Estimation of pi (π) using Quantum Phase Estimation

March 19, 2023

Author: Jeevesh Krishna Arigala

Date: 2023-03-19

1 Estimation of pi (π) using Quantum Phase Estimation Algorithm (QPE)

1.1 1. Quantum Phase Estimation Algorithm

Quantum Phase Estimation (QPE) is a fundamental quantum algorithm that serves as a building block for many other complex quantum algorithms. The problem it solves is relatively simple: given an operator U and a quantum state $|\psi\rangle$ that is an eigenvalue of U with $U|\psi\rangle = \exp(2\pi i\theta)\,|\psi\rangle$, can we obtain an estimate of θ?

The answer is affirmative, and the QPE algorithm provides us with $2^n\theta$, where n is the number of qubits we use to estimate the phase θ.

1.2 2. Estimating π

As a demonstration of the Quantum Phase Estimation (QPE) algorithm, we consider the problem of estimating the mathematical constant π using a quantum circuit.

We choose a quantum operator U and a quantum state $|\psi\rangle$ that are eigenvalues of U with $U|\psi\rangle = \exp(2\pi i\theta)\,|\psi\rangle$. Specifically, we select

$$U = p(\theta), \quad |\psi\rangle = |1\rangle$$

where the gate $p(\theta)$ is defined as

$$p(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & \exp(i\theta) \end{bmatrix}.$$

The action of this gate on the state $|1\rangle$ yields $p(\theta)|1\rangle = \exp(i\theta)|1\rangle$.

To estimate π, we choose the phase of the gate to be $\theta = 1$ and use the QPE algorithm to estimate $2^n\theta$. From the output of the QPE algorithm, we obtain an estimate for $2^n\theta$, and then θ can be computed as $\theta = \text{measured}/2^n$.

From the definition of the $p(\theta)$ gate above, we know that $2\pi\theta = 1$. Therefore, we can solve for π by using the relationship:

$$\pi = \frac{1}{2\theta}.$$

Combining these two relationships, we obtain the estimate for π as:

$$\pi = \frac{1}{2 \times (\text{measured}/2^n)}.$$

The above process demonstrates how the QPE algorithm can be used to estimate π using a quantum circuit. For a more detailed understanding of the QPE algorithm and its applications, refer to relevant literature in quantum computing.

1.3 3. Implementation

Importing the necessary libraries

[1]:
```
from qiskit import *
from qiskit.visualization import plot_histogram
from qiskit.tools.monitor import job_monitor
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import display, Math, Latex
from IPython.display import clear_output
import seaborn as sns, operator
sns.set_style('darkgrid')
pi = np.pi
```

The function `qft_dagger` computes the inverse Quantum Fourier Transform. The QFT is an essential operation in many quantum algorithms, and its inverse is used in many applications such as state preparation, amplitude amplification, and phase estimation.

The function qft_dagger implements the inverse QFT using a set of basic quantum gates, including the Hadamard gate (H) and the controlled-phase gate (CP). The algorithm consists of two parts: the first part swaps the qubits in the circuit, and the second part applies the CP and H gates in a nested loop structure.

The first loop iterates over the qubits in the range [0, n_qubits/2) and performs a swap operation between the qubit at position qubit and the qubit at position n_qubits-qubit-1. This swap operation is necessary because the QFT assumes that the qubits are ordered in reverse bit order, and the swap operation ensures that the qubits are in the correct order for the inverse QFT.

The second loop applies the gates in the inverse QFT circuit in a nested structure, where each iteration j applies a set of CP and H gates to qubit j and all the qubits with smaller indices. The CP gate applies a phase shift of -pi/2^(j-m) to the state of qubit m conditioned on the state of qubit j, while the H gate is applied to qubit j alone. The CP gate is the controlled version of the phase gate, which applies a global phase shift to the state of the qubit.

Note that the phase shift applied by the CP gate is negative, which is why the minus sign is present in the function. The gates are applied in this nested loop structure to simulate the effect of the

QFT on the state of the qubits. The final result is the inverse QFT of the state of the first n_qubits qubits in the circuit circ_.

[2]:
```
# adapted from Qiskit Textbook
def qft_dagger(circ_, n_qubits):
    """n-qubit QFTdagger the first n qubits in circ"""
    for qubit in range(int(n_qubits/2)):
        circ_.swap(qubit, n_qubits-qubit-1)
    for j in range(0,n_qubits):
        for m in range(j):
            circ_.cp(-np.pi/float(2**(j-m)), m, j)
        circ_.h(j)
```

The qpe_pre function prepares the initial state for the phase estimation algorithm. It takes as input a quantum circuit circ_ and the number of qubits n_qubits that will be used in the algorithm.

The first step in qpe_pre is to apply a Hadamard gate to each qubit in the circuit, except for the last qubit. This creates a superposition of all possible states for the first n_qubits-1 qubits.

The second step is to set the last qubit to the state $|1\rangle$. This is done using the x gate, which flips the qubit from $|0\rangle$ to $|1\rangle$.

The final step is to apply a series of controlled-phase gates (denoted by cp in the code) to the qubits in the circuit. The gates are applied in reverse order, starting with the last qubit and working backwards. The number of gates applied to each qubit doubles as the algorithm progresses.

The controlled-phase gate between qubits j and k applies a phase shift of $e^{i\theta}$ to the state $|1\rangle$ of qubit k if qubit j is in the state $|1\rangle$. The angle θ is given by $\theta = 2\pi/2^{j-k+1}$.

Note that in the qpe_pre function, the angle of the controlled-phase gates is set to 1. This is because the specific value of θ will be determined by the QPE algorithm itself.

[3]:
```
def qpe_pre(circ_, n_qubits):
    circ_.h(range(n_qubits))
    circ_.x(n_qubits)

    for x in reversed(range(n_qubits)):
        for _ in range(2**(n_qubits-1-x)):
            circ_.cp(1, n_qubits-1-x, n_qubits)
```

function to run the quantum circuit

[4]:
```
def run_job(circ, backend, shots=1000, optimization_level=0):
    t_circ = transpile(circ, backend, optimization_level=optimization_level)
    qobj = assemble(t_circ, shots=shots)
    job = backend.run(qobj)
    job_monitor(job)
    return job.result().get_counts()
```

[5]:
```
my_provider = IBMQ.load_account()
simulator_cloud = my_provider.get_backend('ibmq_qasm_simulator')
```

```
device = my_provider.get_backend('ibm_perth')
```

Function to estimate pi using n qubits

[6]:
```
def get_estimate(n_qubits):

    circ = QuantumCircuit(n_qubits + 1, n_qubits)

    qpe_pre(circ, n_qubits)

    circ.barrier()

    # apply the inverse fourier transform
    qft_dagger(circ, n_qubits)

    circ.barrier()

    circ.measure(range(n_qubits), range(n_qubits))

    counts = run_job(circ, backend=simulator, shots=10000, optimization_level=0)

    max_counts_result = max(counts, key=counts.get)
    max_counts_result = int(max_counts_result, 2)

    theta = max_counts_result/2**n_qubits
    return (1./(2*theta))
```

[7]:
```
simulator = Aer.get_backend('aer_simulator')
```

Running the function with different number of qubits

[8]:
```
qubits = list(range(2,13))
pi_estimates = []
for qubit in qubits:
    qubit_pi_estimate = get_estimate(qubit)
    pi_estimates.append(qubit_pi_estimate)
    print(f"{qubit} qubits, pi  {qubit_pi_estimate}")
```

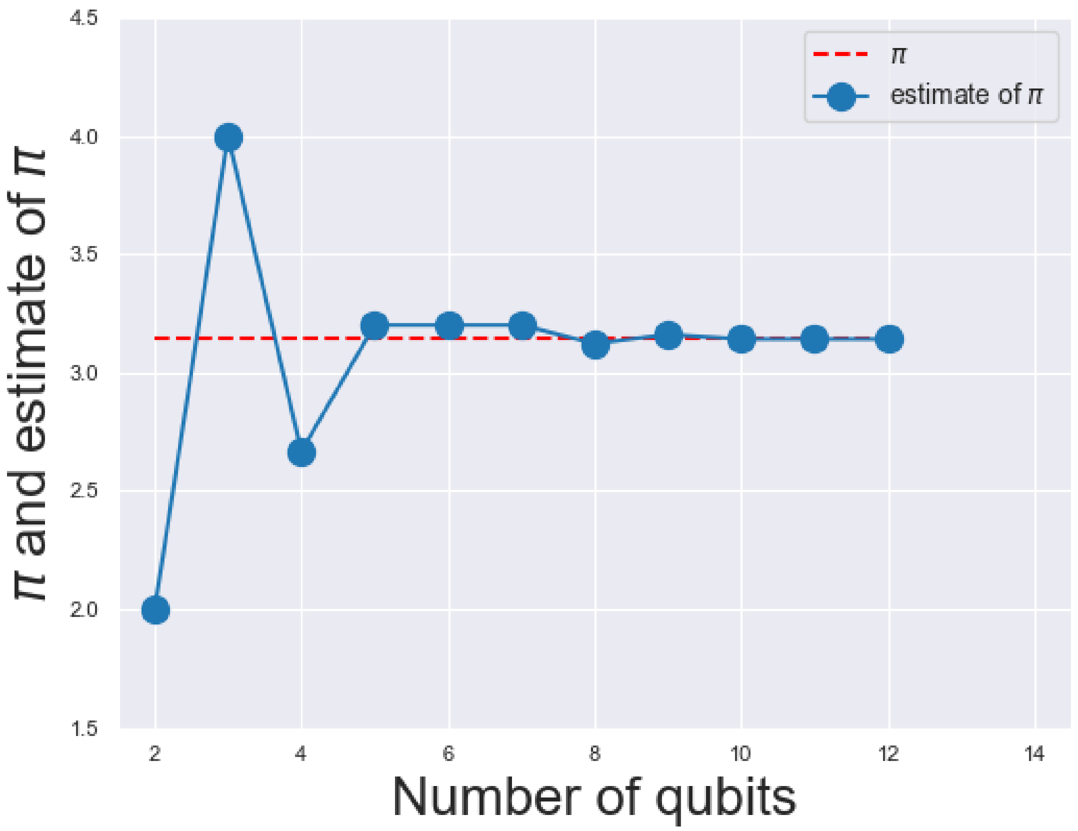
```
Job Status: job has successfully run
2 qubits, pi  2.0
Job Status: job has successfully run
3 qubits, pi  4.0
Job Status: job has successfully run
4 qubits, pi  2.6666666666666665
Job Status: job has successfully run
5 qubits, pi  3.2
Job Status: job has successfully run
6 qubits, pi  3.2
Job Status: job has successfully run
```

```
7 qubits, pi  3.2
Job Status: job has successfully run
8 qubits, pi  3.1219512195121952
Job Status: job has successfully run
9 qubits, pi  3.1604938271604937
Job Status: job has successfully run
10 qubits, pi  3.1411042944785277
Job Status: job has successfully run
11 qubits, pi  3.1411042944785277
Job Status: job has successfully run
12 qubits, pi  3.1411042944785277
```

Plotting the results

1.3.1 When run on a simulator

[9]:
```
plt.plot(qubits, [pi]*len(qubits), '--r')
plt.plot(qubits, pi_estimates, '.-', markersize=20)
plt.xlim([1.5, 14.5])
plt.ylim([1.5, 4.5])
plt.legend(['$\pi$', 'estimate of $\pi$'])
plt.xlabel('Number of qubits', fontdict={'size':20})
plt.ylabel('$\pi$ and estimate of $\pi$', fontdict={'size':20})
plt.tick_params(axis='x', labelsize=8)
plt.tick_params(axis='y', labelsize=8)
plt.show()
```


π
estimate of π
π and estimate of π
Number of qubits
4.5
4.0
3.5
3.0
2.5
2.0
1.5
2
4
6
8
10
12
14