UNIVERSITY OF SCIENCE AND TECHNOLOGY

Information Theory and Coding (CIE 425)

Project Phase 1: Image Compression Using JPEG

Mario Youchia
201902086

Yahya Mohamed
202000776

Dr. Mahmoud Abdelaziz

Academic Year
2023-2024

Table of Contents

Abstract:

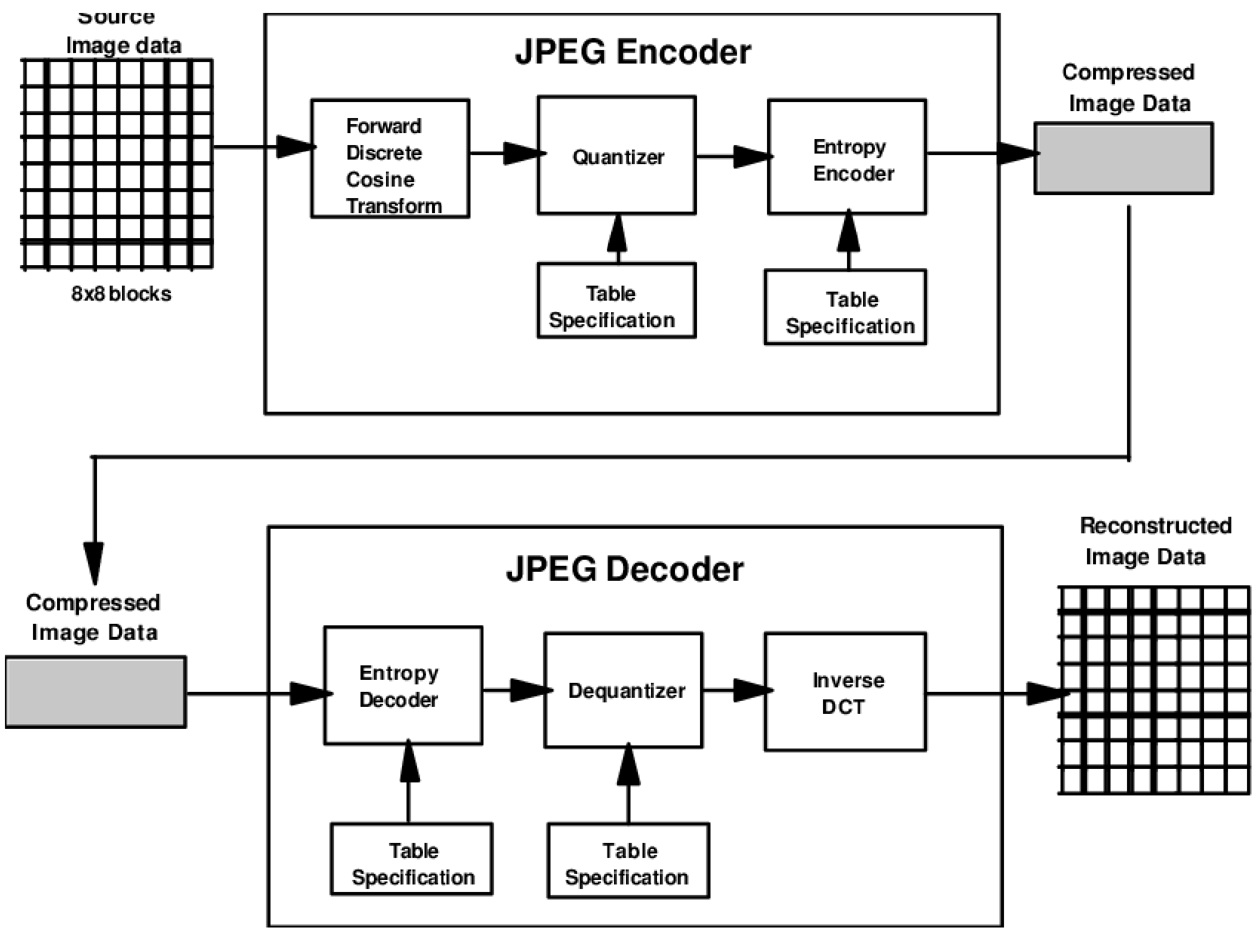
In this phase, we created a JPEG encoder and decoder for grayscale images using MATLAB.

Introduction:

As the number of images transmitted or received per second increases, there is an increasing demand to reduce the size of the image through compression. It is also important to be able to decompress the compressed image and extract information from it. This is exactly what the JPEG encoder and decoder do, as shown in the figure below.

Firstly, the image is divided into 8x8 blocks. These blocks are transformed to the frequency domain using DCT. The result is then quantized using a specific quantization table. The compressed image is obtained after encoding the output of the quantization.

On the other hand, the JPEG decoder performs the reverse of the JPEG encoder's process. It begins by decoding the compressed image, followed by dequantizing it using the same quantization table as used in the encoding part. Finally, the reconstructed image is obtained after transforming the frequency domain back to the spatial domain using the IDCT.



The following parts show the steps of implementing the JPEG encoder and decoder in MATLAB. Each step includes the corresponding output, relevant variables, and a brief description of the code used in this part.

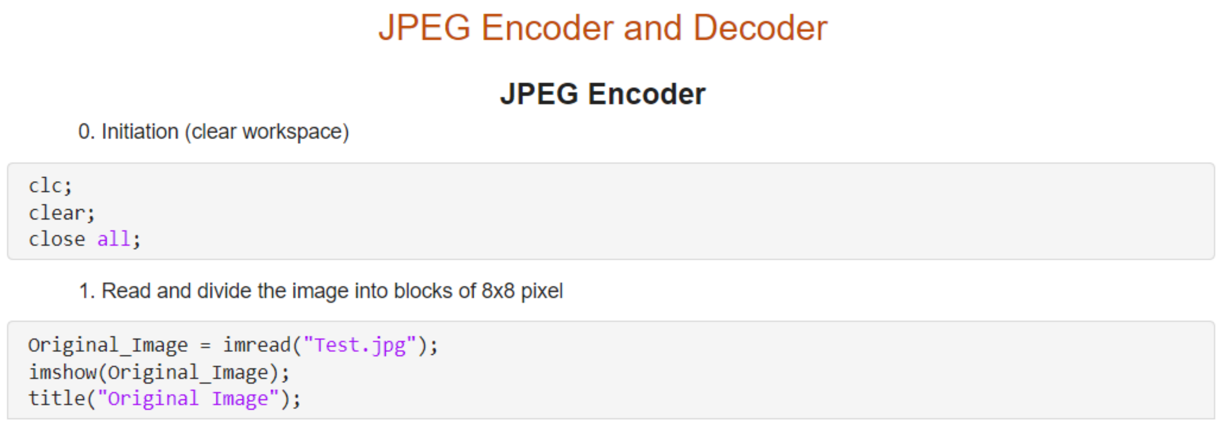
Part 1: Divide image into 8x8 blocks

The code shown below represents the function that divides the image into 8x8 blocks. It takes a 2D array representing the image as input and returns a 2D cell array. Each cell contains an 8x8 block. If the number of rows or columns is not divisible by 8, the image is cropped. The two most important variables in this function are "image" (representing the input image) and "image_blocks" (containing the 8x8 blocks).

```matlab
function image_blocks = divideImageIntoBlocks(image, square)
    num_of_rows_per_block = 8;
    num_of_cols_per_block = 8;
    [num_of_rows, num_of_cols] = size(image);
    switch square
        case 1 % image > square 2D array
            if (num_of_rows < num_of_cols)
                last_dimension = num_of_rows - ...
                    rem(num_of_rows, num_of_rows_per_block);
            else
                last_dimension = num_of_cols - ...
                    rem(num_of_cols, num_of_cols_per_block);
            end
            image = image(1:last_dimension, 1:last_dimension);
        case 0
            last_dimension_rows = num_of_rows - ...
                rem(num_of_rows, num_of_rows_per_block);
            last_dimension_cols = num_of_cols - ...
                rem(num_of_cols, num_of_cols_per_block);
            image = image(1:last_dimension_rows, 1:last_dimension_cols);
        otherwise
            return;
    end

    [num_of_rows, num_of_cols] = size(image);

    % num_of_blocks_rows, and num_of_blocks_cols represent the new number
    % of rows and columns of the image matrix after converting it to blocks
    % of 8x8 elements
    num_of_blocks_rows = num_of_rows / num_of_rows_per_block;
    num_of_blocks_cols = num_of_cols / num_of_cols_per_block;

    % Define an array container where each element represent a block of the
    % image
    image_blocks = cell(num_of_blocks_rows, num_of_blocks_cols);
    for i = 1:num_of_blocks_rows
        for j = 1:num_of_blocks_cols
            rows_range = ...
                (i * num_of_rows_per_block) - (num_of_rows_per_block - 1) : ...
                i * num_of_rows_per_block;
            cols_range = ...
```

```
                (j * num_of_cols_per_block) - (num_of_cols_per_block - 1) : ...
                j * num_of_cols_per_block;
            image_blocks{i, j} = image(rows_range, cols_range);
        end
    end
end
```

The figure below shows the start of the program. It starts by clearing the previous variables in the workspace. Then, it reads the image and displays it before performing any changes to the image.



JPEG Encoder and Decoder

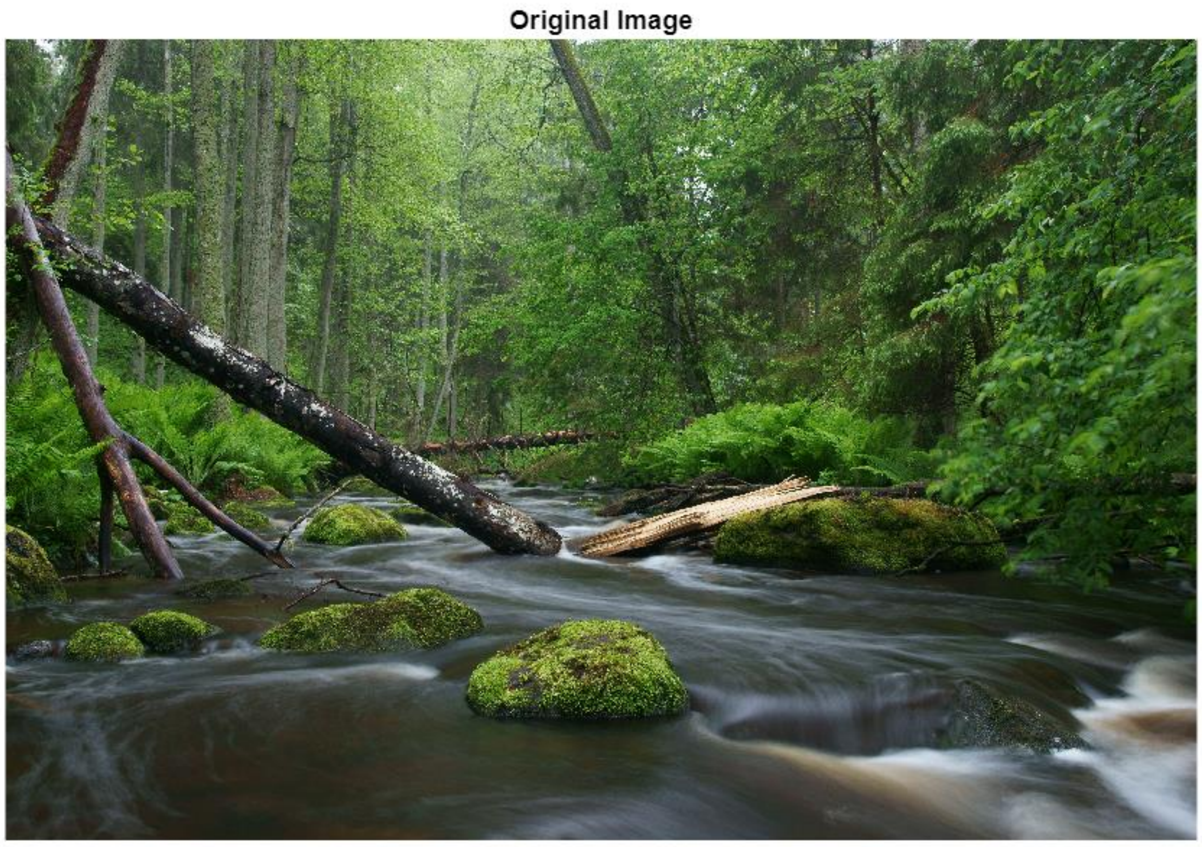
JPEG Encoder

0. Initiation (clear workspace)

```
clc;
clear;
close all;
```

1. Read and divide the image into blocks of 8x8 pixel

```
Original_Image = imread("Test.jpg");
imshow(Original_Image);
title("Original Image");
```

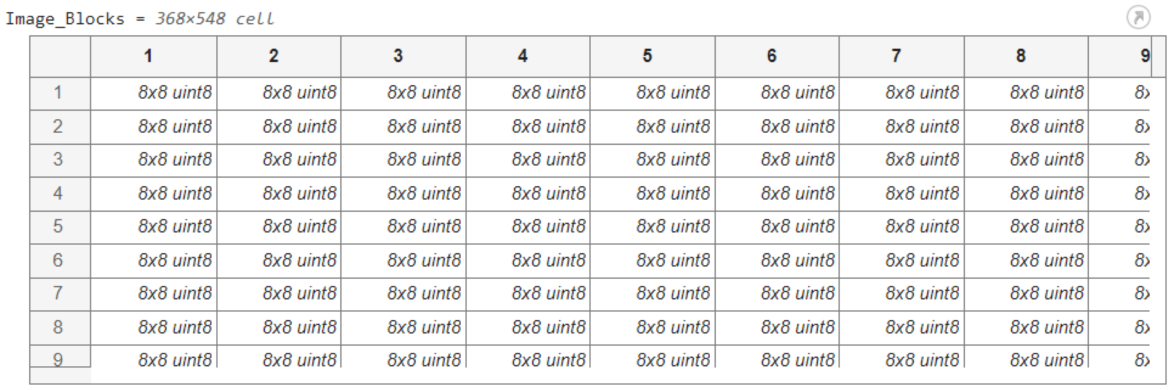


The image is then converted to grayscale, as shown in the following figure on the next page:

```
Gray_Image = rgb2gray(Original_Image);
imshow(Gray_Image)
title("Grayscale Image");
```

The final step in this section is dividing the image into 8x8 blocks, as shown in the following image:

```
% Divide the image into block of 8x8 pixels
Image_Blocks = divideImageIntoBlocks(Gray_Image, 0)
```



```
Image_Blocks = 368×548 cell
```

	1	2	3	4	5	6	7	8	9
1	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8
2	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8
3	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8
4	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8
5	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8
6	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8
7	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8
8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8
9	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8x8 uint8	8

Part 2: Perform DCT on Each Block

The function provided calculates the Discrete Cosine Transform (DCT) basis functions using the formula:

$$b(x, y) = \cos\left(\frac{(2x+1)u\pi}{16}\right)\cos\left(\frac{(2y+1)v\pi}{16}\right)$$

Then, it employs this transformation on the blocks to derive the DCT coefficients. Each DCT coefficient in the DCT table is scaled based on the following criteria:

- The DCT coefficient at $(u = 0, v = 0)$ is divided by 64.
- DCT coefficients in the first row and the first column (excluding the DC component) are divided by 32.
- DCT coefficients in the rest of the table are divided by 16.

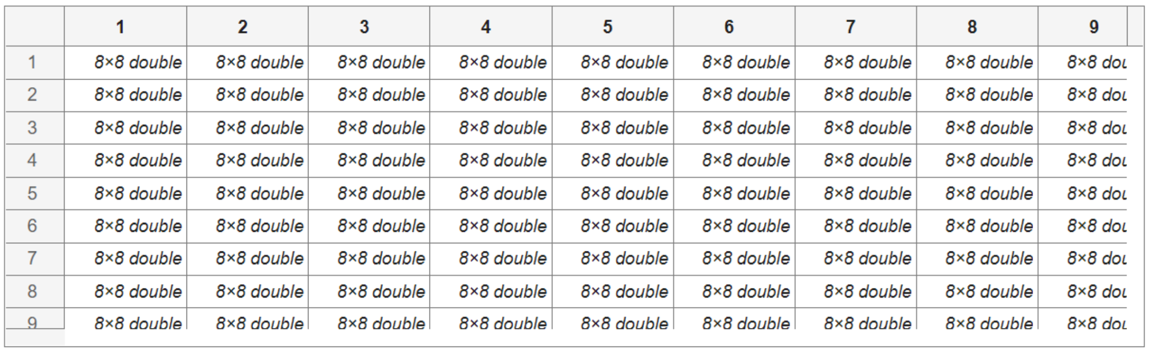
```
function DCT_OUT = DCT(Blocks)
    DCT_OUT = cell(size(Blocks));
    for i = 1:size(Blocks, 1)
        for j = 1:size(Blocks, 2)
            Block = Blocks{i, j};
            Block = double(Block);
            [num_of_rows_per_block, num_of_cols_per_block] = size(Block);
            new_block = zeros(size(Block));
            for u = 0:num_of_rows_per_block-1 % 0 > 7
                for v = 0:num_of_cols_per_block-1 % 0 > 7
                    sum = 0;
                    for x = 0:num_of_rows_per_block-1 % 0 > 7
                        for y = 0:num_of_cols_per_block-1 % 0 > 7
                            sum = sum + Block(x+1, y+1) * ...
                            cos(((2*x+1)*u*pi)/16) * cos(((2*y+1)*v*pi)/16);
                        end
                    end
                    if u == 0 && v == 0
                        new_block(u+1, v+1) = sum / 64;
                    elseif u == 0 || v == 0
                        new_block(u+1, v+1) = sum / 32;
                    else
                        new_block(u+1, v+1) = sum / 16;
                    end
                end
            end
            DCT_OUT{i, j} = new_block;
        end
    end
end
```

The figure below shows the output format generated by the Discrete Cosine Transform (DCT) function.
The function takes the image blocks, then it produces a set of 2D cells. Each cell contains an 8×8 DCT table.

2. Perform DCT on each block

```
image_in_freq_domain = DCT(Image_Blocks)
```

```
image_in_freq_domain = 368×548 cell
```



	1	2	3	4	5	6	7	8	9
1	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
2	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
3	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
4	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
5	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
6	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
7	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
8	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
9	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou

To validate the DCT function, a single 8 × 8 block is entered to the function. Then, the obtained output is entered into the Inverse Discrete Cosine Transform (IDCT) function, *implemented in part 11*. The outcome of the IDCT function is then compared to the original 8 × 8 block through two methods. Firstly, the Root Mean Square Error (RMSE) is calculated. Secondly, the values of the two blocks are displayed and manually compared for validation.

In the initial approach, the Root Mean Square Error (RMSE) has given a very small value, indicating the very close similarity between the two blocks:

```
% Validation
test_block = {Image_Blocks{1,1}};
test_block{1}
```

```
ans = 8×8 uint8 matrix
    95   85   80   67   55   59   64   68
    73   67   72   67   53   49   54   63
    67   57   62   63   51   44   47   59
    70   50   48   52   50   47   47   52
    65   44   38   42   47   51   48   44
    75   55   46   43   44   51   52   50
    88   69   59   50   44   49   55   59
    83   67   62   56   47   48   52   58
```

```
reconstructed_block = IDCT(DCT(test_block));
reconstructed_block{1};
% Calculate the RMSE
RMSE = sqrt(sum(sum((double(test_block{1}) - reconstructed_block{1}).^2))/numel(test_block))
```

```
RMSE = 2.7738e-13
```

The second method, shown in the figure on the following page, involves a manual comparison of values. This comparison ensures that the reconstructed block is identical to the input block.

```
Validation = horzcat( ...
    array2table(test_block{1}, 'VariableNames', compose('Original%d', 1:8)), ...
    array2table(reconstructed_block{1}, 'VariableNames', compose('Reconstructed%d', 1:8)))
```

Validation = *8×16 table*

	Original1	Original2	Original3	Original4	Original5	Original6	Origina
1	95	85	80	67	55	59	
2	73	67	72	67	53	49	
3	67	57	62	63	51	44	
4	70	50	48	52	50	47	
5	65	44	38	42	47	51	
6	75	55	46	43	44	51	
7	88	69	59	50	44	49	
8	83	67	62	56	47	48	

```
% The validation using a single 8x8 block is to ensure that the output
% after performing DCT and IDCT restores the original pixel values.
```

Hence, we can ensure the proper functionality of both the DCT and IDCT functions.

Part 3: Perform Quantization Using At least Two Quantization Tables

The following code shows the quantization step applied to each block after the transformation from the spatial domain to the frequency domain. The quantization step involves an element-wise division between the block and the quantization table.

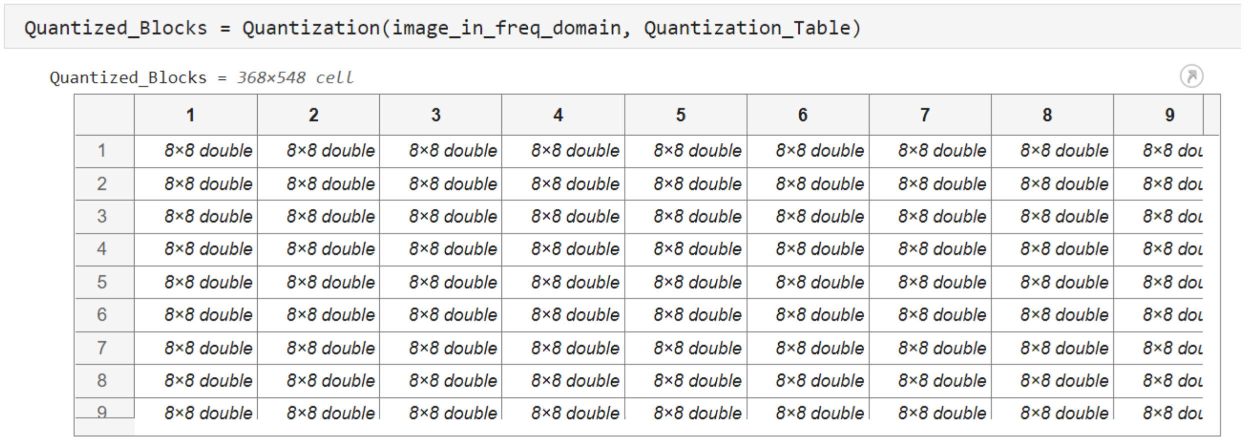
```
function Quant_OUT = Quantization(Blocks, Quant_Table)
    Quant_OUT = cell(size(Blocks));
    for i = 1:size(Blocks, 1)
        for j = 1:size(Blocks, 2)
            Block = Blocks{i, j};
            new_block = Block./Quant_Table;
            Quant_OUT{i, j} = round(new_block);
        end
    end
end
```

The dropdown menu shown in the figure on the right facilitates the selection between two different quantization tables. The first table is used for image compression with a lower loss ratio, while the second table is

```
Low_Compression_Quant_Table = [1, 1, 1, 1, 1, 2, 2, 4;
                               1, 1, 1, 1, 1, 2, 2, 4;
                               1, 1, 1, 1, 2, 2, 2, 4;
                               1, 1, 1, 1, 2, 2, 4, 8;
                               1, 1, 2, 2, 2, 2, 4, 8;
                               2, 2, 2, 2, 2, 4, 8, 8;
                               2, 2, 2, 4, 4, 8, 8, 16;
                               4, 4, 4, 4, 8, 8, 16, 16];
High_Compression_Quant_Table = [1, 2, 4, 8, 16, 32, 64, 128;
                                2, 4, 4, 8, 16, 32, 64, 128;
                                4, 4, 8, 16, 32, 64, 128, 128;
                                8, 8, 16, 32, 64, 128, 128, 256;
                                16, 16, 32, 64, 128, 128, 256, 256;
                                32, 32, 64, 128, 128, 256, 256, 256;
                                64, 64, 128, 128, 256, 256, 256, 256;
                                128, 128, 128, 256, 256, 256, 256, 256];
Quantization_Table = Low Compression
```

used to achieve higher compression, consequently resulting in a greater loss ratio compared to the first quantization table.

The figure below shows the format of the output generated by the quantization function. It is still a set of 2D cells, each containing a 2D block of 8 × 8 values. These values are those from the DCT output, yet they undergo individual element-wise division by the quantization table and subsequent rounding.



```
Quantized_Blocks = Quantization(image_in_freq_domain, Quantization_Table)
```

```
Quantized_Blocks = 368×548 cell
```

	1	2	3	4	5	6	7	8	9
1	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
2	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
3	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
4	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
5	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
6	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
7	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
8	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
9	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou

Part 4: Transform Each 2D Block Into 1D Vector

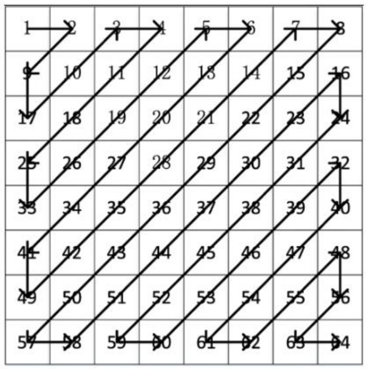
The provided code illustrates the transformation from 2D arrays to 1D vectors using a zigzag reading pattern. This pattern helps group more zeros together, allowing for greater image compression. The actions implemented in this function are inspired by the image displayed following the code.

```
function TwoD_To_1D = From_2D_To_1D(Blocks)
    TwoD_To_1D = cell(size(Blocks));
    for i = 1:size(Blocks, 1)
        for j = 1:size(Blocks, 2)
            Block = Blocks{i, j};
            [num_of_rows, num_of_cols] = size(Block);
            new_block = zeros(1, num_of_rows * num_of_cols);
            % Initial Position (row = 1, col = 1)
            curr_row = 1;
            curr_col = 1;
            % Initialize an upward flag to record the current direction
            upward = true;

            for elem_num = 1:num_of_rows*num_of_cols
                % Copy the current element to the 1D vector
                new_block(elem_num) = Block(curr_row, curr_col);
                % Upward Case
                if upward
                    % if the current element is at the last column, the next action
                    % should be move down, and then go diagonally downward
                    if curr_col == num_of_cols
                        curr_row = curr_row + 1;
                        upward = false;
                    % if the current element is at the first row, the next action
                    % should be move to the right, and then go diagonally downward
                    elseif curr_row == 1
                        curr_col = curr_col + 1;
                        upward = false;
                    % if the current element is not at the first row or at the last
                    % column, then move diagonally upward
                    else
                        curr_row = curr_row - 1;
                        curr_col = curr_col + 1;
                    end

                % Downward Case
                else
                    % if the current element is at the last row, the next action
                    % should be move right, and then go diagonally upward
                    if curr_row == num_of_rows
                        curr_col = curr_col + 1;
                        upward = true;
                    % if the current element is at the first column, the next action
                    % should be move down, and then go diagonally upward
                    elseif curr_col == 1
                        curr_row = curr_row + 1;
                        upward = true;
                    % if the current element is not at the first column or at the
last
                    % row, then move diagonally downward
                    else
                        curr_row = curr_row + 1;
                        curr_col = curr_col - 1;
                    end
                end
            end
        TwoD_To_1D{i, j} = new_block;
        end
    end
end
```

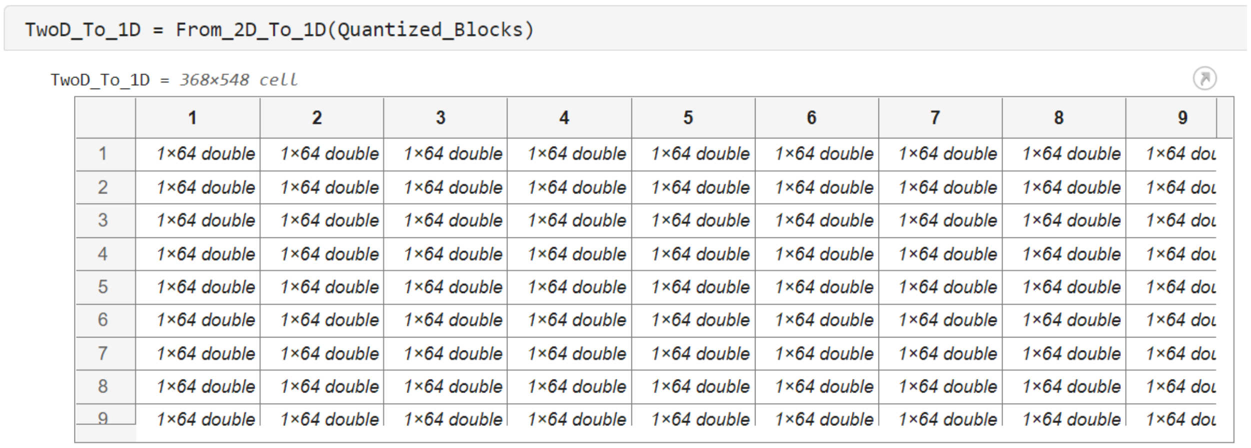
The image from which the implemented actions are inspired:

4. Transform each 2D block into 1D vector

This part and part No. 9 is inspired from the following figure:



Here is the output format resulting from this transformation, with each cell now containing a 1×64 vector:



```
TwoD_To_1D = From_2D_To_1D(Quantized_Blocks)
```

```
TwoD_To_1D = 368×548 cell
```

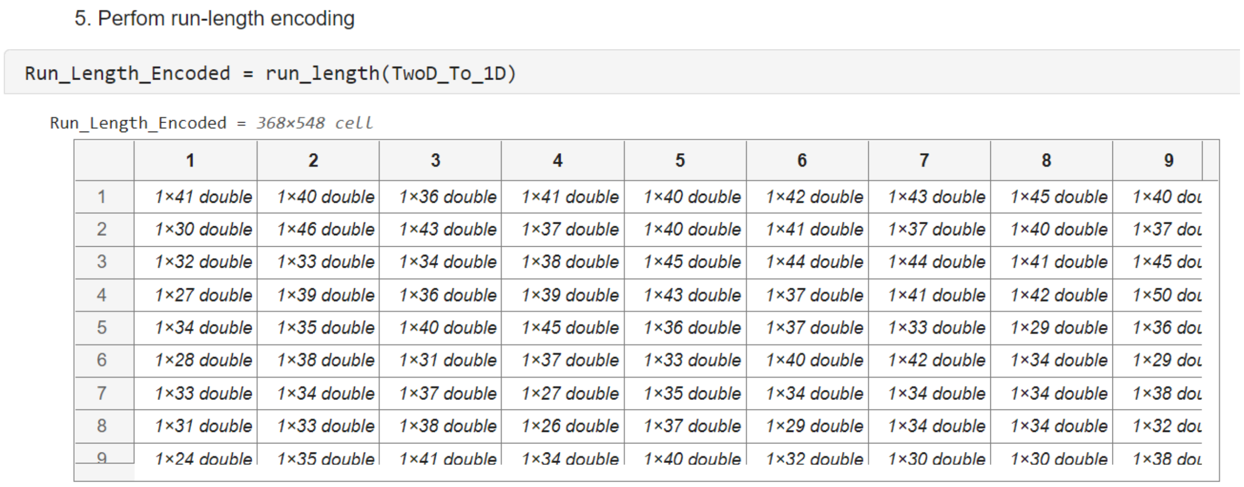
	1	2	3	4	5	6	7	8	9
1	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 dou
2	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 dou
3	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 dou
4	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 dou
5	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 dou
6	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 dou
7	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 dou
8	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 dou
9	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 dou

Part 5: Perform Run-Length Encoding

Here is the implementation of run-length encoding, an algorithm used to make each vector smaller by grouping together neighboring zeros.

```
function Run_Length_OUT = run_length(Blocks)
    Run_Length_OUT = cell(size(Blocks));
    for i = 1:size(Blocks, 1)
        for j = 1:size(Blocks, 2)
            line = Blocks{i, j};
            % Initialize the encoded line
            encoded_line = [];
            % For every element in the original array
            for elem = 1:length(line)
                % If the element is not 0, append it to the encoded line
                if line(elem) ~= 0
                    encoded_line = [encoded_line, line(elem)];
                else
                    % If it is 0, check if it's the first zero or not
                    if length(encoded_line) > 2 && encoded_line(end-1) == 0
                        % If it is not the first zero, increment the last element of
the encoded line by 1
                        encoded_line(end) = encoded_line(end) + 1;
                    else
                        % If it is the first zero, append 0 and 1
                        encoded_line = [encoded_line, 0, 1];
                    end
                end
            end
            Run_Length_OUT{i, j} = encoded_line;
        end
    end
end
```

The image below displays the output format after applying run-length encoding. As observed, each cell now holds a 1D vector with fewer or equal to 64 elements.



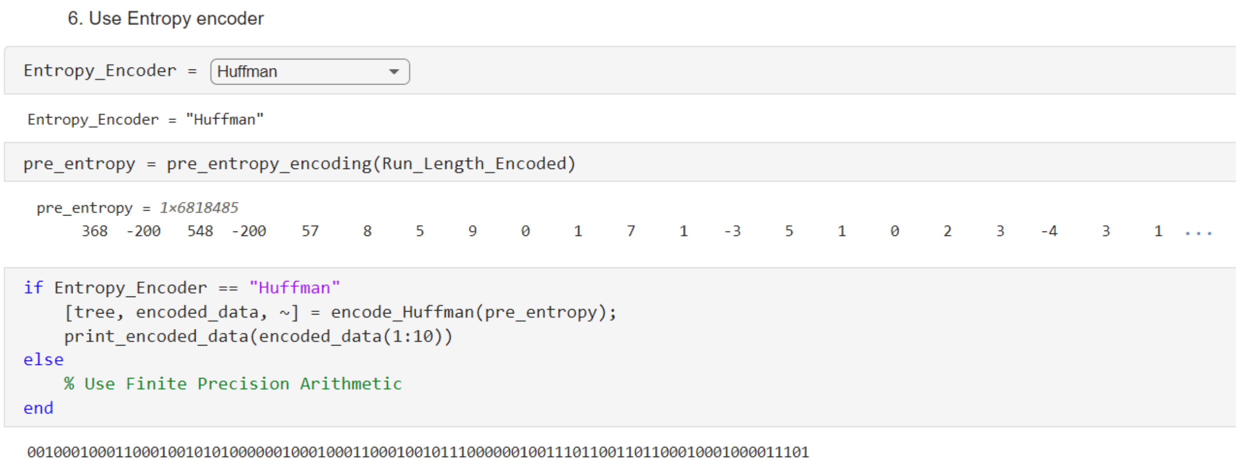
5. Perfom run-length encoding

```
Run_Length_Encoded = run_length(TwoD_To_1D)
```

```
Run_Length_Encoded = 368×548 cell
```

	1	2	3	4	5	6	7	8	9
1	1×41 double	1×40 double	1×36 double	1×41 double	1×40 double	1×42 double	1×43 double	1×45 double	1×40 dou
2	1×30 double	1×46 double	1×43 double	1×37 double	1×40 double	1×41 double	1×37 double	1×40 double	1×37 dou
3	1×32 double	1×33 double	1×34 double	1×38 double	1×45 double	1×44 double	1×44 double	1×41 double	1×45 dou
4	1×27 double	1×39 double	1×36 double	1×39 double	1×43 double	1×37 double	1×41 double	1×42 double	1×50 dou
5	1×34 double	1×35 double	1×40 double	1×45 double	1×36 double	1×37 double	1×33 double	1×29 double	1×36 dou
6	1×28 double	1×38 double	1×31 double	1×37 double	1×33 double	1×40 double	1×42 double	1×34 double	1×29 dou
7	1×33 double	1×34 double	1×37 double	1×27 double	1×35 double	1×34 double	1×34 double	1×34 double	1×38 dou
8	1×31 double	1×33 double	1×38 double	1×26 double	1×37 double	1×29 double	1×34 double	1×34 double	1×32 dou
9	1×24 double	1×35 double	1×41 double	1×34 double	1×40 double	1×32 double	1×30 double	1×30 double	1×38 dou

Part 6: Use Huffman Encoder



6. Use Entropy encoder

```
Entropy_Encoder = Huffman
```

```
Entropy_Encoder = "Huffman"
```

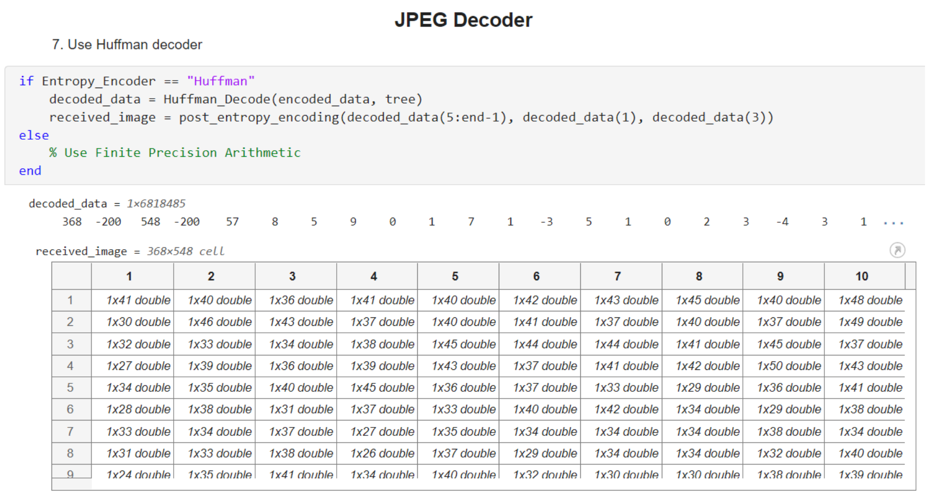
```
pre_entropy = pre_entropy_encoding(Run_Length_Encoded)
```

```
pre_entropy = 1×6818485
   368  -200   548  -200    57     8     5     9     0     1     7     1    -3     5     1     0     2     3    -4     3     1 ...
```

```
if Entropy_Encoder == "Huffman"
    [tree, encoded_data, ~] = encode_Huffman(pre_entropy);
    print_encoded_data(encoded_data(1:10))
else
    % Use Finite Precision Arithmetic
end
```

```
00100010001100010010101000000100010001100010010111000000100111011001101100010001000011101
```

Part 7: Use Huffman Decoder



JPEG Decoder

7. Use Huffman decoder

```
if Entropy_Encoder == "Huffman"
    decoded_data = Huffman_Decode(encoded_data, tree)
    received_image = post_entropy_encoding(decoded_data(5:end-1), decoded_data(1), decoded_data(3))
else
    % Use Finite Precision Arithmetic
end
```

```
decoded_data = 1×6818485
   368  -200   548  -200    57     8     5     9     0     1     7     1    -3     5     1     0     2     3    -4     3     1 ...
```

```
received_image = 368×548 cell
```

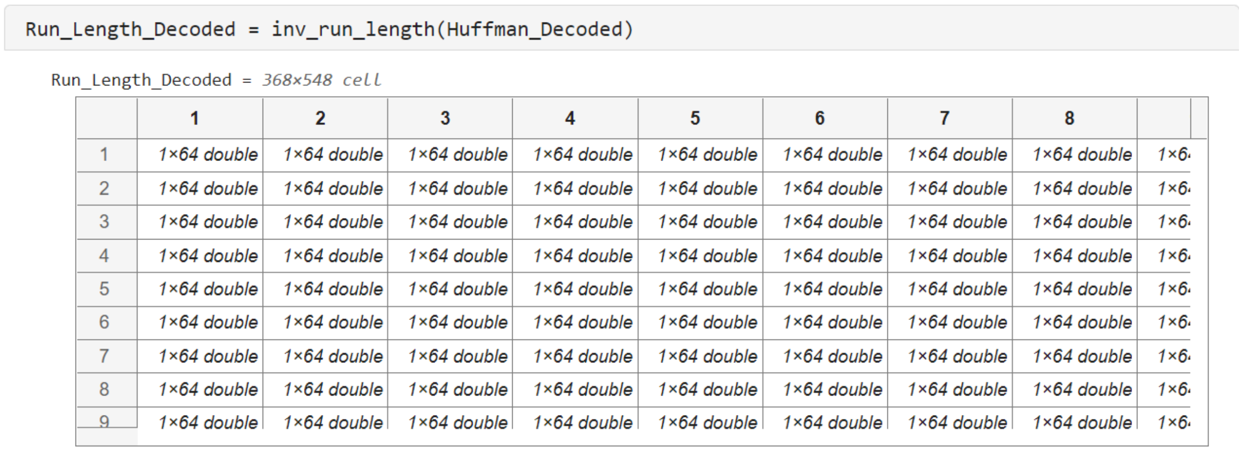
	1	2	3	4	5	6	7	8	9	10
1	1x41 double	1x40 double	1x36 double	1x41 double	1x40 double	1x42 double	1x43 double	1x45 double	1x40 double	1x48 double
2	1x30 double	1x46 double	1x43 double	1x37 double	1x40 double	1x41 double	1x37 double	1x40 double	1x37 double	1x49 double
3	1x32 double	1x33 double	1x34 double	1x38 double	1x45 double	1x44 double	1x44 double	1x41 double	1x45 double	1x37 double
4	1x27 double	1x39 double	1x36 double	1x39 double	1x43 double	1x37 double	1x41 double	1x42 double	1x50 double	1x43 double
5	1x34 double	1x35 double	1x40 double	1x45 double	1x36 double	1x37 double	1x33 double	1x29 double	1x36 double	1x41 double
6	1x28 double	1x38 double	1x31 double	1x37 double	1x33 double	1x40 double	1x42 double	1x34 double	1x29 double	1x38 double
7	1x33 double	1x34 double	1x37 double	1x27 double	1x35 double	1x34 double	1x34 double	1x34 double	1x38 double	1x34 double
8	1x31 double	1x33 double	1x38 double	1x26 double	1x37 double	1x29 double	1x34 double	1x34 double	1x32 double	1x40 double
9	1x24 double	1x35 double	1x41 double	1x34 double	1x40 double	1x32 double	1x30 double	1x30 double	1x38 double	1x39 double

Part 8: Perform Run-Length Decoding

Here is the implementation of run-length decoding.

```
function Run_Length_OUT = inv_run_length(Blocks)
    Run_Length_OUT = cell(size(Blocks));
    for i = 1:size(Blocks, 1)
        for j = 1:size(Blocks, 2)
            % Perform the run length decoding on a line (array) of numbers
            encoded_line = Blocks{i, j};
            % Initialize the decoded line
            decoded_line = [];

            % For every element in the encoded array
            for elem = 1:length(encoded_line)
                % If the previous element is zero, skip as this number represents the 
repeated values of zero
                if elem > 1 && encoded_line(elem-1) == 0
                    continue;
                end

                % If the element is not 0, append it to the original line
                if encoded_line(elem) ~= 0
                    decoded_line = [decoded_line, encoded_line(elem)];
                else
                    % If it is 0, repeat the 0 next element of the encoded line times 
and append all of these 0s to the original line
                    decoded_line = [decoded_line, zeros(1, encoded_line(elem+1))];
                end
            end
            Run_Length_OUT{i, j} = decoded_line;
        end
    end
end
```

The image below shows the result of the run-length decoding algorithm. The output consists of 2D cells, with each cell containing a 1D vector containing 64 elements.



```
Run_Length_Decoded = inv_run_length(Huffman_Decoded)
```

```
Run_Length_Decoded = 368×548 cell
```

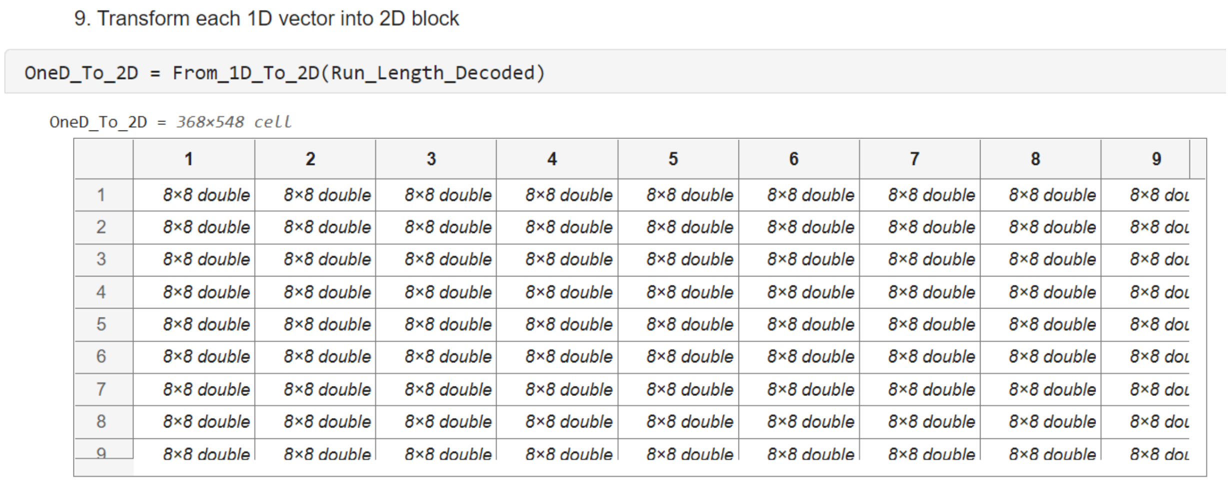
	1	2	3	4	5	6	7	8	
1	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×6
2	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×6
3	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×6
4	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×6
5	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×6
6	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×6
7	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×6
8	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×6
9	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×64 double	1×6

Part 9: Transform Each 1D Vector Into 2D Block

The provided code illustrates the transformation from 1D vectors to 2D arrays using a zigzag reading pattern.

```
function OneD_To_2D = From_1D_To_2D(Blocks)
    OneD_To_2D = cell(size(Blocks));
    for i = 1:size(Blocks, 1)
        for j = 1:size(Blocks, 2)
            Block = Blocks{i, j};
            num_of_rows = sqrt(length(Block));
            num_of_cols = sqrt(length(Block));
            new_block = zeros(num_of_rows, num_of_cols);
            % Initial Position (row = 1, col = 1)
            curr_row = 1;
            curr_col = 1;
            % Initialize an upward flag to record the current direction
            upward = true;

            for elem_num = 1:num_of_rows*num_of_cols
                % Copy the current element from the 1D vector to the 2D block
                new_block(curr_row, curr_col) = Block(elem_num);
                % Upward Case
                if upward
                    % if the current element is at the last column, the next action
                    % should be move down, and then go diagonally downward
                    if curr_col == num_of_cols
                        curr_row = curr_row + 1;
                        upward = false;
                    % if the current element is at the first row, the next action
                    % should be move to the right, and then go diagonally downward
                    elseif curr_row == 1
                        curr_col = curr_col + 1;
                        upward = false;
                    % if the current element is not at the first row or at the last
                    % column, then move diagonally upward
                    else
                        curr_row = curr_row - 1;
                        curr_col = curr_col + 1;
                    end

                % Downward Case
                else
                    % if the current element is at the last row, the next action
                    % should be move right, and then go diagonally upward
                    if curr_row == num_of_rows
                        curr_col = curr_col + 1;
                        upward = true;
                    % if the current element is at the first column, the next action
                    % should be move down, and then go diagonally upward
                    elseif curr_col == 1
                        curr_row = curr_row + 1;
                        upward = true;
                    % if the current element is not at the first column or at the
last
                    % row, then move diagonally downward
                    else
                        curr_row = curr_row + 1;
                        curr_col = curr_col - 1;
                    end
                end
            end
        OneD_To_2D{i, j} = new_block;
        end
    end
end
```

Here is the output format resulting from this transformation, with each cell now containing a 8×8 array:



9. Transform each 1D vector into 2D block

```
OneD_To_2D = From_1D_To_2D(Run_Length_Decoded)
```

```
OneD_To_2D = 368×548 cell
```

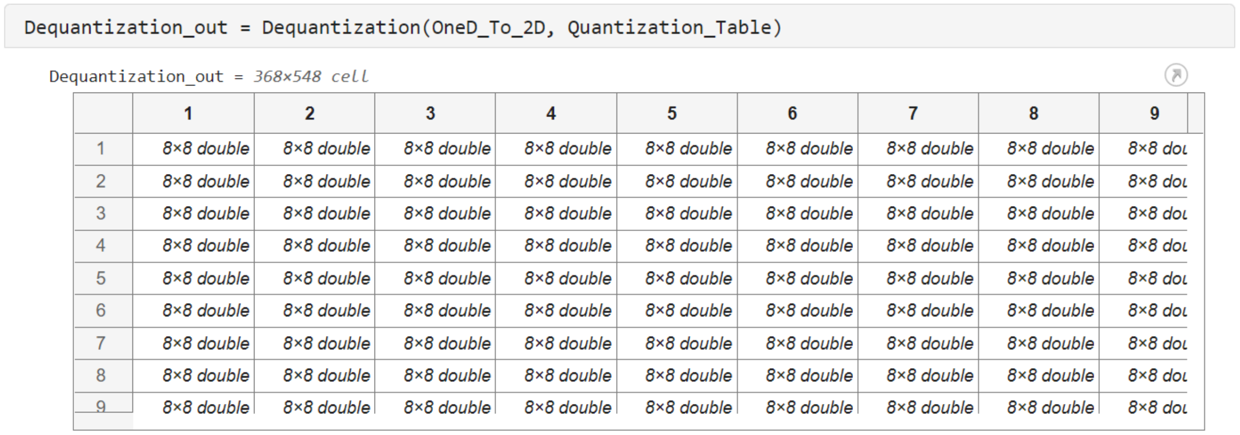
	1	2	3	4	5	6	7	8	9
1	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
2	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
3	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
4	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
5	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
6	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
7	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
8	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
9	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou

Part 10: Perform Dequantization

The following code shows the dequantization step applied to each block after the transformation from the 1D vectors to the 2D blocks. The dequantization step involves an element-wise multiplication between the block and the same quantization table used in part 3.

```
function Dequant_OUT = Dequantization(Blocks, Quant_Table)
    Dequant_OUT = cell(size(Blocks));
    for i = 1:size(Blocks, 1)
        for j = 1:size(Blocks, 2)
            Block = Blocks{i, j};
            new_block = Block.*Quant_Table;
            Dequant_OUT{i, j} = new_block;
        end
    end
end
```

The figure below shows the format of the output generated by the dequantization function. It is still a set of 2D cells, each containing a 2D block of 8 × 8 values. These values are those from the



```
Dequantization_out = Dequantization(OneD_To_2D, Quantization_Table)
```

```
Dequantization_out = 368×548 cell
```

	1	2	3	4	5	6	7	8	9
1	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
2	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
3	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
4	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
5	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
6	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
7	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
8	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou
9	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 double	8×8 dou

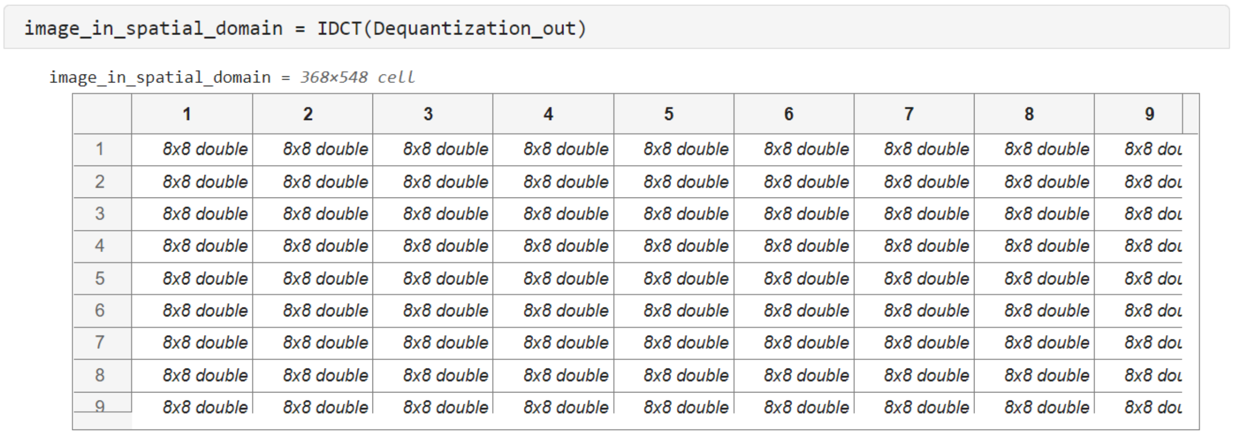
1D to 2D transformation output, yet they undergo individual element-wise multiplication by the quantization table used in part 3 and subsequent rounding.

Part 11: Perform IDCT on Each Block

The following function illustrates the implementation of the IDCT. It just multiplies each DCT coefficient by its corresponding DCT basis function.

```
function IDCT_OUT = IDCT(Blocks)
    IDCT_OUT = cell(size(Blocks));
    for i = 1:size(Blocks, 1)
        for j = 1:size(Blocks, 2)
            Block = Blocks{i, j};
            [num_of_rows_per_block, num_of_cols_per_block] = size(Block);
            new_block = zeros(size(Block));
            for x = 0:num_of_rows_per_block-1
                for y = 0:num_of_cols_per_block-1
                    sum = 0;
                    for u = 0:num_of_rows_per_block-1
                        for v = 0:num_of_cols_per_block-1
                            sum = sum + Block(u+1, v+1) * ...
                            cos(((2*x+1)*u*pi)/16) * cos(((2*y+1)*v*pi)/16);
                        end
                    end
                    new_block(x+1, y+1) = sum;
                end
            end
            IDCT_OUT{i, j} = new_block;
        end
    end
end
```

The figure below shows the output format from the IDCT. It consists of 2D cells, each containing a 2D array of 8×8 pixel values. The validation of this function, as well as the DCT function, is presented in part 2.



```
image_in_spatial_domain = IDCT(Dequantization_out)
```

image_in_spatial_domain = *368×548 cell*

	1	2	3	4	5	6	7	8	9
1	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 dou
2	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 dou
3	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 dou
4	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 dou
5	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 dou
6	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 dou
7	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 dou
8	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 dou
9	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 double	8x8 dou

Part 12: Combine and Save 8x8 Pixel Blocks

The provided code combines the received blocks, after IDCT processing, into a single image.

```
function newImage = combineBlocks(image_blocks)
    num_of_rows_per_block = 8;
    num_of_cols_per_block = 8;

    [num_of_rows, num_of_cols] = size(image_blocks);
    newImage = zeros(num_of_rows * num_of_rows_per_block, ...
        num_of_cols * num_of_cols_per_block);

    for i = 1:num_of_rows
        for j = 1:num_of_cols
            newImage((i - 1) * num_of_rows_per_block + 1 ...
                : i * num_of_rows_per_block, ...
                (j - 1) * num_of_cols_per_block + 1 ...
                : j * num_of_cols_per_block) = image_blocks{i, j};
        end
    end
    newImage = uint8(newImage);
end
```

The image on the right shows the result of combining blocks after the IDCT.

The line of code below is used to save this received image.

```
imwrite(Received_Image, ...
    'result.jpg','JPEG');
```

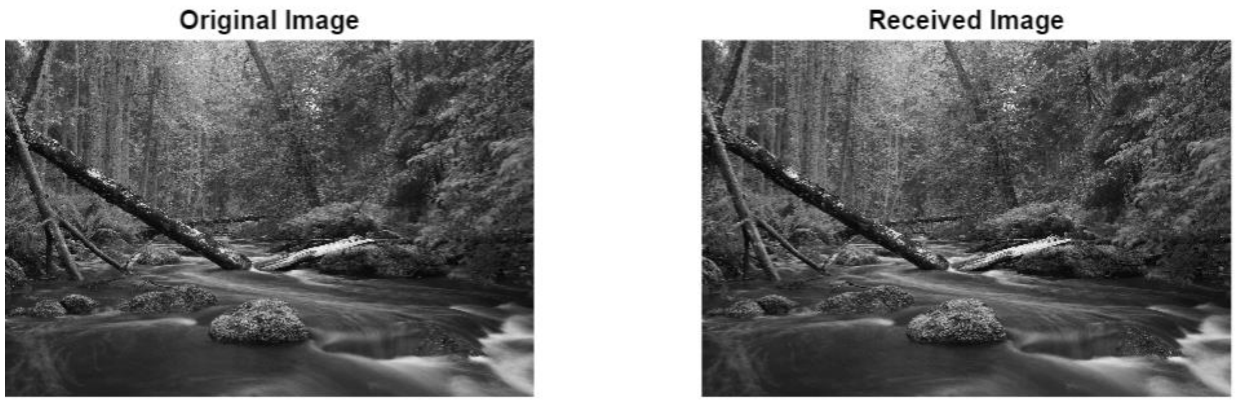
12. Combine the 8x8 pixel groups into a single image and save it back to a file

```
Received_Image = combineBlocks(IDCT_Output_Blocks);
imshow(Received_Image)
title("Received Image");
```

Part 13: Compare the Original Image with the Compressed Image When Using Each Quantization Table

- Using Low Quantization Table

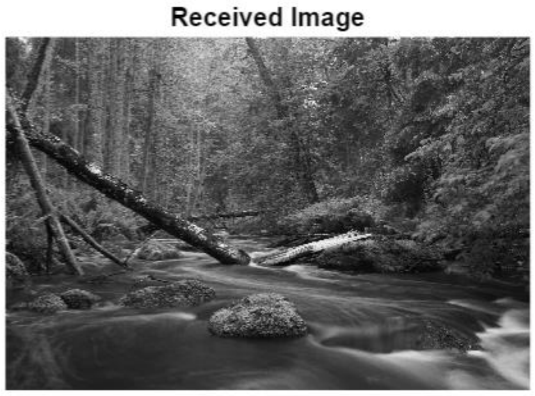
```
CompRatio = getCompRatio(encoded_data, size(Gray_Image, 1), size(Gray_Image, 2));
subplot(1,2,1)
imshow(Gray_Image)
title("Original Image");
subplot(1,2,2)
imshow(Received_Image)
title("Received Image");
```

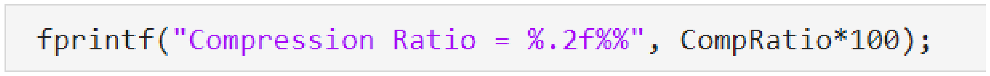


```
fprintf("Compression Ratio = %.2f%%", CompRatio*100);
```

```
Compression Ratio = 71.62%
```

- Using High Quantization Table:





```
fprintf("Compression Ratio = %.2f%%", CompRatio*100);
```

```
Compression Ratio = 88.92%
```

Comment:

By using a high quantization table, we can increase image compression, but at the cost of losing some information. This introduces a tradeoff between compression and quality. As the compression ratio increases, the quality of the resulting image decreases. In situations where images carry minimal information, sacrificing some quality to achieve further compression can be beneficial. Conversely, a low quantization table allows for significant image compression while saving a high percentage of the image quality.