

Problem Set 5: Noise and Codes

Due: December 13, 11:59pm.

Collaboration is allowed and encouraged (teams of at most 3). Please read the syllabus carefully for the guidelines regarding collaboration. In particular, everyone must write their own solutions in their own words.

Write your collaborators here:

Problem 1: 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

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

where $|\bar{0}\rangle, |\bar{1}\rangle$ 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 2: 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 *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:

$$\begin{aligned} |\bar{0}\rangle &= \frac{1}{2} (|00\rangle + |11\rangle) \otimes (|00\rangle + |11\rangle) \\ |\bar{1}\rangle &= \frac{1}{2} (|00\rangle - |11\rangle) \otimes (|00\rangle - |11\rangle). \end{aligned}$$

Problem 2.1

Give an example of two distinct single-qubit unitaries $E_1 \neq E_2$ (which may act on different qubits) such that

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

This shows that this code cannot uniquely identify single-qubit errors, because given $E_1 |\bar{0}\rangle$, one can't be sure whether the original state was $|\bar{0}\rangle$ or $|\bar{1}\rangle$. However, in the next few parts you will show that the code can still detect that **some** error has occurred.

Solution

Problem 2.2

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

Solution

Problem 2.3

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 $\alpha |\bar{0}\rangle + \beta |\bar{1}\rangle$.

Solution

Problem 2.4

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

Solution

Problem 3: Computing on encoded data

In this problem we explore the notion of *logical gates*: these are operations performed on quantum data that is already protected by a quantum error-correcting code (such as Shor's code). Ideally, one would like logical gates to be easier to perform than unencoding, applying the desired gate, and re-encoding. Most ideally, the logical gate should be *transversal*: if $|\bar{\psi}\rangle$ is an n -qubit encoding of the logical single qubit $|\psi\rangle$ and G is a logical single-qubit operation, then applying $K^{\otimes n}$ to $|\bar{\psi}\rangle$ should result in $\overline{G|\psi\rangle}$, where K is some single-qubit gate. Transversal gates are nice because they don't introduce any entanglement, and thus do not spread errors between qubits.

Problem 3.1

Consider the 9-qubit Shor code. How can you implement a logical X (i.e. bitflip operation) on the 9-qubit state $|\bar{\psi}\rangle$ that is the Shor encoding of $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$? What about logical Z ?

Solution

Problem 3.2

Is there a transversal implementation of the logical H (i.e. Hadamard) operation for the 9-qubit Shor code? If there is, demonstrate it and show that it's correct. If not, prove that there isn't.

Solution

Problem 3.3

Suppose $a, b \in \{0, 1\}$ are bits, and consider their Shor encodings $|\bar{a}\rangle, |\bar{b}\rangle$. How can you implement a logical CNOT between these two encoded states, i.e.:

$$\overline{CNOT}\left(|\bar{a}\rangle \otimes |\bar{b}\rangle\right) = \overline{CNOT|a, b\rangle}.$$

Note that the encoded CNOT is an 18-qubit operation.

Hint: try a transversal implementation of CNOT.

Hint: Try solving this problem for the bitflip and phaseflip codes first, before solving it for the Shor code.

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 individual qubits on (a simulation 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, \dots, g_k , and then running the reverse sequence $g_k^{-1}, g_{k-1}^{-1}, \dots, g_1^{-1}$ so that overall the effect should be the identity. Of course, each gate will incur some noise, so the state of the qubit will drift 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: for $k = 10, 20, 30, \dots, 100$, for each qubit $q = 0, 1, 2, \dots, 4$, pick a sequence of gates g_1, \dots, g_k where each g_i is chosen randomly from the gate set $\{X, Y, Z, H\}$. Then, on qubit q run 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 three functions. The first function, `benchmark_qubit`, requires you to fill in some code, and it returns a `QuantumCircuit` object. The second function, `retrieve_job_results`, takes an `IBMQJob` object and gets the measurement counts. The third function, `benchmark_qubit_job`, is a wrapper function around `benchmark_qubit` that runs it on a particular device and returns an `IBMQJob` object. (The reason we're dividing this up into three steps is because later we'll run these circuits on a real device).

```
In [ ]: 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
        from qiskit_ibm_provider import IBMProvider
        from qiskit_ibm_runtime import QiskitRuntimeService
```

```

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
#
# Returns:
#   - the QuantumCircuit object corresponding to the circuit
#####
def benchmark_qubit(q, k):

    circ = QuantumCircuit(5, 1)

    ### WRITE CODE TO GENERATE THE RANDOM SEQUENCE OF LENGTH k, and ITS REVERSE ###

    ### END CODE BLOCK #####

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

#####
# This function takes as input
#   - job: the IBMQJob object corresponding to a circuit being executed
#   - blocking: if True, then this will wait until the circuit results are
#               (either fake or real). If False, then this will first check
#
# Returns:
#   - if the job is done, then it returns the counts as a dictionary (e.g.,
#     otherwise if the job is still running or some other status, then it r
#####
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

#####
# 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 to run on, either simulated or real
#
# Returns:
# - the IBMQJob object corresponding to the circuit
#####
def benchmark_qubit_job(q, k, shots, device):
    circ = benchmark_qubit(q,k)
    compiled_circuit = transpile(circ,device,optimization_level=0)
    job = device.run(compiled_circuit,shots=shots)
    return job

```

Below, write code using `benchmark_qubit_job`, `retrieve_job_results` to get the measurement counts and calculate $p_{q,k}$ for $q = 0, 1, 2, 3, 4$ and $k = 10, 20, 30, \dots, 100$. 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 []:

```

fake_device = AerSimulator.from_backend(FakeEssex())

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

#example code
q = 0
k = 100
shots = 1000
job = benchmark_qubit_job(q,k,shots,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 #####

```

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 know and love. Let C_r denote a circuit that starts with r zeroes and outputs $|\psi_r\rangle$.

Write code to do the same benchmarking as in Problem 4.1, except now 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. However, the gates will be 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 $k = 10, 20, 30, \dots, 100$. 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 example, 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
#
# Returns:
```

```

# - the QuantumCircuit object corresponding to the circuit
#####
def benchmark_circuit(r, k):
    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 COMPUTE P_RK #####

    # measure r of the qubits , and store it in the r classical registers
    #for example, if r = 3, circ.measure([1,3,4],[0,1,2]) would mean measuring

    # <--- WRITE MEASUREMENT CODE HERE

    ### END CODE BLOCK #####

    return circ

#####
# 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
#####
def benchmark_circuit_job(r, k, shots, device):
    circ = benchmark_circuit(r,k)
    compiled_circuit = transpile(circ,device,optimization_level=0)
    job = device.run(compiled_circuit,shots=shots)
    return job

#### WRITE CODE TO CALL benchmark_circuit_job, retrieve_job_results AND PLOT

##### 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_brisbane` , `ibmq_osaka` , `ibmq_kyoto`) with varying parameters, and with different levels of busy-ness (some have more jobs in the queue than others).

For this problem, you will need to run this on the IBM Quantum Lab.

The next code will get the machines that are available to you:

```
In [ ]: provider = IBMProvider(instance="ibm-q/open/main")
        provider.backends()
```

Next, pick a machine to use. In our simulated code above, we choose `ibmq_essex` , but you have to choose some other device (one of `ibmq_kyoto` , `ibmq_osaka` , or `ibmq_brisbane`). You should check <https://quantum-computing.ibm.com/services/resources> to see which ones have the shortest line.

```
In [ ]: real_device = provider.get_backend('your_device_name_here')
        print('Max Circuits:', real_device.max_circuits)
        real_device.status()
```

Now that you've loaded a device, now you can run the same code (`benchmark_qubit` and `benchmark_circuit`) to create the quantum circuits that you want to run, except this time you will transpile and run them on `real_device` -- your circuits will be run on some *real* 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, you will create all of the circuits that you want to run first, then transpile them all to `real_device` , and then create a single job that runs all of your circuits together. (Be sure to test your code first in simulation before submitting jobs to the IBM quantum machines!).

Once you run your code, you can see the jobs being queued here:

<https://quantum.ibm.com/jobs>. If you need to, you can use their jobs manager interface to cancel unwanted jobs.

First, write your code to benchmark qubits/circuits on the IBM device of your choosing. Be sure to frequently check <https://quantum.ibm.com/jobs> to see whether your job are still in progress or have finished.

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.

Note: The 127-qubit machines can only have 3 jobs queued at any time, so you have to be patient about sending your jobs to collect all your data. Furthermore, you are only allocated 10 minutes per month to run circuits on the 127-qubit machines. This entire assignment should take ~10 seconds to run all the circuits for, but if you end up doing lots of tests be careful not to exceed that limit.

```
In [ ]: circuits = []
        ### WRITE CODE TO PREPARE ALL OF YOUR CIRCUITS HERE #####

        ### END CODE BLOCK #####

        ### These lines will run all of your circuits together in a single job
shots = 1000
compiled_circuits = [transpile(circ, real_device, optimization_level=0) for
batch_job = real_device.run(compiled_circuits, shots=shots)

print('Job ID:', batch_job.job_id())
job_id = batch_job.job_id()
```

If your Jupyter notebook is ever closed and you need to retrieve the result of your compute jobs after a while, you can do so by looking up the Job ID on <https://quantum.ibm.com/jobs> and using the following code:

```
In [ ]: # This is to check your Job Status
service = QiskitRuntimeService()
job = service.job(job_id)
print(job.status())
#####
```

```
In [ ]: #if it's done, get the results from the following function
job.result().get_counts()
```

Finally, plot the results just like you did with your simulations.

```
In [ ]: ### PLOT THE REAL MACHINE DATA HERE #####
```

```
#####
```

How do the results from the actual hardware compare with simulation? Quantify the differences, if any.

Solution

In []: `## Solution`