

**Module Title: Image Analysis**

**Module Code: CS3IA16**

**Lecturer responsible: Prof James Ferryman**

**Type of Assignment: Technical report**

**Group Assignment: Group work**

**Hours spent: 24 hours**

**Student ID's: 29014027, 29003671, 29012930**

<b>Name</b>	<b>Contribution</b>	<b>Note</b>	<b>Signature</b>
<b>Hamza Shahid</b>	<b>100%</b>	<b>Coding, Report</b>	<b>HS</b>
<b>Ranjeth Ravichandran</b>	<b>100%</b>	<b>Coding, GUI, Report</b>	<b>RR</b>
<b>Richard Gibson</b>	<b>100%</b>	<b>Coding, Report</b>	<b>RG</b>

## Contents

Abstract.....	3
Introduction .....	3
Methodology.....	3
Algorithm 1- JPEG scheme using Huffman encoding: .....	4
Algorithm 2 SVD theory and implementation.....	5
Metrics .....	5
Additional features.....	5
Results.....	6
JPEG Compression Image .....	6
SVD Compression Image .....	7
Explanation of Dinosaur Image .....	8
Difference between Lossy and Lossless .....	8
JPEG Compression.....	8
SVD Compression .....	9
How to improve the results?.....	10
SVD K-Value .....	10
JPEG Q-Value .....	11
Additional Features.....	12
Conclusion.....	13
Appendix .....	14
Fig 1, GUI, JPEG, SVD, METRICS.....	14
JPEG Compression.....	14
SVD Compression .....	16
Fig 2, ZIG ZAG & RLE Attempted.....	18
Bibliography .....	22

## Abstract

Digital images are captured for a variety of purposes in modern society, ranging from training neural networks for computer imaging to photography for personal use. As images become easier to capture and higher resolution, the storage requirements for them increase. By removing redundancies in digital images, the required storage space for each image can be reduced dramatically.

This report includes a comprehensive overview of the development, implementation, and evaluation of code for image compression and decompression. The primary objective of this exercise is to demonstrate the full compression and decompression cycle, using key metrics such as mean square error, compression ratio and peak signal to noise ratio to evaluate the efficacy of the developed algorithms.

The MATLAB programming language was used to process images. Images are converted to doubles, DCT is applied to each 8\*8 block in the image which is then flattened. A quantization matrix is used to restrict the number of values that can be saved without loss of information. Finally, Huffman encoding is used on the resulting data, as well as that a comparison algorithm singular value decomposition is also used for image compression. Image metrics are then produced and relayed to the user.

## Introduction

The cost of storing information digitally has increased dramatically in recent years, with predictions that the world will produce more than 180 zettabytes of data in 2025 (Taylor, 2023). The availability of high-resolution digital imagery and use cases such as training artificial intelligence and entertainment media has increased the cost associated with storing data. Efficient encoding algorithms are becoming increasingly important to keep up with the growing demand for high resolution imagery. Compressed images take less space to store and are easier to transmit over networks, making compression and decompression algorithms an important and relevant area of study.

This report details the methods used to compress and decompress images using the MATLAB programming language. The aim of this is to compress the image as much as possible whilst retaining an acceptable level of information from the original image using both lossless and lossy compression algorithms. The algorithms explored in this report are DCT, Huffman and SVD. These algorithms and their effects on the data are explained in detail in the methodology section. The code was designed to demonstrate the compression / decompression cycle by loading and displaying the original image, compressing it, and saving it to memory. The compressed image data is loaded from memory, decompressed, and displayed to the user along with data to assist in evaluating how effectively the image has been compressed.

This report demonstrates compression and decompression algorithms written in the MATLAB programming language, along with metrics to evaluate their efficacy.

## Methodology

The development methodology involved is comparing different algorithms to see which was the most efficient compression technique which maximises compression while maintaining a good level of image quality. To implement these, MATLAB was used. A reason why MATLAB is used is due to efficient matrix computing as it allows easy matrix operations to be implemented. Also, it has set of

built in functions for image processing such as reading and writing images to disk and memory and for encoding techniques such as Huffman and RLE.

The algorithms implemented and experimented was Huffman using the JPEG scheme, Run Length encoding (RLE), and singular value decomposition (SVD). SVD and Huffman were implemented successfully, however RLE was not able to run fully (see figure 2 in appendix for attempt). The aim was to use a combination of lossy and lossless techniques to see which process gave us optimal compression and high quality. For Lossy compression the JPEG encoding scheme was implemented. This allowed significant reduction in image file size, but with some information loss. For the lossless process, SVD was implemented which ensured no information was lost. Both algorithms implemented was aimed at removing either spatial, code or psychovisual redundancy. A comparison was conducted as shown in the results section. Below we outline the theory and implementation details of the two successful algorithms:

#### Algorithm 1- JPEG scheme using Huffman encoding:

The first algorithm was the lossy JPEG scheme using the Huffman encoding. To implement this the first step was loading the four images into the MATLAB scene. The decision was made to convert the images to grey scale, allowing the image compression techniques to compress the image to its barest minimum pixel due to the fact greyscale images have only 1 intensity compared to colour images with 3. To conduct this the “`im2gray()`” built in MATLAB function was used to convert each of the coloured represented images into its grey scale representations.

Next DCT and quantization are applied. For this the image is split into 8\*8 blocks. The reason is images usually have local/grouped up variations and structures. The JPEG scheme divides images into 8\*8 blocks so the image compression techniques can adapt to local characteristics in the image. The DCT (discrete cosine transform) is applied to each block. This transforms an image from its spatial domain to its frequency domain. This allows to only concentrate the images data in a smaller number of coefficients and retaining only the most significant parts allowing the removal of spatial redundancy. To implement, the “`DCT2()`” MATLAB function is applied to each of the 8\*8 blocks. Once applied each of these DCT blocks had quantization applied. The aim of quantization is to remove psychovisual redundancy, certain intensity values are not important and removing them has no effect on how the image will look to the human eye hence can be removed. To implement this, mapping is conducted from a range of continuous values to a smaller range of discrete values. This is done by dividing each of the 8\*8 blocks with a quantization matrix (Q). The quantized 8\*8 blocks are then reconstructed together to a whole image which is then ready for Huffman encoding to take place.

Huffman encoding is then applied on the quantized image. This removes code redundancy from the image. Huffman aims to reduce the amount of data i.e., number of bits to represent the image and its individual pixel values. Huffman looks at the data stream that makes up the image, it uses entropy to calculate the probability that certain pixel values occur. Those data points in the image that occur the most are assigned shorter bit representation. This allows to use a lower number of bytes to have to represent the image. To implement, two key MATLAB functions are used. Firstly, “`Huffmandict()`”, this creates a Huffman dictionary based on set of symbols(data points) in the image and how frequently the symbols occur using probabilities. The output of this dictionary is an array showing each symbol in the image and its new corresponding shortened Huffman code. Next, The “`Huffmanenco()`” is used to encode the image data. It takes information from the Huffman dictionary and converts the pixel values to its new representations. Later when the image is decoded the “`huffmandeco()`” function is used. Once all the above steps are completed the images were stored into memory and loaded in again to perform decoding on.

Once this was completed the image needed to be decompressed so that it can be displayed on the screen. The “Huffmandeco()” and “Idct2()” functions were used to allow this process to be possible. Also, dequantization is applied by multiplying each of the 8\*8 image blocks by the quantization matrix.

### Algorithm 2 SVD theory and implementation

To compare to the JPEG scheme, Singular value decomposition was implemented (SVD). The aim was to make this a lossless compression technique meaning no information is lost to see how it compares to JPEG. The process decomposes the image matrix into three simpler matrices to represent images in a more compact way using less data. To implement this in MATLAB, the “svd()” function is applied to the original image causing three new matrices [U,S,V] to be computed. Also, the rank is computed, and the compression level is decided by specifying the number of singular values to retain (K). Once this is decided, the three matrices split are truncated so each only retains K singular values and therefore only keeping the most necessary data. To ensure lossless compression, the K value was increased until it is equal to the rank of the original matrix.

### Metrics

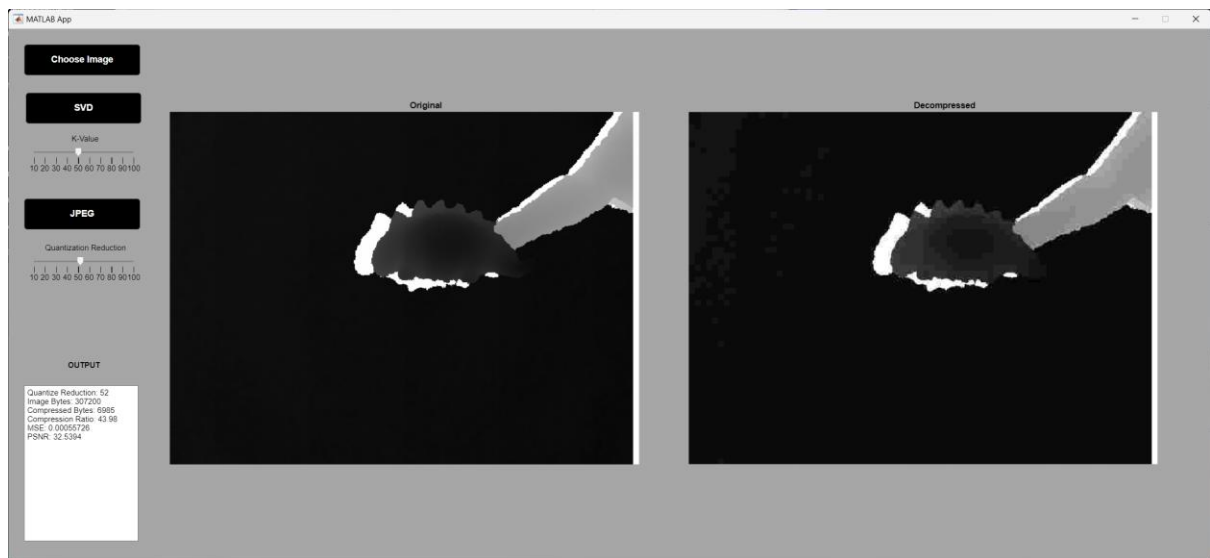
To evaluate the performance of algorithms Metrics are used. The first is the compression ratio (CR) which shows the level of compression achieved. A compression ratio that is greater than 1 indicates compression is achieved. The larger the compression ratio value the more the image data has been compressed. To implement the compression ratio, the size of the original image was divided by the new compressed image, and this was stored in a variable.

However, not only showing high compression is important, but to also ensure quality is remained. For this two additional metrics are applied. The first is mean squared error (MSE), used to quantify the difference between the original image and the compressed image. The aim was to try and replicate the original image as much as possible, so this metric allows numerically to measure the quality of the compressed image and how well the compression algorithm preserves the details of the original image. In order to implement this the MATLAB function “immse()” is used which takes in the original image and compressed image as parameters. The lower the MSE value the better quality has been remained.

Finally, PSNR(peak signal to noise ratio) is calculated which also assesses how well quality is remained. To do this we used the “psnr()” built in function. The higher the PSNR value indicates that better image quality is maintained. If the PSNR is above 30 show that the image quality is very high.

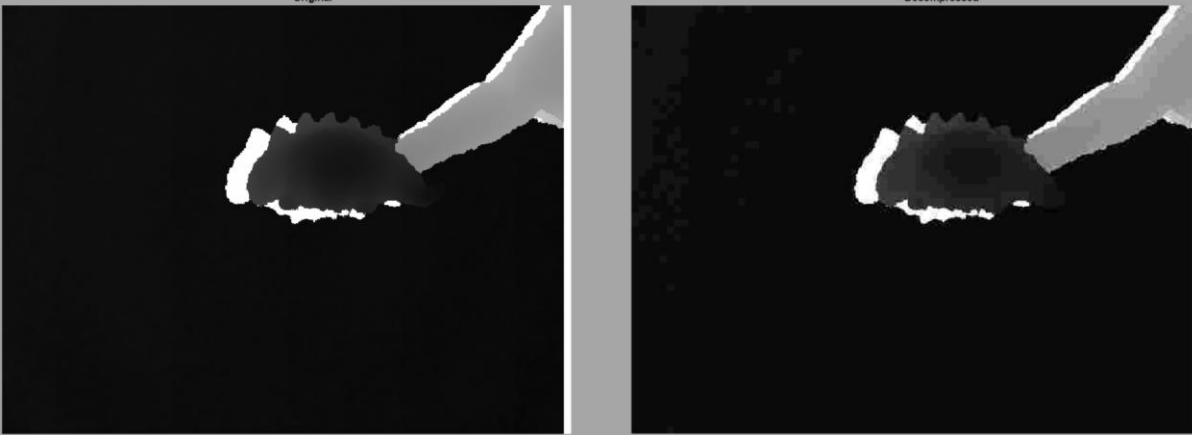

### Additional features

To be able to run different image compression techniques for different algorithms efficiently a GUI was built. The GUI was created using MATLAB Apps. The features included are buttons, sliders for defining the K value, images to select from, and a file select. These features are inbuilt MATLAB objects which can be dragged and dropped onto the scene window. Examples are the choose image button to select the desired input image for the compression algorithm by opening a separate window directory. JPEG button, which calls the JPEG scheme built in MATLAB functions we designed on the desired image and SVD button. The image below shows the screenshot of the GUI page.



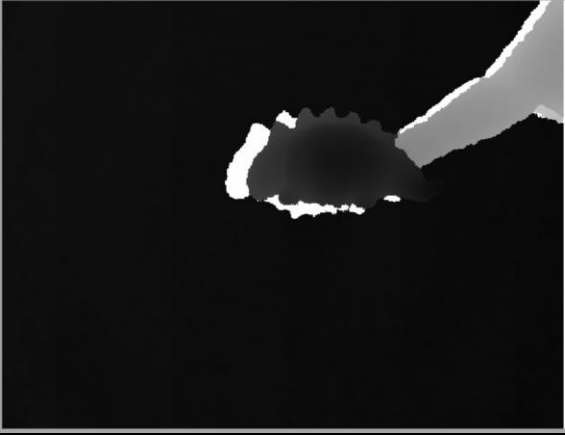
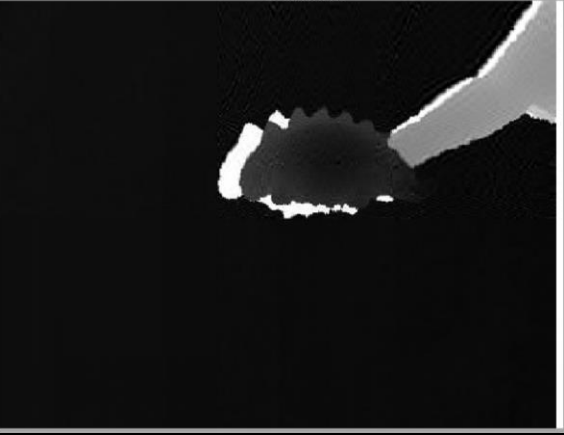
## Results







The results table below shows the screenshots and metrics values of each of four images provided, the decompressed image, and the metrics on each image. Variables were used to store the compressed image data which allowed the extraction of the number of bytes of the compressed image.

JPEG Compression Image		Result
		Quantize Reduction: 52 Image Bytes: 307200 Compressed Bytes: 6985 Compression Ratio: 43.98 MSE: 0.0005726 PSNR: 32.5394
		Quantize Reduction: 52 Image Bytes: 327680 Compressed Bytes: 9627 Compression Ratio: 34.0376 MSE: 0.00034328 PSNR: 34.6435



		<p>Quantize Reduction: 52 Image Bytes: 12192768 Compressed Bytes: 942248 Compression Ratio: 12.9401 MSE: 0.0029642 PSNR: 25.2809</p>
		<p>Quantize Reduction: 52 Image Bytes: 12192768 Compressed Bytes: 1591341 Compression Ratio: 7.6619 MSE: 0.0057543 PSNR: 22.4001</p>

SVD Compression Image		Results
		<p>K-Value: 50 Original Bytes: 307200 Compressed Bytes: 58500 Compression Ratio: 5.2513 MSE: 0.00055296 PSNR: 32.5731</p>

		<p>K-Value: 50  Original Bytes: 327680  Compressed Bytes: 60100  Compression Ratio: 5.4522  MSE: 0.00013827  PSNR: 38.5926</p>
		<p>K-Value: 50  Original Bytes: 12192768  Compressed Bytes: 355300  Compression Ratio: 34.3168  MSE: 0.014898  PSNR: 18.2687</p>
		<p>K-Value: 50  Original Bytes: 12192768  Compressed Bytes: 355300  Compression Ratio: 34.3168  MSE: 0.042185  PSNR: 13.7484</p>

## Explanation of Dinosaur Image

### Difference between Lossy and Lossless

Lossy compression techniques achieve higher compression ratios by discarding some of the data. This results in a smaller file size, but it also means that the original data cannot be perfectly reconstructed from the compressed data. Lossless compression techniques do not discard any data, so the original data can be perfectly reconstructed from the compressed data.

### JPEG Compression

The original image was 307,200 bytes, but the compressed image was only 6,985 bytes. This means that the image has been compressed by a factor of 43.98. The Mean Squared Error (MSE) was 0.00055726, which is a very low value. This means that the compressed image is very close to the



original image in terms of visual quality. The Peak Signal-to-Noise Ratio (PSNR) was 32.5394, which is also a very high value. This means that the compressed image is very clear and detailed.

In general, JPEG compression is a lossy compression method, which means that some information is lost in the compression process. However, in this case, the loss of information is very small and is not noticeable to the human eye. This is because the quantize parameter was set to 52, which is a relatively high value. A higher quantize parameter will result in a smaller compressed file size, but at the expense of some image quality.

The results of compressing this image show that JPEG compression can be a very effective way to reduce the file size of images without sacrificing too much quality. This makes JPEG compression a popular choice for web images, where file size is important.

### SVD Compression

The original image had a file size of 307,200 bytes. After applying SVD compression, the file size was reduced to 58,500 bytes, representing a compression ratio of 5.25. This means that the image was compressed by approximately 5.25 times.

The MSE (Mean Squared Error) is a measure of the difference between the original image and the compressed image. A low MSE indicates that the compressed image is very close to the original image in terms of visual quality. In this case, the MSE is 0.00055296, which is a very low value.


The PSNR (Peak Signal-to-Noise Ratio) is another measure of image quality. A higher PSNR indicates a clearer and more detailed image. In this case, the PSNR is 32.5731, which is also a very high value. This means that the compressed image is very clear and detailed.

Overall, the results of compressing the image using SVD show that it is an effective method for reducing file size without sacrificing too much image quality. The compression ratio of 5.25 is significant, and the MSE and PSNR values are very low, indicating that the compressed image is highly faithful to the original.


### How to improve the results?

To improve the results the user can change the compression values for SVD and JPEG to be lower or higher. In the case for SVD, a higher k value will result in a smaller compressed file size, but at the expense of some image quality. A lower k value will preserve more image quality, but at the expense of a larger compressed file size.

#### SVD K-Value

SVD Compression Image (Low K-Value)		Results
		K-Value: 10 Original Bytes: 307200 Compressed Bytes: 11300 Compression Ratio: 27.1858 MSE: 0.0034535 PSNR: 24.6174

The compressed images become more distorted as the k value decreases. This is because the lower k values correspond to less important components of the image, which are discarded in the compression process.


SVD Compression Image (High K-Value)		Results
		K-Value: 100 Original Bytes: 307200 Compressed Bytes: 122000 Compression Ratio: 2.518 MSE: 0.00018998 PSNR: 37.2128

The higher the k value, the more information will be preserved in the compressed image, and the less distorted the image will appear. The MSE decreases as the k value increases. This means that the compressed image becomes more faithful to the original image as the k value increases.

As you can see, the image with k=100 has the lowest MSE and the highest PSNR, which means that it is the most faithful to the original image. The image with k=10 has the highest compression ratio, which means that it is the smallest in terms of file size.

## JPEG Q-Value

In the case for JPEG the Q-Value can be lowered or increased, which may result in the image being more distorted or true to original, this will either decrease the number of bytes or retain quality.

JPEG Compression (High Q-Value)		Results
		Quantize Reduction: 100 Image Bytes: 307200 Compressed Bytes: 9250 Compression Ratio: 33.2108 MSE: 0.00034369 PSNR: 34.6383

The quantize parameter of 100 has resulted in a very high compression ratio of 33.2108, meaning that the compressed image is only 33.2% of the size of the original image. However, the compression has also resulted in some loss of image quality, as evidenced by the Mean Squared Error (MSE) of 0.00034369 and the Peak Signal-to-Noise Ratio (PSNR) of 34.6383. These values indicate that there is a small amount of noise in the compressed image, but it is not noticeable to the naked eye.

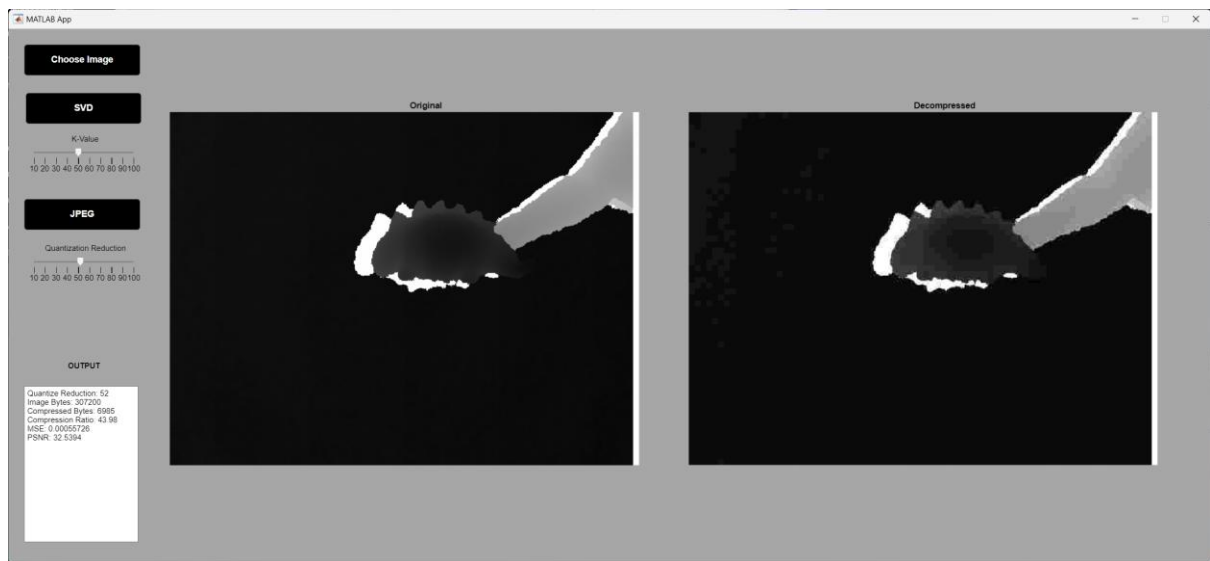
JPEG Compression (Low Q-Value)		Results
		Quantize Reduction: 10 Image Bytes: 307200 Compressed Bytes: 3207 Compression Ratio: 95.7905 MSE: 0.0034172 PSNR: 24.6633

The quantize parameter of 10 has resulted in a relatively low compression ratio of 95.7905, meaning that the compressed image is 4.21% larger than the original image. However, the compression has also resulted in significantly better image quality, as evidenced by the Mean Squared Error (MSE) of 0.0034172 and the Peak Signal-to-Noise Ratio (PSNR) of 24.6633. These values indicate that there is very little noise in the compressed image, and it is very close to the original image in terms of visual quality.

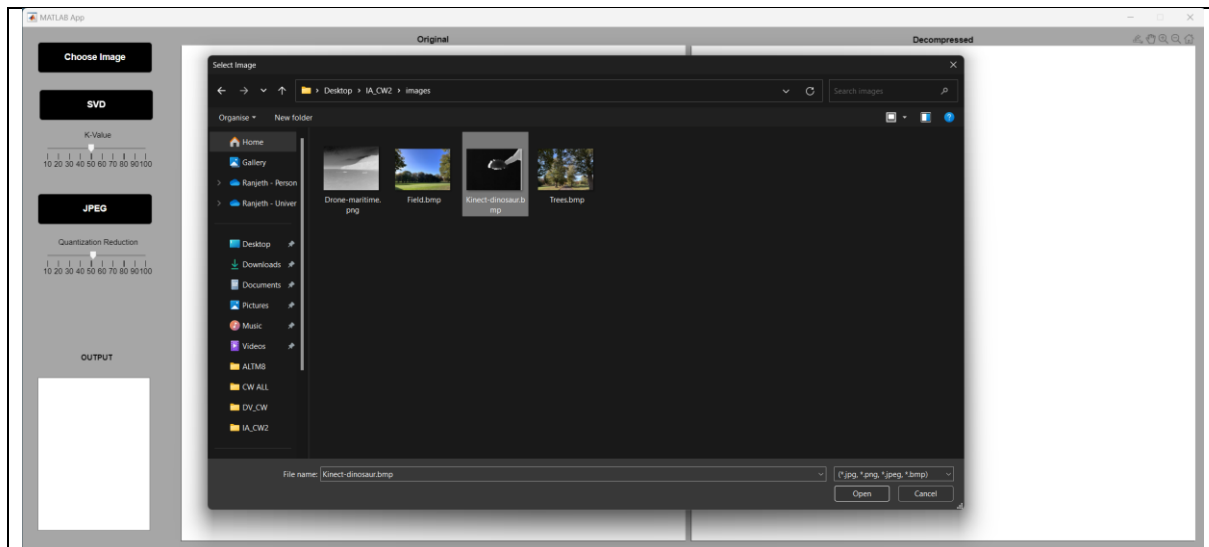
The JPEG compression with a high quantize parameter provides a better compression ratio and a slightly lower MSE than the JPEG compression with a low quantize parameter. However, the JPEG compression with a low quantize parameter provides a much better PSNR and visual quality.

Compression Method	Advantage	Disadvantage
JPEG with high quantize parameter	Smaller file size	Some loss of quality
JPEG with low quantize parameter	Better visual quality	Larger file size

## Additional Features



Feature	Explanation
GUI	<p>Used to visualise the techniques used through the coursework as well as displaying the input and output images after processes have been done. The GUI also allows interactivity which changes how the images are displayed.</p> <p>The GUI was created through MATLAB Apps, features such as the Buttons, Sliders, Images, Select File, can be accessed within the GUI. These features are inbuilt MATLAB objects which can be drag and dropped onto the scene/window. Each object may have their own unique functionality/callback such as the “choose image” button, this allows the user to select their desired input image by opening up a separate window directory to locate their file, another function is the “JPEG” button when interacted upon the button calls a function that applies the JPEG Compression technique onto the loaded input image, which eventually is presented onto the decompressed field.</p>
Selectable Compression Level	This an interactive slider on the GUI which allows the user to change the compression quality for SVD and JPEG.
Save/Read Image	The user can interact with the GUI and select their desired image, which will be read in, and after a compression technique has been run an output image will be saved, loaded, and visualised in the GUI.



Further Metrics

Metrics which has been used to describe the images through quantitative data are, Bytes, MSE, Compression Ratio, and PSNR.

```
app.OUTPUTTextArea.Value = sprintf('Quantize Reduction: %s \nImage Bytes: %s\nCompressed Bytes: %s\nCompression Ratio: %s \nMSE: %s \nPSNR: %s', num2str(size_of_compression), num2str(original_size/8), num2str(compressed_size/8), num2str(compression_ratio), num2str(immse(im2double(img), im2double(img_reconstructed))), num2str(peaksnr));
```

## Conclusion

In conclusion, the effectiveness of different image compression techniques has been thoroughly explored in this report. Techniques such as JPEG using Huffman encoding and SVD are efficient in reducing image size whilst maintaining an acceptable level of image quality. JPEG achieved much higher compression ratios as compared to SVD, however JPEG displayed higher levels of information loss as indicated by the higher MSE values and lower PSNR values.

This suggests that the choice between JPEG and SVD depend on user requirements. If loss of information is acceptable, JPEG is suitable. If not, SVD is the superior choice due to it maintaining a higher image quality.

Further research opportunities include investigating how these techniques could be applied to video, removing redundancies spanning over multiple frames. We would aim to implement the algorithms above and apply for scenarios such as videos, sound, and coloured images.

Future work we would like to implement is more lossy algorithms, currently one lossy and on lossless algorithms introduced into this coursework, implementing a series of lossy algorithms would allow for greater comparisons of metrics, image compression, and image quality, an example of such algorithm would be arithmetic algorithm. We would tweak the algorithms to work for both coloured and grayscale images for future work.



## Appendix

Fig 1, GUI, JPEG, SVD, METRICS

```
classdef app1 < matlab.apps.AppBase
% Properties that correspond to app components
properties (Access = public)
    UIFigure matlab.ui.Figure
    QuantizationReductionSlider matlab.ui.control.Slider
    QuantizationReductionSliderLabel matlab.ui.control.Label
    KValueSlider matlab.ui.control.Slider
    KValueSliderLabel matlab.ui.control.Label
    OUTPUTTextArea matlab.ui.control.TextArea
    OUTPUTTextAreaLabel matlab.ui.control.Label
    SVDButton matlab.ui.control.Button
    JPEGButton matlab.ui.control.Button
    ChooseImageButton matlab.ui.control.Button
    UIAxes2 matlab.ui.control.UIAxes
    UIAxes matlab.ui.control.UIAxes
end
% Callbacks that handle component events
methods (Access = private)
% Button pushed function: ChooseImageButton
function chooseImage(app, event)
    global img
    [filename, pathname] = uigetfile({'*.jpg;*.png;*.jpeg;*.bmp'}, 'Select Image');
    if filename ~= 0
        % Process the selected image file
        img = imread(fullfile(pathname, filename));
        % Perform any desired image processing or operations using the 'image' variable
    end
    % Display the image in the UI axes
    imshow(im2gray(img), 'Parent', app.UIAxes);
    cla(app.UIAxes2);
End

JPEG Compression
% Button pushed function: JPEGButton
function JPEG_Compression(app, event)
    sliderValue = app.QuantizationReductionSlider.Value;
    global img
    img = im2gray(img);
    % Get the size of the original image
    size_img = size(img);
    % Convert the image to double
    img = im2double(img);
    % Define a quantization matrix
    size_of_compression = sliderValue; % User selectable compression level -> the
    lower this value the more compression

Encoding
Q = [16 11 10 16 24 40 51 61;
    12 12 14 19 26 58 60 55;
    14 13 16 24 40 57 69 56;
    14 17 22 29 51 87 80 62;
    18 22 37 56 68 109 103 77;
    24 35 55 64 81 104 113 92;
    49 64 78 87 103 121 120 101;
    72 92 95 98 112 100 103 99];
Q = Q / size_of_compression; % less aggressive quantization
% Initialize an empty matrix for the quantized DCT coefficients
```

```

quantized_img = zeros(size(img));
% Divide the image into 8x8 blocks and apply DCT and quantization to each
for i = 1:8:size(img, 1)
for j = 1:8:size(img, 2)
block = img(i:i+7, j:j+7);
dct_block = dct2(block);
quantized_block = round(dct_block ./ Q);
quantized_img(i:i+7, j:j+7) = quantized_block;
end
end
% Flatten the quantized image into a 1-D array
flattened_img = quantized_img(:);
% Perform Huffman encoding
huff_dict = huffmandict(unique(flattened_img), histc(flattened_img,
unique(flattened_img))/numel(flattened_img)));
huff_code = huffmanenco(flattened_img, huff_dict);
% Save the Huffman encoded image to a file
save('compressed_image.mat', 'huff_code', 'huff_dict', 'Q', 'size_img');
% Get the size of the original image in bytes
original_size = numel(img) * 8; % 8 bits per byte
% Read the size of the compressed data in bytes
compressed_info = dir('compressed_image.mat');
compressed_size = compressed_info.bytes * 8; % 8 bits per byte
% Calculate the compression ratio
compression_ratio = original_size / compressed_size;
% Load the Huffman encoded image from the memory
load('compressed_image.mat');

Decoding
% Perform Huffman decoding
decoded_img = huffmandeco(huff_code, huff_dict);
% Reshape the decoded image back into its original shape
decoded_img = reshape(decoded_img, size(img));
% Initialize an empty matrix for the reconstructed DCT coefficients
idct_img = zeros(size(img));
% Divide the image into 8x8 blocks and apply inverse quantization and IDCT to
each
for i = 1:8:size(img, 1)
for j = 1:8:size(img, 2)
block = decoded_img(i:i+7, j:j+7);
dequantized_block = block .* Q;
idct_block = idct2(dequantized_block);
idct_img(i:i+7, j:j+7) = idct_block;
end
end
% Convert the image back to uint8
img_reconstructed = im2uint8(idct_img);

Metrics
% Metrics and Display
[peaksnr, snr] = psnr(im2double(img), im2double(img_reconstructed));
app.OUTPUTTextArea.Value = sprintf('Quantize Reduction: %s \nImage Bytes:
%s\nCompressed Bytes: %s\nCompression Ratio: %s \nMSE: %s \nPSNR: %s',
num2str(size_of_compression), num2str(original_size/8),
num2str(compressed_size/8), num2str(compression_ratio),
num2str(immse(im2double(img), im2double(img_reconstructed))), num2str(peaksnr));
% Display the image in the UI axes
imshow(img_reconstructed, 'Parent', app.UIAxes2);
End

```

## SVD Compression

```
% Button pushed function: SVDButton
function SVD_Compression(app, event)
    sliderValue = cast(app.KValueSlider.Value, 'int32');
    global img
    % Load the image
    originalImage = im2gray(img);
    % Convert the image to double
    originalImage = im2double(originalImage);
    % Perform SVD on the grayscale image
    [U, S, V] = svd(originalImage);
    % Determine the rank of S
    rankS = rank(S);
    % Compress the image
    k = sliderValue; % Number of singular values to retain
    compressedImage = U(:, 1:k) * S(1:k, 1:k) * V(:, 1:k)';
    % Save the compressed image
    imwrite(compressedImage, 'compressedImage.jpg');
    % MATLAB CODE for calculating compression ratio
    % Get the size of the original image
    originalImageSize = numel(originalImage);
    % Get the size of the compressed image
    compressedImageSize = numel(U(:, 1:k)) + numel(S(1:k, 1:k)) + numel(V(:, 1:k)');
    % Calculate the compression ratio
    compressionRatio = originalImageSize / compressedImageSize;
    % MATLAB CODE for SVD Image Decompression
    % Load the compressed image
    compressedImage = im2gray(imread('compressedImage.jpg'));
    % Convert the image to double
    compressedImage = im2double(compressedImage);
    % Perform SVD on the compressed image
    [U, S, V] = svd(compressedImage);
    % Determine the rank of S
    rankS = rank(S);
    % Reconstruct the image
    k = sliderValue; % Number of singular values to retain
    reconstructedImage = U(:, 1:k) * S(1:k, 1:k) * V(:, 1:k)';
    % Display the image in the UI axes
    imshow(reconstructedImage, 'Parent', app.UIAxes2);
    % Display and calculate metrics
    [peaksnr, snr] = psnr(originalImage, reconstructedImage);
    app.OUTPUTTextArea.Value = sprintf('K-Value: %s \nOriginal Bytes: %s \nCompressed Bytes: %s \nCompression Ratio: %s \nMSE: %s \nPSNR: %s', num2str(sliderValue), num2str(originalImageSize), num2str(compressedImageSize), num2str(compressionRatio), num2str(immse(originalImage, reconstructedImage)), num2str(peaksnr));
end
end
% Component initialization
methods (Access = private)
% Create UIFigure and components
function createComponents(app)
    % Create UIFigure and hide until all components are created
    app.UIFigure = uifigure('Visible', 'off');
    app.UIFigure.AutoResizeChildren = 'off';
    app.UIFigure.Color = [0.651 0.651 0.651];
    app.UIFigure.Position = [100 100 1854 822];
    app.UIFigure.Name = 'MATLAB App';
    app.UIFigure.Resize = 'off';
```

```

% Create UIAxes
app.UIAxes = uiaxes(app.UIFigure);
title(app.UIAxes, 'Original')
app.UIAxes.XTick = [];
app.UIAxes.YTick = [];
app.UIAxes.Position = [244 13 800 800];
% Create UIAxes2
app.UIAxes2 = uiaxes(app.UIFigure);
title(app.UIAxes2, 'Decompressed')
app.UIAxes2.XTick = [];
app.UIAxes2.YTick = [];
colormap(app.UIAxes2, 'gray')
app.UIAxes2.Position = [1043 13 800 800];
% Create ChooseImageButton
app.ChooseImageButton = uibutton(app.UIFigure, 'push');
app.ChooseImageButton.ButtonPushedFcn = createCallbackFcn(app, @chooseImage,
true);
app.ChooseImageButton.BackgroundColor = [0 0 0];
app.ChooseImageButton.FontSize = 14;
app.ChooseImageButton.FontWeight = 'bold';
app.ChooseImageButton.FontColor = [1 1 1];
app.ChooseImageButton.Position = [25 752 179 48];
app.ChooseImageButton.Text = 'Choose Image';
% Create JPEGButton
app.JPEGButton = uibutton(app.UIFigure, 'push');
app.JPEGButton.ButtonPushedFcn = createCallbackFcn(app, @JPEG_Compression, true);
app.JPEGButton.BackgroundColor = [0 0 0];
app.JPEGButton.FontSize = 14;
app.JPEGButton.FontWeight = 'bold';
app.JPEGButton.FontColor = [1 1 1];
app.JPEGButton.Position = [25 515 179 48];
app.JPEGButton.Text = 'JPEG';
% Create SVDButton
app.SVDButton = uibutton(app.UIFigure, 'push');
app.SVDButton.ButtonPushedFcn = createCallbackFcn(app, @SVD_Compression, true);
app.SVDButton.BackgroundColor = [0 0 0];
app.SVDButton.FontSize = 14;
app.SVDButton.FontWeight = 'bold';
app.SVDButton.FontColor = [1 1 1];
app.SVDButton.Position = [27 678 179 48];
app.SVDButton.Text = 'SVD';
% Create OUTPUTTextAreaLabel
app.OUTPUTTextAreaLabel = uilabel(app.UIFigure);
app.OUTPUTTextAreaLabel.HorizontalAlignment = 'right';
app.OUTPUTTextAreaLabel.FontWeight = 'bold';
app.OUTPUTTextAreaLabel.Position = [88 295 54 22];
app.OUTPUTTextAreaLabel.Text = 'OUTPUT';
% Create OUTPUTTextArea
app.OUTPUTTextArea = uitextarea(app.UIFigure);
app.OUTPUTTextArea.Position = [25 35 176 240];
% Create KValueSliderLabel
app.KValueSliderLabel = uilabel(app.UIFigure);
app.KValueSliderLabel.HorizontalAlignment = 'right';
app.KValueSliderLabel.Position = [93 644 47 22];
app.KValueSliderLabel.Text = 'K-Value';
% Create KValueSlider
app.KValueSlider = uislider(app.UIFigure);
app.KValueSlider.Limits = [10 100];
app.KValueSlider.MajorTicks = [0 10 20 30 40 50 60 70 80 90 100];

```

```

app.KValueSlider.MajorTickLabels = {'0', '10', '20', '30', '40', '50', '60', '70',
'80', '90', '100'};
app.KValueSlider.MinorTicks = [];
app.KValueSlider.Position = [40 632 154 3];
app.KValueSlider.Value = 50;
% Create QuantizationReductionSliderLabel
app.QuantizationReductionSliderLabel = uilabel(app.UIFigure);
app.QuantizationReductionSliderLabel.HorizontalAlignment = 'right';
app.QuantizationReductionSliderLabel.Position = [52 476 130 22];
app.QuantizationReductionSliderLabel.Text = 'Quantization Reduction';
% Create QuantizationReductionSlider
app.QuantizationReductionSlider = uislider(app.UIFigure);
app.QuantizationReductionSlider.Limits = [10 100];
app.QuantizationReductionSlider.MajorTicks = [0 10 20 30 40 50 60 70 80 90 100];
app.QuantizationReductionSlider.MinorTicks = [];
app.QuantizationReductionSlider.Position = [40 464 154 3];
app.QuantizationReductionSlider.Value = 52;
% Show the figure after all components are created
app.UIFigure.Visible = 'on';
end
end
% App creation and deletion
methods (Access = public)
% Construct app
function app = app1
% Create UIFigure and components
createComponents(app)
% Register the app with App Designer
registerApp(app, app.UIFigure)
if nargin == 0
clear app
end
end
% Code that executes before app deletion
function delete(app)
% Delete UIFigure when app is deleted
delete(app.UIFigure)
end
end
end

```

---

Fig 2, ZIG ZAG & RLE Attempted

```

clc;
clear all;
close all;
% Images
fieldImage = im2gray(imread('images\Field.bmp'));
kndImage = im2gray(imread('images\Kinect-dinosaur.bmp'));
treesImage = im2gray(imread('images\Trees.bmp'));
dmImage = im2gray(imread('images\Drone-maritime.png'));
imArray = {kndImage}
for image = 1:length(imArray)
img = im2double(imArray{image})
figure, subplot(1,2,1), imshow(img), title('Original');
% Define the block size
block_size = 8;

```



```

% Define the quantization matrix
quantization_matrix = [16 11 10 16 24 40 51 61;
12 12 14 19 26 58 60 55;
14 13 16 24 40 57 69 56;
14 17 22 29 51 87 80 62;
18 22 37 56 68 109 103 77;
24 35 55 64 81 104 113 92;
49 64 78 87 103 121 120 101;
72 92 95 98 112 100 103 99];
% Perform block-based DCT, quantization, zigzag scan, run-length encoding,
% Huffman encoding, inverse Huffman decoding, inverse zigzag scan,
% inverse quantization, inverse DCT, and reconstruct the image
for i = 1:block_size:size(img, 1)
for j = 1:block_size:size(img, 2)
% Extract the block
block = img(i:i+block_size-1, j:j+block_size-1);
% Perform block-based DCT
dct_block = dct2(block);
% Perform quantization
quantized_block = round(dct_block ./ quantization_matrix);
% Perform zigzag scan
zigzag_block = zigzag(quantized_block);
% Perform run-length encoding
rle_block = rle(zigzag_block);
% Perform Huffman encoding
huffman_block = huffman(rle_block);
% Perform inverse Huffman decoding
irle_block = irle(huffman_block);
% Perform inverse zigzag scan
izigzag_block = izigzag(irle_block, block_size, block_size);
% Perform inverse quantization
iquantized_block = izigzag_block .* quantization_matrix;
% Perform inverse DCT
idct_block = idct2(iquantized_block);
% Replace the block in the image
img(i:i+block_size-1, j:j+block_size-1) = idct_block;
end
end
% Display the reconstructed image
subplot(1,2,2), imshow(img, []);
end
function z = zigzag(m)
%ZIGZAG Scans a matrix in a zigzag pattern and returns a vector.
% Z = ZIGZAG(M) scans matrix M in a zigzag pattern and returns the
% resulting vector Z.
% Get the size of the matrix
[mrows, ncols] = size(m);
% Initialize the output vector
z = zeros(1, mrows * ncols);
% Initialize the indices
index = 1;
i = 1;
j = 1;
% Scan the matrix in a zigzag pattern
for k = 1:(mrows + ncols - 1)
if mod(k, 2) == 1
% Scan up the column
while i >= 1 && j <= ncols
z(index) = m(i, j);

```

```

index = index + 1;
i = i - 1;
j = j + 1;
end
if j <= ncols
i = i + 1;
else
i = i + 2;
j = j - 1;
end
else
% Scan down the column
while i <= mrows && j >= 1
z(index) = m(i, j);
index = index + 1;
i = i + 1;
j = j - 1;
end
if i <= mrows
j = j + 1;
else
j = j + 2;
i = i - 1;
end
end
end
end
function r = rle(x)
%RLE Performs run-length encoding on a vector.
% R = RLE(X) performs run-length encoding on the vector X and returns
% the resulting vector R.
% Initialize the output vector
r = [];
% Initialize the current run
current_run = x(1);
current_count = 1;
% Perform run-length encoding
for i = 2:length(x)
if x(i) == current_run
% Increment the current run
current_count = current_count + 1;
else
% Add the current run to the output vector
r = [r current_count current_run];
% Start a new run
current_run = x(i);
current_count = 1;
end
end
% Add the last run to the output vector
r = [r current_count current_run];
end
function x = irle(r)
%IRLE Performs inverse run-length encoding on a vector.
% X = IRLE(R) performs inverse run-length encoding on the vector R and
% returns the resulting vector X.
% Initialize the output vector
x = [];
% Perform inverse run-length encoding

```

```

for i = 1:2:length(r)
x = [x repmat(r(i+1), 1, r(i))];
end
end
function m = izigzag(z, mrows, ncols)
%IZIGZAG Scans a vector in a zigzag pattern and returns a matrix.
% M = IZIGZAG(Z, MROWS, NCOLS) scans vector Z in a zigzag pattern and
% returns the resulting matrix M with MROWS rows and NCOLS columns.
% Initialize the output matrix
m = zeros(mrows, ncols);
% Initialize the indices
index = 1;
i = 1;
j = 1;
% Scan the vector in a zigzag pattern
for k = 1:(mrows + ncols - 1)
if mod(k, 2) == 1
% Scan up the column
while i >= 1 && j <= ncols
m(i, j) = z(index);
index = index + 1;
i = i - 1;
j = j + 1;
end
if j <= ncols
i = i + 1;
else
i = i + 2;
j = j - 1;
end
else
% Scan down the column
while i <= mrows && j >= 1
m(i, j) = z(index);
index = index + 1;
i = i + 1;
j = j - 1;
end
if i <= mrows
j = j + 1;
else
j = j + 2;
i = i - 1;
end
end
end
end
function decoded_sig = huffman(sig)
% Compute the probability of each symbol
p = hist(sig, unique(sig)) / length(sig);
% Create a Huffman dictionary
dict = huffmandict(unique(sig), p);
% Encode the input signal
code = huffmanenco(sig, dict);
% Decode the encoded signal
decoded_sig = huffmandeco(code, dict);
end

```

---

## Bibliography

Muzhir Shaban AL-Ani, F. H. (2013). *THE JPEG IMAGE COMPRESSION ALGORITHM*. Anbar: International Journal of Advances in Engineering & Technology.

Taylor, P. (2023, 12 04). *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025*. Retrieved from Statista: <https://www.statista.com/statistics/871513/worldwide-data-created/>