COM SCI 239

Quantum Programming: Algorithms in PyQuil

May 12, 2020

Author

Arjun Raghavan and Bryan Pan

1 Overview

2 Design

2.1 Mathematical explanation

All four algorithms take as input a function f in the form:

$$f : \{0,1\}^n \to \{0,1\}^m$$

In the case of Deutsch-Josza, Bernstein-Vazirani, and Grover, we have $m = 1$. Simon, on the other hand, has $m = n$.

For Deutsch-Josza, Bernstein-Vazirani, and Simon, the quantum oracle of f, denoted U_f, is defined as:

$$U_f \ket{x} \ket{b} = \ket{x} \ket{b \oplus f(x)}$$

Here, we have $b \in \{0,1\}^m$. The $\oplus$ operator represents bitwise XOR, or equivalently bitwise addition mod 2.

Grover's algorithm makes use of a gate denoted by Z_f, which is defined as:

$$Z_f \ket{x} = (-1)^{f(x)} \ket{x}$$

Notice that this is precisely the application of the phase kickback trick for gates of a form equivalent to U_f where $\ket{b} = \ket{-}$:

$$Z_f \ket{x} \ket{-} = (-1)^{f(x)} \ket{x} \ket{-}$$

Clearly, we can also define Z_f as a quantum oracle for Grover's algorithm in the same way U_f was defined for the three other algorithms. Given that all four algorithms' quantum oracles have the same general form, we created a single function which could generate a quantum oracle and reused it for each program.

Lemma 1. $\sum_{q \in \{0,1\}^k} \ket{q} \bra{q} = I_{2^k}$ *where* I_{2^k} *is the identity matrix in* $\mathcal{H}_{2^k}$ *(corresponding to k qubits).*

Proof. Let $q \in \{0,1\}^k$. We have $\ket{q} = \begin{bmatrix} q_1 & q_2 & \dots & q_{2^k} \end{bmatrix}^T$ where $q_i = 1$ for some $1 \le i \le 2^k$ and $q_j = 0$ for all $j \ne i$. Then, $\ket{q} \bra{q}$ is a $2^k \times 2^k$ matrix of the form $\begin{bmatrix} q_{ab} \end{bmatrix}$ where:

$$q_{ab} = \begin{cases} 1 & \text{if } a = b = i \\ 0 & \text{otherwise} \end{cases}$$

For $\alpha, \beta \in \{0,1\}^k$ such that $\alpha \ne \beta$, we have $\ket{\alpha} \ne \ket{\beta} \implies \ket{\alpha} \bra{\alpha} \ne \ket{\beta} \bra{\beta}$. Thus:

$$\sum_{q \in \{0,1\}^k} \ket{q} \bra{q} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix} + \dots + \begin{bmatrix} 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} = I_{2^k}$$

$\square$

Consider the application of U_f to the outer product of $|x\rangle |b\rangle$ with itself:

$$U_f |x\rangle |b\rangle \langle x| \langle b| = |x\rangle |b \oplus f(x)\rangle \langle x| \langle b|$$
$$= |x\rangle \langle x| |b \oplus f(x)\rangle \langle b|$$

Let $k = n + m$. By taking the sum of this over all bit strings $xb \in \{0,1\}^k$, we get:

$$\sum_{xb \in \{0,1\}^k} U_f \left(|x\rangle |b\rangle \langle x| \langle b|\right) = U_f \left(\sum_{xb \in \{0,1\}^k} |x\rangle |b\rangle \langle x| \langle b| \right)$$
$$= U_f \circ I_{2^k}$$
$$= U_f$$

And so:

$$U_f = \sum_{xb \in \{0,1\}^k} |x\rangle \langle x| |b \oplus f(x)\rangle \langle b| \tag{1}$$

2.2 Python implementation

As shown in Code Block 1, Equation 1 is straightforwardly implemented in Python thanks to the numpy library using methods such as `np.kron` (the Kronecker product is a special case of the tensor product over complex matrices and is the nomenclature used by `numpy`) and `np.outer` [1]. Notice in line 155 that `f` is a dictionary, not a function. Indeed, we chose to treat the input function f for all four programs as a dictionary (or a "mapping" as we mostly referred to it) for a few reasons.

For one, this would allow a user more flexibility in providing f as an input to our programs without having to possibly construct tedious `if-elif` statements. In addition, this allowed us to more easily test our programs—we could randomly generate dictionaries representing functions which followed whatever assumptions were required for each algorithm. An example of this for Deutsch-Josza can be seen in Code Block 2.

Our implementation of generating U_f with this generalized method had the additional benefit of making it very easy to parametrize our solutions in n. The `oracle.gen_matrix` method, along with any other methods which require the dimension of the domain and/or range of f, accepts `n` and `m` (as defined previously) as arguments. While `m` is algorithm-dependent and so assigned inside the code, the user can pass in a value for n using the `-num` option[1].

2.2.1 Readability

There were a couple of possible approaches we could have taken to iterating over all $xb \in \{0,1\}^{n+m}$, including nested `for` loops or generating the entire set of bit strings and iterating

---

[1]Discussed in further detail in the README.

```
148      # Accumulator to hold the value of the summation
149      U_f=np.zeros((2**(n+m),2**(n+m)))
150      for xb in range(0, int(2**(n+m))):
151          # Convert index to binary and split it down the middle
152          x  = f'{xb:0{n+m}b}'[:int(n)]
153          b  = f'{xb:0{n+m}b}'[int(n):]
154          # Apply f to x
155          fx = f[x]
156          # Calculate b + f(x)
157          bfx = f'{int(b, 2) ^ int(fx, 2):0{n}b}'
158
159          # Vector representations of x, b, and b+f(x)
160          xv = np.zeros((2**n,1))
161          xv[int(x, 2)] = 1.
162          bv = np.zeros((2**m,1))
163          bv[int(b, 2)] = 1.
164          bfxv = np.zeros((2**m,1))
165          bfxv[int(bfx, 2)] = 1.
166
167          # Accumulate (|x><x| (*) |b + f(x)><b|) into the sum
168          # (*) is the tensor product
169          U_f = np.add(np.kron(np.outer(xv, xv), np.outer(bfxv, bv)), U_f)
```

Code Block 1: Excerpt from the `gen_matrix` function in `oracle.py` showing the implementation of Equation 1 using `numpy`

over that. We eventually settled on the approach seen in Code Block 1, which we felt was more elegant.

The dimension of $\{0,1\}^{n+m} =: S$ is 2^{n+m}. Furthermore, each element of S is a bitstring which has a decimal equivalent. Thus, it would be just as effective to iterate through each of these decimal equivalents, convert them to binary, and extract x and b, which is precisely what we did. We felt that this was more concise than generating the entirety of S or nesting `for` loops while still maintaining some clarity of our approach, assisting in readability.

In more general terms, we made sure to thoroughly comment our code wherever we felt it was necessary. Our primary functions all have PyDocs (loosely conforming to PEP/8 guidelines [2]), and further comments are added when specific lines of code require further clarification. Overall, though, our code is self-documenting as much as possible. We have also taken care to maintain limited line lengths (keeping 80 characters as a soft limit) and proper indentation (although the Python language just about forces the latter).

3

```
43      if algo is Algos.DJ:
44          # Constant: f(x) returns 0 or 1 for all x
45          if func is DJ.CONSTANT:
46              val = np.random.choice(['0','1'])
47              oracle_map = {i: val for i in qubits}
48          # Balanced: f(x) returns 0 or 1 for all x
49          # val1 represents set of x that f(x) = 1
50          # val0 represents set of x that f(x) = 0
51          elif func is DJ.BALANCED:
52              val1 = random.sample(qubits, k=int(len(qubits)/2))
53              val0 = set(qubits) - set(val1)
54              oracle_map = {i: '1' for i in val1}
55              temp = {i: '0' for i in val0}
56              oracle_map.update(temp)
```

Code Block 2: Excerpt from the `init_bit_mapping` function in `oracle.py` showing how we randomly generate a function for testing Deutsch-Josza.

3 Evaluation

3.1 Shared code

As opposed to copying identical sections of code between each of our four programs, we opted to have a central module, `oracle.py`, which contains common code relating to the generation of functions f (bit mappings, to be precise) and quantum oracles, which was discussed in the previous section.

Of course, there is still some duplication of code between programs. Each program has a `getUf` function (named `getZf` for Grover) which checks if a U_f matrix for the desired value of n has already been generated and stored[2], in which case it loads it from its file; otherwise, it simply generates a new U_f. All the implementations of this function are identical.

Also, our programs use the `argparse` library [3] to read and process user-defined options[3]. The code for doing so is largely the same across programs, save for some minor algorithm-specific differences (like the names).

Overall, we estimate that each program file has an average of approximately 38 lines of code that is shared (within a reasonable threshold of a few characters' difference) with the other files. Excluding `oracle.py`, the average length of a single program's Python file is 122 lines. As such, the percentage of shared code between programs is approximately 31%.

There are certainly avenues for reducing code reuse. We began to adopt a `class` structure for the entire project, having each algorithm inherit from an `Algo` class which itself had methods common to all four programs. This can in fact be seen in the code for `dj.py` and `bv.py`.

---

[2]This generation and storing of matrices ahead of actually running the algorithms is discussed in the README.

[3]Also discussed in the README.

However, this was abandoned mostly due to the fact that the assignment required individual files for each program, and so this would end up unnecessarily increasing the amount of code reused between files.

3.2   Testing

As mentioned earlier, our implementation of generating and storing quantum oracle gate matrices and usage of `argparse` to handle user input greatly facilitated testing. We quickly realized that compiling and running our code was (naturally) CPU-intensive, so we also used a couple of Oracle Cloud Infrastructure Compute instances so that we could run our code while freeing up our own computers for further work.

For our implementation of Grover, we configured the program such that it would automatically calculate the minimum number of trials required to minimize error based on the size of the input using the equation $k = \lfloor \pi \sqrt{N}/4 \rfloor$ with $N = 2^n$. For our implementation of Simon, given that a full "iteration" was $n - 1$ applications of the quantum oracle (to generate $n - 1$ values for y), we multiplied the user's input number of trials by 4 to minimize error using the equation $\mathcal{P}(\text{not linearly independent}) = \exp(-t/4)$ with t being the number of trials.

We tested our code with a variety of values of n (the length of the input bit string). For Simon and Grover, we also tested with differing values of t (the number of trials); this was unnecessary for Deutsch-Josza and Bernstein-Vazirani, which are deterministic algorithms. We found that our results very closely matched the expected outcomes of applying our algorithms.

For each value of n up to about 7-9 depending on the algorithm, we generated randomized functions (following the appropriate constraints for its corresponding algorithm) and subsequently U_f matrices, storing them locally for later use. We limited n to 7-9 mostly because any values greater than those resulted in massive U_f matrices (the file size of a 10-qubit U_f matrix stored as an `.npy` array exceeded 100 MB). This proved to be a reasonable restriction, because none of our programs were able to operate on more than 7 qubits before the PyQuil compiler crashed or the program itself exceeded its allocated heap space.

For smaller n, however, we were able to make some observations:

- Deutsch-Josza was the fastest algorithm of the four, able to reach 7 qubits before crashing.

- Whether the input function to Deutsch-Josza was balanced or constant was a large factor in its compilation and execution time. For constant functions, the quantum oracle was simply a large identity matrix, and it appears that PyQuil optimizes for such a case, as can be seen below.

- Despite being probabilistic, Simon's and Grover's algorithms are in fact quite accurate even for a smaller number of trials. Of course, this is in an ideal setting on a Quantum Virtual Machine without noise.

- Compilation time is orders of magnitude larger than execution time. All of the programs' execution time were under 10 seconds; compilation time, on the other hand, increased dramatically as n increased—for Simon's algorithm, an input of $n = 4$ had a compilation time of a whopping 36 minutes. It would have been beneficial to compile ahead of time,

but this was not feasible for two reasons: a new gate had to be used for each value of n, meaning the quantum circuit would have to be recompiled for each input regardless, and it was not clear whether we could compile a quantum program and store it as a file.

- Simon's algorithm is the slowest to compile, crashing `quilc` at only 5 input qubits. This is expected behavior; while the other three algorithms have one additional helper qubit, Simon's algorithm must use n additional qubits. Thus, increasing the input bitstring by 1 causes the resulting quantum oracle matrix to balloon to $2^4 = 16$ times its original size.

3.3   Scalability

While as mentioned above we used Oracle Cloud Infrastructure, we were limited to 2 CPU cores on each of our instances. So, one of our local machines was used to perform tests for execution time. Said machine has an `8-core Intel(R) Core(TM) i7-8550U @ 1.80GHz` and a RAM of `16GiB System Memory` (x2 `8GiB SODIMM DDR4 Synchronous Unbuffered 2400 MHz`).

We measured both the compilation as well as execution time of each run for the four algorithms. The execution time was obtained from the logged output of `qvm -S`. The compilation time was measured simply by measuring the system clock before and after calling `QuantumComputer.run_and_measure`[4].
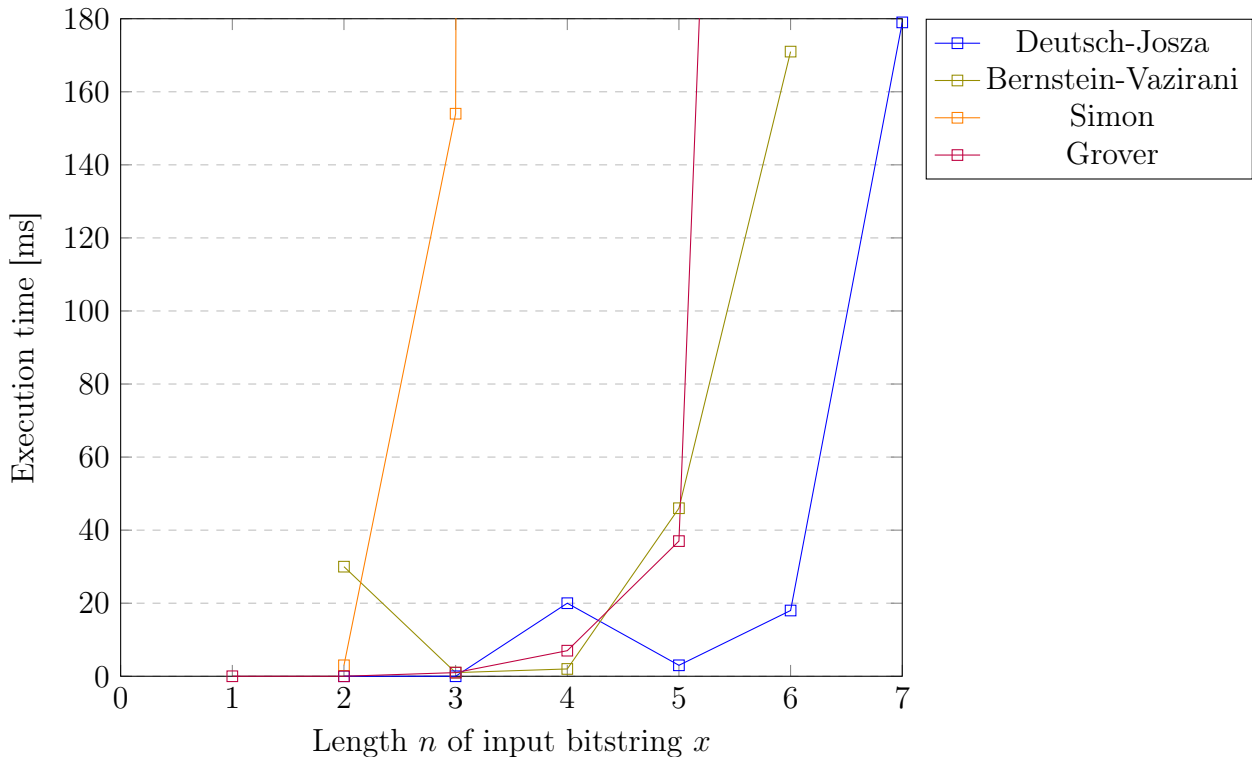

Figure 1: Graph of execution time (ms) vs. the length of the input bitstring.

---

[4]A more accurate measure of this would be to subtract execution time from the total measured time to "run and measure", but as mentioned before, compilation time vastly dwarfs execution time.

Figure 1 portrays execution time against n. As expected, execution time seems to increase exponentially as the size of the input increases. Note the the value of Simon's execution time for $n = 4$ is off the graph as an outlier, as it had a value of $10,015$ m.

There is an apparent discrepancy in the plot for Deutsch-Josza. This is due to the nature of our tests, which generated randomized input functions f. In the case of $n = 5$ and $n = 6$, the generated function was constant, not balanced. We chose to include these points as opposed to running the tests again on a homogenous input function set so as to demonstrate the impact having a constant function versus a balanced function had on execution time. It is likely that PyQuil (or the QVM itself) is performing some sort of optimization when using identity matrices.

Overall, as expected, runtime increases exponentially with n, both in terms of execution and compilation.
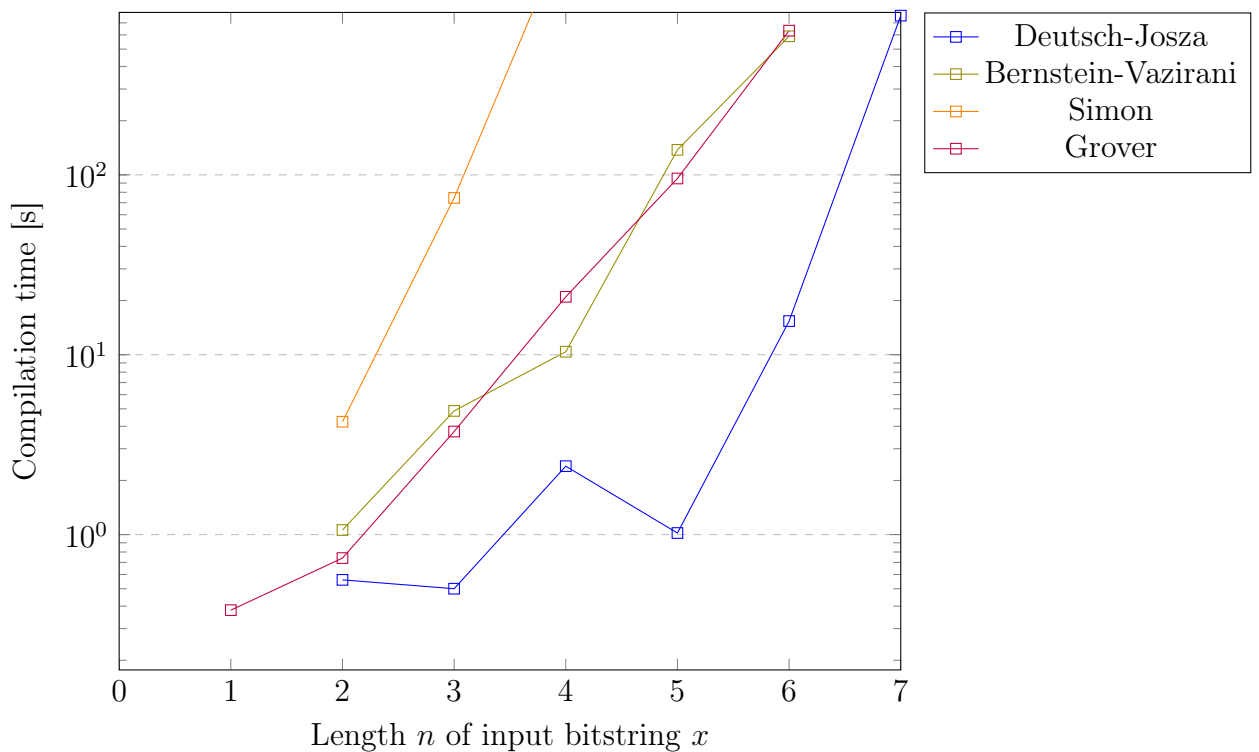

Figure 2: Graph of compilation time (s) vs. the length of the input bitstring. Notice that the y-axis is a logarithmic scale; this is because the time taken to compile grows exponentially as n increases.

Figure 2 portrays compilation time against n. Notice two significant differences from Figure 1. For one, the units of the y-axis are seconds, not milliseconds. Also, the scale is logarithmic, not linear—this is because the growth rate too large to be accurately displayed on a linear scale. It is clear that growth here is once again exponential, as the lines for each algorithm are almost straight. Deutsch-Josza is an outlier in this regard, but it is once again likely because for $n = 4$ and $n = 5$ a constant function was generated.

4   Reflection on PyQuil

4.1   Learning to use PyQuil

Our brief introduction to PyQuil showed us that there were a handful of gates which were provided as part of the library. A such, we expected that it would be challenging to create our own gates. Fortunately, we discovered that it was in fact very simple. We had to generate our own matrix, which was straightforward with the help of `numpy`, but the creation of an actual quantum gate was quite literally only two lines (see Code Block 3).

```
57      U_f_def = DefGate('U_f', getUf(n, reload))
58      U_f = U_f_def.get_constructor()
```

Code Block 3: The two lines needed to create a quantum gate that can be used in a PyQuil program. Excerpt from `simon.py`

In addition, building the program was actually quite easy and intuitive. It seemed quite natural to use simple concatenation operators as `p+=<gate>` to build our program. The ability to "run and measure" was also very helpful, because it entirely abstracted the compilation process from quil → native quil → binary executable → result from the user. For people who are more interested in functionality and output, it is extremely useful to have that process abstracted but also readily available to use. The `quilc` and `qvm` server logs were verbose resources for debugging, whether it be a memory issue or a logical issue in our coding. Indeed, the act of programming these circuits was actually trivial in comparison to debugging and understanding the underlying operation of the simulator.

We of course had some gripes with PyQuil. As mentioned previously, the ability to pre-compile executables or amortize compilation time a lot quicker is something that PyQuil lacks support for. Also, we ran into a lot of problems with the allocated heap for `qvm` running out of space. Some way to mitigate the sheer amount of space used or to understand how to combat it technically would be useful. Finally, it would be nice to better methods of visualizing our quantum circuits. Especially while using our custom gates, we had to resort to a rather unusual notation to expand a tuple into function arguments (which can be seen in our submitted code, as seen in Code Block 4.

```
66      p += U_f(*(tuple(range(2*n))))
```

Code Block 4: Passing a variable number of inputs to a custom quantum gate. Excerpt from `simon.py`

We do have the option of doing `print(to_latex(p))` to generate a Ti*k*z diagram, but it is a cumbersome process to copy the output into a file and compile it into a PDF or PNG when we would much prefer to be able to instantly view the state of our circuit (not unlike how Google's Cirq does it [4]).

4.2   A suggestion

Similar to how a C or C++ program can be compiled by something like `gcc`, it would be extremely beneficial if PyQuil could support externally saving compiled quantum programs. Obviously, it would be somewhat impractical as to the volatility of user input. For example, a compiled program or `PyQuilExecutableResponse` for a `3q-qvm` would not work for a 5 qubit system. Still, a way to reduce programming downtime would be at least to amortize compilation.

We also noticed that one of the biggest challenges in terms of time was the conversion of pyquil program to native quil or (shown in the logs as `quil_to_native_quil`). While PyQuil supports a handful of gates including the Hadamard gate, controlled NOT, and more, Rigetti QPUs (and by extension the QVM we would use) have gate operators constrained to lie in $RZ(\theta)$, $RX(k\pi/2)$, and CZ—in other words, in the Z basis. With simple quantum oracle matrices like identity matrices, the compiler did not have to worry about calculating the correct rotation matrices. However, for most other gates, especially unoptimized ones like our generated U_f gates, the compiler would spend a lot of time transforming them into a combination of the three above gates.

Unfortunately, this always needs to be executed after every step, and the PyquilExecutableResponse is sometimes inspectable. For security and for compilation purposes, it might be better to make the compiled object to be a BinaryExecutableResponse and opaque.

4.3   Reading the documentation

The documentation for PyQuil provides a fine introduction to PyQuil and quantum programming. The **Getting Started** page clearly and concisely explains how to build a quantum circuit, make a custom gate, and more. Also, the documentation provides a lot of sample code for everything it explains, which was especially useful for the both of us, given that we are both not overly experienced with Python[5]. We discovered an **Exercises** section in the documentation which we want to explore further; it provides a set of problems and solutions for implementing quantum algorithms. (We chose not to look at this in more detail during this assignment because one of these problems was in fact Grover's algorithm! We had rather figure that out ourselves.)

However, explaining more intermediate or advanced usage and concepts is where the documentation fell short, in our opinion. We found it difficult to understand things like what exactly `run_and_measure` does under all those layers of abstraction, or how to manually compile and store a PyQuil program. A specific issue we faced was what kind of QVM to choose when running our programs. It seemed there were only a couple of options, such as `9q-qvm` or `9q-square-qvm`. To use a larger number of qubits seemed impossible, until we discovered that the number of qubits in these directives could was in fact variable—we could just as easily use `3q-qvm` or `14q-qvm`. This was not very clearly stated in the documentation, relegated to a single statement under a block of code. However, the documentation seemed to lack information on the implication of choosing arbitrary values like this. It was entirely possible that there are

---

[5]Our fortes lie more in the field of statically-typed languages.

certain numbers of qubits on a virtual QPU that more closely mimic a real QPU, but we were not aware of this.

4.4   Translating key concepts

As is our understanding, below are some key concepts in quantum programming as they are referred to with PyQuil:

- **quil**: pyquil program $(H(0)\ H(1)\ \dots\ H(n))$

- **native quil**: modified quil program that is dependent on the ISA of the compiler, built of all native rotations matrices

- **PyQuilExecutableResponse:** Executable for a QVM or QPU

- **RX gate:** Universal gate for native quil, rotation gate for angle $= +/-\ /2$

- **RZ gate:** Universal gate for native quil, rotation gate for an arbitrary angle

- **CZ gate:** Universal gate for native quil, Controlled-Z gate for neighboring qubits

- **Measure:** Measure instruction to measure qubit state into a classical register

4.5   Joining PyQuil with classical code

Our approach to our PyQuil implementation was to make the most robust program possible to show the true power of quantum computing. We reasoned that user input will most likely lead to undefined behavior and decided to mitigate that by randomizing function parameters. While doing so, we made it easy to substitute our function parameters with whatever desired parameters of your choosing, but for the purposes of showing the prowess of quantum computing, we reasoned it would be best to abstract that classical input.

For example, Deutsch-Josza gave users the ability to choose if they wanted a balanced or constant function as opposed to putting in their own function. If we allowed users to put in their own function, we would first have to check whether the function is balanced or constant classically to determine if it was a valid function to pass into our algorithm. This would entirely defeat the purpose of running the algorithm, hence the utilization of entirely random balanced/constant U_f matrices.

References

[1] S. van der Walt, S. C. Colbert, and G. Varoquaux. The NumPy array: A structure for efficient numerical computation. Computing in Science & Engineering, 13(2):22–30, 2011.

[2] Guido van Rossum, Barry Warsaw, and Nick Coghlan. Style guide for Python code. PEP 8, 2001.

[3] Guido Van Rossum and Fred L. Drake. Python 3 Reference Manual. CreateSpace, Scotts Valley, CA, 2009.

[4] The Cirq Developers. Cirq Documentation, 2020.