COM SCI 239

Quantum Programming: Algorithms in Qiskit

May 29, 2020

Author
Arjun Raghavan and Bryan Pan

1 Design **1**
1.1 Mathematical explanation . . . 1
1.2 Python implementation . . . 2
1.2.1 Readability . . . 3

2 Evaluation **4**
2.1 Shared code . . . 4
2.2 Testing . . . 5
2.3 Scalability . . . 6

3 Reflection on PyQuil **7**
3.1 Learning to use PyQuil . . . 7
3.2 A suggestion . . . 8
3.3 Reading the documentation . . . 8
3.4 Translating key concepts . . . 9
3.5 Joining PyQuil with classical code . . . 9

1 Design

1.1 Mathematical explanation[1]

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:

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

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:

$$Z_f \left|x\right\rangle = (-1)^{f(x)} \left|x\right\rangle$$

Notice that this is precisely the application of the phase kickback trick for gates of a form equivalent to U_f where $\left|b\right\rangle = \left|-\right\rangle$:

$$Z_f \left|x\right\rangle \left|-\right\rangle = (-1)^{f(x)} \left|x\right\rangle \left|-\right\rangle$$

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. *$\sum_{q\in\{0,1\}^k} \left|q\right\rangle \left\langle q\right| = I_{2^k}$ 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 $\left|q\right\rangle = \begin{bmatrix} q_1 & q_2 & \dots & q_{2^k} \end{bmatrix}^T$ where $q_i = 1$ for some $1 \leq i \leq 2^k$ and $q_j = 0$ for all $j \neq i$. Then, $\left|q\right\rangle \left\langle q\right|$ 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 $\alpha \neq \beta$, we have $\left|\alpha\right\rangle \neq \left|\beta\right\rangle \implies \left|\alpha\right\rangle \left\langle\alpha\right| \neq \left|\beta\right\rangle \left\langle\beta\right|$. Thus:

$$\sum_{q\in\{0,1\}^k} \left|q\right\rangle \left\langle q\right| = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix} + \dots + \begin{bmatrix} 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} = I_{2^k}$$

□

[1]**NB:** This section is identical to Section 1.1 in the *Algorithms in PyQuil* report. Regardless, it is repeated here for the sake of completeness.

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

$$\begin{aligned} U_f\,|x\rangle\,|b\rangle\,\langle x|\,\langle b| &= |x\rangle\,|b\oplus f(x)\rangle\,\langle x|\,\langle b| \\ &= |x\rangle\,\langle x|\,|b\oplus f(x)\rangle\,\langle b| \end{aligned}$$

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

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

And so:

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

1.2 Python implementation

In our PyQuil assignment, we noted that we had to use the `numpy` library in order to implement Equation 1 in Python, as PyQuil does not have native methods of creating custom gates from scratch. Qiskit, on the other hand, possesses its own way to define and manipulate custom gates on the matrix level: the `Operator` class [Qiskit, 2020a], as seen in Code Block 1.

We chose to make this change for a few reasons. For one, using the `Operator` class would allow for easier interoperability with actual Qiskit QuantumCircuits [Qiskit, 2020c], and while we could just as easily use `numpy` to calculate U_f and then convert it to an `Operator` just before appending it to a circuit, we felt that using the `Operator` class for said calculation would be more idiomatic Qiskit. Also, we found that interestingly, the `Operator` class was capable of performing the necessary tensor products rather faster than `numpy`. Specifically, the time taken to generate U_f using `Operator` methods was about 0.847% faster than doing so with `numpy` (the full set of readings is omitted for brevity).

Apart from this minor change, our implementation of generating U_f is the same it was with PyQuil. That is, as seen in Code Block 1, we still accept a function definition `f` in the form of a Python dictionary, not a physical function, for the same reasons as discussed in our PyQuil report (it allows more flexibility of input, and it makes it far easier to parametrize our solutions in `n`).

Also, the `oracle.gen_matrix` method, along with any other methods which require the dimension of the domain and/or range of f, still 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[2].

[2]Discussed in further detail in the README.

```
    for xb in range(0, int(2**(n+m))):
        # Convert index to binary and split it down the middle
        x  = f'{xb:0{n+m}b}'[:int(n)]
        b  = f'{xb:0{n+m}b}'[int(n):]
        # Apply f to x
        fx = f[x]
        # Calculate b + f(x)
        bfx = f'{int(b, 2) ^ int(fx, 2):0{n}b}'

        # Vector representations of x, b, and b+f(x)
        xv = Operator(np.zeros((2**n,1)))
        xv.data[int(x, 2)] = 1.
        bv = Operator(np.zeros((2**m,1)))
        bv.data[int(b, 2)] = 1.
        bfxv = Operator(np.zeros((2**m,1)))
        bfxv.data[int(bfx, 2)] = 1.

        # Accumulate (|x><x| (*) |b + f(x)><b|) into the sum
        # (*) is the tensor product
        U_f += (xv * xv.transpose()).tensor(bfxv * bv.transpose())

    return U_f
```

Code Block 1: Excerpt from the `gen_matrix` function in `oracle.py` showing the implementation of Equation 1 using Qiskit's `Operator` class.

1.2.1 Readability

In terms of the readability of our oracle functions such as the one seen in Code Block 1, there is not much we can say here that is different from what we described in our PyQuil report. Our reasons for implementing `gen_matrix` as seen above are the same—we felt it is more concise to iterate through the decimal equivalents of `x` and `b` and extract relevant bits on the fly—and we still comment our code with PyDocs, loosely conforming to PEP/8 guidelines [van Rossum et al., 2001].

In terms of the readability of our actual implementations of the different algorithms, we found that Qiskit's way of building quantum circuits with custom gates was in some ways more intuitive and cleaner than that of PyQuil. Compare the following lines of code:

```
    # p is a PyQuil Program, u_f is a NumPy matrix, n is the number of qubits
    U_f_def = DefGate('U_f', u_f)
    U_f = U_f_def.get_constructor()
    p += U_f_def
    p += U_f(*(tuple(range(n))))
```

```
# circuit is a Qiskit QuantumCircuit, u_f is a NumPy matrix, n is the
number of qubits
U_f = Operator(u_f)
circuit.append(U_f, list(range(n))[::-1])
```

The equivalent Qiskit implementation is much less verbose, without sacrificing clarity. By accepting a list of qubit indices instead of requiring them to be passed as arguments, it is far easier to parametrize a gate in this manner.

However, note the addition of `[::-1]` to the Qiskit code. One perhaps unintuitive aspect of Qiskit's multi-qubit gates is that qubits are ordered in reverse, where qubit 0 is the least significant qubit, and qubit n is the most significant qubit. As such, the way we generate matrices for each U_f require that we pass in a reversed list of qubits; similarly, for multi-qubit outputs, we must again reverse the output.

2 Evaluation

2.1 Shared code

Just as with PyQuil, 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[3], 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 [Van Rossum and Drake, 2009] to read and process user-defined options[4]. The code for doing so is largely the same across programs, save for some minor algorithm-specific differences (like the names). Furthermore, we included code that measures each program's execution time as well.

Overall, we estimate that each program file has an average of approximately 33 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 167 lines (not including . As such, the percentage of shared code between programs is approximately 20%.

Notice that this number is lower than the 31% code reuse percentage quoted for PyQuil. We attribute this drop to the fact that, overall, Qiskit is less verbose than PyQuil, as well as the

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

[4]Also discussed in the README.

fact that we added some more wrapping code in order to measure execution time provide more verbose output if desired.

It is possible for us to minimize code reuse. A lot of the aforementioned code to measure execution time is largely identical between programs, save for small differences. With more time, perhaps this could be removed and placed into a separate utility Python module.

2.2 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. As such, there was no compilation time to speak of, and execution was incredibly fast. Instead, we simply used Qiskit's Aer simulator as the backend for running our programs.

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 5-9 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 5-9 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), or otherwise took too much time to generate such a U_f.

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

- Deutsch-Josza was the fastest algorithm of the four, able to reach 9 qubits before crashing.
- Whether the input function to Deutsch-Josza was balanced or constant did not seem to signficantly affect execution time.
- 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 simulator without noise.
- For PyQuil, we found that compiling our code was incredibly CPU-intensive, taking many minutes to simply compile our quantum circuit into the QUIL assembly language. On the other hand, Qiskit needed to do no such thing, perhaps due to the fact that Qiskit's fundamental gate set is far larger than PyQuil—PyQuil at its core uses the gate operators $R_Z(\theta)$, $R_X(\frac{k\pi}{2})$, CZ [Rigetti, 2020], while Qiskit uses the universal gate set U_1, U_2, U_3, CX and I [Qiskit, 2020b], allowing for much more flexibility in implementing other gates [Zhang et al., 2018].

- We were only able to test Simon's algorithm with up to 5 input qubits; beyond that, the time taken to generate U_f was unreasonably long. 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.

2.3 Scalability

As with PyQuil, we tested our code on a machine with 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`).

As mentioned earlier, there was no compilation time to speak of as there was with PyQuil, and we Thus, we were able to measure execution time simply by using Python's built-in `time` module, measuring the time taken to call our functions (from the creation of a circuit to the obtaining of a `counts` object).

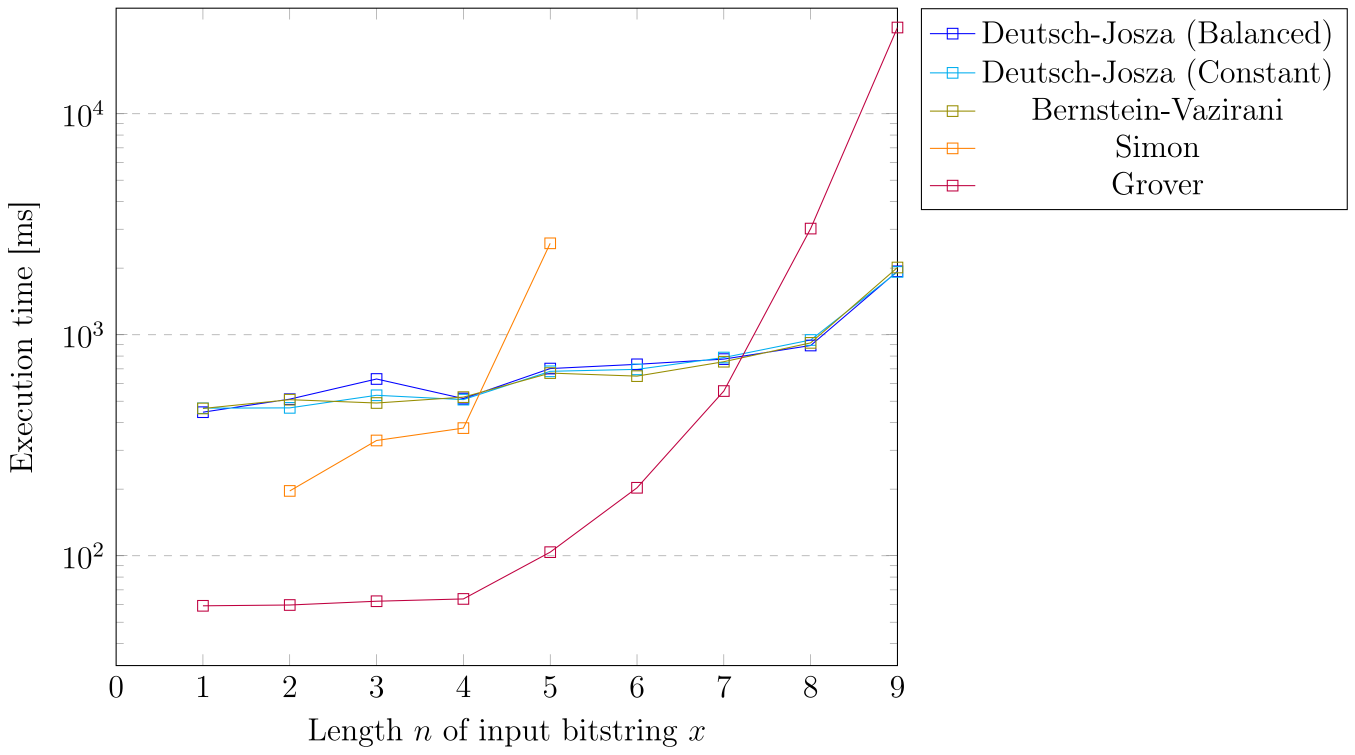

Figure 1: Graph of execution time (ms) vs. the length of the input bitstring. Note that the y-axis is logarithmically scaled.

Figure 1 portrays execution time against n. The execution time growths of Deutsch-Josza and Bernstein-Vazirani are both very similar, with a soft exponentional growth (shown on a logarithmic graph as apparently linear). Simon's algorithm appears to grow exponentially, at a much, much faster rate than DJ and BV.

Note the execution time growth for Grover's algorithm, on the other hand, which has the shape of an exponential curve despite being plotted on a logarithmic scale. Although enough data could not be collected for Simon's algorithm, it is expected that a similar growth would be observed for it as well. This is expected behavior, and appears for a specific reason: the number of times Simon's and Grover's algorithm are to be run is directly dependent on the number of input qubits. In the case of Simon's algorithm, to minimize error, the entire circuit must be executed $(n-1) \times 4m$ times, where m is a user-defined parameter (chosen to be equal to 5 for our tests). For Grover's algorithm, the entire circuit is to be called once, but the Grover rotation gate is appended to the circuit $\lfloor \frac{\pi}{4}\sqrt{2^n} \rfloor$ times. As such, it is natural that the execution time for each of these algorithms increases at a double-exponential rate. On a log-log graph, the plots for these two algorithms' growth rates would indeed appear in a linear form.

3 Reflection on PyQuil

3.1 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 2).

```

    return U_f, s
```

Code Block 2: 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 $\rightarrow$ native quil $\rightarrow$ binary executable $\rightarrow$ 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 3.

```
    ys.append(zero)
```

Code Block 3: 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 [The Cirq Developers, 2020]).

3.2 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.

3.3 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.)

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

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

3.4 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 ($H(0)\ H(1)\ \ldots\ H(n)$).
- **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 $= \pm\pi/2$.
- **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.

3.5 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

[Abraham et al., 2019] Abraham, H., AduOffei, Akhalwaya, I. Y., Aleksandrowicz, G., Alexander, T., Alexandrowics, G., Arbel, E., Asfaw, A., Azaustre, C., AzizNgoueya, Barkoutsos, P., Barron, G., Bello, L., Ben-Haim, Y., Bevenius, D., Bishop, L. S., Bolos, S., Bosch, S., Bravyi, S., Bucher, D., Burov, A., Cabrera, F., Calpin, P., Capelluto, L., Carballo, J., Carrascal, G., Chen, A., Chen, C.-F., Chen, R., Chow, J. M., Claus, C., Clauss, C., Cross, A. J., Cross, A. W., Cross, S., Cruz-Benito, J., Culver, C., Córcoles-Gonzales, A. D., Dague, S., Dandachi, T. E., Dartiailh, M., DavideFrr, Davila, A. R., Dekusar, A., Ding, D., Doi, J., Drechsler, E., Drew, Dumitrescu, E., Dumon, K., Duran, I., EL-Safty, K., Eastman, E., Eendebak, P., Egger, D., Everitt, M., Fernández, P. M., Ferrera, A. H., Frisch, A., Fuhrer, A., GEORGE, M., Gacon, J., Gadi, Gago, B. G., Gambella, C., Gambetta, J. M., Gammanpila, A., Garcia, L., Garion, S., Gilliam, A., Gomez-Mosquera, J., de la Puente González, S., Gorzinski, J., Gould, I., Greenberg, D., Grinko, D., Guan, W., Gunnels, J. A., Haglund, M., Haide, I., Hamamura, I., Havlicek, V., Hellmers, J., Herok, Ł., Hillmich, S., Horii, H., Howington, C., Hu, S., Hu, W., Imai, H., Imamichi, T., Ishizaki, K., Iten, R., Itoko, T., JamesSeaward, Javadi, A., Javadi-Abhari, A., Jessica, Johns, K., Kachmann, T., Kanazawa, N., Kang-Bae, Karazeev, A., Kassebaum, P., King, S., Knabberjoe, Kovyrshin, A., Krishnakumar, R., Krishnan, V., Krsulich, K., Kus, G., LaRose, R., Lambert, R., Latone, J., Lawrence, S., Liu, D., Liu, P., Maeng, Y., Malyshev, A., Marecek, J., Marques, M., Mathews, D., Matsuo, A., McClure, D. T., McGarry, C., McKay, D., McPherson, D., Meesala, S., Mevissen, M., Mezzacapo, A., Midha, R., Minev, Z., Mitchell, A., Moll, N., Mooring, M. D., Morales, R., Moran, N., MrF, Murali, P., Müggenburg, J., Nadlinger, D., Nakanishi, K., Nannicini, G., Nation, P., Navarro, E., Naveh, Y., Neagle, S. W., Neuweiler, P., Niroula, P., Norlen, H., O'Riordan, L. J., Ogunbayo, O., Ollitrault, P., Oud, S., Padilha, D., Paik, H., Perriello, S., Phan, A., Piro, F., Pistoia, M., Pozas-iKerstjens, A., Prutyanov, V., Puzzuoli, D., Pérez, J., Quintiii, Raymond, R., Redondo, R. M.-C., Reuter, M., Rice, J., Rodríguez, D. M., RohithKarur, Rossmannek, M., Ryu, M., SAPV, T., SamFerracin, Sandberg, M., Sargsyan, H., Sathaye, N., Schmitt, B., Schnabel, C., Schoenfeld, Z., Scholten, T. L., Schoute, E., Schwarm, J., Sertage, I. F., Setia, K., Shammah, N., Shi, Y., Silva, A., Simonetto, A., Singstock, N., Siraichi, Y., Sitdikov, I., Sivarajah, S., Sletfjerding, M. B., Smolin, J. A., Soeken, M., Sokolov, I. O., SooluThomas, Steenken, D., Stypulkoski, M., Suen, J., Sun, S., Sung, K. J., Takahashi, H., Tavernelli, I., Taylor, C., Taylour, P., Thomas, S., Tillet, M., Tod, M., de la Torre, E., Trabing, K., Treinish, M., TrishaPe, Turner, W., Vaknin, Y., Valcarce, C. R., Varchon, F., Vazquez, A. C., Vogt-Lee, D., Vuillot, C., Weaver, J., Wieczorek, R., Wildstrom, J. A., Wille, R., Winston, E., Woehr, J. J., Woerner, S., Woo, R., Wood, C. J., Wood, R., Wood, S., Wood, S., Wootton, J., Yeralin, D., Young, R., Yu, J., Zachow, C., Zdanski, L., Zoufal, C., Zoufalc, a matsuo, adekusar drl, azulehner, bcamorrison, brandhsn, chlorophyll zz, dan1pal, dime10, drholmie, elfrocampeador, faisaldebouni, fanizzamarco, gadial, gruu, jliu45, kanejess, klinvill, kurarrr, lerongil, ma5x, merav aharoni, michelle4654, ordmoj, sethmerkel, strickroman, sumitpuri, tigerjack, toural, vvilpas, welien, willhbang, yang.luh, yelojakit, and yotamvakninibm (2019). Qiskit: An open-source framework for quantum computing.

[Qiskit, 2020a] Qiskit (2020a). Operators. https://qiskit.org/documentation/tutorials/circuits_advanced/2_operators_overview.html.

[Qiskit, 2020b] Qiskit (2020b). Proving Universality. https://qiskit.org/textbook/

ch-gates/proving-universality.html.

[Qiskit, 2020c] Qiskit (2020c). QuantumCircuits. https://qiskit.org/documentation/apidoc/circuit.html.

[Rigetti, 2020] Rigetti (2020). Forest SDK Documentation.

[Smith et al., 2016] Smith, R. S., Curtis, M. J., and Zeng, W. J. (2016). A practical quantum instruction set architecture.

[The Cirq Developers, 2020] The Cirq Developers (2020). Cirq Documentation.

[Van Rossum and Drake, 2009] Van Rossum, G. and Drake, F. L. (2009). Python 3 Reference Manual. CreateSpace, Scotts Valley, CA.

[van Rossum et al., 2001] van Rossum, G., Warsaw, B., and Coghlan, N. (2001). Style guide for Python code. PEP 8.

[Zhang et al., 2018] Zhang, X., Xiang, H., Xiang, T., Fu, L., and Sang, J. (2018). An efficient quantum circuits optimizing scheme compared with qiskit. CoRR, abs/1807.01703.