507 Project
DCT for Image Compression

Shantanu Patne

April 29, 2024

1 Introduction

1.1 Introduction to Project

Discrete Cosine Transform (DCT) is a signal transformation technique similar to the Discrete Fourier Transform (DFT) but expressly for real-valued signals. In this project, we experiment on using this technique as a means of compression. We apply this technique on 8×8 non-overlapping blocks of a 512×512 image, a real-valued 2-dimensional signal to obtain the signal coefficients arranged in order of the frequencies, with the highest frequency components at the corners away from the origin, i.e. (0,0) index. We window these coefficients to zero out the higher-frequency components, quantize the remaining coefficients to calculate the transmission requirements, and then apply the inverse DCT (IDCT) to get the compressed DCT to obtain the compressed image. We experiment with different windows and estimate the quality of the image as well as the transmission requirements for the quantized image. The image used for this project is the widely used, feature-rich Lena image shown in Fig. 1.

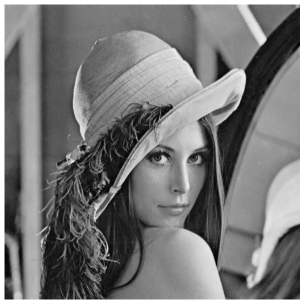

Figure 1: Lena image used in various Image Processing tasks.

The report on this project is arranged as follows: Section 1 introduces the project and briefly highlights the theory of DCT. Sections 2 and 3 discuss the MATLAB implementation of the experiment and the results obtained, while Section 4 provides the concluding remarks of this experiment.

1.2 DCT Theory

Originally devised by Nasir Ahmed, K. R. Rao, and T. Natarajan[1], the DCT algorithm is widely used in modern compression techniques, most prominently in JPEG, MPEG, and H.265[2, 3]. While it has undergone many variations, the DCT used in the project can be seen in Eq. 2. The DCT is a finite, linear, invertible, and real-valued transform, similar to the DFT in that it expresses the input signal in terms of sinusoids. However, the DFT uses complex sinusoids as the basis while the DCT uses only the real-valued cosine, as the name suggests. It overcomes one of the major drawbacks of DFT; even for real data, the DFT is complex-valued[2]. Its energy compaction property makes it highly suitable for compression[4, 5].

For an N-sized, finite, 1-D signal, the DCT is given as,

$$X(k) = \sum_{n=0}^{N-1} 2x(n) cos[\frac{\pi k}{2N}(2n+1)], \tag{1}$$

for $k \in [0, N-1]$.

Extending this to 2-D images, for a signal with support $[N_1 \times N_2]$

$$X(k_1,k_2) = \sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1} 4x(n_1,n_2) cos[\frac{\pi k_1}{2N_1}(2n_1+1)] cos[\frac{\pi k_2}{2N_2}(2n_2+1)], \tag{2}$$

for $k_1, k_2 \in [0, N_1-1] \times [0, N_2-1]$.

Since this is an invertible transform, we can obtain the original image from its DCT as,

$$x(n_1,n_2) = \frac{1}{N_1 N_2}\sum_{k_1=0}^{N_1-1}\sum_{k_2=0}^{N_2-1} W(k_1)W(k_2)X(k_1,k_2) cos[\frac{\pi k_1}{2N_1}(2n_1+1)] cos[\frac{\pi k_2}{2N_2}(2n_2+1)] \tag{3}$$

where the weighting function, $W(k)$ is given as,

$$W(k) = \begin{cases} 1/2, & k = 0 \\ 1, & k \neq 0 \end{cases} \tag{4}$$

From Eqs. 2 and 3[2], it is obvious that the transform is separable into 1-D transforms. This can be done by initially applying the 1-D transform in Eq. 1 to all the rows (or columns) of the image and then applying the transform over the computed values column-wise (or row-wise).

The key property of DCT, which makes it perfect for compression, is the energy compaction property. The energy of the signal is concentrated in the low-frequency components[5, 4]. This is especially useful for image compression since low-frequencies in images generally represent global features and smooth patterns while high-frequency components generally represent details[4]. Thus, since most energy is conserved in the low-frequency components, we can essentially remove high-frequency coefficients in the DCT and still preserve the signal energy to a large extent. This means that even if we zeroed out some high-frequency components, we can still preserve image quality. The percentage of high-frequencies removed does affect the quality of the image, however, since the details in the image are removed.

In fact, as we will see in Section 4, even after removing 50% of the coefficients, we see no degradation in the image or its quality.

2 MATLAB Implementation

The MATLAB implementation for the experiments is straightforward. First, the RAW image is read and displayed. It is then separated into 8×8 non-overlapping blocks for processing. The DCT is applied to each of the blocks sequentially. Then we apply a window to the DCT coefficients to remove the high-frequency components. For this experiment, we zero out 50%, 75%, 90%, and 95% of the DCT coefficients. In each case, we quantize the remaining coefficients using an 8-bit scalar uniform quantizer. We then reconstruct the quantized image to compute the inverse DCT for displaying. The quantized blocks are used to compute the transmission requirements (total bits, average bits per pixel). The 2-D DFT of the image is also displayed along with the reconstructed image to show the effect of compression in the frequency domain. All codes will be listed in the Appendix.

2.1 Block Separation and DCT

The image is segmented into 8x8 blocks in the `computeSegments` function defined in the file with the same name. This function separates the image into the required blocks and saves it in a single 3-dimensional variable of size `(8,8,4096)`. We then pass the output segments to the `computeDCT`

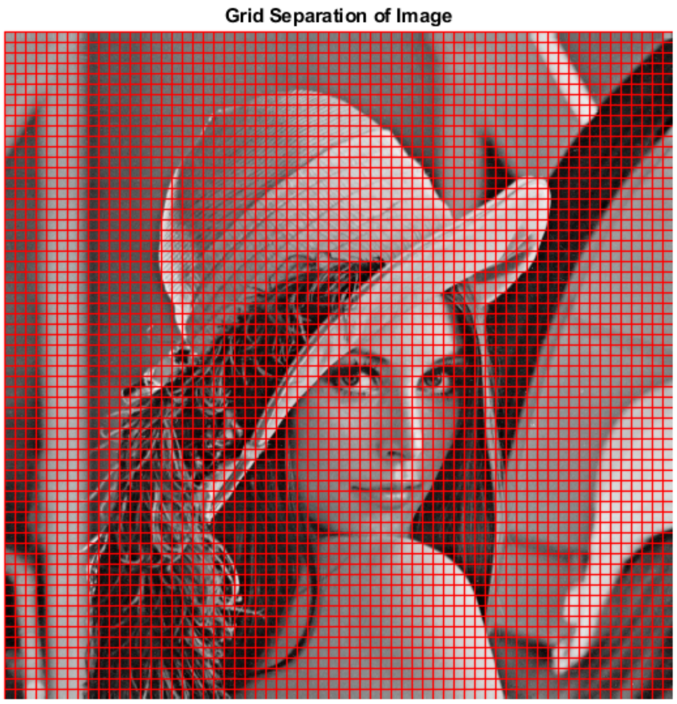

Figure 2: Overlay of the 8x8 blocks over the Lena image

function to obtain the DCT for each of the 8×8 blocks. We can see the grid obtained from this separation in Fig. 2.

The DCT computation is vectorized for efficiency and we iterate over each image and k_1 and k_2 defined in Eq. 2. The algorithm for the computation is highlighted in Algo. 1. Since the image block is a 2-D vector, we calculate the cosines in vector format and compute the sum of the vector product for every DCT index. This vectorization makes the code run in approx. 0.3 seconds for the computation of all 4096 block segments, as compared to 0.6 seconds taken for individual blocks in non-vectorized computation.

Algorithm 1 DCT Computation Algorithm

```
Require: image segments of size $8 \times 8$
  $num\_blocks \leftarrow len(segments)$
  for $block$ in $num\_blocks$ do
      img ← segments(block)
      for $k_1 = 1$ to $N_1$ do
          for $k_2 = 1$ to $N_2$ do
              $n_1 \leftarrow 0 : N_1 - 1$
              $n_2 \leftarrow 0 : N_2 - 1$
              matrix multiplication of $img$ with cosine vectors from $n_1$ and $n_2$
              $X_c(k_1, k_2, block) \leftarrow$ sum of matrix product over all dimensions
          end for
      end for
  end for
  return $X_c$
```

The DCT coefficients obtained from this computation have the DC component in the first index or *origin*, i.e. (0,0) for every block. This DC component is significantly larger than the rest of the coefficients, proving the energy compacting property of the DCT. We can use windowing to remove the high-frequency components from the diagonally opposite end of the origin, i.e. near index (7,7).[2]

2.2 Windowing and Quantization

The next step in the experiment is to remove a certain percentage of the coefficients from the high-frequency end of each block. This is achieved by simply multiplying the block, element-wise, with a window matrix. For 50% window, the window matrix is obtained as,

$$\begin{bmatrix} 1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&0\\ 1&1&1&1&1&1&0&0\\ 1&1&1&1&1&0&0&0\\ 1&1&1&1&0&0&0&0\\ 1&1&1&0&0&0&0&0\\ 1&1&0&0&0&0&0&0\\ 1&0&0&0&0&0&0&0 \end{bmatrix},$$

and so on.

Using this windowing mechanism, we can remove the lower corner frequencies easily. These windows are generated by the `getWindow` function with an input parameter `percent`. The `percent` parameter defines the percentage of coefficients that have to be removed, and calculates the window matrix accordingly.

Following this, we quantize the image using an 8-bit uniform scalar quantization technique. In a scalar quantizer, the input values are mapped to a discrete set of output values[2]. In an 8-bit quantizer, we have 256 discrete output values. The quantizer equation is as given below,

$$Q(x) = \Delta \cdot \lfloor \frac{x}{\Delta} + \frac{1}{2} \rfloor$$
$$and$$
$$\Delta = (x_{max} - x_{min})/256$$

where the forward quantization part is given as,

$$k = \lfloor \frac{x}{\Delta} + \frac{1}{2} \rfloor \tag{5}$$

and the reconstruction is given as,

$$Q(x) = \Delta \cdot k \tag{6}$$

For our purpose, the DCT coefficients are first forward quantized to get the bit information and then reconstructed to get the original coefficients. However, the DC coefficients are too large to be quantized together with the rest of the coefficients. So, the DC coefficients of all blocks are quantized together, while the rest of the block coefficients are quantized block-wise.

2.3 IDCT and Block Reconstruction

The IDCT implementation is similar to that of the DCT, with the only change being the equation used. The IDCT uses Eq. 3 and includes the weighting function. This weighting function can also be coded as a vector for our vectorized implementation to reduce the running time. As with the DCT, vectorization significantly speeds the operation and we can compute the IDCT of all 4096 blocks in 0.3 seconds. Also, like the DCT implementation, we simply pass all 4096 blocks of the windowed, unquantized coefficients to get a the reconstructed image blocks.

The reconstruction of the image from the reconstructed image blocks is simply the opposite of the segmentation portion. We set the 8×8 blocks into an empty 512×512 image by iterating over the blocks. The resulting grid is the same as seen in Fig. 2.

3 Results & Discussion

Following the above operations, we use the MATLAB `fft2()` function to get the 2-D DFT of the original and reconstructed image for each window case. This helps us to visualize the change in the frequency components of the image as well as compare the image quality between the original and compressed images. In each case, we calculate the total bits required to represent the quantized coefficients and the average bits required to represent each pixel of the complete 512×512 image. We also compare the final reconstructed image with the original using the Peak Signal-to-Noise Ratio (PSNR). The equation for PSNR is given as,

$$PSNR = 10 \cdot \log_{10}(\frac{peakval^2}{MSE}), \tag{7}$$

where *peakval* is the maximum possible value in the image (255) and MSE is the Mean Squared Error between the reconstructed and original images.

The quality of the image is dependent on the value of PSNR, i.e. the higher the PSNR, the better the quality. The original image along with its 2-D DFT is seen in Fig. 3

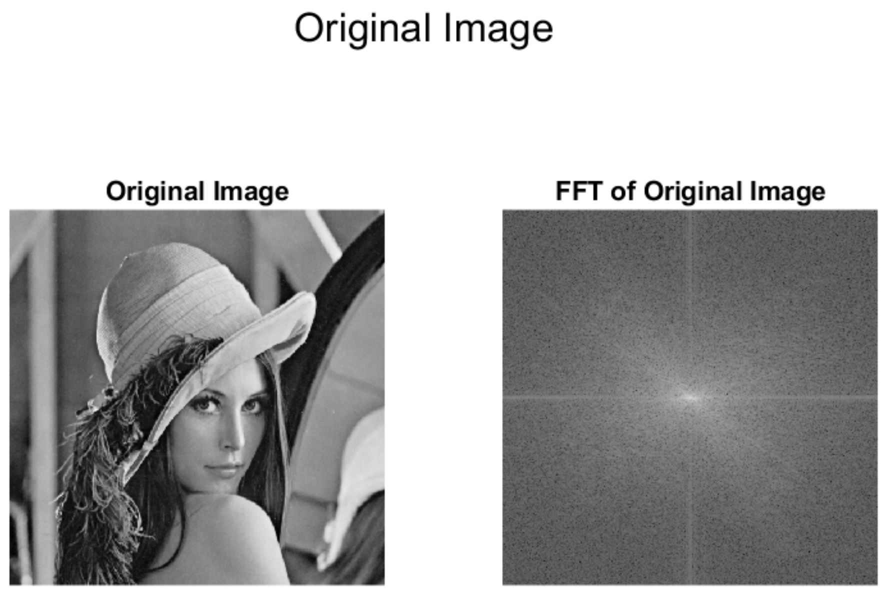

Figure 3: Original Image and DFT

We also generate and show the reconstructed image and its DFT for each window case (50, 75, 90, 95). The table of computed values is given in Table 1

Window	50%	75%	90%	95%
Total bits	1147672	484736	195520	98176
Avg bits/pixel	4.3780	1.8491	0.7458	0.3745
PSNR	40.5144	34.3163	29.8860	27.3358

Table 1: Table of values for each window case

The image degradation is clearly visible as we remove more and more DCT coefficients. The images in Fig. 6 and 7 also show block artifacts. This is due to the non-overlapping nature of the 8×8 blocks[6]. The DFT of each case also shows the high frequencies reduced as we increase the removal percentage.

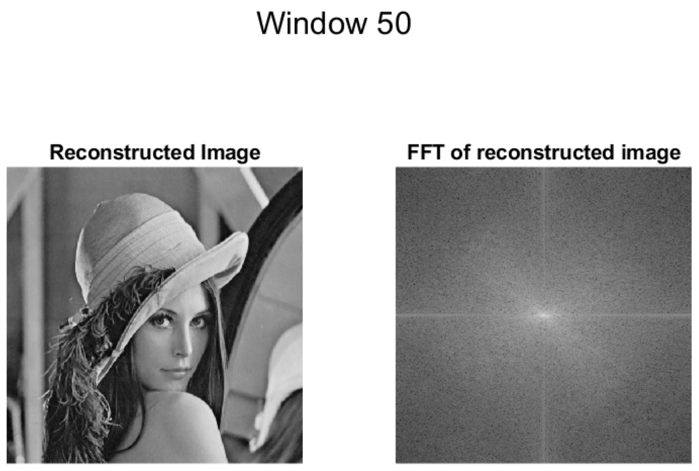

Figure 4: 50% Windowing

In Fig. 4, i.e. 50% coefficients removed, the reconstructed image looks visually identical to the original image. There is no degradation in image quality and no block artifacts can be seen. For compression purposes, this window achieves the best image quality, as is evident in the PSNR value of this image. However, this case has the highest average bits per pixel meaning it has the lowest compression of all cases. It is suitable for high-resolution applications where the image will be displayed largely scaled.

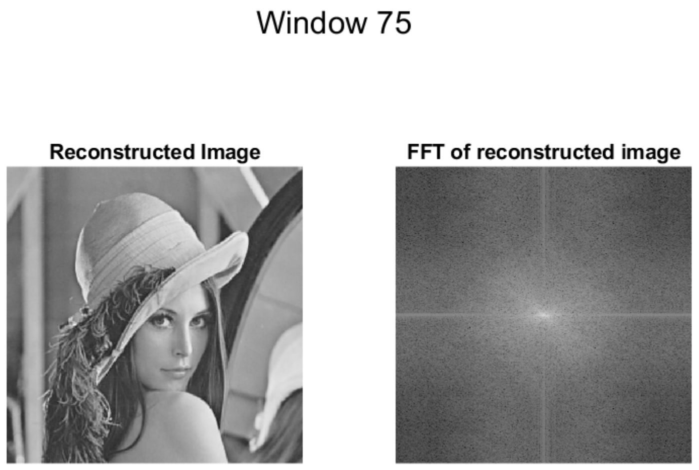

Figure 5: 75% Windowing

In Fig. 5, the image quality is visibly reduced even though there are no block artifacts. The details in the image are removed and the image looks slightly blurred. This is due to the larger number of high-frequency coefficients removed as compared to the 50% case. The average bits per pixel in this case is quite low, giving a good tradeoff between the image quality and compression. The DFT of the image also shows the low-frequency components are intact while the number of high-frequency components removed is larger. This makes it suitable for medium-resolution applications.

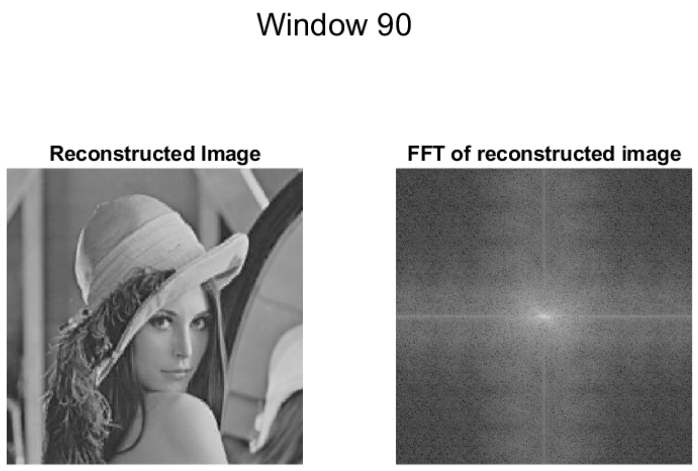

Figure 6: 90% Windowing

Fig. 6 shows the effect of windowing 90% of the DCT coefficients. At a small resolution, this image looks perfectly fine with minor degradation and little to no block artifacts, making this window suitable for applications where the image size will be small and no scaling will occur. However, when scaled, the degradation and artifacts are quite clearly visible. Removing large amounts of coefficients in a non-overlapping manner since there are abrupt discontinuities in the block boundaries[6]. The DFT shows the signal energy is concentrated along the axes and at the origin. With a very low average bits per pixel value, this image has a PSNR of approx. 30, making it suitable only for applications stated above, viz. small image-displaying applications.

The poorest quality image is reconstructed when a 95% window is applied. This is evident in Fig. 7. Even at a small scale, the block artifacts can be seen clearly. The details in the image, mainly the details in the feather hat, are lost and look very blurred. The DFT shows the energy concentrated at the origin with very little energy in medium and high frequencies. This is only suitable for very small images where the degradation won't be visible.

4 Conclusion

In conclusion, the task of creating an image compression system using the 2-D Discrete Cosine Transform (DCT) involved several key steps, including segmenting the image into 8x8 non-overlapping pixel blocks, applying the 2-D DCT independently to each block, and quantizing the DCT coefficients while zeroing out a certain percentage of higher-frequency coefficients.

By varying the percentage of coefficients set to zero (50%, 75%, 90%, and 95%) and quantizing the remaining coefficients using an 8-bit uniform scalar quantizer, we were able to analyze the impact on compression efficiency and image quality. The total number of bits used to code the image and the average number of bits per pixel were computed for each case to evaluate compression performance.

Furthermore, the quality of the reconstructed images was assessed visually, and the Peak Signal-to-

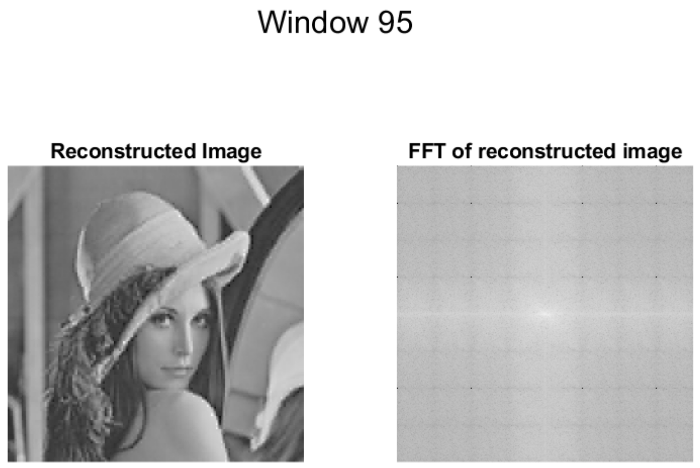

Figure 7: 95% Windowing

Noise Ratio (PSNR) was calculated to provide a quantitative measure of image fidelity. PSNR values were used to compare the reconstructed images with the original image, with higher PSNR values indicating better reconstruction quality.

Additionally, the 2-D Discrete Fourier Transform (DFT) magnitude spectrum of each reconstructed image was plotted and analyzed. This allowed us to examine the frequency content of the reconstructed images and assess the impact of compression on the spectral characteristics.

Overall, this task provided valuable insights into image compression techniques using the 2-D DCT, demonstrating the trade-offs between compression efficiency and image quality. By understanding concepts such as quantization, PSNR, and spectral analysis, we gained a deeper understanding of image compression principles and their practical implications.

References

[1] N. Ahmed, T. Natarajan, and K. R. Rao, “Discrete cosine transform,” *IEEE Transactions on Computers*, 1974.

[2] J. W. Woods, *Multidimensional Signal, Image, and Video Processing and Coding.* Elsevier, 2006.

[3] M. Barbero, H. Hofmann, and N. D. Wells, “Dct source coding and current implementations for hdtv,” *EBU Technical Review*, 1992.

[4] K. Birney and T. Fischer, “On the modeling of dct and subband image data for compression,” *IEEE Transactions on Image Processing*, 1995.

[5] G. Mandyam, N. Ahmed, and N. Magotra, “Lossless image compression using the discrete cosine transform,” *Journal of Visual Communication and Image Representation*, 1997.

[6] J. Luo, C. W. Chen, K. Parker, and T. Huang, “Artifact reduction in low bit rate dct-based image compression,” *IEEE Transactions on Image Processing*, 1996.

Appendix

Code

Listing 1 Image block generation

```
function [segments] = segmentImage(image)
% segmentImage Segments the input image into 8x8 blocks
%   image = Input image to segment
%   segments = segmented blocks

    segments = zeros(8, 8, 64*64);
    for i = 1:64
        h = (i-1)*64;
        for j = 1:64
            segments(:,:,h+j) = image((i-1)*8 + 1: i*8, (j-1)*8 + 1: j*8);
        end
    end
end
```

Listing 2 DCT Computation

```
function [dct_out] = computeDCT(segments)
%   computeIDCT Computes the DCT on all blocks in the input
%
%   segments    size: (8, 8, 4096), segmented image blocks
%   dct_out     size: (8, 8, 4096), computed dct blocks
%

    [N1, N2, blocks] = size(segments);
    dct_out = zeros(size(segments));
    for block = 1:blocks
        img = segments(:,:,block);
        for k1 = 1:N1
            for k2 = 1:N2
                n1 = 0:N1-1;
                n2 = 0:N2-1;
                prod = 4 * img .* cos(pi*(k1 - 1)*(2*n1 + 1)/(2*N1)).' .* cos(pi*(k2
                ↪  - 1)*(2*n2 + 1)/(2*N2));
                dct_out(k1,k2, block) = sum(prod, "all");
            end
        end
    end
end
```

Listing 3 Applying Window to DCT coefficients

```
    % For 50% removal
    window = getWindow(50);
    dct_blocks = computeDCT(segments);
    dct_blocks = win.*dct_blocks;
```

Listing 4 8-bit Uniform Scalar Quantization

```
function [quantized_images, unquantized_images] = quantizeBlock(images)
% quantizeBlock quantize the image blocks. Separately quantize first pixels
%               then quantize the rest values in the block
%
%   images              (8,8,4096) input dct blocks
%   quantized_images    (8,8,4096) forward quantized blocks
%   unquantized_images  (8,8,4096) reconstructed dct blocks


    num_bits = 8;
    quant_levels = 2^num_bits;
    quantized_images = zeros(size(images));
    unquantized_images = zeros(size(images));

    % Quantize the (1,1) pixels of all images together
    first_pixels = images(1, 1, :);
    min_first_pixel = min(first_pixels(:));
    max_first_pixel = max(first_pixels(:));
    delta_first_pixel = (max_first_pixel - min_first_pixel) / quant_levels;
    quantized_first_pixels = round((first_pixels/delta_first_pixel) + 0.5);
    unquantized_first_pixels = quantized_first_pixels * delta_first_pixel;

    for k = 1:size(images, 3)
        non_zero_elements = images(:,:,k);
        non_zero_elements = non_zero_elements(non_zero_elements ~= 0);
        non_zero_elements(1) = [];  % Exclude (1,1) pixel

        min_val = min(non_zero_elements);
        max_val = max(non_zero_elements);
        delta = (max_val - min_val) / quant_levels;

        quantized_image = images(:,:,k);
        unquantized_image = quantized_image;
        for i = 1:numel(non_zero_elements)
            quantized_value = round((non_zero_elements(i)/delta) + 0.5);
            quantized_image(quantized_image == non_zero_elements(i)) =
            ↪  quantized_value;
            unquantized_image(unquantized_image == non_zero_elements(i)) =
            ↪  quantized_value * delta;
        end

        % Save the quantized image
        quantized_images(:,:,k) = quantized_image;
        quantized_images(1,1,k) = quantized_first_pixels(:,:,k);

        % Save the unquantized image
        unquantized_images(:,:,k) = unquantized_image;
        unquantized_images(1,1,k) = unquantized_first_pixels(:,:,k);
    end
end
```

Listing 5 IDCT Computation

```
function [image_out] = computeIDCT(dct_blocks)
%   computeIDCT Computes the inverse DCT on all blocks in the input
%
%   dct_blocks  size: (8, 8, 4096), windowed dct coefficients
%   image_out   size: (8, 8, 4096), reconstructed image blocks
%


    [N1, N2, blocks] = size(dct_blocks);
    image_out = zeros(size(dct_blocks));
    % tic;

    for block = 1:blocks
        dct_img = dct_blocks(:,:,block);
        for n1 = 1:N1
            for n2 = 1:N2
                k1 = 0:N1-1;
                k2 = 0:N2-1;
                w1 = [0.5, ones(1, N1-1)];
                w2 = [0.5, ones(1, N2-1)];
                prod = w1' .* w2 .* dct_img .* cos(pi*k1*(2* n1 - 1)/(2*N1)).' .*
                 ↪  cos(pi*k2*(2 * n2 - 1)/(2*N2));

                image_out(n1, n2, block) = sum(prod, "all") / (N1*N2);
            end
        end
    end
    % toc;
end
```

Listing 6 Recombine Segments to form image

```
function [image] = combineSegments(segments)
% combineSegments Combine all individual image blocks into one image
%   segments        (8,8,4096) input image blocks
%   image           (512, 512) output image

    image = zeros(512, 512);
    for i = 1:64
        h = (i-1)*64;
        for j = 1:64
            image((i-1)*8 + 1: i*8, (j-1)*8 + 1: j*8) = segments(:,:,h+j);
        end
    end
end
```

Listing 7 PSNR Computation

```
function [out] = calcPSNR(img, ref)
%   calcPSNR    computes the PSNR of the reconstructed image
%
%   img         size: (512, 512), reconstructed image
%   ref         size: (512, 512), reference original image
%
%   out         calculated PSNR value
%



    diff = ref - img;
    mse = mean(diff.^2, "all");
    peak = 255^2;
    out = 10 * log10(peak/mse);
end
```