COM SCI 239

Quantum Programming: Algorithms in PyQuil

May 12, 2020

Author
Arjun Raghavan and Bryan Pan

1 Overview

2 Design

2.1 Implementing the black-box quantum oracle

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

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

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

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

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

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

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

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

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

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

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

2.1.1 Mathematical explanation

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

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

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

For $\alpha, \beta \in \{0,1\}^k$ such that $\alpha \neq \beta$, we have $|\alpha\rangle \neq |\beta\rangle \implies |\alpha\rangle\langle\alpha| \neq |\beta\rangle\langle\beta|$. Thus:

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

□

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

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

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

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

And so:

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

2.1.2 Python implementation

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

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

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

[1] Discussed in further detail in the README.

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

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

2.1.3 Readability

There were a couple of possible approaches we could have taken to iterating over all $xb \in \{0,1\}^{n+m}$, including nested `for` loops or generating the entire set of bit strings and iterating over that. We eventually settled on the approach seen in Code Block 1, which we felt was more elegant.

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

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

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

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

3 Evaluation

3.1 Shared code

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

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

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

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

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

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

[3]Also discussed in the README.

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

3.2 Testing

- 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.

3.3 Scalability

- What is your experience with scalability as n grows? Present a diagram that maps n to execution time.

4 Reflection on PyQuil

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

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

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

Some pros of the PyQuil documentation include:

- It provides a fine introduction to PyQuil and quantum programming.

Some cons of the documentation are:

- TODO

TODO: Answer the following:

- (IP) 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.
- List three aspects of quantum programming in PyQuil that the language supports well and list three aspects that PyQuil supports poorly.
- Which feature would you like PyQuil to support to make the quantum programming easier?
- (IP) List three positives and three negatives of the documentation of PyQuil.

- 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.
- 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?

References

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

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

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