CS269Q: Quantum Programming **Spring 2019**

Project #2

Due: 11:59pm on Wed., May 8 2019, by Gradescope

In this project you will implement quantum programs representing two different paradigms of hybrid quantum/classical computation. In the first part you will implement a quantum error correction protocol, which will require you to understand classical control and feedback onto quantum measurements. In the second part you will implement a version of the variational quantum eigensolver. This will give you an opportunity to learn about variational hybrid programming. In the third part you will practice mapping a problem into a quantum form appropriate for variational quantum solution.

Problem 1. Implementing and Simulating the Shor Code

There are two parts to this project. For the first part you will need to submit a Python file called *shorcode.py* to the autograder. The functionality will be as described below. For more details you are referred to the shorcode_template.py template file that accompanies this assignment.

a. Write a function bit_code(q) that, for an arbitrary qubit placeholder q, returns a pyQuil program that creates and applies the 3-qubit bit-flip code to that qubit. You will need to allocate additional qubits for redundancy and to do the parity measurements. You will then need to encode your qubit into the logical qubit space, performing decoding measurements, and apply the right correction. In this project you will not hardcode in qubit register indices. Error correcting codes are useful when they can be used to transform any arbitrary program. You may want to look into QubitPlaceholder objects in pyQuil for this. See the template file for more details.

b. Write a function phase_code(q) that does the same as part (a) but for the 3-bit phase flip code.

c. Write a function shor(q) that does the same but for the full 9-qubit Shor code.

d. Write the functions phase_flip_channel and depolarizing_channel that take a noise probability as an argument and return the Kraus operators corresponding to those noise models.

e. Write a function called simulate that takes as argument a list of Kraus operators, a number of trials, and one of your error correction code functions. The simulate function should then return the probability that the code of choice preserves the correct logical state under the given Kraus operators. Unlike in your last project (where we automatically applied Kraus operators to many qubits and gates) we will treat noise differently in this projects simulation. We will add noise to the Identity gate and specifically apply this noise in a particular place in our circuit. See the accompanying template file for details.
New Hints:

Note that in this assignment you are not only encoding and decoding the qubits. **You must also apply the actual corrections based on the results of the parity measurements.**

This means that your solution would be expected to apply correction gate operations conditional on the classical memory that stores the output of your parity measurements.

In testing there is an input argument called *noise*. This argument is a Python function that operates on the qubit registers in your code to apply gates for a noise model. A good way to check to see if your bit flip code, for example, is working would be to input a function for noise that returns X(q) where q is some qubit that you have used as part of your code. This represents a single bit flip error happening on one of your qubits. Your code should correct for this bit flip so that when all qubits are measured they always remain in the 0 state (which is where they started). Likewise, if a single Z(q) error happens to your phase flip code then you should be able to correct for it. Similarly, since your code only has distance 3 it can only correct for one error. If you have two X errors on the bit flip code then you would expect to get the wrong answer. What do you think you should get when you run the bit flip code and two qubits have X errors?

Problem 2. Implementing a Small Variational Quantum Eigensolver

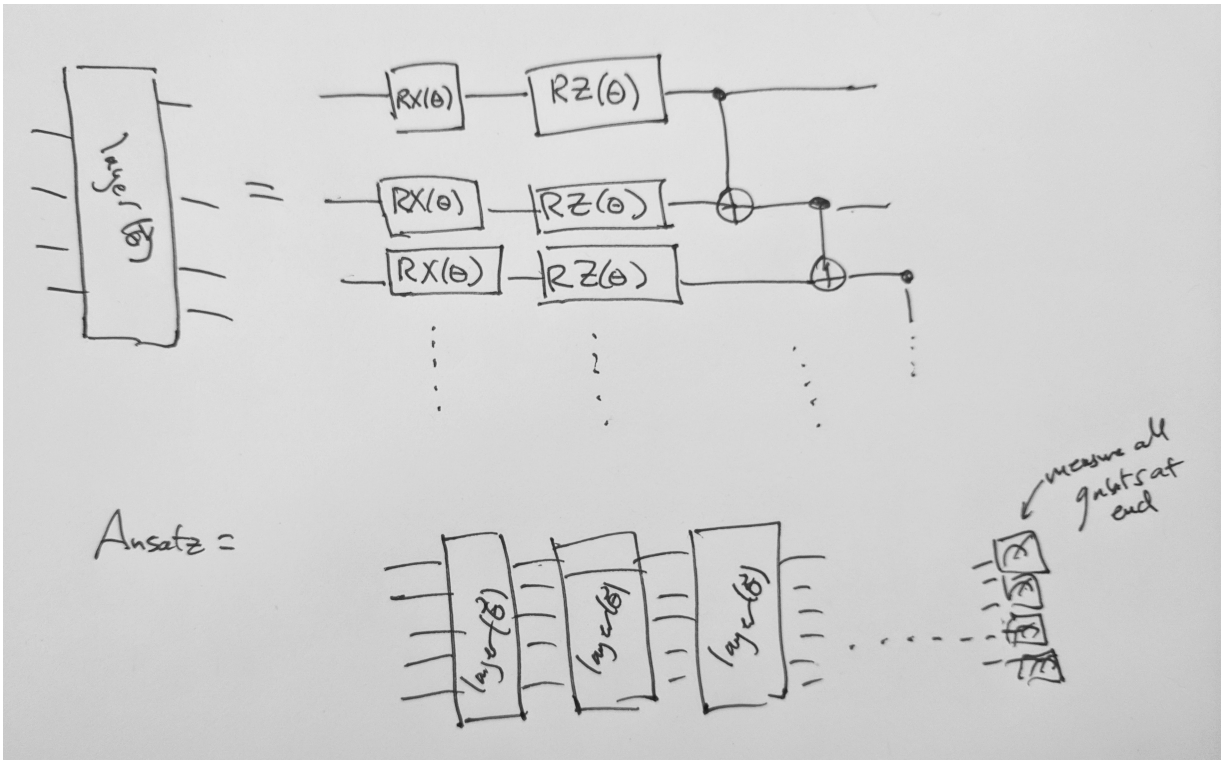
For this second part you will need to submit a Python file called *vqe.py* to the autograder. The functionality will be as described below. For more details you are referred to the vqe_template.py template file that accompanies this assignment.

Write a function called solve_vqe that takes as argument a representation of a matrix H as a PauliSum object and returns the smallest eigenvalue of H by running VQE on an ansatz. You can assume that the PauliSum is defined on at most 5 qubits. In this part of the project you can try this in a simulated mode where you avoid sampling error and calculate the wavefunctions automatically with the QVM. You may find WavefunctionSimulator and it's expectation method useful for this part. Note that you will likely need to experiment with different choices of ansatzes and classical optimizers to be successful here.

New HINT:

Because the problem specifies that you'll be looking at relatively small Hamiltonians of less than 5 qubits, we can first attempt a solution using a generic ansatz based on the hardware ansatz from Kandala et al. 2017. Let the following define a single *layer* of our ansatz. Let N be the number of qubits our ansatz is applied to. Apply RX(θ_i) followed by RZ(θ_{i+1}) to every qubit. Then apply CNOT($q, q+1$) for q in range(N-1). This applies CNOT gates to entangle all of our qubits after doing some rotation that is parameterized by our $\vec{\theta}$.

We then concatenate these parameterized layers some number of times to produce an ansatz of a given depth. More layers means more parameters to train while it also means more flexibility for your ansatz and thus more likelihood of finding the correct answer. A diagram of this ansatz structure is pictured below.



Problem 3. Mapping Traveling Salesman into Quantum Optimization

In the traveling salesman problem we are presented a graph of cities with weighted edges that represent the distance between the cities. We are then asked to find the shortest path in the graph that visits all the cities exactly once.

Let G be a graph of G cities. Let $E(c_1, c_2)$ be an edge in the graph that is labeled by a real number $w_{1,2}$ representing the distance between cities $c_1, c_2 \in G$. A path is a sequence of cities $c_0, ..., c_n$. The TSP cost is then given by the distance of this path. In order to solve this with a variational quantum algorithm such as the quantum approximate optimization algorithm, we will need to represent this problem in a form that we can program.

In this problem we will write down definitions a Hamiltonian whose ground state is a solution to the Traveling Salesman Problem. This would be the first step in preparing to solve TSP on a quantum computer using QAOA.

Let $x_{\alpha,j}$ be a set of n^2 binary variables where α denotes a city and j denotes a time step in some path. Thus $x_{2,0} = 1$ means your path starts in city 2 in the zeroth time step. Likewise $x_{4,n-1} = 1$ means you end in city 4 on the last time step. There are n time steps as we will visit each city exactly once. We form a bitstring that represents a path as follows

$$x_{0,0}x_{1,0}, ..., x_{n-1,0}x_{0,1}, ..., x_{n-1,n-1} \tag{1}$$

Note that not all bitstrings are valid paths.

- $100010001 = (100)(010)(001)$ represents the path $0 \to 1 \to 2$
- 111111111 is not a valid path

- $010001100 = (010)(001)(100)$ represents the path $1 \to 2 \to 0$

We now need to write down the constraints as terms in a Hamiltonian.

a. Each city should only appear once in a given path. Write down a formula for a Hamiltonian term that penalizes bitstrings that violate this condition.

b. Each timestep should only have one city in it. Write down a formula for a Hamiltonian term that penalizes bitstrings that violate this condition.

c. As we go between timesteps, we should only go between cities where an edge exists. Write down a formula for a Hamiltonian term that penalizes paths that violate this condition.

d. We want to find the shortest valid path that visits all cities exactly once. Write down a term that penalizes long paths.

The sum of these terms then gives a Hamiltonian for TSP, i.e. its ground state is a solution to the TSP problem.

Your grade:

The three problems are weighted as 64 pts, 21 pts, and 15 pts respectively. Your score for the first two will come from the autograder while the third problem will be graded manually from your written submission.