Image Compression Project Report

Brandon W. Ardisson

April 29, 2024

1 Introduction

In this project I have utilized the 2-D Discrete Cosine Transform (DCT) to analyze, compress, an reconstruct a sample image, “lena.raw”. The DCT is similar to the Discrete Fourier Transform (DFT) except that the DCT is a real valued transform. For applications where phase relationship information provided by complex data sets is not required, the DCT finds its niche. The DCT is widely used in image and video compression (e.g., JPEG and MPEG) as it naturally tends to compress the useful spatial coefficients into a small number of frequency components.

2 Implementation

2.1 Read-in Original Image

Firstly, the original image is read in to the MATLAB environment using the following commands. The image is interpreted as 8 bit unsigned integer values, and then displayed with a grays cale color map.

```
clc; clear all; close all;


fid = fopen('lena.raw');
a = fread(fid,[512,512],'uchar');
fclose(fid);
a = a';
imagesc(a,[0,255]);
colormap(gray);
axis equal;
axis off;
title('Original Image: lena.raw')
```

Figure 1: "lena.raw" as digitized and imported into the MATLAB environment.

2.2 Block Processing

The two dimensional 512 x 512 array of spatial image gray scale data is then split into an array of 8 x 8 blocks (each being 64 x 64) for further processing. At the top level this is performed by executing the *"block_process"* function. The arguments refer to the image reference data, the number of blocks in one dimension, the percentage of coefficients to drop once the DCT is computed, and a flag indicating whether or not we want to perform the forward or inverse DCT operation.

```
b = block_process(a, block_count, perc_coeff_2_drop, inverse);
```

The complete *block_process* function is provided here below.

```
function [outdata] = block_process(indata, block_count, coeff2drop,
   inverse)

% BLOCK_PROCESS subdivides the input 2D matrix into block_count x
% block_count matricies. It then performs the DCT using a custom 2D DCT
% function titled "dct2custom". Finally this function drops higher
```

```
   order
% coefficients from each block, indicated as a percentage by coeff2drop
   .

[m,n] = size(indata); % m rows, n columns

%% do some simple error checking
if ~isequal(m,n)
    error('Input matrix is not square, i.e. m != n')
end

if mod(m,2) ~= 0
    error('Input matrix dimensionality is not power of 2')
end

%% perform block processing
block_size = m/block_count;
outdata = zeros(m,n);

for hblkcnt = 1:block_count
    for vblkcnt = 1:block_count

        h_idx_start = hblkcnt*block_size - (block_size-1);
        h_idx_stop = hblkcnt*block_size;
        v_idx_start = vblkcnt*block_size - (block_size-1);
        v_idx_stop = vblkcnt*block_size;

        if ~inverse
            outdata(h_idx_start:h_idx_stop, v_idx_start:v_idx_stop) =
               dct2custom(indata(h_idx_start:h_idx_stop, v_idx_start:
               v_idx_stop));
            outdata(h_idx_start:h_idx_stop, v_idx_start:v_idx_stop) =
```

```
            drop_coefficients(outdata(h_idx_start:h_idx_stop,
            v_idx_start:v_idx_stop), coeff2drop);

        else

            outdata(h_idx_start:h_idx_stop, v_idx_start:v_idx_stop) =
            idct2custom(indata(h_idx_start:h_idx_stop, v_idx_start:
            v_idx_stop));
        end

    end
end
end
```

2.3 Discrete Cosine Transform Implementation

The forward DCT was implemented using nested for-loop constructs to perform the discrete summations present in the DCT equation. While this is not the fasted method, it was simple to construct and debug. Since this work is just theoretical deep dive and not designed for performance, the for-loop approach was warranted. The MATLAB code implementation of both the forward and inverse DCT algorithms is provided below.

```
function [outdata] = dct2custom(data)
% Forward DCT 2-D

[M,N] = size(data); % m rows , n columns
outdata = zeros(M,N);

%% dct2

    for k1 = 1:M

        for k2 = 1:N
```

```
            total = 0;

            for n1 = 1:M
                for n2 = 1:N

                    dct_calc = data(n1,n2) * cos(pi * (2*(n1-1)+1) * (
                       k1-1)/(2*M)) * ...
                        cos(pi * (2*(n2-1)+1) * (k2-1)/(2*N));
                    total = total + dct_calc;
                end
            end

            if k1 == 1
                coeff1 = 1/sqrt(M);
            else
                coeff1 = sqrt(2/M);
            end

            if k2 == 1
                coeff2 = 1/sqrt(N);
            else
                coeff2 = sqrt(2/N);
            end

            outdata(k1, k2) = coeff1 * coeff2 * total;

        end
    end
end

function [outdata] = idct2custom(data)
% IDCT 2-D
```

```
[M,N] = size(data); % m rows , n columns
outdata = zeros(M,N);

for n1 = 1:M

    for n2 = 1:N

        total = 0;

        for k1 = 1:M
            for k2 = 1:N

                if k1 == 1
                    coeff1 = 1/sqrt(M);
                else
                    coeff1 = sqrt(2/M);
                end

                if k2 == 1
                    coeff2 = 1/sqrt(N);
                else
                    coeff2 = sqrt(2/N);
                end

                dct_calc = coeff1 * coeff2 * data(k1,k2) * cos(pi *
                   (2*(n1-1)+1) * (k1-1)/(2*M)) * ...
                    cos(pi * (2*(n2-1)+1) * (k2-1)/(2*N));
                total = total + dct_calc;


            end
```

```
        end

        outdata(n1, n2) = total;

    end
end
end
```

2.4 Removal of High Frequency Coefficients

As can be seen in the *block_process* function, after computing the forward DCT, the method then applies the *drop_coefficients* function to the frequency domain data set. This function's implementation is provided below.

```
function [outdata] = drop_coefficients(indata, thresh)

[m,n] = size(indata);

if thresh == 0.9
    k = ceil(0.54 * m);

elseif thresh == 0.95
    k = ceil(0.68 * m);

elseif thresh == 0.75
    k = ceil(0.29 * m);

elseif thresh == 0.5
    k = 0;
end

% fprintf('Extracting %dth diagonal\n', k);
indata = flip(triu(flip(indata,2), k),2);
```

```
outdata = indata;
end
```

For each of the prescribed amounts of coefficients to drop (shown as percentage as a decimal) a specific diagonal index is selected. This diagonal index was selected through brief trial and error, during which the number of remaining non-zero DCT coefficients was compared to the total number of block coefficients to arrive at the correct amount of remaining coefficients.

The higher frequency coefficients correspond to the higher index coefficients in each block being processed. For this reason, I selected the method shown to extract only the upper left triangle of the matrix that resided below the diagonal index chosen. The matrix needs to be flipped about the 2nd dimension both before and after using the MATLAB built-in *triu* function, since this function extracts the true upper triangle of a square matrix which is not what we are looking for exactly (it is flipped about the y-axis).

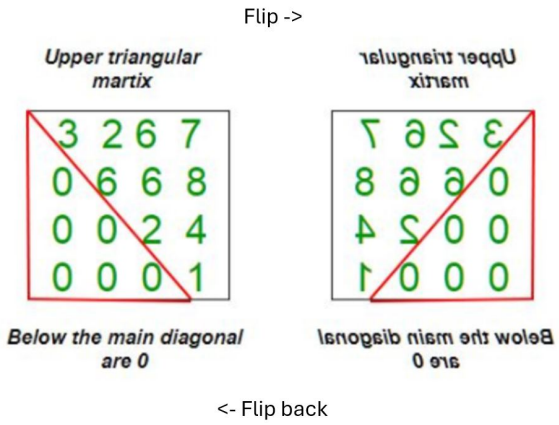

Figure 2: Visual representation of *flip* and *triu* being used to extract the lower frequency coefficients we wish to preserve.

By extracting the the triangle defined below the diagonal index, and placing these coefficients in a new matrix, we have essentially "zeroed out" the higher frequency coefficients corresponding to the percentage we wish to remove. The 2-D plot of the extracted DCT coefficients is provided by Figure 3.

2.5 Quantization

As part of the drive to digitally store the compressed frequency domain representation of the original image, the next stage to implement involved quantization. To perform this step a uniform scalar 8-bit quantizer was implemented. After applying this quantization to the remaining DCT coefficients of the image, rounding was applied to force the truncated coefficients to integer values that can be represented by a standard unsigned

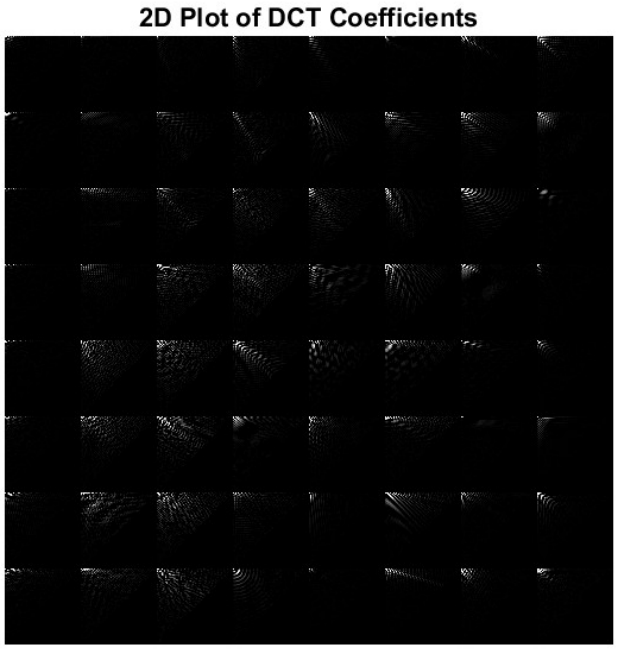

Figure 3: 2-D plot of the DCT coefficients that are preserved after dropping 50% of the higher frequency coefficients

integer (uint8) data type. The code that performs this quantization is provided below.

```
%% quantize
fmin = abs(min(b(:)));
fmax = abs(max(b(:)));
sf = (fmax-fmin)/(2^qbits);
[m,n] = size(b);
c = zeros(m,n);
for i = 1:m
    for j = 1:n
        if b(i,j) ~= 0      % only quantize non-zero coeffs
            c(i,j) = b(i,j)/sf;
        end
    end
end
```

```
c = round(c);
if debug
    figure()
    imshow(c, [0 255]);
    title('8 bit Quantized and Rounded DCT Coefficients');
end
```

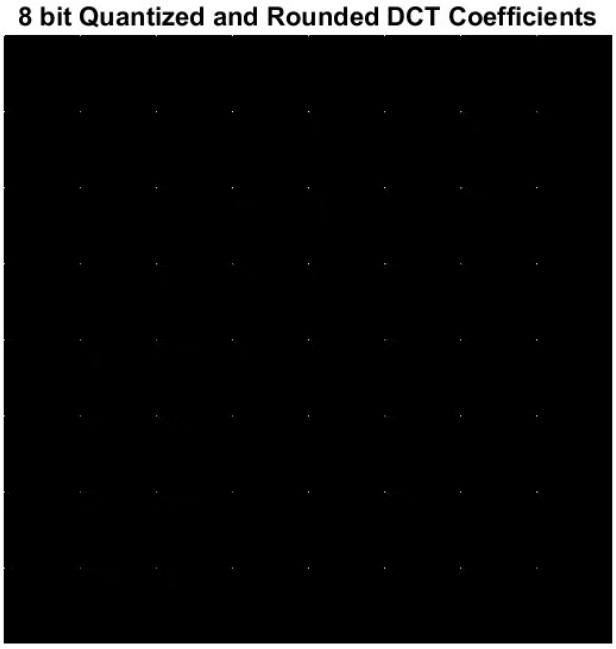

Figure 4: 2-D plot of the DCT coefficients after 8-bit uniform scalar quantization has been performed.

The quantization step leaves only a few bits to represent the lower magnitude coefficients due to the skew in the dynamic range that is introduced due to the large DC offset. In future work, quantization error could be reduced by implementing a separate quantization step for the DC offset of each sub block being processed. **Currently the quantization error due to the phenomenon does not allow for perfect reconstruction with an 8-bit uniform quantizer (since many of the non-DC coefficients have meaningful data represented in their fractional representation, which is then rounded off in order to store the image using uint8 data types)**.

2.6 De-Quantization and Reconstruction

To begin to reconstruct the image, we would need to "dequantize" the quantized and rounded data. This is performed by simply multiplying back the scaling factor that was used to quantize the data originally. The code to perform this is provided below.

```
%% dequantize
d = zeros(m,n);
for i = 1:m
    for j = 1:n
        if c(i,j) ~= 0
            d(i,j) = c(i,j) * sf;
        end
    end
end
if debug
    figure()
    imshow(d, [0 255]);
    title('DE-quantized dct')
end
```

2.7 Inverse DCT

To complete the image reconstruction, we pass the de-quantized DCT coefficients through the *block_process* function once more, this time with the *inverse* flag set to 1.

```
%% idct
inverse = 1;
e = block_process(d, block_count, 0, inverse);
e = round(e);
figure()
imshow(e, [0 255])
colormap(gray);
axis equal;
axis off;
```

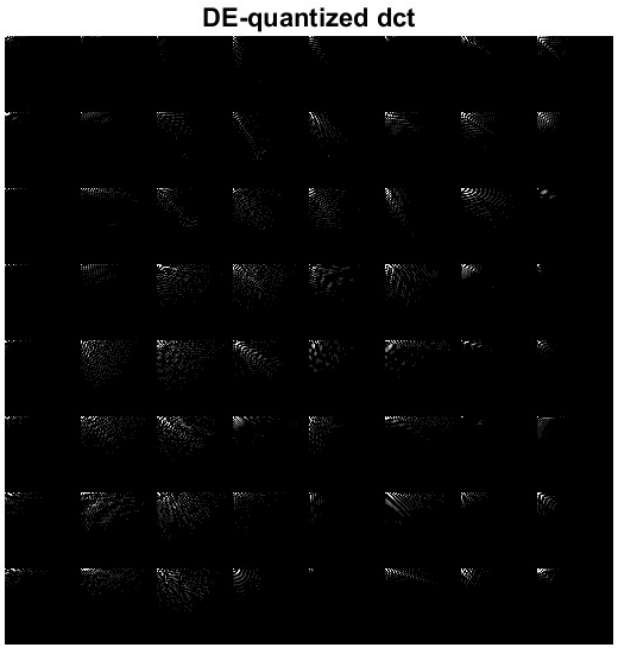

Figure 5: Recovered DCT coefficients after performing de-quantization.

```
title(['Image Reconstructed from Quantized DCT with ',
   perc_coeff_2_drop, ' % of Coefficients Dropped'])
```

2.8 Metrics

Additional performance metrics are output, including the total number of bits used to store the quantized DCT image, the average number of bits per pixel, and the peak signal-to-noise-ratio (PSNR). The MATLAB code that performs these calculation is provided below.

```
%% metrics
valid_pixels = nnz(c);    % number non zero entries in quantized dct
   data
compressed_bit_count = valid_pixels * qbits %
avg_bits_pp = compressed_bit_count / (m*n)
numpixels = m*n;
original_numbits = numpixels * 8;
compression_ratio = compressed_bit_count/original_numbits
```

Figure 6: Reconstructed “lena.raw” after the Inverse DCT is performed.

```
% calculate peak snr in dB
[pksnr, snr] = psnr(e,a);
fprintf('Peak SNR (dB): %f\n', 10*log10(pksnr))

% check to make sure we removed / retained the correct % of
   coefficients
valid = floor((nnz(b)/(m*n))*100);
fprintf('Percent of non-zero coefficients: %f\n', valid);
```

3 Results

This section will provide detail performance metrics for the results from 5 example cases, where each 0%, 50%, 75%, 90%, and 95% of the high frequency DCT coefficients are removed.

Figure 7: Original Image, “lena.raw”.

Figure 8: Reconstructed “lena.raw” with 0% of coefficients dropped.

3.1 0 % Coefficient Removal

For the first case, 0% of high frequency coefficients were removed from the DCT (meaning this should be a completely lossless and invertible trasnformation). However due to the use of a single quantizer and its limited dynamic range being skewed by the large DC coefficient, the reconstruction of the image was not perfect. This was identified to occur during the rounding operation that occurs after initial quantization. Many low magnitude coefficients were rounded off to 0. As suggested before, this could be correct via the

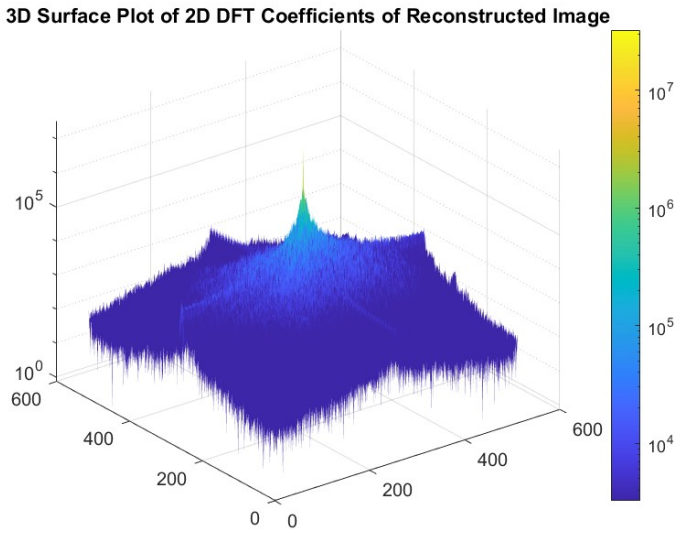

Figure 9: 2-D DFT of the reconstructed image.

use of separate quantization steps for both the large DC offset values and the remaining AC coefficients. See Figure 7, Figure 8, and Figure 9 for reference.

Ideally, we would have seen 512 * 512 * 8 (2 Mb) bits being used to code the compressed image, however my implementation only has 129.8 kb. This represents and average bits per pixel of 0.4952 and a Peak SNR of 11.75 dB. If I removed the round operation after initial quantization we see a compressed bit count of 2,097,157 bits, with an average of 8 bits per pixel and an infinite value Peak SNR, which is more expected for a lossless invertible transformation.

The quality of the reconstructed image is not perfect, due to the above limits of this implementation. However the quality is still present with good edge definition and only slight "graininess" present to indicate degradation.

3.2 50 % Coefficient Removal

For this case, 50 % of the higher frequency coefficients were removed after the initial DCT transform. The coefficients were removed as described in the previous sections, with the diagonal index set to 0, indicating this to be the main diagonal. See Figure 7, Figure 10 and Figure 11.

This resulted in a compressed bit count (number of bits required to code the image) of 128,760. We saw and average bits per pixel ratio of 0.4912 and a PSNR of 11.75 dB

The quality of this reconstruction is very similar to the result with 0 % of the coefficients dropped. This is due to the fact that this case removed many of the lower value coefficients (high frequency) that were

Figure 10: Reconstructed “lena.raw” with 50% of coefficients dropped.

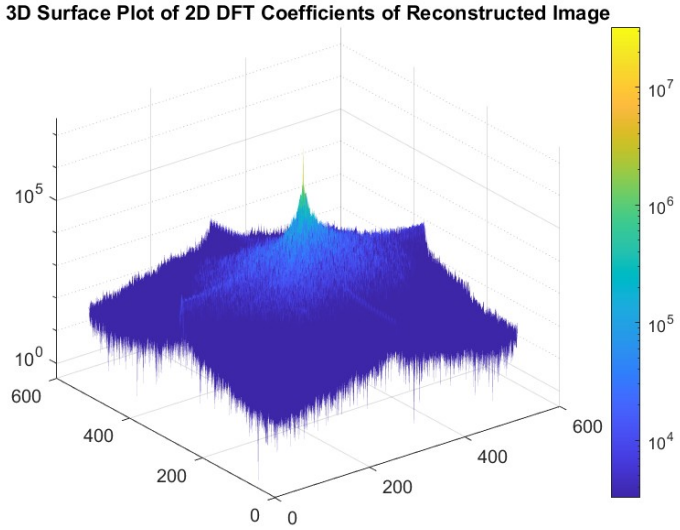

Figure 11: 2-D DFT of the reconstructed image.

otherwise already set to 0 by the rounding process and limited dynamic range of the quantizer.

3.3 75 % Coefficient Removal

For this case, 75 % of the higher frequency coefficients were removed after the initial DCT transform. See Figure 7, Figure 12 and Figure 13.

This resulted in a compressed bit count (number of bits required to code the image) of 116,568. We saw and average bits per pixel ratio of 0.4447 and a PSNR of 11.88 dB

Figure 12: Reconstructed “lena.raw” with 75% of coefficients dropped.

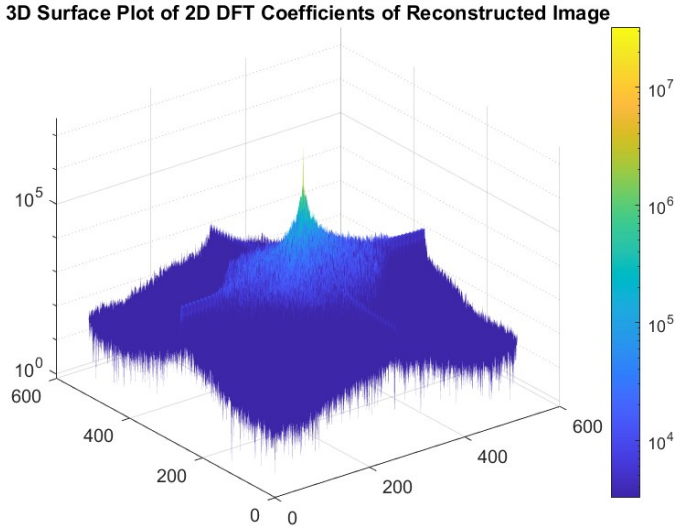

Figure 13: 2-D DFT of the reconstructed image.

The quality of this reconstruction is still not clearly much worse than any of the previous cases. This is still largely due to the fact that the percentage of high frequency coefficients being removed do not contain large amounts of information necessary to reconstruct the image to a high degree of quality.

3.4 90 % Coefficient Removal

For this case, 90 % of the higher frequency coefficients were removed after the initial DCT transform. See Figure 7, Figure 14 and Figure 15.

Figure 14: Reconstructed “lena.raw” with 90% of coefficients dropped.

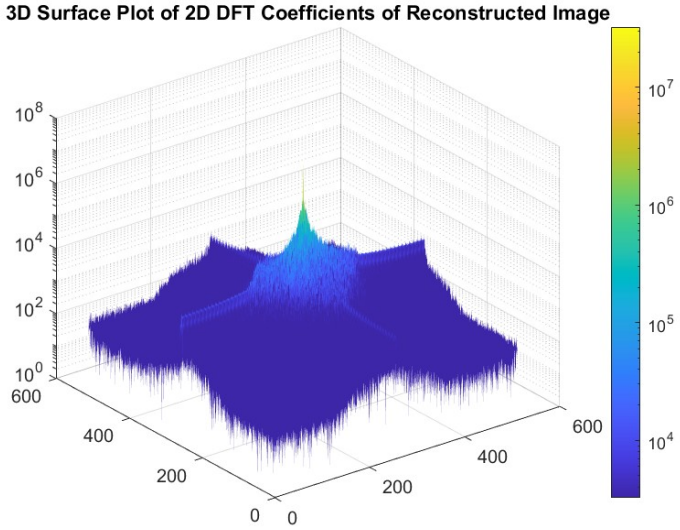

Figure 15: 2-D DFT of the reconstructed image.

This resulted in a compressed bit count (number of bits required to code the image) of 86,064. We saw and average bits per pixel ratio of 0.3283 and a PSNR of 12.41 dB

The quality of this reconstruction is starting to show signs of fading and reduced clarity of the edges found on the woman’s hat and other areas that have contrasting depths or sharp transitions.

Figure 16: Reconstructed “lena.raw” with 95% of coefficients dropped.

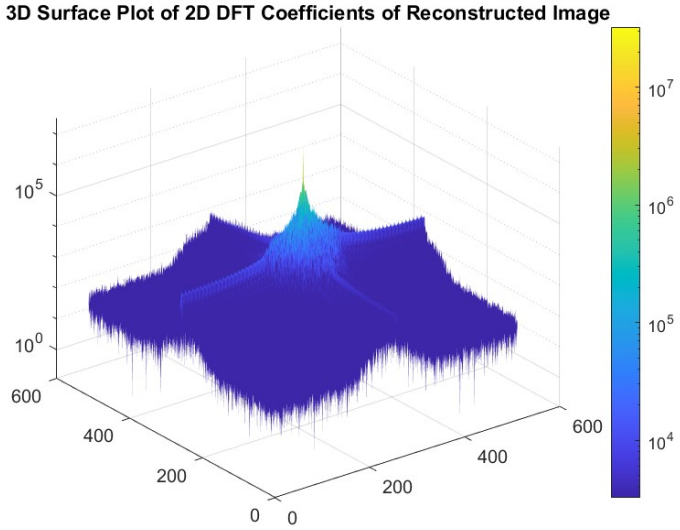

Figure 17: 2-D DFT of the reconstructed image.

3.5 95 % Coefficient Removal

For this case, 95 % of the higher frequency coefficients were removed after the initial DCT transform. See Figure 7, Figure 16 and Figure 17.

This resulted in a compressed bit count (number of bits required to code the image) of 57,736. We saw and average bits per pixel ratio of 0.2202 and a PSNR of 12.9 dB

The quality of this reconstruction is now looking much less clear. There are obvious signs of blurring

or reduced distinction at image edges and we can even see some underlying indication of the subblock boundaries present within the image (due to mismatch at edges of each processed block).

4 Conclusion

It is clear the the DCT process can heavily reduce the number of bits needed to fully encode an image and still reproduce the image with only small changes in observable quality. We were able to witness the correlation between high frequency components and image edges, which gives some insight into edge detection methods. Care must be taken to properly quantize the data after DCT transform compression such that the 8 bit dynamic range is not compressed due to the large DC offsets that are present within each block. Improvements could be made to this MATLAB routine in the future if better performance is desired, such as using fast DFT computational algorithms to speed up the calculation of the DCT and IDCT, and restructuring the quantizer methodology. Care should also be directed to the method in which higher frequency coefficients are removed. Variations in this approach can yield differences in both measured and observable quality as well as compression ratio. The JPEG standard defines specific masks that are used for this purpose.