Grove Documentation

Release 1.1.0

Rigetti Quantum Computing

Sep 22, 2017

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.

Installation and Getting Started

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`.

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)*

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.

Quantum Teleportation

Quantum teleportation is a method for transmitting the information of a qubit from one location to another. The process relies on a previously shared entangled state between the two locations, and classical communication.

Overview

In the canonical description of quantum teleportation[1][2] two parties (Alice and Bob) are trying to transmit a state from one to another. They start with an Alice having half of an entangled bell pair (Bob having the other half) and Alice with a third qubit that she would like to transmit to Bob. Alice then entangles the third qubit with her Bell pair qubit, measures both her qubits, and sends the measurement result $\{0, 1\}^2$ to Bob. Bob uses the classical information from Alice to fix up his state and now has his qubit in the state of Alice's original qubit she wanted to transfer.

Given that Alice's data qubit is labeled 0 and the Bell pair qubits are labeled 1 and 2 (Bob having the qubit labeled 2), a Quil program performing the transfer is as follows:

```
CNOT 0 1
H 0
MEASURE 0 [0]
MEASURE 1 [1]
JUMP-UNLESS @ NOX [1]
X 2
LABEL @NOX
JUMP-UNLESS @NOZ [0]
Z 2
LABEL @NOZ
```

The Bell state preparation can be prepended to the above Quil with:

```
H 1
CNOT 1 2
```

[1] Nielsen, Michael A., and Isaac L. Chuang. Quantum Computation and Quantum Information. Cambridge University Press, 2010.
[2] Wikipedia contributors. "Quantum teleportation." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 4 Jan. 2017. Web. 4 Jan. 2017.

Teleportation in pyQuil

We have included pyQuil code that programatically generates a teleportation program between any two qubits using one ancilla qubit. The ancilla qubit is the piece of the Bell pair that Alice holds. Check the teleportation.py source code for examples.

Source Code Docs

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

grove.teleport.teleportation

`grove.teleport.teleportation.`**`make_bell_pair`**(*q1*, *q2*)
Makes a bell pair between qubits q1 and q2

`grove.teleport.teleportation.`**`teleport`**(*start_index*, *end_index*, *ancilla_index*)
Teleport a qubit from start to end using an ancilla qubit

References

Variational-Quantum-Eigensolver (VQE)

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

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 $\vec{\theta}$.
2. Iterate until convergence.

Practically, the quantum subroutine of VQE amounts to preparing a state based off of a set of parameters $\vec{\theta}$ 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.

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.SyncConnection()
```

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 $\Psi(\vec{\theta})$. 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 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 $\langle \Psi(\vec{\theta}) \mid H \mid \Psi(\vec{\theta}) \rangle$ we calculate for fixed parameters of $\vec{\theta}$.

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

```
-0.4161468365471423
```

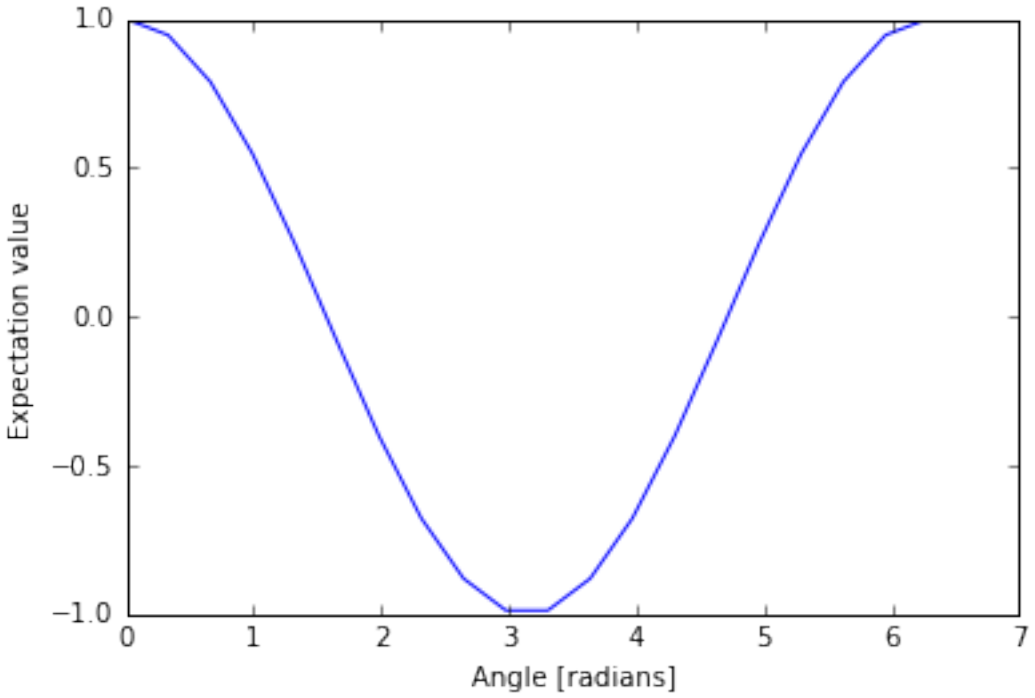
The expectation value was calculated by running the pyQuil program output from `small_ansatz`, saving the wavefunction, and using that vector to calculate the expectation value. We can sample the wavefunction as you would on a quantum computer by passing an integer, instead of *None*, as the samples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

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

```
-0.42900000000000005
```

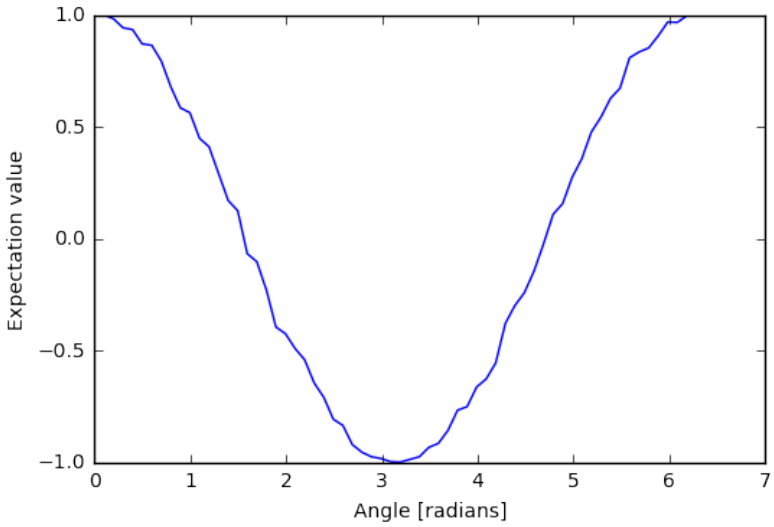
We can loop over a range of these angles and plot the expectation value.

```
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])}
```

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.SyncConnection(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.

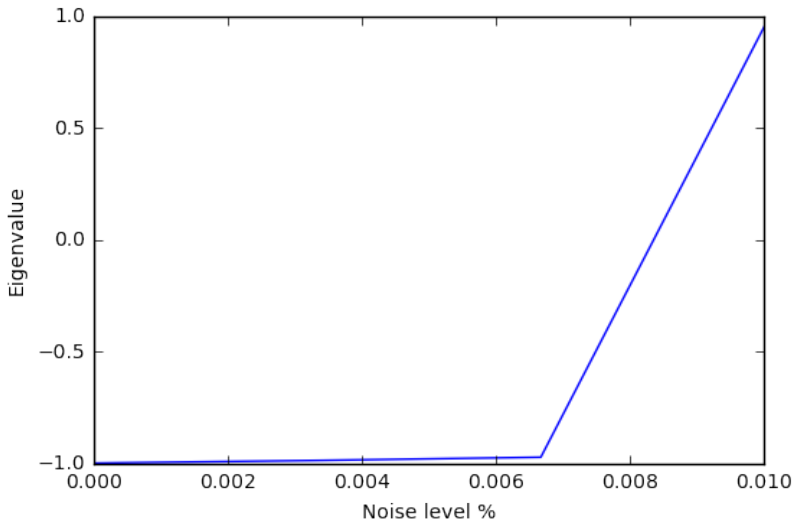
```
vqe_inst.minimizer_kwargs = {'method': 'Nelder-mead', 'options': {'initial_simplex':␣
↪np.array([[0.0], [0.05]]), 'xatol': 1.0e-2}}
result = vqe_inst.vqe_run(small_ansatz, hamiltonian, initial_angle, samples=10000,␣
↪qvm=noisy_qvm)
print result
```

```
{'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.SyncConnection(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.SyncConnection(measurement_noise=meas_channel)
```

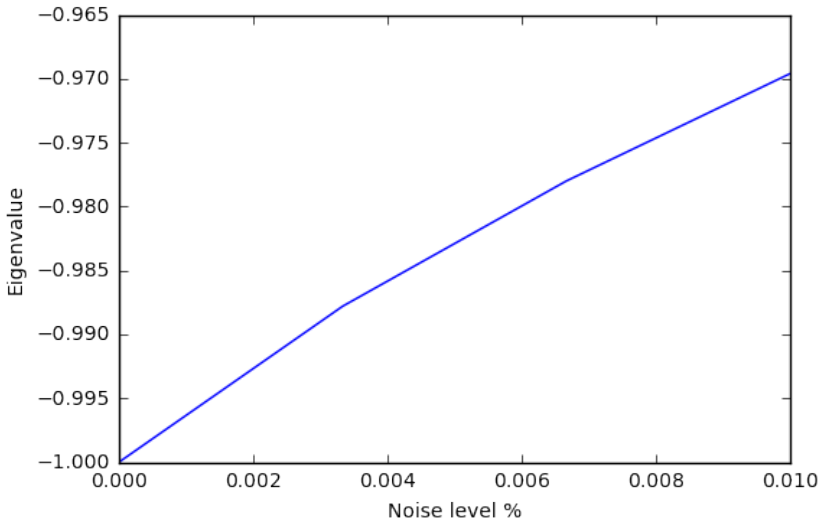
Measurement noise has a different effect:

```
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.SyncConnection(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.

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.

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

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

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.

Quantum Approximate Optimization Algorithm (QAOA)

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.

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.

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 qvm
qvm_connection = qvm.SyncConnection()
```

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)
```

Algorithm and Details

Introduction

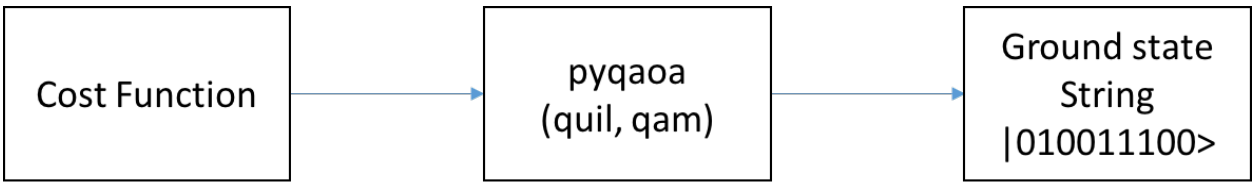
The quantum-approximate-optimization-algorithm (QAOA, pronouced quah-wah), developed by Farhi, Goldstone, and Gutmann, is a polynomial time algorithm for finding "a 'good' solution to an optimization problem" [1, 2].

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.

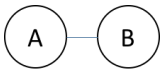
The following section of the pyQAOA documentation describes the algorithm and the NP-hard problem instance used in the original paper.



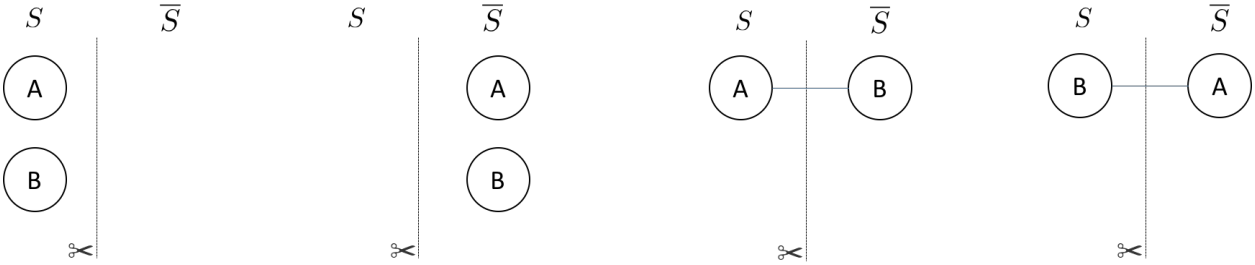
Our First NP-Hard Problem

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

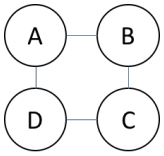


there are 4 ways of partitioning nodes into two sets:



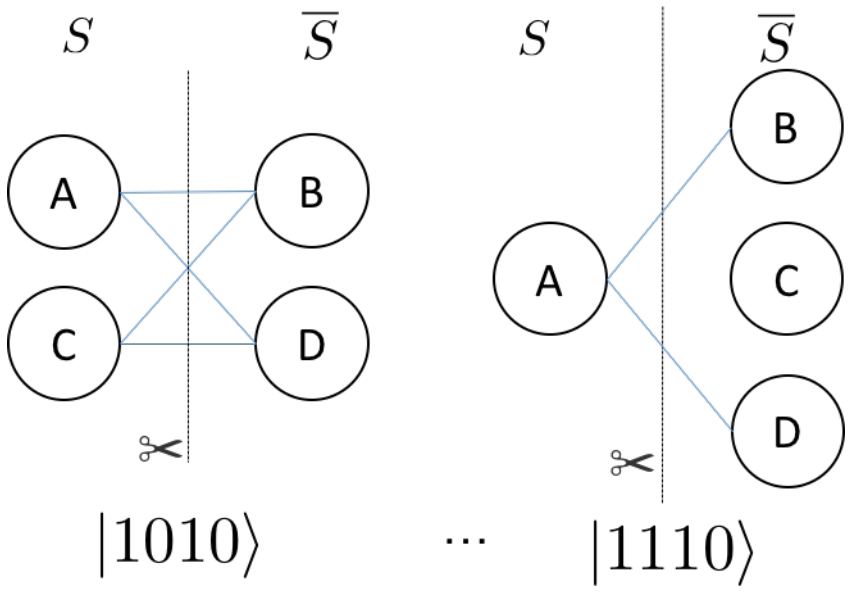
We have drawn the edge only when it connects nodes in different sets. The line with the scissor symbol indicates that we count the edge in our cut. For the barbell graph there are two equal weight partitionings that correspond to a maximum cut (the right two partitonings)–i.e. cutting the barbell in 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...\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.

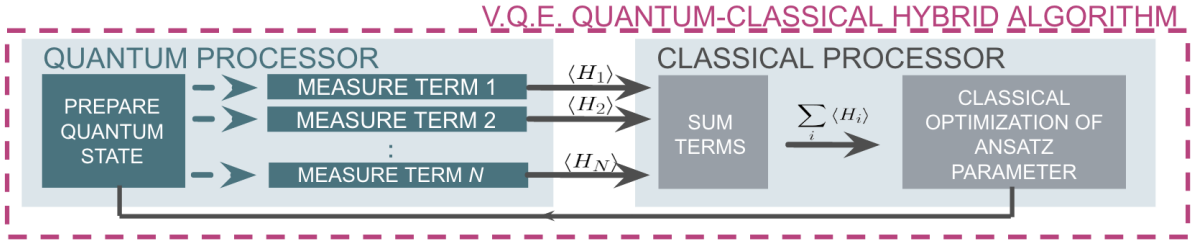
For the \(U(\hat{H}_{\mathrm{ref}}, \beta_{i})\) term (or mixing term) all operators in the sum commute and thus can be split into a product of exponentiated \(\sigma_{x}\) operators. \begin{align} e^{-i\hat{H}_{\mathrm{ref}} \beta_{i}} = \prod_{n = 0}^{1}e^{-i\sigma_{n}^{x}\beta_{i}} \end{align}

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

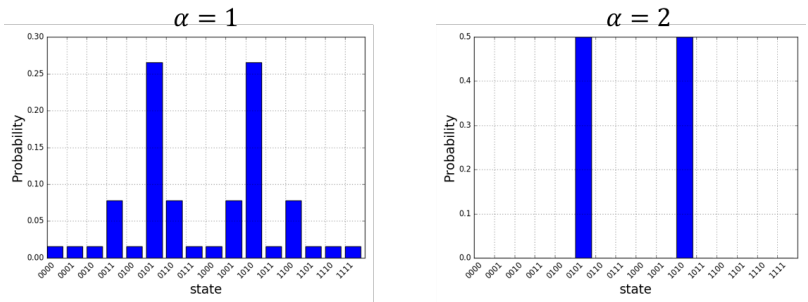
Of course, if RX is in the natural gate set for the quantum-processor this Quil is compiled into a set of RX rotations. The Quil code for the cost function \begin{align} e^{-i \frac{\gamma_{i}}{2}(\mathbf{I} - \sigma_{1}^{z} \otimes \sigma_{0}^{z}) } \end{align} looks like this:

```
X 0
PHASE(gamma{i}/2) 0
X 0
PHASE(gamma{i}/2) 0
CNOT 0 1
RZ(gamma{i}/2) 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,...,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 degnerate cuts of the four ring graph.

Source Code Docs

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

grove.pyqaoa.qaoa

class `grove.pyqaoa.qaoa.`**QAOA**(*qvm*, *n_qubits*, *steps=1*, *init_betas=None*, *init_gammas=None*, *cost_ham=[]*, *ref_hamiltonian=[]*, *driver_ref=None*, *minimizer=None*, *minimizer_args=[]*, *minimizer_kwargs={}*, *rand_seed=None*, *vqe_options={}*, *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.

Parameters

- **qvm** – (Connection) The qvm connection to use for the algorithm.
- **n_qubits** – (int) 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_hamiltonian** – list of clauses in the cost 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 **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.
- **steps** – (int) number of steps approximating the solution.

Quantum Fourier Transform (QFT)

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].

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.

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.

[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/

References

Phase Estimation Algorithm

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].

Source Code Docs

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

grove.phaseestimation.phase_estimation

grove.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.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

Grover's Search Algorithm and Amplitude Amplification

Overview

Quantum search algorithm and more via amplitude amplification.

Source Code Docs

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

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

grove.amplification.amplification

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

For more information, see arXiv:quant-ph/0005055.

grove.amplification.amplification.**amplify**(*A*, *A_inv*, *U_w*, *qubits*, *num_iter*, *init=True*)

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

Parameters

- **A** – a program representing a measurement-less algorithm run on qubits
- **A_inv** – a program representing the inverse algorithm of A. This can be done using the Program object's adjoint() method
- **U_w** – an oracle maps any basis vector to either |0> or |1>
- **qubits** – the qubits to operate on
- **num_iter** – number of iterations of amplifications to run
- **init** – a boolean flag that is set to True if and only if A is to be applied initially on the input qubits. By default, it is set to True.

Returns

grove.amplification.amplification.**diffusion_operator**(*qubits*)

Constructs the (Grover) diffusion operator on qubits, assuming they are ordered from most significant qubit to least significant qubit.

The diffusion operator is the diagonal operator given by (1, -1, -1, ..., -1).

See arXiv:quant-ph/0301079 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.**n_qubit_control**(*controls*, *target*, *u*, *gate_name*)

Returns a controlled u gate with n-1 controls. Useful for constructing oracles.

Uses a number of gates quadratic in the number of qubits, and defines a linear number of new gates. (Roots and adjoints of u.)

See arXiv:quant-ph/9503016 for more information.

Parameters

- **controls** – The indices of the qubits to condition the gate on.
- **target** – The index of the target of the gate.
- **u** – The unitary gate to be controlled, given as a numpy array.
- **gate_name** – The name of the gate u.

Returns The controlled gate.

grove.amplification.grover

Module for Grover's algorithm. Based off standalone grover implementation. Uses the amplitude amplification library.

grove.amplification.grover.**basis_selector_oracle**(*bitstring*, *qubits*)

Defines an oracle that selects the ith element of the computational basis.

Defines a phase filp rather than bit flip oracle to eliminate need for extra qubit. Flips the sign of the state $|x\rangle >$ if and only if x==bitstring and does nothing otherwise.

Parameters

- **bitstring** – The desired bitstring, given as a string of ones and zeros. e.g. "101"
- **qubits** – The qubits the oracle is called on. The qubits are assumed to be ordered from most significant qubit to least significant qubit.

Returns A program representing this oracle.

grove.amplification.grover.**grover**(*oracle*, *qubits*, *num_iter=None*)

Implementation of Grover's Algorithm for a given oracle.

The query qubit will be left in the zero state afterwards.

Parameters

- **oracle** (*Program*) – An oracle defined as a Program. It should send |x> to (-1)^f(x)|x>, where the range of f is {0, 1}.
- **qubits** (*list(int)*) – 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

Bernstein-Vazirani Algorithm

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].

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. Additional information for this algorithm can be found at: http://pages.cs.wisc.edu/~dieter/Courses/2010f-CS880/Scribes/04/lecture04.pdf

grove.bernstein_vazirani.bernstein_vazirani.**bernstein_vazirani**(*oracle*, *qubits*, *ancilla*)

Implementation of the Bernstein-Vazirani Algorithm.

Given a list of input qubits and an ancilla bit, all initially in the $|0\rangle$ state, create a program that can find $\vec{a}$ with one query to the given oracle.

Parameters

- **oracle** (*Program*) – Program representing unitary application of function
- **qubits** (*list(int)*) – List of qubits that enter as state $|x\rangle$.
- **ancilla** (*int*) – Ancillary qubit to serve as input $|y\rangle$, where the answer will be written to.

Returns A program corresponding to the desired instance of the Bernstein-Vazirani Algorithm.

Return type Program

grove.bernstein_vazirani.bernstein_vazirani.**oracle_function**(*vec_a*, *b*, *qubits*, *ancilla*)

Creates a black box oracle for a function to be used in the Bernstein-Vazirani algorithm.

For a function f such that

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

where $(\cdot)$ is the bitwise dot product, this function defines a program that performs the following unitary transformation:

$$|\mathbf{x}\rangle|y\rangle \rightarrow |\mathbf{x}\rangle|f(\mathbf{x}) \text{ xor } y\rangle$$

where xor is taken bitwise.

Allocates one scratch bit.

Parameters

- **vec_a** (*list(int)*) – a vector of length n containing only ones and zeros. The order is taken to be most to least significant bit.
- **b** (*int*) – a 0 or 1
- **qubits** (*list(int)*) – List of qubits that enter as input $|x\rangle$. Must be the same length (n) as $\mathbf{a}$
- **ancilla** (*int*) – Ancillary qubit to serve as input $|y\rangle$, where the answer will be written to.

Returns A program that performs the above unitary transformation.

Return type Program

grove.bernstein_vazirani.bernstein_vazirani.**run_bernstein_vazirani**(*cxn*, *oracle*, *qubits*, *ancilla*)

Runs the Bernstein-Vazirani algorithm.

Given a QVM connection, an oracle, the input bits, and ancilla, find the $\mathbf{a}$ and b corresponding to the function represented by the oracle.

Parameters

- **cxn** (*Connection*) – the QVM connection to use to run the programs
- **oracle** (*Program*) – the oracle to query that represents a function of the form $f(x) = \mathbf{a} \cdot \mathbf{x} + b \pmod 2$
- **qubits** (*list*) – the input qubits
- **ancilla** (*Qubit*) – the ancilla qubit

Returns

a tuple that includes, in order,

- the program's determination of $\mathbf{a}$
- the program's determination of b
- the main program used to determine $\mathbf{a}$

Return type tuple

References

Simon's Algorithm

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].

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.

[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

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.

Source Code Docs

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

grove.simon.simon

Module for the Simon's Algorithm. For more information, see

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

grove.simon.simon.**binary_back_substitute**(*W*, *s*)

Perform back substitution on a binary system of equations.

Finds the $\mathbf{x}$ such that $W\mathbf{x} = \mathbf{s}$, where all arithmetic is taken bitwise and modulo 2.

Parameters

- **W** (*2darray*) – A square $n \times n$ matrix of 0s and 1s, in row-echelon form
- **s** (*1darray*) – An $n \times 1$ vector of 0s and 1s

Returns The $n \times 1$ vector of 0s and 1s that solves the above system of equations.

Return type 1darray

grove.simon.simon.**check_two_to_one**(*cxn*, *oracle*, *ancillas*, *s*)

Check if the oracle is one-to-one or two-to-one. The oracle is known to represent either a one-to-one function, or a two-to-one function with mask s.

Parameters

- **cxn** (*JobConnection*) – the connection used to run programs
- **oracle** (*Program*) – the oracle to query; emulates a classical $f(x)$ function as a black-box.
- **qubits** (*list(int)*) – the input qubits
- **ancillas** (*list(int)*) – the ancillary qubits, where $f(x)$ is written to by the oracle
- **s** (*str*) – the proposed mask of the function, found by Simon's algorithm

Returns true if and only if the oracle represents a function that is two-to-one with mask s

Return type bool

grove.simon.simon.**find_mask**(*cxn*, *oracle*, *qubits*)

Runs Simon's algorithm to find the mask.

Parameters

- **cxn** (*JobConnection*) – the connection used to run programs
- **oracle** (*Program*) – the oracle to query; emulates a classical $f(x)$ function as a black-box.
- **qubits** (*list(int)*) – the input qubits

Returns a tuple t, where t[0] is the bitstring of the mask, t[1] is the number of iterations that the quantum program was run, and t[2] is said quantum program.

Return type tuple

grove.simon.simon.**insert_into_row_echelon_binary_matrix**(*W*, *z*)

Given a matrix W of 0s and 1s in row-echelon form, such that each row is orthogonal to some (unknown) vector $\mathbf{s}$ of 0s and 1s, attempt to insert a new row into W that maintains the above property.

Besides the above property, a vector $\mathbf{z}$ is given that is known to also be orthogonal to $\mathbf{s}$.

If (and only if) no such row can be inserted with certainty, W is return unchanged.

Parameters

- **W** (*2darray*) – a matrix of 0s and 1s in row-echelon form, with rows all orthogonal to some vector of 0s and 1s.
- **z** (*1darray*) – a vector of 0s and 1s known to be orthogonal to $\mathbf{s}$

Returns either the same matrix W, unchanged, or W with one additional row added that maintains the property described above.

Return type 2darray

grove.simon.simon.**is_unitary**(*matrix*)

A helper function that checks if a matrix is unitary.

Parameters **matrix** (*2darray*) – a matrix to test unitarity of

Returns true if and only if matrix is unitary

Return type bool

grove.simon.simon.**make_square_row_echelon**(*W*)

Make W into a square matrix for Simon's algorithm, satisfying a few criteria.

Parameters **W** (*2darray*) – an $(n-1) \times n$ array of 0s and 1s in row-echelon form such that all rows are orthogonal to some length n vector of 0s and 1s $\mathbf{s}$.

Returns a two-element tuple. The first element is an $n \times n$ square array identical to W except with one row added. That row is chosen to keep W in row-echelon form. The second element is the row that the new row is in, where the top row is at index 0.

Return type tuple

grove.simon.simon.**most_significant_bit**(*lst*)

A helper function that finds the position of the most significant bit in a 1darray of 1s and 0s, i.e. the first position where a 1 appears, reading left to right.

Parameters **lst** (*1darray*) – a 1darray of 0s and 1s with at least one 1

Returns the first position in lst that a 1 appears

Return type int

grove.simon.simon.**oracle_function**(*unitary_funct*, *qubits*, *ancillas*, *gate_name='FUNCT'*)

Given a unitary U_f that acts as a function $f : \{0,1\}^n \to \{0,1\}^n$, such that

$$U_f|x\rangle = |f(x)\rangle$$

create an oracle program that performs the following transformation:

$$|x\rangle|y\rangle \to |x\rangle|f(x) \oplus y\rangle$$

where $|x\rangle$ and $|y\rangle$ are n qubit states and $\oplus$ is bitwise xor.

Allocates one scratch bit.

Parameters

- **unitary_funct** (*2darray*) – Matrix representation U_f of the function f, i.e. the unitary transformation that must be applied to a state $|x\rangle$ to get $|f(x)\rangle$
- **qubits** (*list(int)*) – List of qubits that enter as the input $|x\rangle$.
- **ancillas** (*list(int)*) – List of qubits to serve as the ancillary input $|y\rangle$.
- **gate_name** (*str*) – Optional parameter specifying the name of the gate that will represent unitary_funct

Returns A program that performs the above unitary transformation.

Return type Program

grove.simon.simon.**simon**(*oracle*, *qubits*)

Implementation of the quantum portion of Simon's Algorithm.

Given a list of input qubits, all initially in the $|0\rangle$ state, create a program that applies the Hadamard-Walsh transform the qubits before and after going through the oracle.

Parameters

- **oracle** (*Program*) – Program representing unitary application of function
- **qubits** (*list(int)*) – List of qubits that enter as the input $|x\rangle$.

Returns A program corresponding to the desired instance of Simon's Algorithm.

Return type Program

grove.simon.simon.**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 one is always provided. That is, if given the mapping of n qubits, the calculated transformation will be on $n + 1$ qubits, where the zeroth qubit is the scratch bit and the return value of the function is left in the qubits that follow.

Parameters **mappings** (*list(int)*) – List of the mappings of $f(x)$ on all length n in their decimal representations. For example, the following mapping:

- $00 \rightarrow 00$
- $01 \rightarrow 10$
- $10 \rightarrow 10$
- $11 \rightarrow 00$

Would be represented as $[0, 2, 2, 0]$. Requires mappings to be either one-to-one, or two-to-one with unique mask s, as specified in Simon's problem.

Returns Matrix representing specified unitary transformation.

Return type numpy array

References

Deutsch-Jozsa Algorithm

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 classical algorithm, the Deutsch-Jozsa Algorithm can solve this problem with a single iteration, regardless of the input size.

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. Based off description in "Quantum Computation and Quantum Information" by Neilson and Chuang.

grove.deutsch_jozsa.deutsch_jozsa.**deutsch_jozsa**(*oracle*, *qubits*, *ancilla*)

Implementation of the Deutsch-Jozsa Algorithm.

Can determine whether a function f mapping {0,1}^n to {0,1} is constant or balanced, provided that it is one of them.

Parameters

- **oracle** (*Program*) – Program representing unitary application of function.
- **qubits** (*list*) – List of qubits that enter as state |x>.
- **ancilla** (*Qubit*) – Qubit to serve as input |y>.

Returns A program corresponding to the desired instance of the Deutsch-Jozsa Algorithm.

Return type Program

grove.deutsch_jozsa.deutsch_jozsa.**integer_to_bitstring**(*x*, *n*)

Converts a positive integer into a bitstring.

Parameters

- **x** (*int*) – The integer to convert
- **n** (*int*) – The length of the desired bitstring

Returns x as an n-bit string

Return type str

grove.deutsch_jozsa.deutsch_jozsa.**is_unitary**(*mat*)

Checks if a matrix is unitary.

Parameters **mat** (*np.array*) – The matrix to check.

Returns Whether or not mat is unitary.

Return type bool

grove.deutsch_jozsa.deutsch_jozsa.**oracle_function**(*unitary_funct*, *qubits*, *ancilla*)

Defines an oracle that performs the following unitary transformation: |x>|y> -> |x>|f(x) xor y>

Allocates one scratch bit.

Parameters

- **unitary_funct** (*np.array*) – Matrix representation of the function f, i.e. the unitary transformation that must be applied to a state |x> to put f(x) in qubit 0, where f(x) returns either 0 or 1 for any n-bit string x
- **qubits** (*np.array*) – List of qubits that enter as input |x>.
- **ancilla** (*Qubit*) – Qubit to serve as input |y>.

Returns A program that performs the above unitary transformation.

Return type Program

grove.deutsch_jozsa.deutsch_jozsa.**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** (*list*) – List of the mappings of f(x) on all length n bitstrings. For example, the following mapping: { 00 -> 0, 01 -> 1, 10 -> 1, 11 -> 0 } Would be represented as [0, 1, 1, 0].

Returns Matrix representing specified unitary transformation.

Return type numpy array

Arbitrary State Generation

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].

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].

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.

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.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.

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

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.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.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.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.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.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.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.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

Python Module Index

g

Index