ExperimentResult (*class in pyquil.operator_estimation*), 168
ExperimentSetting (*class in pyquil.operator_estimation*), 167
exponential_map() (*in module pyquil.paulis*), 138
exponentiate() (*in module pyquil.paulis*), 138
exponentiate_commuting_pauli_sum() (*in module pyquil.paulis*), 139

F

from_dict() (*pyquil.noise.KrausModel static method*), 167
from_dict() (*pyquil.noise.NoiseModel static method*), 165
from_list() (*pyquil.paulis.PauliTerm class method*), 141

G

gate() (*pyquil.quil.Program method*), 114
gates_by_name() (*pyquil.noise.NoiseModel method*), 165
gates_in_isa() (*in module pyquil.device*), 154
GE() (*in module pyquil.gates*), 135
get_classical_addresses_from_program() (*in module pyquil.quil*), 120
get_default_qubit_mapping() (*in module pyquil.quil*), 119
get_isa() (*pyquil.api.QuantumComputer method*), 146
get_noisy_gate() (*in module pyquil.noise*), 161
get_programs() (*pyquil.paulis.PauliSum method*), 140
get_qc() (*in module pyquil*), 142
get_qubits() (*pyquil.paulis.PauliSum method*), 140
get_qubits() (*pyquil.quil.Program method*), 112
group_experiments() (*in module pyquil.operator_estimation*), 168
GT() (*in module pyquil.gates*), 135

H

H() (*in module pyquil.gates*), 124
HALT (*in module pyquil.gates*), 130

I

I() (*in module pyquil.gates*), 123
ID() (*in module pyquil.paulis*), 136
id() (*pyquil.paulis.PauliTerm method*), 141
if_then() (*pyquil.quil.Program method*), 117
implicitly_declare_ro() (*in module pyquil.quil*), 119
inst() (*pyquil.quil.Program method*), 113
instantiate_labels() (*in module pyquil.quil*), 119
instructions (*pyquil.quil.Program attribute*), 111
IOR() (*in module pyquil.gates*), 131
is_identity() (*in module pyquil.paulis*), 138
is_protoquil() (*pyquil.quil.Program method*), 112
is_zero() (*in module pyquil.paulis*), 138
ISA (*class in pyquil.device*), 152
isa_from_graph() (*in module pyquil.device*), 154
isa_to_graph() (*in module pyquil.device*), 154
ISWAP() (*in module pyquil.gates*), 129

K

KrausModel (*class in pyquil.noise*), 165

L

LE() (*in module pyquil.gates*), 134
lifted_gate() (*in module pyquil.unitary_tools*), 157
lifted_pauli() (*in module pyquil.unitary_tools*), 156
list_quantum_computers() (*in module pyquil*), 143
LOAD() (*in module pyquil.gates*), 132
LT() (*in module pyquil.gates*), 134

M

MEASURE() (*in module pyquil.gates*), 122
measure() (*pyquil.quil.Program method*), 115
measure_all() (*pyquil.quil.Program method*), 116
measure_observables() (*in module pyquil.operator_estimation*), 169
merge_programs() (*in module pyquil.quil*), 120
merge_with_pauli_noise() (*in module pyquil.quil*), 119
MOVE() (*in module pyquil.gates*), 132
MUL() (*in module pyquil.gates*), 133

N

NEG() (*in module pyquil.gates*), 131
no_noise() (*pyquil.quil.Program method*), 115
NoiseModel (*class in pyquil.noise*), 164
NOP (*in module pyquil.gates*), 130
NOT() (*in module pyquil.gates*), 131
NumpyWavefunctionSimulator (*class in pyquil.numpy_simulator*), 156
NxDevice (*class in pyquil.device*), 152

O

operations_as_set() (*pyquil.paulis.PauliTerm method*), 141
OR() (*in module pyquil.gates*), 131
out() (*pyquil.quil.Program method*), 111

P

parse() (*in module pyquil.parser*), 120
parse_program() (*in module pyquil.parser*), 120
pauli_kraus_map() (*in module pyquil.noise*), 159
pauli_string() (*pyquil.paulis.PauliTerm method*), 141
PauliSum (*class in pyquil.paulis*), 139
PauliTerm (*class in pyquil.paulis*), 140
percolate_declares() (*in module pyquil.quil*), 120
PHASE() (*in module pyquil.gates*), 125
pop() (*pyquil.quil.Program method*), 118
Program (*class in pyquil.quil*), 111
program (*pyquil.paulis.PauliTerm attribute*), 141
program_unitary() (*in module pyquil.unitary_tools*), 157
PSWAP() (*in module pyquil.gates*), 129
PyQVM (*class in pyquil.pyqvm*), 150

Q

QAM (*class in pyquil.api._qam*), 149
QPU (*class in pyquil.api*), 149
QPUCompiler (*class in pyquil.api*), 148
QuantumComputer (*class in pyquil.api*), 143
qubit_topology() (*pyquil.api.QuantumComputer method*), 146
qubits() (*pyquil.api.QuantumComputer method*), 145
QVM (*class in pyquil.api*), 149
QVMCompiler (*class in pyquil.api*), 147

R

read_json() (*in module pyquil.operator_estimation*), 170
ReferenceDensitySimulator (*class in pyquil.reference_simulator*), 155
ReferenceWavefunctionSimulator (*class in pyquil.reference_simulator*), 155
RESET() (*in module pyquil.gates*), 130
reset() (*pyquil.quil.Program method*), 116
run() (*pyquil.api.QuantumComputer method*), 144
run_and_measure() (*pyquil.api.QuantumComputer method*), 144
run_symmetrized_readout() (*pyquil.api.QuantumComputer method*), 145
RX() (*in module pyquil.gates*), 121
RY() (*in module pyquil.gates*), 125
RZ() (*in module pyquil.gates*), 121

S

S() (*in module pyquil.gates*), 124
sI() (*in module pyquil.paulis*), 136
simplify() (*pyquil.paulis.PauliSum method*), 140
simplify_pauli_sum() (*in module pyquil.paulis*), 137
Specs (*class in pyquil.device*), 153
specs_from_graph() (*in module pyquil.device*), 154
STORE() (*in module pyquil.gates*), 132
SUB() (*in module pyquil.gates*), 133
suzuki_trotter() (*in module pyquil.paulis*), 139
SWAP() (*in module pyquil.gates*), 128
sX() (*in module pyquil.paulis*), 136
sY() (*in module pyquil.paulis*), 136
sZ() (*in module pyquil.paulis*), 136

T

T() (*in module pyquil.gates*), 124
targeted_einsum() (*in module pyquil.numpy_simulator*), 157
targeted_tensordot() (*in module pyquil.numpy_simulator*), 158
tensor_kraus_maps() (*in module pyquil.noise*), 160
to_dict() (*pyquil.noise.KrausModel method*), 166
to_dict() (*pyquil.noise.NoiseModel method*), 165
to_json() (*in module pyquil.operator_estimation*), 170
TomographyExperiment (*class in pyquil.operator_estimation*), 168
trotterize() (*in module pyquil.paulis*), 139

U

unpack_kraus_matrix() (*pyquil.noise.KrausModel static method*), 166

V

validate_protoquil() (*in module pyquil.quil*), 120

W

WAIT (*in module pyquil.gates*), 130
WavefunctionSimulator (*class in pyquil.api*), 155
while_do() (*pyquil.quil.Program method*), 117
wrap_in_numshots_loop() (*pyquil.quil.Program method*), 117

X

X() (*in module pyquil.gates*), 123

XOR() (*in module pyquil.gates*), 132

Y

Y() (*in module pyquil.gates*), 123

Z

Z() (*in module pyquil.gates*), 124
ZERO() (*in module pyquil.paulis*), 137