CS239 - UCLA
SPRING 2019

HOMEWORK 11 : REPORT

Using Cirq for Quantum Algorithms: Simon's and Grover's

GROUP MEMBERS

Rishab Doshi - 305220397
Srishti Majumdar - 105225254
Vidhu Malik - 305225272

Simon’s and Grover’s

Part 1. Introduction

This section presents a brief summary of the implementation of the solution to the two problems.

For Simon’s algorithm,the utility functions used are same as those used for solving DeutschJozsa and BernsteinVazirani problem.

Description	Function
Get decimal number from binary bit String	getDecimalNo(bitString)
Get qubit vector from the binary bit vector String	getQubitVector(bitString)
Get all possible bitstrings(2^n) for n bits	getAllPossibleNBitStrings(n)

Table 1.0.1: Common Util Functions

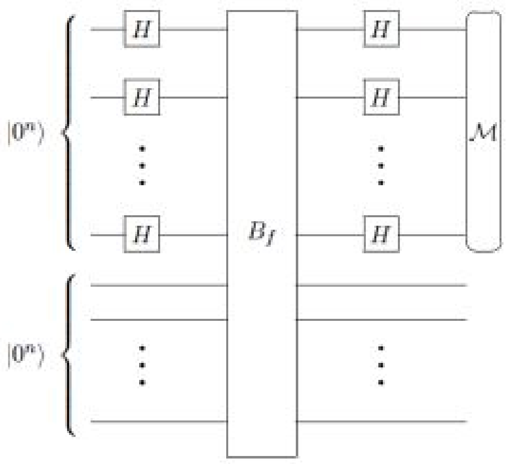
The parameterization and encapsulation of the code is as done previously for DeutschJozsa and BernsteinVazirani problem. We make use of a class Oracle which extends cirq.Gate to create custom gate B_f .

For Solving Grover’s, we make use of a function similar to getAllPossibleNBitStrings(n) called get_possibilities(n). We also make use of class Oracle to help create gates for $-Z_0$ and Z_f .

1.1 Solution to Simon’s problem

Quantum Circuit

This part of the program uses CirQ to create the quantum circuit as below. The circuit used for Simon’s problem is shown below. The B_f gate mentioned in the circuit is what we would be referring to as the U_f gate. We define an oracle class that extends cirq.Gate class defined in cirq to implement U_f. The *constructor(__init__), _unitary_, __str__, num_qubits* functions are then overridden. This class is then instantiated with the UfMatrix as input and the resulting object can easily be appended to the cirq circuit for further use.



Generation of U_f matrix

To implement U_f (the oracle function gate) we make use of the fact that U_f has to be a permutation matrix. We use the below steps to generate U_f matrix are:

1. If the input size of X is N, then generate all bit combinations of size 2N.
2. Map each combination to its decimal counterpart. We would hence get a mapping from the bit combinations to its first 2^(2N) decimal counterparts.
3. The decimals represent the row and column indices of the unitary matrix.
4. We know that Uf |x> |b> = |x> |f(x) + b> where '+' is addition mod 2.Hence, each N bit |x> is associated with N bit possible values for b to get the above-mentioned 2N input bits.
5. Now out of these bit combinations, the first N bits are sent to the function f, and the generated output f(x) and b are added modulus 2 to get the N output bit. The first N bits of input and output are the same. Now, using these input to output mappings we can fill the matrix.
6. The input bits are used to get the row number and the output bits are used to get the column number using the mapping created between bit patterns and indices initially.
7. All other bits are made 0 in that row and column.
8. Once the process is completed for all input bit combination, we would not have the unitary matrix that is also a permutation matrix and represents the function f in its reversible form.

Post-processing on Quantum Circuit[1]

1. We use the results of the quantum circuit to build a matrix of (n-1) linearly independent vectors. The way this was done was that, we kept running the circuit until we had (n-1) linearly independent vectors.

[1] http://lapastillaroja.net/wp-content/uploads/2016/09/Intro_to_QC_Vol_1_Loceff.pdf

2. The way we identified if the given vector was linearly independent to existing vectors in the matrix was by continuously maintaining the row reduced echelon form for all the vectors that were added to the matrix. If the addition of the vector resulted in a vector of all 0s in the row reduced echelon form, then this vector was discarded.
3. The exact implementation of maintaining the row reduced echelon form was done by ensuring that there was only one row that had a 1 in the same MSB of the vector. If the circuit produced a vector whose MSB didn't conflict with other existing vectors then, it was added to the matrix, else we attempted to add the XOR of the conflicting vectors. This process was repeated until either the row was added or the XOR resulted in a bit vector of all 0's, leading to discarding of the row.
4. After obtaining (n-1) linearly independent vectors we add a final vector to the matrix that is not orthogonal to the result. We do this by identifying the row where the matrix is missing a 1 in its diagonal. Corresponding to the same row, we add a 1 to the result(ensuring that the dot product is not 0, hence not orthogonal). Now we have a matrix of n equations and n unknowns, we can solve this using a classical solver. We used the numpy library to solve these equations.

1.2 Solution to Grover's problem

In Grover's algorithm, we have $G = -H^{\otimes n} Z_0 H^{\otimes n} Z_f$.

The solution to Grover's problem is as follows:

Let X be a collection of n qubits that initially is $|0^n\rangle$.
1. Apply $H^{\otimes n}$ to X.
2. Repeat { apply G to X } $O(\sqrt{2^n})$ times.
3. Measure X and output the result.

We know $Z_f|x\rangle = (-1)^{f(x)} |x\rangle$ and $Z_0 = I - 2 \ |0^n\rangle\langle 0^n|$.

Generation of Z_f matrix

From the definition of Z_f , we see that we can build this unitary matrix through a simple trick. For every possibility where f(x)=1 {x is of n bit length}, we multiply that column in the matrix with -1.Hence, effectively converting $|x\rangle$ to $(-|x\rangle)$.

Generation of Negative Z_0 matrix

We notice G has a negative sign and we incorporate this in Z_0 to make negative Z_0 . We, first create an Identity matrix of size 2^n. We, then, convert the element corresponding to position [0][0] to -1. Finally, we negate the whole matrix.

Applying algorithm

1. We create two possibilities for functions:
    a. Has a unique x such that f(x)=1
    b. Has one or more x such that f(x)=1
2. We randomly select the type of function and again randomly generate the function, corresponding Z_f and Z_0. We build gates out of the matrices obtained using a class Oracle.
3. We apply H to all qubits in $|0^n\rangle$
4. For $\{\pi/4\,(\sqrt{2^n}) - (1/2)\}$ iterations:
    a. Apply Z_f.
    b. We apply H to all qubits.
    c. Apply $-Z_0$.
    d. We apply H to all qubits.
5. We measure our outcome.

Part 2. Evaluation

2.1 Results

Discuss your effort to test the two programs and present results from the testing.

Simon's Problem -

1. We tested the Simon's problem on 2 qubits with the secret value being equal to 01, and generated (2-1) = 1 vector, i.e., we run the quantum circuit once, added another vector to the matrix and solved the equation to get the value of S. The values obtained for S from the quantum circuit were consistent with the original value of S.
2. We also tested the Simon's problem on 3 qubits with 4 different secret values and received correct results for all the cases. We created a driver function that abstracts away all the implementation details and runs the quantum circuit till we get (n-1) linearly independent vectors and finally adds the nth vector and solves the set of equations to get S. The details of the runs have been tabulated in the next section.

Grover's Problem

1. Grover's algorithm was checked for two kinds of functions:
    a. Has a unique x such that f(x)=1
    b. Has one or more x such that f(x)=1
2. We checked execution times for various functions across n=1,2,3,4. For all n, for each type of function, we ran multiple simulations.
3. For each kind of function, 3 simulations were chosen for various n such that the Z_f obtained is unique for each point in the plot.

Note: Our Grover's algorithm gives the correct answer with nearly 100% accuracy every time for functions of type 1.a. For functions of type 1.b., we notice that the accuracy dips, this is because the probability of having f(x)=1 gets distributed and hence, the chance of getting the correct answer slightly reduces. This notion was further validated by doing the math for Grover's as learnt in class.

2.2 Execution time for different U_f

Discuss whether different cases of U_f lead to different execution times.

Simon's Problem

For evaluating the change in execution times for different U_f, we tested out the Quantum part of Simon's algorithm for three different functions with n=3 and recorded the execution time for each of them. The functions are as mentioned below:

f1: (s = 110)

x	f1(x)
000	101
001	010
010	000
011	110
100	000
101	110
110	101
111	010

Table 2.2.s.1: Mapping when s=110

f2: (s=001)

x	f2(x)
000	101
001	101
010	000
011	000
100	110
101	110
110	010
111	010

Table 2.2.s.2: Mapping when s=001

f3: (s=010)

x	f3(x)
000	101
001	010
010	101
011	010
100	000
101	110
110	000
111	110

Table 2.2.s.3: Mapping when s=010

For each of the above functions, we evaluated the execution time for three trials and averaged it to obtain the overall execution time for each function. Although each of the functions is designed for n=3, they all have different U_f matrices associated with them. A summary of the execution time is presented below:

Trials	Function In Consideration		
	f1	f2	f3
Trial 1	0.0089	0.0119	0.0079
Trial 2	0.0079	0.0079	0.0069
Trial 3	0.0089	0.0099	0.0089
AVERAGE	0.008566s	0.0099s	0.0079s

Table 2.2.s.4: Execution Time for each function over each Trial

Unlike the results observed for PyQuil, there is very little variation in the results for CirQ. The execution times do not vary much amongst trials. More importantly, the execution time seems independent of the kind of function and hence independent of U_f. All three execution times are in the order of 10^{-3} s and fall well within that range.

In the below tables, we will look at the total time spent on running the quantum circuit for different values of n.

Simulation 1

No. of Qubits	Quantum Runs	S	Time	Avg. Time per Run
2	1	01	0.0025	0.0025
3	3	110	0.009	0.003
3	2	010	0.006	0.003
3	12	010	0.0305	0.0025

Table 2.2.s.5: Simulation 1 data for Simon's Algorithm

Simulation 2

No. of Qubits	Quantum Runs	S	Time	Avg. Time per Run
2	7	01	0.0147	0.0021
3	4	110	0.0104	0.0026
3	5	010	0.0160	0.0032
3	3	011	0.0066	0.0022

Table 2.2.s.6: Simulation 2 data for Simon's Algorithm

Simulation 3

No. of Qubits	Quantum Runs	S	Time	Avg. Time per Run
2	2	01	0.021	0.0105
3	2	110	0.0077	0.00385
3	2	010	0.0064	0.0032
3	3	011	0.0081	0.0027

Table 2.2.s.7: Simulation 3 data for Simon's Algorithm

As can be seen from the above tables, the time taken in seconds for same n value is almost similar. With 2 qubits(simon circuit size 4) we are seeing a runtime of 0.001 seconds - 0.002 seconds and with 3 qubits(simon circuit size 6) we are seeing an average runtime of ~0.003s. In the case where n=3, we have different S values, which means we have different U_f values for these, however we see that the runtimes are almost same. However, for a smaller no. of qubits with another U_f, we see a change in the average time to run.

Using the small sample-set we have seen the conclusion we can draw is that for same no. of qubits, irrespective of different U_f values we will see approximately similar runtimes.

Grover's Problem

We see the following trends:

- When more than one x may exist such that f(x)=1.

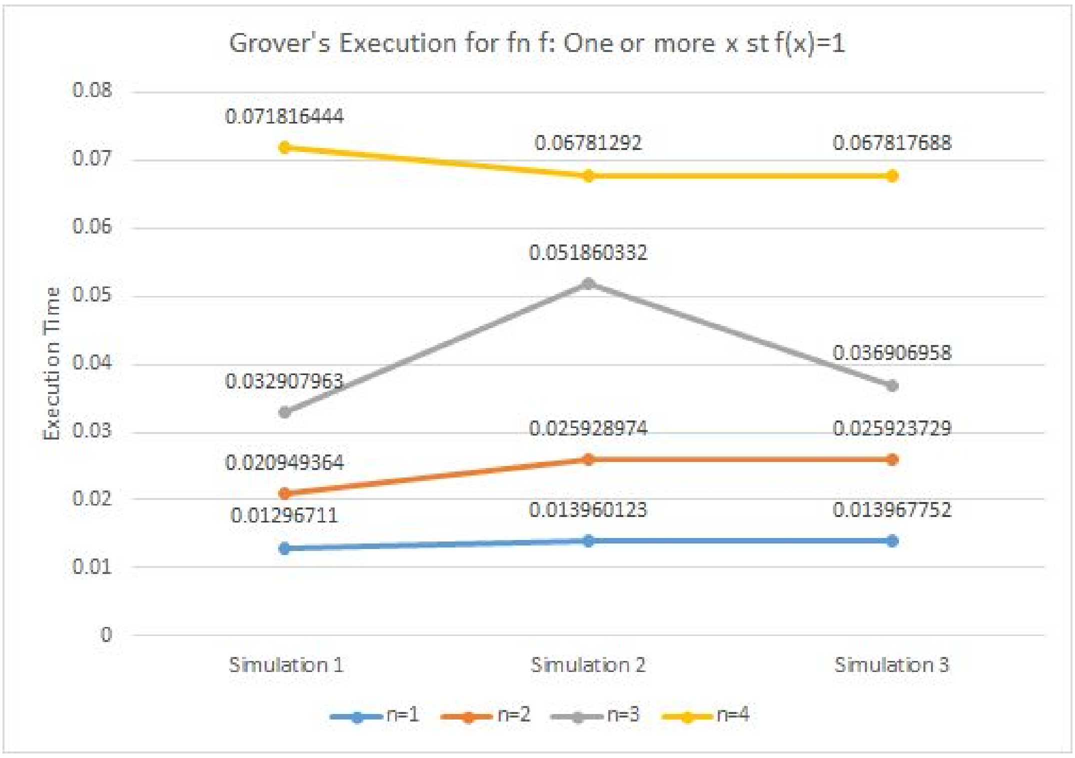


Figure 2.2.g.1: Execution Time for n=1,2,3,4 for Grover's where multiple x st f(x)=1

- When only one x such that f(x)=1.

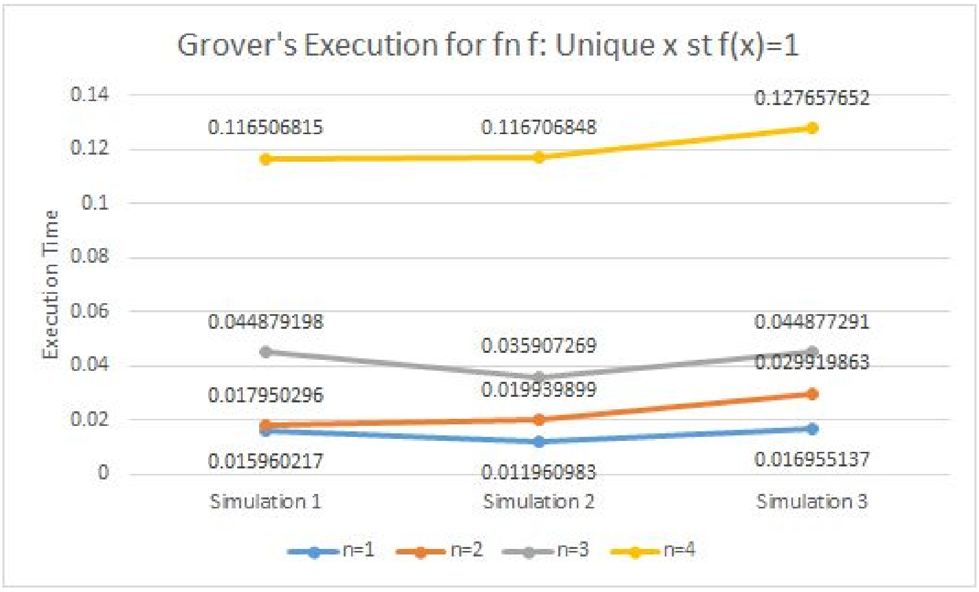


Figure 2.2.g.2: Execution Time for n=1,2,3,4 for Grover's where only one x st f(x)=1

We see that no matter what the function type is, the execution time required is similar across Z_f for the same n.

2.3 Execution time as n grows

Simon's Problem -

In table 2.3.s.1 and Figure 2.3.s.1, we have recorded the time it takes to run the code for functions of different number of qubits ranging from 2-4
The growth is clearly exponential. An increase in even a single qubit can cause significant increase in runtime. Although, it should be noticed that the growth does not seem to be as steep as that of PyQuil results.

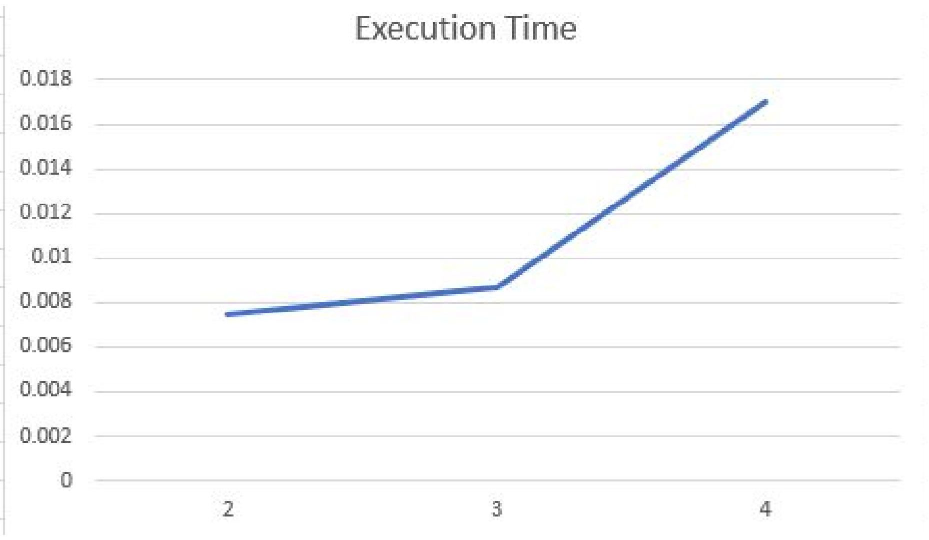


Figure 2.3.s.1: Execution time as a function of no. of qubits

N	Time
2	0.0075
3	0.008786 secs
4	0.017 secs

Table 2.3.s.1: Data corresponding to Figure 2.3.s.1

Grover's Problem -

From figures 2.2.g.1 and 2.2.g.2, we see that as n increases, time required for execution increases.

The increase in time with n can be observed in Figure 2.3.g.1. Similar trends can be observed for all simulations irrespective of type of function.

We see that from n=9, the execution time increases exponentially. This is due to the way the number of iterations increases. Details regarding number of iterations for each n is given in Table 2.3.g.1. At n=13, the execution time lasts more than a minute. So while we observe greater scalability compared to PyQuil (without setting any Timeout), for large n, execution time increases exponentially in Cirq.

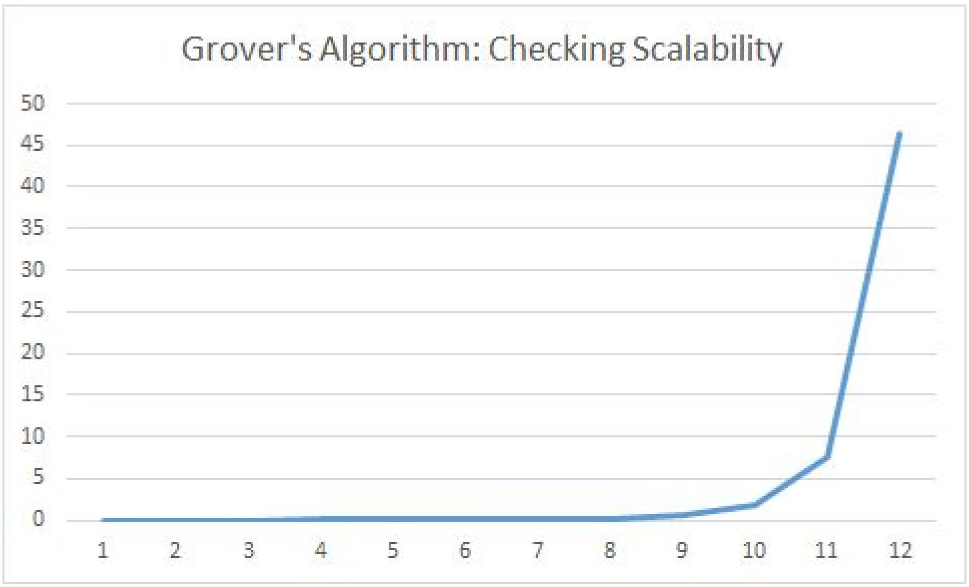


Figure 2.3.g.1: Execution Time as n increases for Grover's Algorithm

n	Number of Iterations	Execution Time
1	1	0.008970976
2	1	0.012965918
3	2	0.016951084
4	3	0.02293992
5	4	0.042884588
6	6	0.098742723
7	8	0.106720209
8	12	0.17054534
9	17	0.549530745
10	25	1.708476782
11	35	7.58490181
12	50	46.43086195

Table 2.3.g.1: Data Corresponding to Figure 2.3.g.1

Part 2. Instruction

The readme file has been attached with the submission files.

Part 3. PyQuil

3.1 List three aspects of quantum programming in Cirq that turned out to be easy to learn and list three aspects that were difficult to learn

Positives:

1. Usage of Quantum Gates: Majority of the common quantum gates like Hadamard, CNOT etc. are available for direct use and there are lots of helpful examples that help programmers get started.

2. Abundance of helper functions and abstractions: Every common requirement for a quantum program like running the circuit, measuring the results, repeating the quantum trial multiple times are available as helper functions which make programming in Cirq easy and fun!

3. Composition of programs: Since quantum circuits are composed of multiple gates, the same analogy is followed to create a Cirq program and that makes it easy to learn how to compose a program to run a Quantum circuit.

Negatives:

1. Custom Gate Definitions and Usage: We created custom gates that were built based on user defined matrices for defining oracle functions for different algorithms. The intuition around how this done is somewhat unclear at first. The documentation implies the extending the abstract class cirq.Gate to define a class that implement a unitary matrix and transform it to a gate. But this is not explicitly mentioned which makes understanding the creation of gates and understanding what code to reuse difficult.

2. Interpretation of measurement results: The results of running the simulations required closer inspection to truly understand the qubit states after

measurement. It was available as an object called TrialResult which had to be further inspected and parsed to obtain the actual qubit states. Although, not difficult, this required some closer inspection.

3. Finding errors in code: While creating custom gates, it is difficult to trace if the unitary matrix being applied is correct, especially when the program supports multiple outcomes probabilistically. Providing unitary matrix in the form of numpy arrays also makes visualising matrix without certain transformations difficult.

3.2 List three aspects of quantum programming in Cirq that the language supports well and list three aspects that Cirq supports poorly

Positives:

1. Intuitive interface to quantum programming: Since quantum programs are essentially a collection of gates, the concept of a Program as a collection of gates that can be incrementally added to the circuit help in making the transformation from quantum algorithms to quantum programming easy.

2. Ability to view Quantum Circuit at all intermediate steps: The fact that we are able to visualize the quantum circuit by simple print statements aids in readability and understanding of the actual overall quantum program.

3. Cirq is very verbose: As compared to PyQuil, cirq is very verbose, clearly defining the circuit, gates, abstract class for new gates, and even measurement gates! Hence coding in Cirq, is very deterministic.

4. Ability to provide qubit ordering and amplitude ordering while performing measurement is a very useful feature as quantum circuits often rely on a range of different qubits and operations and transformations to these different qubits.

Negatives:

1. Better error Messages: When the quantum simulation fails, sometimes the error messages given by cirq are not very informative and actionable as shown in various github issues[2].

[2] https://github.com/quantumlib/Cirq/issues/1514

2. Interpretation of measurement results: TrialResult object is provided as an output to measurement of a quantum circuit in cirq, the interpretability of this object and learning to use it is not trivial. It would be better if the results were provided in a popular python data structure like a pandas dataframe[3].

3. Quantum Circuit Interpretation / Action Verification: Quantum circuits operating on qubits are not very interpretable if they are not one of the standard circuits. i.e., usually, the result of the quantum circuit is possibly known only to the creator, unless the inspector of the circuit performs the math required to understand the result. Unit tests are one of the best ways to understand the functioning of a program, There is scope for Cirq to define how quantum unit tests(QUT) can be run(directly on QVM) and also design of QUT's to aid understandability of quantum circuits.

3.3 Which feature would you like cirq to support to make the quantum programming easier?

Since, all quantum circuits tend to have oracle functions, it would be useful if cirq can provide a way to pass a function pointer and the number of qubits and internally built a quantum oracle Uf based on the given function and n value.

3.4 List three positives and three negatives of the documentation of cirq

Positives:

1. Concise : The cirq documentation, tutorials and library are much more concise and focussed on coding examples than that of PyQuil. The documentation has a lot of useful code snippets such as those for gate usage, defining custom gates,etc that are very helpful for coders who are just starting to use CirQ. The tutorials are very well written and the documentation is filled with a lot of examples, illustrations and code explanations.
2. API Reference: The API Reference provided by CirQ is very clear and well written. Most function references are clear enough for users to not require and extensive detailed definition of the function. It specifies the function protocol as well as a very clear description of what the function achieves.
3. Display of circuit outputs: CirQ has a very useful feature of displaying circuits that it creates to make debugging easier. Most of the documentation provides

[3] https://github.com/quantumlib/Cirq/issues/1689

these circuit snippets for all different features and commands. This makes it easier for readers to visualise the function and hence understand it better.

Negatives:

1. Introduction to Quantum Programming: Unlike PyQuil, the introduction provided towards Quantum programming is rather complex. It will be difficult for an amatuer user to start using the language without having a thorough background or formal education in the subject.
2. Example order can be improved: Along lines similar to the previous negative, the documentation of examples begins with the complex BCS Mean Field which can possibly lead to an amateur finding it intimidating to start using Cirq. It would be beneficial to have a progression by introducing simpler algorithms like Deutsch-Jozsa and then going onto more complex ones. Although, these examples are present, they are not in order of difficulty.
3. Better explanations for extending classes: Since a lot of cirq features depend on extending classes and overriding functions, it would be useful for the documentation to carry enough support for this. For example, to implement the U_f matrix as a gate, the Oracle class needs to override cirq.gate. While this is implied through the examples, the step by step procedure and explanation of every step for doing this is not provided.

3.5 In some cases, cirq has its own names for key concepts in quantum programming. Give a dictionary that maps key concepts in quantum programming to the names used in cirq

Quantum programming concepts	cirq counterparts
Helper Bits	Ancilla
Time Slice of an Operation	Moment
U_f	Parametric gates
Usage of Universal gates	Quantum decomposition

Table 3.5.1: Difference Between Quantum Programming and cirq

3.6 How much type checking does the cirq implementation do at run time when a program passes data from the classical side to the quantum side and back?

Basic checks like ensuring passed in arguments to the gates are of right type is ensured, we verified this by passing in String values as arguments to gates and we received a TypeError. We also observed that extensive type checking is performed when data is passed to the quantum side. If a matrix that is being sent for gate creation via Oracle is not unitary, Cirq recognizes that. This is important as non-unitary operations are not allowed on quantum devices implying that it is not feasible for real qubits to undergo a non-unitary transformation. It also type-checks return types like recognising if the U_f matrix is not in the form of a numpy array while defining our gate.