Run On A Quantum Computer

James King, Matan Shtepel, Jacob Zhang

March 2022

1 How to reproduce our implementation

Here we hope to describe our implementation in enough detail so it can be reproduced by another group. Let us breakdown the simplest

```python
"Deutsch-Jozsa"
import enum #to make enum for constant or balanced -- makes code
    convininet
import sys  # to collect cmdline args

import numpy as np # for more efficient array processing when
    making U_f
import qiskit # to compile and load circuit
from qiskit import IBMQ # to use connect to IBM Quantum accounts
from qiskit.providers.aer import QasmSimulator # the qiskit qasm
    simulator
from qiskit.providers.aer.noise import NoiseModel # use the noise
    model for the IBM sims
from qiskit.quantum_info.operators import Operator # to make a gate
     for a qiskit circuit (for U_f)
```

Listing 1: Python example

These are some standard imports. Each one is commented with why it is needed

```python
USE_IBMQ = True
SIMULATOR_NOISE = True
VERBOSE = True
NUM_SHOTS = 1024

if USE_IBMQ or SIMULATOR_NOISE:
    IBMQ.save_account("YOUR_KEY_HERE")


class Outcome(enum.Enum):
    CONSTANT = enum.auto()
    BALANCED = enum.auto()
```

Listing 2: Python example

Here are some set-up constants that make the development process more friendly. This section of the code essentially serves as a configuration for the rest of our code.

```python
def get_U_f_matrix(n_inp, n_anc, f):
    side = 2 ** (n_inp + n_anc)
    ret = np.zeros((side, side))

    for i in range(side):
        ret[bool(f(i >> n_anc)) ^ i, i] = 1

    return ret
```

Listing 3: Python example

This code makes the U_f for the circuit. First, we determine the size of the gate, then, make the U_f in the standard way, that is, for each possible state of the system (that is, the i in range 2^{side}), then, in the i'th row, we place a 1 at the position where i gets mapped to by U_f. That is, i should be mapped to

$$\text{the first n\_inp bits of , the last n\_anc bits of i } \oplus f(\text{the first n\_inp bits of } i)$$

where by , we mean "append". Perhaps less intuitively, the formula we used in our code cycles through the columns linearly, placing a 1 at the row that gets mapped to column i. That is the row such that

$$\text{the first n\_inp bits of } row, \text{the last n\_anc bits of } row \oplus f(\text{the first n\_inp bits of } row) = i$$

By simple manipulation of the boolean equations, one could find the the condition holds as expected.

We did this reverse engineering strategy because random access is known to be faster on rows then on columns, in numpy arrays, which really matters for the bigger input sizes.

```python
def create_dj_circuit(n, f):
    circuit = qiskit.QuantumCircuit(n + 1, n)

    # Initialize last qubit to 1
    circuit.x(n)

    circuit.barrier()

    for i in range(n + 1):
        circuit.h(i)

    circuit.barrier()

    U_f = Operator(get_U_f_matrix(n, 1, f))
    circuit.unitary(U_f, reversed(range(n + 1)), "U_f")

    circuit.barrier()

    for i in range(n):
        circuit.h(i)

    circuit.measure(range(n), range(n))

    return circuit
```

Listing 4: Python example

This code makes the D-J circuit. We start by specifying the number of qubits. Then flip the last qubit to 1 as expected by the algorithm. The *barrier* is a directive for circuit compilation to separate pieces of a circuit so that any optimizations or re-writes are constrained to only act between barriers. We do it because we want our code to not be unexpectedly optimize, and keep it's main directives as is. For example, we want the inverting of the last qubit to be an independent part of the circuit. The circuit creation utelizes our U_f builder and follows the standard D-J construction. Notice that we have to feed in our qubits reversed, since the ordering of the qubits qiskit perceives is the oppisate of the one we took in class.

Lastly:

```python
if __name__ == "__main__":
    # functions = [
    #     lambda x: x & 1 == 0,
    #     lambda x: x & 1 == 1,
    #     lambda _: True,
    #     lambda _: False,
    # ]
    functions = {
        "balanced": lambda x: x & 1 == 1,
        "constant": lambda _: True,
    }
    try:
        n = int(sys.argv[1])
        f = functions[sys.argv[2]]
        print(f"{n} qubits, {sys.argv[2]} function")
    except Exception:
        print("Usage: python dj.py [n] [type: constant|balanced]")
        exit(1)

    circuit = create_dj_circuit(n, f)

    # print("Circuit:")
    # print(circuit.draw())

    if USE_IBMQ:
        provider = IBMQ.load_account()
        backend = provider.backend.ibmq_quito
    elif SIMULATOR_NOISE:
        provider = IBMQ.load_account()
        noise_backend = provider.backend.ibmq_quito
        noise_model = NoiseModel.from_backend(noise_backend)
        backend = QasmSimulator(noise_model=noise_model)
    else:
        backend = QasmSimulator()

    # Execute the circuit on the qasm simulator.
    # We've set the number of repeats of the circuit
    # to be 1024, which is the default.
    transpiled = qiskit.transpile(circuit, backend)
    job_sim = backend.run(transpiled, shots=NUM_SHOTS)

    # Grab the results from the job.
    result_sim = job_sim.result()

    if VERBOSE:
        print("Circuit Results:", result_sim)

    counts = result_sim.get_counts()
    result = int(max(counts.keys(), key=lambda x: counts[x]), base
=2)
    if result == 0:
        print("\tResult: Constant")
    else:
        print("\tResult: Balanced")
```

Listing 5: Python example

This code is far less ”quantum” and is more here for the sake of control. In particular, we start by constructing a balanced and constant function, we collect arguments from the user for which we should run, make the circuit, based on config at the top of the file, choose if we run on perfect sim, noisy sim, or real QC, we compile the job (using transpile), execute it, and print the results. Noting too special going on.

The creation of other algorithms was quite similar, with the test case built in a main control function, the circuit, based on the algorithm specifications in it's own function. The structure of U_f need not change between algorithms. Hence, this template could be followed to implement pretty much any Quantum algorithm in qiskit, with not too much of an issue.

2 Evaluation

2.1 Testing

In order to test our programs, we ran each program multiple times and aggregated statistics about the results. In general, the testing process for each algorithm/circuit was as follows:

- Test on an ideal simulator, for increasing n
- Test on a noisy simulator, for increasing n
- Test on the IBM Quantum Computer, for n as large as possible (5 qubits)

Each circuit created was run on one of the above platforms for a total of 1024 shots (the default). When possible, we attempted to run our programs multiple times. Also note that when running on the IBM Quantum Computers, we chose to run only on IBM Quito for consistency purposes.

We adopted this approach because running on the ideal simulator first is both the easiest, and can give us an idea of program correctness. That is, since the ideal simulator makes no errors, we can tell if our ported code is running properly before we run it on a machine which obscures this correctness with it's massive unreliability. Then running on the noisy simulator serves as another stepping stone as we build up towards the real thing, allowing us to incrementally check correctness and adjust our expectations for the real QC's running. Finally, we run on the quantum computer, which is our end result. Our pattern of software engineering is at least inspired by Test Driven Development, where the theoretical results of the algorithm, on each of simulator(s)/QC are the tests, and we develop to make them work.

Before we give specific results, we'd like to mention the overall pattern: the noisy simulator produced worse results than the ideal simulator, and the quantum computer produced even worse results than that. This was, of course, expected. We also noticed the general trend of accuracy decreasing as the number of qubits increased, which matched our pre-experiment predictions, since as the number of qubits increases, not only do we have a longer runtime and more gates, but more expensive SWAPs. Our ability to perform each level of testing varied due to limitations surrounding our access to computing resources. While we were able to test ideal and noisy simulations of all circuits, we were sometimes constrained to testing smaller examples of problems due to time constraints. Furthermore, our problem sizes were severely constrained when running on the IBM computer, as we detail in the next few subsections.

We tried to find the properties and bias statistics of IBM's Quito to make better sense of our results, but unfortunately IBM has not made that information public.

Let us discuss the testing of each algorithm, in particular:

2.1.1 Deutsch-Jozsa

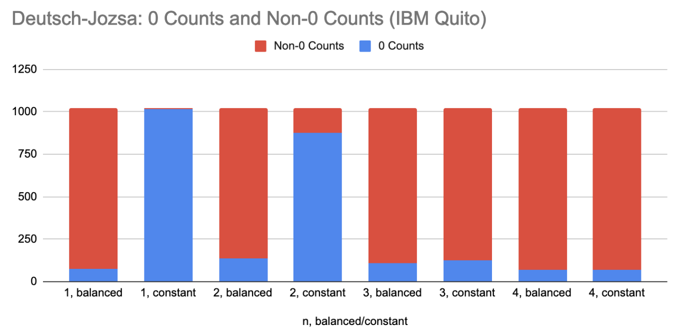
On an ideal simulator, our results for Deutsch-Jozsa matched the theoretical expectations. When f was constant, all executions of the circuit finished with the final state 0, and when f was balanced, none of the executions of the circuit finished with final state 0. This result gave us confidence in the correctness of our circuit, as theory dictates that the constructive and destructive interference leveraged by the circuit gives these exact results.

On a noisy simulator, we began to see a break from the theoretical results. For all values of n, including small n, we saw that we did not have 100% 0 outcomes on constant functions, and did not have 100% non-0 results on balanced functions. However, we still did see a breakdown between 0 and non-0 outcome states that did allow one to correctly infer whether f was balanced or constant. For example, for $n = 1$ and a constant function, 97% of the runs ended in state 0. We can compare this to the circuit for $n = 7$ and a constant function, for which only 76% of the runs ended in state 0. Because the number of gates and qubits increases linearly with n, the overall complexity of the circuit grows, and so noise becomes more and more of a factor. Despite this decrease in accuracy, we are still able to correctly classify between balanced and constant functions by determining whether 0 was the most common output state.

On the IBM Quantum Computer, the accuracy of the results decreased even more dramatically as n increased. As seen in the graph below, or in the table in the section *Ideal Simulation*, for $n = 1$, the balanced function circuit correctly outputted non-zero 92% of the time and the constant function circuit correctly outputted zero 99.5% of the time. However, for even just $n = 3$ (as well as $n = 4$), the circuits outputted an almost uniform distribution over all possible output states. Thus, the number of counts for 0 on larger n became too low to differentiate between balanced and constant circuits. Because of this, it became practically impossible to classify between balanced and constant functions except for by luck - at this point, the strategy of checking the output state with the most counts failed, as this would cause us to output 'balanced' a majority of the time. This is demonstrated very clearly in the graph below - while for larger n, counts would be distributed evenly across many states, we consistently see a low number of counts for state 0, for both balanced and constant functions.

Here's our data for runs on Quito:

Figure 1: Deutsch-Jozsa Result Count Breakdown (IBM Quito)



2.1.2 Bernstein-Vazirani

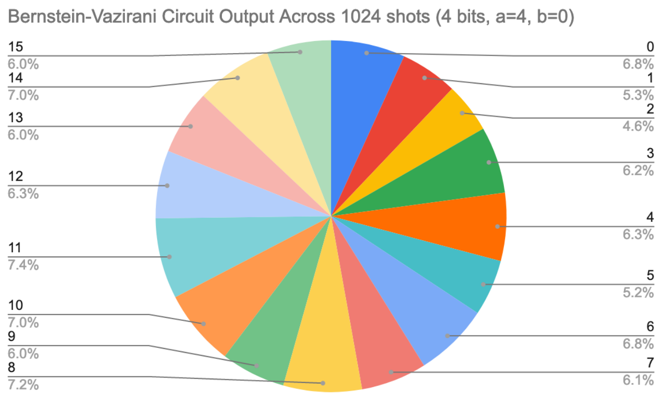
In general, correctly finding b was achieved across all tests. This is because the method for finding b is performed without running any quantum circuit. Instead, we can feed the appropriate 0 value into the function f, and setting the output to b. Thus, the following discussion focuses on finding a.

On an ideal simulator, our results for Bernstein-Vazirani were perfect. Without any noise or imperfection, the circuit's performance matched theoretical expectations across all 1024 shots for each circuit, allowing for ideal circuit runs that exactly yielded the value for a, for all tested $1 \leq n \leq 10$ and both $b = 0$ and $b = 1$.

On a noisy simulator, we noticed that while the circuits would no longer produce perfect results on every shot, running each circuit across enough shots and taking the results with the most counts was good enough to perfectly obtain a across all $1 \leq n \leq 7$. Due to the noise, some circuit shots would result in incorrect outputs. However, the large majority of shots would still be correct, whereas the incorrect shots were largely distributed across possible incorrect states, such that the correct answer had significantly more counts as was relatively easy to pick out. However, if n became too large, it's possible that we would no longer be able to overcome the issues brought up from noise.

On the IBM Quantum Computer, we achieved reasonable results for $n = 1$ and $n = 2$. While the count of correct runs was lower than both ideal and noisy simulations, a large majority of circuit runs ended in correct results. However, the noise and errors for anything larger $n = 2$ (specifically, the cases where $n = 3$ and $n = 4$) seemed to result in terrible results. For these cases, the circuit outputs were distributed almost evenly, meaning that the chance of actually finding the correct a was almost up to chance. To give an example, one job we sent to the IBM quantum computer was based on the job corresponding to a function $f : \{0,1\}^4 \rightarrow \{0,1\}$ defined with $a = 4$ and $b = 0$. In the pie chart below, we see that the counts of outputs of all shots are spread over all 2^4 possible states, each with similar probability. Based on these results for $n = 3$ and $n = 4$, it would be fair to assume that results for even larger n would be just as affected by noise and errors.

Figure 2: Bernstein-Vazirani Result Count Breakdown (IBM Quito)



2.1.3 Simon

On an ideal simulator, our results for Simon were very good. While Simon is an algorithm with some variation in number of circuit runs required (due to the nature of finding constraint equations via quantum circuit), we almost always obtained the correct value for s. While the number of circuit runs used varied based on n, testing on the ideal simulator allowed us to check the correctness of our constraint solving code and the soundness of our circuit generation.

On a noisy simulator, our results were mixed. We found that for most of our program runs, we successfully found s. However, on the remaining attempts, our program found that $s = 0$ (when s actually was not 0). This may be the the result of the fact that if our y_i are linearly independent, then the set of obtained equations has solutions s and 0. In theory, if our y_i are chosen independently of one another, we are not guaranteed that the equations are linearly independent. We also note that the Simon circuit is particularly sensitive to noise. Noting that for a problem of size n, $2n$ qubits are required, this means that as n increases, the number of qubits will increase at twice the rate, so noise will scale even more.

We were not able to run many different tests on the IBM Quantum Computer. As we were constrained to 5 qubits, the only nontrivial tests we could run were for problems with $n = 2$, requiring $2 \cdot 2 = 4$ qubits. Of these, we found similar results to the noisy simulations in that our program would output either s or 0 after solving the constraint equations. One interesting thing to note is that the number of times a circuit was run for a specific problem varied. Across the tests performed, we saw that between 1 and 3 jobs were sent to the IBM Quantum computer, before a sufficient number of constraints were obtained. It's important to note that because a relatively large number of qubits were needed

(4 out of 5 available qubits), noise significantly affected the circuit results. For example, one result (of the many we ran) we obtained for was the following:

Output	Counts
0	228
1	224
2	332
3	240

As can be seen, though there are more counts for 2, all possible outputs had a fairly similar number of counts, which shows the effect of noise on the circuit execution. Similar output distributions were seen for our other results, which explains the relatively poor results we obtained compared to our simulation results.

2.1.4 Grover

On an ideal simulator, our results for Grover were very good. Without any noise or imperfection, the circuit's performance matched theoretical expectations across all 1024 shots for each circuit, allowing for high probabilities of obtaining the correct index for the given function. While we did not achieve 100% success across all n, we did get very high success rates overall (and 100% success for $n > 2$).

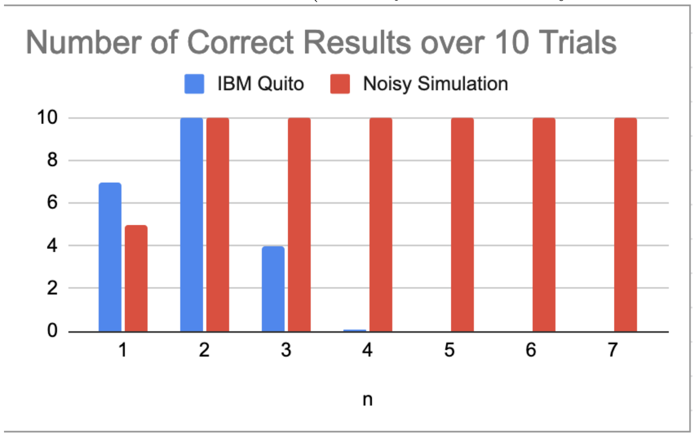
On a noisy simulator, we noticed that while the circuits would no longer produce perfect results on every shot, running each circuit across enough shots and taking the results with the most counts was good enough to match the success seen while running on the ideal simulator. Due to the noise, some circuit runs would result in incorrect outputs. However, the large majority of runs would still be correct, whereas the incorrect runs were largely distributed across possible incorrect states, such that the correct answer had significantly more counts as was relatively easy to pick out in most cases.

On the IBM Quantum Computer, we achieved success rates for $n = 1$ and $n = 2$ that paralleled our success rates while running on the ideal and noisy simulators. However, for the cases where $n = 3$ and $n = 4$, the circuit outputs were widely distributed, and finding the correct index was almost up to chance. in the case where $n = 3$, our success rate dropped drastically. Furthermore, we incorrectly picked an index on every single attempt for $n = 4$. On one example run (for $n = 4$, with the index being 5 such that only $f(5) = 1$), we obtained the following counts, which clearly shows the dramatic effects of noise:

Output	Counts
0	81
1	73
2	64
3	62
4	61
5	54
6	65
7	51
8	76
9	51
10	65
11	57
12	84
13	56
14	69
15	55

The graph below shows the results of running 10 circuits for each n and comparing the circuit output with the actual index, for both our noisy simulation and our run on the IBM Quito machine. Based on these results for $n = 3$ and $n = 4$, it would be fair to assume that results for even larger n would be poor, as well.

Figure 3: Grover Correct Trials (IBM Quito and Noisy Simulator vs n)



2.1.5 QAOA

Testing on an ideal simulator, we found that QAOA performs slightly better than random in some cases, but approximately the same as random overall. Running on graphs of size $n \in [3, 9]$, we found that these results matched the results obtained when running via Cirq simulation, which made sense. These results were similar to the results obtained when testing on a noisy simulator. Our results can be attributed to our use of the `scipy` optimization library, which was an attempt at reducing the number of circuits we would need to run (by being a bit smarter on how we chose parameters γ and β).

Unfortunately, we found that we were limited in our ability to run the entire QAOA algorithm on the IBM Quantum computer. As mentioned above, our algorithm utilizes `scipy` for its optimization algorithms, specifically the `minimize` function. However, our usage of this function requires iteratively generating circuits and executing them to obtain information about their optimality (in hopes of finding good values for γ and β). Thus, in order to compromise between experimenting with running on a real quantum computer, and to avoid taking over the queue, we decided to run only the final circuit on the IBM Quantum Computers (and to run the optimization circuits locally via simulation). Additionally, due to constraints of the IBM Quantum Computer, we ran circuits corresponding to only up to $n = 5$. We ultimately found that even when using a real quantum computer more prone to error, that running the circuit obtained partitions that were comparable to the results from an ideal simulator. In fact, for our $n = 4$ graph, the QAOA algorithm produced a partition that was slightly better than random. One interesting example of a result we obtained for $n = 4$ is the following:

Output	Counts
0	49
1	21
2	29
3	92
4	57
5	69
6	64
7	55
8	90
9	87
10	87
11	57
12	155
13	50
14	44
15	18

Here, we noted that noise clearly affected our results, as seen in the other circuits. In spite of this, we did see that our results were not necessarily uniformly distributed - there actually was a peak at 12, which was an encouraging sign that the QAOA circuit can still produce results even with noise. However, this was one of our more ‘successful’ runs, in that regard, as most other runs did not see spiking as significant as this. The overall trend of the circuits was to have increasingly noisy results as a result of increasing n, which seems to suggest that for larger n, our results may not be too effective.

Ultimately, we concluded that our results were satisfactory, given the circumstances. However, given more computing resources, it would be interesting to run more circuits on the real quantum computer to gain a better understanding of the practical application of QAOA.

2.1.6 Shor

Overall, we found that our implementation of Shor's algorithm was severely limited by the size of the circuit. Because the circuit is fairly complex, and uses a significant number of gates and qubits, even generating a circuit and simulating it on an ideal simulator was not possible for many numbers. For the limited numbers (for example 15) that we were able to test, we found that our implementation succeeded in finding a correct factor (5) on an ideal simulator. Running on a noisy simulator also yielded a factor (also 5), but required many more iterations of running the circuit, as earlier iterations would not find an appropriate factor. Finally, we were unable to run our circuit on any nontrivial numbers on Quito, due to the limitation on the number of qubits available. However, based on the results of our other experiments, it is likely that the results would be even worse than the noisy simulations.

It is unfortunate that our qompute was so limited, as running Shor's Algorithm to factor some legit numbers on a quantum computer would have really been the highlight of this assignment.

2.2 Scalability

One thing we noticed was that the circuits we sent to the IBM quantum computer were severely limited in terms of the number of qubits that could be utilized. Specifically, the circuits we constructed were limited to a maximum of 5 qubits, meaning that it was impossible to send the circuits of some of the more complex algorithms to the IBM quantum computer (apart from circuits for trivial instances of the problem).

To work around this, we also investigated the runtime of the circuits on a noisy simulator that mimicked the behavior of the IBM quantum computer. While this did provide a somewhat feasible alternative, it may not make sense to directly compare the execution times of the IBM quantum computer with the execution times of the simulations. Instead, we compare the overall growth patterns of the runtimes between the IBM quantum computer and the simulator, expecting that in general the runtime of circuits on quantum computers should grow much less significantly, based on theory. The IBM quantum computer consistently takes seconds longer than simulations to complete. This is because the simulations are relatively low overhead and have no physical restrictions compared to the quantum computer. On the other hand, the quantum computer is a real, physical machine that has practical considerations.

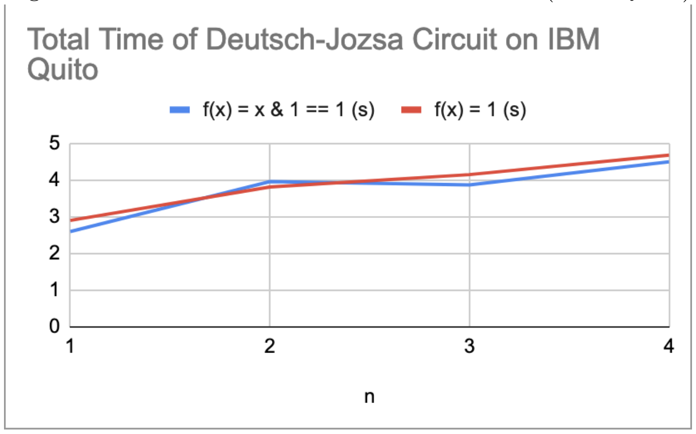
In spite of these challenges, running on both a simulator and on IBM Quito yielded some interesting results on scalability. Many of our results paralleled the theoretical expectations, though performance was obviously hindered through practical considerations of running on a real, physical machine. Having seen these results, we are extremely excited for the potential of quantum computing, especially as hardware advacements and improvements allow for better performance and scalability.

2.2.1 Deutsch-Jozsa

We were able to run the Deutsch-Jozsa circuit on the IBM Quantum Computer for problems focusing on balanced and constant functions taking in bitstrings of length of at most 4. For problems where $n \geq 5$, 5 or more qubits are required (in general, for a function taking in a bitstring of length n, $n+1$ qubits are required).

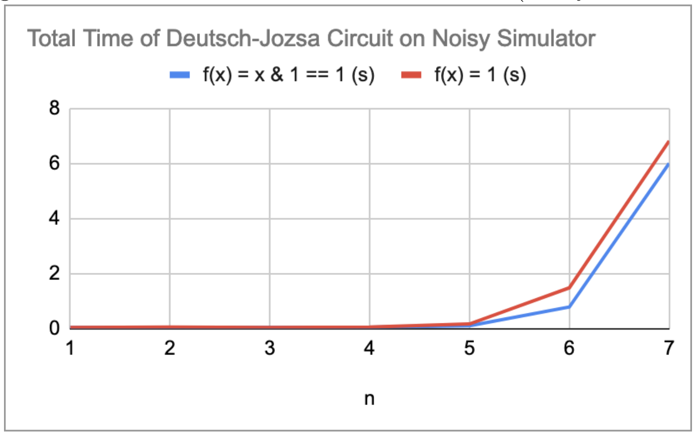
On the IBM Quantum Computer, we saw that for small n, the circuit runtime grew relatively slowly. Our group ran several different circuits with n varying between 1 and 4. Additionally, for each n, we tested one balanced function, and one constant function. We did not see a significant difference in performance between the constant and balanced functions used, though given more compute power and time, it may be an interesting direction to explore. Given the chance to experiment with larger n, I would predict that the runtime of circuits as n will continue to increase, but at a slower rate than the noisy simulator executions. This prediction is based on the theoretical $O(1)$ runtime, compounded with practical implementation limitations of running on a real quantum computer (such as register allocation/swapping and the effectiveness of compiler optimizations).

Figure 4: Deutsch-Jozsa Execution Time vs n (IBM Quito)



On a noisy simulator on our own machines, we saw that for small n, the circuit performed extremely quickly, on the order of tens of milliseconds. However, as n grew beyond $n = 4$, the circuit runtimes both began to grow exponentially. We also noticed a slight discrepancy between the circuits, based on which of the two functions it was based upon. This could have been caused by a slight difference in the complexity of the U_f sub-circuit needed to represent the two functions, and is something that could be further explored.

Figure 5: Deutsch-Jozsa Execution Time vs n (Noisy Simulator)



2.2.2 Bernstein-Vazirani

We were able to run the Bernstein-Vazirani circuit on the IBM Quantum Computer for problems focusing on balanced and constant functions taking in bitstrings of length of at most 4. For problems where $n \geq 5$, 5 or more qubits are required (in general, for a function taking in a bitstring of length n, $n+1$ qubits are required).

Something to note is that the performance of Bernstein-Vazirani circuits paralleled the performance of Deutsch-Jozsa circuits. This is because the two circuits are practically identical - the underlying structure of the circuit is able to be repurposed to solve the two differing problems, and this is clearly reflected in the scalability of these circuits.

Having said this, we see that the growth of circuit runtimes (with respect to n), on both the IBM Quantum Computer and on the noisy simulator, mirrors the growth seen for the Deutsch-Jozsa circuits on the IBM Quantum Computer and on the noisy simulator, respectively. For the IBM Quantum Computer, runtimes grew extremely slowly, though we were not able to run circuits beyond $n = 4$. This corresponds to the theoretical expectations that the algorithm runs in $O(1)$ time, but may also include some practical implementation limitations that result in small runtime growth (factors such as register swapping, for example). For the Noisy Simulator, we see that while runtimes are relatively quick for small n, the exponential growth of runtimes is more apparent for large n.

Figure 6: Bernstein-Vazirani Execution Time vs n (IBM Quito)

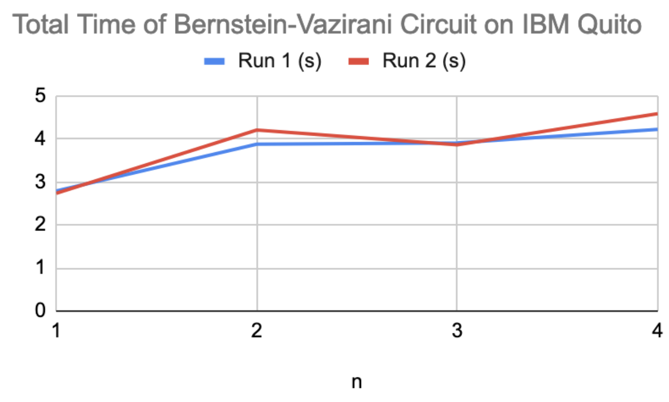
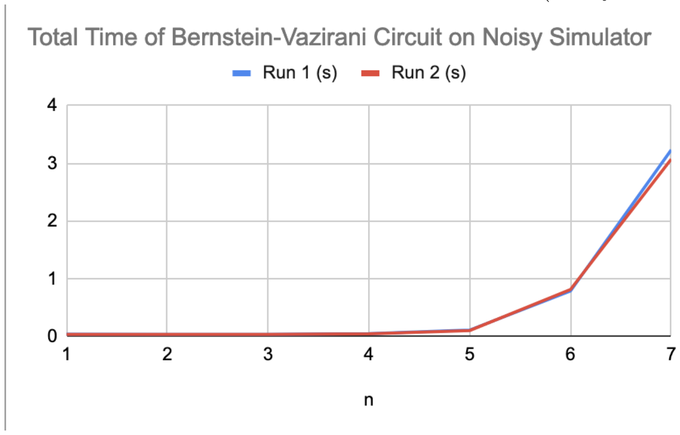


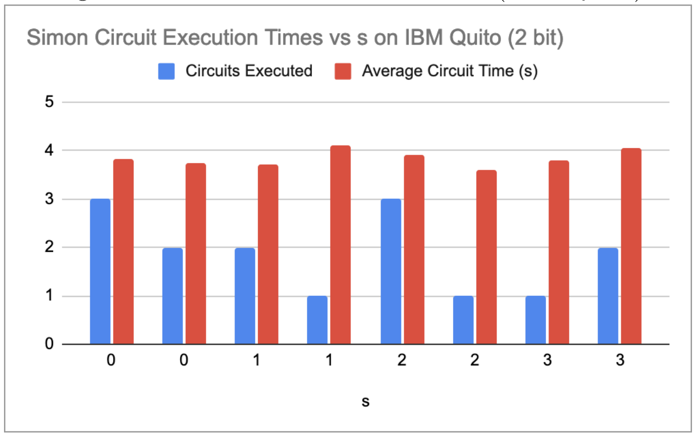
Figure 7: Bernstein-Vazirani Execution Time vs n (Noisy Simulator)



2.2.3 Simon

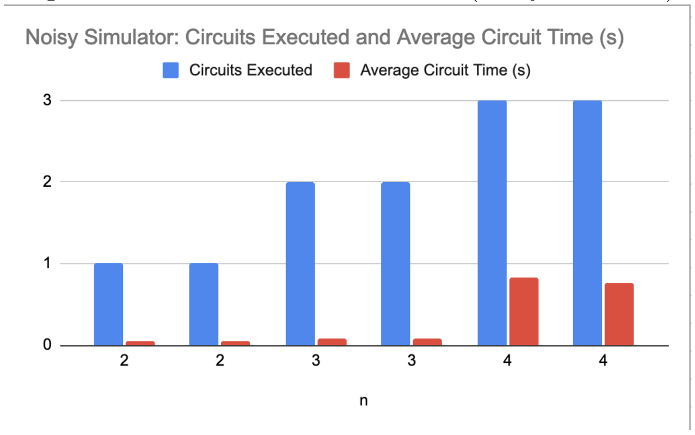
Because Simon uses $2n$ qubits, the only non-trivial case we could experiment with on the IBM machine was the case where $n = 2$. One experiment we attempted was varying the value of s. Across all $s \in \{0, 1, 2, 3\}$, we saw that average circuit time was consistent (approximately 3.5 - 4 seconds). Additionally, the average number of circuit executions before completion was also uniformly distributed, varying between 1 and 3 executions. Both these results make sense - because the complexity and size of the circuit do not depend on the value of s, changing s should give similar results. Theoretically, the runtime of these circuits on a real quantum computer grows less then exponentially (in contrast to the simulator's performance) with respect to n, but the number of circuits executed could grow at roughly the same rate due to the necessary condition of finding a certain number of equations to obtain enough constraints to solve the overall problem.

Figure 8: Simon Execution Time vs s (IBM Quito)



On a noisy simulator, we tested circuits for $n \in \{2, 3, 4\}$. We found that the number of circuits executed scaled with n, as did the average time to execute a circuit. However, the circuit runtime seemed to grow at a much more rapid rate, and roughly resembles an exponential increase. Given more computing resources, we could verify that these trends continue for $n > 4$.

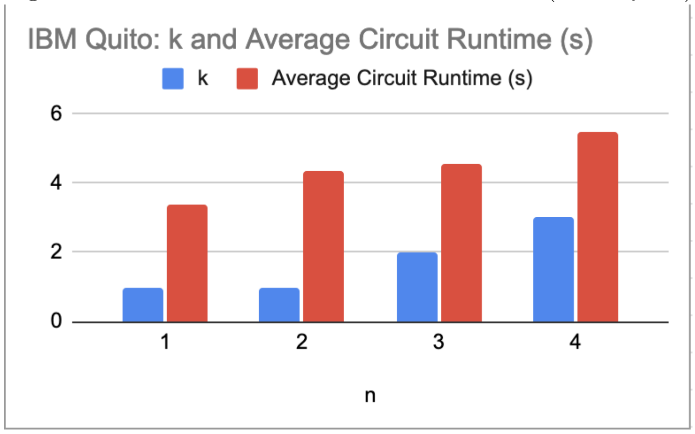
Figure 9: Simon Execution Time vs n (Noisy Simulator)



2.2.4 Grover

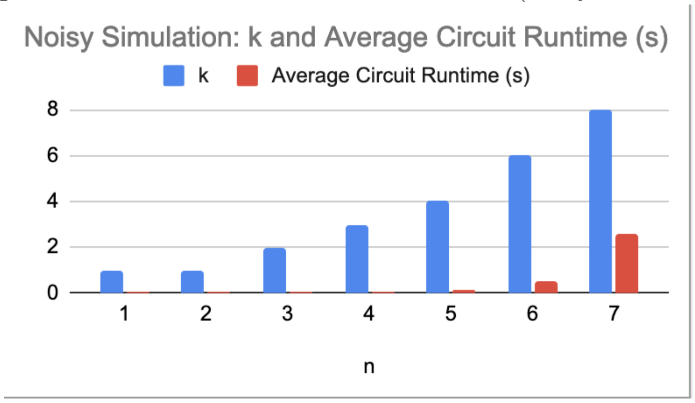
We saw that runtimes grew with respect to n on the actual IBM computer. While we were constrained to experimenting with circuits corresponding to $n \leq 4$, we saw that for small n ($n \in \{0, 1\}$) the runtime was low and for larger n, the runtime began to grow ($n \in \{3, 4\}$). However, it did seem that the growth rate of the runtimes was lower than the runtimes of our simulations, which reflects the powerful properties of quantum computing compared to simulations and classical computing. If we had the opportunity to experiment with circuits corresponding to $n \geq 5$, we would expect for the runtimes to continue to grow, but at a rate significantly lower than the exponential growth seen for the simulator's circuit executions.

Figure 10: Grover Execution Time and k vs n (IBM Quito)



Similarly, on a noisy simulator, we found that the average circuit runtime grew exponentially with respect to n. Having tested for all $n \leq 7$, we see that for small n ($n \leq 4$), the the average circuit runtime is on the order of hundredths of a second. However, for $n \geq 5$, runtime expands significantly, for $n \geq 8$. This makes sense because as n increases, several factors increase as well: the number of qubits needed, the complexity of U_f's gate representation (as our implementation uses U_f as a sub-circuit), and the number of iterations k to apply the Grover gates.

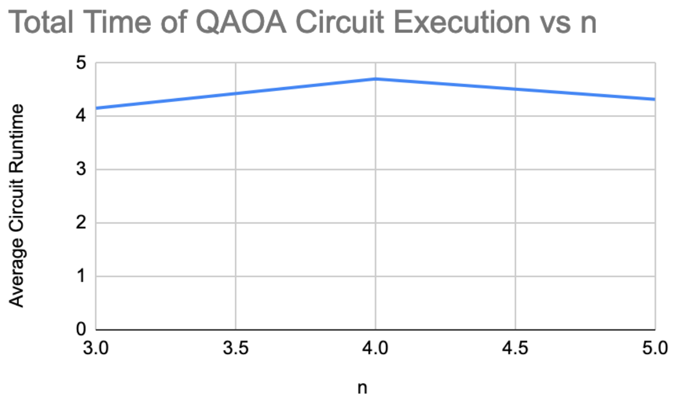
Figure 11: Grover Execution Time and k vs n (Noisy Simulator)



2.2.5 QAOA

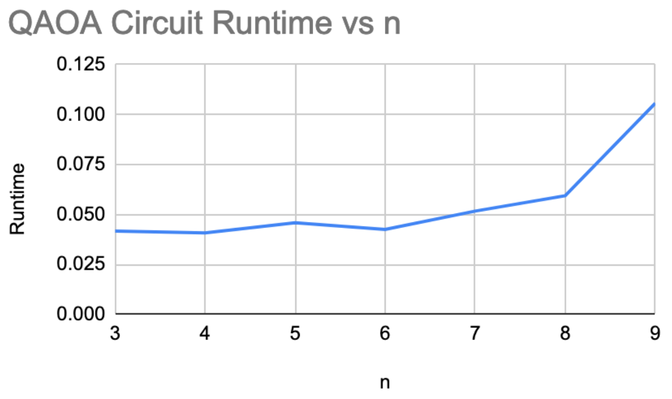
We were relatively constrained in our scalability testing of the QAOA algorithm on the IBM quantum computers. However, we were able to run circuits corresponding to graphs of size $n \in [3, 5]$. Interestingly, we found that running the circuit for these sizes did not yield significant differences; these circuits all ran in roughly the same time. However, it is important to note several caveats about our results. First, it is possible that scalability becomes more apparent as n scales beyond $n = 5$. Additionally, the runtime could be affected the the complexity of the graphs involved. While the QAOA circuit is based on the number of nodes, it is also constructed based on the connections between these nodes. Overall, though, the runtimes of these circuits was extremely impressive, especially due to the fact that the problem it solves is NP-Complete (and no classical algorithm can solve this problem in an efficient way).

Figure 12: QAOA Average Execution Time vs n (IBM Quito)



On a noisy simulator, we saw the runtime of the QAOA circuit simulation grow significantly for larger n. For relatively small n, the runtime grew minimally and was relatively stable. However, once n grew sufficiently large, we saw that the runtime of the corresponding simulations began to grow exponentially. For even larger n, generating and running the circuit became too lengthy to complete. This is to be expected, as the size and complexity of the circuit grows significantly and takes a significant amount of time to simulate on a classical computer. Therefore, the scalability of QAOA on a noisy simulator matched our expectations, and showed the limitations of simulating quantum circuits on classical machines.

Figure 13: QAOA Average Execution Time vs n (Noisy Simulation)



2.2.6 Shor

It was difficult to perform many tests or experiments with Shor's algorithm, due to the circuit's size and use of many qubits. However, we were able to perform some limited analysis on the small number of feasible circuits and their performance on simulators. We saw that typically, we would only need to run a few iterations of our circuit circuits in order to factor 15, the minimum being 1 and the average being 2. Comparing the runtimes of our Cirq and Qiskit implementation, it was clear that our Cirq implementation executed in much less time. On average, our Cirq circuit ran in 0.24 seconds, whereas our Qiskit circuit ran in 13.85 seconds. This discrepancy could be caused the a difference in their implementation. While Cirq's simulator is based on numpy and is described as 'a sparse matrix state vector simulator', Qiskit's execution approach is much more complex. The idea is to transpile an abstract quantum circuit into an equivalent circuit that can run on some machine, whether it be an actual quantum computer or a simulator. Thus, Qiskit offers a lot of power in terms of practicality and running on available machinery, in return for worse simulation performance.

Performing some research gave us insight into the limitations of Shor's algorithm. Shor's algorithm was first demonstrated by IBM in 2001, when 15 was factored using 7 qubits (source). In 2012, factoring 21 was accomplished. And in 2019, there was an unsuccessful attempt to factor 35, which failed due to error accumulation. Therefore, even for a number such as $15 < 2^4$ which can be represented in 4 classical bits, it can only be factored by a circuit requiring more qubits than IBM allows for. In reality, using Python and Cirq features to analyze our own circuits showed that our implementation of Shor's algorithm for factoring 15 would theoretically use 12 qubits, and our implementation of Shor's algorithm for factoring 21 would theoretically use 14 qubits. Scaling Shor's algorithm is a current-day challenge that, once addressed, could lead to crucial breakthroughs in quantum computing.

2.3 Simulation vs Actual

While we did compare simulated results against results obtained from an actual quantum computer, we use this section to explicitly elaborate some of our findings.

2.3.1 Ideal Simulation vs Quantum Computer

Running our circuits in the Qiskit simulation backends yielded dramatically different results than running them on the IBM quantum computer. The main difference is that by default, the Qiskit backends attempt to simulate an ideal quantum computer, while in reality, quantum computers may experience some imperfections as a result of the technology's current state. Thus, while the simulators often match our theoretical expectations for various circuits, the same sometimes cannot be said for their execution on real quantum computers.

One clear example of this is the Deutsch-Jozsa circuit. In theory, we expect the circuit to always evaluate to 0 if f is constant, and to always evaluate to some non-zero value if f is balanced. While the simulators give this result 100% of the time, the IBM quantum computers do not. In fact, as n increases, the success rate of the circuit decreases. The following table records the counts of the results obtained from running our Deutsch-Jozsa circuit on the IBM quantum computers:

Circuit	0 Counts	Non-0 Counts
n=1, balanced	78	946
n=1, constant	1018	6
n=2, balanced	137	887
n=2, constant	876	148
n=3, balanced	107	917
n=3, constant	128	896
n=4, balanced	68	956
n=4, constant	67	957

This table shows that even for small n, the execution of the circuit follows but does not precisely match the theoretical expectations. As n increases, the output counts get distributed amongst all possibilities more and more evenly, meaning that the likelihood of f being falsely labeled as balanced increases significantly.

2.3.2 Noisy Simulation vs Quantum Computer

Our group also compared the results of the IBM quantum computer, with the results of the Qiskit simulator configured to factor in noise based on the real-life IBM quantum computer. Overall, the results matched our expectations. The noise increased the rate at which errors occurred, meaning that the circuits would not yield perfectly theoretical results. However, a quick skim of the data seems to suggest that the real IBM quantum computer featured more noise, or at least exhibited outcomes that were further from the theoretically expected results than the noisy simulator. Noisy simulations were interesting to experiment with because even though our noise profiles were supposedly based on the same machine we ran simulations on (IBM Quito), the simulator was not able to introduce noise and errors on the same order as what we experienced on the real machine. Additionally, we note that while all of our testing was performed on IBM Quito, there could be some variability in performance between different IBM quantum computers, which would be an interesting topic to explore.

2.3.3 Practical Usage Comparisons

One practical difference between the two computation methods was the existence of the job queue for the IBM quantum computer. Though the computer could execute our circuit fairly quickly, it often took several minutes in order to have a turn at sending our circuit to the computer. Thus, gathering data and testing our programs on the actual quantum computer often took significantly more time due to the amount of time spent waiting.

However, attempting to simulate circuits using a noisy simulator also took a significant amount of time. While the actual execution of the circuit took longer than on an ideal simulator, the process of transpiling exhibited the largest increase in wait time. This could have been caused by the added complexity that simulating noise brings to the situation - adding additional steps and imperfections, mimicking real life, requires adjusting the circuit to account for errors.

3 Instructions

Please refer to `README.md` for how to provide input, how to run the program, and how to understand the output.

4 Experience

Overall, the process of converting the programs was extremely challenging. The technique of obtaining a QASM string from the cirq circuit, then sending that to a qiskit backend, was often not possible due to various incompatibilities and issues. For example, the use of a custom matrix-based gate to implement U_f was one issue that caused the generation of a QASM string to fail.

Thus, for all of the programs, it was necessary to re-implement the core quantum computing circuit in Qiskit. This provided an opportunity for us to compare the overall development experience of the two environments. Many of the fundamental aspects of Quantum circuit design were similar - both environments had similar syntax for adding gates to circuits, defining custom gates, measuring, etc. Overall, though, Cirq felt more user-friendly. Cirq's documentation was especially helpful. It was richer and provided more useful examples, as opposed to Qiskit's overly verbose and difficult-to-parse reference pages. Another aspect of Qiskit that was difficult to adjust to was the different tensor ordering it uses. While most courses and cirq have the right-most qubit (of a tensor product) being the n^{th} qubit, Qiskit uses an ordering where the n^{th} qubit is on the left side. This was something that we did not initially notice, and was somewhat frustrating to discover.

In our orcale construction method, we cycled through columns instead of rows for a slight efficiency boost. We did this since in our previous projects, just the construction of U_f on large n seemed to have become a bottleneck. In retrospect, we are not sure that it was beneficial as it made the code more diffciult to understand.

Not to come off complacent, but overall, it seems that most of the issues we ran into were a result of the youthfulness of the field of quantum computing. That is, perhaps the biggest lesson that we took from this class about quantum programming is that quantum computing is currently in such a rudimentary place that it's just too early to worry about interfaces, languages, gate-set optimisations, and software engineering practices. In fact, we feel that it is likely that quantum computing will change so radically before it becomes practical (if it ever will) that much of this high-level progress with languages and such will be rendered completely irrelevant. Since our machine was so small and so unreliable, it hardly mattered how nicely we set up our code and any optimisations that we instilled into it. To fix these shortcomings, one would first have to "fix" quantum computing, a task that we are passionate about.

To end on a positive note: While Qiskit was harder to use, it did provide a lot of interesting functionality. The Qiskit ecosystem provides a wide range of backends that allow for one's circuits to be run on various simulators or even actual Quantum computers. Additionally, Qiskit makes it possible to customize the simulators used to run circuits - one powerful feature it has is the ability to add adjustable noise to a simulator, to better capture the imperfections seen in real Quantum computers as they are now. So while Qiskit was frustrating to use, it did have a lot to offer.

5 Reference

To view the data and tables used to generate the graphs in this document, please see this link.