Quantum PCA to price financial derivates related to DAX index

We will implement an effective qPCA (https://arxiv.org/abs/1307.0401) algorithm to price financial derivites related to the DAX German stock index (https://en.wikipedia.org/wiki/DAX) based on the paper Towards Pricing Financial Derivatives with an IBM Quantum Computer (https://arxiv.org/pdf/1904.05803.pdf). The objective of this algorithm is to approximate a square matrix M of dimension n with a new ρ matrix which will have rank $r << n$, so we need to calculate efficiently the r greatest eigenvalues of the original matrix M as then $\rho = \sum_{i=1}^{r} \lambda_i |u_i\rangle \langle u_i|$ with λ_i being the eigenvalues.

Calculating the covariance matrix

We first calculate the covariance matrix from the data related to the DAX, this is the matrix we want to reduce calculating its principal components.

In [1]:

```
import pandas as pd
import numpy as np

url = "https://raw.githubusercontent.com/ibonreinoso/qiskit-hackathon-bilbao-19/master/DAX_I
data = pd.read_csv(url, sep=';')
data = data.drop(['wkn_500340'], axis = 1)
data = data.loc[:,['wkn_515100', 'wkn_575200']]

covariance_matrix = np.cov(data.values.T)
```

In [2]:

```
print(covariance_matrix)
```

```
[[ 67.38376849  97.4907718 ]
 [ 97.4907718  152.27294829]]
```

Classical approach to the problem

Once the covariance matrix M has been obtained, in a classical PCA algorithm we will have to calculate the eigenvalues. This can be done by calculating the characteristic polynomial (https://en.wikipedia.org/wiki/Characteristic_polynomial) as it follows:

$$P(\lambda) = det(M - \lambda Id)$$

Where the eigenvalues are the values λ such that $P(\lambda) = 0$. The complexity of this operations are bounded by the complexity cost of computing a determinant of the covariance matrix. For our particular case, eigenvalues can be obtained classically as it follows:

In [3]:

```
# Calculate the characteristic polynomial of the covariance matrix
polynomial = np.poly(covariance_matrix)
# Calculate the roots of the polynomial which are the eigenvalues
eigenvalues = np.roots(polynomial)

print(eigenvalues)
```

```
[216.15800524   3.49871154]
```

As you might have noticed, there is an eigenvalue λ_{max} which greater than the other and also $\lambda_{max} = \lambda_1 >> \lambda_2$. Note that this is important in order to carry out effectively a PCA.

First step into the quantum model: calculate a unitary matrix

We could instead of calculating the determinant of the matrix, which is highly costly, use quantum computing to outperform the classical approach. To achieve this we will need to modify the covariance matrix so it is turned into an unitary matrix which fits as a quantum gate. This quantum gate will be used perform operations which will help us to obtain the eigenvalues. Later on we will represent the eigenvalues with qubits, so we need them to be in the range $[0, 1]$. To achieve this, we will normalize the covariance matrix with respect to its trace (in a $N \times N$ matrix the sum of all the eigenvalues equals the trace, so if we normalize with respect to its trace we will end up with eigenvalues falling in the range we wanted to).

In [4]:

```
trace_normalize_matrix = covariance_matrix /np.matrix.trace(covariance_matrix)

print(trace_normalize_matrix)
```

```
[[0.30676853 0.44383242]
 [0.44383242 0.69323147]]
```

Now, we have to create an unitary matrix U from our normalize matrix N. This can be done by just taking the complex exponential value of the matrix:

$$U = e^{2\pi i N}$$

In [5]:

```
from scipy.linalg import expm

unitary = expm(2*1j*np.pi*trace_normalize_matrix)

print(unitary)
```

```
[[9.94996262e-01+0.03988286j 2.19878137e-16-0.09160674j]
 [2.24840803e-16-0.09160674j 9.94996262e-01-0.03988286j]]
```

Eigenvectors and eigenvalues

As when using PCA we assume that there is an eigenvalue $\lambda_{max} = \lambda_1 >> \lambda_2$ then we can use a number of qubits to approximate the value of this eigenvalues. In this notebook we will focus on the implementation with just two qubits, but further investigation on how to generalize this model could be carried out. We will initialize a qubit in a random state and try to change its value until it reaches the value of the eigenvector which corresponds to the eigenvalue λ_{max}. So, the initial state of the vector will be:

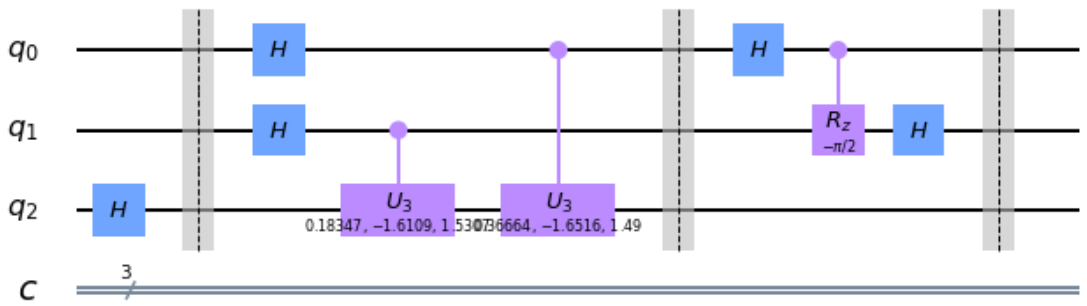
$$|b\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$$

If the value of $|b\rangle$ was the actual eigenvector of the eigenvalue λ_{max} we are over, but it is not. Nevertheless, we can use the Quantum Amplitude Estimation (https://arxiv.org/abs/quant-ph/0005055) to estimate the eigenvector of the matrix U. Once this calculation has been performed the data can be retrived by using the Quantum Fourier Transform (https://en.wikipedia.org/wiki/Quantum_Fourier_transform). This process can be repited in order to obtain a more precise value of the eigenvalue. Here we present this approach step by step:

In [9]:

```
# We will draw the first iteration circuit:
from qiskit import *

num_qubits_eigenvalue = 2
num_qubits_eigenvector = 1
num_qubits = num_qubits_eigenvalue + num_qubits_eigenvector

quantum_circuit = QuantumCircuit(num_qubits, num_qubits)

# Create the random state for |b>
quantum_circuit.h(2)
quantum_circuit.barrier()


# We perform quantum amplitude estimation with the unitary matrix U
quantum_circuit.h(0)
quantum_circuit.h(1)
# Calculate the gate which relates to the unitary matrix
(th1, ph1, lam1) = qiskit.quantum_info.synthesis.two_qubit_decompose.euler_angles_1q(expm(2*
quantum_circuit.cu3(th1, ph1, lam1, 1, 2)
# Calculate the gate which relates to the unitary matrix applied two times, which is necessa
(th2, ph2, lam2) = qiskit.quantum_info.synthesis.two_qubit_decompose.euler_angles_1q(expm(2*
quantum_circuit.cu3(th2, ph2, lam2, 0, 2)
quantum_circuit.barrier()

# Now we perform the quantum fourier transform in the first two qubits
quantum_circuit.h(0)
quantum_circuit.crz(-np.pi/2,0,1)
quantum_circuit.h(1)
quantum_circuit.barrier()

quantum_circuit.draw(output='mpl')
```

Out[9]:



A deep look into the iterative process

After the Quantum Fourier Transformation is performed, the state of the whole system will be (it is remarkable that this state depends on the initial random state $|b\rangle$ so we explicitly denote it):

$$|\Phi_b\rangle = \sum_{i=0}^{3} \frac{1}{\sqrt{2}} |\lambda_i\rangle \otimes |u_i\rangle$$

We obtain a state where eigenvectors and eigenvalues are entangled. So, if one of the eigenvalues is greater than the other, then by projecting in the related eigenvalue based component of the state $|\Phi_b\rangle$ one can efficiently obtain the value of the eigenvector. To accomplish it, we now conduct a measurement and calculate the data projected to the state $|11\rangle$ which matches definitely to the biggest eigenvalue.

After the first iteration results are showed below:

In [7]:

```
# We perform some measurements on the circuit
quantum_circuit.measure([0,1,2],[0,1,2])

# Let execute the circuit on the quantum simulator
results = execute(quantum_circuit, backend=BasicAer.get_backend('qasm_simulator'), shots=10

# Count the states related to the 11 projection
denominator_result = results['111'] + results['011']

# Calculate the coefficient of the eigenvector
alpha1 = np.sqrt(results['011'] / denominator_result)
alpha2 = np.sqrt(results['111'] / denominator_result)

# Eigenvector coefficents are showed
print(alpha1, alpha2)
```

```
0.31622776601683794 0.9486832980505138
```

A complete implementation of the qPCA

We are now able to implement the Quantum Principal Component Analysis algorithm by taking the value calculated at each iteration and reusing it as the initial quantum state of the random vector $|b\rangle$:

In [8]:

```
# We initialize the random vector b to H|0> = 1/sqrt(2) * (|0> + |1>)
state_vector = [1/np.sqrt(2), 1/np.sqrt(2)]

# We establish a limit to the number of iteration and a bound to the accuracy
limit_iteration = 5

shots_per_iteration = 8000

for i in range(0, limit_iteration):
    quantum_circuit = QuantumCircuit(num_qubits, num_qubits)
    quantum_circuit.initialize(state_vector, num_qubits-1)
    quantum_circuit.h(0)
    quantum_circuit.h(1)
    (th1, ph1, lam1) = qiskit.quantum_info.synthesis.two_qubit_decompose.euler_angles_1q(ex
    quantum_circuit.cu3(th1, ph1, lam1, 1, 2)
    (th2, ph2, lam2) = qiskit.quantum_info.synthesis.two_qubit_decompose.euler_angles_1q(ex
    quantum_circuit.cu3(th2, ph2, lam2, 0, 2)
    quantum_circuit.h(0)
    quantum_circuit.crz(-np.pi/2,0,1)
    quantum_circuit.h(1)
    quantum_circuit.measure([0,1,2],[0,1,2])
    results = execute(quantum_circuit, backend=BasicAer.get_backend('qasm_simulator'), shot
    denominator_result = results['111'] + results['011']
    alpha1 = np.sqrt(results['011'] / denominator_result)
    alpha2 = np.sqrt(results['111'] / denominator_result)
    new_state = [alpha1, alpha2]
    state_vector = new_state

# Print the state vector result which is an approximation of the eigenvector
print("Eigenvector: ", state_vector)
```

```
Eigenvector:  [0.5773502691896257, 0.816496580927726]
```