

```
In [1]: # Importing standard Qiskit libraries and configuring account
from qiskit import *
from qiskit import IBMQ
from qiskit.compiler import transpile, assemble
from qiskit.tools.jupyter import *
from qiskit.visualization import *

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import style
style.use('bmh')
```

```
In [2]: # Function for plotting the image using matplotlib
def plot_image(img, title: str):
    plt.title(title)
    plt.xticks(range(img.shape[0]))
    plt.yticks(range(img.shape[1]))
    plt.imshow(img, extent=[0, img.shape[0], img.shape[1], 0], cmap='viridis')
    plt.show()
```

```
In [3]: # Convert the raw pixel values to probability amplitudes
def amplitude_encode(img_data):

    # Calculate the RMS value
    rms = np.sqrt(np.sum(np.sum(img_data**2, axis=1)))

    # Create normalized image
    image_norm = []
    for arr in img_data:
        for ele in arr:
            image_norm.append(ele / rms)

    # Return the normalized image as a numpy array
    return np.array(image_norm)
    print(image_norm)
```

```
In [4]: from PIL import Image
style.use('default')

image_size = 256      # Original image-width
image_crop_size = 32   # Width of each part of image for processing

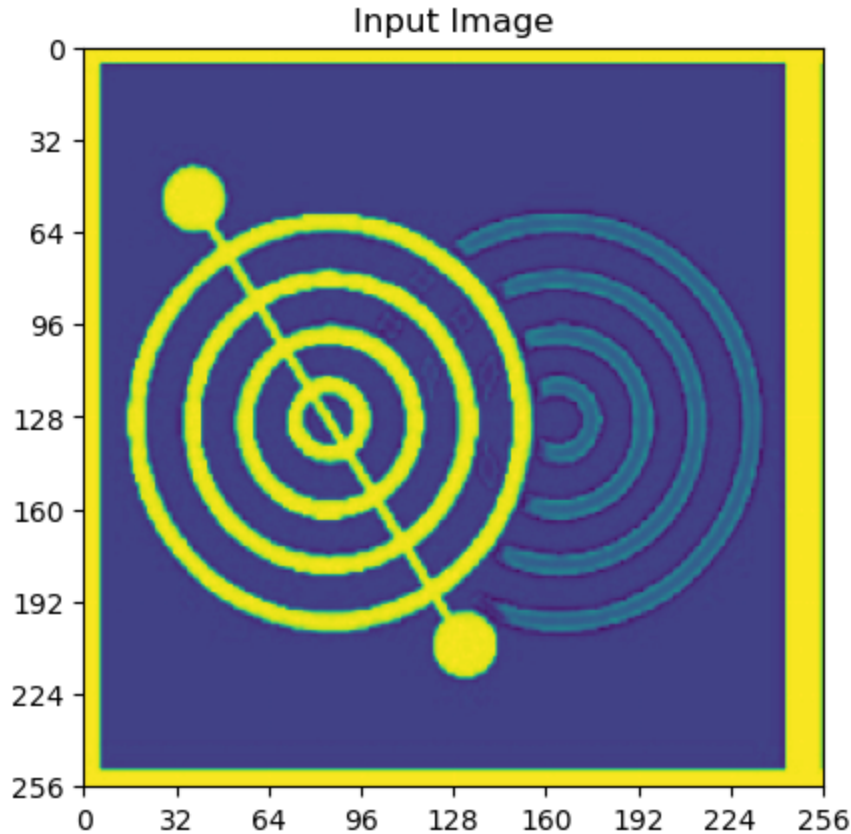
# Load the image from filesystem
image_raw = np.array(Image.open(r'C:\Users\Dell\Downloads\circle.jpg'))
print('Raw Image info:', image_raw.shape)
print('Raw Image datatype:', image_raw.dtype)
# Convert the RGB component of the image to B&W image, as a numpy (uint8) array
image = []
for i in range(image_size):
    image.append([])
    for j in range(image_size):
        image[i].append(image_raw[i][j][0] / 255)

image = np.array(image)
print('Image shape (numpy array):', image.shape)

plt.title('Input Image')
plt.xticks(range(0, image.shape[0]+1, 32))
plt.yticks(range(0, image.shape[1]+1, 32))
```

```
plt.imshow(image, extent=[0, image.shape[0], image.shape[1], 0], cmap='viridis')
plt.show()
```

Raw Image info: (256, 256, 3)
 Raw Image datatype: uint8
 Image shape (numpy array): (256, 256)



```
In [5]: # Initialize some global variable for number of qubits
data_qb = 10
anc_qb = 1
total_qb = data_qb + anc_qb

# Initialize the amplitude permutation unitary
D2n_1 = np.roll(np.identity(2**total_qb), 1, axis=1)
print(D2n_1)

[[0. 1. 0. ... 0. 0. 0.]
 [0. 0. 1. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 1. 0.]
 [0. 0. 0. ... 0. 0. 1.]
 [1. 0. 0. ... 0. 0. 0.]
```

```
In [6]: new_image = image
for i in range(0, 256, 32):
    print(i)
    for j in range(0, 256, 32):
        image1 = image[i:i+32, j:j+32]
        image1_norm_h = amplitude_encode(image1)
        image1_norm_v = amplitude_encode(image1.T)
        # Create the circuit for horizontal scan
        qc_h = QuantumCircuit(total_qb)
        qc_h.initialize(image1_norm_h, range(1, total_qb))
        qc_h.h(0)
        qc_h.unitary(D2n_1, range(total_qb))
        qc_h.h(0)
        # display(qc_h.draw('mpl', fold=-1))
```

```

# Create the circuit for vertical scan
qc_v = QuantumCircuit(total_qb)
qc_v.initialize(image1_norm_v, range(1, total_qb))
qc_v.h(0)
qc_v.unitary(D2n_1, range(total_qb))
qc_v.h(0)
# display(qc_v.draw('mpl', fold=-1))

# Combine both circuits into a single list
circ_list = [qc_h, qc_v]

# Simulating the circuits
back = Aer.get_backend('statevector_simulator')
results = execute(circ_list, backend=back).result()
sv_h = results.get_statevector(qc_h)
sv_v = results.get_statevector(qc_v)
threshold = lambda amp: (amp > 1e-15 or amp < -1e-15)

# Selecting odd states from the raw statevector and
# reshaping column vector of size 64 to an 8x8 matrix
edge_scan_h = np.abs(np.array([1 if threshold(sv_h[2*i+1].real) else 0 for i in
edge_scan_v = np.abs(np.array([1 if threshold(sv_v[2*i+1].real) else 0 for i in

edge_scan_sim = edge_scan_h | edge_scan_v

for a in range (32):
    for b in range (32):
        new_image[i+a][j+b]=edge_scan_sim[a][b]

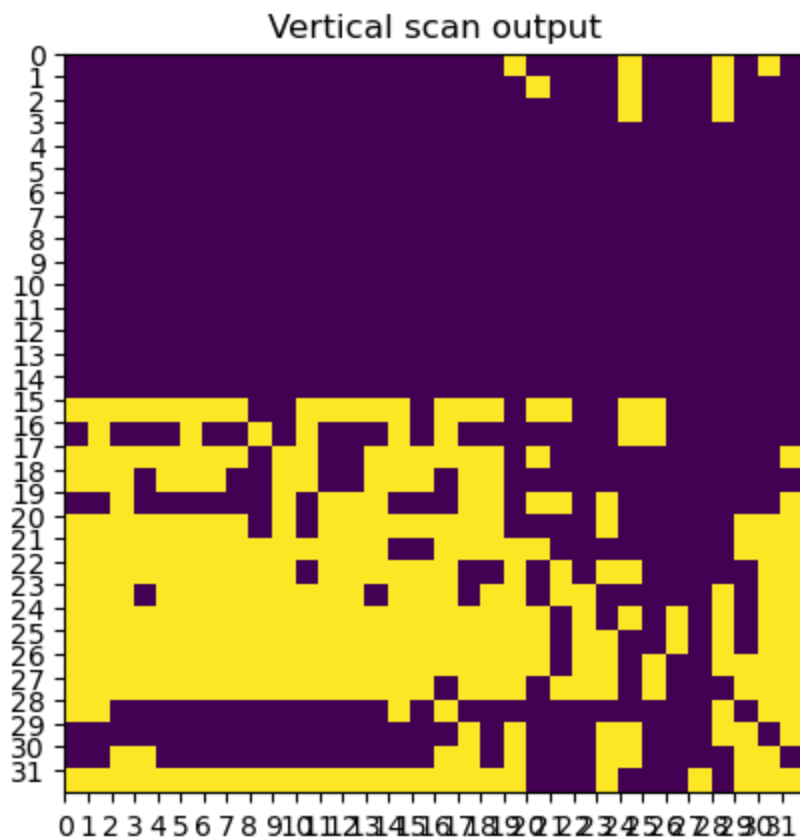
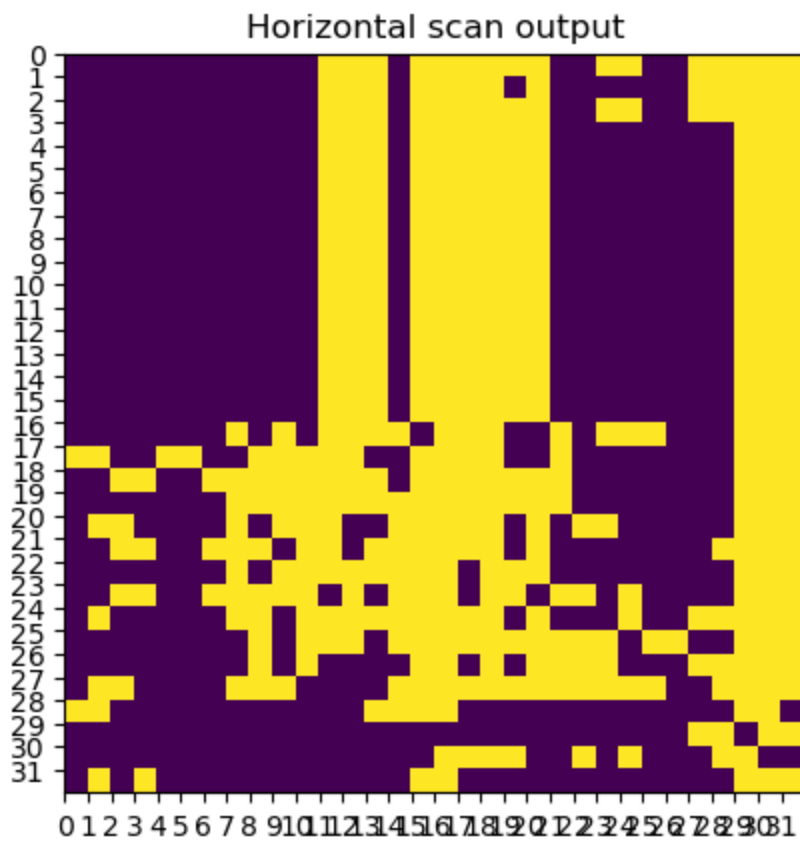
```

0
32
64
96
128
160
192
224

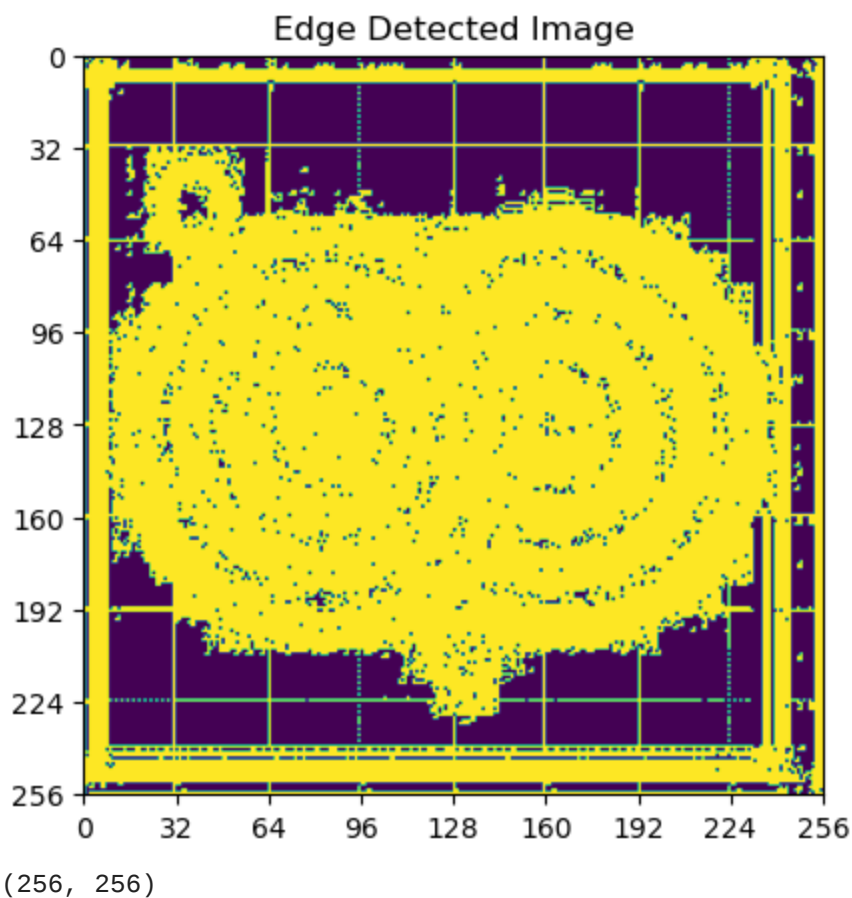
```

In [7]: # Plotting the Horizontal and vertical scans
plot_image(edge_scan_h, 'Horizontal scan output')
plot_image(edge_scan_v, 'Vertical scan output')

```



```
In [8]: plt.title('Edge Detected Image')
plt.xticks(range(0, new_image.shape[0]+1, 32))
plt.yticks(range(0, new_image.shape[1]+1, 32))
plt.imshow(new_image, extent=[0, new_image.shape[0], new_image.shape[1], 0], cmap='virid')
plt.show()
print(new_image.shape)
```



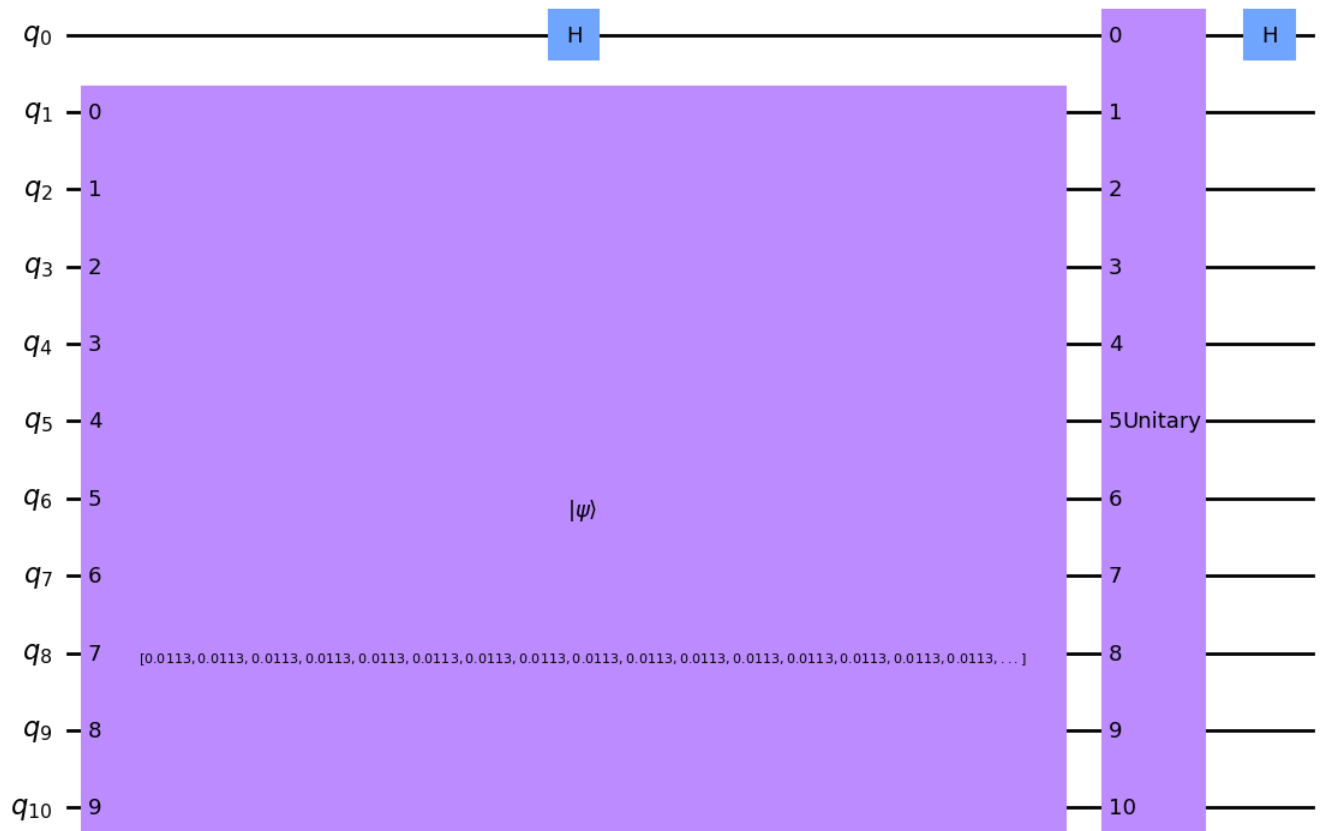
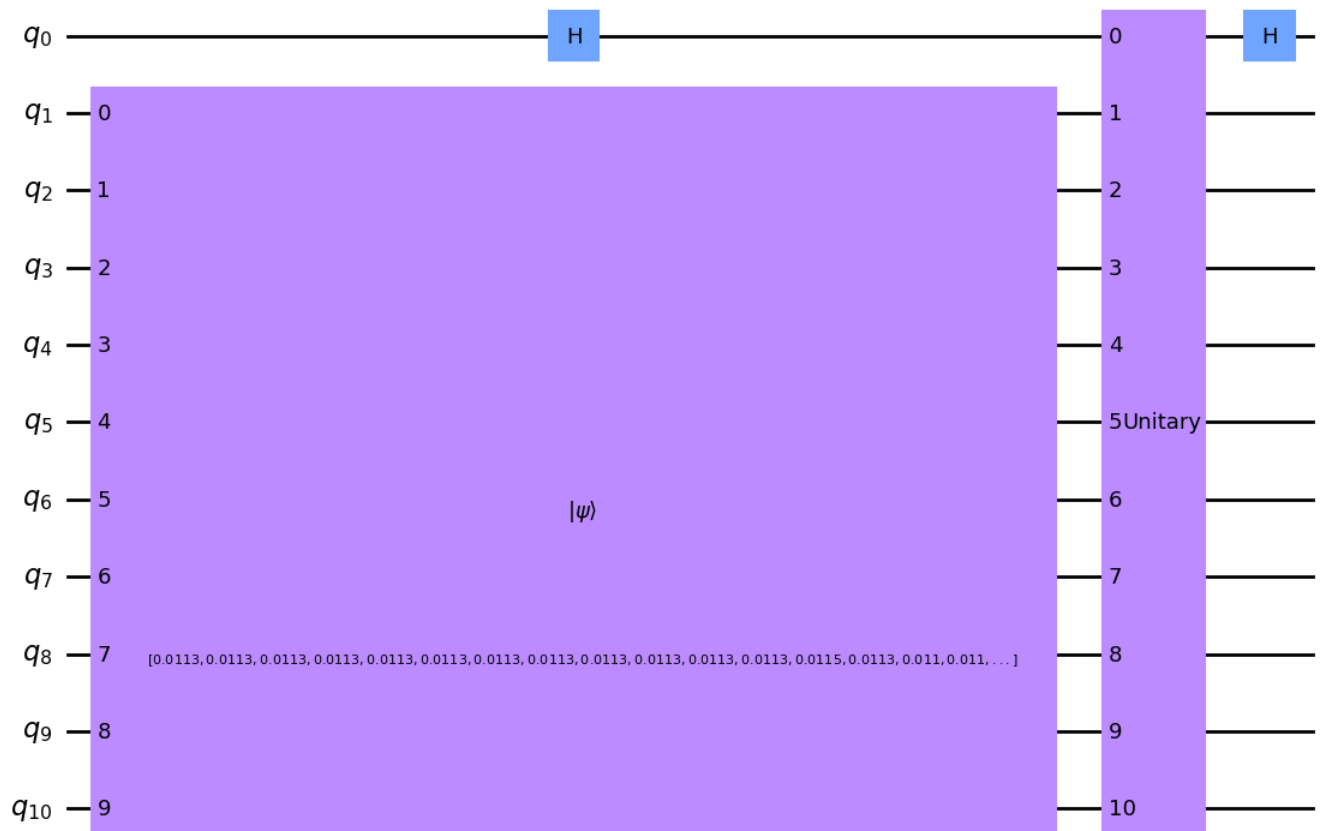
(256, 256)

In [9]:

```
# Create the circuit for horizontal scan
qc_h = QuantumCircuit(total_qb)
qc_h.initialize(image1_norm_h, range(1, total_qb))
qc_h.h(0)
qc_h.unitary(D2n_1, range(total_qb))
qc_h.h(0)
display(qc_h.draw('mpl', fold=-1))

# Create the circuit for vertical scan
qc_v = QuantumCircuit(total_qb)
qc_v.initialize(image1_norm_v, range(1, total_qb))
qc_v.h(0)
qc_v.unitary(D2n_1, range(total_qb))
qc_v.h(0)
display(qc_v.draw('mpl', fold=-1))

# Combine both circuits into a single list
circ_list = [qc_h, qc_v]
```



```
In [10]: # Create the circuit for horizontal scan
qc_small_h = QuantumCircuit(total_qb)
qc_small_h.x(1)
qc_small_h.h(0)
```

```
# Decrement gate - START
qc_small_h.x(0)
```

```

qc_small_h.cx(0, 1)
qc_small_h.ccx(0, 1, 2)
# Decrement gate - END

qc_small_h.h(0)
qc_small_h.measure_all()
display(qc_small_h.draw('mpl'))

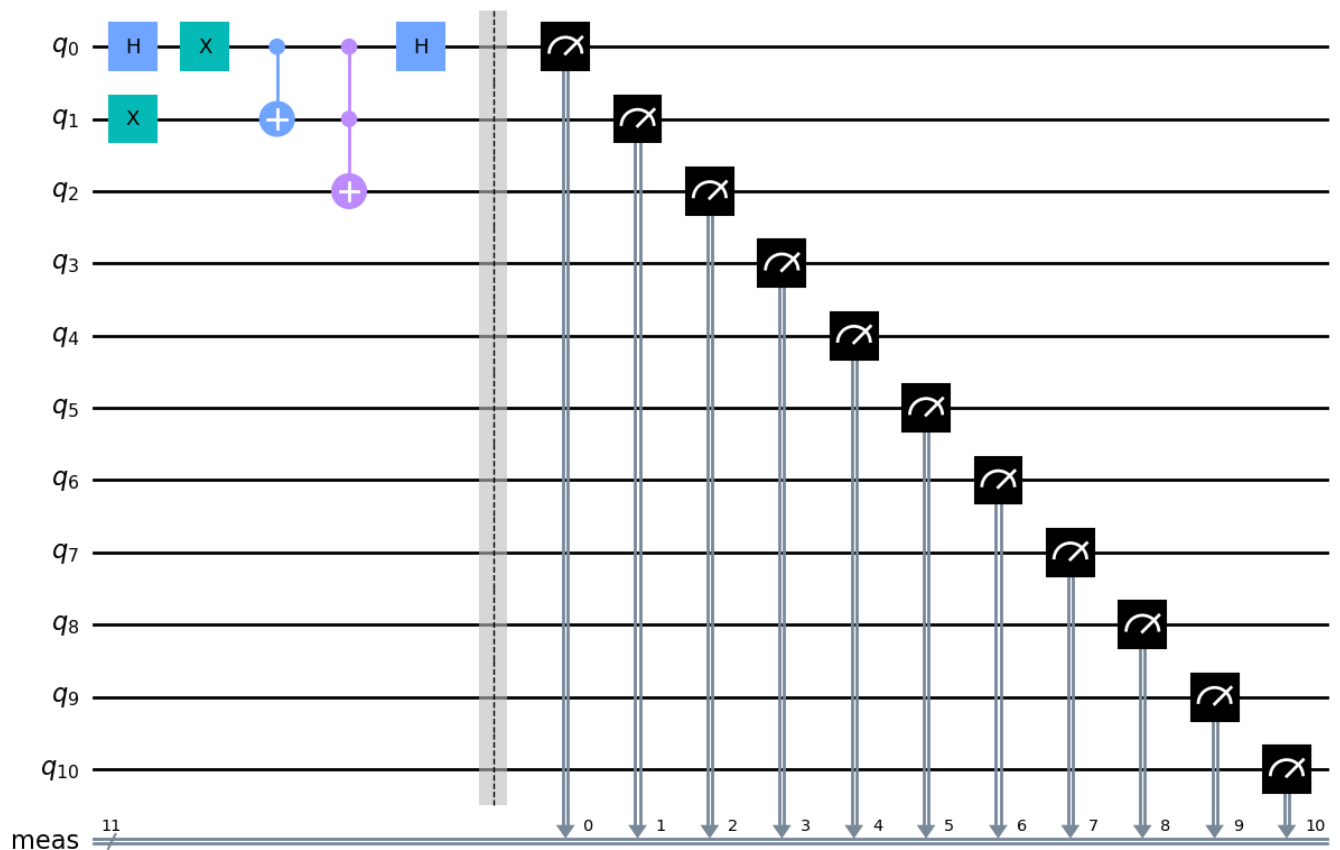
# Create the circuit for vertical scan
qc_small_v = QuantumCircuit(total_qb)
qc_small_v.x(2)
qc_small_v.h(0)

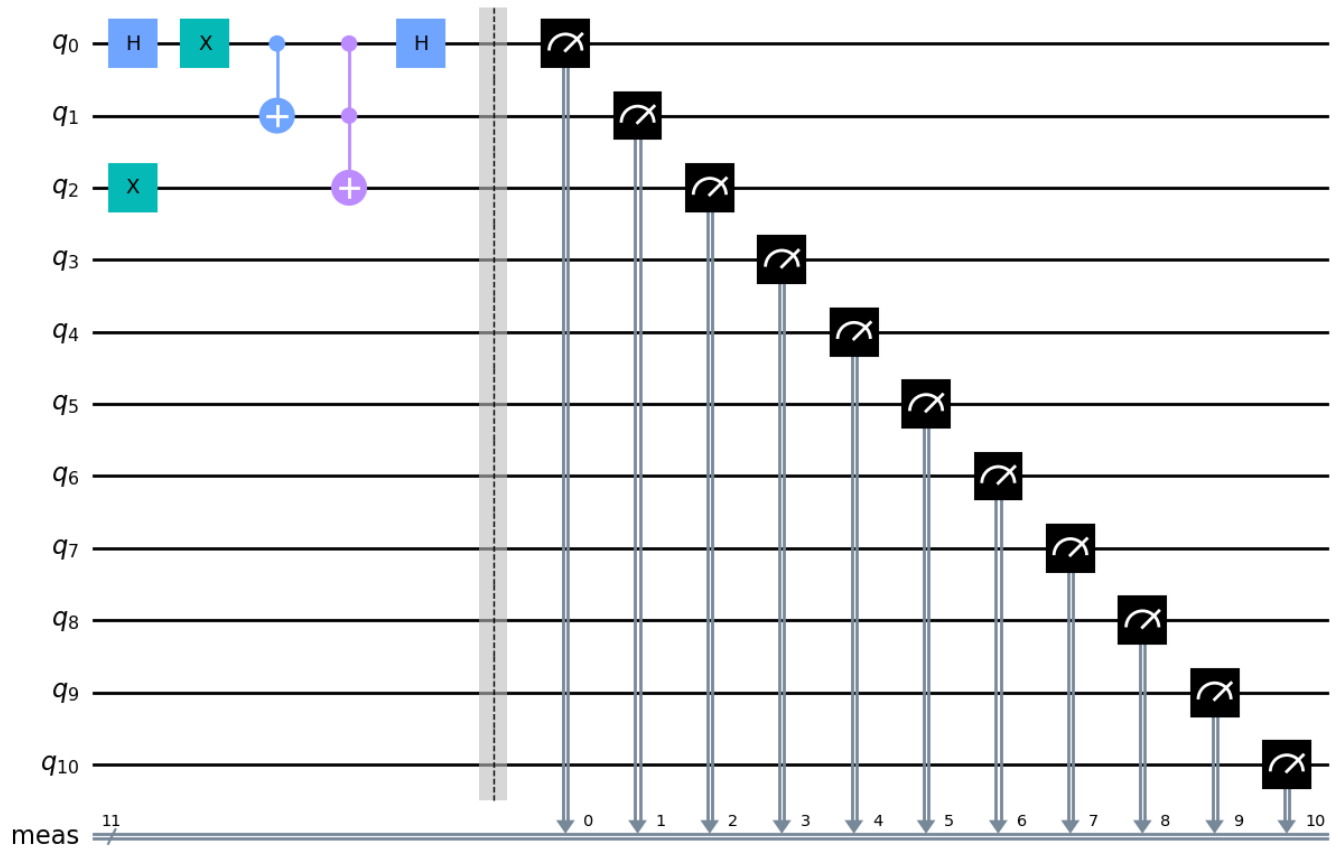
# Decrement gate - START
qc_small_v.x(0)
qc_small_v.cx(0, 1)
qc_small_v.ccx(0, 1, 2)
# Decrement gate - END

qc_small_v.h(0)
qc_small_v.measure_all()
display(qc_small_v.draw('mpl'))

# Combine both circuits into a single list
circ_list = [qc_small_h, qc_small_v]

```





In []:


```
In [1]: # Importing standard Qiskit libraries and configuring account
from qiskit import *
from qiskit import IBMQ
from qiskit.compiler import transpile, assemble
from qiskit.tools.jupyter import *
from qiskit.visualization import *

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import style
style.use('bmh')
```

```
In [2]: # Convert the raw pixel values to probability amplitudes
def amplitude_encode(img_data):

    # Calculate the RMS value
    rms = np.sqrt(np.sum(np.sum(img_data**2, axis=1)))

    if abs(rms) < 0.00001:
        return None

    # Create normalized image
    image_norm = []
    for arr in img_data:
        for ele in arr:
            image_norm.append(ele / rms)

    return np.array(image_norm)
```

```
In [3]: from PIL import Image
style.use('default')

image_size = 256      # Original image-width
image_crop_size = 32  # Width of each part of image for processing

# Load the image from filesystem
image_raw = np.array(Image.open(r'C:\Users\Dell\Downloads\circle.jpg'))
print('Raw Image info:', image_raw.shape)
print('Raw Image datatype:', image_raw.dtype)

# Convert the RGB component of the image to B&W image, as a numpy (uint8) array
image = []
for i in range(image_size):
    image.append([])
    for j in range(image_size):
        image[i].append(image_raw[i][j][0] / 255)

image = np.array(image)#[:, :2]
print('Image shape (numpy array):', image.shape)

# Function for plotting the image using matplotlib
def plot_image(img, title: str):
    #img[0][0] = 0
    plt.title(title)
    plt.xticks(range(0, image.shape[0]+1, 32))
    plt.yticks(range(0, image.shape[1]+1, 32))
    plt.imshow(img, extent=[0, img.shape[0], img.shape[1], 0], cmap='viridis')
    plt.show()
```

```

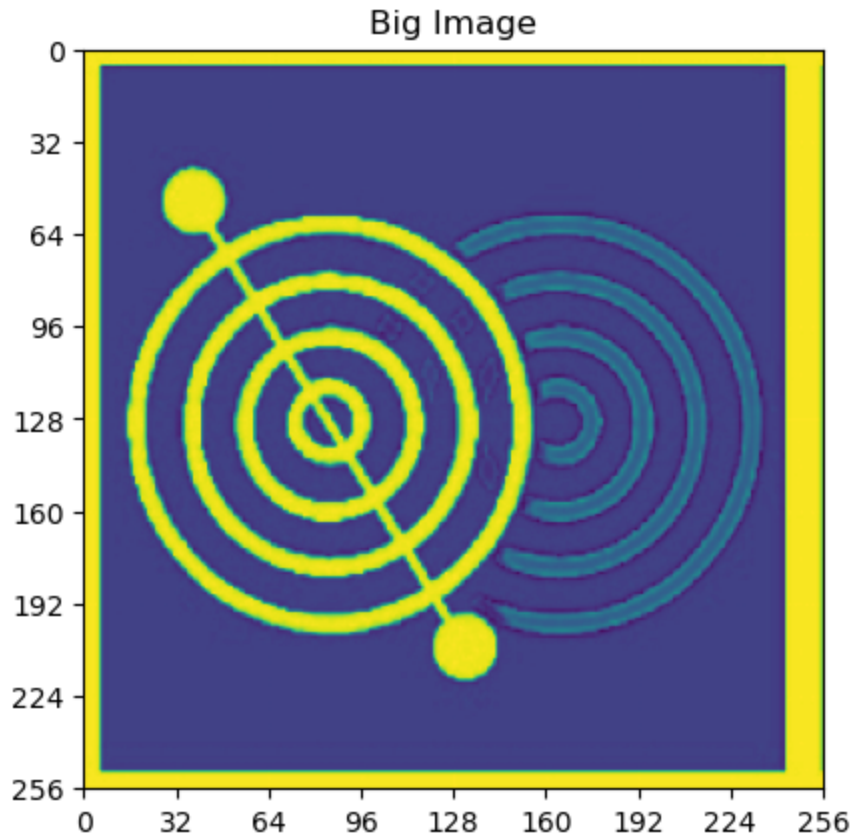
# Display the image
plt.title('Big Image')
plt.xticks(range(0, image.shape[0]+1, 32))
plt.yticks(range(0, image.shape[1]+1, 32))
plt.imshow(image, extent=[0, image.shape[0], image.shape[1], 0], cmap='viridis')
plt.show()

```

Raw Image info: (256, 256, 3)

Raw Image datatype: uint8

Image shape (numpy array): (256, 256)



In [4]: *#Color Quantization is the process of reducing number of colors in an image.*

```

import cv2
import numpy as np

def kmeans_color_quantization(image, clusters=8, rounds=1):
    h, w = image.shape[:2]
    samples = np.zeros([h*w], dtype=np.float32)
    count = 0

    for x in range(h):
        for y in range(w):
            samples[count] = np.float32(image[x][y])
            count += 1

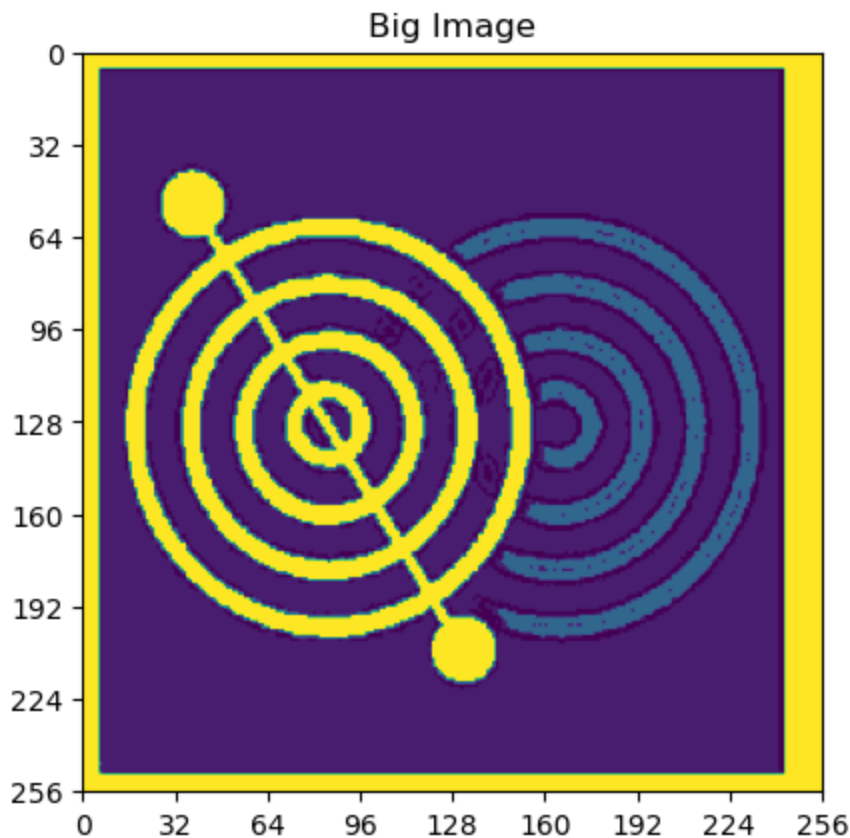
    compactness, labels, centers = cv2.kmeans(samples,
        clusters,
        None,
        (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10000, 0.0001),
        rounds,
        cv2.KMEANS_RANDOM_CENTERS)

    #centers = np.uint8(centers *)
    #print(list(labels))

```

```
return res.reshape((image.shape)), centers, labels
```

```
seg, centers, labels = kmeans_color_quantization(image, clusters=4)  
plot_image(seg, 'Big Image')
```



```
In [5]: unique, counts = np.unique(seg, return_counts=True)  
dict(zip(unique, counts))
```

```
Out[5]: {0.20465949: 4608, 0.26237568: 40651, 0.45362478: 5266, 0.9606226: 15011}
```

```
In [6]: enc = ["{0:b}".format(x) for x in labels.flatten()]  
unique, counts = np.unique(enc, return_counts=True)  
dict(zip(unique, counts))
```

```
Out[6]: {'0': 5266, '1': 15011, '10': 4608, '11': 40651}
```

```
In [18]: # Initialize some global variable for number of qubits  
data_qb = 6  
anc_qb = 1  
total_qb = data_qb + anc_qb  
  
# Initialize the amplitude permutation unitary  
D2n_1 = np.roll(np.identity(2**total_qb), 1, axis=1)
```

```
In [21]: import tqdm  
CHUNK_POWER = 3  
CHUNK_SIZE = 2 ** CHUNK_POWER  
  
#image = result  
w, h = image.shape  
  
chunks = []  
back = Aer.get_backend('statevector_simulator')  
result = image.copy()
```

```

for j in range(h // (CHUNK_SIZE - 1) + 1):
    offsetx = i * CHUNK_SIZE - i
    offsety = j * CHUNK_SIZE - j
    if w - offsetx < CHUNK_SIZE:
        offsetx = w - CHUNK_SIZE
    if h - offsety < CHUNK_SIZE:
        offsety = h - CHUNK_SIZE

    chunk = image[offsetx : offsetx + CHUNK_SIZE, offsety : offsety + CHUNK_SIZE]
    chunk_seg = seg[offsetx : offsetx + CHUNK_SIZE, offsety : offsety + CHUNK_SIZE]

    # Get the amplitude encoded pixel values
    # Horizontal: Original image
    chunk_norm_h = amplitude_encode(chunk)
    if chunk_norm_h is None or np.all(chunk_seg == chunk_seg[0][0]):
        edge_scan_h = np.abs(np.array([0.0 for i in range(2**data_qb)]).reshape(CHUNK_SIZE))
        edge_scan_v = np.abs(np.array([0.0 for i in range(2**data_qb)]).reshape(CHUNK_SIZE))
        result[offsetx : offsetx + CHUNK_SIZE, offsety : offsety + CHUNK_SIZE] = np.concatenate((edge_scan_h, edge_scan_v))
        continue

    chunk_norm_v = amplitude_encode(chunk.T)
    # Vertical: Transpose of Original image
    qc_h = QuantumCircuit(total_qb)
    qc_h.initialize(chunk_norm_h, range(1, total_qb))
    qc_h.h(0)
    qc_h.unitary(D2n_1, range(total_qb))
    qc_h.h(0)
    #display(qc_h.draw('mpl', fold=-1))

    # Create the circuit for vertical scan
    qc_v = QuantumCircuit(total_qb)
    qc_v.initialize(chunk_norm_v, range(1, total_qb))
    qc_v.h(0)
    qc_v.unitary(D2n_1, range(total_qb))
    qc_v.h(0)
    #display(qc_v.draw('mpl', fold=-1))

    # Combine both circuits into a single list
    circ_list = [qc_h, qc_v]
    # Simulating the circuits
    results = execute(circ_list, backend=back).result()
    sv_h = results.get_statevector(qc_h)
    sv_v = results.get_statevector(qc_v)

    # Defining a lambda function for
    # thresholding to binary values
    threshold = lambda amp: (amp > 1e-15 or amp < -1e-15)

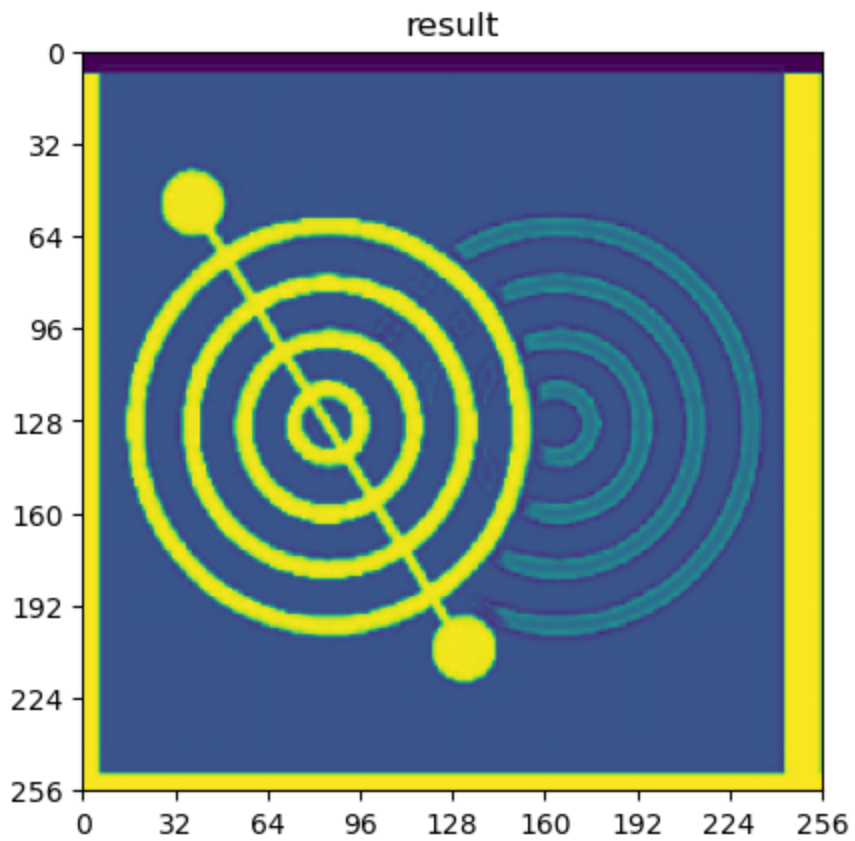
    # Selecting odd states from the raw statevector and
    # reshaping column vector of size 64 to an 8x8 matrix
    # edge_scan_h = np.abs(np.array([1 if threshold(sv_h[2*i+1].real) else 0 for i in range(2**data_qb)]))
    # edge_scan_v = np.abs(np.array([1 if threshold(sv_v[2*i+1].real) else 0 for i in range(2**data_qb)]))

    edge_scan_h = np.abs(np.array([sv_h[2*i+1].real for i in range(2**data_qb)]))
    edge_scan_v = np.abs(np.array([sv_v[2*i+1].real for i in range(2**data_qb)]))
    result[offsetx : offsetx + CHUNK_SIZE, offsety : offsety + CHUNK_SIZE] = np.sqrt(edge_scan_h**2 + edge_scan_v**2)
    #print(j)
# print(i)
if i % 10 == 0:
    plot_image(result, 'result')

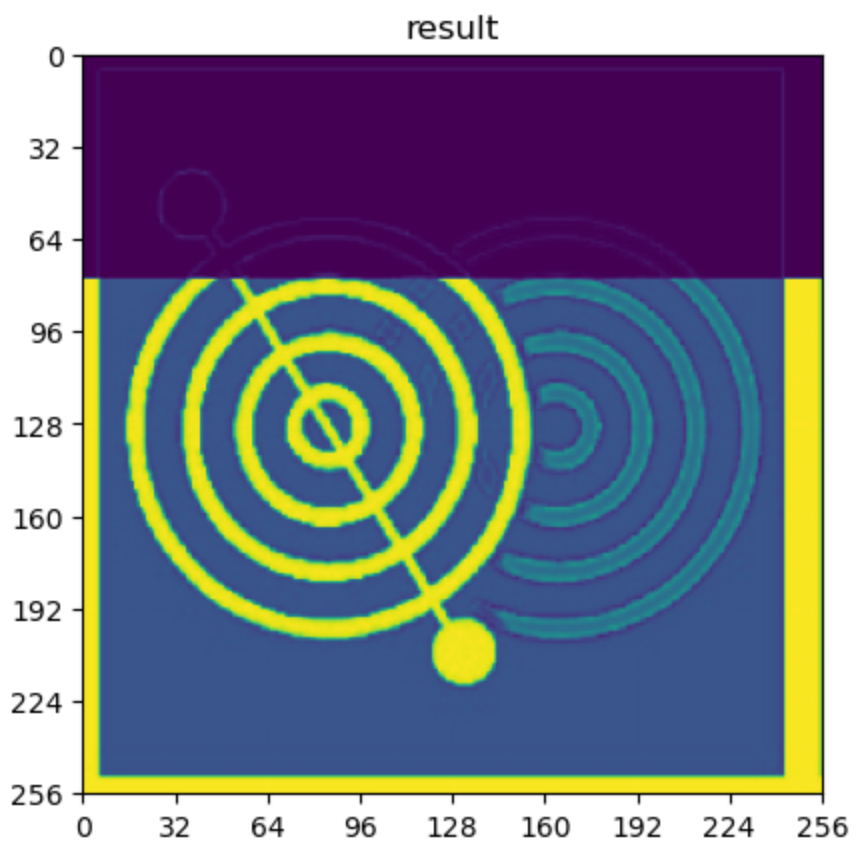
plot_image(result, 'result')

```

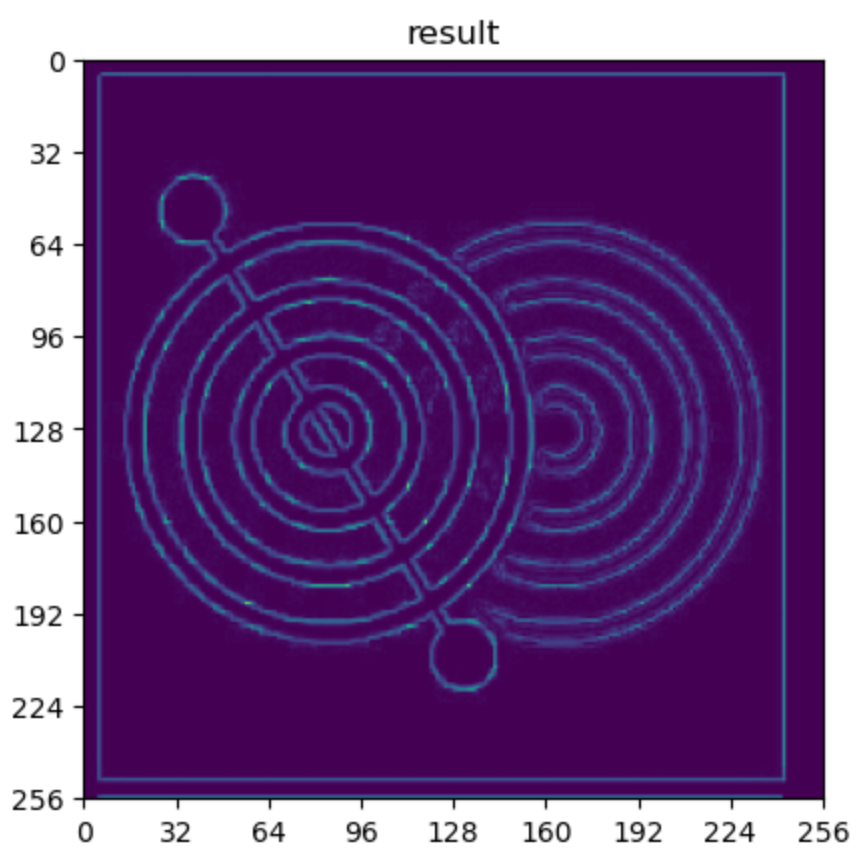
0%|
| 0/37 [00:00<?, ?it/s]



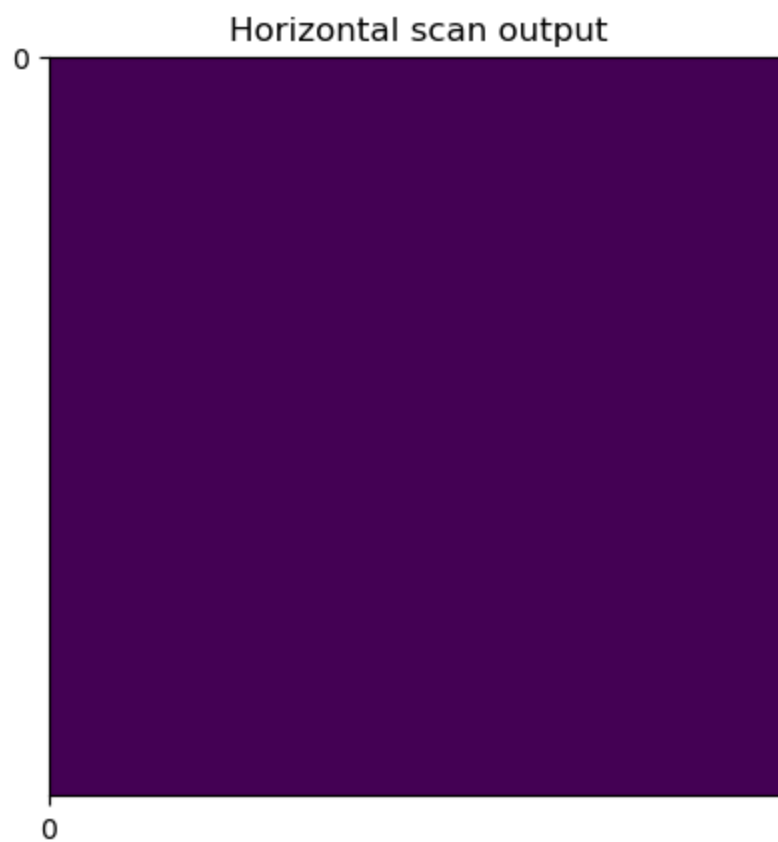
27%|
10/37 [00:02<00:08, 3.03it/s]

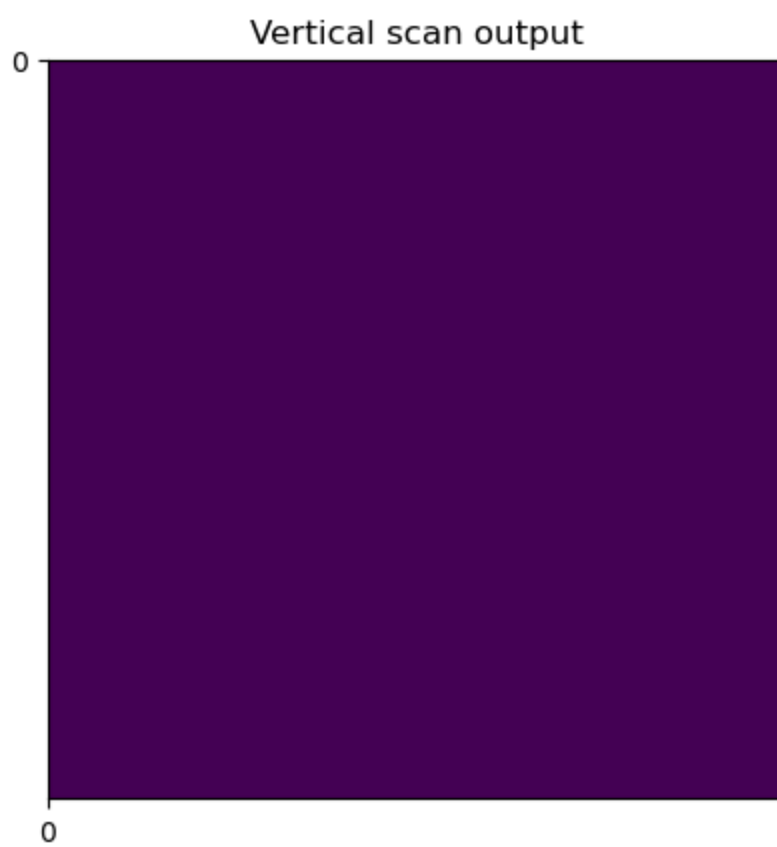


54%|
20/37 [00:09<00:11, 1.53it/s]

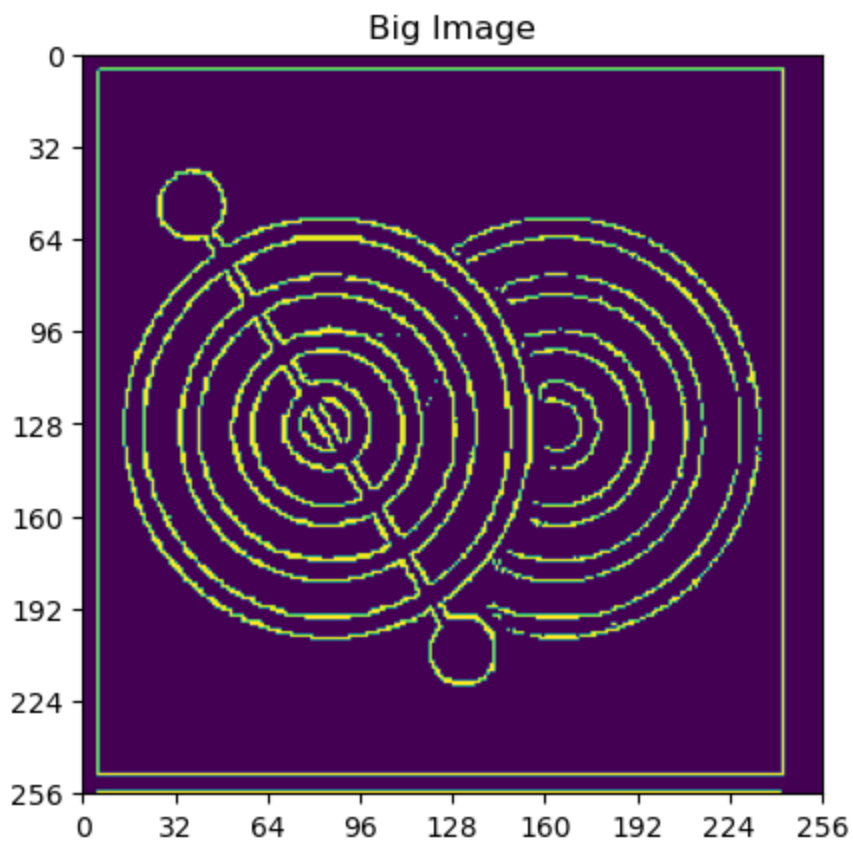


```
In [22]: # Plotting the Horizontal and vertical scans
plot_image(edge_scan_h, 'Horizontal scan output')
plot_image(edge_scan_v, 'Vertical scan output')
```

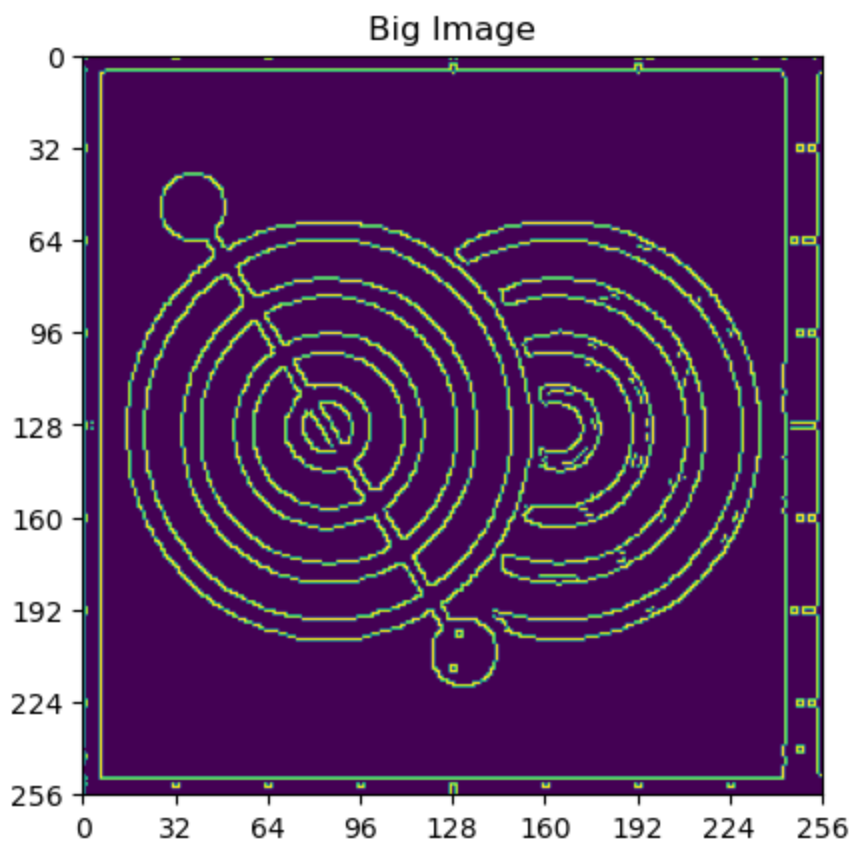




```
In [13]: resseg, centers, labels = kmeans_color_quantization(result, clusters=2)
         plot_image(resseg, 'Big Image')
```



```
In [14]: edges = cv2.Canny((image * 256).astype(np.uint8),100,200)
         plot_image(edges, 'Big Image')
```

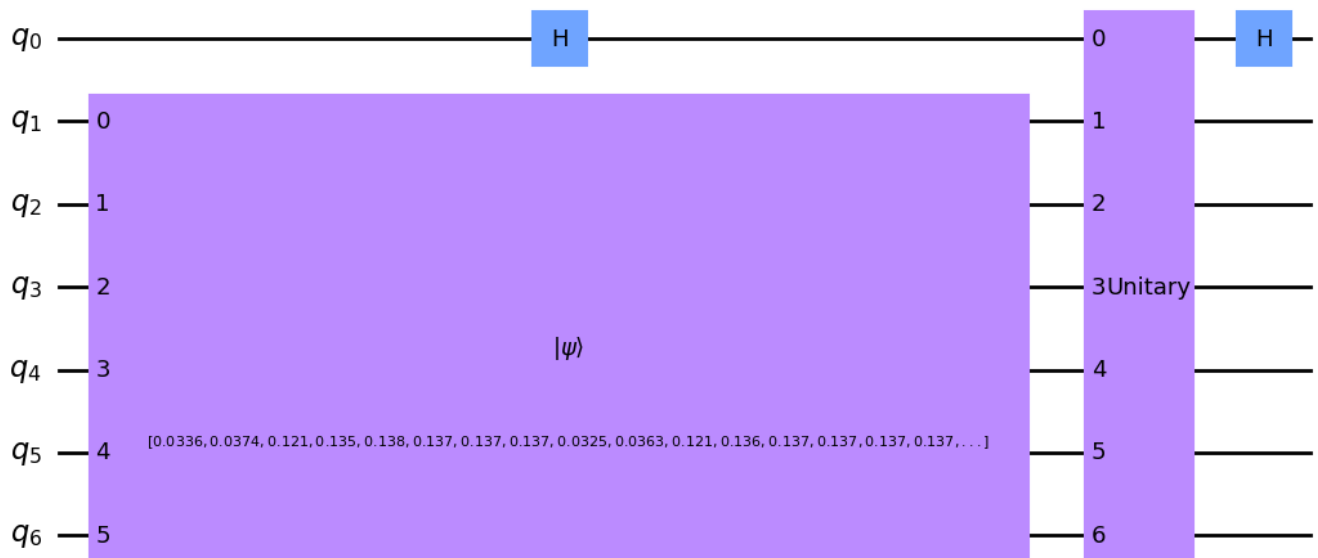
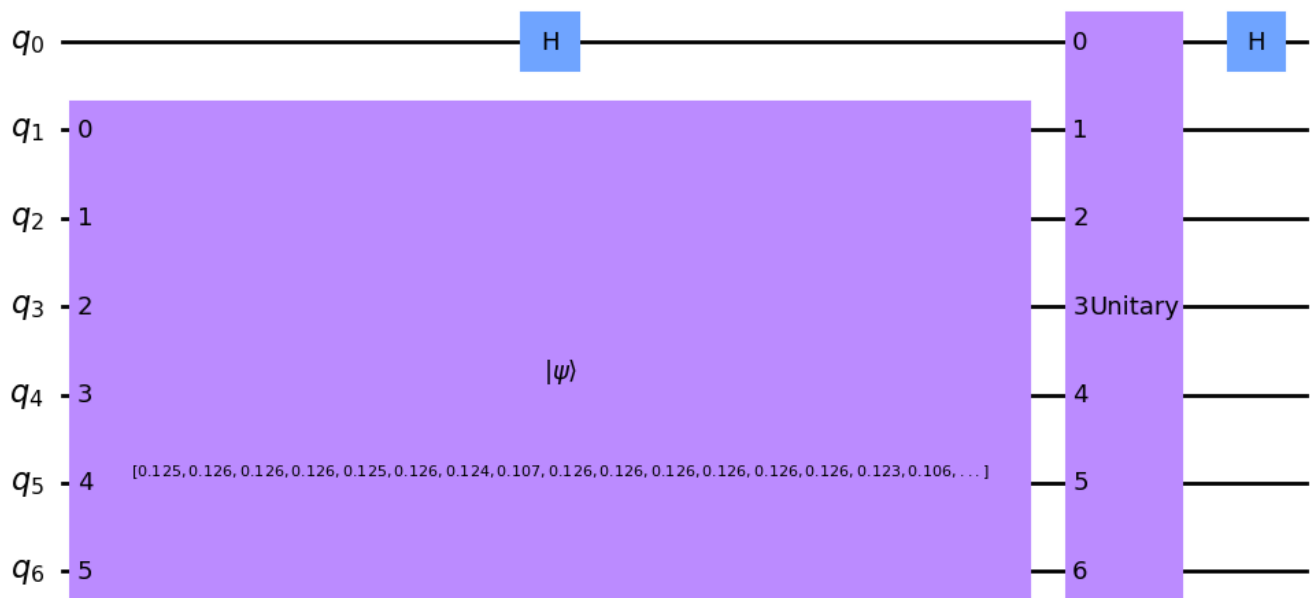



In [25]:

```
# Create the circuit for horizontal scan
qc_h = QuantumCircuit(total_qb)
qc_h.initialize(chunk_norm_h, range(1, total_qb))
qc_h.h(0)
qc_h.unitary(D2n_1, range(total_qb))
qc_h.h(0)
display(qc_h.draw('mpl', fold=-1))

# Create the circuit for vertical scan
qc_v = QuantumCircuit(total_qb)
qc_v.initialize(chunk_norm_v, range(1, total_qb))
qc_v.h(0)
qc_v.unitary(D2n_1, range(total_qb))
qc_v.h(0)
display(qc_v.draw('mpl', fold=-1))

# Combine both circuits into a single list
circ_list = [qc_h, qc_v]
```



In [26]: *# Create the circuit for horizontal scan*

```
qc_small_h = QuantumCircuit(total_qb)
qc_small_h.x(1)
qc_small_h.h(0)
```

Decrement gate - START

```
qc_small_h.x(0)
qc_small_h.cx(0, 1)
qc_small_h.ccx(0, 1, 2)
```

Decrement gate - END

```
qc_small_h.h(0)
qc_small_h.measure_all()
display(qc_small_h.draw('mpl'))
```

Create the circuit for vertical scan

```
qc_small_v = QuantumCircuit(total_qb)
qc_small_v.x(2)
qc_small_v.h(0)
```

Decrement gate - START

```
qc_small_v.x(0)
```

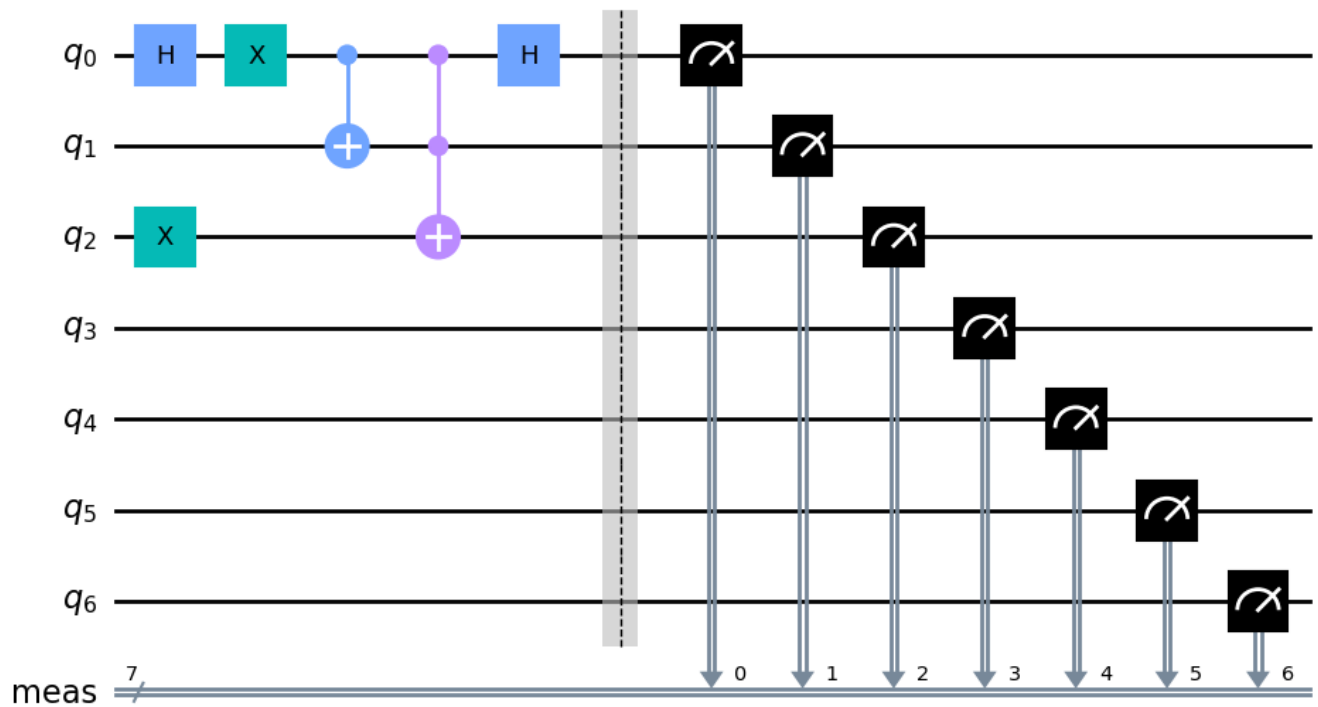
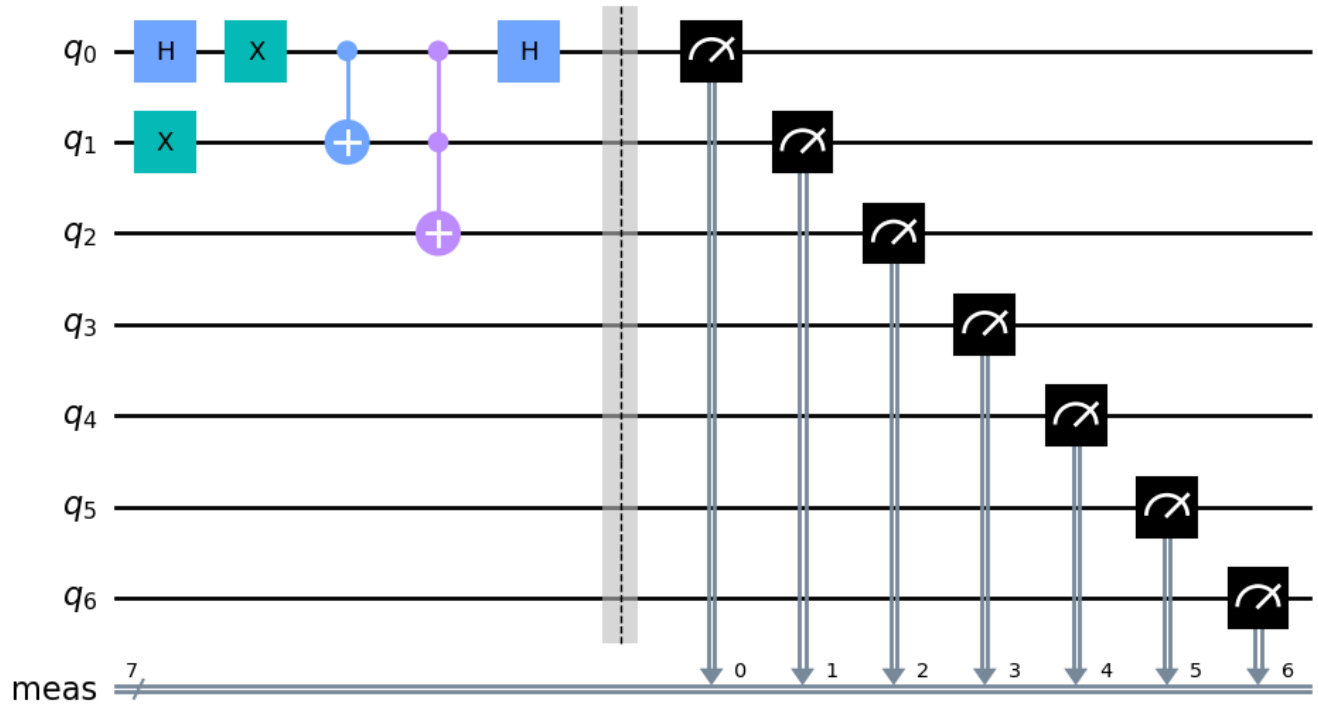
```

qc_small_v.cx(0, 1)
qc_small_v.ccx(0, 1, 2)
# Decrement gate - END

qc_small_v.h(0)
qc_small_v.measure_all()
display(qc_small_v.draw('mpl'))

# Combine both circuits into a single list
circ_list = [qc_small_h, qc_small_v]

```



In []:

Classical image to QPIE state

Let us take a sample image with four pixels which is arranged in 2D as follows:-

Here, the vector (I_0, I_1, I_2, I_3) (or $(I_{00}, I_{01}, I_{10}, I_{11})$ in binary representation of sub-script indices) represents color intensities (in 8-bit B&W color) of different pixels $(00, 01, 10, 11)$ represented as a 2D matrix to form a 2×2 classical image. The image can be represented in terms of it's pixel intensities as follows:

$$I = (I_{yx})_{N_1 \times N_2} \quad (1.2)$$

Therefore, (1.2) represent a 2-dimensional image made of $N_1 \times N_2$ pixels, where I_{yx} is the intensity of the pixel at the position (x, y) in the 2D image starting the coordinate axes from the top-left corner of the image.

Now, we need to represent these pixel intensities as the probability amplitudes of a particular quantum state. To do this, the pixel intensities should be normalized so that the sum of the squares of all the probability amplitudes is 1. For every c_i corresponding to respective I_{yx} , the normalization can be done as follows:-

$$c_i = \frac{I_{yx}}{\sqrt{\sum I_{yx}^2}} \quad (1.3)$$

After aforementioned normalization, the quantum-image looks like,

Finally, assigning the normalized pixel color values of each pixel P_i to the respective quantum state $|i\rangle$, we can write the image state $|\text{Img}\rangle$ as:-

$$|\text{Img}\rangle = c_0|00\rangle + c_1|01\rangle + c_2|10\rangle + c_3|11\rangle \quad (1.4)$$

OR generalizing for n -qubits, we have,

$$|\text{Img}\rangle = \sum_{i=0}^{2^n-1} c_i|i\rangle \quad (1.5)$$

Such a state can be very efficiently prepared just by using a few rotation and CNOT gates as can be seen in [7,

The case of the Hadamard gate

The Hadamard gate H has the following operation on the state of qubit,

$$\begin{aligned} |0\rangle &\rightarrow \frac{(|0\rangle + |1\rangle)}{\sqrt{2}} \\ |1\rangle &\rightarrow \frac{(|0\rangle - |1\rangle)}{\sqrt{2}} \end{aligned} \quad (2.1)$$

The QHED algorithm generalizes this action of H -gate and uses it for edge detection of an image.

Let us assume we have an N -pixel image. The pixels of the image can be numbered using binary bit-strings in the form of $|b_{n-1}b_{n-2}b_{n-3}\dots b_1b_0\rangle$ where $b_i \in \{0, 1\}$.

For two neighboring pixels, the bit-strings can be written as $|b_{n-1}b_{n-2}\dots b_1b_0\rangle$ and $|b_{n-1}b_{n-2}\dots b_1b_1\rangle$, i.e. only the least significant bit (LSB) is different for both of them. The corresponding pixel intensity values (normalized) can be written as $c_{b_{n-1}b_{n-2}\dots b_1b_0}$ and $c_{b_{n-1}b_{n-2}\dots b_1b_1}$ respectively. To simplify the notation, we will resort to the decimal representation of the bit-strings. Hence, the pixel values can be written as c_i and c_{i+1} in decimal representation.

Now, if we apply the H -gate to the LSB of an arbitrary size quantum register, we can represent the resultant unitary like,

Now, if we apply the H -gate to the LSB of an arbitrary size quantum register, we can represent the resultant unitary like,

$$I_{2^{n-1}} \otimes H_0 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 & 0 & 0 & \dots & 0 & 0 \\ 1 & -1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & 1 & \dots & 0 & 0 \\ 0 & 0 & 1 & -1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 & 1 \\ 0 & 0 & 0 & 0 & \dots & 1 & -1 \end{bmatrix} \quad (2.2)$$

Applying this unitary to a quantum register containing pixel values encoded using the QPIE representation $|\text{Img}\rangle = \sum_{i=0}^{N-1} c_i|i\rangle$, as shown in *eq. (1.1)*, we have,

$$(I_{2^{n-1}} \otimes H_0) \cdot \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_{N-2} \\ c_{N-1} \end{bmatrix} \rightarrow \frac{1}{\sqrt{2}} \begin{bmatrix} c_0 + c_1 \\ c_0 - c_1 \\ c_2 + c_3 \\ c_2 - c_3 \\ \vdots \\ c_{N-2} + c_{N-1} \\ c_{N-2} - c_{N-1} \end{bmatrix} \quad (2.3)$$

From the above resultant matrix (2.3), it is clearly visible that we now have access to the gradient between the pixel intensities of neighboring pixels in the form of $(c_i - c_{i+1})$ where, i is even. Measuring the circuit conditioned on the LSB being in state $|1\rangle$, we can obtain the gradients through statistical analysis.

This process results in the detection of horizontal boundaries between the even-pixels-pairs (0 & 1, 2 & 3, and so on). For detecting horizontal boundaries between odd-pixel-pairs (1 & 2, 3 & 4, etc.), we can perform an amplitude permutation on the quantum register to convert the amplitude vector $(c_0, c_1, c_2, \dots, c_{N-1})^T$ to $(c_1, c_2, c_3, \dots, c_{N-1}, c_0)^T$, and then applying the H -gate and measuring the quantum register conditioned on LSB being $|1\rangle$.

However, we can make it more resource-efficient by using an additional auxiliary qubit!

A Variation of QHED (with an auxiliary qubit)

As discussed in the previous sub-section, we still have a quantum register with n -qubits ($n = \lceil \log_2 N \rceil$) for encoding the N -pixel image. However, in this case, we add an extra auxiliary qubit to the register which we can utilize to extend the QHED algorithm and perform computation on both even- and odd-pixel-pairs simultaneously.

Like the last time, we initialize to the state $|Img\rangle = (c_0, c_1, c_2, \dots, c_{N-2}, c_{N-1})^T$. However, the H -gate is now applied to the **auxiliary qubit** this time which is initialized to state $|0\rangle$.

This produces an $(n + 1)$ -qubit redundant image state which can be represented as,

- -

This produces an $(n + 1)$ -qubit redundant image state which can be represented as,

$$|Img\rangle \otimes \frac{(|0\rangle + |1\rangle)}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{bmatrix} c_0 \\ c_0 \\ c_1 \\ c_1 \\ c_2 \\ c_2 \\ \vdots \\ c_{N-2} \\ c_{N-2} \\ c_{N-1} \\ c_{N-1} \end{bmatrix} \quad (2.4)$$

Now, since we get the redundant probability amplitudes obtained in the resultant state in (2.4), we can define an amplitude permutation unitary as follows to transform the amplitudes into a structure which will make it easier to calculate the image gradients further ahead,

$$D_{2^{n+1}} = \begin{bmatrix} 0 & 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 \\ 1 & 0 & 0 & 0 & \dots & 0 & 0 \end{bmatrix} \quad (2.5)$$

The above unitary corresponds to a **Decrement gate**. Hence, we can efficeintly decompose this unitary into a set of single- and multi-controlled-X rotations on a register of multiple qubits as shown by Fijany and Williams in [9] and Gidney in [10].

Applying the above decrement gate unitary $D_{2^{n+1}}$ to the redundant image state, we can transform the state $(c_0, c_0, c_1, c_1, c_2, c_2, \dots, c_{N-2}, c_{N-2}, c_{N-1}, c_{N-1})^T$ to the new redundant image state $(c_0, c_1, c_1, c_2, c_2, c_3, \dots, c_{N-2}, c_{N-1}, c_{N-1}, c_0)^T$.

Now again if we apply the H -gate to the auxiliary qubit, we obtain the gradients for both even- and odd-pixel-pairs at the same time like so,

Now again if we apply the H -gate to the auxiliary qubit, we obtain the gradients for both even- and odd-pixel-pairs at the same time like so,

$$(I_{2^n} \otimes H) \cdot \begin{bmatrix} c_0 \\ c_1 \\ c_1 \\ c_2 \\ c_2 \\ c_3 \\ \vdots \\ c_{N-2} \\ c_{N-1} \\ c_{N-1} \\ c_0 \end{bmatrix} \rightarrow \begin{bmatrix} c_0 + c_1 \\ c_0 - c_1 \\ c_1 + c_2 \\ c_1 - c_2 \\ c_2 + c_3 \\ c_2 - c_3 \\ \vdots \\ c_{N-2} + c_{N-1} \\ c_{N-2} - c_{N-1} \\ c_{N-1} + c_0 \\ c_{N-1} - c_0 \end{bmatrix} \quad (2.6)$$

Finally, measuring this state conditioned on the auxiliary qubit being in state $|1\rangle$, we will get the resultant horizontal gradient values $(c_i - c_{i+1})$ for all possible pairs of adjacent qubits.

NOTE: The above process provides a horizontal scan of the entire image which has edges detected in only the horizontal direction. To obtain the vertical scan edge detected image, we take the traspose of the image matrix and follow the same process again to obtain a vertical scan. These horizontal and vertical scans are then superimposed on each other using some classical post-processing to create the full edge detected image.

For the Horizontal and Vertical scan of the above image, we can see that the operations for state-preparation and decrement gate be written as follows:-

Horizontal Scan:

1. State preparation ($|Img\rangle = |01\rangle$): We can achieve this with a simple $[X(1)]$ operation.
2. Decrement gate: We can achieve this by a sequence of $[X(0), CX(0,1), CCX(0,1,2)]$ operations.

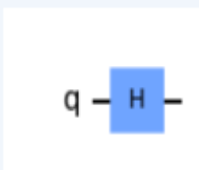
Vertical Scan:

1. State preparation ($|Img\rangle = |10\rangle$): We can achieve this with a simple $[X(2)]$ operation.
2. Decrement gate: We can achieve this by a sequence of $[X(0), CX(0,1), CCX(0,1,2)]$ operations.

GATES AND SYMBOLS USED IN EDGE DETECTION

HGate

The `HGate` class creates a single-qubit **H** gate. It performs a π rotation around the $X+Z$ axis. It also has the effect of changing the computational basis from $|0\rangle, |1\rangle$ to $|+\rangle, |-\rangle$ and vice-versa.

Signature:	Appearance:	Matrix:
<code>HGate(label=None)</code>		$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$

2. The X-Gate

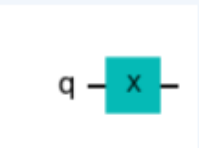
The X-gate is represented by the Pauli-X matrix:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = |0\rangle\langle 1| + |1\rangle\langle 0|$$

To see the effect a gate has on a qubit, we simply multiply the qubit's statevector by the gate. We can see that the X-gate switches the amplitudes of the states $|0\rangle$ and $|1\rangle$:

$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

The `XGate` class creates a single-qubit **X** gate that performs a π rotation around the X axis.

Signature:	Appearance:	Matrix:
<code>XGate(label=None)</code>		$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

3. CXGate

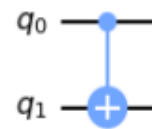
- The CXGate class creates a controlled-X gate, applying the X gate according to the control qubit state.
- The controlled-NOT gate, also known as the controlled-x (CX) gate, acts on a pair of qubits, with one acting as 'control' and the other as 'target'. It performs a NOT on the target whenever the control is in state $|1\rangle$. If the control qubit is in a superposition, this gate creates entanglement.

The CXGate class creates a controlled-X gate, applying the **X** gate according to the control qubit state.

Signature:

```
CXGate(label=None, ctrl_state=None)
```

Appearance:



4. Toffoli gate

- The Toffoli gate, also known as the double controlled-NOT gate (CCX), has two control qubits and one target. It applies a NOT to the target only when both controls are in state $|1\rangle$.
- The Toffoli gate with the Hadamard gate is a universal gate set for quantum computing.

The `CCXGate` class creates a three-qubit gate that has an **X** gate and two control qubits. This is also known as a Toffoli gate.

Signature:

```
CCXGate(label=None, ctrl_state=None)
```

Appearance:

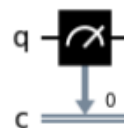


Measure

The `Measure` class creates a measurement instruction for measuring a quantum state in the computational basis, placing the binary result in a classical register (see [“Measuring a quantum circuit”](#)).

Signature: **Appearance:**

```
Measure()
```

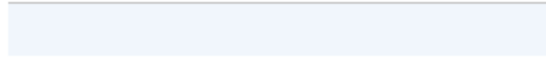


Barrier

The `Barrier` class creates a barrier instruction (see “[Creating a Barrier](#)”) with a given number of qubits. A barrier provides both visual and functional separation between gates on a wire in a quantum circuit.

Signature:

Appearance:



```
Barrier(num_qubits)
```



STEP 1 – IMPORTING THE STANDARD QISKIT LIBRARIES

For the purpose of this demonstration, we can assume that an image is nothing but a collection of pixel values represented as a numpy matrix in python.

```
# Importing standard Qiskit libraries and configuring account
from qiskit import *
from qiskit import IBMQ
from qiskit.compiler import transpile, assemble
from qiskit.tools.jupyter import *
from qiskit.visualization import *

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import style
style.use('bmh')
```

STEP 2 - PLOTTING THE IMAGE USING MATPLOTLIB

Matplotlib is an amazing visualization library in Python for 2D plots of arrays. Matplotlib is a multi-platform data visualization library built on NumPy arrays

The image module in matplotlib library is used for working with images in Python. The image module also includes two useful methods which are `imread` which is used to read images and `imshow` which is used to display the image.

```
# Function for plotting the image using matplotlib
def plot_image(img, title: str):
    plt.title(title)
    plt.xticks(range(img.shape[0]))
    plt.yticks(range(img.shape[1]))
    plt.imshow(img, extent=[0, img.shape[0], img.shape[1], 0], cmap='viridis')
    plt.show()
```

STEP 3 – CONVERT THE RAW PIXELS VALUES INTO PROBABILITY AMPLITUDE

Now that we have defined our image for testing, we can go ahead and encode the pixel intensities as probability amplitudes of different states of the system:-

```
# Convert the raw pixel values to probability amplitudes
def amplitude_encode(img_data):

    # Calculate the RMS value
    rms = np.sqrt(np.sum(np.sum(img_data**2, axis=1)))

    # Create normalized image
    image_norm = []
    for arr in img_data:
        for ele in arr:
            image_norm.append(ele / rms)

    # Return the normalized image as a numpy array
    return np.array(image_norm)
```

STEP 4 – LOADING THE IMAGE FROM FILESYSTEM AND SET THE ORIGINAL AND CROPPED SIZE IMAGE

```

from PIL import Image
style.use('default')

image_size = 128      # Original image-width
image_crop_size = 32   # Width of each part of image for processing

# Load the image from filesystem
image_raw = np.array(Image.open('camera-icon.png'))
print('Raw Image info:', image_raw.shape)
print('Raw Image datatype:', image_raw.dtype)

```

STEP 5 - INITIALIZE INITIALIZE THE NUMBER OF QUBITS AND THE AMPLITUDE PERMUTATION UNITARY

```

# Initialize some global variable for number of qubits
data_qb = 10
anc_qb = 1
total_qb = data_qb + anc_qb

# Initialize the amplitude permutation unitary
D2n_1 = np.roll(np.identity(2**total_qb), 1, axis=1)

```

STEP 5 – CONVERT THE RGB COMPONENT OF GIVEN INUTIMAGE INTO BLACK AND WHITE IMAGE OR GRAYSCALE AS –

they are common in image processing because using a grayscale image requires less available space and is faster, especially when we deal with complex computations.

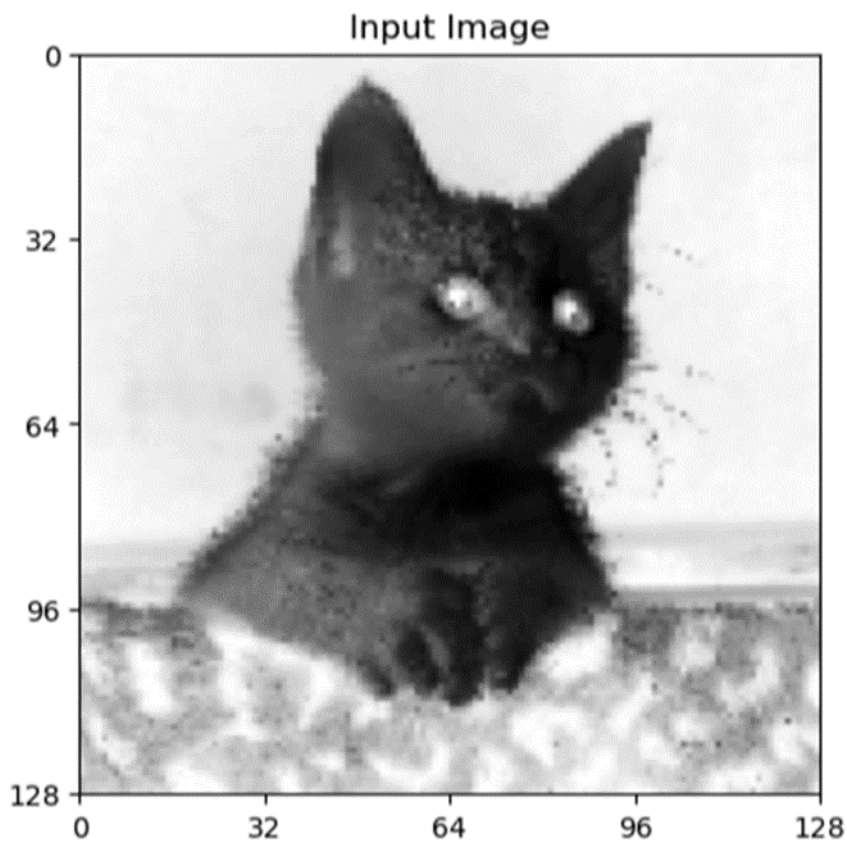

```

# Convert the RGB component of the image to B&W image, as a numpy (uint8) array
image = []
for i in range(image_size):
    image.append([])
    for j in range(image_size):
        image[i].append(image_raw[i][j][0] / 255)

image = np.array(image)
print('Image shape (numpy array):', image.shape)

plt.title('Input Image')
plt.xticks(range(0, image.shape[0]+1, 32))
plt.yticks(range(0, image.shape[1]+1, 32))
plt.imshow(image, extent=[0, image.shape[0], image.shape[1], 0], cmap='gray_r')
plt.show()

```



STEP 6 – CREATING THE CIRCUITS BOTH FOR HORIZONTAL AND VERTICAL AND MERGING THEM INTO SINGLE LIST

- Once, we have normalized the pixel values, converted them to probability amplitudes, and determined the number of qubits necessary for processing the image, we can start making the quantum circuit for the same.
- Since, our image now basically represents the amplitudes of different quantum states, we can directly use the initialize() method to perform the state preparation. After this, we add a Hadamard gate to the auxiliary qubit, then the amplitude permutation unitary, and then again a Hadamard gate to auxiliary qubit. Also, the whole circuit is repeated once more for the vertical scanning of image.

```
new_image = image
for i in range (0, 128, 32):
    print(i)
    for j in range (0, 128, 32):
        image1 = image[i:i+32, j:j+32]
        image1_norm_h = amplitude_encode(image1)
        image1_norm_v = amplitude_encode(image1.T)
        # Create the circuit for horizontal scan
        qc_h = QuantumCircuit(total_qb)
        qc_h.initialize(image1_norm_h, range(1, total_qb))
        qc_h.h(0)
        qc_h.unitary(D2n_1, range(total_qb))
        qc_h.h(0)
        # display(qc_h.draw('mpl', fold=-1))

        # Create the circuit for vertical scan
        qc_v = QuantumCircuit(total_qb)
        qc_v.initialize(image1_norm_v, range(1, total_qb))
        qc_v.h(0)
        qc_v.unitary(D2n_1, range(total_qb))
        qc_v.h(0)
        # display(qc_v.draw('mpl', fold=-1))

        # Combine both circuits into a single list
        circ_list = [qc_h, qc_v]
```

STEP 7 - SIMULATING THE CIRCUITS

we simulate the circuits using the `statevector_simulator` and obtain the statevector of the system as the output.

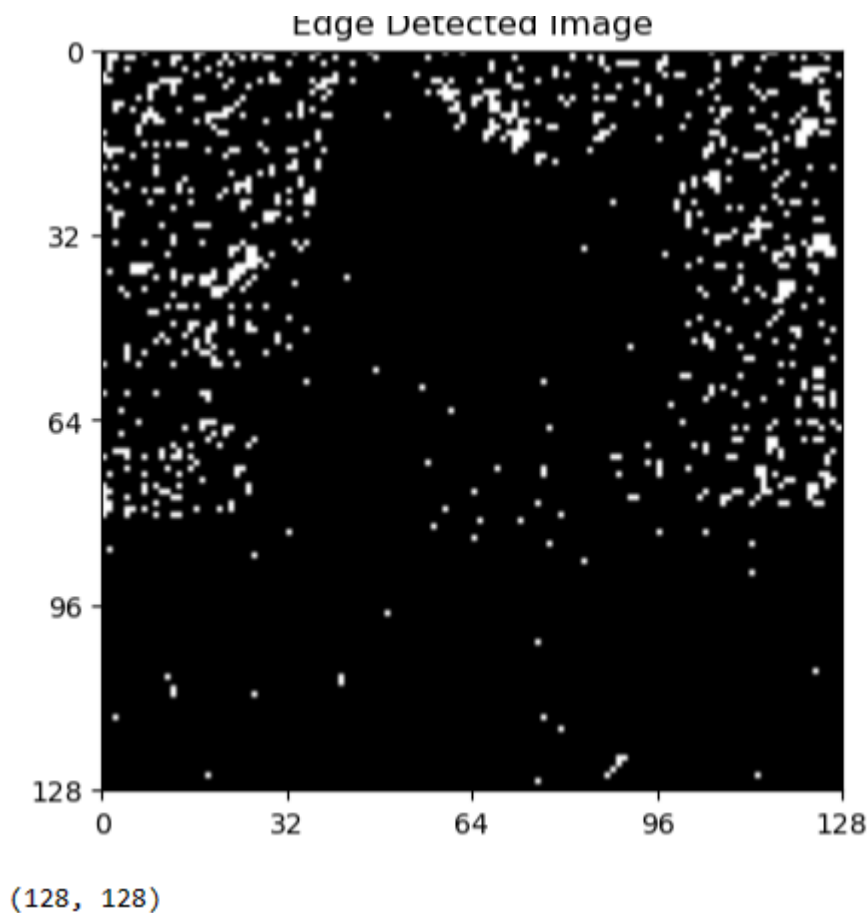
```
# Simulating the circuits
back = Aer.get_backend('statevector_simulator')
results = execute(circ_list, backend=back).result()
sv_h = results.get_statevector(qc_h)
sv_v = results.get_statevector(qc_v)
threshold = lambda amp: (amp > 1e-15 or amp < -1e-15)

# Selecting odd states from the raw statevector and
# reshaping column vector of size 64 to an 8x8 matrix
edge_scan_h = np.abs(np.array([1 if threshold(sv_h[2*i+1].real) else 0 for i in range(2**data_qb)]).reshape(32, 32))
edge_scan_v = np.abs(np.array([1 if threshold(sv_v[2*i+1].real) else 0 for i in range(2**data_qb)]).reshape(32, 32).T)

edge_scan_sim = edge_scan_h | edge_scan_v

for a in range(32):
    for b in range(32):
        new_image[i+a][j+b]=edge_scan_sim[a][b]
```

```
plt.title('Edge Detected Image')
plt.xticks(range(0, new_image.shape[0]+1, 32))
plt.yticks(range(0, new_image.shape[1]+1, 32))
plt.imshow(new_image, extent=[0, new_image.shape[0], new_image.shape[1], 0], cmap='gray_r')
plt.show()
print(new_image.shape)
```



NOTE – THE OUTPUT SHOWN IN THE IMAGE DOESN'T GIVE PROPER EDGES OF THE IMAGE AND MAY TAKE LOT OF MEMORY . TO REDUCE IT , WE CAN USE COLOR QUANTISATION AFTER STEP 5

STEP 1 COLOR QUANTIZATION

Color Quantization is the process of reducing number of colors in an image. One reason to do so is to reduce the memory. Sometimes, some devices may have limitation such that it can produce only limited number of colors. In those cases also, color quantization is performed. Here we use k-means clustering for color quantization.

```

import cv2
import numpy as np

def kmeans_color_quantization(image, clusters=8, rounds=1):
    h, w = image.shape[:2]
    samples = np.zeros([h*w], dtype=np.float32)
    count = 0

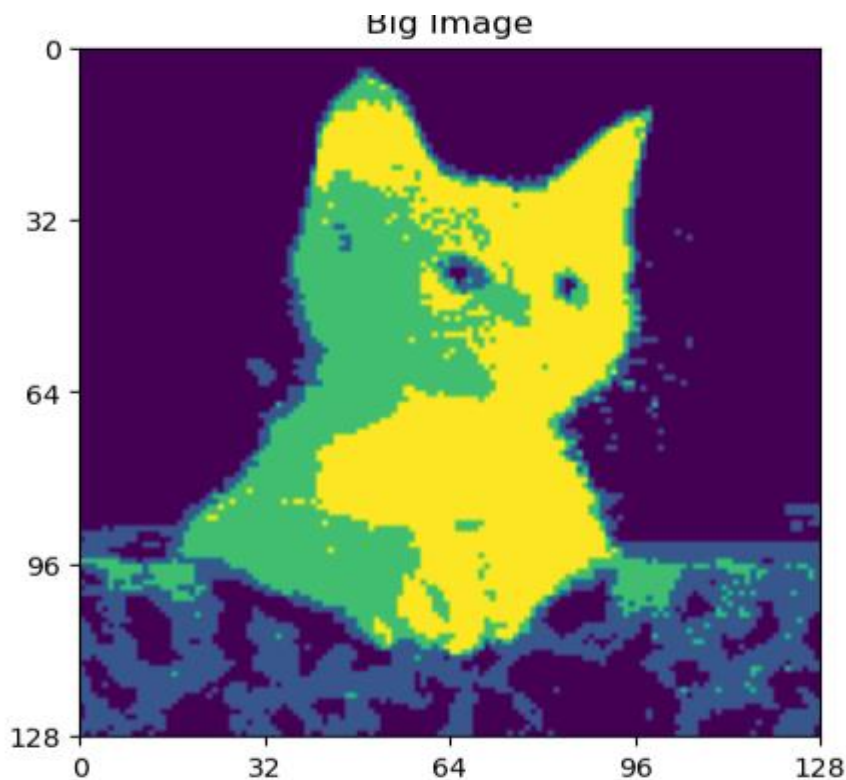
    for x in range(h):
        for y in range(w):
            samples[count] = np.float32(image[x][y])
            count += 1

    compactness, labels, centers = cv2.kmeans(samples,
        clusters,
        None,
        (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10000, 0.0001),
        rounds,
        cv2.KMEANS_RANDOM_CENTERS)

    #centers = np.uint8(centers *)
    #print(List(labels))
    res = centers[labels.flatten()]
    return res.reshape((image.shape)), centers, labels

seg, centers, labels = kmeans_color_quantization(image, clusters=4)
plot_image(seg, 'Big Image')

```



After color quantisation , we initialise the number of qubits and amplitude permutation matrix

STEP 2 - USE OF TQDM: PYTHON PROGRESS BAR LIBRARY

tqdm is a library that is used for creating Python Progress Bars. It gets its name from the Arabic name taqaddum, which means 'progress.'

It can be easily implemented in our loops, functions, or even Pandas. Progress bars are pretty useful in Python because-

- One can see if the Kernel is still working
- Progress Bars are visually appealing to the eyes
- It gives Code Execution Time and Estimated Time for the code to complete, which would help while working on huge datasets

```
import tqdm
CHUNK_POWER = 3
CHUNK_SIZE = 2 ** CHUNK_POWER

#image = result
w, h = image.shape

chunks = [];
back = Aer.get_backend('statevector_simulator')
result = image.copy()

for i in tqdm.tqdm(range(0, w // (CHUNK_SIZE - 1) + 1)):
    for j in range(h // (CHUNK_SIZE - 1) + 1):
        offsetx = i * CHUNK_SIZE - i
        offsety = j * CHUNK_SIZE - j
        if w - offsetx < CHUNK_SIZE:
            offsetx = w - CHUNK_SIZE
        if h - offsety < CHUNK_SIZE:
            offsety = h - CHUNK_SIZE

        chunk = image[offsetx : offsetx + CHUNK_SIZE, offsety : offsety + CHUNK_SIZE]
        chunk_seg = seg[offsetx : offsetx + CHUNK_SIZE, offsety : offsety + CHUNK_SIZE]

        # Get the amplitude encoded pixel values
        # Horizontal: Original image
        chunk_norm_h = amplitude_encode(chunk)
        if chunk_norm_h is None or np.all(chunk_seg == chunk_seg[0][0]):
            edge_scan_h = np.abs(np.array([0.0 for i in range(2**data_qb)]).reshape(CHUNK_SIZE, CHUNK_SIZE))
            edge_scan_v = np.abs(np.array([0.0 for i in range(2**data_qb)]).reshape(CHUNK_SIZE, CHUNK_SIZE).T)
            result[offsetx : offsetx + CHUNK_SIZE, offsety : offsety + CHUNK_SIZE] = np.sqrt(edge_scan_h **2 + edge_scan_v**2)
            continue

        chunk_norm_v = amplitude_encode(chunk.T)
```

THEN , REPEAT THE PROCESS I.E, CREATING CIRCUITS FOR BOTH VERTICAL AND HORIZONTAL SCAN AND COMBINING THEM INTO SINGLE LIST

```
qc_v = QuantumCircuit(total_qb)
qc_v.initialize(chunk_norm_v, range(1, total_qb))
qc_v.h(0)
qc_v.unitary(D2n_1, range(total_qb))
qc_v.h(0)
#display(qc_v.draw('mpl', fold=-1))

# Combine both circuits into a single list
circ_list = [qc_h, qc_v]
# Simulating the circuits
results = execute(circ_list, backend=back).result()
sv_h = results.get_statevector(qc_h)
sv_v = results.get_statevector(qc_v)

# Defining a lambda function for
# thresholding to binary values
threshold = lambda amp: (amp > 1e-15 or amp < -1e-15)

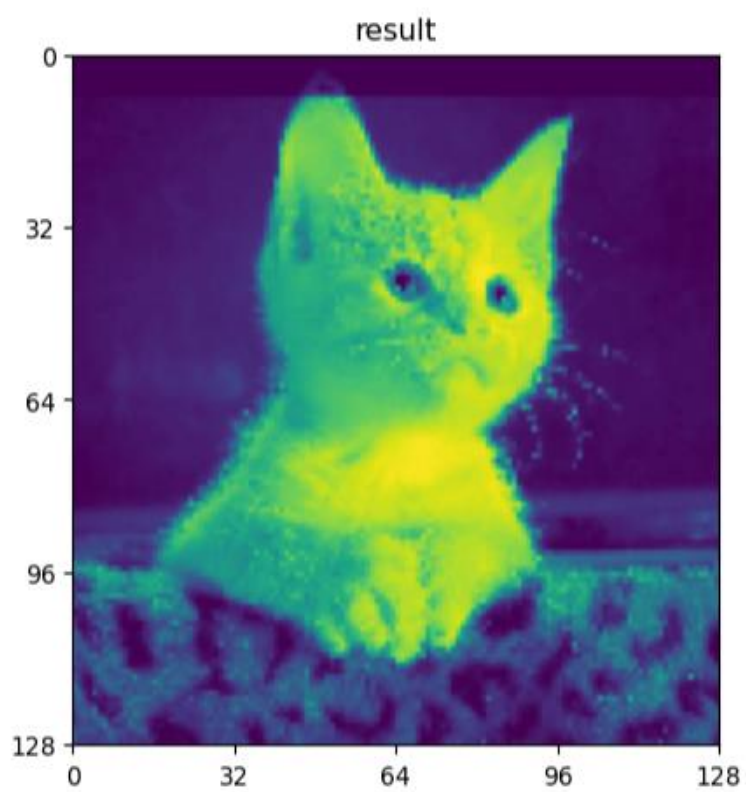
# Selecting odd states from the raw statevector and
# reshaping column vector of size 64 to an 8x8 matrix
# edge_scan_h = np.abs(np.array([1 if threshold(sv_h[2*i+1].real) else 0 for i in range(2**data_qb)])).reshape(16, 16)
# edge_scan_v = np.abs(np.array([1 if threshold(sv_v[2*i+1].real) else 0 for i in range(2**data_qb)])).reshape(16, 16).T

edge_scan_h = np.abs(np.array([sv_h[2*i+1].real for i in range(2**data_qb)])).reshape(CHUNK_SIZE, CHUNK_SIZE)
edge_scan_v = np.abs(np.array([sv_v[2*i+1].real for i in range(2**data_qb)])).reshape(CHUNK_SIZE, CHUNK_SIZE).T

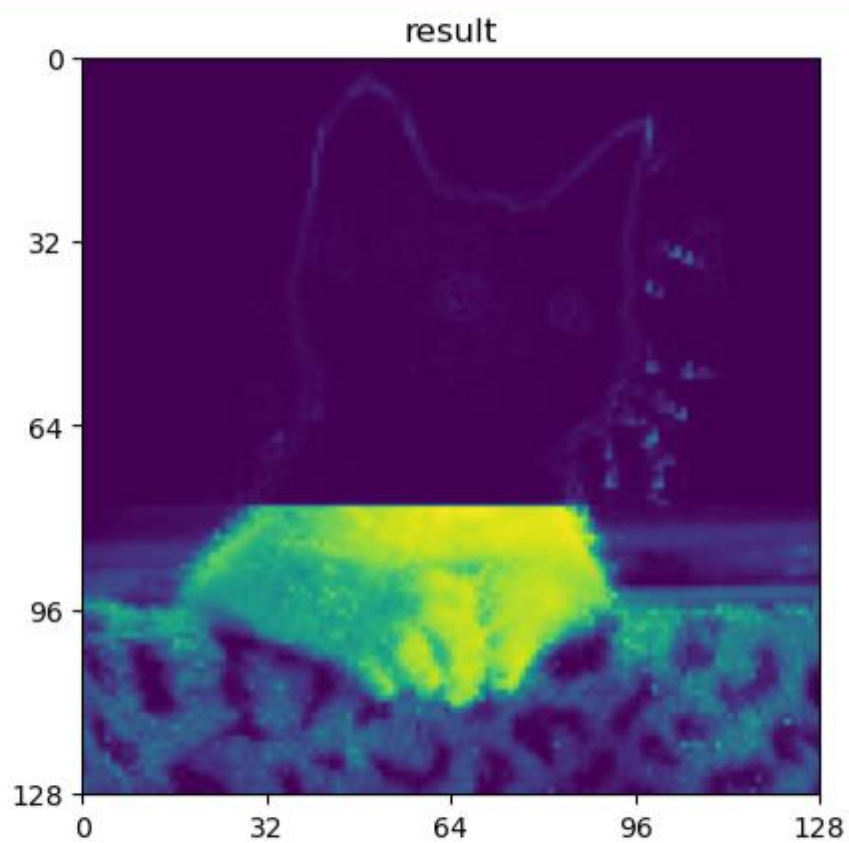
result[offsetx : offsetx + CHUNK_SIZE, offsety : offsety + CHUNK_SIZE] = np.sqrt(edge_scan_h **2 + edge_scan_v**2)
#print(j)
#print(i)
if i % 10 == 0:
    plot_image(result, 'result')

plot_image(result, 'result')
```

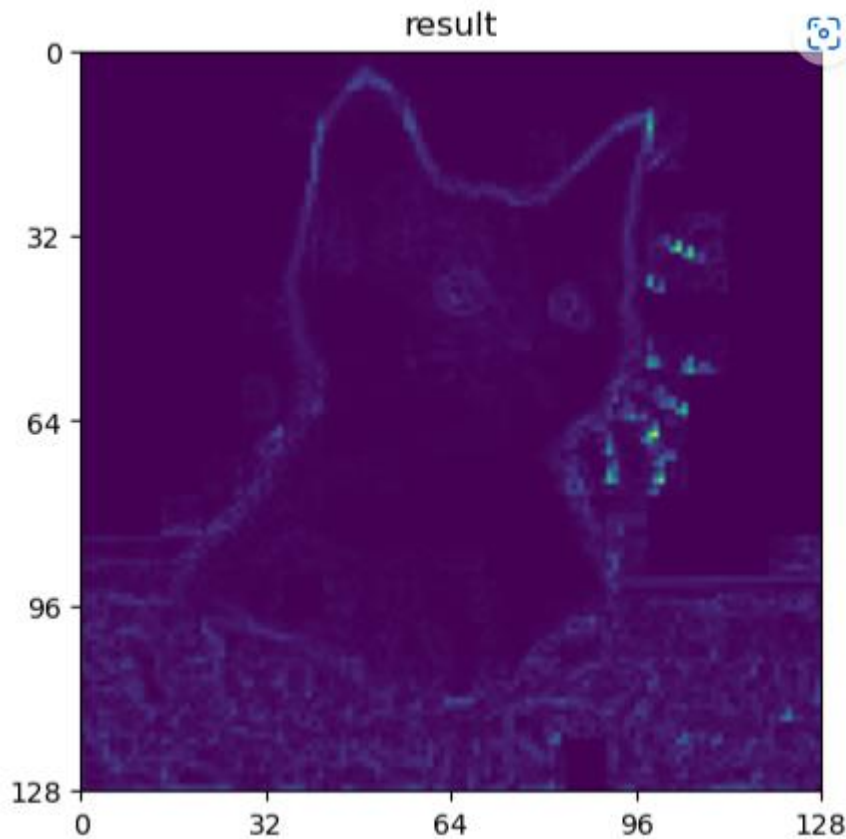
0%| | 0/19 [00:00<?, ?it/s]



53%|██████| 10/19 [00:02<00:02, 4.33it/s]



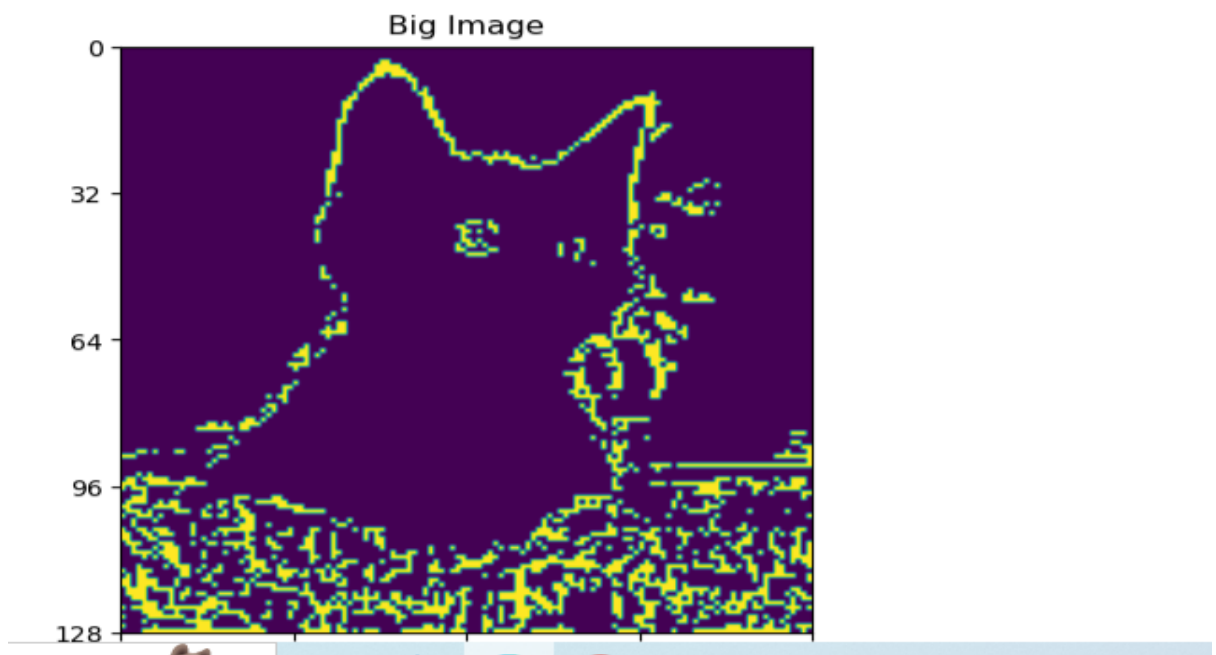
100%|██████████| 19/19 [00:04<00:00, 3.95it/s]



K-Means is an unsupervised algorithm from the machine learning approach. This algorithm tries to make clusters of input data features. The input data objects need to be allocated to separate clusters based on the relationship among them.

- In the beginning step, we shall attempt to determine the number of clusters for our picture.
- After determining the number of colours, it is time to determine the cluster's centroids, which would be the groups' colour representative.
- In the following step, we shall measure the distance of all points of the cluster centroids. And based on the range, we shall appoint every moment to the focus with the least distance.

```
resseg, centers, labels = kmeans_color_quantization(result, clusters=2)
plot_image(resseg, 'Big Image')
```



```
In [28]: edges = cv2.Canny((image * 256).astype(np.uint8),100,200)
plot_image(edges, 'Big Image')
```

