Grove Documentation

Release 1.6.0

Rigetti Quantum Computing

Jun 27, 2018

Contents

Grove is a open source Python library containing quantum algorithms that uses the quantum programming library pyQuil and the Rigetti Forest toolkit.

CHAPTER 1

Structure

Grove is organized into modules for the various quantum algorithms, each of which has its own self-contained documentation.

1.1 Installation and Getting Started

1.1.1 Prerequisites

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

1.1.2 Installation

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

```
pip install quantum-grove
```

To instead install the bleeding-edge version from source, clone the Grove GitHub repository, `cd` into it, and run:

```
pip install -e .
```

This will install Grove's dependencies if you do not already have them. The dependencies are:

- NumPy
- SciPy
- NetworkX
- Matplotlib
- pytest *(optional, for testing)*

- mock *(optional, for testing)*

1.1.3 Forest and pyQuil

Grove also requires the Python library for Quil, called pyQuil.

After obtaining the library from the pyQuil GitHub repository or from a source distribution, navigate into its directory in a terminal and run:

```
pip install -e .
```

You will need to make sure that your pyQuil installation is properly configured to run with a QVM or quantum processor (QPU) hosted on the Rigetti Forest, which requires an API key. See the pyQuil docs for instructions on how to do this.

1.2 Variational-Quantum-Eigensolver (VQE)

1.2.1 Overview

The Variational-Quantum-Eigensolver (VQE) [1, 2] is a quantum/classical hybrid algorithm that can be used to find eigenvalues of a (often large) matrix H. When this algorithm is used in quantum simulations, H is typically the Hamiltonian of some system [3, 4, 5]. In this hybrid algorithm a quantum subroutine is run inside of a classical optimization loop.

The quantum subroutine has two fundamental steps:

1. Prepare the quantum state $|\Psi(\text{vec}\,(\theta))\rangle$, often called the *ansatz*.
2. Measure the expectation value $\langle\, \Psi(\text{vec}\,(\theta)) \,|\, H \,|\, \Psi(\text{vec}\,(\theta)) \,\rangle$.

The variational principle ensures that this expectation value is always greater than the smallest eigenvalue of H.

This bound allows us to use classical computation to run an optimization loop to find this eigenvalue:

1. Use a classical non-linear optimizer to minimize the expectation value by varying ansatz parameters vec (θ).
2. Iterate until convergence.

Practically, the quantum subroutine of VQE amounts to preparing a state based off of a set of parameters vec (θ) and performing a series of measurements in the appropriate basis. The paramaterized state (or ansatz) preparation can be tricky in these algorithms and can dramatically affect performance. Our `VQE` module allows any Python function that returns a pyQuil program to be used as an ansatz generator. This function is passed into `vqe_run` as the `variational_state_evolve` argument. More details are in the source documentation.

Measurements are then performed on these states based on a Pauli operator decomposition of H. Using Quil, these measurements will end up in classical memory. Doing this iteratively followed by a small amount of postprocessing, one may compute a real expectation value for the classical optimizer to use.

Below there is a very small first example of VQE and Grove's implementation of the Quantum Approximate Optimization Algorithm QAOA also makes use of the VQE module.

1.2.2 Basic Usage

Here we will take you through an example of a very small variational quantum eigensolver problem. In this example we will use a quantum circuit that consists of a single parametrized gate to calculate an eigenvalue of the Pauli Z matrix.

First we import the necessary pyQuil modules to construct our ansatz pyQuil program.

```
from pyquil.quil import Program
import pyquil.api as api
from pyquil.gates import *
qvm = api.QVMConnection()
```

Any Python function that takes a list of numeric parameters and outputs a pyQuil program can be used as an ansatz function. We will see some more examples of this later. For now, we just take a parameter list with a single parameter.

```
def small_ansatz(params):
    return Program(RX(params[0], 0))

print(small_ansatz([1.0]))
```

```
RX(1.0) 0
```

This `small_ansatz` function is our $\Psi(\text{vec}\,(\theta))$. To construct the Hamiltonian that we wish to simulate, we use the `pyquil.paulis` module.

```
from pyquil.paulis import sZ
initial_angle = [0.0]
# Our Hamiltonian is just \sigma_z on the zeroth qubit
hamiltonian = sZ(0)
```

We now use the *vqe* module in Grove to construct a `VQE` object to perform our algorithm. In this example, we use `scipy.optimize.minimize()` with Nelder-Mead as our classical minimizer, but you can choose other parameters or write your own minimizer.

```
from grove.pyvqe.vqe import VQE
from scipy.optimize import minimize
import numpy as np

vqe_inst = VQE(minimizer=minimize,
               minimizer_kwargs={'method': 'nelder-mead'})
```

Before we run the minimizer, let us look manually at what expectation values $\langle\, \Psi(\text{vec}\,(\theta)) \,|\, H \,|\, \Psi(\text{vec}\,(\theta)) \,\rangle$ we calculate for fixed parameters of $\text{vec}\,(\theta)$.

```
angle = 2.0
vqe_inst.expectation(small_ansatz([angle]), hamiltonian, None, qvm)
```

```
-0.4161468365471423
```

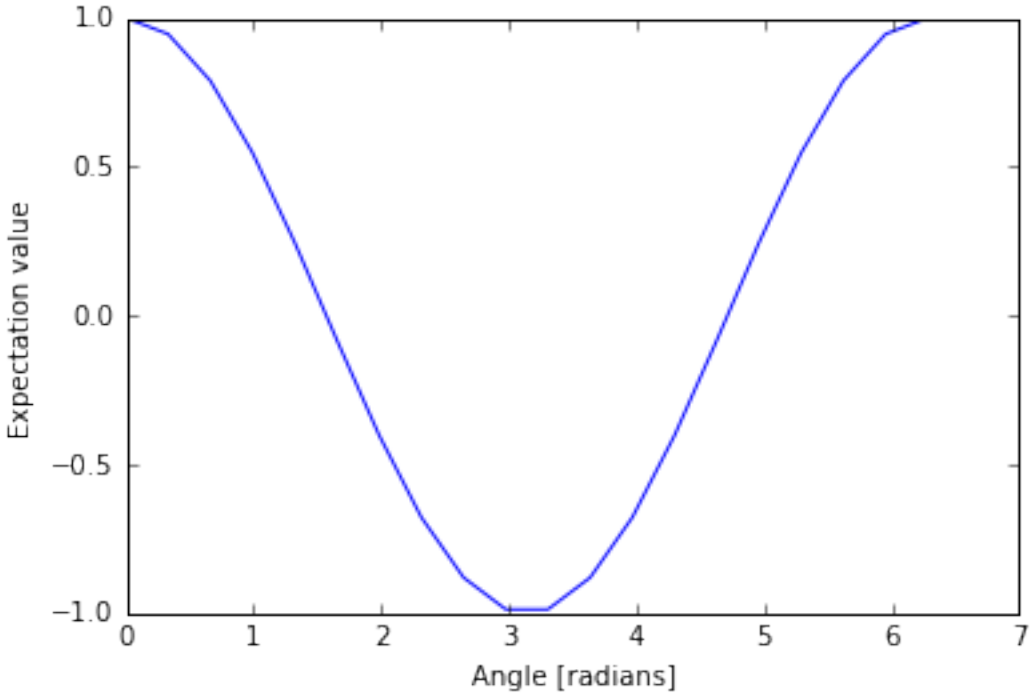
The expectation value was calculated by running the pyQuil program output from `small_ansatz`, saving the wavefunction, and using that vector to calculate the expectation value. We can sample the wavefunction as you would on a quantum computer by passing an integer, instead of *None*, as the samples argument of the `expectation()` method.

```
angle = 2.0
vqe_inst.expectation(small_ansatz([angle]), hamiltonian, 10000, qvm)
```

```
-0.42900000000000005
```

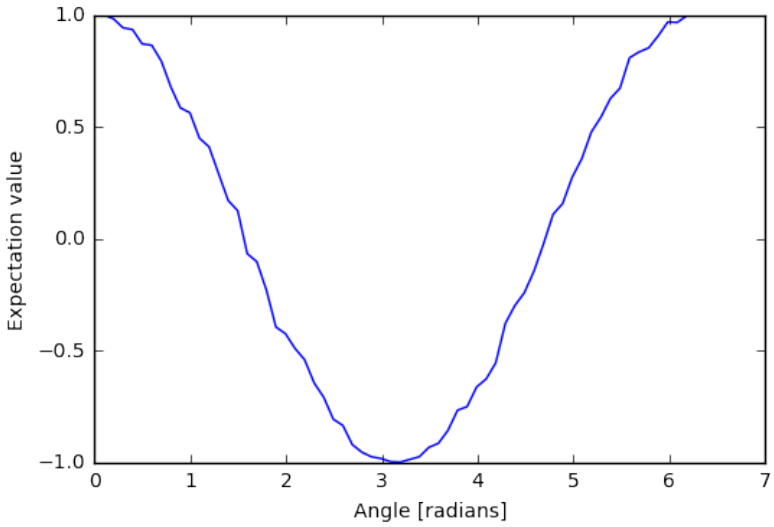
We can loop over a range of these angles and plot the expectation value.

```
angle_range = np.linspace(0.0, 2 * np.pi, 20)
data = [vqe_inst.expectation(small_ansatz([angle]), hamiltonian, None, qvm)
        for angle in angle_range]

import matplotlib.pyplot as plt
plt.xlabel('Angle [radians]')
plt.ylabel('Expectation value')
plt.plot(angle_range, data)
plt.show()
```



Now with sampling…

```
angle_range = np.linspace(0.0, 2 * np.pi, 20)
data = [vqe_inst.expectation(small_ansatz([angle]), hamiltonian, 1000, qvm)
        for angle in angle_range]

import matplotlib.pyplot as plt
plt.xlabel('Angle [radians]')
plt.ylabel('Expectation value')
plt.plot(angle_range, data)
plt.show()
```



We can compare this plot against the value we obtain when we run the our variational quantum eigensolver.

```
result = vqe_inst.vqe_run(small_ansatz, hamiltonian, initial_angle, None, qvm=qvm)
print(result)
```

```
{'fun': -0.99999999954538055, 'x': array([ 3.1415625])}
```

1.2.3 Running Noisy VQE

A great thing about VQE is that it is somewhat insensitive to noise. We can test this out by running the previous algorithm on a noisy qvm.

Remember that Pauli channels are defined as a list of three probabilities that correspond to the probability of a random X, Y, or Z gate respectively. First we'll study the impact of a channel that has the same probability of each random Pauli.

```
pauli_channel = [0.1, 0.1, 0.1] #10% chance of each gate at each timestep
noisy_qvm = api.QVMConnection(gate_noise=pauli_channel)
```

Let us check that this QVM has noise:

```
p = Program(X(0), X(1)).measure(0, [0]).measure(1, [1])
noisy_qvm.run(p, [0, 1], 10)
```

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

We can run the VQE under noise. Let's modify the classical optimizer to start with a larger simplex so we don't get stuck at an initial minimum.

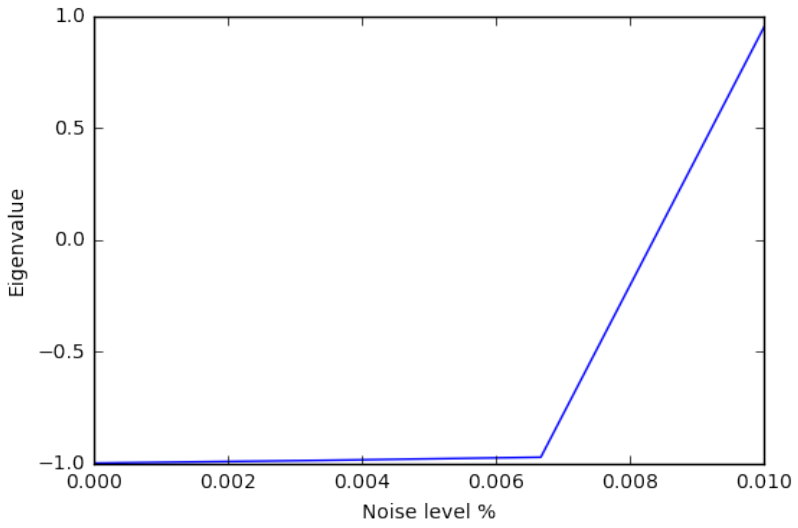
```
vqe_inst.minimizer_kwargs = {'method': 'Nelder-mead', 'options': {'initial_simplex':␣
↪np.array([[0.0], [0.05]]), 'xatol': 1.0e-2}}
result = vqe_inst.vqe_run(small_ansatz, hamiltonian, initial_angle, samples=10000,␣
↪qvm=noisy_qvm)
print(result)
```

```
{'fun': 0.5875999999999999, 'x': array([ 0.01874886])}
```

10% error is a huge amount of error! We can plot the effect of increasing noise on the result of this algorithm:

```
data = []
noises = np.linspace(0.0, 0.01, 4)
for noise in noises:
    pauli_channel = [noise] * 3
    noisy_qvm = api.QVMConnection(gate_noise=pauli_channel)
    # We can pass the noise params directly into the vqe_run instead of passing the␣
↪noisy connection
    result = vqe_inst.vqe_run(small_ansatz, hamiltonian, initial_angle,
                              gate_noise=pauli_channel)
    data.append(result['fun'])
```

```
plt.xlabel('Noise level %')
plt.ylabel('Eigenvalue')
plt.plot(noises, data)
plt.show()
```



It looks like this algorithm is pretty robust to noise up until 0.6% error. However measurement noise might be a different story.

```
meas_channel = [0.1, 0.1, 0.1] #10% chance of each gate at each measurement
noisy_meas_qvm = api.QVMConnection(measurement_noise=meas_channel)
```

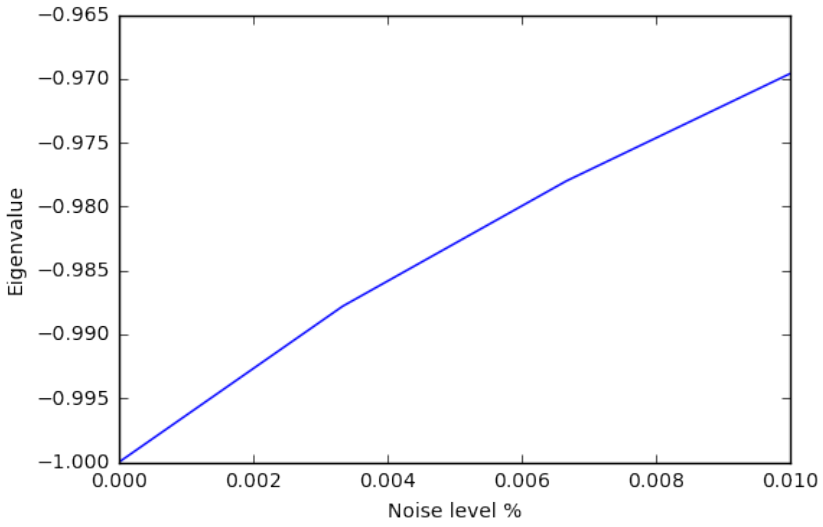
Measurement noise has a different effect:

```
p = Program(X(0), X(1)).measure(0, [0]).measure(1, [1])
noisy_meas_qvm.run(p, [0, 1], 10)
```

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

```
data = []
noises = np.linspace(0.0, 0.01, 4)
for noise in noises:
    meas_channel = [noise] * 3
    noisy_qvm = api.QVMConnection(measurement_noise=meas_channel)
    result = vqe_inst.vqe_run(small_ansatz, hamiltonian, initial_angle, samples=10000,
↪ qvm=noisy_qvm)
    data.append(result['fun'])
```

```
plt.xlabel('Noise level %')
plt.ylabel('Eigenvalue')
plt.plot(noises, data)
plt.show()
```



We see this particular VQE algorithm is generally more sensitive to measurement noise than gate noise.

1.2.4 More Sophisticated Ansatzes

Because we are working with Python, we can leverage the full language to make much more sophisticated ansatzes for VQE. As an example we can easily change the number of gates.

```
def smallish_ansatz(params):
    return Program(RX(params[0], 0), RX(params[1], 0))

print(smallish_ansatz([1.0, 2.0]))
```

```
RX(1.0) 0
RX(2.0) 0
```

```
vqe_inst = VQE(minimizer=minimize,
               minimizer_kwargs={'method': 'nelder-mead'})
initial_angles = [1.0, 1.0]
result = vqe_inst.vqe_run(smallish_ansatz, hamiltonian, initial_angles, None, qvm=qvm)
print(result)
```

```
{'fun': -1.0000000000000004, 'x': array([ 1.61767133,  1.52392133])}
```

We can even dynamically change the gates in the circuit based on a parameterization:

```
def variable_gate_ansatz(params):
    gate_num = int(np.round(params[1])) # for scipy.minimize params must be floats
    p = Program(RX(params[0], 0))
    for gate in range(gate_num):
        p.inst(X(0))
    return p

print(variable_gate_ansatz([0.5, 3]))
```

```
RX(0.5) 0
X 0
X 0
X 0
```

```
initial_params = [1.0, 3]
result = vqe_inst.vqe_run(variable_gate_ansatz, hamiltonian, initial_params, None,␣
↪qvm=qvm)
print(result)
```

```
{'fun': -1.0, 'x': array([  2.65393312e-09,   3.42891875e+00])}
```

Note that the restriction that the ansatz function take a single list of floats as parameters only comes from our choice of minimizer (this is what `scipy.optimize.minimize` takes). One could easily imagine writing a custom minimizer that takes more sophisticated forms of arguments.

1.2.5 Links and Further Reading

This concludes our brief tour of VQE. There is a lot of fascinating literature about this algorithm out there and we encourage you to both explore those topics as well as come up with new ideas using this library. Let us know if you have ideas about anything that you would like to see added!

Here are some links to get you started:

- A Variational Eigenvalue Solver on a Quantum Processor
- The Theory of Variational Hybrid Quantum-Classical Algorithms
- Hybrid Quantum-Classical Approach to Correlated Materials
- A Hybrid Classical/Quantum Approach for Large-Scale Studies of Quantum Systems with Density Matrix Embedding Theory
- Hybrid Quantum-Classical Hierarchy for Mitigation of Decoherence and Determination of Excited States

1.2.6 Source Code Docs

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

grove.pyvqe.vqe

class grove.pyvqe.vqe.**OptResults**
Bases: dict

Object for holding optimization results from VQE.

class grove.pyvqe.vqe.**VQE**(*minimizer*, *minimizer_args=[]*, *minimizer_kwargs={}*)
Bases: object

The Variational-Quantum-Eigensolver algorithm

VQE is an object that encapsulates the VQE algorithm (functional minimization). The main components of the VQE algorithm are a minimizer function for performing the functional minimization, a function that takes a vector of parameters and returns a pyQuil program, and a Hamiltonian of which to calculate the expectation value.

Using this object:

1) initialize with *inst = VQE(minimizer)* where *minimizer* is a function that performs a gradient free minization–i.e scipy.optimize.minimize(. , ., method='Nelder-Mead')

2) call *inst.vqe_run(variational_state_evolve, hamiltonian, initial_parameters)*. Returns the optimal parameters and minimum expecation

Parameters

- **minimizer** – function that minimizes objective f(obj, param). For example the function scipy.optimize.minimize() needs at least two parameters, the objective and an initial point for the optimization. The args for minimizer are the cost function (provided by this class), initial parameters (passed to vqe_run() method, and jacobian (defaulted to None). kwargs can be passed in below.
- **minimizer_args** – (list) arguments for minimizer function. Default=None
- **minimizer_kwargs** – (dict) arguments for keyword args. Default=None

static expectation(*pyquil_prog*, *pauli_sum*, *samples*, *qvm*)
Computes the expectation value of pauli_sum over the distribution generated from pyquil_prog.

Parameters

- **pyquil_prog** – (pyQuil program)
- **pauli_sum** – (PauliSum, ndarray) PauliSum representing the operator of which to calculate the expectation value or a numpy matrix representing the Hamiltonian tensored up to the appropriate size.
- **samples** – (int) number of samples used to calculate the expectation value. If samples is None then the expectation value is calculated by calculating <psi|O|psi> on the QVM. Error models will not work if samples is None.
- **qvm** – (qvm connection)

Returns (float) representing the expectation value of pauli_sum given given the distribution generated from quil_prog.

vqe_run(*variational_state_evolve*, *hamiltonian*, *initial_params*, *gate_noise=None*, *measurement_noise=None*, *jacobian=None*, *qvm=None*, *disp=None*, *samples=None*, *return_all=False*)
functional minimization loop.

Parameters

- **variational_state_evolve** – function that takes a set of parameters and returns a pyQuil program.
- **hamiltonian** – (PauliSum) object representing the hamiltonian of which to take the expectation value.
- **initial_params** – (ndarray) vector of initial parameters for the optimization
- **gate_noise** – list of Px, Py, Pz probabilities of gate being applied to every gate after each get application
- **measurement_noise** – list of Px', Py', Pz' probabilities of a X, Y or Z being applied before a measurement.
- **jacobian** – (optional) method of generating jacobian for parameters (Default=None).
- **qvm** – (optional, QVM) forest connection object.
- **disp** – (optional, bool) display level. If True then each iteration expectation and parameters are printed at each optimization iteration.
- **samples** – (int) Number of samples for calculating the expectation value of the operators. If *None* then faster method ,dotting the wave function with the operator, is used. Default=None.
- **return_all** – (optional, bool) request to return all intermediate parameters determined during the optimization.

Returns

(vqe.OptResult()) object `OptResult`. The following fields are initialized in OptResult: -x: set of w.f. ansatz parameters -fun: scalar value of the objective function

-iteration_params: a list of all intermediate parameter vectors. Only returned if 'return_all=True' is set as a vqe_run() option.

-expectation_vals: a list of all intermediate expectation values. Only returned if 'return_all=True' is set as a vqe_run() option.

`grove.pyvqe.vqe.`**expectation_from_sampling**(*pyquil_program*, *marked_qubits*, *qvm*, *samples*)
Calculation of Z_{i} at marked_qubits

Given a wavefunctions, this calculates the expectation value of the Zi operator where i ranges over all the qubits given in marked_qubits.

Parameters

- **pyquil_program** – pyQuil program generating some state
- **marked_qubits** – The qubits within the support of the Z pauli operator whose expectation value is being calculated
- **qvm** – A QVM connection.
- **samples** – Number of bitstrings collected to calculate expectation from sampling.

Returns The expectation value as a float.

grove.pyvqe.vqe.**parity_even_p**(*state*, *marked_qubits*)
Calculates the parity of elements at indexes in marked_qubits

Parity is relative to the binary representation of the integer state.

Parameters

- **state** – The wavefunction index that corresponds to this state.
- **marked_qubits** – The indexes to be considered in the parity sum.

Returns A boolean corresponding to the parity.

1.3 Quantum Approximate Optimization Algorithm (QAOA)

1.3.1 Overview

pyQAOA is a Python module for running the Quantum Approximate Optimization Algorithm on an instance of a quantum abstract machine.

The pyQAOA package contains separate modules for each type of problem instance: MAX-CUT, graph partitioning, etc. For each problem instance the user specifies the driver Hamiltonian, cost Hamiltonian, and the approximation order of the algorithm.

`qaoa.py` contains the base QAOA class and routines for finding optimal rotation angles via Grove's variational-quantum-eigensolver method.

1.3.2 Cost Functions

- `maxcut_qaoa.py` implements the cost function for MAX-CUT problems.
- `numpartition_qaoa.py` implements the cost function for bipartitioning a list of numbers.

1.3.3 Quickstart Examples

To test your installation and get going we can run QAOA to solve MAX-CUT on a square ring with 4 nodes at the corners. In your python interpreter import the packages and connect to your QVM:

```
import numpy as np
from grove.pyqaoa.maxcut_qaoa import maxcut_qaoa
import pyquil.api as api
qvm_connection = api.QVMConnection()
```

Next define the graph on which to run MAX-CUT

```
square_ring = [(0,1),(1,2),(2,3),(3,0)]
```

The optional configuration parameter for the algorithm is given by the number of steps to use (which loosely corresponds to the accuracy of the optimization computation). We instantiate the algorithm and run the optimization routine on our QVM:

```
steps = 2
inst = maxcut_qaoa(graph=square_ring, steps=steps)
betas, gammas = inst.get_angles()
```

to see the final \(\mid \beta, \gamma \rangle \) state we can rebuild the quil program that gives us \(\mid \beta, \gamma \rangle \) and evaluate the wave function using the QVM

```
t = np.hstack((betas, gammas))
param_prog = inst.get_parameterized_program()
prog = param_prog(t)
wf = qvm_connection.wavefunction(prog)
wf = wf.amplitudes
```

`wf` is now a numpy array of complex-valued amplitudes for each computational basis state. To visualize the distribution iterate over the states and calculate the probability.

```
for state_index in range(2**inst.n_qubits):
    print(inst.states[state_index], np.conj(wf[state_index])*wf[state_index])
```

You should then see that the algorithm converges on the expected solutions of 0101 and 1010!

```
0000 (4.38395094039e-26+0j)
0001 (5.26193287055e-15+0j)
0010 (5.2619328789e-15+0j)
0011 (1.52416449345e-13+0j)
0100 (5.26193285935e-15+0j)
0101 (0.5+0j)
0110 (1.52416449362e-13+0j)
0111 (5.26193286607e-15+0j)
1000 (5.26193286607e-15+0j)
1001 (1.52416449362e-13+0j)
1010 (0.5+0j)
1011 (5.26193285935e-15+0j)
1100 (1.52416449345e-13+0j)
1101 (5.2619328789e-15+0j)
1110 (5.26193287055e-15+0j)
1111 (4.38395094039e-26+0j)
```

1.3.4 Algorithm and Details

Introduction

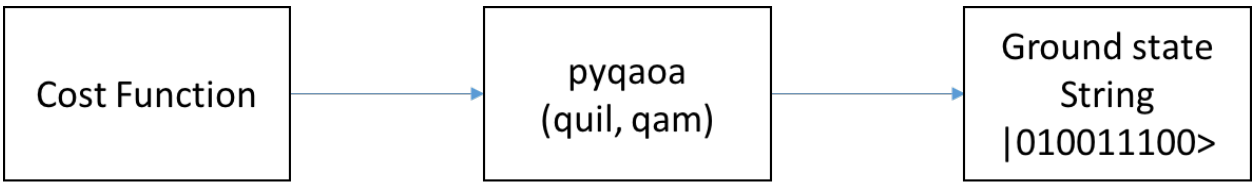
The quantum-approximate-optimization-algorithm (QAOA, pronouced quah-wah), developed by Farhi, Goldstone, and Gutmann, is a polynomial time algorithm for finding "a 'good' solution to an optimization problem" [1, 2].

What's with the name? For a given NP-Hard problem an approximate algorithm is a polynomial-time algorithm that solves every instance of the problem with some guaranteed quality in expectation. The value of merit is the ratio between the quality of the polynomial time solution and the quality of the true solution.

One reason QAOA is interesting is its potential to exhibit quantum supremacy [1].

This package, which is an implementation of QAOA that runs on a simulated quantum computer, can be used as a stand alone optimizer or a plugin optimization routine in a larger environment. The usage pipeline is as follows: 1) encoding the cost function into a set of Pauli operators, 2) instantiating the problem with pyQAOA and pyQuil, and 3) retrieving ground state solution by sampling.

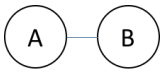
The following section of the pyQAOA documentation describes the algorithm and the NP-hard problem instance used in the original paper.



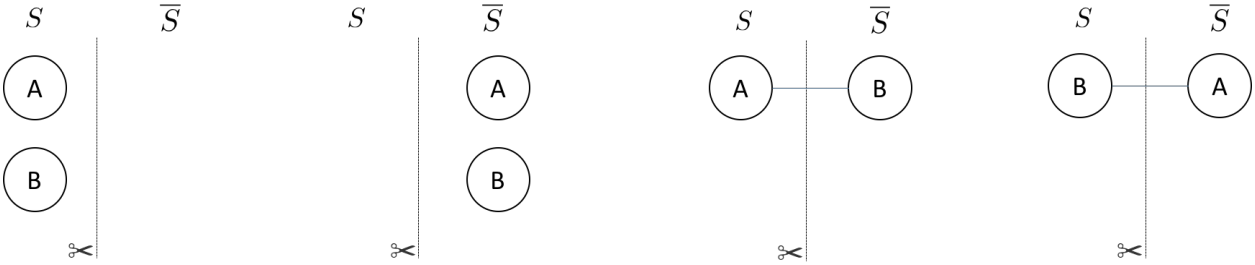
Our First NP-Hard Problem

The maximum-cut problem (MAX-CUT) was the first application described in the original quantum-approximate-optimization-algorithm paper [2]. This problem is similar to graph coloring. Given a graph of nodes and edges, color each node black or white, then score a point for each node that is next to a node of a different color. The aim is to find a coloring that scores the most points.

Stated a bit more formally, the problem is to partition the nodes of a graph into two sets such that the number of edges connecting nodes in opposite sets is maximized. For example, consider the barbell graph

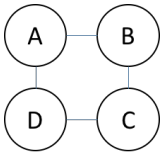


there are 4 ways of partitioning nodes into two sets:



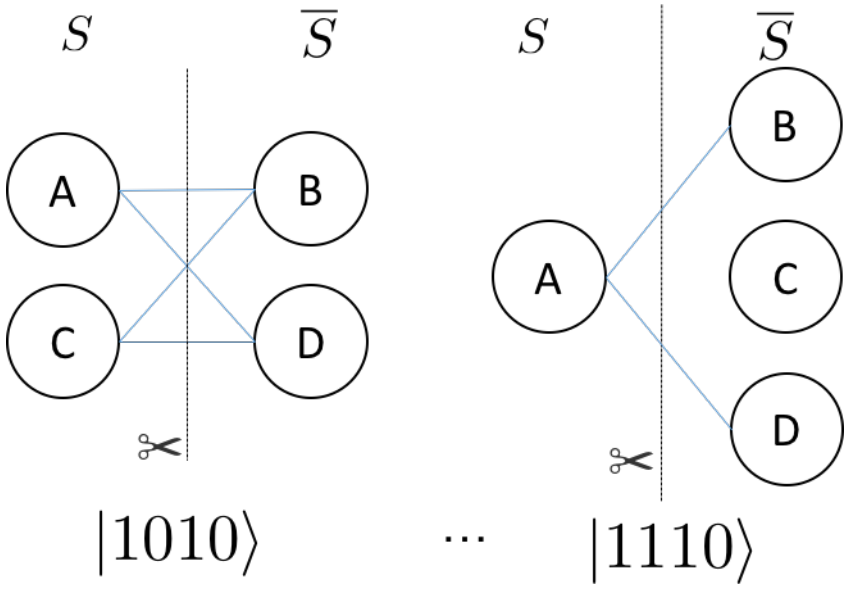
We have drawn the edge only when it connects nodes in different sets. The line with the scissor symbol indicates that we count the edge in our cut. For the barbell graph there are two equal weight partitionings that correspond to a maximum cut (the right two partitonings)–i.e. cutting the barbell in half. One can denote which set \(S \) or \(\overline{S} \) a node is in with either a \(0\) or a \(1\), respectively, in a bit string of length \(N \). The four partitionings of the barbell graph listed above are, \(\{ 00, 11, 01, 10 \} \)—where the left most bit is node \(A\) and the right most bit is node \(B\). The bit string representation makes it easy to represent a particular partition of the graph. Each bit string has an associated cut weight.

For any graph, the bit string representations of the node partitionings are always length \(N\). The total number of partitionings grows as \(2^{N}\). For example, a square ring graph



has 16 possible partitions (\(2^{4}\)). Below are two possible ways of parititioning of the nodes.

The bit strings associated with each parititioning are indicated in the figure. The right most bit corresponds with the node labeled \(A\) and the left most bit corresponds with the node labeled \(D\).



Classical Solutions

In order to find the best cut on a classical computer the obvious approach is to enumerate all partitions of the graph and check the weight of the cut associated with the partition.

Faced with an exponential cost for finding the optimal cut (or set of optimal cuts) one can devise a polynomial algorithm that is guaranteed to be of a particular quality. For example, a famous polynomial time algorithm is the randomized partitioning approach. One simply iterates over the nodes of the graph and flips a coin. If the coin is heads the node is in \(S \), if tails the node is in \(\overline{S} \). The quality of the random assignment algorithm is at least 50 percent of the maximum cut. For a coin-flip process the probability of an edge being in the cut is 50%. Therefore, the expectation value of a cut produced by random assignment can be written as follows: $$\sum_{e \in E} w_{e} \cdot \mathrm{Pr}(e \in \mathrm{cut}) = \frac{1}{2} \sum_{e \in E}w_{e}$$ Since the sum of all the edges is necessarily an upper bound to the maximum cut the randomized approach produces a cut of expected value of at least 0.5 times the best cut on the graph.

Other polynomial approaches exist that involve semi-definite programming which give cuts of expected value at least 0.87856 times the maximum cut [3].

Quantum Approximate Optimization

One can think of the bit strings (or set of bit strings) that correspond to the maximum cut on a graph as the ground state of a Hamiltonian encoding the cost function. The form of this Hamiltonian can be determined by constructing the classical function that returns a 1 (or the weight of the edge) if the edge spans two-nodes in different sets, or 0 if the nodes are in the same set. \begin{align} C_{ij} = \frac{1}{2}(1 - z_{i}z_{j}) \end{align} \(z_{i}\) or \(z_{j}\) is \(+1\) if node \(i\) or node \(j\) is in \(S\) or \(-1\) if node \(i\) or node \(j\) is in \(\overline{S}\). The total cost is the sum of all \((i ,j) \) node pairs that form the edge set of the graph. This suggests that for MAX-CUT the Hamiltonian that encodes the problem is $$\sum_{ij}\frac{1}{2}(\mathbf{I} - \sigma_{i}^{z}\sigma_{j}^{z})$$ where the sum is over \((i,j) \) node pairs that form the edges of the graph. The quantum-approximate-optimization-algorithm relies on the fact that we can prepare something approximating the ground state of this Hamiltonian and perform a measurement on that state. Performing a measurement on the \(N\)-body quantum state returns the bit string corresponding to the maximum cut with high probability.

To make this concrete let us return to the barbell graph. The graph requires two qubits in order to represent the nodes. The Hamiltonian has the form \begin{align} \hat{H} = \frac{1}{2}(\mathbf{I} - \sigma_{z}^{1}\otimes \sigma_{z}^{0}) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}

\end{align} where the basis ordering corresponds to increasing integer values in binary format (the left most bit being the most significant). This corresponds to a basis ordering for the \(\hat{H}\) operator above as \begin{align} (| 00\rangle, | 01\rangle, | 10\rangle, | 11\rangle). \end{align} Here the Hamiltonian is diagonal with integer eigenvalues. Clearly each bit string is an eigenstate of the Hamiltonian because \(\hat{H}\) is diagonal.

QAOA identifies the ground state of the MAXCUT Hamiltonian by evolving from a reference state. This reference state is the ground state of a Hamiltonian that couples all \(2^{N} \) states that form the basis of the cost Hamiltonian—i.e. the diagonal basis for cost function. For MAX-CUT this is the \(Z\) computational basis.

The evolution between the ground state of the reference Hamiltonian and the ground state of the MAXCUT Hamiltonian can be generated by an interpolation between the two operators \begin{align} \hat{H}_{\tau} = \tau\hat{H}_{\mathrm{ref}} + (1 - \tau)\hat{H}_{\mathrm{MAXCUT}} \end{align} where \(\tau\) changes between 1 and 0. If the ground state of the reference Hamiltonian is prepared and \(\tau = 1\) the state is a stationary state of \(\hat{H}_{\tau}\). As \(\hat{H}_{\tau}\) transforms into the MAXCUT Hamiltonian the ground state will evolve as it is no longer stationary with respect to \(\hat{H}_{\tau \neq 1 }\). This can be thought of as a continuous version of the of the evolution in QAOA.

The appproximate portion of the algorithm comes from how many values of \(\tau\) are used for approximating the continuous evolution. We will call this number of slices \(\alpha\). The original paper [2] demonstrated that for \(\alpha = 1\) the optimal circuit produced a distribution of states with a Hamiltonian expectation value of 0.6924 of the true maximum cut for 3-regular graphs. Furthermore, the ratio between the true maximum cut and the expectation value from QAOA could be improved by increasing the number of slices approximating the evolution.

Details

For MAXCUT, the reference Hamiltonian is the sum of \(\sigma_{x}\) operators on each qubit. \begin{align} \hat{H}_{\mathrm{ref}} = \sum_{i=0}^{N-1} \sigma_{i}^{X} \end{align} This Hamiltonian has a ground state which is the tensor product of the lowest eigenvectors of the \(\sigma_{x}\) operator (\(\mid + \rangle\)). \begin{align} \mid \psi_{\mathrm{ref}}\rangle = \mid + \rangle_{N-1}\otimes\mid + \rangle_{N-2}\otimes\ldots\otimes\mid + \rangle_{0} \end{align}

The reference state is easily generated by performing a Hadamard gate on each qubit–assuming the initial state of the system is all zeros. The Quil code generating this state is

```
H 0
H 1
...
H N-1
```

pyQAOA requires the user to input how many slices (approximate steps) for the evolution between the reference and MAXCUT Hamiltonian. The algorithm then variationally determines the parameters for the rotations (denoted \(\beta\) and \(\gamma\)) using the quantum-variational-eigensolver method [4][5] that maximizes the cost function.

For example, if (\(\alpha = 2\)) is selected two unitary operators approximating the continuous evolution are generated. \begin{align} U = U(\hat{H}_{\alpha_{1}})U(\hat{H}_{\alpha_{0}}) \label{eq:evolve} \end{align} Each \(U(\hat{H}_{\alpha_{i}})\) is approximated by a first order Trotter-Suzuki decomposition with the number of Trotter steps equal to one \begin{align} U(\hat{H}_{s_{i}}) = U(\hat{H}_{\mathrm{ref}}, \beta_{i})U(\hat{H}_{\mathrm{MAXCUT}}, \gamma_{i}) \end{align} where \begin{align} U(\hat{H}_{\mathrm{ref}}, \beta_{i}) = e^{-i \hat{H}_{\mathrm{ref}} \beta_{i}} \end{align} and \begin{align} U(\hat{H}_{\mathrm{MAXCUT}}, \gamma_{i}) = e^{-i \hat{H}_{\mathrm{MAXCUT}} \gamma_{i}} \end{align} \(U(\hat{H}_{\mathrm{ref}}, \beta_{i}) \) and \(U(\hat{H}_{\mathrm{MAXCUT}}, \gamma_{i})\) can be expressed as a short quantum circuit.

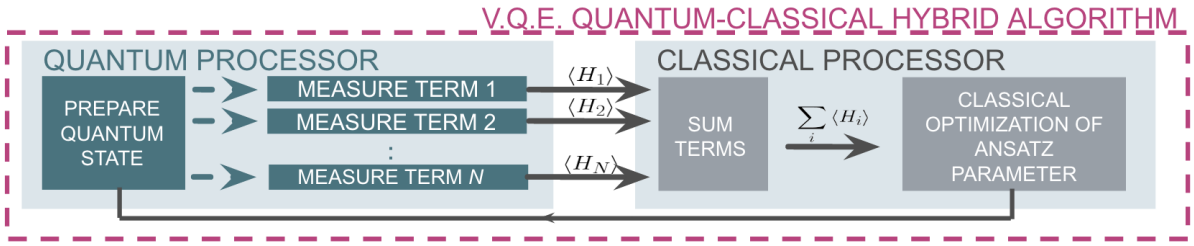
For the \(U(\hat{H}_{\mathrm{ref}}, \beta_{i})\) term (or mixing term) all operators in the sum commute and thus can be split into a product of exponentiated \(\sigma_{x}\) operators. \begin{align} e^{-i\hat{H}_{\mathrm{ref}} \beta_{i}} = \prod_{n = 0}^{1}e^{-i\sigma_{n}^{x}\beta_{i}} \end{align}

```
H 0
RZ(beta_i) 0
H 0
H 1
RZ(beta_i) 1
H 1
```

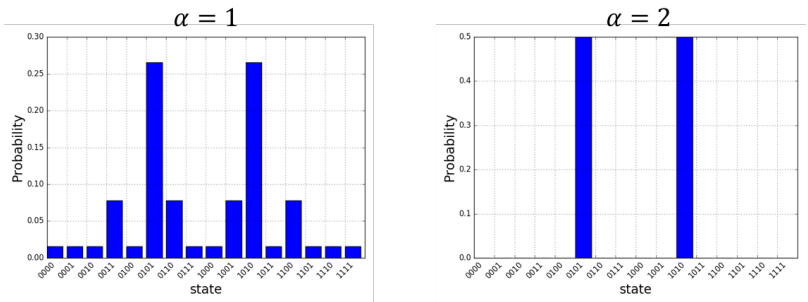
Of course, if RX is in the natural gate set for the quantum-processor this Quil is compiled into a set of RX rotations. The Quil code for the cost function \begin{align} e^{-i \frac{\gamma_{i}}{2}(\mathbf{I} - \sigma_{1}^{z} \otimes \sigma_{0}^{z}) } \end{align} looks like this:

```
X 0
PHASE(gamma{i}/2) 0
X 0
PHASE(gamma{i}/2) 0
CNOT 0 1
RZ(gamma{i}) 1
CNOT 0 1
```

Executing the Quil code will generate the \(\mid + \rangle_{1}\otimes\mid + \rangle_{0}\) state and perform the evolution with selected \(\beta\) and \(\gamma\) angles. \begin{align} \mid \beta, \gamma \rangle = e^{-i \hat{H}_{\mathrm{ref}} \beta_{1}}e^{-i \hat{H}_{\mathrm{MAXCUT}} \gamma_{1}}e^{-i \hat{H}_{\mathrm{ref}} \beta_{0}}e^{-i \hat{H}_{\mathrm{MAXCUT}} \gamma_{0}} \mid + \rangle_{N-1,\ldots,0} \end{align} In order to indentify the set of \(\beta\) and \(\gamma\) angles that maximize the objective function \begin{align} \mathrm{Cost} = \langle \beta, \gamma \mid \hat{H}_{\mathrm{MAXCUT}} \mid \beta, \gamma \rangle \label{expect} \end{align} pyQAOA leverages the classical-quantum hybrid approach known as the quantum-variational-eigensolver[4][5]. The quantum processor is used to prepare a state through a polynomial number of operations which is then used to evaluate the cost. Evaluating the cost (\(\langle \beta, \gamma \mid \hat{H}_{\mathrm{MAXCUT}} \mid \beta, \gamma \rangle\)) requires many preparations and measurements to generate enough samples to accurately construct the distribution. The classical computer then generates a new set of parameters (\(\beta, \gamma\)) for maximizing the cost function.



By allowing variational freedom in the \(\beta \) and \(\gamma \) angles QAOA finds the optimal path for a fixed number of steps. Once optimal angles are determined by the classical optimization loop one can read off the distribution by many preparations of the state with \(\beta, \gamma\) and sampling.



The probability distributions above are for the four ring graph discussed earlier. As expected the approximate evolution becomes more accurate as the number of steps (\(\alpha\)) is increased. For this simple model \(\alpha = 2\) is sufficient to find the two degenerate cuts of the four ring graph.

1.3.5 Source Code Docs

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

grove.pyqaoa.qaoa

class grove.pyqaoa.qaoa.**QAOA**(*qvm*, *qubits*, *steps=1*, *init_betas=None*, *init_gammas=None*, *cost_ham=None*, *ref_ham=None*, *driver_ref=None*, *minimizer=None*, *minimizer_args=None*, *minimizer_kwargs=None*, *rand_seed=None*, *vqe_options=None*, *store_basis=False*)

Bases: object

QAOA object.

Contains all information for running the QAOA algorthm to find the ground state of the list of cost clauses.

N.B. This only works if all the terms in the cost Hamiltonian commute with each other.

Parameters

- **qvm** – (Connection) The qvm connection to use for the algorithm.
- **qubits** – (list of ints) The number of qubits to use for the algorithm.
- **steps** – (int) The number of mixing and cost function steps to use. Default=1.
- **init_betas** – (list) Initial values for the beta parameters on the mixing terms. Default=None.
- **init_gammas** – (list) Initial values for the gamma parameters on the cost function. Default=None.
- **cost_ham** – list of clauses in the cost function. Must be PauliSum objects
- **ref_ham** – list of clauses in the mixer function. Must be PauliSum objects
- **driver_ref** – (pyQuil.quil.Program()) object to define state prep for the starting state of the QAOA algorithm. Defaults to tensor product of |+> states.
- **rand_seed** – integer random seed for initial betas and gammas guess.
- **minimizer** – (Optional) Minimization function to pass to the Variational-Quantum-Eigensolver method
- **minimizer_kwargs** – (Optional) (dict) of optional arguments to pass to the minimizer. Default={}.
- **minimizer_args** – (Optional) (list) of additional arguments to pass to the minimizer. Default=[].
- **minimizer_args** – (Optional) (list) of additional arguments to pass to the minimizer. Default=[].
- **vqe_options** – (optinal) arguents for VQE run.
- **store_basis** – (optional) boolean flag for storing basis states. Default=False.

get_angles()
Finds optimal angles with the quantum variational eigensolver method.

Stored VQE result

Returns ([list], [list]) A tuple of the beta angles and the gamma angles for the optimal solution.

get_parameterized_program()
Return a function that accepts parameters and returns a new Quil program.

Returns a function

get_string(*betas*, *gammas*, *samples=100*)
Compute the most probable string.

The method assumes you have passed init_betas and init_gammas with your pre-computed angles or you have run the VQE loop to determine the angles. If you have not done this you will be returning the output for a random set of angles.

Parameters

- **betas** – List of beta angles
- **gammas** – List of gamma angles
- **samples** – (int, Optional) number of samples to get back from the QVM.

Returns tuple representing the bitstring, Counter object from collections holding all output bitstrings and their frequency.

probabilities(*angles*)
Computes the probability of each state given a particular set of angles.

Parameters **angles** – [list] A concatenated list of angles [betas]+[gammas]

Returns [list] The probabilities of each outcome given those angles.

grove.pyqaoa.maxcut_qaoa

Finding a maximum cut by QAOA.

grove.pyqaoa.maxcut_qaoa.**maxcut_qaoa**(*graph*, *steps=1*, *rand_seed=None*, *connection=None*, *samples=None*, *initial_beta=None*, *initial_gamma=None*, *minimizer_kwargs=None*, *vqe_option=None*)

Max cut set up method

Parameters

- **graph** – Graph definition. Either networkx or list of tuples
- **steps** – (Optional. Default=1) Trotterization order for the QAOA algorithm.
- **rand_seed** – (Optional. Default=None) random seed when beta and gamma angles are not provided.
- **connection** – (Optional) connection to the QVM. Default is None.
- **samples** – (Optional. Default=None) VQE option. Number of samples (circuit preparation and measurement) to use in operator averaging.
- **initial_beta** – (Optional. Default=None) Initial guess for beta parameters.
- **initial_gamma** – (Optional. Default=None) Initial guess for gamma parameters.

- **minimizer_kwargs** – (Optional. Default=None). Minimizer optional arguments. If None set to `{'method': 'Nelder-Mead', 'options': {'ftol': 1.0e-2, 'xtol': 1.0e-2, 'disp': False}`
- **vqe_option** – (Optional. Default=None). VQE optional arguments. If None set to `vqe_option = {'disp': print_fun, 'return_all': True, 'samples': samples}`

grove.pyqaoa.maxcut_qaoa.**print_fun**(*x*)

grove.pyqaoa.numpartition_qaoa

grove.pyqaoa.numpartition_qaoa.**numpart_qaoa**(*asset_list*, *A=1.0*, *minimizer_kwargs=None*, *steps=1*)

generate number partition driver and cost functions

Parameters

- **asset_list** – list to binary parition
- **A** – (float) optional constant for level separation. Default=1.
- **minimizer_kwargs** – Arguments for the QAOA minimizer
- **steps** – (int) number of steps approximating the solution.

1.4 Quantum Fourier Transform (QFT)

1.4.1 Overview

The quantum Fourier transform is the quantum implementation of the discrete Fourier transform over the amplitudes of a wavefunction. Detailed explanations can be found in references[1] and[2]. The QFT forms the basis of many quantum algorithms such as Shor's factoring algorithm, discrete logarithm, and others to be found in the quantum algorithms zoo[3].

1.4.2 Source Code Docs

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

grove.qft.fourier

grove.qft.fourier.**bit_reversal**(*qubits*)

Generate a circuit to do bit reversal.

Parameters **qubits** – Qubits to do bit reversal with.

Returns A program to do bit reversal.

grove.qft.fourier.**inverse_qft**(*qubits*)

Generate a program to compute the inverse quantum Fourier transform on a set of qubits.

Parameters **qubits** – A list of qubit indexes.

[1] Nielsen, Michael A., and Isaac L. Chuang. Quantum Computation and Quantum Information. Cambridge University Press, 2010.
[2] Rieffel, E. G., and W. Polak. "A Gentle Introduction to Quantum Computing." (2011).
[3] http://math.nist.gov/quantum/zoo/

Returns A Quil program to compute the inverse Fourier transform of the qubits.

grove.qft.fourier.**qft**(*qubits*)

Generate a program to compute the quantum Fourier transform on a set of qubits.

Parameters **qubits** – A list of qubit indexes.

Returns A Quil program to compute the Fourier transform of the qubits.

References

1.5 Phase Estimation Algorithm

1.5.1 Overview

The phase estimation algorithm is a quantum subroutine useful for finding the eigenvalue corresponding to an eigenvector \(u\) of some unitary operator. It is the starting point for many other algorithms and relies on the inverse quantum Fourier transform. More details can be found in references[1].

1.5.2 Example

First, connect to the QVM.

```
import pyquil.api as api

qvm = api.QVMConnection()
```

Now we encode a phase into the unitary operator `U`.

```
import numpy as np

phase = 0.75
phase_factor = np.exp(1.0j * 2 * np.pi * phase)
U = np.array([[phase_factor, 0],
              [0, -1*phase_factor]])
```

Then, we feed this operator into the `phase_estimation` module. Here, we ask for 4 bits of precision.

```
from grove.alpha.phaseestimation.phase_estimation import phase_estimation

precision = 4
p = phase_estimation(U, precision)
```

Now, we run the program and check our output.

```
output = qvm.run(p, range(precision))
wavefunction = qvm.wavefunction(p)

print(output)
print(wavefunction)
```

This should print the following:

[1] Nielsen, Michael A., and Isaac L. Chuang. Quantum Computation and Quantum Information. Cambridge University Press, 2010.

```
[[0, 0, 1, 1]]
(1+0j)|01100>
```

Note that .75, written as a binary fraction of precision 4, is 0.1100. Thus, we have recovered the phase encoded into our unitary operator.

1.5.3 Source Code Docs

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

grove.phaseestimation.phase_estimation

grove.alpha.phaseestimation.phase_estimation.**controlled**(*m*)

Make a one-qubit-controlled version of a matrix.

Parameters **m** – (numpy.ndarray) A matrix.

Returns A controlled version of that matrix.

grove.alpha.phaseestimation.phase_estimation.**phase_estimation**(*U*, *accuracy*, *reg_offset=0*)

Generate a circuit for quantum phase estimation.

Parameters

- **U** – (numpy.ndarray) A unitary matrix.
- **accuracy** – (int) Number of bits of accuracy desired.
- **reg_offset** – (int) Where to start writing measurements (default 0).

Returns A Quil program to perform phase estimation.

References

1.6 Histogram based Tomography

1.6.1 Introduction

Quantum states generally encode information about several different mutually incompatible (non-commuting) sets of observables. A given quantum process including final measurement of all qubits will therefore only yield partial information about the pre-measurement state even if the measurement is repeated many times.

To access the information the state contains about other, non-compatible observables, one can apply unitary rotations before measuring. Assuming these rotations are done perfectly, the resulting measurements can be interpreted being of the un-rotated state but with rotated observables.

Quantum tomography is a method that formalizes this procedure and allows to use a complete or overcomplete set of pre-measurement rotations to fully characterize all matrix elements of the density matrix.

1.6.2 Example

Consider a density matrix $\rho = \begin{pmatrix} 0.3 & 0.2i \\ -0.2i & 0.7 \end{pmatrix}$.

Let us assume that our quantum processor's projective measurement yields perfect outcomes z in the Z basis, either z=*+1* or z=*-1*. Then the density matrix ρ will give outcome z=*+1* with probability p=*30%* and z=*-1* with p=*70%*, respectively. Consequently, if we repeat the Z-measurement many times, we can estimate the diagonal coefficients of the density matrix. To access the off-diagonals, however, we need to measure different observables such as X or Y.

If we rotate the state as $\rho \mapsto U\rho U^\dagger$ and then do our usual Z-basis measurement, then this is equivalent to rotating the measured observable as $Z \mapsto U^\dagger Z U$ and keeping our state ρ unchanged. This second point of view then allows us to see that if we apply a rotation such as $U = R_y(\pi/2)$ then this rotates the observable as $R_y(-\pi/2) Z R_y(+\pi/2) = \cos(\pi/2)Z - \sin(\pi/2)X = -X$. Similarly, we could rotate by $U = R_x(\pi/2)$ to measure the Y observable. Overall, we can construct a sequence of different circuits with outcome statistics that depend on all elements of the density matrix and that allow to estimate them using techniques such as maximum likelihood estimation (*[MLE]*).

We have visualized this in *Figure 1*.

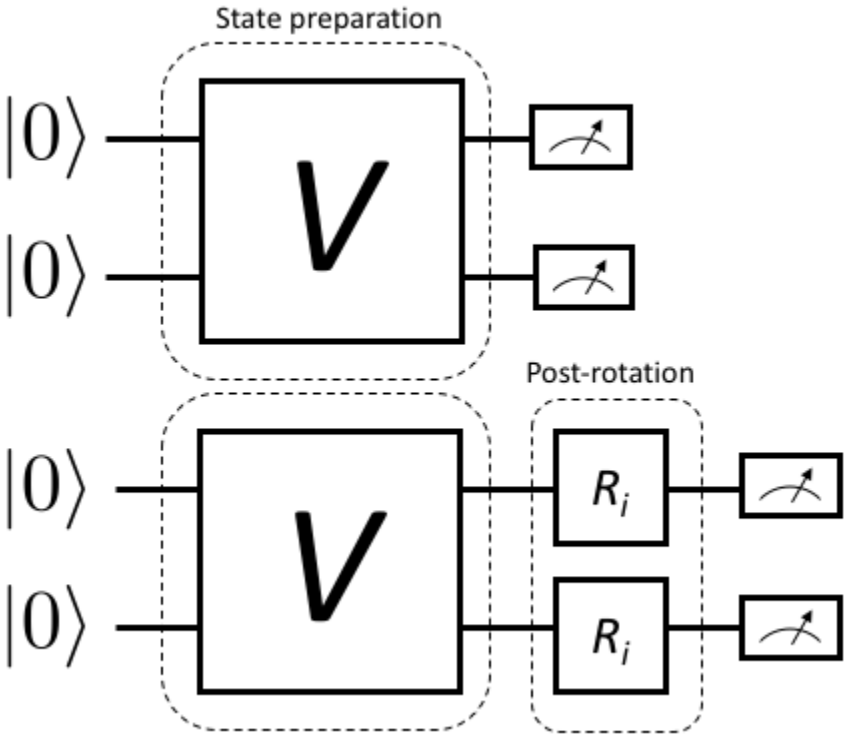


Fig. 1: **Figure 1:** This upper half of this diagram shows a simple 2-qubit quantum program consisting of both qubits initialized in the $|0\rangle$ state, then transformed to some other state via a process V and finally measured in the natural qubit basis.

On occasion we may also wish to estimate precisely what physical process a particular control/gate sequence realizes. This is done by a slight extension of the above scheme that also introduces pre-rotations that prepare different initial states, then act on these with the unknown map V and finally append post-rotations to fully determine the state that each initial state was mapped to. This is visualized in *Figure 2*.

The following sections formally define and introduce our tomography methods in full technical detail. Grove also contains an example notebook with tomography results obtained from the QPU. A rendered version of this can be

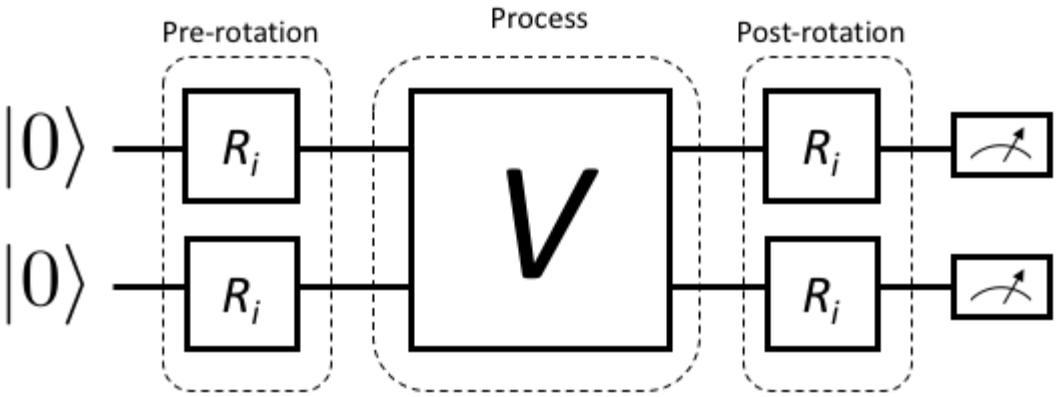


Fig. 2: **Figure 2:** For process tomography, rotations must be prepended and appended to fully resolve the action of *V* on arbitrary initial states.

found in *example_code*.

1.6.3 Useful notation and definitions

In the following we use 'super-ket' notation $|\,\rho\rangle\!\rangle := \mathrm{vec}\,(\hat{\rho})$ where $\mathrm{vec}\,(\hat{\rho})$ is a density operator $\hat{\rho}$ collapsed to a single vector by stacking its columns. The standard basis in this space is given by $\{|\,j\rangle\!\rangle,\ j = 0, 1, 2 \ldots, d^2 - 1\}$, where $j = f(k, l)$ is a multi-index enumerating the elements of a d-dimensional matrix row-wise, i.e. $j = 0 \Leftrightarrow (k, l) = (0, 0)$, $j = 1 \Leftrightarrow (k, l) = (0, 1)$, etc. The super-ket $|\,j\rangle\!\rangle$ then corresponds to the operator $|k\rangle\langle l|$.

We similarly define $\langle\!\langle\rho\,| := \mathrm{vec}\,(\hat{\rho})^\dagger$ such that the inner product $\langle\!\langle\chi|\rho\rangle\!\rangle = \mathrm{vec}\,(\hat{\chi})^\dagger \mathrm{vec}\,(\hat{\rho}) = \sum_{j,k=0}^{d^2-1} \chi^*_{jk}\rho_{jk} = \mathrm{Tr}\left(\hat{\chi}^\dagger\hat{\rho}\right)$ equals the Hilbert-Schmidt inner product. If ρ is a physical density matrix and $\hat{\chi}$ a Hermitian observable, this also equals its expectation value. When a state is represented as a super-ket, we can represent super-operators acting on them as $\Lambda \to \tilde{\Lambda}$, i.e., we write $|\,\Lambda(\hat{\rho})\rangle\!\rangle = \tilde{\Lambda}|\,\rho\rangle\!\rangle$.

We introduce an orthonormal, Hermitian basis for a single qubit in terms of the Pauli operators and the identity $|\,P_j\rangle\!\rangle := \mathrm{vec}\left(\hat{P}_j\right)$ for $j = 0, 1, 2, 3$, where $\hat{P}_0 = \hat{\mathbb{I}}/\sqrt{2}$ and $\hat{P}_k = \sigma_k/\sqrt{2}$ for $k = 1, 2, 3$. These satisfy $\langle\!\langle P_l|P_m\rangle\!\rangle = \delta_{lm}$ for $l, m = 0, 1, 2, 3$. For multi-qubit states, the generalization to a tensor-product basis representation carries over straightforwardly. The normalization $1/\sqrt{2}$ is generalized to $1/\sqrt{d}$ for a d-dimensional space. In the following we assume no particular size of the system.

We can then express both states and observables in terms of linear combinations of Pauli-basis super-kets and super-bras, respectively, and they will have real valued coefficients due to the hermiticity of the Pauli operator basis. Starting from an initial state ρ we can apply a completely positive map to it

$$\hat{\rho}' = \Lambda_K(\hat{\rho}) = \sum_{j=1}^{n} \hat{K}_j \hat{\rho} \hat{K}_j^\dagger.$$

A Kraus map is always completely positive and additionally is trace preserving if $\sum_{j=1}^{n} \hat{K}_j^\dagger \hat{K}_j = \hat{I}$. We can expand a given map $\Lambda(\hat{\rho})$ in terms of the Pauli basis by exploiting that $\sum_{j=0}^{d^2-1} |\,j\rangle\!\rangle\langle\!\langle\,j\,| = \sum_{j=0}^{d^2-1} \left|\,\hat{P}_j\,\right\rangle\!\rangle\!\left\langle\!\left\langle\,\hat{P}_j\,\right|\right. = \hat{I}$ where $\hat{I}$ is the super-identity map.

For any given map $\Lambda(\cdot), \mathcal{B} \to \mathcal{B}$, where $\mathcal{B}$ is the space of bounded operators, we can compute its Pauli-transfer matrix

as

$$(\mathcal{R}_\Lambda)_{jk} := \mathrm{Tr}\left(\hat{P}_j \Lambda(\hat{P}_k)\right), \quad j,k = 0,1,,\ldots,d^2-1.$$

In contrast to *[Chow]*, our tomography method does not rely on a measurement with continuous outcomes but rather discrete POVM outcomes $j \in \{0,1,\ldots,d-1\}$, where d is the dimension of the underlying Hilbert space. In the case of perfect readout fidelity the POVM outcome j coincides with a projective outcome of having measured the basis state $|j\rangle$. For imperfect measurements, we can falsely register outcomes of type $k \neq j$ even if the physical state before measurement was $|j\rangle$. This is quantitatively captured by the readout POVM. Any detection scheme—including the actual readout and subsequent signal processing and classification step to a discrete bitstring outcome—can be characterized by its confusion rate matrix, which provides the conditional probabilities $p(j|k) := p(\text{detected } j \mid \text{prepared } k)$ of detected outcome j given a perfect preparation of basis state $|k\rangle$

$$P = \begin{pmatrix} p(0|0) & p(0|1) & \cdots & p(0|d-1) \\ p(1|0) & p(1|1) & \cdots & p(1|d-1) \\ \vdots & & & \vdots \\ p(d-1|0) & p(d-1|1) & \cdots & p(d-1|d-1) \end{pmatrix}.$$

The trace of the confusion rate matrix (*[ConfusionMatrix]*) divided by the number of states $F := \mathrm{Tr}\,(P)/d = \sum_{j=0}^{d-1} p(j|j)/d$ gives the joint assignment fidelity of our simultaneous qubit readout *[Jeffrey]*, *[Magesan]*. Given the coefficients appearing in the confusion rate matrix the equivalent readout *[POVM]* is

$$\hat{N}_j := \sum_{k=0}^{d-1} p(j|k) \hat{\Pi}_k$$

where we have introduced the bitstring projectors $\hat{\Pi}_k = |k\rangle\langle k|$. We can immediately see that $\hat{N}_j \geq 0$ for all j, and verify the normalization

$$\sum_{j=0}^{d-1} \hat{N}_j = \sum_{k=0}^{d-1} \underbrace{\sum_{j=0}^{d-1} p(j|k)}_{1} \hat{\Pi}_k = \sum_{k=0}^{d-1} \hat{\Pi}_k = \hat{\mathbb{I}}$$

where $\hat{\mathbb{I}}$ is the identity operator.

1.6.4 State tomography

For state tomography, we use a control sequence to prepare a state ρ and then apply d^2 different post-rotations $\hat{R}_k$ to our state $\rho \mapsto \Lambda_{R_k}(\hat{\rho}) := \hat{R}_k \hat{\rho} \hat{R}_k^\dagger$ such that $\mathrm{vec}\,(\Lambda_{R_k}(\hat{\rho})) = \tilde{\Lambda}_{R_k} |\rho\rangle\rangle$ and subsequently measure it in our given measurement basis. We assume that subsequent measurements are independent which implies that the relevant statistics for our Maximum-Likelihood-Estimator (MLE) are the histograms of measured POVM outcomes for each prepared state:

$$n_{jk} := \text{ number of outcomes } j \text{ for an initial state } \tilde{\Lambda}_{R_k} |\rho\rangle\rangle$$

If we measure a total of $n_k = \sum_{j=0}^{d-1} n_{jk}$ shots for the pre-rotation $\hat{R}_k$ the probability of obtaining the outcome $h_k := (n_{0k}, \ldots, n_{(d-1)k})$ is given by the multinomial distribution

$$p(h_k) = \binom{n_k}{n_{0k}\ n_{1k}\ \cdots\ n_{(d-1)k}} p_{0k}^{n_{0k}} \cdots p_{(d-1)k}^{n_{(d-1)k}},$$

where for fixed k the vector $(p_{0k}, \ldots, p_{(d-1)k})$ gives the single shot probability over the POVM outcomes for the prepared circuit. These probabilities are given by

$$\begin{aligned} p_{jk} &:= \langle\langle N_j \,|\tilde{\Lambda}_{R_k}|\, \rho \rangle\rangle \\ &= \sum_{m=0}^{d^2-1} \underbrace{\sum_{r=0}^{d^2-1} \pi_{jr} (\hat{\mathcal{R}}_k)_{rm}}_{C_{jkm}} \rho_m \\ &= \sum_{m=0}^{d^2-1} C_{jkm} \rho_m. \end{aligned}$$

Here we have introduced $\pi_{jl} := \langle\langle N_j | P_l \rangle\rangle = \mathrm{Tr}\left(\hat{N}_j \hat{P}_l\right)$, $(\mathcal{R}_k)_{rm} := \langle\langle P_r \,|\tilde{\Lambda}_{R_k}|\, P_m \rangle\rangle$ and $\rho_m := \langle\langle P_m | \rho \rangle\rangle$. The POVM operators $N_j = \sum_{k=0}^{d-1} p(j|k) \Pi_k$ are defined as above.

The joint log likelihood for the unknown coefficients ρ_m for all pre-measurement channels $\mathcal{R}_k$ is given by

$$\log L(\rho) = \sum_{j=0}^{d-1} \sum_{k=0}^{d^2-1} n_{jk} \log \left(\sum_{m=0}^{d^2-1} C_{jkm} \rho_m \right) + \text{const.}$$

Maximizing this is a convex problem and can be efficiently done even with constraints that enforce normalization $\mathrm{Tr}\,(\rho) = 1$ and positivity $\rho \geq 0$.

1.6.5 Process Tomography

Process tomography introduces an additional index over the pre-rotations $\hat{R}_l$ that act on a fixed initial state ρ_0. The result of each such preparation is then acted on by the process $\tilde{\Lambda}$ that is to be inferred. This leads to a sequence of different states

$$\hat{\rho}^{(kl)} := \hat{R}_k \Lambda(\hat{R}_l \rho_0 \hat{R}_l^\dagger) \hat{R}_k^\dagger \leftrightarrow \left| \rho^{(kl)} \right\rangle\!\!\right\rangle = \tilde{\Lambda}_{R_k} \tilde{\Lambda} \tilde{\Lambda}_{R_l} |\, \rho_0 \rangle\rangle.$$

The joint histograms of all such preparations and final POVM outcomes is given by

$$n_{jkl} := \text{ number of outcomes } j \text{ given input } \left| \rho^{(kl)} \right\rangle\!\!\right\rangle.$$

If we measure a total of $n_{kl} = \sum_{j=0}^{d-1} n_{jkl}$ shots for the post-rotation k and pre-rotation l, the probability of obtaining the outcome $m_{kl} := (n_{0kl}, \ldots, n_{(d-1)kl})$ is given by the binomial

$$p(m_{kl}) = \binom{n_{kl}}{n_{0kl}\ n_{1kl}\ \cdots\ n_{(d-1)kl}} p_{0kl}^{n_{0kl}} \cdots p_{(d-1)kl}^{n_{(d-1)kl}}$$

where the single shot probabilities p_{jkl} of measuring outcome N_j for the post-channel k and pre-channel l are given by

$$\begin{aligned} p_{jkl} &:= \langle\langle N_j \,|\tilde{\Lambda}_{R_k} \tilde{\Lambda} \tilde{\Lambda}_{R_l}|\, \rho_0 \rangle\rangle \\ &= \sum_{m,n=0}^{d^2-1} \underbrace{\sum_{r,q=0}^{d^2-1} \pi_{jr} (\mathcal{R}_k)_{rm} (\mathcal{R}_l)_{nq} (\rho_0)_q (\mathcal{R})_{mn}}_{B_{jklmn}} \\ &= \sum_{mn=0}^{d^2-1} B_{jklmn} (\mathcal{R})_{mn} \end{aligned}$$

where $\pi_{jl} := \langle\langle N_j|l\rangle\rangle = \mathrm{Tr}\left(\hat{N}_j\hat{P}_l\right)$ and $(\rho_0)_q := \langle\langle P_q|\rho_0\rangle\rangle = \mathrm{Tr}\left(\hat{P}_q\hat{\rho}_0\right)$ and the Pauli-transfer matrices for the pre and post rotations R_l and the unknown process are given by

$$(\mathcal{R}_l)_{nq} := \mathrm{Tr}\left(\hat{P}_n\hat{R}_l\hat{P}_q\hat{R}_l^\dagger\right).$$
$$\mathcal{R}_{mn} := \mathrm{Tr}\left(\hat{P}_m\Lambda(\hat{R}_n)\right).$$

The joint log likelihood for the unknown transfer matrix $\mathcal{R}$ for all pre-rotations $\mathcal{R}_l$ and post-rotations $\mathcal{R}_k$ is given by

$$\log L(\mathcal{R}) = \sum_{j=0}^{d-1}\sum_{kl=0}^{d^2-1} n_{jkl}\log\left(\sum_{mn=0}^{d^2-1} B_{jklmn}(\mathcal{R})_{mn}\right) + \mathrm{const.}$$

Handling positivity constraints is achieved by constraining the associated Choi-matrix to be positive *[Chow]*. We can also constrain the estimated transfer matrix to preserve the trace of the mapped state by demanding that $\mathcal{R}_{0l} = \delta_{0l}$.

You can learn more about quantum channels here: *[QuantumChannel]*.

1.6.6 Metrics

Here we discuss some quantitative measures of comparing quantum states and processes.

For states

When comparing quantum states there are a variety of different measures of (in-)distinguishability, with each usually being the answer to a particular question, such as "With what probability can I distinguish two states in a single experiment?", or "How indistinguishable are measurement samples of two states going to be?".

A particularly easy to compute measure of indistinguishability is given by the quantum state fidelity, which for pure (and normalized) states is simply given by $F(\phi,\psi) = |\langle\phi|\psi\rangle|$. The fidelity is 1 if and only if the two states are identical up to a scalar factor. It is zero when they are orthogonal. The generalization to mixed states takes the form

$$F(\rho,\sigma) := \mathrm{Tr}\left(\sqrt{\sqrt{\rho}\sigma\sqrt{\rho}}\right).$$

Although this is not obvious from the expression it is symmetric under exchange of the states. Read more about it here: *[QuantumStateFidelity]* Although one can use the *infidelity* $1-F$ as a distance measure, it is not a proper metric. It can be shown, however that the so called Bures-angle $\theta_{\rho\sigma}$ implicitly defined via $\cos\theta_{\rho\sigma} = F(\rho,\sigma)$ does yield a proper metric in the mathematical sense.

Another useful metric is given by the trace distance (*[QuantumTraceDistance]*)

$$T(\rho,\sigma) := \frac{1}{2}\|\rho-\sigma\|_1 = \frac{1}{2}\mathrm{Tr}\left[\sqrt{(\rho-\sigma)^\dagger(\rho-\sigma)}\right],$$

which is also a proper metric and provides the answer to the above posed question of what the maximum single shot probability is to distinguish states ρ and σ.

For processes

For processes the two most popular metrics are the average gate fidelity $F_{\mathrm{avg}}(P,U)$ of an actual process P relative to some ideal unitary gate U. In some sense it measures the average fidelity (over all input states) by which a physical channel realizes the ideal operation. Given the Pauli transfer matrices $\mathcal{R}_P$ and $\mathcal{R}_U$ for the actual and ideal processes, respectively, the average gate fidelity (*[Chow]*) is

$$F_{\mathrm{avg}}(P,U) = \frac{\mathrm{Tr}\left(\mathcal{R}_P^T\mathcal{R}_U\right)/d+1}{d+1}$$

The corresponding infidelity $1 - F_{\text{avg}}(P, U)$ can be seen as a measure of the average gate error, but it is not a proper metric.

Another popular error metric is given by the diamond distance, which is a proper metric and has other nice properties that make it mathematically convenient for proving bounds on error thresholds, etc. It is given by the maximum trace distance between the ideal map and the actual map over all input states ρ that can generally feature entanglement with other ancillary degrees of freedom that U acts trivially on.

$$d(U, P)_\diamond = \max_\rho T\left((P \otimes I)[\rho], (U \otimes I)[\rho]\right)$$

In a sense, the diamond distance can be seen as a worst case error metric and it is particularly sensitive to *coherent* gate error, i.e., errors in which P is a (nearly) unitary process but deviates from U. See also these slides by Blume-Kohout et al. for more information *[GST]*.

1.6.7 Further resources

1.6.8 Run tomography experiments

This is a rendered version of the example notebook. and provides some example applications of grove's tomography module.

```
from __future__ import print_function
import matplotlib.pyplot as plt
from mock import MagicMock
import json

import numpy as np
from grove.tomography.state_tomography import do_state_tomography
from grove.tomography.utils import notebook_mode
from grove.tomography.process_tomography import do_process_tomography

# get fancy TQDM progress bars
notebook_mode(True)

from pyquil.gates import CZ, RY
from pyquil.api import QVMConnection, QPUConnection, get_devices
from pyquil.quil import Program

%matplotlib inline

NUM_SAMPLES = 2000

qvm = QVMConnection()
# QPU
online_devices = [d for d in get_devices() if d.is_online()]
if online_devices:
    d = online_devices[0]
    qpu = QPUConnection(d.name)
    print("Found online device {}, making QPUConnection".format(d.name))
else:
    qpu = QVMConnection()
```

```
Found online device 19Q-Acorn, making QPUConnection
```

Example Code

Create a Bell state

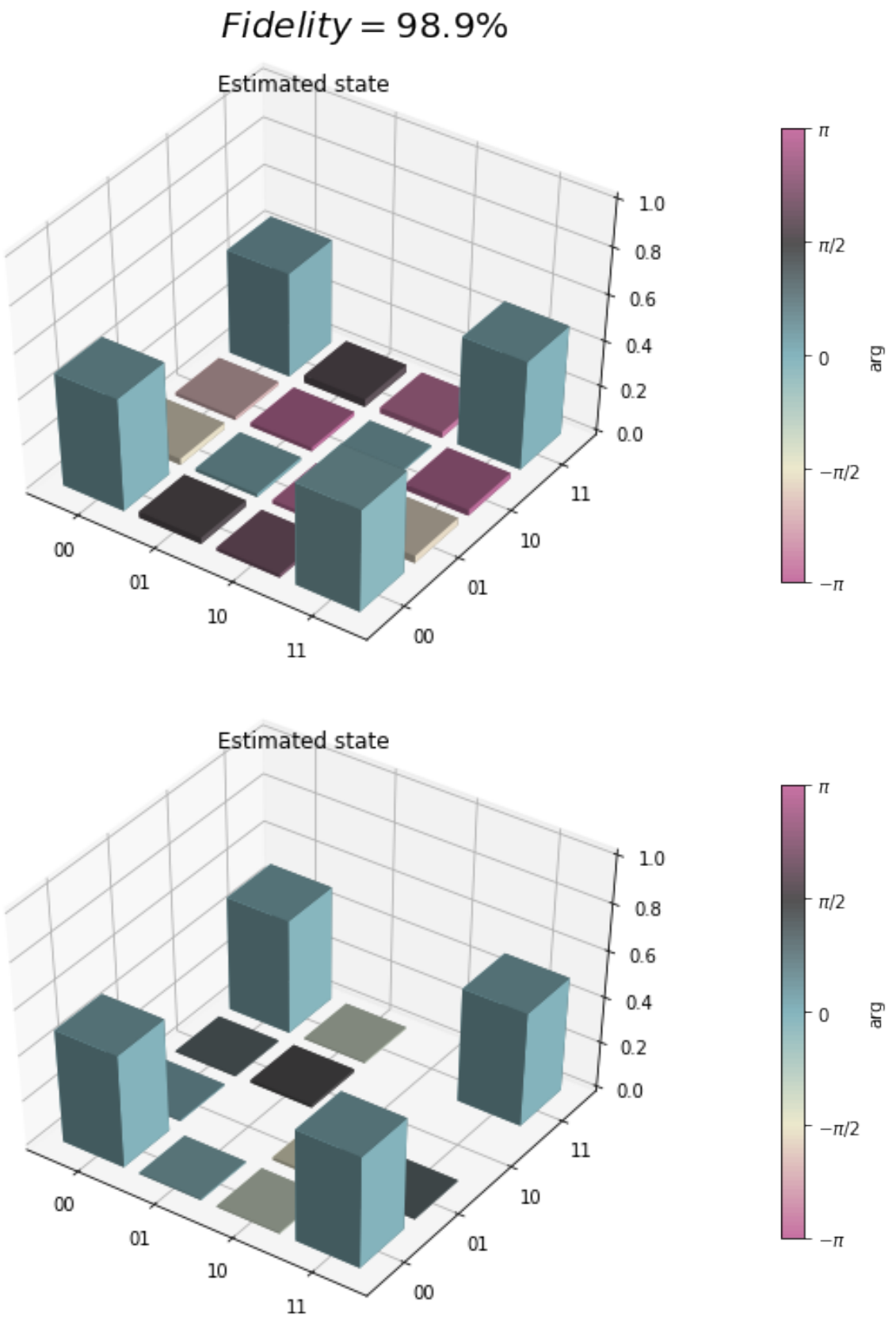
```
qubits = [6, 7]
bell_state_program = Program(RY(-np.pi/2, qubits[0]),
                             RY(np.pi/2, qubits[1]),
                             CZ(qubits[0],qubits[1]),
                             RY(-np.pi/2, qubits[1]))
```

Run on QPU & QVM, and calculate the fidelity

```
%%time
print("Running state tomography on the QPU...")
state_tomography_qpu, _, _ = do_state_tomography(bell_state_program, NUM_SAMPLES, qpu,
↪ qubits)
print("State tomography completed.")
print("Running state tomography on the QVM for reference...")
state_tomography_qvm, _, _ = do_state_tomography(bell_state_program, NUM_SAMPLES, qvm,
↪ qubits)
print("State tomography completed.")
```

```
Running state tomography on the QPU...
State tomography completed.
Running state tomography on the QVM for reference...
State tomography completed.
CPU times: user 1.18 s, sys: 84.2 ms, total: 1.27 s
Wall time: 4.6 s
```

```
state_fidelity = state_tomography_qpu.fidelity(state_tomography_qvm.rho_est)

if not SEND_PROGRAMS:
    EPS = .01
    assert np.isclose(state_fidelity, 1, EPS)

qpu_plot = state_tomography_qpu.plot();
qpu_plot.text(0.35, 0.9, r'$Fidelity={:1.1f}\%$'.format(state_fidelity*100), size=20)

state_tomography_qvm.plot();
```


$Fidelity = 98.9\%$
Estimated state
1.0
0.8
0.6
0.4
0.2
0.0
00
01
10
11
arg
π
$\pi/2$
0
$-\pi/2$
$-\pi$
Estimated state

Process tomography

Perform process tomography on a controlled-Z (CZ) gate

```
qubits = [5, 6]
CZ_PROGRAM = Program([CZ(qubits[0], qubits[1])])
print(CZ_PROGRAM)
```

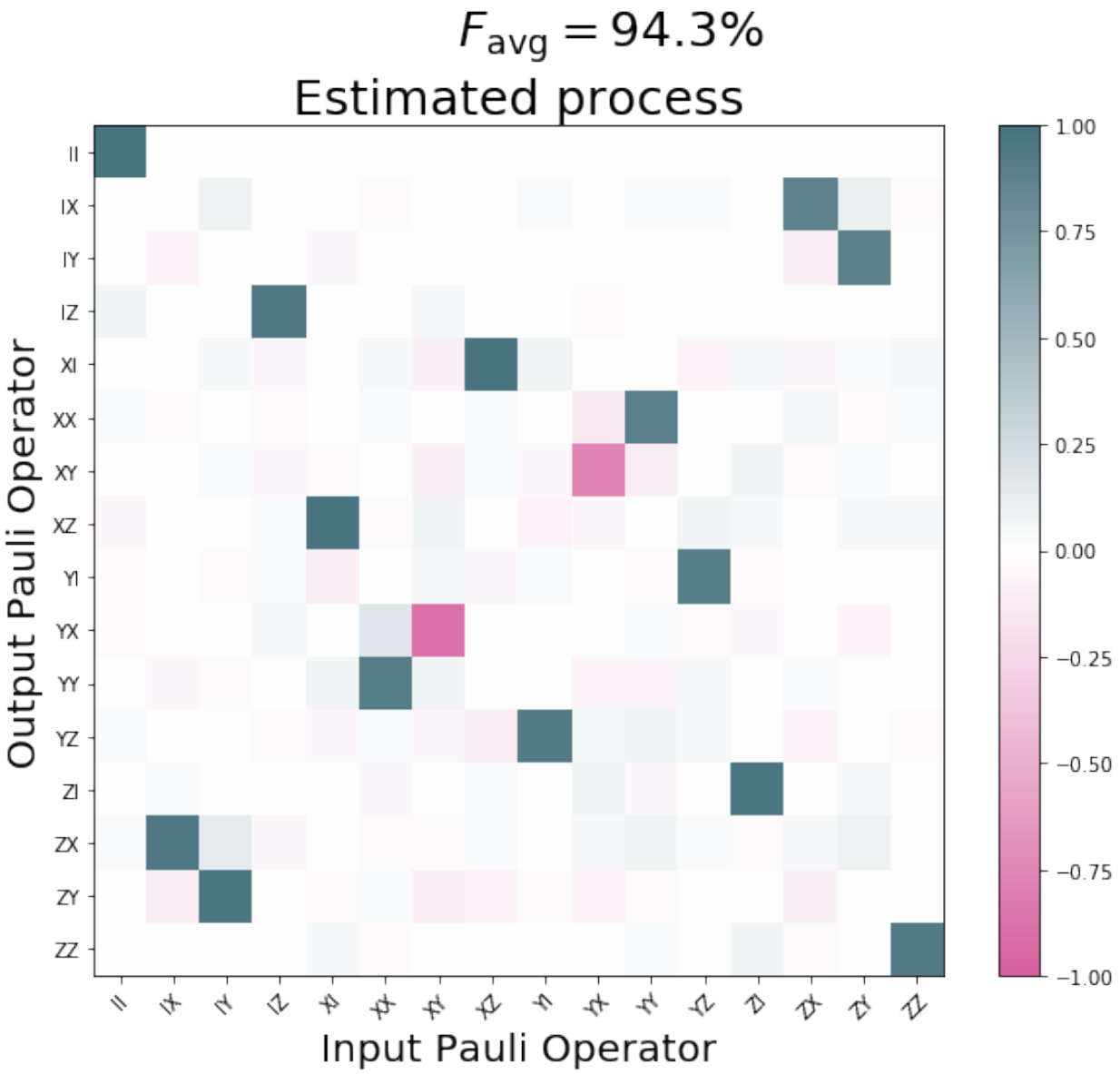
```
CZ 5 6
```

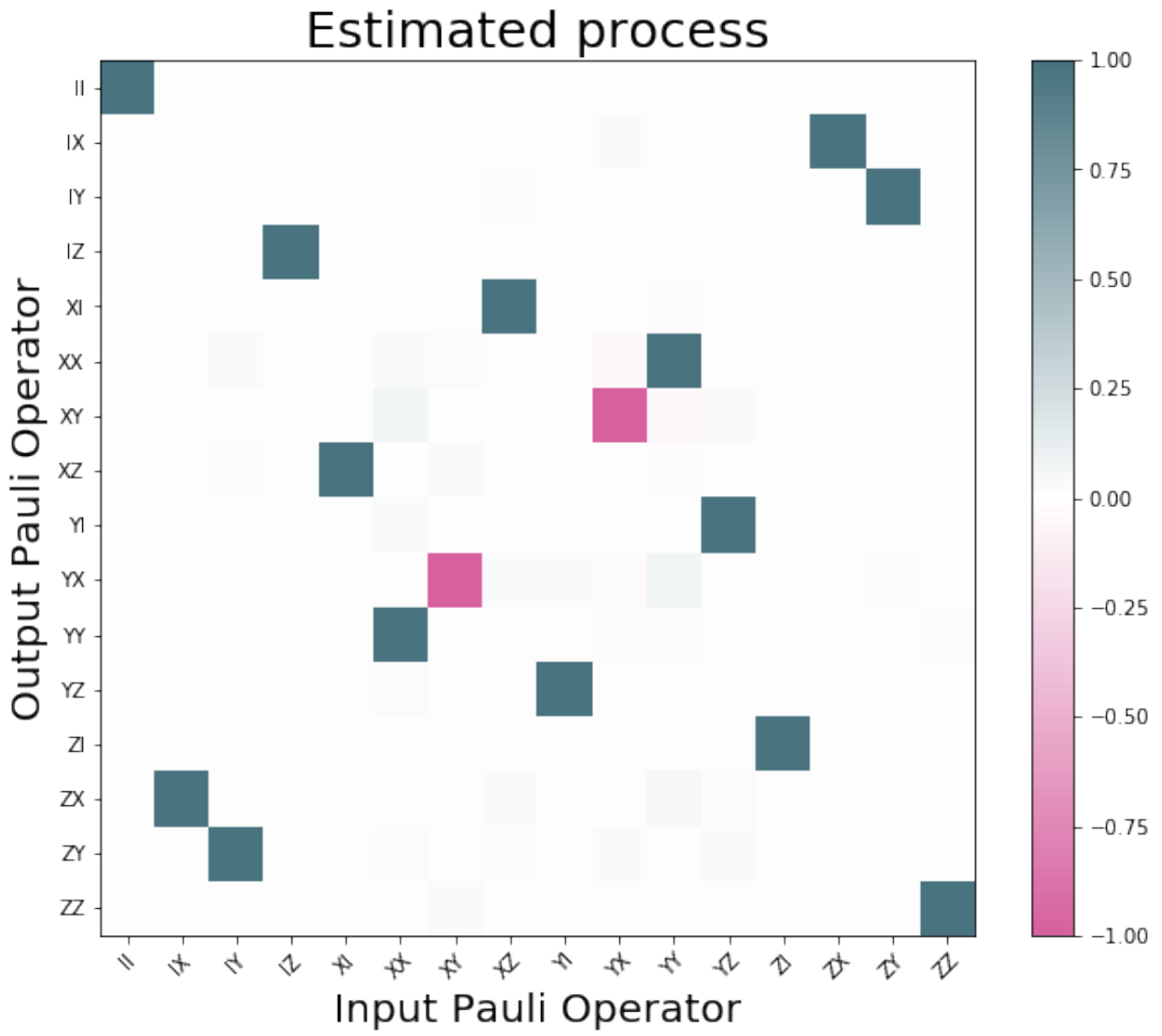
Run on the QPU & QVM, and calculate the fidelity

```
%%time
print("Running process tomography on the QPU...")
process_tomography_qpu, _, _ = do_process_tomography(CZ_PROGRAM, NUM_SAMPLES, qpu,␣
↪qubits)
print("Process tomography completed.")
print("Running process tomography on the QVM for reference...")
process_tomography_qvm, _, _ = do_process_tomography(CZ_PROGRAM, NUM_SAMPLES, qvm,␣
↪qubits)
print("Process tomography completed.")
```

```
Running process tomography on the QPU...
Process tomography completed.
Running process tomography on the QVM for reference...
Process tomography completed.
CPU times: user 16.4 s, sys: 491 ms, total: 16.8 s
Wall time: 57.4 s
```

```
process_fidelity = process_tomography_qpu.avg_gate_fidelity(process_tomography_qvm.r_
↪est)

if not SEND_PROGRAMS:
    EPS = .001
    assert np.isclose(process_fidelity, 1, EPS)

qpu_plot = process_tomography_qpu.plot();
qpu_plot.text(0.4, .95, r'$F_{{\rm avg}}={:1.1f}\%$'.format(process_fidelity*100),␣
↪size=25)

process_tomography_qvm.plot();
```


$F_{\text{avg}} = 94.3\%$
Estimated process
Output Pauli Operator
Input Pauli Operator
II
IX
IY
IZ
XI
XX
XY
XZ
YI
YX
YY
YZ
ZI
ZX
ZY
ZZ
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



1.6.9 Source Code Docs

Module for quantum state and process tomography.

Quantum state and process tomography are algorithms that take as input many copies of a quantum state or process, and output an estimate of what that state or process is. For more information, see the documentation.

exception grove.tomography.tomography.**BadReadoutPOVM**
Bases: *grove.tomography.tomography.TomographyBaseError*

Raised when the tomography analysis fails due to a bad readout calibration.

exception grove.tomography.tomography.**IncompleteTomographyError**
Bases: *grove.tomography.tomography.TomographyBaseError*

Raised when a tomography SignalTensor has circuit results that are all 0. indicating that the measurement did not complete successfully.

exception grove.tomography.tomography.**TomographyBaseError**
Bases: exceptions.Exception

Base class for errors raised during Tomography analysis.

class grove.tomography.tomography.**TomographySettings**
Bases: tuple

Create new instance of TomographySettings(constraints, solver_kwargs)

constraints
Alias for field number 0

solver_kwargs
Alias for field number 1

grove.tomography.tomography.**default_channel_ops**(*nqubits*)
Generate the tomographic pre- and post-rotations of any number of qubits as qutip operators.

Parameters **nqubits** (*int*) – The number of qubits to perform tomography on.

Returns Qutip object corresponding to the tomographic rotation.

Return type Qobj

grove.tomography.tomography.**default_rotations**(**qubits*)
Generates the Quil programs for the tomographic pre- and post-rotations of any number of qubits.

Parameters **qubits** (*list*) – A list of qubits to perform tomography on.

class grove.tomography.state_tomography.**StateTomography**(*rho_coeffs*, *pauli_basis*, *settings*)
Bases: grove.tomography.tomography.TomographyBase

Construct a StateTomography to encapsulate the result of estimating the quantum state from a quantum tomography measurement.

Parameters **r_est** (*numpy.ndarray*) – The estimated quantum state represented in a given (generalized)

Pauli basis. :param OperatorBasis pauli_basis: The employed (generalized) Pauli basis. :param TomographySettings settings: The settings used to estimate the state.

static estimate_from_ssr(*histograms*, *readout_povm*, *channel_ops*, *settings*)
Estimate a density matrix from single shot histograms obtained by measuring bitstrings in the Z-eigenbasis after application of given channel operators.

Parameters

- **histograms** (*numpy.ndarray*) – The single shot histograms, *shape=(n_channels, dim)*.
- **readout_povm** (*DiagognalPOVM*) – The POVM corresponding to the readout plus classifier.
- **channel_ops** (*list*) – The tomography measurement channels as *qutip.Qobj*'s.
- **settings** (TomographySettings) – The solver and estimation settings.

Returns The generated StateTomography object.

Return type *StateTomography*

fidelity(*other*)
Compute the quantum state fidelity of the estimated state with another state.

Parameters **other** (*qutip.Qobj*) – The other quantum state.

Returns The fidelity, a real number between 0 and 1.

Return type float

plot()
Visualize the state.

Returns The generated figure.

Return type matplotlib.Figure

plot_state_histogram(*ax*)
Visualize the complex matrix elements of the estimated state.

Parameters **ax** (*matplotlib.Axes*) – A matplotlib Axes object to plot into.

grove.tomography.state_tomography.**do_state_tomography**(*preparation_program*, *nsamples*, *cxn*, *qubits=None*, *use_run=False*)
Method to perform both a QPU and QVM state tomography, and use the latter as as reference to calculate the fidelity of the former.

Parameters

- **preparation_program** (*Program*) – Program to execute.
- **nsamples** (*int*) – Number of samples to take for the program.
- **cxn** (*QVMConnection|QPUConnection*) – Connection on which to run the program.
- **qubits** (*list*) – List of qubits for the program.

to use in the tomography analysis. :param bool use_run: If `True`, use append measurements on all qubits and use `cxn.run`

instead of `cxn.run_and_measure`.

Returns The state tomogram.

Return type *StateTomography*

grove.tomography.state_tomography.**state_tomography_programs**(*state_prep*, *qubits=None*, *rotation_generator=<function default_rotations>*)
Yield tomographic sequences that prepare a state with Quil program *state_prep* and then append tomographic rotations on the specified *qubits*. If *qubits is None*, it assumes all qubits in the program should be tomographically rotated.

Parameters

- **state_prep** (*Program*) – The program to prepare the state to be tomographed.
- **qubits** (*list|NoneType*) – A list of Qubits or Numbers, to perform the tomography on. If

None, performs it on all in state_prep. :param generator rotation_generator: A generator that yields tomography rotations to perform. :return: Program for state tomography. :rtype: Program

class grove.tomography.process_tomography.**ProcessTomography**(*r_est*, *pauli_basis*, *settings*)
Bases: `grove.tomography.tomography.TomographyBase`

Construct a ProcessTomography to encapsulate the result of estimating a quantum process from a quantum tomography measurement.

Parameters

- **r_est** (*numpy.ndarray*) – The estimated quantum process represented as a Pauli transfer matrix.
- **pauli_basis** (OperatorBasis) – The employed (generalized) Pauli basis.
- **settings** (TomographySettings) – The settings used to estimate the process.

avg_gate_fidelity(*reference_unitary*)

Compute the average gate fidelity of the estimated process with respect to a unitary process. See Chow et al., 2012,

Parameters **reference_unitary** (*(qutip.Qobj|matrix-like)*) – A unitary operator that induces a process as *rho -> other*rho*other.dag()*, alternatively a superoperator or Pauli-transfer matrix.

Returns The average gate fidelity, a real number between 1/(d+1) and 1, where d is the

Hilbert space dimension. :rtype: float

static estimate_from_ssr(*histograms*, *readout_povm*, *pre_channel_ops*, *post_channel_ops*, *settings*)

Estimate a quantum process from single shot histograms obtained by preparing specific input states and measuring bitstrings in the Z-eigenbasis after application of given channel operators.

Parameters

- **histograms** (*numpy.ndarray*) – The single shot histograms.
- **readout_povm** (DiagonalPOVM) – The POVM corresponding to readout plus classifier.
- **pre_channel_ops** (*list*) – The input state preparation channels as *qutip.Qobj*'s.
- **post_channel_ops** (*list*) – The tomography post-process channels as *qutip.Qobj*'s.
- **settings** (TomographySettings) – The solver and estimation settings.

Returns The ProcessTomography object and results from the the given data.

Return type *ProcessTomography*

plot()

Visualize the process.

Returns The generated figure.

Return type matplotlib.Figure

plot_pauli_transfer_matrix(*ax*)

Plot the elements of the Pauli transfer matrix.

Parameters **ax** (*matplotlib.Axes*) – A matplotlib Axes object to plot into.

process_fidelity(*reference_unitary*)

Compute the quantum process fidelity of the estimated state with respect to a unitary process. For non-sparse reference_unitary, this implementation this will be expensive in higher dimensions.

Parameters **reference_unitary** (*(qutip.Qobj|matrix-like)*) – A unitary operator that induces a process as `rho -> other*rho*other.dag()`, can also be a superoperator or Pauli-transfer matrix.

Returns The process fidelity, a real number between 0 and 1.

Return type float

to_chi()
Compute the chi process matrix representation of the estimated process.

Returns The process as a chi-matrix.

Rytpe qutip.Qobj

to_choi()
Compute the choi matrix representation of the estimated process.

Returns The process as a choi-matrix.

Rytpe qutip.Qobj

to_kraus()
Compute the Kraus operator representation of the estimated process.

Returns The process as a list of Kraus operators.

Rytpe List[np.array]

to_super()
Compute the standard superoperator representation of the estimated process.

Returns The process as a superoperator.

Rytpe qutip.Qobj

grove.tomography.process_tomography.**do_process_tomography**(*process*, *nsamples*, *cxn*, *qubits=None*, *use_run=False*)
Method to perform a process tomography.

Parameters

- **process** (*Program*) – Process to execute.
- **nsamples** (*int*) – Number of samples to take for the program.
- **cxn** (*QVMConnection|QPUConnection*) – Connection on which to run the program.
- **qubits** (*list*) – List of qubits for the program.

to use in the tomography analysis. :param bool use_run: If `True`, use append measurements on all qubits and use `cxn.run`

instead of `cxn.run_and_measure`.

Returns The process tomogram

Return type *ProcessTomography*

grove.tomography.process_tomography.**process_tomography_programs**(*process*, *qubits=None*, *pre_rotation_generator=<function default_rotations>*, *post_rotation_generator=<function default_rotations>*)
Generator that yields tomographic sequences that wrap a process encoded by a QUIL program *proc* in tomographic rotations on the specified *qubits*.

If *qubits is None*, it assumes all qubits in the program should be tomographically rotated.

Parameters

- **process** (*Program*) – A Quil program
- **qubits** (*list|NoneType*) – The specific qubits for which to generate the tomographic sequences
- **pre_rotation_generator** – A generator that yields tomographic pre-rotations to perform.
- **post_rotation_generator** – A generator that yields tomographic post-rotations to perform.

Returns Program for process tomography.

Return type Program

exception grove.tomography.operator_utils.**CRMBaseError**
Bases: exceptions.Exception

Base class for errors raised when the confusion rate matrix is defective.

exception grove.tomography.operator_utils.**CRMUnnormalizedError**
Bases: *grove.tomography.operator_utils.CRMBaseError*

Raised when a confusion rate matrix is not properly normalized.

exception grove.tomography.operator_utils.**CRMValueError**
Bases: *grove.tomography.operator_utils.CRMBaseError*

Raised when a confusion rate matrix contains elements not contained in the interval :math‘[0,1]‘

class grove.tomography.operator_utils.**DiagonalPOVM**
Bases: tuple

Create new instance of DiagonalPOVM(pi_basis, confusion_rate_matrix, ops)

confusion_rate_matrix
Alias for field number 1

ops
Alias for field number 2

pi_basis
Alias for field number 0

class grove.tomography.operator_utils.**OperatorBasis**(*labels_ops*)
Bases: object

Encapsulates a set of linearly independent operators.

Parameters **labels_ops** (*(list|tuple)*) – Sequence of tuples (label, operator) where label is a string and operator a qutip.Qobj operator representation.

all_hermitian()
Check if all basis operators are hermitian.

is_orthonormal()
Compute a matrix of Hilbert-Schmidt inner products for the basis operators, and see if they are orthonormal. If they are return True, else, False.

Returns True if the basis vectors represented by this OperatorBasis are orthonormal, False otherwise.

Return type bool

metric()
Compute a matrix of Hilbert-Schmidt inner products for the basis operators, update self._metric, and return the value.

Returns The matrix of inner products.

Return type numpy.matrix

product(**bases*)
Compute the tensor product with another basis.

Parameters **bases** – One or more additional bases to form the product with.

Return (OperatorBasis) The tensor product basis as an OperatorBasis object.

project_op(*op*)
Project an operator onto the basis.

Parameters **op** (*qutip.Qobj*) – The operator to project.

Returns The projection coefficients as a numpy array.

Return type scipy.sparse.csr_matrix

super_basis()
Generate the superoperator basis in which the Choi matrix can be represented.

The follows the definition in Chow et al.

Return (OperatorBasis) The super basis as an OperatorBasis object.

super_from_tm(*transfer_matrix*)
Reconstruct a super operator from a transfer matrix representation. This inverts *self.transfer_matrix(...)*.

Parameters **transfer_matrix** (*(numpy.ndarray)*) – A process in transfer matrix form.

Returns A qutip super operator.

Return type qutip.Qobj.

transfer_matrix(*superoperator*)
Compute the transfer matrix $R_{jk} = r[P_j sop(P_k)]$.

Parameters **superoperator** (*qutip.Qobj*) – The superoperator to transform.

Returns The transfer matrix in sparse form.

Return type scipy.sparse.csr_matrix

grove.tomography.operator_utils.**choi_matrix**(*pauli_tm*, *basis*)
Compute the Choi matrix for a quantum process from its Pauli Transfer Matrix.

This agrees with the definition in Chow et al. except for a different overall normalization. Our normalization agrees with that of qutip.

Parameters

- **pauli_tm** (*numpy.ndarray*) – The Pauli Transfer Matrix as 2d-array.
- **basis** (OperatorBasis) – The operator basis, typically products of normalized Paulis.

Returns The Choi matrix as qutip.Qobj.

Return type qutip.Qobj

grove.tomography.operator_utils.**is_hermitian**(*operator*)
Check if matrix or operator is hermitian.

Parameters **operator** (*(numpy.ndarray|qutip.Qobj)*) – The operator or matrix to be tested.

Returns True if the operator is hermitian.

Return type bool

grove.tomography.operator_utils.**is_projector**(*operator*)
Check if operator is a projector.

Parameters **operator** (*qutip.Qobj*) – The operator or matrix to be tested.

Returns True if the operator is a projector.

Return type bool

grove.tomography.operator_utils.**make_diagonal_povm**(*pi_basis*, *confusion_rate_matrix*)
Create a DiagonalPOVM from a `pi_basis` and the `confusion_rate_matrix` associated with a readout.

See also the grove documentation.

Parameters

- **pi_basis** (OperatorBasis) – An operator basis of rank-1 projection operators.
- **confusion_rate_matrix** (*numpy.ndarray*) – The matrix of detection probabilities conditional

on a prepared qubit state. :return: The POVM corresponding to confusion_rate_matrix. :rtype: DiagonalPOVM

grove.tomography.operator_utils.**n_qubit_ground_state**(*n*)
Construct the tensor product of *n* ground states |*0*>.

Parameters **n** (*int*) – The number of qubits.

Returns The state |*000...0*> for *n* qubits.

Return type qutip.Qobj

grove.tomography.operator_utils.**n_qubit_pauli_basis**(*n*)
Construct the tensor product operator basis of *n* PAULI_BASIS's.

Parameters **n** (*int*) – The number of qubits.

Returns The product Pauli operator basis of *n* qubits

Return type *OperatorBasis*

grove.tomography.operator_utils.**to_realimag**(*z*)
Convert a complex hermitian matrix to a real valued doubled up representation, i.e., for `Z = Z_r + 1j * Z_i` return `R(Z)`:

```
R(Z) = [ Z_r   Z_i]
       [-Z_i   Z_r]
```

A complex hermitian matrix `Z` with elementwise real and imaginary parts `Z = Z_r + 1j * Z_i` can be isomorphically represented in doubled up form as:

```
R(Z) = [ Z_r   Z_i]
       [-Z_i   Z_r]

R(X)*R(Y) = [ (X_r*Y_r-X_i*Y_i)     (X_r*Y_i + X_i*Y_r)]
            [-(X_r*Y_i + X_i*Y_r)   (X_r*Y_r-X_i*Y_i)  ]

          = R(X*Y).
```

In particular, `Z` is complex positive (semi-)definite iff `R(Z)` is real positive (semi-)definite.

Parameters **z** (*(qutip.Qobj|scipy.sparse.base.spmatrix)*) – The operator representation matrix.

Returns R(Z) the doubled up representation.

Return type scipy.sparse.csr_matrix

Utilities for encapsulating bases and properties of quantum operators and super-operators as represented by qutip.Qobj()'s.

`grove.tomography.utils.`**`basis_labels`**`(`*`n`*`)`

Generate a list of basis labels for *n* qubits, ordered from least to greatest, in big-endian format:

['00..00', '00..01', ..., '11..11']

Parameters **n** –

Returns A list of strings of length n that enumerate the n-qubit bitstrings

Return type list

`grove.tomography.utils.`**`basis_state_preps`**`(`*`*qubits`*`)`

Generate a sequence of programs that prepares the measurement basis states of some set of qubits in the order such that the qubit with highest index is iterated over the most quickly: E.g., for `qubits=(0, 1)`, it returns the circuits:

```
I_0 I_1
I_0 X_1
X_0 I_1
X_0 X_1
```

Parameters **qubits** (*list*) – Each qubit to include in the basis state preparation.

Returns Yields programs for each basis state preparation.

Return type Program

`grove.tomography.utils.`**`bitlist_to_int`**`(`*`bitlist`*`)`

Convert a binary bitstring into the corresponding unsigned integer.

Parameters **bitlist** (*list*) – A list of ones of zeros.

Returns The corresponding integer.

Return type int

`grove.tomography.utils.`**`estimate_assignment_probs`**`(`*`bitstring_prep_histograms`*`)`

Compute the estimated assignment probability matrix for a sequence of single shot histograms obtained by running the programs generated by *basis_state_preps()*.

bitstring_prep_histograms[i,j] = #number of measured outcomes j when running program i

The assignment probability is obtained by transposing and afterwards normalizing the columns.

p[j, i] = Probability to measure outcome j when preparing the state with program i.

Parameters **bitstring_prep_histograms** (*list|numpy.ndarray*) – A nested list or 2d array with shape

(d, d), where `d = 2**nqubits` is the dimension of the Hilbert space. The first axis varies over the state preparation program index, the second axis corresponds to the measured bitstring. :return: The assignment probability matrix. :rtype: numpy.ndarray

`grove.tomography.utils.`**`generated_states`**(*initial_state*, *preparations*)
Generate states prepared from channel operators acting on an initial state. Typically the channel operators will be unitary.

Parameters

- **`initial_state`** (*`qutip.Qobj`*) – The initial state as a density matrix.
- **`preparations`** (*`(list|tuple)`*) – The unitary channel operators that transform the initial state.

Returns The states generated from preparations acting on intial_state

Return type list

`grove.tomography.utils.`**`import_cvxpy`**()
Try importing the qutip module, log an error if unsuccessful.

Returns The cvxpy module if successful or None

Return type Optional[module]

`grove.tomography.utils.`**`import_qutip`**()
Try importing the qutip module, log an error if unsuccessful.

Returns The qutip module if successful or None

Return type Optional[module]

`grove.tomography.utils.`**`make_histogram`**(*samples*, *ksup*)
For a list of samples [s1, s2, ..., sN] taking on integer values from 0 to ksup-1, make a histogram of each integer's outcome and return it.

Parameters

- **`samples`** – The samples.
- **`ksup`** – The (exclusive) upper bound

Returns A histogram of outcomes.

Return type numpy.ndarray

`grove.tomography.utils.`**`notebook_mode`**(*m*)
Configure whether this module should assume that it is being run from a jupyter notebook. This sets some global variables related to how progress for long measurement sequences is indicated.

Parameters **`m`** (*`bool`*) – If True, assume to be in notebook.

Returns None

Return type NoneType

`grove.tomography.utils.`**`plot_pauli_transfer_matrix`**(*ptransfermatrix*, *ax*, *labels*, *title*)
Visualize the Pauli Transfer Matrix of a process.

Parameters

- **`ptransfermatrix`** (*`numpy.ndarray`*) – The Pauli Transfer Matrix
- **`ax`** – The matplotlib axes.
- **`labels`** – The labels for the operator basis states.

- **title** – The title for the plot

Returns The modified axis object.

Return type AxesSubplot

grove.tomography.utils.**run_in_parallel**(*programs*, *nsamples*, *cxn*, *shuffle=True*)

Take sequences of Protoquil programs on disjoint qubits and execute a single sequence of programs that executes the input programs in parallel. Optionally randomize within each qubit-specific sequence.

The programs are passed as a 2d array of Quil programs, where the (first) outer axis iterates over disjoint sets of qubits that the programs involve and the inner axis iterates over a sequence of related programs, e.g., tomography sequences, on the same set of qubits.

Parameters

- **programs** (*Union[np.ndarray,List[List[Program]]]*) – A rectangular list of lists, or a 2d array of Quil Programs. The outer list iterates over disjoint qubit groups as targets, the inner list over programs to run on those qubits, e.g., tomographic sequences.
- **nsamples** (*int*) – Number of repetitions for executing each Program.
- **cxn** (*QPUConnection|QVMConnection*) – The quantum machine connection.
- **shuffle** (*bool*) – If True, the order of each qubit specific sequence (2nd axis) is randomized Default is True.

Returns An array of 2d arrays that provide bitstring histograms for each input program. The axis of the outer array iterates over the disjoint qubit groups, the outer axis of the inner 2d array iterates over the programs for that group and the inner most axis iterates over all possible bitstrings for the qubit group under consideration.

:rtype np.array

grove.tomography.utils.**sample_assignment_probs**(*qubits*, *nsamples*, *cxn*)

Sample the assignment probabilities of qubits using nsamples per measurement, and then compute the estimated assignment probability matrix. See the docstring for estimate_assignment_probs for more information.

Parameters

- **qubits** (*list*) – Qubits to sample the assignment probabilities for.
- **nsamples** (*int*) – The number of samples to use in each measurement.
- **cxn** (*QPUConnection|QVMConnection*) – The Connection object to connect to Forest.

Returns The assignment probability matrix.

Return type numpy.ndarray

grove.tomography.utils.**sample_bad_readout**(*program*, *num_samples*, *assignment_probs*, *cxn*)

Generate *n* samples of measuring all outcomes of a Quil *program* assuming the assignment probabilities *assignment_probs* by simulating the wave function on a qvm QVMConnection *cxn*

Parameters

- **program** (*pyquil.quil.Program*) – The program.
- **num_samples** (*int*) – The number of samples
- **assignment_probs** (*numpy.ndarray*) – A matrix of assignment probabilities
- **cxn** (*QVMConnection*) – the QVM connection.

Returns The resulting sampled outcomes from assignment_probs applied to cxn, one dimensional.

Return type numpy.ndarray

grove.tomography.utils.**sample_outcomes**(*probs*, *n*)
For a discrete probability distribution probs with outcomes 0, 1, ..., k-1 draw n random samples.

Parameters

- **probs** (*list*) – A list of probabilities.
- **n** (*Number*) – The number of random samples to draw.

Returns An array of samples drawn from distribution probs over 0, ..., len(probs) - 1

Return type numpy.ndarray

grove.tomography.utils.**state_histogram**(*rho*, *ax=None*, *title=''*, *threshold=0.001*)
Visualize a density matrix as a 3d bar plot with complex phase encoded as the bar color.

This code is a modified version of an equivalent function in qutip which is released under the (New) BSD license.

Parameters

- **rho** (*qutip.Qobj*) – The density matrix.
- **ax** (*Axes3D*) – The axes object.
- **title** (*str*) – The axes title.
- **threshold** (*float*) – (Optional) minimum magnitude of matrix elements. Values below this

are hidden. :return: The axis :rtype: mpl_toolkits.mplot3d.Axes3D

grove.tomography.utils.**to_density_matrix**(*state*)
Convert a Hilbert space vector to a density matrix.

Parameters **state** (*qt.basis*) – The state to convert into a density matrix.

Returns The density operator corresponding to state.

Return type qutip.qobj.Qobj

1.7 Grover's Search Algorithm and Amplitude Amplification

1.7.1 Overview

This module implements Grover's Search Algorithm, and the more general Amplitude Amplification Algorithm. Grover's Algorithm solves the following problem:

Given a collection of basis states $\{|y\rangle_i\}$, and a quantum circuit U_w that performs the following:

$$U_w : |x\rangle |q\rangle \rightarrow |x\rangle |q \oplus f(x)\rangle$$

where $f(x) = 1$ iff $|x\rangle \in \{|y\rangle_i\}$, construct a quantum circuit that when given the uniform superposition $|s\rangle = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |x_i\rangle$ as input, produces a state $|s'\rangle$ that, when measured, produces a state $\{y_i\}$ with probability near one.

As an example, take $U_w : |x\rangle |q\rangle \rightarrow |x\rangle |q \oplus (x \cdot \text{vec}(1))\rangle$, where vec{1} is the vector of ones with the same dimension as ket{x}. In this case, $f(x) = 1$ iff $x = 1$, and so starting with the state $|s\rangle$ we hope end up with a state $|\psi\rangle$ such that $\langle\psi|\text{vec}(1)\rangle \approx 1$. In this example, $\{y_i\} = \{\text{vec}(1)\}$.

1.7.2 Algorithm and Details

Grover's Algorithm requires an oracle U_w, that performs the mapping as described above, with $f : \{0,1\}^n \to \{0,1\}^n$, and $|q\rangle$ a single ancilla qubit. We see that if we prepare the ancilla qubit $|q\rangle$ in the state $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ then U_w takes on a particularly useful action on our qubits:

$$U_w : |x\rangle |-\rangle \to \frac{1}{\sqrt{2}} |x\rangle (|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle)$$

If $f(x) = 0$, then the ancilla qubit is left unchanged, however if $f(x) = 1$ we see that the ancilla picks up a phase factor of -1. Thus, when used in conjunction with the ancilla qubit, we may write the action of the oracle circuit on the data qubits $|x\rangle$ as:

$$U_w : |x\rangle \to (-1)^{f(x)} |x\rangle$$

The other gate of note in Grover's Algorithm is the Diffusion operator. This operator is defined as:

$$\mathcal{D} := \begin{bmatrix} \frac{2}{N} - 1 & \frac{2}{N} & \cdots & \frac{2}{N} \\ \frac{2}{N} & & & \\ \vdots & & \ddots & \\ \frac{2}{N} & & & \frac{2}{N} - 1 \end{bmatrix}$$

This operator takes on its name from its similarity to a discretized version of the diffusion equation, which provided motivation for Grover[2]. The diffusion equation is given by $\frac{\partial \rho(t)}{\partial t} = \nabla \cdot \nabla \rho(t)$, where ρ is a density diffusing through space. We can discretize this process, as is described in[2], by considering N vertices on a complete graph, each of which can diffuse to $N - 1$ other vertices in each time step. By considering this process, one arrives at an equation of the form $\psi(t + \Delta t) = \mathcal{D}'\psi$ where $\mathcal{D}'$ has a form similar to $\mathcal{D}$. One might note that the diffusion equation is the same as the Schruodinger equation, up to a missing i, and in many ways it describes the diffusion of the probability amplitude of a quantum state, but with slightly different properties. From this analogy one might be led to explore how this diffusion process can be taken advantage of in a computational setting.

One property that $\mathcal{D}$ has is that it inverts the amplitudes of an input state about their mean. Thus, one way of viewing Grover's Algorithm is as follows. First, we flip the amplitude of the desired state(s) with U_w, then invert the amplitudes about their mean, which will result in the amplitude of the desired state being slightly larger than all the other amplitudes. Iterating this process will eventually result in the desired state having a significantly larger amplitude. As short example by analogy, consider the vector of all ones, $[1, 1, ..., 1]$. Suppose we want to apply a transformation that increases the value of the second input, and supresses all other inputs. We can first flip the sign to yield $[1, -1, 1, ..., 1]$ Then, if there are a large number of entries we see that the mean will be rougly one. Thus inverting the entries about the mean will yield, approximately, $[-1, 3, -1, ..., -1]$. Thus we see that this procedure, after one iteration, significantly increases the amplitude of the desired index with respect to the other indices. See[2] for more.

Given these definitions we can now describe Grover's Algorithm:

Input: $n + 1$ qubits

Algorithm:

1. Initialize them to the state $|s\rangle |-\rangle$.
2. Apply the oracle U_w to the qubits, yielding $\sum_{0}^{N-1} (-1)^{f(x)} |x\rangle |-\rangle$, where $N = 2^n$
3. Apply the n-fold Hadamard gate $H^{\otimes n}$ to $|x\rangle$
4. Apply $\mathcal{D}$
5. Apply $H^{\otimes n}$ to $|x\rangle$

[2] Lov K. Grover: "A fast quantum mechanical algorithm for database search", 1996; [http://arxiv.org/abs/quant-ph/9605043 arXiv:quant-ph/9605043].

It can be shown[1] that if this process is iterated for $\mathcal{O}(\sqrt{N})$ iterations, a measurement of $|x\rangle$ will result in one of $\{y_i\}$ with probability near one.

1.7.3 Source Code Docs

Here you can find documentation for the different submodules in amplification. grove.amplification.amplification

---

Module for amplitude amplification, for use in algorithms such as Grover's algorithm.

See G. Brassard, P. Hoyer, M. Mosca (2000) Quantum Amplitude Amplification and Estimation for more information.

`grove.amplification.amplification.`**`amplification_circuit`**(*algorithm*, *oracle*, *qubits*, *num_iter*, *decompose_diffusion=False*)

Returns a program that does `num_iter` rounds of amplification, given a measurement-less algorithm, an oracle, and a list of qubits to operate on.

Parameters

- **`algorithm`** (*`Program`*) – A program representing a measurement-less algorithm run on qubits.
- **`oracle`** (*`Program`*) – An oracle maps any basis vector `|psi>` to either `+|psi>` or `-|psi>` depending on whether `|psi>` is in the desirable subspace or the undesirable subspace.
- **`qubits`** (*`Sequence`*) – the qubits to operate on
- **`num_iter`** (*`int`*) – number of iterations of amplifications to run
- **`decompose_diffusion`** (*`bool`*) – If True, decompose the Grover diffusion gate into two qubit gates. If False, use a defgate to define the gate.

Returns The amplified algorithm.

Return type Program

`grove.amplification.amplification.`**`decomposed_diffusion_program`**(*qubits*)

Constructs the diffusion operator used in Grover's Algorithm, acted on both sides by an a Hadamard gate on each qubit. Note that this means that the matrix representation of this operator is diag(1, -1, ..., -1). In particular, this decomposes the diffusion operator, which is a $2 * *len(qubits) imes 2 * *len(qubits)$ sparse matrix, into

:math:‘mathcal{O}(len(qubits)**2) single and two qubit gates.

See C. Lavor, L.R.U. Manssur, and R. Portugal (2003) Grover's Algorithm: Quantum Database Search for more information.

Parameters **`qubits`** – A list of ints corresponding to the qubits to operate on. The operator operates on bistrings of the form `|qubits[0], ..., qubits[-1]>`.

`grove.amplification.amplification.`**`diffusion_program`**(*qubits*)

grove.amplification.grover

Module for Grover's algorithm.

[1] Nielsen, M.A. and Chuang, I.L. Quantum computation and quantum information. Cambridge University Press, 2000. Chapter 6.

class grove.amplification.grover.**Grover**
Bases: object

This class contains an implementation of Grover's algorithm using pyQuil. See these notes by Dave Bacon for more information.

find_bitstring(*cxn*, *bitstring_map*)
Runs Grover's Algorithm to find the bitstring that is designated by `bistring_map`.

In particular, this will prepare an initial state in the uniform superposition over all bit- strings, an then use Grover's Algorithm to pick out the desired bitstring.

Parameters
- **cxn** (*QVMConnection*) – the connection to the Rigetti cloud to run pyQuil programs.
- **bitstring_map** (*Dict[String, Int]*) – a mapping from bitstrings to the phases that the oracle should impart on them. If the oracle should "look" for a bitstring, it should have a `-1`, otherwise it should have a `1`.

Returns Returns the bitstring resulting from measurement after Grover's Algorithm.
Return type str

static oracle_grover(*oracle*, *qubits*, *num_iter=None*)
Implementation of Grover's Algorithm for a given oracle.

Parameters
- **oracle** (*Program*) – An oracle defined as a Program. It should send $|x\rangle$ to $(-1)^{f(x)}|x\rangle$, where the range of f is {0, 1}.
- **qubits** (*list[int or Qubit]*) – List of qubits for Grover's Algorithm.
- **num_iter** (*int*) – The number of iterations to repeat the algorithm for. The default is the integer closest to $\frac{\pi}{4}\sqrt{N}$, where N is the size of the domain.

Returns A program corresponding to the desired instance of Grover's Algorithm.
Return type Program

1.8 Bernstein-Vazirani Algorithm

1.8.1 Overview

This module emulates the Bernstein-Vazirani Algorithm.

The problem is summarized as follows. Given a function \(f\) such that

$$f:\{0,1\}^n \rightarrow \{0,1\} \ \mathbf{x} \rightarrow \mathbf{a}\cdot\mathbf{x} + b\pmod{2} \ (\mathbf{a}\in\{0,1\}^n, b\in\{0,1\})$$

determine \(\mathbf{a}\) and \(b\) with as few queries to \(f\) as possible.

Classically, \(n+1\) queries are required: \(n\) for \(\mathbf{a}\) and one for \(b\). However, using a quantum algorithm, only \(2\) queries are required: just one each both \(\mathbf{a}\) and \(b\).

This module is able to generate and run a program to determine \(\mathbf{a}\) and \(b\), given an oracle. It also has the ability to prescribe a way to generate an oracle out of quantum circuit components, given \(\mathbf{a}\) and \(b\).

More details about the Bernstein-Vazirani Algorithm can be found in reference[1].

1.8.2 Source Code Docs

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

[1] http://pages.cs.wisc.edu/~dieter/Courses/2010f-CS880/Scribes/04/lecture04.pdf

grove.bernstein_vazirani.bernstein_vazirani

Module for the Bernstein-Vazirani Algorithm. For more information, see *[Loceff2015]*

class grove.bernstein_vazirani.bernstein_vazirani.**BernsteinVazirani**
Bases: object

This class contains an implementation of the Bernstein-Vazirani algorithm using pyQuil. For more references see the documentation

check_solution()
Checks if the the found solution correctly reproduces the input.

Returns True if solution correctly reproduces input bitstring map

Return type Bool

get_solution()
Returns the solution of the BV algorithm

Returns a tuple of string corresponding to the dot-product partner vector and the bias term

Return type Tuple[String, String]

run(*cxn*, *bitstring_map*)
Runs the Bernstein-Vazirani algorithm.

Given a connection to a QVM or QPU, find the $\mathbf{a}$ and b corresponding to the function represented by the oracle function that will be constructed from the bitstring map.

Parameters

- **cxn** (*Connection*) – connection to the QPU or QVM
- **String] bitstring_map** (*Dict[String,*) – a truth table describing the boolean function, whose dot-product vector and bias is to be found

Return type *BernsteinVazirani*

grove.bernstein_vazirani.bernstein_vazirani.**create_bv_bitmap**(*dot_product_vector*, *dot_product_bias*)
This function creates a map from bitstring to function value for a boolean formula f with a dot product vector a and a dot product bias b

$$f : \{0,1\}^n \rightarrow \{0,1\}$$
$$\mathbf{x} \rightarrow \mathbf{a} \cdot \mathbf{x} + b \pmod 2$$
$$(\mathbf{a} \in \{0,1\}^n, b \in \{0,1\})$$

Parameters

- **dot_product_vector** (*String*) – a string of 0's and 1's that represents the dot-product partner in f
- **dot_product_bias** (*String*) – 0 or 1 as a string representing the bias term in f

Returns A dictionary containing all possible bitstring of length equal to a and the function value f

Return type Dict[String, String]

References

1.9 Simon's Algorithm

1.9.1 Overview

This module emulates Simon's Algorithm.

Simon's problem is summarized as follows. A function \(f :\{0,1\}^n\rightarrow\{0,1\}^n\) is promised to be either one-to-one, or two-to-one with some nonzero \(n\)-bit mask \(s\). The latter condition means that for any two different \(n\)-bit numbers \(x\) and \(y\), \(f(x)=f(y)\) if and only if \(x\oplus y = s\). The problem then is to determine whether \(f\) is one-to-one or two-to-one, and, if the latter, what the mask \(s\) is, in as few queries to \(f\) as possible.

The problem statement and algorithm can be explored further, at a high level, in reference[1]. The implementation of the algorithm in this module, however, follows[2].

1.9.2 Algorithm and Details

This algorithm involves a quantum component and a classical component. The quantum part follows similarly to other blackbox oracle algorithms. First, assume a blackbox oracle \(U_f\) is available with the property $$U_f\vert x\rangle\vert y\rangle = \vert x\rangle\vert y\oplus f(x)\rangle$$

where the top \(n\) qubits \(\vert x \rangle\) are the input, and the bottom \(n\) qubits \(\vert y \rangle\) are called ancilla qubits.

The input qubits are prepared with the ancilla qubits into the state $$(H^{\otimes n} \otimes I^{\otimes n})\vert 0\rangle^{\otimes n}\vert 0\rangle^{\otimes n} = \vert +\rangle^{\otimes n}\vert 0\rangle^{\otimes n}$$ and sent through a blackbox gate \(U_f\). Then, the Hadamard-Walsh transform \(H^{\otimes n}\) is applied to the \(n\) input qubits, resulting in the state given by $$(H^{\otimes n} \otimes I^{\otimes n})U_f\vert +\rangle^{\otimes n}\vert 0\rangle^{\otimes n}$$

It turns out the resulting \(n\) input qubits are in a uniform random state over the space killed by (modulo \(2\), bitwise) dot product with \(s\). This covers the one-to-one case as well, if one considers it to be the degenerate \(s=0\) case.

Suppose we then measured the \(n\) input qubits, calling the bitstring output \(y\). The above property then requires \(s\cdot y = 0\). The space of \(y\) that satisfies this is \(n-1\) dimensional. By running this quantum subroutine several times, \(n-1\) nonzero linearly independent bitstrings \(y_i\), \(i = 0, \ldots, n-2\), can be found, each orthogonal to \(s\).

This gives a system of \(n-1\) equations, with \(n\) unknowns for finding \(s\). One final nonzero bitstring \(y^{\prime}\) can be classically found that is linearly independent to the other \(y_i\), but with the property that \(s\cdot y^{\prime} = 1\). The combination of \(y^{\prime}\) and the \(y_i\) give a system of \(n\) independent equations that can then be solved for \(s\).

By using a clever implementation of Gaussian Elimination and Back Substitution for mod-2 equations, as outlined in Reference[2], \(s\) can be found relatively quickly. By then sending separate input states \(\vert 0\rangle\) and \(\vert s\rangle\) through the blackbox \(U_f\), we can find whether or not \(f(0) = f(s)\) (in fact, any pair \(\vert x\rangle\) and \(\vert x\oplus s\rangle\) will do as well). If so, we conclude \(f\) is two-to-one with mask \(s\); otherwise, \(f\) is one-to-one.

Overall, this algorithm can be solved in \(O(n^3)\), i.e., polynomial, time, whereas the best classical algorithm requires exponential time.

[1] http://pages.cs.wisc.edu/~dieter/Courses/2010f-CS880/Scribes/05/lecture05.pdf
[2] http://lapastillaroja.net/wp-content/uploads/2016/09/Intro_to_QC_Vol_1_Loceff.pdf

1.9.3 Source Code Docs

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

grove.simon.simon

Module for Simon's Algorithm. For more information, see *[Simon1995]*, *[Loceff2015]*, *[Watrous2006]*

class grove.simon.simon.**Simon**
Bases: object

This class contains an implementation of Simon's algorithm using pyQuil. For more references see the documentation

find_mask(*cxn*, *bitstring_map*)
Runs Simon's mask_array algorithm to find the mask.

Parameters

- **cxn** (*QVMConnection*) – the connection to the Rigetti cloud to run pyQuil programs
- **String] bitstring_map** (*Dict[String,*) – a truth table describing the boolean function, whose period is to be found.

Returns Returns the mask of the bitstring map or raises an Exception if the mask cannot be found.

Return type String

grove.simon.simon.**create_1to1_bitmap**(*mask*)
A helper to create a bit map function (as a dictionary) for a given mask. E.g. for a mask $m = 10$ the return is a dictionary:

```
>>> create_1to1_bitmap('10')
... {
...     '00': '10',
...     '01': '11',
...     '10': '00',
...     '11': '01'
... }
```

Parameters **mask** (*String*) – binary mask as a string of 0's and 1's

Returns dictionary containing a mapping of all possible bit strings of the same length as the mask's string and their mapped bit-string value

Return type Dict[String, String]

grove.simon.simon.**create_valid_2to1_bitmap**(*mask*, *random_seed=None*)
A helper to create a 2-to-1 binary function that is invariant with respect to the application of a specified XOR bitmask. This property must be satisfied if a 2-to-1 function is to be used in Simon's algorithm

More explicitly, such a 2-to-1 function f must satisfy $f(x) = f(x \oplus m)$ where m is a bit mask and $\oplus$ denotes the bit wise XOR operation. An example of such a function is the truth-table

x	f(x)
000	101
001	010
010	000
011	110
100	000
101	110
110	101
111	010

Note that, e.g. both *000* and *110* map to the same value *101* and $000 \oplus 110 = 110$. The same holds true for other pairs.

Parameters

- **mask** (*String*) – mask input that defines the periodicity of f
- **random_seed** (*Integer*) – (optional) integer to set numpy.random.seed parameter.

Returns dictionary containing the truth table of a valid 2-to-1 boolean function

Return type Dict[String, String]

References

1.10 Deutsch-Jozsa Algorithm

1.10.1 Overview

The Deutsch-Jozsa algorithm can determine whether a function mapping all bitstrings to a single bit is constant or balanced, provided that it is one of the two. A constant function always maps to either 1 or 0, and a balanced function maps to 1 for half of the inputs and maps to 0 for the other half. Unlike any deterministic classical algorithm, the Deutsch-Jozsa Algorithm can solve this problem with a single iteration, regardless of the input size. It was one of the first known quantum algorithms that showed an exponential speedup, albeit against a deterministic (non-probabilistic) classical compuetwr, and with access to a blackbox function that can evaluate inputs to the chosen function.

1.10.2 Algorithm and Details

This algorithm takes as input n qubits in state $|x\rangle$, an ancillary qubit in state $|q\rangle$, and additionally a quantum circuit U_w that performs the following:

$$U_w : |x\rangle |q\rangle \rightarrow |x\rangle |f(x) \oplus q\rangle$$

In the case of the Deutsch-Jozsa algorithm, the function f is some function mapping from bitstrings to bits:

$$f : \{0,1\}^n \rightarrow \{0,1\}$$

and is assumed to either be textit{constant} or textit{balanced}. Constant means that on all inputs f takes on the same value, and balanced means that on half of the inputs f takes on one value, and on the other half f takes on a different value. (Here the value is restricted to $\{0,1\}$)

We can then describe the algorithm as follows:

Input: $n+1$ qubits

Algorithm:

1. Prepare the textit{ancilla} ($|q\rangle$ above) in the $|1\rangle$ state by performing an X gate.
2. Perform the $n+1$-fold Hadamard gate $H^{\otimes n+1}$ on the $n+1$ qubits.
3. Apply the circuit U_w.
4. Apply the n-fold Hadamard gate $H^{\otimes n}$ on the data qubits, $|x\rangle$.
5. Measure $|x\rangle$. If the result is all zeroes, then the function is constant. Otherwise, it is balanced.

1.10.3 Implementation Notes

The oracle in the Deutsch-Jozsa module is not implemented in such a way that calling *Deutsch_Jozsa.is_constant()* will yield an exponential speedup over classical implementations. To construct the quantum algorithm that is executing on the QPU we use a Quil *defgate*, which specifies the circuit U_w as its action on the data qubits $|x\rangle$. This matrix is exponentially large, and thus even generating the program will take exponential time.

1.10.4 Source Code Docs

Here you can find documentation for the different submodules in deutsch-jozsa.

grove.deutsch_jozsa.deutsch_jozsa.py

Module for the Deutsch-Jozsa Algorithm.

class grove.deutsch_jozsa.deutsch_jozsa.**DeutschJosza**

Bases: object

is_constant(*cxn*, *bitstring_map*)

Computes whether bitstring_map represents a constant function, given that it is constant or balanced. Constant means all inputs map to the same value, balanced means half of the inputs maps to one value, and half to the other.

Parameters

- **cxn** (*QVMConnection*) – The connection object to the Rigetti cloud to run pyQuil programs.
- **bitstring_map** – A dictionary whose keys are bitstrings, and whose values are bits represented as strings.

Returns True if the bitstring_map represented a constant function, false otherwise.

Return type bool

static unitary_function(*mappings*)

Creates a unitary transformation that maps each state to the values specified in mappings.

Some (but not all) of these transformations involve a scratch qubit, so room for one is always provided. That is, if given the mapping of n qubits, the calculated transformation will be on n + 1 qubits, where the 0th is the scratch bit and the return value of the function is left in the 1st.

Parameters **mappings** (*Dict[String, Int]*) – Dictionary of the mappings of f(x) on all length n bitstrings, e.g.

```
>>> {'00': '0', '01': '1', '10': '1', '11': '0'}
```

Returns ndarray representing specified unitary transformation.

Return type np.ndarray

1.11 Arbitrary State Generation

1.11.1 Overview

This module is concerned with making a program that can generate an arbitrary state. In particular, if one is given a nonzero complex vector \(\mathbf{a}\in\mathbb{C}^N\) with components \(a_i\), the goal is to produce a program that takes in the state \(\vert 0\rangle\) and outputs the state

$$ \vert \Psi \rangle = \sum_{i=0}^{N-1}\frac{a_i}{\vert \mathbf{a}\vert} \vert i\rangle $$

where \(\vert i\rangle\) is interpreted by taking \(i\) in its binary representation.

This problem is approached in two different ways in this module, and will be described in the sections to follow. The first is to directly construct a circuit using a sequence of CNOT, rotation, Hadamard, and phase gates, that produces the desired state. The second is to construct a unitary matrix that could be decomposed into different circuits depending on which gates one would see fit.

More details on the first approach can be found in references[1] and[2].

1.11.2 Arbitrary State Generation via Specific Circuit

The method in this approach follows the algorithm described in[1]. The idea is to imagine beginning with the desired state \(\vert \Psi \rangle\). First, controlled RZ gates are used to unify the phases of the coefficients of consecutive pairs of basis states. Next, controlled RY gates are used to unify the magnitudes (or probabilities) of those pairs of basis states, and hence unify the coefficients altogether. Next, a swap is performed so that in subsequent steps, multiple pairs of consecutive states will have the same pair of coefficients. This process can be repeated, with each successive step of rotations requiring fewer controls due to the interspersed swaps. Finally, with all states having the same coefficient, the Hadamard gate can be applied to all the qubits to select out the \(\vert 0 \rangle\) state. Lastly, a combination of a PHASE gate and RZ gate can be applied to remove the global phase. The reverse of this program, which can be found by applying all gates in reverse and all rotations with negated angles, this provides the desired program for arbitrary state generation.

One key part of this algorithm is that each rotation step is uniformly controlled. This has a relatively efficient decomposition into CNOTs and uncontrolled rotations, and is the subject of reference[2].

1.11.3 Arbitrary State Generation via Unitary Matrix

The method in this approach is to create a unitary operator mapping the ground state of a set of qubits to the desired outcome state. This requires constructing a unitary matrix whose leftmost column is \(\vert \Psi \rangle\). By replacing the left column of the identity matrix with \(\vert \Psi \rangle\) and then QR factorizing it, one can construct such a matrix.

[1] http://140.78.161.123/digital/2016_ismvl_logic_synthesis_quantum_state_generation.pdf
[2] https://arxiv.org/pdf/quant-ph/0407010.pdf

1.11.4 Source Code Docs

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

grove.arbitrary_state.arbitrary_state

Class for generating a program that can generate an arbitrary quantum state. References are available at:

- http://140.78.161.123/digital/2016_ismvl_logic_synthesis_quantum_state_generation.pdf
- https://arxiv.org/pdf/quant-ph/0407010.pdf

Note that the algorithm used creates a circuit that begins with a target state and brings it to the all zero state. Thus, many of this module's functions involve finding gates to be applied in the reversed circuit.

`grove.alpha.arbitrary_state.arbitrary_state.`**`create_arbitrary_state`**(*vector*, *qubits=None*)

This function makes a program that can generate an arbitrary state.

Applies the methods described in references above.

Given a complex vector $\mathbf{a}$ with components a_i (i ranging from 0 to $N-1$), produce a program that takes in the state $|0\rangle$ and outputs the state

$$\sum_{i=0}^{N-1} \frac{a_i}{|\mathbf{a}|}|i\rangle$$

where i is given in its binary expansion.

Parameters

- **`vector`** (*`1darray`*) – the vector to put into qubit form.
- **`qubits`** (*`list(int)`*) – Which qubits to encode the vector into. Must contain at least the minimum number of qubits n needed for all elements of vector to be present as a coefficient in the final state. If more than n are provided, only the first n will be used. If no list is provided, the default will be qubits $0, 1, \ldots, n-1$.

Returns a program that takes in $|0\rangle^{\otimes n}$ and produces a state that represents this vector, as described above.

Return type Program

`grove.alpha.arbitrary_state.arbitrary_state.`**`get_cnot_control_positions`**(*k*)

Returns a list of positions for the controls of the CNOTs used when decomposing uniformly controlled rotations, as outlined in arXiv:quant-ph/0407010.

Referencing Fig. 2 in the aforementioned paper, this method uses the convention that, going up from the target qubit, the control qubits are labelled $1, 2, \ldots, k$, where k is the number of control qubits. The returned list provides the qubit that controls each successive CNOT, in order from left to right.

Parameters **`k`** (*`int`*) – the number of control qubits

Returns the list of positions of the controls

Return type list

grove.alpha.arbitrary_state.arbitrary_state.**get_reversed_unification_program**(*angles*, *control_indices*, *target*, *controls*, *mode*)

Gets the Program representing the reversed circuit for the decomposition of the uniformly controlled rotations in a unification step.

If n is the number of controls, the indices within control indices must range from 1 to n, inclusive. The length of control_indices and the length of angles must both be 2^n.

Parameters

- **angles** (*list*) – The angles of rotation in the the decomposition, in order from left to right
- **control_indices** (*list*) – a list of positions for the controls of the CNOTs used when decomposing uniformly controlled rotations; see get_cnot_control_positions for labelling conventions.
- **target** (*int*) – Index of the target of all rotations
- **controls** (*list*) – Index of the controls, in order from bottom to top.
- **mode** (*str*) – The unification mode. Is either 'phase', corresponding to controlled RZ rotations, or 'magnitude', corresponding to controlled RY rotations.

Returns The reversed circuit of this unification step.

Return type Program

grove.alpha.arbitrary_state.arbitrary_state.**get_rotation_parameters**(*phases*, *magnitudes*)

Simulates one step of rotations.

Given lists of phases and magnitudes of the same length N, such that $N = 2^n$ for some positive integer n, finds the rotation angles required for one step of phase and magnitude unification.

Parameters

- **phases** (*list*) – real valued phases from $-\pi$ to π.
- **magnitudes** (*list*) – positive, real value magnitudes such that the sum of the square of each magnitude is 2^{-m} for some nonnegative integer m.

Returns

A tuple t of four lists such that

- t[0] are the z-rotations needed to unify adjacent pairs of phases
- t[1] are the y-rotations needed to unify adjacent pairs of magnitudes
- t[2] are the updated phases after these rotations are applied
- t[3] are the updated magnitudes after these rotations are applied

Return type tuple

grove.alpha.arbitrary_state.arbitrary_state.**get_uniformly_controlled_rotation_matrix**(*k*)

Returns the matrix represented by M_{ij} in arXiv:quant-ph/0407010.

This matrix converts the angles of k-fold uniformly controlled rotations to the angles of the efficient gate decomposition.

Parameters **k** (*int*) – number of control qubits

Returns the matrix M_{ij}

Return type 2darray

grove.arbitrary_state.unitary_operator

Module for creating a unitary operator for encoding any complex vector into the wavefunction of a quantum state. For example, the input vector $[a, b, c, d]$ would result in the state

$$a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle$$

grove.alpha.arbitrary_state.unitary_operator.**fix_norm_and_length**(*vector*)
Create a normalized and zero padded version of vector.

Parameters **vector** (*1darray*) – a vector with at least one nonzero component.

Returns a vector that is the normalized version of vector, padded at the end with the smallest number of 0s necessary to make the length of the vector 2^m for some positive integer m.

Return type 1darray

grove.alpha.arbitrary_state.unitary_operator.**get_bits_needed**(*n*)
Calculates the smallest positive integer m for which $2^m \geq n$.

Parameters **n** (*int*) – A positive integer

Returns The positive integer m, as specified above

Return type int

grove.alpha.arbitrary_state.unitary_operator.**unitary_operator**(*state_vector*)
Uses QR factorization to create a unitary operator that can encode an arbitrary normalized vector into the wavefunction of a quantum state.

Assumes that the state of the input qubits is to be expressed as

$$(1, 0, \ldots, 0)^T$$

Parameters **array state_vector** (*1d*) – Normalized vector whose length is at least two and a power of two.

Returns Unitary operator that encodes state_vector

Return type 2d array

References

CHAPTER 2

Indices and Tables

- genindex
- modindex
- search

Bibliography

[MLE] https://en.wikipedia.org/wiki/Maximum_likelihood_estimation

[Chow] Chow et al. https://doi.org/10.1103/PhysRevLett.109.060501

[Jeffrey] Jeffrey et al. https://doi.org/10.1103/PhysRevLett.112.190504

[Magesan] Magesan et al. http://dx.doi.org/10.1103/PhysRevLett.114.200501

[POVM] https://en.wikipedia.org/wiki/POVM

[ConfusionMatrix] https://en.wikipedia.org/wiki/confusion_matrix

[QuantumChannel] https://en.wikipedia.org/wiki/Quantum_channel

[QuantumStateFidelity] https://en.wikipedia.org/wiki/Fidelity_of_quantum_states

[QuantumTraceDistance] https://en.wikipedia.org/wiki/Trace_distance

[GST] Blume-Kohout et al. https://www.osti.gov/scitech/biblio/1345878

[Loceff2015] Loceff, M. (2015), "A Course in Quantum Computing for the Community College", Volume 1, Chapter 18, p 484-541.

[Simon1995] Simon, D.R. (1995), "On the power of quantum computation", 35th Annual Symposium on Foundations of Computer Science, Proceedings, p. 116-123.

[Loceff2015] Loceff, M. (2015), "A Course in Quantum Computing for the Community College", Volume 1, Chapter 18, p 484-541.

[Watrous2006] Watrous, J. (2006), "Simon's Algorithm", University of Calgary CPSC 519/619: Quantum Computation, Lecture 6.

Python Module Index

g

Index