quantum_algo_exercises

1

1.a

QFT

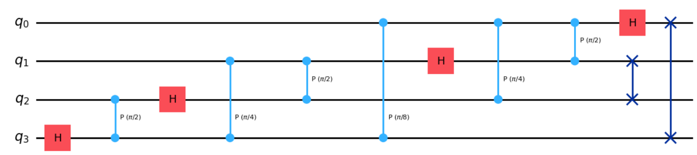


Figure 1: QFT

Inverse QFT

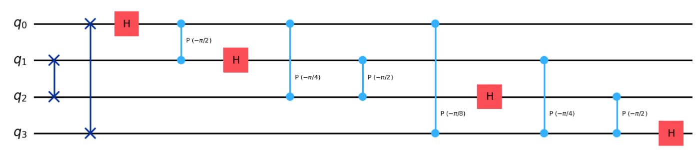


Figure 2: Inverse QFT

1.b

$$U_{QFT}|\phi\rangle = \left(\frac{1}{2\sqrt{2}} \sum_{j=0}^{7} \sum_{k=0}^{7} e^{-2\pi ijk/8} |k\rangle\langle j|\right)\left(\frac{1}{2}\sum_{l=0}^{7} \cos(2\pi l/8)|l\rangle\right)$$

$$= \frac{1}{4\sqrt{2}} \sum_{j=0}^{7}\left[\sum_{k=0}^{7} e^{-2\pi ijk/8} \cos\left(\frac{2\pi j}{8}\right)\right]|k\rangle$$

$$\frac{e^{-2\pi ijk/8} + e^{2\pi jk}}{2} = \cos\left(\frac{2\pi j}{8}\right)$$

$$\Rightarrow \sum_{k=0}^{7} e^{-2\pi ijk/8} \cos\left(\frac{2\pi j}{8}\right) = \sum_{k=0}^{7} e^{-2\pi ijk/8}\left(\frac{e^{2\pi j} + e^{-2\pi j}}{2}\right)$$

$$= \frac{1}{2}\left(\sum e^{-\frac{2\pi ij(k-1)}{8}} + \sum e^{-\frac{2\pi ij(k+1)}{8}}\right)$$

וכל שווה 0 עבור $k \neq 1,7$; 1 עבור $k=1,7$

$\Leftarrow$

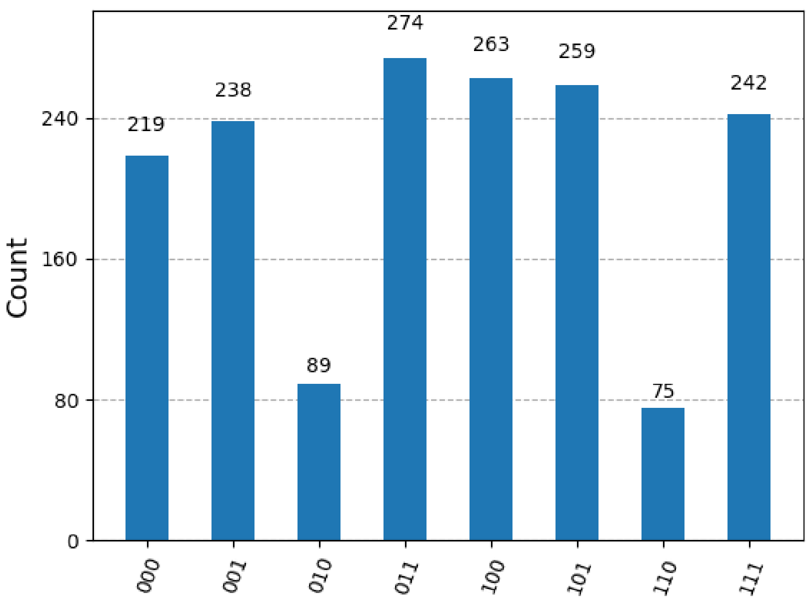
$$U_{QFT}|\phi\rangle = \frac{1}{\sqrt{2}}\left(|1\rangle + |7\rangle\right)$$

1.c

The result from b is encoded to the circuit with simple operations, then to reconstruct the original stage we used the inverse QFT and then $InverseQFT * QFT * \phi = \phi$. For more info see the src code: src/HW1.py

1.d

We can see that for j=2,6 the value is low, this is the excepted result from the original stage.



2

In [1]:
```python
#initialization
import matplotlib.pyplot as plt
import numpy as np
import math

# importing Qiskit
from qiskit import transpile, assemble
from qiskit_aer import AerSimulator
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister

# import basic plot tools
from qiskit.visualization import plot_histogram
```

In [2]:
```python
def qft_dagger(qc, n):
    """n-qubit QFTdagger the first n qubits in circ"""
    # Don't forget the Swaps!
    for qubit in range(n//2):
        qc.swap(qubit, n-qubit-1)
    for j in range(n):
        for m in range(j):
            qc.cp(-math.pi/float(2**(j-m)), m, j)
        qc.h(j)
```

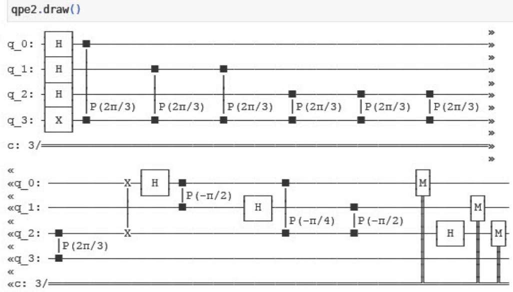
In [6]:
```python
#Aer.get_backend('aer_simulator')
aer_sim = AerSimulator()
```

In [7]:
```python
# Create and set up circuit
qpe2 = QuantumCircuit(4, 3)

# Apply H-Gates to counting qubits:
for qubit in range(3):
    qpe2.h(qubit)

# Prepare our eigenstate |psi>:
qpe2.x(3)

# Do the controlled-U operations:
angle = 2*math.pi/3
```



```python
repetitions = 1
for counting_qubit in range(3):
    for i in range(repetitions):
        qpe2.cp(angle, counting_qubit, 3);
    repetitions *= 2

# Do the inverse QFT:
qft_dagger(qpe2, 3)

# Measure of course!
for n in range(3):
    qpe2.measure(n,n)

qpe2.draw()
```

Out[7]:

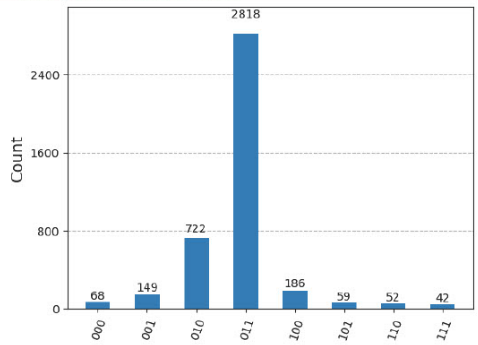


In [8]:
```python
# Let's see the results!
# aer_sim = Aer.get_backend('aer_simulator')
shots = 4096
t_qpe2 = transpile(qpe2, aer_sim)
qobj = assemble(t_qpe2, shots=shots)
```

```
results = aer_sim.run(qobj).result()
answer = results.get_counts()

plot_histogram(answer)
```

```
/tmp/ipykernel_5723/1318575361.py:6: DeprecationWarning: Using a qobj for run() is deprecated as of qiskit-aer 0.14 and will be removed no sooner than 3 months from that release date. Transpiled circuits should now be passed directly using `backend.run(circuits, **run_options).
  results = aer_sim.run(qobj).result()
```

Out[8]:



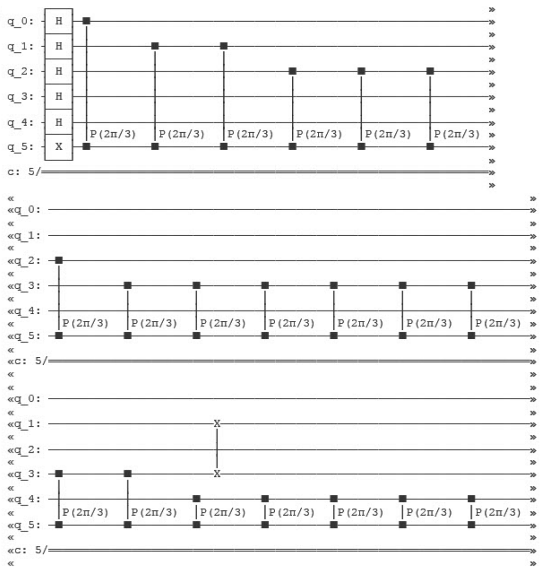
We are expecting the result $\theta = 0.3333\ldots$, and we see our most likely results are `010(bin) = 2(dec)` and `011(bin) = 3(dec)`. These two results would tell us that $\theta = 0.25$ (off by 25%) and $\theta = 0.375$ (off by 13%) respectively. The true value of θ lies between the values we can get from our counting bits, and this gives us uncertainty and imprecision.

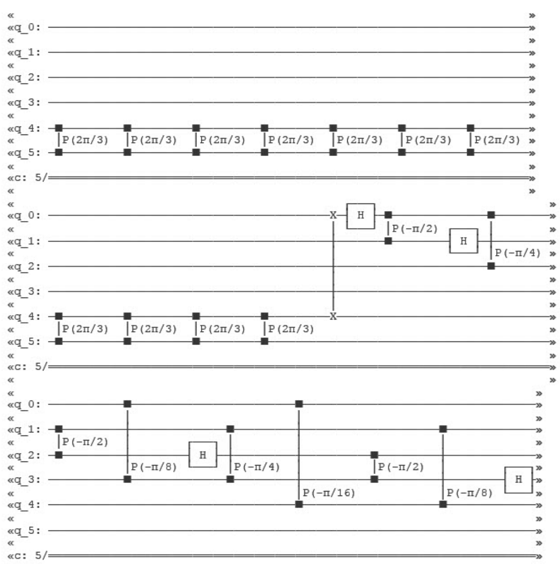
The second question is for t=5

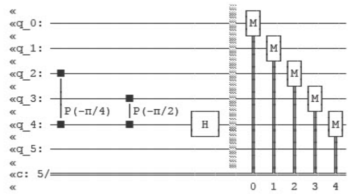
In [10]:

```
# Create and set up circuit
qpe3 = QuantumCircuit(6, 5)

# Apply H-Gates to counting qubits:
for qubit in range(5):
    qpe3.h(qubit)

# Prepare our eigenstate |psi>:
qpe3.x(5)

# Do the controlled-U operations:
angle = 2*math.pi/3
repetitions = 1
for counting_qubit in range(5):
    for i in range(repetitions):
        qpe3.cp(angle, counting_qubit, 5);
    repetitions *= 2

# Do the inverse QFT:
qft_dagger(qpe3, 5)

# Measure of course!
qpe3.barrier()
for n in range(5):
    qpe3.measure(n,n)

qpe3.draw()
```

Out[10]:



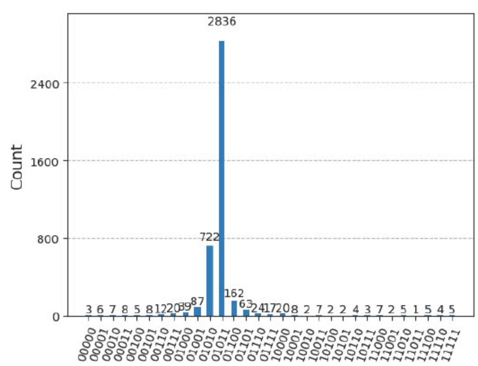




In [11]:
```python
# Let's see the results!
# aer_sim = Aer.get_backend('aer_simulator')
shots = 4096
t_qpe3 = transpile(qpe3, aer_sim)
qobj = assemble(t_qpe3, shots=shots)
results = aer_sim.run(qobj).result()
answer = results.get_counts()

plot_histogram(answer)
```

```
/tmp/ipykernel_5723/652080245.py:6: DeprecationWarning: Using a qobj for run() is deprecated as of qiskit-aer 0.14 and will be removed no sooner than 3 months from that release date. Transpiled circuits should now be passed directly using `backend.run(circuits, **run_options).
  results = aer_sim.run(qobj).result()
```


Out[11]:



The two most likely measurements are now `01011` (decimal 11) and `01010` (decimal 10). Measuring these results would tell us θ is:

$$\theta = \frac{11}{2^5} = 0.344, \text{ or } \theta = \frac{10}{2^5} = 0.313$$

These two results differ from $\frac{1}{3}$ by 3% and 6% respectively. A much better precision!

In [ ]:

3

תרגיל 3

נשתמש בלוגיקה דומה לזו של האלגוריתם במאמר למציאה של מינימום.
בשונה, הפעם יהיה מוגדרת הפונקציה:

$$f_i(j) = \begin{cases} 1 & \text{if } T[j] < T[i] \\ 0 & \text{otherwise} \end{cases}$$

האלגוריתם עצמו וניתוח של הסיבוכיות יהיו כמו שמוסבר במאמר.
נעבור על השלבים שמוסברים במאמר:

(1) בחר אינדקס j אקראי בתחום $\{0..N-1\}$

(2) חזור על הפעולות הבאות $O(\sqrt{N})$ פעמים:

(א) לאתחל מצב כמו כזה $|\psi\rangle = \sum_i \frac{1}{\sqrt{N}}|i\rangle$ בסיבוכיות $O(\log N)$

(ב) הפעלת האלגוריתם של גרובר על מנת למצוא סימונים x שעבורם $f_j(x) = 1$ כלומר $f_j(x)=1$.
במידה ויש כזה x_0 ונשמור אותו כ- j עבור האיטרציה הבאה.

(3) בסוף, נחזיר את j כפלט של האלגוריתם.

הסיבוכיות מוכחת במאמר. אך, נציין כי יש $O(\sqrt{N})$ איטרציות וכל הסיבוכיות של כל איטרציה היא $O(\log N)$

4

4.a

שאלה 4

סעיף א:

$$\sum_a a_i \frac{\Delta_3(\Pi_\phi, \Pi_{a_i}, \Pi_\psi)}{\Delta_2(\Pi_\phi, \Pi_\psi)} =$$

$$\sum_a a_i \frac{Tr(|\phi\rangle\langle\phi|\,|a_i\rangle\langle a_i|\,|\psi\rangle\langle\psi|)}{Tr(|\phi\rangle\langle\phi|\,|\psi\rangle\langle\psi|)} =$$

$$\sum_a a_i \frac{Tr(\langle\phi a_i\rangle\langle a_i\psi\rangle\langle\psi|\phi\rangle)}{Tr(\langle\phi\psi\rangle\langle\psi\phi\rangle)} =$$

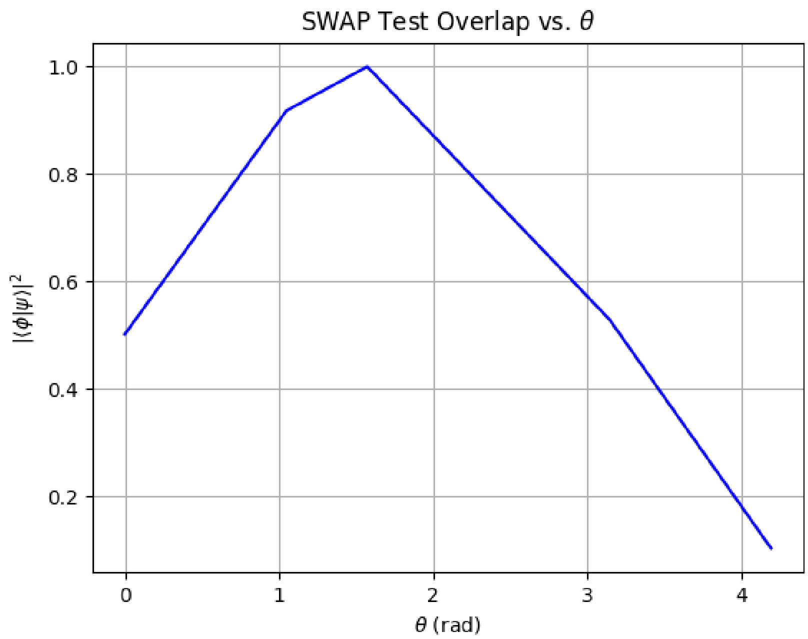
$$\sum_a a_i \frac{Tr(\langle\phi a_i\rangle\langle a_i\psi\rangle)}{Tr(\langle\phi\psi\rangle)} =$$

$$\frac{\langle\phi|\left(\sum_a a_i\rangle\langle a_i\right)|\psi}{\langle\phi|\psi\rangle} =$$

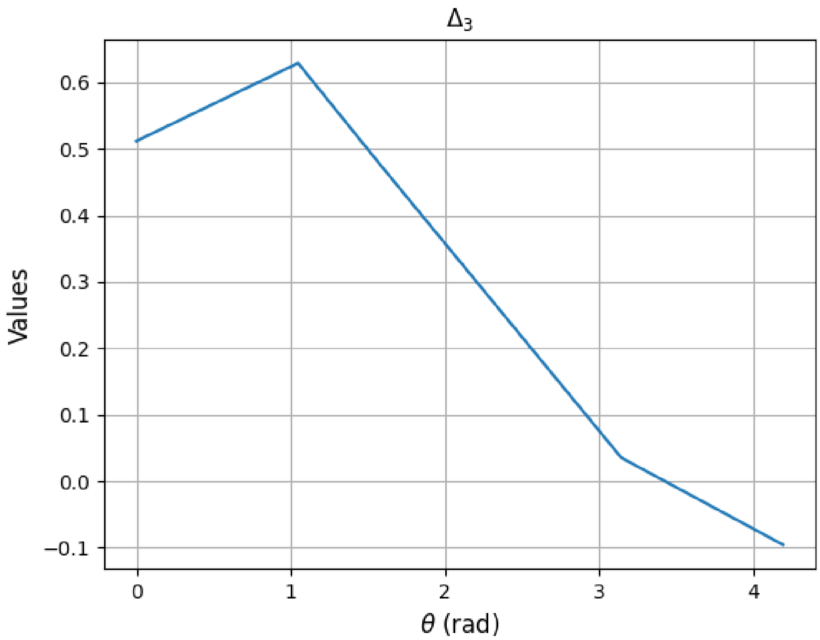
$$\frac{\langle\phi|A|\psi\rangle}{\langle\phi|\psi\rangle}$$

4.b

(i)


SWAP Test Overlap vs. θ
$|\langle\phi|\psi\rangle|^2$
1.0
0.8
0.6
0.4
0.2
0
1
2
3
4
θ (rad)

(ii)



Note that

$$\Delta_3(\Pi_\phi, |1\rangle\langle 1|\Pi_\psi)$$

is computed as: $\Delta_3(\Pi_\phi, |1\rangle\langle 1|\Pi_\psi) = \mathrm{Tr}(\Pi_\phi|1\rangle\langle 1|\Pi_\psi) = \mathrm{Tr}(\langle 1|\Pi_\psi\Pi_\phi|1\rangle) = \Pi_\psi\Pi_\phi(2,2) = \mathrm{Tr}(\Pi_\psi\Pi_\phi) - (\Pi_\psi\Pi_\phi)(1,1) = \Delta_2(\Pi_\psi\Pi_\phi) - \Delta_3(\Pi_\phi|0\rangle\langle 0|\Pi_\psi)$

Since the right-hand side has already been calculated, we can now compute the left-hand side directly without any further measurements.

Answer

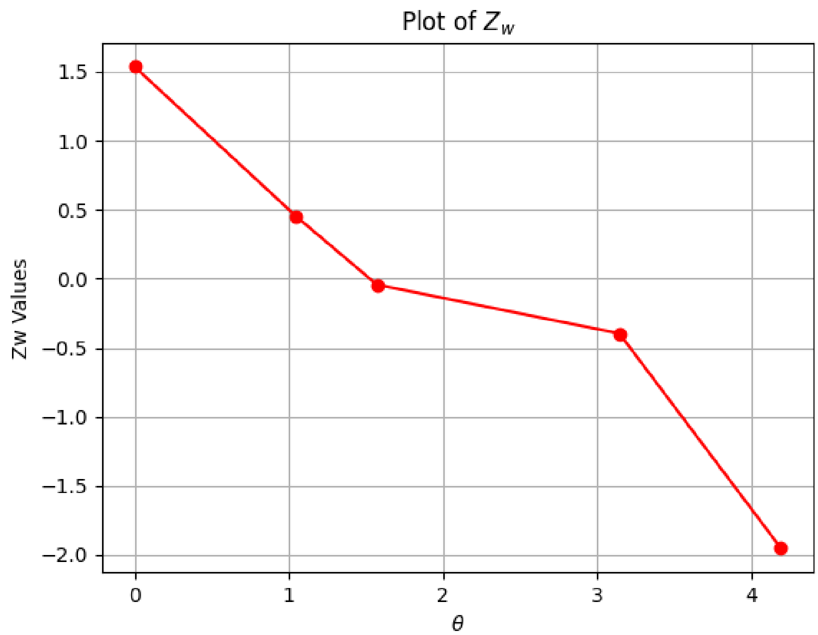
Recall that Z is defined as:

$$Z = 1 \cdot |0\rangle\langle 0| + (-1) \cdot |1\rangle\langle 1|$$

We will now use the identity proven in part 4(a), along with the previously computed values of Δ_2 and Δ_3, to compute Z_w:

$$Z_w = \frac{\Delta_3(\Pi_\phi|0\rangle\langle 0|\Pi_\psi)}{\Delta_2(\Pi_\phi\Pi_\psi)} - \frac{\Delta_3(\Pi_\phi|1\rangle\langle 1|\Pi_\psi)}{\Delta_2(\Pi_\phi\Pi_\psi)}$$

The result Z_w can be computed directly using the above equation.



Code References:

File Structure

```
src
├── file_structure.txt
├── helpers.py
├── HWs
│   ├── HW1.py
│   ├── HW2.pdf
│   └── HW4_b.py
└── main.py

1 directory, 6 files
```

Source Code Files

File: main.py

```python
from pydantic_settings import BaseSettings, SettingsConfigDict
from HWs.HW4_b import run_4_b
from HWs.HW1 import run_1_c


class IBMCredentials(BaseSettings):
    model_config = SettingsConfigDict(env_file=".env")
    token: str


credentials = IBMCredentials()

if __name__ == '__main__':
    run_1_c(credentials.token)  # For explanation see the README and the code
    run_4_b()  # For the explanation see the doc string of the function
```

File: helpers.py

```python
from qiskit.utils import optionals as _optionals
import matplotlib.pyplot as plt
from qiskit import QuantumCircuit

def visualize_circuit(circuit: QuantumCircuit, show_table=True, output=None):
    circuit = circuit.decompose()

    if not show_table:
        circuit.draw("mpl")
        if output is not None:
            print("Saving results to", output)
            plt.savefig(output)
        plt.show()
        return
    ops = circuit.count_ops()
    num_nl = circuit.num_nonlocal_gates()
    _fig, (ax0, ax1) = plt.subplots(2, 1)
    circuit.draw("mpl", ax=ax0)
    ax1.axis("off")
    ax1.grid(visible=None)
    ax1.table(
        [[circuit.name], [circuit.width()], [circuit.depth()], [sum(ops.values())], [num_nl]],
        rowLabels=["Circuit Name", "Width", "Depth", "Total Gates", "Non-local Gates"],
    )
    plt.tight_layout()
    plt.show()
    if output is not None:
        print("Saving results to", output)
        plt.savefig(output)
```

File: HW4_b.py

```python
from qiskit_aer import AerSimulator
from qiskit import QuantumCircuit, transpile
import numpy as np
import matplotlib.pyplot as plt


def cycle_test_circuit(alpha):
```

```
    qc = QuantumCircuit(4, 1)
    qc.h(0)
    qc.ry(alpha, 1)
    qc.h(3)
    qc.cswap(0, 1, 2)
    qc.cswap(0, 2, 3)
    qc.h(0)
    qc.measure(0, 0)
    return qc


def swap_test_circuit(alpha):
    qc = QuantumCircuit(3, 1)
    qc.h(0)
    qc.h(2)
    qc.ry(alpha, 1)
    qc.cswap(0, 1, 2)
    qc.h(0)
    qc.measure(0, 0)
    return qc


def run_circuit(qc):
    simulator = AerSimulator()
    compiled_circuit = transpile(qc, simulator)
    job = simulator.run(compiled_circuit, shots=1024)
    result = job.result()
    counts = result.get_counts(qc)

    # The probability of ancilla being in state |0> gives the overlap as 1 - 2*P(ancilla=1)
    p0 = counts.get('0', 0) / 1024
    overlap_squared = 2 * p0 - 1  # Weak value reconstruction
    return overlap_squared


def compute_and_plot_overlaps2(alphas):
    # List to store the results
    overlaps_2 = []

    # Run the SWAP test for each value of theta
    for alpha in alphas:
        qc = swap_test_circuit(alpha)
        overlap = run_circuit(qc)
        overlaps_2.append(overlap)

    # Plotting the results
    plt.plot(alphas, overlaps_2, color='b')
    plt.xlabel(r'$\theta$ (rad)')
    plt.ylabel(r'$| \langle \phi | \psi \rangle |^2$')
    plt.title('SWAP Test Overlap vs. $\\theta$')
    plt.grid(True)
    plt.savefig("docs/overlap2.png")

    plt.show()

    return np.array(overlaps_2)


def compute_and_plot_overlaps3(alphas):
    overlaps_3 = []

    # Run the cycle test for each value of theta and compute the LHS/RHS expression
    for alpha in alphas:
        qc = cycle_test_circuit(alpha)
        overlap = run_circuit(qc)
        overlaps_3.append(overlap)

    plt.plot(alphas, overlaps_3)
    plt.xlabel(r'$\theta$ (rad)', fontsize=12)
    plt.ylabel('Values', fontsize=12)
    plt.title(r'$\Delta_3$')
    plt.grid(True)

    # Show plot
    plt.savefig("docs/overlap3.png")

    plt.show()
    return overlaps_3


def compute_and_plot_Z(overlaps_2, overlaps_3, alphas):
    # Compute the new array (Zw) using the formula provided
    new_array = (overlaps_3 / overlaps_2) - (overlaps_2 - overlaps_3 / overlaps_2)
    # Plot the result
    plt.plot(alphas, new_array, marker='o', linestyle='-', color='r')
    plt.xlabel(r'$\theta$')
    plt.ylabel('Zw Values')
    plt.title('Plot of $Z_w$')
    plt.grid(True)
    plt.savefig("docs/z.png")
    plt.show()


def run_4_b():
    """
```

```
    4.b(ii)
        Note that Δ3(Π_φ, |1⟩⟨1| Π_ψ) is computed as:
        Δ3(Π_φ, |1⟩⟨1| Π_ψ) = Tr(Π_φ |1⟩⟨1| Π_ψ)
                              = Tr(⟨1| Π_ψ Π_φ |1⟩)
                              = (Π_ψ Π_φ)_(2,2)
                              = Tr(Π_ψ Π_φ) - (Π_ψ Π_φ)_(1,1)
                              = Δ2(Π_ψ Π_φ) - Δ3(Π_φ |0⟩⟨0| Π_ψ)

        Since the right hand side has already been calculated,
        we can now compute the left hand side directly without any further measurements.
    4.b
        Recall that Z is defined as:
        Z = 1 * |0⟩⟨0| + (-1) * |1⟩⟨1|

        We will now use the identity proven in part 4(a),
        along with the previously computed values of Δ2 and Δ3,
        to compute Z_w:

        Z_w = (Δ3(Π_φ |0⟩⟨0| Π_ψ) / Δ2(Π_φ Π_ψ)) - (Δ3(Π_φ |1⟩⟨1| Π_ψ) / Δ2(Π_φ Π_ψ))

        The result Z_w can be computed directly using the above equation.
    """
    alphas = [0, np.pi / 3, np.pi / 2, np.pi, 4 * np.pi / 3]

    overlaps_2 = compute_and_plot_overlaps2(alphas)

    overlaps_3 = compute_and_plot_overlaps3(alphas)

    compute_and_plot_Z(overlaps_2, overlaps_3, alphas)
```

File: HW1.py

```
from qiskit import transpile
from qiskit.circuit.library import QFT
from qiskit import QuantumCircuit
from qiskit_ibm_runtime import QiskitRuntimeService
from qiskit.visualization import plot_histogram

def encode_phi_stage(circuit):
    """
    This circuit encode the stage 1/2(|1> + |7>)
    With simple calculation we can see that applying this circuit create this stage.
    For reconstruct the origin stage we will apply the inverse QFT on this stage.
    """
    circuit.x(2)
    circuit.cx(0, 1)
    # Because qiskit
    circuit.swap(0, 2)


def combine_phi_with_inverse_qft(circuit):
    circuit.append(QFT(3, inverse=True), range(3))
    circuit.measure(range(3), range(3))


def run_qc_on_real_hardware(circuit, token):
    service = QiskitRuntimeService(channel="ibm_quantum", token=token)
    backend = service.least_busy(simulator=False, operational=True)
    shots = 2048
    transpiled_qc = transpile(circuit, backend, optimization_level=3)
    job = backend.run(transpiled_qc, shots=shots)
    # Use the job ID to retrieve your job data later
    print(f">>> Job ID: {job.job_id()}")


def run_1_c(token):
    qc = QuantumCircuit(3, 3)
    encode_phi_stage(qc)
    combine_phi_with_inverse_qft(qc)
    run_qc_on_real_hardware(qc, token)

def plot_results(job_id, token):
    from qiskit_ibm_runtime import QiskitRuntimeService

    service = QiskitRuntimeService(
        channel='ibm_quantum',
        instance='jupiter/internal/default',
        token=token
    )
    job = service.job(job_id)
    counts = job.result().get_counts()
    plot_histogram(counts)


if __name__ == '__main__':
    from src.main import credentials
    # run_1_c(credentials.token) # Job ID: cvxg1zz22dfg008n21xg
    plot_results("cvxg1zz22dfg008n21xg", credentials.token)
```