COMS 4281 - Intro to Quantum Computing

Problem Set 5: Noise and Codes

Due: December 11, 11:59pm.

Collaboration is allowed and encouraged (tea… …ead the syllabus carefully for the guidlines regarding collaboration. In particular, everyone must write their own solutions in their own words.

Write your collaborators here:

Problem 1: Mixed states

Problem 1.1

Compute the single- and two-qubit reduced density matrices of the 3-qubit GHZ state

$$\frac{|000\rangle + |111\rangle}{\sqrt{2}}.$$

Solution

Problem 1.2

Let $\{(p_1, |\psi_1\rangle), \ldots, (\psi_k, |\psi_k\rangle)\}$ denote a probabilistic mixture of states -- not necessarily orthogonal -- where p_i's are probabilities that sum to 1. Consider the density matrix $\sigma = p_1 |\psi_1\rangle\langle\psi_1| + \cdots + p_k |\psi_k\rangle\langle\psi_k|$ on a system A. Show that the following vector $|\theta\rangle_{AB}$ is a valid pure state (i.e. unit length vector), and it purifies σ:

$$|\theta\rangle = \sum_{i=1}^{k} \sqrt{p_i}\, |\psi_i\rangle_A \otimes |i\rangle_B$$

where B is a register of dimension at least k and $\{|i\rangle\}$ is an orthonormal basis for B.

Solution

Problem 1.3

Come up with a two-qubit pure state $|\psi\rangle_{AB}$ whose reduced density matrix on system A is

$$\mathrm{Tr}_B(|\psi\rangle\langle\psi|) = \frac{1}{2}\begin{pmatrix} 1+p & -\sqrt{(1-p)p} \\ -\sqrt{(1-p)p} & 1-p \end{pmatrix}$$

where $0 \leq p \leq 1$ is some parameter.

Hint: Try to reduce to part (2) above.

Solution

Problem 1.4

Let $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ denote an unknown qubit, and suppose it was just teleported from Alice to Bob. But imagine that, before Alice is able to tell Bob the outcomes of her measurements, her internet cuts out. So Bob's qubit is one of the states

$$|\psi\rangle, X|\psi\rangle, Z|\psi\rangle, XZ|\psi\rangle$$

with equal probability. What is the density matrix that describes the mixed state of Bob's qubit?

Solution

Problem 2: Limitations of Shor's code

Show that there are distinct two-qubit errors $E_1 \neq E_2$ (unitaries acting on two qubits) such that

$$E_1\left|\bar{0}\right\rangle = E_2\left|\bar{1}\right\rangle$$

where $\left|\bar{0}\right\rangle, \left|\bar{1}\right\rangle$ represent the logical $|0\rangle, |1\rangle$ state using the 9-qubit Shor code. This shows that the Shor code is not capable of correcting more than 1 error.

Solution

Problem 3: An error *detection* code

Shor's 9-qubit code can **correct** any single-qubit error on one of its 9 qubits. Below we give a 4-qubit code which can \emph{detect} any single-qubit error on its qubits. By this we mean that there is a detection circuit that determines if a single-qubit error has occurred, but it can't necessarily identify which one has occurred.

The encoding map is as follows:

$$\left|\bar{0}\right\rangle = \frac{1}{2}\Big(\,|00\rangle + |11\rangle\,\Big) \otimes \Big(\,|00\rangle + |11\rangle\,\Big)$$
$$\left|\bar{1}\right\rangle = \frac{1}{2}\Big(\,|00\rangle - |11\rangle\,\Big) \otimes \Big(\,|00\rangle - |11\rangle\,\Big)$$

Problem 3.1

Give an example of two distinct single-qubit u[illegible]may act on different qubits) such that

$$E_1 \left|\bar{0}\right\rangle = E_2 \left|\bar{1}\right\rangle \ .$$

This shows that this code cannot uniquely ide[illegible]because given $E_1 \left|\bar{0}\right\rangle$, one can't be sure whether the original state was $\left|\bar{0}\right\rangle$ or $\left|\bar{1}\right\rangle$. However, in the next few parts you will show that the cod[illegible] error has occurred.

Solution

Problem 3.2

Give a procedure (either as a quantum circuit or sufficiently detailed pseudocode) that detects a \emph{bitflip} error on a single qubit of an encoded state $\alpha\left|\bar{0}\right\rangle + \beta\left|\bar{1}\right\rangle$. In other words the procedure, if it's given a valid encoded state $\left|\bar{\psi}\right\rangle$, returns $\left|\bar{\psi}\right\rangle$ and outputs "No error", whereas if it's given $X_i\left|\bar{\psi}\right\rangle$ where X_i is the bitflip operation on some qubit i, then the circuit outputs "X error somewhere".

Solution

Problem 3.3

Give a procedure (either as a quantum circuit or sufficiently detailed pseudocode) that detects a **phaseflip** error on a single qubit of an encoded state $\alpha\left|\bar{0}\right\rangle + \beta\left|\bar{1}\right\rangle$.

Solution

Problem 3.4

Explain how to detect **any** unitary 1-qubit error on one of the 4 qubits.

Solution

Problem 4: Circuits on noisy quantum computers

The IBM Quantum Lab gives public access to a number of their quantum computers. In this problem you'll get to play with them -- and see the effects of noise on your quantum circuits.

Problem 4.1 - Benchmarking individual qubits

In this subproblem, we will benchmark indivi[illegible] of) the 5-qubit `ibmq_essex` device. A typical way to benchmark a qubit is to run a sequence of randomly chosen single-qubit gates g_1, g_2 [illegible] the reverse sequence $g_k^{-1}, g_{k-1}^{-1}, \ldots, g_1^{-1}$ so that overall the effect should be the identity. Of course, each gate will incur some noise, so th[illegible]ft over time. One can measure the noise by measuring the qubit at the end of the sequence to see if it stayed in the $|0\rangle$ state.

You will write code to perform the following: f[illegible]0, for each qubit $q = 0, 1, 2, \ldots, 4$, pick a sequence of gates $g_1, \ldots, g_k$ where each g_i is chosen randomly from the gate set $\{X, Y, Z$ [illegible] the sequence (g_i) forward and then in reverse, and measure the qubit. Do this 1000 times and calculate the percentage $p_{q,k}$ of times that the qubit q ends back in the $|0\rangle$ state.

Below, we've provided two functions. The first function, `benchmark_qubit`, requires you to fill in some code, and it returns a `IBMQJob` object. The second function, `retrieve_job_results`, takes an `IBMQJob` object and gets the measurement counts. (The reason we're dividing this up into two steps is because later we'll run these circuits on a real device).

In [2]:

```
from qiskit import IBMQ, transpile
from qiskit import QuantumCircuit
from qiskit.providers.aer import AerSimulator
from qiskit.providers.fake_provider import FakeEssex
from qiskit.providers.jobstatus import JobStatus

import numpy as np
from matplotlib import pyplot as plt

##########
# This function takes as input
#    - q : qubit number 0 thru 4
#    - k : length of random gate sequence
#    - shots: number of times to run and measure the sequence
#    - device: the device (either simulated or real) to run the circuit on
#
#  Returns:
#    - the IBMQJob object corresponding to the circuit (see https://qiskit.org/documentation/stubs/qiskit.providers.ibmq.job.IBMQJob.html#qis
##########
def benchmark_qubit(q, k, shots,device):

    circ = QuantumCircuit(5, 1)

    ### WRITE CODE TO GENERATE THE RANDOM SEQUENCE OF LENGTH k, and ITS REVERSAL, ON QUBIT q ####################


    ### END CODE BLOCK ###################
```

```
    # measure qubit q, and store it in classical register [0]
    circ.measure([q], [0])

    #this will break down the circuit into gates that can be implemented on the chip
    compiled_circuit = transpile(circ,device,optimization_level=0)

    job = device.run(compiled_circuit,shots=shots)
    return job

##########
# This function takes as input
#    - job: the IBMQJob object corres[illegible]eing executed
#    - blocking: if True, then this w[illegible]cuit results are done executing on the device
#                (either fake or real[illegible] will first check the status of the job.
#
#  Returns:
#    - if the job is done, then it re[illegible]dictionary (e.g., { '0000': 356, '0001': 288, ...})
#      otherwise if the job is still [illegible]status, then it returns None
##########
def retrieve_job_results(job,blocking=True):

    #if it's blocking, then just go ahead and call result()
    if blocking:
        counts = job.result().get_counts(0)
        return counts
    else:
        #first, check the status
        job_status = job.status()

        if job_status is JobStatus.DONE:
            counts = job.result().get_counts(0)
            return counts
        else:
            print("The job ",job.job_id()," has status: ",job_status)
            return None
```

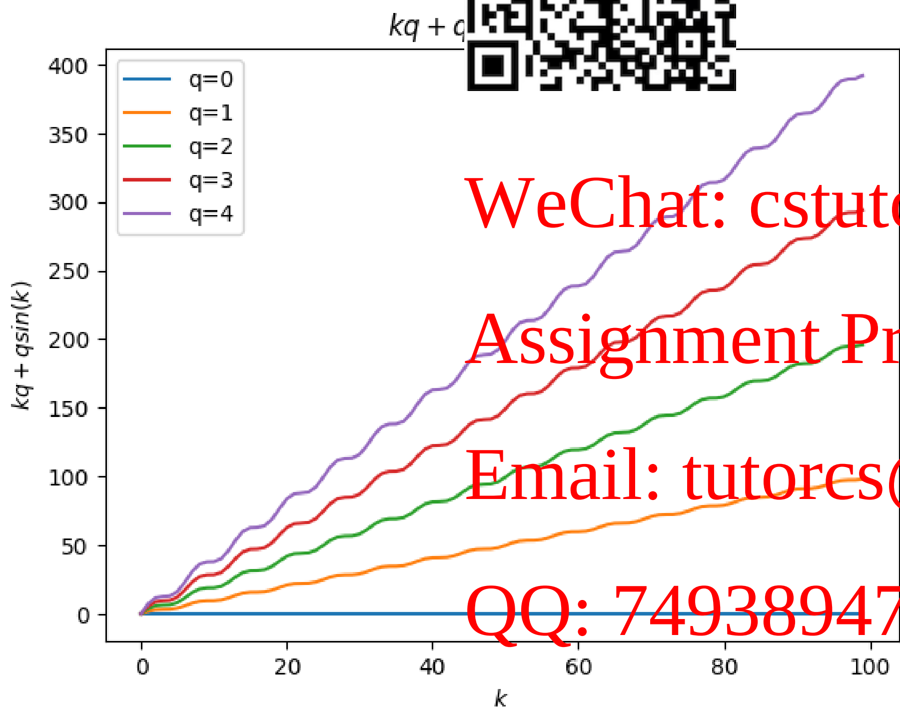
Below, write code using `benchmark_qubit`, `retrieve_job_results` to get the measurement counts and calculate $p_{q,k}$ for $q = 0, 1, 2, 3, 4$ and $k = 10, 20, 30, \ldots, 100$. Then, plot $p_{q,k}$ against k (you can consult https://www.geeksforgeeks.org/using-matplotlib-with-jupyter-notebook/ for an example on how to plot graphs). You should have 5 plots in one graph.

In [6]:

```
fake_device = AerSimulator.from_backend(FakeEssex())

#### WRITE YOUR CODE HERE #################

#example code,
job = benchmark_qubit(0,100,1000,fake_device)
counts = retrieve_job_results(job)
print(counts)
#
```

```
#an example plot, change to display your data instead
for q in range(5):
    ks = np.arange(100)
    p_qks = [k*q + q*np.sin(k) for k in ks]
    plt.plot(ks, p_qks, label=f'q={q}')
plt.xlabel('$k$')
plt.ylabel('$kq + q sin(k)$')
plt.title('$kq + q sin(k)$ vs. k')
plt.legend()

plt.show()

#### END CODE BLOCK ##################################################################
```

```
{'1': 20, '0': 980}
```



Problem 4.2 - Estimating the single qubit gate noise

For each qubit q, find a best function $f(k)$ (linear, quadratic, exponential,...) that fits the plots. For example, if the plot of p_{qk} against k looks linear, then you should come up with parameters a, b such that p_{qk} is close to $ak + b$. If it looks like exponential decay, then you should find an approximate $f(k) = ae^{bk} + c$ for some parameters a, b, c.

This function can be used to give a simple model for how noise accumulates on each qubit from single-qubit gates.

Solution

Problem 4.3 - Benchmarking entanglement generation

Now we benchmark the quantum computer on more complex circuits that involve entangling gates, which will generally be more noisy than single-qubit gates.

For each $r = 2, 3, 4, 5$, let $|\psi_r\rangle$ denote the r-qubit GHZ state

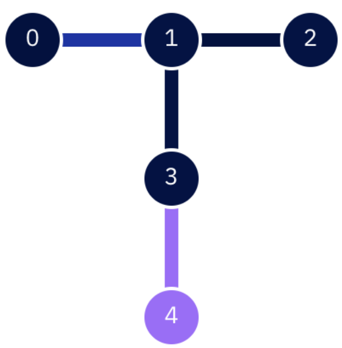
$$\frac{|0\rangle^{\otimes r} + |1\rangle^{\otimes r}}{\sqrt{2}}.$$

When $r = 2$, this is simply the EPR pair we kn… te a circuit that starts with r zeroes and outputs $|\psi_r\rangle$.

Write code to do the same benchmarking as … we perform C_r, then C_r^{-1}, then C_r, and so on k times. Ideally, all of these circuits would cancel out so measuring all r qubits will yield zero. H… noisy so error will accumulate as k grows larger.

Let p_{rk} denote the percentage of times (out of 1000 shots) that doing C_r and C_r^{-1} for k times yields all zeroes in the r qubits. Plot p_{rk} versus k for $r = 2, 3, 4, 5$ and for $k = 10, 20, 30, \ldots, 100$. There should be 4 plots on one graph.

Important: the `ibmq_essex` device has a very particular connectivity of qubits:



You can only apply 2-qubit gates between the connected nodes. For exampe, you can apply a CNOT between qubits 3 and 1, but not 3 and 2. Thus, when coding your circuit C_r, you may want to judiciously choose which r qubits you use to maximize the performance. The darker shading of a qubit means lower noise rate.

In [ ]:
```
##########
# This function takes as input
#    - r : size of GHZ state
#    - k : length of random gate sequence
#    - shots: number of times to run and measure the sequence
#    - device: the device to run on, either simulated or real
#
```

```
#  Returns:
#    - the IBMQJob object corresponding to the circuit (see https://qiskit.org/documentation/stubs/qiskit.providers.ibmq.job.IBMQJob.html#qis
##########
def benchmark_circuit(r, k, shots, device):

    circ = QuantumCircuit(5, r)

    ### WRITE CODE TO GENERATE C_r and its reversal k times ####################

             ## You need to choose your subset of qubits carefully!

    ### END CODE BLOCK ##############


    ### WRITE CODE TO MEASURE AND COM ##########

    # measure r of the qubits , and s ical registers
    #for example, if r = 3, circ.meas ould mean measuring qubits 1,3,4

    # <--- WRITE MEASUREMENT CODE HER

    #this will break down the circuit into gates that can be implemented on the chip
    compiled_circuit = transpile(circ,device,optimization_level=0)
    job = device.run(compiled_circuit,shots=shots)

    return job

#### WRITE CODE TO CALL benchmark_circuit, retrieve_job_results AND PLOT p_rk versus k #################


#### END CODE BLOCK ###############################################################
```

Problem 4.4 - Estimating noise of the GHZ circuit

For each $r = 2, 3, 4, 5$ find a best function $f(k)$ (linear, quadratic, exponential,...) that fits the plots.

These functions can be used to give a simple model for how noise accumulates at a global level for the GHZ circuit. How much worse is this than the single-qubit gates situation?

Solution

Problem 4.5 - Running this on an *actual* quantum computer

So far you've been running all this on a simulated version of the 5-qubit `ibmq_essex` device (which has been retired). Now let's actually run it on a *real* quantum computer -- how exciting!

If you look at the available systems on https://quantum-computing.ibm.com/services/resources, you will see that there are a number of devices (`ibmq_belem` , `ibm_perth` , `ibm_lagos` , `ibm_nairobi` , etc) with varying numbers of qubits, and with different levels of busy-ness (some have more jobs in the queue than others).

If you haven't already, please add your IBM Quantum login email to the spreadsheet

https://docs.google.com/spreadsheets/d/1tB27ZyoDozMzBN9Ya1-bEx6f_pa-VfibDIXVrkA9XCQ/edit?usp=sharing

so that you can get the special education access to their machines (your jobs are processed faster).

For this problem, you will need to run this on the IBM Quantum Lab. The next two commands will show whether you have the special education access.

In [ ]:
```
IBMQ.load_account()
IBMQ.providers()
```

If you don't see something like <AccountPr ... 'ibm-q-education', group='columbia-uni-2', project='computer-science')> in the printed list, then e-mail Prof. Yuen and he ... ator.

The next code will get the machines that are ...

In [ ]:
```
provider = IBMQ.get_provider(hub='ibm-q-education')
provider.backends()
```

Next, pick a machine to use. In our simulated code above, we chose `ibmq_essex`, but you have to choose some other device (because essex has been retired). You should check https://quantum-computing.ibm.com/services/resources to see which ones have the shortest line.

In [ ]:
```
real_device = provider.get_backend('ibmq_PICK_DEVICE_HERE')
real_device.status()
```

Now that you've loaded a device, now you can run the same code (`benchmark_qubit` and `benchmark_circuit`), except by passing `real_device` to the functions this time your circuits will be run on some qubits somewhere in upstate New York! Your goal in this part of the problem is to do the same benchmarking, and compare the results with simulation. Since time on the IBM computers are limited, we only ask you to plot p_{qk} versus k for just one of the qubits, and to only plot p_{rk} versus k for $r = 3$.

Unlike with the simulation code, your circuits will be queued after you call `benchmark_qubit` or `benchmark_circuit`. Each call will return a job that takes time to run (usually a few minutes per job). A hundred circuits will likely take a whole day to finish. (So be sure to test your code first in simulation before submitting jobs to the IBM quantum machines!)

To avoid situations where you lose the result of your jobs, in this part you should structure your code as follows:

1. Call the `benchmark_qubit` and `benchmark_circuit` code all at once, and save the job IDs to a file.
2. Periodically check the status of your jobs (either programmatically or using the "Jobs" control panel in IBM Quantum Lab).
3. Once you see they're done, retrieve the results as before.

Below we provide some helper functions to save jobs to a file.

In [ ]:
```
##########
# This function takes as input
#    - list_of_jobs: self-explanatory
```

```
#    - filename: file to store the job_ids
##########
def save_jobs_to_file(list_of_jobs,filename):

    #open the file
    with open(filename,'w') as f:
        for job in list_of_jobs:
            f.write(job.job_id())
            f.write('\n')

##########
# This function takes as input
#    - name_of_device: this is the na ubmitted the jobs to, such as `ibmq_belem` or `ibm_nairobi`, etc.
#    - filename: file that has all th
#
# Returns: a list_of_jobs
##########
def load_jobs_from_file(name_of_devic
    IBMQ.load_account()
    provider = IBMQ.get_provider(hub=
    device = provider.get_backend(name_of_device)

    list_of_jobs = []
    with open(filename,'r') as f:
        job_ids = f.readlines()
        for job_id in job_ids:
            job = device.retrieve_job(job_id.strip())
            list_of_jobs.append(job)

    return list_of_jobs


##########
# This function takes as input
#    - list_of_jobs: self-explanatory
##########
def print_job_statuses(list_of_jobs):
    for job in list_of_jobs:
        print("id: ",job.job_id()," status: ", job.status())
```


Next, write code for Step 1. You should use `save_jobs_to_file` to store your work.

Warning: If you execute it multiple times, it will resubmit the same jobs! We suggest testing it out with one or two circuit jobs first before scaling up.

In [8]:
```
### INSERT YOUR STEP 1 CODE HERE ######################


################################################
```

Next, write some code to load the jobs ids and check on their status. You can run this block even if you've re-opened the Jupyter notebook.

In [ ]:
```
### INSERT YOUR STEP 2 CODE HERE ######################
```

```
#list_of_jobs = load_jobs_from_file('name_of_device','filename.txt')
#print_job_statuses(list_of_jobs)

###############################################
```

Finally, once you see all the job statuses are done, retrieve the job results. Remember that `retrieve_job_results` takes in an argument called `blocking`, which you should set to `False` here.

In [7]:
```
### INSERT YOUR STEP 3 CODE HERE ####


###############################################
```

How do the results from the actual hardware Quantify the differences, if any.

Solution

In [ ]:
```
## Solution
```

Problem 5: Quantum Tug-of-War!

In this part of the problem set you will write a strategy for a quantum video game. See the accompanying Jupyter notebook `QTugofwar.ipynb` for details.

In [ ]:
```

```