Shor’s Algorithm, Bernstein-Vazirani, and Deutsch-Jozsa Algorithm: A Deep Dive

1st Maria Harrson
Dept. Computer Science (of Aff.)
Vanderbilt University
Nashville, TN
maria.harrison@vanderbilt.edu

Abstract—**Quantum Fourier Transform is the tool behind why quantum algorithms offer exponential speed ups compared to its classical counterparts. serving as a quantum subroutine, it transforms one quantum state into another; encoding the frequency spectrum of the original state. The Quantum Fourier Transform also plays a crucial role in Shor’s, Bernstein-Vazirani, and Deutsch-Jozsa algorithm, as these three all use an oracle and a quantum circuit. The purpose of this paper is to explore these three algorithms and implement them in Python.**

Keywords—hidden shift, integer factorization, quantum computing, modulus operator, quantum key distribution

I. Introduction

Quantum computers are machines that exploit the mechanics of quantum physics to solve large-scale linear algebra problems, that present challenges Classical computers are incapable of solving. With the advancements of quantum computing, there has been an increase interest in quantum cryptography over the past few years, leading to numerous discoveries including the various proposals for cryptographic approaches based on quantum information; Quantum Key Distribution(QKD) being the most prominent. Conversely, the advancement of quantum computing poses a threat to many classical cryptosystems, whose security is based on the difficulty of solving theoretical problems efficiently, like large prime factorization. Rivest-Shamir-Aldeman (RSA) is the most widely used public-key cryptosystem today. RSA relies on the
difficulty of factoring large prime integers in polynomial time for its security. This process is computationally exhaustive to break using a Classical computer, but is possible when using a quantum computer. Shor’s algorithm is the most represented example of a quantum algorithm that can efficiently solve the difficult large prime integer factorization problem that many public-key cryptosystems rely on security.

II. Quantum Fourier Transform

A. Quantum Fourier Transform

Quantum fourier transform (QFT) is a unitary operator that performs on the amplitude of a quantum state utilizing a discrete fourier transform. At a high level, QFT works by applying a series of Hadamard and Controlled Phase Shift Quantum Gates to a register of qubits, simulating a shift of basis states. Each qubit represents a binary digit on the input state, and each quantum gate alters the qubits amplitude and phase according to a specific formula. The result of applying a QFT is a quantum state that has the same number of qubits as the input state but its amplitude and phase correlate to a coefficient of the Fourier series of input states.
QFT uses Zφ gates and they act upon qubits as follows:

$$Z_\varphi|0\rangle = |0\rangle$$

$$Z_\varphi|1\rangle = e^{2\pi i/2\varphi}|1\rangle$$

$$Z_\varphi = \begin{bmatrix}1 & 0\\ 0 & e^{2\pi i/2\varphi}\end{bmatrix}$$

Although a |0⟩ state qubit does not change states, for a given positive number φ, the Zφ gate rotates the |1⟩ state qubit around the Z-axis at an angle of:

$$2\pi i/2\varphi$$

This can be further dissected by rotating |ψ⟩ around the Z-axis at an angle of:

$$2\pi i/2^\varphi J$$

Doing so applies Zφ on |ψ⟩ J times. The matrix for this now is as seen below:

$$Z_\psi^j = \begin{bmatrix}1 & 0\\ 0 & e^{(2\pi i/2^\varphi)j}\end{bmatrix}$$

When $N = 2n$ is the basis state, we can say that:

$$|\widetilde{x}\rangle = QFT|X\rangle$$

$$|\widetilde{x}\rangle = \frac{1}{N}\sum_{y=0}^{N-1} e^{\frac{2\pi ixy}{N}}|y\rangle$$

A single qubit case of a QFT is as follows:

$$|\widetilde{0}\rangle = \frac{1}{\sqrt{2}}\sum_{y=0}^{1} e^{\frac{2\pi i(0)y}{2}}|y$$

$$= \frac{1}{\sqrt{2}}\sum_{y=0}^{1}|y\rangle$$

$$= \frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)$$

$$|\widetilde{1}\rangle = \frac{1}{\sqrt{2}}\sum_{y=0}^{1} e^{\frac{2\pi i(1)y}{2}}|y\rangle$$

$$= \frac{1}{\sqrt{2}}\left(e^{\frac{2\pi i(0)}{2}}|0\rangle \dot{+} e^{\frac{2\pi i(0)}{2}}|1\rangle\right)$$

$$= \frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)$$

The QFT plays an important role in quantum algorithms as a subroutine, specifically Shor's Algorithm because it only needs n gates for the first qubit, $n - 1$ gates for the second, and so on, which sums to:

$$n + (n - 1) + ... + 1 = n(n + 1)/2$$

If you consider a $N/2$ SWAP gates implemented by 3x CNOT gates, the quantum circuit still only has a runtime of $O(n^2)$, which is still a significant speed up. QFT offers significant speed up when used as a subroutine in quantum algorithms, thus offering a more efficient solution to specific problems compared to classical algorithms.

III. Shor's Algorithm

In 1994, Peter Shor introduced a quantum algorithm that efficiently solves the Integer Factorization problem, notably referred to as Shor's Algorithm. Shor's discovery demonstrated those (cryptosystems) whose security relies on the difficulty of solving Integer Factorization or a Discrete Log Problem, can be impotent with a quantum computer. As quantum computing research furthers, it is prevalent to ensure secure communica- tion between governments, businesses, and individuals.

Shor's algorithm does not allow for direct factoring of a number, rather it reduces factoring to find the period of a modular exponential function where:

$N = p \times q$ where p, q are prime

Shor's algorithm uses quantum computation when finding the order of a modulo N, where N is an n-bit integer that we want to factor. The order r of a module N is the least positive integer such that:

$$a^r \equiv 1(mod\ N).$$

A. Greatest Common Divisor

The Greatest Common Divisor (GCD), also called the Highest Common Factor (HCF) is the largest natural number that evenly divides into both numbers without a reminder. Examples of GCD are as follows:

$$GCD(24, 36) = 12$$

$$GCD(1015, 23) = 7$$

$$GCD(15, 22) = 11$$

The best way to calculate the GCD is using Euclid's Division Algorithm.

B. Modulus Operator

The modulus operator performs division on two numbers and returns the remainder. Examples of the Modulus Operator are as follows:

$$17 \quad mod\ 5 = 2$$

$$45 \quad mod\ 9 = 0$$

$$101 \quad mod\ 21 = 17$$

C. Algorithm

This is a step-by-step explanation of Shor's Algorithm in its entirety. Let *N* represent the number to be factored, using the following example:

$$N = 15$$

1. Choose a random number A that satisfies: 1 < A < N, so they are coprimes of each other.

- I've chosen $A = 7$ for this example

2. Calculate the GCD using Euclid's Division Algorithm.

- If the GCD of $(N, A) = 1$ proceed to the next step.

  If N is the number to be factored, A represents the random number chosen from the previous step, then:

$$GCD\ (N, a)$$

$$GCD\ (15, 7)$$

3. Using a quantum computer, find the smallest positive integer r, such that if:

$$f(x) = a^x modN,$$

then:

$$f(A) = f(A + R)$$

- Define a new variable $Q = 1$
- Find $(Q \times A)modN$. If the remainder is 1, proceed to Step 4. If remainder is not 1, set q to the value of

the remainder of times the transformation is performed.

$$q \times a \, mod \, N = []$$
$$1 \times 7 \, mod \, 15 = 7$$
$$7 \times 7 \, mod \, 15 = 4$$
$$4 \times 7 \, mod \, 15 = 13$$
$$13 \times 7 \, mod \, 15 = 1$$

We performed the transformation 4 times total. Let $R = 4$

4. If R is an odd number, go back to Step 2 and choose another value for A and repeat. If R is an even number, continue to the next step.
5. Define P as the remainder in the $(R/2)^{th}$ transformation such that: $p = remainder$ in $(R/2)$ transformation

- If: $P + 1 = N$

  Return to Step 2 to choose a different value of K

- Else: Continue

  When $R = 4$, $P = 4$ then the equation would be $(4/2)$,

  the 2nd transformation: $4 + 1 = 5$ is not equal to $N = 15$

  Proceed to next step

6. Find the factors of N. They are:

$$f_1 = GCD(P + 1, N)$$
$$f_2 = GCD(P - 1, N)$$

Finishing my example where:

$$N = 15, A = 7, P = 4$$

The factors of N are as followed:

$$f_1 = GCD(P + 1, N)$$
$$= GCD(4 + 1, 15)$$
$$= GCD(5, 15)$$
$$f_1 = 5$$

$$f_2 = GCD(P - 1, N)$$
$$= GCD(4 - 1, 15)$$
$$= GCD(3, 15)$$
$$f_2 = 3$$

Therefore 3,5 are factors of 15

IV. Bernstein-Vazirani Algorithm

In 1992, Ethan Bernstein and Umesh Vazirani developed an algorithm, commonly known as the Bernstein-Vazirani algorithm, that solves a hidden shift problem. Utilizing quantum mechanics's superposition and entanglement properties, the Bernstein-Vazirani algorithm efficiently identifies a hidden, unknown binary string in one single run. In comparison to classical algorithms, this provides significant speed up times as it does not require n number of queries to be made. Cryptographic keys vary in length, spanning from 128 to 256 bits, to 2046 bits. The runtime of classical algorithms significantly limits its applications due to its minimum runtime of n runs, for a string of n length. If a classical algorithm was used to crack a cryptographic key of a 128 bit size, the runtime would be a minimum of 128 runs, just to fully traverse the key in its entirety. The Bernstein-Vazirani algorithm poses a threat to cracking classical cryptographic systems through cryptographic key-related attacks because of its significant speedup from the usage of the oracle and quantum circuit.

A. *Hidden Shift Problem and Algorithm*

The Hidden Shift problem states that when given a function, there is a guarantee an undetermined shift exists. The shift is an unknown bit string. The function is guaranteed to be of type $a \times x$ and when given an input, the output will always be of $a \times x \, mod(2)$. When f is a given function of bit strings, a is the phase shift, x is a string of n bits, n is the length of said bit string $x_0, x_1, x_2, x_3, ..., x_n$ the algorithm defines a function $f(x)$to determine the unknown n-bit string a as followed:

$$f(x) = a \times (mod2)$$

B. *Inner Product Oracle*

This algorithm uses a quantum oracle that applies a given function to a superposition of all possible inputs of a quantum state. Often referred to as the inner-product oracle, the quantum oracle is an inner modulus function that receives queries and applies a single-qubit unitary transformation. The complexity of the Bernstein-Vazirani algorithm is in terms of the number of queries to an oracle. The oracle has four core tasks:

- Containing a random n-bit hidden bit string
- Taking in two inputs, the Query and Auxiliary
- Calculating the inner product of the hidden bit string and Query $mod(2)$ without affecting the Query, and sets the Auxiliary to the value
- Outputting the Query and Auxiliary output value

C. *Quantum Circuit*

Similar to Deutsch-Jozsa Algorithm, the Bernstein-Vazirani Algorithm also uses a quantum circuit consisting of n qubits and outputs each bit of the hidden string after measurements are made. The usage of a quantum circuit solves the hidden shift problem in one query, rather than running multiple passes, at a minimum of n runs for a string of length n.

V. Deutsch-Jozsa Algorithm

In 1992, David Deutsch and Richard Jozsa proposed an algorithm known as the Deutsch-Jozsa Algorithm, solving a problem that determines whether a function is balanced or constant. The Deutsch-Jozsa Algorithm solves this problem efficiently by making one query to the function on a quantum computer; likewise solving this problem on a classical computer requires evaluating the function at each possible input. This problem is known as the Deutsch-Jozsa problem, and solving for it on a classical computer requires a minimum of:

$$\frac{n}{2+1}$$

queries to the function. Determining whether a binary function is constant or balanced can be used in cryptography when assessing the strength of an encryption function.

A. *Deutsch-Jozsa Problem*

Given a function that takes n bit string as input and outputs a truth value of one bit:

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

The function is either constant:

$$\forall x \in \{0,\ 1\}^n,\ f(x) = \beta$$

or balanced:

$$\forall x \in H,\ f(x) = 0$$
$$\forall x \notin H,\ f(x) = 1$$

B. *Constant and Balanced Functions on a Single Bit*

TABLE I. Function on a single bit

Constant and Balanced Function		
Function	***x f(x)***	***Type***
$f(x) = 0$	0 0 1 0	Constant
$f(x) = 1$	0 0 1 0	Constant
$f(x) = x$	0 0 1 0	Balanced
$f(x) = x \oplus 1$	0 0 1 0	Balanced

Fig. 1. Constant and balanced function on a single bit

C. *Algorithm*

- Initialize the quantum state using registers. Register 1 holds n input bits and Register 2 holds a single output bit. Both registers are set to 0.

$$|\psi_0\rangle = |01\rangle$$

- Apply a Hadamard gate to each input bit to create a superposition of all possible input states.

$$|\psi_1\rangle = \left(\frac{|0\rangle+|1\rangle}{\sqrt{2}}\right)\left(\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right)$$

- Apply a quantum oracle to the registers. The oracle performs a unitary operator function on each bit and sets the second qubit. This is factorized as:

$$|\psi_2\rangle = \begin{cases} \pm\left(\frac{|0\rangle+|1\rangle}{\sqrt{2}}\right)\left(\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right) & if \quad f(0)=f(1) \\ \pm\left(\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right)\left(\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right) & if \quad f(0)\neq f(1) \end{cases}$$

- Apply a Hadamard gate to each input bit after the Oracle to interfere with the superposition on the first qubit. This returns them to the original base state. Their results in:

$$|\psi_3\rangle = \begin{cases} \pm|0\rangle\left(\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right) & if \quad f(0)=f(1) \\ \pm|1\rangle\left(\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right) & if \quad f(0)\neq f(1) \end{cases}$$

- Measure the input bits and use their values to determine whether the function is balanced or constant

VI. Implementing Bernstein-Vazirani Algorithm

To further understand Bernstein-Vazirani's algorithm, I implemented it using quantum circuits, and used it to guess a Fernet encryption key. My coding environment consisted of:

- Visual Studio Code as an IDE
- Python 3.11 language
- Qiskit, Numpy, Pyenv Python submodules

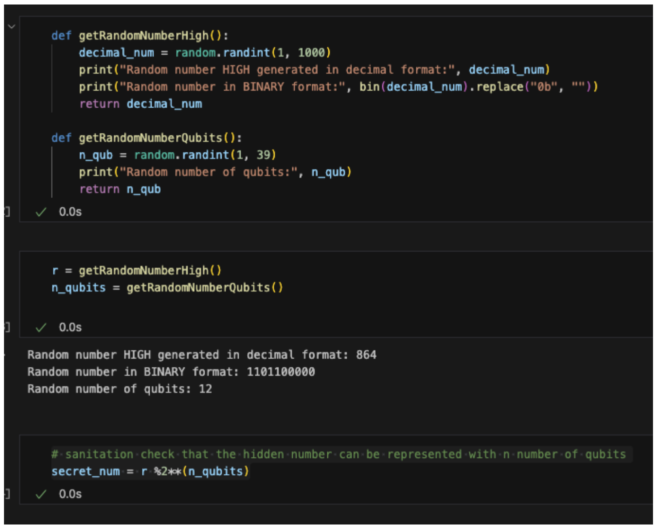
A. *Setting Up for a Quantum Circuit*

First I generated a random int, in decimal type, whose binary format would be used as my hidden string.

Next, I generated a random int between 1, 39 which represented the number of qubits for my circuit.

Then, I performed a modulus function on my hidden string, checking to ensure that my randomly generated integer would be able to be represented with the randomly generated number of qubits.

When f is my inner product oracle function and
r is a randomly generated integer used for my hidden string
n_qubits is a randomly generated integer for the number of qubits my circuit will use
$secret\_num$ is the inner product from a modulus function
The hidden string, number of qubits for my quantum circuit is as followed:



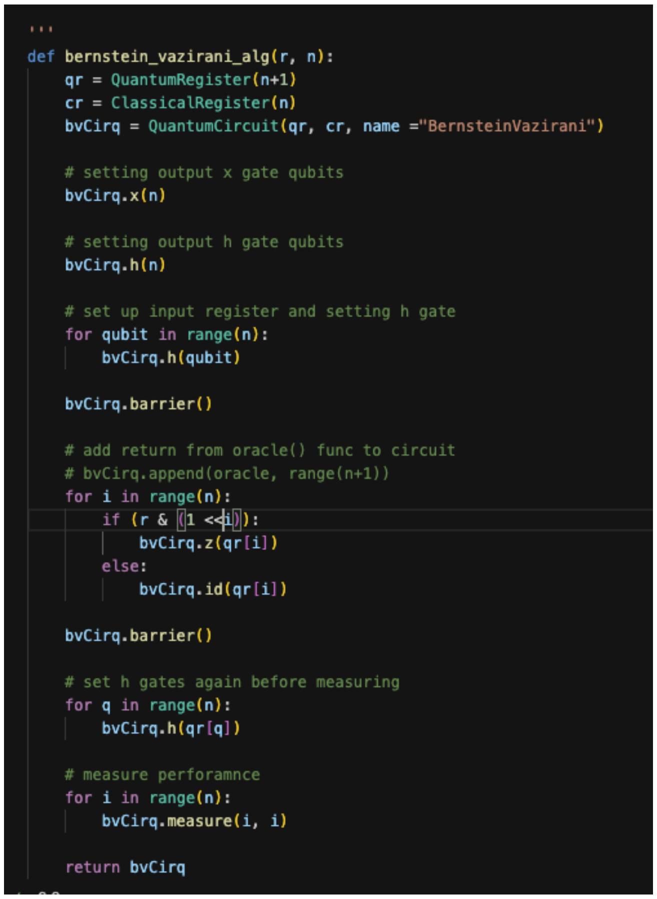
```
def getRandomNumberHigh():
    decimal_num = random.randint(1, 1000)
    print("Random number HIGH generated in decimal format:", decimal_num)
    print("Random number in BINARY format:", bin(decimal_num).replace("0b", ""))
    return decimal_num

def getRandomNumberQubits():
    n_qub = random.randint(1, 39)
    print("Random number of qubits:", n_qub)
    return n_qub
✓ 0.0s

r = getRandomNumberHigh()
n_qubits = getRandomNumberQubits()

✓ 0.0s
Random number HIGH generated in decimal format: 864
Random number in BINARY format: 1101100000
Random number of qubits: 12

# sanitation check that the hidden number can be represented with n number of qubits
secret_num = r %2**(n_qubits)
✓ 0.0s
```

B. *Creating Algorithm*

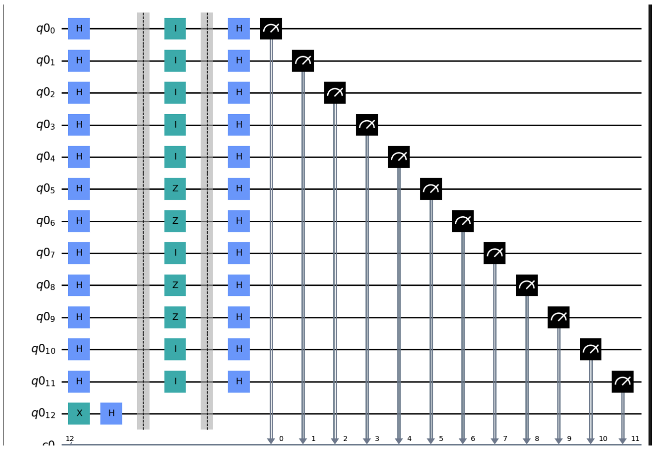
When $bernstein\_vazirani\_alg(r,\ n)$ is the algorithm and

r is the hidden string in decimal format
n is the number of qubits for the quantum circuit
qr is a quantum register of size $n\ +\ 1$
cr is a classical register of size n
$bvCirq$ is a QuantumCircuit
$.x$ is the output $x\ gate$
$.h$ is the output $h\ gate$
$if\ (f\ \&\ 1\ \ <<\ i))$: is the inner product oracle

The newly created quantum circuit implementing Bernstein-Vazirani's algorithm is as followed:

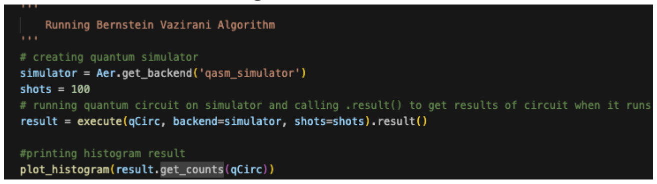


```
'''
def bernstein_vazirani_alg(r, n):
    qr = QuantumRegister(n+1)
    cr = ClassicalRegister(n)
    bvCirq = QuantumCircuit(qr, cr, name ="BernsteinVazirani")

    # setting output x gate qubits
    bvCirq.x(n)

    # setting output h gate qubits
    bvCirq.h(n)

    # set up input register and setting h gate
    for qubit in range(n):
        bvCirq.h(qubit)

    bvCirq.barrier()

    # add return from oracle() func to circuit
    # bvCirq.append(oracle, range(n+1))
    for i in range(n):
        if (r & (1 <<i)):
            bvCirq.z(qr[i])
        else:
            bvCirq.id(qr[i])

    bvCirq.barrier()

    # set h gates again before measuring
    for q in range(n):
        bvCirq.h(qr[q])

    # measure performnce
    for i in range(n):
        bvCirq.measure(i, i)

    return bvCirq
```



C. *Creating Quantum Simulator and Executing Circuit*

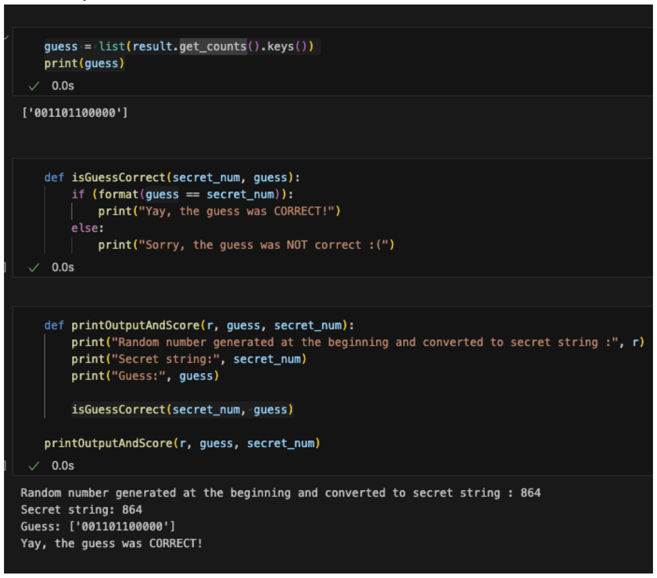
Next I created a quantum simulator and executed my newly created quantum circuit, implemented with Bernstein-Vazirani's Algorithm.



```
'''
    Running Bernstein Vazirani Algorithm
'''
# creating quantum simulator
simulator = Aer.get_backend('qasm_simulator')
shots = 100
# running quantum circuit on simulator and calling .result() to get results of circuit when it runs
result = execute(qCirc, backend=simulator, shots=shots).result()

#printing histogram result
plot_histogram(result.get_counts(qCirc))
```

D. Measuring and Testing Algorithm

Next I measured and tested my algorithm, and compared the hidden string with the algorithm's guess to determine accuracy.



```
guess = list(result.get_counts().keys())
print(guess)
✓ 0.0s

['001101100000']


def isGuessCorrect(secret_num, guess):
    if (format(guess == secret_num)):
        print("Yay, the guess was CORRECT!")
    else:
        print("Sorry, the guess was NOT correct :(")
✓ 0.0s


def printOutputAndScore(r, guess, secret_num):
    print("Random number generated at the beginning and converted to secret string :", r)
    print("Secret string:", secret_num)
    print("Guess:", guess)

    isGuessCorrect(secret_num, guess)

printOutputAndScore(r, guess, secret_num)
✓ 0.0s
Random number generated at the beginning and converted to secret string : 864
Secret string: 864
Guess: ['001101100000']
Yay, the guess was CORRECT!
```

VII. Implementing Deutsch-Jozsa Algorithm

A. Creating an Oracle Function

In order to implement Deutsch-Jozsa, I first created an oracle function. This oracle function accepts two parameters:

returnQGate: a binary flag of 1 or 0

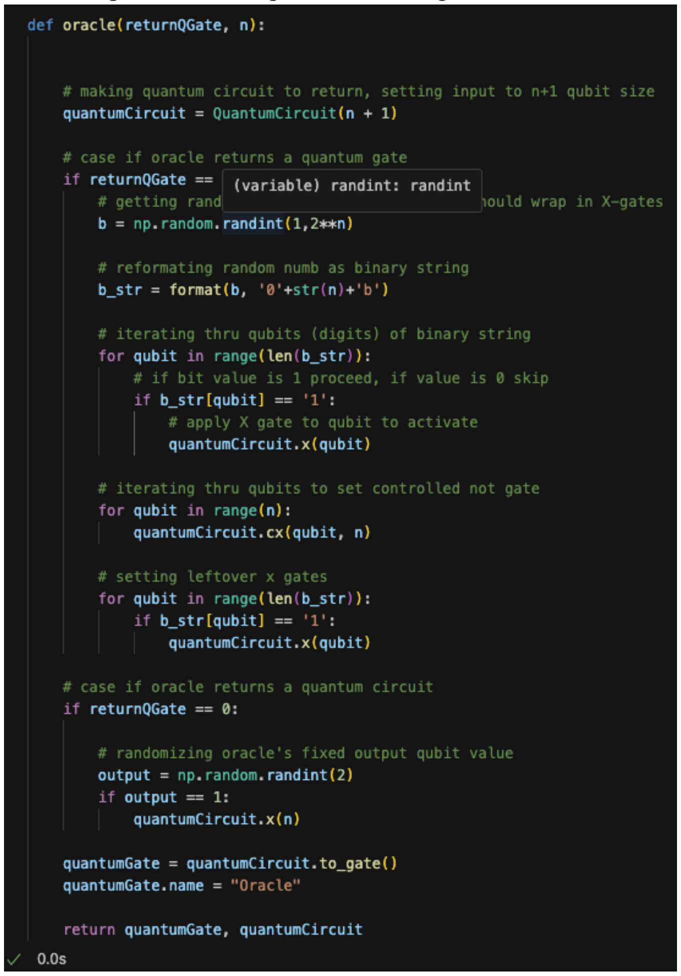
1 returns a quantum gate function

0 returns a circuit that generates bitstrings

n: length of a bit string

In my oracle function, I first create a quantum circuit to return, and set the input to $n + 1$ qubit size. In the case this function returns a quantum gate, I first generate a random number to know which CNOTs should be wrapped in X-gates. Then I reformat the random number as a binary string and iterate through the qubits (digits) of a binary string, finding a qubit with a value of 1 and applying an X gate to it to activate it. Then I iterate through the qubits to set my controlled NOT gates, and set my leftover gates.

In the case this function returns a quantum circuit, I randomly set the output value of a qubit with an X gate.



```
def oracle(returnQGate, n):


    # making quantum circuit to return, setting input to n+1 qubit size
    quantumCircuit = QuantumCircuit(n + 1)

    # case if oracle returns a quantum gate
    if returnQGate == (variable) randint: randint
        # getting rand                        hould wrap in X-gates
        b = np.random.randint(1,2**n)

        # reformating random numb as binary string
        b_str = format(b, '0'+str(n)+'b')

        # iterating thru qubits (digits) of binary string
        for qubit in range(len(b_str)):
            # if bit value is 1 proceed, if value is 0 skip
            if b_str[qubit] == '1':
                # apply X gate to qubit to activate
                quantumCircuit.x(qubit)

        # iterating thru qubits to set controlled not gate
        for qubit in range(n):
            quantumCircuit.cx(qubit, n)

        # setting leftover x gates
        for qubit in range(len(b_str)):
            if b_str[qubit] == '1':
                quantumCircuit.x(qubit)

    # case if oracle returns a quantum circuit
    if returnQGate == 0:

        # randomizing oracle's fixed output qubit value
        output = np.random.randint(2)
        if output == 1:
            quantumCircuit.x(n)

    quantumGate = quantumCircuit.to_gate()
    quantumGate.name = "Oracle"

    return quantumGate, quantumCircuit
✓ 0.0s
```

B. Setting Up for a Quantum Circuit

When *deutsch_jozsa_alg(oracle, n)* is the algorithm and

oracle: is a quantum circuit

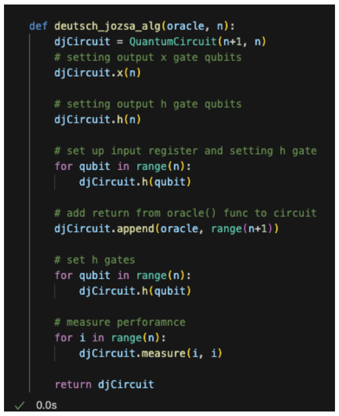
n: is a length of a bit string

djCircuit: is a newly created quantum circuit

with a quantum register of n+1 bits

with a classical register of n bits

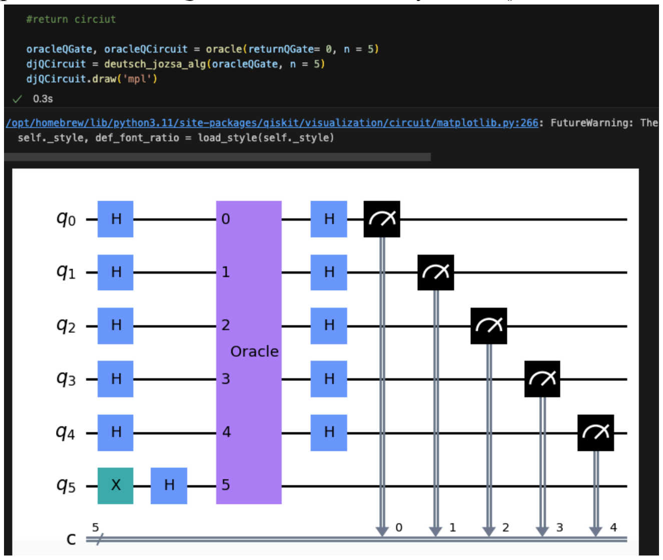
The newly created quantum circuit implementing Deutsch-Jozsa is as followed:



```
def deutsch_jozsa_alg(oracle, n):
    djCircuit = QuantumCircuit(n+1, n)
    # setting output x gate qubits
    djCircuit.x(n)

    # setting output h gate qubits
    djCircuit.h(n)

    # set up input register and setting h gate
    for qubit in range(n):
        djCircuit.h(qubit)

    # add return from oracle() func to circuit
    djCircuit.append(oracle, range(n+1))

    # set h gates
    for qubit in range(n):
        djCircuit.h(qubit)

    # measure perforamnce
    for i in range(n):
        djCircuit.measure(i, i)

    return djCircuit
✓ 0.0s
```

C. Returning a Quantum Circuit

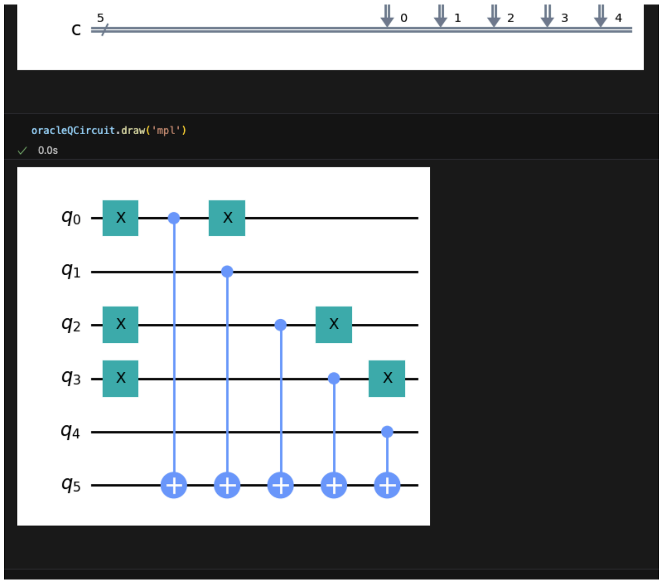
To demonstrate my Oracle returning a circuit, I set the parameter *returnQGate* to 0 and call my oracle() function.



```
#return circiut

oracleQGate, oracleQCircuit = oracle(returnQGate= 0, n = 5)
djQCircuit = deutsch_jozsa_alg(oracleQGate, n = 5)
djQCircuit.draw('mpl')
```
```
✓ 0.3s
/opt/homebrew/lib/python3.11/site-packages/qiskit/visualization/circuit/matplotlib.py:266: FutureWarning: The
  self._style, def_font_ratio = load_style(self._style)
```

D. Returning a Quantum Gate

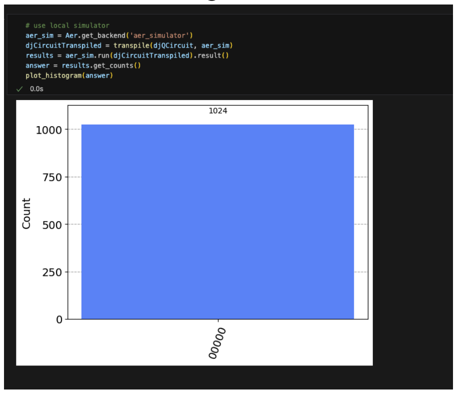
To demonstrate my Oracle returning a quantum gate, I set the parameter *returnQGate* to 1 and call my oracle() function.



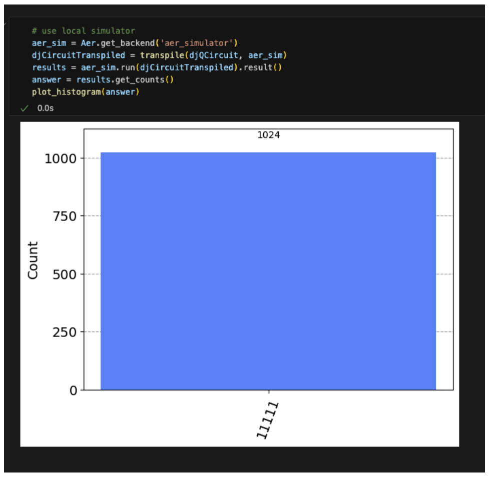
```
oracleQCircuit.draw('mpl')
```
```
✓ 0.0s
```

E. Creating Quantum Simulator and Executing Circuit or Gate

Next I created a quantum simulator and executed my newly created quantum circuit, implemented with the Deutsch-Jozsa's Algorithm.



```
# use local simulator
aer_sim = Aer.get_backend('aer_simulator')
djCircuitTranspiled = transpile(djQCircuit, aer_sim)
results = aer_sim.run(djCircuitTranspiled).result()
answer = results.get_counts()
plot_histogram(answer)
```
```
✓ 0.0s
```



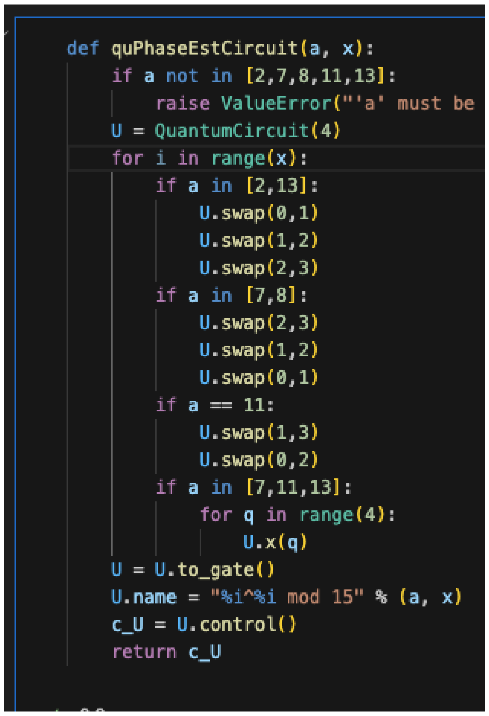
```
# use local simulator
aer_sim = Aer.get_backend('aer_simulator')
djCircuitTranspiled = transpile(djQCircuit, aer_sim)
results = aer_sim.run(djCircuitTranspiled).result()
answer = results.get_counts()
plot_histogram(answer)
```

✓ 0.0s

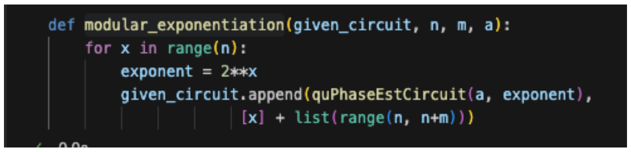
VIII. Implementing Shor's Algorithm

A. *Creating a Quantum Phase Estimation Circuit*

First I created a quantum phase estimation circuit in order to find the period of a function for $f(x) = a^x mod\, 15$. My QPE function accepts a unitary operator and an eigenstate and returns its phase. This is necessary for my quantum circuit later on.
I also created a helper function to do the modulus exponentiation on my circuit (which will be used later).

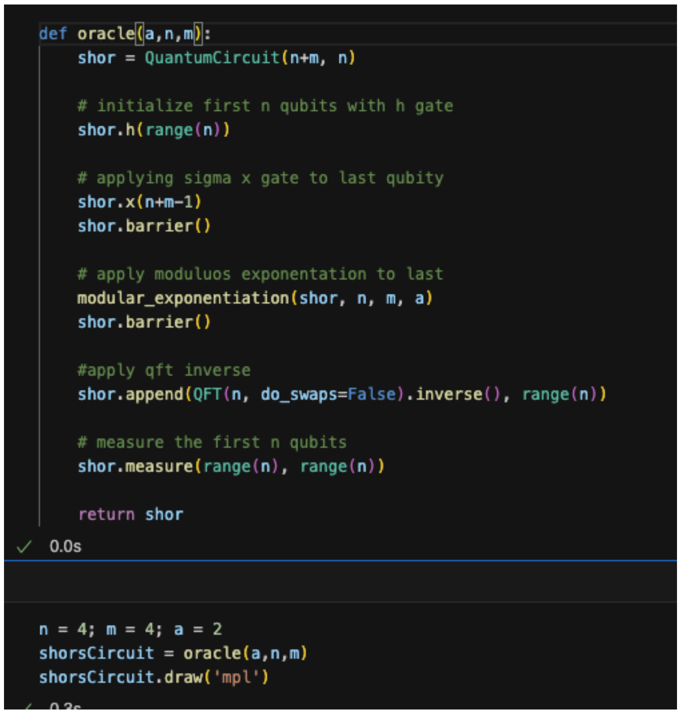


```
def quPhaseEstCircuit(a, x):
    if a not in [2,7,8,11,13]:
        raise ValueError("'a' must be
    U = QuantumCircuit(4)
    for i in range(x):
        if a in [2,13]:
            U.swap(0,1)
            U.swap(1,2)
            U.swap(2,3)
        if a in [7,8]:
            U.swap(2,3)
            U.swap(1,2)
            U.swap(0,1)
        if a == 11:
            U.swap(1,3)
            U.swap(0,2)
        if a in [7,11,13]:
            for q in range(4):
                U.x(q)
    U = U.to_gate()
    U.name = "%i^%i mod 15" % (a, x)
    c_U = U.control()
    return c_U
```



```
def modular_exponentiation(given_circuit, n, m, a):
    for x in range(n):
        exponent = 2**x
        given_circuit.append(quPhaseEstCircuit(a, exponent),
                             [x] + list(range(n, n+m)))
```

B. *Creating an Oracle Function*



```
def oracle(a,n,m):
    shor = QuantumCircuit(n+m, n)

    # initialize first n qubits with h gate
    shor.h(range(n))

    # applying sigma x gate to last qubity
    shor.x(n+m-1)
    shor.barrier()

    # apply moduluos exponentation to last
    modular_exponentiation(shor, n, m, a)
    shor.barrier()

    #apply qft inverse
    shor.append(QFT(n, do_swaps=False).inverse(), range(n))

    # measure the first n qubits
    shor.measure(range(n), range(n))

    return shor
```

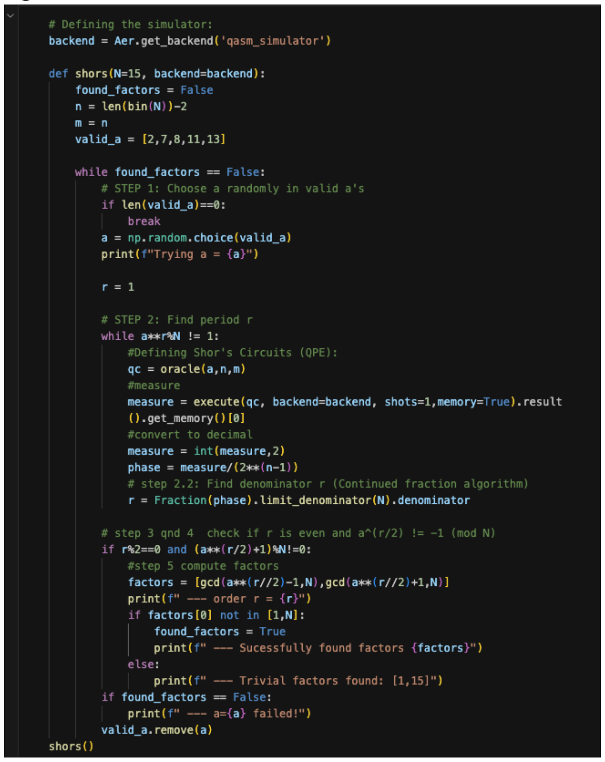
✓ 0.0s

```
n = 4; m = 4; a = 2
shorsCircuit = oracle(a,n,m)
shorsCircuit.draw('mpl')
```

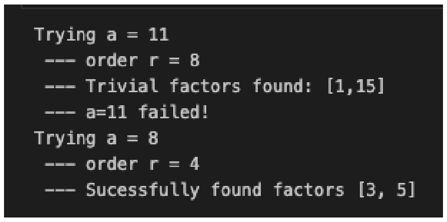
C. *Setting Up for a Quantum Circuit*

When *shors(N, backend)* is the algorithm and
N: is the number to be factorized
backend: is the quantum simulator
found_factors: is a boolean flag to determine if a factor has been found
n: is the binary size of N - 2
valid_a: is a list of valid numbers that meet the criteria

The newly created quantum circuit implementing Shor's algorithm is as followed:



```
# Defining the simulator:
backend = Aer.get_backend('qasm_simulator')

def shors(N=15, backend=backend):
    found_factors = False
    n = len(bin(N))-2
    m = n
    valid_a = [2,7,8,11,13]

    while found_factors == False:
        # STEP 1: Choose a randomly in valid a's
        if len(valid_a)==0:
            break
        a = np.random.choice(valid_a)
        print(f"Trying a = {a}")

        r = 1

        # STEP 2: Find period r
        while a**r%N != 1:
            #Defining Shor's Circuits (QPE):
            qc = oracle(a,n,m)
            #measure
            measure = execute(qc, backend=backend, shots=1,memory=True).result
            ().get_memory()[0]
            #convert to decimal
            measure = int(measure,2)
            phase = measure/(2**(n-1))
            # step 2.2: Find denominator r (Continued fraction algorithm)
            r = Fraction(phase).limit_denominator(N).denominator

        # step 3 qnd 4  check if r is even and a^(r/2) != -1 (mod N)
        if r%2==0 and (a**(r/2)+1)%N!=0:
            #step 5 compute factors
            factors = [gcd(a**(r//2)-1,N),gcd(a**(r//2)+1,N)]
            print(f" --- order r = {r}")
            if factors[0] not in [1,N]:
                found_factors = True
                print(f" --- Sucessfully found factors {factors}")
            else:
                print(f" --- Trivial factors found: [1,15]")
        if found_factors == False:
            print(f" --- a={a} failed!")
        valid_a.remove(a)
shors()
```



```
Trying a = 11
 --- order r = 8
 --- Trivial factors found: [1,15]
 --- a=11 failed!
Trying a = 8
 --- order r = 4
 --- Sucessfully found factors [3, 5]
```

Conclusion

The Quantum Fourier Transform (QFT) serves as a pivotal tool in quantum computing, enabling exponential speed-ups in comparison to classical algorithms.It plays a fundamental role in algorithms such as Shor's Algorithm, Bernstein-Vazirani, and Deutsch-Josza, transforming quantum states into encoded frequency spectra. These algorithms leverage the QFT as a quantum subroutine, to achieve quantum advantages. Shor's Algorithm, for instance, efficiently solves the Integer

Factorization problem, crucial for cryptography. Shor's Algo-rithm highlights the potential threat quantum computing poses on classical cryptographic systems. Similarly, the Bernstein-Vazirani Algorithm efficiently identifies hidden binary strings and the Deutsch-Jozsa Algorithm efficiently distinguishes between constant and balanced functions. Both algorithms are also applicable to cryptography, and are essential for assessing encryption function strength. The application of the QFT as a subroutine in these algorithms highlights its significance in quantum computing, offering efficient solutions for complex problems compared to classical approaches.

References

[1] "A quantum related-key attack based on Bernstein-Vazirani algorithm," ar5iv. Accessed: Mar. 10, 2024. [Online]. Available: https://ar5iv.labs.arxiv.org/html/1808.03266

[2] "Ansatze and Variational Forms | IBM Quantum Learning." Accessed: Mar. 09, 2024. [Online]. Available: https://learning.quantum.ibm.com/course/variational-algorithm-design/ansatze-and-variational-forms

[3] "Bernstein-Vazirani Algorithm.docx," Google Docs. Accessed: Mar. 10, 2024. [Online]. Available: https://docs.google.com/document/d/1w7GQ7qSDchPt0YiqkvZE7YQmTk9r0XO-/edit?usp=drive_web&ouid=104716854945157424942&rtpof=true&usp=embed_facebook

[4] "Cryptographic hash functions | IBM Quantum Learning." Accessed: Mar. 10, 2024. [Online]. Available: https://learning.quantum.ibm.com/course/practical-introduction-to-quantum-safe-cryptography/cryptographic-hash-functions

[5] S. Singh and E. Sakk, "Implementation and Analysis of Shor's Algorithm to Break RSA Cryptosystem Security," Preprints, preprint, Jan. 2024. doi: 10.36227/techrxiv.170259160.05374043/v2.

[6] R. Huang, X. Tan, and Q. Xu, "Learning to Learn Variational Quantum Algorithm," IEEE Transactions on Neural Networks and Learning Systems, vol. 34, no. 11, pp. 8430–8440, Nov. 2023, doi: 10.1109/TNNLS.2022.3151127.

[7] A. M. Childs, "Lecture Notes on Quantum Algorithms".

[8] J. Pipher, "Lectures on the NTRU encryption algorithm and digital signature scheme: Grenoble, June 2002".

[9] "Musty Thoughts." Accessed: Mar. 09, 2024. [Online]. Available: https://www.mustythoughts.com/variational-quantum-eigensolver-explained

[10] "NIST Announces First Four Quantum-Resistant Cryptographic Algorithms," NIST, Jul. 2022, Accessed: Mar. 10, 2024. [Online]. Available: https://www.nist.gov/news-events/news/2022/07/nist-announces-first-four-quantum-resistant-cryptographic-algorithms

[11] I. T. L. Computer Security Division, "Post-Quantum Cryptography | CSRC | CSRC," CSRC | NIST. Accessed: Mar. 09, 2024. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography

[12] I. T. L. Computer Security Division, "Post-Quantum Cryptography Standardization - Post-Quantum Cryptography | CSRC | CSRC," CSRC | NIST. Accessed: Mar. 10, 2024. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization

[13] "Quantum computing is taking on its biggest challenge: noise," MIT Technology Review. Accessed: Mar. 09, 2024. [Online]. Available: https://www.technologyreview.com/2024/01/04/1084783/quantum-computing-noise-google-ibm-microsoft/

[14] "Quantum-Safe Cryptography And the Quantum Threat." Accessed: Mar. 10, 2024. [Online]. Available: https://www.ssh.com/academy/cryptography/what-is-quantum-safe-cryptography

[15] L. Chen et al., "Report on Post-Quantum Cryptography," National Institute of Standards and Technology, NIST IR 8105, Apr. 2016. doi: 10.6028/NIST.IR.8105.

[16] Z.-X. Shang, M.-C. Chen, X. Yuan, C.-Y. Lu, and J.-W. Pan, "Schrödinger-Heisenberg Variational Quantum Algorithms," Phys. Rev. Lett., vol. 131, no. 6, p. 060406, Aug. 2023, doi: 10.1103/PhysRevLett.131.060406.

[17] "textbook/notebooks/ch-applications at main · Qiskit/textbook," GitHub. Accessed: Mar. 09, 2024. [Online]. Available: https://github.com/Qiskit/textbook/tree/main/notebooks/ch-applications

[18] H. Xie and L. Yang, "Using Bernstein–Vazirani algorithm to attack block ciphers," Des. Codes Cryptogr., vol. 87, no. 5, pp. 1161–1182, May 2019, doi: 10.1007/s10623-018-0510-5.

[19] "Variational algorithm design | IBM Quantum Learning." Accessed: Mar. 09, 2024. [Online]. Available: https://learning.quantum.ibm.com/course/variational-algorithm-design

[20] M. Cerezo et al., "Variational Quantum Algorithms," Nat Rev Phys, vol. 3, no. 9, pp. 625–644, Aug. 2021, doi: 10.1038/s42254-021-00348-9.

[21] Q. C. G. Roorkee IIT, "Variational Quantum Algorithms," Medium. Accessed: Mar. 09, 2024. [Online]. Available: https://medium.com/@qcgiitr/variational-quantum-algorithms-66367053a2f3

[22] I. Griol-Barres, S. Milla, A. Cebrián, Y. Mansoori, and J. Millet, "Variational Quantum Circuits for Machine Learning. An Application for the Detection of Weak Signals," Applied Sciences, vol. 11, no. 14, Art. no. 14, Jan. 2021, doi: 10.3390/app11146427.

[23] "Variational quantum eigensolver | IBM Quantum Learning." Accessed: Mar. 09, 2024. [Online]. Available: https://learning.quantum.ibm.com/tutorial/variational-quantum-eigensolver

[24] D. A. Fedorov, B. Peng, N. Govind, and Y. Alexeev, "VQE method: a short survey and recent developments," Materials Theory, vol. 6, no. 1, p. 2, Jan. 2022, doi: 10.1186/s41313-021-00032-6.

[25] "What are quantum algorithms? | Q-CTRL." Accessed: Mar. 09, 2024. [Online]. Available: https://q-ctrl.com/topics/what-are-quantum-algorithms