COM SCI 239 Algorithms in PyQuil Project Report

Design and evaluation

Deutsch-Jozsa (deutsch_jozsa.py)

For Deutsch-Jozsa, implementing the black-box U_f proved to be tricky. Since U_f |x> |b> = |x> |b + f(x)>, where x represents the n input qubits and b is the helper qubit, we initially thought that the matrix representing U_f could be constructed as the simple tensor product of n identity gates and either the identity gate, if f(x) == 0, to preserve b, or the Pauli X gate, if f(x) == 1, to invert b. However, we quickly realized this was not the case since whether the helper qubit is preserved or flipped varies based on x.

Hence, to figure out a general form for U_f, we worked through the case of f(x) = XOR(x) for 2 input qubits and 1 helper qubit. By doing so, we devised an algorithm for constructing U_f. U_f is a 2^(n + 1) by 2^(n + 1) NumPy array (n + 1 because n input qubits and 1 helper qubit), and is initially all zeros. Using a loop, we iterate through each possible n-bit input x, and if f(x) == 0, then we preserve the state of the helper qubit b, otherwise, we invert the state of the helper qubit. In essence, the “diagonal” of U_f consists of 2 by 2 blocks that are either the identity or Pauli X; the i-th diagonal block corresponds to the i-th possible n-bit input x_i to f (going from {0}^n to {1}^n), and if f(x_i) == 0 we insert the identity into the i-th diagonal block, else if f(x_i) == 1, we insert Pauli X into the i-th diagonal block.

The implementation of U_f is pretty neat and easy to read; it clearly follows the logic described above. If we opened up the black-box U_f, it would also be neat and easy to read since all the non-zero entries in the matrix representing the unitary gate are either along or just off the diagonal. In our program, we call a function get_U_f, which takes as input the function f and n and executes the algorithm discussed to return the matrix representing U_f. We then pass this matrix into dj_program (the function that builds the Deutsch-Jozsa quantum circuit), which actually defines the gate U_f using the matrix and specifies it to be applied to the input and helper qubits, as shown below.

```
U_f_def = DefGate("U_f", U_f)
U_f_GATE = U_f_def.get_constructor()
p += U_f_def
p += U_f_GATE(*range(n + 1))
```

With regards to the user accessing the implementation of U_f, the user writes the classical implementation of f in func.py and inputs the name of f as a command line argument when running deutsch_jozsa.py. U_f is constructed dynamically during runtime using get_U_f, so there is no way to access the entries of the matrix for U_f. However, because get_U_f is not

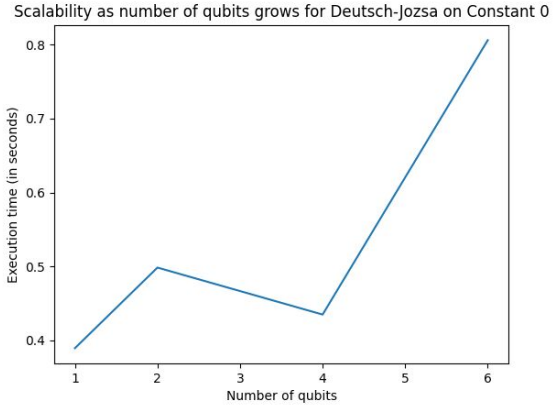
encapsulated in a class, and furthermore, there is no private access modifier on get_U_f, the user could create their own script to call get_U_f and inspect the internals of the matrix returned.

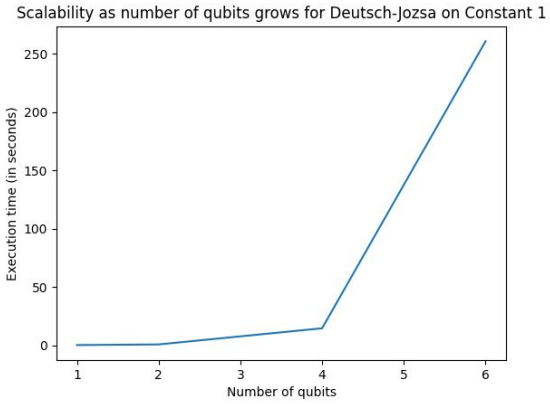
The parameter n is passed into the program as a command line argument. During runtime, n is passed to get_U_f to dynamically construct the matrix for U_f with the appropriate dimensions (i.e. 2^(n + 1) by 2^(n + 1)). n is also passed to dj_program, where it governs the number of qubits to which Hadamard is applied at the beginning of the circuit and at the end and on which qubits U_f operates.

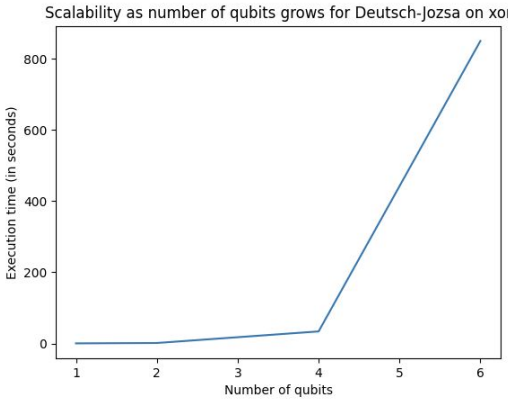
We tested our Deutsch-Jozsa implementation on the functions constant 0, constant 1, XOR, and XNOR (where the former two are constant and the latter two are balanced). Since the PyQuil simulator is a noisy (i.e. not ideal) quantum simulator by default, we set up our test driver to accept the number of trials to run as a command line argument. For each test, we print out the results; we use nice formatting to clearly label the test to which the trials correspond and the total execution time for the entire test (including all the trials), and subsequently, for each trial, present 1) the actual measurements of the n input qubits and 2) the interpretation of the result (i.e. “Constant” if all the qubits are measured to be zero and “Balanced” otherwise).

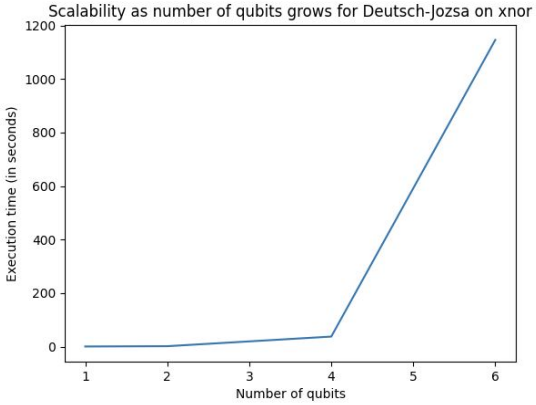
Different cases of U_f definitely lead to different execution times. As stated earlier, for each test, we compute the execution time including all the trials. The execution time does not include the construction of the black-box U_f (since Deutsch-Jozsa assumes U_f is available), but it does include the compilation and running of the quantum circuit, as well as the measurements of the qubits and their interpretation (since the classical logic for interpreting the measurements is part of the Deutsch-Jozsa algorithm). We noticed that with 4 qubits, constant 0 (0.451 sec) led to a much faster execution time than constant 1 (20.471 sec), and XOR (29.889 sec) and XNOR (37.387 sec) were noticeably slower than both. This is likely due to the fact that the constant functions make it easier for the compiler to decompose U_f, for example, into simple single-qubit unitary gates and controlled-X gates (these two types of gates form a universal set). This is because U_f for constant 0 is just the identity and U_f for constant 1 is just $I^{\otimes n} \otimes X$ (i.e. CCNOT), while the balanced functions are more complex since they have an equal amount of and alternating on-diagonal and off-diagonal non-zero entries. Since U_f for constant 0 is simply the identity, it obviously makes it the easiest for the compiler to decompose U_f. The disparity in the execution time between the constant functions and the balanced functions seems to grow exponentially as the number of qubits increases.

With regards to the scalability of n, for each test U_f, the execution time appears to be exponential in number of qubits; this makes sense as the compiler will have a harder time decomposing larger matrices. The plots below show the execution times for the constant 0, constant 1, XOR, and XNOR functions vs. the number of qubits. Unfortunately, we could not successfully generate results for a larger number of qubits because we repeatedly received an RPC error. As a note, because communication with the PyQuil compiler and VM was performed via sockets, execution times were highly variable. We attribute the dip at 4 qubits in the constant 0 plot below to this variability.



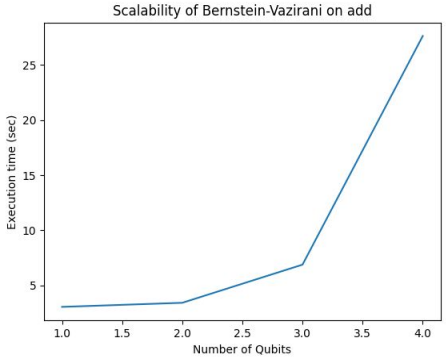


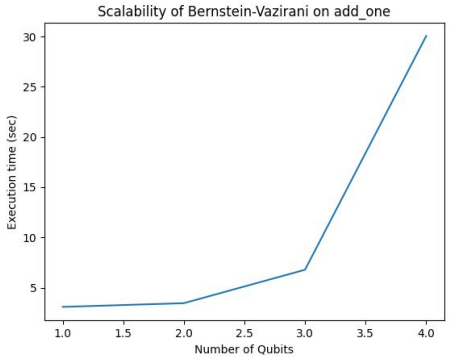


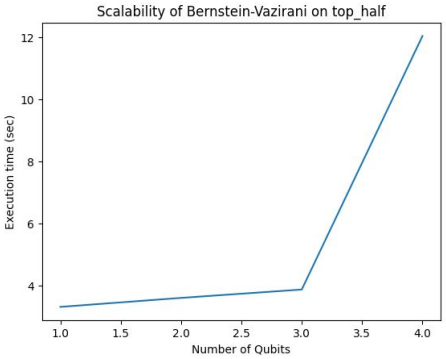


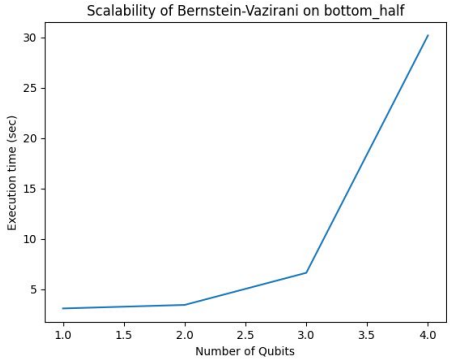
Bernstein-Vazirani (bernstein_vazirani.py)

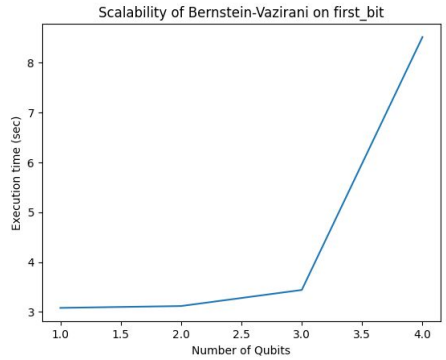
For Bernstein-Vazirani, the logic and implementation of U_f is identical to that for Deutsch-Jozsa, i.e., U_f is a $2^{n+1}x2^{n+1}$ complex unitary matrix that can be divided into a $2^n x 2^n$ grid, where each box in the grid is a 2x2 matrix, such that the boxes are the zero matrix if they are not on the diagonal, and the k^{th} box on the diagonal is $X^{f(k)}$. All the challenges faced were identical to those for Deutsch-Jozsa above. Here are the results from the scalability testing:

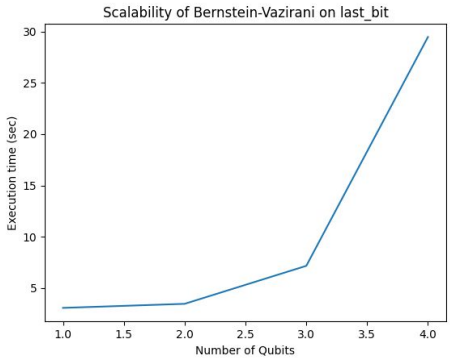












Definitions of functions in above graphs:

- add: returns the sum of all bits in the bit string (mod 2)
- add_one: returns the sum of all bits in the bit string plus 1 (mod 2)
- top_half: returns the sum of the first half, rounded down, of the bit string (mod 2)
- bottom_half: returns the sum of the second half, rounded up, of the bit string (mod 2)
- first_bit: returns the value of the first bit in the bit string
- last_bit: returns the value of the last bit in the bit string

As can be seen above, the execution time appeared to grow exponentially with the number of qubits. Similar to Deutsch-Jozsa, this probably has to do with the fact that the decomposition of the custom matrices into quantum gates becomes exponentially more challenging as the dimensions of the matrices and number of off-diagonal non-zero entries increase. Furthermore,

we can see that top_half and first_bit seem to have significantly lower execution times than the rest of the functions. Since these act on the ‘early’ qubits, this could indicate that operations on later qubits take longer to execute than those on earlier qubits.

Furthermore, during our testing, we found that the algorithm occasionally returned wrong values for a, usually when operating on 4+ qubits. After doing some digging, we discovered that this is because most QVMs in PyQuil are, by default, noisy, which means that they have some chance of giving erroneous results, which increases significantly when the circuit is loaded. To combat this issue, we increased the number of qubits available to 16 and set noisy=False when constructing the QVM.

Grover (grover.py)

Implementing Z_f and Z_0 was relatively straightforward. Z_f is just a 2^n by 2^n diagonal matrix, where the i-th entry in the diagonal corresponds to the 2^i-th possible n-bit input x and is equal to (-1)^f(x). In this way, |x> gets mapped to (-1)^f(x) |x>. We just passed a list of all 2^n diagonal entries to np.diag to obtain the appropriate matrix for Z_f as a NumPy array. Similarly, Z_0 is just a 2^n by 2^n diagonal matrix, where the i-th entry in the diagonal corresponds to the 2^i-th possible n-bit input x and is equal to -1 if x == {0}^n and 1 otherwise. In this way, |x> gets mapped to -|x> if x == {0}^n and |x> otherwise. To efficiently implement this, we used np.eye to create a 2^n by 2^n identity NumPy array and set the top-leftmost entry to -1.

The implementations of Z_f and Z_0 are pretty neat and easy to read; they clearly follow the logic described above. If we opened up the black-boxes Z_f and Z_0, it would also be neat and easy to read since all the non-zero entries in the matrix representing the unitary gates are along the diagonal. In our program, we call the function get_Z_f, which takes as input the function f and n and returns the matrix representing Z_f, and get_Z_0, which takes as input n and returns the matrix representing Z_0,. We then pass this matrix into grover_program (the function that builds the Grover quantum circuit), which actually defines the gates Z_f and Z_0 using the matrices and specifies them to be applied to the input qubits, as shown below.

```
# define the Z_f gate based on the unitary matrix returned by get_Z_f
Z_f_def = DefGate("Z_f", Z_f)
Z_f_GATE = Z_f_def.get_constructor()
p += Z_f_def

# define the Z_0 gate based on the unitary matrix returned by get_Z_0
Z_0_def = DefGate("Z_0", Z_0)
Z_0_GATE = Z_0_def.get_constructor()
p += Z_0_def
```

With regards to the user accessing the implementations of Z_f and Z_0, the user writes the classical implementation of f in func.py and inputs the name of f as a command line argument

when running grover.py. Z_f and Z_0 are constructed dynamically during runtime using get_Z_f and get_Z_0, so there is no way to access the entries of the matrix for Z_f or Z_0. However, because get_Z_f and get_Z_0 are not encapsulated in a class, and furthermore, there is no private access modifier on either, the user could create their own script to call get_Z_f or get_Z_0 and inspect the internals of the matrices returned.

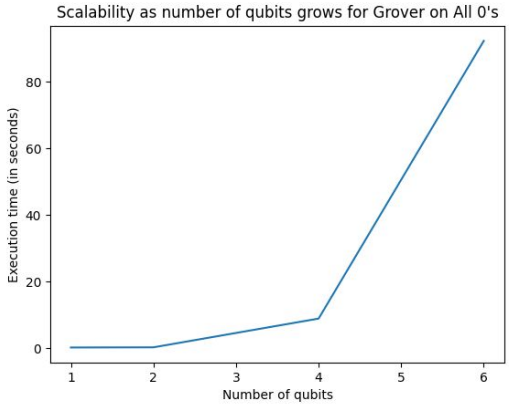
The parameter n is passed into the program as a command line argument. Then, during runtime, n is passed to get_Z_f and get_Z_0 to dynamically construct the matrices for Z_f and Z_0 with the appropriate dimensions (i.e. 2^n by 2^n). n is also passed to grover_program, where it governs the number of qubits to which Hadamard is applied at the beginning of the circuit and on which qubits and the number of times G operates.

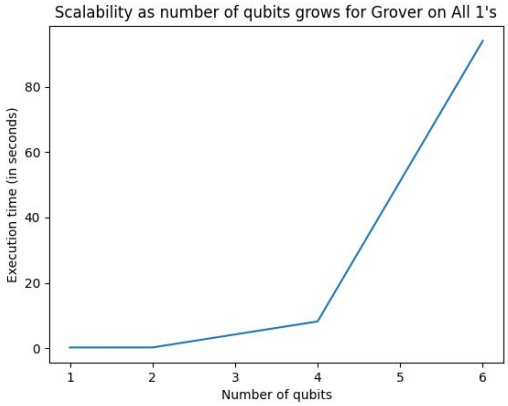
We tested our Grover implementation on the functions constant 0, All 0's (evaluates to 1 only when all input bits are 0), All 1's (evaluates to 1 only when all input bits are 1), and XNOR. Since Grover is not a deterministic algorithm and the PyQuil simulator is a noisy (i.e. not ideal) quantum simulator, we set up our test driver to accept the number of trials to run as a command line argument. For each test, we print out the results; we use nice formatting to clearly label the test to which the trials correspond and the total execution time for the entire test (including all the trials), and subsequently, for each trial, present 1) the actual measurements of the n input qubits and 2) what f evaluates to for the measurements (0 or 1).

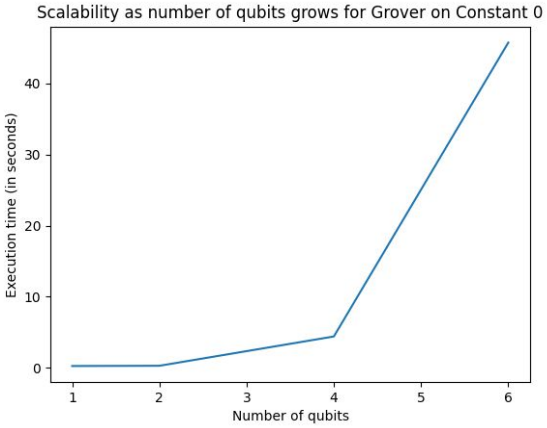
Different cases of Z_f definitely lead to different execution times. As stated earlier, for each test, we compute the execution time including all the trials. This execution time does not include the construction of the black-boxes Z_f and Z_0 (since Grover assumes Z_f and Z_0 are available), but it does include the compilation and running of the quantum circuit, as well as the measurements of the qubits and their interpretation (since the classical logic for interpreting the measurements is part of Grover's algorithm). We noticed that with 4 qubits, constant 0 (4.334 sec) led to a faster execution time than All 0's (11.209 sec) and All 1's (10.764 sec), and XNOR (16.276 sec) was slightly slower than the rest. This is likely due to the fact that the constant function makes it easy for the compiler to decompose Z_f into simpler gates, because Z_f for constant 0 is just the identity, while the other functions admit a Z_f that is more complex due to having -1 entries along their diagonal. The disparity in the execution time between the functions seems to grow exponentially as the number of qubits increases. Notably, the disparity is not due to the construction of Z_0 or the floor(pi/4 * sqrt(2^n)) number of iterations Grover requires because these are consistent across all the test functions in the 4-qubit case. Furthermore, compared to Deutsch-Jozsa, for the same number of qubits, Grover has much faster execution times since all the gates involved have diagonal matrices, which are much easier for the compiler to decompose than matrices with off-diagonal non-zero entries.

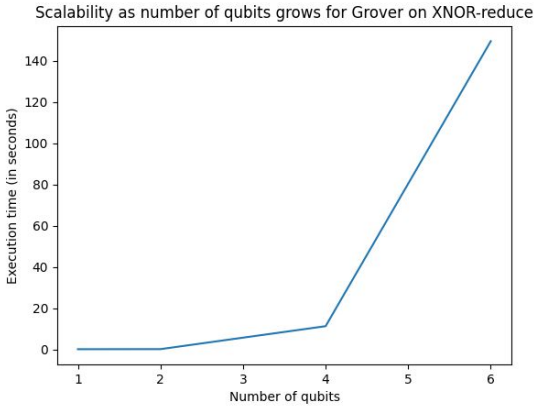
With regards to the scalability of n, for each test function, the execution time appears to be exponential in number of qubits; this makes sense as the compiler will have a harder time decomposing larger matrices. The plots below show the execution time for all the test functions mentioned above vs. the number of qubits. As a note, because communication with the PyQuil

compiler and VM was performed via sockets, execution times were highly variable between runs.









Simon (simon.py)

Algorithm Structure

Implementing the quantum part of Simon's algorithm in PyQuil was pretty straightforward, with the exception of determining U_f:

1. Apply Hadamard to the n qubits (n helper qubits are initialized to 0)
2. Create U_f gate and apply to all (2*n) qubits
3. Apply Hadamard to n qubits
4. Measure first n qubits

U_f Implementation

Determining U_f proved tricky initially, as we tried to follow a method similar to that of Deutsch-Jozsa and Bernstein-Vazarani, except with a U_f of size 2*n by 2*n rather than n+1 by

n+1. However, we soon realized we had to generate 2^2*n permutations of possible input strings and not 2^n. We determined U_f by first computing f(x_a) + b for every possible input x, where x_a is the first n bits and b is the second n bits of the input x. Note that the first n bits are the same for the input and output, and the second n bits are determined by the mod2 sum of f(x_a) and b. Iterating through the list of all possible input values, we mapped each input and output to the index value of that iteration using two dictionaries. We initialized every element in U_f to 0, and then used the input and output pairs to determine which elements were one. The indices mapped to the input values represented the rows of U_f, and the indices mapped to the output values represented the columns. If a given input mapped to index ‘a’ is the output mapped to index ‘b’, then the element U_f[a][b] is 1. It was initially confusing to determine whether the inputs represented the column or row indices, but we worked out the logic by starting with n = 1 where U_f is only a 4 by 4 matrix.

The implementation of U_f is pretty neat and easy to read; it clearly follows the logic described above. However, if we opened up the black-box U_f, it would not be neat and easy to read since there are many non-zero entries in the matrix representing the unitary gate that are not along or just off the diagonal. The specification of f, the design for how the user is prevented from accessing U_f (and the success of this design), and how the solution is parametrized in n are identical to those of Deutsch-Jozsa.

Solving Equations (Constraints) to determine s:

We looped through the quantum circuit 4*m times, where m is an argument passed in by the user, to set up a system of n-1 linearly independent equations (n including the y consisting of all 0s) to solve for s. In the case we reach this requirement before 4*m iterations, which is often the case, we break from the loop and pass our equations to a separate function for solving the constraints. We determined the number of equations which are linearly independent by setting all y’s as rows in a matrix, and then used a numpy function to find the rank of the matrix. In our constraint solving function, we determined s using the idea that the dot product of y and s is always 0. This means that for any index a in y where y[a] is 1, s[a] cannot be 1. We initialized all elements of s to be 1 and changed an element to 0 whenever a y contained a 1 at that same index. This way, we determined the s where the dot product of s and y is 0 for all y’s passed in to the constraint solving function.

Two-to-One Function:

We set s to be a list of n elements alternating between 1 and 0, where the most significant bit is always 1. If the most significant bit of x is 0, the function returns x, otherwise the function returns x XOR s.

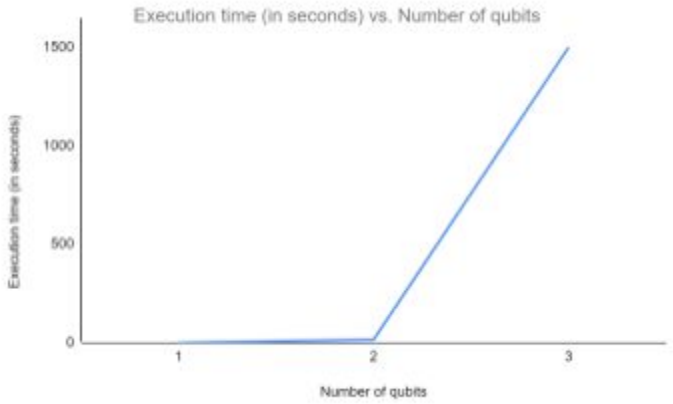
n = 3 example:

x	f(x)

000	000
010	010
001	001
011	011
100	001
110	011
101	000
111	010

s = 101

As n increases, so does the execution time of the algorithm significantly, and our execution time for n = 3 was much higher than expected. This probably has to do with the fact that the decomposition of the custom matrices into quantum gates becomes a lot more challenging as the dimensions of the matrices and number of non-diagonal non-zero entries increase. Compared to the other algorithms, U_f for Simon has a lot more non-diagonal non-zero entries and each dimension of the matrix is relatively twice as large as what would be required for the other algorithms, which explains why Simon's execution time is higher than that of the other algorithms even with only n = 3 qubits. In our future implementations of Simon's, we will try to optimize the algorithm more in areas possible.



Overlap and Reuse of Code

There is significant overlap between the code for Deutsch-Jozsa and the code for Bernstein-Vazirani, since the structure of the quantum portion for both algorithms is identical — the difference is in the interpretation of the qubit measurements in the classical portion. Despite

the overlap, we did not modularize the shared quantum circuit for the reuse in both programs because these are the only two programs that share a significant amount of code, so it seemed excessive to do so. However, for the purposes of testing, we placed the test functions we used to evaluate the accuracy of the implementation of the algorithms in func.py, the same file in which the user defines custom functions to run the quantum programs on, since these test functions can be shared amongst the test drivers for the programs (e.g. the constant 0 function is a good test function for both Deutsch-Jozsa and Grover). Even though our test driver for each program made use of these shared test functions, the interpretation of the qubit measurement results being different for each driver made it impossible to reuse code between the drivers.

Specifically, the shared code sections between deutsch_jozsa.py and bernstein_vazirani.py were the construction of U_f, given f(x), the entirety of the quantum circuit, or 'program' as well as the initialization of the quantum virtual machine (qvm). In the ~100 lines of code in deutsch_jozsa.py and bernstein_vazirani.py, around 70-80 of them were shared, i.e., around 70-80% of the code for Deutsch-Jozsa and Bernstein-Vazirani was shared.

README

Open the file "func.py," which is in the same directory as the quantum program files; it already comes with a lot of test functions. In this file, define the function(s) you want to input into any of the quantum algorithms. The function must take only one parameter, a list of bits, and it should return a value that aligns with the assumptions made by the algorithm you plan on using (e.g. for Grover, you should only return a single bit). Furthemore, the function may NOT make any assumptions about the length of the list of bits. This is an example:

```
# constant 1 function
def one(x):
        return 1
```

All quantum programs must be run from the command line, and they all take the same arguments. There are two options:

1) python3 quantum_program.py **fn_name** **n** **trials**

   where **fn_name** is the name of the function you want to input (exactly the same as in the definition in func.py), **n** is the number of qubits, and **trials** is the number of times you want to run the quantum circuit and measure the qubits. It is not required to manually start the PyQuil compiler and VM. This is an example for Deutsch-Jozsa:

   python3 deutsch_jozsa.py one 4 10

   Each script runs the quantum circuit and measures the qubits for the specified function, number of qubits, and number of trials. It then prints out the results; it clearly labels the

test to which the trials correspond and the total execution time for the entire test (including all the trials), and subsequently, for each trial, presents 1) the actual measurements of the n input qubits and 2) the interpretation of the measurements.

For simon.py, the trials option is used as m, which determines the number of iterations of the circuit to increase confidence in the results.

2) python3 quantum_program.py **fn_name --graph n_1 n_2 n_3…**

Alternatively, each script also runs the quantum circuit for the provided function for the specified numbers of qubits and saves a plot of the execution time vs. the number of qubits to the same directory in which the quantum program files are. This is an example for Deutsch-Jozsa:

python3 deutsch_jozsa.py one --graph 1 2 4

We chose not to apply the --graph option to simon.py, since the trials option is used as m: a confidence measure.

PyQuil Reflection

Some aspects of PyQuil which made it easy to learn include the += operator for building a program, the Python libraries used, through which the syntax of PyQuil was familiar, and the pre-implemented common gates such as Hadamard and Pauli X, which were easy to apply to our circuit. Running and measuring the circuit was also a single function and therefore was straightforward to use. On the other hand, we did struggle with some other aspects of PyQuil such as learning how to define custom gates. It was unintuitive for us to add a definition of a gate to the program and then add the actual gate instance. Furthermore, the results dictionary was keyed by qubits, but we felt it would've been more useful if keyed by trial. This is because a string of measurements for all the qubits for each trial would have made presenting the results of the algorithms easier than needing to iterate through the trials and collect the corresponding measurement of each qubit for that trial. Another idea is that the measurement results could be returned as a 2D NumPy array, where the rows represent the qubits and the columns represent the measurements for the trials, which would allow for equally easy access to the measurements of all the qubits for a trial AND all the measurements for a single qubit. Lastly, we frequently ran into socket issues. For instance, we kept receiving a runtime warning that the specified port for the qvm and quic servers are already in use. We also frequently received RPC errors, which interrupted the execution of our programs, typically with a large number of qubits, and led to frustration.

Some features of quantum programming that we felt PyQuil supports well are individual qubit operations (just need to pass in 0 to the gate instance), creation of custom unitary gates, and the ability to apply gate modifiers like DAGGER, CONTROLLED, and FORKED to gates, which

reduces the the number of custom unitary gates one needs to define from scratch. Some features that PyQuil doesn't support as well are program visualization, as only the names of the gates used and raw matrix entries are printed (it's definitely not anywhere near as helpful or intuitive as Cirq's pretty printing of programs or Qiskit's circuit composer). A more thorough visualization of the circuit would've been useful when debugging the implementation of the algorithms. We felt multiple-qubit operations could've been handled better too. Instead of unpacking a range of qubits using *, we wish we could pass in a list of qubits as a single argument. It would also be useful to have easier methods to implement common operations, such as initializing qubits to be non-zero, which currently has to be done by applying the Pauli X gate in a for loop.

Going off of the aspects of quantum programming that PyQuil supports poorly, a feature that we would like PyQuil to support to make quantum programming easier is better graphical visualization of quantum circuits. As we mentioned previously, when one prints a Program, only the names of the gates used and raw matrix entries are printed. A more thorough visualization of the circuit would be useful when debugging the implementation of the algorithms. In contrast, Cirq's pretty printing of programs and Qiskit's circuit composer are intuitive and helpful when constructing quantum circuits.

The documentation for PyQuil was very easy to follow. The installation steps were outlined clearly for each operating system. There were also a lot of examples used throughout, especially for the definition and use of custom unitary gates, which were useful reference points when first writing our programs. Furthermore, the Exercises section really helps new PyQuil learners walk through the creation of a variety of programs, from Controlled Gates to even Grover's algorithm, which gives them a sense of the expressive power of the library. A more specific documentation detail we felt could have been more visible was the section on preventing compiler client timeout; initially, the network connection with the compiler would time out while we were executing our programs for a large number of qubits, and we had to dig through online forums to figure out how to increase the compiler client timeout: `qc.compiler.client.timeout = 10000`. For the simulator, the documentation should indicate a clear option for "ideal" mode versus "noisy" mode that would make debugging easier. Noisy quantum simulation made it unclear if the issue was with the program. For example, with "noisy" mode, sometimes we received non-deterministic results for Deutsch-Jozsa, which is supposed to run deterministically. The PyQuil documentation could also have been slightly more quantum novice-friendly. For example, although the list of pre-implemented common gates is comprehensive, their descriptions are only useful to individuals who have a decent level of experience and exposure to quantum programming theory. For example, the description for "RX(angle, qubit)" is "Produces the RX gate.", which may require beginners to learn what the RX gate is using an external resource; instead, it would be nice to include a basic description of what operation the RX gate performs in the documentation.

Often when documentation is unclear we use question and answer forums to check if someone has had the same question in the past. However since the field of quantum programming has

generally fewer members than what we're used to in our other classes, it was interesting to see the decrease in the number of external resources for clarifying questions.

Dictionary of key concepts in quantum programming mapped to names used in PyQuil:
{circuit : program,
simulator : quantum virtual machine (qvm),
9-qubit simulator : 9q-square-qvm,
shots : trials,
total number of gates : gate_volume,
largest number of gates on a single qubit : gate_depth}

PyQuil does some level of type checking during runtime. For example, when passing a matrix from the classical side to the quantum side of the program to define a custom gate, PyQuil checks that the matrix is unitary and is of the right dimension. It does so by attempting to decompose an operator as native Quil instructions and checking if the resulting instructions match the original operator. If they do not, then "! ! ! Error: Matrices do not lie in the same projective class" will be thrown. However, when data is passed from the quantum to the classical side however, PyQuil assumes that the quantum output and measurements are compatible with the input required for measurement interpretation in the classical part of the program.