CS239 - UCLA

SPRING 2019

HOMEWORK 9 : REPORT

Using PyQuil 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 1.1: Common Util Functions

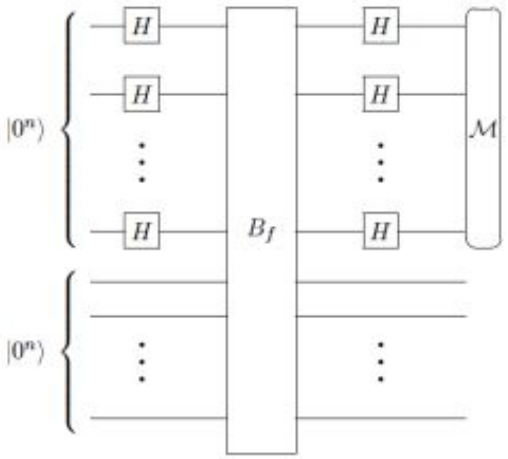
The parameterization and encapsulation of the code is as done previously for DeutschJozsa and BernsteinVazirani problem.

For Solving Grover's, we make use of a function similar to getAllPossibleNBitStrings(n) called get_possibilities(n).

1.1 Solution to Simon's problem

Quantum Circuit

This part of the program uses Pyquil 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 use the DefGate library in PyQuil to create a U_f gate based on the U_f matrix generated.



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 [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.
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 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.

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

f2: (s=001)

x	**f2(x)**
000	101

001	101
010	000
011	000
100	110
101	110
110	010
111	010

f3: (s=010)

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

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:

	f1	**f2**	**f3**
trial1	52.82	39.16	57.8
trial2	51.9	35.7	58.18
trial3	68	45	73.62

AVERAGE	57.57	39.93	63.2s

While the time taken to run the functions seem to fluctuate quite a bit amongst trials, it can be noticed that the consistency between the order of trials across functions is maintained. f2 consistently takes lesser time than f1 that takes lesser time than f3. Clearly the choice of U_f affects the execution time. But irrespective of the choice of U_f the overall order of time remains the same. All the execution times are in the range of 1 minute. A much larger variation in execution times can be seen when n is varied as discussed in the next section.

In the below table, 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	2	01	2.38	1.19
3	2	110	59.17	29.59
3	4	010	123.21	30.80
3	6	010	180.72	30.12

Simulation 2

No. of Qubits	Quantum Runs	S	Time	Avg. Time per Run
2	2	01	2.41	1.2
3	5	110	150.74	30.25
3	2	010	59.57	29.785
3	4	011	115.90	28.975

Simulation 3

No. of Qubits	Quantum Runs	S	Time	Avg. Time per Run
2	2	01	2.5259	1.26
3	2	110	57.40	28.7

3	4	010	125.98	33.99
3	2	011	62.63	31.315

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

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 an average runtime of ~1.25 seconds and with 3 qubits(simon circuit size 6) we are seeing an average runtime of ~30s. 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

n	Sim 1	Sim2	Sim3
1	0.40192771	0.165556192	0.123668671
2	0.388958454	0.225396633	0.283242941
3	1.551850557	1.489017725	2.827439785
4	8.54614687	10.28949165	10.09999466

Table 2.2.g.1: Data values corresponding to Figure 2.2.g.1

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

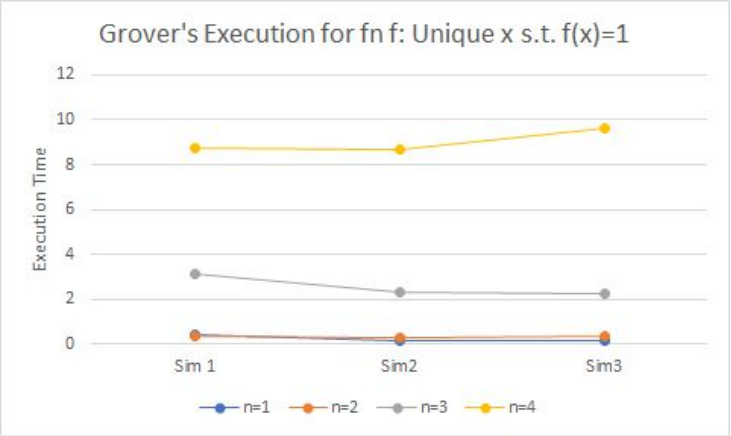


Figure 2.2.g.2: Execution Time for n=1,2,3,4 for Grover's where only one x st f(x)=1

n	Sim 1	Sim2	Sim3
1	0.380981684	0.12781477	0.137633324
2	0.313161373	0.247341633	0.348069668
3	3.15057683	2.304452896	2.239336491
4	8.735645533	8.664830446	9.609306335

Table 2.2.g.2: Data values corresponding to Figure 2.2.g.2

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

What is your experience with scalability as n grows? Present a diagram that maps n to execution time

Simon's Problem -

Below, 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.

N	Time
2	2.26 secs

3	50.2 secs
4	1431 secs = 23.85 mins

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. If timeout not explicitly set, the algorithm times out at n=5.

The increase in time with n is observed to be sort of exponential. We have shown an example below. Similar trends can be observed for all simulations irrespective of type of function.

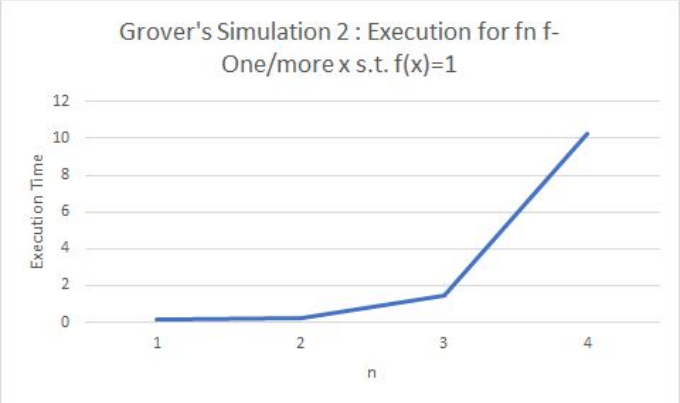


Figure 2.3.g.1: Execution Time as n increases for Grover's where multiple x st f(x)=1

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 PyQuil 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 PyQuil easy and fun!
3. **Composition of programs:** Since quantum circuits are composed of multiple gates, the same analogy is followed to create a PyQuil 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 of DefGate and the associated example don't have an example where the input qubits are parametrized in N, a common requirement for oracle gates.
2. **Interpretation of measurement results:** Although not difficult to interpret, our intuition was that calling run_and_measure would give us results of all n qubits in a single list, and no. of lists in the results would depend on the no. of trials. However, it was the opposite, where the rows are all measurement values for the same qubit and length of each row is no. of trials.
3. **Passing multiple qubits to gates:** It is not straightforward as to how multiple qubits are passed to a quantum gate.

.

3.2 List three aspects of quantum programming in PyQuil that the language supports well and list three aspects that PyQuil 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 Quil Code at all intermediate steps:** The fact that we are able to view the code in the intermediate QUIL language and its readability helps in understanding/visualizing of the actual overall quantum circuit.
3. **Different granular levels of Functions:** Pyquil provides ability to apply operations on specific qubits and also separately measure qubits(although it may collapse the quantum state). This is helpful for someone who might be only concerned with the state of a particular qubit. On the other hand a beginner might want to not be concerned about the internals of measuring specific qubits but would want to run the circuit multiple times, this is made available as a function that operates on all qubits. To summarize, abstractions at different granularity levels make PyQuil useful for even expert programmers and beginners.

Negatives

1. **Debuggability of Quantum Programs:** Since any sort of inspection of qubits will lead to a collapse into that particular basis, debugging quantum circuits seems to be a general problem, not just limited to PyQuil. Since Quil can be simulated on a QVM, it would be useful if PyQuil allowed us to inspect the value of qubits without collapsing its state.
2. **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 PyQuil 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. **Oracle Functions:** Since oracle functions are a common requirement, it would be useful for users to be able to pass in function pointers and get custom oracle gates that can be used in the quantum circuit.

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

Since, all quantum circuits tend to have oracle functions, it would be useful if PyQuil 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 PyQuil

Positives

1. **Introduction:** The PyQuil documentation has a section called “Introduction to Quantum Computing” that introduces new users to the concepts of quantum computing/ is a refresher for amatuer users. It contains all basics of quatum computing, like dirac notation, basic gates, measuring of qubits, etc.
2. **Useful Code snippets and tutorials:** The documentation has a lot of useful code snippets that are very helpful for coders who are just starting to use PyQuil. Code snippets have been provided for gate usage, defining one’s own gates, etc. The tutorials that are provided are very well written and the documentation is filled with a lot of examples, illustrations and code walk throughs.
3. **Availability of extensive changelog for contributors:** PyQuil is one of the few open source project that has maintained a change log that recors the contributions of all contributors and hence helps users in adapting the code for newer releases.

Negatives

1. **Installation instructions:** PyQuil official documentation page does not provide any support for window’s users even though the Rigetti Forest SDK is available for Window’s users. It was also found that under certain constraints like the usage of Window’s powershell instead of Command prompt, the quilc -S command throws an error. There is no mention of this error or steps to avoid it in the documentation
2. **Lack of documentation for contributors:** Although PyQuil is an opensource setup, it seems to lack the instructions required by contributors to effectively contribute their code to the framework and to follow standard guidelines

3. **Expert oriented explanation of the quantum computer architecture:** The documentation that explains the architecture and its components like QAM, QVM and so on is hard to understand for a beginner audience. These components can be explained in layman's terms so that they are easier to understand for a wider range of audience.

3.5 In some cases, PyQuil 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 PyQuil

Quantum programming concepts	**PyQuil counterparts**
Superposition of states	Combined states
Usage of Universal gates	Quantum decomposition
U_f	Parametric gates
b	ancilla

3.6 How much type checking does the PyQuil 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 DefGate is not unitary, PyQuil 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.