

2021-12-07

Optimum Implementation of Compound Compression of a Computer Screen for Real-Time Transmission in Low Network Bandwidth Environments

Conor Paxton

Technological University Dublin, x00043062@mytudublin.ie

Follow this and additional works at: <https://arrow.tudublin.ie/engmas>



Part of the [Electrical and Computer Engineering Commons](#)

Recommended Citation

Paxton, C. (2021). Optimum Implementation of Compound Compression of a Computer Screen for Real-Time Transmission in Low Network Bandwidth Environments. Technological University Dublin. DOI: 10.21427/SYPS-P748

This Theses, Masters is brought to you for free and open access by the Engineering at ARROW@TU Dublin. It has been accepted for inclusion in Masters by an authorized administrator of ARROW@TU Dublin. For more information, please contact arrow.admin@tudublin.ie, aisling.coyne@tudublin.ie, vera.kilshaw@tudublin.ie.

***Optimum Implementation of Compound
Compression of a Computer Screen for Real-Time
Transmission in Low Network Bandwidth
Environments***

A Thesis Presented For the Award of Masters by Research by

Conor Paxton B.eng



Technological University Dublin – Tallaght Campus

Department of Electronic Engineering

For Research Carried Out Under the Guidance of

Mr. Richard Gahan

Submitted to Technological University Dublin

January 2021

DECLARATION

I certify that this thesis which I now submit for examination for the award of Masters by Research, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work.

This thesis was prepared according to the regulations for graduate study by research of the Technological University Dublin (TU Dublin) and has not been submitted in whole or in part for another award in any other third level institution.

The work reported on in this thesis conforms to the principles and requirements of the TU Dublin's guidelines for ethics in research.

TU Dublin has permission to keep, lend or copy this thesis in whole or in part, on condition that any such use of the material of the thesis be duly acknowledged.

Signature _____ Date _____

Candidate

Acknowledgements

I would like to thank my project supervisor Mr Richard Gahan for the help, guidance and confidence that he has given me. I would also like to thank Mr. Patrick O' Friel, Brian Keogh, James Wright and all from the Department of Electronic Engineering.

I would like to thank my family for believing in me.

I would especially like to thank Lisa Smyth for putting up with me.

Lastly, I would like to thank Dillon Waters, Michael King, Kai Wu and the rest of my peers from my undergraduate years.

Contents

1	Introduction	9
1.1	Background	9
1.2	Motivation	12
1.3	Research Question	13
2	Literature Review	14
3	Human Visual System and Colour	27
3.1	The Human Visual System	27
3.2	Colour Space	29
3.3	Chroma Sub Sampling	35
3.3.1	Conclusion on Chroma Subsampling	39
4	Compound Image	40
4.1	Image Data Classes	41
4.1.1	Smooth Class	42
4.1.2	Sparse Class	44
4.1.3	Text Class	46
4.1.4	Fuzzy Class	47
4.1.5	Picture Class	49
4.2	Conclusion On Compound Image	51
5	Classification	53
5.1	The Discrete Wavelet Transform	53
6	Compression	60
6.1	Lossless Compression	64
6.1.1	Run Length Encoding	64
6.1.2	LZ 77 Algorithm	67
6.1.3	Lempel Ziv Welch Encoding	68
6.1.4	Huffman Encoding	76
6.1.5	Arithmetic Encoding	83

6.2	Lossy Compression	87
6.2.1	Image Quality Assessment	87
6.2.2	Block Transform Encoding	92
6.2.3	The Discrete Cosine Transform	92
6.2.4	Comparing speed performance of 3 discrete cosine transform methods	100
6.2.5	Quantization	101
6.2.6	The Discrete Wavelet Transform	103
7	Compression File Formats	111
7.1	JPEG	111
7.2	PNG	114
8	Video Encoding	118
8.1	H.264	118
9	Compound Compression Algorithm	120
9.1	Classification	121
9.1.1	Comparing Presented Classification Algorithm to the Work presented by Wu [48]	142
9.1.2	Colour Counting Analysis	144
9.1.3	Conclusion on Compound Image Classification Using Discrete Wavelet Transform and Colour Counting Analysis	155
9.2	Lossless Compression	157
9.2.1	Smooth Block Compression	157
9.2.2	Lempel-Ziv Welch Compression	159
9.2.3	Deflate Compression	163
9.2.4	Conclusions on Deflate Compression on Compound Image Data Classified as Sparse and Text	167
9.2.5	Differential Index Map Coding	167
9.2.6	Conclusions on Compression Performance of Differential Index Map Coding	182
9.2.7	Compression and Speed Performance Comparison: Lempel-Ziv Welch Vs Differential Indel Map Coding Vs Deflate	183

9.2.8	Conclusion on Lossless Compression testing	188
9.3	Lossy Compression	189
9.3.1	Chroma Sub Sampling Testing	190
9.3.2	Comments on Chroma Sub Sampling Methods	193
9.3.3	Performing the Two-Dimensional Discrete Cosine Transform as Two One-Dimensional Transforms	194
9.3.4	Quantization of Discrete Cosine Transform Coefficients	195
9.3.5	Reordering Discrete Cosine Transform Coefficients	197
9.3.6	Performing Zero Run Length Encoding on Reordered Quantized Discrete Cosine Transform Coefficients	198
9.3.7	Testing Discrete Cosine Transform Based Compression on Blocks that have been Classified as Fuzzy and Picture	200
9.3.8	Conclusions on Fuzzy Block Quantization	202
9.3.9	Conclusions on Picture Block Quantization	204
9.3.10	Entropy Encoding Discrete Cosine Transform Based Compressed Data	205
9.3.11	Discrete Wavelet Transform based Compression: Blocks Clas- sified as Text with High Unique Pixel Count	207
9.3.12	Comments on Accuracy and Speed of Discrete Haar Wavelet Transform	209
9.3.13	Threshold Value Selection for Discrete Haar Wavelet Transform Compression	209
9.3.14	Encoding Non Zero Discrete Wavelet Transform Coefficients After Thresholding	210
9.3.15	Conclusions on Lossy Compression Testing	215
9.4	Algorithm Configuration	217
9.5	Results	220
9.6	Discussion	221
10	Results Summary	223
11	Conclusion	225
11.1	Future Work	226
12	Code	227

12.1 Data Structures	227
12.2 Classification Algorithm	229
12.3 Discrete Cosine Transform Code	236
12.4 Deflate Interface	240
12.5 Differential Index Map Coding Functions	242
12.5.1 Discrete Wavelet Transform Functions	252
12.6 OpenCV and FFMPEG Code	254
13 Appendix	255

Abstract

Remote working is becoming increasingly more prevalent in recent times. A large part of remote working involves sharing computer screens between servers and clients.

The image content that is presented when sharing computer screens consists of both natural camera captured image data as well as computer generated graphics and text. The attributes of natural camera captured image data differ greatly to the attributes of computer generated image data. An image containing a mixture of both natural camera captured image and computer generated image data is known as a compound image.

The research presented in this thesis focuses on the challenge of constructing a compound compression strategy to apply the ‘best fit’ compression algorithm for the mixed content found in a compound image. The research also involves analysis and classification of the types of data a given compound image may contain.

While researching optimal types of compression, consideration is given to the computational overhead of a given algorithm because the research is being developed for real time systems such as cloud computing services, where latency has a detrimental impact on end user experience.

The previous and current state of the art videos codec’s have been researched along many of the most current publishing’s from academia, to design and implement a novel approach to a low complexity compound compression algorithm that will be suitable for real time transmission.

The compound compression algorithm will utilise a mixture of lossless and lossy compression algorithms with parameters that can be used to control the performance of the algorithm.

An objective image quality assessment is needed to determine whether the proposed algorithm can produce an acceptable quality image after processing. Both traditional metrics such as Peak Signal to Noise Ratio will be used along with a new more modern approach specifically designed for compound images which is known as Structural Similarity Index will be used to define the quality of the decompressed Image.

In finishing, the compression strategy will be tested on a set of generated compound images. Using open source software, the same images will be compressed with the previous and current state of the art video codec’s to compare the three main metrics, compression ratio, computational complexity and objective image quality.

1 Introduction

1.1 Background

The basic components of a personal computer consist of hardware, software and peripherals for a user to interact with, such as a display, keyboard and a mouse. The user interacts with the computer, via peripherals, using applications written in a high level computing language which is then interpreted to a low level machine language. The operating system (OS), is a low level software that manages the underlying hardware resources through the hardware abstraction layer (HAL). The HAL provides drivers enabling the commands the operating system interprets from user applications, to run on the hardware. A high level overview of the system architecture for a personal computer is presented in figure 1.

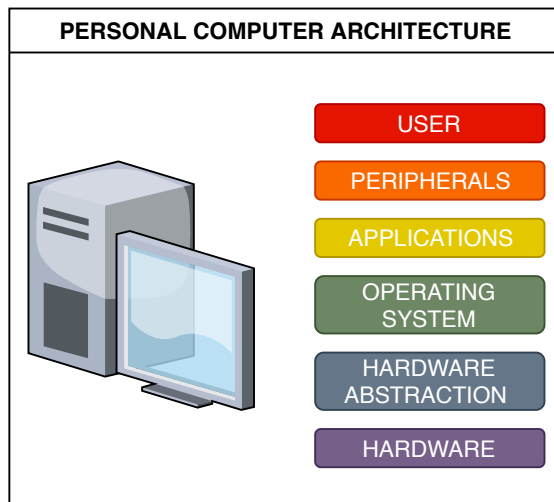


Figure 1: Basic Architecture of a Personal Computer

With continual growth in power and performance in microprocessor technology, even the most basic personal computer is powerful enough to run most software needed in large enterprises and institutions of higher education, without ever using all potential resources.

Virtual desktop infrastructure (VDI), refers to a system where a server will generate instances of the core software components that run on a computer such as the OS and applications and deploy it in a container known as a 'virtual machine'. An instance of a virtual machine can then be accessed remotely by a user, who can interact with it through peripherals, and work at it as though it were a personal computer. The virtual machines are controlled on the server by the hypervisor and share the resources of the server. There are many benefits to this type of deployment, some of which are:

- The users device that accesses the virtual machine, typically called a thin client,

does not need its own resources, as they are on the server side. This means that a low power device can be used which is generally much more energy efficient.

- The cost of hardware is reduced, as the thin clients do not need to be continuously upgraded, even as software becomes more resource intensive.
- Multiple instances of virtual machines can be generated, for deployment on different platforms using different operating systems and applications.
- Mobility is enhanced as the virtual machine can be accessed through the server from anywhere and work as though the client is at a local desktop.
- Security is enhanced as all data is stored centrally. Thin clients typically do not have a hard disk, so sensitive data can only be accessed through correct protocol
- Maintenance and software updates can be undertaken procedurally. All virtual machines running on the server can be up to date and coherent with each other, running the latest software revisions or, if needed regress to a more stable state, which means less down time for an enterprise.

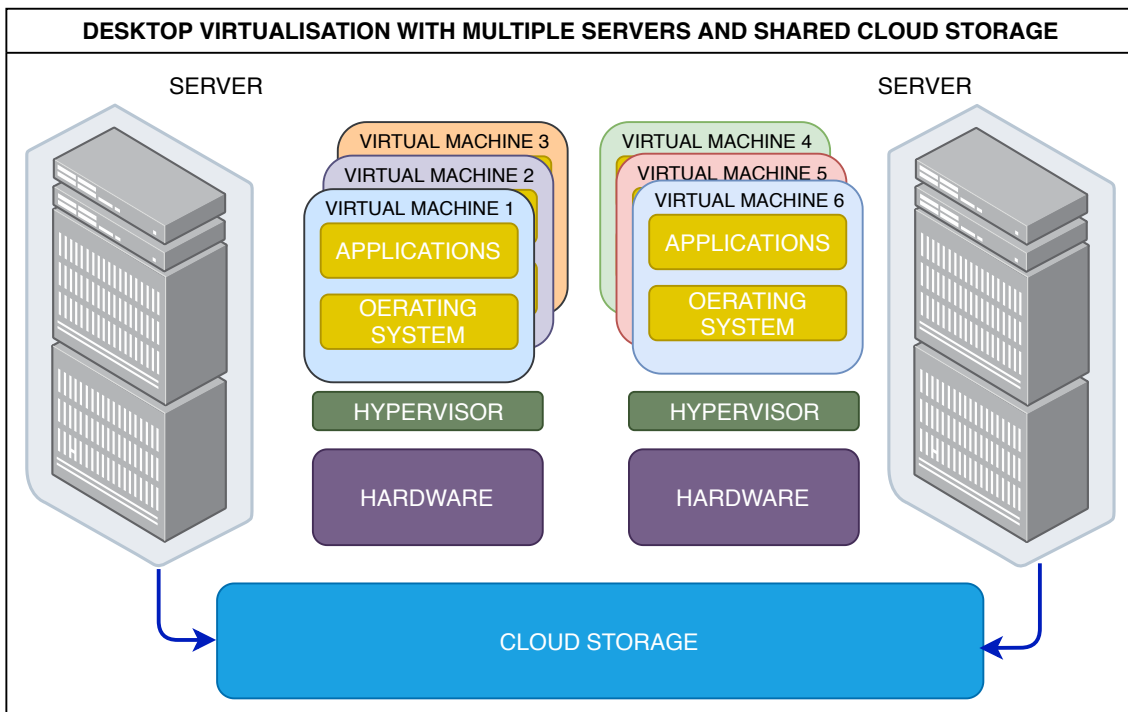


Figure 2: A Multiple Server Virtual Desktop Infrastructure Generating Virtual Machines on Several Operating Systems

Figure 2 Shows a high level overview of a virtual desktop infrastructure. As previously mentioned, the hypervisor takes control over the available resources and distributes

them among the instances of the virtual machines. The virtual machines can be moved from one host server to another, which will provide additional benefits in load balancing and the underlying resilience of the infrastructure. The virtual machine will still operate, irrespective of which server it is hosted from.

In figure 2, the different colours of the virtual machines represent different types of distributions which are tailored to the clients needs, depending on the type of software application that is needed. This has the benefit of provisioning resources efficiently, reducing overheads, such as licensing for specific software and power consumption.

The need for storage may be realised using network or cloud storage which has the added benefits of security as the data is centralised, machines can be deployed faster and the extra benefits of continual backups, in case of malicious attacks or accidents.

There are many challenges in the deployment of a successful virtual desktop infrastructure. Four of the main challenges are:

- User latency
- Transmission bandwidth
- complexity
- Subjective image quality of the decoded image on the user side.

As the desktop image is being generated on the server and sent to the user remotely over the network, this will generate latency for the user. If the latency is too high it will result in a negative impact for the user. The ITU-T recommendation G.114 [40] suggests that one way traffic from server to source should be in bounds of approximately 50 ms, to avoid negative impact with respect to latency for the end user.

The raw file size for a single image frame of resolution 1080p is 1920x1080 pixels is 6.2MB. at 30 frames per second, the data rate is 1.492Gbs which would not be possible to transmit on a band limited network.

There are some advanced compression methods such as High efficiency video encoding (HEVC) [39] That can achieve good compression, however rely on intensive processing, which may not be suitable for low powered simple thin client systems.

To reduce the file size before transmission, the file must be compressed on the server side and decompressed on the client side. This process will introduce latency, as it will take time to compress and decompress. It may introduce distortion in the decompressed image on the client side, if the compression that is used is a lossy type of compression.

1.2 Motivation

A survey from The International Data Group on cloud computing [17], states that in 2020, over 81% of medium to large enterprises have adopted cloud based solutions for computing and software applications, which is up from 73% in 2018. This shows signs of a steady continued growth in this sector.

From the 2020 Cisco Global Networking Trends Report [5], video data will account for over 82% of business internet traffic.

With recent advancements in screen technology, Ultra High Definition (4K) displays are becoming more popular in the consumer market. However, the raw bit rate for a 4096x2160 display, with refresh rate of 30FPS, and 24bit 4:4:4 colour depth, is 8.91Gbps. This would need significantly larger storage and would require extensive bandwidth to transmit, which may not fit the constraints of a bandwidth limited network such as the internet.

Remote desktop services and virtual desktop infrastructure are a mature technology. However, in the authors experience, the deployment of such technologies in areas such as institutions of higher education, are limited. This may be due to a number of reasons, one of which may be the cost of licences per "seat" at a virtual machine. Companies such as Citrix offer virtual desktop infrastructure and remote desktop applications with excellent results for the end user in terms of both quality of image and low in latency. However, such companies use proprietary software and licensing agreements with companies such as Microsoft for using their Remote Desktop Protocol(RDP).

The benefits of using Remote Desktop Protocol is that the screen rendering on the decoder side has prior knowledge of elements of the Window Application Programming Interface (API) and can generate platform specific graphical user interface (GUI) elements, reducing the need to transmit as much data across the network. As previously mentioned, using this technology requires a paid licence, which could prove infeasible for a given institution or enterprise.

Another disadvantage of using proprietary software, is that it may restrict the virtualisation to a limited amount of operating systems. For instance, in a given institution, there may be a need to use applications specific to Microsoft Windows, while another department may need access to Macintosh operating system applications, while another may want to use any number of Linux distributions.

An alternative to the above may be use a system based on Virtual Network Protocol (VNC) [34]. VNC is a display based application allowing remote display of a rendered screen using a Remote Frame Buffer (RFB) protocol, where the only data transmitted from the user side is mouse and keyboard data. VNC has no prior information of the rendered screen image to be decoded, it works by encoding a frame and transmitting a series of difference frames over time, which are updated regularly. It treats the

rendered image as a whole bitmap and updates the areas that have changed, as the user is working with the application. An advantage of using VNC is that it is totally platform agnostic, allowing the user to work on a multitude of platforms, only limited to what the server is able to virtualize. There are companies who deliver VNC technology through software as service (SAAS), such as RealVNC which require a paid subscription to use the service.

The motivation for this research project is to research and design a compound image compression algorithm and compare it to known solutions using image quality assessment metrics. The goal is to research and use compression algorithms that use unrestricted licensing and open source software so that the work done in this project can be given back to the community, without obstruction from licensing agreements.

1.3 Research Question

This research proposes to answer the following questions:

- 1 Can the data contained in a compound image, such as the data typically displayed on a computer screen, be classified accurately based on specific attributes that would be suited for specific types of compression.
- 2 Can a compound compression and decompression algorithm be created that is capable of compressing the mixed content of a compound image effectively, to meet band limited network constraints.
- 3 Will the proposed algorithm be suitable for real time services such as a virtual desktop infrastructure.
- 4 Will the decoded image after processing be of an acceptable subjective and objective quality. To determine the metrics that can be used to define if an image is of acceptable quality.
- 5 Will the proposed compound compression algorithm be comparable to the current state of the art.
- 6 Will the proposed compound compression algorithm be free from restrictive licensing dependencies.

2 Literature Review

There are multiple frameworks in which one could implement a compound compression strategy. A common theme is to first implement a classification algorithm that can identify different types of data in an image, such as computer generated graphics, text and continuous tone image. The classification algorithm will identify areas of an image containing specific types of data based on attributes such as sharp transitions in smooth areas for computer generated data or low gradient change among neighbouring pixels for continuous tone image. The location of the different types of data found within a compound image is stored and used in the compression stage to apply optimised compression algorithms for a given area of the image containing a specific type of data. The three most notable approaches to classification are object-based, layer-based and block-based classification.

Object-based classification extracts objects of an image exactly along their boundary, of any arbitrary shape, using complex image processing techniques such as edge detection. These objects could be compressed using an optimized compression algorithm for a given type of object. This type of classification is the most computationally intensive and may not be suitable for time critical applications.

de Queiroz [6] describes a Mixed Raster Content (MRC) layer based classification and compression framework, which is used in pdf type compressors such as djVU [2]. This method splits an image into multiple rectangular layers, namely background, foreground and binary mask and applies different types of compression to each layer.

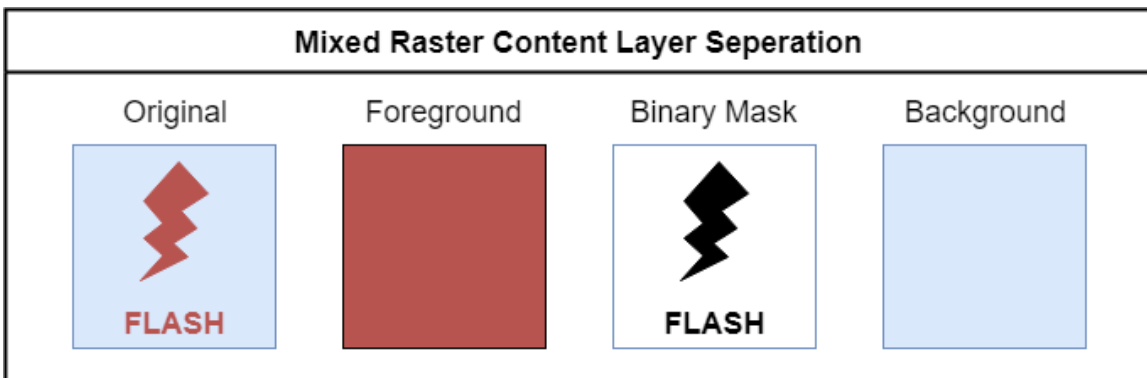


Figure 3: A Simple Image Split Into the three Layer Mixed Raster Content Model

Figure 3 shows a simple image split into 3 layers. In practice an image can be split into any multiple of these three layers, depending on the amount of objects in the image. The binary mask layer is what is used to tell which pixels from what layer are the final image.

In [2] the background and foreground layer are compressed using a proprietary discrete wavelet transform, the IW44, which can efficiently compress the data in a lossy way,

with little discernible impact with respect to subjective image quality. The binary mask layer is compressed with a variant of JBIG2 [21] called JB2, which compresses the data using a type of arithmetic coding at a much higher resolution as to preserve the quality of text and graphic information.

Crucial to this approach is the segmentation process, which is based on a K-means clustering algorithm. While this works very well on images that could be considered low in complexity, with fewer colours and lack of continuous tone image (such as non camera captured images), *Said et al* [36] note that certain objects can get mismatched in the layers (text over image, or natural image with sharp transients). This can lead to poor compression performance for a given layer. *Ding et al* [9] also note that the complexity of generating the mask layer, and implementation of the IW44 transform make it unsuitable for time critical applications and therefore, this approach is not pursued in this thesis.

In their work on a **”Simplified Segmentation for Compound Image Compression”**, *Said, et al* [36], implement a block based classification scheme that will compress blocks with text with jpeg-LS variant and blocks containing picture with lossy jpeg variant. They note while block-based classification may not be as successful at classifying as object based classification, due to all pixels in a block being classified as the same type, even if object boundaries occur within a block, the effects can be mitigated by using a small enough block size. This type of classification and compression is well suited for efficiency and for using multiple standard compression algorithms together to form a compound compression algorithm.

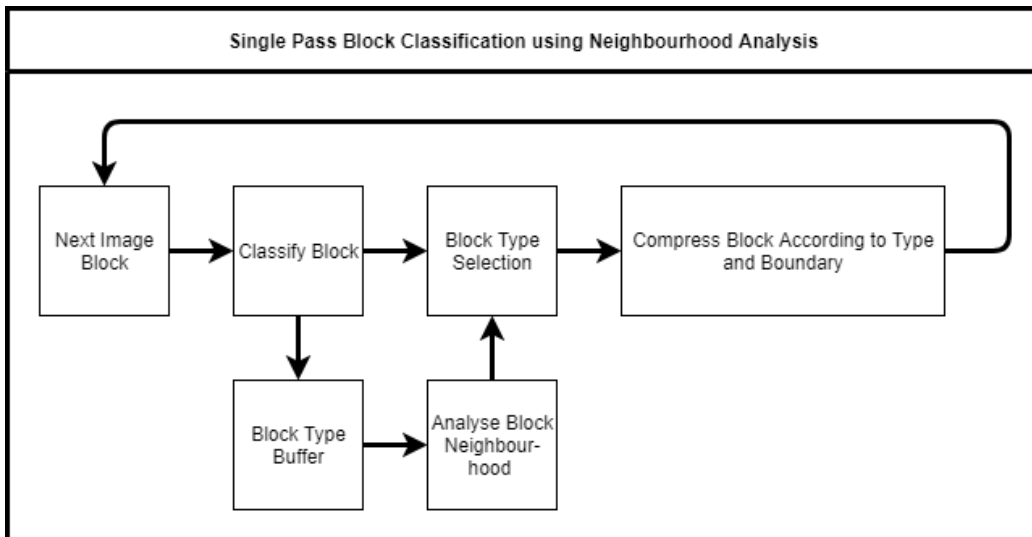


Figure 4: Classification and Compression process, Said et al [36]

Figure 4 shows the implementation in *Said et al* [36]. The literature does not give the specifics on exactly how the classification is performed on the block itself, simply stating a block is classified due to its distribution of pixels. However, the framework

shows that the algorithm also takes into account neighbouring blocks when deciding on what type the current block is. This can be beneficial in helping distinguishing complex pictorial blocks from text/graphic type blocks.

From their research on a "**Block-Based Fast Compression for Compound Images**" [9] Ding et al develop a block-based strategy to to further classify a compound image into more than just text and image blocks, as in *Said et al's* work.

Ding notes that large areas of a compound image, such as a computer screen is constituted by smooth, single colour areas as well as text, graphic and pictorial data. As such, the compound image is to be classified into four distinct types of data blocks: smooth, text, picture and hybrid blocks. The attributes of each type of block are defined by the gradient histogram of each type of block, where the gradient is defined as the rate of change between neighbouring pixels.

The pixels of a block are first grouped into three classes, low-gradient, medium-gradient and high gradient-pixels. *Ding* notes that smooth blocks will only contain low gradient pixels and show a single peak on the low-pixel histogram, while text blocks will contain several peaks at the low and high gradient histograms. Picture blocks contain mainly mid-gradient pixels, while hybrid blocks (which can contain both natural image and text, or sharp transitions), contain a mix of both hi-gradient and mid-gradient peaks.

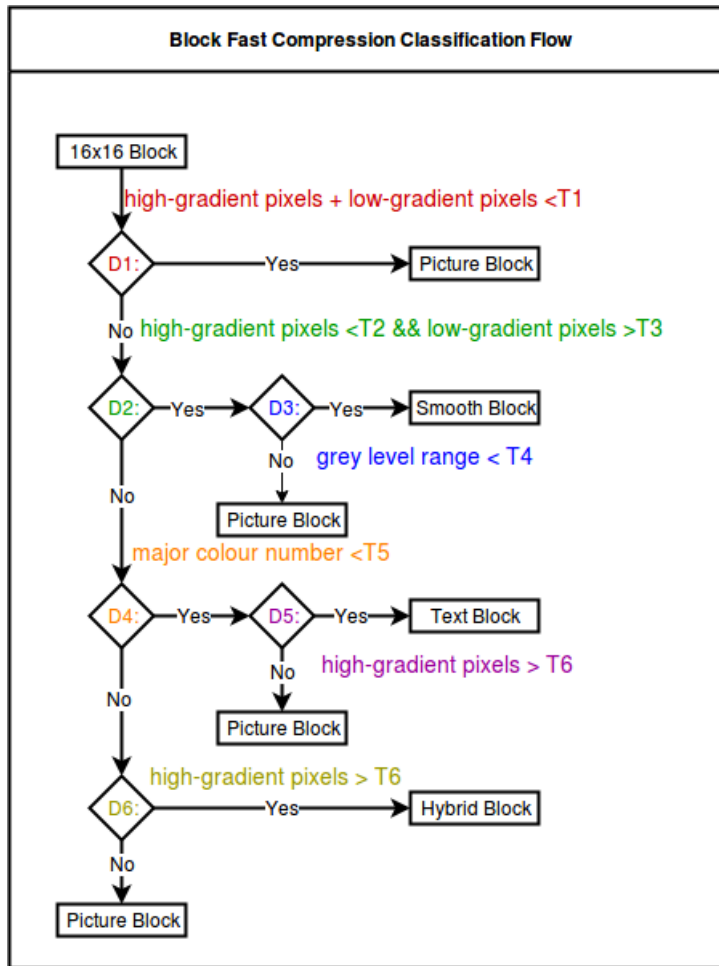


Figure 5: Process flow of the classification used in Block Fast Compression Algorithm

Figure 5 shows the classification process in *Ding's* Block Fast Compression algorithm. The "T" values are threshold values that have been chosen through empirical testing.

Ding chooses a unique compression algorithm for each type of block. Smooth Blocks are defined by having a single dominant colour, so all colours in a smooth block are quantized to the dominant colour, which is coded with an arithmetic coder.

A Jpeg-like encoding is used for picture blocks, as it is a proven efficient coder for natural continuous tone image.

Ding Notes that text blocks contain several dominant colours which are needed to ensure the textual characters are readable. *Ding* chooses up to 4 dominant colours from each block, the colours close to the dominant colours within a given threshold distance, are quantized to that dominant colour. After quantization, the pixels are converted to an index between 0-4, where 0,1,2 and 3 are the dominant colours and the rest of the pixels that are outside of the thresholds are encoded with 4. The indexed pixels are compressed in a raster scan order, where the current pixel, X in Figure 6 can have up to 5 different neighbouring values. As there are 3 indexes associated for

each pixel, there are 125 different contexts that can be encoded by the arithmetic encoder. The indexes are encoded and if the index value is 4, then the pixel values are encoded instead. This is a good approach, however, due to the raster scan order

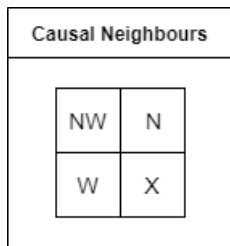


Figure 6: The causal pixel neighbourhood for encoding indexed pixels in Block Fast Compression Algorithm

encoding, it does not fully take advantage of the spatial correlation of pixels in text blocks have in both the horizontal and vertical direction, which is developed in [23] and further again in the research presented in this thesis.

Finally, the hybrid blocks of an image are compressed with a discrete wavelet transform coder, using a Haar transform. *Ding* notes that discrete cosine transform coding used in JPEG [18] work well on compacting the energy in low frequency signals, such as the slowly varying pixel values of natural image. As hybrid blocks can contain sharp transients, this can lead to poor compression performance and ringing artefacts, if transformed with a DCT. Using short wavelet bases such as the Haar wavelet provides is useful in reducing ringing artefacts for blocks that contain high frequency information and gives good compression performance, at the expense of poor compression performance for the low frequency information. *Ding* shows in his research that, when using the Haar transform for hybrid blocks and the discrete cosine transform for picture blocks, the resulting image shows a significant improvement in terms of visual quality and PSNR compared to using a discrete cosine transform alone.

Ding's research on text block coding can be seen as the basis for an efficient alternative in compressing compound images. In his work "**Enable Efficient Compound Image Compression in H.264 AVC Intra Coding**" [10], *Ding* develops the Base Colour and Index Map (BCIM) Compression algorithm and incorporates it into the H.264 video codec framework. The benefits of implementing BCIM as a new method in H.264 intra mode are two fold: The first is the algorithm uses less computation compared to performing a discrete cosine transform, thus is more efficient for time critical applications. The second benefit is the use of the "Rate Distortion Optimisation" algorithm in the H.264 framework. This algorithm is a good choice at discerning which blocks should be compressed with the Base Colour and Index map, or the other standard methods in H.264.

In Base Colour and Index Map, as in the previous incarnation, the dominant colours of a text block are used to create an index map and the index map is then encoded

using context adaptive binary arithmetic coding.

For this, Ding uses a two pass quantization technique. The first pass is a local quantization to cluster the pixel values into groups, which will give a more defined structure to the index map. The second step is to further quantize the pixels to form the dominant or base colours. there can be up to 8 base colours used per block.

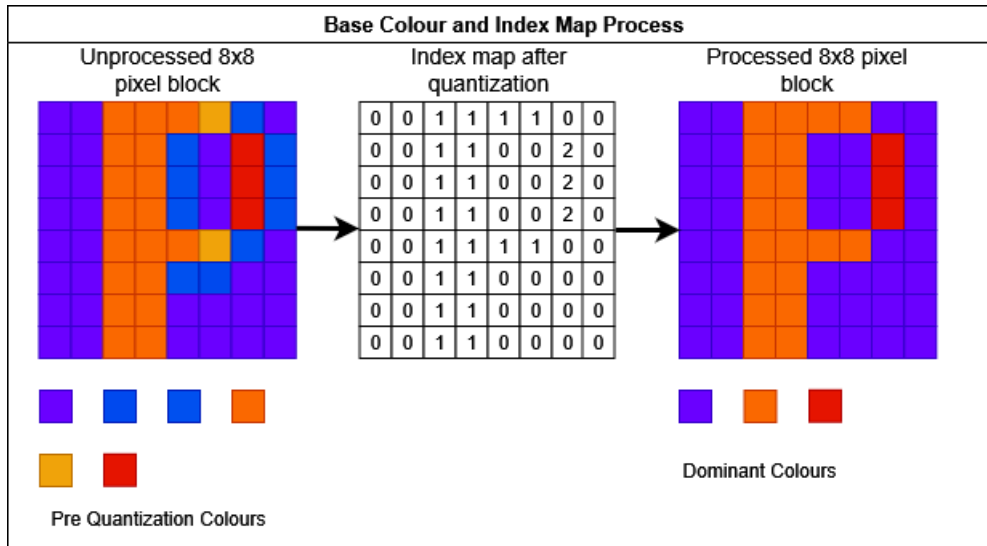


Figure 7: Generating the index Map from an 8x8 block of pixels, using Base Colour and Index Map Algorithm

The generation of an index map from a block of pixels can be observed in figure 7. The index map is compressed in a raster scan order using the Context Adaptive Binary Arithmetic Coding (CABAC) of H.264. Ding has shown impressive results with the implementation of this algorithm, with an improvement in Peak Signal to Noise Ratio (PSNR) and bit-rate (bpp) over traditional H.264 for compound image compression. However, by compressing the blocks in a raster scan order, Ding has not fully exploited the correlation of the index map in both the vertical and horizontal position, which is a trait that will be developed in the research presented in this work.

Like *Ding, Lan et al* [23] take the approach of exploiting the H.264 framework for compound image compression in their work in **”Compress Compound Images in H.264 by Fully Exploiting Spatial Correlation”**, [23]. Their approach incorporates Base Colour and Index Map coding along with another new method called **”Residual Scalar Quantization”** for intra-mode coding. The residual scalar quantization method is based on the intra-prediction method inherent in H.264. However in H.264, boundary samples of a block of pixels are first low passed filtered before prediction, because this method will be applied to areas of the screen that may contain pixel values with lower correlation such as window borders, or graphic areas, the values are directly used for prediction. Prediction involves subtracting the pixel values of the current block from a block in its neighbourhood to produce whats called

a residual. After prediction, the residual is directly quantized and encoded.

Lan et al show impressive results for their work and note a significant improvement in Peak Signal to Noise Ratio, compared to standard H.264, as much as a 10dB, however, this comes at a cost of added complexity to the algorithm.

The complexity of H.264 for classification and processing of compound images is also noted by *Juliet* in his work "**Efficient Block Prediction-Based Coding of Computer Screen Images with Precise Block Classification**" [22]. In his work, Juliet proposes a single-pass classification algorithm that classifies an image into text/graphic blocks or picture/background blocks. *Juliet* successfully accomplishes this by using the statistical properties of a block of pixels that has been transformed using a Discrete Wavelet Transform.

Juliet implements a block based approach, where a given compound image is segmented into 8x8 non overlapping blocks. A discrete wavelet transform is performed on a single channel per block to decompose it into four sub-bands. The sub-bands are essentially filter banks that can be used to determine specific properties of the block based on statistical calculations.

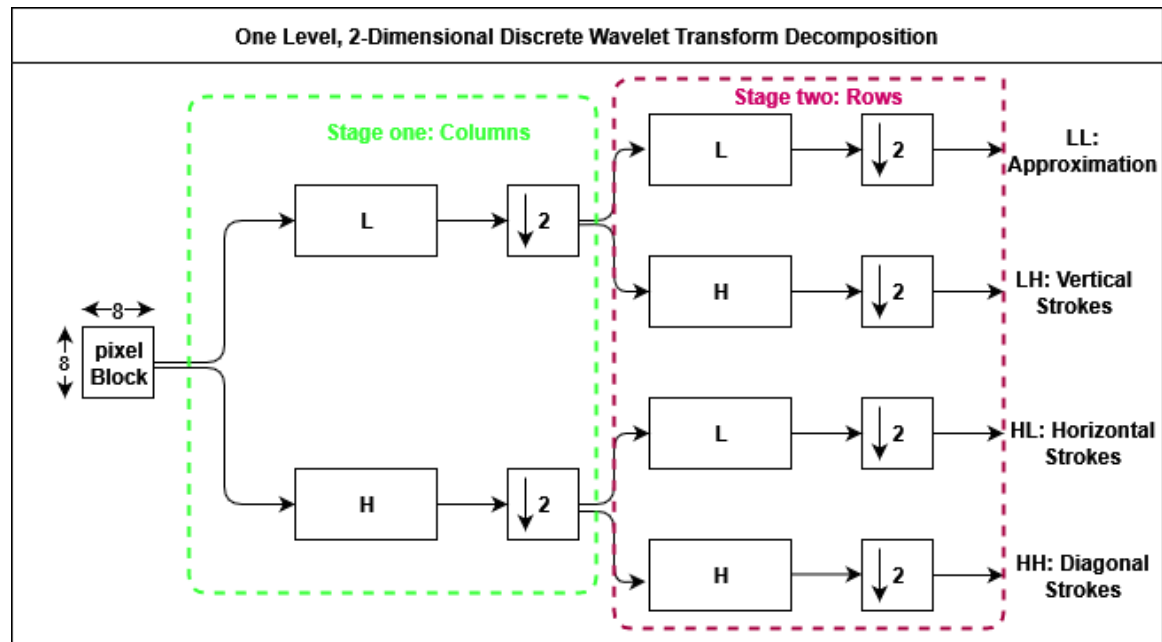


Figure 8: Decomposing an 8x8 pixel block into four sub bands using one level Discrete Wavelet Transform

Figure 8 shows the decomposition process using a discrete wavelet transform. There are two stages: the first stage filters the block into low and high pass bands and down samples the block by a factor of two. The low-pass coefficients contain an approximation of the original image, while the high-pass coefficients contain the details. The second stage applies the same technique along the rows which effectively decomposes

the block into four sub-bands. *Juliet* notes that text characters are dominated by strokes in 3 positions, vertical, horizontal and diagonal and after decomposition, the sub-bands LH,HL,HH can be used to identify these directional characteristics respectively. The discrete wavelet transform is a powerful tool and will be discussed in greater detail further on in this Thesis

After decomposition, *Juliet* calculates the standard deviation of each sub-band. If the value is above a given threshold in either of the LH,HL,HH sub-bands, the block is classified as a text/graphics block, else it is classified as a picture/background block.

Text blocks are coded losslessly by first using a prediction method, where the residual block is coded using a Huffman coder, while the picture blocks are coded using a JPEG style encoding. *Juliet* compares his work to JPEG, JPEG2000 and H.264 intra-mode coding and shows gains in both Peak Signal to Noise Ratio and bit-rate, without an increase in computational complexity.

Wu [48] builds upon the classification algorithm presented by *Juliet* in his work "**Block Based Classification Method for Computer Screen Images**". *Wu* further classifies the types of blocks within a compound image, into five categories: *Smooth, Sparse, Text, Fuzzy and Picture*. The first 3 block types contain computer generated data, whilst the last two typically contain continuous tone natural camera captured data.

The distinction between blocks classified as sparse and as text is within the structure contained within the block. A sparse block is a highly structured block that contains few colours and highly defined areas, which would typically be elements of a graphical user interface, such as window boxes, etc. The distinction between blocks classified as fuzzy and blocks classified as picture is the texture characteristics of the the block. Blocks classified as fuzzy contain slowly varying pixel values that are similar, with no discernible pattern. An example would be a block taken from a picture of the sky or grass or sand, etc. A block classified as a picture block may be more structured, with higher localised variance among the pixel values.

The classification algorithm presented by *Wu* implements a discrete wavelet transform to classify the blocks of a compound image. To help improve the speed performance of the classification process by reducing the amount of times the discrete wavelet transform is performed, two prediction strategies are implemented. *Wu* implements a prediction strategy for text blocks and a prediction strategy for smooth blocks. The prediction strategy for smooth blocks checks if the block directly to the left and the block directly above a given block have been classified as a smooth block. If the condition is true, a given block can be predicted as a smooth block. The same strategy is used for text blocks. To negate the propagation of error due to a given block being misclassified, a lattice approach is taken, where every second block can only be predicted. To ensure a predicted block is accurate, a refined colour counting approach is then applied to a predicted block. A single row and column is selected in

the block and the unique pixel values are counted. If the amount of unique colours is below an empirically tested threshold value, the prediction is a success. If the unique values are above a given threshold value, the discrete wavelet transform classification process will be applied to the block.

The classification algorithm presented in the research by *Wu* has been shown to be highly accurate and is the basis of the classification implemented in this thesis. The classification stage of the compound compression presented in this research is compared to the work presented by *Wu* further on in this Thesis.

Wang et al, propose a compound compression strategy which uses dictionary based coding and the H.264 framework in their work "**Compound Image Compression Based on unified LZ and Hybrid Coding**" [45]. Their work leverages the ability of the GZIP compression algorithm to efficiently compress repetitive patterns inherent in computer generated image data and H.264 efficiency at compressing natural image.

Central to GZIP is the Deflate algorithm [7], which uses Lempel-Ziv 77 (LZ77) dictionary coding [51] and Huffman coding [16]. The Deflate algorithm is also used in the lossless image compression format Portable Network Graphics (PNG), however it is performed in a raster scan order over the full row of an image. As *Wang et al* are using a block based approach, they modify the scanning procedure to work on blocks instead of rows. They note from empirical testing, greater compression gains are achieved by compressing in a "packed" pixel format, rather than each colour channel separately.

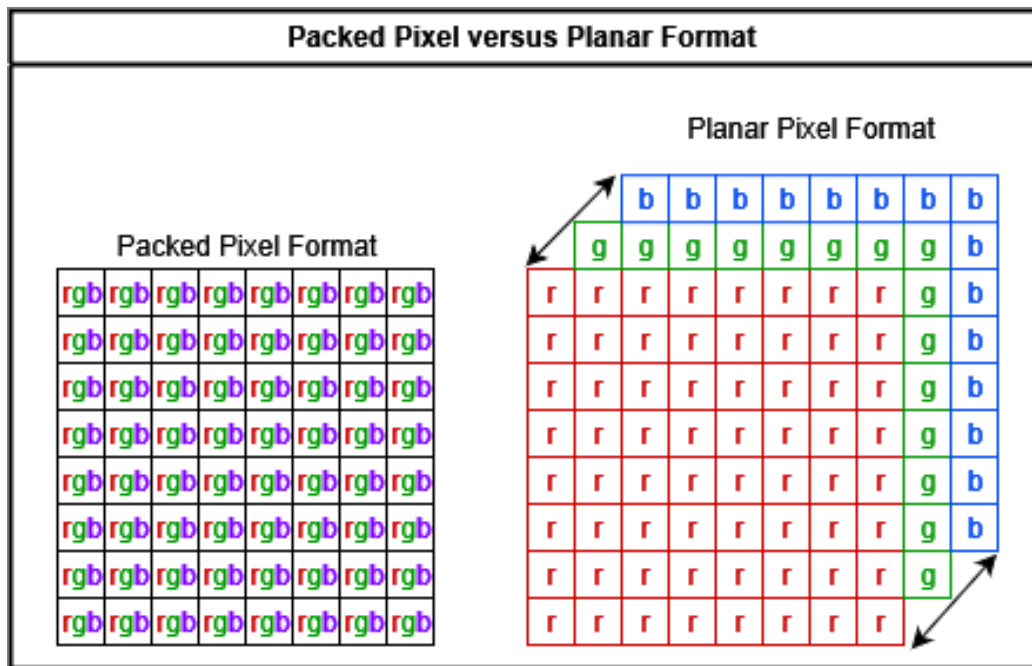


Figure 9: Showing the difference between a packed pixel format versus a planar pixel format

Figure 9 illustrates the difference between a packed pixel format and a planar pixel format. *Wang et al* also modify the scan direction, based on their empirical testing, they found that processing a block column-wise rather than row-wise improves compression performance.

Wang et al note the computational overhead of GZIP-like coding is a fraction of the computational complexity of H.264, so for their proposed method they choose to simultaneously compress each block with both GZIP-like coding and H.264 coding and use a metric based on rate-distortion optimisation to choose between the output of each method to encode in the final bit stream. However, as the GZIP-like coding is lossless, the distortion will be zero, so they modify the algorithm to accommodate for this.

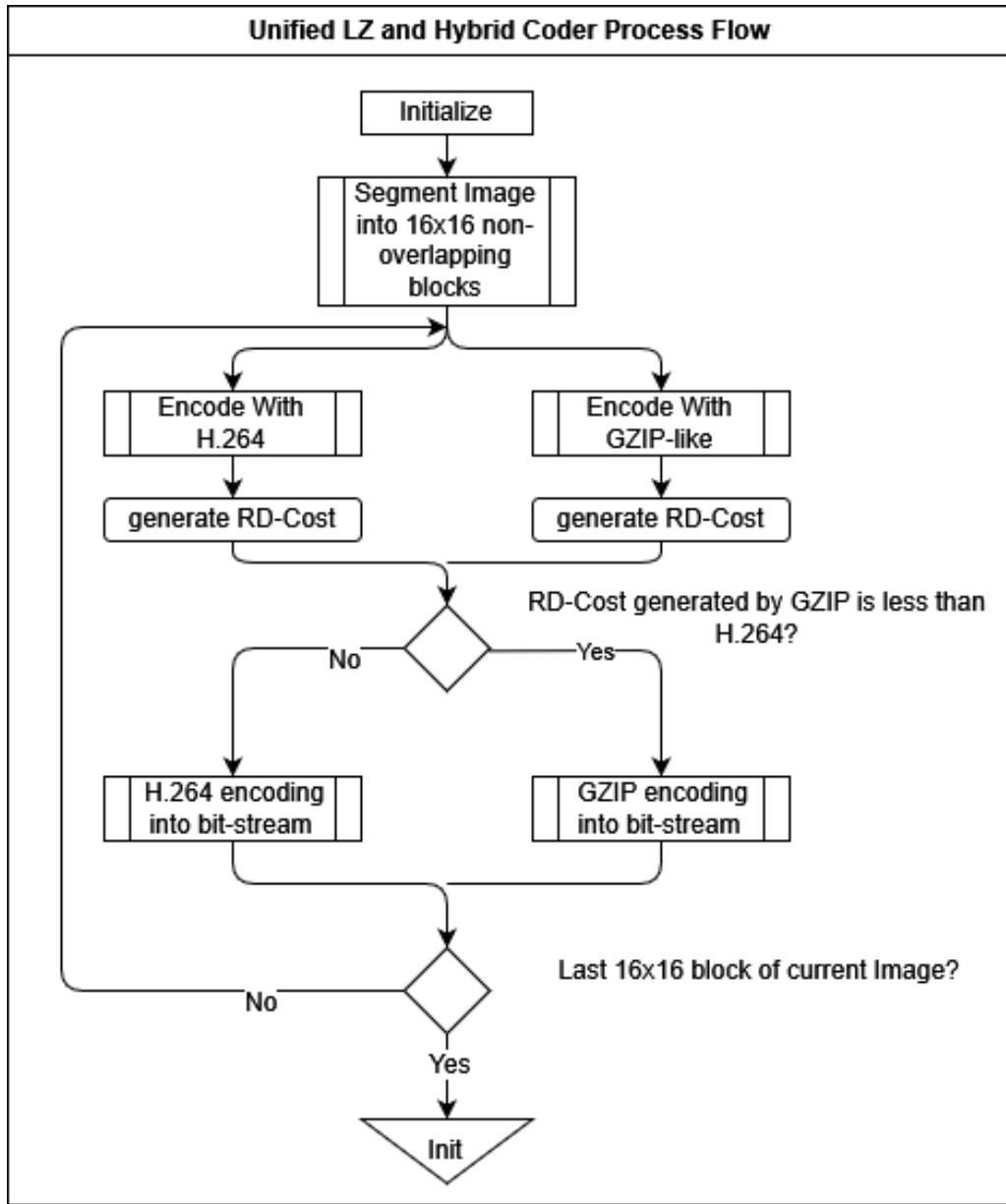


Figure 10: Overview of the Unified LZ and Hybrid Coding framework

Figure 10 shows an overview of *Wang et al* algorithm. They show impressive gains in both compression performance and Peak signal to noise ratio over standard H.264 and GZIP, however their strategy does have added computational complexity by simultaneously using both H.264 and GZIP-like compression. As they have noted in their research, the computational complexity of GZIP is a fraction of H.264, if They chose an approach that could analyse a given block before the compression stage, they could reduce the computational complexity greatly by only using a single compression algorithm per block.

Chen et al examine the benefits of using multiple types of transforms in their work **A Staircase Transform Coding Scheme for Screen Content Coding** [4]. In

their work they note that modern video codec's make use of prediction coding, where the transform is performed on the residual of a prediction. However the prediction residuals for screen content may vary dramatically from the residual of natural tone image. The prediction residuals for screen content can take the form of impulse or staircase signals because of the sharp transitions due to text and graphical content in smooth regions. They note that The Discrete Cosine Transform basis functions cannot accurately represent these staircase signals and so look to other types of transforms that can accurately and efficiently represent these basis. The staircase transforms used in this research are the Haar transform and the Walsh-Hadamard transform, both of which are actually discrete wavelet transforms.

Conclusion

There are three clear goals of a successful compound image compression strategy that would be used in modern day cloud services such as virtual desktop infrastructure and Desktop as a service.

- The system must be able to efficiently compress the data such that it meets the network bandwidth requirements.
- The system must be low in computational complexity. Using less resources enables a greater amount of instances that can be used by end users.
- Decoding should be less computationally expensive than the encoding process, such that it can be performed by devices with limited resources, limiting the amount of latency for the end user to an acceptable level.
- The system must be able to produce an acceptable output image in terms of both objective and subjective quality, such that the end user is fully able to discern the data within the image, free from obstruction.

It is clear from the material presented in this literature review that no single compression strategy fully succeeds in realising the three goals simultaneously, so a certain balance needs to be struck. It is then even more important to fully understand the data within a compound image such that the overall system can be optimised to a good balance.

While block-based compression strategies may not be as optimal in classification as object-based classification, they make up for this in their efficiency and ability to incorporate proven compression techniques such as JPEG style compression based on DCT coding for natural image. But there are attributes of the compound image such as text, smooth regions and graphics that are not suitable for transform coding and if compressed by these methods, will introduce distortion artefacts, degrading the

quality of the output image, as well as incurring additional latency to the end user, due to calculation of the transform for each block.

A block-based compression strategy has the ability to incorporate other compression algorithms more suited to these types of computer generated data. However, this gives rise to another challenge, that is, how to choose which type of compression to apply to each block?

Using the pixel histogram and gradient is a good method but requires three histograms, along with a clustering algorithm. Using the Rate Distortion Optimisation algorithm in H.254 and HEVC requires multiple passes through the data. Using a Wavelet transform on a single channel to find directional information can be done in a single pass and achieve accurate results, however, it still has to be performed on every block.

A good balance between efficiency and accuracy for classification would be to find attributes of the different types of data that are in a block that can be gathered without intensive processing. One attribute that stands out is the count of unique pixel values per block. Data in a compound image that is computer generated data is typically text, structural information, such as elements of the graphical user interface and smooth regions such as the background of text editors or back drops to applications. For an end user to be able to get the information from computer generated data, the information has to be clear and free from obfuscation. The hypothesis that there is a correlation between blocks containing computer generated data and having a low pixel count will be explored in this thesis, to see if it is a good metric for classification. Counting the unique pixels in a block is radically less computationally complex, compared to performing mathematically intensive discrete transforms, so if unique pixel counting can be shown to be an accurate metric for classification, the algorithm should receive a significant enhancement in speed performance, resulting in a more efficient classification algorithm.

3 Human Visual System and Colour

3.1 The Human Visual System

This sections aims to give a brief overview of the structure and attributes of the human visual system and how it plays a central role in the implementation of image processing techniques.

The cornea is a transparent and durable tissue at the front of the eye which covers the pupil, iris and anterior chamber. As well as providing protection, it is responsible for refracting light in through the pupil onto the retina.

The lens is responsible for absorbing close to 8% of the visible light spectrum and has higher absorption functionality with shorter wavelengths. It focuses light on the retina and by manipulating its shape, it controls the focal distance of the eye so that it can focus on a given object at a specific distance. This allows for a clear image of a given object to be formed on the retina.

Figure 11 shows a cross sectional view of the human eye. It is comprised of many sections:

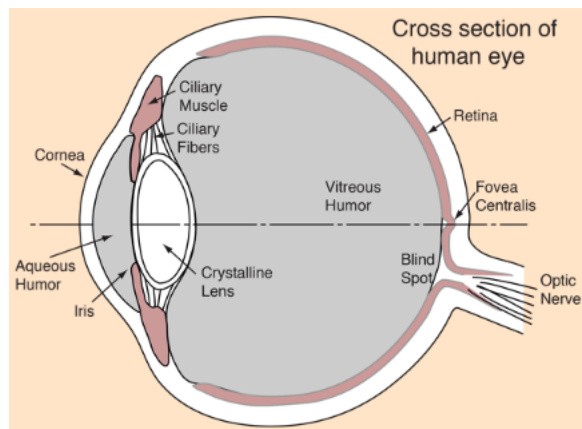


Figure 11: Cross-Sectional view of the human eye structure [42]

The retina lines the rear wall of the eye. When in focus, light reflected from an object outside of the eye is imaged on to it. The ability to discern patterns from the reflected light is due to fields of discrete light receptors which cover the surface of the retina. There are two unique types of light receptors known as retinal rods and cones.

The cones of the human eye are clustered around the centre of the retina in an area known as the fovea. Cones are highly sensitive to colour and because each cone is attached to its own individual nerve ending, the human visual system is able to discern a wide palette of colours. There are three defined types of retinal cones: Long, Medium and Short. Each type of cone is sensitive to a particular wavelength band: 650nm[long], 510nm[Medium] and 475nm[short] and the human visual system

interprets these bands as red, green and blue (trichromatic colour vision) respectively [30]. The type of vision produced by cones is known as photopic vision [3].

There are up to 7 million cones in the retina which is comparatively smaller than the 75 - 150 million rods [14]. Rods differ to cones in multiple ways: they take up more surface area than cones and multiple rods are connected to the same nerve ending. Rods are not used in discerning colour information but are very sensitive to slight variation in luminance levels. Rods produce a general overview of the scene imaged to the retina and can be stimulated in low luminance levels when cones are not stimulated. An example of rod stimulation would be an objects outline being visible by moonlight without colour information, while in day light, colour information is clearly visible. This type of vision is known as scopic vision [14].

The human visual system's higher sensitivity to light allows it to extract structural information from an image. This is an important attribute that should be taken into consideration when applying compression to image data. By separating an image into light data (luminance) and colour data (chrominance), an image can be processed by preserving structural information about the image in the luminance channel, while discarding some information in the chrominance channel in favour of smaller image file size. Distortion is a penalty incurred by discarding some of the original data, however as the human visual system is less sensitive to colour information, the reduction in data may not have as great an impact on the end viewer. This is done to reduce the file size of the image to meet transport and storage requirements. This technique is exploited by most modern lossy compression codecs, such as the JPEG standard [18], H.264 [47] and HEVC [39]

The human visual systems ability to extract structural information from an image also has to be considered when applying compression algorithms to non camera captured images. Any distortion to structural information, which is inherent in computer generated images, can impact negatively on the end user experience. However, there are compression algorithms that can efficiently compress structural information, free from distortion, such as PNG [13].

The design of a compound compression algorithm should take into account the attributes of the human visual system and exploit them whenever necessary, to provide a processed image that is free from user discernible distortion, while the size of the compressed file fits storage and network requirements.

To gauge success of a compression algorithm, metrics are needed. Common metrics used in image processing are root mean square error (RMSE) and peak signal to noise ratio (PSNR). Both metrics are a measure of the error in a signal from original image to processed image and do not take into account the human visual system. However, a metric known as structural similarity index (SSIM) [50] which is based upon the attributes of the human visual system can be used as a good image quality assessment metric and will be used as a metric for the work presented in this thesis.

3.2 Colour Space

The perception of light depends on three main properties:

1. Intensity, which can be defined as the total energy within the light.
2. Colour, which can be defined as the fundamental frequency of the light.
3. Saturation, which can be described as how close a light source appears to be a pure spectral colour.

The term luminance is used to refer to the intensity of light within digital images, while the term chrominance is used to collectively refer to the colour and saturation within an image.

To represent an image digitally, there needs to be an agreed upon method to interpret the chrominance and luminance information using discrete values. There are many models that can be used to represent this information and the model that is chosen will be defined as the colour space of the medium.

There are many models that can be used for the colour space. A popular model used in computing is the RGB model.

RGB Colour Model

The RGB model is an additive model which decomposes a given colour into three primary components: Red, Green and Blue. By varying the level of intensity of each component, the full colour spectrum can be represented, which can be compared to a spatial vector which can be defined about three axis x,y,z . By normalising the Red, Green and Blue components to the unit vector, all definable colours are constrained in a unit cube, in which any given discrete colour can be directly translated into three positive integer values. The unit vector RGB colour cube can be seen in figure [12](#).

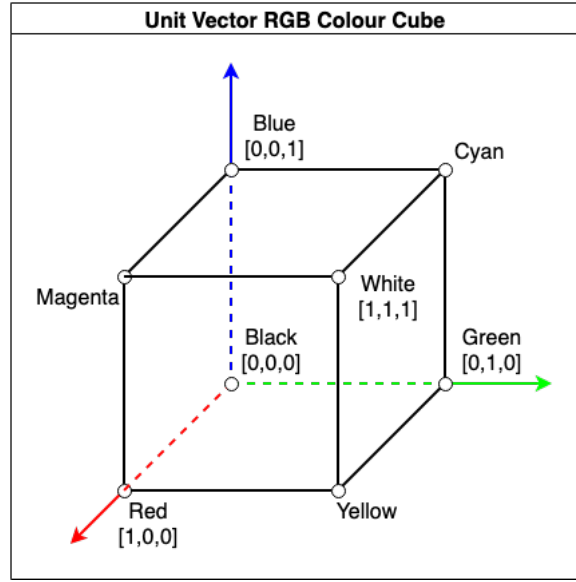


Figure 12: The unit vector RGB colour cube

Using the colour cube defined in figure 12, a colour C can be defined as:

$$C = \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Where

$$0 \leq R, G, B \leq 1$$

The contribution of intensity for each colour is denoted with r, g, b and so the colour C can be defined as

$$C = (rR + gG + bB) \quad (1)$$

The process of adding colours together, which is known as blending has to satisfy certain constraints. Just as adding two points in space is prohibited, a linear combination of two discrete colours is only true if the sum of coefficients is equal to one.

$$C_3 = \alpha_1 C_1 + \alpha_2 C_2 \quad (2)$$

where

$$0 \leq \alpha_1, \alpha_2 \quad \text{and} \quad \alpha_1 + \alpha_2 = 1$$

Equation 2 ensure that the combination of colours will lie with the unit colour cube.

YUV Colour Model

The RGB model is suited for colour image display on devices such as computer monitors, as each colour component is mutually exclusive with a discrete value given to each component. However, it is not an efficient model for image processing for natural image and video compression. It is a mathematical model that does not take into account that the human visual system is more sensitive to luminance than chrominance. A more effective approach would be to separate the brightness information from the colour information. By separating luminance and chrominance, a higher resolution can be assigned to the luminance component while a lower resolution can be assigned to the chrominance components in favour of improved compression performance while reducing the impact of distortion for the end user.

The YUV model which has been traditionally used for colour television broadcasting is derived from the RGB colour model, It is described by the luminance component Y , and two colour difference components, UV .

The luminance component is a calculated weighted sum of the red, green and blue component, while the chrominance components are calculated by subtracting the blue and red components from the the luminance component.

The equations for the transform from RGB space to YUV space are described by the ITU-R recommendation BT.601-7[41]:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.148 & -0.289 & 0.437 \\ 0.615 & -0.515 & -0.11 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (3)$$

When using 8-bit values for each sample in a channel, the range of values for the luminance channel are uni polar and lie in the range from $[0,255]$, whereas the values for the colour difference component are bipolar where:

$$-144 \leq U \leq 111 \quad \text{and} \quad -98 \leq V \leq 157 \quad (4)$$

YCbCr Colour Model

The YCbCr Model is related to the YUV model and is used in the JPEG and MPEG standards. The chrominance values, Cb , Cr , are scaled and shifted to ensure they are a positive value unlike the YUV model.

Computing the YCbCr transform values from RGB components can be done with

$$Y = k_r R + k_g G + k_b B$$

$$C_b = \frac{B - Y}{2(1 - k_b)} + 0.5$$

$$C_r = \frac{R - Y}{2(1 - k_r)} + 0.5$$

(5)

where:

$$k_r + k_b + k_g = 1$$



(a) Image in 24 bit RGB colour space.



(b) Image in 24 bit YCbCr colour space

Figure 13: Standard test image 'lena' converted from RGB colour Space to YCbCr colour space

Figure 13a shows the standard test image 'Lena' represented in 24 bit true colour RGB, while figure 13b shows the same image transformed into YCbCr colour space.

One of the main purposes of transforming an image from RGB colour space to YCbCr colour space is to process the luminance and chrominance data independently. Figure 14a and figure 14b represent a 4x4 pixel sub image extracted from the top left corner of figure 13a and 13b respectively.

125 137 226	133 137 223	128 136 223	120 138 226
125 137 226	133 137 223	128 136 223	120 138 226
125 137 226	133 137 223	128 136 223	120 138 226
125 137 226	133 137 223	128 136 223	120 138 226

(a) 4x4 sub image sampled from 13a space.

162 174 107	162 171 112	161 172 109	162 174 104
162 174 107	162 171 112	161 172 109	162 174 104
162 174 107	162 171 112	161 172 109	162 174 104
162 174 107	162 171 112	161 172 109	162 174 104

(b) 4x4 sub image sampled from 13b space

Figure 14: A 4x4 pixel matrix sub-sampled from the top left corner of the images in 13a and 13b

Each square in figure 14a and 14b represent a 3 byte pixel. In 14a the values from top to bottom represent the blue, green and red contribution, while in 14b the values represent, the Y, Cr (difference between red and Y value) and Cb (difference between blue and Y value) contribution. It can be observed that the red channel contribution is the predominant value in 14a, with all values greater than 220 and lower values for the blue and green channels. Comparing the red contribution, with respect to the Cr contribution of 14b, It can be seen that the Cr values are lower, however, because of the colour space transform, some of the information about the red channel is captured in the Y channel, as is information from both the blue and green channel. To show the information compaction of the colour space transform, the standard deviation can be calculated on all channels of both figure 14a and figure 14b. The standard deviation can be defined as a measure of the amount of variation of a set of values and can be computed with:

$$\sqrt{\frac{\sum_{i=1}^N (x_i - \bar{x})^2}{N - 1}} \quad (6)$$

where $\{x_1, x_2, \dots, x_N\}$ are the channel values, \bar{x} is the mean value of the channel and N is the number of values in the channel.

Colour Space	Channel	Mean	Standard Deviation
RGB	Red	180.22	49.05
	Green	99.05	52.87
	Blue	105.41	34.057
YCbCr	Y	124.05	47.86
	Cb	117.49	13.63
	Cr	168.06	12.80

Table 1: Results of mean and standard deviation calculations on each colour channel of figure 13a and 13b

Table 1 shows the results of calculating the mean and standard deviation of each channel of 13a and 13b. It can be observed that the standard deviation of both the Cb and Cr channels are significantly lower than any of the channels in the RGB colour space, while the Y channel, sits mid way between the highest and lowest of the channels of the RGB colour space. A low standard deviation indicates that the pixel values tend to be close to the mean of the values of a given channel, while a high standard deviation indicates that the values are spread out over a wider range. The low standard deviation of the Cb and Cr channels is a significant result, if one was to replace some of the pixels with common a value, to reduce the amount of unique values being stored, the error in the processed signal would be less than that, if the same process was performed on a channel with high standard deviation. The process of replacing groups of pixels within the colour channels with a common value is known as chroma subsampling and its use in lossy compression algorithms is wide spread.

3.3 Chroma Sub Sampling

Chroma subsampling can be defined as a type of compression that discards colour information in an image in favour of reducing the file size for storage and to meet network bandwidth requirements. Subsampling is performed on the colour or chrominance information, rather than the brightness or luminance information because of the human visual systems ability to discern more brightness information than colour information. A chroma subsampling scheme is defined by its sample rate, typically for digital images, the three most popular are YCC 4:4:4, YCC 4:2:2 and YCC 4:2:0, where YCC is an abbreviation of YCbCr colour space.

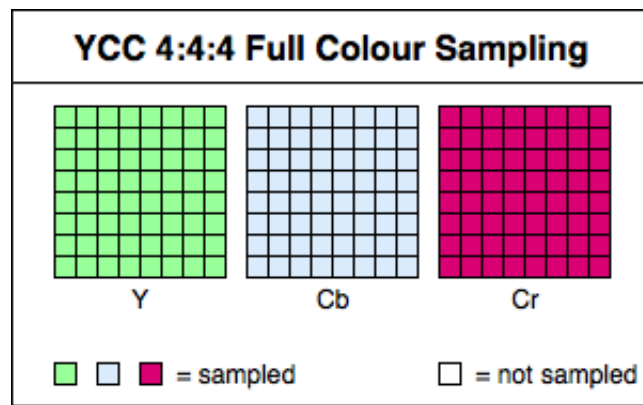


Figure 15: Full colour sampling pattern

Figure 15 Shows the full resolution sampling of YCC 4:4:4. The Cb and Cr colour channel values are sampled in a 1:1 ratio with the Y channel values. This is the highest sample rate, which is lossless. Because of the full resolution, the file size is very large, as an example a single 16x16 block of 3 byte pixels contains 768 bytes. Codecs such as JPEG and MPEG-2 both support YCC 4:4:4 sample rate, however it would not be suitable for real time transmission or storage.

Figure 16 Shows the YCC 4:2:2 subsampling pattern. Both the Cb and Cr channels are sampled at half the horizontal resolution of the Y channel. This method still produces high resolution images after processing with the resultant processed image being $\frac{2}{3}$ The size of the image. YCC 4:2:2 is supported in JPEG, H.264 [47] and HEVC [39].

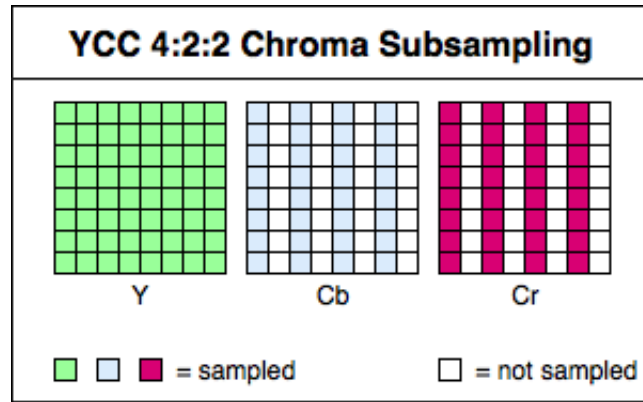


Figure 16: YCC 4:2:2 Chroma subsampling pattern

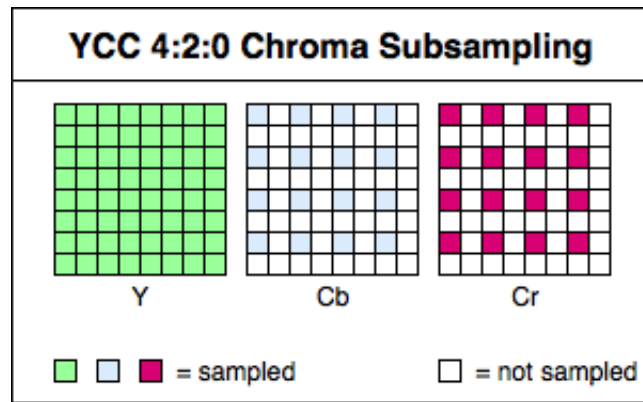


Figure 17: YCC 4:2:0 Chroma subsampling pattern

Figure 17 Shows the YCC 4:2:0 chroma subsampling pattern. This ratio is one of the most commonly used subsampling ratios, as used in most codec's which employ lossy compression, such as JPEG. The chrominance channels are sampled in half the resolution in both horizontal and vertical directions, compared to the luminance channel. After subsampling with 4:2:0, the file size is effectively halved, while still being able to produce an image of good subjective quality. There are many ways to implement the sampling process, in figure 17, the top left value in every 2x2 block of pixels is sampled for both the Cb and Cr channels. another method is to calculate the mean value of the 2x2 block of a given channel. As previously shown in table 1, the standard deviation in both colour difference channels is low, which would indicate that the values in each respective 2x2 block, or neighbourhood of pixels are highly correlated.

Figure 18 Shows a high level over view of the process of performing a colour space transform and YCC 4:2:0 chroma subsampling on a 16x16 block of 3 byte pixels in RGB colour space. The output of the subsampling process is four 8x8 Y channel blocks and two 8x8 colour space blocks. The purpose of decomposing the original size block into 8x8 blocks is for further processing, usually performing a discrete cosine

transform to further decorrelate the information in the pixels. The discrete cosine transform will be discussed further on in this thesis.

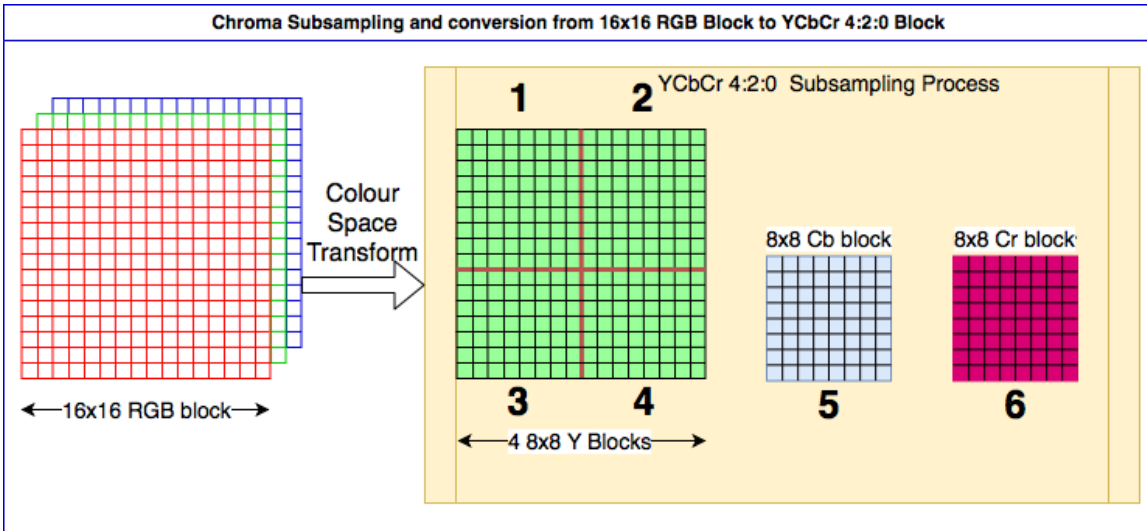


Figure 18: A 16x16 RGB Block Chroma sub sampled into 4 8x8 Y Blocks, 1 Cb block and 1 Cr Block

Figure 19a shows the original test image in full colour 24 bit RGB, while figure 19b shows the test image after processing with YCC 4:2:0 chroma subsampling and transformed back to RGB colour space. It can be observed from figure 19b that there is little observable difference in subject image quality, even though in compressed form, figure 19b is half the size of figure 19a. This is a significant result, as it demonstrates that it is possible to discard some information in an image in favour of compression performance, without having dramatic effects on subjective image quality.



(a) Image in 24 bit RGB colour space.



(b) Image Processed with YCC 4:2:0 chroma subsampling space

Figure 19: Comparing the original 24 bit RGB test image vs processed with YCC 4:2:0 chroma subsampling

Chroma sub sampling is a lossy type of compression. Once the original data has been discarded, it is lost forever, which means the output after the process is only an approximation of the original image. Because of this, there will be some error in the reconstructed image. By calculating the absolute difference between the processed image and the original image, an image residual can be generated. The residual image calculated from figure 19a and 19b can be seen in figure 20a. As the difference is quite small, all the values are close to zero, which is interpreted as near black. The residual image is a powerful tool that is used in video compression standards such as H.264 and HEVC to compress successive frames in a sequence. A single frame is compressed fully and then successive frames are compared to it. As there is not much change between sequential frames in a sequence, the dynamic range in pixel information is greatly reduced. Compression techniques are then performed on the residual, which contains significantly less information and is more efficient to compress. Then on the decoder side, the compression information from the residual frames is then summed with the first frame of the sequence for reconstruction.

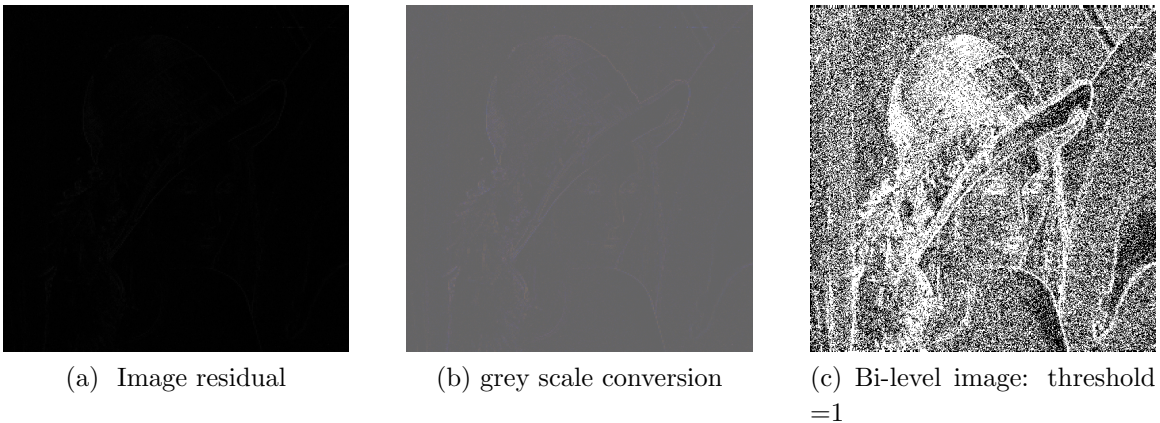


Figure 20: Comparing the absolute difference between original image and chroma subsampled image

Figure 20b is the residual image transformed to a grey scale image, with a uniform scalar value of 100 added to every pixel in each channel, the error is slightly more visible. Figure 20 shows the residual image transformed to a bi-level image with two levels, 0 and 255. Hard thresholding is performed on the residual, where any non zero value in each channel is interpreted as 255. It is clear to see where the error is in the signal using this method. Viewing the error with the bi-level image, brings to light an important point, even though there are many pixel values that are not the same as the original, the distortion to the image caused by the process is not that visible to the human visual system. Two standard metrics for gauging the quality of a compression algorithm are Root Mean Squared Error (RMSE) and Peak Signal to Noise Error (PSNE). Both metrics are measures of the error in a signal, but do not take into account the human visual systems ability to perceive the error or not. The error that is exposed in the bi-level image is not perceivable when comparing figure 19a and 19b. Image quality metrics will be discussed further in this thesis.

3.3.1 Conclusion on Chroma Subsampling

Chroma subsampling is an efficient way to compress natural and camera captured image as the distortion, as shown, can be masked in the processed image, so it has little effect on the human visual system. However, chroma subsampling may not be suitable for all types of images, such as computer generated image and text.

4 Compound Image

A compound image is defined in this thesis as a digital image containing two or more distinct types of data class, such as computer generated text and continuous tone natural image. In this section, the types of data class and their attributes that make up a compound image will be analysed.

There has been extensive research in the field, [36] [9] [23] [48], on image classification, which decompose a compound image into blocks and classify each block based on unique attributes. The purpose of this is two fold:

- Different compression algorithms can be applied to blocks classified as a specific type based on their attributes for optimal compression performance.
- Certain compression algorithms are less computationally expensive compared to others. By choosing a sufficient compression algorithm for a given class of data, the overall resources required will be reduced. As an example, compressing a block of single colour pixels by simple counting algorithm will require less resources than compressing a block of pixels containing continuous tone image by means of a discrete transform function. This improves computing efficiency and will reduce latency.

Figure 21 shows a segment of a typical compound image that may be displayed on a computer screen. It can be observed that the image contains a combinations of continuous tone image, graphical elements like the elements of the browser, text data, areas of the screen that are single colour and also computer generated image, such as icons and logos.

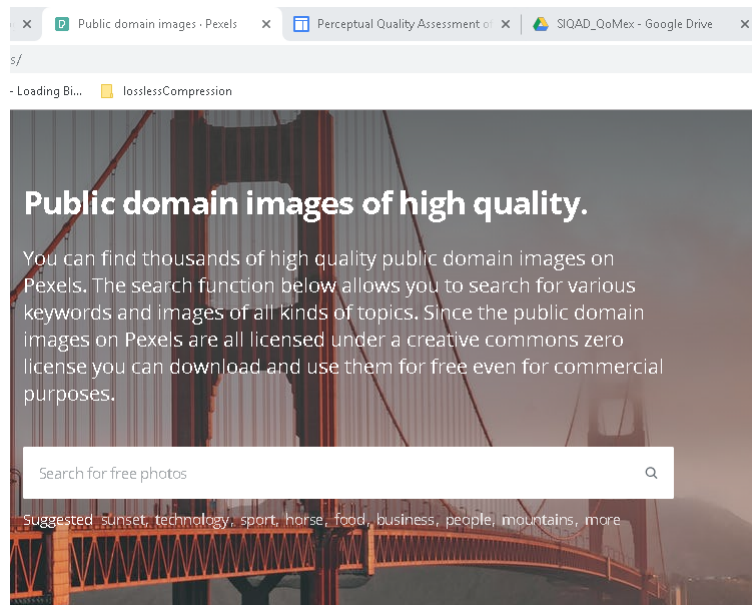


Figure 21: Section of a typical computer desktop displaying a compound image

4.1 Image Data Classes

The research presented in [36] chose to segregate the data in a compound image using two data classes which are *Text* and *Picture*. The research presented in [9] expanded upon this by choosing four data classes which are *Smooth*, *Text*, *Picture* and *Hybrid*. Image data that is classified as *Smooth* contains a single colour and image data classified as *Hybrid* contains a mixture of computer generated graphics and continuous tone image. The research presented in [48] further classifies the image data that contains continuous tone image by introducing another data class called *Fuzzy*. Image data classified as *Fuzzy* contains a high unique pixel count, however without any discernible pattern and all with a similar intensity.

The research presented in [48] chooses 5 image data classes which are *Smooth*, *Sparse*, *Text*, *Fuzzy* and *Picture*, which are the same image data classes used in this thesis. The remainder of this section will describe each class of data.

Figure 22 Shows sample 16x16 pixel blocks extracted from a compound image and expanded to show the individual pixels. The first three types of block are image data that has been computer generated and generally contain fewer unique colours, have definitive structure and are free of noise. The remaining two types of blocks are generally image data that has been camera captured containing continuous tone image. They have a higher distribution of unique pixel values, contain less uniform structure and have a more textured appearance.

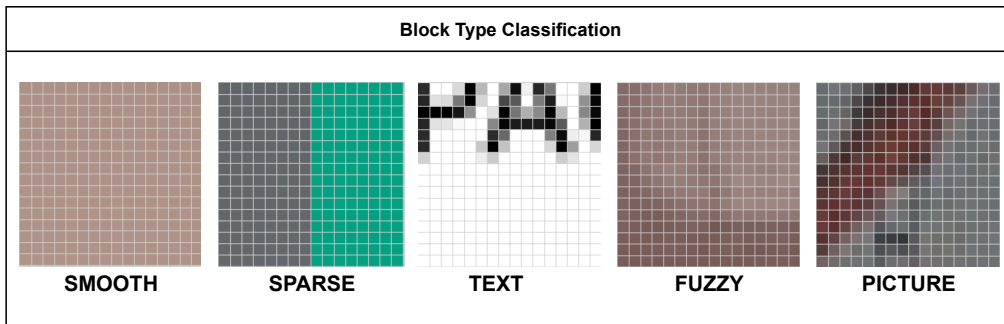


Figure 22: Five Classifications of Blocks Associated with Compound Images

4.1.1 Smooth Class

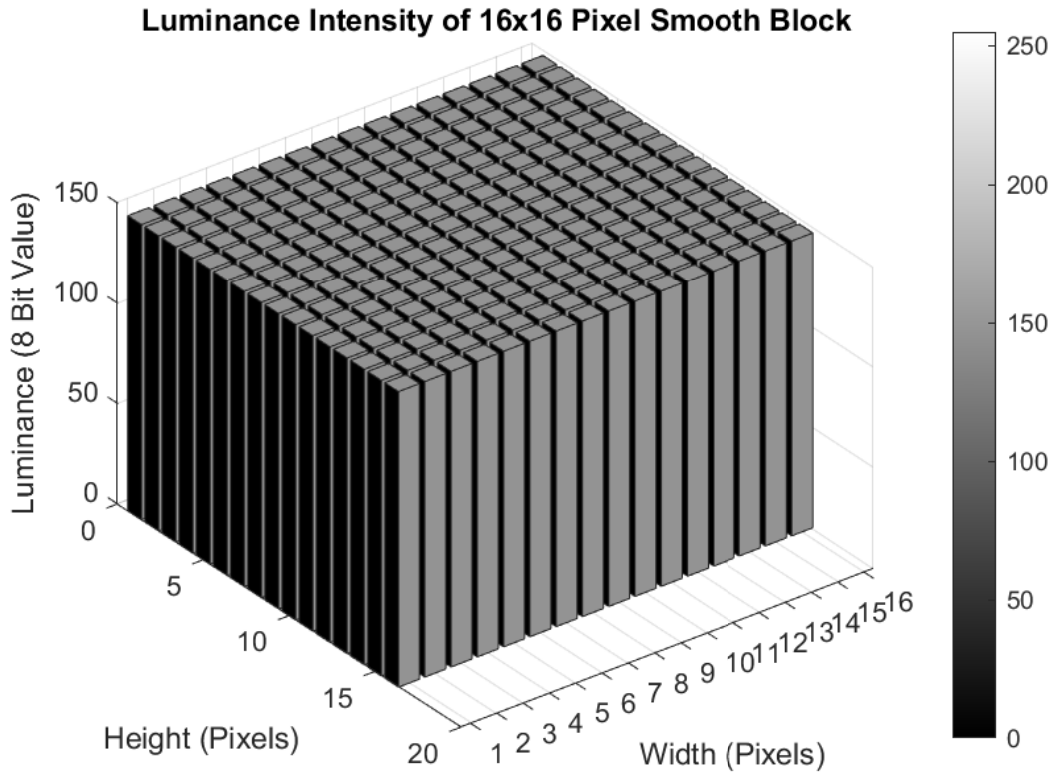


Figure 23: Luminance Channel intensity Graph of 16x16 Pixel Block That is Classified as a Smooth Block

Figure 23 represents the luminance channel of the smooth block in figure 22 in 3 dimensions. The block has been transformed from RGB colour space to YCbCr colour space and the height of each column represents the luminance intensity for a given pixel in the block. The name smooth is derived from the texture of the block, which can be seen as completely uniform and flat. Smooth blocks can be found in a compound image displayed on a computer when a user may be using applications such as word processing applications or pdf viewers, generally, many smooth blocks will be located in large areas of the screen and are in close proximity to one another.

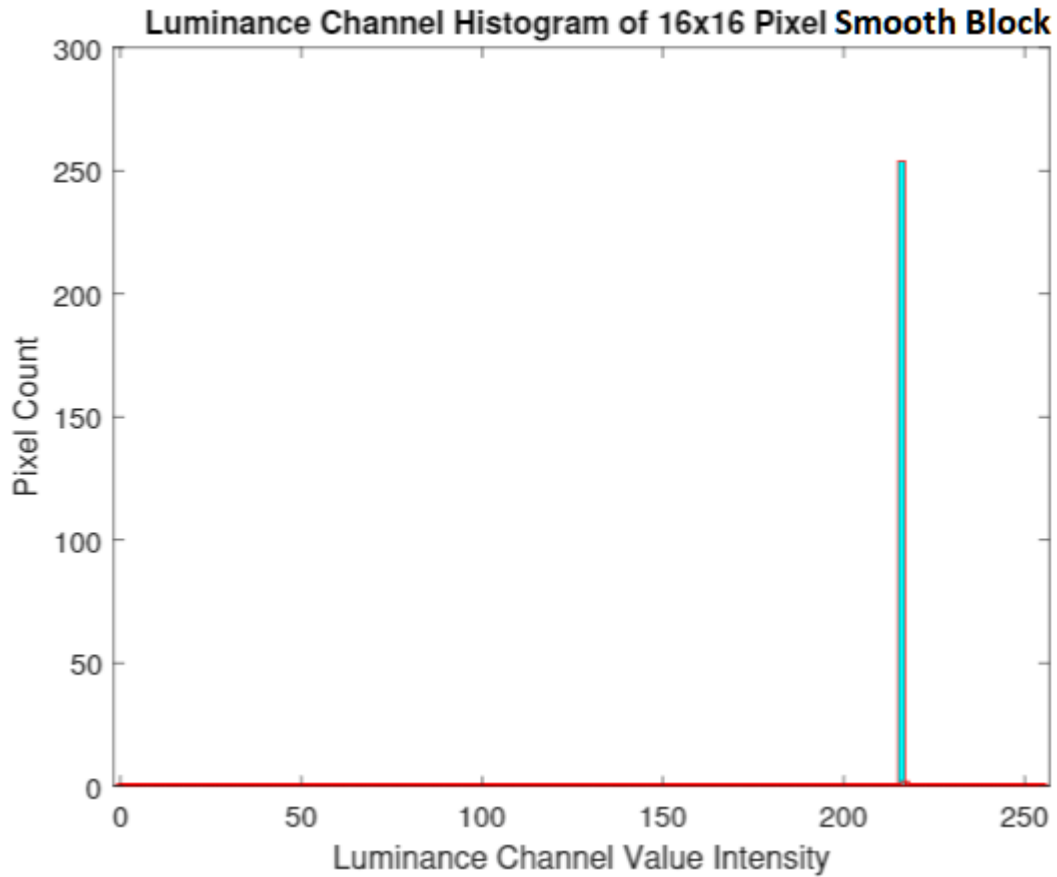


Figure 24: Luminance Channel Histogram of 16x16 Pixel Block Classified as Smooth

Figure 24 shows the histogram of the luminance channel of the smooth block. It can be observed that the smooth block is dominated by a single value. The data contained in a smooth block is completely computer generated data, as there is no variance in values of the pixels.

Smooth blocks have ideal properties for compression. When a given 16x16 block of pixels can be fully represented by a single value which would give a theoretical compression ratio of 256:1, which is very good compression performance. As smooth blocks are generally found to be located in succession, many smooth blocks will contain the same value, which would allow for further processing to achieve further compression performance. Smooth block compression will be developed further on in this thesis.

4.1.2 Sparse Class

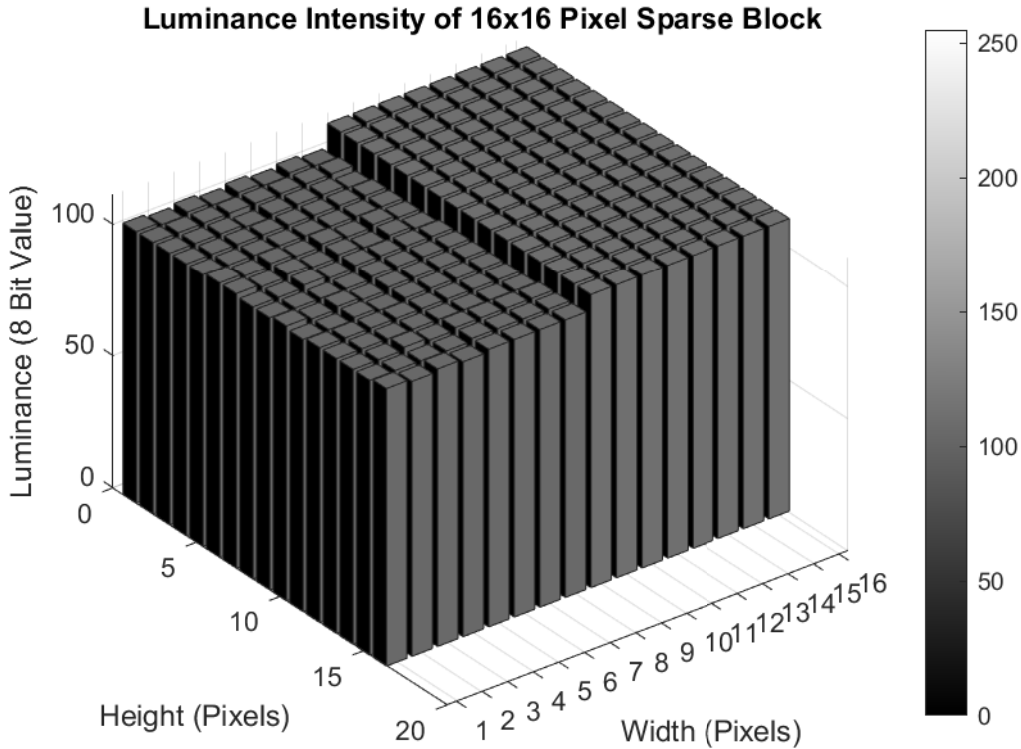


Figure 25: Luminance Channel intensity Graph of 16x16 Pixel Block That is Classified as a Sparse Block

Figure 25 represents the luminance bar graph of the sparse block from figure 22. It can be observed from the bar graph that the texture of the block is largely smooth, but with a definitive partition running through it. This type of image data does not regularly occur in natural image and is most likely computer generated. Sparse blocks contain structural information and typically can be found on perimeter borders of applications running on a desktop.

There can be multiple sparse blocks in succession typically beside each other in a vertical or horizontal direction.

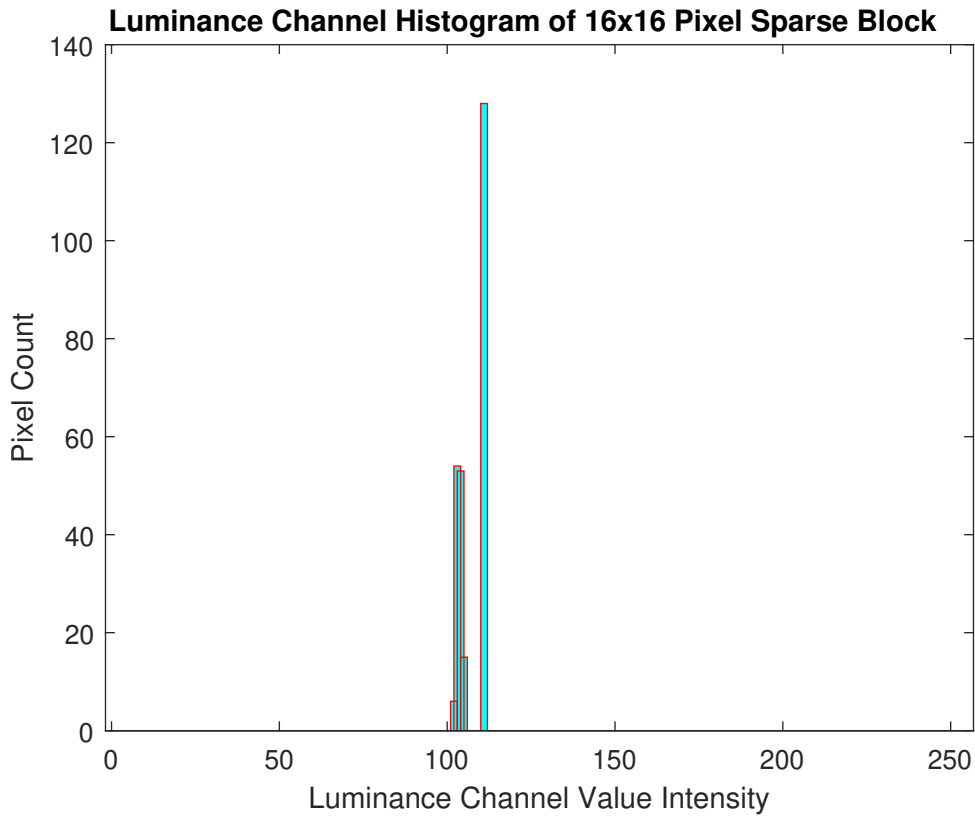


Figure 26: Luminance Channel Histogram of 16x16 Pixel Block Classified as Sparse

Figure 26 shows the luminance channel histogram of the sparse block. It can be observed that there are few dominant colours. The histogram shows that the dominant colour values are within proximity to each other, but sparse blocks may contain colours that are spread out from each other also. Sparse blocks will contain fewer dominant colours than natural image and have defined structures in the data. The defined structure typically consists of repetitive patterns which allow for efficient compression using lossless compression algorithms. However, because the structural information in the data is generally linear, transform techniques such as the discrete cosine transform which is used in lossy compression techniques may induce compression artifacts and poor compression performance.

To ensure efficient compression performance and objective image quality, compression techniques that can exploit spatial correlation and pattern matching should be used for blocks that have been classified as sparse.

4.1.3 Text Class

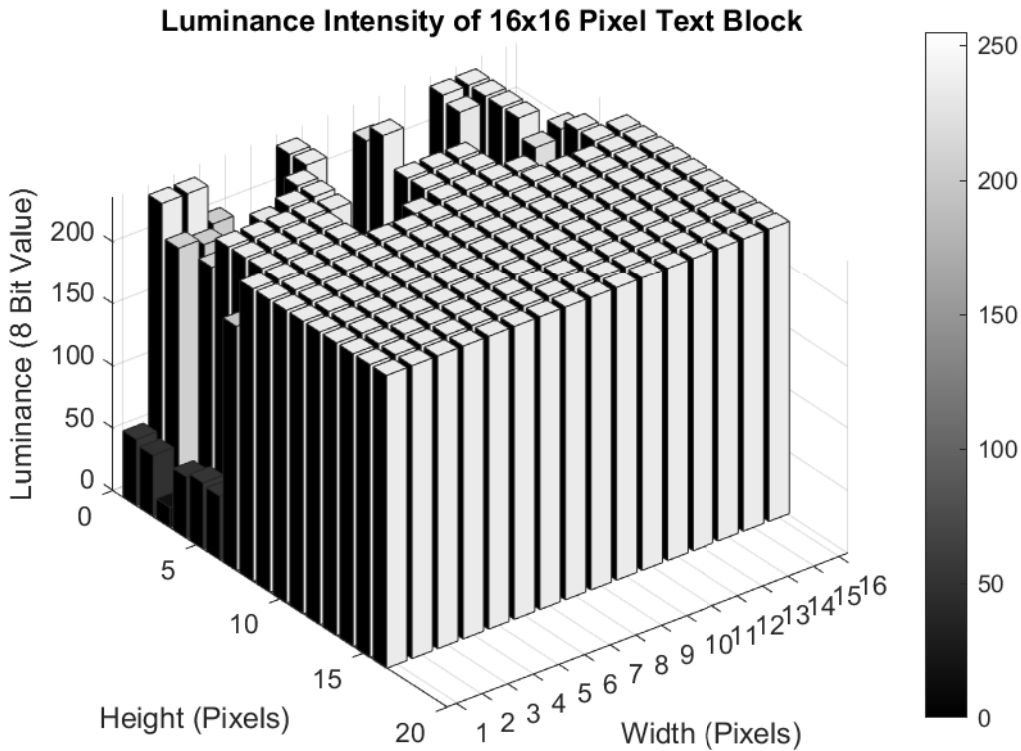


Figure 27: Luminance Channel intensity Graph of 16x16 Pixel Block That is Classified as a Text Block

Figure 27 Shows the luminance channel bar graph of a text block from figure 22. It can be observed that the texture of the block is different to both smooth and sparse blocks. Text blocks can contain complex structural information that make up the characteristics of text, which are sharp strokes in vertical horizontal and diagonal directions. Text blocks may contain multiple unique pixel values but are usually lower in unique pixel values than blocks containing natural image.

To ensure that the information contained in the text can be retrieved by the human visual system, the intensity values of the pixels of a text object should have a high contrast to the pixel values in the background. As the Human visual system is tuned to extract structural information from an image, it is essential that textual data in images is coded with high spatial resolution. Any artifacts introduced in the compression and decompression of textual information will be more noticeable to the end user leading to a degraded user experience.

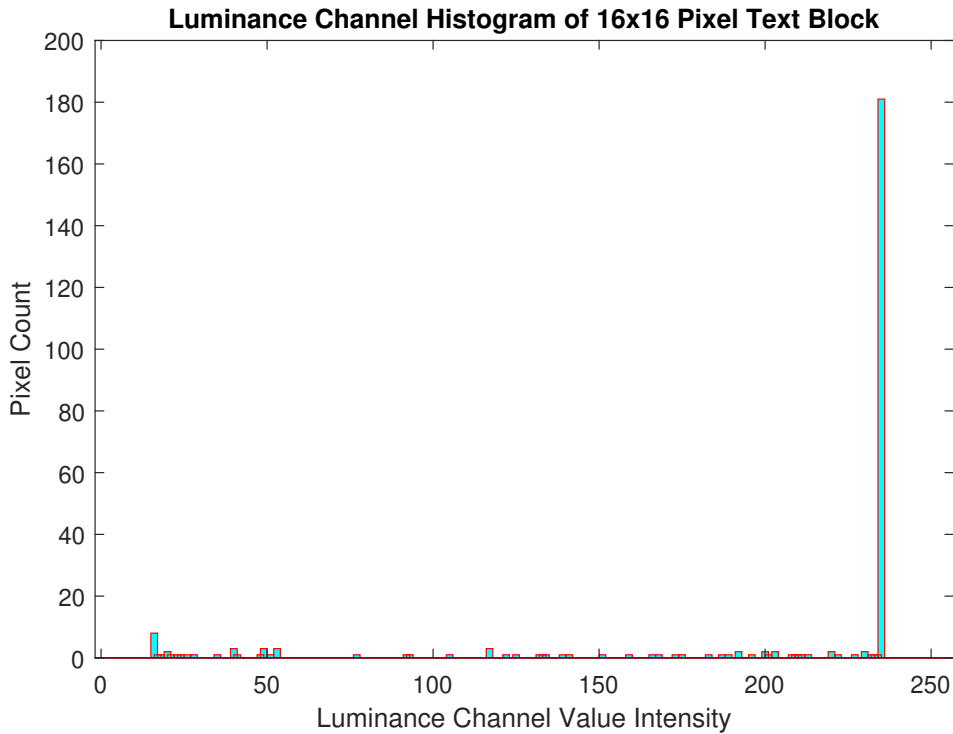


Figure 28: Luminance Channel Histogram of 16x16 Pixel Block Classified as Text

Figure 28 shows the luminance channel histogram for the text block. The dominant value that can be observed is the background colour, while the smaller spikes which are sparsely distributed are what make up the text. It is essential to not lose any of this information in the coding process, which is why lossless compression techniques such as dictionary style compression are best for image data containing text.

4.1.4 Fuzzy Class

Figure 29 shows the luminance channel bar graph of a fuzzy block. The name fuzzy is derived from the texture of the block. It can be observed from figure 29 that there is low variation in intensity levels among the pixels with no discernible pattern. Further, the degree of intensity is similar for the whole block.

Fuzzy blocks are usually located in areas of the screen which show natural camera captured image data, such as desktop background images or photos. An example of an image that would contain many fuzzy blocks would be a picture of a field of grass or a picture of a sky. Fuzzy blocks are ideal for compressing with lossy compression algorithms, such as using the discrete cosine transform, followed by quantization, such as in JPEG. This is due to the lack of structure that is needed to be replicated, coupled with the discrete cosine transform's ability to compact most of the energy in a signal into just a few coefficients, which can then be transmitted and decoded to give a close approximation of the original image.

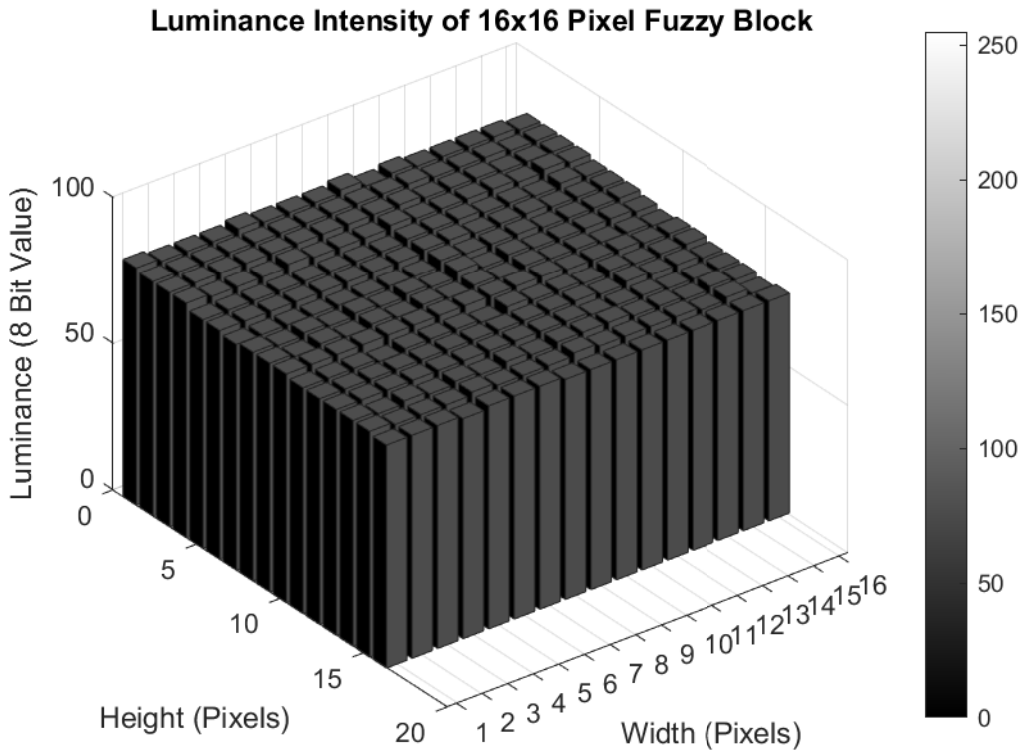


Figure 29: Luminance Channel intensity Graph of 16x16 Pixel Block That is Classified as a Fuzzy Block

Figure 30 shows the luminance channel histogram of the fuzzy block. It can be observed that the distribution of the intensity values are quite close to the normal distribution curve, which indicates the lack of structure within the block. Fuzzy blocks may contain fewer unique pixel values than blocks classified as picture. As the human visual system is less sensitive to colour information compared to luminance information, the colour channels of the fuzzy block can allow for heavier quantization in the compression stage, to improve compression performance without much discernible loss of information to the end user.

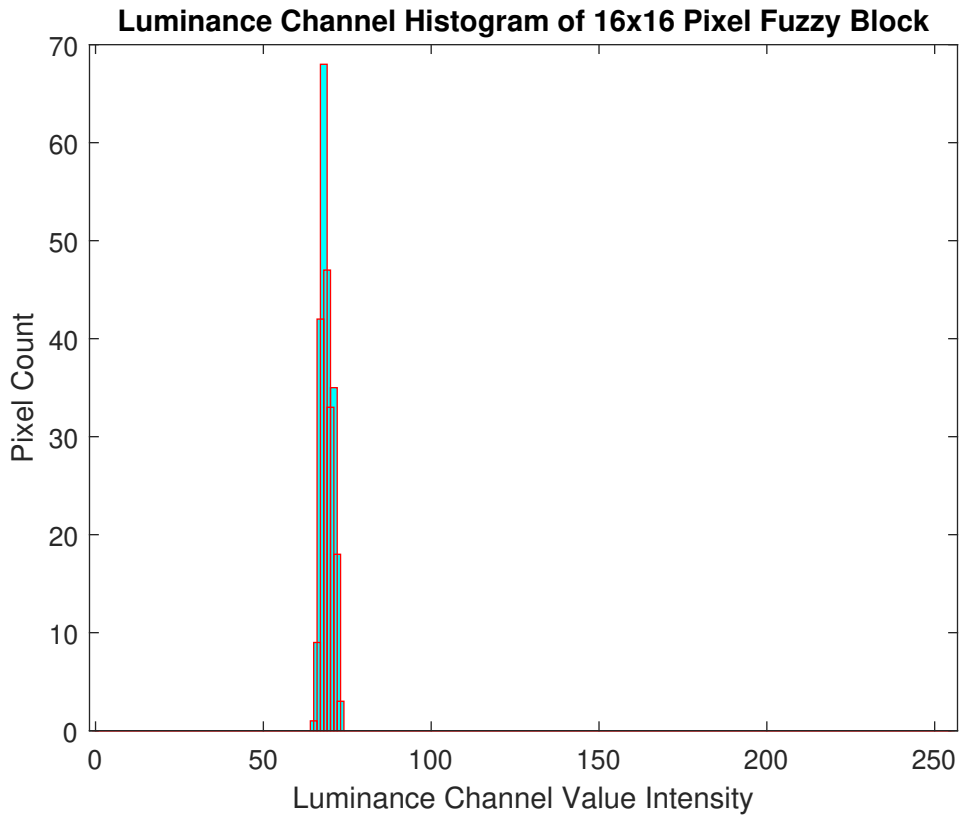


Figure 30: Luminance Channel Histogram of 16x16 Pixel Block Classified as Fuzzy

4.1.5 Picture Class

Figure 31 shows the luminance channel bar graph of the picture block. The texture of the block is more coarse than that of the fuzzy block. It can be observed that there is structure in the block, although it is not linear like the structure of the text and sparse block. However, the transition in intensity between neighbouring pixels develop slower than the transients in both text and structure blocks.

It can be observed from figure 31 that there is correlation between neighbouring pixel values in a picture block. The discrete cosine transform leverage's this correlation among neighbouring pixels and the slowly varying transitions within the block itself, by comparing each sample to a harmonic of a cosine series: if the value is similar, the difference will be close to zero, such that the value is not needed in the decoder stage to generate a good approximation of the original image. The discrete cosine transform and its ability to compact a two-dimensional signal's energy into relatively small amount of coefficients will be analysed further in this thesis.

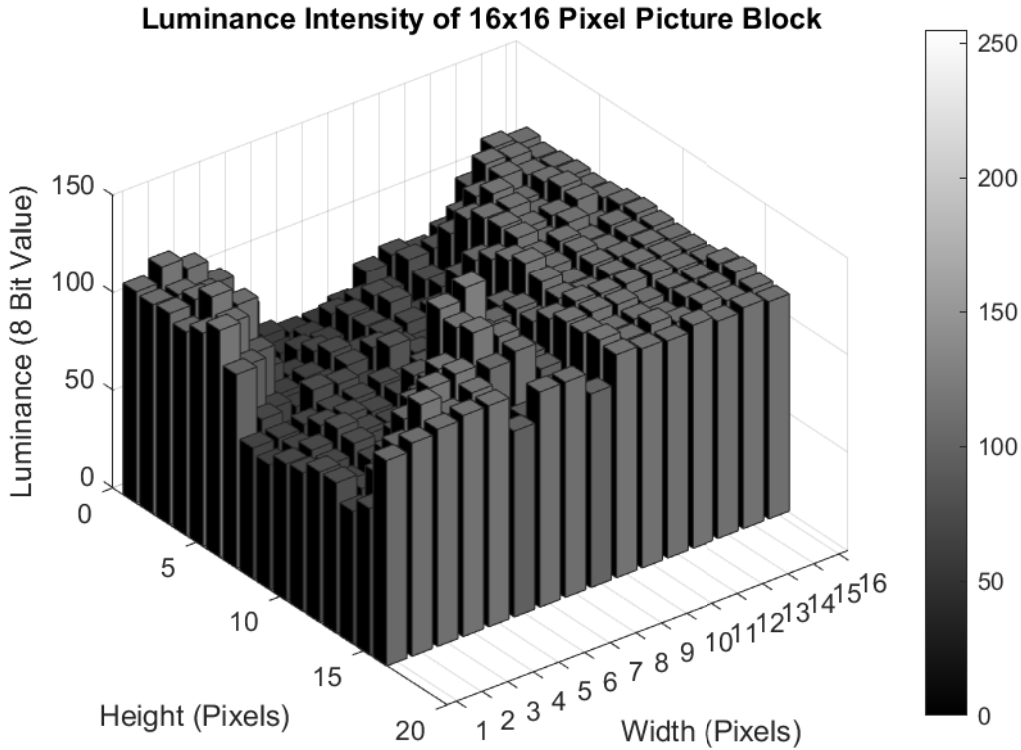


Figure 31: Luminance Channel intensity Graph of 16x16 Pixel Block That is Classified as a Picture Block

Figure 32 shows the luminance histogram of the picture block. It can be seen that it has the widest distribution of unique pixel values with significant pixel count. As previously stated, there is correlation among neighbouring pixels so performing a discrete cosine transform has the ability to compact the signal energy into a smaller amount of coefficients, however a lighter quantization should be used compared to the quantization used on fuzzy blocks as to preserve as much of the structural information in the data as possible. There are many types of quantization matrices that have been researched and will be discussed further on in the thesis in section 6.2.3.

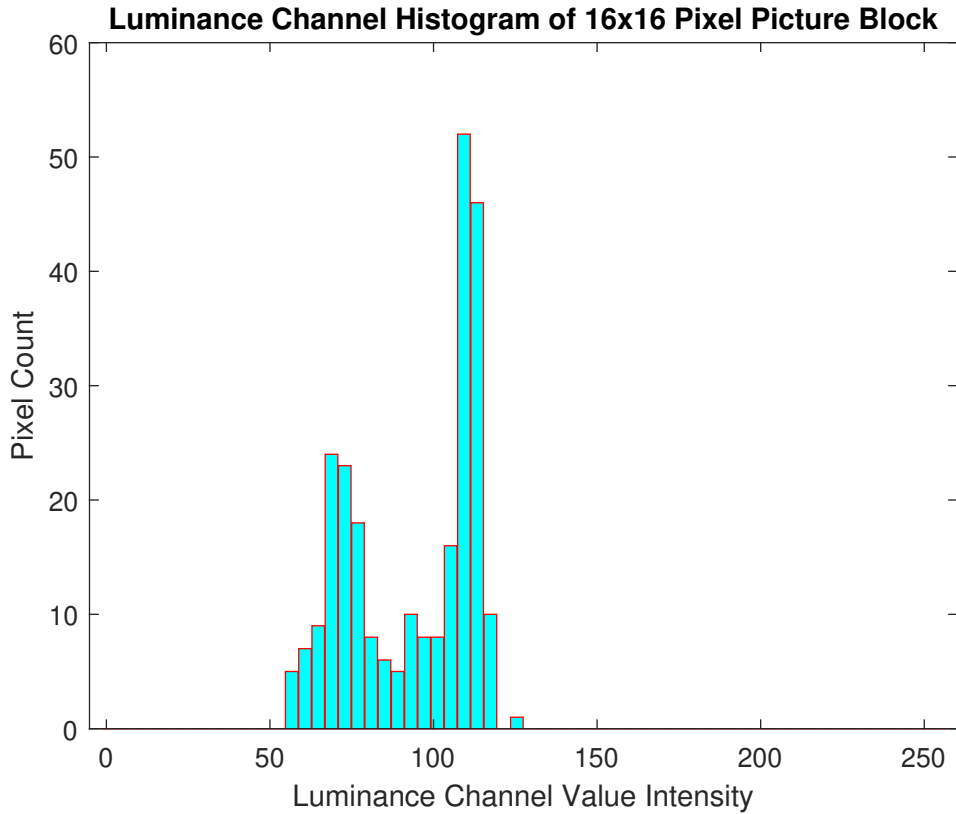


Figure 32: Luminance Channel Histogram of 16x16 Pixel Block Classified as Picture

4.2 Conclusion On Compound Image

In this section, five classification types of data that each have specific attributes that are inherent in compound images have been presented and analysed. Suggestions have been proposed on the types of compression algorithms to use for each type. The following example is aimed to illustrate some of the compression artifacts incurred by using sub optimal compression techniques for a given block type.

Figure 33 shows a 16x16 block of pixels which contains data that is classified as text before processing and after processing using a combination of discrete cosine transform and quantization. It can be observed that there are significant compression artefacts around the structure of the text. This illustrates the negative impact that occurs when using an inappropriate type of compression algorithm for a given type of data.

However, figure 34 shows a 16x16 block of pixels which contains data that could be classified as picture or fuzzy. It shows that, using the same processing that was used in figure 33, there is little discernible subjective difference between the non processed and processed blocks. An efficient compound compression algorithm should then be able to choose a compression algorithm that is suited for a block containing a given type of data.

The next section in this thesis presents ways in which to classify the data in a given compound image.

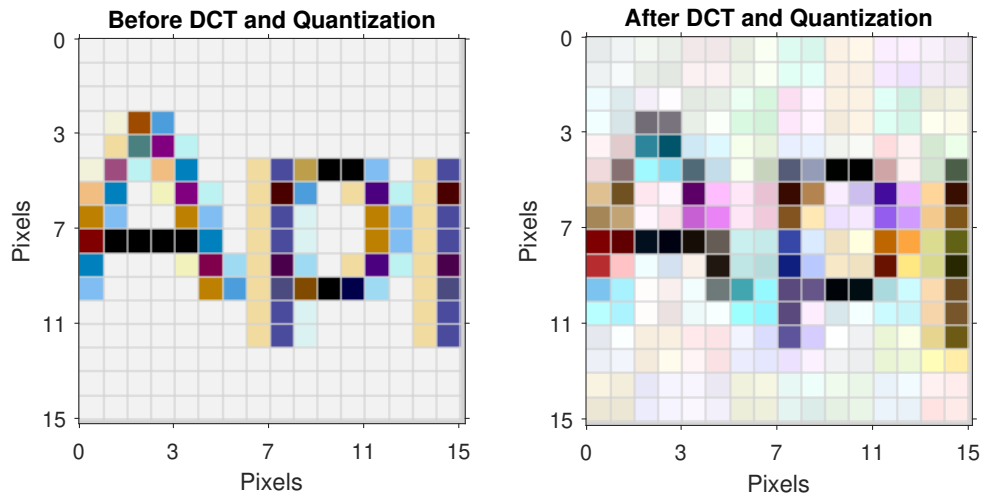


Figure 33: Compression Artifacts of The Discrete Cosine Transform and Quantization on a 16x16 Pixel Block Containing Text

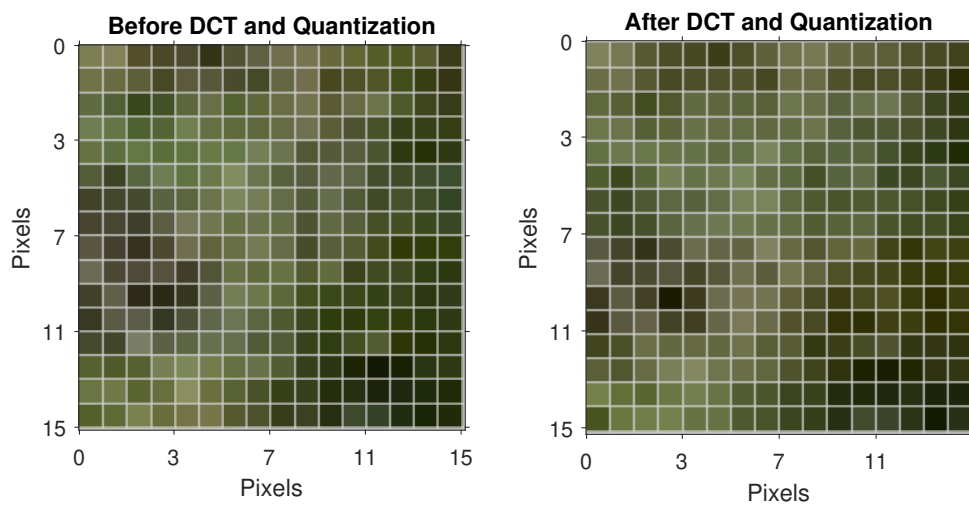


Figure 34: Compression Artifacts of The Discrete Cosine Transform and Quantization on a 16x16 Pixel Block Containing Natural Image

5 Classification

The purpose of classification in a compound compression algorithm is to separate the different types of image data within a compound image, so each class of data can be encoded with an algorithm that is suited to its attributes. The goal is to be able to choose the type of compression algorithm in a 'first pass' over the data. This is especially important for coding systems that should run in real time, such as a virtual desktop infrastructure or cloud computing applications, where added latency can be detrimental to an end user's experience of the system.

There are different architectures that can be used in classification, one such architecture is the layer based classification used in djVU [2]. However, this type of approach processes the whole image at a time and uses computationally expensive algorithms in the segmentation process, which is noted in [36], [9].

Another approach is to use a block based architecture. This approach lends itself to efficiency in terms of being able to process an image one block at a time, which reduces the amount of memory needed and also allows for parallel processing. Also, block based classification is a natural fit for block based compression algorithms such as JPEG [18], H.264 [47] and HEVC [39].

The classification presented in this thesis is inspired by the research done by *Juliet* [22] and *Wu* [48]. Both of these use a discrete Haar wavelet transform to decompose a block of pixel values into sub bands, which can then be analysed by their statistical properties to determine whether a given block contains computer generated data such as text or graphics, or natural image. The purpose of this is to choose a lossless compression algorithm for blocks that will contain computer generated data and a lossy compression algorithm for blocks that contain natural image.

The discrete Haar wavelet transform is introduced in this section, with emphasis on how it can be used for classification. It will also be introduced in the compression section on how it is implemented for compression. The classification algorithm using the discrete Haar wavelet transform will be developed in the testing section of this thesis and the implementation in code can be seen in the code section of this thesis in listing 22 .

5.1 The Discrete Wavelet Transform

In image processing, the discrete wavelet transform is an extremely powerful tool that can be used in multiple ways such as image de-noising, edge extraction, filtering, compression and classification. In this research the discrete wavelet transform will be analysed for both image classification and compression.

Like the discrete Fourier transform (DFT) and discrete cosine transform (DCT), the discrete wavelet transform (DWT) converts a discrete signal from the time domain

to the frequency domain. However, unlike the DFT and DCT transforms which can only decompose into frequency domain with no information of the locality of the frequency content (both transforms can describe what frequencies are in the signal, but not where the frequency content is located in time), the discrete wavelet transform decomposes the signal using a set of wavelets which are localised in the time-frequency scale.

There are many forms of discrete wavelet transform which vary in degrees of computational complexity, such as the Daubechies wavelet transform series , IW44 and Mexican hat. Research done by Juliet [22] and Wu [48] have shown that using a discrete Haar wavelet transform shows good performance with respect to computational complexity and accuracy.

The discrete wavelet transform decomposes an image into four sub bands by applying a combination of low pass and high pass filtering. Essentially, each band is a filter bank. The sub bands are described by the type of filtering applied along the rows and columns. The four sub bands are LL, HL, LH and HH, where L stands for Low pass and H stands for High pass.

The discrete Haar wavelet transform is both separable and reversible, which means that the two dimensional transform that is needed for an image in two dimensions can be separated into two one dimensional transforms, which improves the performance of the transform with respect to computation time. Being reversible means that it is suitable for compression and decompression.

The forward discrete Haar transform can be described as a pair-wise average and difference function. The average values for a pair of samples is computed, where the computed value can be considered low pass filtered, while the difference calculation between a pair of neighbouring values can be considered as high pass filtered.

If X is an array of pixel values, with length N , the average function of the forward discrete wavelet transform can be described as

$$s_k = \frac{X_{2k} + X_{2k+1}}{2} \quad k = 0, \dots, N/2 \quad (7)$$

where S_k is the pairwise average between two samples.

The difference function can be described by:

$$d_k = \frac{X_{2k+1} - X_{2k}}{2} \quad k = 0, \dots, N/2 \quad (8)$$

To show how the discrete wavelet transform works, let X equal a row of pixel intensity values:

$X =$	80	80	80	80	80	40	40	40	60	80	80	60	60	80	80	80
-------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

It can be observed from the values of X that there is a strong correlation among neighbouring pixels in the row. The type of pixel data in X is typical of computer

generated data such as text or graphics. Figure 35 Shows the array of pixel intensity values, X_N plotted in the pixel domain. It can be observed from figure 35 that there are uniform areas and sharp gradients.

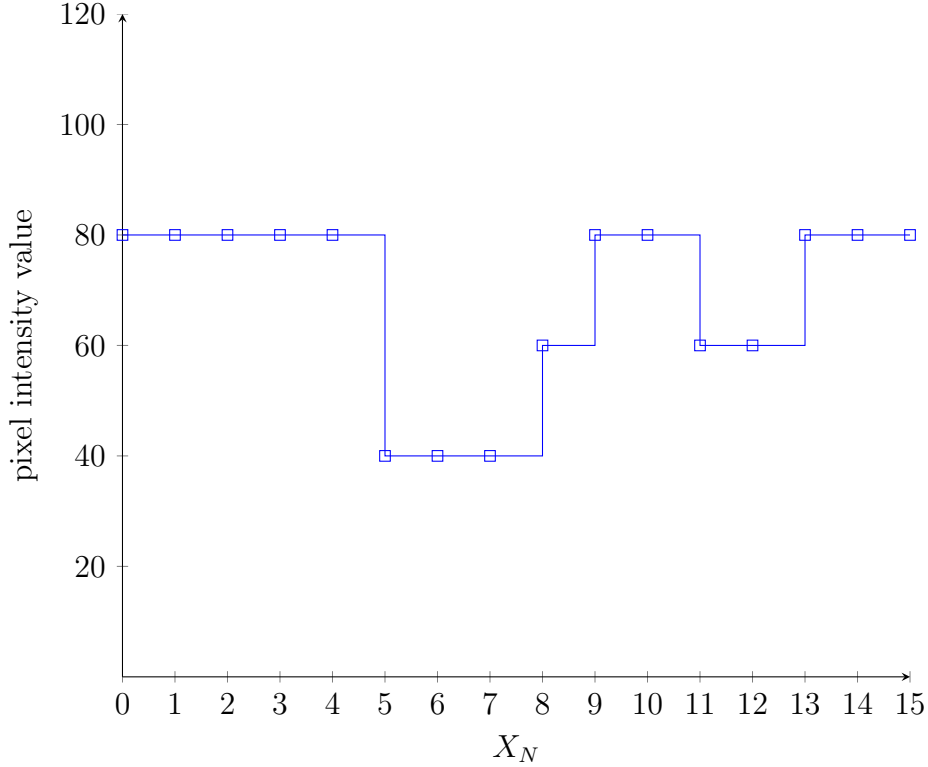


Figure 35: Pixel intensity values vs Location in array

Performing the forward discrete Haar wavelet transform along the columns of X would output an array with the average values computed with equation(7) (Low Pass) on the left and the difference values computed with equation(8) (High Pass) on the right:

$$X_T = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline s_0 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & d_0 & d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 \\ \hline \end{array}$$

The computed values for the discrete Haar wavelet transform, X_T , are :

$$X_T = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 80 & 80 & 60 & 40 & 70 & 70 & 70 & 80 & 0 & 0 & -20 & 0 & 10 & -10 & 10 & 0 \\ \hline \end{array}$$

The pairwise averaged coefficient values, s_n are plotted in figure 36. It can be observed that figure 36 is an approximation of figure 35 with only half the amount of sample values. The strong correlation between neighbouring pixels can be observed from figure 36 because of the close similarity to figure 35. However, the high frequency changes towards the right hand side of figure 35 has been filtered out in figure 36.

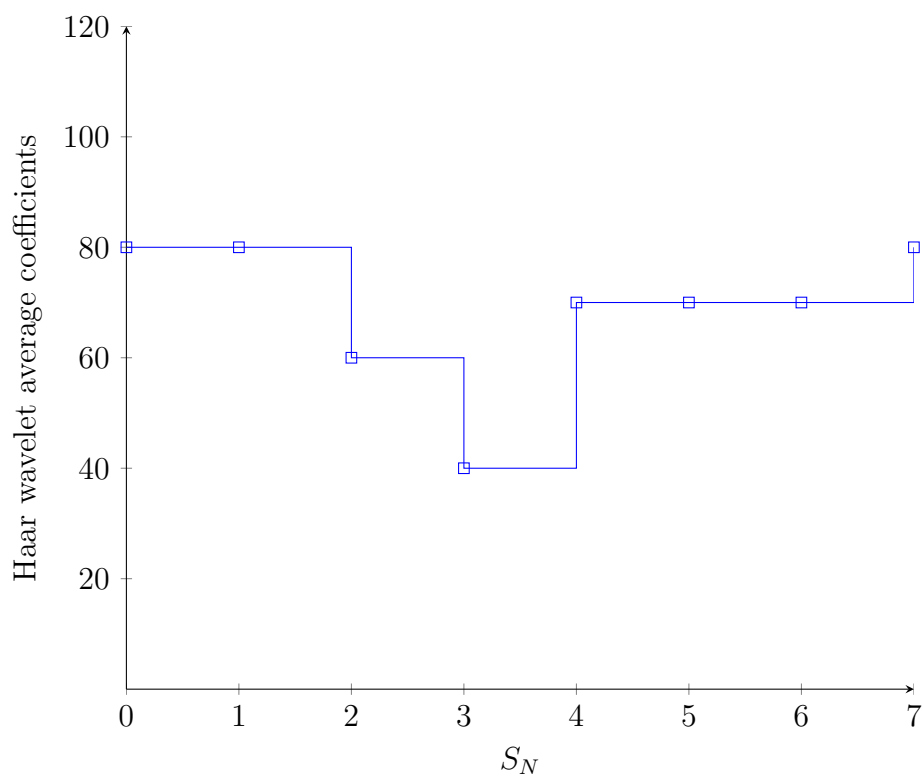


Figure 36: Discrete Haar Wavelet average coefficients

The pairwise difference coefficients, d_N , after performing the discrete Haar transform on figure 35 can be observed in figure 37. The transient information that has been lost in figure 36 can now be observed. One of the fundamental differences between the discrete wavelet transform is that it can distinguish frequency information about a signal, but also the locality of the frequency content, this is shown in figure 37 and is fundamental to the algorithm's success in image classification.

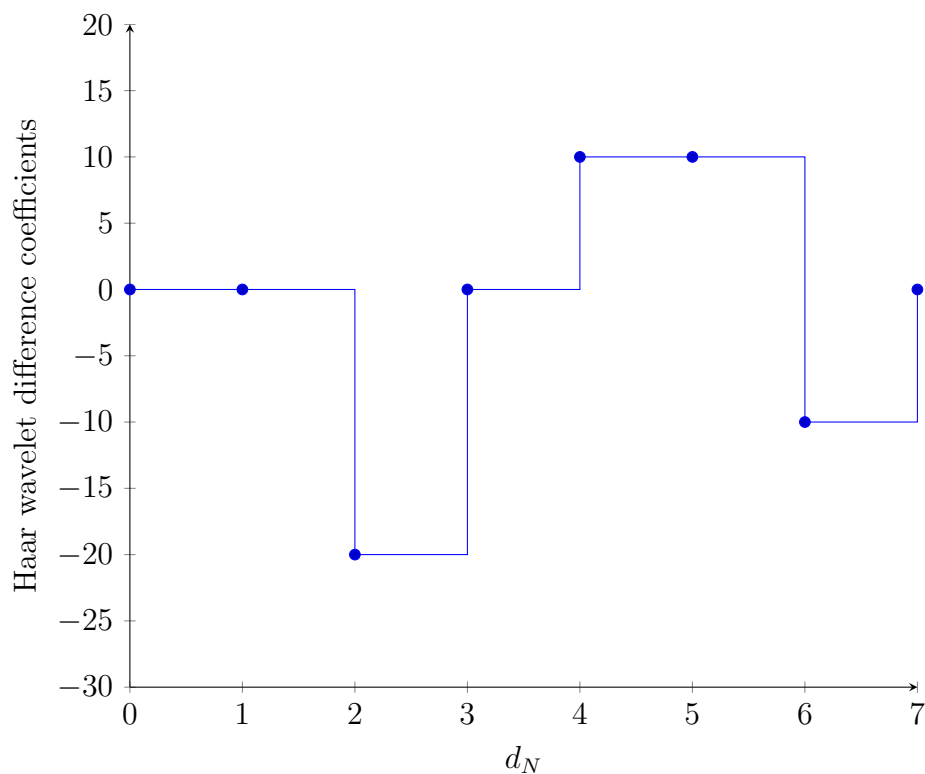


Figure 37: Discrete Haar Wavelet average coefficients

Figure 38 is an example of processing an input image with a one level, two-dimensional discrete wavelet transform, performed as two one-dimensional transforms.

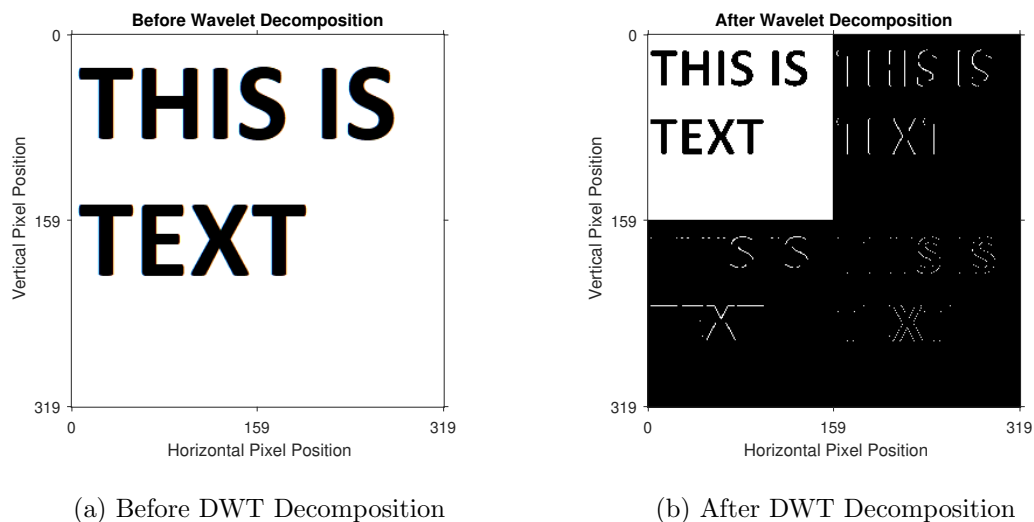


Figure 38: Discrete Haar transform Decomposing a Single Channel 320x320 Pixel Image Containing Text

The processed image in in figure 38(b) has the same dimensions as the figure 38(b),

however is split into 4 distinct regions, which represent the four subbands: LL, HL, LH and HH. The top left corner of 38(b) represents the LL sub band. It contains an approximation of the original image that has been low pass filtered in both horizontal and vertical directions and scaled to $\frac{1}{4}$ size of the original image.

The top right hand corner represents the HL sub band and has been high pass filtered in the horizontal direction and low pass filtered in the vertical direction. It can be observed that there is information about vertical strokes.

The bottom left hand corner represents the LH sub band and has been low pass filtered in the horizontal direction and high pass filtered in the vertical direction. It shows information about horizontal strokes.

The bottom right hand corner represents the HH sub band and has been high pass filtered in both directions. Only very fine details are shown in this sub band, such as diagonal strokes.

The discrete Haar wavelet transform's ability to localise high frequency transient information can be used in image classification to distinguish computer generated data such as text and graphics from natural image. Using the discrete haar wavelet transform in this approach is similar to applying an edge detection filter such as the Canny or Sobel filter. However, the time complexity for the Canny and Sobel filter is $O(N \log N)$ while the discrete Haar wavelet transform is $O(N)$, which is more efficient.

Juliet [22] and Wu [48] incorporated the discrete wavelet transform into a block based classification algorithm that decomposes an input image into 8×8 [22] or 16×16 [48] non overlapping blocks of pixels. The discrete wavelet transform is then applied to a single channel of each block. Statistical analysis is then performed on each of the sub bands after the forward discrete wavelet transform, by computing the standard deviation of each sub band.

The standard deviation, σ is the square root of the variance in a signal. The standard deviation for a two-dimensional signal such as a block of discrete wavelet coefficients is defined by:

$$\sigma_{\theta} = \sqrt{\frac{1}{W \times H} \sum_{i=0}^{W-1} \sum_{j=0}^{H-1} [F_{\theta}(i, j) - \mu_{\theta}]^2} \quad (9)$$

Where θ is a block of discrete wavelet transform coefficients and σ_{θ} is the standard deviation of a given 16×16 block of discrete wavelet transform coefficients. W is the width and H is the height of F_{θ} , where F_{θ} is the input block of coefficients. μ_{θ} is the statistical mean calculation of F_{θ} and is defined by:

$$\mu_{\theta} = \sqrt{\frac{1}{W \times H} \sum_{i=0}^{W-1} \sum_{j=0}^{H-1} [F_{\theta}(i, j)]^2} \quad (10)$$

The root squared mean is used used in equation (10) because the values of discrete wavelet transform coefficients can be both positive and negative.

The hypothesis is that computer generated data contains more uniform regions and sharp transient information than natural camera captured image, thus there should be a higher degree of variance within the sub bands of block that contains text or graphics, compared to blocks that contain natural camera captured data. By performing the discrete Haar wavelet transform on a single channel of a 16x16 block of pixels and then computing the standard deviation of each sub band, there should be a threshold value that can be found that will aid in classifying the type of data within a given block. The Hypothesis is tested and discussed further on in this thesis in section 9.1.

6 Compression

Data compression, in terms of digital processing is the process of reducing the amount of bits that is needed to represent a given quantity of information. There is a distinction to be made between data and information. With respect to digital images, the information is the content of the image, for example, a picture of the sky, a person, or some text, that a viewer of the image can read and get the information from. The data of the same image is the underlying array of numeric values that a computer processes to generate the image.

With the above distinction made, it can be shown that the same information can be described using different sets of data. Information that contains repeated patterns can be said to contain redundant data. As an example, consider an array of pixel intensity values that have been extracted from a 16x16 block of pixel values:

80	80	80	80	20	20	80	80	80	80	20	20	80	80	80	80
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

The same information can be represented as a series of run length pairs, where the run is equal to the value and the length is the amount of repetitions of the run:

80	4	20	2	80	4	20	2	80	4
----	---	----	---	----	---	----	---	----	---

The unit of information, named after the seminal author in the field of information theory, C.E Shannon [38] is also known as the bit. If the top array is denoted as b and the bottom array is denoted as b' the redundancy, R can be calculated with:

$$R = 1 - \frac{1}{C_r} \quad (11)$$

where C_r describes the compression ratio and is defined as:

$$C_r = \frac{b}{b'} \quad (12)$$

In this particular example the compression ratio is equal to 1.6:1 and the data redundancy = 0.375, which means that 37.5% of the data is redundant.

In image and video compression there are four distinct types of redundancy that can be exploited:

- Coding Redundancy
- Spatial Redundancy
- Temporal Redundancy
- Spectral Redundancy

Coding Redundancy

The symbols used to describe a body of information, such as characters, integers, bits etc, can be described as the code book for a medium. The information is described by sequences of symbols which are known as code words. With respect to digital images, an 8 bit value is used to describe the pixel intensity of a given channel in an image. It is highly unlikely that all 255 unique values associated with a code word with code length of 8 bits would be used within a single image.

As an example, consider a single channel from a 3 channel image, with intensity values from 0 to 255. a given intensity value can be treated as a discrete random variable and can be represented with r_k which lies in the interval $[0, L - 1]$ where L represents the range of intensity values available in a $M \times N$ Image, where M represents the rows and N represents the columns. The probability of each r_k occurring is represented with $P_r(r_k)$ and can be described by:

$$P_r(r_k) = \frac{n_k}{M \times N} \quad k = 0, 1, 2, \dots, L - 1 \quad (13)$$

Where n_k is the count of instances of value r_k and L is the amount of unique intensity values for an 8 bit code length. The number of bits (with respect to information carrying units) can be denoted with $l(r_k)$ and the average number of bits needed to represent each intensity value can be defined with:

$$L_{avg} = \sum_{k=0}^{L-1} l(r_k) P_r(r_k) \quad (14)$$

Figure 39 is a 16x16 pixel image with 3 colour channels. Each pixel intensity for each colour channel is described with an 8 bit value, code length = 8. Table 2 Shows the count of intensity values, r_k , for a single channel of the image along with the associated probability $P_r(r_k)$ and the 8 bit code word

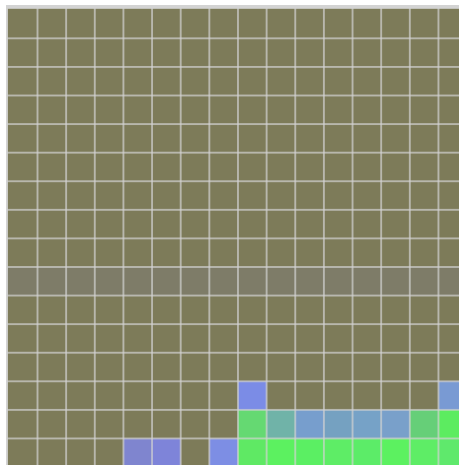


Figure 39: 16x16 block of sample values

r_k	value	count	$P_r(r_k)$	$l(r_k)$	code word
0	89	192	0.75	8	0101 1001
1	104	16	0.063	8	0110 1000
2	230	4	0.017	8	1110 0110
3	211	10	0.039	8	1101 0011
4	114	3	0.012	8	0111 0010
5	168	6	0.023	8	1010 1000
6	205	10	0.039	8	1100 1101
7	103	15	0.059	8	0110 0111

Table 2: frequency table of intensities, associated probabilities, code lengths and code words

Using equation (14), the average code length is equal to 8 bits, as expected as a fixed code length is used for each intensity and using the equation (12), the compression ratio, $C_r = 1:1$ which shows that there is no compression.

If the image in figure 39 was to be compressed in isolation, As there is only 8 unique intensities in the table, each r_k could be assigned a fixed code length $l(r_k)$ of 3 bits. using 12, the compression ratio $C_r = 2.67:1$ and $L_{avg} = 3$.

Instead of using a fixed length code book, a variable length prefix-free coding scheme, such as Huffman coding [16] could be used, which is presented in table 3

r_k	value	count	$P_r(r_k)$	$l(r_k)$	code word
0	89	192	0.75	1	1
1	104	16	0.063	3	010
7	103	15	0.059	3	001
6	205	10	0.039	4	0111
3	211	10	0.039	4	0110
5	168	6	0.023	4	0000
2	230	4	0.017	5	00011
4	114	3	0.012	5	00010

Table 3: Using variable length prefix free coding

using equation (14) to calculate the average bits, L_{avg} :

$$\begin{aligned}
 L_{avg} &= 0.75(1) + 0.063(3) + 0.059(3) + 0.039(4) + \\
 &\quad 0.039(4) + 0.023(4) + 0.017(5) + 0.012(5) \\
 L_{avg} &= 1.665bits
 \end{aligned}$$

Using the value calculated for L_{avg} , the number of bits needed to represent figure 39 can be found by $MxNxL_{avg}$ where $M = N = 16$

$$b' = 16 \times 16 \times 1.665 = 320$$

Where b' is the size of the data needed to represent the information in figure 39 after using variable length coding. using equation (12) to calculate C_r :

$$C_r = \frac{2048}{320} = 6.4 : 1$$

It can be observed from the above example, that using variable length coding can increase compression performance and reduce the amount of data needed to represent a body of information. Shannon [38] provides the mathematical framework that describes the lowest amount of data required to represent a body of information without loss. In his work, Shannon describes Entropy, H as the average information per source output and can be modelled, with respect to the previous example as:

$$H = - \sum_{k=0}^{L-1} P_r(r_k) \log_2(P_r(r_k)) \quad (15)$$

Spatial Redundancy

Images that contain structure and objects typically have high correlation in neighbouring pixel values. The gradient change in natural image tends to be slow, which suggests that the variance between neighbouring pixels may small. Images containing computer generated data are highly structured, with repeated patterns forming the structure, which means there is unnecessary replication within the data.

Temporal Redundancy

In successive frames in a sequence of images, the rate of change is slow, which means that there is high correlation between each successive frame, leading to unnecessary repetition in the data.

Spectral Redundancy

Spectral redundancy, or irrelevancy of information is a phenomena of the Human visual system. As the human visual system may not be able to acutely discern variance within colour information, the inclusion of data that the human visual system is unaware of can be considered as irrelevant data.

6.1 Lossless Compression

Lossless compression algorithms allow for the original data to be reconstructed exactly from the encoded data. In relation to compression of a computer screen, they are generally employed when it is important that the decompressed data must not contain compression artifacts such as "ringing" which is associated with Gibbs phenomenon. Areas containing such data would typically be classified as text and graphic areas.

Lossless compression algorithms can also be incorporated into lossy compression algorithms. An example of this is using Huffman coding[16] in the JPEG Standard, for losslessly compressing quantized discrete cosine transform coefficients.

Lossless compression algorithms exploit repeated patterns in the data to be encoded. Such repeated patterns can be found in large smooth areas of a computer screen where pixel values do not change, or for computer generated text or graphics using single colours and containing sharp contrasts, often in a linear fashion (text characters, window borders, etc). Because of these attributes, transform based compression algorithms, which generally tend to be lossy, are inefficient at compressing this type of data. As noted in [24], certain transform based encoding algorithms cannot compact the energy in the pixel values to low frequency components, but may spread it to high frequency bands.

Some of the repercussions of using lossless compression algorithms include lower compression ratios, typically between 1 and 5, as no quantization of data occurs, i.e, no information is discarded. As well as lower compression ratios, using lossless compression algorithms on unsuitable data can lead to data expansion, rather than compression. As such, it is very important to ensure that lossless compression algorithms are only used on suitable data. Lossless compression algorithms, such as Arithmetic Coding, can be computationally complex, and are bounded by the amount of floating point precision a computer handles.

This section introduces several lossless compression algorithms, the theory of operation and their implementation in code.

6.1.1 Run Length Encoding

Run Length Encoding is a lossless encoding algorithm, documented for application in television broadcasting in [35]. It has been subsequently standardised by the *ITU* recommendation T. 45,[20].

Run Length Coding is optimal on long repeated runs of the same number or character, as such it is suitable for areas of the screen that contain the same colour or that has some repeated pattern, an example of which can be seen in Figure40

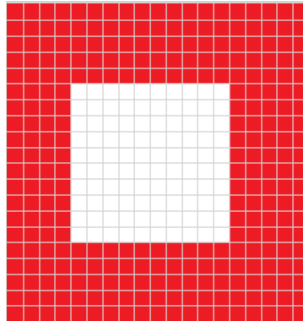


Figure 40: 19x20 3 channel 24 bit 3 computer generated image

Figure 40 shows a 3 channel 24-bit bitmap image. The 3 channels hold the red, green and blue pixel information. This type of image data is computer generated and is free from noise that may have resulted from electronically capturing continuous tone image (such as natural picture). As such, a classification algorithm may classify this type of data as either sparse or text (depending on the classification algorithm). The decomposition of figure 40 into its single channel components can be seen in figure 41

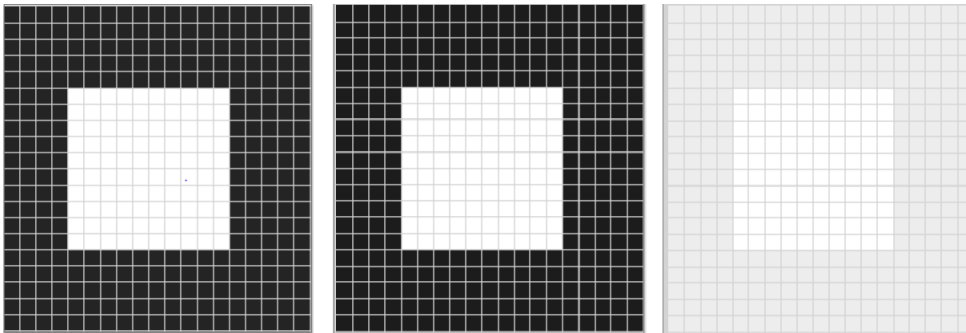


Figure 41: 3 Separate 8 bit channels representing Blue, Green and Red pixel intensity channels

Figure 41 shows the image in figure 40 separated into 3, 8-bit channels, each channel represents the contribution of Blue, Green and Red to the final image. Each pixel value in the channel can have a value between 0 and 255, 0 having no contribution and 255 having highest contribution, (when a pixel's three channels are 0, the resulting colour is black, when 255, the resulting pixel is white and if they are all the same value, they are a shade of grey). The higher the value means the more contribution that colour channel has to the overall image. It can be observed from Figure 41 that the red channel, (the block on the far right), has the highest contributing factor to the image, where the blue and green channel have a low contribution.

To illustrate how the Run length encoding algorithm works, Listing 1 shows a run of the first row of the red channel in figure 41 taken in isolation

Listing 1: Run Length Encoding Psuedo Code

```
1 //data to be encoded
2 Row_0[19] = {241, 241, 241, 241, 241, 241,
3             241, 241, 241, 241, 241, 241, 241, 241,
4             241, 241, 241, 241, 241}
5
6 // run_Length stores the length of the run,
7 // increments when the next byte in the data is the same as the previous byte
8 run_Length =1;
9 //index is used to iterate through the data
10 index;
11 // next_byte used to fetch the next number from the data
12 next_byte
13 // previous_byte is used to hold the value of the run
14 previous_byte
15 while(get another byte from row_0)
16 {
17     previous_byte = next_byte;
18
19     next_byte=row_0[++index];
20     if(next_byte ==previous_byte)
21     {
22         run_Length++;
23     }
24     write run_length;
25     write previous_byte;
26 }
27 Encoded_File[] = {19 241}
```

The above listing shows the Pseudo-code of an implementation of a Run Length encoder. The encoded output would be just two numbers. giving a compression of close to 10. The above example demonstrates that run length encoding is suitable for computer generated data that has a limited amount of colours and repetitive patterns.

When implementing a run length encoder, it is important to consider what type of variable that is used to store the run length. For instance, if an unsigned char is used to store the run length, the maximum length of the recorded run would be 255, in which case it would restart and increment again. On the other hand, if an integer is used, the run length could be over 4 billion, which has a low probability of occurrence. An integer requires 4 bytes to store the value, while an unsigned char requires a single byte, so it makes sense to use shorter variables for data that contains runs that may not be that long, to reduce redundancy on the encoded data.

6.1.2 LZ 77 Algorithm

LZ77[51], is a lossless and dictionary based compression algorithm that is from the Lempel-Ziv family of compression algorithms. Along with LZ78[52], it is the basis of many variants of compression algorithms used in modern compressors. For example, LZ77 is used in the DEFLATE[7] algorithm in GZIP and it is also Incorporated with a Markov chain algorithm to form the basis of LZMA which is used in 7ZIP which is a modern compression algorithm. It is one of the first compression algorithms to implement the concept of a 'sliding window'

Taking a data stream 'X'

$$X = TO\ BE\ OR\ NOT\ TO\ BE?$$

The algorithm works by splitting the data stream into two segments: The Search Buffer and the Lookahead Buffer. Typically, the search buffer will contain thousands of symbols, while the lookahead buffer will contain significantly less symbols that have not been seen yet.

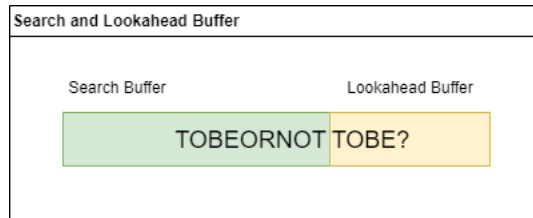


Figure 42: Search and Lookahead Buffer

The encoder reads a symbol from the lookahead buffer and will attempt to find a match for it in the search buffer (Figure 42). If the symbol is found, the encoder will go to read more symbols from the lookahead buffer and continue searching backwards in the search buffer until a match is found (Figure44). When a match is settled on, the encoder will output a token.

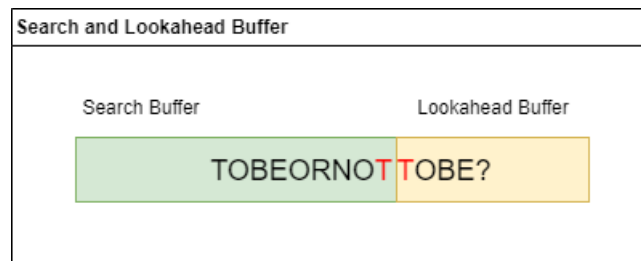


Figure 43: Found match for symbol from the lookahead buffer

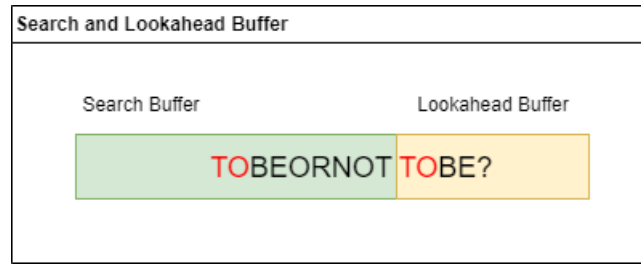


Figure 44: Longest matching string

The Token comes in three parts:

$$[Offset, Length, NT]$$

Offset is the distance to the match from the window, *length* is the length of the match in symbols and *NT* is the next token, which is the next symbol in the lookahead buffer. The token for the example above would be:

$$[9, 2, B]$$

As soon as the encoder is finished outputting the token, the window gets shifted to the right, hence the term 'Sliding Window'

6.1.3 Lempel Ziv Welch Encoding

First published in 1984 by Welch [46], the Lempel-Ziv-Welch (LZW) is a lossless data compression algorithm, which was designed to be an improvement on Abraham Lempel and Jacob Ziv's LZ78, [52], compression algorithm. It is a mature compression algorithm and has been proven to be a successful method of compressing graphical content as it is implemented in the GIF image format which has been widely used to compress content with smooth regions, sharp transitions and few colours, optimally.

The LZW algorithm can be classified as a dictionary coder. Dictionary coders take advantage of data that has low entropy, containing many repeated patterns, such as smooth areas of a screen or large text documents. It associates sequences of data with indexes, (which occupy less space), in a table. The compressor works by substituting repeated sequences with the corresponding index and outputting the index to the stream. The table grows as the data comes in. the longer the sequences replaced and the higher the frequency of occurrence, the better the compression will be.

The LZW coder works by building up a table of code words which are a combination of symbols read in from an input and incrementally outputs the index of these code words to the output stream. In [46], Welch refers to each code word as a **STRING** which are stored in a dictionary or table and the symbols that are input from the data set are stored in a **CHAR**. It reads the input stream a byte at a time. Char's

are 8 bit Bytes, while strings are variable length arrays of char and with a maximum amount of strings equal to the size of the dictionary-1. As an example, using a 16 bit table can store 65535 code words. Each string has an index which is fixed length, which is the amount of bits used for the table. A larger table can store more store strings with the possibility of better compression, but tables that are too large will have diminishing returns as the width of the indexes get wider as more bits get used for the table. For instance, each index output to the encoded stream for a 64 bit table would be 64 bits wide, and the table itself would keep growing using up too much memory resources.

Another feature of the LZW coder is that it resets to an initial condition, once the table has been filled with strings. the initial condition is to populate the table with every one byte entry. The LZW coder can then be said to be adaptive as it does not need any prior knowledge of the data to be compressed to build its table of code words. the decoder is also adaptive, all it needs to know is the size of the Encoder table to rebuild it and to fully decode the message.

The Pseudo code for the Lempel Ziv Welch encoder can be seen in Listing 2.

Listing 2: Lempel Ziv Welch Encoding Psuedo Code

```

1  TABLE      : Table of Codes with unique indexes
2  STR         : empty string
3  CHAR       : byte
4  IS         : Input Stream
5  EF         : the encoded file
6
7  TABLE = all single byte entries with unique INDEX; //Initialise 0-255
8
9  while ( read byte from IS into CHAR )
10     {
11         STR = STR + CHAR // append The Byte to STR
12         if (STR + CHAR is not in TABLE )
13             {
14                 Table[STR] = ++INDEX; // increment the size of the table.
15                 STR = STR - LB; // remove the last byte from STR.
16                 WRITE TABLE[STR] to EF; // TABLE[STR] returns the unique
17                                     // INDEX for each code in the table.
18                 STR = CHAR; // STR is set equen to CHAR
19                                     // and is only a Byte in Length.
20             }
21     }
22 if (STR is not empty)
23     {
24         WRITE TABLE[STR] to EF;
25     }

```

To illustrate the Lempel-Ziv Welch encoding process an example will be demonstrated on the sentence, "the_rain_in_Spain_falls_mainly_on_the_plain_". The sentence contains 44 characters, uncompressed this would be 44 indexes written to the output stream. A full run of the LZW coder can be seen in Table 6. The input characters and associated ASCII values can be seen in Table 4 and the output stream can be seen in Table 7.

t	h	e	-			
116	104	101	95			
r	a	i	n	-		
114	97	105	110	95		
i	n	-				
105	110	95				
S	p	a	i	n	-	
83	112	97	105	110	95	
f	a	l	l	s	-	
102	97	108	108	115	95	
m	a	i	n	l	y	-
109	97	105	110	108	121	95
o	n	-	t	h	e	-
111	110	95	116	104	101	95
p	l	a	i	n	-	
112	109	97	105	110	95	

Table 4: Input Stream in characters and ASCII decimal value

ITR	ITERATION: each read of the input stream
CHAR	next byte to be read from input stream
STR + CHAR	operation appending the most recent byte read in from the input stream
STR	STRING: Character array with an index
NEW STRING	Operation set STRING to current Byte read from the input stream

Table 5: Legend

ITR	CHAR	STR + CHAR	IN TABLE?	OUTPUT	ADD To TABLE	INDEX	NEW STRING
-----	------	---------------	--------------	--------	-----------------	-------	---------------

1	t	t	yes	NOP	no		t
2	h	th	no	t	yes	256	h
3	e	he	no	h	yes	257	e
4	-	e_	no	e	yes	258	-
5	r	_r	no	-	yes	259	r
6	a	ra	n	r	yes	260	a
7	i	ai	no	a	yes	261	i
8	n	in	no	i	yes	262	n
9	-	n_	no	n	yes	263	-
10	i	_i	no	-	yes	264	i
11	n	in	262	NOP			in
12	-	in_	no	262	yes	265	-
13	S	_S	no	-	yes	266	S
14	p	Sp	no	S	yes	267	p
15	a	pa	no	p	yes	268	a
16	i	ai	261	NOP			ai
17	n	ain	no	261	yes	269	n
18	-	n_	263	NOP			n_
19	f	n_f	no	263	yes	270	f
20	a	fa	no	f	yes	271	a
21	l	al	no	a	yes	272	l
22	l	ll	no	l	yes	273	l
23	s	ls	no	l	yes	274	s
24	-	s_	no	s	yes	275	-
25	m	_m	no	-	yes	276	m
26	a	ma	no	m	yes	277	a
27	i	ai	261	NOP			ai
28	n	ain	269	NOP			ain
29	l	ainl	no	269	yes	278	l
30	y	ly	no	l	yes	279	y
31	-	y_	no	y	yes	280	-
32	o	_o	no	-	yes	281	o
33	n	on	no	o	yes	282	n
34	-	n_	263	NOP			n_
35	t	n.t	no	263	yes	283	t
36	h	th	256	NOP			th
37	e	the	no	256	yes	284	e
38	-	e_	258	NOP			e_
39	p	e_p	no	258	yes	285	p
40	l	pl	no	p	yes	286	l

41	a	la	no	l	yes	287	a
42	i	ai	261	NOP			ai
43	n	ain	269	NOP			ain
44	-	ain_	no	269	yes	288	-
45	EOF			-			

Table 6: LZW Encoder Run

116	104	101	196	114	97	105	110	196	262
196	83	112	261	263	102	97	108		
108	114	196	109	269	108	121	196	111	
263	256	258	112	108	269	196			

Table 7: Output Stream

Table 5 explains the abbreviations used in the LZW encoder run.

Following the psuedo code for the LZW encoder and reading Table 6 from left to right:

1. 't' is stored in char. char is added to string (which is empty). table is searched and entry found at table[t] = 104 't' is not added to table. no output to encoded file. string is set to 't'.
2. 'h' is stored in char. char is appended to string 'th'. table is searched and no entry is found. 'th' is added to table with index 256. truncate the last byte from 'th' and output table[t]116. string is set to 'h'.
3. 'e' is stored in char. char is appended to string 'he'. table is searched and no entry is found. 'he' is added to table with index 257. truncate the last byte from 'he' and output table[h]104. string is set to 'e'
4.
5.
6.
7.
8.
9.

10. "i" is stored in char. char is appended to string = *.i*. table is searched and no entry is found. ' *i* ' is added to table with index value 264. Truncate the last byte from ' *i* ' and output table[*i*] = 196. string is set to ' *i* '
11. ' n ' is stored in char. char is appended to string = 'in'. table is searched and entry is found at at table[in] = 262. in is not added to table. no output to encoded file. string is set to 'in'

The run through the LZW coder above illustrates how the encoder stores new code words in the table and how it outputs the unique index for every new code word. The final output stream is shown in Table 7. the numbers is bold are the new indexes created by the LZW coder. The size of the array is 34 vs 44 for the uncompressed input, which shows a compression ratio of 1.29 if the data set is larger, and contains more repeated patterns, as an example a 16x16 block of pixels, the compression ratio would increase.

The pseudo code for the decoder can be seen in Listing 3.

Listing 3: Decoder Pseudo code

```

1  TABLE      : Table of Codes with unique indexes
2  STR         : empty string
3  K           : Holds an INDEX read in from the encoded file
4  EF         : the encoded file
5  DS         : Decoded Stream
6
7
8  TABLE = all single byte entries with unique INDEX; // 256 entries to begin
9
10 while ( could read a new INDEX from EF into k )
11 {
12     if (k > TABLE size)
13     {
14         cannot decode
15     }
16 else
17     if (k = TABLE size)
18     {
19         TABLE[INDEX+1] = STR +First Byte of STR;
20     }
21
22 WRITE TABLE[k] to the output stream;
23
24 }
```

The decoder works inversely to the encoder. It takes in an index, builds the exact same table that the encoder uses and outputs the original message. As stated, the

decoder just needs to know the the maximum size of the original table to correctly decode the message.

There are some conditions for the decoder to work which can be seen in the Pseudo code above. The first condition is that the decoder cannot decode an index that is larger than the table in its present state. This is the main reason why the encoder encodes the message incrementally, where the next code to be indexed will never exceed the size of the table. A full run of the decoder can be seen in Table 8.

ITR	STR	READ [K]	TABLE [k]	STR + K[First Byte]	INDEX	OUTPUT TABLE [K]	NEW STR
1	empty	116	t	t	116	t	t
2	t	104	h	th	256	h	h
3	h	101	e	he	257	e	e
4	e	196	-	e_	258	-	-
5	-	114	r	_r	259	r	r
6	r	97	a	ra	260	a	a
7	a	105	i	ai	261	i	i
8	i	110	n	in	262	n	n
9	n	196	-	n_	263	-	-
10	-	262	in	_i	264	in	in
11	in	196	-	in_	265	-	-
12	-	83	S	S_	266	S	S
13	S	112	p	Sp	267	p	p
14	p	261	ai	pa	268	ai	ai
15	ai	263	n_	ain	269	n_	n_
16	n_	102	f	n_f	270	f	f
17	f	97	a	fa	271	a	a
18	a	108	l	al	272	l	l
19	l	108	l	ll	273	l	l
20	l	114	s	ls	274	s	s
21	s	196	-	s_	275	-	-
22	-	109	m	_m	276	m	m
23	m	269	ain	ma	277	ain	ain
24	ain	108	l	ainl	278	l	l
25	l	121	y	ly	279	y	y
26	y	196	-	-y	280	-	-
27	-	111	o	_o	281	o	o
28	o	263	n_	on	282	n_	n_
29	n_	256	th	n.t	283	th	th
30	th	258	e_	the	284	e_	e_
31	e_	112	p	e_p	285	p	p
32	p	108	l	pl	286	l	l
33	l	269	ain	plai	287	ain	ain
34	ain	196	-	ain_	288	-	-

Table 8: LZW Decoder Run

6.1.4 Huffman Encoding

Huffman coding is a form of lossless compression that uses variable length codes to represent the numbers or symbols of a data set. The process is to replace the most frequently seen symbols with smaller codes than the symbols that are not seen frequently.

To implement Huffman coding, the data first needs to be pre-processed. The frequency of all symbols is recorded and the probability of each symbol is calculated and assigned to create a code dictionary table. The table is organised such that the symbol with the highest frequency is assigned the greatest probability and is stored at the top of the table with symbols with lower frequency stored beneath. To ensure that the decoder is able to correctly decode the new code words generated by the Huffman encoding process, each new code word for a symbol must be prefix free. This means that a given symbol with a higher frequency cannot be a prefix for a symbol with a lower frequency.

124	116	148	148	148	148	116	148
116	148	148	148	148	148	132	148
132	148	108	148	148	148	132	148
124	148	148	148	148	148	132	148
132	148	148	132	148	148	132	148
124	148	148	148	132	148	156	148
108	148	148	148	132	148	116	148
124	148	148	148	148	156	164	148

Figure 45: 8x8 block of pixel values

Figure 45 represents an 8x8 block of pixel intensity values that will be processed with

the Huffman encoding algorithm. Each pixel intensity value is stored using 8 bits giving a dynamic range of 0-255.

r_k	value	frequency	$P_r(r_k)$	$l(r_k)$	code word
0	148	46	$\frac{46}{64}$	8	1001 0100
1	132	9	$\frac{11}{64}$	8	1000 0100
2	116	3	$\frac{3}{64}$	8	0111 0100
3	124	2	$\frac{5}{64}$	8	0111 1100
4	108	2	$\frac{11}{64}$	8	0110 1100
5	156	2	$\frac{1}{64}$	8	1001 1100

Table 9: Counting the frequency of pixel intensity values from figure 45

Table 9, shows the amount of unique symbols (where symbols is used to describe pixel intensity values) in figure 45. It can be observed that there are 6 unique symbols in the table. If figure 9 was to be compressed in isolation, then the code redundancy could be reduced by representing each symbol in table 9 with 3 bits instead of 8 bits. Information about a mapping of the code words to the intensity values could then be provided in meta data that would be sent to the decoder.

using equation (12) to calculate the compression ratio when the average code length for each symbol is 3 bits would give $C_r = 2.67 : 1$.

To encode the pixel intensity values of figure 45 using variable length Huffman encoding, the first step is to create a Huffman table which sorts the symbols by frequency in descending order. The Huffman table for figure 45 can be seen in table 10.

Symbol	frequency
148	46
132	9
116	3
124	2
108	2
156	2

Table 10: Huffman table for figure 45

The Huffman algorithm process is to combine the two lowest frequencies among in the table to form a new node that has two children. Of the children, the symbol that has the higher frequency gets assigned a "0", and the symbol with the lower frequency gets assigned a "1". If both symbols have the same frequency, the symbol that is higher in the table gets assigned a "0". The new nodes frequency gets inserted into the table, again in descending order. If the new nodes frequency is equal to current frequencies in the table, it gets inserted to the top of the symbols with the

same frequency. This process is repeated until there is only a single node, which is known as the "root" of the Huffman tree. When the root of the tree has been found, the variable length codes for each symbol in the dictionary are created by tracing the path back from the root to the symbols original frequency. The Huffman tree process is shown in figure 46

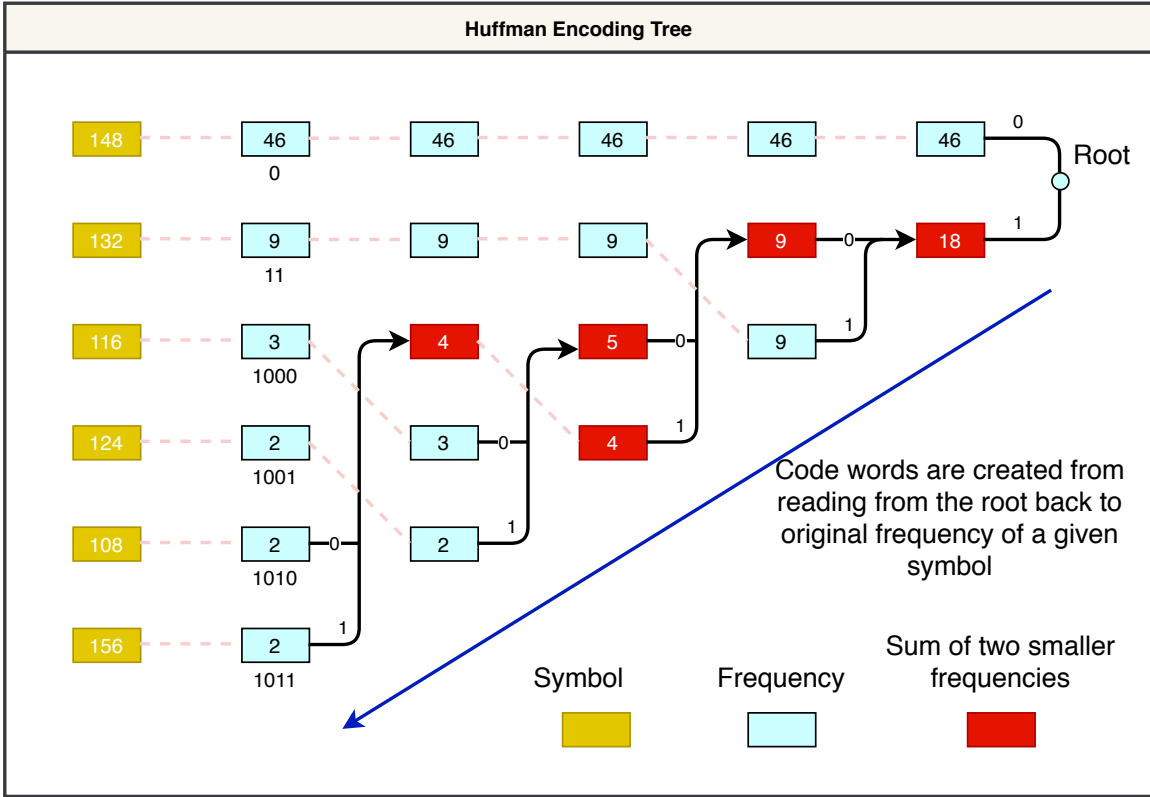


Figure 46: Huffman tree encoding process for figure 45

The new code book for figure 45 can be generated from the Huffman tree in figure 46. The new code book can be seen in table 11

Symbol	Frequency	Code
148	46	0
132	9	11
116	3	1000
124	2	1001
108	2	1010
156	2	1011

Table 11: Variable length Huffman code book for figure 45, generated from figure 46

It can be observed from table 11 That the symbol with the highest frequency is assigned a single bit for encoding and that no code is a prefix of a subsequent code

in the table.

Using equation (14) to calculate the average bits per symbol, L_{avg} the average bits needed is :

$$L_{avg} = \frac{46}{64}(1) + \frac{9}{64}(2) + \frac{3}{64}(4) + \frac{2}{64}(4) + \frac{2}{64}(4) + \frac{2}{64}(4)$$

$$L_{avg} = 1.565bits/symbol$$

And using equation (15) to calculate the entropy, H :

$$H = - \left[\left(\frac{46}{64} \right) \log_2 \left(\frac{46}{64} \right) + \left(\frac{9}{64} \right) \log_2 \left(\frac{9}{64} \right) + \left(\frac{3}{64} \right) \log_2 \left(\frac{3}{64} \right) \right.$$

$$\left. + \left(\frac{2}{64} \right) \log_2 \left(\frac{2}{64} \right) + \left(\frac{2}{64} \right) \log_2 \left(\frac{2}{64} \right) + \left(\frac{2}{64} \right) \log_2 \left(\frac{2}{64} \right) \right]$$

$$H = 1.421bits/symbol$$

It can be observed that the variable length codes that have been generated from the Huffman tree in figure 46 are close to the bounds of entropy. However to correctly decode the code book, the meta data that must be sent with the coded data can be quite large for data that has larger code books. Compression algorithms such as JPEG [18] and PNG [13], use a form of Huffman encoding known as canonical Huffman coding that reduces the amount of meta data that must be sent with the encoded data.

To perform Canonical Huffman coding, some further steps must be taken. The first step is to reorganise the Huffman table in figure 10 to be organised first by the number of bits that is needed for each symbol's Huffman code and then in lexicographical order. This means that symbols with code words that have the lowest number of bits go to the top. Symbols with codewords that have the same amount of bits are then ordered by value, in ascending order, that is lowest to highest. This is represented in table 12

Symbol	No of bits.
148	1
132	2
116	4
108	4
124	4
156	4

Table 12: Huffman table for figure 45 in lexicographical order

Once the Huffman table is organised in lexicographical order, new code words for each symbol is generated. To generate new codes, a binary increment and zero padding

is used. The first symbol in table 12 needs one bit for encoding. it is assigned "0". the second symbol in table 12 needs two bits for encoding. It binary increments the value assigned to code symbol one and appends a zero to the end because two bits are needed to encode symbol two. Table 13 shows the canonical Huffman codes generated for figure 45

Symbol	bits	Canonical Code	comment
148	1	0	one bit needed to encode, assigned 0
132	2	10	binary increment symbol one code, pad with one "0"
116	4	1100	binary increment symbol two code, pad with 2 "0"
108	4	1101	binary increment symbol three code, no pad
124	4	1110	binary increment symbol four code, no pad
156	4	1111	binary increment symbol five code, no pad

Table 13: Canonical Huffman code table for figure 45

It can be observed from the new canonical Huffman codes in table 13 that the new code words are still prefix free. One of the main advantages of applying canonical codes is in the decoding process. Because the symbols are known, the amount of bits needed to encode each symbol is known and that the symbols are ordered in a lexicographical order, the actual code words do not need be transmitted as meta dat to the decoder as it can generate the code words itself. This reduces the amount of meta data that needs to be transmitted, improving the coding performance of the algorithm.

Listing 4: Canonical Huffman Decoder Pseudo Code

```

1 decode(symbolList[], symbolBitsCount[], encodedDataChunk, streamPosition){
2     int codeLength=0;
3     int code = 0;
4     int first = 0;
5     int index = 0;
6     bool bit;
7     int count;
8     uint16_t bitsToShift = 32768 // 1000 0000 0000 0000
9
10    for(codeLength =1; codeLength <15; codeLength++)
11    {
12        bit = (bitsToShift & encodedDataChunk) // extract a bit from the bitstream
13        code |= bit; // logically or it to code;
14        bitsToShift >>= 1; // move to the next bit of the bitstream
15        streamPosition++; // keep track of bit in the bitstream
16        count = symbolBitsCount[codeLength];
17        if(code -count <first)
18            return symbolList[indexs +(code -first)];

```

```

19     index += count;
20     first += count;
21     first <<= 1;
22     code << =1;
23 }
24 return outOfBits; // not enough bits to process the next symbol
25 }

```

Listing 4 shows the pseudo code of implementing the decoder for canonical Huffman encoding. The decode function takes the list of symbols, the amount of bits each symbol needs to be encoded, a chunk of encoded data to decode and locator to keep track of where in the encoded stream the program is. To show how the decoder works, the first row of figure 45 will be encoded and decoded. let X equal a row of 45:

$X =$	132	148	108	148	148	148	132	148
-------	-----	-----	-----	-----	-----	-----	-----	-----

Using the canonical codes that have been generated in table 13, the code words used for X are:

Code Words =	10	0	1101	0	0	0	110	0
--------------	----	---	------	---	---	---	-----	---

and the bitstream is 10011010001100

The meta data that is sent to the decoder first is the list of symbols and the amount of bits that each symbol needs:

symbolList[]=	148	132	116	108	124	156
----------------	-----	-----	-----	-----	-----	-----

The array that holds the count of the amount of bits is:

symbolBitsCount[]=	0	1	1	0	4
---------------------	---	---	---	---	---

This describes the amount of bits for a symbol based on its location in the array. Reading the array from left to right, location[0] is not used. There is one symbol that uses one bit for encoding. There is one symbol that uses two bits for encoding. There are no symbols that use three bits for encoding and there are four symbols that use four bits for encoding.

The following steps are taken when decoding a canonical Huffman encoding bitstream:

1 first iteration through for loop

- bitsToShift = 1000 0000 0000 000

- $\text{encodedDataChunk} = 1001\ 1010\ 0011\ 00$: $\text{streamPosition} = 1$
- $\text{code} = 0000$: $\text{first} = 0000$: $\text{index} = 0$: $\text{codeLength} = 1$
- $(\text{bitsToShift} \& \text{encodedDataChunk}) = 1000\ 0000\ 0000\ 000$: $\text{bit} = \text{true} = 1$.
- $\text{code} = (\text{code} | \text{bit}) = 0001$
- $\text{bitsToShift} \gg 1 = 0100\ 0000\ 0000\ 000$: streamPosition is incremented to keep track of what state the decoder is in.
- $\text{count} = \text{symbolBitsCount}[1] = 1$.
- $(\text{code} - \text{count}) = 0$: $\text{first} = 0$: $(\text{code} - \text{count})$ is not less than first .
- $(\text{index} + \text{count}) = 1$: $(\text{first} + \text{count}) = 0001$: $\text{first} \ll 1 = 0010$: $\text{code} \ll 1 = 0010$

2 Second iteration through for loop

- $\text{code} = 0010$: $\text{first} = 0010$: $\text{index} = 1$: $\text{codeLength} = 2$
- $(\text{bitsToShift} \& \text{encodedDataChunk}) = 1000\ 0000\ 0000\ 000$: $\text{bit} = \text{false} = 0$.
- $\text{code} = (\text{code} | \text{bit}) = 0010$
- $\text{bitsToShift} \gg 1 = 0010\ 0000\ 0000\ 000$: streamPosition is incremented to keep track of what state the decoder is in.
- $\text{count} = \text{symbolBitsCount}[2] = 1$.
- $(\text{code} - \text{count}) = 1$: $\text{first} = 2$: $(\text{code} - \text{count})$ is less than first .
- $\text{return symbolList}[1+(2-2)] = \text{symbolList}[1] = 132$: correctly decoded symbol

3 decoded first symbol, return from function and start process again

- returned values: $\text{decoded symbol} = 132$: $\text{streamPosition} = 3$

The above process is carried out until the end of the bitstream, each time a symbol from the symbol list is found, the function returns that value along with correctly incrementing the marker that correctly identifies where in the bitstream the decode function finished. This value will be passed to the decode function again on the next iteration.

both PNG [13] and JPEG[18] limit the amount of bits for the length of code words to 15. Which is the limit that has been applied in this research too.

6.1.5 Arithmetic Encoding

Arithmetic coding is a lossless compression algorithm. It is typically implemented as an entropy encoding technique after pre processing with other compression algorithms. Like Huffman coding, the concept is to represent frequently seen symbols using fewer bits, compared to less frequently seen symbols.

One of the benefits of Arithmetic coding over Huffman encoding is it uses an adaptive model which can be tailored to the input stream to achieve a higher compression ratio than Huffman coding. A downside to implementing Arithmetic coding is that the computational requirement may be higher than that of Huffman coding.

Arithmetic coding is a precursor to binary arithmetic coding used in JPEG2000 and also to the Context Adaptive Binary Coding (CABAC) that is implemented in the video coding standards H.264 and HEVC.

First, a source alphabet, X , is defined as:

$$X = \{0, 1, 2\}$$

X is an alphabet of numerals, but X can be any finite set, such as characters or bytes etc.

The next step is to get the probability mass function. In this ideal example, the probabilities of each number are known and defined from the start, but in a practical implementation, a model is used to compute the probability of each symbol of the set. A good model should represent the input stream efficiently to achieve good compression ratios. The separation of the model from the coding implementation also allows for the method to be made adaptive to different types of content that makes up the input stream.

The probability mass function, P , is defined as:

$$P = (P_0, P_1, P_2) = (.2, .4, .4)$$

The example will code a three number sequence, X_3 as:

$$X_3 = 210$$

The algorithm can be broken up into two conceptual parts. The first part takes the sequence to be coded and will associate to it a sub interval of the interval between zero and one.

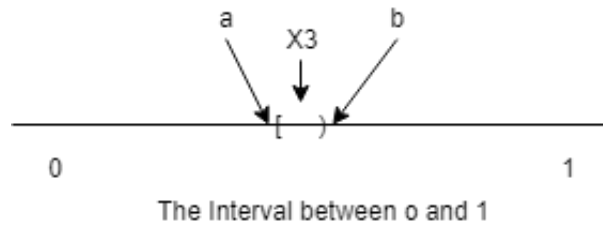


Figure 47: The message as a sub-interval of the interval 0-1

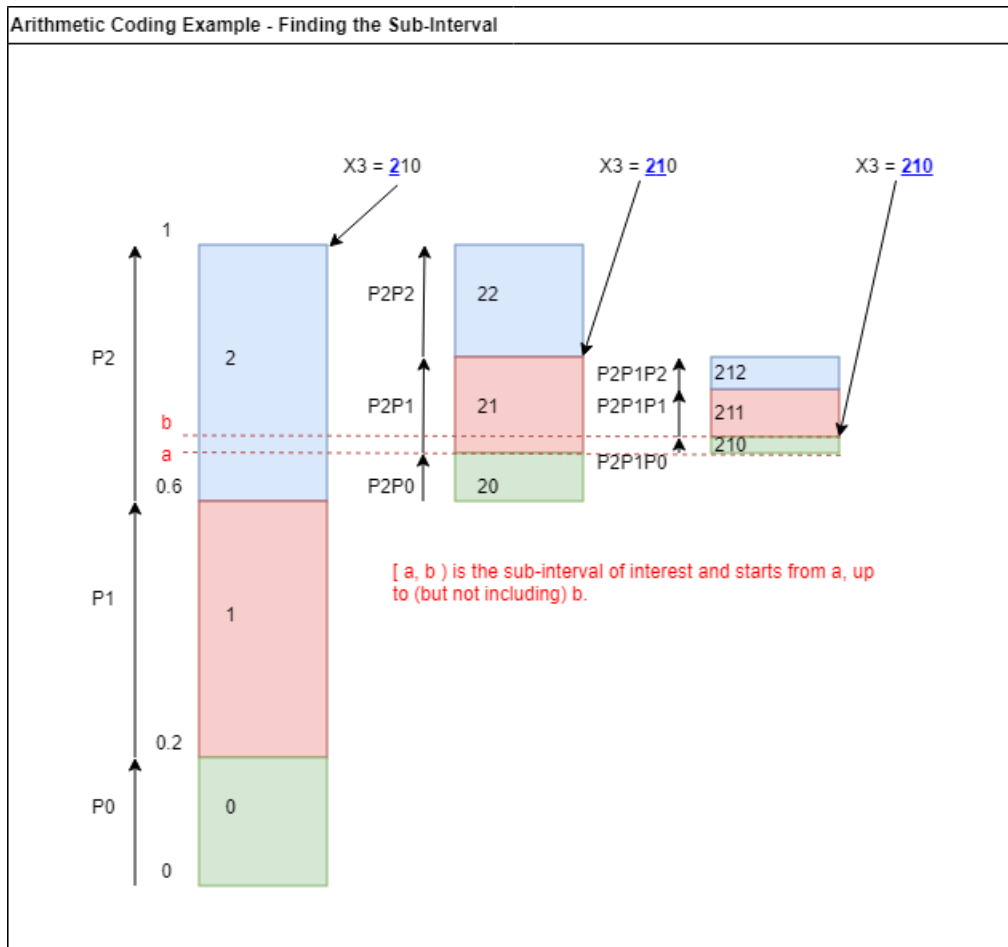


Figure 48: Finding the Sub-interval on the line zero to one

In Figure 48, The Interval between 0 and 1 is sub divided according to the probability mass function P . The first number in the sequence is 2, so the blue shaded region representing the sub division associated with P_2 gets further sub divided, again according to the probability mass function. The new height of each section is the product of P_2 times the respective probability of each section, which gives P_2P_2 , P_2P_1 and P_2P_0 . adding the next number of the sequence yields 21, so the same step as before is applied to the pink region (which is of height P_2P_1).

Finally, the last number in the sequence is added and the previous step is applied again. this time the whole message is contained in the green shaded area (of height $P2P1P0$). The sub-interval of interest is then seen on the the interval between 0-1 from a up to (but not including b).

To solve for a and b:

$$a = 0.6 + P2P0 = 0.68$$

$$b = a + P2P1P0 = 0.712$$

So, the sequence will be contained in the interval $[a, b)$, which goes from 0.68 to 0.712 on the interval between 0 and 1.

The next part of the encoding process uses binary splits to divide up the interval until a segment is found that lies completely between $[a, b)$.

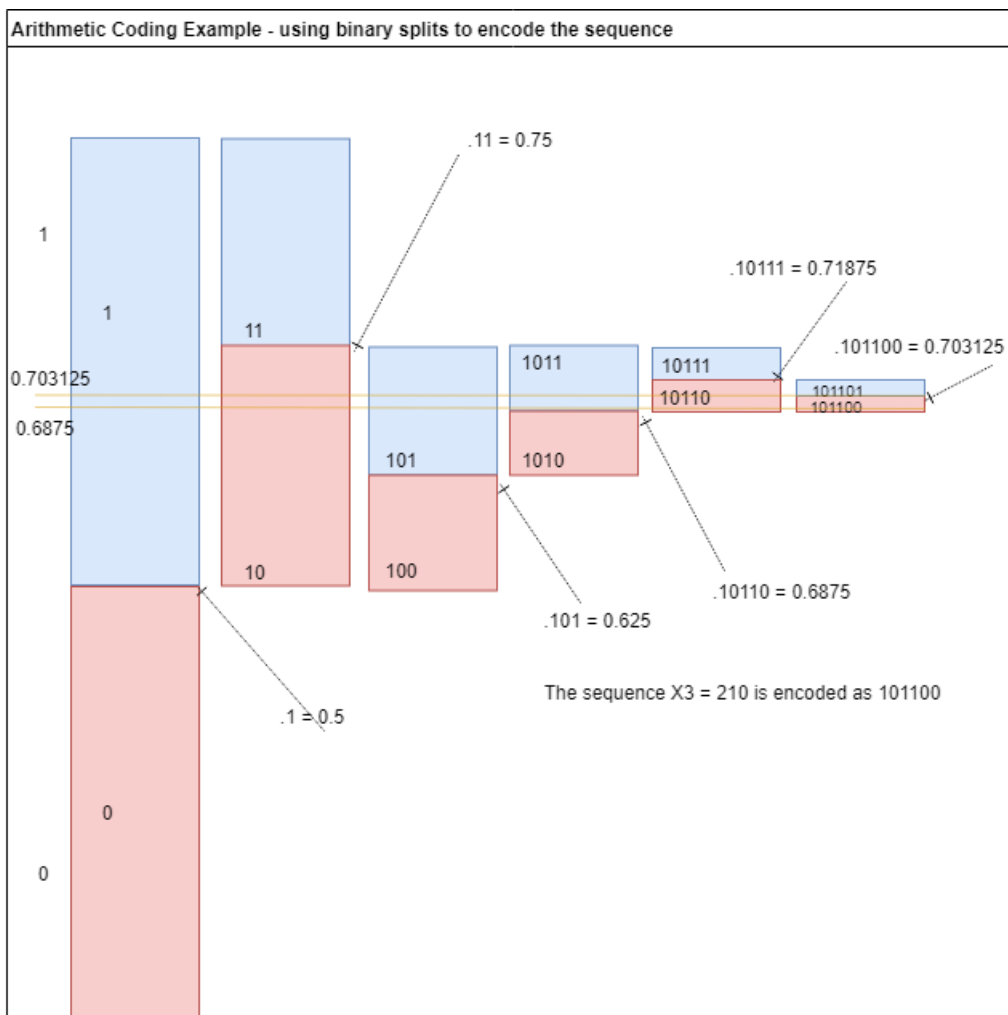


Figure 49: Using binary splits to encode the sub interval $[a, b)$

This part makes use of binary expansions, such as:

$$.1101_2 = \frac{1}{2^1} + \frac{1}{2^2} + \frac{0}{2^3} + \frac{1}{2^4}$$

which will give a real number between zero and one. The binary split can be seen in Figure 49. To ensure that the sequence is uniquely encoded, the line must be subdivided until a sub interval is found that lies completely inside the interval $[a, b)$ which previously has been solved for.

Decoding

Two methods can be used to ensure the message can be decoded successfully. The first method is to encode the message with the length included in it. This ensures that the inverse process can take place easily but has the problem of creating extra overhead. The Second method involves using End of File Symbols in the message. An End of File symbol should be a unique symbol that is only used to signify the end of a sequence.

in the example above, if 0 was chosen to be an end of file symbol, the decoder could recover the original message quite easily. Examples of valid sequences using zero as an End of File symbol can be seen in the table.

Valid	Not Valid
120	12041
213120	1221

Table 14: Valid and Not Valid messages using End of File Symbol, 0

Arithmetic coding takes an input file (can be made up of any type of symbols) and converts it to a floating point number which is between the interval of zero up to 1. Essentially, the whole file to be encoded is a single floating point number, which introduces problems. end of file symbols or message lengths will need to be incorporated.

The second issue is that normal data structures will no longer be able to represent the data, so a new way of representing this data will be needed.

The third issue is that arithmetic coding relies on a model to describe the input message. The role of the model is to relay to the encoder, the probability of each symbol in a message which means that arithmetic coding is only as good as its model. When the model relays an accurate probability of the symbols in a message, it will be encoded very close to full entropy. On the other hand, if the model misjudges the probabilities of a symbol, the encoder can expand the message rather than compress.

6.2 Lossy Compression

Lossy compression is a method of encoding that discards some of the data that is used to describe a body of information in favour of reducing the size of the file for storage and transmission. A decoded message after being compressed using a lossy encoder is an approximation of the original message, once the original data has been discarded, there is no way to retrieve it.

Lossy compression algorithms aim at reducing the spectral redundancy, or irrelevant information contained in a body of information. With respect to image compression, lossy compression algorithms will aim to reduce the amount of unique pixel values contained in an image, based on how the human visual system can perceive colour, or can process objects in the foreground of an image with higher fidelity than objects in the background, based on how the human visual system processes information within an image.

6.2.1 Image Quality Assessment

An effect of discarding data from an image in the processing stage is the resulting distortion in the decoded image. Some distortion may not have an impact on the end user, such as reduced amounts of individual shades in a picture of the sky, or the ordering of pixels in a picture of a sand. However, some distortion, in particular around structured information in an image, such as text, will have a negative impact on the end user. In order to gauge the quality of a compression algorithm, metrics are needed. Subjective methods such as screening tests can be very expensive and time consuming, involving many subjects and analyst's to review the data, so objective methods to assess the quality of an image are necessary for testing.

Standard objective image quality assessment metrics that are common include Root Mean Square Error (RMSE) and Peak Signal to Noise Ratio (PSNR). Both of these metrics are classed as full reference metrics, which compare the error in the signal in a processed image against a reference image, where the reference image can be quantitatively said to be error free.

Root Mean Square Error

Given a sample from a reference image $f(x, y)$ and a sample from the same image after processing, $\hat{f}(x, y)$ the error $e(x, y)$ can be described with

$$e(x, y) = \hat{f}(x, y) - f(x, y) \quad (16)$$

To measure the strength of the error between an image and a processed image with a resultant positive value, the Root Mean Square Error can be described with:

$$RMSE = \left[\frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} [f(\hat{x}, y) - f(x, y)]^2 \right]^{\frac{1}{2}} \quad (17)$$

Where M is the total amount of rows and N is the total amount of columns of a given image.

Peak Signal to Noise Ratio

The processed sample value $f(\hat{x}, y)$ can be described as the sum of the reference $f(x, y)$ and the error $e(x, y)$ which can be considered noise. The mean square signal to noise ratio for a processed image can be described by:

$$SNR_{rms} = \frac{\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(\hat{x}, y)^2}{\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} [f(\hat{x}, y) - f(x, y)]^2} \quad (18)$$

And the Peak Signal to Noise Ratio can be described with:

$$PSNR = 20 \times \log_{10} \left(\frac{MAX_I}{RMSE} \right) \quad (19)$$

Where $RMSE$ is described in equation (17) and MAX_I is the maximum value of a pixel sample can have. Using 8 bits to represent a single channel pixel value, $MAX_I = 255$.

The PSNR is a measure of the error in a signal and is commonly used to evaluate the quality of lossy compression algorithms. However, PSNR does not take into account the quality as perceived by the Human Visual System. Figure 50 shows a reference image that has been processed to produce 3 approximations of the original image. The three images have a low psnr value. However, it can be observed that subjectively, figure 50(d) is clearly more legible than the other two distorted images. This shows that PSNR may not be the most optimum image quality assessment metric to use for images that contain computer generated data.

Name	Date modified	Name	Date modified
16x16Blocks	25/07/2018 16:56	16x16Blocks	25/07/2018 16:56
2018_06_07_Conor_TestResults_LZW	06/07/2018 10:24	2018_06_07_Conor_TestResults_LZW	06/07/2018 10:24
AcCompressedFiles	12/06/2018 13:17	AcCompressedFiles	12/06/2018 13:17
AcDecompressedFiles	12/06/2018 13:43	AcDecompressedFiles	12/06/2018 13:43
BinaryVectors	09/07/2018 10:43	BinaryVectors	09/07/2018 10:43
classified_cg_images	23/10/2018 11:46	classified_cg_images	23/10/2018 11:46
colourQuantized	05/06/2018 12:09	colourQuantized	05/06/2018 12:09
compoundImages	25/01/2019 14:11	compoundImages	25/01/2019 14:11
compoundImagesHQ_100	25/02/2019 09:36	compoundImagesHQ_100	25/02/2019 09:36
compoundImagesHQ_125	22/02/2019 15:56	compoundImagesHQ_125	22/02/2019 15:56
compPictures	22/11/2018 11:31	compPictures	22/11/2018 11:31
computerGeneratedImage	16/10/2018 17:18	computerGeneratedImage	16/10/2018 17:18
computerScreenImage	09/10/2018 12:10	computerScreenImage	09/10/2018 12:10
JPGComputerScreenImage	11/12/2018 00:13	JPGComputerScreenImage	11/12/2018 00:13
LZWCompressedVec	05/07/2018 12:44	LZWCompressedVec	05/07/2018 12:44
LZWDeCompressedVec	05/07/2018 09:30	LZWDeCompressedVec	05/07/2018 09:30
lzwVer5CompressedFiles	06/06/2018 10:17	lzwVer5CompressedFiles	06/06/2018 10:17

(a) Reference image

Name	Date modified	Name	Date modified
16x16Blocks	25/07/2018 16:56	16x16Blocks	25/07/2018 16:56
2018_06_07_Conor_TestResults_LZW	06/07/2018 10:24	2018_06_07_Conor_TestResults_LZW	06/07/2018 10:24
AcCompressedFiles	12/06/2018 13:17	AcCompressedFiles	12/06/2018 13:17
AcDecompressedFiles	12/06/2018 13:43	AcDecompressedFiles	12/06/2018 13:43
BinaryVectors	09/07/2018 10:43	BinaryVectors	09/07/2018 10:43
classified_cg_images	23/10/2018 11:46	classified_cg_images	23/10/2018 11:46
colourQuantized	05/06/2018 12:09	colourQuantized	05/06/2018 12:09
compoundImages	25/01/2019 14:11	compoundImages	25/01/2019 14:11
compoundImagesHQ_100	25/02/2019 09:36	compoundImagesHQ_100	25/02/2019 09:36
compoundImagesHQ_125	22/02/2019 15:56	compoundImagesHQ_125	22/02/2019 15:56
compPictures	22/11/2018 11:31	compPictures	22/11/2018 11:31
computerGeneratedImage	16/10/2018 17:18	computerGeneratedImage	16/10/2018 17:18
computerScreenImage	09/10/2018 12:10	computerScreenImage	09/10/2018 12:10
JPGComputerScreenImage	11/12/2018 00:13	JPGComputerScreenImage	11/12/2018 00:13
LZWCompressedVec	05/07/2018 12:44	LZWCompressedVec	05/07/2018 12:44
LZWDeCompressedVec	05/07/2018 09:30	LZWDeCompressedVec	05/07/2018 09:30
lzwVer5CompressedFiles	06/06/2018 10:17	lzwVer5CompressedFiles	06/06/2018 10:17

(b) Gaussian blur psnr = 21.964dB

Name	Date modified	Name	Date modified
16x16Blocks	25/07/2018 16:56	16x16Blocks	25/07/2018 16:56
2018_06_07_Conor_TestResults_LZW	06/07/2018 10:24	2018_06_07_Conor_TestResults_LZW	06/07/2018 10:24
AcCompressedFiles	12/06/2018 13:17	AcCompressedFiles	12/06/2018 13:17
AcDecompressedFiles	12/06/2018 13:43	AcDecompressedFiles	12/06/2018 13:43
BinaryVectors	09/07/2018 10:43	BinaryVectors	09/07/2018 10:43
classified_cg_images	23/10/2018 11:46	classified_cg_images	23/10/2018 11:46
colourQuantized	05/06/2018 12:09	colourQuantized	05/06/2018 12:09
compoundImages	25/01/2019 14:11	compoundImages	25/01/2019 14:11
compoundImagesHQ_100	25/02/2019 09:36	compoundImagesHQ_100	25/02/2019 09:36
compoundImagesHQ_125	22/02/2019 15:56	compoundImagesHQ_125	22/02/2019 15:56
compPictures	22/11/2018 11:31	compPictures	22/11/2018 11:31
computerGeneratedImage	16/10/2018 17:18	computerGeneratedImage	16/10/2018 17:18
computerScreenImage	09/10/2018 12:10	computerScreenImage	09/10/2018 12:10
JPGComputerScreenImage	11/12/2018 00:13	JPGComputerScreenImage	11/12/2018 00:13
LZWCompressedVec	05/07/2018 12:44	LZWCompressedVec	05/07/2018 12:44
LZWDeCompressedVec	05/07/2018 09:30	LZWDeCompressedVec	05/07/2018 09:30
lzwVer5CompressedFiles	06/06/2018 10:17	lzwVer5CompressedFiles	06/06/2018 10:17

(c) Gaussian blur psnr = 19.401dB

Name	Date modified	Name	Date modified
16x16Blocks	25/07/2018 16:56	16x16Blocks	25/07/2018 16:56
2018_06_07_Conor_TestResults_LZW	06/07/2018 10:24	2018_06_07_Conor_TestResults_LZW	06/07/2018 10:24
AcCompressedFiles	12/06/2018 13:17	AcCompressedFiles	12/06/2018 13:17
AcDecompressedFiles	12/06/2018 13:43	AcDecompressedFiles	12/06/2018 13:43
BinaryVectors	09/07/2018 10:43	BinaryVectors	09/07/2018 10:43
classified_cg_images	23/10/2018 11:46	classified_cg_images	23/10/2018 11:46
colourQuantized	05/06/2018 12:09	colourQuantized	05/06/2018 12:09
compoundImages	25/01/2019 14:11	compoundImages	25/01/2019 14:11
compoundImagesHQ_100	25/02/2019 09:36	compoundImagesHQ_100	25/02/2019 09:36
compoundImagesHQ_125	22/02/2019 15:56	compoundImagesHQ_125	22/02/2019 15:56
compPictures	22/11/2018 11:31	compPictures	22/11/2018 11:31
computerGeneratedImage	16/10/2018 17:18	computerGeneratedImage	16/10/2018 17:18
computerScreenImage	09/10/2018 12:10	computerScreenImage	09/10/2018 12:10
JPGComputerScreenImage	11/12/2018 00:13	JPGComputerScreenImage	11/12/2018 00:13
LZWCompressedVec	05/07/2018 12:44	LZWCompressedVec	05/07/2018 12:44
LZWDeCompressedVec	05/07/2018 09:30	LZWDeCompressedVec	05/07/2018 09:30
lzwVer5CompressedFiles	06/06/2018 10:17	lzwVer5CompressedFiles	06/06/2018 10:17

(d) Injected noise psnr = 17.467dB

Figure 50: A Reference image distorted with Gaussian blurring with 3x3 kernel, Gaussian Blurring with 5x5 kernel and injected noise

Structural Similarity Index (SSIM)

Structural Similarity Index (SSIM) [50] is an image quality assessment metric that takes into account the attributes of the Human Visual System. It is a perception based model that takes into account luminance masking, contrast masking and structure.

Luminance masking is the effect of distortion in an image that is less visible in bright areas (such as a white background). **Contrast masking** is the effect of dis-

tortion in an image being less visible in areas that are highly textured (such as in image containing natural image such as grass or the sky).

The structural information that is taken into account is based on pixel values in close proximity are highly correlated and the correlation contains information about the structure of objects in an image.

SSIM is a full reference image that compares a reference image, quantitatively considered error free and compares it to a processed image to generate a value between 0 and 1. It is described by:

$$SSIM_{(x,y)} = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \quad (20)$$

where x is the reference image and y is the processed image.

μ_x	the average of x
μ_y	the average of y
σ_x^2	the variance of x
σ_y^2	the variance of y
σ_{xy}	the covariance of x and y
$c_1 = (k_1L)^2$	used to keep the equation from dividing by values very close to 0
$c_2 = (k_2L)^2$	used to keep the equation from dividing by values very close to 0
$c_3 = c_2/2$	used to keep the equation from dividing by values very close to 0
L	the dynamic range of pixel values, typically 255
k_1	0.01
k_2	0.03

Figure 51 shows the flow of the algorithm. It is based on three functions for luminance measurement, contrast measurement and structural comparison.

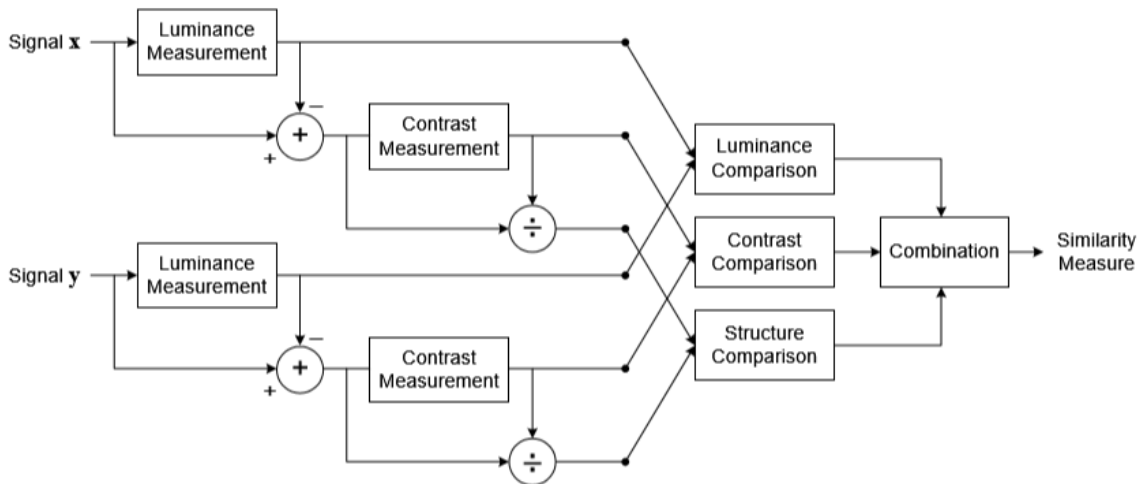


Figure 51: Diagram showing the structural similarity measurement process [50]

The luminance measurement function is described by :

$$l(x, y) = \frac{2\mu_x\mu_y + c_1}{\mu_x^2 + \mu_y^2 + c_1} \quad (21)$$

The contrast measurement function is described by

$$c(x, y) = \frac{2\sigma_x\sigma_y + c_2}{\sigma_x^2 + \sigma_y^2 + c_2} \quad (22)$$

The structural function is described by

$$s(x, y) = \frac{\sigma_{xy} + c_3}{\sigma_x\sigma_y + c_3} \quad (23)$$

To highlight the difference between SSIM and PSNR as a metric for image quality assessment on image containing computer generated data, the SSIM values are calculated for the distorted images in figure 50 and compared to the calculated PSNR value. This can be seen in table 15. A subjective quality metric is also included in the table for each image.

It can be observed from table 15 that figure 50 (d) has the worst PSNR value, yet has the highest SSIM value. Also subjectively, it is the most legible of the distorted images. This is a significant observation, as it shows that, for image data containing computer generated data the SSIM index is aligned with subjective viewing quality.

Figure 50	Distortion Type	PSNR (dB)	SSIM Value	Subjective Quality
(b)	Gaussian Blur 3x3	21.96	0.83	Poor
(c)	Gaussian Blur 5x5	19.4	0.76	Non-Usable
(d)	Injected Noise	17.47	0.97	Acceptable

Table 15: Comparing PSNR Vs SSIM values of distorted images from figure 50

A combination of PSNR and SSIM will be used in this thesis as metrics to compare the performance of the compound compression algorithm presented in this research. The authors of the algorithm published the implementation of the algorithm in matlab and transcribed into c++ [43]

6.2.2 Block Transform Encoding

This section will develop the process of using a discrete cosine transform and discrete wavelet transform to decorrelate the information contained in blocks of pixels and discard the information that may not have a negative impact on the human visual system in favour of better compression performance. Aside from rounding errors that occur from using finite precision to calculate the forward and reverse DCT and DWT transforms, both algorithms can be performed with little to no loss of information of the original image. The process of discarding data in a lossy compression algorithm comes from the quantization process.

In block transform coding, a reversible transform is performed on each sub-block of an image decomposed into $N \times N$ blocks. The transform maps the pixels of the sub block to a set of transform coefficients, which are then quantized to remove information irrelevancy (in favour of better compression performance) and finally processed by an entropy encoder to remove coding redundancy to generate the compressed output.

Transforms such as the Karhunen Loeve transform, The two-dimensional discrete Fourier transform and the two-dimensional discrete cosine transform will transform the blocks from the pixel domain to the frequency domain, while the discrete wavelet transform will transform from the pixel domain to the time-frequency domain.

Block transform coding has been shown to be efficient in terms of compression performance and computational complexity and is implemented in codec's used for image compression such as JPEG [18] and video encoding such as H.264 [47] and HEVC [39].

6.2.3 The Discrete Cosine Transform

The discrete cosine transform is at the heart of compression algorithms that are optimised for natural image compression such as JPEG [18] and video coding standards such as H.264 and HEVC.

In image compression, the discrete cosine transform takes an $N \times N$ block of real valued pixels as an input signal and produces an $N \times N$ block of transformed coefficients in the frequency domain as an output. The purpose of the transform is to compact most of the energy of the input signal into just a few of the output coefficients, usually located in the top left corner of the $N \times N$ block, such that the remaining values can be coarsely quantized or even discarded, reducing the amount of data required for storage or transmission. The transform is reversible, such that if infinite precision is used in the computation, the inverse transform will yield an exact replica of the original signal, however it is infeasible to use large scale precision for calculations for time sensitive applications so either fixed precision or scaled integer versions of the transform are used, which introduce rounding errors. The process of performing

the discrete cosine transform for compression is generally called the forward DCT, or simply, DCT, while the process in recovering the original data from the transform coefficients is known as the inverse DCT or IDCT.

The discrete cosine transform is used in image compression due to its close approximation to another transform known as the Karhunen-Loeve transform (KLT). The KLT transform minimises the mean square error [32] when choosing the minimum amount of transform coefficients to reproduce the original signal. However, the basis functions of the KLT transform are of the auto-correlation of the image, which means that they are dependant on the input signal and must be computed for each image, unlike the discrete cosine transform, who's basis functions are set regardless of the input signal. The constant computation required to generate the basis functions to perform the KLT reduces its effectiveness for time critical applications.

The discrete cosine transform works on the principle that the distribution of pixels that make up natural image can be said to be Markovian, which means that a given pixel depends on the pixel before it. As an image is in two dimensions, there is correlation in both the horizontal and vertical neighbourhood of a given pixel. This is true for natural image, for example images of the sky, grass or skin tone, there may be many different values but the gradient between values is slowly varying. This assumption does not hold for computer generated image data such as graphics and text, where sharp transitions in a smooth background happen frequently.

The cosine function has been chosen for its periodicity. Its a symmetrical even $2n$ -point function (input to the transform is generally taken as a 2^n multiple) who's boundaries do not cause discontinuity. Figure 52 shows the periodicity of a 1 dimensional discrete Fourier transform (DFT) versus a 1 dimensional discrete cosine transform. It can be observed that the periodicity of the DFT leads to discontinuity at boundaries, which gives rise to substantial high frequency content. As the Fourier transform fails to converge uniformly at discontinuities (known as Gibbs phenomenon), the coefficients at boundary points take on erroneous values, which may lead to blocking artifacts in the decompressed image [14]. As the DCT is taken as a $2n$ point function, the values at the boundary points are closer together, which helps reduce the high frequency content and alleviate some of the blocking artifacts.

The two-dimensional discrete cosine transform (2-D DCT) calculates the transform $F(U,V)$ of a two-dimensional input signal $f(i,j)$ of size $N \times N$ samples, where the size 8×8 is typically used. The equation for the 2D DCT can be defined as:

$$F_{xy} = \alpha \alpha \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} f_{ij} \cos\left(\frac{(2j+1)u\pi}{2xN}\right) \cos\left(\frac{(2i+1)v\pi}{2xN}\right) \quad (24)$$

Where F_{xy} is the transformed value at location f_{ij} of the original block to be transformed. α is a normalisation factor for the DCT coefficients and is defined as: $\alpha = \sqrt{\frac{1}{N}}$ $N = 0$ and $\alpha = \sqrt{\frac{2}{N}}$ $N \neq 0$:

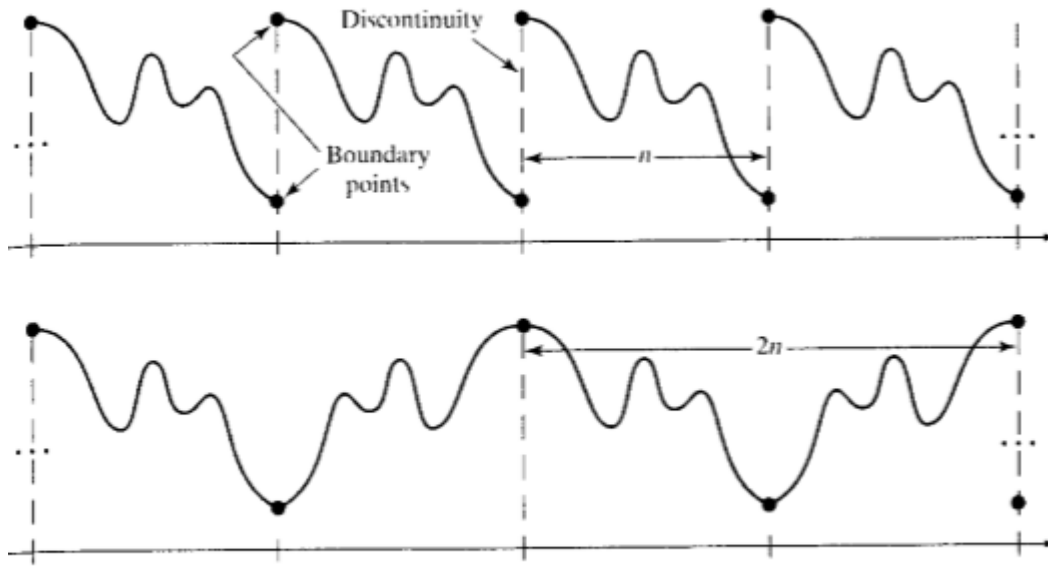


Figure 52: Comparing the periodicity of a 1-D DFT function (top) with a 1-D DCT function (bottom)[14]

The 2-D DCT is also a reversible function, where the inverse 2D-DCT can be described by:

$$\hat{f}_{ij} = \alpha \alpha \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} F_{xy} \cos\left(\frac{(2y+1)u\pi}{2xN}\right) \cos\left(\frac{(2x+1)v\pi}{2xN}\right) \quad (25)$$

where \hat{f}_{ij} is the recovered value after inverse transform, after incurring any rounding error, due to finite precision calculation.

2-D Discrete Cosine Transform Basis Functions

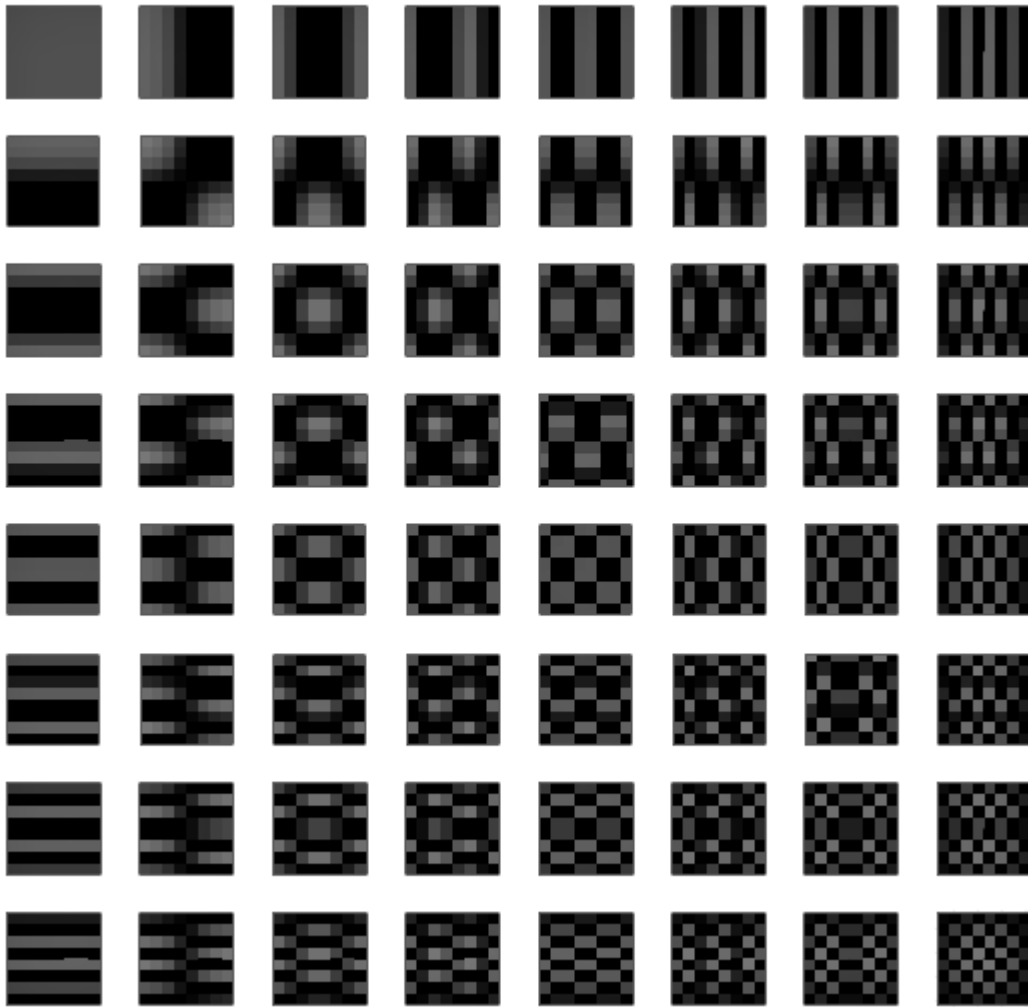


Figure 53: The 64 Basis Functions of the 2-D DCT plotted using MATLAB

Figure 53 shows the 64 2D-DCT basis functions when $N=8$, computed and plotted using MATLAB software. Each block is an 8×8 matrix function, where each output transform coefficient is a weighted value corresponding to each function, the sum of which will return the original image.

The pseudo code in listing 5 shows the implementation of the 2D DCT. To show the energy compaction of the 2D-DCT transform, an 8×8 block of Y channel values is extracted from an image, which can be seen in figure 54.

Correlation in intensity values in the horizontal, vertical and diagonal directions can be observed in figure 54, where the intensity values vary slowly from one pixel to the next. This type of distribution is indicative of natural image.

Listing 5: 2-D Direct DCT transform

```
1  /*DIRECT 2-D Discrete Cosine Transform:*/
2
3  input_values[8][8];    // 8x8 block of 8 bit values
4  output_coefs[8][8];   // 8x8 block of transformed DCT coefficients
5
6  alpha[8]; // normalizing factor:
7             // alpha[0] =  $1/\sqrt{8}$ 
8             // alpha[1-7] =  $2/\sqrt{8}$ 
9
10 for (U = 0; U < 8; ++U)    //U = coefficient matrix rows
11 {
12     for (V = 0; V < 8; ++V) //V=coefficient columns
13     {
14         sum = 0.0; // sum =64 computations per input value
15         for (i = 0; i < 8; ++i)
16         {
17             for (j = 0; j < 8; ++j)
18             {
19                 coef = cos((2 * i + 1) * U * PI / 16)
20                 * cos((2 * j + 1) * V * PI / 16);
21
22                 sum += input_values[i][j] * coef;           //f[i][j] * coef
23             } //for j
24         } //for i
25         output_coefs[u][v] = alpha[u] * alpha[v] * sum;
26     } //for U
27 } //for V
```

139	144	149	153	155	155	155	155
144	151	153	156	159	156	156	156
150	155	160	163	158	156	156	156
159	161	162	160	160	159	159	159
159	160	161	162	162	155	155	155
161	161	161	161	160	157	157	157
162	162	161	163	162	157	157	157
162	162	161	161	163	158	158	158

Figure 54: 8x8 block of Y channel pixel values

Transform coefficient values							
1260	-1	-12	-5	2	-2	-3	1
-23	-17	-6	-3	-3	0	0	1
-11	-9	-2	2	0	-1	-1	0
-7	-2	0	1	1	0	0	0
-1	-1	1	2	0	-1	1	1
2	0	2	0	-1	1	1	-1
-1	0	0	-1	0	2	1	-1
-3	2	-4	-2	2	1	-1	0

Table 16: The transform coefficient values of Figure 54

Table 16 shows the resultant transform coefficient values, (rounded up to the nearest integer) after performing the 2-D DCT on figure 54. It can be observed that the value in top left corner is significantly larger than the rest of the values. This value is known as the dc coefficient and can be calculated by:

$$\frac{1}{N} \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} f_{i,j} \quad (26)$$

Where N=8. This value is thought of as the mean value of the block and holds most

of the energy of the input signal. The remaining coefficients are known as the ac coefficients and hold the frequency content of the transform.

It can be observed that the energy of the ac coefficients reduces significantly as it approaches the bottom right hand corner of the output transform coefficient matrix, compared to the top left corner. As there are no high sharp transients in the original data, the low frequency basis functions have greater weighting for this type of data than the high frequency coefficients. The higher valued coefficients of the output matrix represent how much of the corresponding basis function is used to represent the original image. When the value of the output coefficient is low, the corresponding basis function's contribution to the original image is lower and thus can be quantized or discarded.

The act of using quantization to discard data that is not needed to reconstruct a suitable approximation of the original data is what makes the compression technique lossy. Quantization will be discussed further in section 6.2.5.

It can be observed from the pseudo code in listing 5 that there are four nested loops, which will give a total of 64 multiplications and 49 additions per coefficient, totalling 4096 multiplications per 8x8 block of samples. This gives a computing complexity of $O(N^4)$. This implementation is computationally complex, which would increase latency in a time sensitive application. Fortunately, a property of the 2D-DCT function is that it is a separable function, which means the functions can be implemented by using two 1-D DCT transforms, first along the rows, then along the columns. A transform is said to be separable if it satisfies the following conditions:

$$y = AxAT^T \quad (27)$$

Where y is an $N \times N$ block of output coefficients, A is an Orthonormal transform of size $N \times N$ (the DCT basis functions), x is the input $N \times N$ block of values and A^T is the transposition of the transform matrix. Performing the operation as two 1-D transforms requires two matrix multiplications of size $N \times N$, instead of vector of size $1 \times N^2$ with a $N^2 \times N^2$ matrix, which reduces the complexity from $O(N^4)$ to $O(N^3)$. The 1-D Dct function can be expressed by:

$$G(m) = \sqrt{\frac{2}{N}} \sum_{m=0}^{N-1} g(u) c_m \cos\left(\frac{(2j+1)m\pi}{2xN}\right) \quad (28)$$

where $G(m)$ is the transform output coefficient, $g(u)$ is the current input value. This reduces the amount of multiplications to 1024 multiplication operations per 8x8 block, However the time complexity of $O(N^3)$ would be still too great for real time applications.

The seminal research done in [1] provides a method to greatly reduce the amount of multiplications needed to perform a one-dimensional discrete cosine transform.

Figure 55 shows the butterfly diagram of the AAN [1] discrete cosine transform and figure 56 shows the operations that are carried out. From the legend in figure 56, It can be observed that there are 13 multiplication operations performed per row of 8 values, some of which can be implemented as bit shifts instead of multiplications. This reduces the amount of multiplication operations to 208 per block of 8x8 pixel sample values. The implementation for the fast AAN DCT coded in C++ can be seen in Listing 25 in the code section of this thesis.

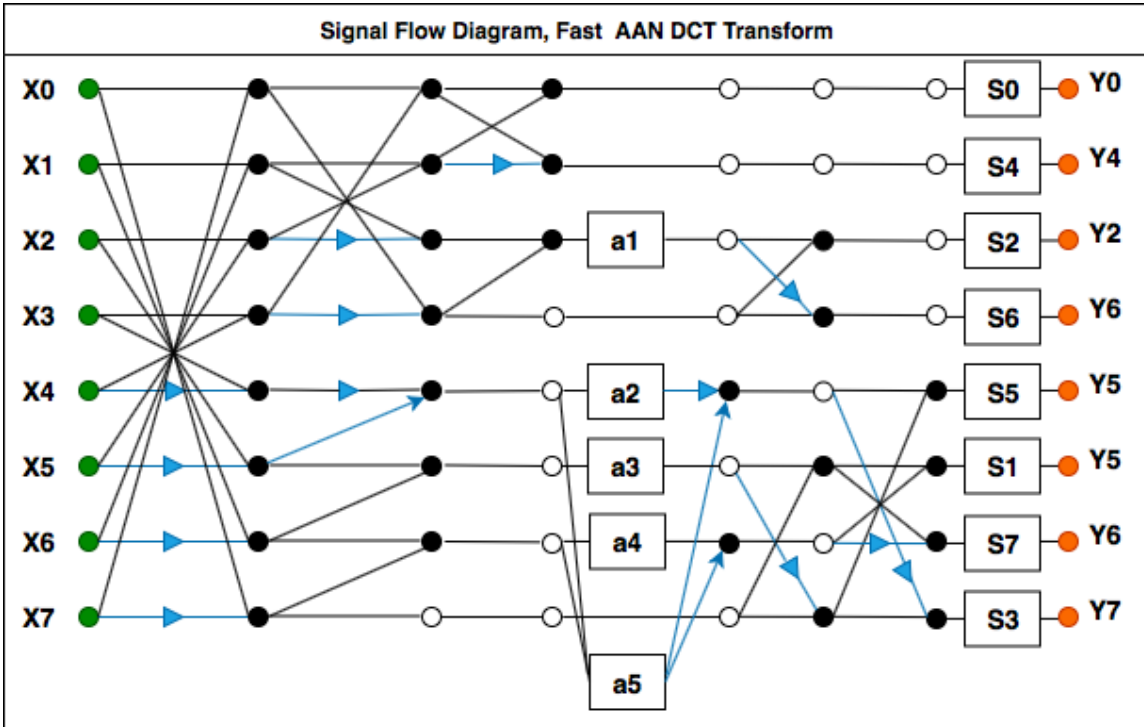


Figure 55: A signal flow diagram of the forward AAN fast DCT transform

Legend				
Values	Arithmetic	Multiplication	a values	formulae
<ul style="list-style-type: none"> ● Input value ● Intermediate value Non-Changed value ● Output value 	<ul style="list-style-type: none"> a ● ——— ● a+b b ● ——— ● a+b a ● ——— ● a-b b ● ——— ● a-b 	<ul style="list-style-type: none"> c ● ——— d ——— c x d 	<ul style="list-style-type: none"> a1 = C4 a2 = C2 - C6 a3 = C4 a4 = C6+C2 a5 = C6 	<ul style="list-style-type: none"> $S_0 = \frac{1}{2\sqrt{2}}$ $S_k = \frac{1}{4C_k}$ k = 1, ..., 7 $C_k = \cos(\frac{\pi}{16}k)$

Figure 56: Legend for figure 55

6.2.4 Comparing speed performance of 3 discrete cosine transform methods

A test is configured to compare the computational time to perform the discrete cosine transform on 2400 blocks of 8x8 pixel intensity values. The first method performs the discrete cosine transform as a two-dimensional transform, as shown in Listing 5. The second method performs the discrete cosine as two one-dimensional transforms, first along the columns, then along the rows. The third method performs the discrete cosine transform using the fast AAN method, shown in figure 55.

In the experiment, each method is used 20 times, in random order. The same 2400 blocks of 8x8 pixel intensity values are used each time, by all three methods. The experiment is carried out in two blocks. The average time will then be calculated for each method. The discrete cosine transform methods are coded using c++ language and can be seen in the code section of this thesis in listings 23 - 25 .

Discrete Cosine Transform method Vs Computation time

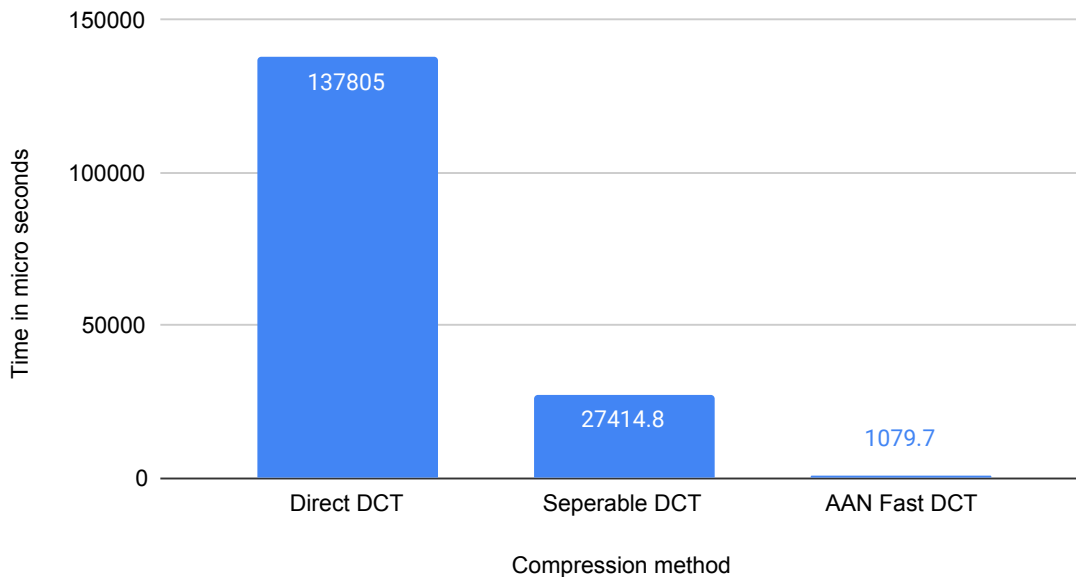


Figure 57: Comparing three methods of discrete cosine transform, with respect to computational time

Figure 57 Shows the results of test. It can be clearly seen that the fast AAN discrete cosine transform is the most efficient in terms of computational time. This is the basis of the discrete cosine transform that will be used in this thesis.

6.2.5 Quantization

After the discrete cosine transform has processed a block of pixel values, the next step is to quantize the coefficient's. The process of quantization will reduce the dynamic range of the coefficient values and discard some of the information, in favour of better compression performance.

The process of quantizing involves dividing a transform coefficient by an integer value and rounding the resultant value to the nearest integer on the encoder side. on the decoder side, the value is de-quantized by multiplying the same coefficient by the same integer value, to get an approximation of the original value. Using a small quantization coefficient reduces the amount of error in the decoded image, but does not reduce the size of the dct coefficient as much as using a larger quantization coefficient. Better compression performance is achieved by using large quantization values at the expensive of distortion in the decoded image. The discrete cosine transform function's role is to compact as much of the energy in the image into just a small number of coefficients. The low frequency dct coefficients are located around the top left of a block of samples. The quantization coefficients used for low frequency dct coefficients is smaller compared to the quantization coefficients used for high frequency dct coefficients

It is important to note that performing the discrete cosine transform does not compress the image data and that apart from minor rounding error, is not responsible for the loss in information when used in a lossy compression algorithm. Information is discarded when the discrete cosine transform coefficients are quantized.

the JPEG standard gives two examples of quantization matrices that can be used. One is for the luminance channel and one is for both chrominance channels. The luminance quantization matrix can be seen in table 17

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

Table 17: Luminance quantization matrix described in JPEG [18]

it can be observed from the quantization matrix in table 17 that the quantization coefficient values are lower toward the top left of the matrix and increase in value towards the bottom right. This is reflected in the low to high frequency coefficients

of the discrete cosine transform. To illustrate the function of quantization, the pixel sample values from figure 54 are processed with the quantization matrix in table 17. The resultant quantized values can be seen in table 18

79	0	-1	0	0	0	0	0
-2	-1	0	0	0	0	0	0
-1	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Table 18: Quantized discrete cosine transform values from figure 54

It can be seen that most of the values have been reduced to zero. The quantized coefficients can be further processed by a form of run length encoding, where the coefficients are first reorganised in a zig-zag pattern to optimise the run length of zero's. The zig-zag pattern is shown in figure 58

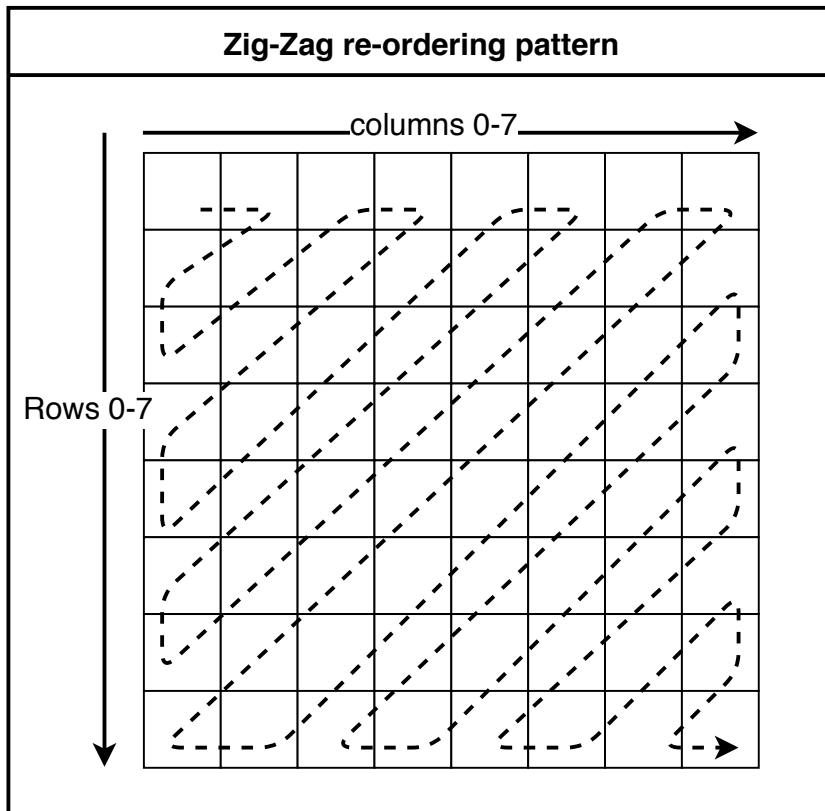


Figure 58: Zig-Zag re-order pattern for quantized DCT coefficients

After reordering the dct coefficient values using the zig-zag pattern in figure 58 the

discrete cosine transform coefficients can be described using a run length encoding pattern, where the zero's are used as the run: $[0, 79]$, $[1, -2]$, $[0, -1][0, -1][0-1]$, $[2, -1]$ [Rest are zeros]. This can be interpreted as no 0's before 79, one 0 before -2, no 0's before -1, etc. In compression algorithms such as JPEG [18], a special character is used to denote the end of sequence, when all the remaining characters are zero. It is generally coded in the bit stream as 0,0. The DC coefficient [79] is generally not included in the run length scanning of a block as the value is much larger than the ac coefficients.

The decoding of data that has been processed with the outlined process, is simply the inverse of the steps carried out, but in reverse order. The encoding and decoding process can be seen in figure 59

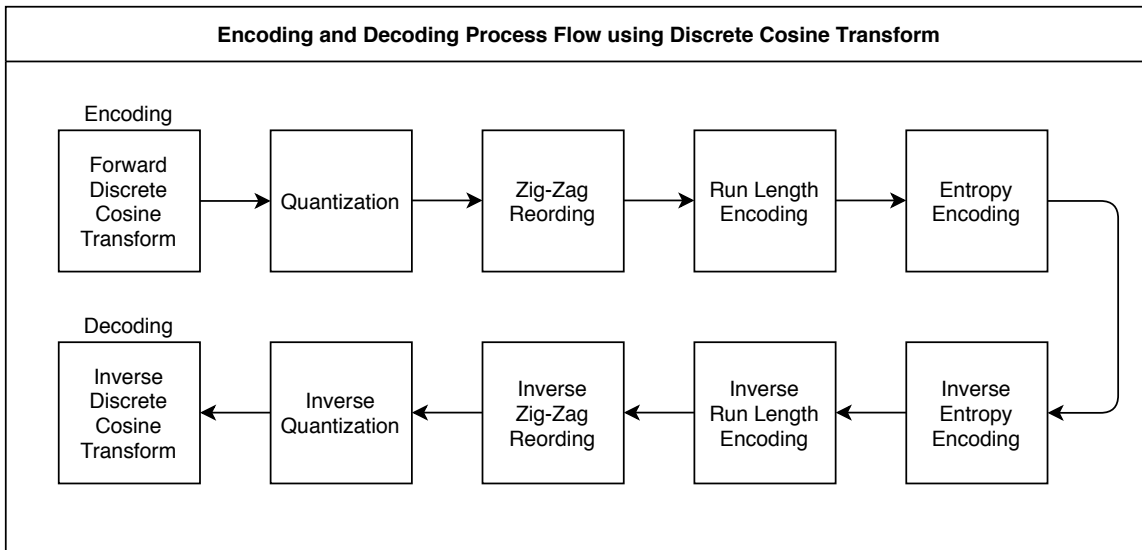


Figure 59: The encoding and decoding process using DCT, quantization and Entropy encoding

6.2.6 The Discrete Wavelet Transform

The discrete Haar wavelet transform was introduced in this thesis in the section on classification. In this section, the discrete wavelet transform will be discussed with respect to compression.

The properties of the discrete wavelet transform so far that have been discussed are its ability to localise frequency content in the time domain and that the two-dimensional wavelet transform needed for image compression is separable into two one-dimensional functions, which reduces time complexity. Another property of the discrete wavelet transform is that it can be applied in a multi-resolution application. This means that the process can be applied recursively. As previously discussed, a one level two-dimensional discrete wavelet transform decomposes an input image into four

sub bands: LL, HL, LH and HH, where L stands for low-pass filtering and H stands for high-pass filtering. The LL sub band is an approximation of the original image at $\frac{1}{4}$ of the resolution. This process compacts a significant amount of information of the original image into a much smaller space, thus decorrelating the information among neighbouring pixels. If the same two-dimensional discrete wavelet transform is performed on the values of the LL sub band, the result is a two level, two-dimensional discrete wavelet transform which has compacted most of the information of the signal into a space that is $\frac{1}{8}$ the original resolution of the original image.

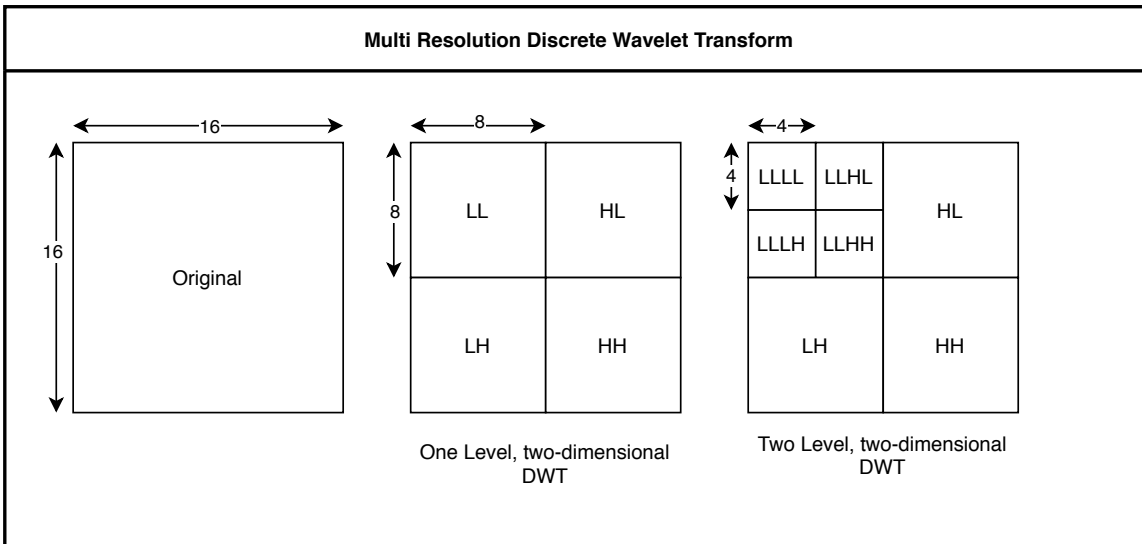


Figure 60: Multi resolution discrete wavelet transform analysis

The location of the sub bands with respect to the original image is shown in figure 60, which shows the decomposition of a 16x16 block of pixel values decomposed with a one level and two-level two-dimensional discrete wavelet transform.

By compacting as much of the information of the original image into a smaller amount of pixels, data irrelevancy can be exploited, by discarding some of the information in the sub bands that do not contribute much to the image, in favour of a better compression performance. This is done by thresholding the discrete wavelet transform coefficients: values below a given threshold are set to zero, while values above the threshold are left as is.

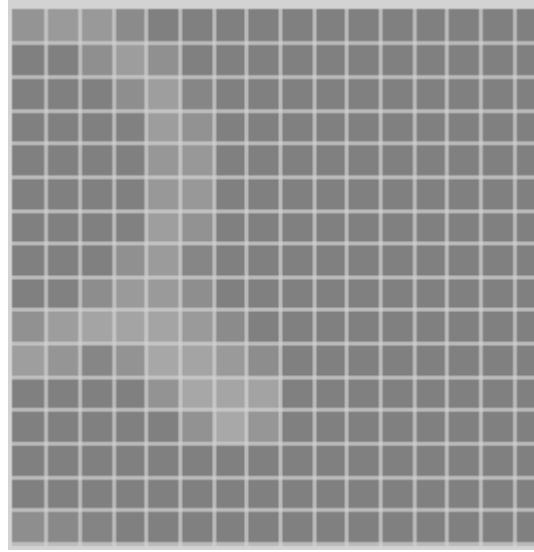


Figure 61: 16x16 block of pixels with graphical information

148	154	153	139	128	128	128	128	128	128	128	128	128	128	128	128
128	128	143	157	144	128	128	128	128	128	128	128	128	128	128	128
128	128	128	142	158	134	128	128	128	128	128	128	128	128	128	128
128	128	128	129	157	146	128	128	128	128	128	128	128	128	128	128
128	128	128	128	151	152	128	128	128	128	128	128	128	128	128	128
128	128	128	128	152	152	128	128	128	128	128	128	128	128	128	128
128	128	128	129	158	146	128	128	128	128	128	128	128	128	128	128
128	128	128	144	155	134	128	128	128	128	128	128	128	128	128	128
128	129	141	154	154	142	128	128	128	128	128	128	128	128	128	128
145	157	164	164	164	154	139	128	128	128	128	128	128	128	128	128
158	149	134	147	167	164	154	141	128	128	128	128	128	128	128	128
128	128	128	128	147	167	165	163	128	128	128	128	128	128	128	128
128	128	128	128	128	146	168	150	128	128	128	128	128	128	128	128
128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	128
128	128	128	128	128	128	128	128	128	128	128	128	128	128	128	128
142	137	132	128	128	128	128	128	128	128	128	128	128	128	128	128

Table 19: Pixel intensity values for figure 61

Figure 61 represents a single channel of a 16x16 block that contains computer generated text. The type of data contained in figure 61 would not be suitable to be compressed using a discrete cosine transform, as the structured values that make up the character against a uniform background, which can be seen in the values in table 19, do not follow a low frequency distribution. This would lead to poor energy compaction and ringing artefact's after decompression. However, the Haar wavelet transform is suited to compress such data, because of its ability to use short wavelet

basis to extract the high frequency content, while compacting the low frequency information into fewer coefficients.

The first step in compressing data using a forward discrete wavelet transform is to apply a level shift to each pixel intensity, so each value is centred about zero. This is achieved by subtracting a scalar value of 128 from each intensity value in the encoding side and in the decoding side, adding 128 to each retrieved value. The purpose of this is to lower the dynamic range in the transform coefficients. The pixel intensity values of an image are non negative values, however the discrete wavelet transform produces both positive and negative values. Centering the pixel intensity values about zero helps reduce the size of the magnitude of the transform coefficients.

Listing 6: forward discrete Haar wavelet transform, C++

```

1 //s0,s1,d0 are the weighting values
2 // =0.5 to represent dividing by 2
3 //d1 = -0.5 to represent the difference equation
4 forward_column_transform(data[16][16],length)
5 {
6     for (row = 0; row < 16; row++) {
7         temp_values[16] = { 0 };
8         h = length >> 1;
9         for (i = 0 ; i < h; i++) {
10            int k = (i << 1);
11            /* averaging calculation */
12            temp[i] = data[row][k] * s0 + data[row][k + 1] * s1;
13
14            /* difference calculation */
15            temp[i + h] = data[row][k] * d0 + data[row][k + 1] * d1;
16        }
17        /* store the basis calculations 'In place' in the original 2D array */
18        for (int i = 0; i < length; i++)
19            data[row][i] = temp[i];
20    }
21 }
22 forward_row_transform(data[16][16],length)
23 {
24     for (int col = 0; col < 16; col++) {
25         /* temp hold the values while calculating the average and difference */
26         temp[16] = { 0 };
27         int h = length >> 1;
28         for (i = 0; i < h; i++)
29         {
30             k = (i << 1);
31             temp[i] = data[k][col] * s0 + data[k + 1][col] * s1;
32             temp[i + h] = data[k][col] * d0 + data[k + 1][col] * d1;
33         }
34         for (i = 0; i < length; i++)
35             data[i][col] = temp[i];

```

36
37

```

    }
}
    
```

To illustrate the transformation and compression process using a discrete wavelet transform, the pseudo code in listing 6 for the two-dimensional forward Haar discrete wavelet transform split into two one-dimensional Haar transforms (along the columns and the rows), will be used to process the values in table 19.

Table 20 shows the transform coefficients after performing the transform as one level. Looking at the values in table 20, it is clear to see that a significant amount of coefficients are zero. It can be also observed that the dynamic range of the coefficient values have been greatly reduced, which could be a potential benefit in compression performance.

11	20	4	0	0	0	0	0	-1	0	4	0	0	0	0	0
0	3	20	0	0	0	0	0	0	-3	8	0	0	0	0	0
0	0	23	0	0	0	0	0	0	0	0	0	0	0	0	0
0	4	20	0	0	0	0	0	0	-4	8	0	0	0	0	0
11	27	25	2	0	0	0	0	-3	-3	5	2	0	0	0	0
12	6	33	27	0	0	0	0	2	-3	-4	3	0	0	0	0
0	0	4	15	0	0	0	0	0	0	-4	4	0	0	0	0
5	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0
11	-2	-4	0	0	0	0	0	-1	7	-4	0	0	0	0	0
0	3	-2	0	0	0	0	0	0	-3	3	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	-3	3	0	0	0	0	0	0	3	-2	0	0	0	0	0
-11	-8	-5	-2	0	0	0	0	2	-3	0	-2	0	0	0	0
12	6	4	-8	0	0	0	0	2	-3	5	2	0	0	0	0
0	0	4	15	0	0	0	0	0	0	-4	4	0	0	0	0
-5	-1	0	0	0	0	0	0	-1	-1	0	0	0	0	0	0

Table 20: One level two-dimensional forward Haar discrete wavelet coefficients top left = LL sub- band, top right = HL sub-band, bottem left = LH sub=band, bottom right = HH sub-band: 79 non-zero coefficients

Table 21 shows the output coefficients after performing the second level forward Haar discrete wavelet transform on the sub band LL of table 20 (top right 8x8 block). In this instance, it can be observed that there is a slight increase in the number of non zero coefficients, however the dynamic range of coefficient values has been reduced further.

8	6	0	0	-3	6	0	0	-1	0	4	0	0	0	0	0
1	11	0	0	-1	11	0	0	0	-3	8	0	0	0	0	0
14	22	0	0	-2	7	0	0	0	0	0	0	0	0	0	0
1	5	0	0	0	-2	0	0	0	-4	8	0	0	0	0	0
6	-4	0	0	-1	-4	0	0	-3	-3	5	2	0	0	0	0
-1	0	0	0	1	0	0	0	2	-3	-4	3	0	0	0	0
5	-8	0	0	-5	4	0	0	0	0	-4	4	0	0	0	0
-1	5	0	0	-1	-2	0	0	1	1	0	0	0	0	0	0
11	-2	-4	0	0	0	0	0	-1	7	-4	0	0	0	0	0
0	3	-2	0	0	0	0	0	0	-3	3	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	-3	3	0	0	0	0	0	0	3	-2	0	0	0	0	0
-11	-8	-5	-2	0	0	0	0	2	-3	0	-2	0	0	0	0
12	6	4	-8	0	0	0	0	2	-3	5	2	0	0	0	0
0	0	4	15	0	0	0	0	0	0	-4	4	0	0	0	0
-5	-1	0	0	0	0	0	0	-1	-1	0	0	0	0	0	0

Table 21: Two level two dimensional forward Haar discrete wavelet coefficients: 81 non-zero coefficients, values have a lower dynamic range

The next step in encoding is to use a thresholding process to reduce code irrelevancy in the data. Aside from some small rounding error from converting floating point to integer, the thresholding process is what introduces distortion and makes the algorithm a lossy compression algorithm.

Thresholding involves comparing a coefficient value to a specified value. If a given coefficient value is under the specified threshold value, it is set to zero. As previously discussed, transient information is captured in the sub bands HL, LH and HH, while an approximation of the original block is stored in the sub-band LL. If there is low transient information in a block, the dynamic range of values in the sub-bands HL, LH and HH will be low and likewise, if there is transient information in a block, the dynamic range of the coefficient values in these sub bands will be high. The purpose of using a discrete wavelet transform for lossy compression is to retain as much transient information as possible about a block, while discarding information that does not.

A novel approach to thresholding has been implemented in this thesis. It is a recursive thresholding process, which occurs after each level of discrete wavelet transform decomposition. The Threshold value decreases after each level and only the newly computed values of each level are processed. The pseudo code for rounding coefficients to integer values and then applying thresholding can be seen in listing 7. The function is passed an integer value of 10 for threshold after the first level of transformation and is passed an integer value of 2 for the threshold after the second level of transformation.

Listing 7: Discrete Wavelet Transform round and threshold function

```

1 round_and_threshold(data[16][16],threshold,level)
2 {
3     /* this is for rounding and thresholding */
4     for (int Row = 0; Row < level; Row++)
5     {
6         for (int Col = 0; Col < level; Col++)
7         {
8             /* rounding the floating point number to now decimal places */
9             data[Row][Col] = round(data[Row][Col]);
10            /* if the absolute value is less than threshold it is set to 0 */
11            if (abs(data[Row][Col]) < threshold)
12            {
13                data[Row][Col] = 0;
14            }
15        }
16    }
17 }

```

Table 22 shows the resultant values after thresholding the values of table 21

8	5	0	0	-2	5	0	0	0	0	0	0	0	0	0
0	11	0	0	0	11	0	0	0	0	0	0	0	0	0
13	22	0	0	0	8	0	0	0	0	0	0	0	0	0
0	4	0	0	0	-4	0	0	0	0	0	0	0	0	0
8	-5	0	0	-2	-5	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	-9	0	0	-7	5	0	0	0	0	0	0	0	0	0
0	4	0	0	0	-4	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-11	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	16	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 22: After two level forward discrete Haar wavelet transform. thresholding value 6: 25 non zero coefficients

It can be observed from table 22 that the majority of values are now zero, with only 25 non zero coefficients, that have a greatly reduced dynamic range than the original pixel intensity values.

Using the discrete wavelet transform in the above example has compacted the infor-

mation about the image in figure 61 into just a few coefficients, which is suitable for further processing. However, unlike processing with a discrete cosine transform, the location of the non zero coefficients are spread out in the sub bands and is highly dependent on the transient information in the block itself. As such, a zig-zag run length scan pattern as used in JPEG [18], may not be suitable to process the data prior to entropy encoding. A novel method to process the non zero coefficients prior to entropy encoding will be presented further on in this thesis in section 9.3.14.

The process flow for encoding and decoding using a discrete wavelet transform can be seen in figure 62

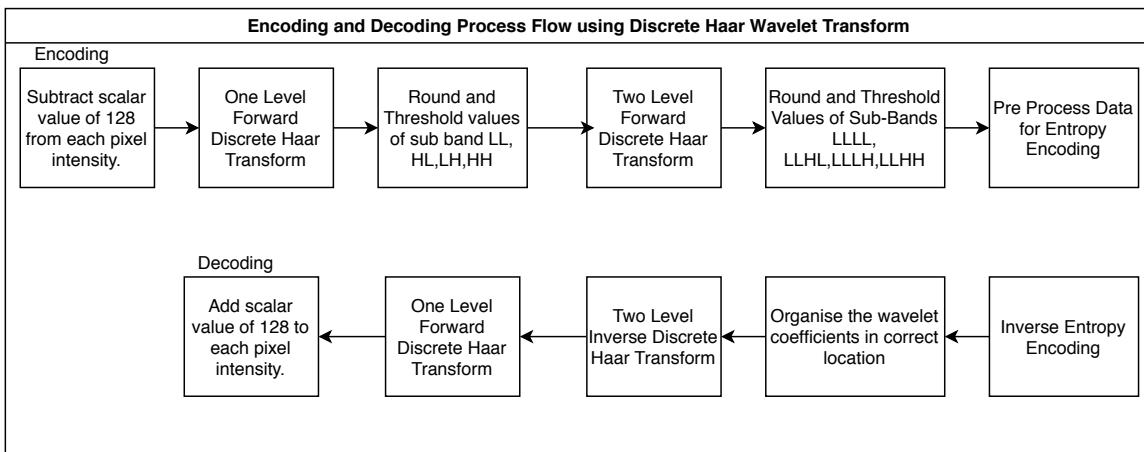


Figure 62: Encoding and decoding using discrete Haar wavelet transform

Figure 63 Shows the decoded approximation of figure 61 after processing. It can be observed that there are little to no human visible distortion artefacts in the recovered image. This validates the use of a discrete wavelet transform for blocks containing computer generated data.

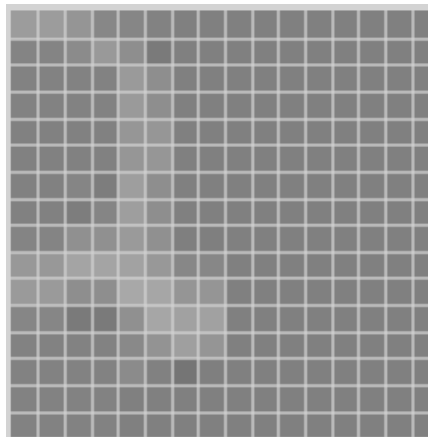


Figure 63: decoded approximation of figure 61 after thresholding, PSNR = 37.12dB

7 Compression File Formats

7.1 JPEG

JPEG (Joint Photographic Experts Group) [18] is a popular method of compression for continuous tone images in both colour and grey scale. It is optimised for image data where neighbouring pixels have similar values. JPEG has both lossy and lossless modes of compression, for continuous tone image, the former typically yields a compression ratio of 10:1 to 20:1 with little discernible impact to the end user, while the latter typically yields a compression ratio from 2:1 to 5:1.

The core values of JPEG are:

- To achieve high compression performance for continuous tone image, such as natural camera captured image.
- To have uniform compression performance, independent from the size of the image.
- To be highly parameterized. Allowing the user to configure settings to optimise the algorithm for particular cases. For example to optimise for compression performance, for speed or for image quality.
- To describe a file format for transmission such that an image can still be decoded if the encoding and decoding have been implemented separately.

JPEG has multiple operation modes. The most common mode is sequential mode where each colour channel (known as a component) is compressed sequentially via a single scan from left to right, top to bottom. Progressive mode is used to compress an image in multiple scans and each scan contains information about the image from coarse to fine detail. Hierarchical mode compresses an image at different resolutions enabling viewing at a lower resolution without fully decoding the image.

The following steps are undertaken when compressing an image using JPEG:

Colour Space Transform

Raw image data or a bitmap file is transformed from RGB colour space to another colour space that will separate the colour information from the intensity information. Typically the colour space used is YCbCr colour space or a variant of it. YCbCr colour space stores the brightness or luminance information in the Y channel, the colour difference between Y and red in the Cr and colour difference between Y and blue in the Cb channel. The reason the image is transformed from RGB colour space is to process the luminance and chrominance information independently. The human

visual system can perceive small changes in luminance information, but is less sensitive in chrominance information, which means that the more data can be discarded in the chrominance data in favour of better compression performance, with little discernible impact to the end user after decompression. This process is not possible in RGB colour space, as each channel is as significant as the other.

Chroma Sub Sampling

Chroma sampling is optional in JPEG compression. It is used to discard redundant information in the chrominance information in favour of better compression performance. This is a lossy step in the encoding process, as once the information is discarded, it cannot be retrieved, however, the noticeable impact is low in terms of subjective observation.

A common pattern that is used for chroma subsampling is YCbCr 4:2:0, which means there is one Cb and Cr value for a neighbourhood of 4 Y channel values.

Decomposition into Data Units

Each colour channel (component) is segmented into 8x8 non overlapping blocks, known as data units. A data unit holds 64 values, each is an eight bit value from 0-255. If the original images size in width and height is not a multiple of 8, the last row and furthest right column are duplicated to make up the difference.

A JPEG encoder can process data units in two modes: non-interleaved and interleaved. In non-interleaved mode, each component is processed fully before moving on to the next component. In interleaved mode, the data units are processed in group of three: first data unit from the Y component, first data unit from Cr component, first data unit from Cb component. Then moving on to the next group of three in a sequential order, from left to right, top to bottom.

The benefit of each data unit being processed individually is the reduction in memory space needed to store intermediate values while compressing a data unit. Also because each data unit is mutually exclusive, the process can be paralleled to improve speed performance. However, one of the main attributing factors to ringing artefacts caused from Gibbs phenomena comes from the fact that each 8x8 block is processed independently, leading to discontinuity around each blocks perimeter.

Discrete Cosine Transform

The forward discrete cosine transform is performed on each data unit, the output is an output map of frequency coefficients, generally called dct coefficients. The dct function is used to decorrelate the information in the input signal and store as much

energy as possible into just a few output coefficients. The most energy, which is the average value is stored in the first output coefficient, known as the DC coefficient, while the remaining higher frequency coefficients, known as the AC coefficients contain the detail of the block.

As the input signal is in two dimensions, the forward transform must be a 2 dimensional transform. To reduce the complexity of the transform and improve on speed, the two dimensional transform is split into two, one dimensional transforms, first performed along the rows and then performed along the columns. This is possible as the key attributes of the discrete cosine function are that it is both separable and reversible. Further improvements can be made to reduce the amount of calculations needed to perform the transform [1] which can reduce the amount of multiplications from 64 per coefficient to 5 coefficient multiplications and 8 scaling multiplications, totalling 13 multiplications.

Even though the discrete cosine transform is both separable and reversible, due to the finite precision in arithmetic used by computers, there will be some error in the transformed signal, due to rounding error, however in practical terms, the error can be negligible.

Quantization

Each of the 64 dct coefficient values are quantized by dividing the value by another value from what's generally known as a quantization matrix. The quantization matrices used in JPEG can be user defined, however there are two suggested matrices in [18] for luminance values and for chrominance values. The values of the quantization matrix are typically called quantization coefficients. A dct coefficient is divided by its quantization coefficient and then rounded to an integer value. The low frequency dct coefficients get divided by smaller quantization coefficients, while the high frequency dct coefficients get divided by larger valued quantization coefficients. The larger the quantization coefficient, the more error that is introduced, but the better the compression performance achieved.

Entropy Encoding

After the dct coefficients have been quantized, the data unit is processed by run length encoding, followed by one of two methods to reduce the entropy in the processed data. The baseline version of JPEG uses a form of Huffman encoding called canonical Huffman coding. The other method of entropy encoding is based on arithmetic coding entropy encoder. Shabahrani et al [37] have performed a series of tests comparing Huffman encoding and Arithmetic coding, Their results show that arithmetic coding can achieve a higher compression ratio but at the cost of speed performance. Their work shows that as the size of the data to be compressed increases, the compression

performance for arithmetic coding is greater, Where as Huffman encoding compression performance is more uniform and independent of file size.

Decompression

The steps for decompression for JPEG encoding are the reverse of the compression stages. This enables the compression and decompression time to be symmetric.

7.2 PNG

PNG (Portable Network Graphics) is a popular lossless compression file format for images which is used extensively in web applications, mobile application and computer game applications. It is a non patented technology which has allowed for its integration on a large scale in most applications that need to transmit image data from one source to another. It is a fully lossless compression format and processes image data in two stages. The first stage is the filtering process and the second stage is the compression process. Central to the compression stage is the Deflate [7] algorithm which is a combination of LZ 77 [51] and Huffman encoding [16].

PNG compression is both efficient in terms of speed performance and achieves good compression performance for images containing a mixture of computer generated data and natural continuous tone image. However for images that fully contain natural image, such as images for medical and security processing, compression performance is degraded, with typical compression ratios of 1.1:1 to 2:1 [49] [15]

PNG compression processes the image in a raster scan order and processes the colour channels of an image separately. PNG does not implement colour space transformation as its a lossless method where each pixel value of each colour channels contribution is of equal importance.

Filtering

Before applying compression, a filtering process is performed on each row of a channel of the image. Delta coding is used to implement filtering. The role of delta coding is to take advantage of data that may be linearly correlated. In delta coding, a given sample value can be represented as the difference between its numeric value and a previous sample value. This type of of processing can result in groups of repeated numbers, which will improve the performance of the compression algorithm in the second stage.

Table 23 shows X_n which is an array of numbers that have a linear correlation that would be suitable for delta encoding. using the formula

$$Y_n = X_n - X_{n-1} \quad (29)$$

$X_n =$	128	129	130	131	131	131	131	131
---------	-----	-----	-----	-----	-----	-----	-----	-----

Table 23: X_n is an array of values that have linear correlation

The output can be seen in Table 24.

$Y_n =$	128	1	1	1	0	0	0	0
---------	-----	---	---	---	---	---	---	---

Table 24: Y_n is an array of values after delta encoding

It can be observed from table 24 that the dynamic range of sample values have been greatly reduced and the runs of repetitive patterns would be suitable for lossless compression.

There are five filter types that are used in the filtering stage in PNG. The filter choice can be chosen per scan line but the same filter must be used for each channel per scan line. The first filter choice is to use no filtering.

The second filter choice **Sub**, is shown in equation 30, where X is a row of an image and N is a single channel pixel values at a given coordinate. The difference value is calculated between a given channel pixel value and the value directly to the left in the same row.

$$Y_{[N]} = X_{[N]} - X_{[N-1]} \quad (30)$$

The third filter, **Up**, is shown in equation 31. The difference calculation is calculated between a given channel pixel value and the the sample directly above it, in the previous row of the image.

$$Y_{[N]} = X_{[N]} - [X - 1]_{[N]} \quad (31)$$

The fourth filter **Average** is shown in equation 32. The difference value is calculated by subtracting a given pixel value from the average value calculation of the value directly to the left and directly above a given pixel.

$$Y_{[N]} = X_{[N]} - \frac{X_{[N-1]} + [X - 1]_{[N]}}{2} \quad (32)$$

The fifth filter **PAETH** [29] computes a linear function using the sample directly to the left, the sample directly above and the sample to the top left of a given channel pixel value, then chooses the neighbouring value as a predictor that is closest to the computed value. The psuedo code to implement the paeth filter is described in [29] and shown in listing 8

Listing 8: PAETH Filter Pseudo code


```

1 function PaethPredictor (a, b, c)
2   begin
3     // a = left, b = above, c = upper left
4     p := a + b - c      // initial estimate
5     pa := abs(p - a)   // distances to a, b, c
6     pb := abs(p - b)
7     pc := abs(p - c)
8     // return nearest of a,b,c,
9     // breaking ties in order a,b,c.
10    if pa <= pb AND pa <= pc then return a
11    else if pb <= pc then return b
12    else return c
13  end

```

Compression using Deflate

After filtering has been performed on each row of each channel of the input image, the processed data is ready for compression using a compression algorithm called deflate which is a combination of LZ 77 and Huffman encoding.

The LZ 77 algorithm which has been described in section 6.1.2 iterates the length of the input data sequentially and will store any pattern matches into a buffer. This approach is known as a sliding window, which is made up of a search buffer and a look ahead buffer. Potential patterns in the look ahead buffer are searched for in the search buffer and if found, the output is in the form of a triple, (o,l,c):

- o = offset: represents the number of positions that is needed to move backwards in order to find the start of the pattern match
- l = length: represents the length of the match
- c = character: represents the character directly after the match.

The search algorithm is a linear search, which would give a time complexity of $O(n^2)$. For a large image this would have a significant impact on speed performance and because of this, the standard [19] limits the size of the sliding window to 32KB.

The LZ 77 data is then processed with Huffman encoding. There are two methods of Huffman encoding available. The first method uses predefined Huffman trees which are defined in [7]. This method can be used for an improvement in speed performance as the data does not need to be parsed again to update the frequency of symbols. another benefit is the Huffman trees are not needed to be sent with the compressed data as the encoder already has knowledge of them. The second method is to define the Huffman trees from the input data. This method incurs additional overhead in processing time, however may achieve better compression.

Compressed file format ordering

A PNG file is split up into chunks. There are three mandatory chunks and many optional chunks. Table shows an example of the mandatory chunks and one optional and their ordering

PNG Chunk name and ordering		
Chunk	Mandatory	description
IHDR	Yes	Header. Comes first in the stream
PLTE	no	Pallete. Comes second only if defined colour set is used
IDAT	yes	Data. Can be any multiple of data chunks in order
IEND	yes	End. Signalling end of stream

Table 25: Description of Chunks in a PNG file

8 Video Encoding

8.1 H.264

H.264 [47] is a mature video coding technology first published in 2003 and is still in use as an industry standard for video compression and in streaming video content online.

The encoder process in H.264 is broken down into the following steps

- 1 Frame decomposition
- 2 Prediction
- 3 Transform
- 4 quantization
- 5 Entropy encoding the bit stream

Frame decomposition

Similar to JPEG [18], the frames to be encoded with H.264 are first transformed from RGB colour space to YCbCr colour space. H.264 can incorporate chroma subsampling, Using YCbCr 4:4:4, 4:2:2 and 4:2:0 ratios.

A frame is then decomposed into 16x16 pixel blocks which are called Macro Blocks. Depending on which chroma subsampling rate is used, a Macro Block will contain 4 8x8 blocks of Y channel samples and between 1-4 blocks of 8x8 samples for each colour channel.

Prediction

H.264 can use encoded data from previous frames (inter prediction) to encoded a given Macro Block in a current frame. The values of a given Macro block are subtracted from a Macro Block from a previous frame, with the result is known as a Residual. It can also apply prediction from within the current frame (intra prediction). Intra prediction can use variable block sizes from 16x16 to 4x4 to predict the Macro Block from previously encoded blocks in the neighbourhood of directly left, top left and top.

Prediction is used to reduce redundant information in frames of a sequence that have highly correlated data, such as data in the background that may only slightly change from one frame to the next. The values of the Residual are sparse and have a low dynamic range, which will improve the performance of the compression applied in further steps.

Motion estimation is a type of prediction that can be used in H.264. It is used to map motion in objects in successive frames. Motion estimation is described and coded using Motion Vectors.

Transform

The residual sample values from prediction are then processed with a variant of the discrete cosine transform which is implemented using integer arithmetic. This step is similar to JPEG but differs by using an integer implementation and that it can choose to perform the transform on blocks of size 8x8 or 4x4. A Macro block can use any combination of 8x8 and 4x4 transforms in any given order.

Quantization

The output coefficients of the integer transform are further processed by quantization, where each integer output coefficient is divided by a weighted quantization coefficient. H.264 allows the Quality of quantization to be determined, by adjusting the quantization parameter (QP). High QP values result in more of the integer output coefficients reduced to zero, in turn increasing compression performance at the expense of subjective decoded image quality.

Entropy encoding the bit stream

H.264 uses a type of entropy encoding called Context Adaptive Binary Arithmetic Coding (CABAC) [25]. CABAC implements the following steps:

- 1 Binarization: using binary arithmetic coding, the alphabet size is 2. a symbol such as a transform coefficient is transformed into a binary string before arithmetic coding.
- 2 Context model selection: A probability model is selected for each bit in the binary string depending on the statistics of previously encoded binary strings.
- 3 Arithmetic coding: each bit in the binary string is encoded with an arithmetic coder according to the probability model that is selected.
- 4 Probability Update: Once the bit in the binary string has been encoded, the context model is updated.

9 Compound Compression Algorithm

In this section, the image classification and compression techniques required to create a compound compression algorithm will be developed and tested. The goal is to design the framework of a compound compression algorithm that will be efficient in terms of compression performance, computational complexity and can deliver acceptable levels of both subjective and objective image quality.

A set of 40 high quality 24 bit bitmap compound images have been prepared and will be used as the test set, for both classification and compression, with some additional images used as a training set for the classification algorithm. The test set of images can be seen in the appendix of this thesis in figures 134 - 137. The set of test images were generated on a computer using Windows 10 operating system, using the Windows 10 application programming interface to capture the screen image.

The classification algorithm will first be described and tested, followed by lossless and lossy compression algorithm analysis and testing. A novel lossless compression algorithm will be proposed, which has been developed from the findings of the analysis on the amount of unique pixel values in a given block of pixels that contain computer generated data. The novel compression algorithm will be compared to standard compression algorithms in terms of both compression performance and computational complexity.

9.1 Classification Algorithm Configuration

A series of tests are configured to develop a classification algorithm which builds upon the research presented by *Wu* in [48]. The classification algorithm used for compound image compression decomposes an input image into 16x16 non overlapping blocks and classifies each block as one of five types, which are: smooth, sparse, text, picture and fuzzy and have been described in section 4.1

Block classification is done by a combination of colour counting and a discrete wavelet transform. The discrete wavelet transform is used to decompose a block into four sub bands which contain transient information which can be used to discern if a given block contains properties of text, computer generated data or natural image. The sub bands are in essence four filter banks which are described in table 26 and their location with respect to a 2 dimensional array is shown in table 26.

Sub Band	Row Filter	Column Filter	Description
LL	Low	Low	averaged and scaled approximation of original image
HL	Hi	Low	contains horizontal stroke data
LH	Low	Hi	contains vertical stroke data
HH	Hi	Hi	contains diagonal stroke data

Table 26: Description of the discrete wavelet transform sub bands

LL	HL
LH	HH

Table 27: Location of the four sub bands within a block

Statistical analysis is performed by computing the standard deviation on the sub bands LH, HL and HH. The hypothesis is that blocks containing text and computer generated data will contain sharp transitions in vertical, horizontal and diagonal directions. Blocks containing sharp transitions in these sub bands should have a higher standard deviation than blocks that do not contain text or computer generated data. Blocks that contain natural image but are less structured, such as fuzzy blocks, should have a lower standard deviation than blocks that contain natural image with structure.

The purpose of the following test is to examine a set of specifically chosen images that contain a majority of blocks that could be classified as sparse, text, fuzzy and picture blocks. The test will process each image by decomposing it into 16x16 blocks, colour space transform, discrete wavelet transform and statistical analysis to ascertain whether it can distinguish a type of block by analysing the sub bands. Images that

only contain smooth blocks have been omitted from this test as smooth blocks contain a single colour, thus will have no variance or deviation from the mean value. The computed values will then be used in the classification algorithm and tested on a set of forty compound images, which contain a mixture of all of the above types of blocks.

Experimental Design

Twelve images are selected for the test. All images are size 512x512 pixel 24-bit bitmaps and are public domain images. Each image will contain a majority of one of the classification types. This initial classification is performed subjectively by viewing the images.

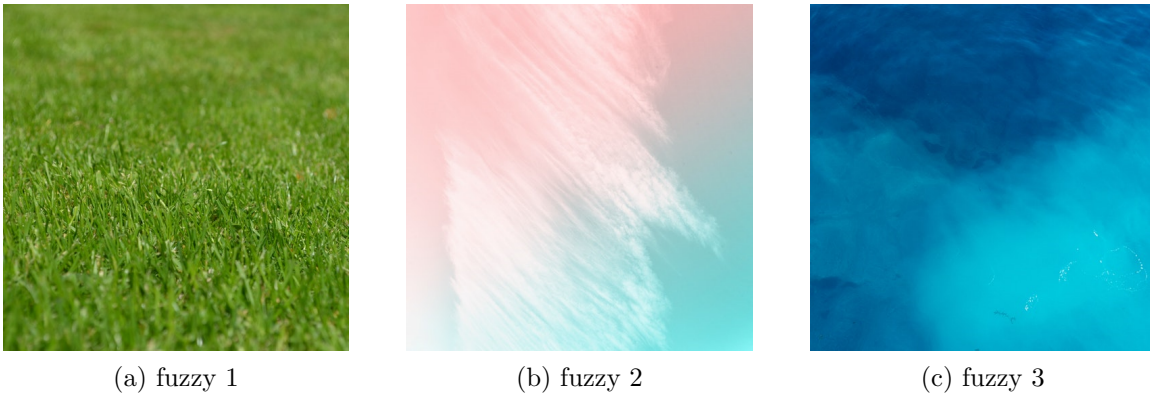


Figure 64: Test images that contain data subjectively classified as fuzzy blocks

Figure 64 Shows the three images that have be chosen as fuzzy. Each image contains natural continuous tone image with no apparent structure.

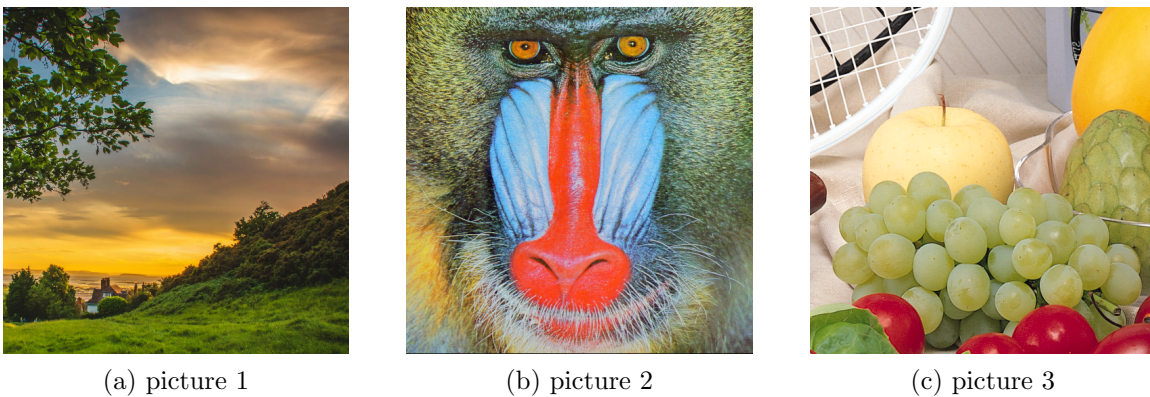


Figure 65: Test images that contain data subjectively classified as picture blocks

Figure 65 Shows the three images that have be chosen as picture. Each image contains natural camera captured image data with defined objects such as landscape, animals

and fruit as well as depth of perception such as background and foreground. The content in each image is more complex than that of images that have been classified as predominately fuzzy.

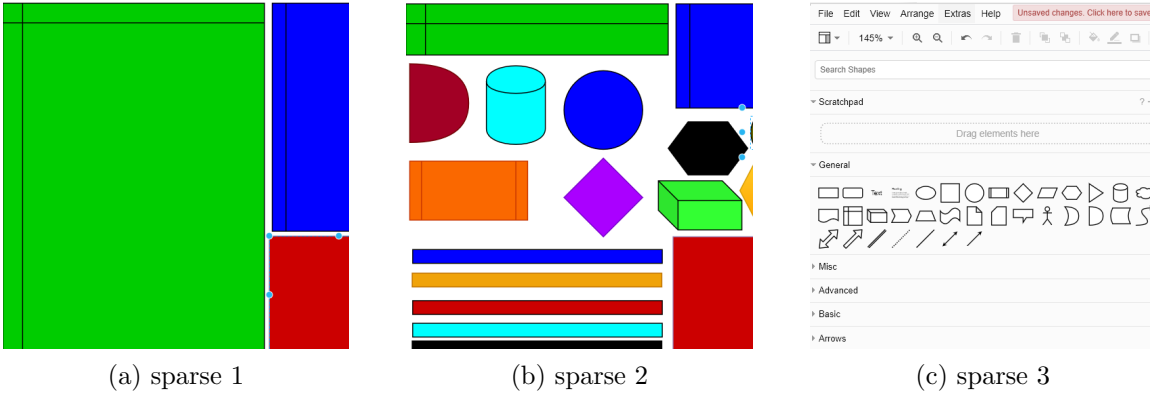


Figure 66: Test images that contain data subjectively classified as sparse blocks

Figure 66 Shows the three images that have been chosen as containing sparse data. Each image is computer generated with sharp lines in vertical, horizontal and diagonal directions in smooth regions, as well as areas of single colour.

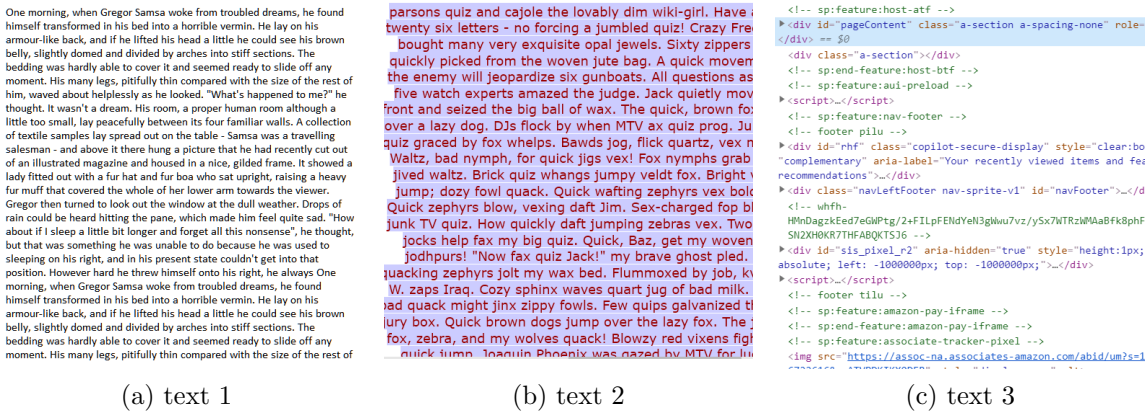


Figure 67: Test images that contain data subjectively classified as text blocks

Figure 67 shows the three images that have been chosen as containing text data. The first image is plain text on white background, the second image is coloured text with colour background and the third image contains multicoloured text with many symbols and lines. The three text images have been chosen to be representative of typical text editing applications that may be run on a computer.

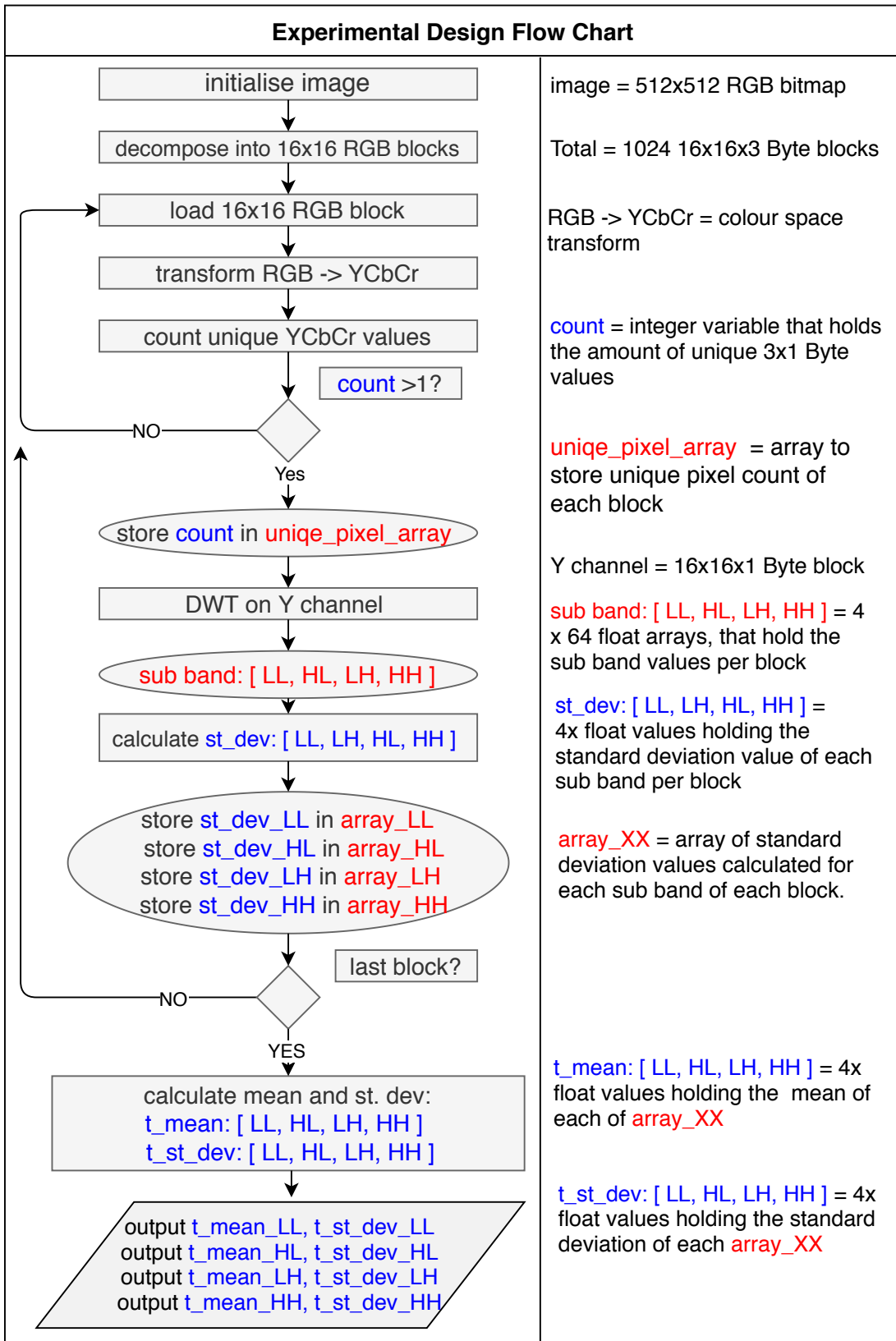


Figure 68: flow chart for experimental design to calculate threshold values for classification algorithm

Figure 68 is a flow chart of the experiment. It implements the following steps:

- A test image is loaded into the program.
- The image is decomposed into 16x16 non overlapping blocks.
- An iterative for loop is started that will process each block.
- A block is transformed from RGB colour space to YCbCr colour space (with no chroma subsampling)
- Every 3 byte pixel in the block is counted. The total count of unique 3 byte pixels in the block is stored in an array. If there is only 1 unique pixel value, the block is discarded as a smooth block and the next block is loaded. This experiment is concentrated on sparse, text, fuzzy and picture blocks.
- The forward discrete wavelet transform is performed on the Y channel of the block. The Y channel is chosen as it contains information about all colour channels. The 256 samples of the Y channel are decomposed into four 64 sample array sub bands: LL, HL, LH and HH, by process of discrete Haar wavelet transform.
- The standard deviation for each sub band is computed and the result for each block is stored in a separate array.
- After all the blocks have been processed, there are four arrays that contain the standard deviation value for each sub band per block and also an array that holds the count of how many unique pixels there are per block. To get base line values to use in the full classification algorithm that will be used to classify compound images, The mean value along with the standard deviation is calculated on the four arrays holding the per block standard deviation values. The mean will set the baseline for the threshold, while the standard deviation will inform whether or not a specific sub band can be used in the classification.

Experimental Results of total mean standard deviation and associated standard deviation

Table 28 shows the results of calculating the total mean of array_LL, array_HL, array_LH and array_HH from figure 68 and their associated standard deviation, for each image in the fuzzy image set. It can be observed for *fuzzy 2* and *fuzzy 3*, the mean standard deviation value in sub bands HL, LH and HH is low, indicating that there is low transient information in the image. This is confirmed by observing the images in figure 64.

The LL sub band is a scaled and averaged approximation of the original block, at $\frac{1}{4}$ of the size. As the standard deviation is calculated on the Y channel, which holds the

luminance information, it can be seen that the mean value in *fuzzy 1* is greater than both *fuzzy 2* and *fuzzy 3*. This is due to figure 64a having finer detail in the blades of grass in the centre of the image, compared to the foreground and background. The luminance in both *fuzzy 2* and *fuzzy 3* is quite uniform, which is confirmed by the low standard deviation in the LL sub band while the larger standard deviation for *fuzzy 1* is due to higher definition that can be observed in the grass in 64a.

Fuzzy	Calculation	Sub Band LL	Sub Band HL	Sub Band LH	Sub Band HH
fuzzy1	t_mean	12.32	4.85	3.23	1.44
	t_st_dev	7.05	4.94	3.00	1.93
fuzzy2	t_mean	2.34	0.56	0.60	0.13
	t_st_dev	1.91	0.75	0.79	0.36
fuzzy3	t_mean	7.26	6.82	8.84	5.98
	t_st_dev	1.54	2.46	3.04	2.81

Table 28: standard deviation analysis for fuzzy image set

Table 29 shows the resulting calculations for the picture image set. The first observation is both the total mean standard deviation and its associated standard deviation in sub band LL for each image in the picture set is larger than that of the fuzzy set. This is because the images in figure 65 contain defined objects, where the brightness of the image is localised around a given object.

The total mean standard deviation value of The HL, LH and HH sub bands is also greater than that of the fuzzy image set, This is due to more structural information in the images, where there is transient information about the outline of each object in the images, however the mean standard deviation value is still low as the outline of the objects in the images are softer and less defined than that of computer generated data. The gradient is slower between object boundaries as the brightness levels change due to shadowing. This is an inherent attribute to natural camera captured image data.

Picture	Calculation	Sub Band LL	Sub Band HL	Sub Band LH	Sub Band HH
picture1	t_mean	14.38	7.04	9.32	5.25
	t_st_dev	12.05	6.18	6.89	4.35
picture2	t_mean	20.72	11.65	16.79	8.97
	t_st_dev	9.58	5.87	9.87	4.56
picture3	t_mean	13.21	7.23	8.39	3.86
	t_st_dev	14.55	5.28	5.22	1.88

Table 29: standard deviation analysis for picture image set

Sparse	Calculation	Sub Band LL	Sub Band HL	Sub Band LH	Sub Band HH
sparse1	t_mean	34.45	12.12	14.11	0.33
	t_st_dev	31.52	11.50	21.92	1.39
sparse2	t_mean	54.25	7.62	22.63	2.07
	t_st_dev	34.28	12.18	19.59	4.07
sparse3	t_mean	19.51	11.20	14.93	5.48
	t_st_dev	20.11	15.32	15.11	8.20

Table 30: standard deviation analysis for sparse image set

Table 66 shows the resulting calculations from the sparse image set. The sparse images contain highly structured objects with well defined object borders. The luminance of the object in the images is uniform and localised to each object. This is reflected in the mean standard deviation in the LL sub band and its associated standard deviation. The mean standard deviation value is high because the luminance value of each object is different and the transition from one object to the next is sharp.

The mean standard deviation of the HL and LH sub bands is high because there is a lot of well define horizontal and vertical strokes in the images. The HH sub band is low, as it can be observed that there is a lower amount of diagonal stroke information, which would be more prominent in text data.

The high associated standard deviation values in the LL sub band for sparse blocks would limit its effectiveness for classification, as it shows that there is a high dynamic range for that sub band.

Sparse	Calculation	Sub Band LL	Sub Band HL	Sub Band LH	Sub Band HH
text1	t_mean	53.86	41.23	42.96	22.04
	t_st_dev	10.98	14.55	12.72	6.15
text2	t_mean	42.94	26.44	23.59	9.89
	t_st_dev	9.73	10.46	11.21	3.98
text3	t_mean	29.86	19.26	27.50	11.62
	t_st_dev	8.65	8.38	12.77	4.99

Table 31: standard deviation analysis for text image set

It can be observed in table 31, that text information has the highest total mean standard deviation values for the HL, LH and HH sub bands. This is due to the attributes of text data. Text characters are highly structured and can contain strokes in vertical, horizontal and diagonal directions, with the luminance level of the text in high contrast to the smooth background. This is indicative that the hypothesis

is correct and that image data that contains computer generated text information will have a higher standard deviation in the LH, HL and HH sub bands compared to continuous tone image. This is a fundamental result, as it suggests that the data types in an image can be classified by their inherent statistical properties when decomposed using a discrete wavelet transform.

Type	Calculation	Sub Band LL	Sub Band HL	Sub Band LH	Sub Band HH
Fuzzy	mean	7.31	4.08	4.22	2.52
	standard dev	4.07	2.61	3.44	2.51
Picture	mean	16.10	8.64	11.50	6.03
	standard dev	3.3	2.13	3.76	2.16
Sparse	mean	36.07	10.31	17.22	2.63
	standard dev	14.23	1.94	3.84	2.14
Text	mean	42.22	28.98	31.35	14.52
	standard dev	9.81	9.15	8.36	5.37

Table 32: Final mean and standard deviation values to derive threshold values for classification algorithm

Table 32 shows the final stage of calculations to generate threshold values for use in the compound image classification algorithm. The insight gained by performing the above test is as follows:

- The standard deviation in the LL sub band of blocks in the fuzzy image set is the lowest of any of the image types. This is an expected outcome, as the luminance values of a 16x16 block of pixels should be consistent, but with each individual value slightly varying to the next. As the LL sub band is an approximation of the original block, but scaled by a half in both horizontal and vertical directions, it is a good indication of the overall texture of the block and so can be used in classification.
- The amount of unique pixels per 16x16 block in the fuzzy image set is also greater than the sparse and text image set. This is also an important observation, as it will help to distinguish fuzzy blocks from certain sparse blocks that may have a similar standard deviation in the LL sub band.
- The unique pixels per 16x16 block in the sparse image set is the lowest of any of the image types.
- Sparse blocks are generally computer generated data are associated with a computers graphical user interface (GUI) and are found around the perimeter of windows or tool bars, and may have very sharp transitions in luminance, as

such they usually have a dominant value in one of the sub bands HL, LH or HH.

- However, some blocks from the sparse image set may have a similar standard deviation in the LL sub band as blocks from the fuzzy image set. this is because the majority of a given block from the sparse block set may be the same colour, with the exception of a small amount of pixels. To be able to to classify sparse blocks the two attributes needed is a low unique pixel count and a dominant standard deviation in one of the sub bands LH, HL or HH.
- The amount of unique pixel values per 16x16 block in the picture block are the largest of any of the image sets.
- The standard deviation in the sub bands LH, HL and HH are generally lower than those in the text image set, however some of the block do contain sharp transient information, such as the fine hair detail in the image of the baboon in figure 65. To correctly identity blocks as picture blocks, the two attributes that are needed are a high unique pixel count and a threshold value for the LH, HL and HH sub bands that a picture block will not crossover.
- Blocks in the text image set have a higher unique pixel count than the sparse image set, but a lower unique pixel count than the fuzzy and picture set.
- Blocks in the text image set have high transient information in the sub bands HL, LH and HH, which can be used for classification.

From the above insights a classification algorithm has been created to be used for classifying the data in compound images in preparation for compression. Figure 70 shows the flow chart of the classification algorithm that will be used to classify the data types of a compound image. The conditional logic for the flow chart can be seen in figure 69.

Conditional Logic for Classification Algorithm	
condition 1 = [count < 5] && [st_dev_LL < 9] condition 2 = [count > 5] && [st_dev_LL < 12] condition 3 = [count < 10] && [st_dev_HL > (st_dev_LH * factor && st_dev_HH * factor)] condition 4 = [count < 10] && [st_dev_LH > (st_dev_HL * factor && st_dev_HH * factor)]	condition 5 = [count < 10] && [st_dev_HH > (st_dev_HL * factor && st_dev_LL * factor)] condition 6 = [st_dev_HH < 25] && [st_dev_HL < 25] && [st_dev_LL < 25] condition 7 = [st_dev_HH > 25] [st_dev_HL > 25] [st_dev_LL > 25]
st_dev_LL = standard deviation of LL sub band st_dev_HL = standard deviation of HL sub band st_dev_LH = standard deviation of LH sub band st_dev_HH = standard deviation of HH sub band count = amount of unique 3 byte pixels per block factor = integer multiple = 4	

Figure 69: Logic description and legend for figure 70

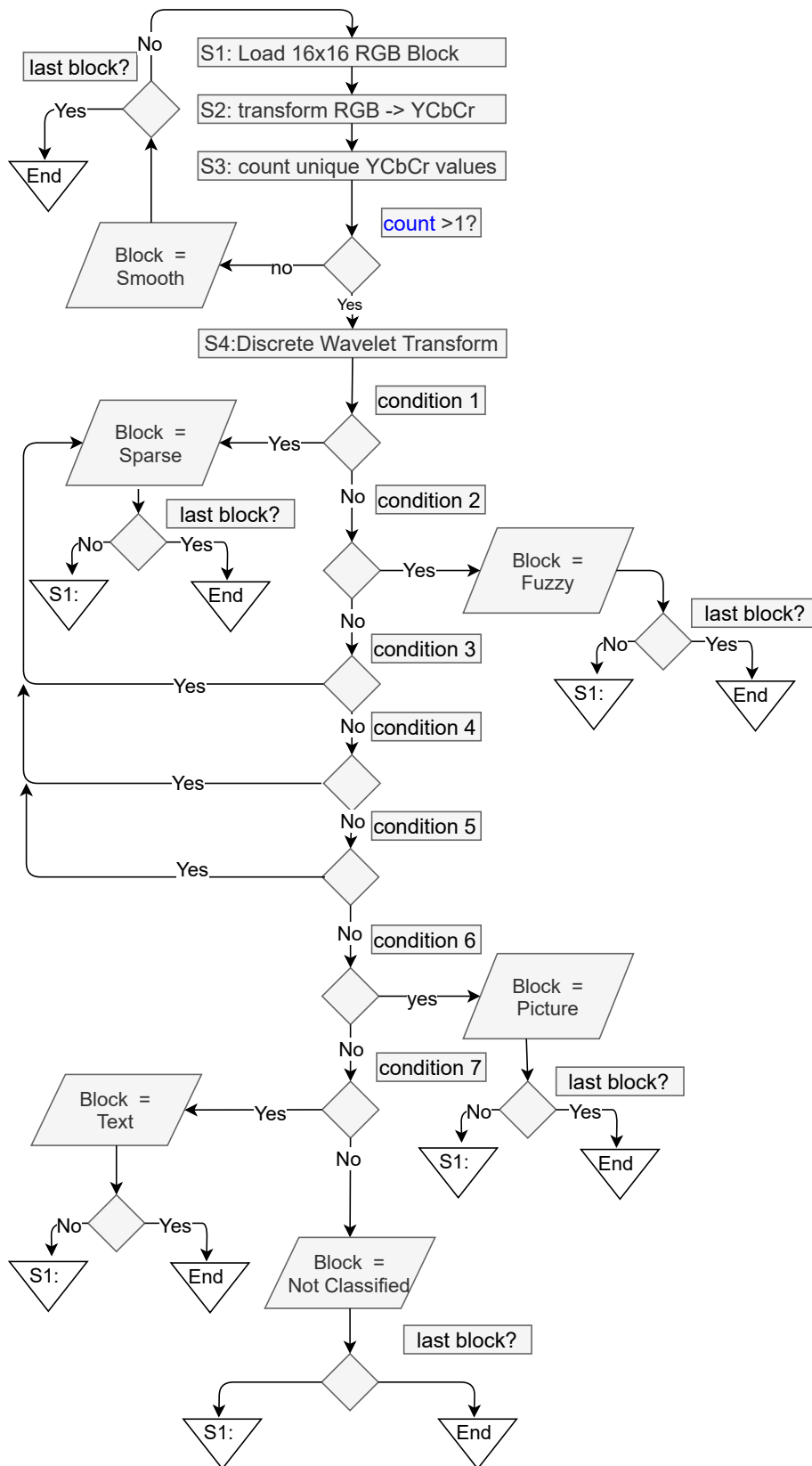


Figure 70: Flow Chart of Classification algorithm for compound images

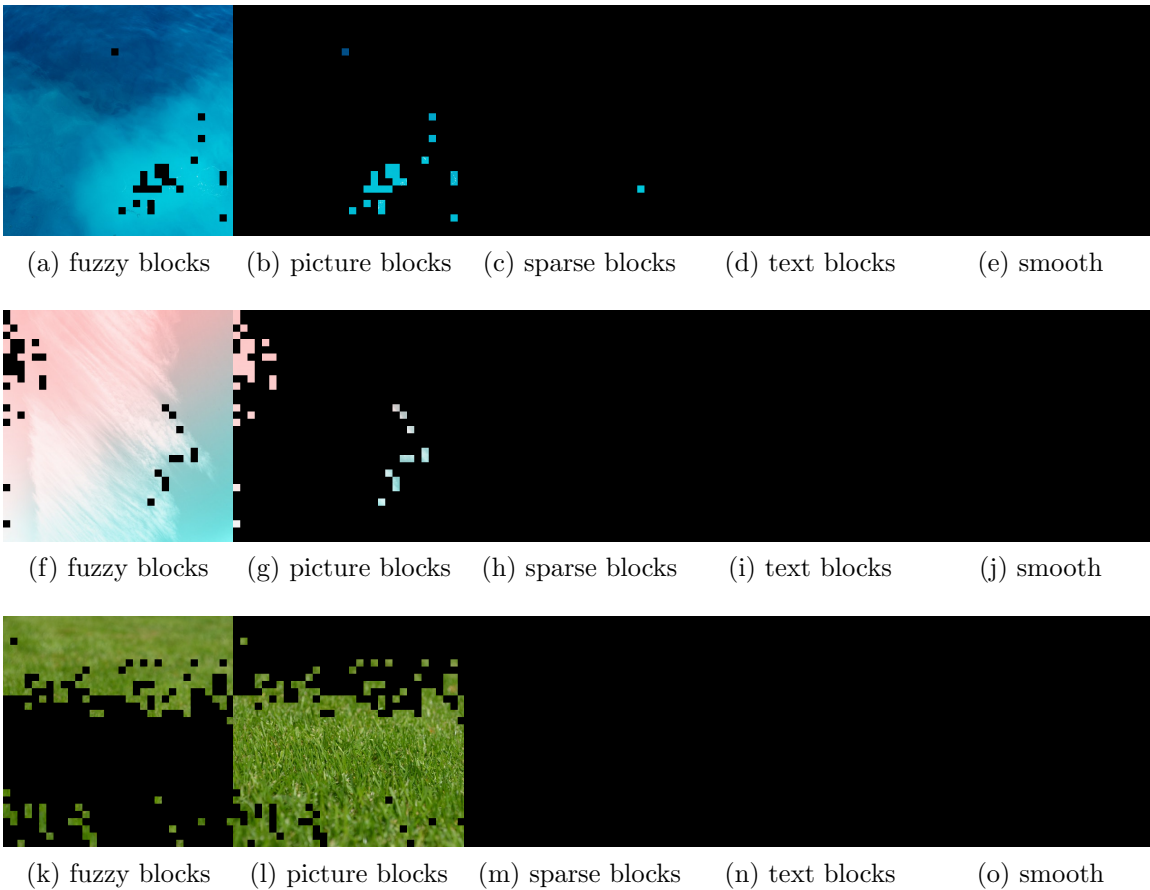


Figure 71: Fuzzy image classification results

Figure 71 shows the fuzzy image set after classification and decomposed into each block type. There were no blocks that were not classified in this test. There were also no blocks misclassified as text and only a single block classified as sparse. It can be observed that the classification based on the LL sub band is accurate in classifying fuzzy blocks.

It can be seen in figure 71 that the majority of blocks in the image of the grass have been classified as picture. This is because there is finer resolution in the detail of the grass, which is an expected result.

The intended compression method to compress blocks classified as fuzzy and picture is based on discrete cosine transform followed by quantization, albeit with stronger quantization used for fuzzy, compared to picture.

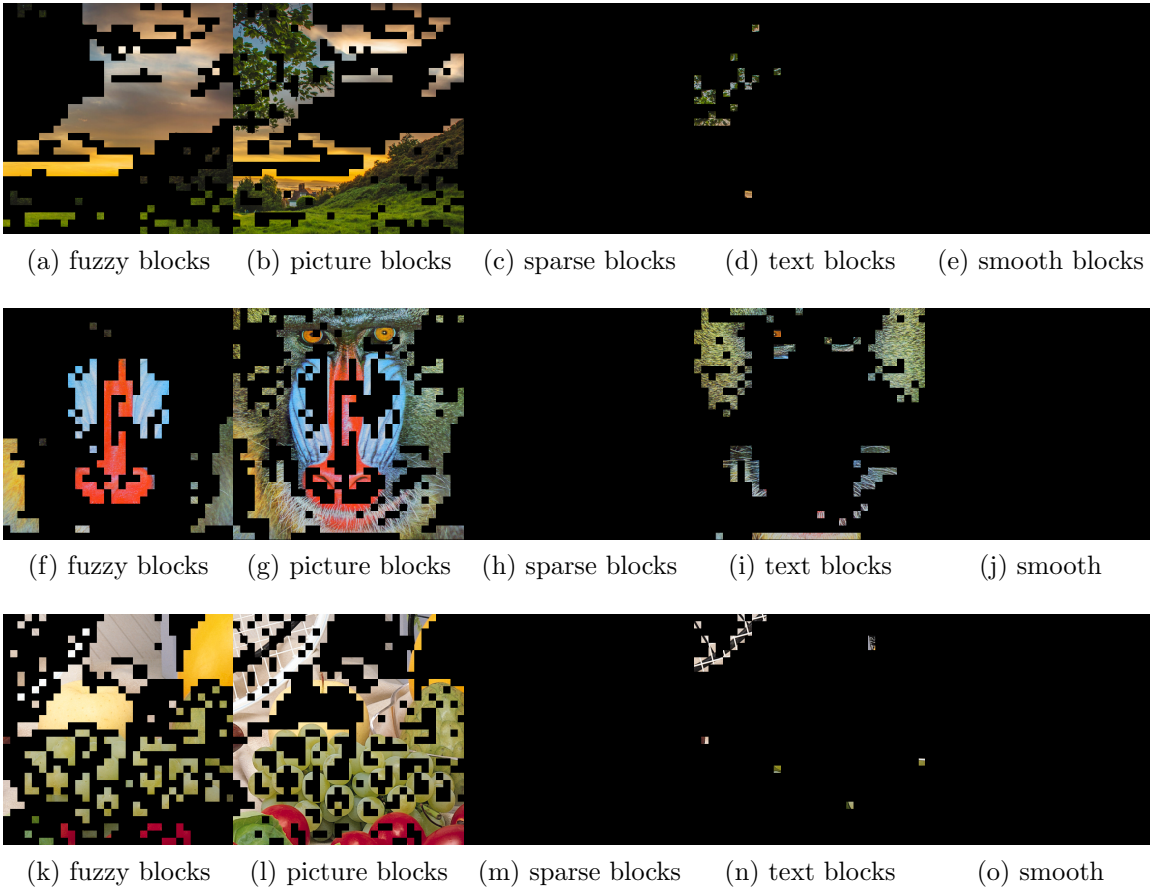


Figure 72: Picture image classification results

Figure 72 shows the results of classification on the picture image set shown in figure 65. The majority of the blocks in the first and third image of the set have been classified correctly as either picture blocks or fuzzy blocks. In both the first and third image, the blocks that are classified as picture blocks contain significantly more detail than the blocks that are classified as fuzzy blocks, which have a more uniform luminance level per block.

There are 24 blocks in the first image and 30 blocks in the third image that are misclassified as text, which gives an accuracy of approximately 98%. However, there are a significant amount of blocks that have been misclassified as text in the second image. The second image of the picture image set depicts an image of a baboon, which has very fine detail in in the fur, with many sharp horizontal and diagonal strokes. When a discrete wavelet transform is performed on the blocks with fur, there would be a large amount of transient information in the HL, LH and HH sub bands, which would be similar to text, which is why the blocks are misclassified. Another condition is needed to catch these types of blocks.

Image number from figure 65	1	2	3
No. of blocks classified as picture	425	580	510
picture blocks with 32+ unique pixels	425	580	478
percentage (%)	100	100	93.7
No. of blocks classified as fuzzy	575	240	484
fuzzy blocks with 32+ unique pixels	575	229	478
percentage (%)	100	95.4	98.7
No. of blocks classified as text	24	204	30
text blocks with 32+ unique pixels	24	204	30
percentage (%)	100	100	100

Table 33: Breakdown of classification of picture image set, with colour counting

Table 33 shows a breakdown of the classification process for the picture image test set. It shows the amount of blocks that are classified as picture, fuzzy and text, as well as the number of each type that has 32 or more unique pixel values and what the percentage is. It can be observed that the majority of blocks that are classified as fuzzy and picture have a unique pixel count that is greater than or equal to 32 unique pixel values. Also, every block that has been misclassified as text in each image has a unique pixel count that is greater than 32. This is a very important result, because it shows that using the amount of unique pixels per block can be used as another condition to aid classification for picture blocks.

```

1 condition 6 = [ count > 31
2                 && st_dev_HL < 25
3                 && st_dev_LH < 25
4                 && st_dev_HL < 25 ]

```

By modifying the conditional logic for condition 6 in figure 69 to the condition above, the classification algorithm is re-run on the picture test image set.

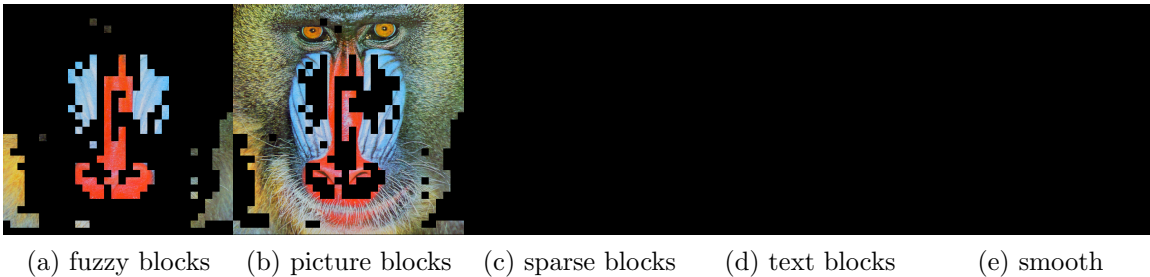


Figure 73: Picture image classification results with modified condition including unique colour count

Figure 73 shows the results of the image 2 from the picture image set classified with the modified condition. It shows that all blocks have been classified correctly.

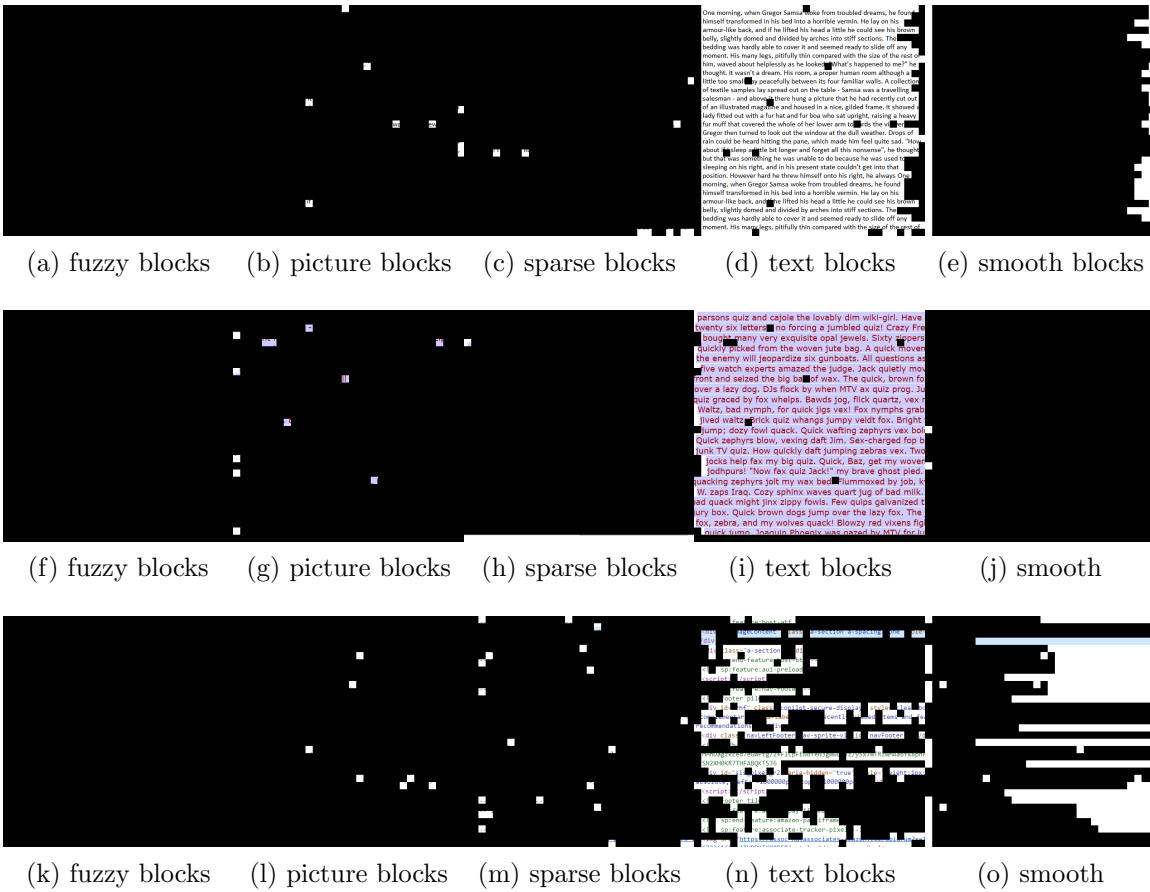


Figure 74: Text image classification results

Figure 74 shows the results of the text image set after classification. Most notably, it can be observed that there are 9 blocks in 74b and 7 blocks in 74b that are misclassified as picture blocks. Picture blocks are intended to be compressed with a lossy compression, while text blocks are to be compressed with a lossless method. If blocks containing text are compressed with a lossy compression method such as one based on the discrete cosine transform which is intended for use with picture and fuzzy blocks, there will be compression artefacts that will reduce the overall quality of the image. Table 34 shows a count of the amount of blocks classified as text in the text image set that have over 32 unique pixel values

Image	1	2	3
No. of blocks classified as text	937	965	652
of classified, blocks with 32+ unique pixels	2	6	0
percentage (%)	.002	.006	0

Table 34: Table of blocks that have been classified as text from the text image set containing more than 32 unique pixel values

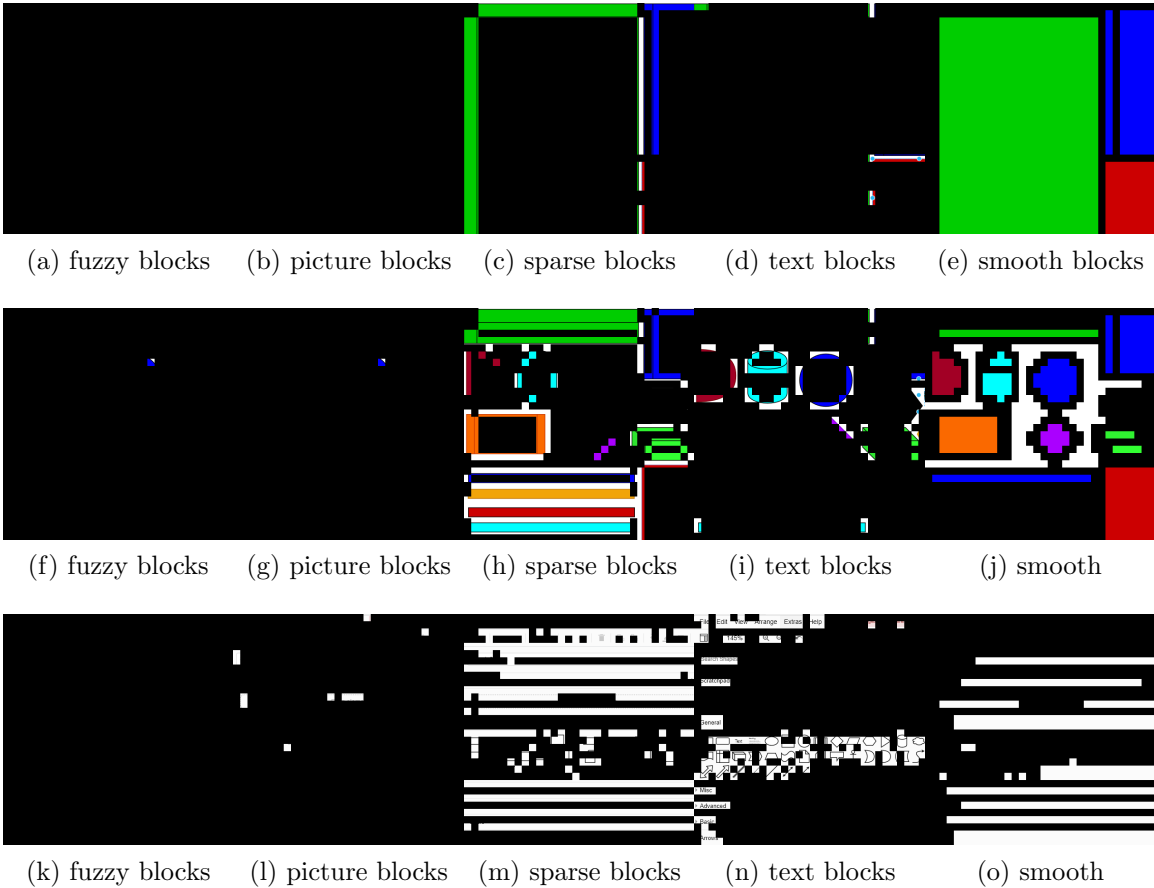


Figure 75: sparse image classification results

Figure 75 shows the results of the sparse image set after classification. Sparse classification has a high degree of accuracy of 99.7%

It can be observed from 75(n) that the blocks that are classified as text represent blocks that contain text when visually observing 66.

The insight gained by performing the above test is as follows:

- The classification algorithm, with base line threshold values is successful in classifying the data type in each image of the test sets.
- sparse block classification is the most accurate as there is well defined information in the LH, HL and HH sub bands.
- using the LL sub band is an effective method to classify fuzzy blocks in combination with a count of the unique pixel values of a block is a success method to classify fuzzy blocks.
- There is a possibility to misclassify picture blocks as text blocks and vice versa if the only attribute that is being analysed is the information in the HL, LH and HH sub bands.

- There is much larger unique pixel count in picture and fuzzy blocks than sparse and text blocks

Using the insight gained from the previous tests, the classification algorithm will be tested on a full compound image from a set of 40 high resolution compound images.

Figure 76 shows a compound image containing a mixture of the five classes of data found within a compound image. The image is representative of a typical computer desktop image, with a file explorer application open and an image viewing application which show attributes of the graphical user interface along with text data. The background is smooth in texture and predominately made up of a single colour.

The image of the almonds is chosen as it contains highly structured objects with varying degrees of brightness and colour. The image is from a public domain standard test set of lossless images of size 768x512 pixels, originally released by Kodak retrieved from [27].

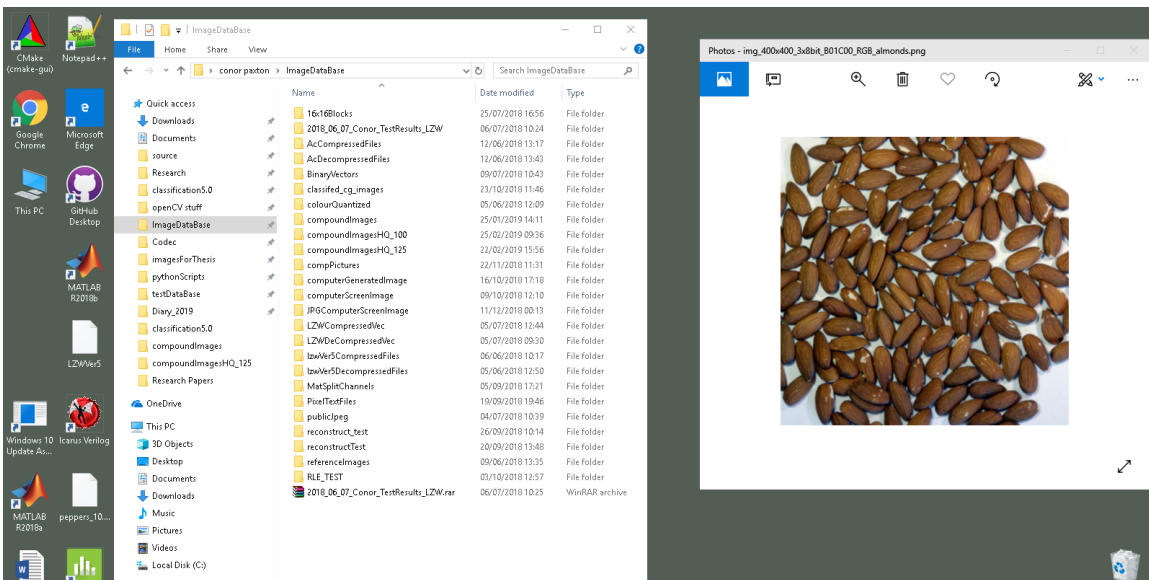
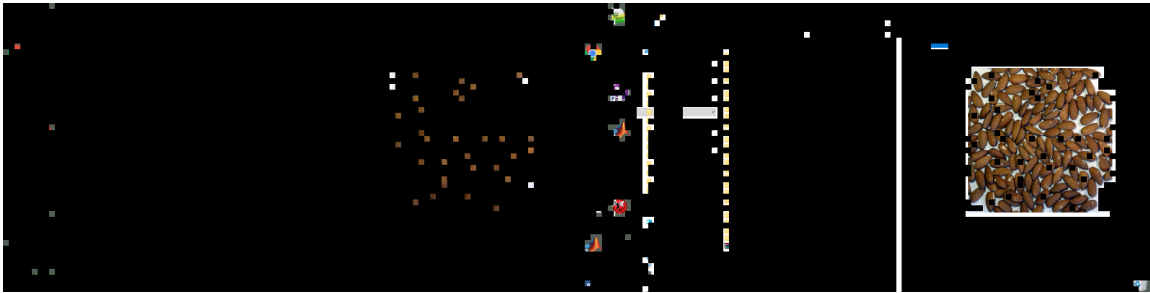


Figure 76: A sample for a set of 40 high resolution compound images



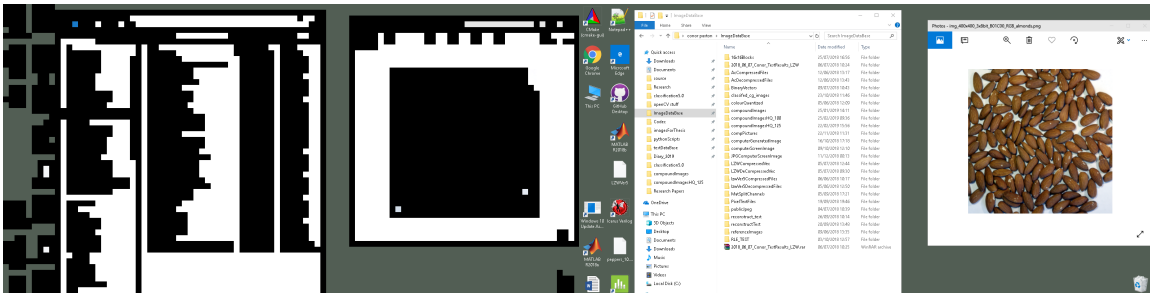
(a) Sparse Blocks

(b) Text Blocks



(c) Fuzzy Blocks

(d) Picture Blocks



(e) Smooth Blocks

(f) Full Image

Figure 77: Decomposition of compound image into Sparse, Text, Fuzzy and Picture blocks using classification algorithm and the recombined image

The results of the classification algorithm performed on figure 76 can be observed in figure 77. The five images in figure 77 show the compound image decomposed into five categories, sparse, text, fuzzy, picture and smooth.

By visual observation, it can be seen that the classification algorithm can decompose a compound image into the five data types with a high degree of accuracy. 100 percent of blocks have been classified. All smooth blocks have been accurately classified, as well as all text and fuzzy blocks. However, there are a number of sparse blocks that have been misclassified as picture blocks that can be observed in figure 77d.

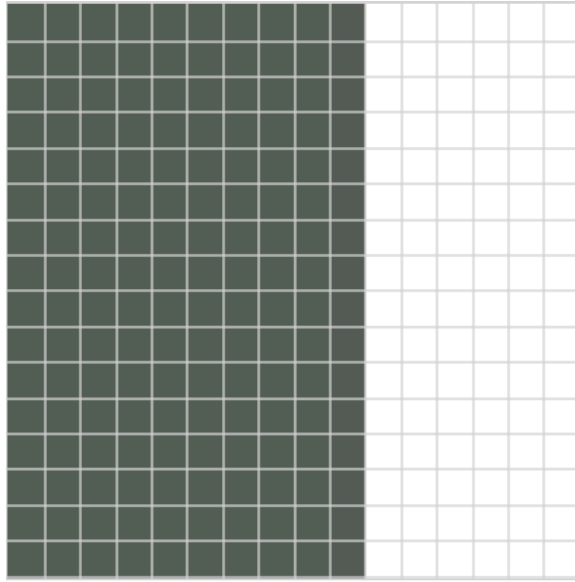


Figure 78: A sparse block that has been misclassified as a picture block from figure 77

Figure 78 shows one of the blocks that should be classified as sparse, but has been classified as picture. Visual inspection of the block shows that there are two areas that are uniform in colour, with distinct structure, with a definitive separation in colour. Table 35 shows the Y channel values of the block after colour space transformation. It confirms that there are only two unique values in the luminance channel for this block.

89	89	89	89	89	89	89	89	89	88	255	255	255	255	255	255
89	89	89	89	89	89	89	89	89	88	255	255	255	255	255	255
89	89	89	89	89	89	89	89	89	88	255	255	255	255	255	255
89	89	89	89	89	89	89	89	89	88	255	255	255	255	255	255
89	89	89	89	89	89	89	89	89	88	255	255	255	255	255	255
89	89	89	89	89	89	89	89	89	88	255	255	255	255	255	255
89	89	89	89	89	89	89	89	89	88	255	255	255	255	255	255
89	89	89	89	89	89	89	89	89	88	255	255	255	255	255	255
89	89	89	89	89	89	89	89	89	88	255	255	255	255	255	255
89	89	89	89	89	89	89	89	89	88	255	255	255	255	255	255
89	89	89	89	89	89	89	89	89	88	255	255	255	255	255	255
89	89	89	89	89	89	89	89	89	88	255	255	255	255	255	255
89	89	89	89	89	89	89	89	89	88	255	255	255	255	255	255
89	89	89	89	89	89	89	89	89	88	255	255	255	255	255	255
89	89	89	89	89	89	89	89	89	88	255	255	255	255	255	255
89	89	89	89	89	89	89	89	89	88	255	255	255	255	255	255

Table 35: Y Channel Values of sparse block miss classified as picture block

An observation that is of fundamental importance is that there are an even number of columns containing the unique values. There are ten columns containing the luminance value 89, while there are six columns containing the value 255.

89	89	89	89	88	255	255	255
89	89	89	89	88	255	255	255
89	89	89	89	88	255	255	255
89	89	89	89	88	255	255	255
89	89	89	89	88	255	255	255
89	89	89	89	88	255	255	255
89	89	89	89	88	255	255	255
89	89	89	89	88	255	255	255

(a) Sub Band LL

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(b) Sub Band HL

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(c) Sub Band LH

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(d) Sub Band HH

Table 36: Forward Discrete Haar Wavelet Transform decomposition into four sub-bands LL, HL, LH, HH

Figure 36 shows the forward 2D Haar wavelet transform decomposition of figure 78 into the respective sub bands LL, HL, LH and HH. It can be observed that all values in the HL, LH and HH sub bands are zero. The Haar transform calculates the mean and difference of consecutive pairwise samples, if two consecutive values are the same, the mean is equal to the value of each sample and the difference will always be zero. If the Haar transform is performed on a block of samples that have an even uniform structure, like that in figure 78, the resulting sub bands HL, LH and HH will not contain any variance, as such the standard deviation of the particular sub bands will also be zero. To cater for this type of block more logic has to be included in the classification algorithm.

```

1   additional condition sparse = [ count < 10
2                                   && st_dev_HL == 0
3                                   && st_dev_LH == 0
4                                   && st_dev_HL == 0 ]

```

The classification algorithm in figure 70 is modified to include the condition above. The compound image in figure 76 is processed with the final iteration of the classification algorithm used in this thesis and the results are shown in figure 79. It can be observed that the classification algorithm is now highly accurate at decomposing a compound image into smooth, sparse, text and fuzzy blocks. The c++ code implementation of the functions for the classification algorithm can be seen in the code section of this thesis in listing 22.

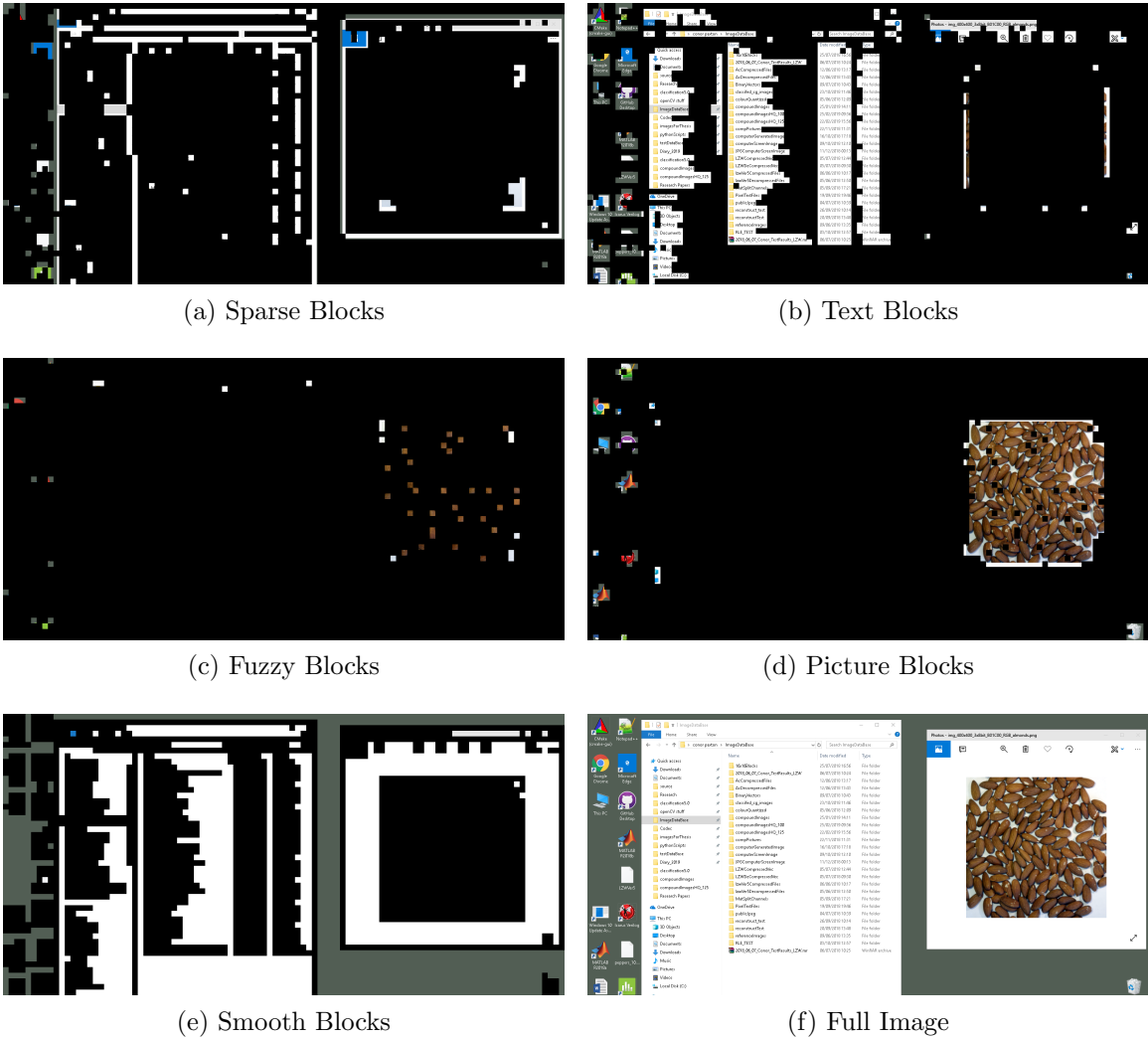


Figure 79: Decomposition of compound image into Sparse, Text, Fuzzy and Picture blocks using modified classification algorithm and the recombined image

Figure 79 shows the results of classification with modified logic. It can be observed that the classification errors have now been greatly reduced.

9.1.1 Comparing Presented Classification Algorithm to the Work presented by Wu [48]

As stated, the classification algorithm presented in this thesis is inspired by the work presented by Wu [48], with fundamental differences. The key differences in the classification algorithm presented so far in this thesis are as follows:

- The discrete wavelet transform is performed on the luminance channel in YCbCr colour space instead of the green channel in RGB colour space. This is a fundamental change which greatly reduces the error in classification. As an example, given a 16x16 block containing pure blue text on a black background, if a discrete wavelet transform is performed on the green channel, each sub band will contain no deviation as all the green channel values will be zero. This case is shown in 133 in the appendix of this thesis.
- Because the discrete Haar wavelet transform is a pairwise average and difference calculation, if it is performed on a block that has multiple distinct uniform colour regions with an even amount of columns, there will be no difference in the sub bands so the standard deviation calculation will return zero. This case has been found in blocks that should be classified as sparse. If a block has more than one unique colour and a standard deviation of zero in the sub bands HL, LH and HH, it will be a sparse block.
- The LL sub band contains an approximation of the original block. If the standard deviation is low, in the LL sub band, it is a strong indication that the colour distribution is uniform, thus coupled with colour counting, can be used to discern between fuzzy and sparse block, which may contain similar attributes in the HL, LH and HH sub bands.
- To improve speed performance in classification, the work presented by Wu [48] used sub sampling of a block in RGB colour space, to reduce the size of a 16x16 block to an 8x8 block before performing a discrete Haar wavelet transform. The classification algorithm presented here does not use sub sampling, but implements the discrete Haar wavelet transform using integer arithmetic, multiplication and division to improve on performance.

A test is set to compare the classification algorithm described so far in this thesis to the work presented by Wu. Five compound images are selected to compare the two algorithms classification accuracy and performance in speed. Four of the images are from a set of forty high resolution compound images captured for this thesis and one image is from the work presented by Wang *et al* [44] on their work for consideration of screen content image for testing the HEVC algorithm.

A sample of two of the images used in this test which have been decomposed into smooth, sparse, fuzzy, text and picture blocks can be seen in the appendix of this thesis in figures 123 - 133.

The algorithm presented in this thesis is labelled algorithm A, while the work presented by Wu is labelled algorithm B. It can be observed that the classification Accuracy of algorithm A is greater than the classification by Algorithm B based on the amount of blocks that have been subjectively classified in error. Of significant interest, the error in classification caused by performing the discrete wavelet transform in RGB colour space is prominent in figure 133. It shows a significant number of blocks classified in error as smooth blocks, when they contain text and other computer generated data. This type of error would be detrimental to the overall subject quality of the image after decompression and decoding, as truly smooth blocks contain a single pixel colour, as such only one pixel is chosen to represent a 16x16 block in compression.

The time taken for classification is recorded and can be seen in table 37. It can be seen that classification algorithm A out performs Algorithm B in each case. However, there is a larger variance in time taken by classification A, while classification B is more uniform.

Algorithm A (ms)	Algorithm B (ms)
23.9	42.6
18.8	43.5
24.8	43.3
16.1	45.6
17.7	46.1

Table 37: Classification time comparison of the classification algorithm presented in this thesis (Algorithm A vs the algorithm presented by Wu in [48] Algorithm B

The insight gained by performing the above tests are as follows:

- The classification algorithm presented in this thesis is highly accurate at classification
- Even though there is an improvement in speed, the time taken for classification would be still too long for real time applications, where latency can have a negative impact for the end user's experience.
- The variance in classification time presented in 37 for algorithm A is due to the colour counting process. To ensure a uniform classification time, a threshold must be formulated to limit the time spent on colour counting.

9.1.2 Colour Counting Analysis

A test will be performed on a set of forty compound images prepared for this thesis, to count the amount of unique pixel values that are in each type of block. The purpose of this test is to understand if there is a correlation between the unique pixel count per block and which data type the block will be classified as. Further, to decide upon a threshold value of unique pixels that can be used to distinguish between computer generated data, such as text and natural camera captured data.

Classification Breakdown

Contribution of each data type after classification algorithm on a set of 40 compound images

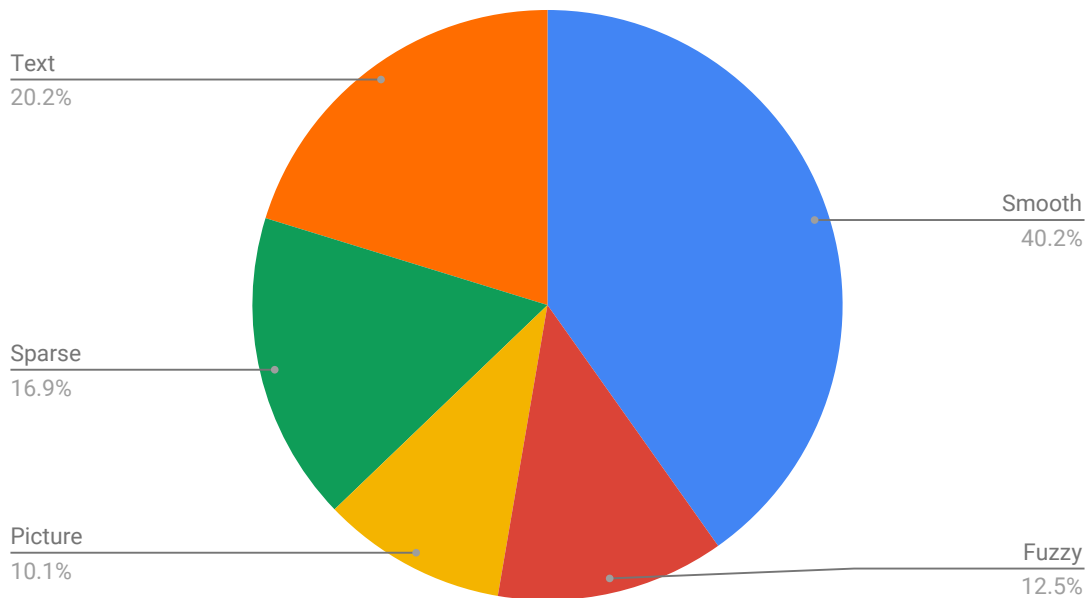


Figure 80: Breakdown of classification Results

Figure 80 Shows the results of running the classification algorithm presented in this thesis on a set of 40 compound images. The average value of each classification type is calculated and the value is shown in the graph. It can be observed that the dominant classification type is smooth, with approximately 40 percent of the distribution. The combination of blocks classified as sparse and text which should be compressed using a lossless compression algorithm is approximately 37 percent, while the combination of blocks that would be potentially compressed using a lossy compression type image, fuzzy and picture blocks, is approximately 23 percent.

Analysis of Pixel Counting on Blocks Classified as Fuzzy

Blocks classified as fuzzy vs unique pixel count

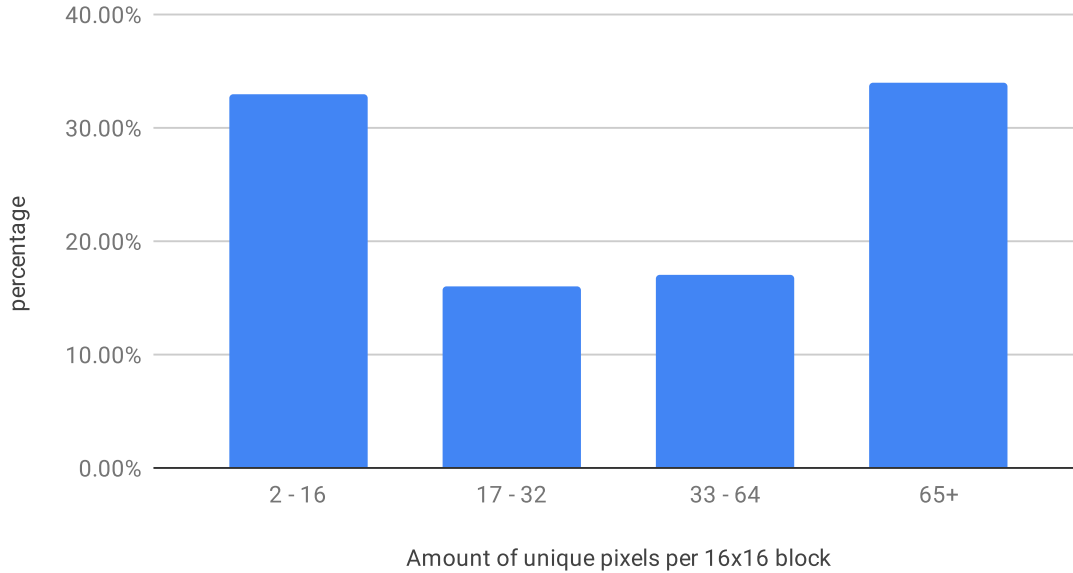


Figure 81: Breakdown of unique pixel count on blocks classified as fuzzy from a set of 40 compound images

Figure 81 shows the distribution of unique pixel count for blocks that have been classified as fuzzy. The four bins of the histogram are blocks containing 2-16, 17-32, 33-64 and 65 plus unique pixel values. The dominant bin is the 65 or more unique pixel values per block, with approximately 37 percent of the total distribution. This result agrees with the hypothesis that blocks classified as fuzzy are typically blocks containing natural camera captured image, which contain many unique colours, but have a uniform luminance values with slight variance.

A significant observation from figure 81 is that the bin representing 2-16 unique pixel values per block is the second highest with approximately 34 percent of the distribution. To further analysis this, an image from the test set is selected that has a significant amount of blocks that have been classified as fuzzy containing from 2-16 unique pixel values. This image can be seen in figure 82.

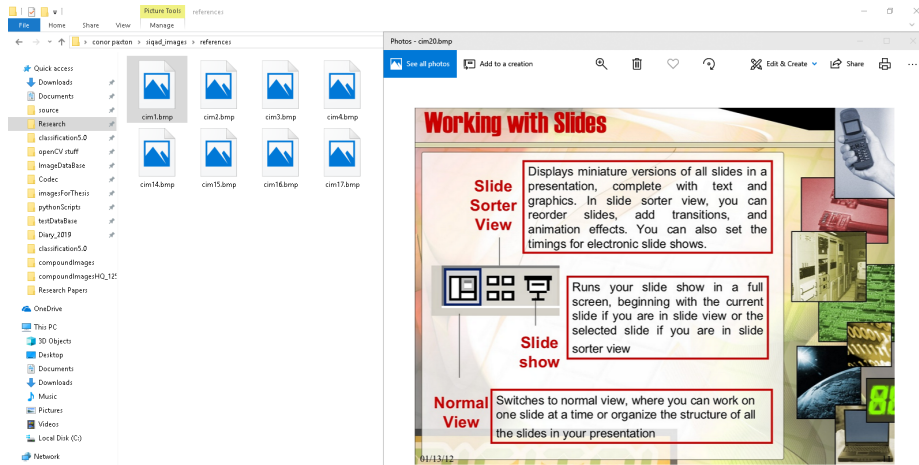


Figure 82: Compound image with blocks classified as fuzzy, with approximately 50 percent having 2-16 unique values

Table 38 shows the distribution of unique pixel values in blocks that have been classified as fuzzy. It can be observed that over 32 percent of blocks classified as fuzzy in figure 82 contain 2 -16 unique pixel values. The fuzzy blocks containing 2-16 pixels from 38 are isolated and shown in figure 83.

Total amount of fuzzy Blocks	403	
Fuzzy blocks with 2-16 unique pixel count	129	32.01%
Fuzzy Blocks with 17-32 unique pixel count	24	5.95%
Fuzzy blocks with 33-64 unique pixel count	96	23.82%
Fuzzy blocks with 65 plus unique pixel count	154	38.28%

Table 38: Distribution of unique pixel values in fuzzy blocks from figure 82



Figure 83: Only blocks classified as fuzzy that contain 2-16 unique pixel values from figure 82

It can be observed from figure 83 that the blocks containing 2-16 unique pixels are not in areas of the image that largely contain computer generated data. A sample of one of the blocks classified as fuzzy containing 2-16 unique pixel values is shown in figure 84.

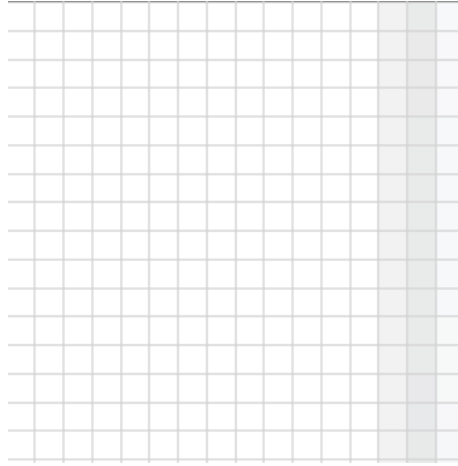


Figure 84: A 16x16 block classified as fuzzy from figure 82 containing less than 16 unique pixel values

It can be observed from figure 84 that the block contains defined structure with vertical lines to the right hand side, while the rest of the block seems smooth. This type of data is not data from natural camera captured image and should potentially be classified as either sparse or text.

The reason that this block has been classified as fuzzy is that the luminance value of the block overall is fairly uniform, with the vertical lines structured elements luminance values not varying too much from the background. This type of data is typically caused by shadow type effects generated by the graphical user interface around desktop items and borders to help them stand out.

The potential consequence of the block in figure 84 being classified as fuzzy would be that it will get compressed with a lossy type compression algorithm that could lead to compression artifacts, which would reduce the quality for the end user.

Analysis of Pixel Counting on Blocks Classified as Picture

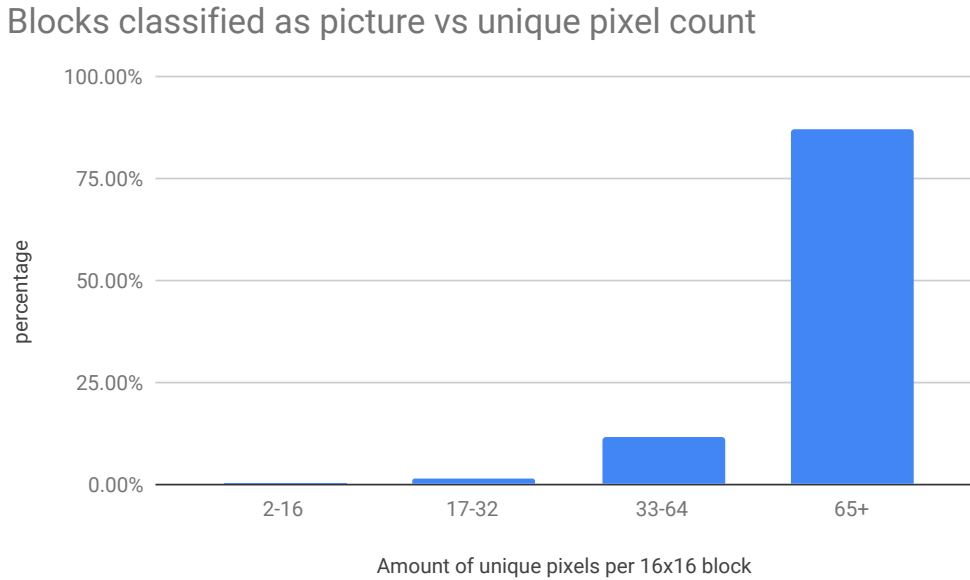


Figure 85: Breakdown of unique pixel count on blocks classified as picture from a set of 40 compound images

Figure 85 shows the distribution of unique pixel count for blocks that have been classified as picture from a set of 40 compound images. It can be observed from the histogram that the majority of blocks that have been classified as picture contain 65 or more unique pixel values. This agrees with the hypothesis that blocks containing natural camera captured image contain a larger amount of unique pixel values.

The large number of unique pixel values in blocks classified as picture suggests that neighbouring pixels in a block are highly correlated. Because there is high correlation, there is typically redundant information contained in the data. A lossy compression algorithm such as one based on a discrete cosine transform would be suited to compressing the data found in blocks classified as picture efficiently

Analysis of Pixel Counting on Blocks Classified as Sparse

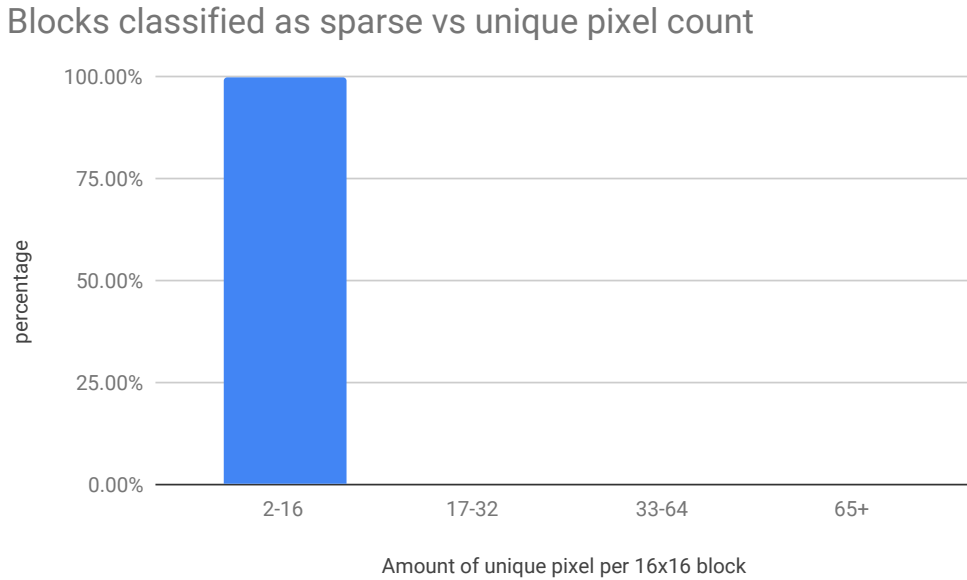


Figure 86: Breakdown of unique pixel count on blocks classified as sparse from a set of 40 compound images

Figure 86 shows the distribution of unique pixel count for blocks that have been classified as sparse from a set of 40 compound images. It can be observed that all blocks that have been classified as sparse contain less than 32 unique pixel values. This agrees with the hypothesis that blocks classified as sparse should be low in complexity and have a limited amount of unique pixel values.

Due to the low unique pixel value count in sparse blocks, it suggests that there are uniform and highly structured patterns in the data contained in each block. A lossless type of compression algorithm can be tuned to compress blocks of this nature efficiently and quickly.

Analysis of Pixel Counting on Blocks Classified as Text

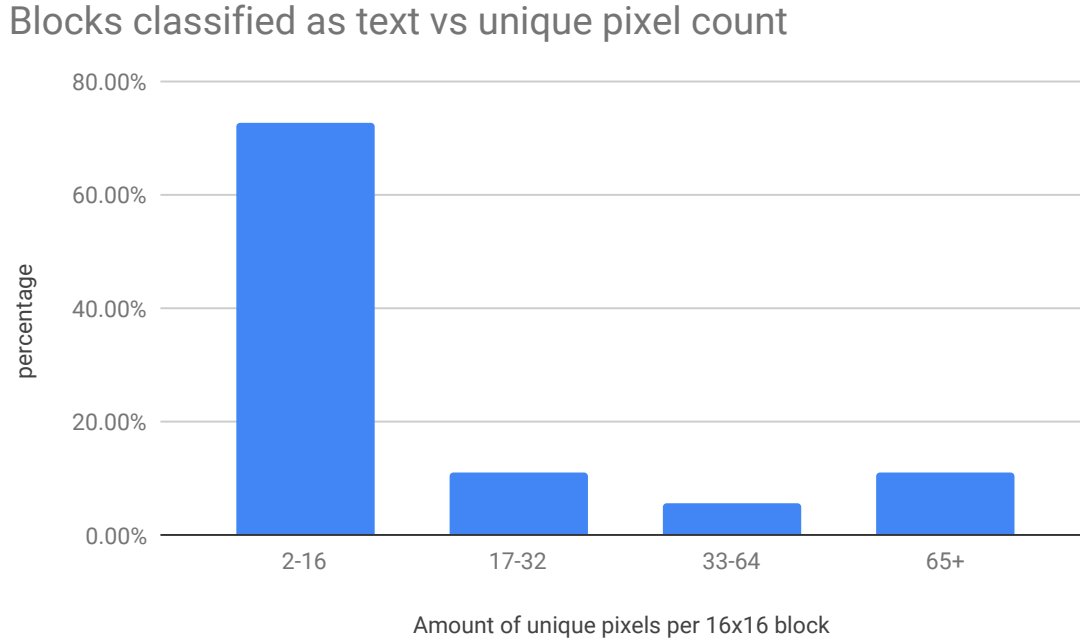


Figure 87: Breakdown of unique pixel count on blocks classified as text from a set of 40 compound images

Figure 87 shows the distribution of unique pixel count for blocks that have been classified as text from a set of 40 compound images.

It can be observed from the histogram that the dominant bin is blocks containing 2-16 pixels, with approximately 73 percent of the total distribution. a blocks classified as text containing 17-32 unique pixel values have approximately 11 percent of the distribution.

An important observation from 85 is that the combination of blocks greater than 32 unique pixel values account for approximately 16 percent of the distribution. An image is selected from the test set that contains a significant amount of blocks that have been classified as text with unique pixel count greater than 32. The image can be seen in figure 88

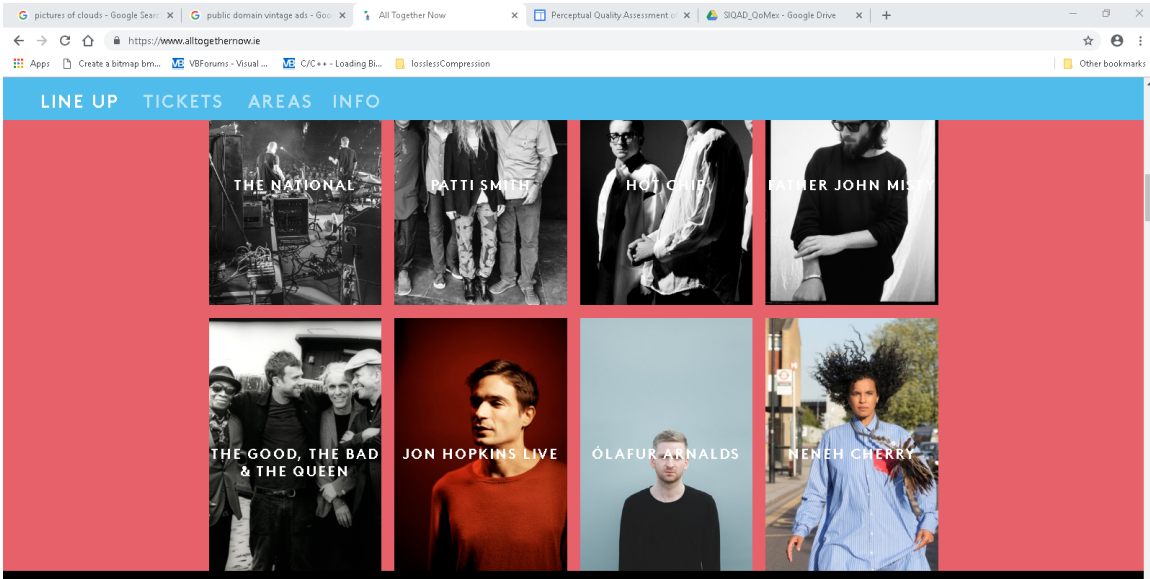


Figure 88: Sample from compound image set with over 50 percent of blocks classified as text containing greater than 16 unique pixel values

Total amount of text Blocks	596	
Text blocks with 2-16 unique pixel count	224	40.93%
Text Blocks with 17-32 unique pixel count	181	30.37%
Text blocks with 33-64 unique pixel count	13	2.2%
Text blocks with 65 plus unique pixel count	158	26.51%

Table 39: Distribution of unique pixel values in text blocks from figure 88

Table 39 shows the distribution of unique pixel values in blocks that have been classified as text from figure 88. It can be observed that the combination of blocks greater than 32 unique pixel values make up approximately 29 percent of the total distribution. The text blocks containing greater than 32 unique pixels from 38 are isolated and shown in figure 89.

It can be observed from figure 89 That the majority of blocks that have been classified as text that contain greater than 32 unique pixel values are in areas of figure 88 that do not contain text. From observation only 21 of the 171 blocks with greater than 32 unique pixel values actually contain text

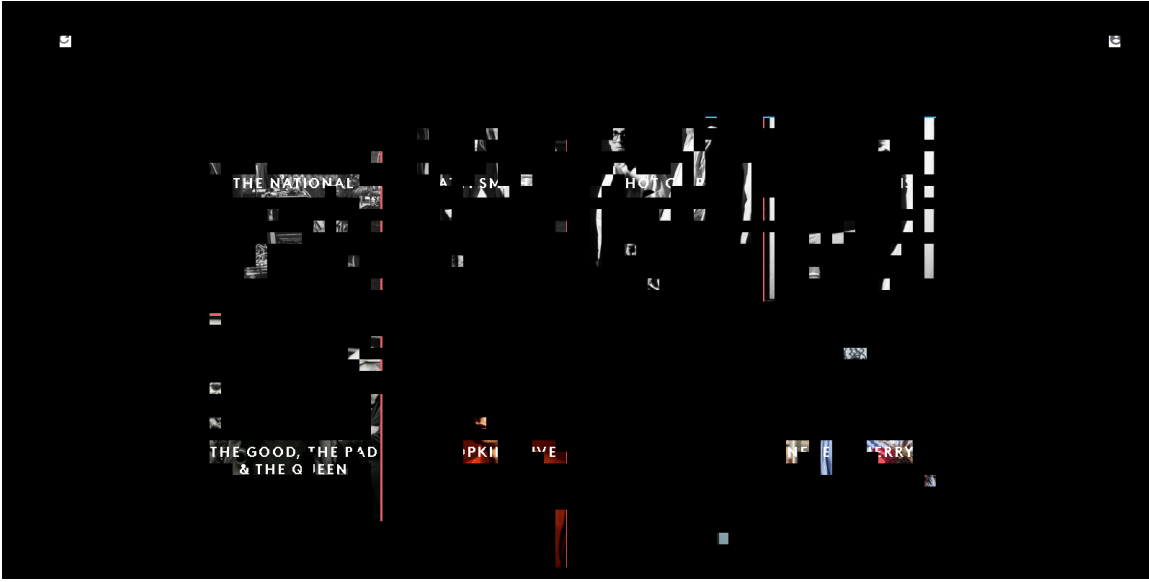


Figure 89: blocks classified as text containing greater than 32 unique pixel values from figure 88

A sample of two of the blocks from 88 that have been classified as text that have greater than 32 unique pixel values have been extracted and can be seen in figure

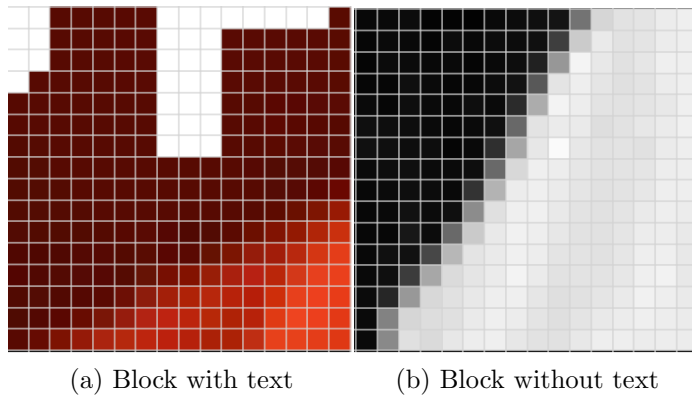


Figure 90: Sample Blocks from figure 88 that have been classified as text with greater than 32 unique pixel values

It can be observed from figure 90a, That there is text data, while the background contains natural image, While figure 90b does not contain text, but it does have the characteristics of text, such as sharp transients in the diagonal direction.

Both blocks contain very defined structure which should be preserved during the compression process, however due to both blocks containing a high unique pixel count, compressing types of blocks similar to this using a lossless compression algorithm would incur a significant penalty in compression performance. To achieve better

compression performance, a lossy transform based compression algorithm could be used to de-correlate the information in the pixels relating to natural image. However, by using a lossy compression algorithm based on the discrete cosine transform could lead to visible compression artifacts and distortion to the structure contained in both blocks. A lossy type compression that can successfully preserve the structure of the data within the block, while removing redundancy in the colour information in the block

Analysis of Unique Pixel Value Distribution on the Compound Image Test Set

Average distribution of unique pixel values per 16x16 pixel block on a set of 40 compound images

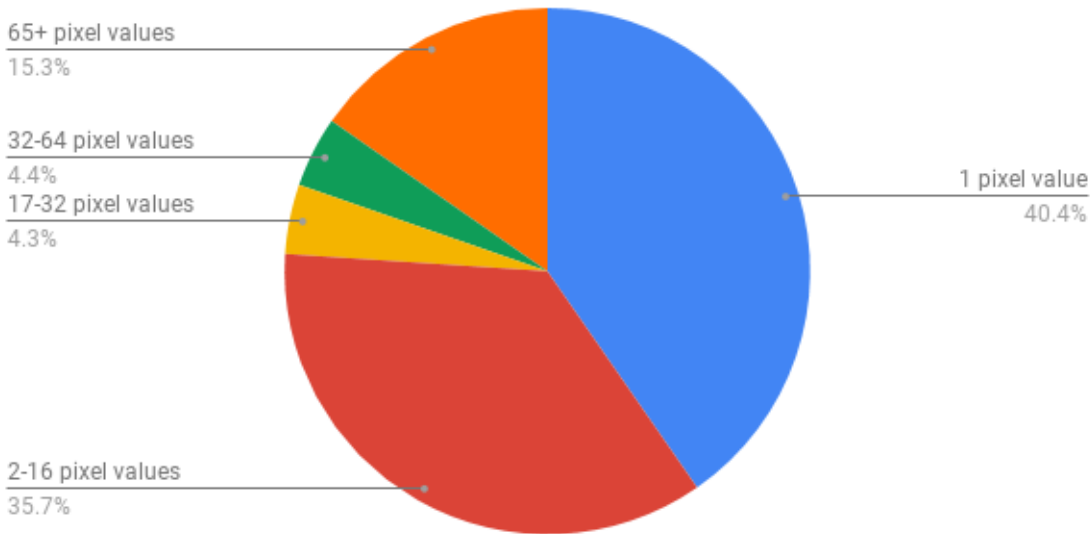


Figure 91: Showing the average value of the distribution of unique pixel values per 16x16 block of pixels in the compound image test set

Figure 91 shows the breakdown of unique pixel values per block on the set of compound images used in testing for this thesis. It can be observed that the dominant section is blocks containing one single pixel value with approximately 40 percent of the distribution. while the combination of blocks with up to 32 unique pixel values accounts for over 80 percent of the distribution. It can be observed that the amount of blocks contain greater than 32 unique pixel values is significantly smaller, accounting for approximately 20 percent of the distribution.

Insight Gained from Colour Count testing

By performing colour counting on the compound image test set, significant insight has been gained about the relationship between the unique pixel count of a 16x16 block and what type of data that block contains. The key points found are:

- The majority of blocks classified as fuzzy have a pixel count greater than 16 unique pixels. However, a significant number of blocks that have been classified as fuzzy contain a unique pixel count between the range 2-16 unique pixel values. On visual inspection, the majority of these blocks do not contain data from natural camera captured image, but contain computer generated data.

The problem presented with blocks that should be classified as sparse but get classified as fuzzy is that, typically a lossy type compression would be used to compress data from camera captured image, while a lossless type of compression should be used for computer generated data.

- Blocks classified as picture contain the highest unique pixel count, with the majority of blocks having greater than 64 unique pixel values
- Blocks that have been classified as sparse have a low pixel count
- The majority of blocks that have been classified as text have a pixel count less than 32 unique pixels. However, there is a significant number of blocks that have been classified as text that have a unique pixel count greater than 32 unique pixel values. on visual inspection, the blocks that have been classified as text that contain greater than 32 unique pixel values do not actually contain text data, but have the characteristics of text, such as sharp transitions in horizontal, vertical and diagonal directions. These type of blocks have been observed to be a type of hybrid block that contain both computer generated data and natural camera captured data. The problem presented with this type of block is that, typically a lossless type of compression should be used on data containing text, to preserve the structure in the block and to ensure that the decompressed image is free from artifacts around the text. The high pixel count in these type of blocks would significantly impact the performance of the compression.
- On average, approximately 80 percent of blocks contain less than 32 unique pixel values from the compound image test set.

9.1.3 Conclusion on Compound Image Classification Using Discrete Wavelet Transform and Colour Counting Analysis

The tests presented in this section have used a combination of discrete wavelet transform, statistical analysis and colour counting to gain insight about the characteristics of data that is found in a compound image. The classification algorithm that has been developed so far is the first stage in creating a codec suitable for compound image compression. The role of the classification algorithm is to decide what type of compression to use on a given block, based on its classification.

Initially, working on the premise that there are five distinct classification types such as smooth, sparse, fuzzy, picture and text, the objective is to compress computer generated data with a lossless compression algorithm as to preserve the quality of text and graphical content and to compress non computer generated data with a lossy compression algorithm to reduce the redundancy in colour information that may not have an impact for the end user.

From initial testing, it was found that the classification algorithm based on discrete Haar wavelet transform with the aid of colour counting is extremely accurate at classifying each classification data type on images that contained a majority of a given data type, such as sparse, fuzzy, text and picture.

Important findings were uncovered when using the discrete Haar wavelet transform for classification on compound images, such as the edge case where a block containing structural information has an even number of row or columns, leading it to be misclassified and logic has been implemented to mitigate this phenomena.

Of significant importance, is the results of colour counting analysis, in particular on blocks that have been classified as fuzzy and text. The observation was made that a significant number of fuzzy blocks that have a low pixel count in fact contain computer generated data. This presents a challenge when a compression algorithm is to be selected for such a block. If the block is compressed with a lossy type compression algorithm, there may be compression artefacts visible in the decompressed image.

At the other end, the observation was made about a significant number of blocks classified as text have a unique pixel count greater than 32 unique pixel values. On visual observation, the majority of these blocks did not actually contain text, but have properties similar to text in the sub bands after discrete wavelet transform, such as sharp transitions in the vertical, horizontal and diagonal directions. The challenge when choosing a compression algorithm suitable for compressing these types of blocks are, if a lossless compression algorithm is chosen, the high pixel count will have a negative impact on the compression performance. If a lossy type of compression algorithm is chosen, it could have a negative impact on the final image quality by imparting compression artefacts around structural information.

One of the key findings from performing colour counting analysis is that, on aver-

age, approximately 80 percent of blocks in a compound image from the test set have a unique pixel less than 32 unique values. This information is fundamental to the development of compression algorithms to be chosen for the compound compression algorithm. From the findings, It can be seen that there is a high correlation between blocks with low pixel count and computer generated data such as text and graphical content. As the objective is to use a lossless type of compression for computer generated data, A majority of the focus should be on implementing efficient lossless, or approximately lossless compression algorithms for the final compound compression algorithm. Further, as there are a significant number of blocks that have similar statistical properties as text, but do not contain text, compression algorithms that support retaining structural information, while reducing redundancy in colour information should be implemented.

So far, the development of the classification algorithm has not put a great deal of emphasis on optimisation for computational complexity, except for implementing the discrete Haar wavelet transform using integer arithmetic, instead of floating point arithmetic. As the discrete wavelet transform is a mathematical transform, it is far more computationally expensive than simply counting the amount of unique pixel values per block. It can be seen from the above tests that a significant amount of information can be found in the analysis of colour counting, with the value of 32 unique pixel values per block being a sufficient indicator on whether a block contains computer generated data or natural camera captured data. These findings will be exploited in the final iteration of the codec developed in this thesis.

The next section of this thesis will be on the implementation and testing of both lossless and lossy compression algorithms for the specified classified data types.

9.2 Lossless Compression Testing

In this section, lossless compression algorithms will be tested on the three types of block class that contain computer generated data, which are smooth, sparse and text. Industry standard compression algorithms such as the LZW [46] and Deflate [7] will be implemented and compared to a novel compression algorithm proposed for this thesis, called *Differential Index Map Coding*. The goal is to find the most efficient compression algorithm for each type of classified block to use in the framework for the compound compression algorithm.

9.2.1 Smooth Block Compression

Compression of blocks that have been classified as smooth is the most straight forward type of compression that will be implemented in the final compound compression algorithm presented in this thesis. The defining attribute of a smooth block is a single 3-byte pixel value per 16x16 block. From the analysis on the compound image test set, smooth blocks are among the most dominant type of block found within a compound image and luckily, can be compressed very efficiently with respect to compression performance and computational complexity.

Smooth blocks have a theoretical compression ratio of 256:1, as one pixel can be used to describe a 16x16 block, if all values are the same. Smooth blocks can be found in large areas of the screen often surrounded by other smooth blocks. However, not all smooth blocks are contiguous, as a compound image may have different regions of the screen which contain smooth blocks. Because of this, the coordinates of smooth blocks will have to be recorded to with the data, so that the smooth block data gets repopulated in the decompressed image. By taking the x and y coordinates of where a block is with respect to the compound image and a single 3 byte pixel, the compression ratio for smooth block compression is 154:1. This is extremely efficient compression, however further processing can be used to improve compression performance.

Smooth blocks tend to be grouped together, such as a single colour background of an open text editor on a computer screen. A significant number of neighbouring smooth blocks will have the same value. The redundancy in same valued neighbouring smooth blocks can be exploited using a simple run length encoding, where the run value is a 3 byte pixel value and the length is the amount of consecutive blocks that are the same. As well as run length encoding, the coordinate data of the smooth blocks can also be exploited. As the scanning order is from left to right, top to bottom, the smooth blocks coordinates are in ascending formation. By recording the consecutive difference between coordinate values, a simple delta encoding can be used to improve compression performance.

Figure 92 shows the results of applying smooth block compression on blocks classified as smooth in a set of 40 compound images. It can be observed that by applying run

length encoding and delta encoding the block coordinates, extremely high compression can be achieved. As such, this is an optimal compression method for blocks classified as smooth in a compound image

Compression of data classified as smooth vs compression ratio

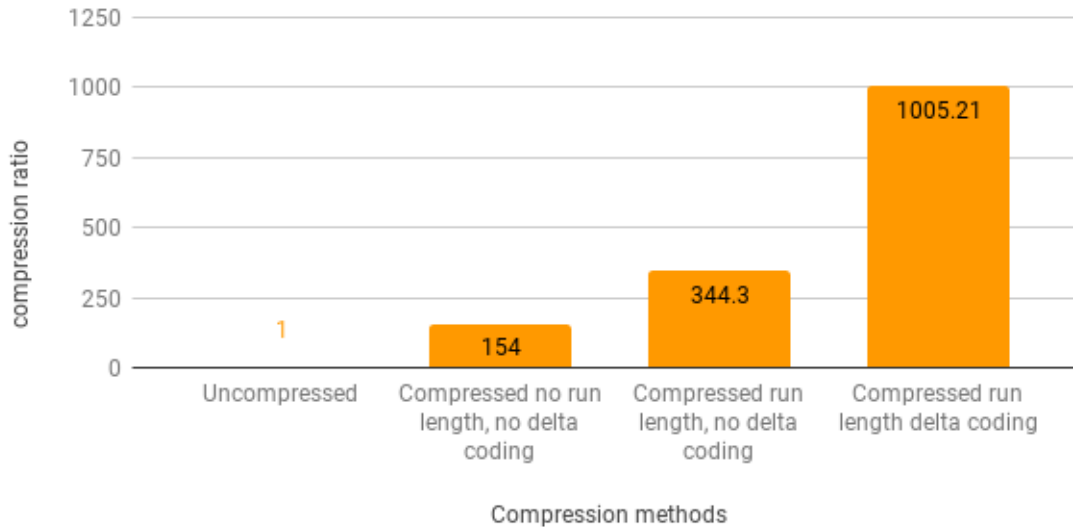


Figure 92: Compressing data classified as smooth from a test set of compound images with multiple compression methods

To illustrate the effectiveness of smooth block compression on reducing the size of smooth data in a compound image, table 40 shows the average uncompressed size of smooth data from the test set of 40 compound images, along with the above compression techniques. It can be observed that on average, approximately 1.54MB has been reduced to little over 1.6KB, which is extremely efficient.

Data Compressed with Smooth Compression	
Compression Method	Average Size (bytes)
Uncompressed	1547268.05
Compressed no run length, no delta coding	10047.19
Compressed run length, no delta coding	4487.75
Compressed run length delta coding	1637.60

Table 40: Table comparing the file size of uncompressed data classified as smooth vs compressed data file size from the compound image test set

9.2.2 Lempel-Ziv Welch Compression

An implementation of a Lempel-Ziv Welch compression algorithm will be used to losslessly compress blocks that have been classified as Sparse and Text from a test set of 40 compound images.

The Lempel-Ziv Welch compression algorithm is a dictionary based coding algorithm, as such the blocks of pixels to be encoded must first be serialized. Two types of serialization are used in testing: *Packed* and *Channel*. The packed method stores each channel value of a 3 byte pixel in an interleaved mode, where each value of a given pixel are ordered directly after each other. In the Channel method, all the channel values of a block are stored sequentially, followed by the next channel. To illustrate this, consider four 3-byte pixels: p1, p2, p3 and p4, where each pixel has a data member for blue =b, green=g and red=r. Table 41 illustrates Packed serialization and table 42 illustrates Channel serialization.

p1.b	p1.g	p1.r	p2.b	p2.g	p2.r	p3.b	p3.g	p3.r	p4.b	p4.g	p4.r
------	------	------	------	------	------	------	------	------	------	------	------

Table 41: Packed pixel serialization format

p1.b	p2.b	p3.b	p4.b	p1.g	p2.g	p3.g	p4.g	p1.r	p2.r	p3.r	p4.r
------	------	------	------	------	------	------	------	------	------	------	------

Table 42: Channel pixel serialization format

Listing 9: Serializing Interface Psuedo Code

```

1  SerializeBlocks(src_Image){
2      /*max_row = amount of 16x16 blocks in a row of src_Image*/
3      max_row = src_Image.rows /16;
4      /*max_col = amount of 16x16 blocks in a column of src_Image*/
5      max_col = src_Image.columns/16;
6      serializeType; //0=Packed, 1 = Channel
7      serialData[]; // array to hold serialized Sparse or Text data
8
9      blockChoice; //0 = Sparse, 1 = Text
10
11     for(i =0; i < max_row; i++) {
12         for(j=0; j< max_col; j++) {
13
14             blockType=classifyBlock(src_Image, i j); //block at coordinate i,j
15
16             if(blockType= blockChoice)
17                 serializeBlock(src_Image,i,j,&serialData)
18         }
19     }
20 }
```

Listing 9 shows the psuedo code for iterating through an image and serializing each block that has been classified as either sparse or text, using the classification algorithm presented in this thesis. Once the data has been serialized, it can be processed with A Lempel-Ziv Welch encoder.

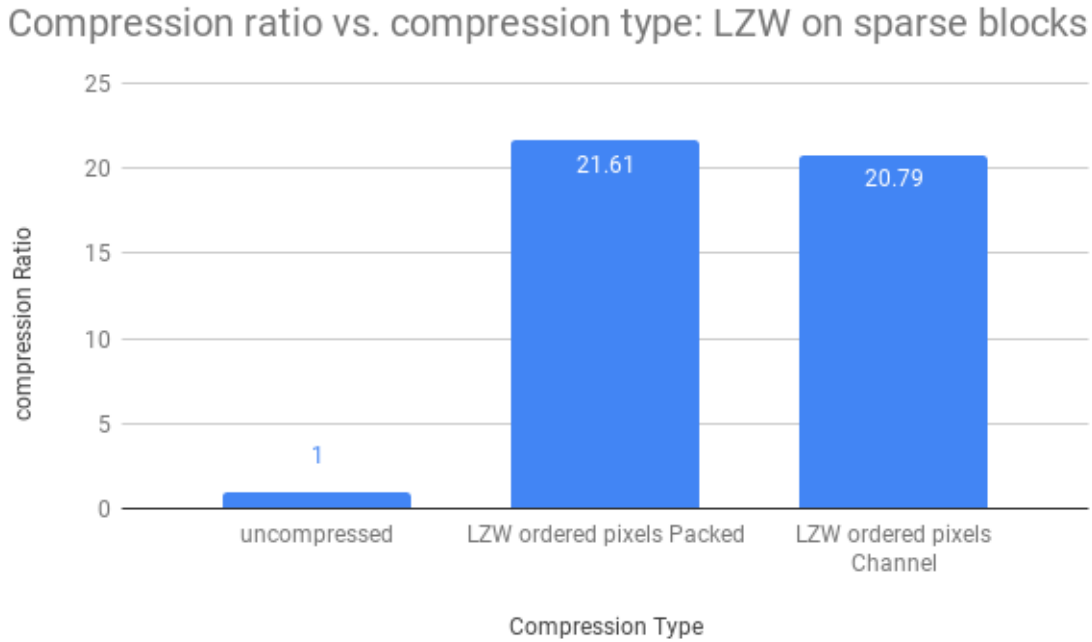


Figure 93: Average compression ratio comparison of Lempel Ziv Welch compression on blocks classified as sparse, with pixel ordering

Figure 93 illustrates the average compression ratio achieved using Lempel Ziv Welch compression on blocks classified as sparse from a set of 40 compound images, using two different types of serialization. It can be observed that Packed serialization performs better by approximately 4 percent.

Lempel-Ziv Welch compression can be seen to perform efficiently on blocks classified as sparse, which is shown in table 43 which describes the average compressed file size with respect to the average uncompressed size of raw data.

Compression Method	Average Size (kB)
uncompressed	696.20
LZW ordered pixels Packed	35.86
LZW ordered pixels Channel	35.17

Table 43: Table comparing the average file size of uncompressed data classified as sparse vs compressed data file size from the compound image test set

Figure 94 illustrates the average compression ratio achieved using Lempel Ziv Welch compression on blocks classified as text from a set of 40 compound images, using two different types of serialization. Again, it can be observed that Packed serialization performs better than Channel serialization by approximately 10 percent.

Compression ratio vs. compression type: LZW on text blocks

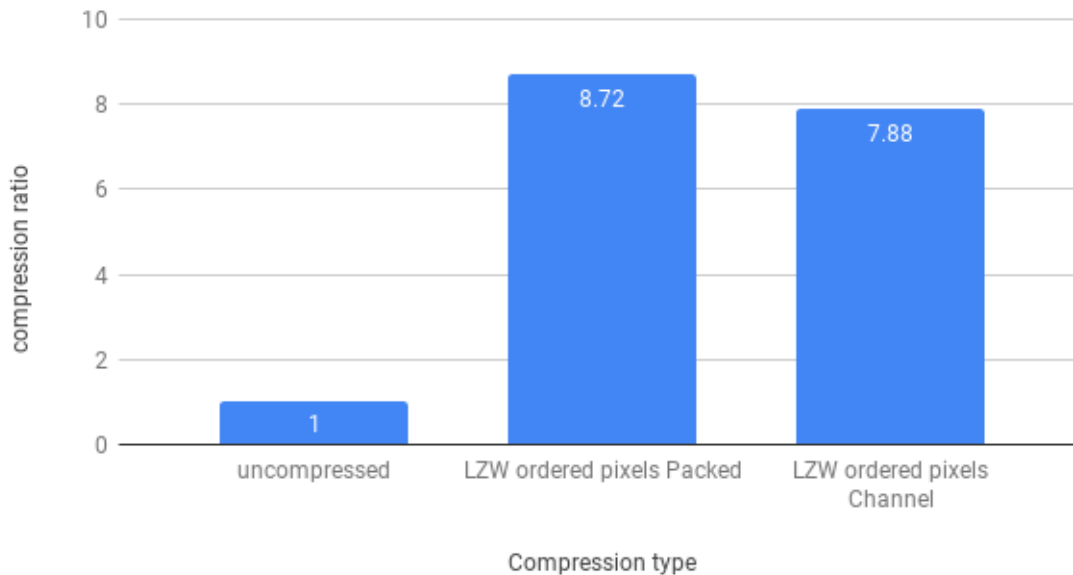


Figure 94: Average compression ratio comparison of Lempel Ziv Welch compression on blocks classified as text, with pixel ordering

It can be observed from figure 94 and table 44 that compressing compound image data classified as text with Lempel-Ziv Welch compression is not as efficient as compressing data classified as sparse. This is due to the complex structural information in the pixel data for text. However, Lempel-Ziv Welch compression for data classified as text is still good compression considering that it is a lossless compression algorithm.

Average Size of Text Blocks Compressed with LZW Compression	
Compression Method	Average Size (kB)
uncompressed	827.47
LZW ordered pixels Packed	117.1
LZW ordered pixels Channel	123.08

Table 44: Table comparing the average file size of uncompressed data classified as sparse vs compressed data file size from the compound image test set

Comments on Lempel-Ziv Welch compression on compound image data classified as Sparse and Text

The Lempel-Ziv Welch compression algorithm can achieve good compression for data in a compound image classified as sparse and adequate compression for data classified as text. However, as the compression algorithm is a dictionary based algorithm, the raw data to be compressed must be acquired first and fully serialized, before processing, which means a second pass through the data. Passing through the image data multiple times will incur the penalty of added latency. Further, Lempel-Ziv Welch encoding is known as a "greedy" algorithm, in that it constantly adds new codewords into the code book until the maximum size of the code book has been reached, which can also introduce latency. Table 45 shows the average time to compress the serialized sparse and text data per image from the compound image test set. The compression time using Lempel Ziv welch encoding may not be suitable for real time applications.

Data type	Averaged Compression time (ms) per image
Sparse	39.47
Text	58.595

Table 45: Average Compression time per image on data classified as Sparse and Text from a set of 40 compound images

9.2.3 Deflate Compression

The Deflate compression algorithm [7] is a lossless compression algorithm that is a combination of LZ 77 [51] and Huffman encoding [16]. It is at the heart of the lossless image compression file format PNG [13], and was first introduced to be an unencumbered by licences, method for compression which is independent of operating system, cpu and character set. It is also the basis algorithm for archival compression format, GZIP.

The implementation of the Deflate algorithm used in this thesis is provided by the permissive free software licenced library, zlib [8].

Similar to Lempel-Ziv Welch encoding, Deflate is a dictionary based coding method, so the two-dimensional image data of each block must be serialized before it can be compressed using Deflate. The same two serialization methods, *Packed* and *Channel* are used in testing and the serialization interface which has been described in listing 9 is used to serialize the data.

The zlib library provides the functions to compress and decompress streams of data to and from files so an interface is needed to use Deflate functionality with vectors of data in running program memory. The interface implementation in c++ can be seen in the code section of this thesis in Listing 26.

Zlib implements different modes to aid in optimisation for either compression or speed. The three modes that effect compression performance are **Z_BEST_COMPRESSION**, **Z_BEST_SPEED** and **Z_DEFAULT_COMPRESSION**.

Z_BEST_COMPRESSION utilizes a longer search buffer in the LZ 77 process and generates dynamic huffman tables based on the frequency of symbols in the data to be compressed.

Z_BEST_SPEED utilizes a shorter search buffer and uses static Huffman tables that are predefined and that have been created from empirical testing.

Z_DEFAULT_COMPRESSION is a middle ground between the other methods. it uses a shorter search buffer and can use a combination of static and dynamically created Huffman tables.

Average Compression Performance vs. Block Type and Mode: Deflate Algorithm

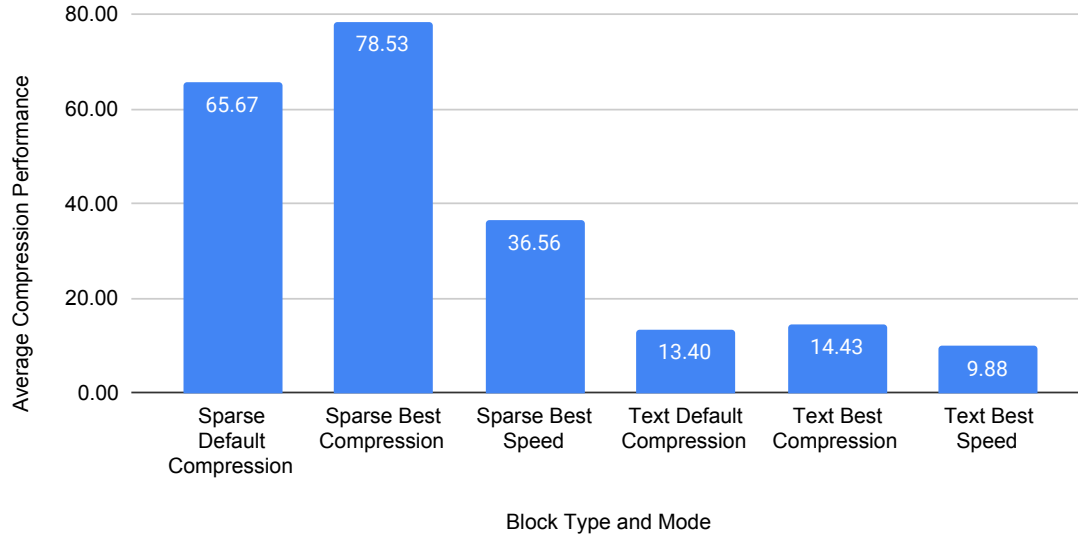


Figure 95: Average compression ratio using Deflate to compress blocks classified as Sparse and Text from a set of 40 compound images

Figure 95 shows the results of performing deflate based compression on blocks that have been classified as Sparse and Text from a set of 40 compound images. Some observation can be made:

- Deflate achieves excellent compression on blocks that have been classified as sparse
- Deflate achieves good compression performance on blocks that have been classified as text
- There is a significant difference in compression performance between optimising for speed or for compression
- Deflate performs better than Lempel-Ziv Welch compression in all three modes

Taking into consideration the performance of each of the modes of the deflate algorithm, the next step is to analyse the performance with respect to speed of processing. Figure 96 shows the average time taken to compress data that has been classified as Sparse and Text from a set of 40 compound images. The results do not include the time taken to classify the data using the classification algorithm and to serialize the pixel block data so it can be processed using the deflate algorithm.

Time (micro seconds) vs. Block Type and Mode: Deflate Algorithm Compression

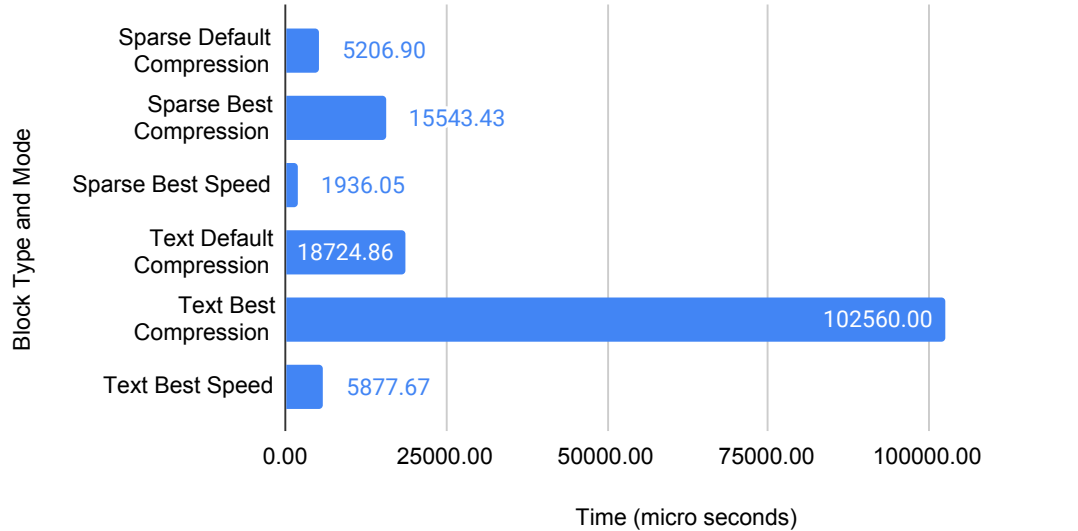


Figure 96: Average time taken to compress using Deflate algorithm with different modes on blocks classified as sparse and text from a set of 40 compound images

It can clearly be observed from figure 96 that using the deflate algorithm with the best compression mode enabled is orders of magnitude slower than the other two modes, In particular, the compression time for text blocks exceeds 100 ms, which would greatly reduce the algorithms ability to be used in real time systems.

When optimized for best speed, the deflate algorithm is extremely efficient, particularly for blocks that have been classified as sparse, however the compression performance with respect to the other modes is significantly less favourable. The default mode is efficient in both speed and in compression performance and the results of this test suggest that It would be a suitable compression algorithm to use for time critical applications, which need both good compression performance, and low through put. As Deflate is a lossless compression algorithm, there will be no error in the recovered data, thus increasing both the objective and subjective image quality of the decoded image.

Time (micro seconds) vs. Block Type and Mode: Deflate Algorithm Decompression

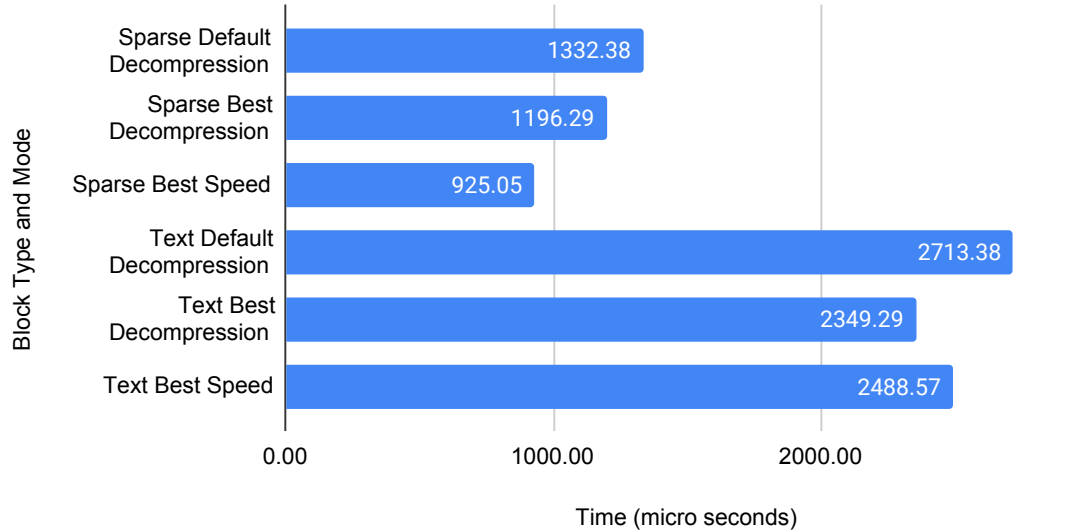


Figure 97: Average time taken to decompress using Deflate algorithm with different modes on blocks classified as sparse and text from a set of 40 compound images

For applications such as virtual desktop infrastructure or thin client systems, possibly of greater importance than compression speed is decompression speed. In a thin client system, the client machine will usually have less processing power than the server that compresses the data. Figure 97 shows the average time it takes to decompress data that has been classified as Sparse and Text from a set of 40 compound images. There are two key points to be observed from the data presented in figure 97: the first point to be made is that the time taken to decode a compressed stream is significantly less than the encoding time. This suggests that Deflate encoding is an asymmetric coding algorithm and may be suitable for both time critical applications and systems where the decoder is running on a machine with low resources.

The second point is that the decoding time is comparatively similar, regardless of which mode was used in encoding. Coincidentally, it can be observed that decoding a stream that has been compressed with the mode for best compression takes slightly less time for the data that has been classified as text. This may be due to the longer symbol matches found during the encoding process.

9.2.4 Conclusions on Deflate Compression on Compound Image Data Classified as Sparse and Text

From the tests performed, it can be seen that the type of data that is compressed using Deflate has a significant impact on the coding performance. The Deflate algorithm has excellent performance in terms of both compression and processing time for data that has been classified as Sparse. It also shows very good performance with respect to compression and speed with data classified as text.

The compression performance has been shown to be lower for data classified as text, compared to data classified as sparse. There are multiple reasons that attribute to this. The complex structure inherent in blocks containing text, will reduce the size of the matching strings generated from the LZ 77 process, which leads to reduced compression performance. Another factor is that a small percentage of blocks that have been classified as text do not contain text, but have some of the characteristics of text, such as sharp transitions in continuous tone images. An example of this is pictures containing buildings, or perhaps text laid over natural image. Although the percentage of this "hybrid" type of block is small, when compressed with a lossless encoding process such as Deflate, it can actually cause bloat in the encoded data, leading to reduced efficiency and larger compressed file size. If one was to incorporate Deflate encoding in a compound compression algorithm, it is vital to ensure that only suitable data for the algorithm is processed by it.

A final key point about integrating the Deflate algorithm into a compound compression algorithm is that all of the data that is to be compressed using it has to be identified and serialized prior to processing. The algorithm relies on having a large buffer of data to search through to find matches. Luckily, Deflate encoding is efficient in terms of speed, however it means that the compression must be done in a second pass through the data which adds time on to the overall process.

9.2.5 Differential Index Map Coding

Differential Index Map Coding is a novel compression algorithm proposed in this research. The algorithm is suitable for compressing highly structured blocks of pixels with a low unique pixel count, which typically contain computer generated text or graphics.

It is a block based coding technique that counts the number of unique 3 byte pixel values in a block and assigns each unique pixel an integer value known as an index. It then generates a map of the index values. The rows of the index map get compared to each other in a unique scanning pattern, where only the difference and the coordinates of the difference are recorded.

The concept of using an index map to represent highly structured data was first

presented by by *Ding et al*[23], for their *Block Fast Compression* algorithm. However, their approach is to use the index map in a lossy context, where the unique pixel values of a block are quantized into 4 dominant colours from which the index map is then constructed from.

The Differential Index Map Coding algorithm assigns an index value to each unique 3 byte pixel in a block of 16x16 pixels and generates an index map. The range of index values are limited. The limit has been derived from the section on colour counting analysis in 9.1.2, which shows that over 90% of all blocks that contain computer generated data from the test set of compound images have a unique pixel count less than 32. This low unique pixel count is fundamental to the structure of the Differential Index Map Coding algorithm framework.

To illustrate the process of Differential Index Map Coding, a 16x16 block of 3 byte pixels is extracted from an image and is shown in figure 98. The block of pixels contains computer generated data, which is evident from the highly structured content, containing sharp transitions and repetitive patterns.

The first stage is to count the unique 3 byte pixels in the block and store them in an array. The location in the array that the unique pixel value is stored is its given index value. A graphical representation of the index table can be seen in fig 99.

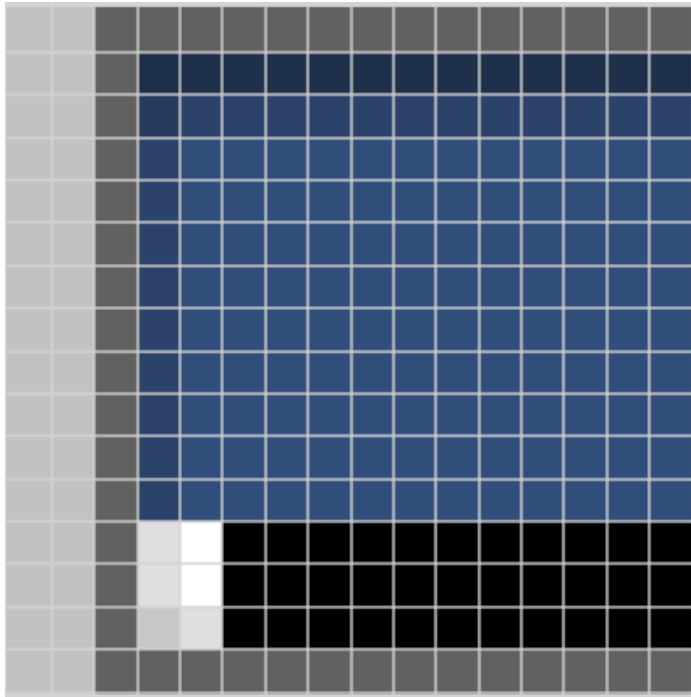


Figure 98: Highly structured block containing computer generated data, with 11 unique 3 byte pixel values












Index Table											
Index	0	1	2	3	4	5	6	7	8	9	10
Colour											
Pixel Value	194	97	29	31	38	43	49	223	255	0	199
Pixel Value	194	97	46	48	60	67	77	223	255	0	199
Pixel Value	194	97	72	75	94	106	121	223	255	0	199

Figure 99: Index table of figure 98 showing pixel values

The next stage is to generate the index map from the values stored in the index table. The process is to iterate through the block and map the current pixel with its given index from the index table. The index map generated from figure 98 can be seen in table 46.

row	Index Map														
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	2	3	3	3	3	3	3	3	3	3	3	3
2	0	0	1	4	5	5	5	5	5	5	5	5	5	5	5
3	0	0	1	5	6	6	6	6	6	6	6	6	6	6	6
4	0	0	1	5	6	6	6	6	6	6	6	6	6	6	6
5	0	0	1	5	6	6	6	6	6	6	6	6	6	6	6
6	0	0	1	5	6	6	6	6	6	6	6	6	6	6	6
7	0	0	1	5	6	6	6	6	6	6	6	6	6	6	6
8	0	0	1	5	6	6	6	6	6	6	6	6	6	6	6
9	0	0	1	5	6	6	6	6	6	6	6	6	6	6	6
10	0	0	1	5	6	6	6	6	6	6	6	6	6	6	6
11	0	0	1	5	6	6	6	6	6	6	6	6	6	6	6
12	0	0	1	7	8	9	9	9	9	9	9	9	9	9	9
13	0	0	1	7	8	9	9	9	9	9	9	9	9	9	9
14	0	0	1	10	7	9	9	9	9	9	9	9	9	9	9
15	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 46: Index map for figure 98

It is very clear from table 46 that there is substantial repetition in the rows, which can be exploited. It can be observed that rows 3-11 all follow the same pattern.

The next stage in the process involves comparing consecutive rows with each other and recording the difference. The first part of this stage is to choose a base line row for initial comparison. Row 8 from table 46 is chosen as the base-line row, where the comparing takes place in two phases. For the top half of the index map, row 7 gets compared to row 8 and the difference gets recorded. row 6 will get compared to row

7 and so on until row 0 is compared to row 1. Likewise, for the lower half of the index map: row 9 gets compared to row 8, row 10 gets compared to row 9, etc..

The row-wise comparison can be seen in table 47. The white space in a row is used to describe that there is no difference to the previous row. It can be observed that there are whole rows that have no difference and by exploiting this will greatly improve compression performance.

Row Compare	Difference Map															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
compare 0-1				1	1	1	1	1	1	1	1	1	1	1	1	1
compare 1-2				2	3	3	3	3	3	3	3	3	3	3	3	3
compare 2-3				4	5	5	5	5	5	5	5	5	5	5	5	5
compare 3-4																
compare 4-5																
compare 5-6																
compare 6-7																
compare 7-8																
base line 8	0	0	1	5	6	6	6	6	6	6	6	6	6	6	6	6
compare 9-8																
compare 10-9																
compare 11-10																
compare 12-11				7	8	9	9	9	9	9	9	9	9	9	9	9
compare 13-12																
compare 14-13				10	7											
compare 15-14				1	1	1	1	1	1	1	1	1	1	1	1	1

Table 47: compare

It can be observed from table 47 that row 8 has not changed from the index map. It is essential that the baseline row retains all of the original information, because the difference values that will be encoded for the preceding rows depend on it entirely. Without having the base line row, the rest of the block would not be able to be decoded.

Rather than explicitly coding the values of row 8, a simple run length coding can be used to reduce the coding redundancy. Using run length encoding, Row 8 could be described by (2,0)(1,1)(1,5)(12,6), where the first of the pair is the length of the run and the second is the run value. There are many rows that have the same value as the baseline row. to illustrate how to encode these rows, a symbol, **(nc)** (no change), can be used in place. To illustrate how a row with differences is to be encoded, the comparison between rows 2-3 can be described as pair-wise values:

3,4	4,5	5,5	6,5	7,5	8,5	9,5	10,5	11,5	12,5	13,5	14,5	15,5
-----	-----	-----	-----	-----	-----	-----	------	------	------	------	------	------

where the first value of the pair is the location in the row and the second value is the change value, in this case, between row 2 and row 3 from table 46

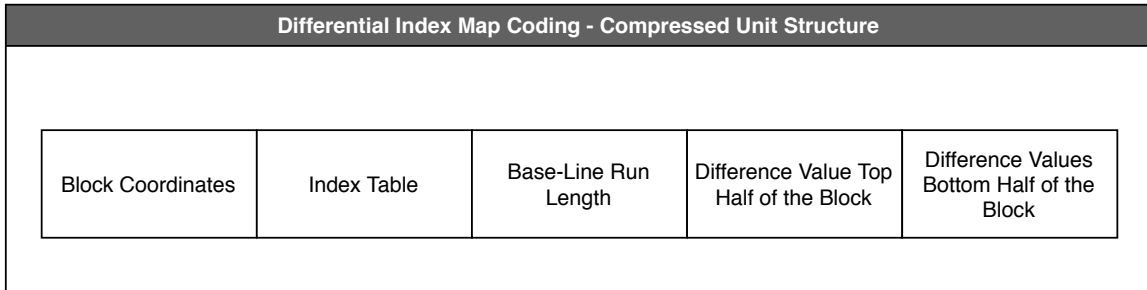


Figure 100: The Differential Index Map Coding compression unit structure

Figure 100 shows the structure of a block that would be compressed with Differential Index Map Coding. The coordinates of where the block is located, with respect to the original image are placed at the top. The next chunk is the index table, which contains the count of unique pixels in the block, with the intensity value of the unique pixels. The order that they are in, represent their index. The following chunk is the base-line run length encoding values. The next two sections are the pair-wise difference values recorded for the top half of the block, followed by the bottom half of the block.

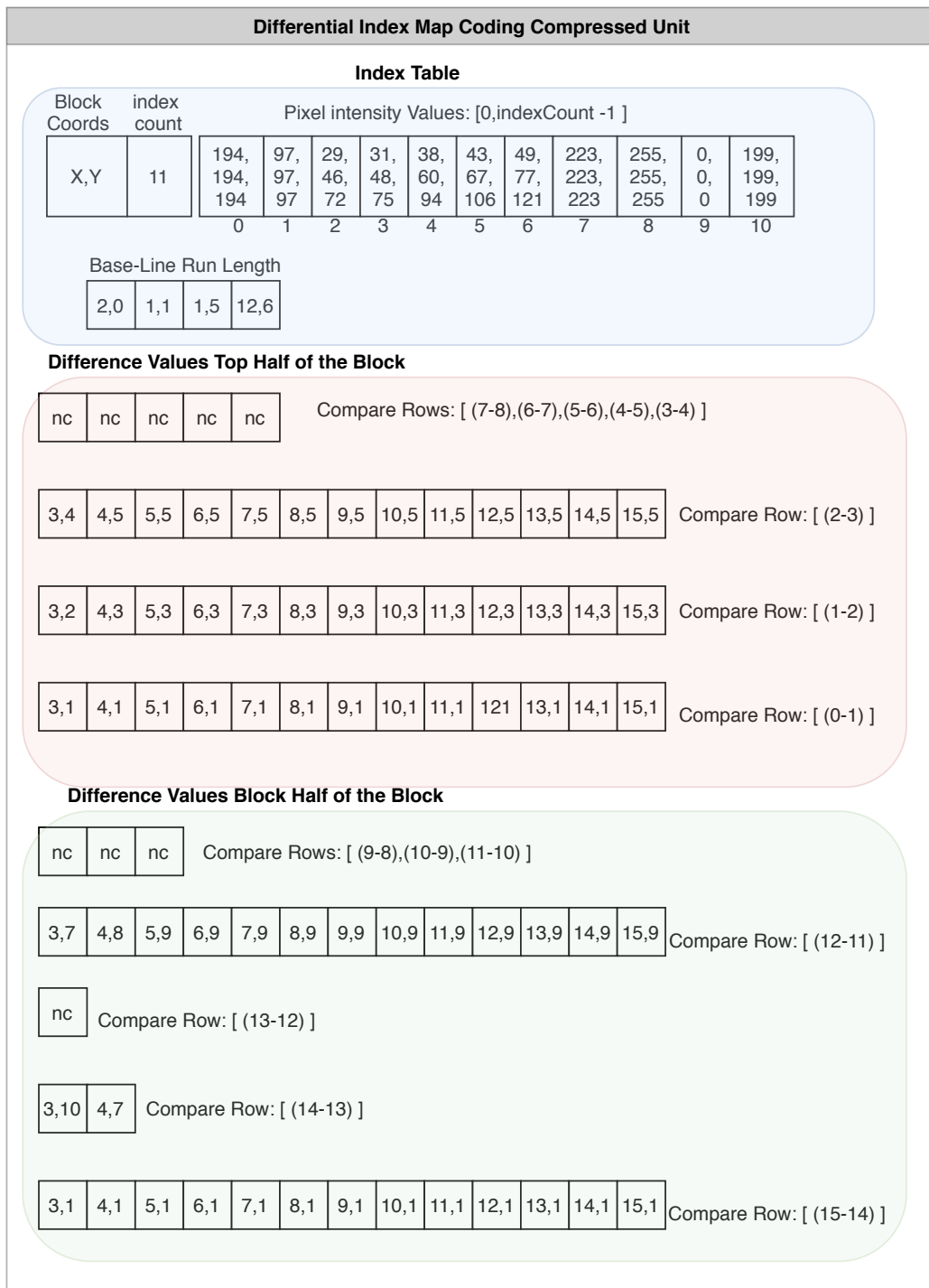


Figure 101: Illustrated Differential Index Map Coding Compression unit structure for figure 98

Figure 101 illustrates what the contents of a Differential Index Map Coding compressed unit would contain if the the block in figure 98 was compressed using the algorithm. As previously stated, the algorithm is designed specifically to compress blocks that are highly structured with a low unique pixel count. Differential Index

Map Coding can be configured in two ways: The first method will process a 16x16 block and perform row wise comparisons on rows of length 16. This method can be performed on blocks with up to a maximum of 15 unique pixel values. The second method will process a 16x16 block and will decompose it into four 8x8 blocks, which will do row-wise comparisons on rows of length 8. This method can be performed on blocks that have between 16-31 unique pixel values. The choice of the limit on unique pixel values was based upon the analysis of the amount of unique pixels contained in blocks containing computer generated data presented in this thesis.

The choice of the limit of up to 15 unique pixel values for the first method is based upon how the values can be encoded in the encoded stream in an efficient way. Only four bits are needed to represent any one of the 16 columns in a row of a block of 16x16 values. When a comparison is made between two rows, the location of the change in the row gets recorded, along with the difference value. To illustrate this, consider the comparison between row 14 and row 13 from the index map presented in table 46, shown again in table 48:

Rows	Columns															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Row 13	0	0	1	7	8	9	9	9	9	9	9	9	9	9	9	9
Row 14	0	0	1	10	7	9	9	9	9	9	9	9	9	9	9	9

Table 48: Two row comparison

Comparing row 14 to row 13 it can be observed there are two changes: column location 3 with an index value of 10 and column location 4, with an index value of 7. Only 4 bits are needed to uniquely identify each column coordinate in a row. Further, if the algorithm is limited to 15 unique pixel values per block, only 4 bits are needed to represent each index. Therefore a single byte can then be used to describe the location of the change and also the the difference index value. Using the top most significant nibble of an 8 bit byte to represent the change location and the least significant to represent the difference index value, the encoding for the two row comparison of table 48 is described in table 49

location	value	location	value	EOL
3	10	4	7	255
0011	1010	0100	0111	1111 1111

Table 49: Encoded bitstream of two row comparison

Where **EOL** represents the **End of Line** of a two row comparison and is given the value of 255 (all ones in binary). It is a special symbol that is used in Differential Index Map Coding, that is necessary in the decoding process. This is the reason that

15 unique pixel values are the limit instead of 16 for this method and 31 unique pixel values instead of 32 for the second method.

Along with the **EOL** special symbol, another special symbol is used in Differential Index Map Coding to represent the result of no change when comparing rows. The **nc** (no change symbol) is a special symbol which is an 8 bit byte, with the least significant nibble set to all ones in binary. The most significant nibble gets incremented depending on the amount of rows that have not changed during the comparing stage.

Num of No Change Rows	Most Significant Nibble	Least Significant Nibble
1	0000	1111
2	0001	1111
3	0010	1111
4	0011	1111
5	0100	1111
6	0101	1111
7	0110	1111
8	0111	1111

Table 50: Table of **nc** special symbols

Table 50 lists the binary values of the **nc** symbols. The process of comparing rows is performed in two parts, The top half of the block, with respect to the base line row and the bottom half of the block. If row 8 is chosen as the base line row for comparison, there is a maximum of 8 no change symbols needed to represent up to 8 rows that have no difference to the base line row.

To illustrate using the **nc** special symbol, consider the scanning process of the bottom half of the index map of table 46, which is shown again in table

Rows	Columns															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	0	0	1	5	6	6	6	6	6	6	6	6	6	6	6	6
9	0	0	1	5	6	6	6	6	6	6	6	6	6	6	6	6
10	0	0	1	5	6	6	6	6	6	6	6	6	6	6	6	6
11	0	0	1	5	6	6	6	6	6	6	6	6	6	6	6	6
12	0	0	1	7	8	9	9	9	9	9	9	9	9	9	9	9
13	0	0	1	7	8	9	9	9	9	9	9	9	9	9	9	9
14	0	0	1	10	7	9	9	9	9	9	9	9	9	9	9	9
15	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 51: Base line row and bottom half of index map from table 46

It can be observed that comparing row 9 to row 8, there is no difference, likewise rows: 10-9 and 11-10. The no change symbol that would be used to represent the results of the three comparisons would be:

Num of No Change Rows	Most Significant Nibble	Least Significant Nibble Nibble
3	0010	1111

nc 0010 1111	l=3,d=7 0011 0111	l=4,d=8 0100 1000	l=5,d=9 0101 1001	l=6,d=9 0110 1001	l=7,d=9 0111 1001	l=8,d=9 1000 1001
l=9, d=9 1001 1001	l=10,d=9 1010 1001	l=11,d=9 1011 1001	l=12,d=9 1100 1001	l=13,d=9 1101 1001	l=14,d=9 1110 1001	l=15,d=9 1111 1001
EOL 1111 1111	nc 0000 1111	l=3,d=10 0011 1010	l=4,d=7 0101 0111	EOL 1111 1111	l=3,d=1 0011 0001	l=4,d=1 0100 0001
l=5, d=1 0101 0001	l=6,d=1 0110 0001	l=7,d=1 0111 0001	l=8,d=1 1000 1001	l=9,d=1 1001 0001	l=10,d=1 1010 0001	l=11,d=1 1011 1001
l=12, d=1 1100 0001	l=12,d=1 1101 0001	l=14,d=1 1110 0001	l=15,d=1 1111 0001			

Table 52: Encoding process for bottom half of index map 46

Table 52 shows the full scan for the lower half of the index map for figure 98. Applying Differential Index Map Coding to the full block results in a compressed size of 94 bytes, which yields a compression ratio of approximately 8.2:1, which is very efficient for lossless compression, considering the size of the data set (768 bytes) and the highly structured and complex nature of the information within the block.

The second method in Differential Index Map coding expands the amount of unique 3 byte pixel values per block up to a value of 31 unique pixel values. The principle of operation is the same: the coordinates of the block are recorded and an index table of between 16 to 31 unique pixel values is recorded, however the comparison process differs. A given 16x16 block is split into 4 8x8 blocks and an index map is generated for each sub block, where all sub blocks use a common index table, to reduce the overhead.

To illustrate this, figure 102 is a 16x16 block that contains text and has 18 unique 3 byte pixels that has been extracted from an image. Table 53 Shows the four index maps that describe figure 102 and highlight the rows that are used for base line run length encoding.

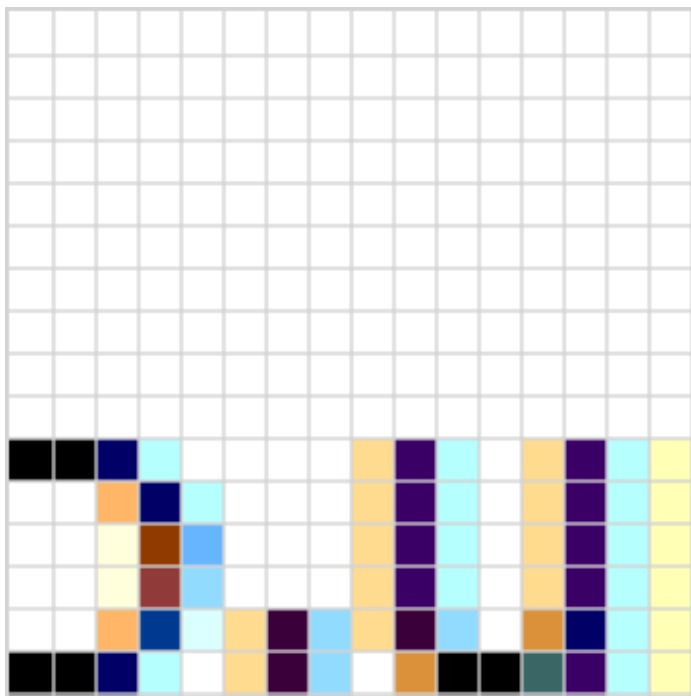


Figure 102: Block with 18 unique pixel values

Rows	Columns															
	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	1	2	3	0	0	0	0	4	5	3	0	4	5	3	6
3	0	0	7	2	3	0	0	0	4	5	3	0	4	5	3	6
4	0	0	8	9	10	0	0	0	4	5	3	0	4	5	3	6
5	0	0	8	11	12	0	0	0	4	5	3	0	4	5	3	6
6	0	0	7	13	14	4	15	12	4	15	12	0	16	2	3	6
7	1	1	2	3	0	4	15	12	0	16	1	1	17	5	3	6

Table 53: Using four 8x8 index maps for to describe figure 102

For the decoder to be able to decide which of the two modes used in encoding for a

given block, an extra byte is inserted into the header information for each block that is compressed with Differential Index Map coding, which is shown in table 54

bit	Description
0	0 = IndexMap16x16 : 1 = IndexMaps8x8
1	Not currently used
2	Not currently used
3-7	Amount of unique pixel values: 0-31

Table 54: Byte describing which Differential index Map Coding mode and count of unique pixel values

Figure 103 illustrates the process flow of Differential Index map Coding, While figure 104 Illustrates the flow of the actual compression algorithm. For illustrative purposes, classification of a given block has been omitted in figure 103, in favour of clarity.

Differential Index Map coding has been designed to compress in a "first pass" through the image data, As to improve on performance with respect to latency. As the algorithm compresses a block, the output is serially stored in a vector, which can be processed with an entropy coder, such as Huffman encoding when all blocks have been processed. As Each block that is compressed with Differential Index Map Coding is processed in isolation, without any need of prior knowledge of other blocks, the algorithm would be suitable for parallel processing.

The implementation of Differential Index Map Coding in c++ can be seen in the code section of this thesis in Listing 27

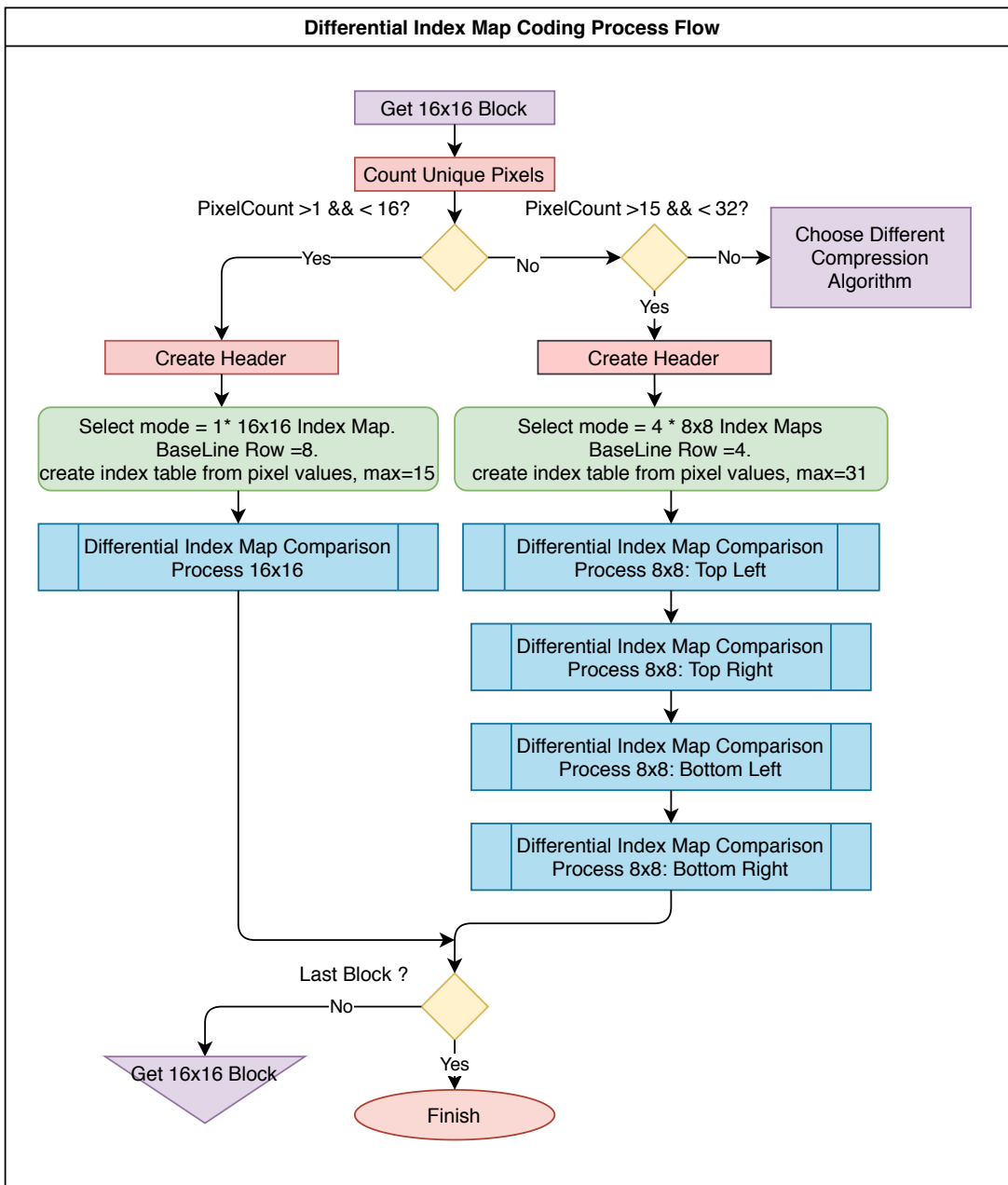


Figure 103: The process flow of Differential Index Map Coding Algorithm

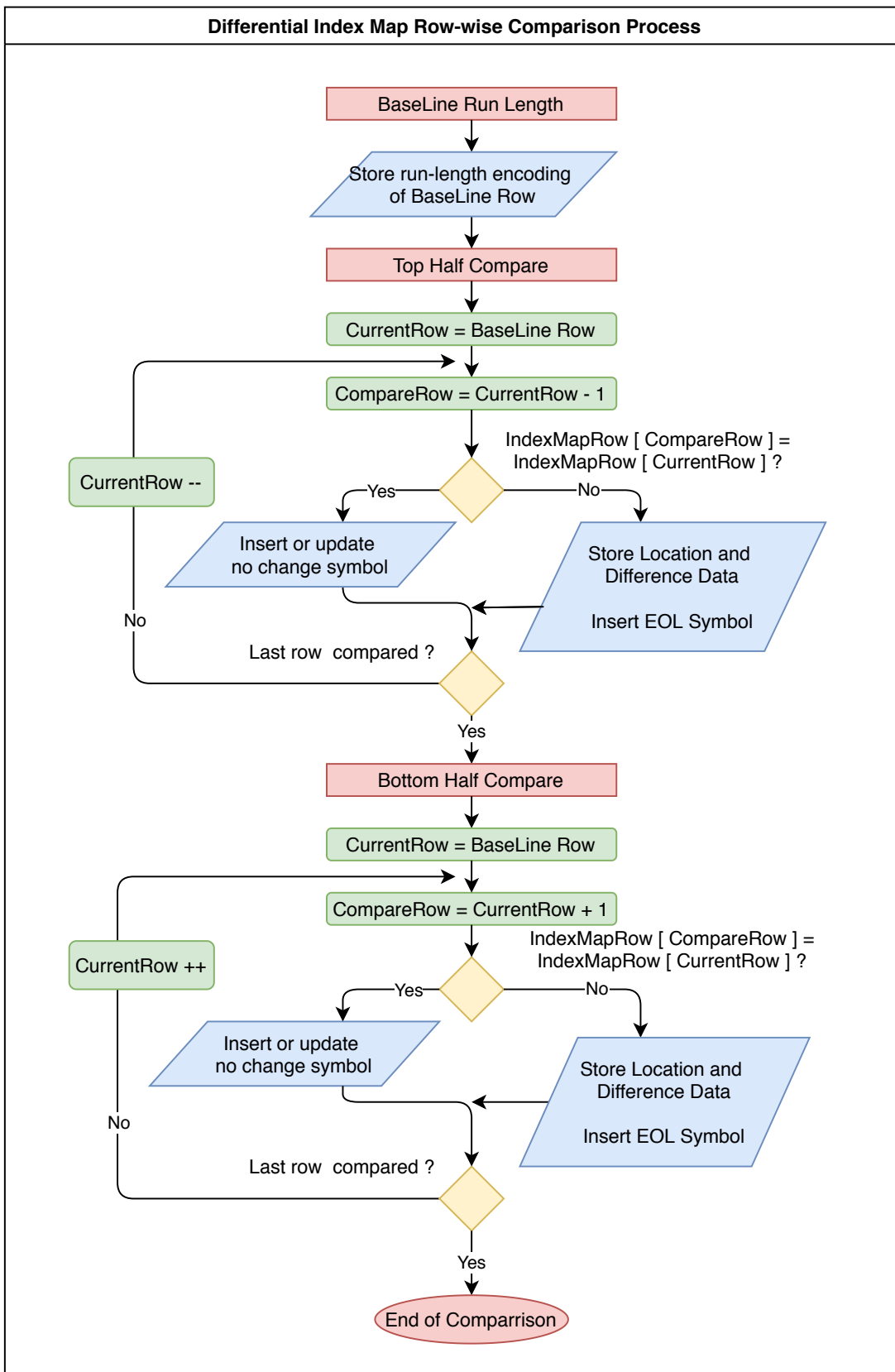


Figure 104: The process of row wise comparison in the Differential Index Map Coding Algorithm

A test is configured to compress blocks classified as sparse and text from a set of 40 compound Images. Sparse blocks will be encoded with Differential Index Map Coding mode one (as all blocks classified as sparse will contain less than 15 unique pixels) and Text blocks with up to 31 unique pixel values with a combination of Differential Index Map Coding mode one and mode two. Text blocks that have greater than 31 unique pixel values are omitted from this test. The pseudo code for the interface for this test can be seen in listing 10

Listing 10: Differential index Map Coding on Sparse and Text Blocks Pseudo Code

```

1 Compress(src_Image){
2     /*max_row = amount of 16x16 blocks in a row of src_Image*/
3     max_row = src_Image.rows /16;
4     /*max_col = amount of 16x16 blocks in a column of src_Image*/
5     max_col = src_Image.columns/16;
6     DIMCSparseData[]; // array to hold serialized encode Sparse data
7     DIMCTextData[]; // array to hold serialized encoded Text data
8     blockChoice; //Sparse, Text
9
10    huffCodedSparseData[]; // array to hold Huffman encoded DIMC Sparse Data
11    huffCodedTextData[]; // array to hold Huffman encoded DIMC Text Data
12
13    for(i =0; i < max_row; i++) {
14        for(j=0; j< max_col; j++) {
15
16            blockType=classifyBlock(src_Image, i j); //block at coordinate i,j
17
18            if(blockType=Sparse)
19                DIMC_Compress_Mode_1(src_Image,i,j,DIMCSparseData)
20                Sparse_Block_Count++
21
22            else if(blockType=Text){
23
24                if (pixelCount < 16)
25                    DIMC_Compress_Mode_1(src_Image,i,j,DIMCTextData)
26                    Text_Block_Count++;
27
28                else if (pixelCount < 32)
29                    DIMC_Compress_Mode_2(src_Image,i,j,DIMCTextData)
30                    Text_Block_Count++;
31                else:
32                    Break;
33            }
34            else:
35                Break;
36        }
37    }
38    Perform_Huffman_Encoding(DIMCSparseData,huffCodedSparseData)
39    Perform_Huffman_Encoding(DIMCTextData,huffCodedTextData)

```

40 }

Average Compression Ratio vs. Block Type: Differential Index Map Coding No Huffman

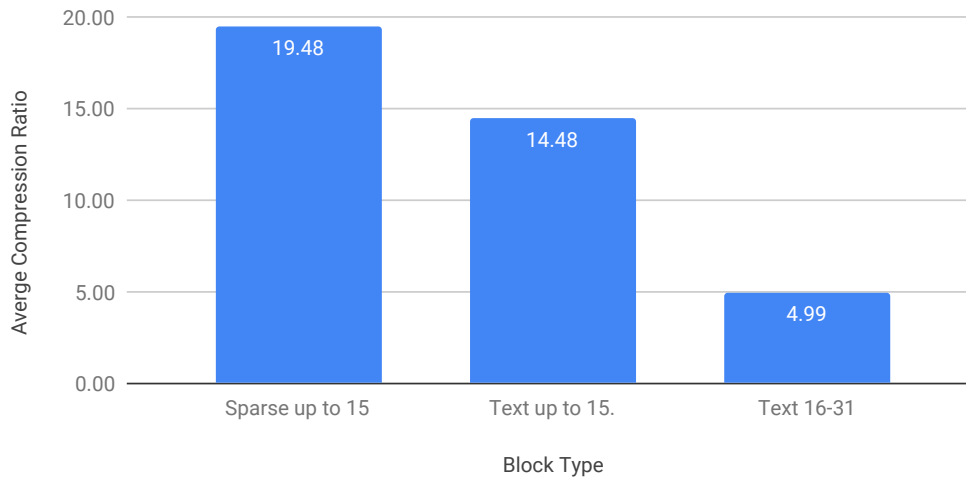


Figure 105: The average compression ratio achieved using Differential Index Map Coding on blocks classified as Sparse or Text with up to 31 unique 3-byte pixel values from a set of 40 compound images

Figure 105 Shows the results of compressing sparse and text blocks with Differential Index Map Coding. It can be observed that good compression performance is achieved for both data types with up to 15 unique pixels and average compression for text blocks with unique pixel counts between 16 and 31 unique pixels.

To reduce code redundancy and improve compression performance of the Differential Index Map Coding algorithm further, the compressed data stream can be processed using an entropy encoder such as Huffman encoding. The Huffman encoding algorithm that has been implemented in this thesis is from the zlib library [8].

Figure 106 shows the results of the same test with the encoded Differential Index Map Coding data further processed with Huffman encoding. It can be observed that the combination of both algorithms produces very high compression performance, especially for blocks classified as sparse.

Average Compression Ratio vs. Block Type: Differential Index Map Coding with Huffman

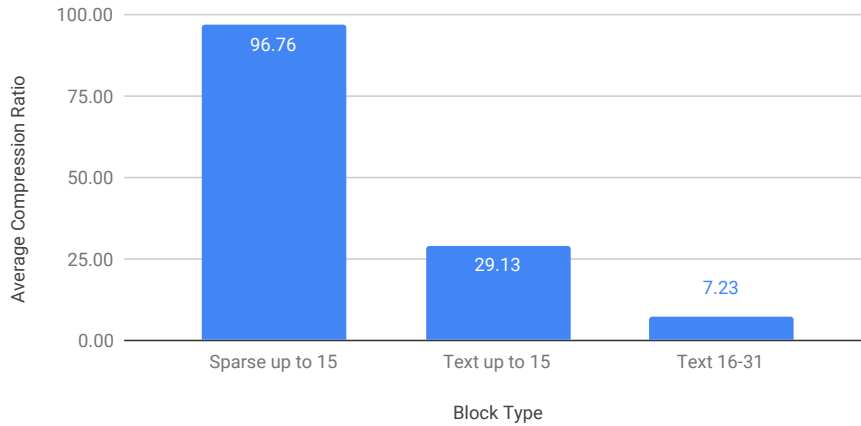


Figure 106: The average compression ratio achieved using a combination of Differential Index map Coding and Huffman coding on blocks classified as Sparse and Text with up to 31 unique 3-byte pixel values from a set of 40 compound images

9.2.6 Conclusions on Compression Performance of Differential Index Map Coding

The results from tests using Differential Index Map Coding algorithm to compress data that has been classified as sparse and data that has been classified as text, with up to 31 unique pixel values show that it has excellent compression performance. It is a lossless compression algorithm and can achieve, extremely high compression ratios, comparable and better to lossy type compression.

Key benefits of the algorithm over fully dictionary based methods such as Lempel-Ziv Welch and the Deflate algorithm include the ability to perform compression in a "first pass" fashion, working in tangent with the classification algorithm. As soon as a block has been identified to be suitable, the block can be processed. Also, each block can be compressed in isolation, which could potentially improve the performance of the algorithm if distributed or parallel processing is used when compressing on the host side.

From research performed on unique pixel count analysis which has been shown, the Differential Index Map Coding Algorithm has been specifically designed to encode blocks of pixels with low pixel count. As such, it has not been designed to process blocks that have greater than 31 unique pixel values. It would be possible to implement the algorithm to recursively divide down the block size and use a greater amount of unique pixel values, but this would require extra meta data in the header for each block which would significantly reduce the efficiency of the compression performance.

9.2.7 Compression and Speed Performance Comparison: Lempel-Ziv Welch Vs Differential Index Map Coding Vs Deflate

In this section, three lossless compression algorithms will be compared with respect to compression performance and compression speed. The algorithms that will be compared are Lempel-Ziv Welch, Deflate and Differential Index Map Coding.

Compression Test Experimental Design

The algorithms will be used to compress blocks that have been classified as sparse and blocks that have been classified as text from a set of 40 compound images. The Differential Index Map Coding algorithm is designed to only compress blocks that have a unique pixel count of up to 31, thus, only blocks with less than 32 unique pixel values will be compressed by all three algorithms, to ensure the accuracy of the comparison.

By design, all blocks that are classified as sparse have less than 15 unique pixel values, and on average, approximately 16 percent of blocks classified as text have greater than 31 unique pixel values. Text blocks with greater than 31 unique pixel values will be omitted from testing.

The first test is to compare the compression performance of blocks that have been classified as sparse for all three algorithms. The configuration settings for the test are as follows:

- The blocks that have been classified as sparse will be serialized as "packed pixels" for both Lempel-Ziv Welch and for Deflate as this method has been shown to be the most optimal for pre processing with respect to compression performance.
- Deflate is used in the mode for best compression performance.
- Huffman entropy encoding is used with Differential Index Map Coding.

Compression Performance Comparison: Sparse Blocks

Figure 107 shows the average compression performance of the three algorithms when compressing data that has been classified as sparse. It can be observed that Differential Index Map Coding clearly has the best compression performance.

Average Compression Ratio vs. Compression Algorithm: Sparse Blocks

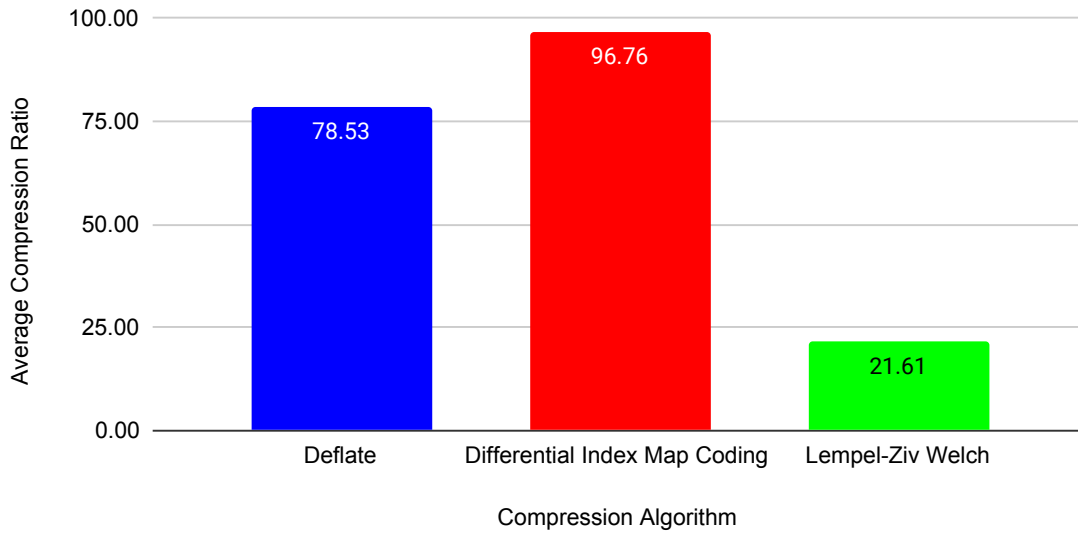


Figure 107: Comparing Deflate, Differential Index Map Coding and Lempel-Ziv Welch encoding on blocks that have been classified as Sparse

The standard deviation is calculated for the results for each algorithm to give an indication on the uniformity of each compression algorithm. Having a low standard deviation is important because it allows the right amount of resources to be provisioned which is important when considering an algorithm that should be used on a network with limited bandwidth.

Compression Algorithm	Standard Deviation
Deflate	35.61
Differential Index Map Coding	44.23
Lempel-Ziv Welch	4.89

Table 55: Comparing sample standard deviation on results for figure 107

Table 55 shows the results of calculating the sample standard deviation of the results for the three compression algorithms on blocks that have been classified as sparse.

It can be observed that, with respect to the average compressed size, the standard deviation for each algorithm is low.

Compression Performance Comparison: Text Blocks

The next test will compress blocks that have been classified as text using the three algorithms. only text blocks with up to 31 unique pixel values will be chosen. packed pixel serialization will be used for Lempel-Ziv Welch and Deflate. Huffman entropy encoding will be used with Differential Index Map Coding.

The first set will be be text blocks with up to 15 unique pixel values and the second set will be blocks that have 16-31 unique pixel values.

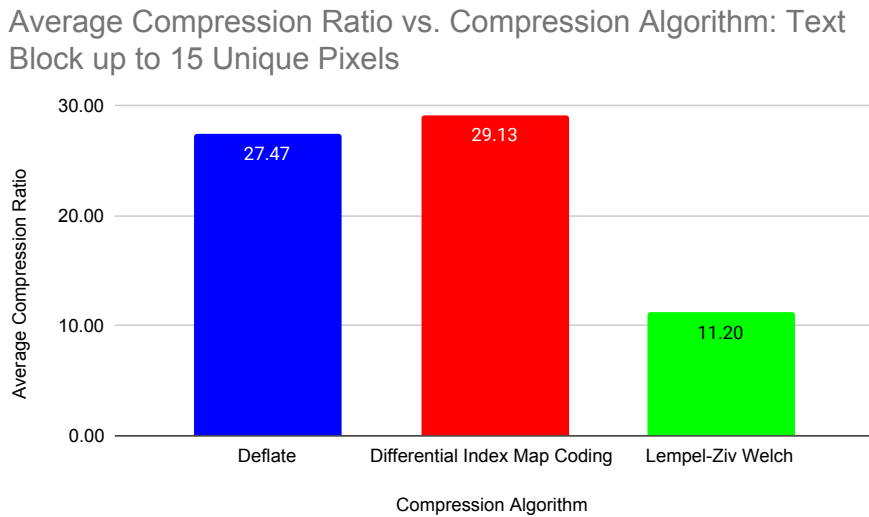


Figure 108: Comparing Deflate, Differential Index Map Coding and Lempel-Ziv Welch encoding on blocks that have been classified as Text with up to 15 unique pixel values

It can be observed from figure 108 that Differential Index Map Coding out performs both Deflate and Lempel-Ziv Welch. This is a significant result, as it has been previously shown that on average, approximately 73 percent of blocks that are classified as text contain up to 16 unique pixel values, which is the vast majority of blocks that have been classified as text. Deflate is an industry standard compression algorithm, it is so because of its efficiency and speed performance. In this research, it has been further optimised to only compress image data that is suitable for the algorithm, increasing its performance with respect to image data. The results show that Differential Index Map Coding has out performed Deflate encoding for the sample set of images used in this research.

Average Compression Ratio vs. Compression Algorithm: Text Blocks 16-31 Unique Pixel Values

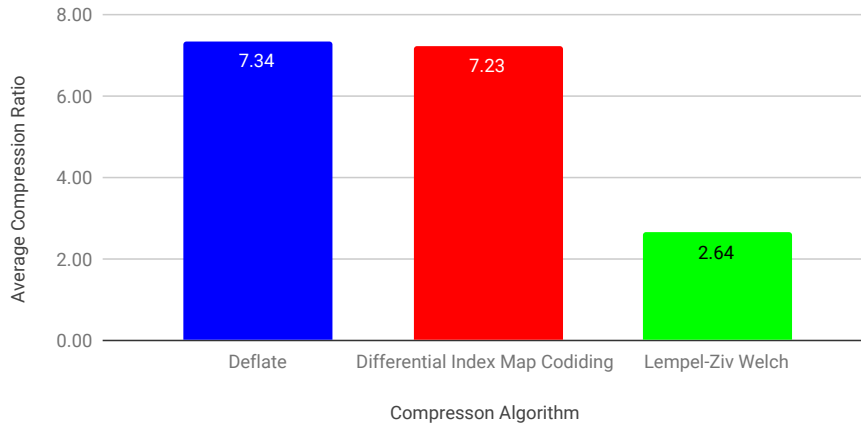


Figure 109: Comparing Deflate, Differential Index Map Coding and Lempel-Ziv Welch encoding on blocks that have been classified as Text with 16-31 unique pixel values

Figure 109 Shows the comparison of the three algorithms compression performance on text blocks with between 16-31 unique pixel values. Here It can be seen that Deflate algorithm and Differential Index Map Coding are similar, with Deflate out performing Differential index Map coding by approximately 1.4 percent

Compression Speed Test Experimental Design

The speed performance of Deflate, Differential Index Map Coding and Lempel-Ziv Welch will now be compared. An Image has been selected from the set of 40 compound images that has a significant number of blocks that are classified as text. From the blocks that have been classified as text, the blocks with up to 31 unique pixels have been identified. To compare the speed performance of the three algorithms, 1000 blocks that have been classified of text with up to 31 unique pixel values will be compressed. The time incurred from using the classification algorithm is not included in the test. The steps of the test are as follows:

- For Deflate and Lempel-Ziv Welch encoding, each block will be serialized as packed pixels. The time taken to serialize the data is included in the overall time, as it is part of the processing.
- The Deflate algorithm has three modes for optimisation: best speed performance, best compression performance and default. All three modes will be tested.
- Colour counting is a function of the Differential Index Map Coding algorithm. Even though the blocks that have been chosen for the test have already been

identified as having up to 31 unique pixel values, colour counting is still performed for each block as it is part of the processing.

- The test will be run 10 times, where the result will be the average value of the set.
- The test is timed using the standard library steady clock class implemented in C++ 11, which is a highly accurate timing class.
- The tests are run on a Lenovo Thinkpad 440p with intel i5-4300M CPU, with 2 cores and 4 threads.

Compression time (micro seconds) vs. Compression Algorithm: 1000 Blocks Classified as Text.

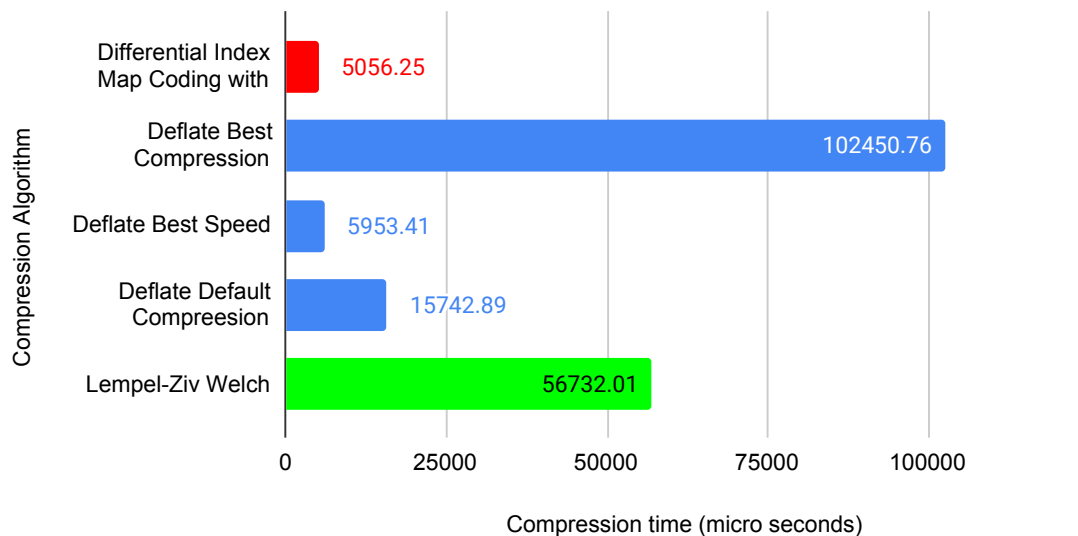


Figure 110: Showing Compression time of 1000 blocks classified as text with up to 31 unique pixel values

Figure 110 shows the results of the comparison of Deflate, Differential Index Map Coding and Lempel-Ziv Welch coding with respect to compression time. It can be seen that Differential Index Map Coding out performs both Deflate and Lempel-Ziv Welch.

This is significant result. It can be observed that Differential Index Map Coding has lower computational time than the Deflate, even with Deflate optimised for best speed performance and is orders of magnitude faster than Deflate when it is optimised for best compression.

9.2.8 Conclusion on Lossless Compression testing

Multiple compression algorithms have been presented and tested for losslessly compressing the data that would be computer generated within a compound image. A novel approach to compress blocks of pixels which contain a single colour has been presented and shows extremely efficient compression with average compression ratios of greater than 1000:1, when applied to the test set of images. This efficient compression method is both computationally simple and significantly reduces the size of the amount of data needed for transmission.

A novel compression algorithm called Differential Index Map Coding which has been designed to compress blocks that are highly structured with low pixel count has been presented. It has been tested and compared against two industry standard compression algorithms (Deflate and Lempel-Ziv Welch) and has been shown to achieve higher compression performance with reduced computational overhead for the data that it is designed to compress. It is fully capable to compress all blocks that have been classified as sparse and the majority of blocks that have been classified as text which can be tightly integrated in to the classification algorithm that has already been presented in this thesis.

For the remaining blocks that cannot be compressed using Differential Index Map Coding, The Deflate algorithm may be a possible candidate. The Deflate algorithm can be used in different modes to optimise for best speed or best compression. Tests have shown that Deflate is capable of compressing data classified as text with good compression performance. However, using the mode for best compression introduces a significant amount of added time, whereas optimising for best speed reduces the compression performance. A good balance between compression performance and speed is to use the default compression mode and to try find other means to improve compression performance with the algorithm. Optimisations in the serialization of the data using a packed pixel method have been shown to improve its performance and thus it may be suitable to use in a compound compression algorithm for time critical applications.

The Lempel-Ziv Welch algorithm that has been presented has shown moderate to good compression, however both the processing time and compression performance is inferior to both Deflate and Differential Index Map coding. As such it will not be considered for the final compound compression algorithm.

9.3 Lossy Compression Testing

In this section, the compression algorithms to process blocks that have been classified as fuzzy and as picture will be tested. The majority of blocks that have been classified as fuzzy and picture using the classification algorithm developed for this thesis contain natural continuous tone image.

Compression algorithms such as JPEG [18], video encoding algorithms such as H.264 [47] and HEVC [39] and research presented in [23], [9] and [36] have shown success in compressing continuous tone image with a transform based compression strategy, incorporating the discrete cosine transform, chroma subsampling and quantisation. Based on the published results in [23], [9] and [36] and the wide spread adoption of JPEG [18], designing a compression algorithm based on the discrete cosine transform seems like a good choice to use on blocks that have been classified as fuzzy and picture and will be used in this thesis.

When using a block based coding algorithm for compound images, it is inevitable that there will be some blocks that will contain a mixture of continuous tone image and computer generated data, which results in a highly structured block which may have a large unique pixel count. Such a block may contain text, however compressing such a block with a lossless compression algorithm may not reduce the amount of data sufficiently for transport or storage. To overcome this challenge, the work presented in [9] also includes a transform based compression algorithm based on the discrete wavelet transform. The discrete Haar wavelet transform, which has already been described in this thesis for use in both classification and compression will be tested with blocks that have been classified as text with a pixel count greater than 31 unique pixel values.

When performing chroma sub sampling and quantisation, as used with the discrete cosine transform, or thresholding as used with the discrete wavelet transform, when used for compression, some of the original data will be discarded in favour of better compression performance which makes the compression of a lossy type. This will effect both the subjective and objective image quality after decoding as well as the size of the encoded data. As a result of this, the image quality assessment metrics which have been described in section 6.2.1 will be used to decide the effectiveness of the compression algorithm, as well as compute time and compression ratio.

The parameters for the compression algorithm that incorporates the discrete cosine to be tested are as follows:

- Chroma sub sample strategy.
- The quantisation matrices used.

In section 6.2.4 three implementations of the discrete cosine transform were tested and the fastest implementation of the transform is the AAN [1] transform. This is

the method used in this research, for which the implementation in C++ can be seen in the code section of this thesis in listing 25.

The parameters for the discrete wavelet transform based compression to be tested are as follows:

- The base implementation in floating point and integer.
- The resolution of the transform.
- The threshold values.
- The non zero coefficient ordering.

9.3.1 Chroma Sub Sampling Testing

The first test is to compare two strategies for chroma sub sampling to be used when encoding blocks that have been classified as fuzzy and blocks that have been classified as picture. As previously discussed in this thesis in the section on the discrete cosine transform, chroma subsampling is the process of using less sample values for the chrominance channels with respect to the luminance channels. In this thesis the sample rate of 4:2:0 will be used which means that there are four luminance samples taken for every 1 of each colour channel, Cb and Cr. This effectively reduces the amount of source data by half. As the sampling is done in the YCbCr colour space, some of the colour information is stored in the Y channel, so the effects of chroma sub sampling are not as severe as if performed on an image in the RGB colour space, where each channel is mutually exclusive.

Listing 11: Chroma Sub Sampling Top Left, C++

```

1 sub_sample_first_val(Block16x16)
2 {
3     for(row =0; row < 16; row++)
4     {
5         for(column =0; column <16;column++)
6         {
7             // get the 3 byte pixel value at (row,column)
8             pixel_pointer = Block16x16[row,column]
9
10            // statement evaluates true for every two pixels:row and column
11            // ex: (row,col) (0,0), (0,2), (2,0) (2,2)
12            if(!(row & 1) && !(column & 1) {
13                store pixel_pointer[row,column].Y_val
14                store pixel_pointer[row,column].Cb_val
15                store pixel_pointer[row,column].Cr_val
16            } else
17                store pixel_pointer[row,column].Y_val

```

```

18     }
19   }
20 }

```

Listing 11 shows the first method of chroma subsampling. The top left pixel in a neighbourhood of a 2x2 block of pixels is select and its Cb and Cr value is used to represent the remaining three pixel's Cb and Cr value. The benefit of choosing a single sample to represent the other 3 pixels is that no calculation has to be performed to compute the average value, which may improve computational time. The adverse effect may be that it might accumulate more error in the data that will be further processed by the discrete cosine transform and quantization.

Listing 12: Chroma Sub Sampling Average Value, C++

```

1  sub_sample_avg_val(Block16x16)
2  {
3      for(row =0; row < 16; row++)
4      {
5          for(column =0; column <16;column++)
6          {
7              // get the 3 byte pixel value at (row,column)
8              pixel_pointer = Block16x16(row,column)
9
10             // statement evaluates true for every two pixels:row and column
11             // ex: (row,col) (0,0), (0,2), (2,0) (2,2)
12             if(!(row & 1) && !(column & 1))
13             {
14                 store pixel_pointer[row,column].Y_val
15
16                 avg_Cb = ( pixel_pointer[row,column].Cb
17                     + pixel_pointer[row,column+1].Cb
18                     + pixel_pointer[row+1,column].Cb
19                     + pixel_pointer[row+1,column+1].Cb ) / 4
20
21                 avg_Cr = ( pixel_pointer[row,column].Cr
22                     + pixel_pointer[row,column+1].Cr
23                     + pixel_pointer[row+1,column].Cr
24                     + pixel_pointer[row+1,column+1].Cr ) / 4
25
26                 store avg_Cb
27                 store avg_Cr
28             }
29             else
30                 store pixel_pointer.Y_val
31         }
32     }
33 }

```

34 } 

The second chroma sub sampling method calculates the average value of 4 pixel sample values and uses that to represent each pixel in the neighbourhood's Cb and Cr value. The pseudo code for this method can be seen in listing 12. The possible benefit of taking the average value for a group of 4 samples is that it may reduce some of the error in the data to be further processed. The possible negative effect of method two is that it introduces more arithmetic which could impact the speed performance of the algorithm.

A test is configured to measure the error that the two described methods for chroma sub sampling introduce into the image data to be processed. The steps of the test are outlined in the following experimental design:

Experimental Design: Peak Signal to Noise Ratio test on Chroma Sub Sampling methods

- An image is processed using the classification algorithm.
- If a given block is classified as picture or fuzzy, it is chroma subsampled by each method.
- Inverse subsampling is performed on the chroma subsampled data and a 16x16 block is reconstructed for both methods.
- The peak signal to noise ratio is calculated for the reconstructed block, which takes the original unprocessed block as a reference.
- The PSNR value is recorded for each block that has been processed and the mean value is calculated for both methods.
- All steps repeated for each image of the test set of compound images for this thesis.

The results of the above test can be seen in table 56. It can be observed that Method 2, which takes the average value of a neighbourhood of 4 pixels performs significantly better, with close to 3dB in the difference.

Chroma Sub Sample Method	Average PSNR Value (dB)
Method 1 (top left)	42.186
Method 2 (avg value)	44.984

Table 56: Results of Peak Signal to Noise testing using two Chroma Sub Sampling methods

The next step is to compare the difference in processing time between both modes of Chroma Sub Sampling. The steps for the test are outline in the following Experimental Design

Experimental Design: Comparing computation time between two Chroma Sub Sampling methods

- 1000 blocks are selected from an image.
- Each mode processes the blocks and the time is recorded.
- The process is run 10 times for each mode.
- The results are recorded each time and the average value is calculated for each mode.
- The tests are run on a Lenovo Thinkpad 440p with intel i5-4300M CPU, with 2 cores and 4 threads.

The results of the test outline above can be seen in table 57. It can be seen that mode 1 (top left) out performs mode 2 (average value). It completes the same processing in approximately 82 percent of the time.

Chroma Sub Sample Method	Average time to process 1000 Blocks (μS)
Method 1 (top left)	573.34
Method 2 (avg value)	694.09

Table 57: Results of Peak Signal to Noise testing using two Chroma Sub Sampling methods

9.3.2 Comments on Chroma Sub Sampling Methods

The tests performed shows that taking an average value of four pixel samples to represent the Cr and Cb value in a neighbourhood of 4 pixels, reduces the amount of error in the signal significantly compared to taking a single value of the neighbourhood of pixels. However, it comes at a price of reduced speed performance. This presents a choice that can be used to optimise the final compound compression algorithm, one can choose to fully optimise for compression performance or for speed.

Even though mode two is slower, it can be argued that both methods are efficient with respect to processing time and that the reduction in error that incurs from using mode one may have greater significance to the over all performance of the final compound compression algorithm.

9.3.3 Performing the Two-Dimensional Discrete Cosine Transform as Two One-Dimensional Transforms

In the section describing the discrete cosine transform in this thesis, it has been shown that performing the two-dimensional discrete cosine transform can be performed as two one-dimensional transforms, to reduce the amount of computations. It has also been shown that the method known as the AAN transform [1] further reduces the amount of computations needed to perform the transform significantly.

The one-dimensional discrete transform must first be computed along the columns of a block of 8x8 one byte pixel samples. The discrete cosine function which can be seen in the code section of this thesis in listing 25, takes a reference to an array of 8 values as an argument and performs the computations. The call to the function can be seen in the pseudo code in listing 13. [H]

Listing 13: Calling the one-dimensional discrete cosine function pseudo code

```

1 one_d_transform_8x8(block[8][8])
2 {
3     for(row =0; row<8;row++)
4     {
5         FastDct8::transform(block[row][0])
6     }
7 }
```

The 8x8 block must then be transposed and then the one-dimensional discrete cosine is computed, this time it will be computed along the row values of an 8x8 block of one byte pixel samples. The code for the transpose function can be seen in listing 14

Listing 14: Transpose a block of 8x8 one byte pixel values pseudo code

```

1 transpose(block[8][8])
2 {
3     for(row =0; row <8; row ++)
4     {
5         //set col = row, so it does not overwrite previous swaps
6         for(col =row; col <8; col ++)
7         {
8             // temp variable to hold intermediate swap value
9             temp = dest[col][row]
10            //perform a swap
11            block[col][row] = block[row][col]
12            block[row][col] = temp
13        }
14    }
15 }
```

The next stage after performing the discrete cosine transform in transform encoding, is to perform quantization on the discrete cosine transform coefficients

9.3.4 Quantization of Discrete Cosine Transform Coefficients

The second stage of discarding information in favour of better compression performance using a discrete cosine transform is quantisation. A quantization matrix is a term given to a block of quantization values that are used to quantize transform coefficients, with a specific quantization coefficient used for each discrete cosine transform coefficient.

As previously stated, blocks that are classified as picture and fuzzy, generally contain continuous tone image, primarily from natural camera captured image. The purpose for the the distinction is that blocks that are classified as fuzzy are generally structureless and exhibit no discernible pattern, so the hypothesis is that blocks that have been classified as fuzzy can be quantized harder in favour of better compression performance, without having as much of an impact on subjective image quality.

For this thesis, five quantization matrices have been selected:

- The reference luminance and chrominance quantization matrices from the JPEG standard [18].
- The intra mode quantization matrix used in H.264 presented in the book by *Iain Richardson*, "H.264 and MPEG-4 Video Compression : video coding for next-generation multimedia" [33].
- The reference intra mode quantization matrix from the HEVC standard, which is presented in the research done by *Prangnell* in his work "Minimizing Compression Artifacts for High Resolutions with Adaptive Quantization Matrices for HEVC" [31] and also the AQM matrix presented by the same author in [31]

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

(a) JPEG Luminance Quantization Matrix [18]

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

(b) JPEG Chrominance Quantization Matrix [18]

Table 58: Reference quantization matrices from the JPEG standard [18]

6	10	13	16	18	23	25	27
10	11	16	18	23	25	27	29
13	16	18	23	25	27	29	31
16	18	23	25	27	29	31	33
18	23	25	27	29	31	33	36
23	25	27	29	31	33	36	38
25	27	29	31	33	36	38	40
27	29	31	33	36	38	40	4

(a) H.264 Intra Mode Quantization Matrix[33]

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

(b) HEVC Intra Mode Quantization Matrix[31]

16	16	16	16	16	17	18	18
16	16	16	16	17	17	18	18
16	16	16	17	18	18	18	19
16	16	17	18	19	19	20	20
16	17	18	19	20	21	21	21
17	17	18	19	21	22	22	22
18	18	18	20	21	22	22	23
18	18	19	20	21	22	23	23

(c) Adaptive Quantization Matrix [31]

Table 59: Reference quantization matrices from the video coding standards H.264 and HEVC and matrix presented in [31]

The quantization matrices used in this thesis can be seen in tables 58 and 59. The process of quantization involves dividing a given dct coefficient by the quantization coefficient at the same coordinate. The resultant value is then rounded up to the nearest integer value. The pseudo code for quantization can be seen in Listing

Listing 15: Quantize a block of DCT coefficients pseudo code

```

1  quantize(dct_coefs[8][8],quantize_matrix)
2  {
3      for (row =0; row <8;row++)
4          {
5              for(col =0; col <8; col++)
6                  {
7                      // round the result up to nearest int and store in place
8                      dct_coefs[row][col] = round( dct_coefs[row][col]
9                      / quantize_matrix[row][col] )
10                 }
11             }
12 }

```

The resultant block of quantized discrete coefficient values will generally be significantly smaller than the original values, where the majority of values will be reduced to zero. After quantization, the first compression stage happens, which is a type of run-length encoding that records the amount of zeros between the quantized coefficients that are non zero. To improve the length of the run of zeros (which improves compression) The quantized discrete coefficients will be re-ordered before processing. it is important to note that the very first coefficient of a block of discrete coefficients, known as the the DC coefficient is not included in the zero run length scanning process. The DC coefficient is the lowest frequency component of a block of 8x8 dct coefficients, It is a weighted summation of all 64 coefficients and thus, generally has a larger amplitude than the remaining coefficients, known as the ac coefficients. The DC Coefficient is removed from each block and processed by a type of delta encoding, where each subsequent DC value is stored as the difference between itself and the preceding DC coefficient.

9.3.5 Reordering Discrete Cosine Transform Coefficients

The zig zag coefficient reordering pattern that has been described in the section 6.2.3 of this thesis is taken from the JPEG standard [18]. The reordering pattern can be seen in table 60.

0	1	8	16	9	2	3	10
17	24	32	25	18	11	4	5
12	19	26	33	40	48	41	34
27	20	13	6	7	14	21	28
35	42	49	56	57	50	43	36
29	22	15	23	30	37	44	51
58	59	52	45	38	31	39	46
53	60	61	54	47	55	62	63

Table 60: Zig zag discrete cosine transform coefficient reordering pattern [18]

To reorder the values of a block of quantized discrete coefficients, the block must first be serialized from a two-dimensional structure, to a single array of 64 values, then the values are reordered according to the coordinate of the zig zag scan. The pseudo code for this process can be seen in Listing 16.

Listing 16: serialize and reorder a block of quantized dct coefficients pseudo code

```

1 reorder_coefs(block[8][8],reordered_vals[64]
2 {
3     // a temp array to hold the flatt
4     flattened_8x8[64] = {0}
5
6     // variable that increments each time a value from the block
7     // dct coeffs has been added to flattend_8x8
8     locator =0
9     for (row =0; row <8;row++)
10    {
11        for(col =0; col <8; col++)
12        {
13            flattened_8x8[locator] = block[row][col]
14        }
15    }
16
17    //Block has now been serialized, time to reorder:
18    for(i =0; i < 64; i ++)
19    {
20        // return the value of zig zag order at location i
21        // and populate reordered with its value from the
22        // serialized block of quantized coefficients
23        reordered[i] = flattened_8x8[ Zig_zag[ i ] ]
24    }
25 }

```

9.3.6 Performing Zero Run Length Encoding on Reordered Quantized Discrete Cosine Transform Coefficients

Once the block of quantized discrete cosine transform coefficients has been serialized and reordered, the zero run length processing can be performed. The algorithm stores a pair of values: the amount of zeros preceding a non zero coefficient and the value of the non zero coefficient. The scan starts from the second element in the array, as the first element is the DC coefficient and is stored separately. A special sequence is needed to indicate the end of a scan of blocks, so after the last non zero coefficient has been found, two zeros are stored, which indicates the end of the scanning process. The pseudo code for zero run length encoding can be seen in listing [17](#)

Listing 17: Zero run length encoding on reordered quantized discrete cosine transform ac coefficients

```
1 zero_run_length(reordered_vals[64], encoded_data[], dc_vals[])
2 {
3     // variable to keep count of the amount of zeros
4     // preceding a non zero coefficient
5     zero_run_length =0;
6
7     // store the dc coefficient in a separate stream
8     // to be processed separately
9     store reordered[0] in dc_vals[]
10
11    // iterate through the serialized
12    // reordered quantized dct coeffs
13    for(i =1; i < 64; i ++){
14        {
15            // if the coef at location i is zero
16            if(reordered[i] ==0)
17            {
18                //increment non zero count
19                zero_count++;
20            }
21            else
22            {
23                // a non zero coef has been found
24                // store the non zero count and the non zero value
25                // in the encoded data array
26                store zero_count in encoded_data[];
27                store reordered[i] in encoded_data[];
28                // reset the zero count
29                zero_count =0;
30            }
31        }
32        if(zero_count >0)
33        {
34            // the last non zero coefficient has been found
35            // store the special sequence of two zeros in the encoded data array
36            store 00 in encoded_data[]
37        }
38    }
```

After zero run length encoding has been performed on each block 8x8 block of sample values, the resultant bit stream is generated. Using chroma sub-sampling, one block of 16x16 3-byte pixels results in four 8x8 blocks of Y channel values, one 8x8 block of Cb channel values and one 8x8 block of Cr channel values. Each block has to follow

all of the above steps that make up the discrete cosine transform encoding method.

The compressed data is segregated based on channels. All of the Y blocks are compressed and the data is stored in one bit stream and both Cb and Cr channels are stored together, in a different bit stream. The purpose of segregation is so that an entropy encoder such as Huffman encoding can take advantage of the statistical properties of the values of each channel to generate optimum code lengths, reducing the size of the compressed data.

9.3.7 Testing Discrete Cosine Transform Based Compression on Blocks that have been Classified as Fuzzy and Picture

A test is configured to process blocks that have been classified as fuzzy and picture using discrete cosine transform based compression using different quantization matrices. The compression performance will be tested as will the objective image quality using the image quality assessment metric SSIM [50].

This test will be run first on blocks that have been classified as fuzzy, then run again for blocks that have been classified as picture.

Experimental Design

- An image from the test set of compound images is processed using the classification algorithm.
- Blocks that have been classified as the chosen classification type will be processed using discrete cosine based transform encoding.
- the blocks that are classified as the chosen classification type will be compressed fully, four times using a different quantization matrix for each run.
- The runs are (1) Reference quantization matrices for luminance and chrominance. (2) H.264 reference quantization matrix for both luminance channel and chrominance channels. (3) HEVC reference quantization matrix for luminance channel and chrominance channels. (4) Adaptive quantization matrix for luminance channel and chrominance channels.
- The compressed blocks will be decoded and inserted into the original image, such that the rest of the original image is unprocessed. The Structural Similarity index will be computed for each image, with each quantization matrix combination used.
- The test will run for every image from the test set of 40 compound images.

Average Compression Ratio vs. Quantization Matrix: Fuzzy Blocks

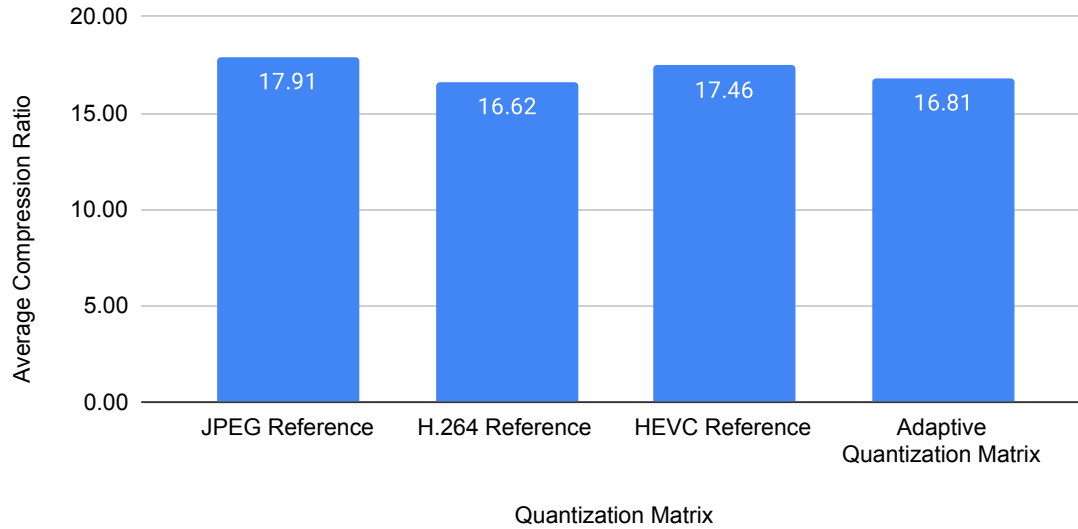


Figure 111: Results of using different quantization matrices to compress blocks classified as fuzzy

Figure 111 shows the average compression performance of each matrix when compressing blocks that have been classified as fuzzy from the test set of 40 compound images. It can be observed that each matrix produces similar results with respect to compression performance.

Quantization Matrix	Average PSNR Value of Block
JPEG Reference	42.90
H.264 Reference	42.93
HEVC Reference	42.88
Adaptive Quantization Matrix	42.89

Table 61: Results of PSNR calculated per block for each type of quantization matrix

Table 61 shows the average PSNR value computed on each 16x16 block after processing with discrete cosine transform encoding.

Quantization Matrix	Average SSIM Index Value
JPEG Reference	0.9904
H.264 Reference	0.9930
HEVC Reference	0.9926
Adaptive Quantization Matrix	0.9935

Table 62: Results of Structural Similarity Index calculation for each type of quantization matrix

Table 62 shows the Average SSIM Index value for the the images of the test set reconstructed with the processed fuzzy blocks. Values above 0.99 are considered very good. while values below 0.9 are considered acceptable, while values below 0.8 are considered poor.

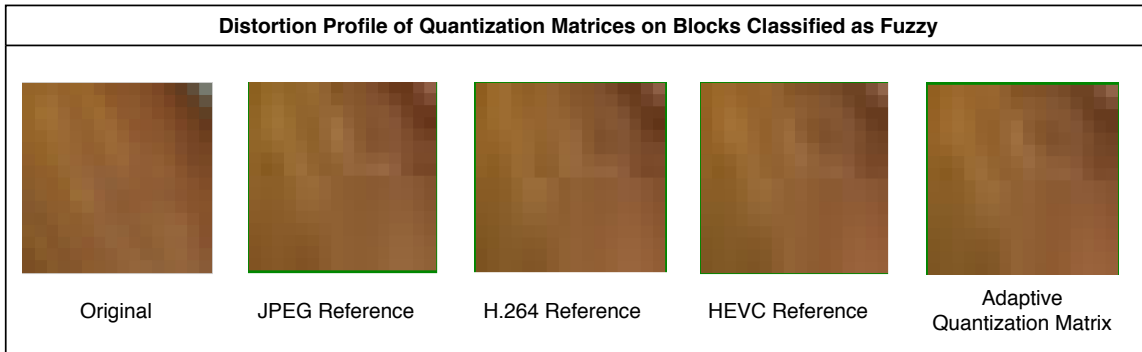


Figure 112: Comparing distortion artifacts incurred from multiple quantization matrices, fuzzy blocks

Figure 112 shows the distortion profile of the quantization matrices used. The block in figure 112 is the same 16x16 block of pixel values before and after processing, which has been expanded to give visual feedback. It can be observed that there are some slight blocking artifacts resulting from processing. The blocking artifacts are most visible when processed with the JPEG reference matrices and least visible with the HEVC Reference and Adaptive Quantization Matrix.

9.3.8 Conclusions on Fuzzy Block Quantization

It can be seen from the results shown, that each matrix performs equally well when compressing blocks that have been classified as fuzzy. The compression ratios achieved are good with respect to compression performance, which is generally greater than 15:1. The structural similarity index computation suggests that the decoded fuzzy blocks do not impede on the objective image quality of the overall image, which is a significant result, considering that chroma subsampling has been implemented.

By subjective observation, using the Adaptive Quantization Matrix [31] incurs the least amount of distortion on blocks that have been classified as fuzzy, followed by the HEVC reference quantization matrix. This agrees with the results of the SSIM computations, seen in table 62 and which an example can be seen in figure 112. However, this disagrees with the results of computing the average peak signal to noise ratio for each processed fuzzy block, shown in table 61. This shows the relevance of using an objective image quality assessment metric such as Structural Similarity, when optimising for image quality. The compression performance using the HEVC reference quantization matrix is slightly better, approx 3 percent on average, while the SSIM index value for the Adaptive Quantization matrix is better.

Average Compression Ratio vs. Quantization Matrix: Picture Blocks

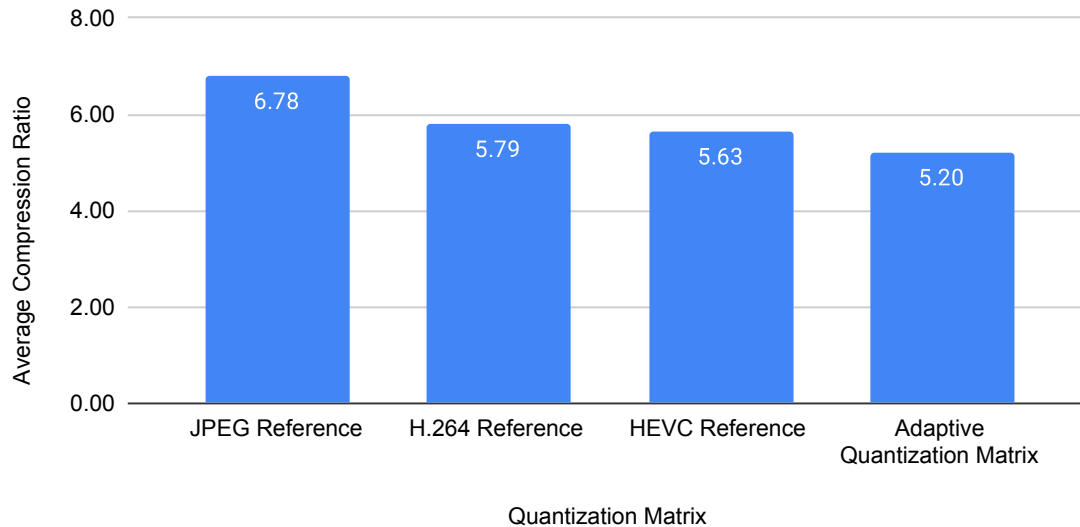


Figure 113: Results of using different quantization matrices to compress blocks classified as picture

Figure 113 shows the average compression performance of each matrix when compressing blocks that have been classified as picture from the test set of 40 compound images. It can be observed that each matrix produces similar results with respect to compression performance and that the compression performance is significantly lower than blocks that have been classified as fuzzy. This is as expected, as blocks that are classified as picture may contain more structure and a higher gradient of change within a given block.

Quantization Matrix	Average SSIM Index Value
JPEG Reference	0.9791
H.264 Reference	0.9870
HEVC Reference	0.9878
Adaptive Quantization Matrix	0.9904

Table 63: Results of Structural Similarity Index calculation for each type of quantization matrix

Table 62 shows the Average SSIM Index value for the the images of the test set reconstructed with the processed picture blocks. Values above 0.99 are considered very good. while values below 0.9 are considered acceptable, while values below 0.8 are considered poor.

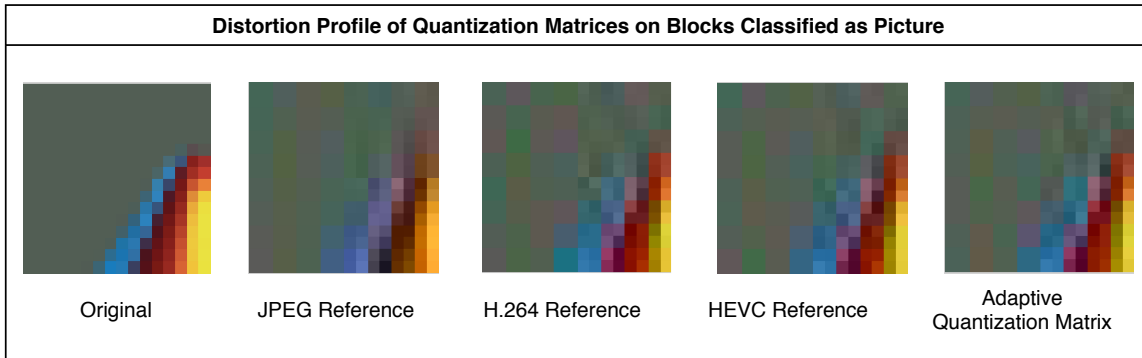


Figure 114: Comparing distortion artifacts incurred from multiple quantization matrices, picture blocks

Figure 114 shows the distortion profile of the quantization matrices used. The block in figure 112. It can be observed that there are severe compression artifacts visible in all cases. The Adaptive Quantization matrix, subjectively has the lowest amount of distortion, however it also has the lowest compression performance.

The block from 114 was specifically chosen to highlight distortion artifacts as a result of compressing using a discrete cosine transform and quantization. It contains computer generated image data, which is complex in structure and has a high unique pixel count. In this case, it is important to choose the quantization matrix that introduces the least amount of compression artifacts.

9.3.9 Conclusions on Picture Block Quantization

It is clear from the compression performance results that blocks that are classified as picture are harder to compress. Further, the choice of quantization used can negatively impact the subjective quality of the decoded image when the image data is

highly structured, an example of which has been shown in figure 114. However, The average SSIM index value computed for the test set of images show that as a whole, the objective image quality of the decoded images after processing is still good. A subjective view of the processed images shows the distortion introduced from discrete cosine transform and quantization is highly masked for blocks that actually contain natural continuous tone image, while some compression artifacts can be seen on blocks that contain some computer generated image data.

When using lossy style compression, there will always be a trade-off between compression performance and processed image quality. In the case of blocks that have been classified as picture, it may be of greater benefit to optimise for image quality by using the quantization matrix that introduces the least amount of distortion.

9.3.10 Entropy Encoding Discrete Cosine Transform Based Compressed Data

To reduce the coding redundancy in the discrete cosine transform based compressed bit stream, the Huffman encoding algorithm from the zlib library [8] is used. To optimise compression, the compressed fuzzy block data is processed independantly to the compressed picture block data. Further, The dc component data is independently processed from the ac component data. This is the same approach as JPEG [18]

Before Huffman encoding on the dc component bit stream, the data is processed with delta encoding which stores consecutive values as a difference value from the preceding values, this reduces the dynamic range of values, improving compression performance. The pseudo code for delta encoding can be seen in listing

Listing 18: Delta encoding pseudo code

```
1 delta_encode(dc_components[])
2 {
3     // variable to keep track of the comparison
4     last = 0;
5     // iterate to the end of the array
6     for (i = 0; i < dc_components.length; i++)
7     {
8         current = dc_components[i];
9         // subtract the current value from the last value, store the difference
10        dc_components[i] = current - last;
11        // increment the comparing location
12        last = current;
13    }
14 }
```

Average Compression Ratio vs. Quantization Matrix: Huffman Encoded Fuzzy Blocks

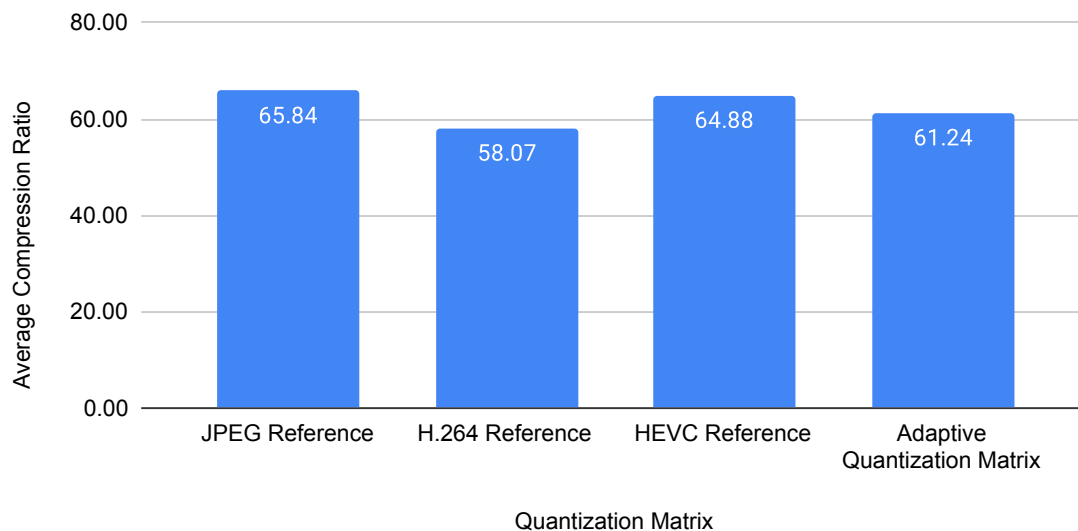


Figure 115: Compression performance of Huffman encoded compressed fuzzy block data

Average Compression Ratio vs. Quantization Matrix: Huffman encoded Picture Blocks

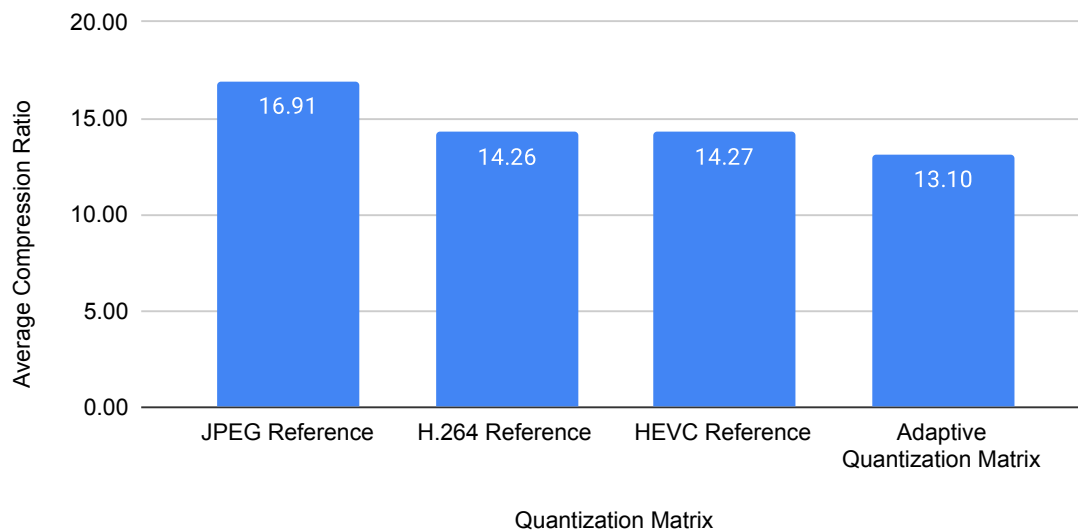


Figure 116: Compression performance of Huffman encoded compressed picture block data

Figure 115 and figure 116 show the compression performance of encoding fuzzy and picture data using Huffman encoding. It can be seen that entropy encoding significantly improves overall compression performance for discrete cosine transform based compression, particularly for fuzzy compressed data.

9.3.11 Discrete Wavelet Transform based Compression: Blocks Classified as Text with High Unique Pixel Count

This section presents the discrete Haar wavelet transform with the use of thresholding for compression of blocks that have been classified as text that have a high unique pixel count. This type of block may not be suitably compressed using a lossless compression algorithm, with respect to compression performance. Further, the highly structured information in this type of block can not be easily compacted into a small number of low frequency discrete cosine transform coefficients. Thus, using a compression strategy based on a discrete cosine transform is inefficient in terms of compression performance and also will result in distortion artifacts in the decoded processed image.

Discrete Haar wavelet transform based compression will be compared to discrete cosine transform based compression and Deflate compression for blocks classified as text with pixel count greater than 31 unique pixel values.

The comparison is made by:

- Compression performance.
- Speed performance.
- Subjective and objective image quality.

The discrete Haar wavelet transform functions with respect to compression have been described in section 6.2.6 in listings 6. The resultant transform coefficients can be positive and negative floating point values.

The first test will compare two implementations of the discrete Haar wavelet transform. The first method uses floating point computation, while the second method uses integer computation. To use integer computation, the weighted values of s_0 , s_1 , d_0 , d_1 shown in listing 6 are shifted right 10 bits from their original values of 0.5, 0.5, 0.5, -0.5, so that the calculations can be performed with integer values. After computing the discrete wavelet coefficients, each value is then left shifted by 10 bits to negate the right shift.

Experimental Design

- 400 16x16 3-byte pixel blocks from a compound image are selected.

- The forward discrete Haar wavelet transform using floating point computation is applied to each channel of the block. No thresholding is applied.
- The inverse discrete Haar wavelet transform using floating point computation is applied to each channel of the block. resulting values are rounded to nearest integer values.
- The PSNR is computed on the process block, using the unprocessed block as a reference.
- The amount of blocks recovered with no error is recorded.
- The PSNR value is calculated for the set.
- The above steps are repeated for the Integer implementation of the forward and inverse discrete Haar wavelet transform, using Integer computation.

Table 64 shows the result of the experiment. It can be observed that using floating point arithmetic has greater accuracy when computing the discrete Haar wavelet transform. However, with an average PSNR value above 90dB, both implementations are approximately lossless and any error would not be visible to the human visual system. This is confirmed by subjective analysis of the processed blocks.

Computation Type	Recovered Blocks No Error	Average PSNR (dB)
Floating	67	98dB
Integer	0	92dB

Table 64: Comparing accuracy of floating point and integer implementation of the discrete Haar Wavelet Transform

To compare the performance of computation time between both methods of forward Haar wavelet transform, the experiment is run 10 times, with the average time recorded for both method. The results can be seen in table. 65

Computation Type	time (μS)
Floating	9085
Integer	2004

Table 65: Comparing the time taken to perform the forward discrete Haar wavelet transform on 400 Blocks of 16x16 3-byte pixels using floating point and integer calculation

It can be observed from the results in table 65 that using integer calculation is significantly faster than using floating point calculation to compute the discrete Haar wavelet transform. This is a significant result, by reducing the time taken to perform the transform, will reduce the overall processing time for the final compound compression algorithm.

9.3.12 Comments on Accuracy and Speed of Discrete Haar Wavelet Transform

Implementing the discrete Haar wavelet transform using integer computation significantly improves the performance of the algorithm with respect to speed, with the side effect of reducing the overall accuracy of the transform. However by subjectively viewing the blocks before and after processing, the distortion is imperceptible to the human eye. As always, there is a trade off when trying to optimise for speed, performance or quality, in this case, however, the improvements in speed performance outweigh the reduced amount of objective quality.

9.3.13 Threshold Value Selection for Discrete Haar Wavelet Transform Compression

To discard some of the irrelevant data in favour of better compression, using a form of quantization called thresholding is used on the discrete Haar wavelet coefficients. The act of thresholding is to use a set value and compare a discrete wavelet coefficient to it: if the coefficient is below the threshold value, the coefficient is set to zero. The algorithm for thresholding is shown in listing 7

A staggered thresholding approach is used in this thesis. As a two level discrete Haar transform is used, a different threshold value is used for each level. The wavelet coefficients of the second level of the transform have a higher weighting than the coefficients from the first level of the transform, thus using a smaller threshold value for the second stage should reduce the amount of error in the processed data.

Chowdhury, et al in their work presented on "compression using discrete wavelet transform" [26] suggests the threshold value of 30 as a good trade off between compression performance and decoded image quality. This suggested value is used as a base value for testing.

In section 6.2.6 of this thesis describing the discrete wavelet transform for compression, it was posited that, unlike discrete cosine transformation, the discrete wavelet coefficients can not be reordered in a suitable ordering as to optimise zero run-length encoding. This is because the location of the discrete wavelet transform coefficients is highly dependent on the input. This is primarily the reason for using the discrete wavelet transform for processing blocks containing high transient information, the

transient information is localised in the sub bands, which can be preserved once the transient magnitude is greater than the threshold values. It has been shown that after performing the discrete wavelet transform and thresholding, there are many coefficients that are set to zero. Two methods are proposed in this thesis to encode the non zero coefficients. The first method is to use a zero run length encoding, the same as used in discrete cosine transform encoding, which records the number of zeros before the next non zero coefficient. The second method is a novel method which is presented in this thesis.

9.3.14 Encoding Non Zero Discrete Wavelet Transform Coefficients After Thresholding

The second method involves creating an index map of the non zero coefficients after thresholding and stores that first, followed by the payload of the non zero coefficient values. The index map starts with a two byte header, where each bit of the header represents a row of the 16x16 block of values after thresholding. If a row contains all zeros, the corresponding bit of the two byte header is set to zero. After the two byte header, there is a following two byte pair for every row that contains a non zero value. The column in a row that contains a non zero value is indicated by a bit of the corresponding two byte pair and is set to 1. To illustrate the process of generating an index map to encode the non-zero DWT coefficients, consider the block of transformed and thresholded coefficients in table 66

Table 22 shows the resultant values after thresholding the values of table 21

8	15	0	0	-12	15	0	0	0	0	0	9	0	0	0	0
0	11	0	0	0	11	0	0	0	0	0	0	0	0	0	0
13	22	0	0	0	8	0	0	0	0	0	0	0	0	0	0
0	4	0	0	0	-4	0	0	0	0	0	0	0	0	0	0
8	-8	0	0	-12	-10	0	0	-12	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	-9	0	0	-7	10	0	0	0	0	0	0	0	0	0	0
0	8	0	0	0	-8	8	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-11	0	0	0	0	0	0	0	0	0	0	10	0	0	20	0
13	0	0	0	0	0	0	0	0	0	0	10	0	0	20	0
0	0	0	16	16	0	0	0	0	0	0	10	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 66: After two level forward discrete Haar wavelet transform with thresholding

The two byte header to describe the rows with non-zero coefficients for table 66 would be:

1111 1011	1000 1110
-----------	-----------

The two bytes to describe where the non zero coefficients in the first row would be:

1100 1100	0001 0000
-----------	-----------

There is no need store two bytes for rows that do not contain any non zero coefficients, as the decoder will know how many rows contain non zero coefficients from the two byte header and subsequently from the following index map that is stored. The implementation in C++ can be seen in the code section of this thesis in listing 28.

A series of tests to find efficient threshold values to use in discrete wavelet transform encoding have been implemented. A threshold value is selected for each level of the two levels of the discrete wavelet transform. Also, different combinations of threshold values for the luminance channel and the chrominance channels are tested. A sample of the combination of threshold values used in testing can be seen in table 67

Channel	Level one DWT T Val	Level Two DWT T Val	Channel	Level One DWT T Val	Level Two DWT T Val
Y	10	05	Cb Cr	10	05
Y	10	05	Cb Cr	20	10
Y	15	10	Cb Cr	20	10
Y	20	10	Cb Cr	30	20
Y	30	15	Cb Cr	40	20

Table 67: A Set of Threshold value combinations used in testing the discrete wavelet transform

After thresholding, the discrete wavelet transform coefficients are compressed using two different methods: method one is the zero run length encoding similar to the implementation used for discrete cosine transform, without reordering the coefficients. the second method is the novel approach presented in this thesis which is called non zero index map encoding.

Figure 117 shows the average compression ratio achieved using a selection of threshold value combinations The threshold values in figure 117 are described in table 67

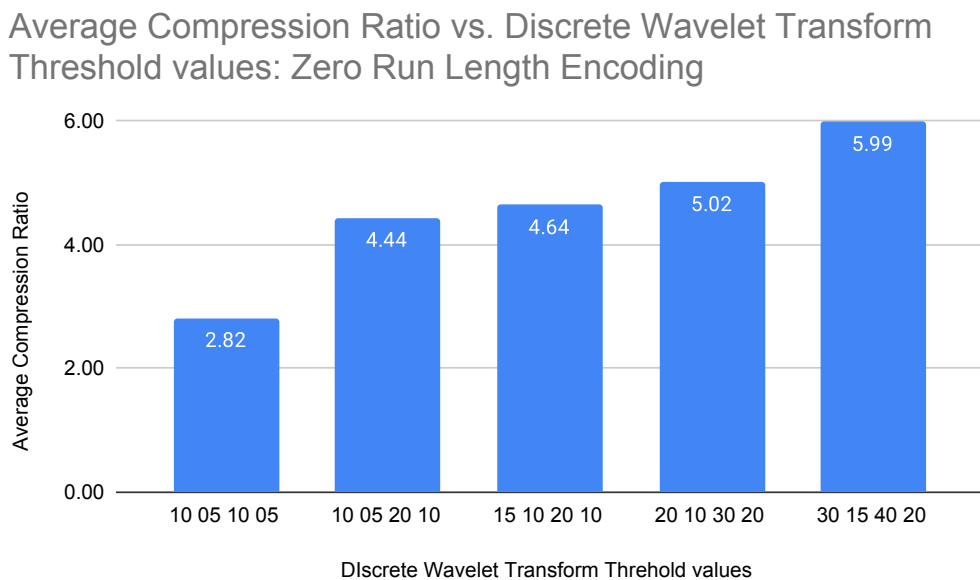


Figure 117: Average Compression Ratio Using Different Threshold Value Combinations with Zero Run Length Encoding

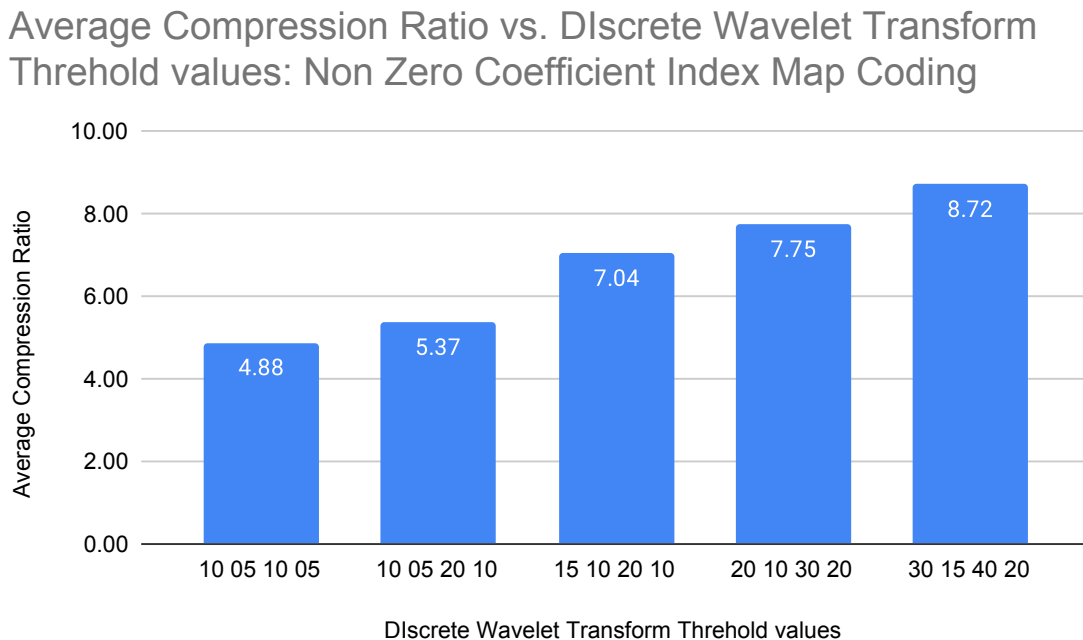


Figure 118: Average Compression Ratio Using Different Threshold Value Combinations with Non Zero Index Map Encoding

Figure 118 shows the average compression ratio achieved using a selection of threshold value combinations with non zero index map coding. It is clear from the comparing

the results from figure 117 and 118 that using the novel non zero coefficient index map coding performs better.

Figure 119 shows the average peak signal to noise ratio computed for each block that has been processed with the discrete wavelet transform and combinations of threshold values. It can be seen that as the threshold value increases, so does the error in the signal.

Average PSNR Block Value vs. Discrete Wavelet Transform Threshold values

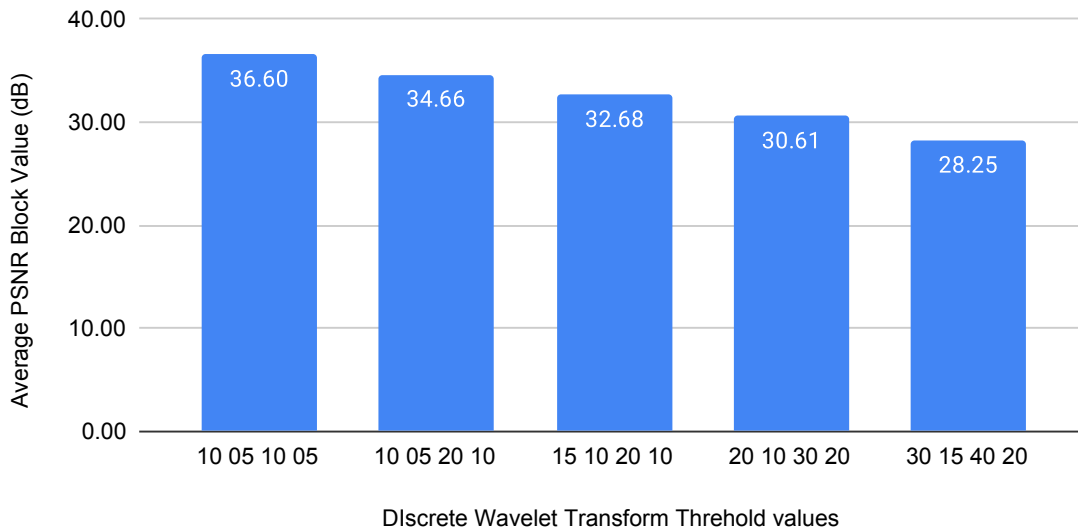


Figure 119: Average PSNR Value calculated per Block for combinations of threshold values

Compression Algorithm	Compression Ratio	PSNR (dB)
Discrete Wavelet Transform	7.04	34.68
Discrete Cosine Transform	3.9	31.72
Deflate	2.37	Lossless

Table 68: Comparing Discrete Wavelet Transform Compression to Discrete Cosine Transform Compression and Deflate Compression

Table 68 compares the average compression performance of the discrete wavelet transform encoding against discrete cosine transform encoding and deflate encoding. The non zero index map method is used along with threshold values 15 10 20 10 for the discrete wavelet encoding, The HEVC reference quantization method is used for the

discrete cosine transform and the default compression method is used for deflate. It can be observed that the discrete wavelet transform encoding method performs better, with respect to compression ratio than both discrete cosine transform and deflate. The average psnr values is similar for both discrete wavelet transform and discrete cosine transform encoding, however it can be seen in figure 120 that the distortion incurred from both methods of processing is very different and, subjectively, the distortion caused by the discrete wavelet transform and thresholding is less severe than that caused by the discrete cosine transform and quantisation.

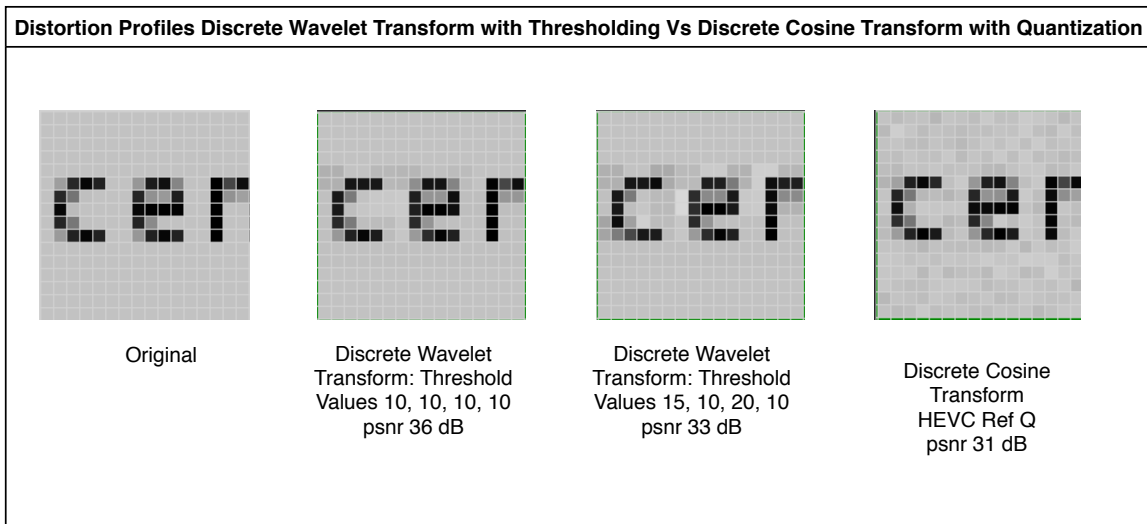


Figure 120: Distortion profile of Discrete Wavelet Transform with thresholding compared to Discrete Cosine Transform on a block classified as text

To reduce the code redundancy after encoding with discrete wavelet transform encoding and thresholding, Huffman encoding is used as the entropy encoder. Figure 121 shows the average compression performance of implementing Huffman entropy encoding with the discrete wavelet transform encoding previously described. The results show that the combination of discrete wavelet transform, thresholding, non zero index map coding and Huffman entropy encoding produce very efficient results in terms of both compression performance and subjective image quality

Average Compression Ratio vs. Discrete Wavelet Transform Threshold values: With Huffman Entropy Encoding

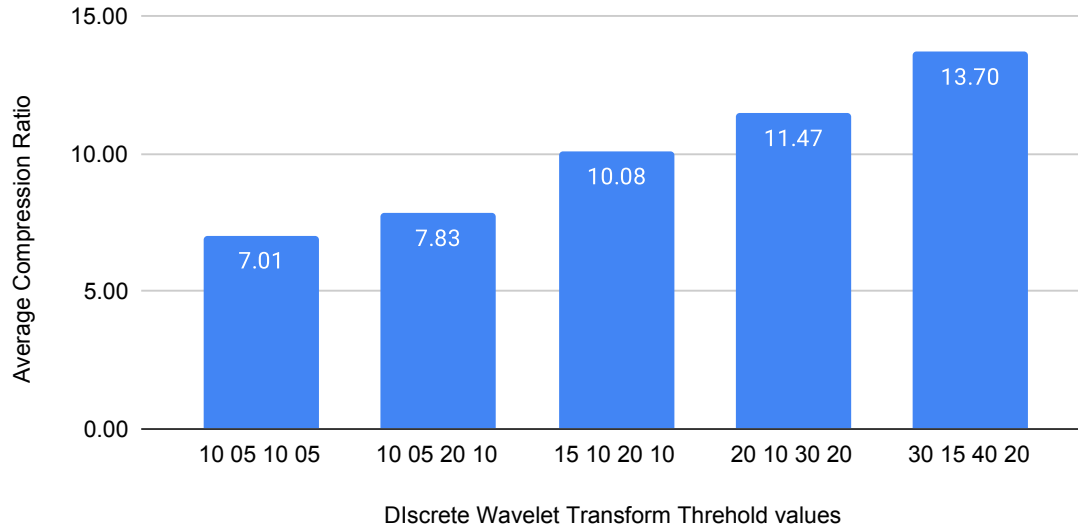


Figure 121: Compression performance of the discrete wavelet transform using combinations of threshold values, non zero index map coding and Huffman entropy encoding

9.3.15 Conclusions on Lossy Compression Testing

In this section of the thesis, lossy transform based encoding has been described and tested on blocks that contain continuous tone image and also blocks that contain computer generated data with a high unique pixel count.

It has been shown that blocks that have been classified as fuzzy are easily compressed using the discrete cosine based transform with any of the quantization matrices discussed. Both the subjective and objective image quality of decoded fuzzy blocks is of acceptable to good quality, with most of the compression artifacts being masked. This suggests optimising the compression of fuzzy blocks for the best compression performance.

However, blocks that have been classified as picture may contain structural elements and are susceptible to compression artifacts which may be noticed by the end user. In this case, it would be best to optimise for objective image quality over compression performance. Moreover, the gains in compression performance using different quantization matrices are not large enough to sacrifice the overall image quality.

The discrete wavelet transform has been presented in this section for compressing blocks that have been classified as text but have a high unique pixel count. It has been shown that compressing these types of blocks with a lossless compression algorithm yields poor compression performance. Using a discrete cosine transform based

compression yields acceptable compression performance but reduces both the subjective and objective image quality of the decoded image. Using the discrete wavelet transform with a combination of the novel approach of compacting the non zero coefficients using an index map has been demonstrated to be efficient, both in terms of compression performance, objective and subjective image quality.

Using combinations of threshold values, the discrete wavelet transform can be optimised for compression performance or for objective image quality. It is very important to reduce the amount of compression artifacts in the decoded image, especially for text data. Many combinations of threshold values were tested and a sample of threshold value and their compression performance has been demonstrated. Striking a balance between compression performance and image quality, a suitable combination of threshold values used in this thesis can be seen in table 69

Channel	Level one DWT T Val	Level Two DWT T Val	Channel	Level One DWT T Val	Level Two DWT T Val
Y	15	10	Cb Cr	20	10

Table 69: A suggested set of threshold values for Discrete Wavelet Transform based compression

The discrete wavelet function that is used in this thesis is the Haar wavelet function. It was chosen because of its ability to extract structural information from a block of pixels by using an average and difference function. The preservation of structural information within blocks of pixels by thresholding relies on a given block having structural information such as lines and sharp transitions in pixel intensity values. Performing the discrete Haar wavelet transform on blocks without structural information, such as continuous tone image will introduce blocking artifacts, which is why the discrete Haar wavelet transform has not been chosen to process picture and fuzzy blocks in this thesis.

9.4 Full Compound Compression Algorithm Configuration

Previously in this section, lossless and lossy compression algorithms have been described and tested on the different types of classified data within a compound image. This section presents the results of combining the compression algorithms that are optimal for the specific type of classified data into a single compound compression algorithm which chooses a compression algorithm that is best suited to the attribute within a given block of pixels.

Blocks that have been classified as smooth will be compressed with the novel smooth compression algorithm presented in this thesis which shows exceptional compression performance.

The novel lossless compression algorithm, Differential Index Map Coding has been shown to achieve the best compression and speed performance on blocks classified as sparse and blocks classified as text that have less than 32 unique pixel values. For the remaining blocks that have been classified as text, The discrete wavelet transform coding has been shown to have the best compression performance and subjective and objective quality, compared to the discrete cosine transform encoding. However, It is a lossy compression algorithm, which presents a choice to optimise for compression performance or optimise for quality. If the algorithm is optimised for image quality, the Deflate compression algorithm can be used to compress blocks that are classified as text with a pixel count greater than 31. This comes at a reduction on compression performance but an increase in decoded image quality.

Discrete cosine transform coding is implemented for blocks that have been classified as fuzzy and picture. Blocks that have been classified as fuzzy will be quantized stronger than those that have been classified as picture.

When all blocks have been compressed with the chosen compression algorithm, the encoded data from each compression algorithm will be entropy encoded using Huffman encoding provided by the zlib library [8]. This results in the compressed data payload. The process flow for the compound compression algorithm (CCA) can be seen in figure

122

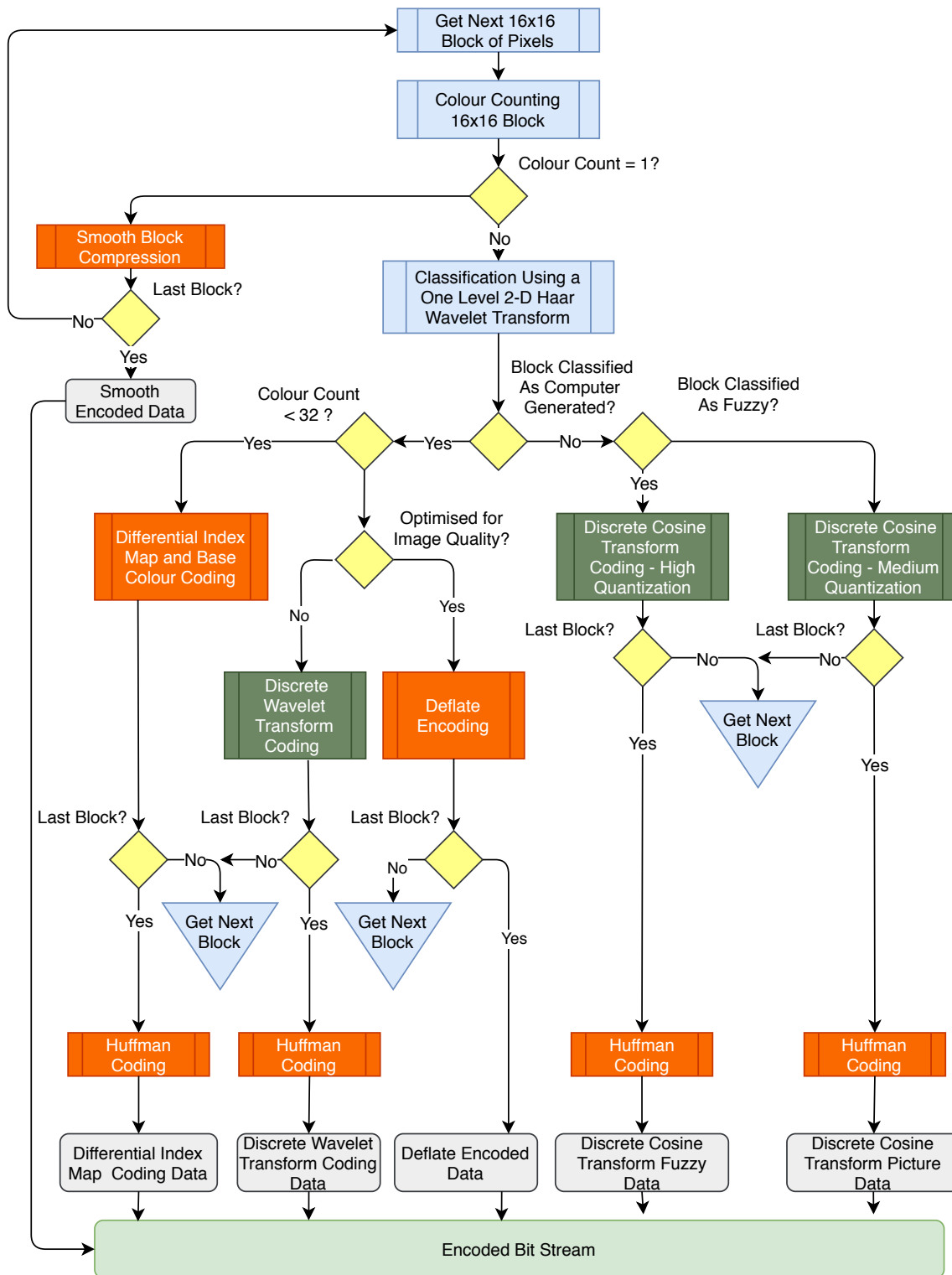


Figure 122: Compound compression algorithm

A series of tests are undertaken to compare the final compound compression algorithm presented so far in this thesis against three compression standards: JPEG [18], H.264 [47] and HEVC [39]. JPEG is an image compression standard, while H.264 and HEVC are video coding standards. To test H.264 and HEVC a given image will be encoded as a single frame of a video which means it will be encoded as an Intra mode - or "I" frame.

40 compound images will be compressed by each algorithm and the results will be presented. The compound images used for testing can be seen in the appendix of this thesis in figures 134 - 137.

JPEG compression allows for a parameter known as QF (Quality Factor) to be modified, which can increase compression performance at the expense of image quality and vice versa.

JPEG compression is implemented using "OpenCV" open source computer vision software library [28]. To test the compression and speed performance of JPEG compression, different values of QF are used. The QF variable can be a value from 0-100. Values below 50 incur severe distortion and compression artifacts, so they are omitted from this test. QF values from 50 - 90 inclusive, incremented by 10 are tested. The implementation in c++ can be seen in the code section of this thesis in listing 29

To encode an image as a single frame using H.264 and HEVC, an open source video encoding framework called FFMPEG [12] is used. FFMPEG implements the official open source distributions of x264 (H.264) and x265(HEVC) under GPL licence.

Similar to JPEG, a parameter called "CRF" (constant rate frame) can be used to control the quality vs compression performance for both H.264 and HEVC. CRF can be a value between 0-50, where 0 is the highest quality and 50 is the worst. A value of 10 is good to excellent and a value of 20 is acceptable. Values below 20 incur severe distortion and compression artefacts, so are omitted from testing.

The compound compression algorithm,"CCA" presented in this thesis is configured in two modes during this test. When optimised for compression performance, Differential Index Map Coding is used for all blocks that are classified as sparse. Blocks that are classified as text with up to 31 unique pixel values are also compressed using Differential Index Map Coding. Blocks that are classified as text with a pixel count greater than 32 are compressed with discrete wavelet transform coding. Blocks that are classified as fuzzy are compressed using discrete cosine transform encoding using JPEG reference quantization matrix. Blocks classified as picture are compressed with discrete cosine transform coding using HEVC reference quantization matrix. The resultant encoded data from each compression algorithm is compressed independantly using Huffman encoding implemented using the zlib [8] library.

When "CCA" is optimised for image quality, blocks that are classified as text with a

unique pixel count greater than 31 are compressed with the Deflate algorithm. Blocks that are classified as fuzzy use the HEVC quantization matrix and blocks that are classified as picture use the AQM [31] matrix.

9.5 Compound Compression Algorithm Results

Table 71 tabulates the average results per image of the test. The test compares compression performance, objective and subjective image quality and speed performance on a set of 40 compound images. The compression algorithms tested are: "CCA" (compound compression algorithm), JPEG, H.264 and HEVC.

To accurately measure the speed performance, the suite of tests were run 10 times in blocks and randomised. The value for time was then averaged over each set.

By viewing samples of the test images after processing, a scale has been developed to associate the subjective image quality of processed compound images with SSIM index value. The scale is described in table: 70

SSIM Index	Label	Description
1	Lossless	Full replication of original image, free from error
0.990 - 0.999	A.Lossless	Approximately lossless, distortion imperceptible to human visual system
0.980 - 0.989	Excellent	Excellent visual quality, text legible distortion artifacts only noticeable if one knows what to look for
0.970 - 0.979	Acceptable	visible signs of blurring of text and visible distortion artifacts
0.960 - 0.969	Passable	Blurred text and signs of blocking artifacts
0.900 - 0.959	Poor	Text not legible, blocking artifacts.
0 - 0.899	NA	Not suitable for compound image compression

Table 70: SSIM Index value scale vs subjective image quality

Codec	Comp Ratio	PSNR (dB)	SSIM Index	Comp Time (μS)	Decomp Time (μS)	Subjective Quality
CCA Optimise Compression	42.27	40.88	0.983	42,285	16,465	Excellent
CCA Optimise Quality	34.39	42.63	0.991	39,423	15,511	A.Lossless
JPEG QF = 50	32.17	31.38	0.943	37,108	18,402	Poor
JPEG QF = 60	28.99	32.32	0.949	38,871	18,561	Poor
JPEG QF = 70	25.31	33.73	0.952	36,424	18,351	Poor
JPEG QF = 80	21.17	35.53	0.968	39,387	22,207	Acceptable
JPEG QF = 90	15.71	38.12	0.980	41,357	26,651	Excellent
H.264 CRF = 20	49.58	33.71	0.971	374,236	32,045	Acceptable
H.264 CRF = 15	36.15	34.71	0.9791	371,898	40,613	Acceptable
H.264 CRF = 10	27.13	35.71	0.981	480,674	31,020	Excellent
HEVC CRF = 20	46.78	34.51	0.978	386,871	31,045	Acceptable
HEVC CRF = 15	34.63	35.33	0.981	396,011	32,987	Excellent
HEVC CRF = 10	25.85	35.79	0.988	464,674	39,574	Excellent

Table 71: Compound Compression Algorithm vs JPEG vs H.264 vs HEVC

9.6 Discussion

The results shown in table 71 are significant. With respect to compression performance, the compound compression algorithm, "CCA", presented in this thesis significantly out performs JPEG at all quality levels that have been tested. Moreover, it significantly out performs both H.264 and HEVC when both are optimised for excellent quality.

Of possible greater significance, is that the compound compression algorithm presented has the highest objective image quality scores of all iterations of the test compression frameworks. In particular, when the compound compression algorithm is optimised for image quality, both the objective results of SSIM Index value and PSNR are extremely high, with respect to the compressed file size. When subjectively viewing the decompressed images, there is very little to no compression artifacts around areas of the screen containing text. When optimised for compression performance, there are some mild blocking artifacts that are noticeable when there are areas of natural continuous tone image on the boundary of computer generated data. Blocking artifacts are an inherent problem when using a block based compression algorithm and can be even more exaggerated when using a combination of both lossless and lossy compression algorithms in the same image. However, when subjectively viewing

the image, the blocking artifacts are only noticeable when one knows what to look for. In other words, its a trained bias that brings the artifacts to the fore.

With respect to speed performance, the compound compression algorithm is very close to JPEG. However, it can be considered that the decompression time holds greater significance in a thin client system or virtual desktop infrastructure, when the client side has less processing power than the server. To this end, It can be seen that the decompression times for both modes of the compound compression algorithm out perform JPEG and greatly out perform both H.264 and HEVC.

A significant contributing factor to the loss of compression performance when using discrete cosine transform coding in the compound compression algorithm presented in this thesis is that a generic Huffman entropy coder is used. The Huffman entropy encoder used is not as optimised for the encoded discrete cosine transform data as the canonical Huffman coding or arithmetic coding that is used in JPEG. Both H.264 and HEVC use a patented entropy encoder called Context Adaptive Binary Arithmetic Coding (CABAC) which is highly optimised for discrete cosine transform data.

The choice to use a standard generic Huffman encoder implemented using the zlib [8] library for entropy encoding was made for multiple reasons, three of the main reason are:

- There are multiple types of compression algorithms used in the compound compression algorithm. The compressed data resulting from each compression algorithm have very different attributes. Optimising an entropy encoder for one specific compressed data type may not be optimal for another.
- The Huffman entropy encoder from the zlib library is free from license restrictions, allowing it to be used freely without penalty.
- The Huffman entropy encoder from zlib library is both efficient, lightweight and very fast, which means it would be suitable for devices with low computational power

For future work, optimising the entropy encoder for the compound compression algorithm could enhance the compression performance of the compound compression algorithm presented in this thesis.

10 Results Summary

Compound Image Classification

In section 9.1.1 (page 142), the compound image classification algorithm presented in this thesis is compared to the algorithm presented in the work by Wu [48]. Table 37 (page 143), from section 9.1.1 shows that the classification time presented in this thesis is at least twice as fast as the algorithm presented by Wu [48], which is a significant improvement.

There is also a significant improvement in classification accuracy by the algorithm presented in this thesis compared to Wu [48], which can be subjectively viewed in figures 123 - 133, in the appendix of this thesis (page 256 - 266).

Lossless Compression

The novel compression algorithm "Differential Index Map Coding" (DIMC) developed in this thesis in section 9.2.5 (page 167), is compared to the industry standard compression algorithm Deflate [7] in section 9.2.7 (page 184 - 187). With respect to compression performance, it is shown that DIMC significantly out performs Deflate's best compression setting for compound image data that contains low pixel count such as sparse blocks and text blocks. This is shown in figures 107 and 108 (page 184 - 185).

With respect to speed performance, DIMC significantly out performs Deflate, even when Deflate is configured for best speed performance. Of particular significance is DIMC out performs Deflate when configured for best compression. It processes the data approximately 20 times faster. This can be seen in figure 110 (page 187)

Full Compound Compression Algorithm

The full compound compression algorithm (CCA) described in section 9.4 (page 218), is compared to industry standard compression algorithms JPEG [18] and H.264 [47], as well as the state of the art HEVC [39]. The results, which are tabulated in table 71 (page 221), show that CCA achieves the highest objective and subjective image quality score, which is a significant result.

With respect to compression performance, the CCA algorithm achieves significantly higher compression ratios compared to the other compression algorithms tested, when the objective and subjective image quality is similar. Using the subjective quality metrics described in table 70 (page 220), The results in table 71 (page 221), show that CCA with a subjective quality of "Excellent" achieves on average, compression ratios of over [42:1]. This is compared to JPEG which achieves over [15:1], H.264 which achieves over [27:1] and HEVC which achieves over [34:1].

With respect to speed performance, CCA significantly out performs H.264 and HEVC in compression time. The results in table 71 (page 221), show that CCA is on average, up to 10 times faster at compressing compared to H.264 and HEVC.

With respect to decompression time, the results in table 71 (page 221), show that CCA has the fastest decompression time of all algorithms tested. Compared to H.264 and HEVC, the decompression time is over twice as fast.

11 Conclusion

The goal of the work presented in this thesis is to research and implement a strategy to compress the highly complex attributes of a compound image. To do this, one must have a deep understanding of the characteristics of a compound image and the nature of digital imagery and the human visual system. By analysing the different attributes of a compound image and separating them into defined classes of data type, one can start to design a system that is capable of compressing each data type optimally.

Some of the most immediate attributes of the classes of data include:

- Elements of a graphical user interface that are smooth.
- Sharp transients in text data.
- Highly structured information for computer generated data such as sparse blocks.
- Non discernible slowly varying pixel value variations for blocks of continuous tone image that can be considered fuzzy.

These attributes have been well researched in the area of compound image compression, but to the authors knowledge, little has been done on an overlooked attribute of a compound image and that is the unique pixel count of blocks of pixels that contain computer generated data.

From analysis on the amount of unique pixel values in a each block of a compound image, it has been shown that there is a very strong correlation between unique pixel count and computer generated data. This simple and overlooked attribute has lead to the development of a very efficient novel lossless compression algorithm called Differential Index Map Coding which has been presented in this thesis and tested against industry standard compression algorithms.

The compound compression algorithm presented in this thesis is highly capable at classifying the data types within a compound image. Moreover, it is a block based strategy, which lends itself well to work with many "off the shelf" compression algorithm, which leaves rooms for continuous improvement. Being a block based compression algorithm, it is modular by nature. This is a desirable attribute as the different compression algorithms may be implemented in parallel, which may improve on compression and decompression time.

Image quality is inherently subjective. However, it is important to be able to quantify objectively the image quality of a decoded image to help in development and to control data rate. Legacy metrics such as Root Mean Square Error (RMSE) and Peak Signal to Noise Ratio (PSNR) are a good indication of quality, but do not give the whole story. Modern image quality metrics such as Structural Similarity Index (SSIM) help give confidence in the design of the compound compression algorithm presented in

this thesis, as the results of compression have both a high PSNR value and SSIM index value.

The compound compression algorithm has been tested on a set of high quality test images and has out performed state of the art video codecs in terms of compression performance and objective image quality. A significant amount of research in the area of compound compression involves implementing new modes for industry standard, state of the art video codecs such as H.264 and HEVC. This can be seen in the research presented in [23] [9] [10] [4] [31]. However, implementing these video codecs in a product that would be used commercially involves restrictive licensing criteria, which reduces the scope of any potential project.

11.1 Future Work

The compound compression algorithm presented in this thesis could be considered as the basis for real time video encoding strategy for video containing high amounts of compound image, such as virtual desktop infrastructure, cloud computing and thin client infrastructure. As video is temporal by nature, the next stage would be how to compress successive frames of images efficiently. There is typically quite an amount of temporal redundancy at a frame rate of 25- 30 frames per second. Much of the screen may stay the same from frame to frame. A good step might be to leverage the inter frame redundancy by implementing residual coding similar to the approach used in other coding standards such as H.264 and HEVC.

With respect to the algorithms that make up the compound compression algorithm, The first stage in future work would be to research on improving the entropy encoding technique. As stated previously, H.264 and HEVC use a patented entropy encoder known as "Context Adaptive Binary Arithmetic Coding", which is an Arithmetic coder. Research presented by Duda [11] describes a new form of entropy coding known as Asymmetrical Numeric Systems which leverage's the speed of Huffman coding with the performance of Arithmetic coding. This would be very beneficial for a real time codec which must operate in a low bandwidth environment.

12 Code

12.1 Data Structures

This sections will describe the data structures and defined variables that are used in the code. As the code has has been implemented using object orientated programming, the data structures used have been defined mainly using classes instead of structs.

Listing 19: PIXEL Class, C++

```

1  class PIXEL
2  {
3  public:
4      uint8_t channel0; //Blue in RGB colour space, Y in YCbCr colour space
5      uint8_t channel1; //Green in RGB colour space, Cr in YCbCr colour space
6      uint8_t channel2; //GRed in RGB colour space, Cb in YCbCr colour space
7      bool friend operator==(const PIXEL &p1, const PIXEL &p2)
8      {
9          if((p1.channel0 == p2.channel0)
10         && (p1.channel1 == p2.channel1)
11         && (p1.channel2 == p2.channel2))
12             return true;
13         else
14             return false;
15     };
16 };

```

The PIXEL class contains three data members of type 8 bit unsigned integer that will hold the a value for each colour channel of a 24 bit pixel. the range of each data member is 0-255. The PIXEL class implements operator overloading of the equality '==' operator. The purpose of this is to compare two PIXEL objects together.

Listing 20: PIXEL_Q Class, C++

```

1  class PIXEL_Q
2  {
3  public:
4      uint8_t channel0; //Blue in RGB colour space, Y in YCbCr colour space
5      uint8_t channel1; //Green in RGB colour space, Cr in YCbCr colour space
6      uint8_t channel2; //GRed in RGB colour space, Cb in YCbCr colour space
7      bool friend operator ==(const PIXEL &p1, const PIXEL &p2)
8      {
9          if((abs(p1.channel0 -p2.channel0) < 2)
10         && (abs(p1.channel1 - p2.channel1) < 2 )
11         && (abs(p1.channel2 - p2.channel2) < 2 ))
12             return true;

```



```

13     else
14         return false;
15     };
16 };

```

The PIXEL_Q class is similar to the PIXEL class but has a different overloading of the equality operator. The equality operator is overloaded with a function that will perform light quantization on a per pixel pair basis. The absolute value of each data member of the first pixel minus the second pixel is computed, if the value is below a given threshold, then the pixels are said to be equal.

Listing 21: Block Classes, C++

```

1  class YCbCr_420_Block {
2  public:
3      unsigned char Y[256];           //16x16
4      unsigned char Cb[64];           //8x8
5      unsigned char Cr[64];
6  };
7
8  class YCbCr_444_Block {
9  public:
10     unsigned char Y[256];
11     unsigned char Cb[256];
12     unsigned char Cr[256];
13 };
14
15 class DCT_Coeffs_420 {
16 public:
17     short DCT_Y[256];
18     short DCT_Cb[64];
19     short DCT_Cr[64];
20 };
21
22 class DCT_Coeffs_444 {
23 public:
24     short DCT_Y[256];
25     short DCT_Cb[256];
26     short DCT_Cr[256];
27 };

```

The Block classes contain arrays for data members that are used to hold transformed values for one 16x16 pixel block. A YCbCr_420_Block object stores the values of a 16x16 RGB block transformed and chroma sub-sampled into YCbCr colour space. There is one Y sample for every four Cb and Cr sample. A YCbCr_444_Block object holds values for a 16x16 rgb block that has been transformed into YCbCr colour

space, without any chroma sub-sampling.

A DCT_Coeffs_420 object holds the values of a YCbCr_420_Block which has been transformed using a DCT function. The data type of each data member is 16 bit signed integer as the values after dct transform are both positive and negative

12.2 Classification Algorithm

Listing 22: Classification Algorithm Function Definitions, C++

```

1  /* member functions from Differential Index Map Coding compression
2     algorithm used for classification of 16x16 blocks into one of
3     four categories: Sparse, Text, Picture or Fuzzy
4     parameters: input argument is a count of unique
5     pixel values in a block.
6     Return value is val - an indicator of which category the
7     block under test is in
8     1 = fuzzy, 2 = picture, 3 = sparse, 4 = text*/
9
10 /****** Start of Classification Function *****/
11
12 int DIMC_COMPRESS::new_classify(int colourCount)
13 {
14     // Y values of the original Block
15     int y_channel_values[256] = { 0 };
16
17     // calls a function to populate y_channel value array
18     get_y_values(y_channel_values);
19
20     // hold the values of the sub bands after DWT processing
21     int sub_band_low_low[64] = { 0 };
22     int sub_band_low_high[64] = { 0 };
23     int sub_band_high_low[64] = { 0 };
24     int sub_band_high_high[64] = { 0 };
25
26     /* perform the DWT on the columns along the rows */
27     forward_col_transform_16x16(y_channel_values);
28
29     /* perform the DWT on the rows along the columns */
30     forward_row_transform_16x16(y_channel_values);
31
32     /* populate the sub_band arrays */
33     populate_sub_bands(y_channel_values, sub_band_low_low, 0);
34     populate_sub_bands(y_channel_values, sub_band_high_low, 8);
35     populate_sub_bands(y_channel_values, sub_band_low_high, 128);
36     populate_sub_bands(y_channel_values, sub_band_high_high, 136);
37     int locator = 0;
38
39     // calculate the rootmean of each sub band

```

```
40 // the root mean or quadratic mean is used
41 // as the values in the sub bands can be positive
42 // or negative
43 int meanVal_lo_lo = rootmean(sub_band_low_low, 64);
44 int meanVal_hi_lo = rootmean(sub_band_high_low, 64);
45 int meanVal_lo_hi = rootmean(sub_band_low_high, 64);
46 int meanVal_hi_hi = rootmean(sub_band_high_high, 64);
47
48 // calculate the variance of each sub band
49 int variance_hi_lo = variance(sub_band_high_low, 64, meanVal_hi_lo);
50 int variance_lo_lo = variance(sub_band_low_low, 64, meanVal_lo_lo);
51 int variance_lo_hi = variance(sub_band_low_high, 64, meanVal_lo_hi);
52 int variance_hi_hi = variance(sub_band_high_high, 64, meanVal_hi_hi);
53 int val = 0;
54 // calculate the standard deviation of each sub band
55 int standard_dev_lo_lo = sqrt(variance_lo_lo);
56 int standard_dev_hi_lo = sqrt(variance_hi_lo);
57 int standard_dev_lo_hi = sqrt(variance_lo_hi);
58 int standard_dev_hi_hi = sqrt(variance_hi_hi);
59
60 /***** Logic for classification*****/
61
62 // colourCount is a variable holding the count
63 // of unique 3 byte pixels in a 16x16 block
64
65 // if this condition is true, block = sparse block
66 if (colourCount < 5 && standard_dev_lo_lo < 12)
67 {
68     val = 3;
69 }
70
71 // if this condition is true, block = fuzzy block
72 else if (colourCount > 5 && standard_dev_lo_lo < 12)
73 {
74     val = 1;
75 }
76
77 // if this condition is true, block = sparse block
78 else if (((standard_dev_hi_lo == 0)
79         && (standard_dev_lo_hi == 0)
80         && (standard_dev_hi_hi == 0)))
81 {
82
83     val = 3;
84 }
85 // if this condition is true, block = sparse block
86 else if (colourCount < 10
87         && standard_dev_lo_hi > 4 * standard_dev_hi_lo
88         && standard_dev_lo_hi > 4 * standard_dev_hi_hi)
89 {
```

```
90     val = 3;
91 }
92 else if (colourCount<10
93     && standard_dev_hi_lo>4 * standard_dev_lo_hi
94     && standard_dev_hi_lo> 4 * standard_dev_hi_hi)
95 {
96     val = 3;
97 }
98 else if (colourCount<10
99     && standard_dev_hi_hi>4 * standard_dev_lo_hi
100     && standard_dev_hi_hi> 4 * standard_dev_hi_lo)
101 {
102     val = 3;
103 }
104
105 // if this condition is true, block is a text block
106 else if (colourCount<31 ||
107     (standard_dev_hi_lo >25
108     || standard_dev_lo_hi >25
109     || standard_dev_hi_hi> 25))
110 {
111     //4=text
112     val = 4;
113 }
114
115 // if this condition is true, block = picture block
116 else if (colourCount>31 || ((standard_dev_hi_lo < 10)
117     && (standard_dev_lo_hi < 12)
118     && (standard_dev_hi_hi < 8)))
119 {
120     //2 = picture
121     val = 2;
122 }
123
124 // if this condition is true, block is a text block
125 else if (colourCount<31 ||
126     (standard_dev_hi_lo >25
127     || standard_dev_lo_hi >25
128     || standard_dev_hi_hi> 25 ))
129 {
130     //4=text
131     val = 4;
132 }
133 else
134 { //picture
135     val = 2;
136 }
137
138 return val;
139 }
```

```

140
141 /***** End of Classification Function *****/
142
143 /***** Classification Helper functions *****/
144
145 /* function to populate an array with 256 Y channel values
146    Block16x16 is a data member of DIMC_COMPRESS class
147    it is a openCV Mat object, which is a smart pointer
148    to a sub matrix of an image.
149    y_channel_pointer is pointer of type unsigned
150    char that points to the data held in Block16x16 */
151
152 void DIMC_COMPRESS::get_y_values(int y_vals[])
153 {
154     uint8_t* y_channel_pointer;
155     int y = 0;
156     for (int row = 0; row < 16; row++)
157     {
158         y_channel_pointer = Block16x16.ptr(row);
159         /*
160         a mat object stores channel values like:
161         Block[row][0] = Y
162         Block[row][1] = Cr
163         Block[row][2] = Cb
164         so thats why you iterate to 48:
165         */
166         for (int col = 0; col < 48; col++, y++)
167         {
168             /* get the Y value */
169             y_vals[y] = y_channel_pointer[col];
170             /* skip the other values and increment the locator */
171             col += 2;
172         }
173     }
174 }
175
176 /* populate_sub_bands is a member function from
177    differential Index Map compression algoritm. arguments:
178    transformed_vals is a pointer to an array of integer
179    values of discrete wavelet transform values of length 256.
180    sub_band is a pointer to one of four integer array's
181    that will hold the value of each sub band. locator is a
182    value that is used to navigate the transformed_vals */
183
184 void DIMC_COMPRESS::populate_sub_bands( int transformed_vals[],
185                                         int sub_band[],
186                                         int locator)
187 {
188     /* point to the beginning of each sub_band
189     in the 1D array of coefs */

```

```

190     int *coef_pointer = transformed_vals+ locator;
191     /* each row has 8 columns*/
192     int offset = 16;
193     /* where to put the coef in the subband */
194     int next_coef = 0;
195
196     for (int i = 0; i < 8; i++)
197     {
198         coef_pointer = transformed_vals + locator+(i*offset);
199         for (int col = 0;col<8; col++,next_coef++)
200         {
201             sub_band[next_coef] = coef_pointer[col];
202         }
203     }
204
205 }
206
207
208 /* Forward 2D Discrete Wavelet Transform
209 Split into two 1D DWT Transforms */
210 void DIMC_COMPRESS::forward_col_transform_16x16(int sub_band[])
211 {
212     /* the weights s0,s1,w0,w1 are 0.5,0.5,0.5,-0.5 respectively
213     they have been shifted up 10 bits so the can be
214     performed with integer calculations */
215     int offset = 16;
216     for (int row = 0; row < 16; row++)
217     {
218         double temp[16] = { 0 };
219         /* forward transform level one*/
220         int length = 16;
221         /* if length is 16, h is 8 */
222         int h = length >> 1;
223         for (int i = 0; i < h; i++)
224         {
225             /*if i =0, k=1, if i =2, k =4, etc */
226             int k = (i << 1);
227             /*averaging calculation*/
228             temp[i] = sub_band[row*offset+k] * s0
229                 + sub_band[row*offset+(k + 1)] * s1;
230
231             /* differencing calculation */
232             temp[i + h] = sub_band[row*offset + k] * w0
233                 + sub_band[row*offset +(k + 1)] * w1;
234         }
235
236         /* store the basis calculations 'In place'
237         in the original 2D array */
238         for (int i = 0; i < length; i++)
239         {

```

```

240     // pushback is the resultant value after dividing by 1024
241     // equivalent of a right shift by 10 bits
242     double pushBack = ((temp[i]) / 1024);
243     sub_band[row*offset + i] = pushBack;
244
245     }
246
247     }
248 }
249
250 void DIMC_COMPRESS::forward_row_transform_16x16(int sub_band[])
251 {
252     int offset = 16;
253     for (int current_col = 0; current_col < 16; current_col++)
254     {
255         double temp[16] = { 0 };
256
257         /* forward transform level one*/
258         int length = 16;
259         /*
260          * if length is 16, h is 8,
261          * temp[0]->[7] contain the averages,
262          * temp[8]->[15] contain the differences
263          */
264         int h = length >> 1;
265
266         for (int i = 0; i < h; i++)
267         {
268             /*if i =0, k=1, if i =2, k =4, etc */
269             int k = (i << 1);
270             /*averaging calculation*/
271             temp[i] = sub_band[current_col+(offset * k)] * s0
272                 + sub_band[current_col+ offset*(k + 1)] * s1;
273
274             /* differencing calculation */
275             temp[i + h] = sub_band[current_col + (offset * k)] * w0
276                 + sub_band[current_col + offset * (k + 1)] * w1;
277         }
278
279         for (int i = 0; i < length; i++)
280         {
281             //double roundedVal = (temp[i] / 1024);
282             //if(roundedVal)
283             sub_band[(i*offset)+current_col] = (temp[i]/1024);
284         }
285     }
286 }
287
288 /* root mean calculates the quadratic mean of an array of values
289    that contain one of the four sub bands after Discrete

```

```

290 Wavelet Tranform arguments: sub_band = an integer pointer
291 to an array of sub band values, size = the size of the sub band.
292 return the rootmean */
293 int DIMC_COMPRESS::rootmean(int sub_band[], int size)
294 {
295     /* holds the total some of all values of
296     the block that is passed */
297     int sum_of_channel_values = 0;
298
299     for (int i = 0; i < size; i++)
300     {
301         sum_of_channel_values += sub_band[i] * sub_band[i];
302     }
303     /* performing the calculation as floating point,
304     then rounding up to an integer. */
305     int mean = sqrt(ceil(static_cast<float>(sum_of_channel_values) /
306         static_cast<float>(size)));
307     return mean;
308 }
309
310 int DIMC_COMPRESS::variance(int sub_band[], int size, int mean)
311 {
312     int sum = 0;
313
314     /* declaring an array to hold the variance
315     calculations for an 8x8 block of DWT coefs */
316     int squared_values[64] = { 0 };
317
318     /* pointing to the index value of the
319     64 byte array of DWT coefs */
320     int *value_ptr = sub_band;
321
322     for (int i = 0; i < size; i++)
323     {
324         /* subtracting the mean from each value in the
325         sub band and storing it in squared_values*/
326         squared_values[i] = value_ptr[i] -= mean;
327
328         /* squaring each values */
329         squared_values[i] = squared_values[i] * squared_values[i];
330
331         /*taking the sum */
332         sum += squared_values[i];
333     }
334
335     /* returning the variance as an integer
336     - will incur rounding error */
337     int variance = round(static_cast<float>(sum)
338         / static_cast<float>(size));
339

```



```

340     return variance;
341 }

```

12.3 Discrete Cosine Transform Code

Listing 23: Direct 2-Dimensional DCT, C++

```

1 void DCT_Function::dct_direct_2D( double *f, double *F )
2 {
3     // a = Normalization (alpha) values,
4     // coef = value of dct coef at M[i][j], M = dct coefficient Matrix
5     // sum = total of weighted dct coef's times input sample times alpha values
6     double a[8], sum, coef;
7     //N = 8 for 8x8 block of pixels
8     uint8_t N=8;
9
10    // calculate the alpha values
11    a[0] = sqrt ( 1.0 / N );
12    for (uint8_t b = 1; b < N; ++b )
13    {
14        a[b] = sqrt ( 2.0 / N );
15    }
16    for (uint8_t u = 0; u < N; ++u )
17    {
18        for (uint8_t v = 0; v < N; ++v )
19        {
20            sum = 0.0;
21            for (uint8_t i = 0; i < N; ++i )
22            {
23                for (uint8_t j = 0; j < N; ++j )
24                {
25                    coef = std::cos ((2*i+1) * u * PI / (2 * N))
26                        * std::cos ((2 * j+1) * v * PI/(2 * N));
27                    sum += *(f+i * N+j) * coef; //f[i][j] * coef
28                } //for j
29                *(F+u * N+v) = a[u] * a[v] * sum;
30            } //for i
31        } //for u
32    } //for v
33 }

```

The DCT function in listing 23 computes the 2-Dimensional Discrete Cosine Transform of an 8x8 block of samples. It takes two pointers to arrays as arguments: *f* is the input array, and *F* is the output array. This transform implementation is fully reversible. If the input array are YCbCr values, the output array are DCT Coeffi-

cients and vice versa. This implementation has a time complexity of $O(N^4)$, which is too slow for real-time application

Listing 24: 1-D DCT Transform, C++

```
1  #define PI 3.141592653589
2  double factor = PI / 8;
3
4  void DCT_Function::forward_1D_Dct(double input[8], double output[8])
5  {
6      // N = length of input array
7      uint8_t N = 8;
8      for (uint8_t i = 0; i < N; i++)
9      {
10         double sum = 0;
11         double coef = 0;
12         for (uint8_t j = 0; j < N; j++)
13         {
14             coef = std::cos((j + 0.5) * i * factor);
15             sum += input[j] * coef;
16         }
17         output[i] =sum;
18     }
19 }
20
21 void DCT_Function::inverse_1D_Dct(double input[8], double output[8])
22 {
23     uint8_t N = 8;
24     for (size_t i = 0; i < N; i++) {
25         double sum = input[0] / 2;
26         double coef =0;
27         for (size_t j = 1; j < N; j++)
28         {
29             coef = std::cos(j * (i + 0.5) * factor)
30             sum += input[j] * coef ;
31         }
32         output[i] =(sum/4);
33     }
34 }
```

Listing 24 shows the implementation of the forward and inverse 1 dimensional Discrete cosine transform. The functions take two arguments, a reference to an input array and output array. performing the 2-D discrete as two 1-D transforms, first along the columns, then the rows, reduces the time complexity to $O(N^3)$

Listing 25: Fast Arai,Agui,Nakajima (AAN) 1D DCT Transform, C++

```

1 void FastDct8::transform(double vector[8]) {
2     const double v0 = vector[0] + vector[7];
3     const double v1 = vector[1] + vector[6];
4     const double v2 = vector[2] + vector[5];
5     const double v3 = vector[3] + vector[4];
6     const double v4 = vector[3] - vector[4];
7     const double v5 = vector[2] - vector[5];
8     const double v6 = vector[1] - vector[6];
9     const double v7 = vector[0] - vector[7];
10
11     const double v8 = v0 + v3;
12     const double v9 = v1 + v2;
13     const double v10 = v1 - v2;
14     const double v11 = v0 - v3;
15     const double v12 = -v4 - v5;
16     const double v13 = (v5 + v6) * A[3];
17     const double v14 = v6 + v7;
18
19     const double v15 = v8 + v9;
20     const double v16 = v8 - v9;
21     const double v17 = (v10 + v11) * A[1];
22     const double v18 = (v12 + v14) * A[5];
23
24     const double v19 = -v12 * A[2] - v18;
25     const double v20 = v14 * A[4] - v18;
26
27     const double v21 = v17 + v11;
28     const double v22 = v11 - v17;
29     const double v23 = v13 + v7;
30     const double v24 = v7 - v13;
31
32     const double v25 = v19 + v24;
33     const double v26 = v23 + v20;
34     const double v27 = v23 - v20;
35     const double v28 = v24 - v19;
36
37     vector[0] = S[0] * v15;
38     vector[1] = S[1] * v26;
39     vector[2] = S[2] * v21;
40     vector[3] = S[3] * v28;
41     vector[4] = S[4] * v16;
42     vector[5] = S[5] * v25;
43     vector[6] = S[6] * v22;
44     vector[7] = S[7] * v27;
45 }

```

The fast implementation is from [1]

12.4 Deflate Interface

These are the functions to interface with the zlib dynamically linked library

Listing 26: Interface for zlib dll, Deflate compression Functions, C++

```

1  int Z_LIB::compress(vector<uint8_t> &src, vector<uint_8t> &dest)
2  {
3      int level = 9; // for best compression
4
5      constexpr int CHUNK = 256000;
6
7      int ret, flush;
8      unsigned have;
9      z_stream strm;
10     unsigned char out[CHUNK];
11
12     /* allocate deflate state */
13     strm.zalloc = Z_NULL;
14     strm.zfree = Z_NULL;
15     strm.opaque = Z_NULL;
16     ret = deflateInit(&strm, level);
17
18     /* input data specified in one big chunk */
19     strm.avail_in = src.size();
20     flush = Z_FINISH; // instead of Z_NO_FLUSH, as this is the entire data
21     strm.next_in = src.data();
22
23     /* run deflate() on input until output buffer not full, finish
24     compression if all of source has been read in */
25     do {
26         strm.avail_out = CHUNK;
27         strm.next_out = out;
28         ret = deflate(&strm, flush); /* no bad return value */
29         assert(ret != Z_STREAM_ERROR); /* state not clobbered */
30         have = CHUNK - strm.avail_out;
31         dest.insert(dest.end(), out, out+have);
32     } while (strm.avail_out == 0);
33     assert(strm.avail_in == 0); /* all input will be used */
34
35     assert(ret == Z_STREAM_END); /* stream will be complete */
36
37     /* clean up and return */
38     (void)deflateEnd(&strm);
39
40     return 0;
41 }
42 int Z_LIB::decompress(vector<uint8_t> &src, vector<uint_8t> &dest)
43 {
44

```

```

45     constexpr int CHUNK = 256000;
46
47     int ret;
48     unsigned have;
49     z_stream strm;
50     unsigned char out[CHUNK];
51
52     /* allocate inflate state */
53     strm.zalloc = Z_NULL;
54     strm.zfree = Z_NULL;
55     strm.opaque = Z_NULL;
56     strm.avail_in = 0;
57     strm.next_in = Z_NULL;
58     ret = inflateInit(&strm);
59     if (ret != Z_OK)
60         throw exc:: error_from_inflateInit_t{ret};
61
62     strm.avail_in = src.size();
63     strm.next_in = src.data();
64
65     /* run inflate() on input until output buffer not full */
66     do {
67         strm.avail_out = CHUNK;
68         strm.next_out = out;
69         ret = inflate(&strm, Z_NO_FLUSH);
70         assert(ret != Z_STREAM_ERROR); /* state not clobbered */
71         switch (ret) {
72             case Z_NEED_DICT:
73                 ret = Z_DATA_ERROR; /* and fall through */
74             case Z_DATA_ERROR:
75             case Z_MEM_ERROR:
76                 (void)inflateEnd(&strm);
77                 throw exc:: error_from_inflateInit_t{ret};
78         }
79         have = CHUNK - strm.avail_out;
80         dest.insert(dest.end(), out, out+have);
81     } while (strm.avail_out == 0);
82
83     // deflated one big chunk of input, so ret
84     // will be Z_STREAM_END now
85     assert(ret == Z_STREAM_END);
86
87     /* clean up and return */
88     (void)inflateEnd(&strm);
89 }

```

12.5 Differential Index Map Coding Functions

Listing 27: Differential Index Map Coding compression Functions, C++

```

1  /*
2     colour_count is a class member function that counts the unique 3-byte pixels
3     in a 16x16 block and stores them in an array, which is the index table for
4     the index map(s).
5     Takes no arguments, populates class data member array indexTable[]
6  */
7  int DIMC_COMPRESS::colour_count()
8  {
9     /* pixel_pointer used to point to the beginning of
10     a 3 byte pixel in a 16x16 block of pixels */
11     uchar* pixel_pointer;
12
13     /* recovers the RGB values of the pixel pointed to */
14     PIXEL pixel;
15
16     for (int row = 0; row < 16; row++)
17     {
18         pixel_pointer = Block16x16.ptr<uchar>(row);
19
20         /* col goes to 48 as the pixels are in a "packed" format, not planar */
21         for (int col = 0; col < 48; col++)
22         {
23             pixel.channel0 = pixel_pointer[col];
24             ++col;
25             pixel.channel1 = pixel_pointer[col];
26             ++col;
27             pixel.channel2 = pixel_pointer[col];
28
29             /*the '==' operator is overloaded and will return true if a pixel
30             and its neighbor are equal
31             after being shifted over by a specified range*/
32
33             /*searches the vector of unique pixels, if not found, puts it in*/
34
35             if (std::find(indexTable.begin(), indexTable.end(), pixel)
36                 == indexTable.end())
37             {
38                 indexTable.emplace_back(pixel);
39             }
40             /* Break out of the function to use DCT on block*/
41             if (indexTable.size() > 30)
42             {
43                 return -1;
44             }
45         }
46     }

```

```

47     return indexTable.size();
48 }
49
50 /*Calling function to generate four 8x8 index maps*/
51 void DIMC_COMPRESS::make_index_map()
52 {
53     index_map8x8(0, 0, i_Map_Top_L);
54     index_map8x8(0, 24, i_Map_Top_R);
55     index_map8x8(8, 0, i_Map_Bottom_L);
56     index_map8x8(8, 24, i_Map_Bottom_R);
57 }
58
59 /*Function to create an 8x8 index map, which is a sub block from
60 a 16x16 block. the indexMap values will be stored in a linear array instead
61 of a 2-d array to speed up processing */
62 void DIMC_COMPRESS::index_map8x8(int row_offset, int col_offset, uint8_t i_Map[])
63 {
64     /* iterator is used to put the index value into the index map */
65     int iterator = 0;
66     int column_offset = col_offset+24;
67     for (int row = 0; row < 8; row++)
68     {
69         /* pixel_pointer points to the beginning of a 3 byte pixel in Block16x16 */
70         uint8_t* pixel_pointer = Block16x16.ptr<uchar>((row + row_offset));
71         for (int col = col_offset; col < column_offset; col++)
72         {
73
74             /* used to put a matched index value in the index map */
75             vector<PIXEL>::iterator it;
76             PIXEL pixel;
77             pixel.channel0 = pixel_pointer[col];
78             ++col;
79             pixel.channel1 = pixel_pointer[col];
80             ++col;
81             pixel.channel2 = pixel_pointer[col];
82
83             /* iterate through the pixel value array to find a match */
84             for (it = indexTable.begin(); it != indexTable.end(); it++)
85             {
86                 if (*it == pixel)
87                 {
88                     /* loc = location in vector of pixel value*/
89                     uint8_t loc = it - indexTable.begin();
90                     /* sets the value in the index map equal to the index in
91                     the pixel_value vector */
92
93                     i_Map[iterator] = loc;
94                     iterator++;
95                     break;
96                 }

```



```

97     }
98   }
99 }
100 uint8_t last = 0;
101 int pause = 0;
102 }
103
104 void DIMC_COMPRESS::index_map16x16(uint8_t i_Map[])
105 {
106     int iterator = 0;
107     PIXEL pixel;
108     uint8_t *pixel_pointer = NULL;
109
110     for (int row = 0; row < 16; row++)
111     {
112
113         pixel_pointer = Block16x16.ptr<uchar>(row);
114         for (int col = 0; col < 48; col++)
115         {
116             vector<PIXEL>::iterator it;
117             pixel.channel0 = pixel_pointer[col];
118             ++col;
119             pixel.channel1 = pixel_pointer[col];
120             ++col;
121             pixel.channel2 = pixel_pointer[col];
122
123             /* iterate through the pixel array to find a match */
124             for (it = indexTable.begin(); it != indexTable.end(); it++)
125             {
126                 if (*it == pixel)
127                 {
128                     /*loc = location in vector of pixel value*/
129                     uint8_t loc = it - indexTable.begin();
130                     /* sets the value in the index map equal to the index
131                     in the pixel_value vector*/
132                     i_Map[iterator] = loc;
133                     iterator++;
134                     break;
135                 }
136             }
137         }
138     }
139 }
140
141
142 void DIMC_COMPRESS::create_header(vector<uint8_t> &coded)
143 {
144     /*
145     header byte:
146     bit[0]      0 = 16x16 block comparison, 1 = 8x8 block comparrsion

```

```

147     bit[1]          0 = horizontal comparison, 1 = vertical comparison
148     bit[2]-[7]     amount of different pixels 0-31
149     */
150
151     bool block8x8Flag = 0;
152     bool block16x16Flag = 0;
153
154     /*header: comparison size and amount of colours
155     example 0010 0000 = block uses 16x16 compare, 8 unique colours */
156     uint8_t header = 0;
157
158     // insert the amount of colours of the block into the header
159     header = (header | indexTable.size());
160
161     // shift the pixel count up by 2 bits
162     header = header << 2;
163     /* bit[0] =0 if block is 16x16 */
164     if (indexTable.size() < 16)
165     {
166         header = (header | 0);
167         block16x16Flag = 1;
168     }
169     else
170     {
171         //bit[0] =1 if 8x8 blocks
172
173         header = (header | 1);
174         block8x8Flag = 1;
175     }
176
177     coded.emplace_back(header); // store header byte in data vector
178
179     /*X Y Coordinates With respect to the 16x16 block in the
180     Original Image*/
181
182     uint8_t block_row = 0;
183
184     block_row = (block_row | block_Row);
185
186     uint8_t block_col = 0;
187
188     block_col = (block_col | block_Column);
189
190     coded.emplace_back(block_row);
191     coded.emplace_back(block_col);
192
193     /* Write The colours to the output file */
194
195     for (int i = 0; i < indexTable.size(); i++)
196     {

```

```

197     // pushing the unique pixel values into the coded stream
198     coded.emplace_back(indexTable[i].channel0); // channel0 value
199     coded.emplace_back(indexTable[i].channel1); // channel1 Value
200     coded.emplace_back(indexTable[i].channel2); // channel2 Value
201
202 }
203 /* Base Line Run Length */
204 if (block8x8Flag)
205 {
206     base_line_run_length(i_Map_Top_L, coded);
207     base_line_run_length(i_Map_Top_R, coded);
208     base_line_run_length(i_Map_Bottom_L, coded);
209     base_line_run_length(i_Map_Bottom_R, coded);
210
211     compare_rows(coded, i_Map_Top_L);
212     //coded.emplace_back(0);
213     compare_rows(coded, i_Map_Top_R);
214     //coded.emplace_back(0);
215     compare_rows(coded, i_Map_Bottom_L);
216     //coded.emplace_back(0);
217     compare_rows(coded, i_Map_Bottom_R);
218     //coded.emplace_back(0);
219 }
220
221 if (block16x16Flag)
222 {
223     base_line_run_length_16(i_Map_16x16, coded);
224     compare_rows_16(coded, i_Map_16x16);
225 }
226 /*End Of Base Line Run Length*/
227 }
228
229
230
231 void DIMC_COMPRESS::base_line_run_length(uint8_t i_Map[], vector<uint8_t> &coded)
232 {
233     /*
234     extractor is a byte that will contain
235     the run value in ms nibble and length in ls nibble
236     */
237     uint8_t extractor = 0;
238
239     /* set pointer to row 4 of an 8x8 block */
240     uint8_t *centre_Row_Ptr = i_Map + 32;
241
242     /* declare a variable to hold run length */
243     uint8_t run_length = 0;
244
245     for (int i = 0; i < 8; i++)
246     {

```

```

247     run_length = 0;
248
249     /* if the next val + 1 is less than the
250     No of columns and index i+ 1 = index i */
251     while (i + 1 < 8 && centre_Row_Ptr[i] == centre_Row_Ptr[i + 1])
252     {
253         run_length++; // increase the run
254         i++; // move the iterator
255     }
256     /* extractor will contain the run data and length for
257     a specific run ex: 0010 1111 = index 1 from 0-15 */
258
259     uint8_t extractor = 0;
260
261     // The run value index
262     extractor = (extractor | centre_Row_Ptr[i]);
263     //push it up to the top 3 bits
264     extractor = extractor << 3;
265     // the length of the run
266     extractor = (extractor | run_length);
267     coded.emplace_back(extractor);
268 }
269 }
270
271 void DIMC_COMPRESS::base_line_run_length_16(uint8_t i_Map[], vector<uint8_t> &coded)
272 {
273     /*
274     extractor is a byte that will contain
275     the run value in ms nibble and length in ls nibble
276     */
277     uint8_t extractor = 0;
278
279     /* set pointer to row 4 of an 8x8 block */
280     uint8_t *centre_Row_Ptr = i_Map + 128;
281
282     /* declare a variable to hold run length */
283     uint8_t run_length = 0;
284
285     for (int i = 0; i < 16; i++)
286     {
287         run_length = 0;
288
289         /* if the next val + 1 is less than the
290         No of columns and index i+ 1 = index i */
291         while (i + 1 < 16 && centre_Row_Ptr[i] == centre_Row_Ptr[i + 1]) //
292         {
293             run_length++;
294             i++;
295         }
296

```

```

297     uint8_t extractor = 0; /
298     extractor = (extractor | centre_Row_Ptr[i]);
299     extractor = extractor << 4;
300     extractor = (extractor | run_length);
301     coded.emplace_back(extractor);
302 }
303 }
304
305 void DIMC_COMPRESS::compare_rows(vector<uint8_t> &coded, uint8_t i_Map[])
306 {
307     uint8_t increment_no_change = 32; // 0010 0000
308     uint8_t nc_symbol = 31; // 0001 1111
309
310     /* the return value of the two row comparison
311     if no change, it returns a 1 */
312     int no_change_flag = 0;
313
314     /* holds the amount of non changed rows*/
315     int no_change_count = 0;
316     for (int row = 4; row > 0; row--)
317     {
318         /* offset is a variable that is used to set
319         the correct row of the index map */
320         int offset = row * 8;
321         // no_change_flag = 1: no change
322         // no_change_flag = 0: there is a change
323
324         no_change_flag = two_row_compare(coded, i_Map, offset, -8);
325
326         /* no change count gets incremented on successive no change rows */
327         no_change_count += no_change_flag;
328
329         /* check the flag */
330         if (no_change_flag == 1)
331         {
332             if (no_change_count > 1)
333             { /* increment the no change symbol ahead in code stream */
334                 coded.back() += increment_no_change;
335             }
336             else
337             {
338                 /* insert a no change symbol */
339                 coded.emplace_back(nc_symbol);
340             }
341         }
342         else
343             /* this condition is true when there has been a change on a row */
344             no_change_count = 0;
345     }
346     no_change_count = 0;

```

```

347     for (int row = 4; row < 7; row++)
348     {
349         /* offset is a variable that is used to
350         set the correct row of the index map */
351         int offset = row * 8;
352
353         // no_change_flag = 1: no change
354         // no_change_flag = 0: there is a change
355
356         no_change_flag = two_row_compare(coded, i_Map, offset, +8);
357         /* no change count gets incremented on successive no change rows */
358         no_change_count += no_change_flag;
359         /* check the flag */
360         if (no_change_flag == 1)
361         {
362             if (no_change_count > 1)
363             { /* increment the no change symbol ahead in code stream */
364                 coded.back() += increment_no_change;
365             }
366             else
367             {
368                 /* insert a no change symbol */
369                 coded.emplace_back(nc_symbol);
370             }
371         }
372         else
373             /* this condition is true when there has been a change on a row */
374             no_change_count = 0;
375     }
376 }
377
378 void DIMC_COMPRESS::compare_rows_16(vector<uint8_t> &coded, uint8_t i_Map[])
379 {
380     uint8_t increment_no_change = 16; // 0001 0000
381     uint8_t nc_symbol = 15; // 0000 1111
382
383     /* the return value of the two
384     row comparison if no change, it returns a 1 */
385
386     int no_change_flag = 0;
387     /* holds the amount of non changed rows*/
388     int no_change_count = 0;
389     for (int row = 8; row > 0; row--)
390     {
391         // offset is a variable that is used to set
392         //the correct row of the index map
393
394         int offset = row * 16;
395         // no_change_flag = 1: no change

```

```
397     // no_change_flag = 0: there is a change
398
399     no_change_flag = two_row_compare_16(coded, i_Map, offset, -16);
400     /* no change count gets incremented on successive no change rows */
401     no_change_count += no_change_flag;
402     /* check the flag */
403     if (no_change_flag == 1)
404     {
405         if (no_change_count > 1)
406         { /* increment the no change symbol ahead in code stream */
407             coded.back() += increment_no_change;
408         }
409         else
410         {
411             /* insert a no change symbol */
412             coded.emplace_back(nc_symbol);
413         }
414     }
415     else
416         /* this condition is true when there
417         has been a change on a row */
418
419         no_change_count = 0;
420 }
421 no_change_count = 0;
422 for (int row = 8; row < 15; row++)
423 {
424     int offset = row * 16;
425     // no_change_flag = 1: no change
426     // no_change_flag = 0: there is a change
427
428     no_change_flag = two_row_compare_16(coded, i_Map, offset, +16);
429     /* no change count gets incremented on successive no change rows */
430     no_change_count += no_change_flag;
431     /* check the flag */
432     if (no_change_flag == 1)
433     {
434         if (no_change_count > 1)
435         { /* increment the no change symbol ahead in code stream */
436             coded.back() += increment_no_change;
437         }
438         else
439         {
440             /* insert a no change symbol */
441             coded.emplace_back(nc_symbol);
442         }
443     }
444     else
445         /* this condition is true when there has been a change on a row */
446         no_change_count = 0;
```

```

447     }
448 }
449
450
451
452 uint8_t DIMC_COMPRESS::two_row_compare(vector<uint8_t> &coded,
453                                     uint8_t i_Map[], int offset, int of)
454 {
455     uint8_t no_change_flag = 1;
456     uint8_t compare_row_offset = offset;
457     uint8_t current_row_offset = compare_row_offset + of;
458     uint8_t extractor = 0;
459
460     /* points to the row that the current row will be compared to */
461     uint8_t* compare_ptr = i_Map + compare_row_offset;
462     uint8_t* current_ptr = i_Map + current_row_offset;
463
464     for (int col = 0; col < 8; col++)
465     {
466         /* comparrison of two rows column by column */
467         if (col < 8 && current_ptr[col] != compare_ptr[col])
468         {
469             // 0000 0000 used to logically or with
470             // position (upper nibble) and colour (lower nibble)
471             uint8_t extractor = 0;
472             // extracting the location in the row.
473             extractor = (col | extractor);
474
475             // shifting the location in the row up five bits
476             extractor = extractor << 5;
477
478             // extracting the value of the index map.
479             extractor = (extractor | current_ptr[col]);
480             coded.emplace_back(extractor);
481             no_change_flag = 0;
482         }
483
484     }
485     /* insert end of row symbol */
486     if (no_change_flag == 0)
487     {
488         coded.emplace_back(255);
489     }
490
491     return no_change_flag;
492 }
493
494 uint8_t DIMC_COMPRESS::two_row_compare_16(vector<uint8_t> &coded,
495                                         uint8_t i_Map[], int offset, int of)
496 {

```



```

497     uint8_t no_change_flag = 1;
498     uint8_t compare_row_offset = offset;
499     uint8_t current_row_offset = compare_row_offset + of;
500     uint8_t extractor = 0;
501
502     /* points to the row that the current row will be compared to */
503     uint8_t* compare_ptr = i_Map + compare_row_offset;
504     uint8_t* current_ptr = i_Map + current_row_offset;
505
506     for (int col = 0; col < 16; col++)
507     {
508         /* comparrison of two rows column by column */
509         if (col < 16 && current_ptr[col] != compare_ptr[col])
510         {
511
512             // 0000 0000 used to logically or with
513             // position (upper nibble) and colour (lower nibble)
514
515             uint8_t extractor = 0;
516             extractor = (col | extractor);
517             extractor = extractor << 4;
518             extractor = (extractor | current_ptr[col]);
519             coded.emplace_back(extractor);
520             no_change_flag = 0;
521         }
522     }
523
524     /* insert end of row symbol */
525     if (no_change_flag == 0)
526     {
527         coded.emplace_back(255);
528     }
529
530     return no_change_flag;
531 }

```

12.5.1 Discrete Wavelet Transform Functions

Listing 28: Non Zero Index Map Function to encod Thresholded Discrete Wavelet Transform Coefficients, C++

```

1 void DWT_INT_Compression::make_Index_map(int data[16][16],
2     uint8_t indexMap[16][16],
3     vector<uint8_t>&coded)
4 {
5
6     uint16_t zero_row = 1 << 15;

```

```

7  uint16_t header = 0;
8  int non_zero_count = 0;
9
10 int header_byte_1 = coded.size();
11 coded.emplace_back(0);
12 int header_byte_2 = coded.size();
13 coded.emplace_back(0);
14
15 vector<uint8_t> temp;
16 for (int i = 0; i < 16; i++)
17 {
18     uint16_t nz_locator = 1 << 15;
19     uint16_t nz_val = 0;
20     for (int j = 0; j < 16; j++)
21     {
22         if (data[i][j] != 0)
23         {
24             indexMap[i][j] = 1;
25             temp.emplace_back(data[i][j]);
26             non_zero_count++;
27             nz_val |= nz_locator;
28         }
29         nz_locator >>= 1;
30     }
31     if (nz_val > 0)
32     {
33         // get the row 0->15
34         header |= zero_row;
35         //
36
37         uint8_t LSB = nz_val & 255;
38         nz_val >>= 8;
39         uint8_t MSB = nz_val & 255;
40
41         coded.emplace_back(MSB);
42         coded.emplace_back(LSB);
43     }
44     zero_row >>= 1;
45 }
46 //get the 2 byte header
47 uint8_t LSB = header & 255;
48 header >>= 8;
49 uint8_t MSB = header & 255;
50
51
52 coded.insert(coded.end(), temp.begin(), temp.end());
53 coded[header_byte_1] = MSB;
54 coded[header_byte_2] = LSB;
55 }

```

12.6 OpenCV and FFMPEG Code

Listing 29: Interface for JPEG compression, C++

```

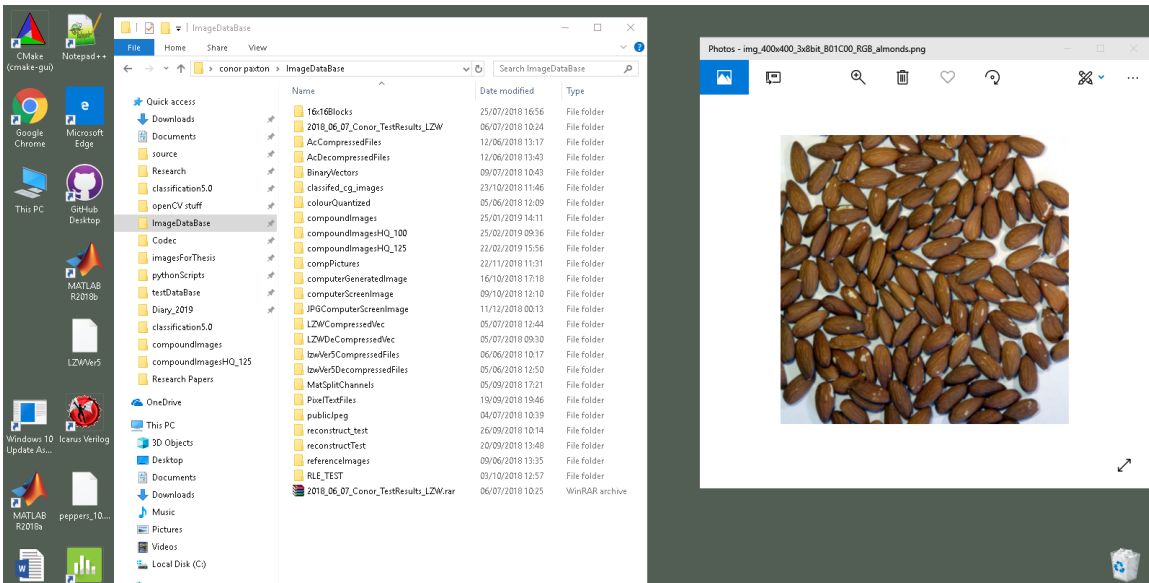
1 void JPEG_Compression()
2 {
3     Utilities utils;
4     int JPEG_QUALITY = utils.convert_Arg_int(argv[2]);
5     //int JPEG_QUALITY = 50, 60, 70, 80, 90;
6
7     // start the time
8     auto start = chrono::steady_clock::now();
9
10    // compress and write image to file with quality factor
11    cv::imwrite("test.jpg", OriginalImage,
12        vector<int>({ CV_IMWRITE_JPEG_QUALITY, JPEG_QUALITY }));
13
14    // stop timer
15    auto end = chrono::steady_clock::now();
16
17    // calculate time take
18    auto elapsed_time =
19        chrono::duration_cast<chrono::microseconds>(end - start).count();
20
21    // read size of compressed file
22    std::ifstream in("test.jpg", std::ifstream::ate | std::ifstream::binary);
23    int c_size = in.tellg();
24
25    // load compressed image into a mat object
26    cv::Mat RecoveredJpeg = cv::imread("test.jpg");
27
28    // call ssim function calculate ssim index
29    double ssim_val = SSIM(OriginalImage, RecoveredJpeg);
30    // calculate peak signal to noise ratio
31    double psnr_val = PSNR(OriginalImage, RecoveredJpeg);
32
33    double bit_rate = static_cast<double>(c_size) /
34        (static_cast<double>(imgHeight * imgWidth));
35
36    double compression_ratio =
37        static_cast<double>(imgHeight*imgWidth * 3)
38        / static_cast<double>(c_size);
39
40    cout << compression_ratio << " , " << bit_rate
41        << " , " << psnr_val << " , " << ssim_val
42        << " , " << elapsed_time << endl;
43 }

```

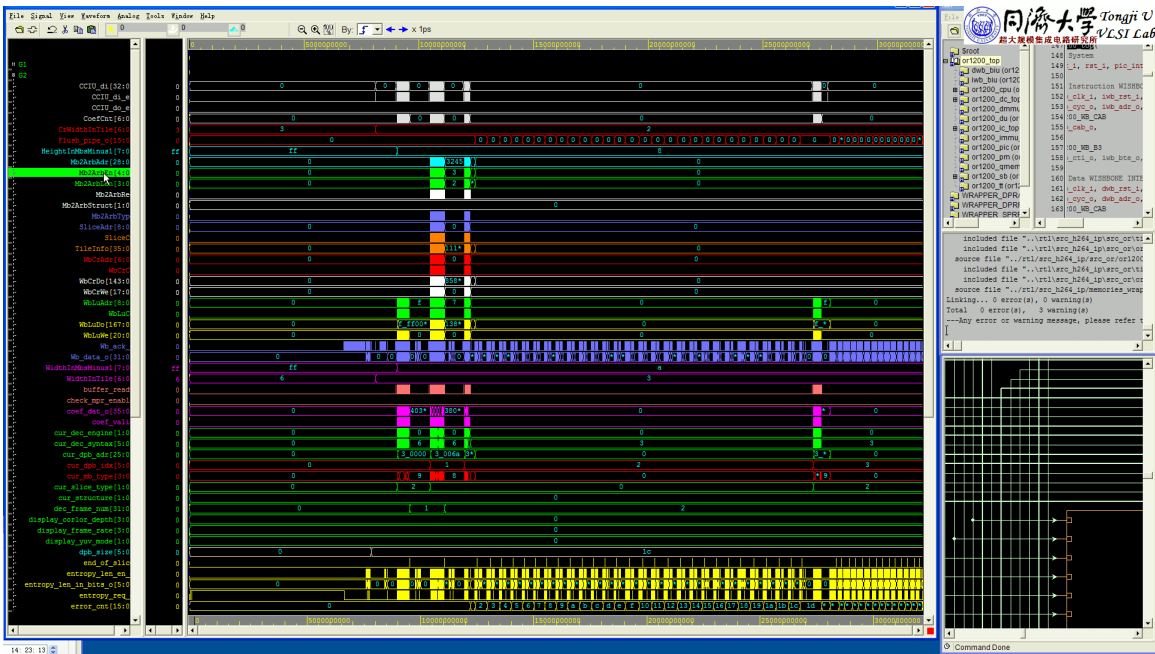
13 Appendix

Results of test that compares the accuracy of two classification algorithms. Classification algorithm A is the work presented in this thesis, while classification algorithm B is the classification algorithm presented in the research done by Wu [48].

Figure 123 shows a sample of two of the images that have been used in the test. Figure 123(a) is from a set of 40 high definition compound image bitmaps, while 123(b) is from a series of screen content images presented By Lin *et al* [44] to the Joint Collaboritve Team on Video Coding (JCT-VC) for testing HEVC coding with screen content image.



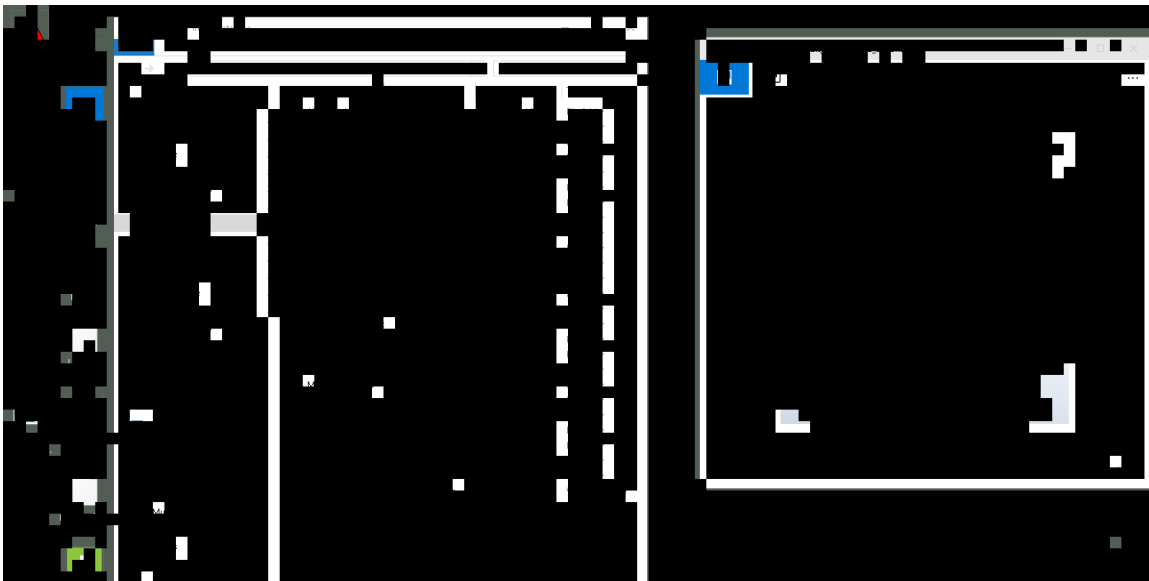
(a) Image 1



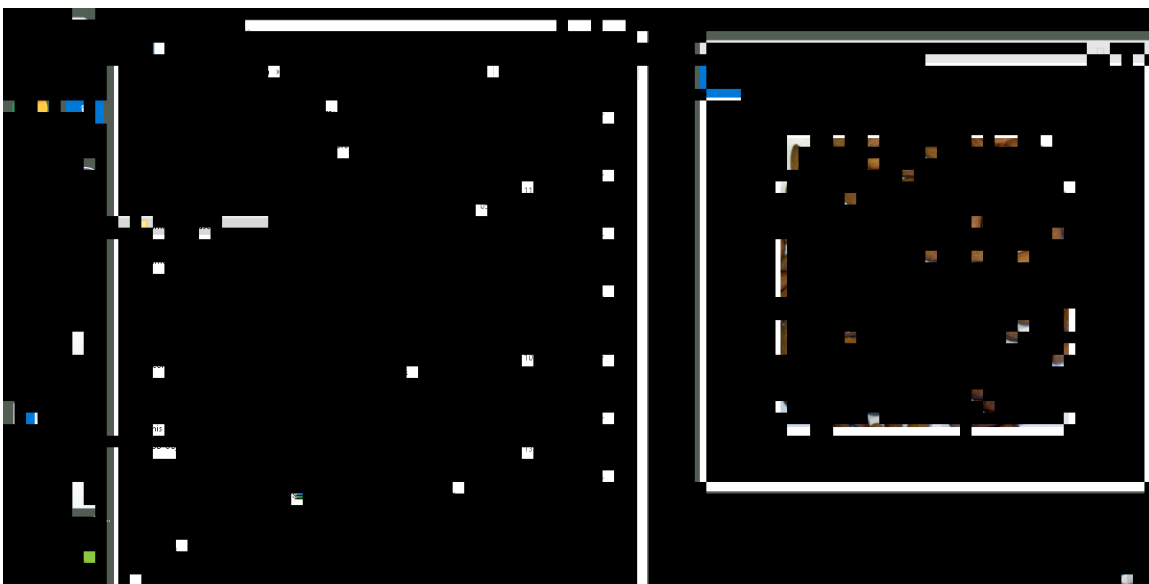
(b) Image 2

Figure 123: Two compound images used in test comparing accuracy of classification algorithm

Figure 124 shows the results of sparse block classification of image (a) presented in figure 123 for both classification algorithms. It can be observed that Algorithm B has classified in error blocks that should be classified as picture, as sparse.



(a) Algorithm A Sparse



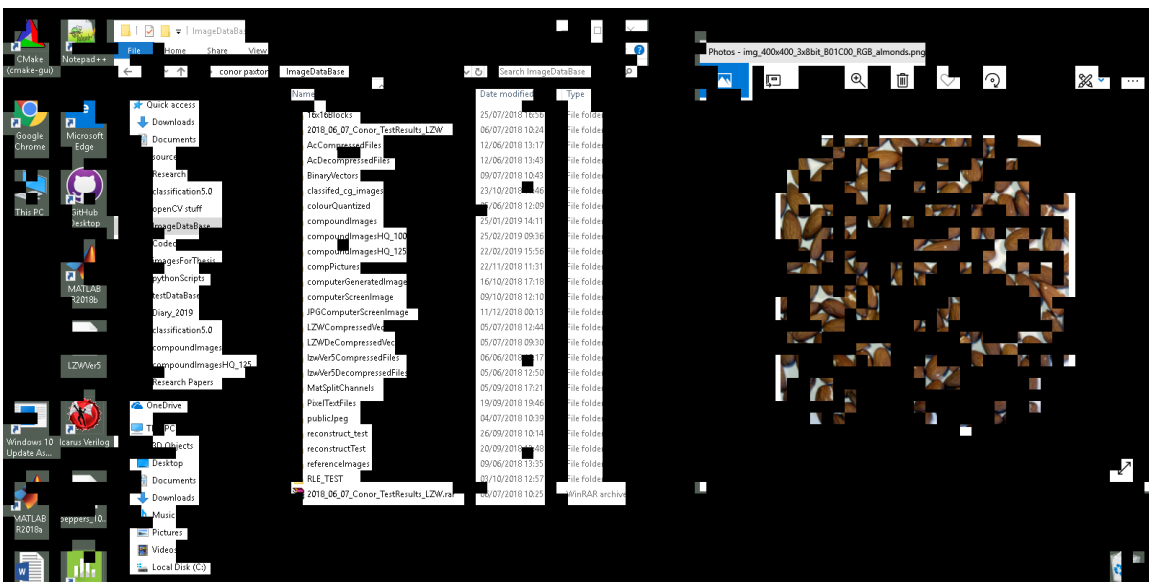
(b) Algorithm B Sparse

Figure 124: Comparing sparse classification results image one

Figure 125 shows the results of text block classification of image (a) presented in figure 123 for both classification algorithms. It can be observed that Algorithm A has successfully classified all blocks containing text with approximately 2 percent of blocks classified in error. Classification algorithm B has misclassified a significant number of blocks as text that should be classified as picture blocks.



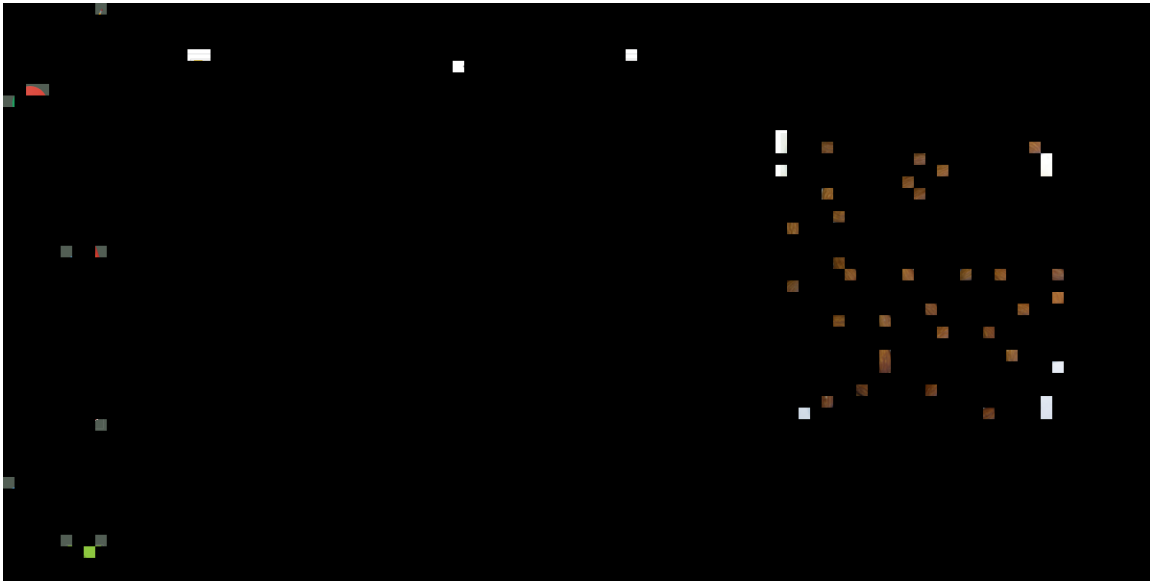
(a) Algorithm A Text



(b) Algorithm B Text

Figure 125: Comparing text classification results image one

Figure 126 shows the results of fuzzy block classification of image (a) presented in figure 123 for both classification algorithms. It can be observed that classification algorithm B has a substantial amount of blocks misclassified as fuzzy that should be classified as sparse. Classification algorithm A has less than 1 percent classification error.



(a) Algorithm A Fuzzy



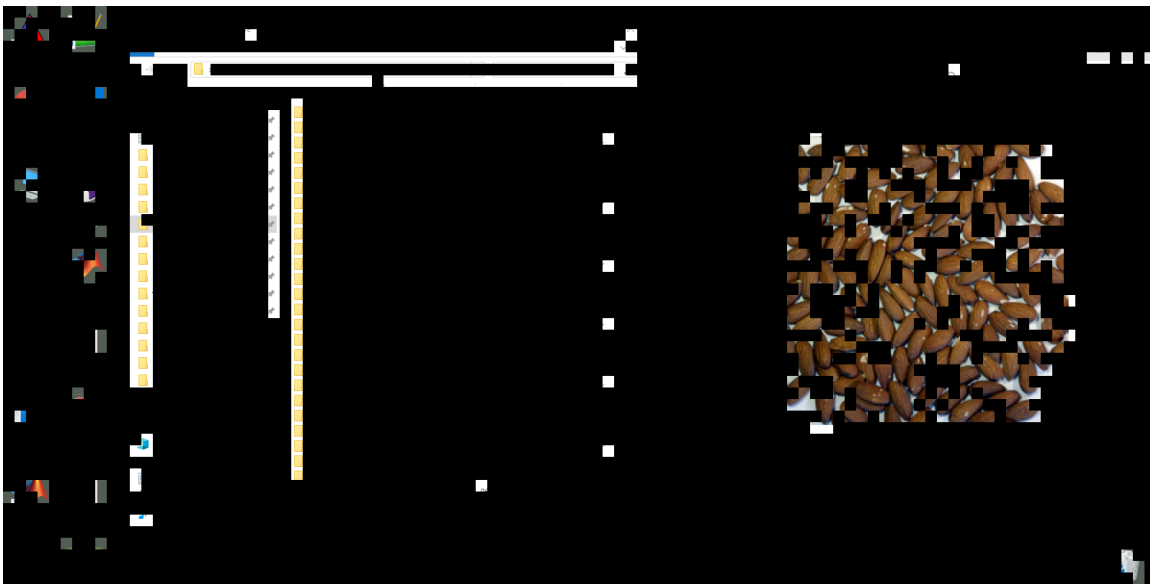
(b) Algorithm B Fuzzy

Figure 126: Comparing fuzzy classification results image one

Figure 127 shows the results of picture block classification of image (a) presented in figure 123 for both classification algorithms. It can be observed that classification algorithm A successfully classifies blocks containing natural camera capture image as picture blocks. Classification algorithm B has classified a significant amount of blocks that contain computer generated data as picture blocks.

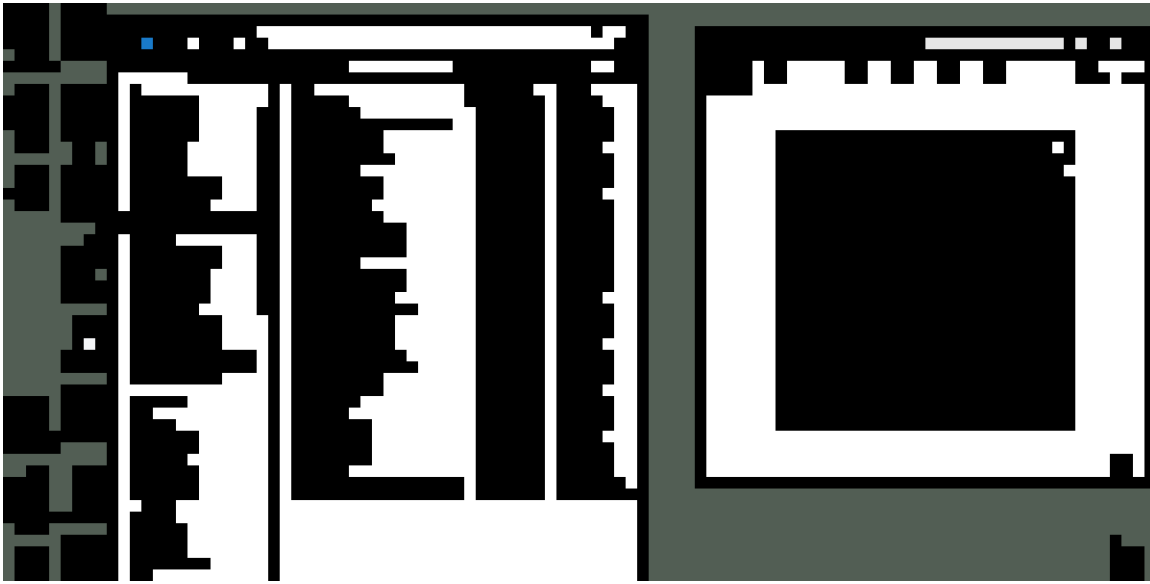


(a) Algorithm A Picture

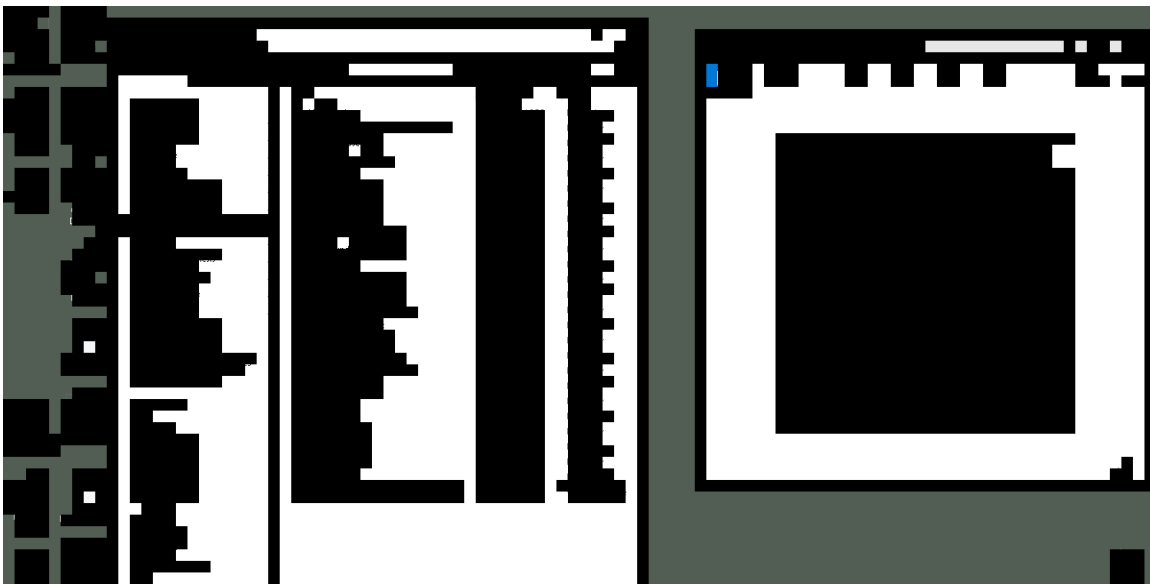


(b) Algorithm B Picture

Figure 127: Comparing picture classification results image one

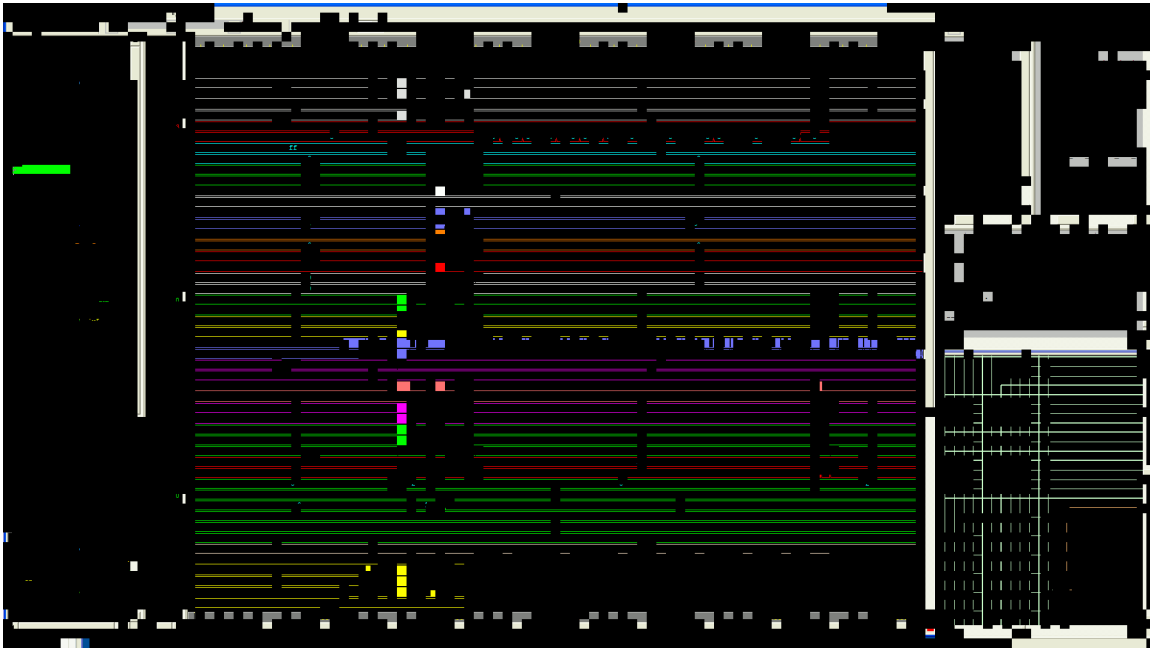


(a) Algorithm A Smooth

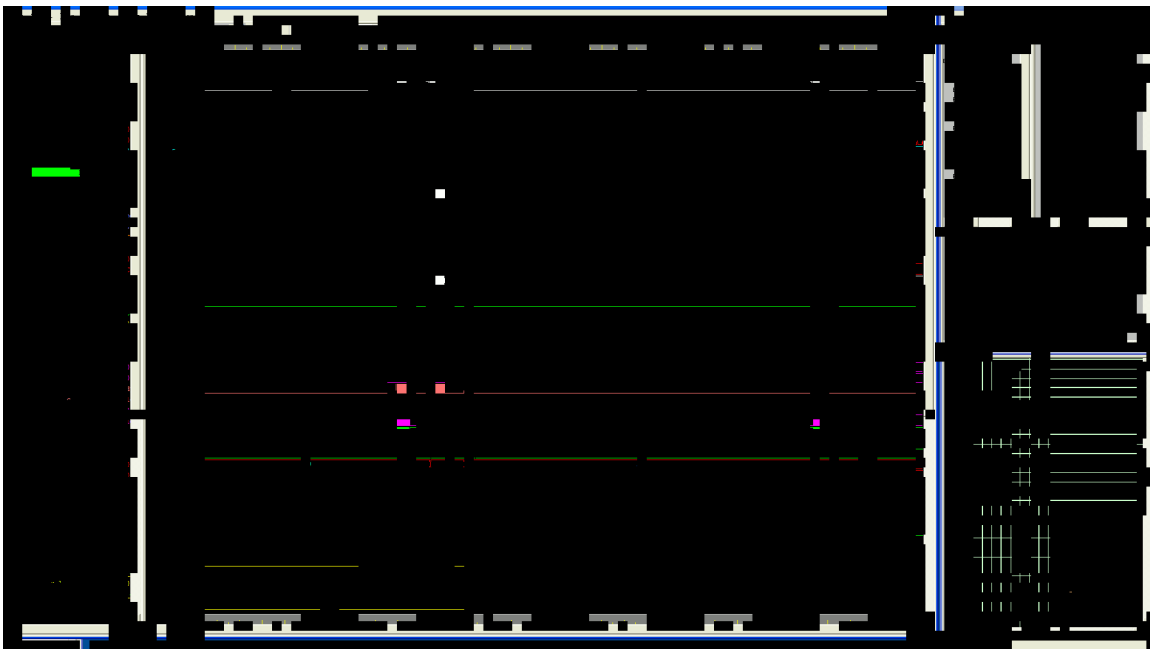


(b) Algorithm B Smooth

Figure 128: Comparing smooth classification results image one

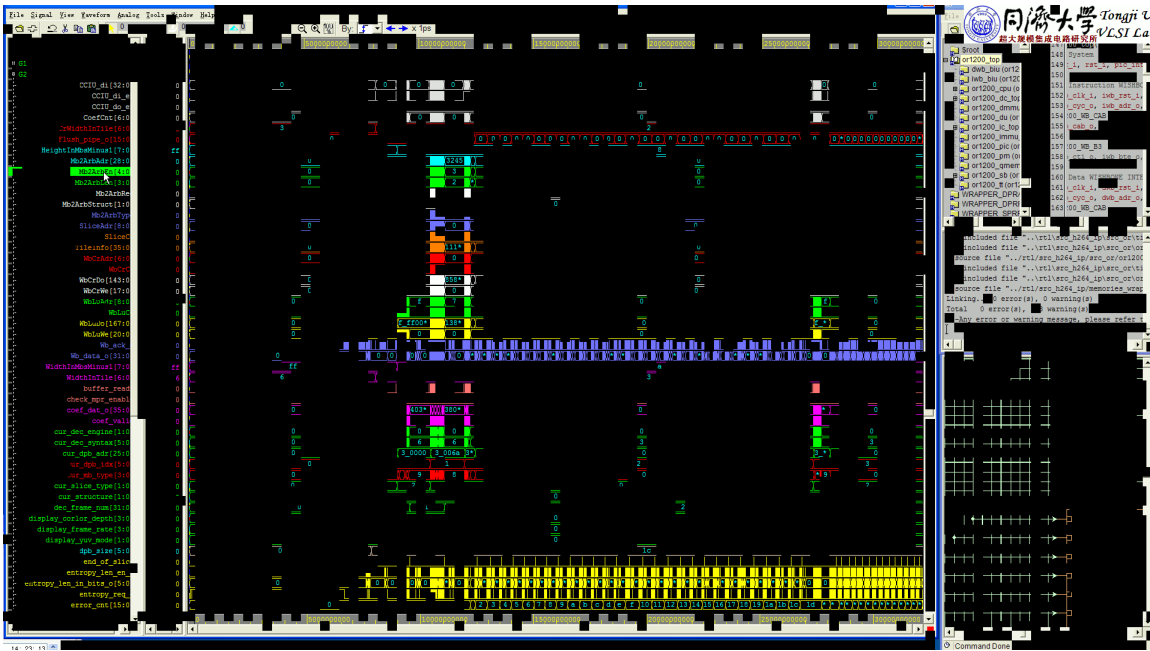


(a) Algorithm A Sparse

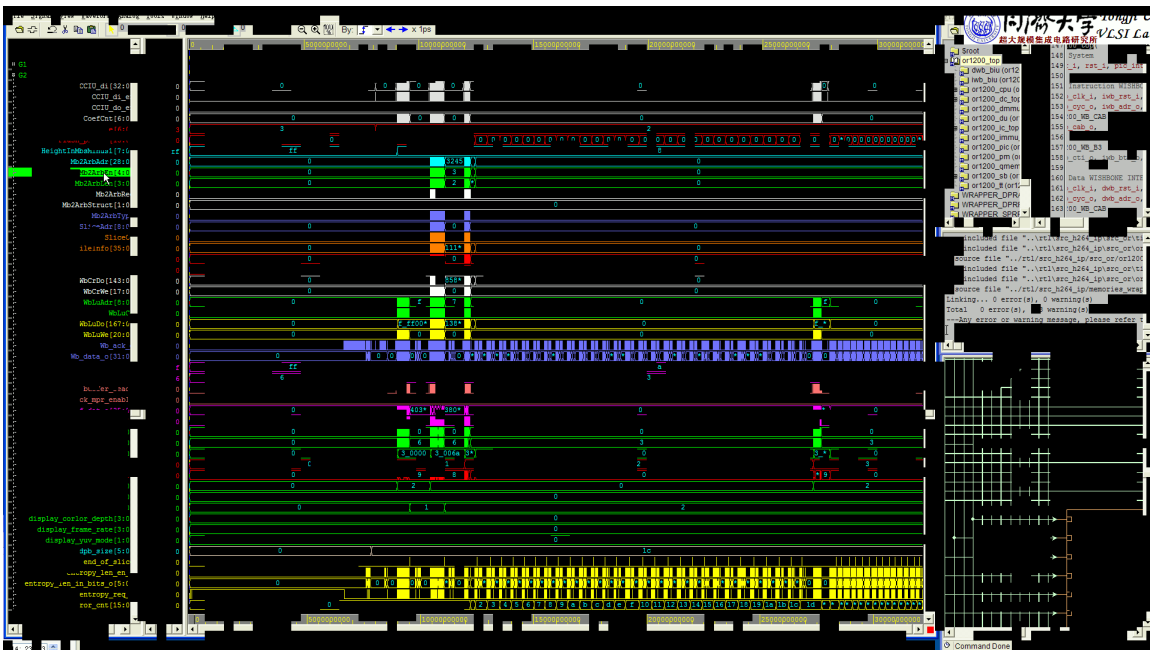


(b) Algorithm B Sparse

Figure 129: Comparing sparse classification results image one

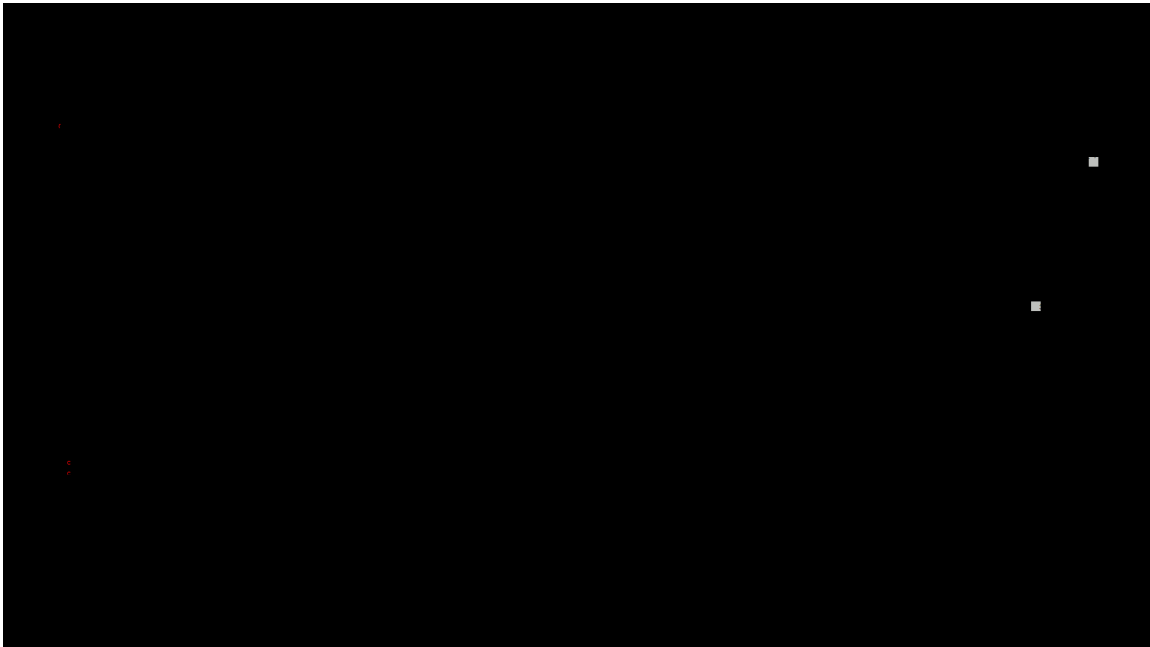


(a) Algorithm A Text

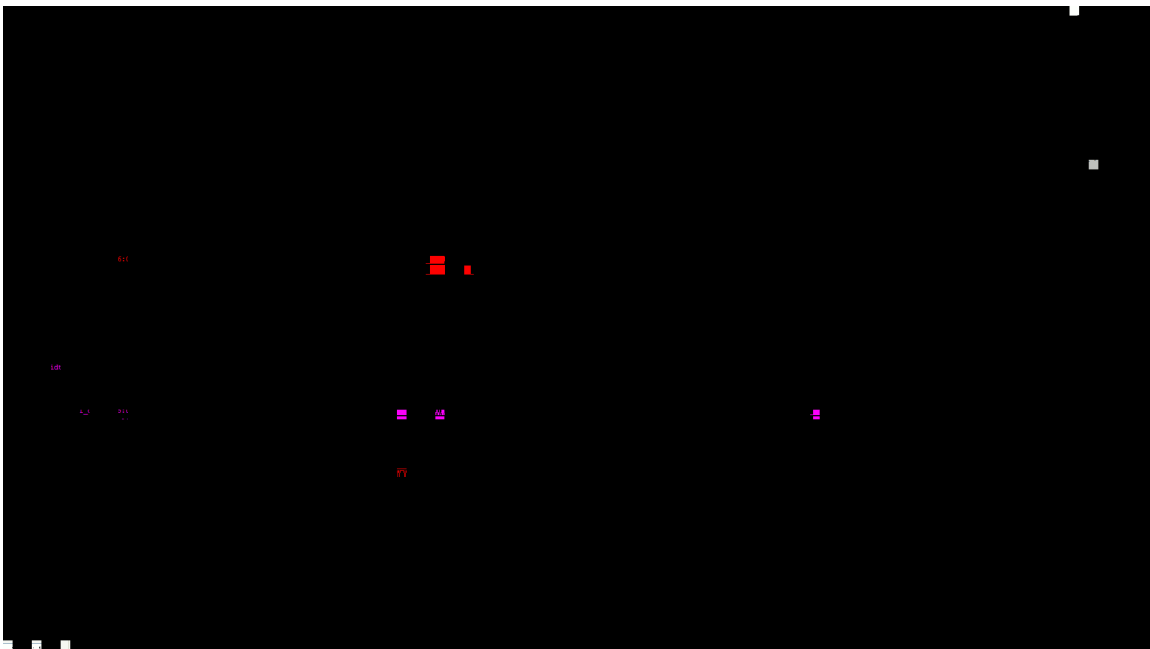


(b) Algorithm B Text

Figure 130: Comparing text classification results image one

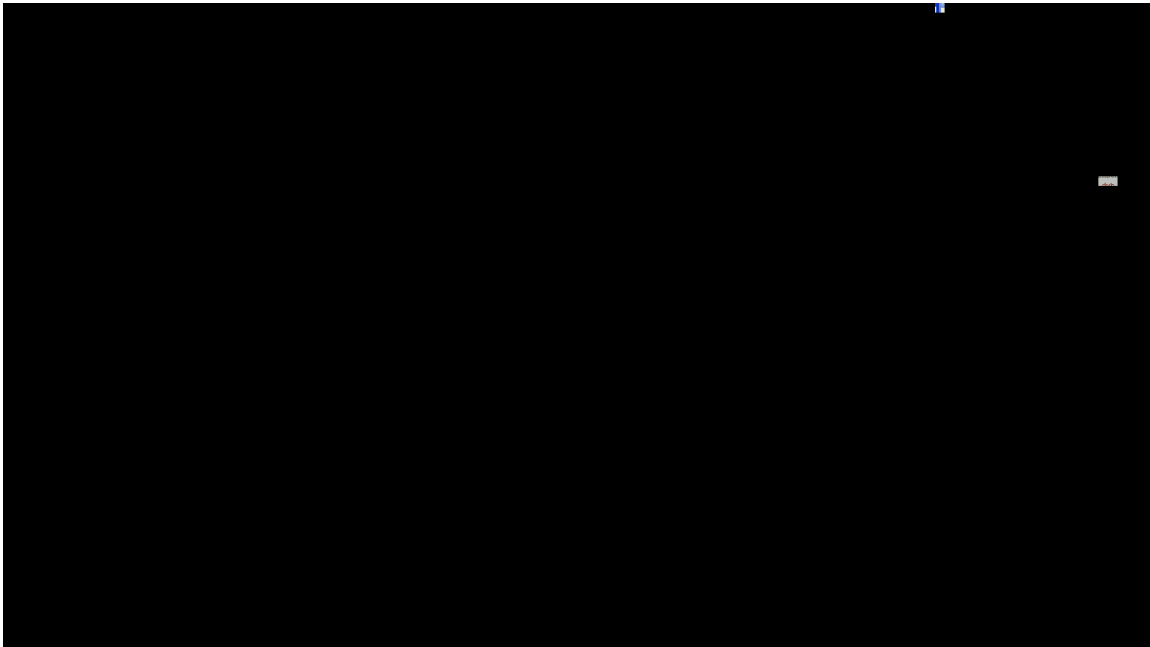


(a) Algorithm A Fuzzy

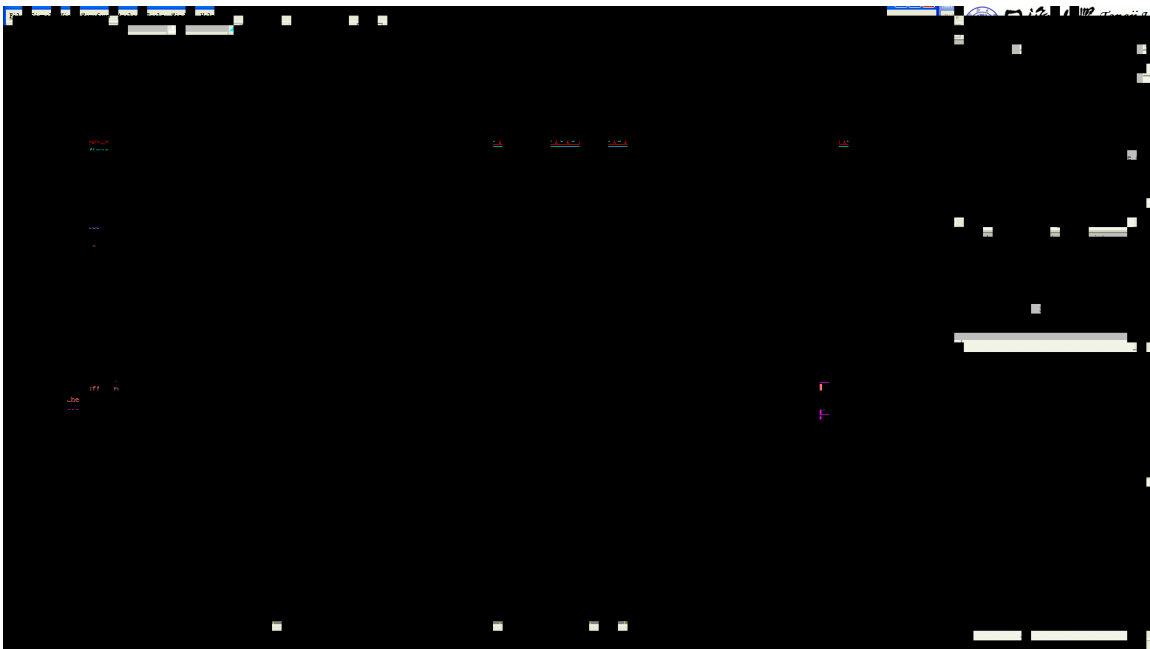


(b) Algorithm B Fuzzy

Figure 131: Comparing fuzzy classification results image one

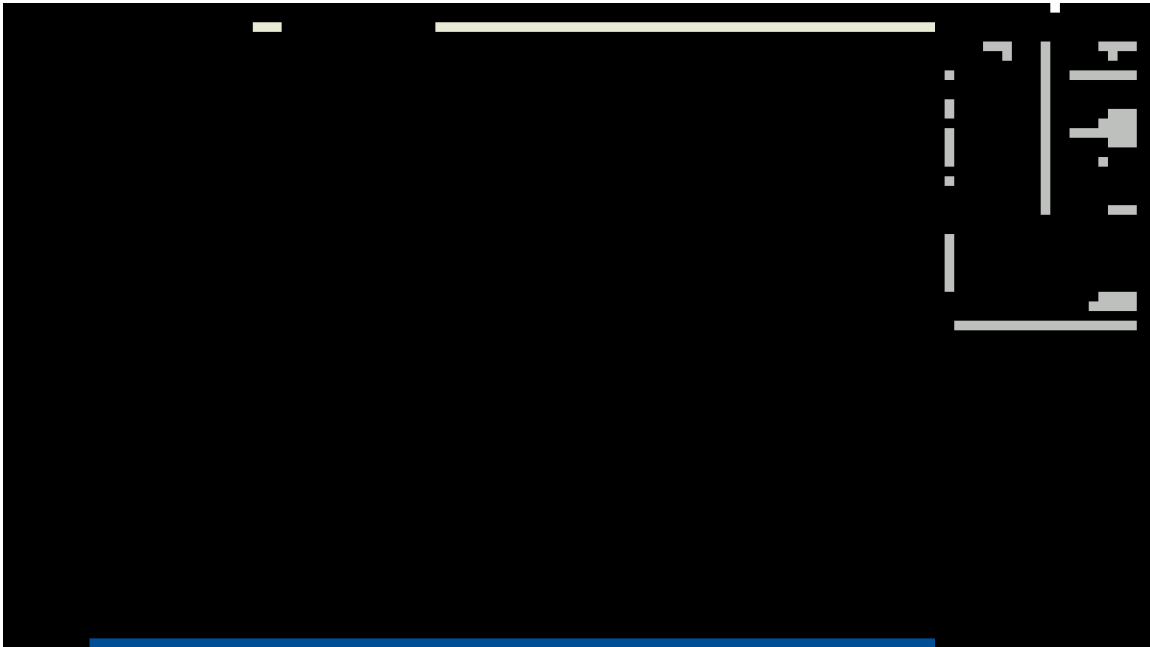


(a) Algorithm A Picture

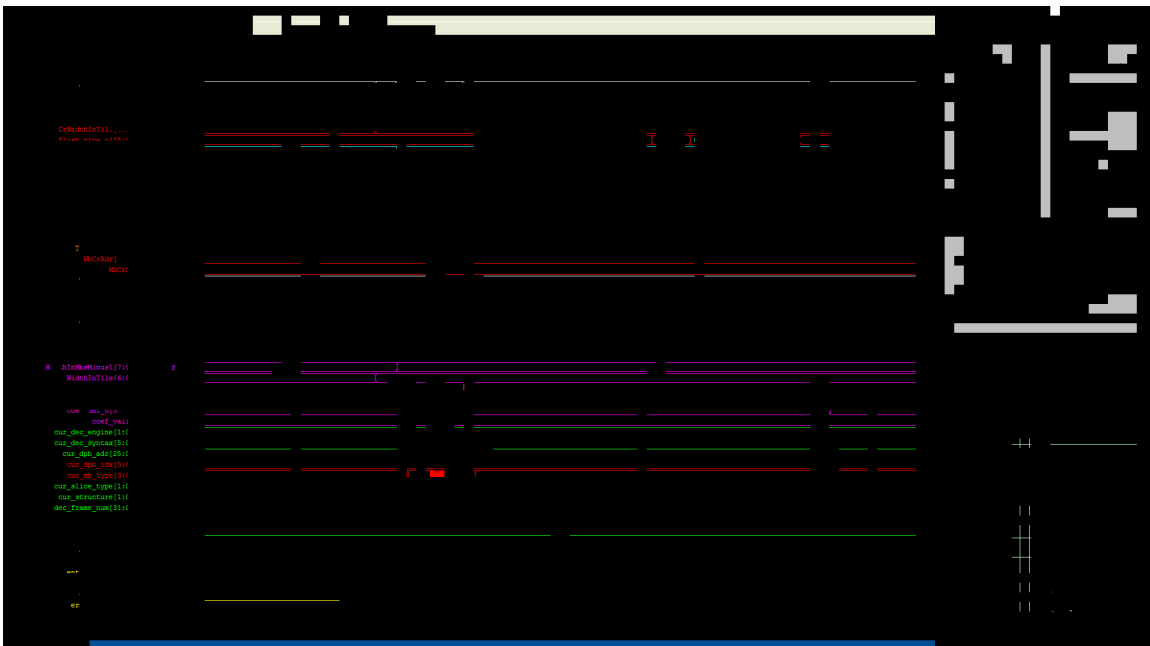


(b) Algorithm B Picture

Figure 132: Comparing picture classification results image one



(a) Algorithm A Smooth



(b) Algorithm B Smooth

Figure 133: Comparing smooth classification results image one

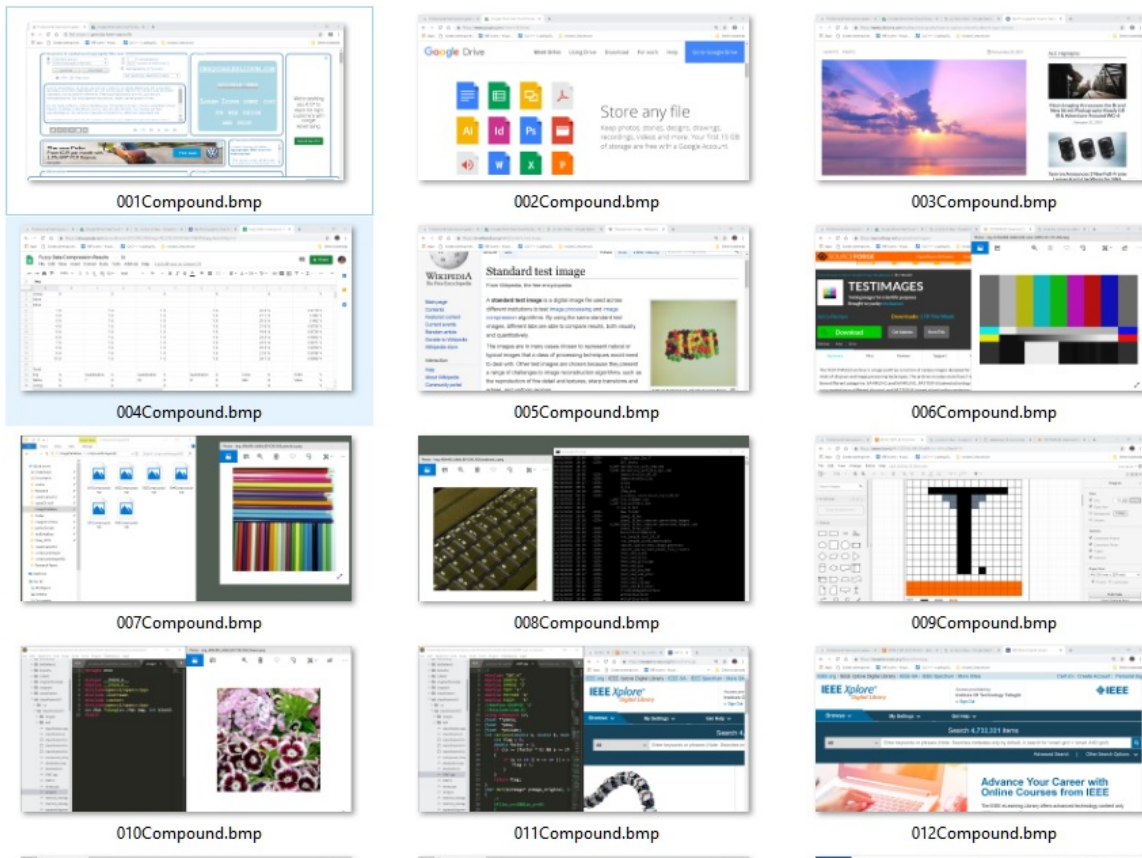


Figure 134: Compound images 1-12

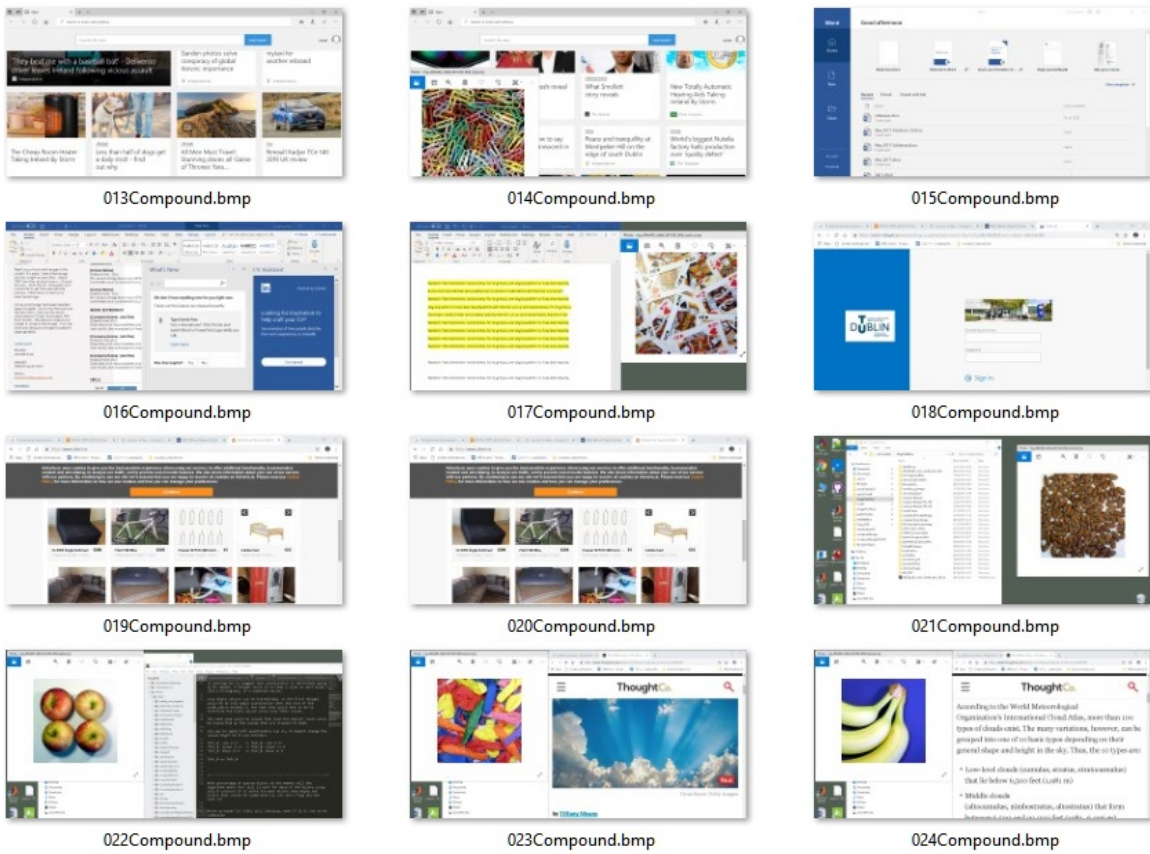


Figure 135: Compound images 13-24

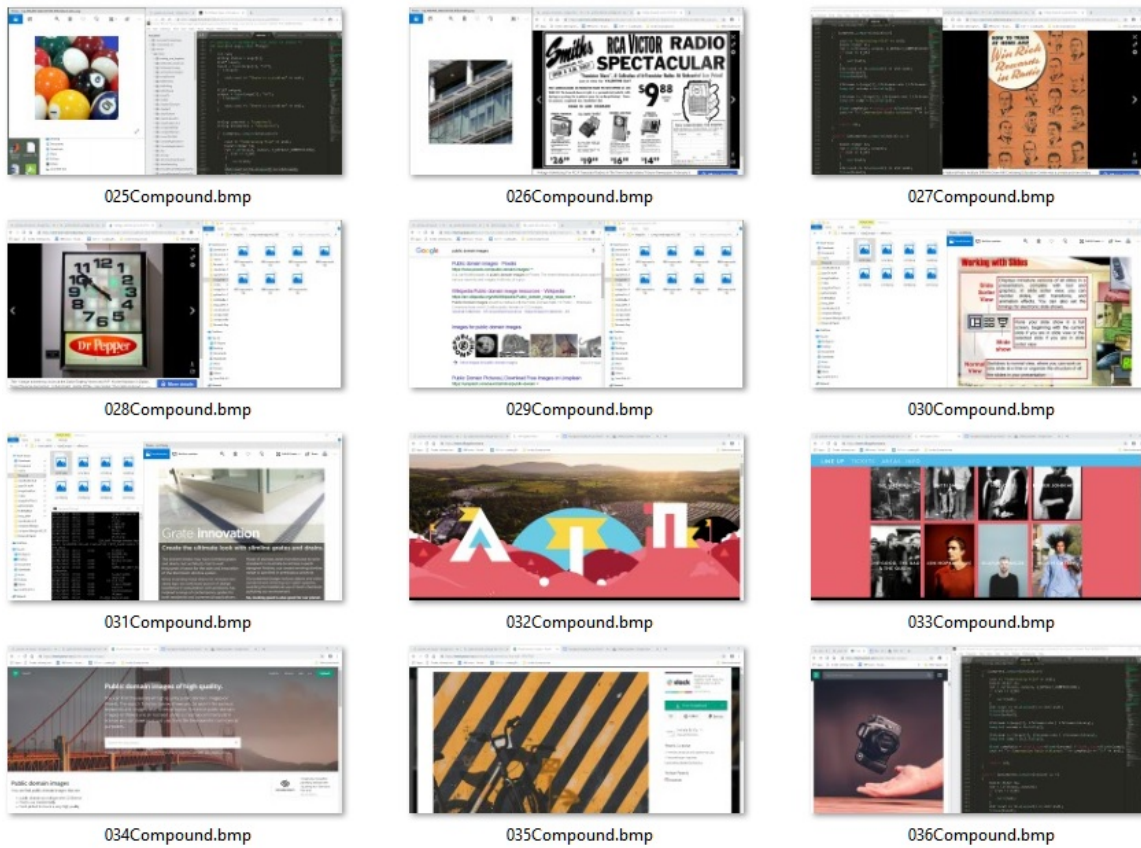


Figure 136: Compound images 25-36

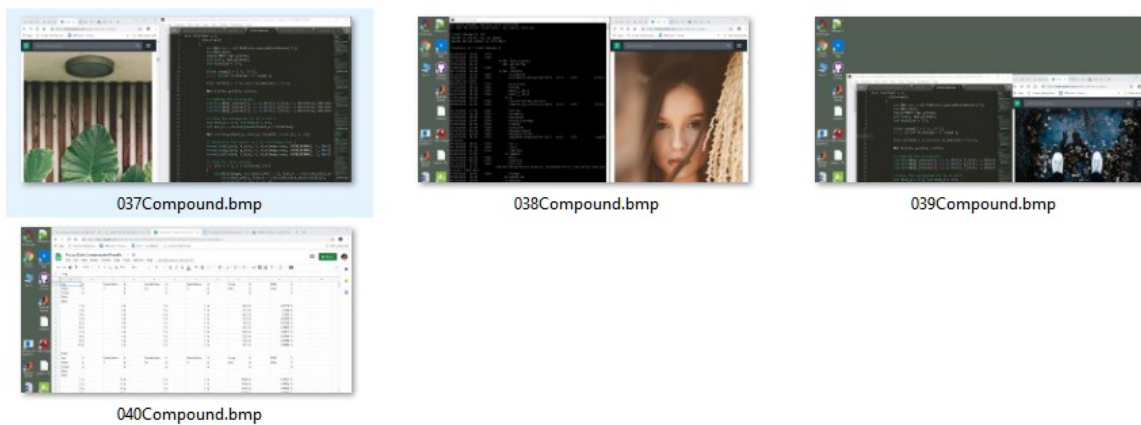


Figure 137: Compound images 37-40

References

- [1] Yukihiro Arai, Takeshi Agui, and Masayuki Nakajima. A fast dct-sq scheme for images. 1988.
- [2] Léon Bottou, Patrick Haffner, Paul G. Howard, Patrice Simard, Y Bengio, and Yann Lecun. High quality document image compression with "djvu". *J. Electronic Imaging*, 7:410–425, 07 1998.
- [3] Dingcai Cao. Chapter 10 - color vision and night vision. In Stephen J. Ryan, Srinivas R. Sada, David R. Hinton, Andrew P. Schachar, Srinivas R. Sada, C.P. Wilkinson, Peter Wiedemann, and Andrew P. Schachar, editors, *Retina (Fifth Edition)*, pages 285 – 299. W.B. Saunders, London, fifth edition edition, 2013.
- [4] C. Chen, J. Han, Y. Xu, and J. Bankoski. A staircase transform coding scheme for screen content video coding. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 2365–2369, Sep. 2016.
- [5] CISCO. *Global Networking Trends Report 2020*, 2020 (accessed December 5, 2020). https://www.cisco.com/c/m/en_us/solutions/enterprise-networks/networking-report.html.
- [6] R. L. de Queiroz. Compression of compound documents. In *Proceedings 1999 International Conference on Image Processing (Cat. 99CH36348)*, volume 1, pages 209–213 vol.1, Oct 1999.
- [7] P Deutsch. Deflate Compressed Data Format Specification version 1.3. RFC 1951, RFC Editor, May 1996.
- [8] P. Deutsch and J.-L. Gailly. Rfc1950: Zlib compressed data format specification version 3.3. 1996.
- [9] W. Ding, D. Liu, Y. He, and F. Wu. Block-based fast compression for compound images. In *2006 IEEE International Conference on Multimedia and Expo*, pages 809–812, July 2006.
- [10] W. Ding, Y. Lu, and F. Wu. Enable efficient compound image compression in h.264/avc intra coding. In *2007 IEEE International Conference on Image Processing*, volume 2, pages II – 337–II – 340, Sep. 2007.
- [11] Jarek Duda. Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. 2013.
- [12] FFMPEG. *A complete, cross-platform solution to record, convert and stream audio and video.*, 2020(accessed February 3, 2020). <https://ffmpeg.org>.

- [13] Borko Furht, editor. *Portable Network Graphics (Png)*, pages 729–729. Springer US, Boston, MA, 2008.
- [14] Rafael C. Gonzalez and Richard E. Woods. *Digital image processing*. Prentice Hall, Upper Saddle River, N.J., 2008.
- [15] J. Hu, S. Song, and Y. Gong. Comparative performance analysis of web image compression. In *2017 10th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*, pages 1–5, 2017.
- [16] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, Sept 1952.
- [17] IDG. *International Data Group Cloud Computing Survey 2020*, 2020(accessed July 19, 2020). <https://www.idg.com/tools-for-marketers/2020-cloud-computing-study/>.
- [18] ISO. *ISO/IEC 10918-1:1994: Information technology — Digital compression and coding of continuous-tone still images: Requirements and guidelines*. 1992.
- [19] ISO/IEC. Information technology — Computer graphics and image processing — Portable Network Graphics (PNG): Functional specification. Standard, International Organization for Standardization, Geneva, CH, November 2003.
- [20] "ITU-T". "itu-t rec 45".
- [21] ITU-T. Information technology - lossy/lossless coding of bi-level images. Recommendation T.88, International Telecommunication Union, November 2000.
- [22] S. E. Juliet and D. J. Florinabel. Efficient block prediction-based coding of computer screen images with precise block classification. *IET Image Processing*, 5(4):306–314, June 2011.
- [23] C. Lan, F. Wu, and G. Shi. Compress compound images in h.264/mpeg-4 avc by fully exploiting spatial correlation. In *2009 IEEE International Symposium on Circuits and Systems*, pages 2818–2821, May 2009.
- [24] C. Lan, J. Xu, Wenjun Zeng, and F. Wu. Compound image compression using lossless and lossy lzma in hevc. In *2015 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6, June 2015.
- [25] D. Marpe, G. Blattermann, G. Heising, and T. Wiegand. Video compression using context-based adaptive arithmetic coding. In *Proceedings 2001 International Conference on Image Processing (Cat. No.01CH37205)*, volume 3, pages 558–561 vol.3, 2001.

- [26] M Mozammel, Hoque Chowdhury, and Amina Khatun. Image compression using discrete wavelet transform. *International Journal of Computer Science Issues*, 9, 07 2012.
- [27] USC University of Southern California. *The USC-SIPI Image Database*, 2020(accessed September 10, 2019). <http://sipi.usc.edu/database/>.
- [28] OpenCV. *Open Source Computer vision Library*, 2020(accessed February 3, 2020). <http://opencv.org>.
- [29] Alan W. Paeth. li.9 - image file compression made easy. In JAMES ARVO, editor, *Graphics Gems II*, pages 93 – 100. Morgan Kaufmann, San Diego, 1991.
- [30] Lee Prangnell. Visible light-based human visual system conceptual model. *CoRR*, abs/1609.04830, 2016.
- [31] Lee Prangnell and Victor Sanchez. Minimizing compression artifacts for high resolutions with adaptive quantization matrices for hevc. 2016.
- [32] A. S. Ragab, A. S. A. Mohamed, and M. S. Hamid. Efficiency of analytical transforms for image compression. In *Proceedings of the Fifteenth National Radio Science Conference. NRSC '98 (Cat. No.98EX109)*, pages B16/1–B1610, Feb 1998.
- [33] Iain Richardson. H.264 and mpeg-4 video compression : video coding for next-generation multimedia / iain e. g. richardson. *SERBIULA (sistema Librum 2.0)*, 01 2004.
- [34] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, Jan 1998.
- [35] A. H. Robinson and C. Cherry. Results of a prototype television bandwidth compression scheme. *Proceedings of the IEEE*, 55(3):356–364, March 1967.
- [36] A. Said and A. Drukarev. Simplified segmentation for compound image compression. In *Proceedings 1999 International Conference on Image Processing (Cat. 99CH36348)*, volume 1, pages 229–233 vol.1, Oct 1999.
- [37] Asadollah Shahbahrami, Ramin Bahrampour, Mobin Sabbaghi Rostami, and Mostafa Ayoubi Mobarhan. Evaluation of huffman and arithmetic algorithms for multimedia compression standards. *ArXiv*, abs/1109.0216, 2011.
- [38] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.
- [39] Vivienne Sze, Madhukar Budagavi, and Gary J. Sullivan. *High Efficiency Video Coding (HEVC): Algorithms and Architectures*. Springer Publishing Company, Incorporated, 2014.

- [40] International Telecommunication Union. Series G: Transmission systems and media, digital systems and networks. One-way transmission time (G.114). Recommendation, JUNE 2003.
- [41] International Telecommunication Union. Studio encoding parameters of digital television standard 4:3 and wide-screen 16:9 aspect ratios (ITU-R BT.601). Recommendation, MARCH 2011.
- [42] Georgia State University. Perspective view of lens structure, 2016.
- [43] Bovik Wang. *Structural Similarity Index Implementation scripts.*, 2008(accessed Marc 21, 2020). <https://www.cns.nyu.edu/~lcv/ssim/>.
- [44] S. Wang and T. Lin. Update on full-chroma (yuv444) screen content test sequences of jctvc-h0294. In *AHG:7 JCTVC-K0207, Shanghai*, Oct 2012.
- [45] S. Wang and T. Lin. Compound image compression based on unified lz and hybrid coding. *IET Image Processing*, 7(5):484–499, July 2013.
- [46] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, June 1984.
- [47] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h.264/avc video coding standard. *IEEE Trans. Cir. and Sys. for Video Technol.*, 13(7):560–576, July 2003.
- [48] K. Wu, R. Gahan, and P. O’Friel. Block-based classification method for computer screen image compression. In *2018 29th Irish Signals and Systems Conference (ISSC)*, pages 1–6, June 2018.
- [49] Song Zhao, Yan Xu, Hengjian Li, and Heng Yang. A comparison of lossless compression methods for palmprint images. *Journal of Software*, 7, 03 2012.
- [50] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, April 2004.
- [51] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.
- [52] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, September 1978.