Daivarsi Malik

CPE-462A

Image Compression

**Abstract**

The aim of this project is to develop a subband image coder utilizing the wavelet transform technique, which has shown enhanced performance relative to the conventional JPEG standard. This initiative encompasses the creation of an encoder that executes subband decomposition, scalar quantization, and entropy encoding, alongside a decoder that reconstructs the image from the encoded bitstream. The coder features a variable quantization step size parameter, which facilitates the management of the compression ratio and file size.

The implementation was performed using MATLAB, capitalizing on its robust signal processing and matrix manipulation functionalities. The procedure initiates subband decomposition via wavelet transforms, which segregates the image into various frequency bands. Subsequently, quantization and entropy encoding are employed to effectively compress the data. The decoder then reverses these operations to reconstruct the image from the encoded file. Performance assessment was conducted by computing the Peak Signal-to-Noise Ratio (PSNR) between the original and reconstructed images.

The findings indicate that the developed subband image coder achieves a high PSNR, reflecting excellent reconstruction quality. Additionally, the ability to modify the quantization step size enables effective balancing between compression ratio and image quality. This project emphasizes the potential of wavelet-based coding techniques, highlighting their significance in contemporary image compression standards such as JPEG2000.

**Description of Work Accomplished**

The execution of the subband image coder involved several essential phases to ensure effective image compression and reconstruction. The initial phase was image preprocessing, during which the input image was either read in grayscale or color format, depending on the user's choice, and subsequently converted to double-precision format for computational efficiency. This implementation was designed to accommodate both grayscale and color images, allowing for a thorough assessment of the coder's performance across various image types.

Following this, wavelet decomposition was applied to the input image. This phase utilized a multi-level wavelet transform to divide the image into approximation (LL) and detail coefficients (LH, HL, HH) at successive levels of decomposition. A specialized 2D wavelet decomposition function was created to ensure precision and effective management of the image data. The next step involved scalar quantization, where a variety of step sizes were employed to quantize the wavelet coefficients. This process allowed for control over the compression ratio, presenting trade-offs between image quality and file size. Smaller quantization step sizes preserved more detail but resulted in larger file sizes, whereas larger step sizes achieved greater compression at the potential expense of quality.

Entropy encoding, specifically through Huffman encoding, was then implemented to efficiently compress the quantized coefficients. The resulting Huffman tables were stored as metadata, which facilitated accurate decoding during the reconstruction phase. For the decoding and reconstruction process, the compressed data underwent entropy decoding, dequantization, and inverse wavelet reconstruction. A custom inverse wavelet transform function was developed to accurately restore the original image from the compressed data.

The coder's performance was assessed by computing the Peak Signal-to-Noise Ratio (PSNR) for the reconstructed images, which acted as an indicator of compression quality. Furthermore, compression ratios were established by comparing the size of the original image with that of the compressed data.

To demonstrate the trade-offs between compression and quality, the interplay among quantization step sizes, PSNR, and compression ratio was graphically represented. Reconstructed images in both grayscale and color formats were preserved to visually illustrate the effects of compression under different quantization scenarios.

It is important to note that grayscale images displayed more straightforward compression properties due to their single-channel format. Conversely, color images, which comprise multiple channels (R, G, B), necessitated individual analysis for each channel. This approach offered valuable insights into how each color channel responds differently to quantization and compression.

**Results**

A person wearing a hat

Description automatically generated

Figure 1 (Original Grayscale BMP Image: 256KB)



Figure 2 (Reconstructed Grayscale PNG Image with Step Size of 5: 122 KB)



Figure 3 (Reconstructed Grayscale PNG Image with Step Size of 10: 79 KB)



Figure 4 (Reconstructed Grayscale PNG Image with Step Size of 15: 58 KB)

For all reconstructed images, 3 decomposition levels were used, allowing for adequate compression while maintaining a high fidelity. The original image was 258KB, as seen in Figure 1. The reconstructed image with a step size of 5, as seen in Figure 2, had an average Peak-to-Signal-Ratio of 44.62, indicating very high-fidelity reconstruction. PSNR above 40 dB is usually considered to be visually lossless. The compression ratio of this image is 2.1:1. The reconstructed image with a step size of 10, as seen in Figure 3, had an average Peak-to-Signal-Ratio of 39.93 and a compression ratio of 3.24:1. There is still very little loss of detail in this reconstruction, while the image size has been reduced to 79KB from 256KB. The reconstructed image with a step size of 15, as seen in Figure 4, had an average Peak-to-Signal-Ratio of 37.61 and a compression ratio of 4.45:1. This image has some noticeable loss of quality, so a step size of 15 may be too large for practical uses of compression where image quality is desired.

In the case of color images, additional procedures are required to manage the various channels, such as Red, Green, and Blue in RGB images, unlike grayscale images that consist of only one channel. Each color channel undergoes independent processing through wavelet decomposition, quantization, entropy encoding, and reconstruction. This methodology guarantees the preservation of the distinct features of each channel during the compression phase. The compressed data for each channel is stored separately, and during the image reconstruction, the channels are decoded and reassembled to create the final color image. This process necessitates supplementary functions for saving and loading the compressed data of each channel, facilitating the efficient management of the multiple elements that constitute the color image. These additional steps are crucial for accurately maintaining the image's color information throughout the compression and reconstruction stages.

A person wearing a hat

Description automatically generated

Figure 5 (Original Color TIFF Image: 769 KB)



Figure 6 (Reconstructed Color PNG Image with Step Size of 5: 430 KB)



Figure 7 (Reconstructed Color PNG Image with Step Size of 10: 342 KB)



Figure 8 (Reconstructed Color PNG Image with Step Size of 15: 255 KB)



Figure 9 (Reconstructed Color PNG Image with Step Size of 20: 171 KB)

The original color image is 769 KB, as seen in Figure 5. The reconstructed image with a step size of 5, as seen in Figure 6, had an average Peak-to-Signal-Ratio of 44.38, indicating very high-fidelity reconstruction. The compression ratio of this image is 1.79:1. The reconstructed image with a step size of 10, as seen in Figure 7, had an average Peak-to-Signal-Ratio of 39.04 and a compression ratio of 2.25:1. The reconstructed image with a step size of 15, as seen in Figure 8, had an average Peak-to-Signal-Ratio of 36.39 and a compression ratio of 3.02:1. There is still little loss of detail in this reconstruction visually, while the image size has been reduced to 255KB from 769KB. The reconstructed image with a step size of 20, as seen in Figure 9, had an average Peak-to-Signal-Ratio of 34.78 and a compression ratio of 3.02:1. This image has some noticeable loss of quality, so a step size beyond 20 may be too large for practical uses of compression where image quality is desired.

It is evident that compression is more efficient for grayscale images than for color images, likely due to several factors. Firstly, grayscale images consist of a single channel, resulting in a reduced amount of data that needs to be processed. In contrast, color images are composed of multiple channels—typically three: Red, Green, and Blue—thereby increasing the volume of data that requires compression. Furthermore, color images frequently exhibit significant redundancy among the channels. The red, green, and blue channels usually display similar patterns and variations; however, because they are compressed separately, this can lead to inefficiencies. Grayscale images, with their singular channel, present less redundancy, facilitating the application of compression techniques.

Additionally, color images often necessitate more sophisticated methods to maintain color accuracy and avoid visual artifacts, which can diminish compression effectiveness. These complexities typically render grayscale images easier to compress with greater efficiency, as they do not involve the additional challenges associated with managing multiple channels and ensuring color fidelity.

**Compiling the Script**

To implement this code for image compression, begin by configuring the input parameters that govern the compression procedure. The primary parameter is input\_image\_path, which indicates the location of the image to be processed. This image may be either a grayscale image (.bmp file) or a color image (.tiff file), based on your selection. The color variable specifies the type of image in use, with 0 representing grayscale and 1 indicating color images. After loading the image, it is transformed into a double-precision matrix to enable mathematical operations necessary for wavelet decomposition and subsequent processing stages.

Subsequently, the decomposition\_levels parameter dictates the number of times the image undergoes decomposition via wavelet transforms. A greater number of decomposition levels yields finer granularity, enhancing compression capabilities, although it may also elevate computational demands. The quantization\_steps array outlines various quantization levels applied to the wavelet coefficients during the compression phase. This parameter balances the compression ratio against image quality; larger quantization steps lead to increased compression but may compromise quality, as indicated by a reduction in PSNR (Peak Signal-to-Noise Ratio).

During the compression procedure, each color channel (for color images) or the single grayscale channel is processed separately. The huffman\_encode and huffman\_decode functions are employed to compress and decompress the wavelet coefficients, with the resulting compressed image being saved in a binary file. You can adjust the step\_size (from the quantization steps) and modify the number of decomposition levels to influence the efficiency of the compression. Following the compression, the script computes the PSNR for each channel and the overall compression ratio, providing insights into image quality and the extent of data reduction. Ultimately, the reconstructed image is saved, typically in .png format, though this can be altered to other formats based on the desired output specifications.

**Conclusion**

The implementation of a multi-level wavelet decomposition and compression technique for image data represents a highly effective strategy for achieving substantial image compression. Utilizing wavelet transforms, the image is broken down into multiple frequency subbands, which are subsequently quantized and encoded to minimize data size. The provided code allows for considerable flexibility in modifying essential parameters, including the level of decomposition, quantization steps, and the type of image (grayscale or color), facilitating a careful adjustment of the balance between compression ratio and image quality.

Grayscale images benefit more from this compression method due to their less complex structure, whereas color images necessitate additional processing for each channel, resulting in a less efficient compression for multi-channel images. The incorporation of Huffman encoding further enhances the optimization of quantized coefficients storage, thereby improving the overall compression ratio. Evaluations based on PSNR and compression ratio calculations reveal the influence of varying quantization steps on the trade-off between image quality and file size.

In summary, the methodology presented in this report establishes a strong basis for image compression, with significant implications for areas where efficient image storage and transmission are essential. By comprehending and adjusting the input parameters, users can customize the compression process to align with their specific requirements, achieving a balance between efficiency and the maintenance of image quality.

Source Code:

% Multi-Level Wavelet Decomposition and Compression

% Specify the full path to the input image

color = 1;

if color == 0

input\_image\_path = 'C:\Users\DSM\Desktop\cpeProject\lena\_gray.bmp';

else

input\_image\_path = 'C:\Users\DSM\Desktop\cpeProject\lena\_color.tiff';

end

% Read the input image

original\_image = imread(input\_image\_path);

% Convert to double for processing

original\_image = double(original\_image);

% Decomposition levels

decomposition\_levels = 3;

% Compression parameters

quantization\_steps = [5, 10, 15, 20]; % Different compression levels

% Process for different quantization steps

for i = 1:length(quantization\_steps)

% Current quantization step

step\_size = quantization\_steps(i);

% Process each color channel separately

num\_channels = size(original\_image, 3);

compressed\_channels = cell(1, num\_channels);

reconstructed\_channels = cell(1, num\_channels);

channel\_sizes = cell(1, num\_channels);

for channel = 1:num\_channels

% Extract current channel

current\_channel = original\_image(:,:,channel);

% Multi-level Wavelet Decomposition

[coeffs, sizes] = multi\_level\_wavelet\_decomposition(current\_channel, decomposition\_levels);

% Scalar Quantization of coefficients

quantized\_coeffs = cell(size(coeffs));

for j = 1:length(coeffs)

quantized\_coeffs{j} = round(coeffs{j} / step\_size);

end

% Entropy Encoding (Huffman)

encoded\_data = cell(size(coeffs));

huffman\_tables = cell(size(coeffs));

compressed\_sizes = zeros(size(coeffs));

for j = 1:length(quantized\_coeffs)

[encoded\_data{j}, huffman\_tables{j}] = huffman\_encode(quantized\_coeffs{j});

compressed\_sizes(j) = numel(encoded\_data{j});

end

% Store compressed channel data

compressed\_channels{channel} = encoded\_data;

channel\_sizes{channel} = sizes;

% Huffman decoding

decoded\_coeffs = cell(size(coeffs));

for j = 1:length(encoded\_data)

decoded\_coeffs{j} = huffman\_decode(encoded\_data{j}, huffman\_tables{j}, ...

size(quantized\_coeffs{j}));

end

% Dequantization

dequantized\_coeffs = cell(size(decoded\_coeffs));

for j = 1:length(decoded\_coeffs)

dequantized\_coeffs{j} = decoded\_coeffs{j} \* step\_size;

end

% Reconstruct Channel

reconstructed\_channel = multi\_level\_wavelet\_reconstruction(dequantized\_coeffs, sizes, decomposition\_levels);

reconstructed\_channels{channel} = max(min(round(reconstructed\_channel), 255), 0);

end

% Combine reconstructed channels

reconstructed\_image = zeros(size(original\_image));

for channel = 1:num\_channels

reconstructed\_image(:,:,channel) = reconstructed\_channels{channel};

end

% Save the compressed data

[folder, name, ~] = fileparts(input\_image\_path);

compressed\_filename = fullfile(folder, sprintf('%s\_compressed\_step\_%d.bin', name, step\_size));

% Save compressed data with color channel information

save\_color\_compressed\_data(compressed\_filename, compressed\_channels, huffman\_tables, ...

channel\_sizes, decomposition\_levels, step\_size, size(original\_image));

% Calculate PSNR for each channel and average

psnr\_values = zeros(1, num\_channels);

for channel = 1:num\_channels

psnr\_values(channel) = calculate\_psnr(original\_image(:,:,channel), ...

reconstructed\_image(:,:,channel));

end

psnr\_value = mean(psnr\_values);

% Calculate Compression Ratio

original\_size\_bits = numel(original\_image) \* 8;

compressed\_size\_bits = sum(compressed\_sizes) \* num\_channels;

compression\_ratio = original\_size\_bits / compressed\_size\_bits;

% Save reconstructed image

filename = fullfile(folder, sprintf('%s\_reconstructed\_step\_%d.png', name, step\_size));

imwrite(uint8(reconstructed\_image), filename);

% Print compression details

fprintf('Quantization Step: %d\n', step\_size);

fprintf('Decomposition Levels: %d\n', decomposition\_levels);

fprintf('Average PSNR: %.2f dB\n', psnr\_value);

fprintf('Individual Channel PSNRs: R=%.2f, G=%.2f, B=%.2f dB\n', psnr\_values);

fprintf('Compression Ratio: %.2f:1\n', compression\_ratio);

fprintf('Original Size: %.2f KB\n', original\_size\_bits/8/1024);

fprintf('Compressed Size: %.2f KB\n', compressed\_size\_bits/8/1024);

fprintf('Reconstructed Image Saved: %s\n\n', filename);

end

% Visualization of results

figure;

subplot(2,1,1);

plot(quantization\_steps, calculate\_psnr\_values(original\_image, quantization\_steps, decomposition\_levels), '-o');

title('PSNR vs Quantization Step Size');

xlabel('Quantization Step Size');

ylabel('PSNR (dB)');

subplot(2,1,2);

plot(quantization\_steps, calculate\_compression\_ratios(original\_image, quantization\_steps, decomposition\_levels), '-o');

title('Compression Ratio vs Quantization Step Size');

xlabel('Quantization Step Size');

ylabel('Compression Ratio');

function [coeffs, sizes] = multi\_level\_wavelet\_decomposition(image, levels)

% Initialize coefficients storage

coeffs = cell(1, levels \* 3 + 1);

sizes = cell(1, levels + 1);

% Current image for decomposition

current\_image = image;

% Store original image size

sizes{1} = size(current\_image);

% Perform multi-level decomposition

for level = 1:levels

% Ensure even dimensions

[rows, cols] = size(current\_image);

rows = floor(rows/2)\*2;

cols = floor(cols/2)\*2;

current\_image = current\_image(1:rows, 1:cols);

% Perform 2D wavelet decomposition

[LL, LH, HL, HH] = perform\_2d\_wavelet\_decomposition(current\_image);

% Store coefficient matrices

coeffs{level\*3-2} = LH;

coeffs{level\*3-1} = HL;

coeffs{level\*3} = HH;

% Store size of current decomposition

sizes{level + 1} = size(LL);

% Continue decomposition on LL band

current\_image = LL;

end

% Store the final LL band

coeffs{end} = current\_image;

end

function [LL, LH, HL, HH] = perform\_2d\_wavelet\_decomposition(image)

% Get dimensions

[rows, cols] = size(image);

% Ensure even dimensions

rows = floor(rows/2)\*2;

cols = floor(cols/2)\*2;

image = image(1:rows, 1:cols);

% Initialize subbands

LL = zeros(rows/2, cols/2);

LH = zeros(rows/2, cols/2);

HL = zeros(rows/2, cols/2);

HH = zeros(rows/2, cols/2);

% Process rows

temp\_L = zeros(rows, cols/2);

temp\_H = zeros(rows, cols/2);

for i = 1:rows

for j = 1:cols/2

temp\_L(i,j) = (image(i,2\*j-1) + image(i,2\*j))/sqrt(2);

temp\_H(i,j) = (image(i,2\*j-1) - image(i,2\*j))/sqrt(2);

end

end

% Process columns

for j = 1:cols/2

for i = 1:rows/2

LL(i,j) = (temp\_L(2\*i-1,j) + temp\_L(2\*i,j))/sqrt(2);

LH(i,j) = (temp\_L(2\*i-1,j) - temp\_L(2\*i,j))/sqrt(2);

HL(i,j) = (temp\_H(2\*i-1,j) + temp\_H(2\*i,j))/sqrt(2);

HH(i,j) = (temp\_H(2\*i-1,j) - temp\_H(2\*i,j))/sqrt(2);

end

end

end

function reconstructed\_image = multi\_level\_wavelet\_reconstruction(coeffs, sizes, levels)

% Start with the lowest frequency band

current\_image = coeffs{end};

% Reconstruct from lowest to highest frequency

for level = levels:-1:1

% Get detail coefficients for current level

LH = coeffs{level\*3-2};

HL = coeffs{level\*3-1};

HH = coeffs{level\*3};

% Verify all subbands have the same size

if ~isequal(size(current\_image), size(LH), size(HL), size(HH))

% Resize all subbands to match the expected size from sizes array

target\_size = sizes{level};

current\_image = imresize(current\_image, target\_size);

LH = imresize(LH, target\_size);

HL = imresize(HL, target\_size);

HH = imresize(HH, target\_size);

end

% Perform inverse wavelet transform

current\_image = perform\_2d\_inverse\_wavelet\_reconstruction(current\_image, LH, HL, HH);

end

% Final resize to original image size if needed

if ~isequal(size(current\_image), sizes{1})

reconstructed\_image = imresize(current\_image, sizes{1});

else

reconstructed\_image = current\_image;

end

end

function reconstructed = perform\_2d\_inverse\_wavelet\_reconstruction(LL, LH, HL, HH)

% Get dimensions from input

[rows, cols] = size(LL);

% Pre-allocate arrays with correct sizes

temp\_L = zeros(2\*rows, cols);

temp\_H = zeros(2\*rows, cols);

reconstructed = zeros(2\*rows, 2\*cols);

% Process columns - inverse vertical transform

for j = 1:cols

for i = 1:rows

% Ensure indices are within bounds

idx1 = min(2\*i-1, 2\*rows);

idx2 = min(2\*i, 2\*rows);

% Apply inverse transform

temp\_L(idx1,j) = (LL(i,j) + LH(i,j))/sqrt(2);

temp\_L(idx2,j) = (LL(i,j) - LH(i,j))/sqrt(2);

temp\_H(idx1,j) = (HL(i,j) + HH(i,j))/sqrt(2);

temp\_H(idx2,j) = (HL(i,j) - HH(i,j))/sqrt(2);

end

end

% Process rows - inverse horizontal transform

for i = 1:2\*rows

for j = 1:cols

% Ensure indices are within bounds

idx1 = min(2\*j-1, 2\*cols);

idx2 = min(2\*j, 2\*cols);

% Apply inverse transform

reconstructed(i,idx1) = (temp\_L(i,j) + temp\_H(i,j))/sqrt(2);

reconstructed(i,idx2) = (temp\_L(i,j) - temp\_H(i,j))/sqrt(2);

end

end

end

function [encoded\_data, huffman\_table] = huffman\_encode(data)

% Convert input data to a column vector of integers

data\_vector = round(data(:));

% Get unique values and their frequencies

unique\_vals = unique(data\_vector);

freq = histcounts(data\_vector, [unique\_vals; max(unique\_vals)+1]);

% Create Huffman dictionary

p = freq/sum(freq);

dict = huffmandict(unique\_vals, p);

% Encode the data

encoded\_data = huffmanenco(data\_vector, dict);

% Store the Huffman table for decoding

huffman\_table = dict;

end

function decoded\_data = huffman\_decode(encoded\_data, huffman\_table, target\_size)

% Decode the data

decoded\_vector = huffmandeco(encoded\_data, huffman\_table);

% Reshape to original dimensions

decoded\_data = reshape(decoded\_vector, target\_size);

end

% Added helper function to store sizes in metadata

function psnr\_value = calculate\_psnr(original, reconstructed)

mse = mean((original(:) - reconstructed(:)).^2);

max\_pixel = max(original(:));

psnr\_value = 10 \* log10((max\_pixel^2) / mse);

end

function psnr\_values = calculate\_psnr\_values(original\_image, quantization\_steps, decomposition\_levels)

psnr\_values = zeros(size(quantization\_steps));

for i = 1:length(quantization\_steps)

step\_size = quantization\_steps(i);

% Perform decomposition

[coeffs, sizes] = multi\_level\_wavelet\_decomposition(original\_image, decomposition\_levels);

% Quantize

quantized\_coeffs = cell(size(coeffs));

for j = 1:length(coeffs)

quantized\_coeffs{j} = round(coeffs{j} / step\_size);

end

% Dequantize

dequantized\_coeffs = cell(size(quantized\_coeffs));

for j = 1:length(quantized\_coeffs)

dequantized\_coeffs{j} = quantized\_coeffs{j} \* step\_size;

end

% Reconstruct

reconstructed = multi\_level\_wavelet\_reconstruction(dequantized\_coeffs, sizes, decomposition\_levels);

% Adjust brightness

reconstructed = reconstructed - min(reconstructed(:));

reconstructed = reconstructed / max(reconstructed(:)) \* 255;

% Calculate PSNR

psnr\_values(i) = calculate\_psnr(original\_image, reconstructed);

end

end

function compression\_ratios = calculate\_compression\_ratios(original\_image, quantization\_steps, decomposition\_levels)

compression\_ratios = zeros(size(quantization\_steps));

original\_size\_bits = numel(original\_image) \* 8;

for i = 1:length(quantization\_steps)

step\_size = quantization\_steps(i);

% Decomposition

[coeffs, sizes] = multi\_level\_wavelet\_decomposition(original\_image, decomposition\_levels);

% Quantize

quantized\_coeffs = cell(size(coeffs));

for j = 1:length(coeffs)

quantized\_coeffs{j} = round(coeffs{j} / step\_size);

end

% Huffman encode

total\_compressed\_bits = 0;

for j = 1:length(quantized\_coeffs)

[encoded\_data, ~] = huffman\_encode(quantized\_coeffs{j});

total\_compressed\_bits = total\_compressed\_bits + length(encoded\_data);

end

% Calculate compression ratio

compression\_ratios(i) = original\_size\_bits / total\_compressed\_bits;

end

end

function save\_color\_compressed\_data(filename, compressed\_channels, huffman\_tables, ...

channel\_sizes, decomposition\_levels, step\_size, image\_size)

% Save metadata

metadata\_filename = [filename(1:end-4) '\_meta.mat'];

save(metadata\_filename, 'huffman\_tables', 'channel\_sizes', 'decomposition\_levels', ...

'step\_size', 'image\_size', '-v7.3');

% Save encoded binary data

fid = fopen(filename, 'wb');

% Write number of channels

num\_channels = length(compressed\_channels);

fwrite(fid, num\_channels, 'uint32');

% For each channel

for channel = 1:num\_channels

encoded\_data = compressed\_channels{channel};

% Write number of subbands

fwrite(fid, length(encoded\_data), 'uint32');

% Write each subband's encoded data

for i = 1:length(encoded\_data)

% Store the original length

data\_length = length(encoded\_data{i});

fwrite(fid, data\_length, 'uint32');

% Write the actual data

fwrite(fid, encoded\_data{i}, 'uint8');

end

end

fclose(fid);

end

% Add new function to load color compressed data

function [compressed\_channels, huffman\_tables, channel\_sizes, decomposition\_levels, ...

step\_size, image\_size] = load\_color\_compressed\_data(filename)

% Load metadata

metadata\_filename = [filename(1:end-4) '\_meta.mat'];

metadata = load(metadata\_filename);

huffman\_tables = metadata.huffman\_tables;

channel\_sizes = metadata.channel\_sizes;

decomposition\_levels = metadata.decomposition\_levels;

step\_size = metadata.step\_size;

image\_size = metadata.image\_size;

% Load binary data

fid = fopen(filename, 'rb');

% Read number of channels

num\_channels = fread(fid, 1, 'uint32');

compressed\_channels = cell(1, num\_channels);

% For each channel

for channel = 1:num\_channels

% Read number of subbands

num\_subbands = fread(fid, 1, 'uint32');

encoded\_data = cell(1, num\_subbands);

% Read each subband's data

for i = 1:num\_subbands

data\_length = fread(fid, 1, 'uint32');

encoded\_data{i} = fread(fid, data\_length, 'uint8')';

end

compressed\_channels{channel} = encoded\_data;

end

fclose(fid);

end