Hardware Efficient Lossless Realtime Compression of Raw Image Data

MÅNS ÅHLANDER & AXEL JONSSON
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Hardware Efficient Lossless Realtime Compression of Raw Image Data

Måns Åhlander & Axel Jonsson
mans.ahlander@gmail.com axelolof94@gmail.com

Department of Electrical and Information Technology
Lund University

Supervisor: Stefan Höst

Co-Supervisor: Imran Iqbal

Examiner: Maria Kihl

May 31, 2018

© 2018
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

When transmitting data it is often desired to lower the bitrate in the transmission. In video capture devices, such as video cameras, with a high resolution and high capturing rate the bitrate in the transmission is high and it is often desired to lower this bitrate. Some of the constraints that were put on the compression techniques was that no information could be lost in the compression, no significant amount of delay may be introduced and that the compression techniques should have as small hardware requirements as possible. Another limitation was that the compression techniques were limited to store less than one frame of memory. One exception of this was tested to see how the compression ratio would be improved when compressing using more than a frame of memory. How transmission in a single directional transmission link with compressed data could be done has also been investigated during this thesis.

The compression techniques in this paper are two-step based compression techniques called linear prediction coding. The first step is to predict the pixel values in the image and the second step is to encode the prediction error. In order to recover the image, these codes need to be decoded to recover the prediction error which can then be used to reconstruct the image. This is an efficient compression technique due to that there is much redundant information in an image and instead of encoding each pixel value it is more efficient to encode the differences between pixels. The differences contains the same information but may be encoded using shorter codewords.

The average compression ratios for the techniques presented in this paper achieves a compression ratio close to 40% when compressing raw image data. If noise reduction techniques are applied to the image to compress, the compression techniques may improve their compression ratio with over 10%. Another way to improve the compression, without losing any significant amount of information, is presented in this paper and it is done by measuring the noise in the image and then allowing some quantization based on this measurement.

A solution to the problem on how to transmit data in a safe way from the encoder to the decoder have been suggested. The data is sent in packets where each packet contains a header field and a data field. The prediction errors are encoded in the data field and the number of symbols encoded in the data field are represented in the header field. These two got a fixed number of bytes and by using error correcting code, the data can be safely be transmitted.

Popular Science Summary

Efficient Lossless Video Stream Compression

Modern video with high resolution puts high demands on the transmission of data, which is specifically true for image data coming from a camera sensor. For efficient transmission of data, compression techniques can be used.

Data compression is a technique that allows one to reduce the amount of space, in bits, the data needs, and it must be possible to decompress the compressed data to retrieve the original data. The goal in this project is to find and evaluate different compression techniques that operates on very limited hardware. This limitation excludes many common techniques such as the MPEG standards. When evaluating several primitive lossless compression techniques a compression around 40% was achieved in our tests. This can be compared with the well known Lossless *JPEG* standard that provided results with compression around 30 to 40%.

During lossless compression no information may be lost, unlike with lossy compression which is a more common technique in video and image compression. This was a limitation because the raw images from the camera sensor must reach a processing unit untouched so the processing unit can provide the best quality in the image. To ensure that no information were lost in transmission a transmission packet protocol was designed which was used to transmit the data in a safe way.

The lossless compression techniques developed and evaluated in this thesis are all based on the same two steps. The first step was a prediction method, that uses the fact that neighboring pixels in an image are usually very similar. Instead of sending every pixel value, trying to estimate a pixel value with already sent pixels and then the difference between the actual pixel value and its estimation was sent. Before transmission, the difference was encoded in a smart way, which was the second step of the compression. The most common values were sent with short codewords while uncommon values were sent with longer codewords. The first step provides data for the encoder whose probability distribution is very skewed, which helps the encoder to send less long codewords. This means that combination of these two steps is what compresses the data.

Because the data is raw sensor data, an image that was compressed was not preprocessed in any way, meaning that the image contained noise, which was extra unwanted information in the image. The noise appears with a random value in

every pixel, which lowers the quality of the image. This random noise in every pixel affects the compression ratio, and without the noise the compression could be improved by over 10%. Due to this noise which makes the pixel values inaccurate, the pixel values may be near-lossless quantized. Meaning that every pixel value may be quantized based on the noise level in the pixel. This introduces error in the pixel value but since this introduced error is much lower than the present noise level of the pixel, it does not significantly reduce the quality of the image. Introducing this near lossless quantization improves the compression ratio.

The thesis is written by Måns Åhlander and Axel Jonsson and can be found at *https://www.overleaf.com/14590377fqvfdfngjrrx#/57088500/* under the title "Hardware Efficient Lossless Realtime Compression of Raw Image Data".

Table of Contents

1 Introduction **1**
- 1.1 Background . . . 2
- 1.2 Data Transmission . . . 3
- 1.3 Problem Formulation . . . 4
- 1.4 Thesis Outline . . . 4

2 Theory **7**
- 2.1 Correlation . . . 7
- 2.2 Entropy . . . 9
- 2.3 Prediction . . . 9
- 2.4 Encoding . . . 12
- 2.5 Horizontal and Vertical Blanking . . . 18
- 2.6 Near-Lossless Compression . . . 19
- 2.7 Camera Serial Interface 2 . . . 20
- 2.8 ISO Compression Standards . . . 21
- 2.9 The Burrows-Wheeler Transform . . . 24
- 2.10 Forward Error Correction . . . 26

3 Implementation of Algorithms **29**
- 3.1 Correlation Analysis . . . 29
- 3.2 Predictor Implementation . . . 29
- 3.3 Encoders . . . 33
- 3.4 Benchmark . . . 34
- 3.5 The Run-length Encoder . . . 35
- 3.6 Hardware Requirement Estimation . . . 35
- 3.7 Implementation of Near-Lossless Compression Technique . . . 36
- 3.8 Horizontal and Vertical Blanking . . . 38
- 3.9 Lossy JPEG . . . 38
- 3.10 Comparison with PNG . . . 38
- 3.11 Error Handling . . . 38

4 Results **41**
- 4.1 Correlation Analysis . . . 41
- 4.2 Benchmark . . . 45

4.3 Run-length Encoding . . . 49
4.4 Sensor Noise . . . 49
4.5 Analysis of Horizontal and Vertical Blanking . . . 52
4.6 Lossy JPEG . . . 53
4.7 Comparison with PNG . . . 56
4.8 Computational Complexity . . . 56
4.9 Error Handling . . . 57

5 Discussion **61**
5.1 Correlation Analysis . . . 61
5.2 Benchmark . . . 63
5.3 Hardware Demands . . . 66
5.4 The Run-length Encoder . . . 68
5.5 Noise Handling . . . 69
5.6 Worst-case Statistics versus Standard Deviation . . . 70
5.7 Prediction Technique Evaluation . . . 71
5.8 Horizontal and Vertical Blanking . . . 74
5.9 Lossy JPEG . . . 74
5.10 Comparison with PNG . . . 75
5.11 Arithmetic Coding . . . 75
5.12 Transforms . . . 76
5.13 Effects of Error Handling . . . 76
5.14 Further work . . . 78

6 Conclusions **79**

References **81**

A Images Used for Testing Spatial Compression Techniques. **85**

B Images Used for Testing Temporal Compression Techniques. **93**

List of Figures

1.1 Bayer color filter. 3

2.1 Example of prediction error from an image. 18
2.2 Percentage of ones for different bit levels. 18
2.3 Pixels used for predicting the current sample. 22
2.4 Flow scheme for the Burrows-Wheeler compression algorithm. U is the input sequence which should be compressed and u is the compressed sequence. 24

3.1 Pixels used for predicting the current sample. 31

4.1 Illustration of correlation in image **test12.pgm**. 42
4.2 Illustration of correlation in image **test24.pgm**. 42
4.3 probability distribution of pixel values. 43
4.4 Probability distribution of two sided prediction error. 44
4.5 Probability distribution of one sided prediction error. 45
4.6 Compression ratio for different amount of sections that each row are divided into. 48
4.7 Methods used in the combined predictor when using 128 section for each row. 48
4.8 Compression ratio for different SQNR parameter values. 51
4.9 Relative Compression ratio for different SQNR parameter values. . . . 52
4.10 The image **test1.pgm** with horizontal and vertical blanking. 53
4.11 Compressed image using $quality = 100$. 54
4.12 Compressed image using $quality = 10$. 55
4.13 Original image. 56
4.14 Example of how a bit error in the transmitted bitstream affects the decoded image. 58

A.1 First set of images used for the benchmark application. 85
A.2 Second set of images used for the benchmark application. 86
A.3 Third set of images used for the benchmark application. 87
A.4 Fourth set of images used for the benchmark application. 88
A.5 Fifth set of images used for the benchmark application. 89

A.6 Sixth set of images used for the benchmark application. 90
A.7 Seventh set of images used for the benchmark application. 91

B.1 First set of temporal images used for the benchmark application. . . 94
B.2 Second set of temporal images used for the benchmark application. . 95
B.3 Third set of temporal images used for the benchmark application. . . 96

List of Tables

2.1 Compression scheme for *Camera Serial Interface 2* 20
2.2 Prediction for lossless JPEG coding. 22
2.3 Prediction for lossless JPEG coding. 23
2.4 The Burrow-Wheeler Transform, the fifth row corresponds to the original sequence of letters. 25

3.1 Prediction for lossless JPEG coding. 31

4.1 Average compression ratio (%) benchmark results. 46
4.2 Worst-case compression ratio (%) benchmark results. 46
4.3 Entropy benchmark results. 47
4.4 Standard deviation of compression ratio benchmark results. 47
4.5 Comparison of compression ratio (%) using *Time* predictor with other predictors. 49
4.6 Comparison of compression ratio (%) using run-length encoding with other encoders using *Mean2L* predictor. 49
4.7 Compression ratios (%) for averaged image. 50
4.8 Compression ratios (%) for non-averaged image. 50
4.9 Compression ratio (%) for image with and without horizontal and vertical blanking. 53
4.10 Compression ratio (%) using *Lossy JPEG* for different quality settings. 54
4.11 Estimated computational cost for the different techniques. 57
4.12 How dividing a video frame into blocks effects the compression ratio of the *Mean2L-Adaptive Golomb-Rice* combination. 59

Chapter 1
Introduction

In various applications large data rates are a problem that leads to risk in transmission errors and might limit the bandwidth of the transmission channel. One area where the amount of transmitted data is huge and the data rate is a problem, is within cameras between the image sensor and the backend processor. If the data transmitted from the image sensor should be compressed, some additional hardware would be needed to be implemented at the sensor. It could be desirable to use as little hardware as possible while achieving as high compression ratio as possible. This is done in order to reduce product cost, risk of overheating and in order to keep the sensor unit small. To be able to determine which hardware that should be implemented compression techniques has been simulated and and their hardware demands evaluated. It has been done in order to find a technique with satisfying and stable compression ratio while at the same time has low computational complexity and memory requirements. Stable compression ratio means that the bitrate is, in the worst case, not higher than the original bitrate and that the average compression ratio to worst case compression ratio rate is not too large. From the code used for implementing the compression techniques the hardware requirements has been estimated.

Originally, the limitation on the hardware was not to use more memory than for one frame and correspondingly the compression algorithm is not allowed to take more time than one frame time interval unit. Other limitations were to only have a few computations for each pixel, so that the compression techniques does not require to much processing power, and the compression should be lossless in order to not lose any data or change any data during the compression. The requirement to not lose or change any data when compressing the data is an important limitation because data that is changed can be seen as noise that is introduced in the image, that will reduce the quality of the image in restoration at the processing unit. A desirable property for the compression would be that it produces good compression results for all types of images. The image compression should always work and the products using it should always be considering the worst-case scenario.

There is a lot of research in the field of compression for all sorts of data, both on lossless and lossy compression. Only lossless and near-lossless research has been relevant for this thesis and the lossless research can be divided into multiple categories, such as, batch mode compression, video compression using correlation between frames, image compression techniques, and more simple algorithms.

Video compression techniques using correlation between more than two frames have not been studied due to the memory and time delay limitations and batch mode compression techniques are also not of any interest, no more than as for comparison. Standard image compression technique such as the online *JPEG-ls* compression technique has been studied, which is a composition of a prediction algorithm and *Golomb-Rice* encoding. Prediction is described in Section 2.3 and *Golomb-Rice* is described in Section 2.4.3. Based on the idea of *JPEG-ls* other techniques has been developed using different prediction techniques and different encoding algorithms. Simpler and more general lossless compression techniques such as *Huffman* coding, *arithmetic* coding and *run-length* coding have also been studied and evaluated.

1.1 Background

In digital cameras, there is a complex chain of image processing steps, starting from capturing of the image, going through, for instance, white balancing and noise filtering all the way to compression of the image/video and then streaming it out or saving into the memory. The image is captured using image sensor and then this raw image is transmitted to the backend processor for processing. Transmission of the raw image from the sensor to the backend processor requires a very high speed link. This is particularly true for video cameras that provide frame rates of 30 frames per second or higher together with HD (720p) resolution or higher. This high bandwidth requirement imposes several challenges and reducing the bandwidth would result in reducing the risk of transmission errors.

The purpose of this thesis is to evaluate different lossless compression techniques in association with Axis communications, which could be used for lowering the bandwidth of data transmitted from the camera sensor. This purpose was extended to also include near-lossless methods. The guidelines from Axis Communications were that the duration of the compression should not exceed more than one frame time interval, the memory is restricted to the size of one frame and the computational cost for the compression techniques should be kept as small as possible.

Each pixel from the image sensor is represented by 12 bits, which means each pixel will be able to have an intensity level between 0 and 4095. When the image sensor captures a frame, each pixel row is sent after another. The pixel in the upper left corner is sent first, then the pixel to the right of it is sent and so on until the whole row is sent. After this the next row is sent in the the same way and this is repeated until all pixels of the frame is sent. On each pixel of the sensor there is a color filter that lets only light of a certain color pass. The filter is a grid formed in a way that every other pixel on a row receives light of the same color. The color grid being used is shown in figure 1.1, where the letter in each square represent its color. Red is represented by R, green is by G and blue by B. This filter is called the Bayer color filter[14], where each pixel is represented in gray scale but using Bayes color scheme can the color for each be restored through interpolation using surrounding pixels.

G	B	G	B	G	B	G	B	G	B
R	G	R	G	R	G	R	G	R	G
G	B	G	B	G	B	G	B	G	B
R	G	R	G	R	G	R	G	R	G
G	B	G	B	G	B	G	B	G	B
R	G	R	G	R	G	R	G	R	G

Figure 1.1: Bayer color filter.

Compressing data is about encoding data using fewer bits than the number of bits that originally each symbol is represented by, which can be done using encoding algorithms. Encoding is a part of information theory, where each symbol, or sequence, is given a code based on how probable the symbol is. In general, a common symbol will be given a short code consisting of few bits, while symbols which are not as likely to occur are given a longer code using more bits. The encoder and decoder needs to share the same statistical data so that the compressed data can be decompressed data back to its original form. This can be done either with perfect reconstruction, meaning that it is lossless compression, or with some loss of information in the compression, which is called lossy compression. Even if lossy compression looses some information in the compression, the reconstructed image, according to the human eye, will have the same quality. Lossy compression techniques are often not suppose to reduce the Quality of Experience which means that a human should not see any differences between the original image and an image which have been compressed using a lossy compression technique.

There is often much correlation between pixels in an image, as a pixel is often very similar to its neighboring pixels. This can be seen as that the neighboring pixels shares the same information. In order to reduce the information required to represent each pixel it is more efficient to represent the difference between the values of the pixels instead of the actual pixel value since they have the same information. This is a common lossless compression method if one assume that it requires fewer bits to represent difference between the pixel values rather than the actual pixel value. This assumption in image- and video compression are in general an accurate assumption, which will be explained later in this paper.

1.2 Data Transmission

In order to transmit the data from an encoder to a receiver in a safe way, a transmission protocol needs to be used. There are several reasons to why some form of protocol is needed in any type of transmission, where in the case of sending encoded data it is of high importance.

The bits sent from the encoder to the decoder should be sent using some sort of protocol that could check if anything went wrong in transmission. This is needed because if there would be a bit error in the encoded bitstream of a video frame, then that error might propagate and interfere with the decoding of following symbols.

The decoder would decode the symbol inaccurately in the bitstream where the bit error occurred. This means that that symbol would get a different value from what it should have been. Also, because the symbols might be encoded using different amount of bits in its codeword it is likely that the decoder would read a different amount of bits than the original codeword. This would lead to that the encoder and decoder would become unsynchronized with each other. What will happen then is that the next codeword will be starting to be read at the wrong position and therefore the codeword will represent a different symbol than it was supposed to do. The effects of this is that the decoded symbol error propagates until the encoder and decoder performs some form of reset to become synchronized again.

1.3 Problem Formulation

The goal with this thesis is to develop and evaluate different lossless compression techniques for high speed image data and determine how the hardware limitations affects the compression ratio in order to find a good trade off. It will be done by first doing a literature study to find compression techniques and then implementing them using a high level language like Python. The hardware demands for these compression techniques will be evaluated based on literature study, the code used when implementing the techniques and their algebraic expression. The compression ratio will be tested by compressing data that have been saved from a camera that uses the Bayer color filter.

The main goal will be to provide a group of algorithm alternatives that could be suitable for a camera which has the hardware limitations, as mentioned in the background section. In addition to these algorithms, some alternative techniques, such as near-lossless quantization, will be discussed to see if they are suitable for this thesis.

Using real-time lossless compression it is not possible to achieve the same compression ratio as with lossy compression techniques, such as the *lossy JPEG* technique. It will still be investigated how close to the *lossy JPEG* the lossless or near-lossless techniques developed during this thesis might come. The lossless compression techniques evaluated and developed will also be compared with the *JPEG-ls* and *PNG* compression techniques which will be a more fair comparison since both of those techniques are lossless.

In order to make the results practical, a brief study on data transmission will be made and a method on how to do this will be proposed.

1.4 Thesis Outline

Each following chapter in this report will begin with a few sentences that describes what the chapter and its sections' contents. In the *theory* Chapter the theory used in this thesis will be presented together with a new compression technique and together with an explanation on why it could work.

The chapter "Implementation" will present how the techniques used in this thesis have been implemented and a description of the methods used in order to verify and test these techniques.

In the chapter "Result" there will be tables and graphs presenting the main results that will later be discussed in the following chapter "Discussion". The images used to obtain the results in the chapter "Result" are shown in Appendix A and B. Finally, in the end of the report there will be a chapter with the conclusions from this thesis.

Throughout this report it will be assumed that Bayer split is applied if nothing else is mentioned, meaning that this pattern can be disregarded as if the image was a grayscale image. This means that when a pixel's closest neighbors are mentioned in this report, it does not mean the closest neighboring pixels in the image but rather the closest neighboring pixels in the same color according to the Bayer color grid.

Chapter 2
Theory

Compression is about lowering the number of words, symbols or bits used when representing any type of data. Images can be interpreted as a sequence of pixels where each pixel is represented using a fixed number of bits. Image compression can be achieved if the pixels can be represented in such a way that the average amount of bits needed to represent each pixel is lower than the original number of bits used for representing each pixel. This can be done using various techniques, but the most common technique for lossless compression, where no information is lost during the compression, is based on using a pixel prediction method and then an encoder which encodes the prediction errors. The achieved compression is then measured with the compression ratio described in the equation below. In this report it will be presented in either decimal form or as a percentage.

$$CR = 1 - \frac{\text{Compressed data size}}{\text{Original data size}} \tag{2.1}$$

This chapter will describe the main concept about the prediction that is used in image compression and also the background to why it works for compression of data. Various prediction algorithms such as linear prediction, non-linear prediction and adaptive prediction will be described. Later in this chapter different encoding techniques will be described. Following is a section about how an image sensor is connected with a digital interface in order to obtain a digital image. After that comes two section about near-lossless compression. The first one contains theory which will be used in a proposed near-lossless compression technique. In the second section will *Camera Serial Interface 2* (*CSI-2*) be described. *CSI-2* is a lossy compression standard for compressing raw image data from a sensor. Following will other compression standards be described. This will be followed by a section where a transformation based technique will be described in order to give an example of how transform based lossless compression techniques works. In the last section will protocols for transmitting data be discussed.

2.1 Correlation

Correlation is a measurement of how much different variables depend on each other. In this report, the variables are usually the compressed image's pixels. It is often referred to as how close two variables are to having a linear relationship with each

other[31]. It then becomes natural when it comes to any sort of prediction to find how a variable that should be predicted correlates with other, known, variables.

In most lossless image compression techniques there is some form of pixel prediction that is based on correlation between pixels. The main idea is to perform some form of prediction on a pixel sample and transmit the prediction error instead of the pixel value itself. As an example, a perfect prediction method would give an output of only $0's$. However, this is not possible in practice as value of pixels in an image are usually of stochastic nature. If the prediction method performs well however, the error values will be very close to zero. This will result in a probability distribution which can be assumed to be Laplace distributed [19]. The question is however, why it is possible to achieve this prediction. Is there any method of measuring how good a prediction can be made and if prediction can be made optimal. In the following subsections the concept of correlation between pixels will be presented, and how the correlation can be computed.

2.1.1 Correlation Between Pixels

When trying to predict the value of a specific pixel in an image there are several factors that affects how well the prediction may perform. One factor is how much known data that is available for prediction and another important factor is how high the correlation between the known data and the pixel that should be predicted is. Generally speaking, the correlations between pixels might differ for pixels at different locations in an image, but in average over the whole image, or in a local region, they will tend towards some values.

One way to calculate the correlation in an offline manner would be to go through many, if not all, pixels and create lists of samples consisting of a pixel value together with its neighbors values and then calculate the covariance between the list of the pixels with the lists of its neighboring pixels. The covariance is proportional to the correlation[30] and works therefore as a tool of measuring the correlation. This would work as each pixel value can be seen as a stochastic variable where there might be some form of dependency between the variables.

2.1.2 Static Statistics

The method described above can be used in several ways. One method would be to gather the correlation statistics before the compression takes place. Enough of samples could be taken until a satisfying precision of the correlation is reached. This is efficient in a computational sense as the predictor will have these values ready before the compression takes place and does not need to perform any computations during the compression. The downside of this method is that it does not give a precise correlation estimation for a specific regions, but instead only as an average of the whole image. The statistics of the image might also change drastically if the image scene is changed and the current known statistics will become invalid. The adaptive version of correlation estimate which is described below, avoids these problems, but at the cost of extra computational complexity.

2.1.3 Adaptive Statistics

The adaptive variant would be an algorithm that in a way learns and estimates the current correlations between the neighboring pixels. Using a method that learns to adapt to the statistics could result in that the pixels at every position would be predicted more accurately and the prediction error would be closer to zero, compared to the static case. However, there are several questions that needs to be answered about whether an adaptive method would be feasible. The most straightforward problem would be if it is even possible to find a way to learn about the statistics efficiently and precise enough and if the algorithm would be stable. More in-depth questions could for example be, what the starting correlation would be set to before learning and how much computational power is required to do the adaptation. The adaptive method proposed in this paper would be to use a row-scanning LMS algorithm which updates its weights according to how previous predictions has performed. It will be described in detail later on in this paper.

2.2 Entropy

The definition of entropy of a random process, X, is shown below [23].

$$H(\mathrm{X}) = E[-\log P(\mathrm{X})] = -\sum_{x} p(x)\log p(x) \tag{2.2}$$

$P(\mathrm{X})$ is the probability of the random process X while $p(\mathrm{x})$ is the probability of the actual value x of the random variable X. The assumption $0 \cdot \log 0 = 0$ is used. The Equation 2.2 can be rewritten in the sometimes more convenient form.

$$H(p_1, p_2, ..., p_k) = -\sum_{i=1}^{k} p_i \log p_i \tag{2.3}$$

Where p_i is the probability for the outcome i. Entropy can be used to determine the uncertainty in the outcome of a random variable, and equivalent the level of compression that can theoretical be achieved when compressing a sequence. Therefore, the entropy of a sequence can be used as a lower limit of the mean codeword length of each symbol in the sequence that can be achieved.

The entropy can easily be limited by the following expression [23].

$$0 \leq H(X) \leq \log k \tag{2.4}$$

Where k is the number of outcomes from the process X. The entropy is equal to $H(X) = \log k$ only in the special case when each outcome of the process has the same probability. This limits the compression ratio to the following expression.

$$CR \leq 1 - \frac{H(X)}{\lceil \log k \rceil} \tag{2.5}$$

2.3 Prediction

The method of predicting the current pixel based on old data and then, instead of sending the pixel values, send the difference between the actual value and the

predicted value is a efficient compression technique. This is based on the idea that there are mutual redundancy, or correlation, between a pixel and its neighbors. They basically contain the same information, so it is unnecessary to send the same information more than once and instead the difference between the pixels can be sent. This works because, when prediction is applied on an image, the probability distribution of the errors value is denser than the probability distribution of the original pixel values. This means that the error signal can be encoded more efficiently than the original signal using for example Huffman or Arithmetic coding because these encoders are more efficient for coding data that has narrow banded probably distribution of its values [20].

Prediction is a lossless compression technique because no information is lost in the prediction. The original pixel value can be reconstructed by using the predictor's corresponding reconstruction filter. It does the same prediction as the the prediction filter, but the difference between the prediction- and the reconstruction filter is that the reconstruction filter is provided with the error from the prediction done by the prediction filter. The reconstruction filter can then use the prediction and the prediction error in order to determine the original pixel values.

The prediction can be made in various ways, with different complexities and qualities. The simplest prediction is based on only using a past pixel value, preferably of the previous pixel, as a prediction of the current pixel.

$$\hat{x}(n) = x(n-\delta) \tag{2.6}$$

$\hat{x}$ is the prediction of the current pixel, $x(n)$ is the current pixel and δ is the distance between the current pixel and the pixel used for estimating the current pixel. The predictor can make use of more than one old pixel and these could be weighted differently, which is called the autoregressive model.

$$\hat{x}(n) = \sum_{k=1}^{M} w_k x(n-k) = \boldsymbol{w}^T \cdot \boldsymbol{u} \tag{2.7}$$

Where

$$\boldsymbol{u} = (x(n-1), x(n-2), ..., x(n-M))^T \tag{2.8}$$

$$\boldsymbol{w} = (w_1, x_2, ..., x_M)^T \tag{2.9}$$

The optimal solution to the autoregressive prediction is called the Wiener solution[7].

The error signal is modeled as followed.

$$e(n) = x(n) - \hat{x}(n) \tag{2.10}$$

The drawback with prediction is that it can create an error that may contain outliers which can take any value in the range $-\alpha \leq e \leq (\alpha - 1)$ where $\alpha = 2^\beta$ and β is the number of bits used for representing the pixels in the image. In practice, the prediction error e can only take α different values if the error is represented with the same number of bits as the pixels. In order to compensate for this may the prediction error be mapped computing modulo α.

$$e = \begin{cases} e_{old} + \alpha & \text{if } e_{old} < -\frac{\alpha}{2} \\ e_{old} - \alpha & \text{if } e_{old} > \frac{\alpha}{2} \end{cases} \tag{2.11}$$

e is the remapped error while e_{old} is the error before remapping. This does not affect the Laplace probability distribution of the prediction error in a significant way since the "tails" of the distribution will be remapped onto the the main-lobe of the distribution which already got much larger amplitude. Besides, the probability of a an error being outside the range $-\frac{\alpha}{2} \leq e \leq \frac{\alpha}{2} - 1$ is so low that it can be assumed that it never happens. This is due to that the error distribution is assumed to decay exponentially away towards zero[29].

2.3.1 Least Mean Squared Filters

The weights in the autoregressive model, shown in Equation (2.7), can be estimated using an adaptive filter, namely a Least Mean Squared (*LMS*) filter. *LMS* is an adaptive algorithm which adapts in order to make the prediction error as small as possible based on the local statistics. In order to reduce the prediction error, a cost function which should be minimized, which is created from Equation 2.7 and 2.10, is described using the mean squared error as,

$$J(n) = E\{e^2\} = E\Big\{\Big(x(n) - \widehat{\boldsymbol{w}}(n)^T \boldsymbol{u}(n)\Big)^2\Big\}, \tag{2.12}$$

where $x(n)$ is the current sample that should be predicted, $\hat{\mathbf{w}}(n)$ are the estimated weights in the autoregressive model, and $\mathbf{u}(n)$ are the old pixel values which are used as the input signal to the adaptive filter. The gradient of the cost function can be calculated and it points towards the direction in which the cost function grows the most. The gradient can be obtained by

$$\nabla_{\boldsymbol{w}} J(n) = -E\Big\{\boldsymbol{u}(n)\Big(x(n) - \boldsymbol{u}(n)^T \widehat{\boldsymbol{w}}(n)\Big)\Big\}. \tag{2.13}$$

Because the cost function and thus the error should be minimized, the algorithm takes a step in the negative gradient direction in order to find a local minimum of the cost function. The new filter coefficients will be updated according to the update rule shown below. It is done by using the actual values of the samples for calculating the gradient, by estimating the mean and not actual the mean itself. The LMS updating equation using actual sample values can be written as

$$w(n+1) = w(n) - \mu \cdot \nabla_w J(n) = w(n) + \mu \mathbf{u}(n)\Big(d(n) - \mathbf{u}(n)^H \hat{\mathbf{w}}(n)\Big), \tag{2.14}$$

where μ is the step size, which is the parameter that affects how far in the opposite gradient direction that the new filter coefficient will be. There is a risk that the step size, μ, is too large and will result in that the filter will not converge. The following condition needs to be fulfilled to reach convergence[7].

$$0 < \mu < \frac{2}{\lambda_{\max}}, \tag{2.15}$$

where $\lambda_{\max}$ is the largest eigenvalues of the autocorrelation matrix ($E\{\mathbf{u}(n)\mathbf{u}(n)^T\}$) of the system.

2.4 Encoding

In the following sections encoders will be presented, both static and adaptive encoders. The static encoders is based on some sort of hard-coded probability distribution, while the adaptive encoders adapts and uses the local statistics in the image when encoding.

2.4.1 Huffman Coding

In short, the *Huffman* encoder takes a list of symbols and assigns a unique binary code for each symbol depending on their respective occurrence probability. The algorithm does this optimally in the sense that the most probable symbols are given the shortest binary codes and the less likely symbols are given longer binary codes. This results in that the mean binary code-length for the symbols becomes as small as possible. The algorithm steps for Huffman is described as following:

1. Produce a set of nodes where each node contains a symbol and its probability of occurring.
2. Take the two nodes from the set with the smallest probabilities and create a new probability which is the sum of the two nodes probabilities.
3. Produce a parent node for these two nodes with the new probability and mark the left branch with a 1 and the right branch with a 0.
4. Update the node set by replacing the two selected nodes with the new parent node. Repeat the steps 2 - 4 until the node set only contains one node.

The result will be a binary tree where the code of each symbol will be the numbers of the branches you take by traversing from the root of the tree to each respective symbol [20].

2.4.2 Adaptive Huffman Coding

The *Huffman* coding algorithm uses a predefined list of probabilities for each symbol and it does never change throughout the execution of the algorithm. This requires that the probabilities are obtained in some procedure before the encoding procedure[20]. The idea behind *Adaptive Huffman* coding is to change the binary code of each symbol depending on an estimation of the probabilities, which is updated during the execution. This would mean that the binary tree and its paths will have to be changed during encoding.

Initialization

The *Adaptive Huffman* algorithm starts with all symbols having the same probabilities [20]. The probabilities are then updated by estimating the probabilities with a weight for each symbol, where the weight will represent the amount of times each symbol has occurred. Symbols which never have occurred would be marked as not yet transmitted (NYT) in a NYT node with a starting weight of 0. Each symbol will have to have a initial code which both the encoder and decoder have

agreed upon. A simple code variant for an alphabet $(a_1, a_2, a_3, \ldots, a_m)$ of size m could be to set the starting codes as following:

1. Pick e and r such as $m = 2^e + r$ and $0 \leq r < 2^e$
2. For each symbol a_k where $1 \leq k \leq 2r$, assign $(e+1)$-bit binary representation as the code.
3. For the rest of the symbols assign the e-bit binary representation.

For an alphabet with the size $m = 4096$ the values $e = 12$ and $r = 0$ can be used. When a symbol is encountered for the first time at the encoder, the code for the NYT will be sent together with the assigned code for the symbol and then it will be removed from the NYT list. A new node for the symbol is created and its weight is increased by one. By using this procedure, both the encoder and decoder are ensured to have the same tree structure.

Because the tree will become a binary tree, the maximum amount of nodes in the tree will be $m \cdot 2 - 1$. The set of nodes with equal weights is referred to as a *block*. Parent nodes will have the combined weight of its children. For the tree to still be a binary Huffman tree the *sibling property* needs to fulfilled[3]. The property will restrict us in the following way:

1. Each node will be numbered from the top to bottom and from right to left in a descending order starting from the maximum value of $m \cdot 2 - 1$.
2. A node must always have a higher number than an another node with a lower weight.

Updating

After a new symbol has been encoded the tree needs to be updated. One of the two following situations may happen[20]:

1. If a symbol is occurring for the first time and is in the NYT list, the NYT node will create two child nodes. The left child node will now be the NYT node and the right child node will have the new symbol assigned to it with the weight 1. The old NYT node is now the parent and its weight is updated to 1. The nodes are now numbered according to property 1, i.e the parent node will maintain the old NYT node's number k and the new NYT node will be assigned the number $k - 2$ and the new symbol node the number $k - 1$. The symbol is lastly removed from the NYT list.

2. If the encoded symbol already existed in the tree, the corresponding node for that symbol will update its weight.

In any of these two situations, the parent nodes in the chain up to the root will have to update their weights. The update of the weights is done from the leaves of the tree to the root. Before the weight of a node is updated, the tree has to be searched in order to check if the node belongs to a block of nodes where another node has a higher node number. If this is the case they swap position and thus changing node numbers. Afterwards, the node's weight will be increased by 1 and

the parent node is then updated in the same way. Doing this will ensure property 2. The same steps will be applied until the root has been reached.

Encoding

As mentioned earlier, both the encoder and the decoder has agreed upon starting codes for the symbols. This data needs to be synchronized and sent in some way before the encoding procedure starts. Both the encoder and decoder will start with the same tree structure consisting of only the NYT node as the root. As previously mentioned, there are two situations that may happen and the encoder will act accordingly:

1. If a symbol occurs for the first time and is in the NYT list, the encoder will output the code for the NYT node, which is obtained by traversing from the root to the NYT node like in the standard Huffman encoding procedure. The encoder will also send the predetermined codeword for the appeared symbol.
2. If the symbol does not exist in the NYT list, the encoder will output the code which is obtained by traversing from the root to the symbol node like in the standard Huffman encoding procedure.

2.4.3 Golomb-Rice Encoding

The *Golomb* encoding is based on the assumption that the larger the value of the symbol is the smaller its probability of occurrence is. The *Golomb* encoder then assumes the symbol distribution to be of a certain shape and therefore some parameters has to be estimated. Assume that the probability of an event is geometrically distributed, then the probability of an event, X can be modeled as

$$p(X = k) = (1 - p)p^k, \tag{2.16}$$

where the expected value of X is given by

$$E(X) = \frac{p}{1 - p}. \tag{2.17}$$

By estimating the mean of X using old samples the estimation of the probability, p can be calculated[22] as

$$p = \frac{\widehat{E}(X)}{1 + \widehat{E}(X)}. \tag{2.18}$$

Optimal *Golomb* coding is achieved by selecting a coding parameter m which relates to p according to the following expression.

$$m = \left\lceil -\frac{\log(1 + p)}{\log(p)} \right\rceil \tag{2.19}$$

The parameter m is often chosen so that $m = 2^k$. This is known as *Golomb-Rice* encoding. *Golomb-Rice* encoding is a good choice of *Golomb* implementation due

to its simplicity. The parameter k can be chosen in an optimal way as [13].

$$k = \max\left\{0, 1 + \left\lceil \log_2\left(\frac{\log(\phi - 1)}{\log\frac{\widehat{E}(X)}{\widehat{E}(X)+1}}\right)\right\rceil\right\}, \tag{2.20}$$

where

$$\phi = \frac{\sqrt{5}+1}{2}. \tag{2.21}$$

If the mean outcome $\widehat{E}(X)$ would be less than ϕ, then k is set to $k = 0$.

Encoding

Rice's version of *Golomb*'s algorithm is the special case when m is equal to 2 to the power of k, $m = 2^k$, where k is a positive integer. The *Golomb* code for the symbol with the integer value n where $(n > 0)$, can easily be obtained[20][13] as

$$q = \left\lfloor \frac{n}{m} \right\rfloor, \tag{2.22}$$

where $\lfloor x \rfloor$ is the integer part of x. Then the remainder part r needs to be calculated, which can be done by

$$r = n - q \cdot m. \tag{2.23}$$

From the resulting values of q and r the encoding can be made. q is always coded as a unary code, while r is coded using binary coding if m is a power of two. The codeword for n is the combination of q put together with r as followed [20].

$$\text{codeword} = q, r \tag{2.24}$$

If m is a power of two, the implementation and decompression will be of lower complexity[4]. This is true because the division with m will be easier to calculate. When m is a power of two, the division can be replaced by a shift operation instead, which is a much less computationally costly function.

2.4.4 Arithmetic Encoding

Arithmetic encoding is an encoder that put multiple symbols into a sequence which then is encoded. A symbol sequence is given a unique binary code[20]. It performs significant better than *Huffman* encoding when using data with a small alphabets that has a skewed probability distribution. This is true because it codes sequences of symbols rather than single symbols and it can be shown that the mean output code-length is closer to the entropy when coding sequences instead of single symbols[20]. The *Arithmetic encoder* can be implemented as a adaptive encoder. Then it does not depend on context and it may gather information about the data statistics as it is processing the data[33].

The downside with *Arithmetic coding* is that it is more computationally costly than *Huffman*. Because of this and that the symbol alphabet used during this thesis is rather large, the *Arithmetic encoder* will not be implemented and its performance will not be evaluated further. A large symbol alphabet means that the

performance between *Arithmetic* and *Huffman* encoding should not differ significantly and therefore is it unnecessary to implement the *Arithmetic encoder* which is a more demanding algorithm than *Huffman.*

What *arithmetic* coding does is that it compress an arbitrary long sequence of symbol codes to a tag, which is a unique number. Each symbol are assigned a interval between $[0, 1)$ where the upper bound of the symbols' intervals will be referred to as $F_X(s)$ where s is the symbol order in this interval. The intervals' sizes relates to their probabilities of occurring. The generation of such a tag is explained in detail in the following section. This generation of the tag is however not feasible on computers with limited precision and another way of implementing the generation must be performed, but it is still useful to show this tag generation method as it is easier to understand.

Generating a Tag

To encode a sequence of symbols a tag has to be generated which is created by going through each symbol until the end of the message is reached[20]. The tag will start by having an upper bound, u and a lower bound l chosen as $[l_0, u_0) = [0, 1)$. The algorithm will then go through each symbol one at a time from start to end and update the lower and upper bound according to the update rule shown below.

$$l_{s+1} = l_s + (u_s - l_s) \cdot_X (s-1) \tag{2.25}$$

$$u_{s+1} = l_s + (u_s - l_s) \cdot_X (s) \tag{2.26}$$

If n_j is the amount of occurrences of the symbol j and the total count of symbols are N, then the symbols' intervals can be estimated as

$$F_X(s) = \frac{\Sigma_{k=1}^{s} n_k}{N}. \tag{2.27}$$

Using this the new updating equations can be rearranged as into the following equations.

$$l_{s+1} = l_s + \frac{(u_s - l_s) \cdot \Sigma_{k=1}^{s-1} n_k}{N} \tag{2.28}$$

$$u_{s+1} = l_s + \frac{(u_s - l_s) \cdot \Sigma_{k=1}^{s} n_k}{N} \tag{2.29}$$

As the number of symbols to encode, s, increases the upper and lower bounds will be more narrow and eventually their most significant bits (MSB) will become equal. This means that both bounds are confined within either the upper or lower half of the unit interval $[00...0, 11...1]$. After the MSB become equal they will never change and a value between the upper and the lower bound may be transmitted as the tag, the lower and upper bounds can then be reseted. This is repeated until the end of the symbol sequence has been reached.

2.4.5 Run-length Encoding

Run-length encoding is based on the idea that there might be a high correlation between neighboring values in a stream or between the pixels of an image. Instead

of outputting a sequence of identical values, the algorithm instead only outputs the value and the number of times it is sent in a row[15].

An example could be a compression of the following sequence of characters,

WWWWWWWWWWWWBWWWWWWWWWWWWBBB,

which will become after encoding:

12W1B12W3B.

In this case the number tells how many times the following symbol will be repeated. So the decoder will simply expand the string back to its original form using the given information.

Run-length with Bit-plane Coding

An idea based on run-length encoding is the idea of bit plane coding. Bit-plane coding views the image as a three-dimensional array of bits[26]. Each pixel value of the two-dimensional image can be seen as a vector containing bit-values, thus creating the three-dimensional array. In this paper method based on some form of prediction and run-length encoding will be proposed. First is the image preprocessed by the prediction technique

$$e(n) = x(n) - \hat{x}(n), \tag{2.30}$$

where n is the current sample. If the prediction is good, this would lead to, as mentioned earlier, a low value of $e(n)$. This means that it is more likely that bits that represent lower numbers in $e(n)$ are '1' than bits representing higher numbers. The idea is to remove some of the more significant bits from $e(n)$ and encode them separately using run-length encoding. This is assumed to be effective since it is assumed to be almost only be zeros in the highest bit levels of the error.

Below is an example where this technique is applied on a image. From the image in Figure 2.1a was the prediction error in Figure 2.1b calculated. It can be seen that there is still much redundancy in the image and it can be seen because most of the image is black. This redundancy is due to the fact that there are only a few error estimations which are bad while the most estimations are good.

In Figure 2.2 are the probability of a bit being '1' for the different bit levels in the error image shown in Figure 2.1b shown. On the horizontal axis are the bit levels represented, from bit level 0 which represent the least significant bit to bit level 10 which represents the most significant bit. The sign bit is not included in this plot since it is assumed that it appears random with being just as likely to be '1' as '0'. It can be seen that the more significant bits barely becomes '1'. This gives the idea that if you go through the bits plane-wise, there will be long sequences with only '0's.

(a) Original image. (b) Error prediction from original image.

Figure 2.1: Example of prediction error from an image.

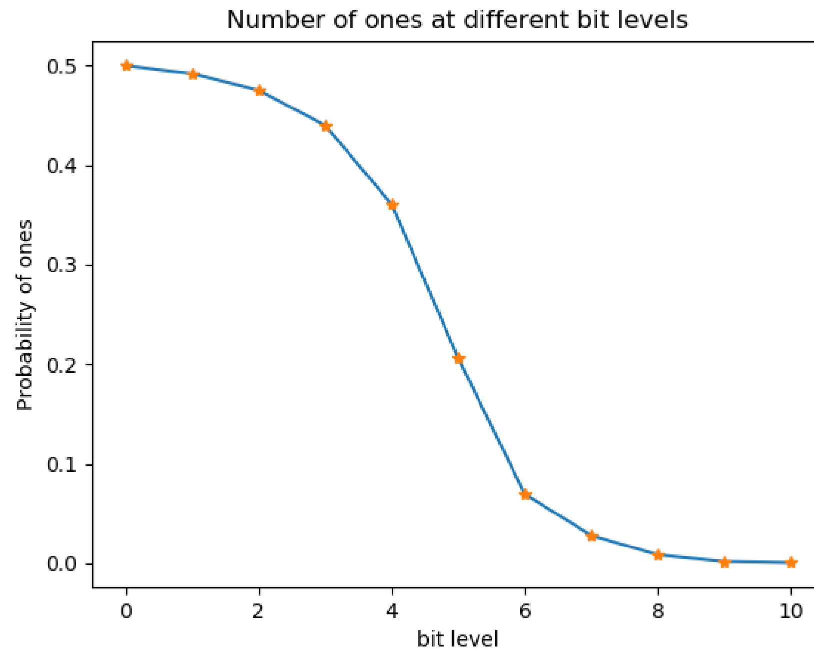


Figure 2.2: Percentage of ones for different bit levels.

2.5 Horizontal and Vertical Blanking

A camera sensor needs to be connected with some digital interface[34]. Depending on the type of interface used, the output might differ and the interface might provide output with unnecessary data. This is true when the digital interface needs to run at a constant clock speed and the sensor is read with faster scanning manner. This will make the camera to perform some additional operations in some cases before it can read a new line on the active video region.

Before the first line there might be a blank space consisting of a few lines because of vertical synchronization and likewise, before each line there might be a blank space for horizontal synchronization. Depending on the camera sensor used and the settings for the camera, the amount of blank space added might vary. During these blank sections, the camera will output only 0's. This effect is called horizontal and vertical blanking.

2.6 Near-Lossless Compression

During this section, noise in the image sensor will be discussed. It will also be discussed how this motivate the use of near-lossless compression and why this may be a good idea to implement, but also the drawbacks on how near-lossless affects the pixel quality in an image.

2.6.1 Noise in the Image Sensor

In an image sensor there are mainly two types of noises, namely read noise and shot noise. Read noise is a combination of noise, from electronics and from the analog to digital converter, that is a sort of ground noise, that basically has a fixed value for each pixel independently of the pixel value. Less read noise results in higher accuracy in the pixels, that determines the contrast resolution that the sensor is able to achieve. Shot noise on the other hand increases with increasing pixel values. The noise is based on the number of photons that enter each pixel in the sensor during an image capture. It has a standard deviation equal to the squared root of the input light signal. For high photon intensity in a pixel, or high pixel value, the shot noise becomes the dominant noise factor in the image sensor [8].

A simple method, that will be used in this thesis, to reduce the noise in a image is to take the average of multiple images of the exact same scene. Since the scene in all of the images is the same and deterministic while the noise is an Gaussian distributed stationary stochastic process may the noise be removed. This while the scene stays the same by averaging a set of images. The averaging affects the noise in the way that it reduces the variance by a factor k where k is the number if images used for averaging[25], which means that when averaging with large enough dataset the noise is removed, meaning that the noise takes a mean value of zero and variance close to zero.

2.6.2 Quantization Noise Model

As mentioned in the previous section, different image sensors used in cameras may introduce different levels of noise in the pixel values.This means that, depending on the amplitude of the noise and the numerical precision the camera has, some bits in the pixel values might be affected by noise. Noise is hard to compress due to their random behavior, which makes the removal of these bits very useful. This already present noise in the image makes it possible to truncate the less significant bits of the pixels without affecting the image quality in a significant way. The truncation will introduce another noise which can be considered having a uniform

distribution[12] that will take values between $[-2^s + 1, 2^s - 1]$ where s is the amount of bits that is truncated. The original goal with this thesis was to perform completely lossless compression of the original sensor data, but when considering the already present noise, doing truncation to some extent is very motivated.

How much to truncate depends on how much added quantization noise that is tolerable and is a question that does not have a clear answer. The rough idea is that the added noise should be much smaller than the already present noise to not interfere with the image processing at the back-end processor. A measurement of this is defined as the signal-to-quantization-noise ratio (SQNR), which is a ratio between the maximal value of the signal and the noise's. By allowing different levels of SQNR between the sensor noise and the quantization noise, different compression ratios could be achieved.

2.7 Camera Serial Interface 2

Camera Serial Interface 2 (*CSI-2*) is a standard compression technique of raw image data[18]. The compression is done in a lossy environment and is achieved by first predicting each pixel by a simple pixel scheme and then encode it. The encoding is not done according to any encoding described earlier in this paper. The codeword length of the encoded symbols are predefined and fixed. The first of these bits are used as a header to tell the decoder in what range the prediction error is between. Then there is a sign bit followed by a quantization of the prediction error. The quantization will be rougher the larger the prediction error is.

This means that the resulted compression ratio from this technique will be fixed while the level of how much this technique affects the quality of the image will vary depending on how well the prediction method works. This compression technique is defined in six different schemes shown in Table 2.1. Later in the report it will be seen that the compression ratio from these different schemes is inferior to the near-lossless compression technique suggested by the authors in this thesis. Another thing which makes the near-lossless technique suggested by the authors of this thesis superior is that the degree of allowed quantization noise is fixed and that the quantization only affect the bits that are affected by the noise which the image sensor introduces. In order to know how the two different methods affect the image quality after the image processing steps, a further study needs to be done but it is assumed that the quality would be less affected by the proposed method. Therefor *CSI-2* will not be investigated further in this thesis.

Original bits in pixel	Encoded bits per pixel	Bits in pixel after reconstruction
12	8	12
12	7	12
12	6	12

Table 2.1: Compression scheme for *Camera Serial Interface 2*

2.8 ISO Compression Standards

In the following subsections two of the most widely used lossless compression algorithms will be presented, namely *JPEG* and *PNG*. They are both rather similar and both are based on doing first a prediction step and then a encoding step.

2.8.1 JPEG

The ISO/CCITT committee known as *JPEG* (Joint Photographic Experts Group) established a compression standard for images in 1992, for both gray-scaled and colored images. They released a JPEG standard which includes two basic compression methods where one is a lossless method which is based on prediction[28] and the other is a lossy method that is based on Discrete Cosine Transform (*DCT*). The methods will briefly be described in the following subsections. The lossy method will be described in order to increase the understanding of image compression in general and it will be investigated later as a reference of how much lossless- and lossy compression differs. However, the lossless version of JPEG is more relevant to this paper. The lossless compression can be divided into two sub-techniques but both of them are based on the same idea. They are based on the technique of first predicting each pixel in the image using old pixels and then encode the error using an entropy encoder. This can be done either in the "old" way by using linear prediction in different modes or in the "new" way with a non-linear predictor.

2.8.2 Lossless JPEG

The standard *JPEG-ls* algorithm is an offline, prediction based compression algorithm, that produces compression which is fairly close to the state of the art for lossless compression [9]. In the two following subsections the linear and non-linear JPEG techniques will be presented.

The Linear Algorithm

The predictor in the *JPEG-ls* combines values from three different neighbor samples, as shown in Figure 2.3. The neighboring pixels are used in eight different modes. The first one makes no prediction, the following three modes makes prediction from one dimension and the four remaining modes makes a prediction in two dimensions. All of these modes are tried out in a non-real-time environment and the mode which produces the lowest prediction error is then used for transmission. The mode used to perform the prediction is coded in the 3-bit header of the compressed file. The modes are described in the Table (2.2). For color images the *JPEG-ls* is normally able to compress the data up to around 50% [28].

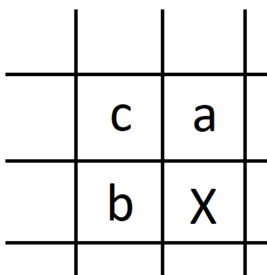


Figure 2.3: Pixels used for predicting the current sample.

Mode	Prediction
0	no prediction
1	$\hat{x} = b$
2	$\hat{x} = a$
3	$\hat{x} = c$
4	$\hat{x} = a + b - c$
5	$\hat{x} = b + \frac{a-c}{2}$
6	$\hat{x} = a + \frac{b-c}{2}$
7	$\hat{x} = \frac{a+b}{2}$

Table 2.2: Prediction for lossless JPEG coding.

The prediction error from the predictor can then be efficiently encoded using either Huffman or arithmetic coding [15].

The Non-linear Algorithm, LOCO-I

Lossless JPEG can also be implemented using an online, non-linear predictor, where in that case the compression technique is called *LOCO-I*, which stands for *LOw COmplexity LOssless COmpression for Images*. The equation for the prediction of the current pixel $\hat{X}$ used in *LOCO-I* [32] is shown below.

$$\hat{X} = \begin{cases} \min(a,b) & \text{if } c \geq \max(a,b) \\ \max(a,b) & \text{if } c \leq \min(a,b) \\ a+b-c & \text{else} \end{cases} \tag{2.31}$$

This is a simple edge detector which can detect horizontal and vertical edges. If a horizontal edge appears in the image, the horizontal neighbor b will be used for prediction. If instead a vertical edge appears, the vertical neighbor a will be used. This is done either directly if c is the largest or smallest value in the set, or by the subtraction $a + b - c$. Here c can be assumed to have almost the same value as either a or b depending on if the edge is horizontal or vertical. In the case where there is no edge close to the pixel, then it can be assumed that the image

is smooth in the area of the pixel which means that $a = b = c$. It results in that $\hat{x}$ can be approximated by $\hat{x} = a$. The error is then encoded using a *Golomb-Rice* encoder[20].

2.8.3 Lossy JPEG

Using *lossy-JPEG* the "lossyness" can be adjusted as a parameter to the algorithm. It is a Discrete Cosine Transform-based (DCT) compression algorithm, that divides the signal into lower and higher frequency components.

First, each pixel in the image are subtracted by $2^{\beta-1}$ to center the data values around 0, where β is the number of bits used to represent each pixel. This makes each pixel to take on a value in between $-2^{\beta-1} \leq x \leq 2^{\beta-1} - 1$ instead of $0 \leq x \leq 2^{\beta} - 1$. The image is then divided into groups of 8x8 pixels which then are transformed using the forward DCT-transform. The transform results in that the low frequency components that are essential for the image quality are put in the upper left corner and the high frequency are put in the bottom right corner of each block. Low frequency components have in general higher values than the high frequency components. After the transpose the values are quantized according to a quantization table that have to be generated for the image. In general, the high frequency components are quantized using a larger step size, so that quantization will be rougher than for the low frequency components. This leads to that the elements in the bottom right corner are truncated down to fewer quantization levels for the possibility of a higher compression ratio at the cost of loss in quality. The remaining parts are then encoded and the high- and low frequency parts are coded separately[20].

2.8.4 PNG

The *PNG* compression technique was accepted by ISO in 2004 as a lossless, portable compression technique for computer graphics image transmission over the Internet standard[10]. *PNG* is a prediction based and lossless image compression technique that utilizes, just as *JPEG-ls*, different prediction modes. The differences with *JPEG-ls* are that *PNG* only uses five different prediction modes and that the *PNG* predictor uses one mode for each row in the image to compress. A header to each row is added to tell which mode that were used for that row[21]. Using the same notations as in Figure 2.3, the prediction methods can be described as show in Table 2.3.

Mode	Prediction
0	no prediction
1	$\hat{X} = a$
2	$\hat{X} = b$
3	$\hat{X} = \frac{a+b}{2}$
4	$\hat{X} = \text{Paeth}(a, b, c)$

Table 2.3: Prediction for lossless JPEG coding.

Paeth is a more complicated predictor than the other four. The predictor is described below in its pseudo-code [6].

```
def Paeth(a, b, c):
    p = a + b - c
    pa = abs(p - a)
    pb = abs(p - b)
    pc = abs(p - c)
    if (pa<=pb) and (pa<=pc): return a
    elif (pb<=pc): return b
    return c
```

The predictor tries to determine which of the pixel values of a, b, and c that is closest related to the pixel tried to predict, x. After the prediction, the prediction error is compressed using the DEFLATE compression method[5].

DEFLATE is a *LZ77*-derived algorithm, that is fundamentally based on the concept of sliding window. One can assume that information are repetitive to some degree, no matter if it is a text, song or an image. The sliding window is sliding over the image using a certain width of the windows and once a sequence is found in the window that has occurred before, the distance to the previous occurrence will be encoded instead of the encoding of that sequence. This encoding is done using Huffman encoding [5] and the sequences that have not appeared before will be encoded using Huffman as well.

2.9 The Burrows-Wheeler Transform

This theory section about the Burrows-Wheeler transform is optional to read, it does not contribute to the understanding of the rest of the paper, but it will be used for discussion of transform methods.

The *Burrows-Wheeler Compression Algorithm* (BWCA) can be divided into four stages. The first stage is the Burrows-Wheeler transform which sorts the data so that data with similar context are grouped closely together. The number of symbols during the transform is kept constant. The second stage is called *Global Structured Transform* (GST) or *move-to front* (MTF), which transforms the local context of the variables to a global context. The third stage is a run-length encoder that is used for reducing the number of symbols used for encoding the data. It can be done efficiently because the symbols that are alike is grouped together during the move-to front phase. The last stage is a entropy coding stage which compresses the output data, that can be done using Huffman or arithmetic coding[24]. The flow scheme for the algorithm can be seen in Figure 2.4.

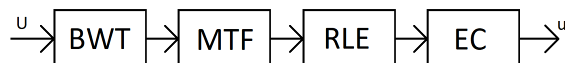


Figure 2.4: Flow scheme for the Burrows-Wheeler compression algorithm. U is the input sequence which should be compressed and u is the compressed sequence.

2.9.1 The BWT Algorithm

Instead of using a linear prediction as i.e. *JPEG-ls* does, BWT is using the whole image or parts of pixel context of the image in order to rearrange the pixels in an order which is easier to code. Originally BWT was used for text compression. The BWT performs a permutation of the symbols to compress this permuted sequence then get sorted in a manner so that symbols which are similar to each other is grouped together. The Forward BWT is done in three steps[2].

1. From an input sequence U of length N, create $N-1$ sequences where all of them is a shifted version of the original sequence U by cyclic rotations of the characters in U. The permutations form a $(N\cdot)$ matrix M', where each row of M' represents one permutation of U.
2. Sort the rows of M' lexicographically to form an other matrix M. M includes the original sequence U as one of its rows.
3. Record L, the last column of the sorted permutation matrix M, and id, the row number for the rows number for the row in M that corresponds to the original sequence U.

As an example, suppose the input sequence which should be compressed is $U =$ **mississippi**. The permutation matrix M' is created and sorted in order to create the matrix M. Let F and L each represent the fist and last element of each row in M respectively. Then, $F =$ **iiiimppssss** and $L =$ **pssmipissii**. The output of the transform will be the pair $(L, id) = ($**pssmipissii**$, 5)$. The example is illustrated in Table 2.4, where M' is the matrix of permuted sequences of U and M is the lexicographically sorted version of M'. Generally, the sequence L can be compressed more efficiently than the original sequence U[2].

M'	M	F	L
mississippi	imississipp	i	p
ississippim	ippimississ	i	s
ssissippimi	issippimiss	i	s
sissippimis	ississippim	i	m
issippimiss	mississippi	m	i
ssippimissi	pimississip	p	p
sippimissis	ppimississi	p	i
ippimississ	sippimissis	s	s
ppimississi	ssippimissi	s	i
pimississip	sissippimis	s	s
imississipp	ssissippimi	s	i

Table 2.4: The Burrow-Wheeler Transform, the fifth row corresponds to the original sequence of letters.

After the transform, the sequence L is rearranged by a so called *move-to-front* method that sorts L even more so that it becomes more suitable for compression.

In the *move-to-front* algorithm each symbol in the sequence is assigned one number. The lowest valued symbol is assigned 0 and next the unique symbol with higher value is assigned 1 and so on. These numbers are stored in a list where each number correspond to its number. Then the sequence L is iterated through, starting from the beginning of the sequence. When a symbol occurs in the sequence L, its value from the corresponding list is transmitted and the number in the list is then moved to the top of the list. This results in that, if there is a run of a certain symbol in L, then a sequence of 0s is transmitted [20]. If *move-to-front* is applied to the example above where $L =$ **pssmipissii** then the following codeword is obtained.

The sequence L contains the following symbols.

$$A = \{i, m, p, s\} \tag{2.32}$$

From which the following table can be created

0	1	2	3
i	m	p	s

The first element in L is 'p' which is encoded as 2 and 'p' is then moved to the top of the table.

0	1	2	3
p	i	m	s

The next symbol is 's' which is encoded as 3 and 's' is then moved to the top of the list.

0	1	2	3
s	p	i	m

Next symbol is also a 's' which is encoded as 0, because 's' is already at the top of the list is nothing changed. After iterating through L, the following sequence is obtained and will be transmitted.

$$\{2, 3, 0, 3, 3, 3, 1, 3, 0, 1, 0\} \tag{2.33}$$

It is possible to achieve better compression if the sequence to compress, U, is a long sequence compared to its alphabet. After this step the sequence should be compressed using run-length coding.

2.10 Forward Error Correction

When transmitting data there is, due to noise in the transmission medium, a risk of a bit being read incorrectly. There are several methods available to both detect and possibly correct these errors by adding redundant bits to the data

being transmitted. This technique is usually referred to as Error Correcting Code (ECC) and a technique using ECC is Forward Error Correction[27](FEC). The idea behind FEC is to add redundant information to the data sent by using some form of ECC. One property of FEC is that the receiver can correct transmission errors on a one-way link. Error correcting techniques are a bit outside the scope for this thesis and therefore only a simple method will be proposed while other possibilities will only be discussed. Common techniques which will not be explained in this paper includes techniques such as LDPC-codes.

2.10.1 Repetition Codes

A very simple FEC technique is Repetition Code[27] that basically means that each bit is sent a fixed number of times and then the receiver will decode the sent bit by doing a "majority vote". The bit will be interpreted as a 0 if most bits received were 0's and interpret it as a 1 if most received bits were 1's.

The downside to this method is rather evident, as it is clear that this would add extra redundant data by a factor of k. The benefits of using this method is however clear, as one would need more than half of the bits to be free from errors to correctly decode the symbol. For example, if $k = 3$, the risk of a bit-error is $p = 10^{-9}$ and the bits' bit-errors are independent of each other, then 2 or more of the bits would need to have its symbols switched, which would only happen with a probability of

$$\binom{3}{2}(p)^2(1-p) + \binom{3}{3}(p)^3 \approx 3 \cdot 10^{-18}. \tag{2.34}$$

The situation where the error-risks are independent of each other is in most cases unlikely and it will depend on the type of noise that is present in the transmission medium. When the errors are grouped together, it is referred to as *Burst Error*[1]. Burst errors are said to be more likely in serial transmission and the length of the burst depends on the bitrate and the type of the noise.

Chapter 3

Implementation of Algorithms

In this chapter, the implementation of the scripts used for obtaining results are presented but also how the computational cost of the implemented methods are evaluated. A correlation estimation method is presented and it is described how it is used to determine prediction methods. Then, the prediction methods' used in this report are described. Afterwards, the entropy encoders used for encoding, which lowers the amount of bits needed to represent each symbol, are presented. All implementations of the predictors and encoders were made in the scripting language *Python*. The benchmark script used for obtaining results is described. Thereafter, the evaluation process of the complexity of the different techniques used is described. The *Run-length* encoder is kept separately in a different section for how its potential is investigated. Near-lossless quantization is presented next, where the reason why it could be acceptable to remove bits in pixels without affecting the image quality significantly. Finally, the *Lossy JPEG* and the *PNG* compression techniques will be tested and compared to the other suggested compression methods.

3.1 Correlation Analysis

To gain better understanding of why prediction methods could produce a good estimate of a pixel value, the correlation in an image was studied. It was done by using scripts written in *Python* that work as described in Section 2.1.1. The correlation was illustrated in a 2D-plot, where the value of the left neighbor pixel to the current pixel was plotted on the y-axis and the value of the current pixel on the x-axis. This was performed after a Bayer split, meaning only one color was used to produce the result. The green pixels were chosen to be used for analysis as they are the most common pixel in the Bayer pattern. The reasoning for this was that the pixel would have a much stronger correlation with pixels of its own color. A correlation surface plot of multiple neighbors was obtained in similar way in order to find how pixels further away from each other correlate with each other.

3.2 Predictor Implementation

As mentioned in the theory chapter of this thesis, pixel value prediction is an efficient way to achieve compression. Therefore several prediction techniques were

implemented based on the results obtained from studying the correlation in images. Initially a simple predictor, that used the value of the pixel to the left of the pixel that was supposed to be predicted, was implemented. When it was done, it could be seen that the prediction error that it produced behaved as expected. Afterwards, several other prediction methods were then implemented in an attempt to get a prediction error with a as narrow-banded probability distribution as possible in order to achieve good compression ratio. The predictors have been designed based on the idea that the pixels close to the pixel that should be predicted are highest correlated with the pixel to predict. The pixels that are close to the pixel to predict are combined and weighted to utilize more correlation in the prediction.

To show these effects visually, a script was written in *Python* to produce plots that shows how the probability distribution is shaped for the original image and the image of the errors after the predictions. This was done for the two images **test12.pgm** and **test24.pgm** to show how the results may vary depending on the image. These two images were selected for this test because **test12.pgm** has much less visual structure than **test24.pgm**. The distribution after the prediction was also plotted before and after the mapping of the negative and large values. This mapping is described in detail later on in this section. To get a numerical measurement of how well the prediction performed, the entropy was calculated before and after each prediction.

3.2.1 Predictor Design

The predictors that were designed and tested are shown in Table 3.1 where the prediction rule is presented or referred to. Here $\hat{x}$ is the predicted pixel value. The pixels used in these predictors are referred to by letters as shown in Figure 3.1, where x is the pixel to predict. The prediction methods that are written in bold text are methods which have not been discovered during the literature study. They are methods that we propose ourselves. With that in mind it is not certain that no one else have discovered these methods before us.

Each predictor was implemented together with a reconstructor that can, given the error that corresponding predictor produce, losslessly reconstruct the image that was originally given to the predictor. The predictor was made sure to have a working reconstructor before it was considered to be complete and working as intended. This was done to prove that the prediction methods actually do perform lossless compression, meaning that no information is lost.

Predictor	Prediction rule
Simple	$\hat{x} = b$
Mean2	$\hat{x} = \frac{a+b}{2}$
Mean4	$\hat{x} = \frac{a+b+c+e}{4}$
Mean4-R	$\hat{x} = \frac{\text{sum}(a,b,c,d,e,f)-\min(a,b,c,d,e,f)-\max(a,b,c,d,e,f)}{4}$
Mean2L	$\hat{x} = \frac{b+\frac{a+e}{2}}{2}$
Mean2L-2	Multiple predictions, description below.
Median	$\hat{x} = \text{median}(a,b,c)$
LOCO-I	Shown in Equation 3.2
Paeth	Described below.
Combined	Combines multiple predictors, description below.
LMS	$\hat{x} = a \cdot w_0 + b \cdot w_1 + c \cdot w_2 + d \cdot w_3 + e \cdot w_4$
LMS2	Multiple predictions, description below.

Table 3.1: Prediction for lossless JPEG coding.

		f	
	c	a	e
d	b	x	g

Figure 3.1: Pixels used for predicting the current sample.

Mean2L-2, initializes by performing the *Mean2L* prediction to obtain $\hat{x}_{est}$. Afterwards $\hat{g}$ is estimated also using the *Mean2L* predictor with the $\hat{x}_{est}$, e and the pixel right to the e pixel. The final estimate of $\hat{x}$ is performed after $\hat{g}$ is estimated by the following equation.

$$\hat{x} = \frac{a + b + \hat{x}_{est} + \hat{g}}{4} \tag{3.1}$$

LOCO-I performs the prediction according to the following Equation.

$$\hat{x} = \begin{cases} \min(a,b) & \text{if } c \geq \max(a,b) \\ \max(a,b) & \text{if } c \leq \min(a,b) \\ a + b - c & \text{else} \end{cases} \tag{3.2}$$

Paeth is inspired by the prediction done in PNG described in Section 2.8.4. The prediction $\hat{x}$ is set to the value returned from the following function.

```
def Paeth(a, b, c):
    p = a + b - c
```

```
pa = abs(p - a)
pb = abs(p - b)
pc = abs(p - c)
if (pa<=pb) and (pa<=pc): return a
elif (pb<=pc): return b
return c
```

The *LMS* predictor uses adaptive weights w_k that are updated after each pixel as described in Section 2.3.1.

The *LMS2* predictor first uses the same prediction as *Mean2L-2* to predict the value of $\hat{x}$ and $\hat{g}$. The final prediction of $\hat{x}$ is $\hat{x} = a \cdot w_0 + b \cdot w_1 + c \cdot w_2 + d \cdot w_3 + e \cdot w_4 + \hat{x} \cdot w_5 + \hat{g} \cdot w_6$, where w_k are weights adapted by a LMS algorithm.

3.2.2 The Combined Predictor

Combined is a prediction method that combines the prediction method used in *Mean2*, *Mean2L*, *LOCO-I*, *Mean4*, *Paeth* and the prediction $\hat{x} = a + \frac{b-c}{2}$ which is the prediction done in the sixth mode of the linear *JPEG-ls* method 2.8.2. Each of these predictors performs its prediction on each pixel while only the error from one of the predictors is used to represent the pixels. Each row of the image to compress is divided into a number of sections, the error from each predictor are summarized separately over each section. At the end of a section, the summarized errors are compared, the predictor that produced the lowest summarized prediction error is chosen to produce the errors used for representing the pixels in the corresponding section of the following row. This prediction of which section to use can be done in a more advanced way by comparing the summarized prediction error from multiple sections that are close to the section to predict. In other words, the *Combined* predictor tries to predict what prediction method to use.

Additional script was made to produce results showing how the compression ratio varies for different amount of sections. These results are instead shown under the Benchmark section in Results, as the benchmark will suit as a good reference for how good the compression ratio usually is.

3.2.3 Prediction using Temporal Data

The prediction methods mentioned so far have only been using data from the current frame. If there would be a possibility to allow using temporal data, which would be pixel value data from previous frames, more advanced prediction methods could be developed. The high demands on limited hardware will put severe limitations on the amount of temporal data accessible. For example, to access the corresponding pixel of the pixel to predict from the previous frame it requires hardware with enough memory capacity for more than one entire frame. Because of the high definition resolution (1080p) that was being used, the hardware needs to have over $3MB$ of memory, depending on how many bytes each pixel should represented with. Therefore, only the case where one old frame is accessible was considered in this thesis.

In order to try this method, a new combined predictor has been implemented that uses the old pixel value in one of its predictors. The other predictors used in

this combined predictor would be *Mean4*, *LOCO-I* and *Mean2Long*. The prediction rule of the predictor using temporal data is described in the equation below.

$$\hat{x} = \frac{a + b + x_{\text{prev}} + g_{\text{prev}}}{4} \tag{3.3}$$

Where x_{prev} and g_{prev} was taken from the previous frame as the corresponding pixels as shown in Figure 3.1. The new predictor using temporal data will be referred to as *Time*. The *Time* predictor predicts which predictor to use in the same way as the *Combined* predictor does. The results from the *Time* predictor being combined with *Adaptive Golomb-Rice* and *Adaptive Huffman* encoders was compared with the *LMS* and the *Mean2Long* predictors using the same encoders. The limitation of comparisons here was made just to make the results easier to present. The main thing being of interest in this test was to show how temporal data could be used in order to improve the compression performance compared to using no temporal data at all. This was a method which have not been found in any literature, it was designed by ourselves.

To show these results a benchmark script was made that compares this new predictor to other high performing predictors. These results will be shown in the Benchmark section in Results, as the benchmark suits as a good reference for how good the compressions usually are.

3.2.4 Mapping of the Probability Distribution

Since the prediction error may take both positive and negative values as described in Section 2.3, the prediction error will be distributed as a two-sided geometric distribution. In order to get the resulting prediction error to be one-sided geometric distributed, the actual prediction error was replaced with the absolute value of itself. The original sign of the error was saved as the most significant bit used for representing the error. Also, the values of the prediction errors that required the same number of bits as the values of the pixel for representation was remapped as mentioned in Section 2.3. This was done to be able to represent the value of the error in one bit less than the number of bits used for representing the pixel values, so that the value of the errors combine with its sign bit would require the same number of bits as each pixel. During encoding, the sign bit was separated from the symbol value of the prediction error and only the symbol value was encoded. This was done because, due to the sign bits' random nature they can not be encoded in an efficient way. The sign bit were stored separately as it were.

3.2.5 Predictor Benchmarking

All the predictors were then used for the benchmarking script that was made in *Python*. This script is explained in detail in the Benchmark section.

3.3 Encoders

The encoders used for encoding the prediction errors are *Huffman*, *Adaptive Huffman*, *Golomb-Rice* and *Adaptive Golomb-Rice*.

In order to verify that each of these encoders works, a decoder was implemented for each of these encoders to verify that the compressed image could be decompressed properly.

3.3.1 The Huffman Encoders

Both the *Huffman* and the *Adaptive Huffman* encoder's implementations were based on their descriptions in the Sections 2.4.1-2.4.2.

When encoding using the regular *Huffman* encoder a predefined probability distribution was used that was based on the average error probability distribution of the images shown in Appendix A. This distribution was obtained using the *LOCO-I* predictor. A predefined probability distribution was used because if this method would be implemented on real hardware then it would require a static probability distribution that is not based on the current image. Therefore, using a predefined probability distribution that was hard-coded did seem to be good solution.

The *Adaptive Huffman* encoder starts off with an empty Huffman tree that is updated during runtime according to the adaptive Huffman updating algorithm.

3.3.2 The Golomb-Rice Encoders

Both the *Golomb* and the *Adaptive Golomb-Rice* encoders' implementations were based on the description in Section 2.4.3 in Theory of Golomb-Rice. The *Golomb-Rice* encoder was used with static parameters k and m, meaning they are the same for the whole image.

The *Adaptive Golomb-Rice* encoder on the other hand starts with predefined values of k and m that were then updated multiple times for each row.

The *Adaptive Golomb-Rice* encoder that was implemented divides each row of the image into a number of sections, which is similar to the section division in the *Combined* predictor. Then, using statistics gathered from the last section and the neighboring sections the parameter k was updated. This update algorithm was based on the parameter estimate shown in Equation 2.20. m was then also updated after k, as mentioned in Section 2.4.3. In order to reduce the number of operations needed in the *Adaptive Golomb-Rice* encoder, the output from Equation 2.20 was calculated and mapped for different μ. This resulted in a table where k can be obtained from different values of μ.

3.3.3 Encoder Benchmarking

All the encoders were then used in the benchmarking script that was made in *Python*. This script is explained in detail in the upcoming section.

3.4 Benchmark

In order to compare all the different combinations of predictors and encoders, a benchmarking script was written in *Python*. The benchmark was made in a way to produce various results for each combination together with a set of different

images. Each predictor-encoder combination produced a compressed file for each image. These files' sizes were then used to determine the compression ratios for each combination. The different compression ratios produced for that combination on all of the images were averaged and saved. The worst compression ratio achieved for each combination was saved separately.

After gathering all the compression ratio results, the standard deviations for the samples were estimated. The idea behind this was to try to get a grasp of each compression method's stability. The stability in this sense would mean how much the compression ratio might differ between different images or in other words, how small the standard deviation was.

In order to get a better idea on the predictors performance independently of the encoders, the entropy was calculated for each predictor for each and every image. Similar to the compression ratio statistics, the average entropy value was saved together with the worst achieved entropy. From the worst case entropy statistics, the lowest theoretically possible compression ratio was calculated for each predictor.

3.5 The Run-length Encoder

The *Run-length* encoder implemented was an encoder that used a form of bit-layered run-length encoding combined with a *Huffman* encoder. It used a form of run-length encoding for the upper bit layers to mark where the 1:s were, as described in Section 2.4.5. The encoder worked on one bit-layer at the time. The *Run-length* encoder left the 8 least significant bit-layers untouched. The idea was that this processing was very quick and had little overhead data needed if the upper bit layers consisted of almost only 0:s. The 8 least significant bit layers were then used to create a new image consisting of pixels that only had symbol values between 0 and 255. This allowed the encoder to quickly build a *Huffman* tree for every image and then compress it using *Huffman* codes.

The method was not used in the same benchmark as the others encoding methods, as it earlier was excluded as a potential encoder for this thesis. However, it was still compared to other encoders' compression ratios in different situations to ensure the reader's understanding that run-length encoding was not a stable choice of encoder.

To show the reason why run-length encoding could have been a good choice on a first glance, the execution time for building the tree for the normal *Huffman* encoder together with the execution time for building the tree for the run-length encoder were compared with each other.

3.6 Hardware Requirement Estimation

To get an estimate of how computationally complex a certain method was, assumptions had to be made and during this thesis a rough estimation had been performed. The computational cost estimate was based on how many operations that were performed by each algorithm for each pixel in the image and what type

of operations these were. One thing to consider regarding the results that was provided by these estimations was that they were very imprecise because they were made after our own implementation of the methods, which might not betimal. The complexity for a specific operation varies depending on the hardware implemented on. Furthermore, the cost for different conditional statements in the code were hard to evaluate. In order to get a rough estimate of the operation costs, the operation weighing was based on very general values from a programming website[11], which explains operation costs in clock-cycles.

In practice, the execution times of our scripts could be measured, but the results would not give a realistic result of the complexity because the code was made in *Python*. *Python* is a scripting language and a lot of computational power is directed towards type conversions and needs to be interpreted during runtime, compared to a programming language like C, where code is compiled before it is run. Using Pythons execution time to determine performance of a method was considered to give very skewed results and therefore a more hypothetical performance was approximated.

In addition to the computational complexity, an estimate of the required memory has also been made. This was done by looking at how the algorithms work in theory. This estimate would consider how many rows that has been used for the prediction algorithm and what temporary variables were required for the calculations. The encoders were also examined by looking at how much information they needed to store while the encoders were running. The results were roughly rounded to a multiple of kilobytes in order to easily get a good idea of the magnitude of memory needed. This was done because many of the algorithms were close to each other when it comes to memory requirements and looking at the small differences was not of any interest.

3.7 Implementation of Near-Lossless Compression Technique

3.7.1 Impact of noise on Compression

The image quality is affected by temporal noise from the sensor as mentioned in Section 2.6.1. It has been investigated in this work how this noise affects the compression of the image.

To be able to determine if the noise profile in the image affects the compression ratio, a simple noise removal filter was implemented and applied to a set of images. 50 images, taken one after another of the exact same scene. In order to reduce temporal noise, these 50 images were averaged into a single image. The resulted averaged image was then compressed using different compression techniques, the resulting compression ratio were then compared to the compression ratio obtained from compressing one of the images used for averaging, using the same compression techniques.

3.7.2 Near-Lossless Quantization

If the sensor in the camera introduces noise N_S, it would be good to understand how allowing different levels of quantization actually affected the compression ra-

tio. A proposed method using this technique was implemented to test this and is described in this section. When this was simulated, a rough noise model was used to describe the sensor noise. In the simulation, the present noise in each pixel were expected to have a maximum value of $N_{S,\max}$, which was obtained using the model as explained in Section 2.6.2.

After the prediction of a pixel, the prediction could be used to estimate the pixel intensity and thus, the amplitude of the noise could be calculated. By allowing a certain signal-to-quantization-noise ratio (SQNR) the value of the prediction error could be truncated by shift operations before it is encoded. The quantization error N_Q became uniformly distributed in $[-2^s + 1, 2^s - 1]$ where s is the total amount of shift operation made for truncating the prediction error. If the sensor noise was expected to have the maximum value of $N_{S,\max}$, the amount of shift operations that could be used was calculated by using the equation

$$SNQR \leq \frac{N_{S,\max}}{2^s - 1}, \tag{3.4}$$

which can be rearranged to

$$2^s \leq \frac{N_{S,\max}}{SNQR} + 1. \tag{3.5}$$

From this expression, s could be obtained. This is done by counting the amount of shift-right operations on the right-hand side of the expression until the result becomes equal to 0, the amount of shift operations was $s + 1$. The added 1 to the equation was done in order to not include the most significant bit in the noise, but only the numbers leading up to it, to make Expression 3.4 hold. The result was the truncated pixel error value by shifting s times. This introduced the quantization noise N_Q that was limited by

$$2^s - 1 = N_{Q,\max} \leq \left\lfloor \frac{N_{S,\max}}{SNQR} + 1 \right\rfloor, \tag{3.6}$$

where $\lfloor x \rfloor$ was the integer part of x. By doing this truncation, the prediction error values could be truncated by doing a shift-right operation s times before encoding. The decoder did, after decoding the error value, perform a shift-left operation s times to obtain the original, but quantized, prediction error value.

As an example, the following case could be considered. If the maximum sensor noise $N_{S,max}$ is 64 for a specific pixel and the allowed SQNR is 4, the restriction will become $2^s \leq \frac{64}{4} + 1 = 17$. In binary representation, 17 is equal to 10001_2 and can be shifted to the right 5 times until the result becomes 0. This means that s can be obtained as $s = 5 - 1 = 4$.

The procedure of bit-shifting the pixel error values before the encoding step was implemented in a new benchmarking script. To produce clear results, only two images were used for calculating the compression ratios. However, these two images were different when it comes to how much they are being able to be compressed.

One of the two images was the **test1.pgm** image, which was an image which has produced high compression ratios for all methods in the benchmark program. The other image called **test24.pgm** that has always produced low compression ratios was added for comparison. The choice of the images was based on the fact

that the technique of quantization might be much more worthwhile for images where good compression ratio was not possible without it. A high average compression ratios was a desired outcome, but the worst-case scenario was still a big factor when it came to evaluating a methods performance. Therefore, a plot of the relative increase in compression ratio was also made to highlight this significance.

3.8 Horizontal and Vertical Blanking

As mentioned in Section 2.5, the raw image from an image sensor does often include some sort of areas which almost only contains zeros. This due to the horizontal and vertical blanking. In order to see how this affects the compression ratio of the algorithms developed and evaluated during this thesis were rows and columns of zeros added to images and then compressed. The resulted compression ratio is then compared with the compression ratio of the original image where no zeros were added. From this the compression ratio for the areas with zeros is calculated.

3.9 Lossy JPEG

To get a reference of how well the compression techniques that were developed developed and evaluated during the thesis performed, they were compared with the lossy JPEG compression technique. It was done in *Python* by taking a raw image file and then save and compress it as an other file using the module imageio which got a lossy JPEG function. The size of the two images files was compared and the compression ratio was calculated. This was done for all of the images in Appendix A and then the average and worst-case were presented.

3.10 Comparison with PNG

To understand how well PNG performs in comparison to the benchmark tests in this thesis, an image was compressed using a PNG image converter application. The images were Bayer separated to remove the advantage the algorithms in the benchmark has. This advantage exists because the algorithms were hardcoded to do predictions while taking the Bayer pattern in consideration, and the PNG converter application can not be expected to do the same. The same image was then also compressed in the benchmark program with the **Adaptive Golomb-Rice** and **Mean2L** combination.

3.11 Error Handling

To be able to see the effects of an introduced bit error, a simulation of this was implemented. The simulation produced the decoded image as if no error handling method was being used. To solve the problem of bit errors, a simple method was developed and analyzed.

3.11.1 Proposed Method

To reduce the effects of a bit error in the transmission between the encoder and decoder it was suggested that each frame should be divided into sequences. Say that the rows of a frame are divided into 16 sequences by dividing the rows into 16 equally large blocks. This means, each block would include 68 rows for $1080p$ images. After entering a new block both the encoder and decoder assume that only the information within this new block are valid information. This means that the prediction rule, prediction statistics and encoding statistics which the compressor uses are reset to a default value so that the encoder and decoder can get back to become synchronized in the case that there was a transmission error in the precious block. The predictor is also then restricted to only use values from within its own block. If an error would occur in a block then the block would be destroyed and the corresponding block from the last frame could be used when representing the image instead. We assume that a block can be replaced by a block from a past frame without a person watching the video stream noticing, not anymore than a very small lag, as the frame rate is as high as 30 fps.

To be able to detect when a bit error occurs in the bitstream from the encoder to the decoder the bitstream needs to be sent in packets. The packets will also be used to keep count on how many symbols that has been transmitted in each block. These packets is suggested to have a fixed length of 1032 bytes where 8 of these bytes are used to represent how many symbols there are in this packet. Since the data which represent how many symbols there are in the packet are sensitive information is the value which represent how many symbols the packet contains repeated 5 times. This is based on the idea described about repetition codes as described in Section 2.10.1. Which means that there has to be a bit error in three of these five symbol fields in order for the decoder to misinterpret the symbol field's original value. In order to avoid the burst error as mentioned in Section 2.10.1, the 5 symbol fields will be spread out at predefined locations. Upon the end of each block of rows, the last packet sent in that block will only contain as many symbols as there are left in the block and the rest of the packet will be filled with zeros. This means that the next block will start of with a new packet which only contains symbols from the new block.

3.11.2 Testing the Proposed Method

In order to evaluate the proposed method some calculations were made to find out what limitations the method used for the transmission puts on the compression. Among these restrictions there is the amount of overhead data that will be introduced because of the symbol fields and how much the symbol field's size restricts the maximum compression ratio.

To test how much the block division will worsen the compression ratio, a predictor and an encoder combination using this division was implemented and tested in the benchmark. The combination used was *Adaptive Golomb-Rice* with *Mean2L*.

Chapter 4
Results

In the following sections, the results from the implementation will be presented. They will be presented in the order they have been presented in the previous chapter. In the next chapter, the results will be discussed and later on a conclusion based on the results will be made. The set of images used during this chapter for obtaining results is shown in Appendix A.

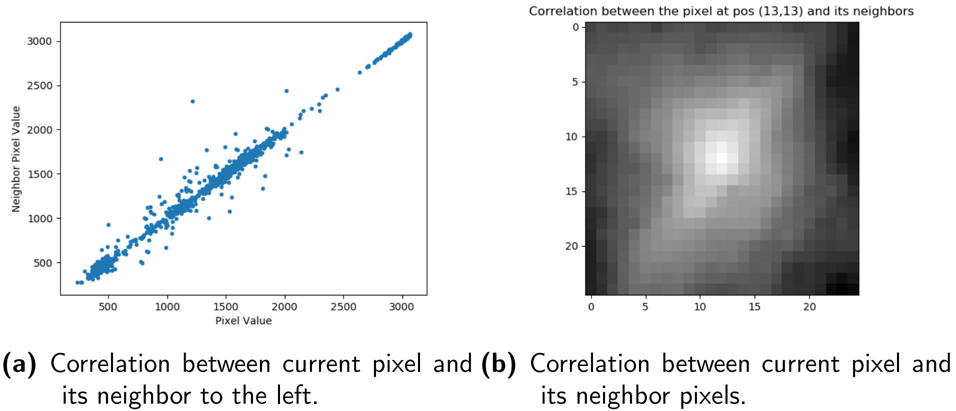
In the tables where results are presented, method names marked in bold font are methods which we have designed by ourselves without finding any information about them in literature. This does not mean that nobody has not used or published these methods before but rather that we discovered them on our own.

4.1 Correlation Analysis

From the *Python* scripts several plots were obtained. In Figure 4.1 the correlation in image **test12.pgm** is illustrated. As a reminder for the reader, these scripts were only testing the green pixels after a Bayer split.

In the plot in Figure 4.1a, each dot is represented by two values, x and y, where x is a chosen pixel's value and y is its neighbor's pixel's value. These two values can be seen as two stochastic variables, which implies that the closer to a straight line they appear to be the higher is the correlation between them.

In Figure 4.1b the Pearson correlation between a pixel, at position $(13, 13)$ in the plot and its neighbors are plotted. Here the neighbors are in different directions and distances that is represented by their position in the graph in relation to the center pixel. In the plot, the color intensity is normalized, meaning in this case, that the highest correlation is completely white and the lowest value is completely black.

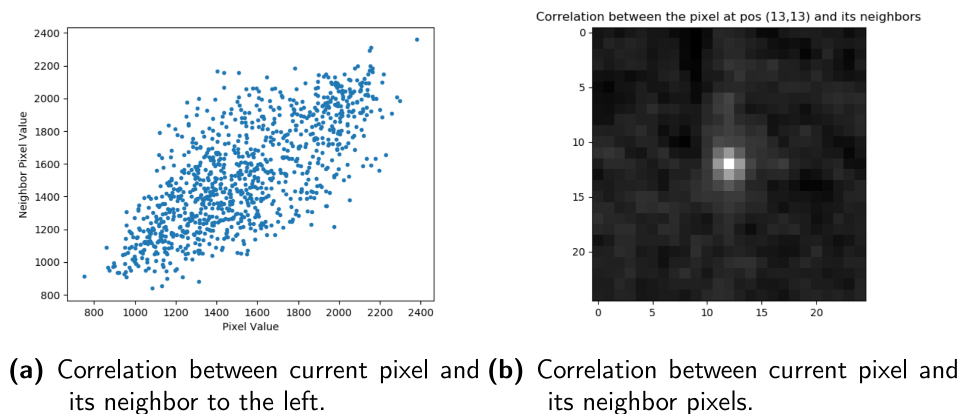


(a) Correlation between current pixel and its neighbor to the left.

(b) Correlation between current pixel and its neighbor pixels.

Figure 4.1: Illustration of correlation in image **test12.pgm**.

The corresponding figures are show in Figure 4.2, where the neighbor pixel correlation in image **test24.pgm** is investigated. Here it can be seen than the correlation between the pixels are not as strong in image **test24.pgm** as in **test12.pgm**. It can be seen from the higher spread of the dots in Figure 4.2a compared to 4.1a. What also can be seen is that the correlation is decreasing at a higher rate with growing distance from the center pixel in Figure 4.2b compared with 4.1b.



(a) Correlation between current pixel and its neighbor to the left.

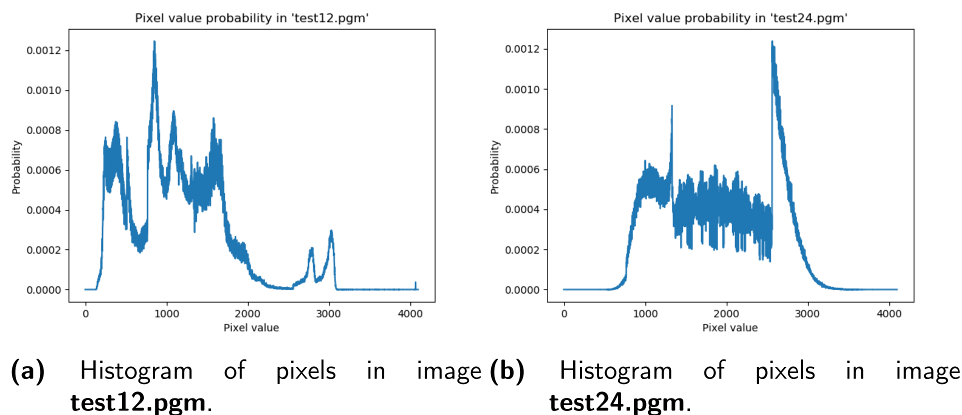
(b) Correlation between current pixel and its neighbor pixels.

Figure 4.2: Illustration of correlation in image **test24.pgm**.

The normalization of the values in Figure 4.1b and Figure 4.2b shows the relative decrease in correlation. The absolute values for the minimum values are around 0.7888 for image **test12.pgm** and around 0.3291 for image **test24.pgm**.

4.1.1 Predictor Implementation

Before any prediction has been made, the probability distribution of the image's pixel values is examined. The probability distribution of the pixels in the images **test12.pgm** and **test24.pgm** were obtained by running a *Python* script. The results are shown in the Figure 4.3.



(a) Histogram of pixels in image **test12.pgm**.

(b) Histogram of pixels in image **test24.pgm**.

Figure 4.3: probability distribution of pixel values.

These probability distributions resulted in the entropy 10.97 bits respective 11.10 bits for the two images. When predicting the two images using the *LOCO-I* predictor, without making the error only take positive values and without remapping it, the probability distribution shown in Figure 4.4 was obtained.

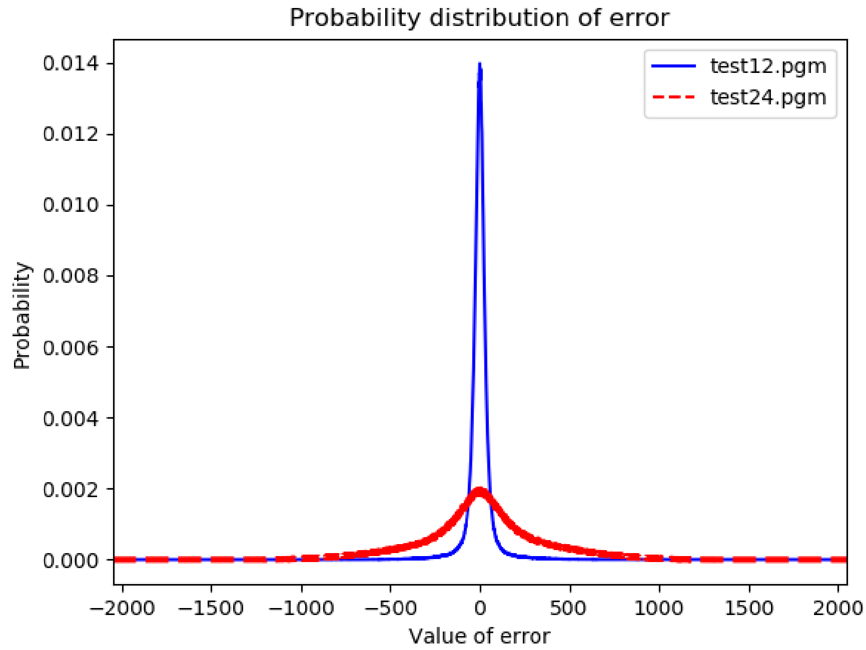


Figure 4.4: Probability distribution of two sided prediction error.

As it can be seen, the probability distribution of the predicted error in image the images **test12.pgm** and **test24.pgm** are two-sided geometrically distributed centered around zero. The entropy of the prediction error in image **test12.pgm** is 7.59 bits and the entropy of the prediction error in image **test24.pgm** is 10.33. If the error is instead mapped to be one-sided, the following probability distribution of the error is obtained.

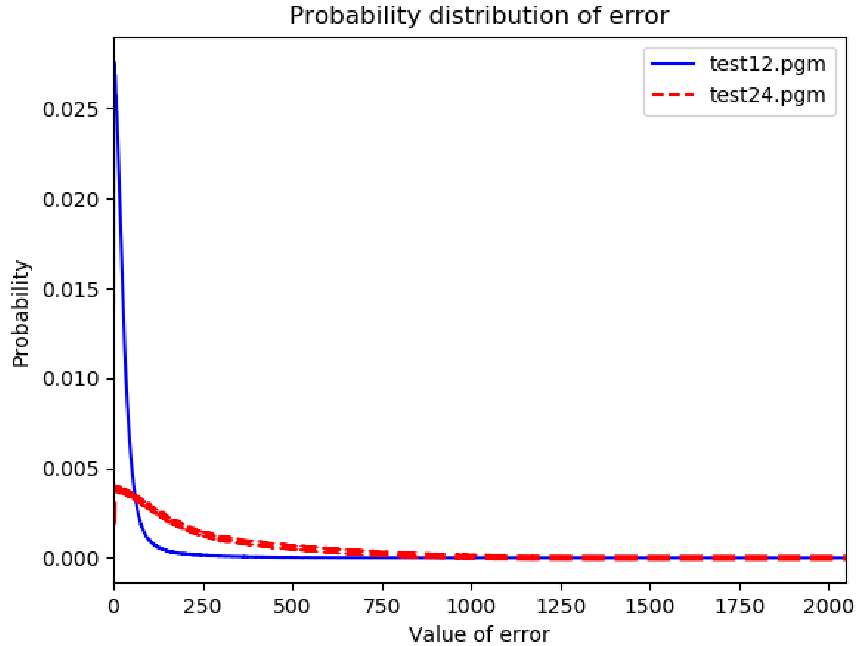


Figure 4.5: Probability distribution of one sided prediction error.

From the probability distribution in Figure 4.5 the entropy was calculated. For **test12.pgm** the entropy was 6.60 bits, while it was 9.33 bits for the image **test24.pgm**. Since the errors were one-sided and the sign of the number is lost, one additional bit must be added to these entropy values.

4.2 Benchmark

In the benchmark the 50 images shown in Appendix A were used. The mean of the compression ratios are shown in Table 4.1 and the worst achieved compression ratio for each combination are shown in Table 4.2. The results are represented in percentages and any negative values means an increase of the file size of that amount instead of a compression of that amount.

As a reminder for the reader, as mentioned in the beginning of Theory, the compression ratio is defined as: $CR = 1 - \frac{\text{Compressed data size}}{\text{Original data size}}$.

	Golomb-Rice	**Adaptive Golomb-Rice**	Huffman	Adaptive Huffman
Simple	0.575332	28.9613	24.6330	28.2528
Mean2	32.4150	38.7146	37.0990	37.7708
Mean4	31.2499	38.8285	36.8677	37.5603
Mean4-R	30.3280	38.6002	36.6311	37.3157
Mean2L	32.6824	39.0323	37.3337	38.0468
Mean2L-2	32.6461	39.0814	37.2994	37.9980
Median	28.6938	37.2351	36.0190	36.6333
LOCO-I	32.6665	38.1162	36.6196	37.3433
Paeth	31.3420	37.4897	36.0812	36.7943
Combined	34.8596	39.7543	38.2790	39.0870
LMS	33.7290	38.9185	37.3286	38.0081
LMS2	33.7882	38.9632	37.3326	38.0209

Table 4.1: Average compression ratio (%) benchmark results.

	Golomb-Rice	**Adaptive Golomb-Rice**	Huffman	Adaptive Huffman
Simple	-246.302	0.286949	-27.6244	4.13832
Mean2	-62.2371	14.2381	0.366498	14.3990
Mean4	-68.6471	13.5352	-1.23117	13.7576
Mean4-R	-72.5712	13.1383	-1.79703	13.3445
Mean2L	-62.4526	14.2263	0.0805504	14.3902
Mean2L-2	-61.4786	14.3451	0.177408	14.4965
Median	-78.9143	12.4331	-2.37588	12.7388
LOCO-I	-70.4697	13.2378	-0.780611	13.5963
Paeth	-76.4615	12.5966	-1.70422	12.9908
Combined	-64.2633	13.9828	0.00191483	14.1760
LMS	-60.4147	14.4592	0.464888	14.6168
LMS2	-60.1835	14.4890	0.435336	14.6416

Table 4.2: Worst-case compression ratio (%) benchmark results.

The entropy results are shown in the Table 4.3. The theoretically maximum compression ratio achievable in the worst-case image compression are presented in the *Compression Potential* column. As mentioned in 3.4, the maximum compression ratio is obtained from entropy.

	Average Entropy	Worst-case Entropy	Compression Potential (%)
Simple	8.57931	11.4638	0.446770
Mean2	7.44380	10.2398	14.6680
Mean4	7.46785	10.3120	14.0663
Mean4-R	7.49611	10.3602	13.6642
Mean2L	7.41089	10.2398	14.6681
Mean2L-2	7.41639	10.2270	14.7746
Median	7.57509	10.4325	13.0618
LOCO-I	7.49446	10.3367	13.8604
Paeth	7.55772	10.4052	13.2898
Combined	7.28607	10.2654	14.4548
LMS	7.41509	10.2142	14.8814
LMS2	7.41362	10.2104	14.9037

Table 4.3: Entropy benchmark results.

The standard deviation was calculated using the results. The standard deviation results are shown in table 4.4

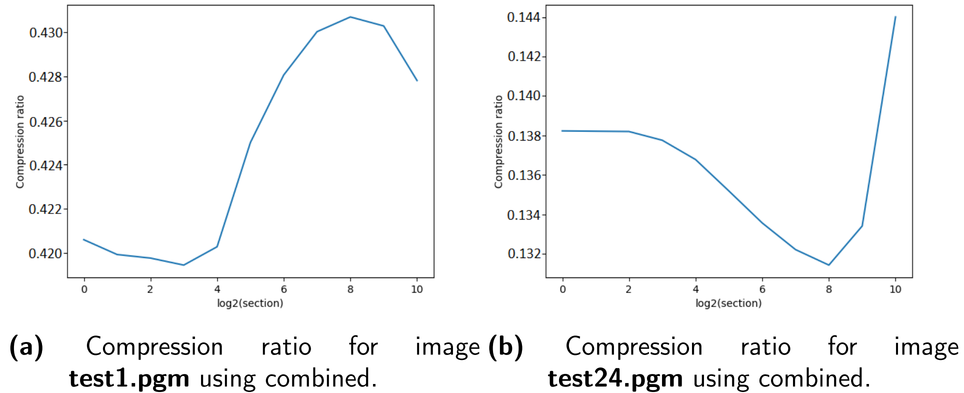
	Golomb-Rice	**Adaptive Golomb-Rice**	Huffman	Adaptive Huffman
Simple	0.446427	0.0796449	0.121084	0.0747887
Mean2	0.146384	0.0443408	0.0608004	0.0448311
Mean4	0.155837	0.0459244	0.0632387	0.0463743
Mean4-R	0.161257	0.0467571	0.0642144	0.0470724
Mean2L	0.147686	0.0449563	0.0617643	0.0457435
Mean2L-2	0.146023	0.0447823	0.0615062	0.0454549
Median	0.167423	0.0448861	0.0639109	0.0460396
LOCO-I	0.158601	0.0452430	0.0626440	0.0454501
Paeth	0.166083	0.0452543	0.0634507	0.0455170
Combined	0.150931	0.0456304	0.0627778	0.0467877
LMS	0.144754	0.0447317	0.0613412	0.0455852
LMS2	0.144517	0.0447952	0.0614665	0.0457211

Table 4.4: Standard deviation of compression ratio benchmark results.

4.2.1 The Combined Predictor

The combined predictor's performance dependence was analyzed. In Figure 4.6 the compression ratios of the combined predictor using different amount of sections

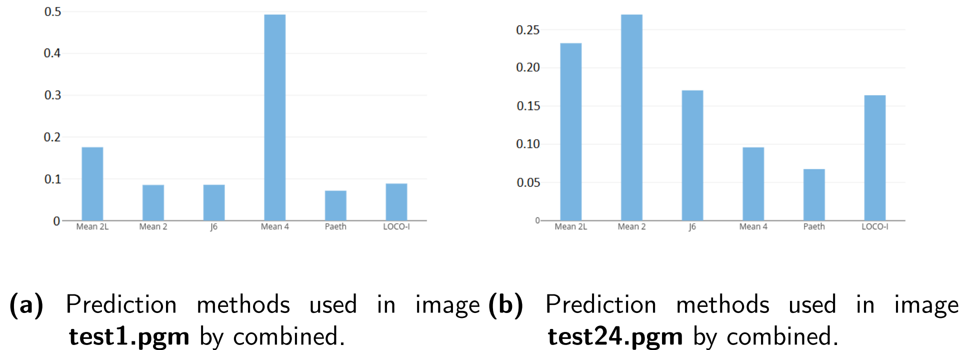
for the images **test1.pgm** and **test24.pgm** are shown. The scale of the x-axis is 2-logarithmic, while the y-axis is linear.



(a) Compression ratio for image **test1.pgm** using combined.
(b) Compression ratio for image **test24.pgm** using combined.

Figure 4.6: Compression ratio for different amount of sections that each row are divided into.

When dividing each row into 128 sections, how much each prediction method was used in the *Combined* predictor for the images **test1.pgm** and **test24.pgm** is shown Figure 4.7. The results are in percentages.



(a) Prediction methods used in image **test1.pgm** by combined.
(b) Prediction methods used in image **test24.pgm** by combined.

Figure 4.7: Methods used in the combined predictor when using 128 section for each row.

4.2.2 Prediction using Temporal Data

The comparisons made using the *Time* predictor using temporal data was compared against other compression methods. The results are shown in Table 4.5. The images used are shown in Appendix B.

	Adaptive Golomb-Rice	Adaptive Huffman	**Adaptive Golomb-Rice** Worst-case	Adaptive Huffman Worst-case
Time	37.6033	36.5367	27.9825	26.6209
LMS	36.2077	35.0452	25.7009	23.7687
Mean2L	36.0564	34.9646	25.2689	23.5140

Table 4.5: Comparison of compression ratio (%) using *Time* predictor with other predictors.

4.3 Run-length Encoding

In Table 4.6 some compression ratios are shown for two different images. The image **test1.pgm** is an image which has provided good compression results for all predictors and encoders in the benchmark, while **test24.pgm** has provided poor compression results.

	Adaptive Golomb-Rice	Huffman	**Run-Length**
test1.pgm	43.0157	41.7430	41.9911
test24.pgm	14.2195	0.01746	-0.133836

Table 4.6: Comparison of compression ratio (%) using run-length encoding with other encoders using *Mean2L* predictor.

The execution for the time in Python to build the tree for Huffman and all 4096 symbols could take more than 4 seconds, while building the tree for the run-length encoder did not take more than 0.1 second.

4.4 Sensor Noise

4.4.1 Impact of Noise on Compression

A new image was created by averaging the set of 50 images in Appendix A. It was put in to the compression benchmark and compared against another image from the original image set. For the averaged image, the benchmark produced the results shown in Table 4.7. For the other, not averaged image, the produced results are shown in Table 4.8.

	Adaptive Golomb-Rice	Adaptive Huffman
Mean2L	55.6455	55.3711
LOCO-I	53.7829	53.4618
Combine	54.6360	55.4679

Table 4.7: Compression ratios (%) for averaged image.

	Adaptive Golomb-Rice	Adaptive Huffman
Mean2L	42.1613	42.0778
LOCO-I	39.7043	39.5638
Combine	41.7973	42.0243

Table 4.8: Compression ratios (%) for non-averaged image.

4.4.2 Near-Lossless Compression

As mentioned earlier, compressing the image **test1.pgm** has achieved good compression ratio results in the benchmark, while the compression of image **test24.pgm** performed worse.

A graph was made to show how compression ratio varies depending on the allowed SQNR parameter for each image, this graph is shown in Figure 4.8. Notice that the x-axis are in a logarithmic scale and that the y-axis is zoomed in to only display the relevant part.

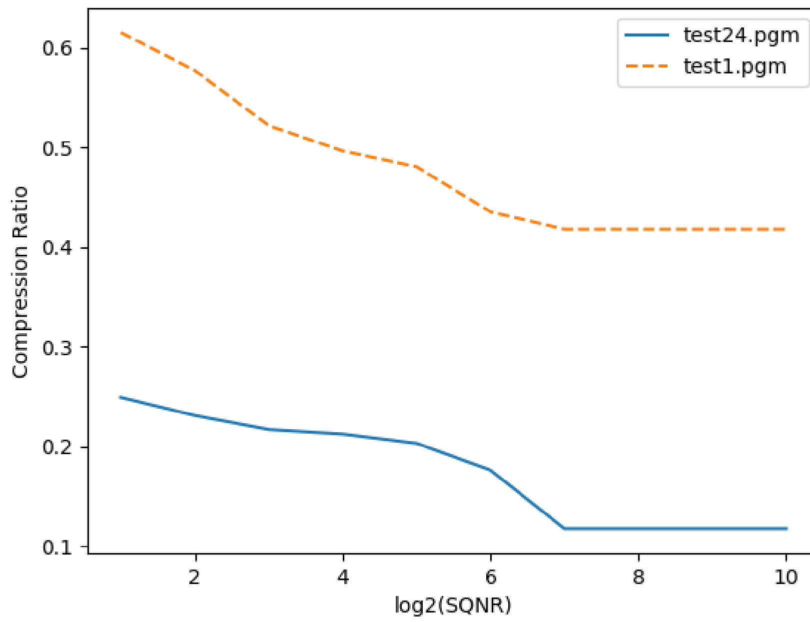


Figure 4.8: Compression ratio for different SQNR parameter values.

Figure 4.9 highlights how the compression ratio using noise quantization relates to the compression ratio without any quantization. It instead shows the increase in compression ratio in comparison with without using quantization.

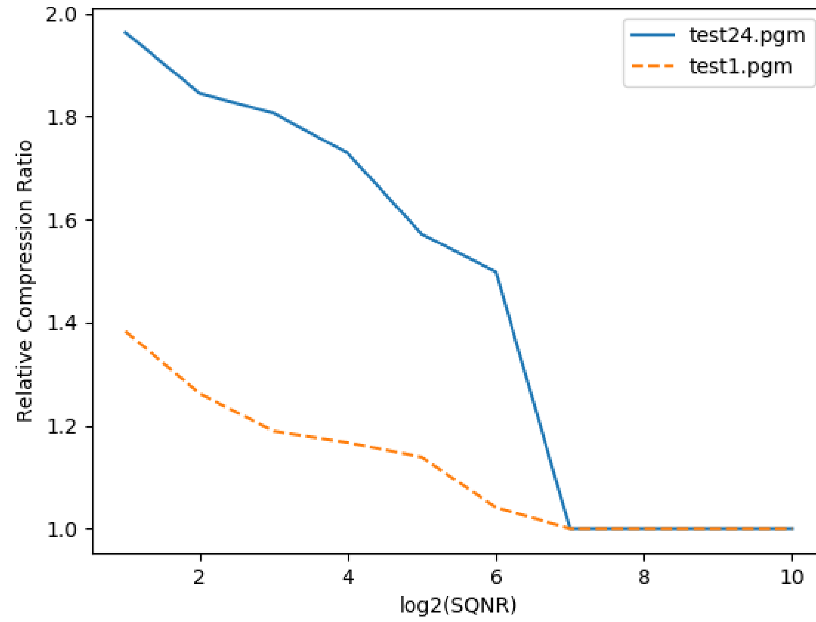


Figure 4.9: Relative Compression ratio for different SQNR parameter values.

4.5 Analysis of Horizontal and Vertical Blanking

When adding 100 rows below and 100 columns to the left of the image **test1.pgm** shown in appendix with zeros it resulted in the image shown in Figure 4.10. Both of the images were then compressed. The resulting compression ratio shown in Table 4.9 were then obtained. The compressing was done using the predictor-encoder combination of *Mean2L* and *Adaptive Golomb-Rice*.

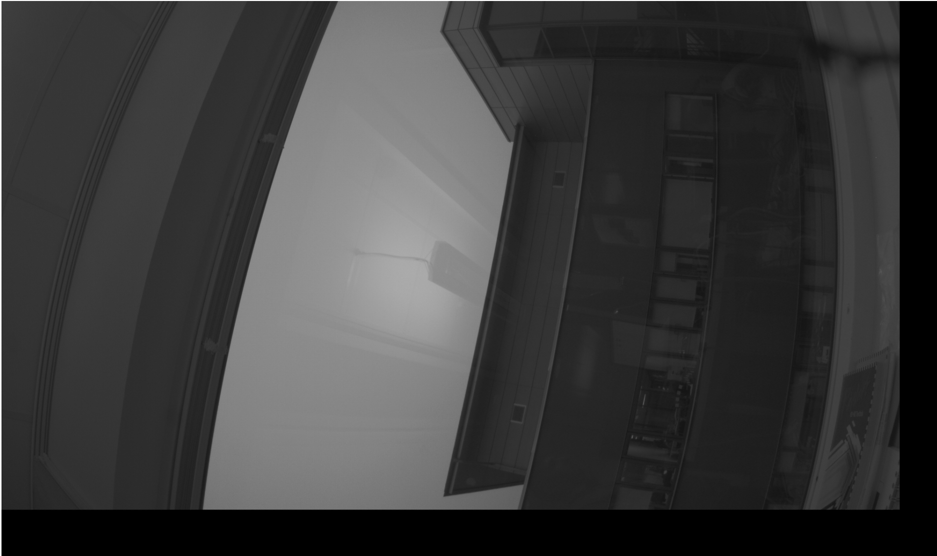


Figure 4.10: The image **test1.pgm** with horizontal and vertical blanking.

	Mean2L - Adaptive Golomb-Rice
test1.pgm	43.0119
Padded image	46.1391

Table 4.9: Compression ratio (%) for image with and without horizontal and vertical blanking.

If compressing the horizontal and vertical blanked image and then comparing the size of the compressed file and the size of the original image **test1.pgm**, then a compression ratio of 38.1255% could be achieved.

When adding zeros in the *test1.pgm* image it increases the file size by 3729600 bits. The compressed version of the image in Figure 4.10 are 1224894 bits larger than the compressed version of the original **test1.pgm** image. From this one can approximate the compression ratio for the blank area to be

$$1 - \frac{1224894}{3729600} \approx 0.6715 \tag{4.1}$$

.

4.6 Lossy JPEG

When compressing using *lossy JPEG*, a parameter that controls the quality of the image after compression may be set. The resulting compression ratio in average

and in worst-case are shown in Table 4.10. In this table, the standard deviation and mean squared error between the pixel values in the original and compressed image are also shown. This was done using the images in Appendix A.

	$quality = 100\%$	$quality = 10\%$
Average compression rate	78.8774	97.5650
Worst-case compression rate	62.6362	92.9070
Standard deviation	4.56383	1.60324
Mean squared error	29.8582	29.7264

Table 4.10: Compression ratio (%) using *Lossy JPEG* for different quality settings.

In Figure 4.11 and Figure 4.12 the compressed images are shown using different quality settings. It can be seen how the compression affects the quality. In Figure 4.13 the original image can be seen.

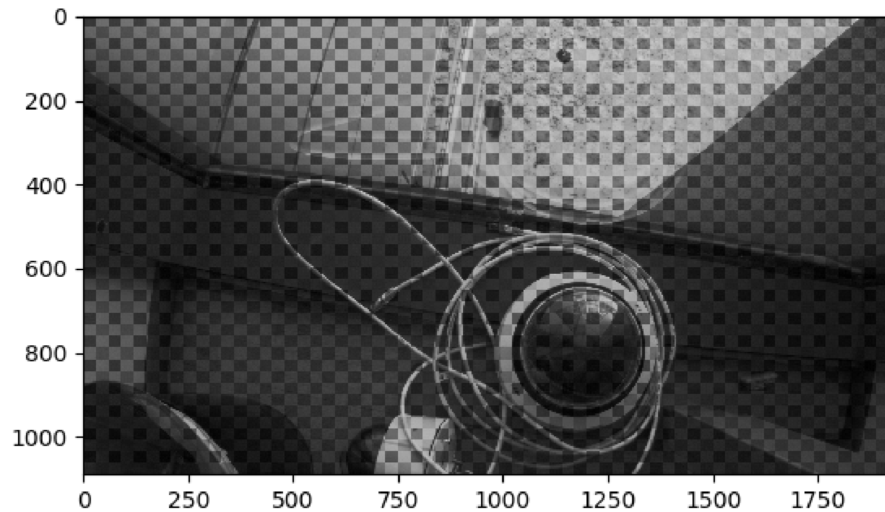


Figure 4.11: Compressed image using $quality = 100$.

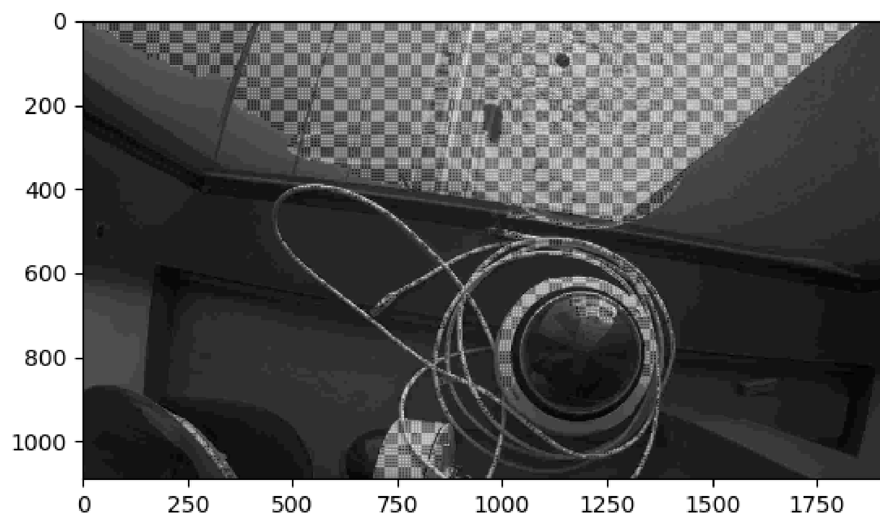


Figure 4.12: Compressed image using $quality = 10$.

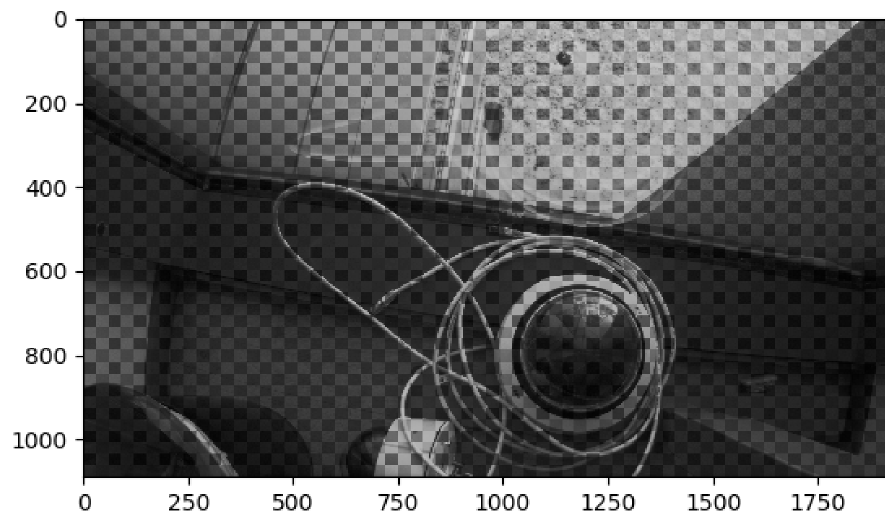


Figure 4.13: Original image.

4.7 Comparison with PNG

To get some reference results the Bayer separated version of the image shown in Figure 4.13 was compressed using the *PNG* standard. The PNG compression provided a file size that was 23.74% smaller compared to the original file size. When the same Bayer separated image was compressed in the benchmark program with *Adaptive Golomb-Rice* and *Mean2L*, it provided a file with a size that was 50.00% smaller than the original file size.

4.8 Computational Complexity

The computational complexity was estimated for every predictor and encoder. Table 4.11 shows the estimated computational cost for each pixel. The estimated memory requirements are also shown in a separate column.

	Computational Cost	Memory Requirements (KB)
Simple	9	<1
Mean2	12	8
Mean4	16	8
Mean4-R	42	3
Mean2L	15	8
Mean2L-2	26	8
Median	25	8
LOCO-I	25	8
Paeth	26	8
Combined	87	15
LMS	66	8
LMS2	102	8
Golomb-Rice	21	0
Adaptive Golomb-Rice	24	8
Huffman	1	20
Adaptive Huffman	120	256

Table 4.11: Estimated computational cost for the different techniques.

Once for each section in the *Combined* predictor, the next predictor needs to be predicted. The computational cost for this operation was not included in the estimation shown in Table 4.11 since the computational cost in the table are on pixel basis. The cost for predicting which prediction method to use in *Combined* is estimated to require 30 clock-cycles per section.

The memory requirements shown in Table 4.11 are all rounded to a multiple of kilobytes. These estimates were made as if the variables would be represented by 4 bytes. If the variable values would be represented by 2 bytes, the memory requirements would be halved. In a few cases, some floating point variables are needed. However, there will only be a few of these, which makes the additional cost negligible in comparison to the 2176 integer values that are needed for just two rows of pixels.

4.9 Error Handling

If a bit error would occur in a frame without any error handling the error would propagate and spread as shown in Figure 4.14. The original image is **test1.pgm**, that is shown in Appendix A.

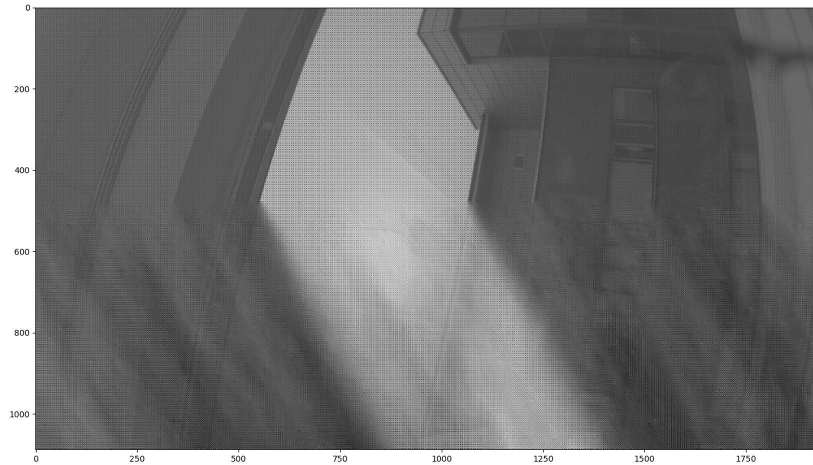


Figure 4.14: Example of how a bit error in the transmitted bitstream affects the decoded image.

It can be seen that somewhere in the middle of the image the transmission error occur and all the pixels that com after that become inaccurate.

4.9.1 Proposed Error Handling

Dividing Image into Blocks

The error handling suggested in Section 3.11 puts some constrains on the compression ratio. When dividing an image into blocks the compressing ratio possible to achieve is reduced. In Table 4.12 are the resulting compression ratios from when compressing the images shown in Appendix A using the *Mean2L* predictor combined with the *Adaptive Golomb-Rice* encoder dividing the video frame into blocks compared with the compression ratio when not dividing the image into blocks. The prediction-encoder combination dividing the video frame into blocks are refereed to as "With error handling". Note that the average compression ratio without error handling differs from the one presented in Table 4.1 as an older version of the set of images were used to perform this experiment.

	Average compression ratio	Worst case compression ratio
Without error handling	0.399972	0.139738
With error handling	0.398461	0.138567

Table 4.12: How dividing a video frame into blocks effects the compression ratio of the *Mean2L-Adaptive Golomb-Rice* combination.

Packet Protocol

The design of the packets puts constraints on the maximum compression ratio that possible to achieve. When using 12 bits in each of the 5 symbol fields, the symbol fields are able to represent at most the value 4095 which limits the number of symbols that can be included in the packet. If using 1024 bytes for representing the codewords of the symbols, then the highest achievable compression ratio can be calculated as

$$\begin{aligned} \mathrm{CR}_{max} = 1 - \frac{\text{(Bytes used in protocol)} \cdot \text{(bits per byte)}}{\text{(Max symbols in protocol)} \cdot \text{(bits per symbol)}} \\ = 1 - \frac{1024 \cdot 8}{4095 \cdot 12} = 0.83. \end{aligned} \quad (4.2)$$

Besides the limitation mentioned above, the header data also needs to be sent. By using 8 bytes for the header the transmitted data will increase by

$$8/1032 \approx 0.007752, \quad (4.3)$$

which is less than one percentage. Besides this header data the last packet of a block needs to fill the rest of its codeword field with zeros after the last symbol for that block. The amount of zeros in the last packet could be considered to be uniformly distributed with an average of 512 Bytes. Since it varies how many packets each block contains based on the compression ratio, it can not be determined how much this is in average for all packets. If it is assumed as an example that the compression ratio is constantly 33% it means that each symbol will be compressed to require 8 bits and thus each packet will contain 1024 symbols. If there are 16 blocks per frame it means that each block contains 68 rows, where each row contains 1920 pixels. This means that there are

$$\frac{68 \cdot 1920}{1024} = 127.5 \quad (4.4)$$

packets per block. Rounded up it gives 128 packets per block and this means that in there are in average

$$\frac{512\text{B}}{128} = 4\text{B/packet} \quad (4.5)$$

of wasted space per packets, which comes from not filling up the last packet in the block. With this we can calculate the total overhead data for each packet as

$$\frac{(8+4)\text{B}}{1032\text{B}} \approx 0.01163. \quad (4.6)$$

Chapter 5

Discussion

The results from the implementation in this thesis will be discussed and connected to the theory. The discussion chapter has a structure based on the ordering in the result chapter. First, the correlation will be discussed and how it has been used in prediction and why the performance differs significantly for different images. Then, the results from the benchmark will be discussed and how different predictors and encoders affect the compression results. The hardware demands are mentioned and why the method used for estimating hardware demands is not exact. Afterwards, near-lossless quantization and how higher compression ratio can be achieved, without affecting the image quality significantly, will be discussed. Later on, transforms are discussed and why it was decided not to implement any transform algorithms. At the end of the chapter, what can be done next in order to extend this research will be discussed, together with criticism on this report.

5.1 Correlation Analysis

How well a pixel is being able to be predicted from its neighbors is highly dependent on how much they are correlated. In the Results chapter this is clearly shown and in the following sections, the correlation test results will be discussed in order to get a better understanding.

5.1.1 Correlation to Neighbors

When comparing the correlation between a pixel and its left neighbor in image **test12.pgm** and **test24.pgm** some differences can be seen. The correlation in image **test12.pgm** is stronger since most of the dots in Figure 4.1a are aligned, this means that the pixels' values are dependent on each other. Given a pixel value, its neighbors can be estimated based on the given pixel value. The dependency between the pixels is not as strong in the image **test24.pgm**, as it can be seen in Figure 4.2a. The pixels spread is much larger than in the Figure 4.1a, the spread is almost diamond shaped. This means that given a pixel value, the pixel's neighbors' values can not be determined with equally high accuracy. The pixels in **test24.pgm** are probably less correlated to each other than the pixels in image **test12.pgm** because of the image **test24.pgm** contains more structures

and appears more random than the image **test12.pgm**. The visible structure is what makes the image less predictable.

5.1.2 Spatial Correlation

From the Figures 4.1b and 4.2b it can be understood why only the close neighboring pixels are useful to use for prediction and why the predictors that uses this information to their advantage obtains a much higher compression ratio. In the figures it can be seen that the correlation between the pixels that are close to each other are strong, while pixels with some distance between them have low correlation. As one might expect, the correlation between pixels decrease the further away the pixels are from each other. Therefore, since the correlation is very strong with neighboring pixels, an image that uses a much higher resolution might expect a greater correlation between its pixels and their neighbors. This might be true in most cases because the spatial distance between the pixels decreases when the resolution increases and they become more densely packed.

It is rather important to not misinterpret these graphs. Without an explanation they might be rather misleading. The absolute values of the correlation is also of interest. The darkest spot in Figure 4.1b is much higher, around 0.7888, than the darkest spot in Figure 4.2b which is around 0.3291. So apart from the correlation just fading off quickly, it also fades down to a much lower value. In other words, the pixels that are pitch black in Figure 4.1b would still have a rather high correlation of around 0.7888 that could be used for prediction. The minimum value could be seen as a direct measurement of how good any prediction method could be, while the relative values tells more about which pixels to use for prediction.

5.1.3 Error Probability Distribution

When using the *LOCO-I* predictor, the pixels in image **test12.pgm** are predicted more accurately than the pixels in image **test24.pgm**. This can be seen in Figure 4.5 that the probability distribution of the errors predicted from the image **test12.pgm** are more narrowly concentrated and centered around zero compared to the errors predicted from the image **test24.pgm**. This means that the error predictions of **test12.pgm** are better and therefore the image can theoretically get a higher compression ratio.

In general, the predictors are used to get a more structured pixel probability distribution. In Figure 4.3a and Figure 4.3b it can be seen that the probability distribution of the pixels values of the images **test12.pgm** and **test24.pgm** are random and very unstructured. After prediction, the probability distributions of the errors of both images become structured in a similar way. They will both have a geometric distribution of the prediction error, but they will fade towards zero at different paces. This can be seen in Figure 4.4, where the figure shows the two-sided geometric distribution of the prediction error. Since Golomb-Rice encoders are, according to theory, optimal for one sided geometric distributions, the prediction error have been remapped to only take positive values. This is done without losing any information and each prediction error is remapped to its positive value. One extra bit is saved that used for representing the prediction error's sign and handled

separately. The obtained one-sided probability distribution can be seen in Figure 4.5.

5.2 Benchmark

Apart from being a very time-saving tool, the benchmark program has helped by producing results that shows the performance for each predictor-encoder combination. In general, the compression ratio tends to become better the higher the computational complexity of the prediction-encoder combination. However, this tendency is not strictly true and in the following sections this will be discussed together with the benchmark results in detail.

5.2.1 Entropy Analysis

As mentioned in Theory, the entropy is a measurement of the theoretical minimum amount of bits required to encode each symbol in a sequence of symbols. Therefore, you could conclude that the lower the entropy, the better compression ratio. This is however not always true. For example, the *Mean2* predictor produces a worst-case entropy of **10.239840** which is worse than *Mean2L*'s **10.239828**, but actually gives a better compression ratio of **14.2381535** versus *Mean2L*'s **14.2263135** when using the adaptive Golomb-Rice encoder. These results were given from the same image, image **test24.pgm** shown in Appendix A.

These results are almost identical, but proves the fact that there are more factors than the entropy that affects the compression ratio. One possible reason for this result could be that the Golomb-Rice encoder works optimally when the probability distribution is geometrically distributed and the predictors might produce an prediction errors where the distribution is not shaped to the encoders advantage. In other words, resulting distribution's shape makes the encoder to assign codes to the symbols in a sub-optimal way, thus creating longer codewords than necessary. The entropy is still a important measurement when it comes to judging a predictors performance. However, when selecting a predictor to use, it is very important to test it with a big set of sample images, together with different encoders, in order to find the one that performs the best.

5.2.2 Average Compression

In general, the compression ratio from the encoders follows the average entropy. There is only one combination that does not produce any compression in average. This would be the *Golomb-Rice-Simple* combination. This might be because of how the *simple* predictor does not perform well, and therefore the error distribution is not geometrically distributed evenly throughout the whole image, thus giving a very low or negative compression ratio for many images. A negative compression ratio would result in an expansion in the file, meaning it takes more space than the original file. The *Adaptive Golomb-Rice* handles this by tuning the parameter k, which determines how the symbols should be encoded, for different sections of the image.

Adaptive Encoders

The encoder that produces the best average compression ratios is the *Adaptive Golomb-Rice* encoder, as it has the best ratio for each and every predictor. The *Adaptive Huffman* encoder is not far from it, but lacks the fast adaptation which the *Adaptive Golomb-Rice* encoder has. The current implementation of the *Adaptive Huffman* is not using any forgetting factor when it comes to its statistics, which means that when enough statistics have been gathered, any new pixel value that is processed will barely affect the overall statistics. *Adaptive Huffman* can also be implemented as a windowed version which means that it would only use statistics within the current window. Neither a version of *Adaptive Huffman* using a forgetting factor or window was implemented. Thus, *Adaptive Huffman* will have no adaptation to local variations within the image. This might be the reason why the *Adaptive Golomb-Rice* encoder outperforms *Adaptive Huffman* on average. In addition to this, the *Adaptive Huffman* starts encoding each image with an empty huff-tree which it has to build up. If it would have an already built tree at the beginning of each image, it might perform better. This could be done by saving the huff-tree between every frame or image. This was however not tested in the benchmark and could probably improve the performance slightly. This implementation was however left out from this thesis, as it is expected to provide very little improvement and there is much work to be implemented, like some form of forgetting of older statistics.

Static Encoders

The normal *Huffman* encoder performed considerably well and produced results that are just a few percentages lower than its adaptive counterpart. The normal *Huffman* uses statistics that are hard-coded, and not the statistics from the image which it is compressing. Since the statistics that the encoder uses is predefined, the normal *Huffman* encoder is extremely fast and requires no computations but only direct memory accesses to obtain the code for a given symbol. This makes *Huffman* an interesting candidate for this thesis. The same could be said about the normal *Golomb-Rice* encoder, which produces results slightly worse than *Huffman*. The advantage *Golomb-Rice* has is that no statistics is needed to be estimated but only a parameter.

5.2.3 Worst-case Compression

The wost compression ratios are all produced when trying to compress the **test-24.pgm** image, which has a very "noise-like" appearance. Because of this, the predictors struggle with predicting the pixel values and it results in that the encoders can not encode the error values efficiently. This leads to lower compression ratios, or even negative compression ratios in some cases. Negative compression ratio means that the size of the file has been increased.

Static Encoders and Negative Results

Negative results are only produced by the *Golomb-Rice* and the *Huffman* encoder and the reason for this is easy to understand. Since the *Golomb-Rice* encoder expects the probability distribution to be a geometric distribution with a certain shape. The *Huffman* encoder is implemented in a way which uses a predefined probability distribution, thus expecting the images' distributions to have a certain shape. The encoders will therefore encode the symbols with codewords that are mis-adjusted and too long if the image's real symbol probability distribution differs too much from expected model. This will lead to larger file sizes and for this reason, these two encoders will be left out of the discussion for the rest of this section.

Adaptive Encoders and Sections

The first thing that one might notice is how *Adaptive Huffman* is better in the worst-case scenario with all predictors in comparison with *Adaptive Golomb-Rice*. This could seem a bit surprising, as the average values were better for *Adaptive Golomb-Rice*. Because of how these two encoders have been implemented, there is a difference in how they might perform in images with high pixel correlation compared to in images with a lower pixel correlation. The *Adaptive Golomb-Rice* encoder is implemented in such a way that it divides each row of the image in a number of sections and tries to estimate what parameter value it should use for the next section to encode. The more noise-like characteristics the image has, the more difficult it becomes for the *Adaptive Golomb-Rice* to accurately predict the parameter value. *Adaptive Huffman* on the other hand has a much smoother way of tuning itself, as described in Section 3.3. This smoother way of tuning itself makes the *Adaptive Huffman* encoder more suitable for images having varying statistics. Therefore, the *Adaptive Huffman* will produce much better results with the worst-case image.

The *Combined* predictor is based on the same principle of trying to tune itself in sections as *Adaptive Golomb-Rice*. It can be seen from the results how the advantage *Combined* had in the average case, has instead turned into a disadvantage in the worst-case. The reason for this might be that the statistics in the images may change rapidly enough that the statistics in one section differs a lot from the statistics in the next section. This would make the prediction method predictor in *Combined*, more inaccurate for the next section. This would make the *Combined* predictor not to use the optimal prediction setting. The prediction settings might vary so much that they alter between a couple predictors but never uses the correct one because the one that it uses was optimal for the past section which had different statistics compared to the current section. Likewise, the *Adaptive Golomb-Rice* encoder suffers from the same issue when estimating its encoding parameter in a image with higher local variation of the statistics. This is why in the average case, the *Combined* predictor performs best among the predictors and *Adaptive Golomb-Rice* among the encoders, while they both perform the worse when compressing images with varying statistics.

5.2.4 Standard Deviation Results

The standard deviation of the compression ratio gave very consistent results for all different combinations of predictors and encoders. Only the combination using the *Simple* predictor with the *Golomb-Rice* encoder produced a high standard deviation that stands out from the rest. It is probably because that predictor-encoder combination is sensitive to images of varying environments. There are not many conclusions that can be drawn from the standard deviation except that most of the predictor-encoder combinations are stable and the compression ratio that they produce do not vary a lot. The standard deviation results will be used for further discussion later on in the report.

5.3 Hardware Demands

The computation costs in Table 4.11 are just estimations and are not exact. This means that if two methods are estimated to require the same number of clock-cycles per pixel it is not certain that the one that is more expensive actually is more expensive. The table can still be useful to get a general idea of how the techniques differs in computational cost. The more demanding techniques can be separated from the medium demanding and even less demanding techniques.

5.3.1 Computational Cost

The computational cost is a rough estimate, because it is hard to get a accurate estimate of how many clock-cycles different operations require. How many clock-cycles different operations requires is hard to estimate because it is hardware dependent. One other factor that makes the estimate inaccurate is that it is hard to calculate the mean number of operations that will be used for each pixel in the different algorithms. The mean number of operations per pixels can vary depending on how the operations are performed, especially when considering conditional branches in the program. During the computational cost evaluation in this report the main focus of the predictors were on the algebraic expressions of their prediction rule. For each prediction rule the different required operations were counted and then weighted based on how many clock-cycles the operation is estimated to require. For example, an addition operation was assumed to only require one clock-cycle while a multiplication required four clock-cycles. The computational cost of the encoders are based on their implementation. The assumptions made on how many clock-cycles each operation cost might be inaccurate, therefore making the Table 4.11 a bit untrustworthy. As mentioned earlier, the table still gives the general idea and a good overview of the computational cost.

5.3.2 Execution Time as Measurement

One might think that measuring execution time would be a good idea to get a rough idea of the hardware requirements, but this would most likely be even more inaccurate. The language used to implement the algorithms has been Python,

which is a high-level scripting language. A script has to be interpreted and translated into instructions and because of the flexibility of for example, data types, there is little optimization done. This makes the program take a long time to execute, no matter the type of instruction made. Another factor to consider is that Python is run on a operating system, such as Windows or Linux, that are not made for real-time applications like high-speed real-time video compression. To get a good idea of the exact performance for an algorithm, the best way would be to program it directly on the hardware and do real-time measurements. This is however, outside the scope of this thesis.

5.3.3 Memory Requirements

The memory requirement estimates presented in Table 4.11 are rounded to an amount of kilobytes. This was done because the differences between many of the predictors were so small that it is not worth mentioning. It is also appropriate to do so because the estimates are very roughly made and presenting in kilobytes provides a good overview. Most prediction algorithms do only require two rows of memory at most. This is because if you want to be able to keep the previous row of data you will need to have it stored separately. However, this requirement might differ between different sensors, because of how they provide the data. The sensor, that has been used in this thesis, provides the pixel data by sending one row at a time. For this reason, some of the algorithms need to have two whole rows of data stored simultaneously, but in theory it could be done better if the pixels were sent in a different manner. If the pixels would be received one at a time, a buffer could be used that stores the previous pixel values until the one that is just above the current pixel is stored. This would require a buffer of the size of only one row. By doing this, a circular buffer would have to be implemented which is not required in the case where they are sent row by row. A circular buffer would add some more computational requirements. So in other words, there is a trade-off between memory needs, computational performance and complexity.

Memory Cost of Predictors

The results are in general straightforward for the predictors. The simple conclusion would be that, the more rows that are required by the algorithm the more memory is needed. There only exception to this would be the *Combined* predictor which uses method prediction that requires memory depending on the amount of sections used. The sections used in our benchmarks were 128 per row but the amount of sections could be lowered while not losing much compression ratio. It is up to the user of the algorithm to decide how much memory that could be set a side for the method prediction. This means that, in this case there is a trade-off between memory needs and compression performance.

Memory Cost of Encoders

The encoders requires different amount of memory for several reasons. The *Golomb-Rice* and *Adaptive Golomb-Rice* encoder does not need much memory for their encoding. The implementation of the *Adaptive Golomb-Rice* encoder uses sections

just like the *Combined* predictor, this leads to a slight increase in the memory requirements. This is still relatively small compared to the *Huffman* encoder which needs to store each symbol together with their codewords. Because the code length varies for the codewords, the code length of the codewords also needs to be stored as a value in the list of codewords.

The *Adaptive Huffman* encoder stores a tree, made out of nodes that contains all of the used symbol values. Based on the values each node needs to store, together with all the pointer variables that points to the leaves, the memory requirements become large. The normal *Huffman* do not need the tree stored as the codes are static and will never changes, and thus the codes can be predefined in a table which is faster to access and also requires less memory.

Both encoders based on a Huffman tree suffers from the fact that the memory requirement is exponentially dependent on the amount of bits used for each symbol. A smaller symbol set of for example, 8 bits, would require a table of 256 entries, which is much less compared to using 12 bits, that needs 4096 table entries. The *Adaptive Huffman* is however an algorithm that has provided, in the worst-case, the best results in the benchmark. If the memory is not a problem for the user, as well as the computational complexity, the *Adaptive Huffman* encoder could be considered for use.

5.4 The Run-length Encoder

From Table 4.6 it can clearly be seen that, for an image with a good prediction as in the case with the image **test1.pgm**, the compression ratio for *Run-Length* is good and even better than *Huffman*. In this test the *Huffman* encoder uses a static tree that is based on the current image's statistics. This is an interesting result as it is significantly faster to build a tree for the *Run-Length* encoder compared to *Huffman*. It takes more than ten times as long time to build the Huffman-tree for the *Huffman* encoder if the *Huffman* encoder would build a tree based on the statistics in the predicted image by fist iterate through it. This time measurement is, however, based on a Python script and should not be generalized. A real hardware implementation might provide different results, but the Python implementation still gives a rough idea of which one is the fastest. This behavior has also already been discussed in the Hardware Requirement section, where the size of the Tree is mentioned and how it depends on the amount of bits the symbols originally need. The *Huffman* encoder encodes the 12 bit symbols, while the *Run-Length* encoder only encodes the last 8 bits using a Huffman tree.

However, if instead trying to encode an image which is hard to predict accurately, *Run-Length* performs worse and even increases the file size instead of compressing it. This is because of the overhead data which is created for the run-length encoding part. If there is not enough long runs of 0:s, there is not much to gain by using the run-length technique. Even when considering how well it performs for some images, it is still not a viable encoder as in the worst-case situation the encoder makes the file size much larger. A compression technique needs to be more stable for different environments in order to be a trustworthy technique that could be used in practice.

5.5 Noise Handling

Comparing Table 4.7 to Table 4.8 it can be seen that the compression ration becomes over 10% better when a simple noise removal technique has been applied to an image. The compression ratio becomes higher because when there are less noise in the image then the prediction methods does perform better. This is because the prediction methods are unable to predict the noise in the pixels since the noise in every pixel is independent and has a random appearance. This means that the random noise value in the pixels are transfered to the prediction error making the prediction error contain a part, which is from the actual noise-free values of the pixels used in the prediction, and a part which is from the noise in pixels used in the prediction. The noise-free part of the prediction error becomes small upon good prediction while the noise part in the prediction error is only dependent on the noise level in the image. This means that if there is much noise in an image, the prediction error gets a wider error probability distribution than if there is less noise in the image. This will then lead to a worse compression ratio when there more noise in the image. When implementing the compression techniques on hardware one should consider to also implement a simple noise reduction technique or use image sensors which have less noise to obtain higher compression ratio.

5.5.1 Near-Lossless Compression

When it comes to the topic of "near-lossless" compression there is no right answer to what is the best way to remove information. Everything comes down to what the user consider as a reasonable amount of noise, or loss of information, that is added to the image for the gained compression ratio. Because of how the sensor-processor setup works, there is no point in actually having a completely lossless compression. The image sensor introduces noise in the pixels and the processor does not know what is noise and what is not. By introducing some quantization noise that is lower than the original noise, the process will be working as before except that the noise in the pixels is slightly increased.

The desired quantization noise should be small compared to the already present noise in order to make it seem like no extra noise was added. The graph in Figure 4.8 shows how the compression ratio is increasing when the allowed level of quantization noise increases. When allowing more quantization noise a rather stable increase of compression ratio be can seen.

Because of how steady the compression ratio increases when allowing more and more quantization noise, the gain is relatively seen, much greater for images that are hard to compress. The image **test24.pgm** has always been a tough image to compress because of its noise-like appearance and when taking a look at Figure 4.9, the advantage of quantization for difficult images is clear. When allowing $SQNR = 2 \approx 3dB$, a compression of around 95% greater compared to no quantization could be achieved. This is a significant increase for all worst-case situations and provides a very good trade-off between quality and compression.

5.6 Worst-case Statistics versus Standard Deviation

When setting up a camera solution, one would want a very stable capture of video data. If there would be a spike in the amount of data being transmitted with time, some data might be lost due to more transmission errors or buffers overflowing. Therefore, it might be a good idea to choose a compression method that performs the best on average, but also does not vary too much by being sensitive to changes.

By calculating the standard deviation for the compression ratio for all image samples, you will be able to get a measurement of how much the compression ratio might vary between different images. The results was however very similar and they all had almost the same standard deviation with just minor differences that could not be taken into any conclusion. The differences were so small that it might just be the uncertainty of the estimation that makes one method get a higher value than another. The standard deviation depends probably more on the image set used for calculating the standard deviation than on the compression techniques. If all of the images are similar then the compression techniques will have similar compression ratio for all of the images and have a low standard deviation. If on the other hand the images might be from a small dataset where each images has a very unique visual structure, the compression techniques will probably vary more and the standard deviation will be larger.

The images used in the benchmark in this thesis, the images in Appendix A, are images taken from our workspace where a limited set of different types of environments could get captured. A set of images of an office with a lot of corridors and flat walls, may provide a very high compression ratio with a very small standard deviation. While a set of images of a construction site containing much visual structures may provide a low compression ratio, but still having a small standard deviation. This might be true for any environment and therefore the standard deviation measurement is too biased for our workspace environment. One idea which will be discussed in the Section 5.14 Further Research is that one could try to get many image sets of different types of environment in an attempt to see how different compression methods' stability gets influenced by different types of settings.

One of the images captured, **test24.pgm**, was intentionally captured to be as tough as possible to compress. The amount of visual structure within the image makes it very hard to predict because of the low pixel-neighbor correlation, and therefore difficult to compress. The worst-case compression results from Table 4.2 shows the compression ratio that, in practice, the camera can be subjected to over a long time if the camera is set up in a harsh environment. Because of how closely related prediction is to correlation, there is, theoretically no way to predict such an image in an efficient way.

The conclusion one can draw from this would be that, no matter what compression method used, one will have to be aware of how the environment will affect the compression performance. In practice, there might not be anything else to do than to be careful not to set up the camera in a situation where the camera monitors an area with much visual structure, or else it might increase the bit-rate. The images that were used for the benchmark consisted of many different types of still images and almost every one of the images provided a 30% compression ratio or

higher, thus providing average values as high as shown in Table 4.1. Most of the images got large areas that are monocoloured of a wall or something similar, these areas can be compressed efficiently and in practice a camera will never monitor an area without any monocoloured part such as shown in **test24.pgm**.

As discussed earlier in this chapter, there are more methods that can be applied in order to increase the compression ratio. For example, a near-lossless quantization method could be used in order to improve a bad compression ratio. The same thing could be done by using a predictor that uses temporal data. However, both of these methods have a very high price to pay in order to be used. The price of quantization is reduction in the image quality and the price of using temporal data is that it requires much more memory.

If the compression ratio becomes lower, the risk of the bitrate becoming to large increases, which might lead to frames being dropped. Therefore the price to pay for quantization might not be a bad option. The problem turns into finding which one of the two options, either using quantization or not using it, that is the best. If quantization would be used, the question is how much quantization should be allowed to maintain this advantage. The level of quantization could probably be automatically tuned by a program, or tuned in some way when the camera is set up, in order to obtain a desired compression ratio.

5.7 Prediction Technique Evaluation

Most of the predictors are pretty similar as they all utilize the correlation between the pixels close to the pixel to predict. The predictions are made by combining the pixel's neighbors' values to predict in various ways. The techniques vary a lot in what types of mathematical operations that are performed in order to predict, which would affect the computational cost among other things.

5.7.1 LMS Predictors and Hardware Demands

There is a vague relationship between computational cost for prediction and prediction performance. The *LMS2* and *Combined* predictors are the two most computationally costly, but also the ones that produced the lowest 'average-Entropy' or 'Worst-case-Entropy' in the benchmark. The *LMS* predictor is also a interesting method because it had almost as good performance as the *LMS2* predictor while at the same time has lower computational cost than the *Combined* predictor. The gain in compression ratio that the *LMS2* predictor has compared to *LMS* is of a cost of increasing the computational cost of over 50%. This can be compared to the compression ratio increase between the *Simple* and the *Mean2* predictor where 30% increase of hardware introduces over 10% increase of compression ratio accordingly to the 'worst-case' Table 4.2. It can also be seen in the tables that the compression ratio differences between a simple method as the *Mean2* and a more computational costly prediction method as *LMS* are very low compared to the increase of computational cost. This shows that simple prediction methods are easy to improve and only require small computational cost increases till one point where it requires large computational cost increases to get small compression ratio gain.

One thing worth mentioning is that there is a risk in using the predictors based on *LMS* adaption. The risk is that if the step size in the *LMS* algorithms is not small enough, the prediction weights might diverge and the compression technique will crash. This could be avoided if the step size would be normalized based on the input to the *LMS* prediction filter. Normalization on the other hand requires lots of computations and is therefore not advisable.

5.7.2 Averaging Predictors

It might be a bit surprising that prediction methods based on averages performed better predictions if they used a low number of neighbors for prediction such as *Mean2*, *Mean2L* and *Mean2L-2* than if more neighbors were used as in the predictors *Mean4* and *Mean4-R*. It is probably due to, as it can be seen in Figure 4.1b and Figure 4.2b, how the correlation between a pixel and its neighbors decreases rapidly with growing distance between the pixels. If the predictor then weighs all the pixels equally independent of their distance to the pixel they predict, the error will become higher than if the predictor would just use less neighboring pixels for the prediction. A solution like LMS or similar is required to actually use some form of weighing of the values to handle this problem. Even the *simple* predictor preformed well in average, but the very important measurement, worst-case compression ratio, gave very bad results.

5.7.3 Combined Predictor

The *Combined* predictor gave interesting results that are shown in Section 4.2.1. The figures 4.6a and 4.6b show how the number of sections affects the compression ratio. These results are only from two different images, but it shows how the compression ratio might get an increase for some images, but a decrease for some images when using the same number of sequences.

The benchmark results shown in Table 4.1 and Table 4.2 shows how the *Combined* predictor performed in general and with the image **test24.pgm**. The later being true because the worst-case was the case when compressing the **test24.pgm** image. As a reminder for the reader, these benchmarks scripts used 128 sections per row in the *Combined* predictor. It is clear that *Combined* performed on average better than the other prediction methods, while the compression ratio decreased for worst-case image. Despite this, one may conclude that the compression ratio usually depends on the number of sections in a way that Figure 4.6a illustrates.

An exception to this would be where the images visual structure makes it difficult for the combined predictor to estimate what method to use. This would be true in the case when compressing **test24.pgm**, because the surrounding sections would not provide reliable statistics for the method estimation. For this reason, when the number of sections reaches a point where it is basically just 1 or 2 pixels per section, the compression becomes better. This can be seen in Figure 4.6b where the graph rises when the number of sections becomes high enough. By taking a look back at Figure 4.2b it can be seen how this pattern matches with the new observation. The correlation of nearby pixels fades off too quickly as you move further away from the pixel in question.

The Figures 4.7a and 4.7b shows how much the different methods were used during the prediction. One clear observation is that the selection of method is depending on the image and its statistics. For example, *Mean4* is the most common used in **test1.pgm** but is the second least used in image **test24.pgm**. The reason for this could be that the *Mean4* predictor uses pixel data that is too far away from the pixel that is being predicted to produce a good prediction in the **test24.pgm** image.

5.7.4 Prediction using Temporal Data

The compression ratio when using the prediction method called *Time* that uses data from the previous frame is shown in Table 4.5. It provides a higher compression ratio than the two prediction methods *LMS* and *Mean2L*. Considering that *Time* uses temporal data that could be highly correlated when predicting, one might expect an even higher compression ratio. This is however not the case with this predictor and the compression ratio only increases around 1 to 2 %. There are several factors that might affect the results.

One factor might be due to that the resolution being used is very high. A small movement between two consecutive frames, for example, when something moves in the screen, or the camera moves, will cause the temporal data to become less correlated. This is because a small movement in the image will make the pixels to take on values from the new environment in the new frame, which have a low correlation to the values in the old frame. Therefore, a predictor that uses the surrounding pixels of the same frame might predict just as well or even better due to that the correlation between neighboring pixels in a image are normally high. This is the same reasoning as described in the Section 2.1.1. The high resolution used in the image sensor makes the predictor that uses temporal data to perform worse, while a predictor using spatial data performs better because with higher resolution the pixels will be closer to each other and therefore the correlation will be higher.

However, the *Time* predictor is implemented like the *Combined* predictor and uses both temporal data and spatial data. The *Time* predictor chooses the appropriate prediction method depending on the current section in the image and is therefore able to achieve high compression ratio. This means that the sections that have not changed between frames have a higher chance to be compressed better using temporal data and the sections which has been changed between frames has a higher chance to be predicted by a prediction method using spatial data.

The reasoning above might lead one to think that the *Time* predictor is a great option because it almost strictly improves the compression ratio. However, because of the very high resolution that is being used, the memory requirements for having a whole frame stored would be high. For an image with a 1080p resolution, using two bytes per pixel would require $1920 \times 1080 \times 2 \approx 4MB$ of memory! This might be a very high requirement for a camera sensor that is supposed to have very light hardware and can not be recommended, considering how small the compression ratio improvements are.

The prediction method that uses temporal data in this thesis was implemented to use a simple technique to predict the next value. The hardware limitations

restricted the predictor to use more advanced techniques. Therefore, one must consider that there might be much more efficient methods to predict using previous frames. More advanced techniques could be, for example, to find movement patterns between frames. Another way could be to just use more complex prediction methods using both spatial and temporal data. This is something that could be researched further.

5.8 Horizontal and Vertical Blanking

As seen from the results in Section 4.5 one can see that the added padding has a huge impact on the compression ratio for the selected predictor-encoder combination. Even though there is almost no extra information added to the image. This is because how the encoder work, and that it treats every pixel the same, regardless if it is a part of the actual image or not. The predictor might not work as good in the edges of the image as before and it might be influenced by the surrounding zeros, and the extra sign-bits still needs to be transmitted.

It is easy to say that another method would be preferred to tackle this problem. However, because of how the amount of blanking might vary, it is hard to just crop out the image and just send the actual image data. Some method of finding where the image starts and ends would be the best case, but this might require extra processing power and memory.

Another method could be that you would modify the encoder with some special marker codewords that could represent "end of the line" and "end of frame". This would of course be of the cost of taking those codewords position and any pixel that would've needed that codeword would have to be remapped. In the case of having 12-bit data, trading one codeword for one of these markers would only make one 1 of 4095 codewords to be lost, which could be a great increase of compression for a very small loss of image quality.

5.9 Lossy JPEG

Comparing the results from lossless or even near lossless compression with lossy compression it can be seen that *lossy JPEG* provides much higher compression ratio. Even if the quality setting in *lossy JPEG* was set to $quality = 100\%$ the compression ratio was much larger and more stable than in any lossless technique. This is because the *lossy JPEG* is allowed to change any pixel value so that the algorithm is able to compress the images more efficient. The values of the pixels can sometimes be changed a lot before any visual quality degradation that the human eye is able to detect occurs. From Table 4.10 it can be seen that the mean squared error of the change in pixels is about the same for both the *lossy JPEG* using $quality = 100\%$ and the *lossy JPEG* using $quality = 10\%$. In the Figures 4.11 and 4.12, it can be seen that the quality of the two images is quite different. For this reason, the mean squared error of the change in each pixel value would be a bad measurement for determining if the quality has been reduced.

The loss from *lossy JPEG* can not be compared in a good manner with the results obtained in the Near-Lossless Quantization Section 4.4.2. This is due to

the fact that there is no control of where or how the loss in *lossy JPEG* occurs. The Near-Lossless Quantization loss only occurs in noisy pixels where the real pixel value is uncertain. The loss is a percentage of the noise already present in the pixels and if there is much noise or uncertainty in a pixel the algorithms may introduce some loss, but if there is no noise in a pixel the algorithm then there should be no loss. The *lossy JPEG* does not depend on the noise level in the pixels but rather introduces loss in a way which suits the algorithm. This means that the compression ratio and loss from *lossy JPEG* can not be compared equally with the compression ratio and loss from the Near-Lossless Quantization.

5.10 Comparison with PNG

The results for the *PNG* compression technique provided was obtained by only compressing one image. It was decided to not investigate the *PNG* technique further because the compression ratio obtained from this test was so poor. The *PNG* technique does not manage to obtain a compression ratio that is near the ratio that is obtained using the benchmark program. There may be some details that might have been disregarded in this test. A proper test by implementing *PNG* and including it in the benchmark might be needed to exclude it entirely as a possible candidate. However, because *PNG* is using a *Lempel-Ziv* encoding algorithm, that requires extra computations and needs to keep track of dictionaries in the memory, it was decided not to research *PNG*'s potential further in this thesis.

The reason that the compression was compared using a Bayer separated image was that it was assumed that the *PNG* algorithm was not implemented for compressing images where the pixels are in a Bayer pattern. This assumption was done because when trying to compress images without using Bayer separation, barely any compression were achieved. If the *PNG* would have been implemented during this thesis some small adjustments could been done in the algorithm. These adjustments could probably make the compression technique perform better by making the algorithm have special cases in mind and by adjusting the algorithms to the structure in the raw image which were used during this thesis.

5.11 Arithmetic Coding

The *arithmetic coder* was not implemented since according to Theory, it may reduce the gap between the entropy and the mean codeword length generated by the *Huffman* encoder. The downside would be that it is more computationally costly. This is mostly important when dealing with symbols in a small alphabet. When compressing images where each pixel is represented by 12 bits, the alphabet is rather large and the advantages of *arithmetic* encoding over *Huffman* encoding decreases. The *Huffman* encoder, or rather *Adaptive Huffman* is so close to the entropy that *arithmetic* coding is redundant. By taking a look at the maximum compression possible in Table 4.3 it can be seen that the *LMS2* has 14.90%, which is based on the worst-case entropy. From Table 4.2 which shows the compression ratio for the worst-case image, *LMS2* has 14.64% already when combined with *Adaptive Huffman*. If the encoder would be even more optimized with for example,

a starting tree and using a forgetting method to find local variations, the encoder would perform even better. Considering how close it already is to the entropy it was decided not to investigat *arithmetic* encoding in detail. Improvements has to be made in the predictor in order to obtain a significant increase in compression ratio.

Another problem with *Arithmetic coding* is the same as with *Huffman*, it needs predetermined statistics in order to achieve efficient encoding. There is an adaptive solution to *Arithmetic coding*, but it is more computationally costly than what could be considered for the application of this thesis. The *arithmetic* encoder would have to recalculate its symbol probability intervals at regular time intervals. These calculations would require divisions and multiplications to be performed. As multiplications and especially divisions are heavily computationally costly, the arithmetic encoder methods are not of any interest of this thesis.

5.12 Transforms

Initially in this thesis, the idea was that there might be some useful lossless transform technique for image compression. During the literature study, the *Burrows-Wheeler transform* (BWT) was the only lossless transform technique to be found. Even though BWT is mainly used for text compression it is supposed to be useful for image compression with a low amount of available symbol values as well. Later on it was decided that the BWT should not be implemented since it was assumed not to be an efficient compression method for such a large symbol alphabet represented with 12 bits. The BWT was also assumed to be too hardware demanding for the scope of this thesis. For example, when BTW compress a sequence of N pixels, it creates a $N \times N$ matrix that needs to be sorted. For BWT to be able to achieve a good compression, N has to be very large, larger than the number of symbols values available. Say $N = 4096$, which is a ratio of $1 : 1$ between the sequence length and the number of symbol values (this ratio is in practice too low), then a matrix of size 4096×4096 would be needed to be created and that requires more memory than what would be available. This means that there is no interest in implementing this technique.

5.13 Effects of Error Handling

When dealing with transmission of encoded data it is important to transmit the data in a safe way otherwise the transmission may cause an error, if it occurs, it might propagate as shown in Figure 4.14. After an error has occurred the encoder and decoder has to be reset in some way in order to become synchronized again. This can be done in various ways with varying complexity.

5.13.1 Dividing Image into Blocks

Taking care of the potential hazards when using variable length compressed data is necessary and a solution to this was proposed in Section 3.11. This method was only made to show how the hazards could be taken care of. More effort could

be put into finding an optimal solution for this purpose with increased efficiency. However, the solution we proposed is still good enough to be a viable solution for test purposes and to show that it is possible to do transmission of variable length compressed data.

The results from the benchmark after splitting the image into sections, as depicted in Table 4.12, show how little impact this has to the compression ratio. If a bit error would occur in a frame, it would only affect the predictions in a 16:th part of the image. This means that the visual quality have been significant increase at the cost of almost no compression ratio loss. This is a technique that should be considered to secure that the image will not be destroyed due to bit errors.

5.13.2 Transmission Protocol

The suggested solution for the error handling was a packet based transmission protocol that does provide a solution to make sure the encoder and decoder always stay synchronized. It does not provide any protection against actual pixel errors in the transmission. The only purpose of the protocol was to detect when an error occurs but also reliably provide the number of symbols was originally sent to keep the encoder and decoder in sync.

When adding all the overhead data that was created for this method, it did only add up to about 1.5% extra data. The packet size is what could be tweaked in order to find a solution which provides the same functionality, but with less overhead data. It is a trade-off between having a bigger packet, which means that less part of it is header data, and having excess space for when an image or section is completed. Simulations could be done in order to test this and there is most likely a better solution.

5.13.3 Error Correction Potential

There are many encoding techniques, like the LDPC codes, that could be implemented in order to minimize the risk of a bit error occurring. This could potentially be just as stable, or even more stable than our proposed method as described in Section 3.11.1. It will also correct the values that becomes wrong in the transmission instead of just detecting them. However, there is a cost to these error correcting codes, which is that they are much more computationally complex than simple repetition codes and they will most likely cause more overhead data than the proposed packet protocol. The later statement might not be true, as there will be no need to divide the image in more sections to prevent error propagation, because the errors will be corrected in this case. A simulation would have to be created in order to test this and show how much the overhead data will differ.

It is mostly up to the one implementing to decide whether error correction coding is worth to be implemented or not. If the risk of a packet being partly destroyed is not an issue, the extra cost of implementing error correction might not be worth it.

5.14 Further work

The prediction rule of how to predict which predictor to use in the suggested *Combine* prediction method could be further investigated to find a prediction rule which optimally finds which prediction rule to use for predicting the pixel values. Similarly, the parameter estimation in *Adaptive Golomb-Rice* could be investigated to find an optimal way to do the parameter estimation.

Before applying near-lossless compression in a real product, the effects from it should be evaluated. It is important to study how the quantization affects the image processing in the back-end processor. How the quantization from the lossy *JPEG* affect the image quality after the image processing in the back-end processor should also be investigated. If these effects are small then lossy *JPEG* could be an interesting choice when it comes to near-lossless compression if it can be implemented in a way that makes it run efficiently on hardware.

The usage of error correcting codes was rather overlooked in this paper, but there are many variants of error correcting codes that can be tested and evaluated in a similar manner as how compression techniques were evaluated in this paper. One might consider computational complexity and added overhead data as factors when evaluating this. A good error correcting code might replace any need of a protocol like the one proposed in this paper, and provide even better results.

If the transmission protocol that was proposed, or a similar one, was used, it would be interesting to see how much the image quality worsens as transmission errors occurs. In this paper this was not investigated and it would be valuable with some form of measure on this error to be able to evaluate different methods.

Chapter 6

Conclusions

There has been many things that had to be considered in this thesis. As seen from the results, there are many techniques that has an advantage, but also a disadvantage. The work performed in this thesis has brought up these advantages and disadvantages in order to provide the best trade-off for a specific situation.

The benchmark program has provided results that has pointed out a few techniques that are good candidates for the scenario presented in the background section of this report. Most of the techniques, except for *LMS* and *Combined*, has been lightweight enough to make a direct comparison of which technique is the best. The *Mean2L* predictor which was proposed in this paper, has given impressive results that does compete with the more complex prediction techniques. Because of how strong the correlation is between neighboring pixels in a high resolution image, the simple methods that only use nearby pixels' information is good enough to make a very good compression. Because of the hardware restrictions that had to be considered in this thesis, one can conclude that a method like *Mean2L* is to be preferred for its simplicity and high performance.

The encoders that provided high compression ratio and at the same time was stable were the adaptive techniques. The efficiency difference between *Adaptive Golomb-Rice* and *Adaptive Huffman* were small, but because the computational complexity of the *Adaptive Golomb-Rice* was significantly lower one can conclude that *Adaptive Golomb-Rice* is to be preferred.

Near-lossless compression has given very high compression ratios with a stable improvement of compression. Visual quality versus compression ratio, and what one might define "near-lossless" as, are difficult topics and the results in this paper only shows the efficiency improvement, but does not compare the visual differences. The visual differences were so small that it was decided not to be discussed further in this paper. The noise has a significant role when it comes to compression and instead one might just focus on better hardware instead of near-lossless compression.

How transmission errors affect the visual quality or the functionality of the transmission, has been seen to have an important role with compressed data. To make good use of any of the results in this paper, one must create a reliable transmission protocol to ensure the synchronization between the encoder and the decoder. The proposed method in this paper shows a way of how this could be done, but it could be improved or even be replaced by a error correction coding technique.

References

[1] "Data Link Control". From CSE IIT, Kharagpur `http://nptel.ac.in/courses/106105080/pdf/M3L2.pdf`, Retrieved 2018-03-22

[2] Don Adjeroh and Kalyan V. Bhupathiraju, On Lossless Image Compression using the Burrows-Wheeler Transform, IEEE International Conference on Image Processing, 2011.

[3] Gallagher, R.G. (1978) Variations on a Theme by Huffman. *IEEE Transactions on Information Theory*, 24(6), pp. 668-674.

[4] Golomb W. (1966) Run-Length Encodings, IEEE Transactions on Information Theory, 12(3), pp. 399-401.

[5] Greg R. (1999) *PNG The Definitive Guide*, Sebastopol United States, O'Reily Media

[6] Hakkennes E.A. Vassiliadis and S. "Hardwired Paeth codec for portable network graphics (PNG)," Proceedings 25th EUROMICRO Conference. Informatics: Theory and Practice for the New Millennium, Milan, 1999, pp. 318-325 vol.2.

[7] Haykin S (2014) Adaptive Filter Theory, Fifth Edition. Pearson.

[8] Ho Derek, et al. (2013) CMOS Tunable-Color Image Sensor With Dual-ADC Shot-Noise-Aware Dynamic Range Extension, *IEEE Transactions on circuits and systems-I: Regular papers.*

[9] Hudson, G.P. (1983) The development of photographic videotex in the UK. *Proceedings of the IEEE Global Telecommunications Conference,IEEE Communication Society*, pp. 319-322

[10] International Organization for Standardization, "ISO/IEC 15948:2004 – Information technology – Computer graphics and image processing – Portable Network Graphics (PNG): Functional specification". `https://www.iso.org/standard/29581.html`, Retrieved 2018-02-19.

[11] IT Hare, "No Bugs" Hare. (2016) *Infographics: Operation Costs in CPU Clock Cycles.* `http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/`, Retrieved 2018-02-22

[12] Kavanagh R.C, Murphy J.M.D, (1998) The effects of quantization noise and sensor nonideality on digital differentiator-based rate measurement, *IEEE Transactions on Information Theory.* 47(6), pp. 1457-1463.

[13] Kiely, A. (2004) Selecting the Golomb Parameter in Rice Coding, *IPN Progress Report*, pp. 42-159.

[14] Koh, C.C. and Mitra S.K. (2003) Compression of Bayer color filter array data, Department of Electrical and Computer Engineering, University of California,

[15] Kou, W. (1995). *Digital Image Compression: Algorithms and Standards.* AH Dordrecht. Kluwer Academic Publisher Group.

[16] Lacan J., Roca V.,Peltotalo J. and Peltotalo S. (2009) Reed-solomon forward error correction (FEC) schemes, RFC 5510. Institute of Electrical and Electronics Engineers.

[17] Leon, W.D., Balkir, S., Sayood, K. and Hoffman, M.W. (2005) An Analog-to-Digital Converter with Golomb-Rice Output Codes, University of Nebraska-Lincoln, *209N Walter-Scott Engineering Center.*

[18] MIPI Alliance, (2012), MIPI® Alliance Specification for Camera Serial Interface 2 (CSI-2).

[19] Netravali, A.N. (1980) Picture Coding: A Review, *Proceedings of the IEEE*, 68(3), pp. 366-406

[20] Sayood, K. (2000). *Introduction to Data Compression: Second Edition.* San Francisco, CA. Morgan Kaufmann Publishers.

[21] Sayood, K. (2003). *Lossless Compression Handbook.* Academic Press. Lincoln, Nebraska.

[22] Shen H., Pan W. D., Wu D. and Lubna M., "Fast Golomb coding parameter estimation using partial data and its application in hyperspectral image compression," SoutheastCon 2016, Norfolk, VA, 2016, pp. 1-7.

[23] Stefan Höst. (2017) Information Theory and Communication Engineering, KFS AB, p. 41.

[24] Syahrul E., Dubois J., Vajnovszki V., Saidani T. and Atri M., "Lossless Image Compression Using Burrows Wheeler Transform (Methods and Techniques)," 2008 IEEE International Conference on Signal Image Technology and Internet Based Systems, Bali, 2008, pp. 338-343.

[25] Sörnmo L. and Laguna P. (2005), *Bioelectrical signal processing in Cardiac and and Neurological Applications*, Elsevier Academic Press.

[26] Tinku A. and Ray K. A. (2005) *Image Processing: Principles and Applications.* Hoboken, New Jersey: John Wiley & Sons, Inc.

[27] Venkataramanan R. "1B Paper 6: Communications" From University of Cambridge Signal Processing and Communications Laboratory. `https://www.sigproc.eng.cam.ac.uk/foswiki/pub/Main/IBComms/P6_handout6.pdf`, Retrieved 2018-03-22

[28] Wallace, G.K. (1992) The JPEG still picture compression standard. IEEE Transactions on Consumer Electronics, 38(1), pp. xviii-xxxiv.

[29] Weinberger, M. J., Seroussi, G. and Sapiro, G. (1996) LOCO-I: A low complexity, context-based, lossless image compression algorithm. In: *Data Compression Conference, 1996. DCC '96. Proceedings*, Snowbird, UT, pp. 140–149.

[30] Weisstein, Eric W. "Covariance." From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/Covariance.html`, Retrieved 2018-03-16

[31] Weisstein, Eric W. "Correlation." From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/Correlation.html`, Retrieved 2018-03-16

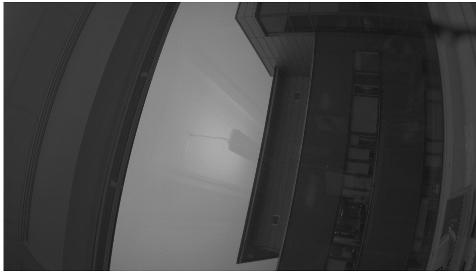
[32] Weinberger, M. J., Seroussi, G. and Sapiro, G. (2000) LOCO-I Lossless Image Compression Algorithm: Principles and Standardization into JPEG-LS, *IEEE Transactions on Image Processing*, 9(8), pp. 1309-1324.

[33] Witten I.H., Neal R.M. and Cleary J.G. (1987). Arithmetic Coding for data compression, *Communications of the ACM*, 30(6) pp. 520-540

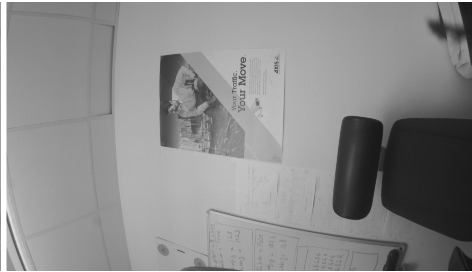
[34] "What is the Camera Parallel Interface?" From FTDI Chip. `http://www.ftdichip.com/Support/Documents/TechnicalNotes/TN_158_What_Is_The_Camera_Parallel_Interface.pdf`, Retrieved 2018-04-13

Appendix A

Images Used for Testing Spatial Compression Techniques.



(a) test1.pgm

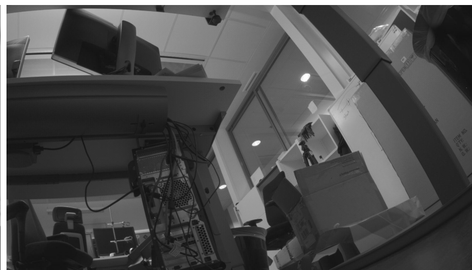


(b) test2.pgm

(c) test3.pgm

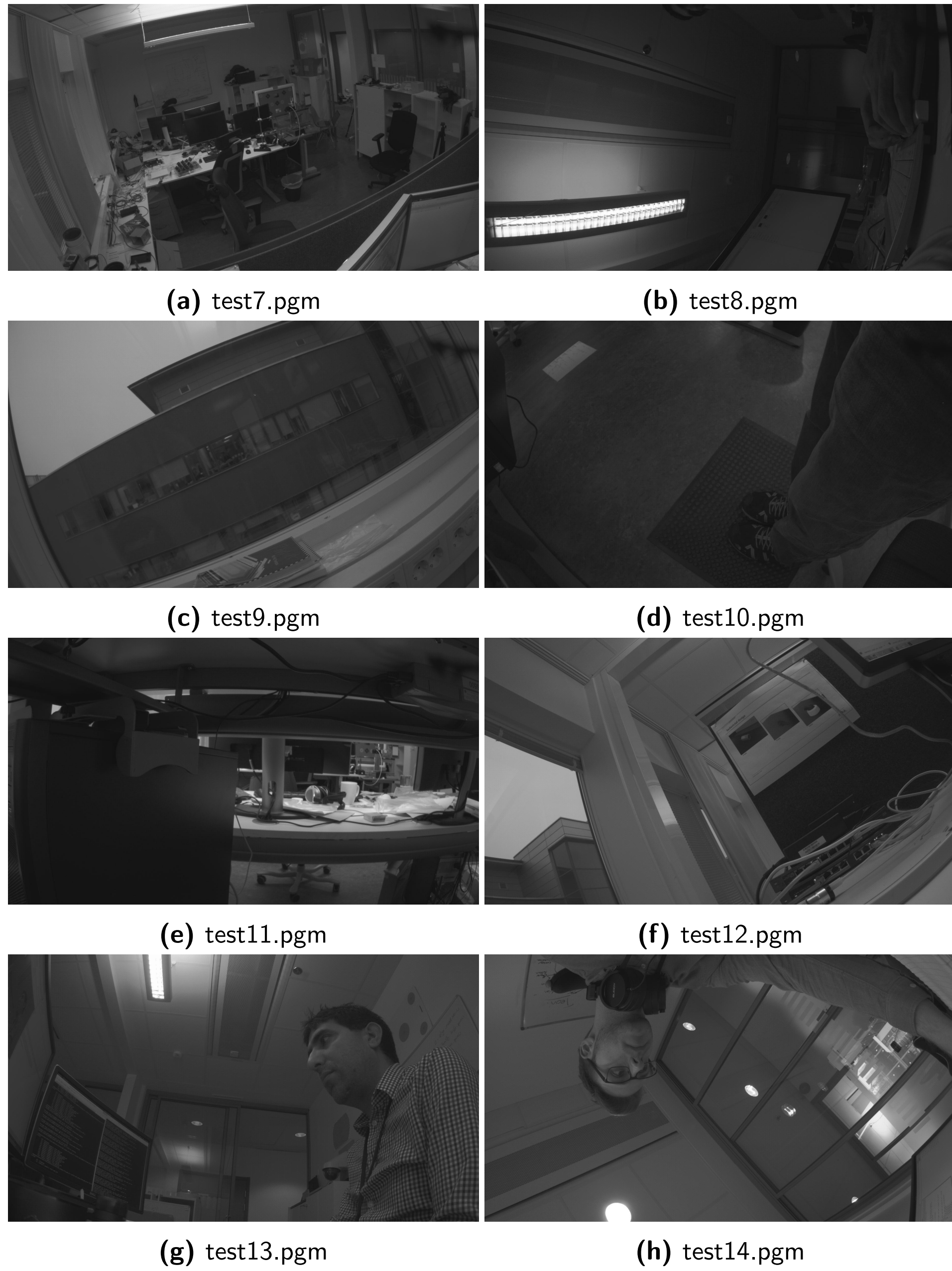
(d) test4.pgm

(e) test5.pgm



(f) test6.pgm

Figure A.1: First set of images used for the benchmark application.



(a) test7.pgm

(b) test8.pgm

(c) test9.pgm

(d) test10.pgm

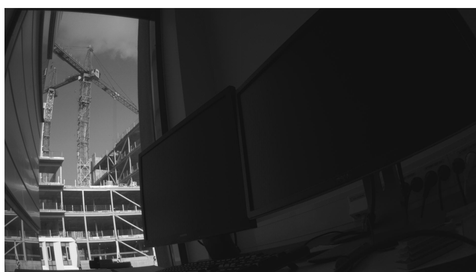
(e) test11.pgm

(f) test12.pgm

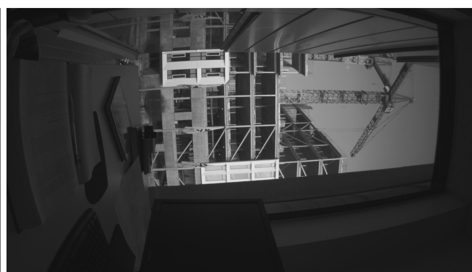
(g) test13.pgm

(h) test14.pgm

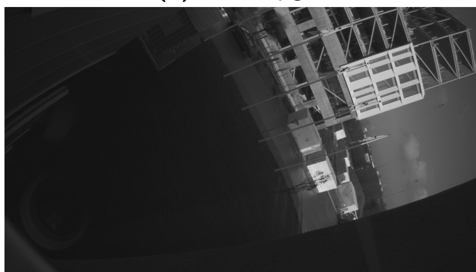
Figure A.2: Second set of images used for the benchmark application.



(a) test15.pgm

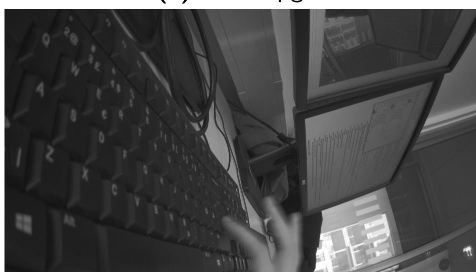


(b) test16.pgm



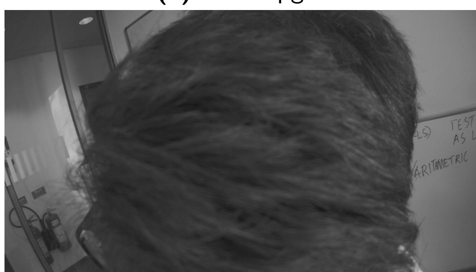
(c) test17.pgm

(d) test18.pgm



(e) test19.pgm

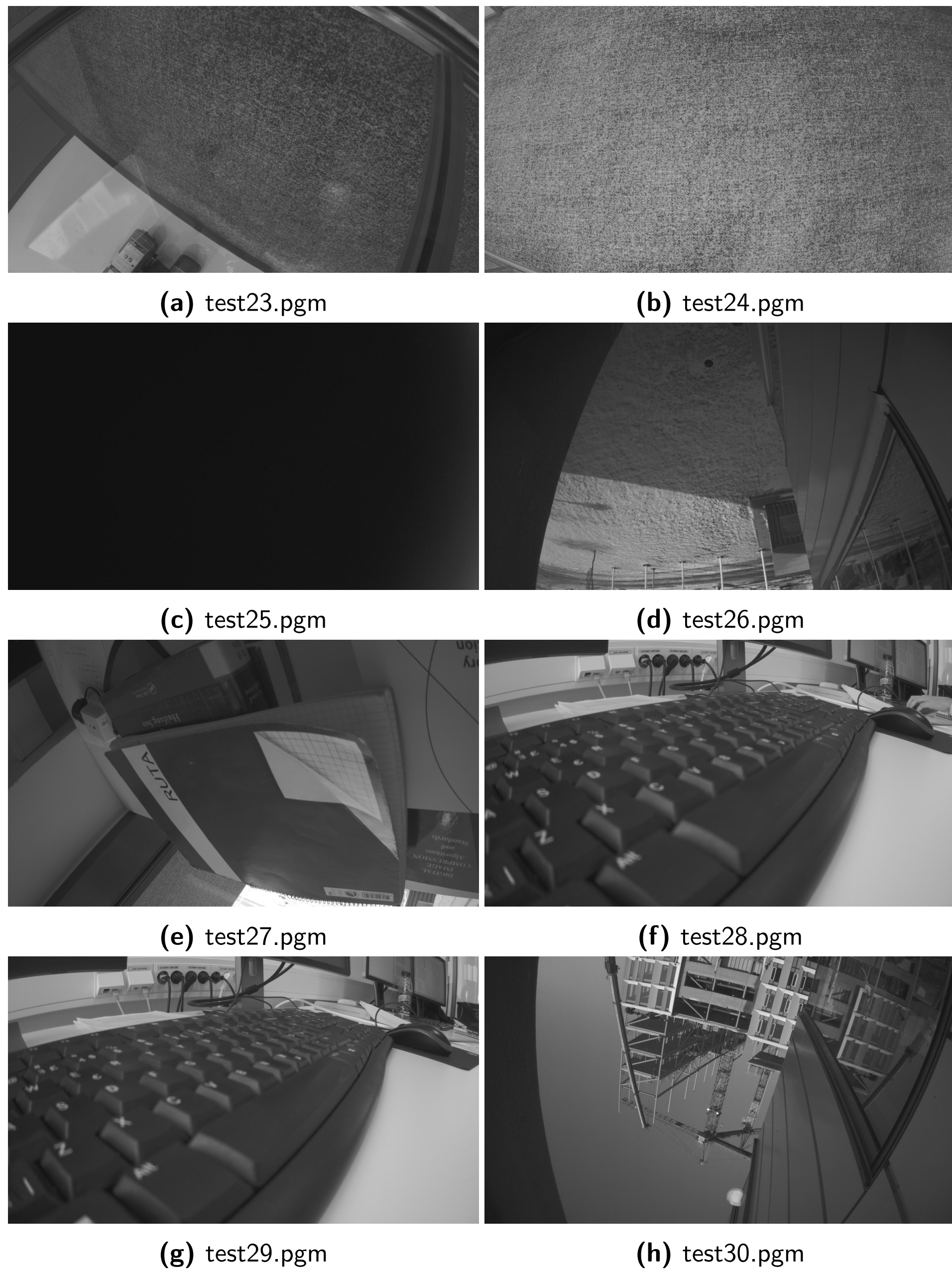
(f) test20.pgm



(g) test21.pgm

(h) test22.pgm

Figure A.3: Third set of images used for the benchmark application.



(a) test23.pgm

(b) test24.pgm

(c) test25.pgm

(d) test26.pgm

(e) test27.pgm

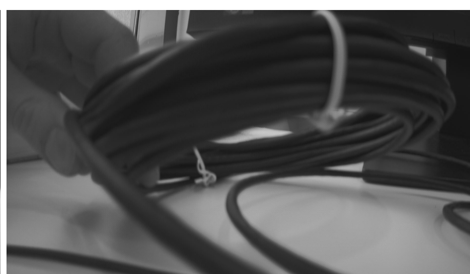
(f) test28.pgm

(g) test29.pgm

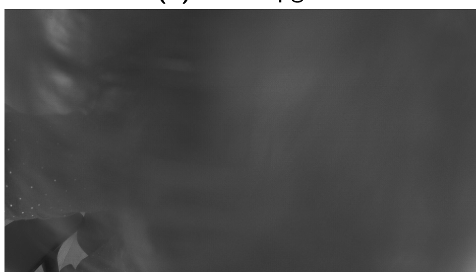
(h) test30.pgm

Figure A.4: Fourth set of images used for the benchmark application.

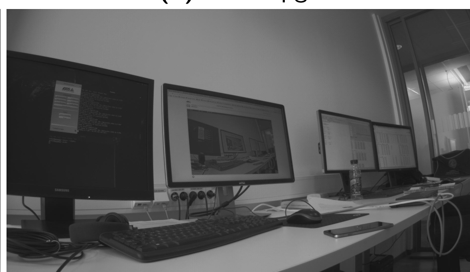
(a) test31.pgm



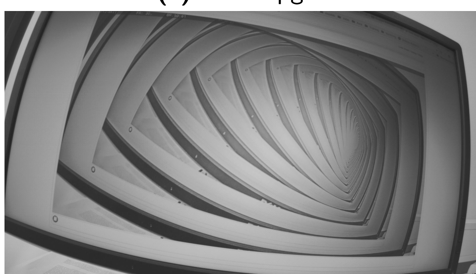
(b) test32.pgm



(c) test33.pgm

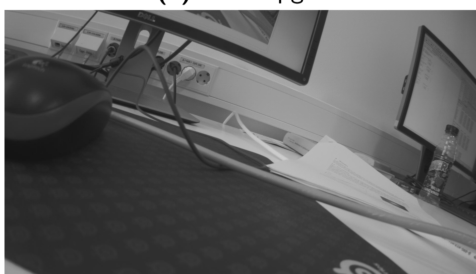


(d) test34.pgm

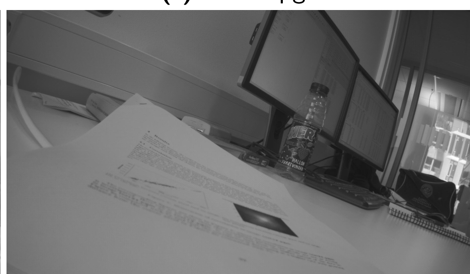


(e) test35.pgm

(f) test36.pgm



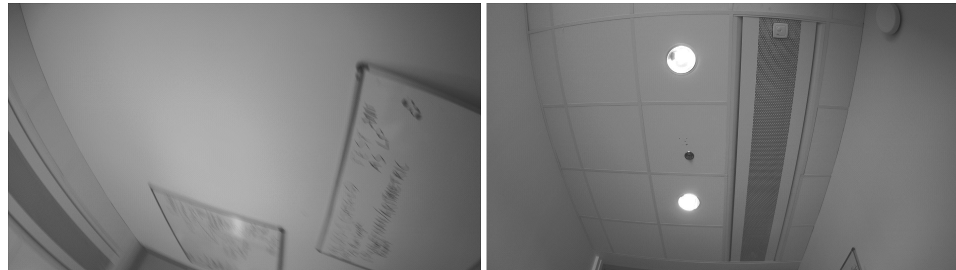
(g) test37.pgm



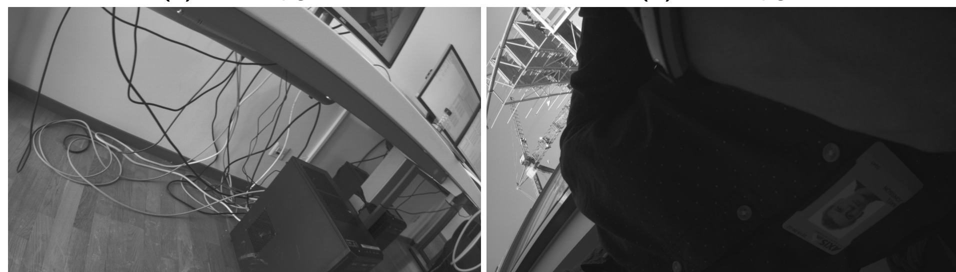
(h) test38.pgm

Figure A.5: Fifth set of images used for the benchmark application.

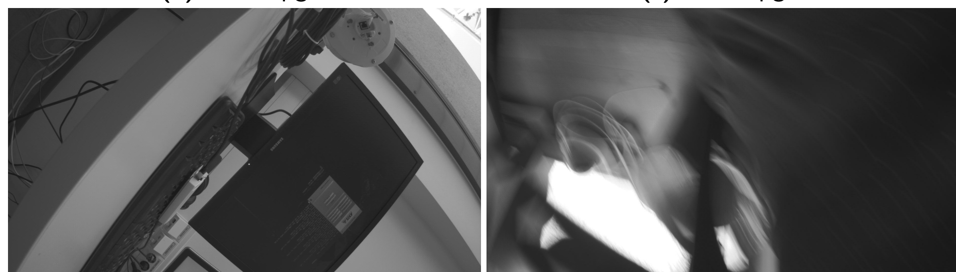
(a) test39.pgm **(b)** test40.pgm



(c) test41.pgm **(d)** test42.pgm

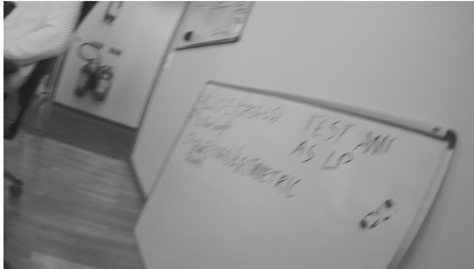


(e) test43.pgm **(f)** test44.pgm



(g) test45.pgm **(h)** test46.pgm

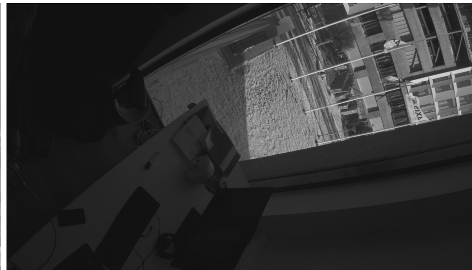
Figure A.6: Sixth set of images used for the benchmark application.



(a) test47.pgm

(b) test48.pgm

(c) test49.pgm

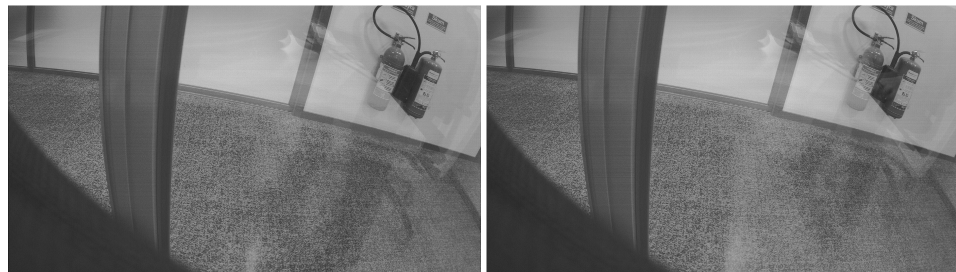


(d) test50.pgm

Figure A.7: Seventh set of images used for the benchmark application.

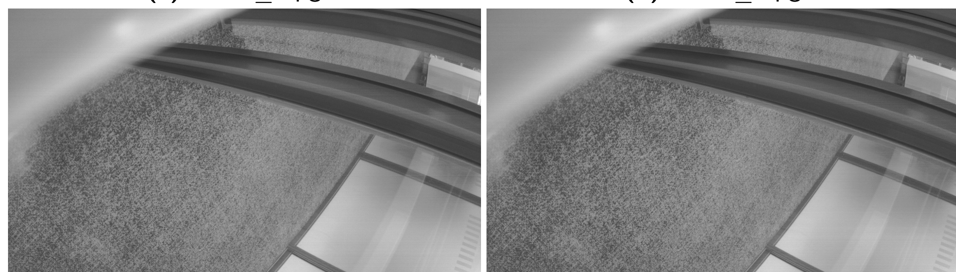
Appendix B

Images Used for Testing Temporal Compression Techniques.



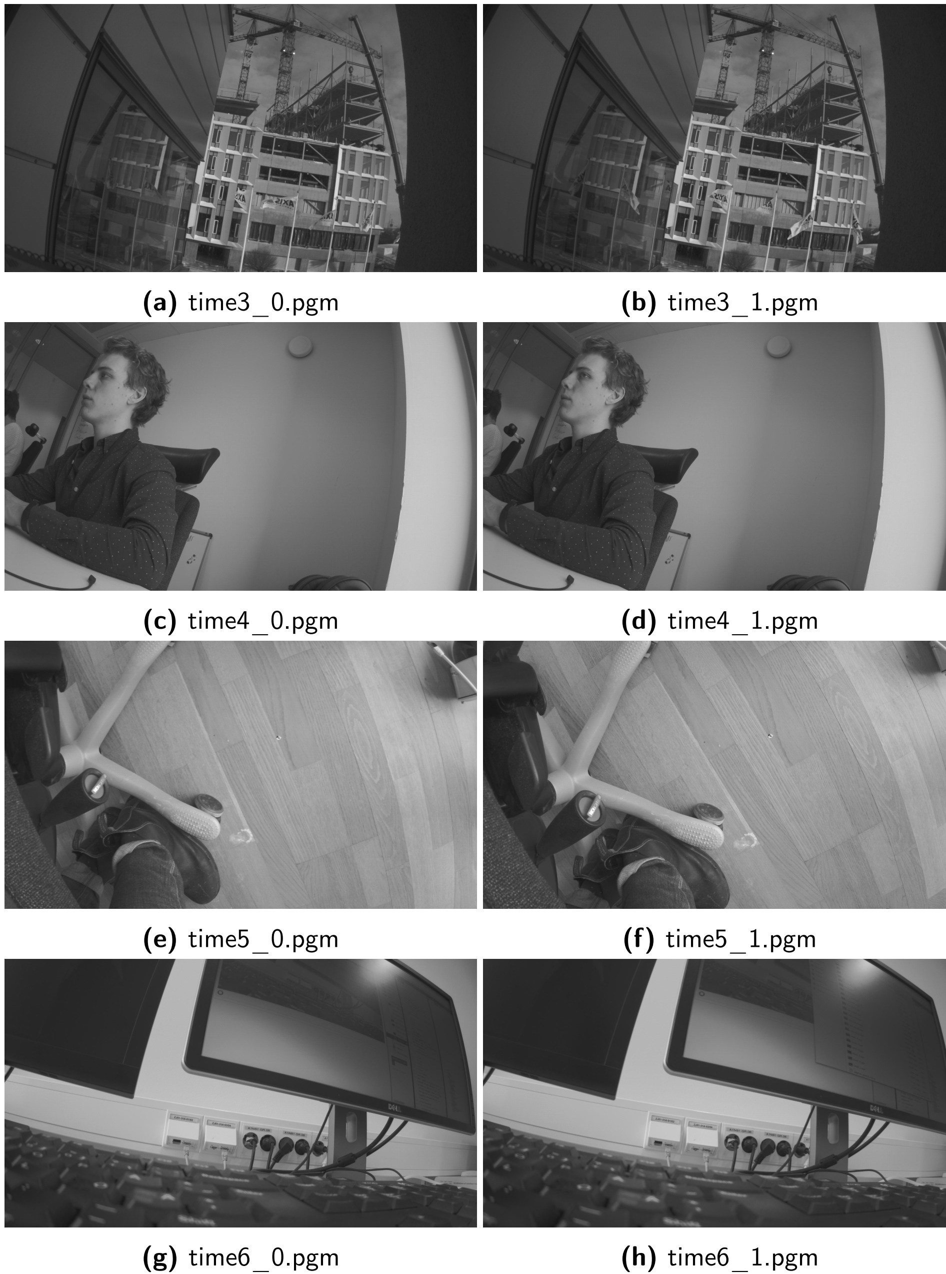
(a) time0_0.pgm **(b)** time0_1.pgm

(c) time1_0.pgm **(d)** time1_1.pgm



(e) time2_0.pgm **(f)** time2_1.pgm

Figure B.1: First set of temporal images used for the benchmark application.



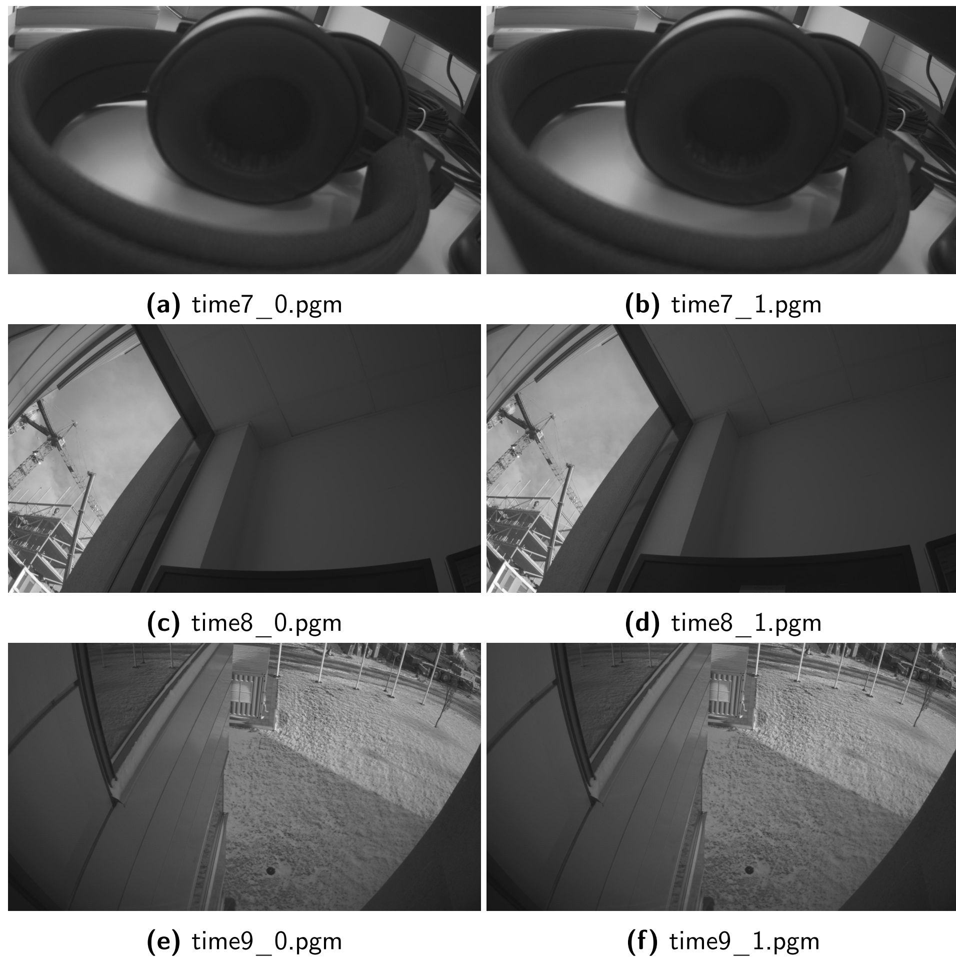
(a) time3_0.pgm **(b)** time3_1.pgm

(c) time4_0.pgm **(d)** time4_1.pgm

(e) time5_0.pgm **(f)** time5_1.pgm

(g) time6_0.pgm **(h)** time6_1.pgm

Figure B.2: Second set of temporal images used for the benchmark application.



(a) time7_0.pgm (b) time7_1.pgm

(c) time8_0.pgm (d) time8_1.pgm

(e) time9_0.pgm (f) time9_1.pgm

Figure B.3: Third set of temporal images used for the benchmark application.

Printed by Tryckeriet i E-huset, Lund 2018

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2018-633

http://www.eit.lth.se