

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:

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

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:

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

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:

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

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

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

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.**  $\sum_{q \in \{0,1\}^k} |q\rangle \langle q| = I_{2^k}$  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  $|q\rangle = [q_1 \ q_2 \ \dots \ q_{2^k}]^T$  where  $q_i = 1$  for some  $1 \leq i \leq 2^k$  and  $q_j = 0$  for all  $j \neq i$ . Then,  $|q\rangle \langle q|$  is a  $2^k \times 2^k$  matrix of the form  $[q_{ab}]$  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:

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

□

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:

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

And so:

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

### 2.1.2 Python implementation

As shown in [Code Block 1](#), [Equation 1](#) is straightforwardly implemented in Python thanks to the [numpy](#) library. 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. And example of this for Deutsch-Josza can be seen in [Code Block 2](#)

```

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

```

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

### 4 Reflection on PyQuil