PyQuil - The Engineering Group

Quantum Programming

Attiano Purpura-Pontoniere attiano@cs.ucla.edu **Nishant Sabharwal** nsabharwal@ucla.edu

Pratik Sathe psathe@physics.ucla.edu

May 14, 2020

Abstract

We use pyQuil (version = 2.19.0) to simulate four popular quantum algorithms (Deutsch-Jozsa, Bernstein-Vazirani, Simon's and Grover's search algorithms) on a classical computer. For each of the algorithms, we benchmark the performance of our simulations, measuring the scalability of compilation and execution with the number of qubits and dependence of the execution time on the oracle U_f. We then discuss our code organization and implementation of quantum circuits. Finally, we provide some comments about pyQuil and its documentation, and provide some suggestions for improving the user experience.

1 Design and Evaluation

1.1 Deutsch-Josza algorithm

First, we summarize the problem statement:

Given a function $f : \{0,1\}^n \rightarrow \{0,1\}$ which is either balanced or constant, determine whether it is balanced or constant.

While a classical algorithm requires $2^{n-1} + 1$ calls to the function f, the Deutsch-Jozsa algorithm finds a solution with just one call to the oracle U_f, which is defined by its action on the computational basis as follows:

$$U_f \left|x\right\rangle \otimes \left|y\right\rangle = \left|x\right\rangle \otimes \left|y \oplus f(x)\right\rangle, \tag{1}$$

for all $x \in \{0,1\}^n$ and $y \in \{0,1\}$. The Deutsch-Jozsa circuit is shown in Fig. 1. If the measured state is $\left|0\right\rangle^{\otimes n}$, then the algorithm concludes that f is constant, and balanced otherwise.

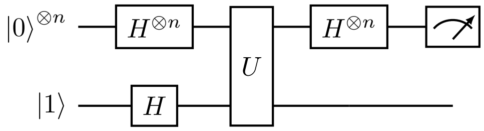


Figure 1: Deutsch-Jozsa algorithm

Now we discuss various aspects of our code design:

- **Implementation of** U_f: Before implementing the unitary matrix U_f, it is first necessary to obtain the matrix representation in the computational basis. To that end, we implement the following pseudo-code:

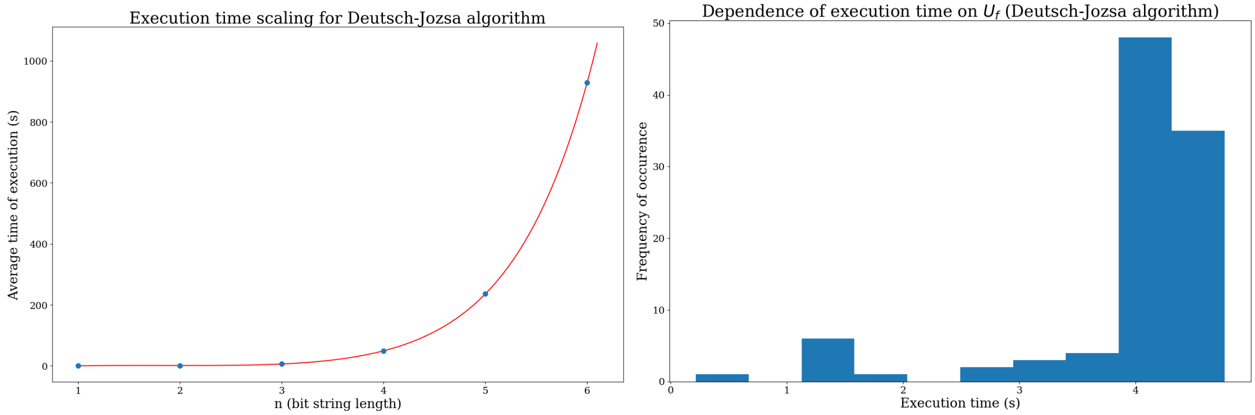
```
1. Initialize $U_f$ to be a $2^n \times 2^n$ zero matrix.
```

2. For every possible bit string a of length n, and bit b, obtain $x = a + b$ and $y = a + (b \oplus f(a))$, with '+' denoting string concatenation. Convering the two strings to their decimal representations, we set the row-y, column-x entry of U_f to 1.

This procedure ensures that U_f maps all the computational basis vectors according to the definition (1). Once the matrix is defined, we use `DefGate` and `get_constructor()` in PyQuil in order to get the Gate object, and its constructor respectively. These can then be inserted in the Pyquil `Program` object when we define the circuit.

- **Code readability:** We split our code into multiple small functions, each of which executes a simple task. There are two ways a user can access the implementation. For a higher level call, the user can input the function f to the function `run_DJ`, or for a lower level implementation (i) call the function `create_Uf` to creates the oracle U_f matrix, and (ii) call the `DJ()` function which executes the Deutsch-Jozsa algorithm, and concludes whether the function is constant or balanced.
  We will briefly describe the role of each function in our code below:
  1. `DJ(f,n)`: Calls `create_UF` and `DJ` functions for higher level access.
  2. `DJ(UF_def, n, time_out_val)`: Executes the Deutsch-Jozsa algorithm and interprets the result (balanced/constant). Returns 1 if the function is determined to be constant, and 0 otherwise. The inputs are the oracle gate definition, the value of n, and (optional) the timeout value for circuit execution.
  3. `binary_add(s1,s2)`: Obtains a binary XOR sum of two bit strings s_1 and s_2.
  4. `give_binary_string(z,n)`: Obtains a length n binary representation of integer z.
  5. `create_all_functions(n)`: Creates all possible functions which are balanced or constant for a given value of n.
  6. `get_random_f(n)`: Create a balanced/constant function for length n bit strings randomly. Out of the $({}^{N}C_{N/2} + 2)$ possible functions f (with $N := 2^n$), each is chosen with the same probability.
  7. `create_Uf(f)`: Given a function f, calculates the corresponding oracle matrix U_f in the computational basis.
  8. `verify_dj_output(DJ_output, f)`: Determines whether the algorithm succeeded or failed.

  By modularizing the algorithm simulation process into small functions executing well-defined tasks, we have made our code readable.
- **Preventing access to U_f implementation:** We discuss this in detail at the end of the document.
- **Parametrizing the solution in n:** The circuit is different for every value of n. In the function `DJ()`, we dynamically create the circuit corresponding to any given value of n, compile it using the `nq-qvm` quantum virtual machine, run the circuit, and interpret the result of the measurement to determine whether the function is balanced or constant.
- **Code testing:** For any given function f, our function `DJ()` returns 1 if it determines that f is constant, and 0 if the function is balanced. In order to verify that the algorithm's output is correct, we define a function `verify_dj_output`, which determines (classically) whether f is constant or balanced, and checks whether the output from `DJ` matches this conclusion. The Deutsch-Jozsa algorithm successfully determined the nature of f correctly every single time in our simulations, after running the simulations for five randomly chosen f's for every $n \in \{1, 2, 3, 4, 5, 6\}$, and for 100 randomly chosen f's for $n = 3$.
- **Dependence of execution time on U_f:** We observed that although a majority of U_f's lead to similar execution times, there are some functions which are executed much faster. Typically, we observed that the closer the U_f matrix is to the identity matrix (i.e. the smaller the number of non-zero off-diagonal entries in U_f), the faster the simulation of the circuit. This observation is in line with the intuition that multiplication with a unitary matrix which has more number of 1's on the diagonal is easier and quicker to implement, than by a unitary matrix with non-zero off-diagonal matrix elements. Choosing $n = 3$, we simulated the circuits for 100 randomly chosen U_f's. A histogram of the recorded execution times can be seen in figure 2b.
- **Scalability:** We observed that the execution time for the circuit simulation grows exponentially with the number of qubits involved, i.e. $n + 1$. For each $n \in \{1, 2, 3, 4, 5, 6\}$, we executed the Deutsch-Jozsa algorithm for five randomly chosen U_f's, and obtained the average execution time for each value of n. As can be seen in figure 2a, the execution time grows exponentially, with $n = 6$ requiring about 20 minutes of execution time. We decided to limit our analysis to $n \leq 6$, since we observed that $n = 7$ required more than an hour of execution time for a single U_f making it unfeasible to simulate multiple circuits. The exponential growth of execution time with n is in line with the fact that the matrix sizes grow exponentially with the number of qubits.



(a) Execution time scaling.

(b) Execution time histogram ($n = 3$).

Figure 2: Deutsch-Jozsa algorithm: Benchmarking.

1.2 Bernstein-Vazirani algorithm

First, we summarize the problem statement:

Given a function $f : \{0,1\}^n \rightarrow \{0,1\}$ of the form $f(x) = a.x + b$, with $a \in \{0,1\}^n$ and $b \in \{0,1\}$ are unknown bit strings, obtain the value of a.

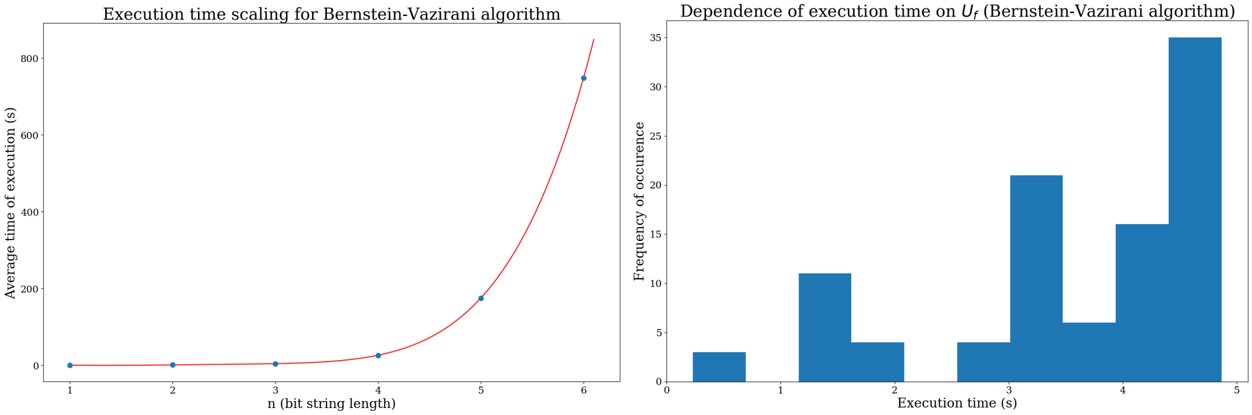
While a classical algorithm requires $\mathcal{O}(n)$ calls to the function f, the Bernstein-Vazirani (BV) algorithm obtains the bit string a with just one call to the oracle U_f, which satisfies equation (1). The quantum circuit for the BV algorithm is the same as that of the DJ algorithm (Fig. 1). The BV algorithm determines a to be the state of the n qubits measured at the end of the circuit.

Now we discuss various aspects of our code design:

- **Implementation of U_f**: Given a function f which satisfies the required criteria, our procedure for obtaining U_f for the BV algorithm is identical to that of the DJ algorithm. Once the matrix is defined, we use `DefGate` and `get_constructor()` in PyQuil in order to get the Gate object, and its constructor respectively. We insert these in the Pyquil `Program` object, when we define the circuit.
- **Code readability:** We split our code into multiple small functions, each of which executes a simple task. There are two ways a user can implementation the code. For a higher level implementation, the user can input the function f to the function `run_BV`, or for a lower level implementation (i) call the function `create_Uf` to create the oracle U_f matrix, and then (ii) call the `BV()` function which executes the Bernstein-Vazirani algorithm, to determine the value of a. We split our code into multiple small functions, each of which executes a simple task. A user with a function f only needs to access two functions in our module: (i) The function `create_Uf` which creates oracle U_f matrix, and (ii) `BV()` which executes the BV algorithm, and returns the value of a.
  In our implementation of the BV algorithm, we use the functions `binary_add(s1,s2)`, `give_binary_string(z,n)`, `create_Uf(f)` from the DJ algorithm without modification. The only new functions used are:
  1. `run_BV(f)` for higher level access to the implementation of the BV algorithm (calls `create_Uf` and `BV` functions.
  2. `BV(UF_def, Uf_gate, n, time_out_val)`: Executes the BV algorithm and returns the determined value of a.
  3. `get_random_f(n)`: Generates the strings a and b randomly, and returns the corresponding function f.
  4. `verify_BV_output(BV_output, f)`: Determines whether the algorithm succeeded or failed.

  By modularizing the algorithm simulation process into small functions executing well-defined tasks, we have made our code readable.
- **Preventing access to U_f implementation:** We discuss this aspect at the end of the report.

- **Parametrizing the solution in n**: The circuit is different for every value of n. In the function `BV()`, we dynamically create the circuit corresponding to any given value of n, compile it using the `nq-qvm` quantum virtual machine, run the circuit, and interpret the result of the measurement to determine whether the function is balanced or constant.
- **Code testing:** For any given function f, the function `BV()` returns the string a which it determines f to correspond to. In order to verify that the algorithm's output is correct, we define a function `verify_BV_output`, which determines (classically) whether the determined value of a is correct for the given function f. (`verify_BV_output` first obtains b by evaluating $f(0)$, and then calculates a function $f' = a.x + b$, with the value of a determined by the DJ algorithm. If f and f' are identical, it determined that the BV algorithm succeeded, and failed otherwise.) The BV algorithm successfully determined the string a every single time in our simulations, after running the simulations for five randomly chosen f's for every $n \in \{1, 2, 3, 4, 5, 6\}$, and for 100 randomly chosen f's for $n = 3$.
- **Similarities in our codes for the BV and DJ implementations**: We estimate about 80% code similarity between our BV and DJ algorithms. In particular, the functions `binary_add(s1,s2)`, `give_binary_string(z,n)` and `create_Uf(f)` are identically defined in both the implementations. Moreover, the only difference between the `BV` and `DJ` functions is in the interpretation of the outputs, with the `DJ` function returning 0 or 1, and the `BV` function returning a bit string of length n. The similarity in the codes is due to the fact that the circuits and oracle definition in both these algorithms are identical.
- **Dependence of execution time on U_f**: Similar to the DJ algorithm, we observe a variance in the execution time with U_f. For $n = 3$, we simulated 100 randomly chosen U_f's, and plotted the histogram in figure 3b. Clearly, there is a higher variance in the execution times. This can be justified by the fact that the allowed U_f's in the BV problem have a larger variation from the identity matrix, compared with the allowed U_f's in the DJ problem.



(a) Execution time scaling.

(b) Execution time histogram ($n = 3$).

Figure 3: Bernstein-Vazirani algorithm: Benchmarking.

- **Scalability:** Similar to the DJ algorithm, we observed an exponential increase in the execution time with n. For each $n \in \{1, 2, 3, 4, 5, 6\}$, we obtained the average execution time for five randomly chosen U_f's. As can be seen in figure 2a, the execution time grows exponentially, with $n = 6$ requiring about 15 minutes of execution time. We decided to limit our analysis to $n \leq 6$, since we observed that $n = 7$ required more than an hour of execution time for a single U_f making it unfeasible to simulate multiple circuits. The exponential growth of execution time with n is in line with the fact that the matrix sizes grow exponentially with the number of qubits.

1.3 Simon's algorithm

Let $f : \{0,1\}^n \rightarrow \{0,1\}^n$ be the function given to us. f satisfies the property that $f(x) = f(y)$ iff. $x \oplus y \in \{0, s\}$. The problem is finding the secret s. The classical implementation requires testing up to $2^{n-1} + 1$ inputs until two inputs with the same output are found. If none are found, s is 0 and f is 1 to 1.

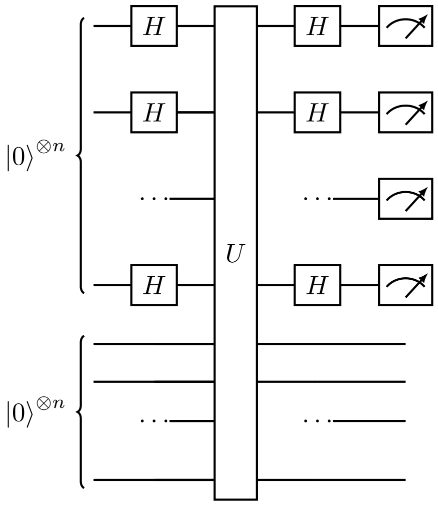


Figure 4: Simon algorithm

The quantum solution involves a circuit with 2n qubits - n for each bit and n helper bits. The oracle, U_f is as described in (1). Fig. 4 shows all qubits are initially set to $|0\rangle$ and have Hadamard applied to the first n, followed by the oracle. Only the first n qubits have Hadamard applied again and measured.
Each time Simon is measured, the result is a y which satisfies $y \cdot s = 0$. If the circuit is measured n-1 times, and all values of y are independent, the system of equations can be solved with 2 values of s, one of them being 0. However, the n-1 values of y may not be all independent so it may need to be measured more times. Though it's possible to keep measuring dependent vectors, there is a 99% probability that $4 * m * (n - 1)$ measurements, where $m = 5$, will result in n-1 independent equations.

- **Implementation of** U_f: Similarly, we construct a matrix from the function f. The pseudo-code is described below.

  1. Initialize U_f to be a $2^{2n} \times 2^{2n}$ zero matrix. In
  2. For every possible bit string x of length n, compute the bit string $f(x)$ of length n. For every bit string b of length n, set the entry whose row is the integer representation of b and whose column is the integer representation of $(b \oplus f(x))$.

  Python allows this to be done in under 7 lines of code. It is easy to read with a single loop from 0 to 2^{2n}. Each iteration represents the concatenation of x and b and $f(x) \oplus b$ can easily be calculated in a few lines. This is abstracted away in it's own function `create_Uf`.

- **Classical Considerations**: Simon involves both a quantum portion and classical portion. The classical algorithm involves taking n-1 measurements trying to solve it, and another n-1 measurements if they are not independent. With the quantum version, $4m$ iterations ensure there is a 99% probability of success. However, time becomes an issue. With $n = 4$, it takes upwards of 3 minutes to measure once. Therefore, It is wasteful to throw away all n-1 equations when some of them are independent.
  We implemented a separate class called `Solver` to handle the classical portion. Each time we measure a vector y, we add it to the solver and check if it's linearly independent. We terminate as soon as we have n - 1 equations so no measurements are wasted. The pseudocode for adding is:

  1. If it is 0^n discard it and terminate.
  2. Find most significant bit
  3. If this is the first vector with that MSB, it is independent so save it. Otherwise, add the vector with that MSB to this one bitwise and repeat.

  When we terminate, we have n-1 independent equations. We solve as follows:

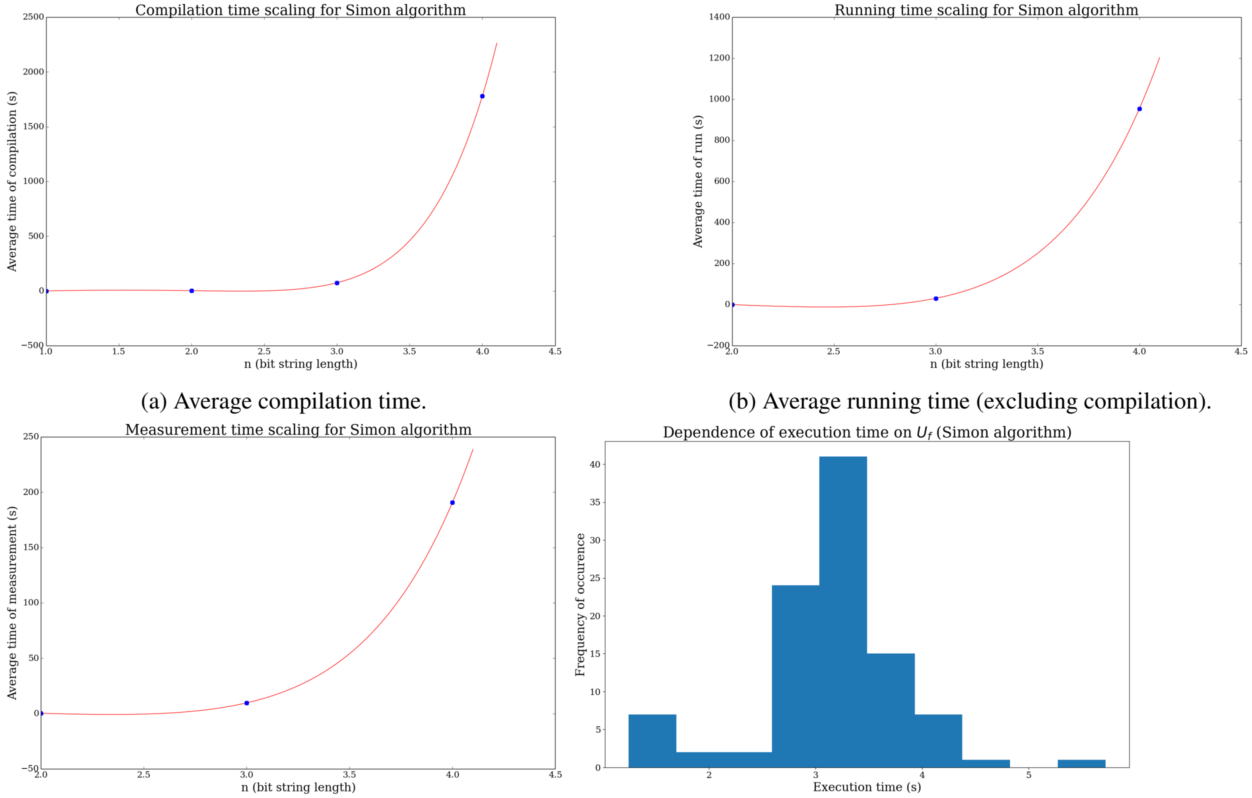
  1. Let m be the missing MSB, assume $y_m = 1$.
  2. Order the vectors by increasing MSB.
  3. Perform gaussian elimination by plugging each value in the equations of the higher MSB.

In the end we have a value for s. We are faced with two choices as the problem definition indicates $f(x) = f(y) \in \{0, s\}$. If f is 1 to 1, the calculated value of s incorrect. We must call $f(0)$ and $f(s)$ and check if they are equal. If they are, s is correct, otherwise s is 0. For testing we confirmed this method of verification works, however if we are strict about not having access to f, we must instruct the user that s is restricted to be nonzero. Since this is a probabilistic solution, there is exceedingly small probability that we keep measuring and keep getting dependent equations. As a fail safe, after $4m(n-1)$ measurements, if still n - 1 independent equations have not been found, we terminate. We still know x bits of information though since we have x independent equations, where x is the number of independent we have after terminating. This is a way to reduce the possible values of s remaining after knowing x bits. This is an exceedingly rare case with increasing m, so in this case we can simply apply brute force as the number of unknown bits n-x is hypothetically small. We try all possible values for s against the equations we do have and return all the possibles.

- **Design Considerations**: We need to measure the circuit a dynamic amount of times and compilation was determined to be the bottleneck. We opted not to use `run_and_measure` but rather split compilation and measure, so we save time by only compiling once per circuit.
- **Parametrizing the solution in n**: This is discussed in more detail at the end but U_f is created dynamically from f, the structure is created based off n, and gates are added to the first n wires in a for loop so the program works for any arbitrary n.
- **Testing**: An exhaustive approach was taking for testing. For a given n, there are 2^n possible secrets. However, there is also an exponential number of mappings since we can choose any output and assign it to any 2 inputs. This seemed too large to test since compilation and measurement time already increased exponentially. Instead we tested each of the 2^n possible secrets and selected 1 mapping at random.
  We wrote a test driver, `generate_functions(n)` which generated 2^n functions, one for each secret. We then compiled it 5 times and ran Simon 5 times for each compilation. As a result, each secret was tested 25 times. This was ran from $n = 1$ to $n = 3$. At $n = 4$, the PyQuil server began to crash inconsistently due to memory and taking over 30 minutes to compile since we were now dealing with a 2^8 by 2^8 matrix. As a result, for n = 4, we selected 5 secrets at random and compiled and tested once.
  To be able to test $s = 0$, in testing we assumed access to f. Once we calculated s, if $f(0) \neq f(s)$ we reported as $s = 0$.
- **Code readability**: The code was modularized as much as possible into functions. Most of the differences arose from the classical solver, the test drivers and minor differences in the circuit. The modules are described below:
  1. `run_simon(f, can_be_zero)`: This a high level function to determine s from f. It converts f to U_f, compiles the program, and takes the measurements. If `can_be_zero` is true, it calls f twice to confirm $s! = 0$. If the user know $s \neq 0$, they can call it with False and f will not be called.
  2. `Solver(n)`: This is unique to Simon. The classical solver is implemented as a class. For submission rules it was left in the same file but it could easily be imported to improve readability.
  3. `compile_simon(Uf_quil_def, n, time_out_val)`: We opted to split running and compilation since we wanted to measure multiples times. The separate modules allowed us to benchmark compilation and run time separately.
  4. `simon(num_bits, qc, executable)`: This runs the circuit. It keeps measuring and feeding the y values into the classical solver. It returns the calculated value of s. This allowed measuring the run time, and how many vectors were measured before being able to solve the equation.
  5. `create_Uf(f)`: This program takes f and calculates the U_f matrix. This is similar to the other programs. The logic is different since the arity is different.
  6. `generate_functions(n)`: This is a helper for testing. It generates all possible secrets of length n and selects a random mapping. This is different from other programs since it is specific for generating simon test programs.
- **Scalability**: In order to benchmark Simon's algorithm, while we were conducting the exhaustive testing, we recorded the time of each compilation, the total time to measure, and the total number of measurements before n - 1 equations were found. Fig. 5 shows the results for $n = 1$ through $n = 4$. For each value of n, there are $2n$ qubits. It should be noted that the benchmarking for $n = 4$ is less accurate since most runs did not succeed. The plots clearly show that both compilation time and running time scale exponentially with n.
  To see the effect U_f has on running time, we created a histogram for the running time of differnt U_f's selected at random for $n = 2$. The chart shows that U_f has a strong effect on the running time. Most center around 3 s but there are some outliers with much faster and slower run times. These results hold across both compilation and run time. Looking at data for $n = 3$, the average compilation time was 75 s. However, one circuit had an

average compilation time of 50 s and another other had a compilation time of 108 s. Different circuits showed a similar range in average run time. U_f has a strong influence on both compilation time and running time.

- **Measurements**: Fig 6 shows a histogram of the number of measurements that were taken before n-1 independent equations were found over 200 trials for $n = 3$. Note the data is a little skewed for 2 since 25 trials have s=0, so any equation except $y = 0^n$ can be used. For $n = 3$, we need 2 independent vectors before we can solve the system of equations to find s. We measured an average of 3.1 times in order to get 2 independent values of y. In the worst case, we measured 12 times before finding an independent set. Our fail-safe setting to stop measuring was set to fail after $4 * m * (n - 1) = 40$. In 200 runs, the closest we got to 40 was 12 which shows it would be an exceedingly rare occurrence to fail. A significant number, 53 out of the 200 runs resulted in 3 measurements needed to determine the necessary number of independent equations. This highlights the benefit of our classical solver optimization since the normal solver would have required 4 measurements.



(a) Average compilation time.

(b) Average running time (excluding compilation).

(c) Average time for a single measurement.

(d) Total execution time histogram ($n = 4$).

Figure 5: Simon's algorithm: Benchmarking. Running time and measurement is not shown for $n = 1$ since we only need $n - 1$ equations.

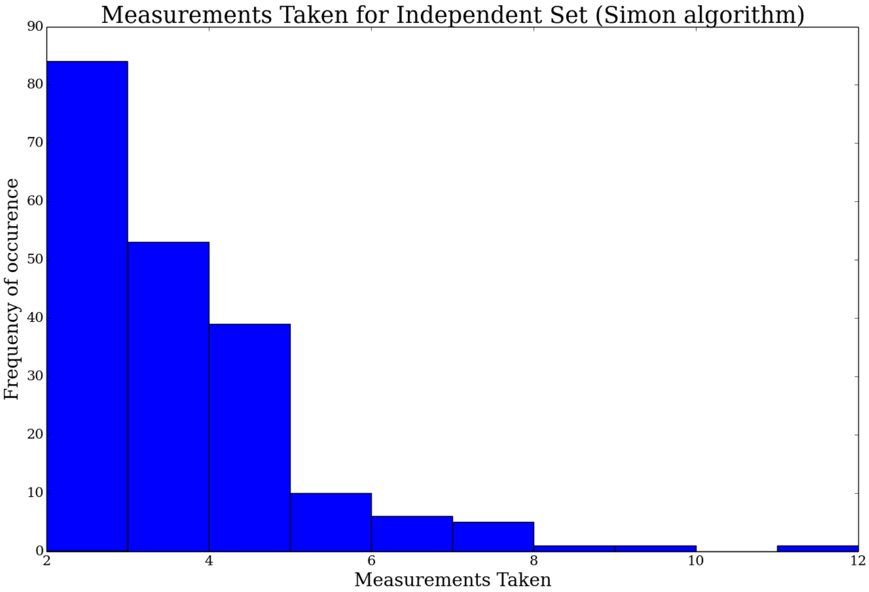


Figure 6: Measurements taken before finding $n-1$ equations for $n=3$.

1.4 Grover's search algorithm

First, we summarize the problem statement:
Given a function f : $\{0,1\}^n \to \{0,1\}$ that returns $f(x)=1$ for exactly $|x| = a << n$ inputs, find one of the inputs x that make f return 1.

Solving this problem on a classical computer requires 2^n queries to f, but with Grover's algorithm we can solve it with high probability with $O(\sqrt{2^n})$ operations of the oracle Z_f. The algorithm involves the so-called Grover's operator G, defined as

$$G := -H^{\otimes n} Z_0 H^{\otimes n} Z_f \left|x\right\rangle, \tag{2}$$

with Z_f and Z_0 defined through their action on computational basis vectors $|z\rangle$ as follows:

$$Z_f \left|x\right\rangle = (-1)^{f(x)} \left|x\right\rangle, \tag{3}$$

$$\text{and } Z_0 \left|x\right\rangle = \begin{cases} -\left|x\right\rangle & \text{, if } \left|x\right\rangle = 0^n \\ \left|x\right\rangle & \text{, if } \left|x\right\rangle \neq 0^n. \end{cases} \tag{4}$$

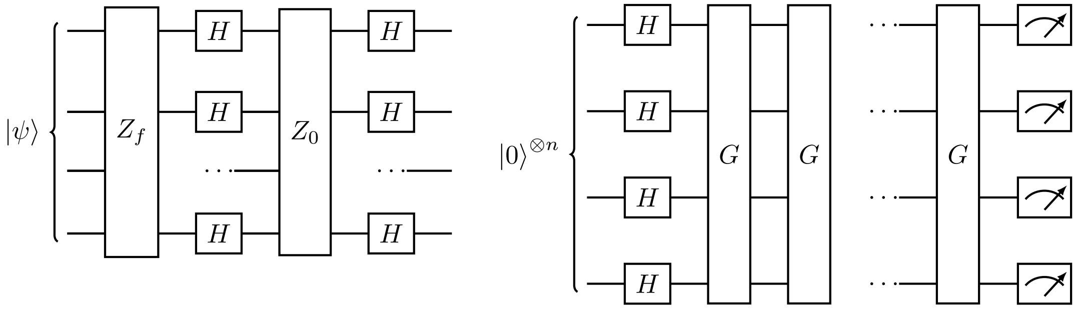
Grover's operator, G, is shown in Fig. 7a. In order to maximize the probability of success, the operator G is applied k number of times, with k being

$$k = CI\left(\frac{\pi}{4\theta} - \frac{1}{2}\right) \qquad \text{with } \theta := \arcsin\frac{a}{N}. \tag{5}$$

Here, $CI(y)$ denotes the integer closest to y. The complete circuit is shown in figure 7b. The probability of success (i.e. the probability of the final measurement resulting in a bitstring satisfying $f(x) = 1$) is given by

$$P_{success} = \sin^2((2k+1)\theta). \tag{6}$$

k (5) number of Grover's iterations G results in measurement of state $|x\rangle$ which satisfies $f(x) = 1$ with high probability, on the order of $95 - 99\%$.



(a) Grover's iteration G.

(b) Grover's circuit: Grover's operator G occures $k = \mathcal{O}(\sqrt{N})$ times.

Figure 7: Grover's search algorithm

Now we discuss various aspects of our code design:

- **Implementation of U_f, i.e.** G: Before implementing the unitary matrix U_f that corresponds to (2), it is first necessary to obtain the matrix representation in the computational basis. To that end, we need to implement Z_0 and Z_f with the following pseudo-code:

  1. Initialize Z_0 and Z_f to be a $2^n \times 2^n$ identity matrix.
  2. For Z_0 set $I_{00} = -1$. For Z_f set all $I_{f(x)=1} = -1$.
  3. Obtain G using (2)
  4. Obtain k (number of Grover iterations) which maximize the success probability.

  This procedure ensures that G in accordance with (3),(4),(2). Once the matrix is defined, we call `DefGate` and `get_constructor()` in PyQuil create a Gate object for (3),and (4). `Get_constructor()` gets the gate constructor once defined. These can then be inserted in the Pyquil `Program` object, when we define the circuit.

- **Code readability:** We split our code for Grover's algorithm into multiple small functions, each of which executes a simple task. A user with a function f only needs to access three functions in our module: (i) The function `create_zf(f,n)` which creates oracle Z_f matrix,(ii) The function `create_z0(n)` which creates oracle Z_0 matrix and (iii) `Grover(z0,zf,n,a)` which executes the Grover algorithm, and returns a $|x^n\rangle$ in the form of a list that matches an input x to the function f(x) that evaluates to 1. The gate definition of Z_f and Z_0 are passed in to `Grover(z0,zf,n,a)` as well as n, the number of qubits, and a, the number of inputs x that evaluate to f(x) = 1.
  We will briefly describe the role of each function in our code below:

  1. `grover(z0,zf,n,a)`: Executes the Grover algorithm and returns the result, the compile time, and the computation time. The inputs are the oracle gate definitions, the value of n, the number of qubits, and the size of the domain of f that evaluates to 1.
  2. `run_grover(f)`: runs Grover's algorithm and constructs G by first constructing Z_f and Z_0, assumes $a = 1$. Returns $|x\rangle$
  3. `check_correctness(func,result)`: Checks to see if the result from Grover's algorithm evaluates to 1, assumes func is an array that maps the domain to range of f by indexing func[x]. The implementation does type checking to see if the func passed in is a function definition and or an array and checks correctness. This assumes the input to f is an int representing an unsigned bit string.

4. `create_minus_gate(n)`: Creates an 2^nx2^n identity matrix with -1 on the diagonal. Used to implement the -$H^{\otimes n}$ at the end of (4).
5. `create_zf(f,n)`: Creates Z_f as a 2^nx2^n identity matrix with -1 on the diagonal value that corresponds to $|x^{\otimes n}\rangle$.
6. `create_z0(n)`: Creates Z_0 as a 2^nx2^n identity matrix with -1 on the diagonal value that corresponds to $|0^{\otimes n}\rangle$.
7. `all_f(n,a)`: Given a function f, creates and returns as a numpy matrix the corresponding oracle matrix Z_f in the computational basis for all possible functions f with a given number of qubits and number of values in the domain that evaluate to 1. There are $\binom{2^n}{a}$ such functions
8. `calc_lim(a,n)`: Given a the number of qubits and the size of the domain of f that evaluates to 1, `calc_lim(a,n)` returns k, the number of times to run Grover's algorithm and the theoretical probability of success given that k. This function returns the floor or ceil of k that maximizes the probability of Grover's algorithm successfully finding a bit-string that is part of the domain of f that evaluates to 1.
9. `convert_n_bit_string_to_int(x,n)`: Converts a list x of length n with entries of 0/1 that correspond to an unsigned encoding of an integer to an int x. `convert_int_to_n_bit_string(integ, n)`: Converts an int integ to a list of length n with entries of 0/1 that correspond to an unsigned encoding of an integer.

By modularizing the algorithm simulation process into small functions executing well-defined tasks, we have made our code more readable and reduce code replication.

- **Preventing access to U_f implementation:** We discuss this aspect at the end of the report.
- **Parametrizing the solution in n:** The circuit is different for every value of n. In the function `Grover(z0,zf,n,a)`, we pass in a dynamically created Z_0 *and* Z_f by calling `create_zf(f,n)` and `create_z0(n)` with an arbitrary value of n. We then compile the quantum program with these dynamically sized gates using the `nq-qvm` quantum virtual machine, run the circuit a sufficient number of times, k, such that the $|x\rangle$ input vector is rotated to match a vector $|x\rangle$ that evaluates as input on the oracle function to 1.
- **Code testing:** In order to verify that the algorithm's output is correct, we define a function `check_correctness`, which determines (classically) whether the measured state x satisfies $f(x) = 1$. Grover's algorithm is probabilistic, succeeding with a high probability, but with a small probability of failure. In order to test that our implementation succeeds at a rate similar to the theoretical success rate (6), we implemented our code for a variety of combinations of n and a. For each combination, we ran our code over 100 times, recording the fraction of times the algorithm succeeded. The theoretical results match the numerical results very well, as can be seen in the table below:

Row	n	a	Theoretical failure rate	Numerical failure rate
1	2	1	50.0%	49.0%
2	3	1	5.46%	4.25%
3	3	2	0.0%	0.0%
4	3	3	15.6%	19.6%
5	4	1	3.87%	3.75%
6	4	2	5.46%	7.5%
7	4	3	5.08%	3.75%

Table 1: Grover's Algorithm: Numerical rate of success matches with the theoretical value

- **Dependence of execution time on U_f:** In order to analyze the variance of the execution time with U_f, we implemented the following:
  1. We tested all 16 possible functions for $n = 4$ and $a = 1$. The fastest U_f was approximately 60% faster than the slowest U_f (see 8).
  2. We expect that for a fixed n and small a's, increasing the value of a should increase the execution time. We expect this because the larger the value of a, the more the deviation of G from the identity matrix. Hence, for $a = 1, 2, 3$, we computed the average compilation and computation time over 100 iterations for each case. As expected, the execution time increases with increasing a (see 9).
- **Scalability**: We benchmarked the average execution and compilation time of the virtual quantum computer for every $n \in \{2, 3, 4, 5, 6\}$ and $a = 1$. We determined the average execution time for a given n by averaging over all possible functions for that n and $a = 1$. We observed that the execution time for the quantum circuit

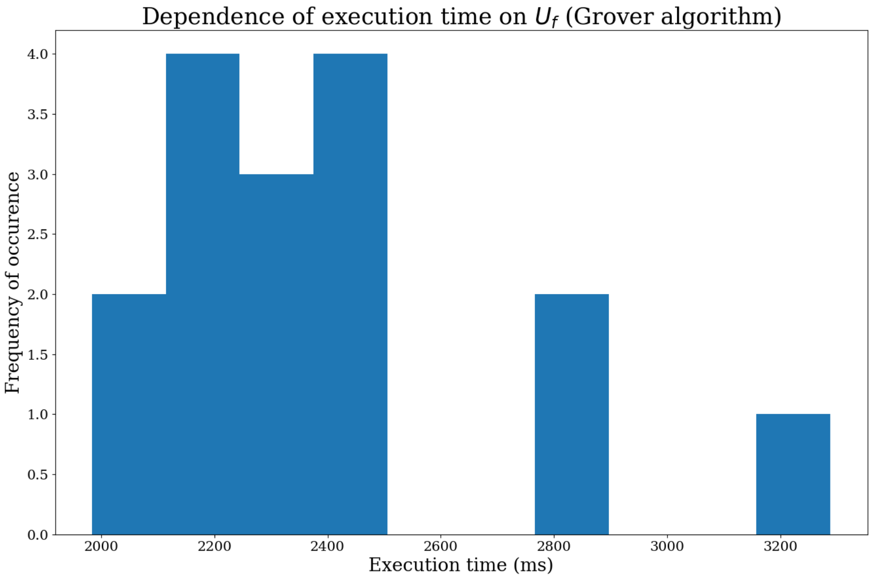


Figure 8: Execution time for all 16 possible U_f when $n = 4$ and $a = 1$.

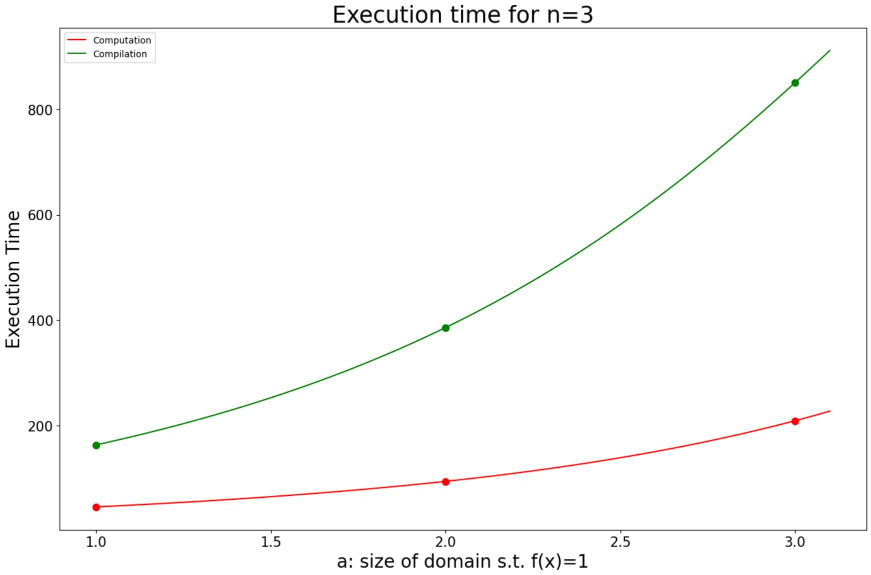
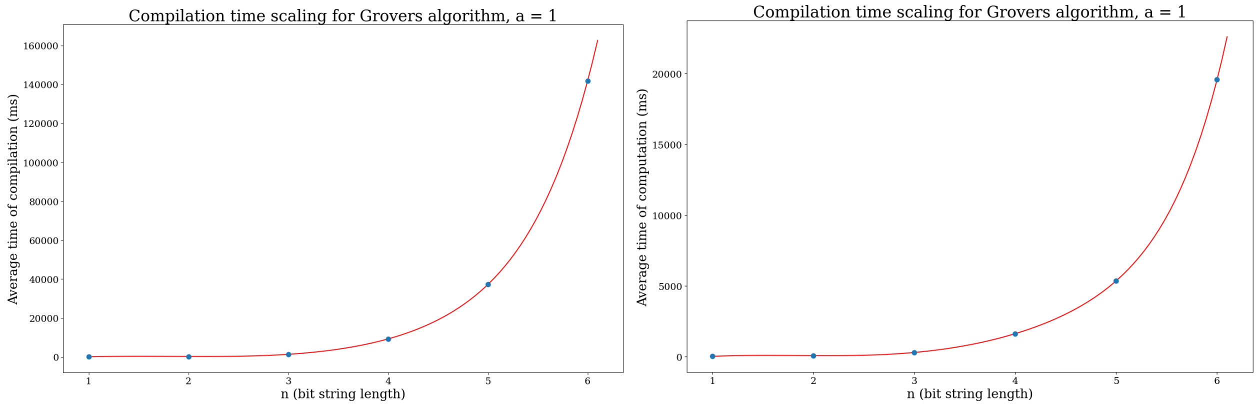


Figure 9: Average Execution time for U_f when $n = 3$ and $a = 1 \rightarrow 3$.

simulation grows exponentially with n. As can be seen in figures 10a and 10b the execution and compilation times both grow exponentially, with $n = 6$ requiring about 3 minutes of compilation time and about half a minute of execution time. We decided to limit our analysis to $n \leq 6$, since we observed that $n = 7$ making it unfeasible to average over multiple runs of the algorithm. Our observations are in line with the fact that the matrices grow larger exponentially as we increase n.



(a) Compilation time scaling for Grover's algorithm.

(b) Execution time scaling for Grover's algorithm.

Figure 10: Grovers algorithm: Benchmarking.

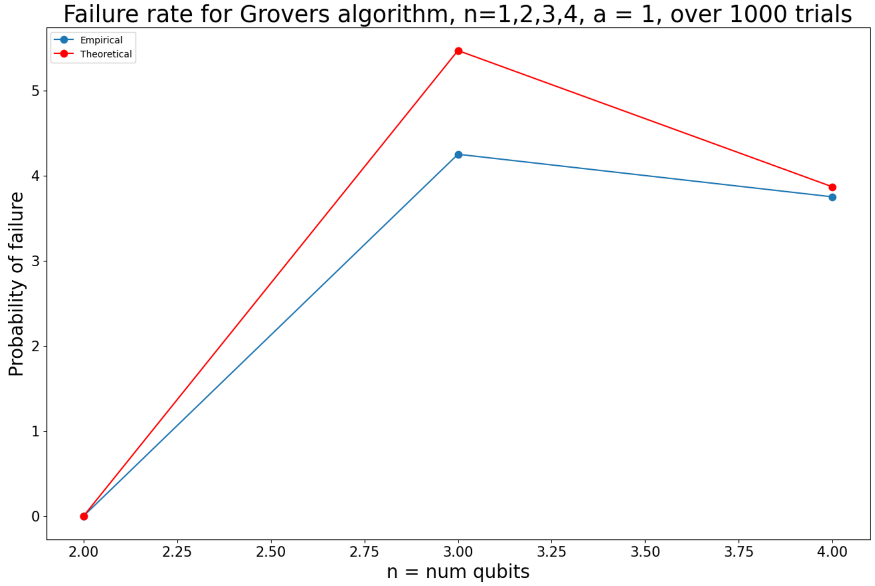


Figure 11: Theoretical vs. numerical probability of failure for Grovers algorithm.

2 Outline: The Language & Documentation

Most of the assignment prompts have been answered in the sections above. We also summarize our approach and observations below.

1. Design and evaluation
   (a) Present the design of how you implemented the black-box function U_f. Assess how visually neat and easy to read it is.
   $\rightarrow$ In order to create a readable and clean code, for each of the algorithms, we have a separate `.py` module. Within each module, based on functionality, we split the procedeure into multiple small and readable functions, which can be run for a lower level implementation. We also have a provision for a higher level implementation, where the only input the user needs to provide is the function f. We provide docstrings and useful code comments in order to make the code easier to understand.
   As for the implementation of the oracle, for each algorithm, given a function f, we compute the corresponding matrix U_f, and add the gate object corresponding to it to the circuit in pyQuil. For more details, we refer the reader to the relevant sections above.

(b) Present the design for how you prevent the user of U_f from accessing the implementation of U_f. Assess how well you succeeded.
→ We interpreted this instruction in two ways, and provide comments on the two interpretations below:

i. If Alice provided Bob with an oracle U_f, and Bob uses a pyQuil script to simulate a quantum circuit, is it possible for Alice to ensure that Bob cannot easily determine the matrix elements of U_f, which would be sufficient for Bob to solve the problem classically?
It is our understanding that currently, it is always possible for Bob to access the internal working of U_f in pyQuil. The reason for this is that in order for Bob to implement U_f in his circuit, he needs the `DefGate` and `DefGate.get_constructor()` objects corresponding to U_f. Once Bob adds these objects to his `Program`, the python `print` command prints the matrix elements of U_f. Alternatively, Bob can use the `DefGate.matrix()` command to view the matrix corresponding to U_f. Once Bob accesses the U_f matrix elements, he can classically reverese engineer to get f and solve the problem classically. In conclusion, the internal workings of U_f cannot be hidden from a pyQuil user.

ii. If Bob provides Alice (the user) with a pyQuil script which takes as input the function f, can Bob ensure that Alice will not be able to access the implementation of U_f? (This was the interpretation suggested by an answer on Piazza).
Although it is possible for Alice to view the script itself in order to figure out how U_f has been implemented, if Bob provides Alice with only a way for a high-level implementation of the quantum algorithm, Alice will simply input f, and receive the final output of the quantum circuit at the end, without interacting with pyQuil at all. We have implemented this approach in our programs, with the user having access to a high-level function, e.g. `run_DJ()`, which only needs f as the input, and provides 0 or 1 as the output.
We also note however, that the user can use `inspect.getsource` in Python to inspect the implementation of U_f in our codes.

(c) Present the design of how you parameterized the solution in n.
→ Each value of n requires a different circuit. Hence our programs create the circuits dynamically. Whenever only f is provided, the matrix U_f is created and the circuit is constructed, compiled and executed. In order to add a variable number of gates (which depend on n), we used `for` and `while` loops.

(d) Discuss the number of lines and percentage of code that your two programs share. Assess how well you succeeded in reusing code from one program to the next.
→ DJ and BV algorithms share the same circuit, and as a consequence many of the functions we used in their implementations were identical. However, Simon's algorithm and Grover's algorithm involved qualitatively different circuits, and dealt with distinct problems, resulting in less code sharing with other algorithms. Simon's algorithm required a classical solver which was unique to it. The testing generation functions were different as well. Individual pieces of the circuit shared code though, such as adding Hadamard gates in a for loop and applying U_f. Creating U_f shared some code as well but there were some differences because of the varity of n. PyQuil allowed a good portion of the circuit code to be shared.

(e) Discuss your effort to test the two programs and present results from the testing. Discuss whether different cases of U_f lead to different execution times.
→ Different cases of U_f lead to different execution times. We observed that the closer the oracle matrix U_f is to the identity matrix, the faster the execution. For histograms for each algorithm, please see the relevant sections above.
Since Grover's search algorithm has a possibility of failure, we implemented our code 100 times for a variety of values of n and a, and observed a close match between the success rate of our implementation, and the success rate expected theoretically. We also observed (and provided an explanation for) an increase in execution time for a fixed n, as a was increased for small values of a.

(f) What is your experience with scalability as n grows? Present a diagram that maps n to execution time.
→ We discussed the scalability for each algorithm and observed an exponential increase in execution time with the number of qubits. For values of n larger than 10, the server would crash with a heap space error. The largest successful run, which included compilation and run was for a circuit with 8 qubits. With 8 qubits though, it took many tries as the server would crash with out of memory. We have also provided plots for average execution time scaling for each of the four algorithms in their respective sections above.

2. PyQuil:
As a reflection on both homeworks on programming in PyQuil, address the following points.

(a) 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.
→ Aspects of pyQuil which were easy to learn:

i. Constructing gates: Construction of custom gates was straightforward, since it only involved providing the numpy array defining the unitary matrix.
ii. Constructing circuits: Construction of circuits is straightforward, all one has to do is add it to the program in sequential order.
iii. Creating a variety of quantum computer objects: It is very convenient in pyQuil to simulate quantum computers with a variety of qubit topologies.

→ Aspects of pyQuil which were difficult to learn:

i. Declaring memory: It was unintuitive to have to create read only memory and that had to be written to every single time you measure to record the output.
ii. The difference and benefits between `run_and_measure()` and compiling separately and measuring was intially unclear.
iii. Errors popping up while starting `quilc` and `qvm`: Sometimes, upon executing a python script, we would get an error saying that a particular port was not functional, and debugging was difficult.

(b) List three aspects of quantum programming in PyQuil that the language supports well and list three aspects that PyQuil supports poorly.
→ Good support:

i. Measuring was handled very well. It offered plenty of flexibility and it didn't take a while to figure out. While learning how to use the system, `run_and_measure(p, trials)` gives a dictionary of the measurement of trials. With finer grained control, the user can add a measure command to the program and read from the results each time the program is run.
ii. The process of adding gates to a circuit was supported very well. It was intuitive to use += to add to a program and use the index of the qubit.
iii. The interface between the quantum computer and the classical program is pretty seamless. Wrapping a program in a while loop calling run to repeatedly measure matches the thought process perfectly.

Poor support:

i. There wasn't much support for the obfuscation or hiding of the implementation of U_f as you have to pass in a gate definition object to the function that implements the quantum algorithm in question.
ii. In order to simulate a quantum circuit, one can either use `pyquil.api.local_forest_runtime()`, or first start the `qvm` and `quilc` servers in the command line before running a pyQuil program. In our experience, the former method sometimes failed to work, giving errors such as 'ConnectionError's, which were hard to debug. The latter method was more reliable.
iii. Although this quantum virtual machine is a quantum simulator, running it did not seem to give us an indication of how the scaling of execution time or memory would occur on an actual quantum computer.

(c) Which feature would you like PyQuil to support to make the quantum programming easier?

i. The `run_and_measure()` command returns the measured values for all the qubits in the simulator, even when many of the qubits may not have been used by the program. It would be more efficient to allow the users to choose to measure only those qubits which are used by the `Program`.
ii. Improving the implementation of `pyquil.api.local_forest_runtime()`, so that commonly experienced errors are avoided.

(d) List three positives and three negatives of the documentation of PyQuil.
→ Positives:

i. The documentation was fairly detailed. It provided examples for anything we needed. It showed examples of defining gates in different ways which helped give ideas on how to program.
ii. The documentation was aesthetically pleasing and followed the standard convention of python modules documentation.
iii. The documentation provides an introduction to quantum computing. All the basics are really well explained and serve as a good review before starting any programs.

Negatives:

i. Debugging, especially connection issues related to the servers could be a topic worth having more documentation about.
ii. PyQuil often has multiple ways for a variety of tasks. For example, we can run `run_and_measure()` or explicitly compile the program, and run it. The documentation doesn't talk about it trade-offs of using one or the other, although in our experience, the former is faster than the latter in certain cases.
iii. Information about the same topic was spread over multiple links e.g. ”The Quantum Computer” & ”New in Forest 2 - QuantumComputer” & ”API REFERENCE: Quantum Computer”. The reference

being separate makes sense but perhaps the other two can be combined into a single more thorough resource.

iv. The documentation states that the commands `man qvm` and `man quilc` should open the manuals for qvm and quilc respectively, when entered into the command line. However, this did not work when tried on a Windows 10 system.

(e) 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.
→ Dictionary

i. **Declare**: Allocating memory for reading.
ii. **Instruction**: Gate.
iii. **Program**: Circuit
iv. **Dagger**: Conjugate transpose.
v. **Readout Results**: Results of measurement.
vi. **QVM**: Quantum Virtual Machine. The software tool that runs quantum instruction language generated by the QC on a virtual machine.
vii. **QC**: Quantum Compiler. The software tool that compiles the python code into quantum instruction language instruction.
viii. **Quil**: Quantum Instruction Language.

(f) 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?
→ We interpreted this question in two ways, and discuss the two below:

i. Type checking done by PyQuil itself: Python in general does type checking at run time. This tends to be a problem with applications like quantum computing that run for a while. We tested a variety of PyQuil functions to see the extent of type checking defined in PyQuil, and observed a robust type checking. For example, trying to create a gate from a matrix of strings or passing in a string to a gate that needs the index of the qubit catches the error and reports it immediately.
ii. Type checking implemented by us (the interpretation suggested by an instructor answer on Piazza): We included type checking in the functions we defined, in order to catch type errors which the user is most likely to commit. For example, in DJ and BV algorithms, we ensure that f is a numpy array, has an acceptable size, and includes only zeros and ones. Similarly, we check that any PyQuil objects that were passed between function calls such as gate definitions or executables were type checked to catch errors immediately.