Palawan State University
College of Sciences
Computer Studies Department

DIGITAL IMAGE PROCESSING LABORATORY EXERCISE #10

Image Compression by DCT, DPCM, HUFFMAN coding

Image compression by DCT (Discrete Cosine Transform), DPCM (Differential Pulse Code Modulation), and Huffman coding involves reducing the size of an image file while preserving its visual quality to some extent. Here's an overview of each technique:

1. **Discrete Cosine Transform (DCT)**:
    - DCT is a widely used technique for lossy image compression.
    - It transforms spatial information into frequency information, which allows for efficient representation of image data.
    - DCT divides the image into small blocks and applies a mathematical transformation to each block.
    - The transformed coefficients are then quantized to reduce the number of bits required to represent them.
    - DCT-based compression is used in popular image formats like JPEG.

2. **Differential Pulse Code Modulation (DPCM)**:
    - DPCM is a lossless compression technique that exploits spatial redundancy in images.
    - It predicts the value of each pixel based on the values of neighboring pixels.
    - The prediction error (the difference between the predicted and actual pixel values) is encoded and transmitted.
    - DPCM is particularly effective for images with smooth regions or slowly varying intensities.

3. **Huffman Coding**:
    - Huffman coding is a lossless data compression algorithm that assigns variable-length codes to different symbols in the input data.
    - The length of each code depends on the frequency of occurrence of the corresponding symbol.
    - Huffman coding achieves compression by replacing frequently occurring symbols with shorter codes and less frequent symbols with longer codes.
    - It is commonly used to encode the quantized DCT coefficients or the prediction error in DPCM.
    -

The combination of these techniques allows for efficient compression of image data while minimizing perceptual loss. DCT handles spatial redundancy, DPCM addresses temporal redundancy, and Huffman coding reduces the entropy of the encoded data, resulting in significant compression ratios with acceptable image quality degradation.

1. Discrete Cosine Transform (DCT):
    - load an image using skimage.io.imread() and convert it to grayscale.
    - perform a 2D DCT using scipy.fftpack.dct().
    - specify a compression ratio, which determines how much of the DCT coefficients we will keep.
    - calculate a threshold based on the compression ratio.
    - create a mask for the DCT coefficients, keeping only those above the threshold.
    - reconstruct the image using the inverse DCT (scipy.fftpack.idct()).
    - display the original and reconstructed images using matplotlib.pyplot.imshow().

Make sure to replace 'input_image.jpg' with the path to the image you want to compress. You may need to install the necessary libraries (scikit-image, scipy, matplotlib) if you haven't already.

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import io
from scipy.fftpack import dct, idct
# Load the image
image = io.imread('input_image.jpg', as_gray=True)
# Perform 2D DCT
dct_image = dct(dct(image, axis=0, norm='ortho'), axis=1, norm='ortho')

# Compression ratio (keep only top-left corner coefficients)
compression_ratio = 0.1
# Determine the threshold for keeping coefficients
threshold = np.percentile(np.abs(dct_image), 100 - compression_ratio * 100)
# Mask the coefficients below the threshold
masked_dct_image = dct_image * (np.abs(dct_image) > threshold)
# Reconstruct the image using inverse DCT
reconstructed_image = idct(idct(masked_dct_image, axis=0, norm='ortho'), axis=1, norm='ortho')
# Display the original and reconstructed images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(reconstructed_image, cmap='gray')
plt.title('Reconstructed Image (Compressed)')
plt.axis('off')
plt.show()
```

2. Differential Pulse Code Modulation (DPCM):
   - load an image using skimage.io.imread() and convert it to grayscale.
   - define a function dpcm_encode() to perform DPCM encoding. This function encodes each pixel value as the difference between the current pixel and the previous pixel in the same row.
   - apply the dpcm_encode() function to the input image to obtain the encoded image.
   - display the original and encoded images using matplotlib.pyplot.imshow().

   Make sure to replace 'input_image.jpg' with the path to the image you want to encode. You may need to install the necessary libraries (scikit-image, matplotlib) if you haven't already.

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import io
# Load the image
image = io.imread('input_image.jpg', as_gray=True)
# Define DPCM function
def dpcm_encode(image):
    # Initialize output array with zeros
    encoded_image = np.zeros_like(image)
        # Loop through each pixel row by row
    for i in range(image.shape[0]):
        # For the first pixel in each row, encode as is
        encoded_image[i, 0] = image[i, 0]
            # For subsequent pixels, encode the difference from the previous pixel
```

```
        for j in range(1, image.shape[1]):
            encoded_image[i, j] = image[i, j] - image[i, j - 1]
        return encoded_image
# Perform DPCM encoding
encoded_image = dpcm_encode(image)
# Display the original and encoded images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(encoded_image, cmap='gray')
plt.title('Encoded Image (DPCM)')
plt.axis('off')
plt.show()
```

3. Huffman Coding:
   - load an image using plt.imread() and convert it to grayscale.
   - flatten the grayscale image to create a 1D array of pixel values.
   - define functions huffman_encoding() and huffman_decoding() to perform Huffman encoding and decoding, respectively.
   - use the huffman_encoding() function to encode the flattened image data.
   - Optionally, use the huffman_decoding() function to decode the encoded data (this step is included for demonstration purposes).
   - reshape the decoded data back to the original image shape and display the original and decoded images using matplotlib.pyplot.imshow().

   Make sure to replace 'input_image.jpg' with the path to the image you want to encode. You may need to install the necessary libraries (matplotlib) if you haven't already.

```
import heapq
import numpy as np
import matplotlib.pyplot as plt
from collections import defaultdict
# Load the image
image = plt.imread('input_image.jpg')
# Convert the image to grayscale
gray_image = np.dot(image[..., :3], [0.2989, 0.5870, 0.1140])
# Flatten the image
flattened_image = gray_image.flatten()
# Function to generate Huffman encoding
def huffman_encoding(data):
    freq_dict = defaultdict(int)
    for value in data:
        freq_dict[value] += 1
    heap = [[weight, [symbol, ""]] for symbol, weight in freq_dict.items()]
    heapq.heapify(heap)
```

```
    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
     huffman_dict = dict(sorted(heapq.heappop(heap)[1:], key=lambda p: (len(p[-1]), p)))
    encoded_data = ''.join(huffman_dict[value] for value in data)
     return encoded_data, huffman_dict
# Function to decode Huffman encoded data
def huffman_decoding(encoded_data, huffman_dict):
    reverse_dict = {v: k for k, v in huffman_dict.items()}
    current_code = ""
    decoded_data = ""
    for bit in encoded_data:
        current_code += bit
        if current_code in reverse_dict:
            decoded_data += reverse_dict[current_code]
            current_code = ""
    return decoded_data
# Perform Huffman encoding
encoded_data, huffman_dict = huffman_encoding(flattened_image)
# Perform Huffman decoding (optional)
decoded_data = huffman_decoding(encoded_data, huffman_dict)
# Reshape the decoded data back to the original image shape
decoded_image = np.reshape(np.array(list(decoded_data), dtype=np.uint8), gray_image.shape)
# Display the original and decoded images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(gray_image, cmap='gray')
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(decoded_image, cmap='gray')
plt.title('Decoded Image (Huffman)')
plt.axis('off')
plt.show()
```

Exercise #10
Image Compression by DCT, DPCM, HUFFMAN coding

Name:
Year/Block:
Application/Software:

1. Codes
2. Output
3. Answer the following questions:

   A. How do various parameters, including the size of the DCT coefficient matrix, the quantization step size, and the choice of frequency domain techniques like DFT or DWT, influence the quality, compression ratio, and computational complexity of image compression?

   B. How does the choice of predictive coding technique, including DPCM, impact image compression, considering factors such as prediction method (spatial vs. temporal), suitability for different image types, and the effectiveness of adaptive schemes in optimizing compression efficiency based on image characteristics?

   C. How does entropy influence the efficiency of Huffman coding, considering its role in encoding symbol probabilities? Furthermore, how does the construction of Huffman trees affect compression ratios and decoding complexity, and what are the limitations of Huffman coding, especially concerning images with non-uniform pixel value distributions?