COM SCI 239

Quantum Programming: Running Qiskit on a Quantum Computer

June 14, 2020

Author
Arjun Raghavan and Bryan Pan

1 Experience

For all the programs, we added a couple of functions that were specific for IBMQ. We decided that it would be best to maintain the local backend implementation in case users didn't have a valid API Token.

We added a `load_api_token()` function that looks at a `.env` file to add and load the user's API Token. We also utilized Qiskit's functions such as `least_busy` and `job_monitor` to add some optimization for user experience purposes (specifically, to choose the least busy IBMQ backend for speediest execution, and to wait for execution to complete).

We use a function, `check_validity`, to read the `counts` dictionary returned from an executed job, as we did in the previous assignment. However, due to the fact that real qubits are more volatile and prone to decoherence and other inaccuracies, we had to slightly change many of these functions. We also made adjustments to our circuits because we found that the U_f matrices we constructed based (see section 1.1 of our previous report) were not optimized for real devices. We noticed that one of the biggest issues with using real devices was that two-qubit operations required said qubits to be *physically adjacent* to one another. With a simulator, we got away with treating our circuit as fully-connected, allowing for our gates to be directly applied. However, we found that with real devices, this adjacency constraint added significant overhead to our circuits, causing them to produce the following unfortunate error message:

```
Circuit runtime is greater than the device repetition rate [8020].
```

We went about changing our programs to utilize only basic gates—specifically, Z, X, and $CNOT$ (a.k.a. CX) to approximate the algorithms we learned in class. They are detailed individually below.

1.1 Deutsch-Josza

Circuit

This algorithm is simple enough that even the original oracle matrix held up fine on actual devices for up to 4-bit strings. However, on 5 and more qubits, the above error reared its head. Luckily, implementing a Deutsch-Josza oracle for a given function f using only basic gates is rather simple:

- If f is constant and only returns 1, apply X to the helper qubit, as seen in Figure 1.
- If f is balanced, simply apply $CNOT(q_i, b)$ for each $q_i \in x$, where x is the input qubit register and b is the helper qubit. In doing so, any input qubit with an odd number of 1s will set the helper qubit to 1, and any input qubit with an even number of 1s will set the helper qubit to 0. This is shown in Figure 2.

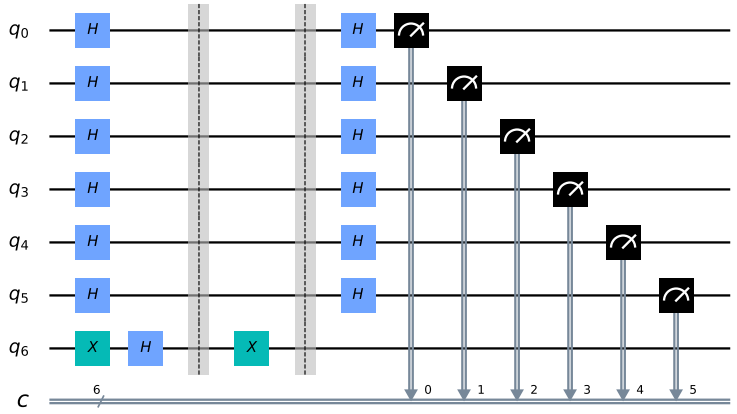

Figure 1: Circuit diagram for 6-bit Deutsch-Josza for a constant function that always returns 1.

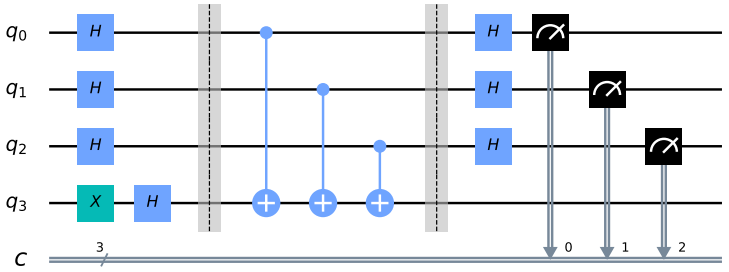

Figure 2: Circuit diagram for 3-bit Deutsch-Josza for a balanced function.

Validity

Because of noise in the results of running this circuit, it is not helpful to check for whether the output is only `00...0` in the case of a constant function, and otherwise for a balanced function—in most cases, a variety of outputs will be recorded. Instead, validity is determined by checking whether the number of occurrences of `00...0` is above a certain threshold percentage of the total number of shots, such as 90% of 1000 shots (the values chosen for this report).

1.2 Bernstein-Vazirani

Circuit

We adjusted the Bernstein-Vazirani circuit to apply an oracle constructed as follows. If the secret bit string's ith bit was a `1` then apply a $CNOT(q_i, b)$ where b is the helper qubit. We

found this to be an adequate approximation of the U_f matrix that we implemented in the previous two assignments, and it had a significantly smaller circuit size. Figure 3 portrays such a circuit.

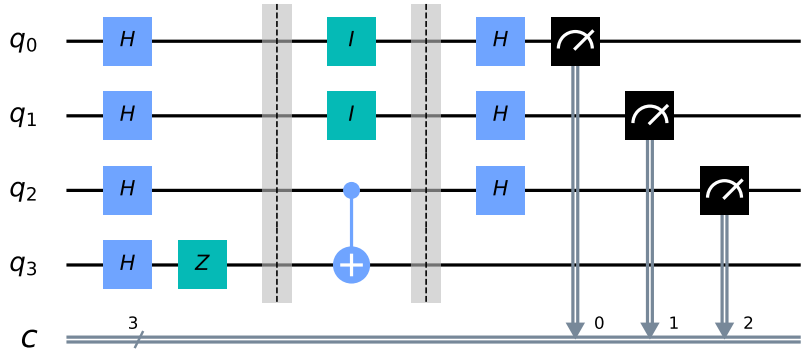

Figure 3: Circuit diagram for 3-qubit Bernstein-Vazirani, with $a =$ `011` and $b =$ `0`.

Validity

Similar to Deutsch-Josza, Bernstein-Vazirani was adjusted to look at the bit string with the highest return rate across 1000 shots to account for qubit volatility.

1.3 Simon

Circuit

Simon's algorithm proved the most difficult algorithm by far to implement as a generalized set of simple gates. We attribute this drastic spike in difficulty to the fact that instead of working with a single helper qubit, as is the case with the other three algorithms, we were in fact working with a helper qubit register of size n in addition to the input n qubits.

However, detailed below are some of our attempts to arrive at a way to produced a generalized set of gates for this algorithm:

- We looked for actionable patterns in the U_f matrices we had used in the previous assignment. As can be seen in Figure 4, for example, it turns out there were noticeable patterns in the arrangement of block matrices along the diagonal of the overall matrix.

  To be specific, consider each 4-by-4 block matrix along the diagonal of U_f. Let them be indexed in increasing order, beginning with 0, considering the binary equivalent of each index. That is, for a 2-qubit U_f as in Figure 4, the top-left block matrix is matrix `00`, the next one is matrix `01`, the one after that is `10`, and the bottom-right one is `11`.

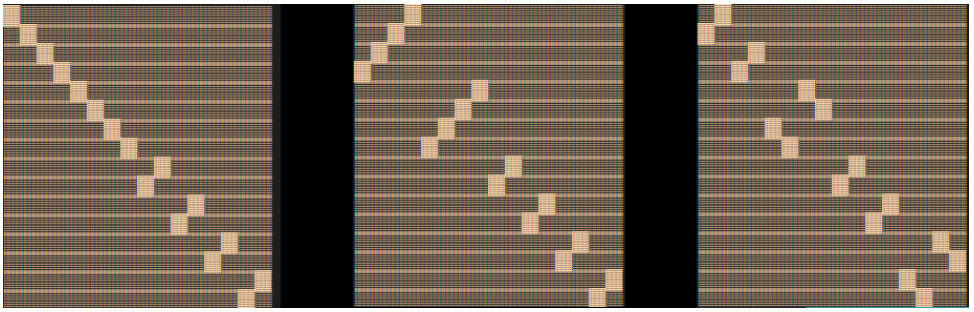

Figure 4: U_f matrices for 2-qubit Simon with mappings (from left to right): `{00→00←01, 10→01←11}`, `{00→11←01, 10→01←11}`, and `{00→01←10, 01→10←11}`. The solid block represents a `1`, and the other blocks are `0`. Notice the regular patterns along the diagonal, which consists entirely of tensor products of I and X gates in varying orders.

Then, each matrix represents an "encoding" of the output of f when applied to its index, where a `0` corresponds to the identity I and a `1` corresponds to the Pauli X gate. Suppose for index `10`, we have $f(\texttt{10}) = \texttt{01}$. Then the corresponding block matrix at that index along the diagonal is of the form $I \otimes X$.

Unfortunately, our discoveries stopped at recognizing this pattern. We were unsure how to proceed with this knowledge—how can we decompose a block matrix such as this one into fundamental gates? Perhaps with more time, we could arrive at a reasonable solution.

- We attempted to apply some of the principles outlined in Qiskit's documentation on implementing Simon's algorithm [Qiskit, 2020]. However, this proved futile, because while the documentation's steps were useful in directly producing a randomized quantum oracle, it would not be useful for our architecture, which involved the creation of a function mapping for f and a choice of s which is then used to produce an oracle. We were not willing to compromise and change our architecture specifically for Simon because it would render us unable to draw any meaningful comparisons with the other algorithms and implementation.

In the end, for the single case of $n = 2$, we resorted to a method inspired by the Qiskit documentation mentioned above [Qiskit, 2020], as seen in Figure 5.

For each bit s_i of s, whenever $s_i = \texttt{1}$, we have $CNOT(q_{n-1-i}, b_i)$. This is only an approximation, and is not precise—hence why the algorithm is not accurate for $n > 2$, on top of the already-present noise of a real quantum device. However, for $n = 2$, it works perfectly.

Validity

No changes needed to be made to the linear independence solver, thankfully. For checking validity, we thought we would have to make some changes, but we found that we did not have to. Initially, because of noise in the results returned by the quantum computer, we decided to choose the $n - 1$ *most common* output bit strings, as we felt that this was a reasonable approximation of determining a set of linearly independent ys. However, we later realized that this approach was flawed, because it was entirely possible that a set of linearly independent ys in one set of runs was not linearly independent with the ys in a different set. We then discovered

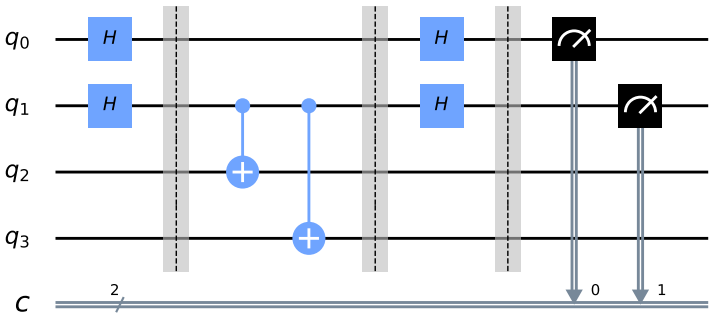

Figure 5: Circuit diagram for 2-bit Simon with $s =$`10`.

that an IBMQ `Result` actually had another function, `get_memory`, which returned an array of every single output bit string for each shot (so long as the `memory` option for the `execute` call was `True`). So, we elected to use this output in post processing, and so we could use the exact same process we had used for the previous report, using the IBMQ simulator.

1.4 Grover

Circuit

We kept the oracle unchanged, as we found that converting it to gates was difficult and didn't dramatically change the output.

Validity

Again, Grover was adjusted to look for the bit string with highest return rate, and if the number of successful calls was in the majority (in case there were multiple bit-strings x that f(x) = 1).

That is, we discovered that our implementation of Grover did not necessarily return a bit string on which the application of the input function f would return `1`. In reality, we found that we received as output one of the inputs on which the application of f produced whatever output formed the *majority* of outputs, which could very well be (and in some of our randomly-generated test cases, was) equal to `0` instead of `1`.

2 Evaluation

2.1 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. Also, in manually using basic gates to implement the quantum oracles, we limited transpilation time to only adding those gates required to swap adjacent qubits (so as to allow virtual operations on multiple qubits). As such, there was no compilation time to speak of, and execution was incredibly fast.

We tested our code with a variety of values of n (the length of the input bit string). We found that even despite the noise, our results still closely matched the expected outcomes of applying our algorithms. We found that (naturally) a larger number of shots ended up producing more accurate results, so we decided on a fixed number of **1000 shots** for all of our trials.

Unfortunately, due to limitations in our implementation of Simon's algorithm, we were unable to implement it for values of n greater than 2, as noted in the previous section. As such, it has been omitted from the below graphs. Instead, we decided to focus on its runtime variance for different functions f, i.e. varying U_f matrices.

Also, we found that Deutsch-Josza, being the simplest of the four algorithms, was able to be run for up to 14 qubits, the largest number we were able to achieve yet. Therefore, we used it as a benchmark for testing the reliability of IBM's largest device, Melbourne (`imbq_16_melbourne`).

2.2 Noise and volatility

Naturally, the volatility of physical qubits means that results will more often than not be extremely noisy, to the point where probabilistic algorithms such as Simon and Grover may produce entirely incorrect answers. This was the case for Simon moreso than Grover, and so, as mentioned above, we have chosen to use 2-qubit Simon as a benchmark in charting the noisiness of a few of IBM's devices.

We tested Simon's algorithm specifically for a function f for which we had $s =$ `10`. As such, based on an implementation of the oracle similar to that shown in Figure 5, we expect the circuit to produce `00` and `01` in relatively equal probability, i.e. approximately 50% each.

As Simon's algorithm on a 2-bit string input requires 4 qubits, we were able to run the circuit on `ibmq_rome`, `ibmq_essex`, `ibmq_burlington`, `ibmq_london`, `ibmq_ourense`, `ibmq_vigo`, `ibmqx2`, and of course, `ibmq_16_melbourne`, providing us with a wide variety of results. In order to gauge the variation of results from the expected probabilities of our outputs, we used the formula for the root mean squared error:

$$\text{RMSE} = \sqrt{\frac{1}{2^n} \sum_{q \in \{0,1\}^2} \left(\Pr(q) - \widehat{\Pr(q)} \right)^2}$$

Here, n is the length of the output bit string ($n = 2$ in our case), $\widehat{\Pr(q)}$ is the expected probability of producing the bit string `q`, and $\Pr(q)$ is the actual such probability.

The RMSE for each of the mentioned devices are plotted on the histogram in Figure 6.

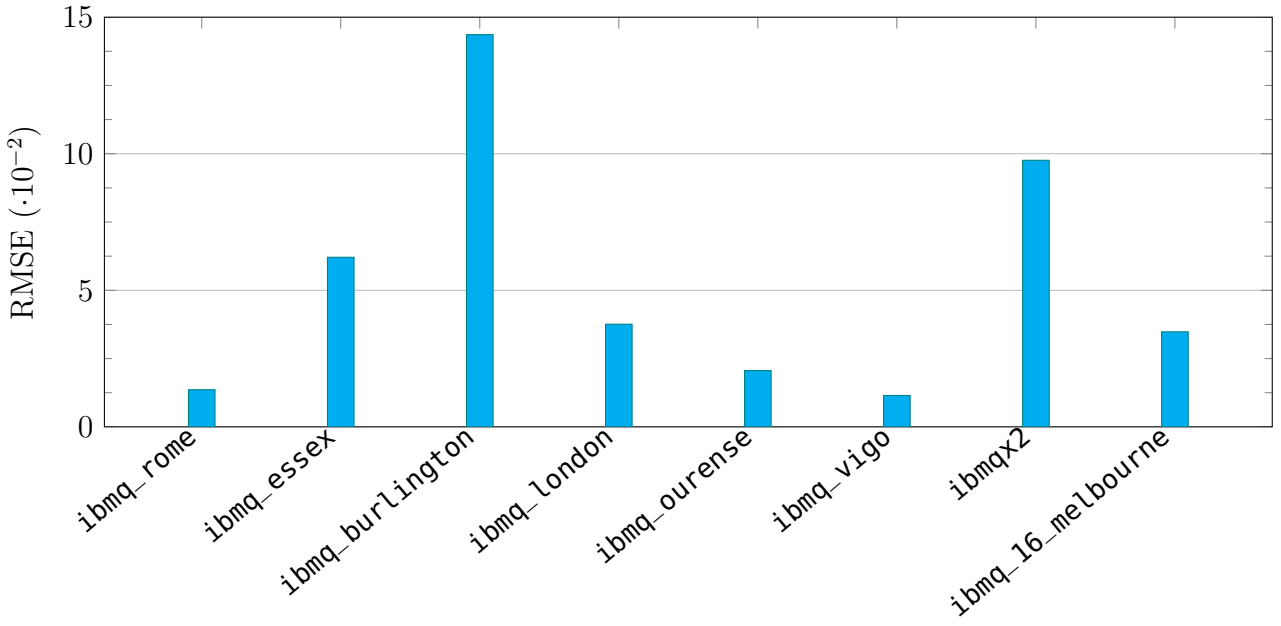

Figure 6: Root mean squared errors in outputs for 2-qubit Simon on each quantum device. Notice the scale of the y-axis is 10^{-2}

Notice the high error rates for `ibmq_burlington`, `ibmq_essex`, and `ibmqx2`. This is expected, as some of the qubits used in the ciruit are those which produced the highest error rate. For example, Figure 7 portrays the circuit that was actually run on `ibmq_burlington`; notice the use of qubit 2.

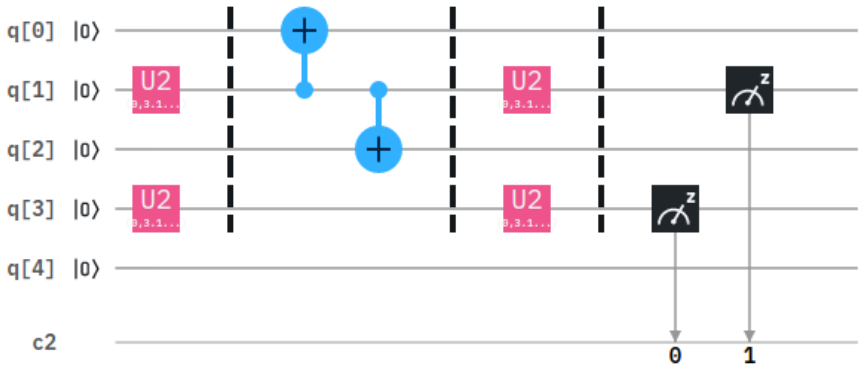

Figure 7: The circuit run on `ibmq_burlington`.

IBM allows us to view the structure of each device and the reliability of each qubit in it. As can be seen in Figure 8, the individual error rates for a single-qubit unitary gate and a CNOT on qubit 2 are comparatively extremely high.

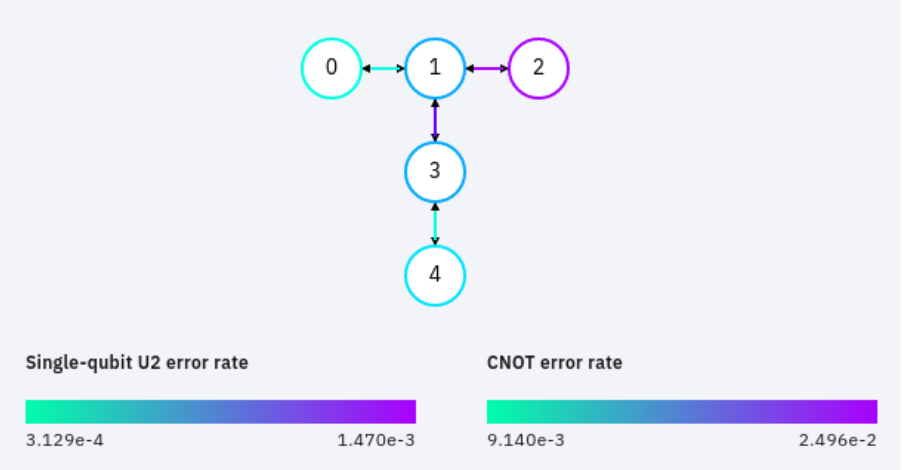

Figure 8: The structure and reliability of `ibmq_burlington`.

The lower error rates for the various other devices correlates with their popularity; for `ibmq_rome`, we experience the longest queue time of all our tests (coming in at 19 entire minutes).

2.3 Scalability

Figure 9 plots the running time versus the length of the input for each of the three other algorithms. Clearly, the growth in runtime for each algorithm is clearly far lower than it was for the simulator (which at its core exhibits behavior similar to that of classical algorithms), inching higher at a linear pace, as expected.

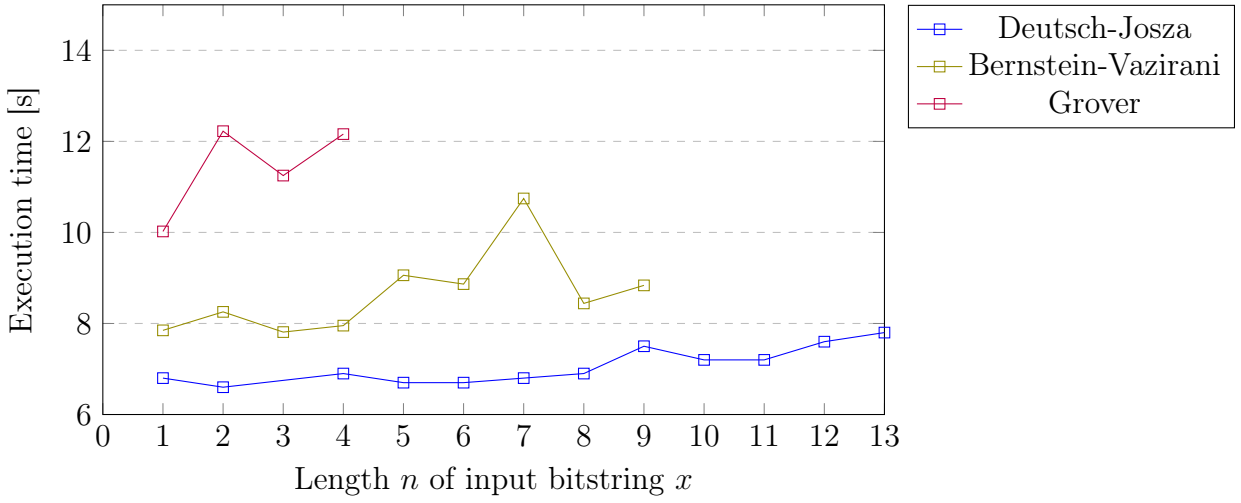

Figure 9: Graph of runtime (s) vs. the length of the input bitstring.

Differences from simulation

Notice how the y-axis is in seconds, and *not* milliseconds like it was for the simulation. This is also expected behavior, in fact. These algorithms are *asymptotically better* than their classical

counterparts, meaning that for extremely large values of n—that is, of an order of magnitude higher than, say 10^5—quantum algorithms will reign supreme. However, we are nowhere near capable of using quantum algorithms for such values of n; as such, it remains clear that classical algorithms are still faster for these extremely low values of n.

Regardless, as noted, the growth of runtime for these algorithms is very, very slow. Compare that to the exponential growth of classical algorithms; it is evident that eventually, the quantum algorithms will be faster.

For our implementation of Grover on a simulation, 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 $N = 2^n$. Similarly, for our implementation of Simon on the simulator, 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.

However, as noted above, running on an actual quantum computer threw a wrench into things. Because of rate limitations, queue times, and other inhibitors, we had no choice but to run every iteration of our circuit with a single `execute` call and a parametrized number of shots. This meant that our readings for runtime formed an unfair comparison; while we were able to prematurely stop executing another iteration of the circuit on a simulator, we had no choice but to run the total number of shots on the quantum computer.

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, 2020] Qiskit (2020). Simon. https://qiskit.org/textbook/ch-algorithms/simon.html#oracle.