Quantum Hadamard Edge Detection Algorithm (QHED)

Ellen Zhang
Army Educational Outreach Program
High School Apprenticeship Program

Edge Detection

- Extracting the outline of a shape from an image
- Basic algorithm using in computer vision, image processing, and machine vision
- Classical implementation
  - Using filters to find light intensity changes
  - Need to process all the pixels: 2^N number of operations
- Quantum Advantages
  - Exponential speed-up
  - More efficient at tackling large images

Quantum Probability Image Encoding

We first import an image and change it from the classical interpretation to the Quantum Probability Image Encoding (QPIE) state. To do this, we need to normalize all the pixel intensities.

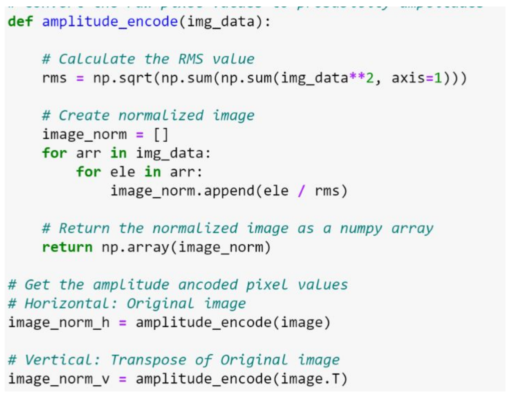
Let the original image with n pixels be represented by vector

$$(a_0 \, a_1 \, a_2 \, \dots \, a_{n-1}).$$

We can define each pixel of the QPIE state of the image to be

$$b_i = \frac{a_i}{\sqrt{\sum_{k=0}^{n-1} a_k^2}}.$$

Using QPIE, we are able to exponentially reduce memory for storing BW or RGB images. A n pixel image would only need $\lceil \log_2 n \rceil$ qubits. QPIE also uses less resources than similar quantum image representation methods like FRQI or NEQR.



```
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)

# Get the amplitude ancoded pixel values
# Horizontal: Original image
image_norm_h = amplitude_encode(image)

# Vertical: Transpose of Original image
image_norm_v = amplitude_encode(image.T)
```

Auxiliary Qubit

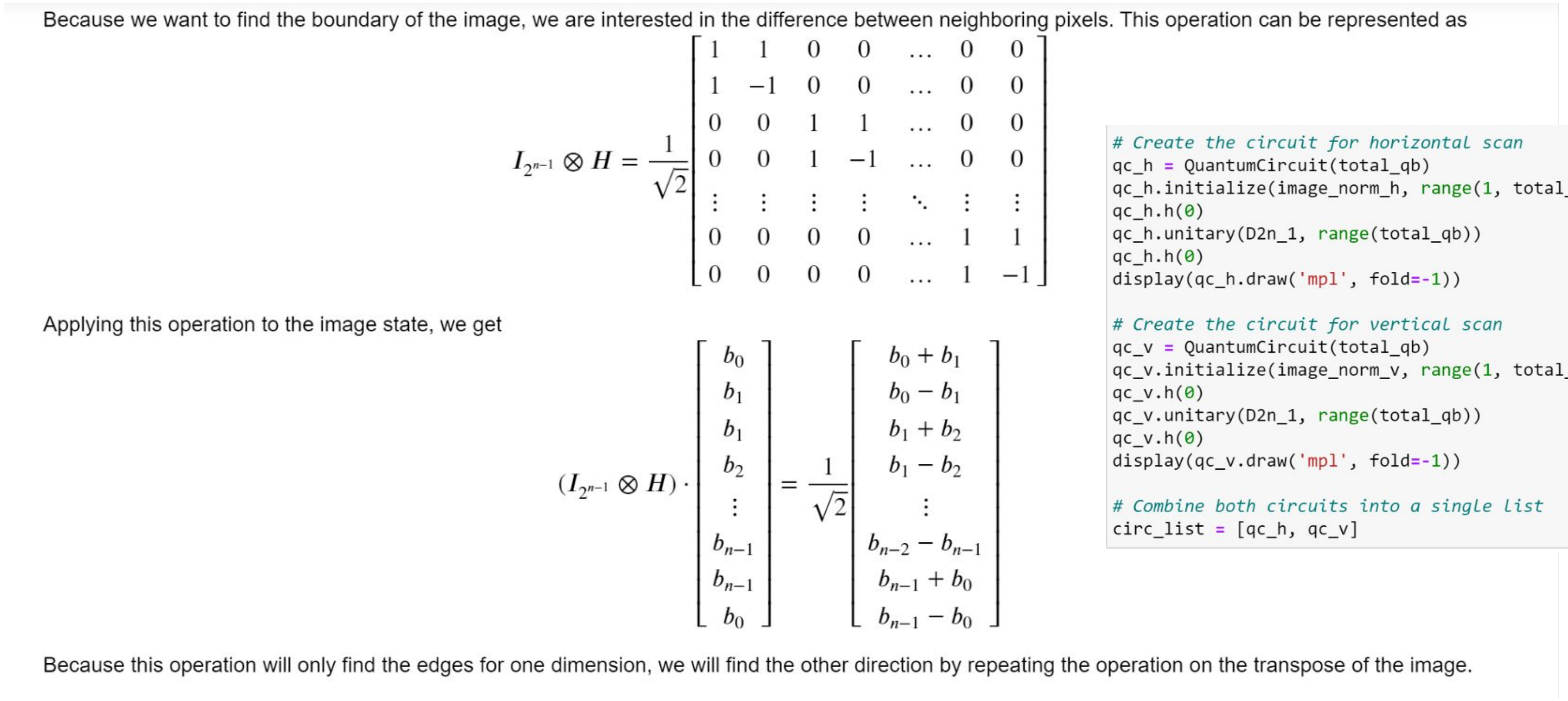
We begin by adding an auxiliary qubit initialized to state $|0\rangle$. The purpose of the auxiliary qubit is to be able to find all the edges in one direction (horizontal or vertical) in one pass. We apply the H-gate to the auxiliary qubit and get the following image state:

$$|\text{image}\rangle \otimes \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{bmatrix} b_0 \\ b_0 \\ b_1 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_{n-1} \end{bmatrix}$$

We then apply a unitary decrement gate to the image state to get $(b_0, b_1, b_1, b_2, b_2, \ldots, b_{n-1}, b_{n-1}, b_0)^T$, shifting the first element to the end.

```
# Initialize some global variable for number of qubits
data_qb = math.ceil(np.log(image.size) / np.log(2))
anc_qb = 1
total_qb = data_qb + anc_qb

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

Hadamard Gates



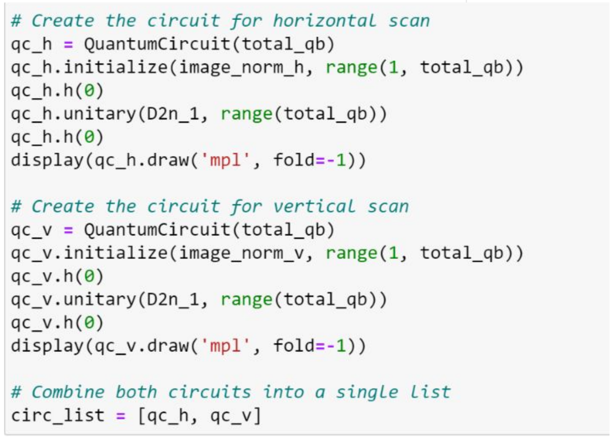
Because we want to find the boundary of the image, we are interested in the difference between neighboring pixels. This operation can be represented as

$$I_{2^{n-1}} \otimes H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 & 0 & 0 & \dots & 0 & 0 \\ 1 & -1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & 1 & \dots & 0 & 0 \\ 0 & 0 & 1 & -1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 & 1 \\ 0 & 0 & 0 & 0 & \dots & 1 & -1 \end{bmatrix}$$

Applying this operation to the image state, we get

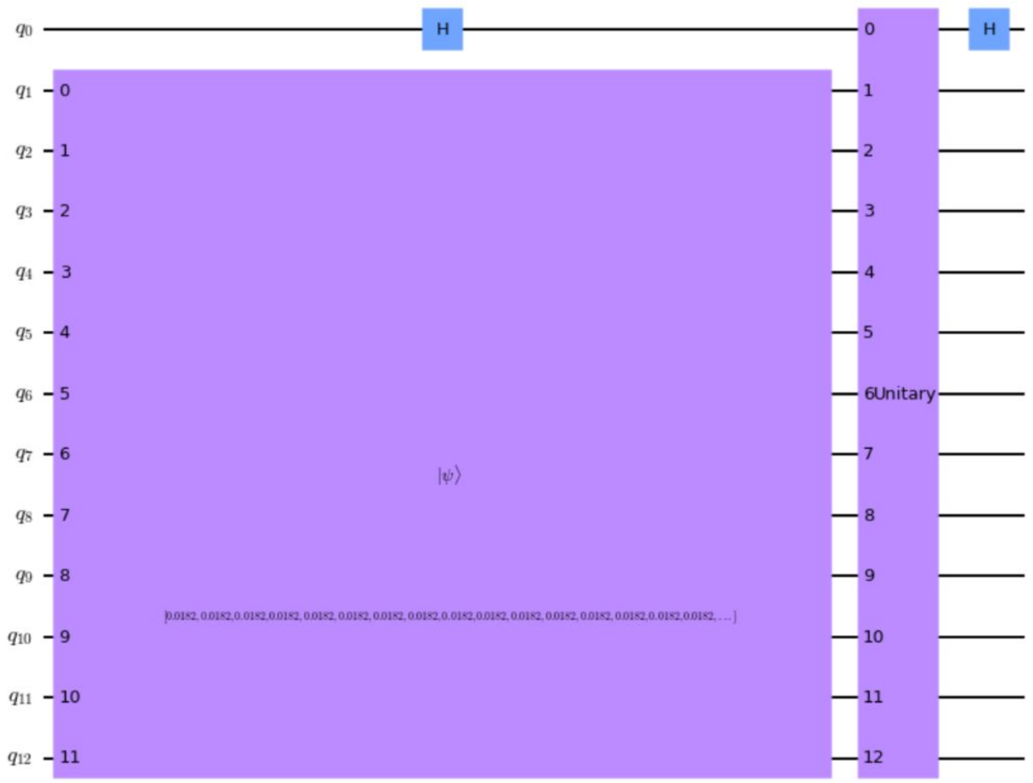
$$(I_{2^{n-1}} \otimes H) \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_{n-1} \\ b_0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} b_0 + b_1 \\ b_0 - b_1 \\ b_1 + b_2 \\ b_1 - b_2 \\ \vdots \\ b_{n-2} - b_{n-1} \\ b_{n-1} + b_0 \\ b_{n-1} - b_0 \end{bmatrix}$$

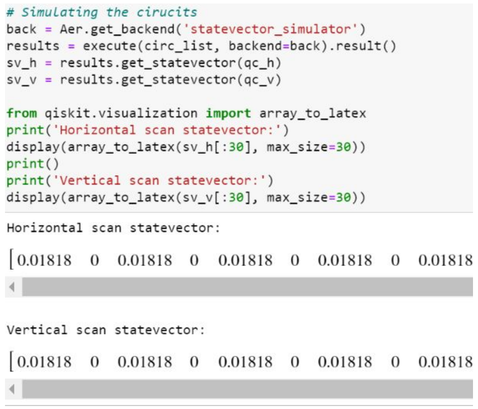
Because this operation will only find the edges for one dimension, we will find the other direction by repeating the operation on the transpose of the image.



```
# Create the circuit for horizontal scan
qc_h = QuantumCircuit(total_qb)
qc_h.initialize(image_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(image_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]
```

Quantum Circuit

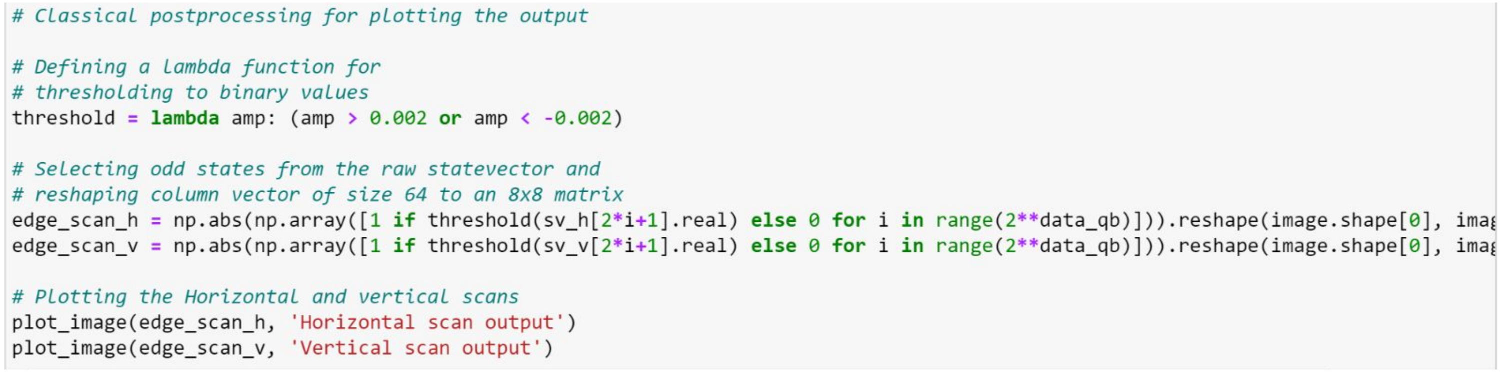



```python
# Simulating the cirucits
back = Aer.get_backend('statevector_simulator')
results = execute(circ_list, backend=back).result()
sv_h = results.get_statevector(qc_h)
sv_v = results.get_statevector(qc_v)

from qiskit.visualization import array_to_latex
print('Horizontal scan statevector:')
display(array_to_latex(sv_h[:30], max_size=30))
print()
print('Vertical scan statevector:')
display(array_to_latex(sv_v[:30], max_size=30))
```

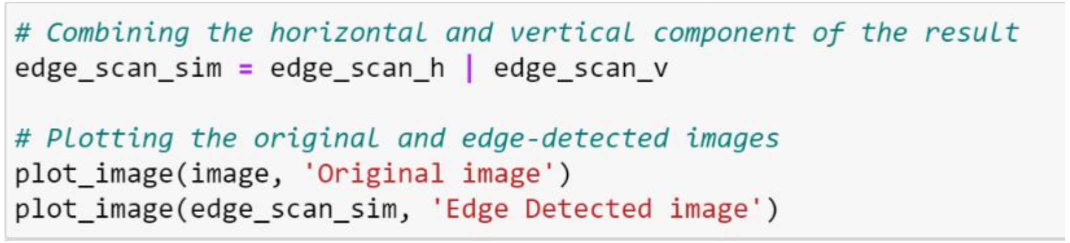
```
Horizontal scan statevector:

[0.01818   0   0.01818   0   0.01818   0   0.01818   0   0.01818

Vertical scan statevector:

[0.01818   0   0.01818   0   0.01818   0   0.01818   0   0.01818
```

Post-Processing

To find the edges of the image, we only need look at the differences between adjacent pixels. To do this, we measure the auxiliary qubit on state $|1\rangle$. The threshold value can be adjusted accordingly. After doing this to both the horizontal and vertical components, we can combine the edges to get the final result.

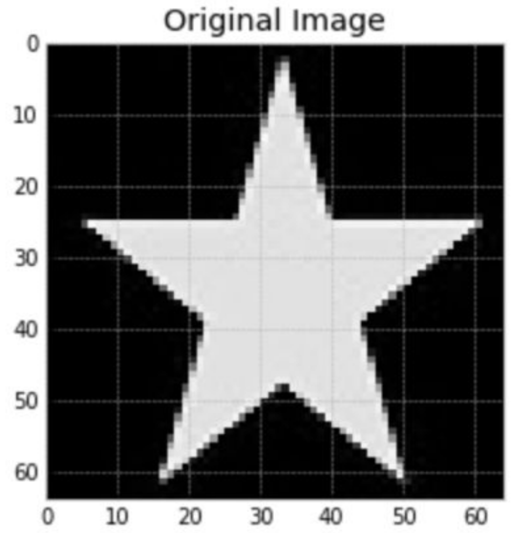


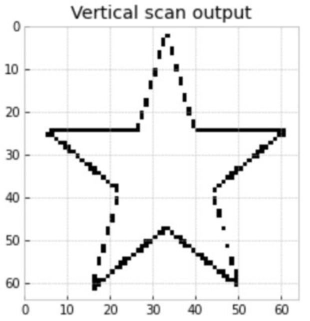
```
# Classical postprocessing for plotting the output

# Defining a lambda function for
# thresholding to binary values
threshold = lambda amp: (amp > 0.002 or amp < -0.002)

# 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(image.shape[0], imag
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(image.shape[0], imag

# Plotting the Horizontal and vertical scans
plot_image(edge_scan_h, 'Horizontal scan output')
plot_image(edge_scan_v, 'Vertical scan output')
```

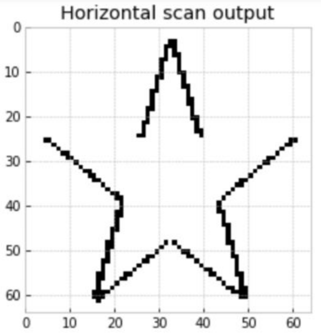


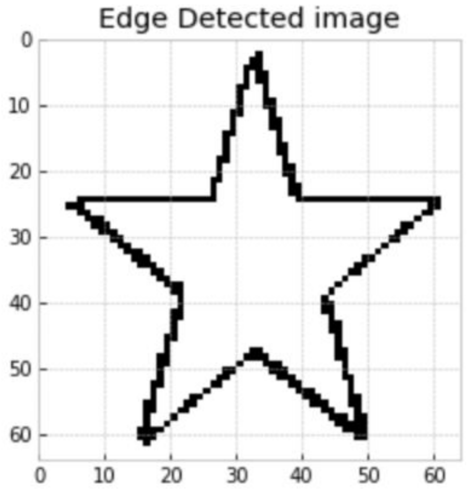
```
# Combining the horizontal and vertical component of the result
edge_scan_sim = edge_scan_h | edge_scan_v

# Plotting the original and edge-detected images
plot_image(image, 'Original image')
plot_image(edge_scan_sim, 'Edge Detected image')
```

Example on a 64x64 image



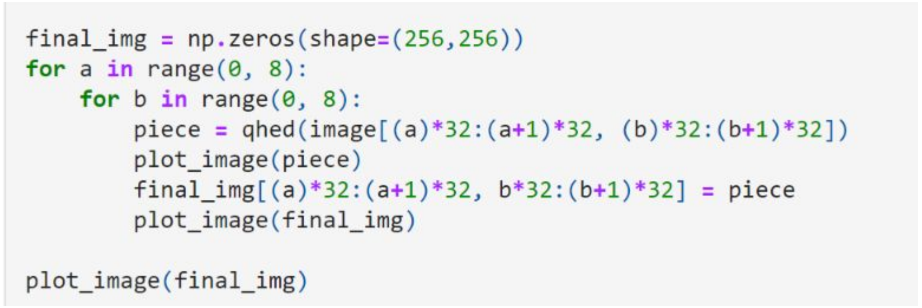






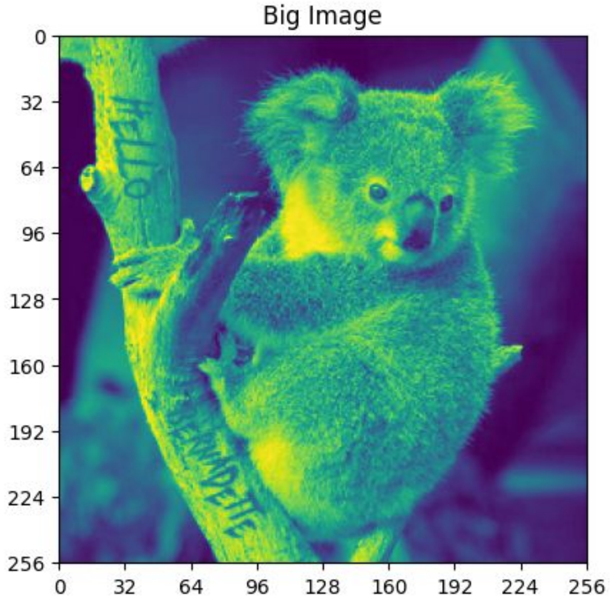
Larger Images

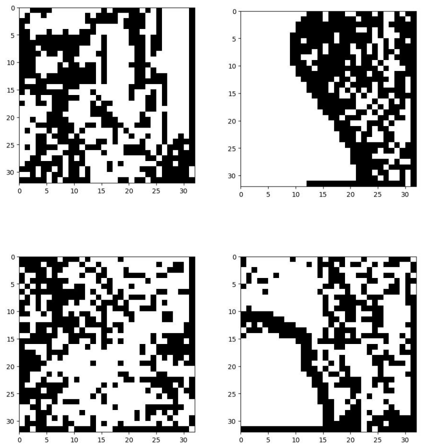
- To directly process large images, fault-tolerant quantum computer needed
- To be able to apply the QHED algorithm efficiently on larger images, one method we can take is to split up large images into smaller ones and recombine at the end.

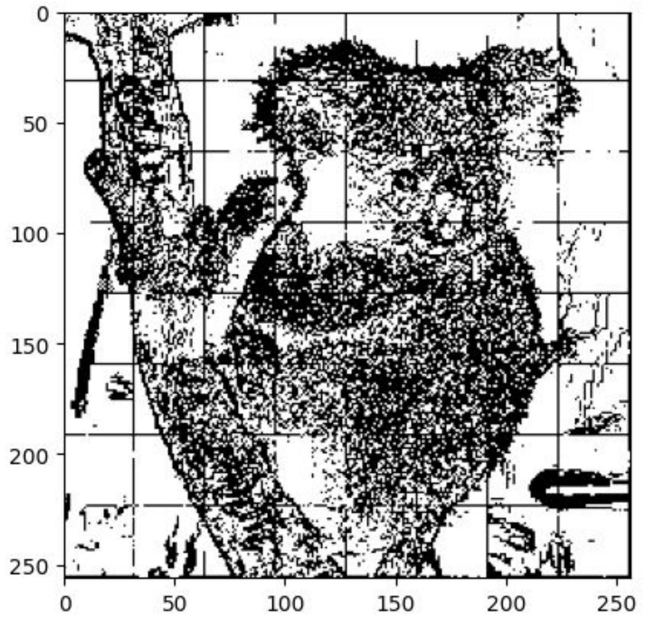


```
final_img = np.zeros(shape=(256,256))
for a in range(0, 8):
    for b in range(0, 8):
        piece = qhed(image[(a)*32:(a+1)*32, (b)*32:(b+1)*32])
        plot_image(piece)
        final_img[(a)*32:(a+1)*32, b*32:(b+1)*32] = piece
        plot_image(final_img)

plot_image(final_img)
```

Example on a 256x256 image







Time Complexity

- Classical algorithms
  - Worst case of $O(2^N)$
  - Best algorithms take $O(mn*\log(mn))$
- QSobel
  - $O(n^2)$
  - FRQI state set-up:$O(n) + O(\log^2 n)$
    - Requires 1+2N qubits
- QHED
  - QPIE state set-up: $O(n^2)$
    - Only requires $\log_2 N$ qubits
  - QHED algorithm: $O(1)$
    - Doesn’t include state-preparation and amplitude permutation
- Quantum algorithms are less precise

Thank you for listening!