Zach Harris, 105221897
Trevor Rollins, 105227465

# Simon and Grover in PyQuil

## I. Evaluation

Both algorithms were rigorously tested to verify the predicted with simulated value. Each experiment is evaluated using the `run_simon` and `run_grover` function calls. The circuit is first built using the corresponding `Simon` or `Grover` class, and then a qvm is created. We use `with local_qvm()` to spin up and connect to the servers provided by PyQuil. We then compile the qvm with the previously-created circuit and independently measure the circuit a number of times. We measure more than once to show that although the circuits may not give the same result each time, the correct result is indeed the most frequent.

### Simon

For each cycle of the quantum portion of Simon's algorithm produces a y such that n-1 cycles are run to produce n-1 y vectors. From here, this series of dot products are used to find a set of valid values for s. Simon's algorithm is probabilistic in that there is at least a ¼ chance that all y's returned will be linearly independent. This means that there is a high chance over overlap between y's, and a single value for s will likely not be found from n-1 cycles. To test the algorithm, we first implemented a naive solver that checked every possible value of s with the set of y's returned from the quantum part. We ran many iterations (where each iteration is n-1 cycles) and compare the sets of valid s values returned. After enough of these iterations, the intersection of these sets will be of size one with a high probability. This is the non-zero value for s. In testing we found that a good value for number of iterations is 5, though this can be adjusted as a parameter of the `run_Simon()` function. The return value will be the set of valid s values found (ideally one value).

To test our algorithm, we used 4 oracle functions with input sizes (n) between 1-4. Each of these functions have a valid non-zero s value. In testing, when iterations < 5, a set of values of size > 1 was sometimes returned. This is to be expected, because if there is significant repetition between the y's used to generate the s values, then there will be many legal s values possible. Though a set of size > 1 is not ideal, these sets always included the real s value. This indicates that the algorithm was valid, and that the iterations just needed to be increased.

At iterations = 5, for every test run, a set of size 1 was returned and included the real s. We these results, we were confident that the algorithm performs correctly.

We implemented U_f for this algorithm by creating controlled X-gates (CX). A CX was created for each $2^n$ possible inputs for each output bit, b, when b = 1. The CX is controlled on the n input bits. For functions that have fewer output bits equal to 1, less CX's will need to be created, and execution time will be better. Also if CX are needed on input bits that are more heavily weighted towards equaling 1, these bits will not have to be NOT'd before and after the CX (because all control bits need to be 1 inside CX), which can also improve execution time. An example is shown in figure 3.

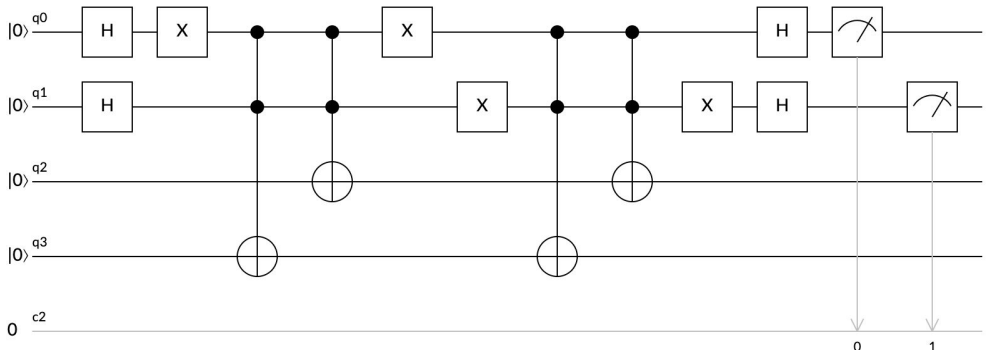


**Figure 3:** Example U_f for the function below

| x | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| f(x) | 00 | 11 | 11 | 00 |

On average however, half of the output bits will need CX's, and half the input bits will need X-gates before and after the CX. This leads to the number of CX gates needed to be: $2^n \cdot \frac{n}{2} = n2^{n-1}$ with each CX being controlled on n bits. And the number of X gates needed will be n(CX gates)= $n^2 2^{n-1}$ . This means that as n increases, runtime increases exponentially. This effect can be seen in Figure 4. Execution time does not scale well in this implementation of U_f.

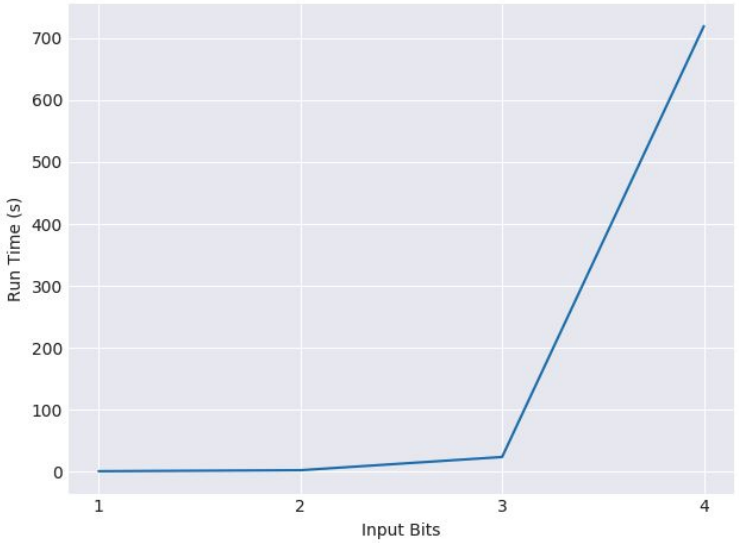


**Figure 4:** Run time of Simon's algorithm

### Grover

Because Grover is not guaranteed to give the correct answer with probability 1, it is necessary to test the outcome multiple times. We can, however, verify the probabilities by using the Wavefunction Simulator built into PyQuil. Albeit straightforward, the Wavefunction Simulator was giving different probabilities each time we reran the simulation. We suspect the Wavefunction Simulator can give incorrect probability estimates when using a particular QPC architecture. We argue this because the Quirk circuit simulator never has this problem.

In order to produce a more accurate measurement for the system, we simply run the simulation multiple times using the `numshots` parameter from the qvm and count the most common-occurring situation as the correct answer. We found that running the simulation for $2^n$ times was enough to ensure that the correct answer was the dominant result from the simulation. For example, when running n=4 and k=2, we get the following output from the terminal:

```
{'[1 0 0 0]': 2, '[1 1 1 0]': 2, '[0 0 1 0]': 5, '[0 0 0 1]': 2,
 '[0 1 1 1]': 2, '[1 1 0 0]': 2, '[0 0 0 0]': 1}
Most probable result: k=[0 0 1 0]
```

It is evident that k=0010 (or 2) is the correct answer, because the frequency of occurences is much higher than the others. After running multiple trials, we find that the probability of getting k=2 from the Aspen QPC is 50% on average. This is close to the probability

predicted by the Quirk circuit simulator, which is 47.3% (proof). We performed the same tests with n up to 6 to verify our results were accurate.

Our circuit scales very well with n. An example of a G block is shown in the figure below. The red box represents $z_f$, which is where f is introduced into the circuit. The Z gate is applied to any bit (in this case the first qubit), and a pair of X gates are added to either side for each bit in k that is 0. In this case, k=0 so there are a pair of X gates for each qubit. Notice that $z_0$, which negates the output if x=0, is equivalent to $z_f$ in this case since k=0.

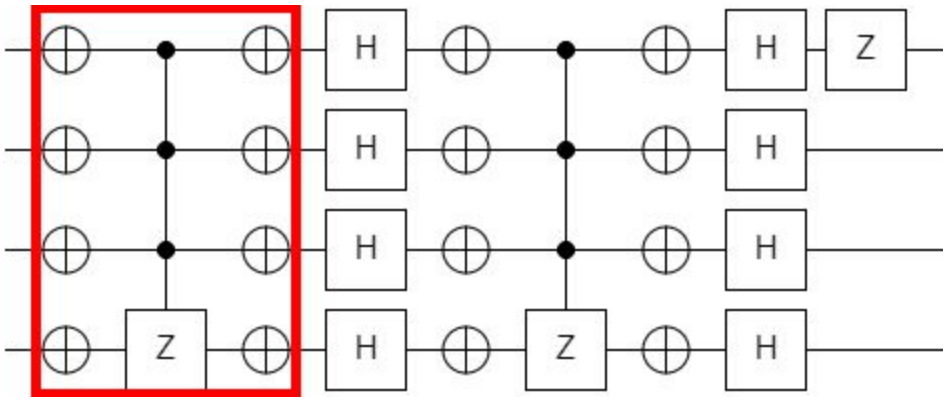


**Figure 1** G circuit block

In order to scale to any n, we simply use (n-1)-controlled Z gates and insert the X gate pairs where needed. Although this presents an easy problem to code, the number of controls scales linearly with n. Figure 2 below shows relationship between execution time and n. Compile time is included in this measurement. It is evident that although the number of controlled bits increases linearly, the execution time increases exponentially. This suggests that by adding controls to a gate will increase the execution time exponentially. There is little change from when the least number of gates are required (orange line, k is all 1's) to when the most number of gates are required (blue line, k is all 0's). This suggests that the time increase is entirely due the impact of adding more controls to the Z gates.

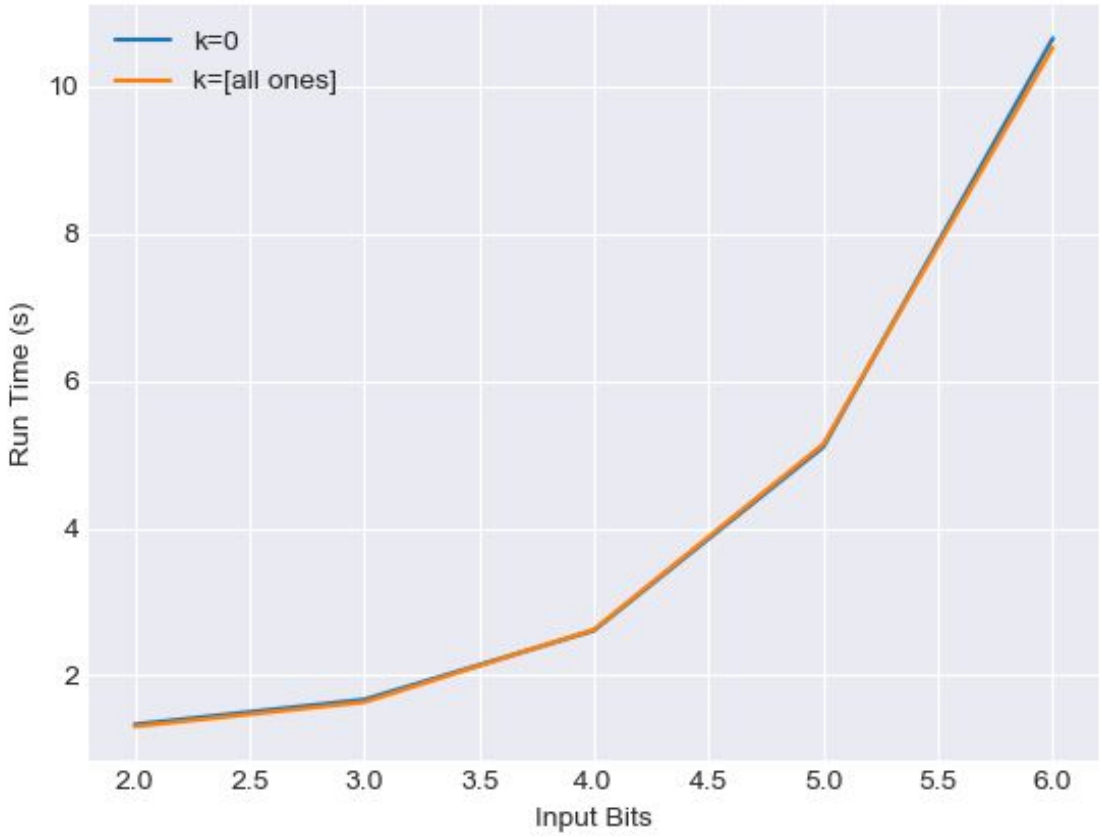


**Figure 2** Relationship between Grover execution time and n

## II. Instructions

The README.md file describes the packages necessary and steps involved to run the tests. Please read the descriptions below that summarize this file for each experiment.

**Simon**

`simon.py` is the script used to run Simon's algorithm on a given oracle function (f). To use this script, you need to type out f as a list of lists.

The outer list will consist of 2^n inner lists, where n is then number of input bits. Each inner list will be 2n in length.
The first n elements will be the input bitstring (x) to f, and the final n element will be the resulting output f(x).

Example:

```
f = [[0,0, 0,0], [0,1, 1,1], [1,0, 1,1], [1,1, 0,0]]
```

Here n = 2 and s = [1,1].
f must have a valid s value, or the algorithm will fail.

Once you have your f ready, all that is necessary is to call `run_Simon(f, naive=False)`.
This function will run Simon's algorithm on f. `naive=True` will use an naive solver to find s. False will use matrix null_space method (which is more efficient); naive defaults to False. This function will print out the non-0 value for s.

Currently there are 3 calls to run_Simon() present at the end of the script.
You can add your call after these or simply delete them.

**Grover**

**`grover.py`** is the script used run the Grover algorithm on a given function. To use this script, you will need to define the integers n and k. The variable n is used to represent the number of input bits. The variable k is used to represent the value of x where f(x)=1. It is assumed that in all other cases of f, f(x)=0.

Simply pass these values into **`run_grover(n, k)`** and it will create the circuit, compile the program, and run it on a qvm. The function takes the following extra parameters: `numshots`, `sim_wave`, `print_p`, and `use_aspen`. `numshots` governs how many times the simulation will run. `sim_wave`, when `True`, will provide the probabilities from the Wavefunction Simulator instead of running the simulation using a qvm. `print_p`, when `True`, will print the Quil code for debugging. `use_aspen`, when `True`, will set the qvm to use the Aspen QPC instead of the default QPC.

The result and execution time is returned by this function. You may add your calls to **`run_grover()`** at the end of the `main()` function.

## III. PyQuil Reflections

**Three aspects of programming in PyQuil that turned out to be easy to learn**

1. Adding gates to a PyQuil circuit only requires arithmetic addition. This made it very easy to add new columns of gates.
2. PyQuil was a Python library which meant that it was easy to integrate with other existing Python libraries. This was good for quick implementation and post-processing.
3. Debugging was very straightforward, because PyQuil lets us print the program and simulate the circuit prior to running.

**Three aspects of programming in PyQuil that turns out to be difficult to learn**

1. Simulating circuits using the qvm sometimes gives different results than expected. This is a very rare occurrence, however sometimes when running the same circuit with a known output we get an unexpected result. We are unsure why this occurs.
2. There are multiple ways of running experiments. It was difficult to learn of the different ways because they were spread throughout the documentation.
3. Qubit ordering is already a easy source of bugs, and it was not made easier here. Different QPU topologies has a different set of legal qubits indices that can be used, and it is on the user to implement these indices correctly.

**Three aspects of programming in PyQuil that the language supports strongly**

1. By using `with local_qvm()` to start the necessary server components, the programmer doesn't have to do anything but worry about running their program.
2. The python `print` feature gives reasonable output for program instances.
3. Gate creation supports python list compression, which is very handy when adding many of the same gate to a circuit.

**Three aspects of programming in PyQuil that the language supports poorly**

1. Little support for more complicated gates like those found in the Quirk circuit simulator. Of course, these can be composed of the simpler gates they do present.
2. Large unitary matrices are decomposed in the same way every time. If a large unitary matrix is known to exhibit specific properties, then it may speed up compilation time to use a decomposer that takes advantage of these properties.
3. A secondary print method which displays a circuit diagram of what the quil code would build would be beneficial for debugging purposes.

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

PyQuil presents a system when you build your circuit from left to right. It may be useful to support a way to add gates in between ones that have already been added to the program. Of course, this could be solved by using a python list to store the layers of gates prior to inserting them into the program. But native support for this may add more flexibility to programming in PyQuil.

**List three positives of the documentation of PyQuil**

1. It includes a getting-started page.
2. It is direct and to-the-point.
3. It includes an advanced-usage page.

**List three negatives of the documentation of PyQuil**

1. Specific sections do not reference each other when they should. There are multiple ways of running simulations and taking measurements, however they appear disconnected in the documentation.
2. More examples should be inserted to demonstrate the functionality of each definition in PyQuil (much like the numpy docs).
3. Existing examples should be explained a little more clearly, with some references to other sources if the readers to not understand the core concepts of the example.

**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**

| **PyQuil jargon** | **QP jargon** |
|---|---|
| Program | Circuit |
| Quantum Virtual Machine | Quantum Simulator |
| Quantum Abstract Machine | Quantum/Classical Hybrid Computer |
| Device | Quantum Computer/Simulator |
| Shots | Runs on Quantum Computer/Simulator |

**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?**
PyQuil does type checking both during compilation and with `run()`. The compilation stage will check that the object to be compiled is a valid `Program()`, and it will also check that the qubits used are legal with the given topology.

The run() method will check that the object passed in is a `PyQuilExecutableResponse` object. It will not run with an uncompiled `Program()`.

PyQuil will also always return a list of lists (with 0/1) when returning from a run. I'm assuming it is doing some checking behind the scenes before it returns classical values.