final_report_12232509

May 31, 2023

1 Image and Video Compression and Network Communication

12232509 FAN Qingyuan

1.1 Introduction

Image compression is an indispensable tool in modern society. It involves the process of reducing the size of an image file without degrading the quality of the image below an acceptable level. This reduction in file size allows more images to be stored within a certain amount of disk or memory space. It also reduces the time required for images to be sent over the internet or downloaded from web pages.

In this report, we delve into the concept of image compression by implementing the basic modules and algorithms, including Discrete Wavelet Transform (DWT), fix step size quantization, Embedded Zero-tree Waveform (EZW) encoding, and (size, amplitude) representation in Python. After the compression, we simulated the transmission of the compressed image over the internet using a file server and use the client function to download and reconstruct the image.

1.2 Import the necessary packages

Before proceeding, We need import the necessary packages first. The functions in the `lib` could be found from `lib` folder.

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import json
from lib import dwt, huffman, ezw, size_amp, enc_dec, entropy
```

1.3 Encode

1.3.1 Read the image

The provided image are 512x512 grayscale image. We first read the image and display it.

[2]:
```
# open /home/fanqy/research-temp/eee5347/final-project/image1.512
with open('/home/fanqy/research-temp/eee5347/final-project/img/image1.512',␣
↪'rb') as f:
    data = f.read()
    data = list(data)
    # reshape data to numpy 512x512
    data = np.array(data)
    img = data.reshape((512, 512))
    plt.imshow(img, cmap='gray')
    plt.colorbar()
```

1.4 Discrete Wavelet Transform

In this section, we implement the DWT algorithm. Specifically, we will apply the Cohen-Daubechies-Feauveau (CDF) 5/3 wavelet, a widely used wavelet in image compression, to our images for four continuous iterations. The objective of this process is to obtain an array of coefficients where the low-frequency components are located in the top-left corner, while the high-frequency components are situated in the bottom-right.

To compute the DWT, we first need to define the low-pass and high-pass filters as follows:

```
# Filter coefficients
lp_analysis = np.array([-1, 2, 6, 2, -1]) / 8
hp_analysis = np.array([-1, 2, -1]) / 2
lp_synthesis = np.array([1, 2, 1]) / 2
hp_synthesis = np.array([-1, -2, 6, -2, -1]) / 8
```

Then, we could implement the DWT by convolution and the downsampling operation. The idea behind Discrete Wavelet Transform involves these two fundamental steps, typically known as filtering and downsampling.

In the `decomposition` function, we first apply convolution to the image with a high-pass filter `hp_analysis` and a low-pass filter `lp_analysis`. These filters are essential as they help separate the image details and approximations, respectively. This process involves multiplying each pixel of the image with the filter coefficients to calculate the high-frequency and low-frequency components of the image.

Following convolution, we proceed with the downsampling operation, which effectively reduces the size of the image by half. We keep every other pixel from the convolution output, both for the low-frequency part (lf) and the high-frequency part (hf). The low-frequency component, `lf`, which carries the approximate details of the image, is sampled at even indices, while the high-frequency component, `hf`, containing the detailed parts of the image, is sampled at odd indices. The resultant `input_decomposition_img` is an image with half the size, with the left side containing the low-frequency part and the right side containing the high-frequency part.

The `decomposition_to_4` function repeats the `decomposition` operation but on the transposed image, essentially performing a 2D DWT.

The `decomposition_to_specify_level` function takes in the image and the desired level of DWT decomposition. It repetitively applies the `decomposition_to_4` function for a specified number of levels, thereby achieving multi-level DWT. Each level of DWT provides a further level of details and approximations, enhancing the compression potential.

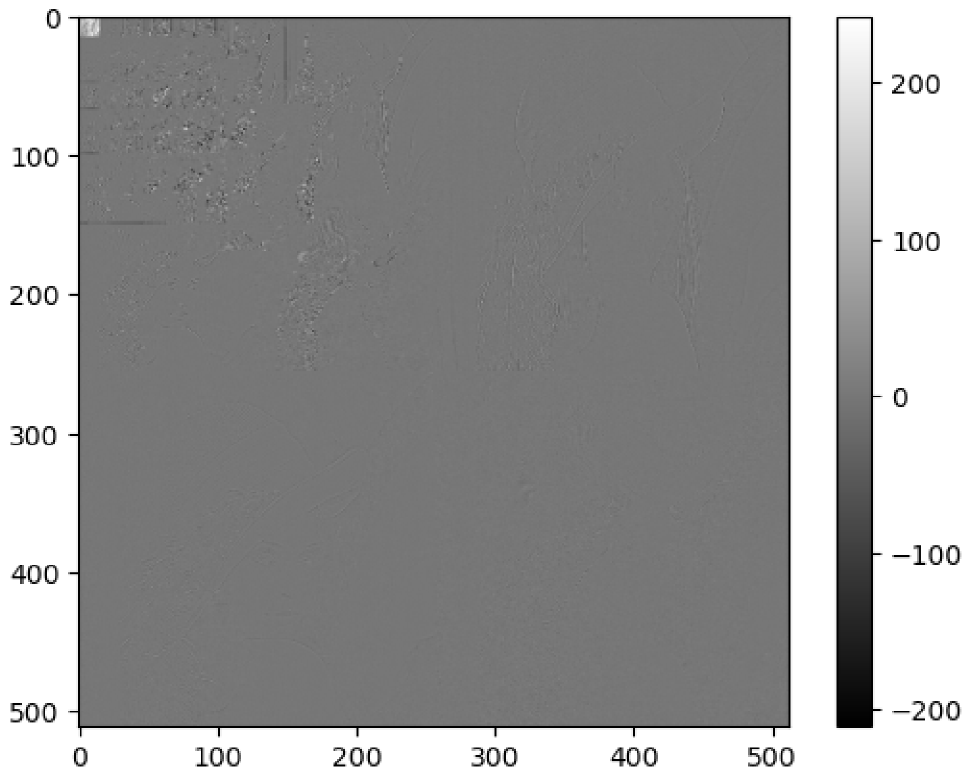
```python
def decomposition(input_img):
    input_decomposition_img = np.zeros(input_img.shape)
    # decomposition
    hf = pad_and_convolve_img(input_img, hp_analysis)
    lf = pad_and_convolve_img(input_img, lp_analysis)
    input_decomposition_img[:,0:input_img.shape[1]//2] = lf[:,1::2]
    input_decomposition_img[:,input_img.shape[1]//2:input_img.shape[1]] = hf[:,::2]
    return input_decomposition_img

def decomposition_to_4(input_img):
    # decomposition
    L1 = decomposition(input_img)
    L2 = decomposition(L1.T).T
    return L2

def decomposition_to_specify_level(input_img, level):
    input_img = input_img.astype(np.float64)
    L = input_img.astype(np.float64)
    height = input_img.shape[0]
    width = input_img.shape[1]
    for i in range(1,level+1):
        # print(width//i, height//i)
        L[0:height//i,0:width//i] = decomposition_to_4(L[0:height//i,0:width//i])
    return L
```

[3]:
```python
# decomp
l5_decomp = dwt.decomposition_to_specify_level(img, 5)
plt.imshow(l5_decomp, cmap='gray')
plt.colorbar()
```

[3]:
```
<matplotlib.colorbar.Colorbar at 0x7f61be70b3a0>
```

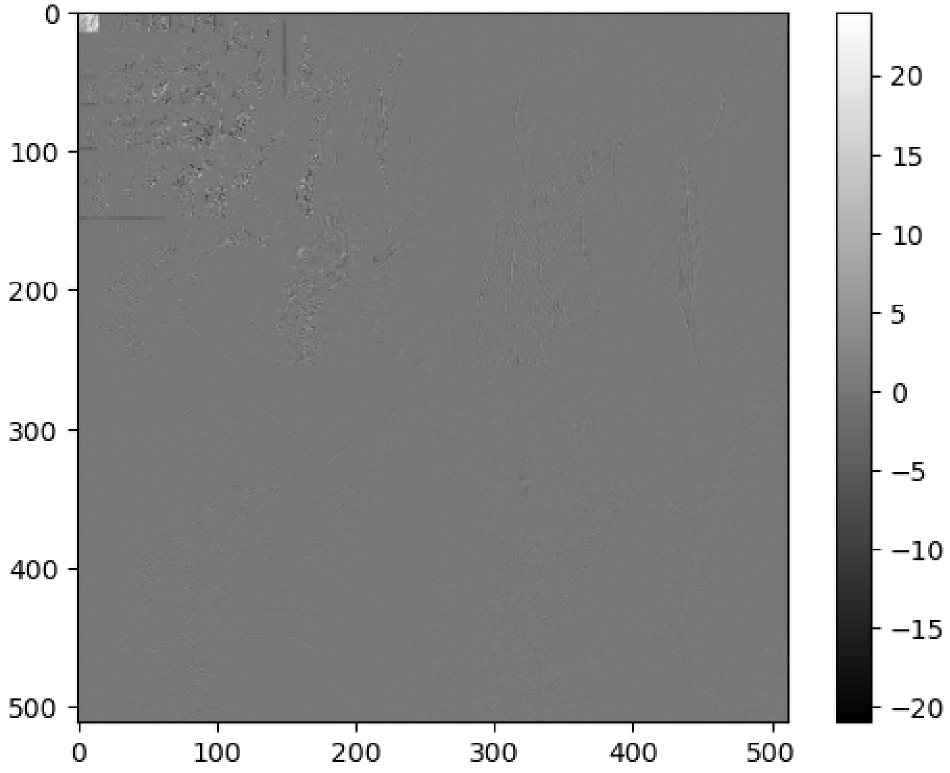


1.4.1 Fix step size quantization

In this section, we implement the fix step size quantization algorithm. The objective of this process is to reduce the number of bits required to represent the image coefficients. This process involves dividing the coefficients by a constant value, known as the step size, and rounding the result to the nearest integer. The step size is a positive integer that determines the number of bits required to represent the coefficients. The smaller the step size, the more bits are required to represent the coefficients, and the reconstructed image will be closer to the original image. However, the compression ratio will be lower. On the other hand, a larger step size will result in a higher compression ratio but a lower quality reconstructed image.

[4]:
```python
# quant
quant_step = 10
l5_decomp_quant = np.round(l5_decomp / quant_step)
plt.imshow(l5_decomp_quant, cmap='gray')
plt.colorbar()
```

```
[4]: <matplotlib.colorbar.Colorbar at 0x7f61bca2c910>
```



1.4.2 Prediction to the lowest frequency subband

In this section, we implement the prediction to the lowest frequency subband algorithm. As part of this algorithm, we focus on the lowest frequency components of the image, which are located in the top-left 16*16 corner of the image after multi-level DWT. This subset of the image represents the most significant components of the image, and their accurate prediction is crucial for effective compression.

We start by extracting this 16*16 array from our decomposed image using `l5_decomp_quant[0:16, 0:16].flatten().astype(int)`, then flatten it into a one-dimensional array for processing convenience.

Following this, we compute the difference between every two consecutive elements in this one-dimensional array using the `np.diff(top_16_16)` function. This process creates a new array that represents the difference between each pixel value and the next, effectively transforming the original pixel values into a series of differences or deltas. These differences often have smaller absolute values, thereby occupying less space, and hence aiding in image compression.

Finally, the difference array is combined with the first element of the original array, using the `np.insert(top_16_16_diff, 0, top_16_16[0])` function. We add the first element back because it's necessary for reconstructing the original data during the decompression process. The result is stored in the `top_16_16_diff` array, ready for the next step of the compression process.

```
[5]: # encode the top 16*16
top_16_16 = l5_decomp_quant[0:16, 0:16].flatten().astype(int)
top_16_16_diff = np.diff(top_16_16)
top_16_16_diff = np.insert(top_16_16_diff, 0, top_16_16[0]).tolist()
```

1.4.3 Scan the high-frequency subbands using zero-tree scan and EZT symbols

We also need to use the zero-tree scan to encode the high-frequency part, which could significantly improve the efficiency of image compression. For this purpose, we utilize the Embedded Zero-tree Wavelet (EZW) algorithm, which exploits the statistical dependence among wavelet coefficients of an image.

In our EZW implementation, we first construct a hierarchical data structure `class EZWTree`, similar to a tree. Each node in the tree represents a pixel of the high-frequency part of the image, with its value, level, quadrant, and other information. The children of each node correspond to the pixels in the next level of the DWT, in the same spatial area as the parent pixel. This spatial correspondence is key to the efficiency of the EZW encoding.

The `build_tree` function builds this tree from the image. Starting from a given pixel, it recursively creates child nodes until it has processed all pixels within the image boundary. After the tree is built, the `dfs_set_value` function traverses the tree in depth-first order, setting the `ezwcode` for each node. The `ezwcode` could be 'P' for positive or negative non-zero, 'Z' for isolated zero, and 'T' for zero-tree root.

Then, we use the `bfs_encode_list` function to conduct a breadth-first traversal of the tree, encoding the significant pixels first. This breadth-first traversal ensures that we will stop encoding its children when we meet the first zero-tree root. The `bfs_encode_list` function returns a list of encoded pixels, which is then stored in the `ezwcode` attribute of each node.

Finally, the `enc_dp_sp` function applies the encoding to each pixel in the image, transforming the image into an encoded list that takes up less space and can be transmitted or stored more efficiently.

```
# create a EZW tree class
class EZWTree:
    def __init__(self, value, level, quadrant, coordinates, children, parent):
        self.value = value
        self.level = level
        self.children = children if children is not None else []
        self.quadrant = quadrant
        self.coordinates = coordinates
        self.ezwcode = None
        self.parent = parent if parent is not None else None

def build_tree(image, level=0, coordinates=(1,0), quadrant=None, parent=None):
    i, j = coordinates
    # Base case: if the coordinates are out of the image boundary, return None
    if i >= image.shape[0] or j >= image.shape[1]:
        return None

    # create the list of children coordinates / quadrant 2,1,3,4
    child_coordinates = [(2*i, 2*j), (2*i, 2*j+1), (2*i + 1, 2*j), (2*i + 1, 2*j + 1)]

    node = EZWTree(image[i, j], level, quadrant, coordinates, children=[], parent=parent)

    # recursively create the children
    for coord in child_coordinates:
        ci, cj = coord
        # Check if the child coordinates are within the image boundary
        if ci < image.shape[0] and cj < image.shape[1]:
            child = build_tree(image, level+1, coord, quadrant, parent=node)
            node.children.append(child)

    return node

def dfs_set_value(node):
    # base case: if node is None, return False
    if node is None:
        return False

    # first process the children
    children_significant = [dfs_set_value(child) for child in node.children]

    # process the current node
    if np.abs(node.value) != 0:
        node.ezwcode = 'P' #Positive or negative non-zero
        return True
    else:
        # does code have significant descendant?
        if any(children_significant):
            node.ezwcode = 'Z' #Isolated zero
            return True
        else:
            node.ezwcode = 'T' #ZTR
            return False

def bfs_encode_list(root):
    # Use deque for faster pop from start of queue
    queue = deque(root.children)
    dominant_pass_result = []

    while queue:
        node = queue.popleft()  # O(1) operation
        if node.ezwcode in {'P', 'Z'}:
            dominant_pass_result.append(int(node.value))
        elif node.ezwcode == 'T':
            dominant_pass_result.append(node.ezwcode)
            continue
        if node.children:
            queue.extend(node.children)
    return dominant_pass_result


def transverse_encode(root):
    # use DFS to traverse the tree and set the ezwcode
    dfs_set_value(root)
    # use BFS to export the dominant_pass_result
    dominant_pass_result = bfs_encode_list(root)
    return dominant_pass_result

def enc_dp_sp(root_nodes_list):
    dpr_list = [[None for _ in range(16)] for _ in range(16)]
    for i in range(0, 16):
        for j in range(0, 16):
            dpr_list[i][j] = transverse_encode(root_nodes_list[i][j])
            # print('coor', root_nodes_list[i][j].coordinates, 'done')
    return dpr_list
```

[6]:
```
# ezw_encode
# enumerate the img[0,16:0,16]
root_nodes = [[None for _ in range(16)] for _ in range(16)]
for i in range(0, 16):
    for j in range(0, 16):
        # in order of quadrant 1,3,4
        child_1 = ezw.build_tree(l5_decomp_quant, 1, (i,j+16), 1, None)
        child_3 = ezw.build_tree(l5_decomp_quant, 1, (i+16,j), 3, None)
        child_4 = ezw.build_tree(l5_decomp_quant, 1, (i+16,j+16), 4, None)
        root_nodes[i][j] = ezw.EZWTree(l5_decomp_quant[i, j], 0, 2, (i,j),␣
 ↪[child_1, child_3, child_4], None) # type: ignore
        child_1.parent = root_nodes[i][j] # type: ignore
        child_3.parent = root_nodes[i][j] # type: ignore
        child_4.parent = root_nodes[i][j] # type: ignore

dpr_list= ezw.enc_dp_sp(root_nodes)
```

1.4.4 (size, amplitude) Representation

Completing the zero-tree scan and the prediction of the low-frequency band, we need to use the (size, amplitude) Representation to do further compression on the image data. This method distinguishes itself by using two separate arrays for size and amplitude, which reduces the need for a fixed length encoding and decreases the overall size of the encoded data.

The `size_amplitude_single_list` function carries out the conversion from the original image data to the (size, amplitude) representation. This function goes through every pixel in the EZW tree of input image and for each non-zero pixel, calculates its binary length (`bin_len`) and converts it to a binary string. These binary lengths are then stored in the `result_size` array, while the binary strings representing pixel values are stored in the `result_amplitude` array. For zero and 'T' (embedded zeroTrees root) pixels, corresponding identifiers 'Z' and 'T' are added to the `result_size` array. At the end of each list, we append 'E' to indicates the end of each incides.

```
def convert_to_bin(num, bin_len):
    if num >= 0:
        return bin(num)[2:].zfill(bin_len)
    else:
        return "0" + bin(abs(num))[3:].zfill(bin_len-1)

def convert_from_bin(num):
    if num[0] == "1": # positive
        return int(num, 2)
    else:
        return -int("1" + num[1:], 2)


def size_amplitude_single_list(input_list):
    input_list = input_list.copy()
    result_size = [] #len of amp, 0, T
    result_amplitude = []
    for num_to_enc in input_list: # avoid pop
        if num_to_enc == "T":
            result_size.append("T")
        elif num_to_enc == 0:
            result_size.append("Z")
        else:
            bin_len = num_to_enc.bit_length()
            result_size.append(bin_len)
            result_amplitude.append(convert_to_bin(num_to_enc, bin_len))
    result_size.append("E") #EOB
    return result_size, result_amplitude
```

[7]:
```
# encode the top 16*16
rs_16_16, ra_16_16 = size_amp.size_amplitude_single_list(top_16_16_diff)
ra_16_16_str = ''.join(str(i) for i in ra_16_16)

# encode the ezw tree
result_size = []
result_amplitude = []
for i in range(16):
    for j in range(16):
        rs, ra = size_amp.size_amplitude_single_list(dpr_list[i][j])
        result_size = result_size + rs
        result_amplitude = result_amplitude + ra
```

```
result_amplitude_str_lf = ''.join(str(i) for i in result_amplitude)
```

1.4.5 Huffman coding and binary encoding

The last step of the encoding module is the Huffman coding and binary encoding. This process primarily involves the 'size' array generated from the (size, amplitude) representation, as the 'amplitude' array is already composed of binary values and thus does not require further encoding.

Firstly, the `convert_code_table_to_binary` function takes the Huffman dictionary (`code_table`) as input and converts it into a binary format. It does this by transforming the Huffman dictionary into a JSON string and then converting this string into binary.

Next, the `pad_encoded_data` function adds necessary padding to the binary data to ensure it's a multiple of 8, allowing it to be stored as bytes. It adds zeros at the end of the binary data and also includes an 8-bit representation of the padding size at the start of the data string.

Finally, the `write_to_binary_file` function writes the combined binary data, including the Huffman code table, quantization step size, encoded 'size' array, and the 'amplitude' array, to a binary file. The data is chunked into 8 bits at a time, with each chunk converted into a byte and written into the file. This sequence of operations effectively compresses the image data while maintaining its essential characteristics, making it ready for efficient transmission or storage.

To distinguish the different parts of the binary data, we also define a seperator `separator` to seperate the different parts of the binary data. The `separator` is a 32-bit binary number `11110000101001011111000010100101` in our implementation.

```
import json
import struct
import requests
# encode

def pad_encoded_data(data):
    # Calculate the number of bits we need to pad
    pad_size = 8 - (len(data) % 8)

    # Pad with zeros
    data += '0' * pad_size

    # Also add the size of the padding to the start of the string
    pad_info = "{0:08b}".format(pad_size)
    data = pad_info + data

    return data

def convert_code_table_to_binary(code_table):
    # Convert the code table to a JSON string
    code_table_str = json.dumps(code_table)

    # Convert the JSON string to binary
    code_table_binary = ''.join(format(ord(i), '08b') for i in code_table_str)

    return code_table_binary

def write_to_binary_file(data, filename):
    with open(filename, 'wb') as file:
        for i in range(0, len(data), 8):
            chunk = data[i:i+8]
            byte = int(chunk, 2).to_bytes(1, 'big')
            file.write(byte)
```

[8]:
```
# huffman encode
huffman_dict = huffman.huffman_encode(result_size + rs_16_16)
huffman_encoded_result_size = huffman.encode_data(result_size, huffman_dict)
huffman_encoded_rs_16_16 = huffman.encode_data(rs_16_16, huffman_dict)

code_table = huffman_dict

# Convert the code table to binary
code_table_binary = enc_dec.convert_code_table_to_binary(code_table)

quant_step_binary = enc_dec.float_to_binary_str(float(quant_step))

# Combine the code table and the encoded data
# Define a separator
separator = '1111000010100101111100001010010l'

# Combine table, quant step, encoded data
combined_data =  code_table_binary + separator + quant_step_binary + separator␣
 ↪+ enc_dec.pad_encoded_data(huffman_encoded_result_size) + separator +␣
 ↪enc_dec.pad_encoded_data(result_amplitude_str_lf) + separator + enc_dec.
 ↪pad_encoded_data(huffman_encoded_rs_16_16) + separator + enc_dec.
 ↪pad_encoded_data(ra_16_16_str)

# Write the combined data to file
enc_dec.write_to_binary_file(combined_data, 'encoded_data.bin')
```

1.5 Network Transmission

In our implementation, we host a file server at the server side and use the `requests` library to fetch the binary file from the server at the client side. The `requests` library is a Python HTTP library that allows us to send HTTP requests in Python. After the fetch, we directly put the received binary data into the decoder.

1.6 Decode

1.6.1 Read the data from binary file

The first step of the decoding module is to read the data from the binary file. This is done using the `read_from_binary_file` function which reads byte data from the file and converts it into a binary string. An alternative `read_from_url` function is provided for cases where the encoded data is available online.

After the data is read, it is split into various components using the `split_data` function with a predefined separator. These components include the binary Huffman code table, the quantization step, the Huffman-encoded 'size' array, the plain 'amplitude' array, and the encoded raster scan arrays.

The binary Huffman code table is converted back into a dictionary using the `convert_binary_to_code_table` function, which converts binary strings back into characters and parses them as JSON.

The `decode_data_with_huffman` function is used to decode the Huffman-encoded 'size' array and the raster scan arrays, and the `decode_data_plain` function is used for the 'amplitude' array. The padding size is first extracted from the data to correctly remove any added padding. Then, the data is traversed bit by bit, building up a buffer until a matching Huffman code is found, which is then replaced by the corresponding value from the reversed Huffman code table.

Finally, the quantization step is converted from binary back to a floating point value using the `binary_str_to_float` function, and the arrays of decoded values are returned for further processing.

```python
# decode
def read_from_binary_file(filename):
    with open(filename, 'rb') as file:
        data_bytes = file.read()

    return ''.join(format(byte, '08b') for byte in data_bytes)

def read_from_url(url):
    response = requests.get(url)
    data_bytes = response.content

    return ''.join(format(byte, '08b') for byte in data_bytes)

def split_data(data, separator):
    return data.split(separator)

def convert_binary_to_code_table(binary_str):
    json_str = ''.join(chr(int(binary_str[i:i+8], 2)) for i in range(0, len(binary_str), 8))
    return json.loads(json_str)

def float_to_binary_str(f):
    return ''.join(bin(c).replace('0b', '').rjust(8, '0') for c in struct.pack('!f', f))
```

```
def binary_str_to_float(b):
    return struct.unpack('!f', bytearray(int(b[i : i + 8], 2) for i in range(0, len(b), 8)))[0

def decode_data_with_huffman(data, code_table):
    reversed_code_table = {v: k for k, v in code_table.items()}

    # Extract the padding size from the start of the data
    pad_size = int(data[:8], 2)
    data = data[8:]

    # Remove the padding from the end of the decoded data
    data = data[:-pad_size] if pad_size != 0 else data

    # Decode the data
    decoded_data = []
    buffer = ''
    for bit in data:
        buffer += bit
        if buffer in reversed_code_table:
            decoded_data.append(reversed_code_table[buffer])
            buffer = ''

    return decoded_data

def decode_data_plain(data):

    # Extract the padding size from the start of the data
    pad_size = int(data[:8], 2)
    data = data[8:]

    # Remove the padding from the end of the decoded data
    decoded_data = data[:-pad_size] if pad_size != 0 else data

    return decoded_data
```

[9]:
```
# Read the binary data from the file
combined_data = enc_dec.read_from_binary_file('encoded_data.bin')

# Split the combined data into the code table and the encoded data

code_table_binary, quant_step_binary, recon_huffman_encoded_result_size,␣
 ↪recon_encoded_data_lf, recon_huffman_encoded_rs_16_16, recon_ra_16_16_str =␣
 ↪enc_dec.split_data(combined_data, separator)

# Convert the binary code table back to a dictionary
code_table = enc_dec.convert_binary_to_code_table(code_table_binary)
```

```
# Convert the quantization step from binary to decimal
quant_step = enc_dec.binary_str_to_float(quant_step_binary)

# Decode the encoded data
recon_result_size = enc_dec.
 ↪decode_data_with_huffman(recon_huffman_encoded_result_size, code_table)
recon_result_amplitude_str_lf = enc_dec.decode_data_plain(recon_encoded_data_lf)
recon_rs_16_16 = enc_dec.
 ↪decode_data_with_huffman(recon_huffman_encoded_rs_16_16, code_table)
recon_ra_16_16_str = enc_dec.decode_data_plain(recon_ra_16_16_str)
```

1.6.2 Reconstruct the ezw tree

We call the `size_amplitude_to_2D_list_optimized` to reconstruct the ezw tree and convert to the 2D list to store the 256 (16*16) ezw root of the image.

```
def size_amplitude_to_2D_list_optimized(result_size, result_amplitude):
    # Avoids the need for pop(0), which is an O(n) operation.
    result_size = [(int(x), 'N') if x.isdigit() else (x, 'C') for x in result_size]

    result_amplitude_index = 0
    recon_dpr_list = [[[] for _ in range(16)] for _ in range(16)]
    dpr_list_index = 0

    # Use enumerate for faster access
    for size, type_ in result_size:
        dpr_list_index_div, dpr_list_index_mod = divmod(dpr_list_index, 16)
        if type_ == 'C':
            if size == "T":
                recon_dpr_list[dpr_list_index_div][dpr_list_index_mod].append("T")
            elif size == "Z":
                recon_dpr_list[dpr_list_index_div][dpr_list_index_mod].append(0)
            elif size == "E":
                dpr_list_index += 1
        else:
            amplitude = result_amplitude[result_amplitude_index:result_amplitude_index+size]
            recon_dpr_list[dpr_list_index_div][dpr_list_index_mod].append(convert_from_bin(''.
            result_amplitude_index += size

    return recon_dpr_list
```

[10]:
```
recon_dpr_list = size_amp.
 ↪size_amplitude_to_2D_list_optimized(recon_result_size,␣
 ↪recon_result_amplitude_str_lf)
```

We also build a tree which is identical to the ezw tree and a blank image, call `dec_dp_sp` function and use BFS to traverse the tree and put the high frequency value back to the image.

```
def decode_tree(root, dominant_pass, recon_img):
    # Use deque for faster pop from start of queue
    queue = deque(root.children)
    idx = 0

    while queue:
        node = queue.popleft()  # O(1) operation
        ezwcode = dominant_pass[idx]
        i, j = node.coordinates
        if ezwcode != "T":
            recon_img[i, j] = ezwcode
            idx += 1
        elif ezwcode == 'T': # ZTR
            idx += 1
            # print('T')
            continue
        if node.children:
            queue.extend(node.children)
    return


def dec_dp_sp(dpr_list, recon_root_nodes, recon_img):
    for i in range(0, 16):
        for j in range(0, 16):
            decode_tree(recon_root_nodes[i][j], dpr_list[i][j], recon_img)
    return recon_img
```

[11]:
```
# enumerate the img[0,16:0,16]
recon_img = np.zeros((512,512)).astype('float64')
recon_root_nodes = [[None for _ in range(16)] for _ in range(16)]
for i in range(0, 16):
    for j in range(0, 16):
        # in order of quadrant 1,3,4
        child_1 = ezw.build_tree(recon_img, 1, (i,j+16), 1, None)
        child_3 = ezw.build_tree(recon_img, 1, (i+16,j), 3, None)
        child_4 = ezw.build_tree(recon_img, 1, (i+16,j+16), 4, None)
        recon_root_nodes[i][j] = ezw.EZWTree(recon_img[i, j], 0, 2, (i,j),␣
  ↪[child_1, child_3, child_4], None) # type: ignore
        child_1.parent = recon_root_nodes[i][j] # type: ignore
        child_3.parent = recon_root_nodes[i][j] # type: ignore
        child_4.parent = recon_root_nodes[i][j] # type: ignore

recon_img = ezw.dec_dp_sp(recon_dpr_list, recon_root_nodes, recon_img)
```

1.6.3 Reconstruct the low frequency band

After reconstructing the difference of the low frequency band `recon_top_16_16_diff` from the (size, amplitude) Representation, we add back the difference bewtween each index and put the reconstructed low frequency band back to the image.

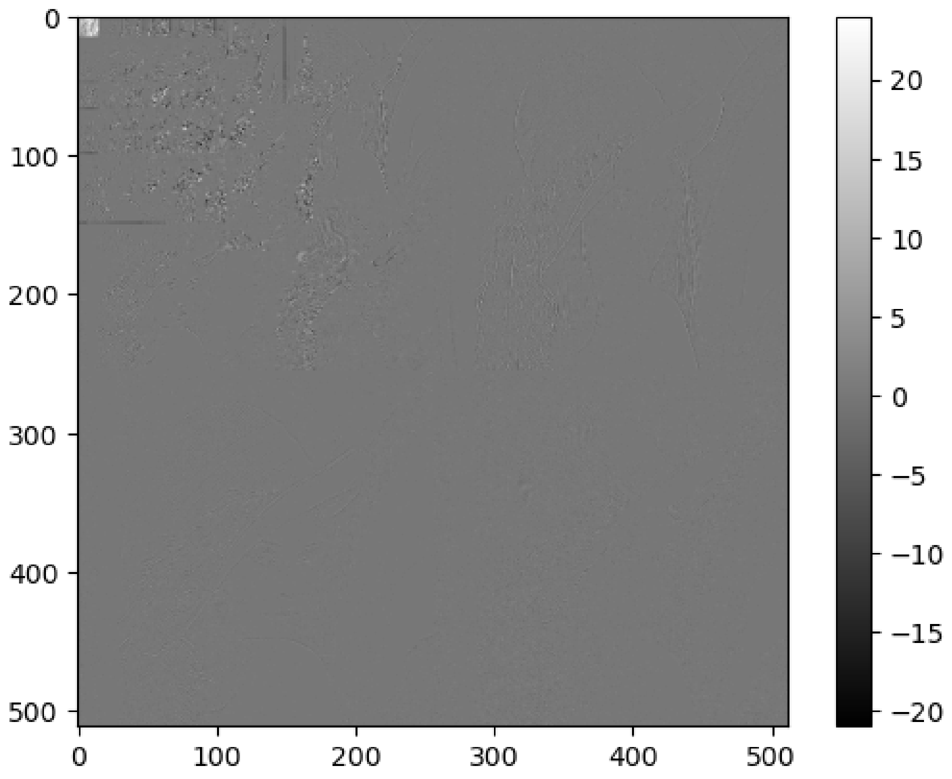
[12]:
```
# decode top 16*16
recon_top_16_16_diff = size_amp.size_amplitude_to_2D_list(recon_rs_16_16,␣
    ↪recon_ra_16_16_str)[0][0]

recon_top_16_16 = np.zeros(256).astype('int')
recon_top_16_16[0] = recon_top_16_16_diff[0]
for i in range(1, 256):
    recon_top_16_16[i] = recon_top_16_16[i-1] + recon_top_16_16_diff[i]
# reshape to 16*16
recon_top_16_16 = recon_top_16_16.reshape((16,16))

recon_img[0:16, 0:16] = recon_top_16_16
```

[13]:
```
plt.imshow(recon_img, cmap='gray')
plt.colorbar()
```

[13]:
```
<matplotlib.colorbar.Colorbar at 0x7f61ac0c1bb0>
```

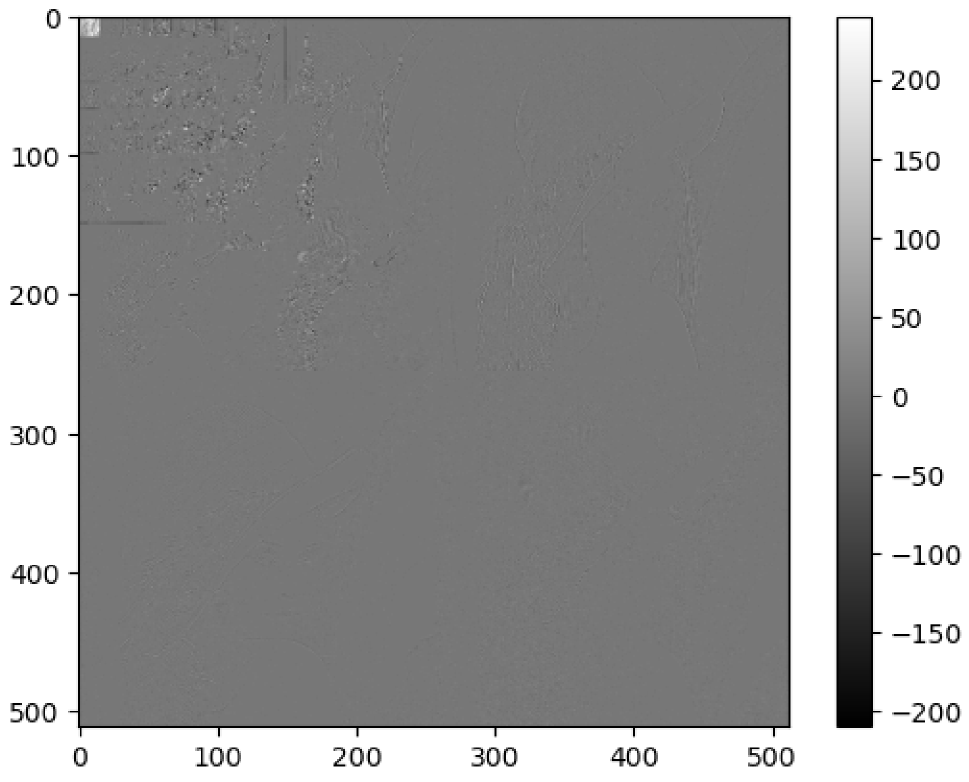


1.6.4 Reverse quantization

In this section, we multiply the quantization step back to the image and convert the image back.

```
[14]: # reverse quantization
      recon_img_after_quant = recon_img * quant_step
      plt.imshow(recon_img_after_quant, cmap='gray')
      plt.colorbar()
```

```
[14]: <matplotlib.colorbar.Colorbar at 0x7f61ac0225b0>
```

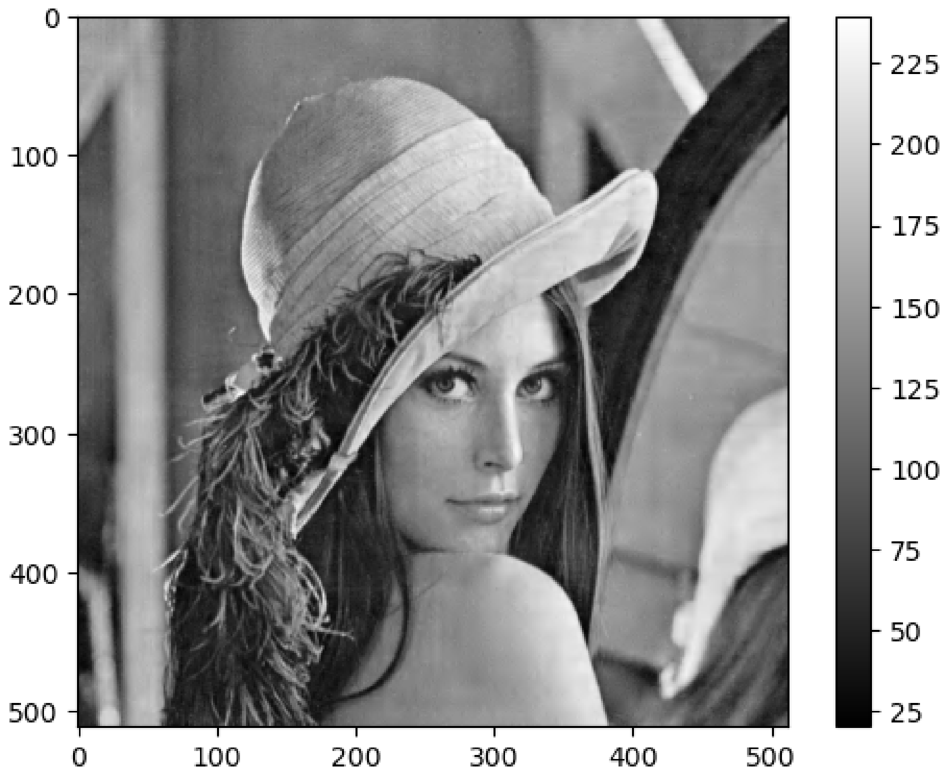


1.6.5 Inverse DWT

Lastly, we call `reconstruction_from_specify_level` to reconstruct the image from the 2D list. the procedure of the function is basically the reverse of the `decomposition_to_specify_level`.

```
[15]: # reconstruct the image
      l5_recon_img_after_quant = dwt.
       ↪reconstruction_from_specify_level(recon_img_after_quant, 5)
      plt.imshow(l5_recon_img_after_quant, cmap='gray')
      plt.colorbar()
```

```
[15]: <matplotlib.colorbar.Colorbar at 0x7f61abf0d700>
```

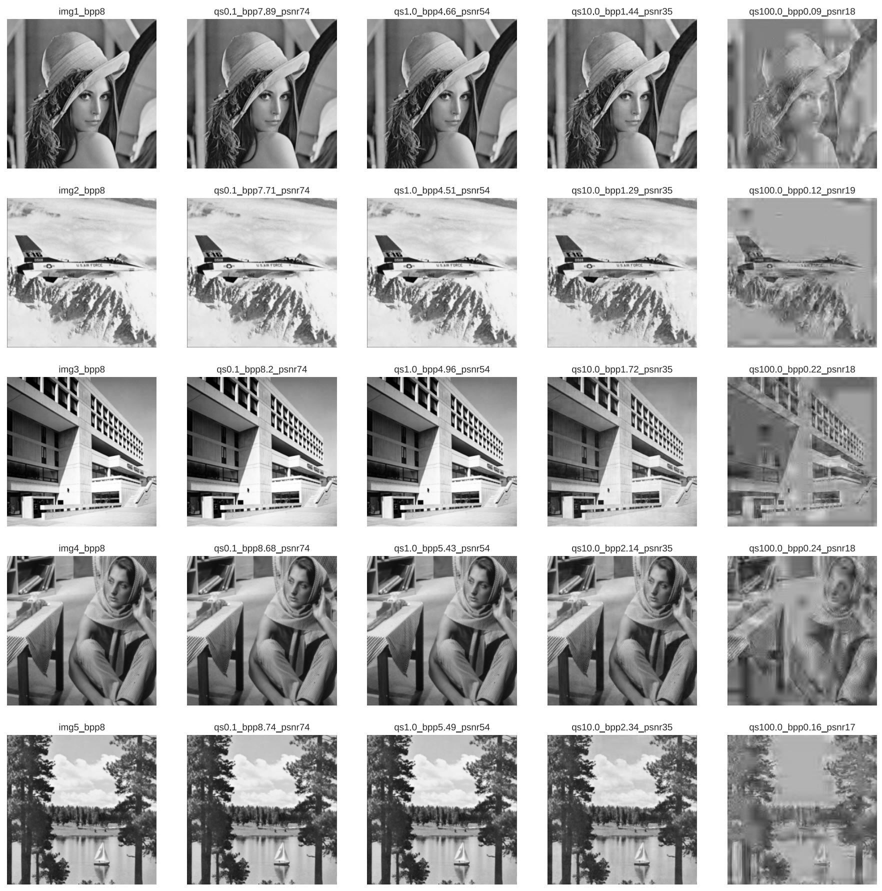


1.7 Results

[16]:
```python
from IPython.display import SVG, Image, display
def show_svg(fn):
    display(SVG(filename=fn))
def show_image(fn):
    display(Image(filename=fn))
```

In the following figure, we plot the result of the original image and the reconstructed image with the quantization step from 0.1 to 100. We can see that at quantization step below than 10, the reconstructed image is almost identical to the original image. However, as the quantization step increases, the reconstructed image becomes more and more blurry.

[17]:
```python
show_svg('/home/fanqy/research-temp/eee5347/final-project/report-img/5x5.svg')
```



1.8 Metrics

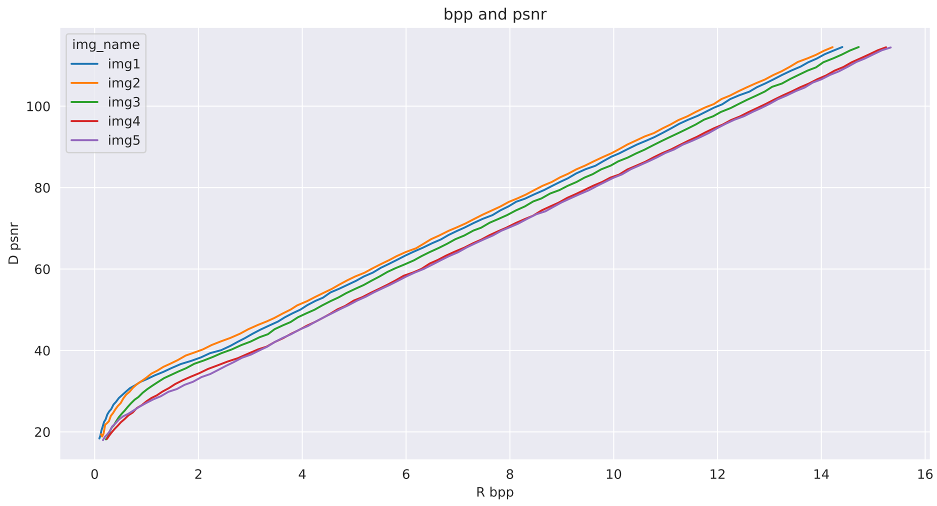
1.8.1 R-D curve

We use the bpp (bit per pixel) to represent the Rate, and PSNR (Peak Signal-to-Noise Ratio) to represent the distortion of the reconstructed image. According to the figure, we experimented with quantization steps ranging from 0.01 to 100. The results show a clear pattern: as the quantization step increases, both the bpp and the PSNR show an exponential increase.

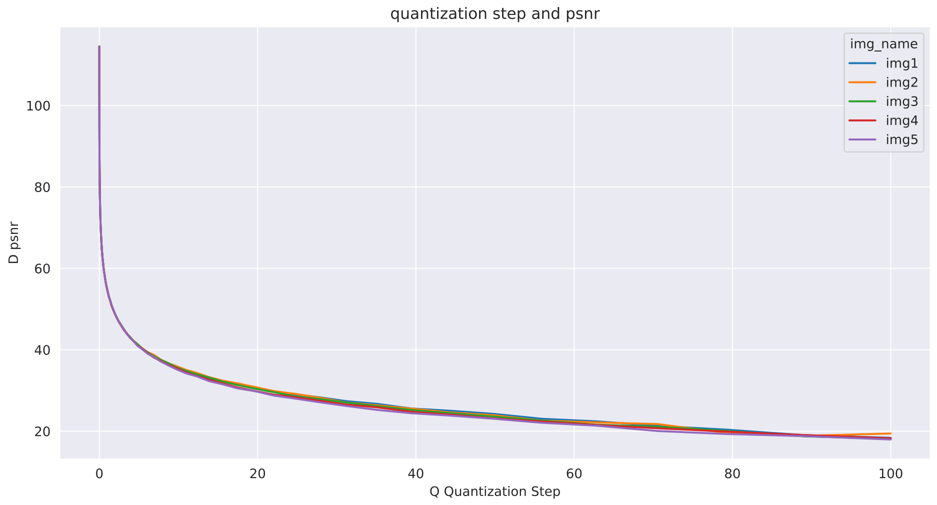
This trend can be explained by the fact that a larger quantization step means that we are keeping more bits to represent the image, leading to higher bpp. Meanwhile, the increase in bpp results in more accurate image representation, thereby reducing distortion and increasing PSNR. However, it's also important to balance the trade-off, as increasing the bpp also results in larger file sizes, which might not be desirable in scenarios with limited storage or bandwidth.

The Rate-Distortion (RD) curve generated from these results can be used to find an optimal quantization step that balances the trade-off between bpp (Rate) and PSNR (Distortion). This is crucial for various applications in image and video compression where the goal is to achieve the highest possible image quality at the lowest possible bit rate.

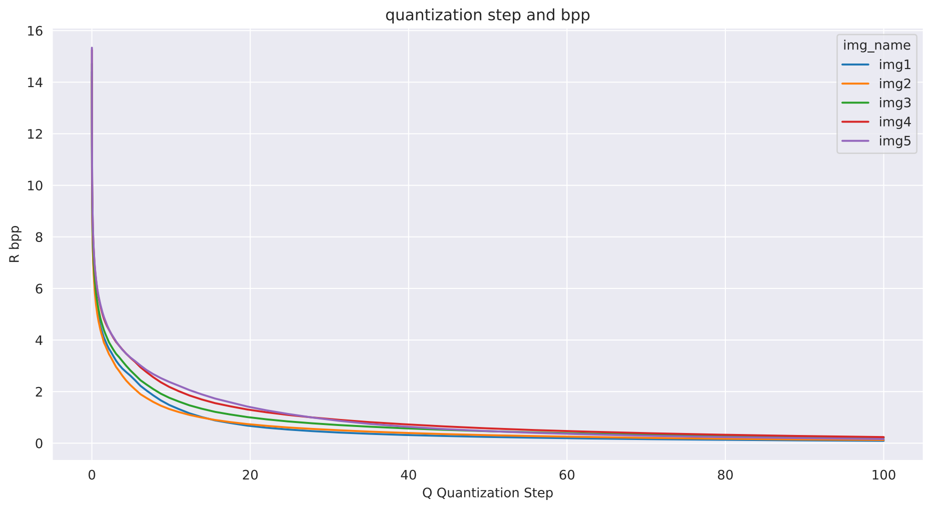
```
[18]: show_svg('/home/fanqy/research-temp/eee5347/final-project/report-img/
        ↪bpp_and_psnr.svg')
```



```
[19]: show_svg('/home/fanqy/research-temp/eee5347/final-project/report-img/
        ↪qs_and_psnr.svg')
```



[20]:
```
show_svg('/home/fanqy/research-temp/eee5347/final-project/report-img/qs_and_bpp.
↪svg')
```



1.9 Conclusion

In conclusion, we presents a comprehensive implementation of the key steps in image compression. It incorporates the application of Discrete Wavelet Transform (DWT), quantization, the zero tree scan, and Huffman encoding. These techniques, when applied in synergy, have proven to be effective in achieving significant compression rates while maintaining a satisfactory level of image quality.

In the course of our investigation, we've managed to reach an impressive compression ratio of up to 6.2 times under conditions where the subjective visual quality of the image is virtually identical to the original (with a PSNR of 35 or higher). This achievement not only validates the effectiveness of our compression strategy but also represents a meaningful contribution to the broader field of image compression.