THE UNIVERSITY OF CALGARY

Adaptive Sampling and Interpolation Methods for Digital Image and Video Coding

by

Chad W. J. Dreveny

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

CALGARY, ALBERTA

September, 1999

Abstract

When using digital images or video, the amount of storage space or transmission bandwidth required can be quite large when the media is in its raw form. Recently, there has been a dramatic increase in the usage of these digital media types. Consequently, there has also been an increase in the research devoted to reduce the data required to represent these types of digital signals. In this thesis, a study is presented of a method that uses adaptive sampling and interpolation for image and video data compression. A recursive splitting method that creates an adaptive sampling grid, is described, along with a discussion concerning the interpolation of these samples for the reconstruction of the original image or video. Implementation and optimization issues concerning the presented image and video data compression algorithms are discussed. Examples showing the effects of these methods are given and compared to existing standard data compression techniques.

Acknowledgements

I would first like to express my appreciation to Dr. Bruton for his supervision, advice, and especially the encouragement he has given me throughout the course of this thesis. Although I was occasionally frustrated with my results, Dr. Bruton would always keep me focused on my goals and working on this thesis turned out to be a very valuable and rewarding experience.

I would also like to thank NSERC, The University of Calgary, and Dr. Bruton for the generous financial support they have given to me throughout my degree, without which this thesis would have never been written.

Lastly, I wish to thank Norm Bartley for making the lab run so smoothly, providing suggestions and support, and for injecting humour into, what could sometimes be, humour-less days. To my lab-mates, Remi Gurski, James Gordy, and Joseph Provine, thank you for your friendship, input, suggestions, and lively conversation and especially to Remi for his happy greetings every morning. I also want to thank Mark Chakravorti, Neil Cumbria, and Oliver Tozser for their friendship and providing a little (in)sanity in my everyday life. Finally, to the ladies in the Electrical Engineering office, Ella, Val, Jenny, Judy, and Angela, thank you for keeping the paperwork on track, reminding me when things were due, and for generally putting up with me.

To my family,

without whos love and

support this thesis would

not have been possible.

Contents

List of Tables

List of Figures

List of Symbols and Abbreviations

AC	Non-zero frequencies
ALSBI	Adaptive Least-Squares Bilinear Interpolation
ALSTI	Adaptive Least-Squares Trilinear Interpolation
BSE	Block Size Equalization
codec	coder/decoder
$\delta(n)$	Unit impulse function
DC	Zero frequency
DCT	Discrete Cosine Transform
DDA	Digital Differential Analyzer
D_{max}	Maximum Block Dimensions
D_{min}	Minimum Block Dimensions
DPCM	Differential Pulse Code Modulation
FIR	Finite Impulse Response
HVS	Human Visual System
JPEG	Joint Photographic Experts Group
LSBI	Least-Squares Bilinear Interpolation

LSLI	Least-Squares Linear Interpolation
LSTI	Least-Squares Trilinear Interpolation
LZW	Lempel-Ziv Welch
m-D	m-dimensional. One-, two-dimensional, etc. are denoted by 1-D, 2-D, etc.
MJPEG	Motion JPEG
MSE	Mean Squared Error
pixel	picture element
PSNR	Peak-Signal-to-Noise-Ratio
Q	Vertex Quantization Factor
voxel	volume element

Chapter 1

Introduction

Over the last few decades there has been a dramatic increase in digital image and video usage, which has resulted in a similar increase in storage and transmission requirements to accommodate the large amount of data associated with these types of contents [1]. While it has been said that a picture is worth a thousand words, this description is somewhat of an understatement. For a simple greyscale digital image composed of 256×256 values, where each value is described by 8 bits (one character) of data, this single image would require over 65000 characters to describe it, which is much larger than 1000 words—even if very large words are used!

Digital video, which is simply composed of a sequence of images, requires even more data for its representation because it has one added dimension: time. The amount of data is directly proportional to the number of images (or frames) that are in the sequence, and the number of images is equal to the length, in time, of the video sequence multiplied by the frame rate. Thus, it is easy to see that if the temporal sampling rate is doubled, so is the data required to represent the image sequence.

Whether the images, being alone or contained in an image sequence, are to be stored or transmitted, this large amount of information can be quite undesirable.

Therefore, along with the increased use of digital images and video, there has also been an increased desire to reduce, or compress, the information requirements of these types of media [1].

1.1 Digital Image and Video Data Compression

Data compression techniques generally involve taking advantage of redundancy within the input data to reduce the average number of bits per element required to represent and properly reconstruct that data [2]. These elements are actually digitized samples of intensity values within the image or image sequence. For colour images or video, each sample contains multiple intensity values: one for each colour component. In order to represent greyscale images or video, each sample is simply one intensity value indicating the brightness of the image or video at that particular point.

In general, there are two different classes of data compression methods: lossless and lossy data compression [3]. Lossless data compression (or lossless data coding) is defined as a system that reduces the number of bits used to represent a set of data while allowing the data to be recovered perfectly by decompression. Conversely, lossy data compression (or lossy data coding) actually removes some of the information from the data set in order to further reduce the number of bits used to represent it. This results in a difference occurring between the coder input and decoder output. The information usually removed during lossy data compression is subjectively less important to the output quality of the entire data set because it can be reasonably recovered by using the remaining information [4]. Lossless coding is usually limited to small compression ratios. On the contrary, the output file size resulting from lossy coding is only limited by the amount of acceptable distortion present in the reconstructed output.

Selection of lossy or lossless coding is based on the data itself and the requirements of the application using the data. For example, a text compression algorithm must be lossless otherwise the decompressed (decoded) output could be meaningless if distortion occurred. In image compression, the application determines what type of coding is desired. For medical imaging, lossless coding may be required because any distortion in the output image could lead to disastrous results like a misdiagnosis. However, for compression of holiday snapshots, some distortion may be acceptable in order to achieve smaller file sizes or less transmission bandwidth.

With the large amount of storage required for image and video data, there have been many developments in image and video data compression. There are many techniques, both lossless and lossy, based on a wide variety of algorithms and requirements [1]. Generally, all these methods take advantage of certain redundancies found in image and video data. Image coding involves the exploitation of spatial redundancies found within an image [5]. Video coding also does this on a per-frame basis, which is also known as *intra*-frame coding. Furthermore, video coders may also perform *inter*-frame coding which involves taking advantage of the similarities between frames [5]. Both video and image coders, when coding colour images or image sequences, can also use chromatic redundancy to their advantage [6]. As this thesis deals mainly with greyscale images and image sequences, the exploitation of chromatic redundancy will not be discussed.

A lossy coder/decoder (codec) system, by definition, will introduce distortion, or what is known as visual artifacts, into the output. There are various types of distortion caused by lossy coding. Certain distortion types can be seen in the output of most codec systems, while some types of artifacts can be particular to an individual codec. Some types of distortion caused by image coding are blockiness, texture and fine pattern degradation, waviness in smooth areas and around edges, and blurring of

image features [7]. These artifacts can also be seen on a per-frame basis in the output of various video codecs. However, when using video codecs, the above spatial effects are not the only distortions visible. Video codecs can introduce a number of temporal artifacts as well. Some of these temporal artifacts include blurring, "ghosting", and removal of moving objects. In non-moving areas, such as a constant background, defects such as jittering and intensity changes with time, can be seen.

1.1.1 Image and Video Coding Performance Metrics

In order to evaluate the effectiveness of different digital image and video codecs, some performance metrics must be defined. There are two general measurements that are associated with data compression techniques: compression and quality.

Compression can be measured either by compression ratio or bits per element, where the element is either a picture element (pixel) when image coding or a volume element (voxel) when video coding. Compression ratio is a measurement that is defined as the ratio between the input file size and the output file size. Since there are various factors that can artificially inflate the compression ratio of an image or video coder, such as the input file format or tricks with up-sampling [8], comparing methods solely based on compression ratio is not always best. A better compression performance measurement that is widely used in image coding is the average number of bits/pixel required to represent the output image. Knowing the number of pixels within the image and the bits/pixel, it is also possible to calculate the compressed file size. Due to the unambiguous nature of the bits/pixel measurement, digital image compression will be measured in bits/pixel for the remainder of this thesis. For video coding, a popular compression metric used is the bit rate of a video sequence, which is the average number of bits per second required for the compressed data

stream [9]. This measurement can be somewhat deceptive because it depends on the spatial resolution and the frame rate of the video sequence. However, it does define the transmission bandwidth required for the compressed video data. A more independent measurement is the number of bits/voxel. If bit rate is required, bits/voxel is easily converted by multiplying it by the number of pixels-per-frame and the frame rate of the image sequence. Thus, in this thesis, bits/voxel will be the preferred measurement for video compression performance.

When dealing with data compression, the concept of *entropy* is sometimes used. Entropy is the theoretical minimum average number of bits per element—pixels or voxels in the case of image or video compression—required to reversibly represent a sequence of elements [2]. Entropy is based on the statistical occurrence of elements within a sequence. Suppose the input sequence is chosen from a set of N elements, where, in the sequence, these elements occur with respective probabilities of p_i for $i = 1, \ldots, N$, and so that $\sum p_i = 1$. Thus, the input sequence, on average, will require at least

$$H = -\sum_{i=1}^{N} p_i \log_2 p_i \quad \text{bits/element} \tag{1.1}$$

where H is the first-order entropy of the probability distribution, or more simply, the entropy of the input sequence [2]. It can be seen that in the case of equiprobable elements, with all $p_i = 1/N$, compression is not possible since $H = \log_2 N$. Thus, a completely random sequence is not compressible. Conversely, for any other set of p_i (i.e. where at least one element is the same as another), a smaller entropy results, allowing for possible compression.

Quality can be measured in many different ways. Since, in most cases, the reconstructed images or image sequences are viewed by human eyes, quality can be a somewhat subjective measurement that can be hard to quantify. Some quality mea-

surements are based on complex Human Visual System (HVS) models or use human test subjects to rate the quality of reconstructed images or image sequences [10]. Other measurements which simply deal with reconstruction error are much more simple [10]. For lossless coding, quality is meaningless as the reconstructed image is identical to the original. When lossy coding is performed, the visual artifacts or errors that result, contribute to the loss of visual quality. These errors can be thought of as noise introduced by the coding/decoding process. A simple and widely used quality measurement is Peak-Signal-to-Noise-Ratio (PSNR), measured in decibels (dB), which relates the maximum signal power to the noise power, where the noise power is simply the Mean Squared Error (MSE) [10]. PSNR is defined as [10]:

$$\text{PSNR} = 10 \log_{10} \frac{[\text{maximum intensity value}]^2}{\text{mean squared error}} \tag{1.2}$$

So for an 8-bit greyscale signal (image or image sequence) composed of N elements, the PSNR is

$$\text{PSNR} = 10 \log_{10} \frac{[255]^2}{\frac{1}{N} \sum_{i=0}^{N-1} [y(i) - x(i)]^2} \tag{1.3}$$

where $y(i)$ is the reconstructed signal and $x(i)$ is the original signal [11]. When used as a measurement of perceptual image quality, PSNR tends to be somewhat image dependent. However, for the same image, it does provide a good comparison when relating different reconstruction qualities. Furthermore, it has been stated in [10] that "coders that incorporate techniques to minimize the MSE are ranked at the top in *both perceptual and objective tests*!" This means that PSNR can provide almost as much insight into the quality of a reconstruction as some, more complex, HVS models. For these reasons, PSNR will be used to measure the reconstructed quality of both images and image sequences in this thesis.

1.2 Sample Image and Video Coding Standards

The goal of most digital image and video codecs is to reduce the storage or transmission requirements of the data required to represent the image or video, while maintaining a certain level of output quality. The following sections give a general overview of a standard image codec and a standard video codec.

1.2.1 JPEG Image Compression

JPEG is a standard set of image coding algorithms used for the compression of "natural" digital images. JPEG is an acronym for Joint Photographic Experts Group, which is the name of the International Organization for Standardization (ISO) committee that defined the image compression algorithms [12]. While the JPEG standard consists of 16 different image coding algorithms, the simplest, or baseline, algorithm is the most popular [7, 12, 13] and hereafter the baseline JPEG algorithm will simply be referred to as JPEG.

The use of JPEG is very widespread. Anyone who has browsed the world-wide web has experienced JPEG image coding. Usually, for both greyscale and colour images, the threshold of visible difference between the source and reconstructed images is around 1–2 bits/pixel [3].

At the core of the baseline JPEG algorithm lies the Discrete Cosine Transform (DCT), which performs a spatial-domain to frequency-domain transformation. The image is first subdivided into blocks of 8×8 pixels. Then a Block DCT is performed on each of these blocks so that certain frequency components can be removed or adjusted by the JPEG algorithm without affecting other components. This results in each block having 64 frequency elements, ranging from DC (zero frequency) to half the sampling frequency in each direction. Each of these frequency components is

then divided by a separate quantization coefficient, from a quantization table, and the results are rounded to integers. Depending on how heavily the DCT coefficients are quantized, there may be a large number of zero-valued AC (non-zero frequencies) coefficients. The AC coefficients are then run-length encoded in a zig-zag pattern across each 8×8 block so that runs of zeros can be increased in length. The DC components are difference encoded to take advantage of average intensity similarities between neighboring blocks. Finally, all this information is coded using variable length codes that exploit statistical redundancies to reduce the overall file size [1, 3, 13].

The JPEG algorithm, although computationally complex, works quite well for natural "continuous tone" images. However, at high compression ratios artifacts such as blockiness, corruption of textures and fine details, and waviness in smooth areas and around edges, are easily detectable. Furthermore, JPEG performs quite poorly on black-and-white and other two-toned images. This is due to the large amount of high contrast edges found in these types of images [3, 7].

A video codec can also be implemented by using the JPEG algorithm by coding an image sequence one frame at a time; when used in this fashion, it is called Motion JPEG (MJPEG). It should be noted that MJPEG is not part of any "official" video coding standard [12]. The MJPEG frame-by-frame coding can be performed by any image coder in order to implement a video coder. However, since each frame is coded independently, these types of video codecs cannot take advantage of inter-frame correlations, which limit their video compression performance.

1.2.2 MPEG, H.261, and H.263 Video Compression

MPEG, H.261, and H.263 are three very closely related video codecs. MPEG, an acronym that stands for Moving Picture Experts Group, is an international stan-

dard of the ISO, while H.261 and H.263 are recommendations of the International Telecommunications Union (ITU) [9]. There have been multiple MPEG standards defined or are in the process of standardization: MPEG 1, MPEG 2, MPEG 4 (versions 1 and 2), and MPEG 7. This discussion mainly concerns the more closely related MPEG 1 and MPEG 2—hereafter referred to as MPEG. The MPEG codec is based on H.261 and JPEG, while H.263 is based on H.261 and MPEG [8]. MPEG is designed for the storage and playback of high quality video and is can produce VHS videotape quality or better when operating at around 1–2 Mbits/s [7, 11]. The MPEG video codec is extremely popular and is used in many popular products such as DBS (Direct Broadcast Satellite), DVD (Digital Video Disc), and has been adopted for use by the all-digital HDTV (High Definition Television) consumer transmission standard [9]. The other two codecs, H.261 and H.263, are intended for teleconferencing applications [7]: the target bit rate for H.261 is 64–2048 Kbits/s while H.263 has a wider bit rate range of 10–2048 Kbits/s [11].

These three image sequence codecs are based on the Block DCT, predicted frames, and motion estimation. In a similar fashion to JPEG, each of these codecs makes use of the Block DCT to encode frames. It is the predicted frames and motion estimation that take advantage of the inter-frame correlation, so that these codecs can provide better performance than MJPEG.

There are three different frame types: Intra (I), Predicted (P), and Bidirectionally Predicted (B) [3]. I-frames are simply coded as a still image, not using any past or future information. Since they can be decoded independently from any other frame, they provide random access to the image sequence—decoding can begin at any I-frame. P-frames are predicted using past information from the most recent I- or P-frame. Lastly, B-frames are also a type of predicted frame, but they are predicted both in the forward and backward direction from the previous and next I-

or P-frames [3]. Only MPEG and H.263 include the use of B-frames. H.261 does not [7, 11]. It should be noted that if only I-frames are used, the coding performance will be very similar to MJPEG.

A predicted frame is a difference frame, which is the difference between the current frame and a previously encoded and reconstructed frame. Usually, the difference values will be quite small over the entire image, except around the edges of moving objects and where new objects are introduced. The small dynamic range of the difference frame enables it to be encoded with a fewer number of bits.

Motion estimation is the estimation of translational object motion in the current frame with respect to another frame. By using motion estimation, it is possible to lower the energy in the frame difference (P- and B-frames) by moving pixels around to simulate object motion [7]. The coder must then include a small amount of motion information in the compressed data so that the decoder can replicate the pixel motion exactly. The amount of motion information is kept low by only estimating motion for blocks of pixels: 8×8 or 16×16.

As stated above, MPEG and H.263 make use of B-frames to enhance prediction and motion estimation. B-frames use previous and next frames as predictors for the current frame, which generally results in about one third the amount of data required for a P-frame [7]. The use of future frames implies out-of-order encoding (and decoding) because the coder can encode a B-frame only after encoding the required previous and future frames. This significantly increases the codec complexity.

The Block DCT core of these complex video codecs can produce visual artifacts similar to the JPEG codec. These include blockiness and distortion around the edges of objects. The artifacts are especially noticeable around scene changes and fast moving objects. Another problem with these codecs is the large number of computations required: the largest proportion of the calculations performed during

motion estimation.

1.3 Motivation Behind Thesis

Both the above mentioned image and video codecs have undesirable properties associated with them. When using JPEG or MPEG, artifacts related to the Block DCT can sometimes be seen, especially at high compression ratios around sudden jumps in intensity and, with MPEG, around fast moving objects. Furthermore, the JPEG algorithm does not perform well on images with few greyscales or simple two-toned images. Lossless coding is also not possible for MPEG or JPEG.

Coding and decoding complexity is also an issue, especially with MPEG. The JPEG algorithm involves many steps and many Block DCT calculations. MPEG also requires Block DCT calculations, possible out-of-order coding/decoding, and motion estimation, which is very computationally demanding.

The combination of algorithm complexity and undesirable output artifacts creates the desire to implement codecs that reduce the computational complexity while increasing the reconstructed output quality. Thus, it is the goal of this thesis to study two closely related codecs that use adaptive sampling and interpolation for image and video coding. Interpolation can be very simple and will not produce visible "ringing" in the output. It has also been found that adaptive sampling methods can reconstruct sharp edges well, make lossless coding possible, and do not have the blockiness in the output that is associated with other fixed-block-size-based coders [14]. Furthermore, by treating an image sequence as a 3-D volume of intensity values, the adaptive sampling and interpolation method can be used to code and decode digital video—all without the need for computationally expensive motion estimation.

1.4 Thesis Overview

This first chapter has presented the background and reasons behind the need for digital image and image sequence data compression. Included was a discussion of various image and video coding metrics used in measuring the quality and compression performance of various codecs. Also, an example of a current image compression standard, JPEG, was described along with an overview of a closely related group of digital image sequence codecs: MPEG, H.261, and H.263. The effects and disadvantages of using these codecs have also been presented: the undesirable effects being the motivation behind the work presented in this thesis.

In Chapter 2, interpolation in both one and two dimensions is described in detail. This then leads to the description of an existing image coder that uses adaptive sampling and 2-D interpolation [15]. In the image coder, bilinear interpolation is used along with an adaptive sampling grid to reduce the data requirements of a digital image.

The discussion in Chapter 3 focuses on implementation issues concerning the image coder presented in Chapter 2. Included in this discussion is exactly how the adaptive sampling grid can be efficiently generated and represented, as well as how the samples themselves can be stored [15]. It is possible to increase the output image quality by using a least-squares error minimization technique, which is also described in this chapter [15, 16]. Some image coding results are also presented.

An extension of the 2-D image coder into 3-D image sequence coding is detailed in Chapter 4. Trilinear interpolation, which is the basis of this video codec [17], is discussed along with implementation specifics of the system. A least-squares method is introduced that reduces the trilinear interpolation error and increases the output image sequence quality. Various image sequence coding results are also given.

Finally, in Chapter 5 the performance of the image and video codecs are summarized, along with their problems, and possible improvements. Some advice on further research in this area is also given.

Chapter 2

Image Coding via Adaptive Two-Dimensional Sampling and Interpolation

This chapter will examine an image coding method that is based on the adaptive sub-sampling of a digital image and the reconstruction of the image from the sub-sampled points by using bilinear interpolation. In Section 2.1, a brief overview of the image coding method will be shown. Then, in order to have a proper introduction to bilinear interpolation, Section 2.2 will describe linear interpolation in detail as well its effects in the frequency domain. One-dimensional processes are examined in great detail because working in one dimension is usually easier than multiple dimensions and in this chapter most of the 2-D operations can be decomposed into multiple 1-D operations. This is apparent in Section 2.3 where the bilinear interpolator is constructed from multiple linear interpolations. Also, a 2-D frequency analysis of the bilinear interpolator will be given. To show how bilinear interpolation can be used in image coding, Section 2.4 is an illustrative example describing how the bilinear

interpolator can be used to reconstruct an image from a set of equi-spaced samples generated by sub-sampling on a uniform 2-D grid. Following that, Section 2.5 introduces an adaptive method for image sub-sampling on a non-uniform 2-D grid to retain the output image quality without impacting compression performance.

2.1 Overview of the Image Coding Scheme

Interpolation has been extensively used in many 1-D data compression techniques, such as audio coding [4]. Compression can be achieved by reducing the the total number of samples which represent the signal. Then, in order to reconstruct the original sequence, the missing data sample values can be estimated using interpolation. Depending on the complexity of the signal, it may be possible to reduce the error due to estimation by using a higher-order interpolator. However, if the complexity of the coding system is to remain low and/or high speed operation is needed, then a low order interpolator would be a more suitable choice.

Image data compression involves the coding of a 2-D signal, where the signal is composed of discrete intensity samples or values, which are also known as pixels. In the hierarchical image coding method described in [15] and which is further detailed in this chapter, a 2-D interpolation of intensity samples on a plane is required by the decoder to fill in samples removed by the coder. For fast computation, a simple bilinear interpolator is selected to fulfill this task, which basically involves filling in values between four known points that lie on a rectangle in two dimensions.

In Figure 2.1 an overview of the image coder is shown. In the coder a recursive block coding algorithm is used to split the original image into smaller blocks that share common corner pixels and lie on a non-uniform grid. Each block, when split, results in four smaller blocks. This splitting information, which is crucial in the reconstruction

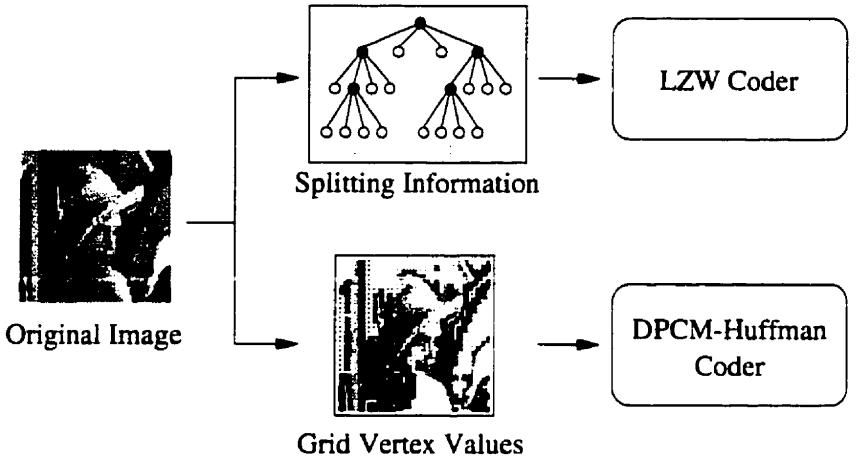


Figure 2.1: Block diagram of the image coder.

stage, is stored in a tree structure where a block that is split (represented by a black node in the figure) becomes the "parent" node for four new sub-blocks. The splitting process is performed until the coder determines that the bilinear interpolation error for each block is smaller than a given threshold. It is possible to retain all the image information by setting the error threshold to zero. That is, "lossless" coding is possible.

After the image has been subdivided and the splitting tree generated, the information stored in the tree structure is compressed via the use of a Lempel-Ziv Welch (LZW) coder [18]. The intensity values that lie on the non-uniform grid vertices resulting from the splitting process are retained while all other samples are discarded. This is the main information removal step. The amount of data required to store the retained grid vertex values is compressed further by using a Differential Pulse Code Modulation (DPCM) algorithm followed by a Huffman entropy coder [2, 15, 17].

The image decoder shown in Figure 2.2 is quite similar to the image coding system but their operations are more or less reversed and the decoder requires less of a the computational load than is required by the coder. In order to reconstruct

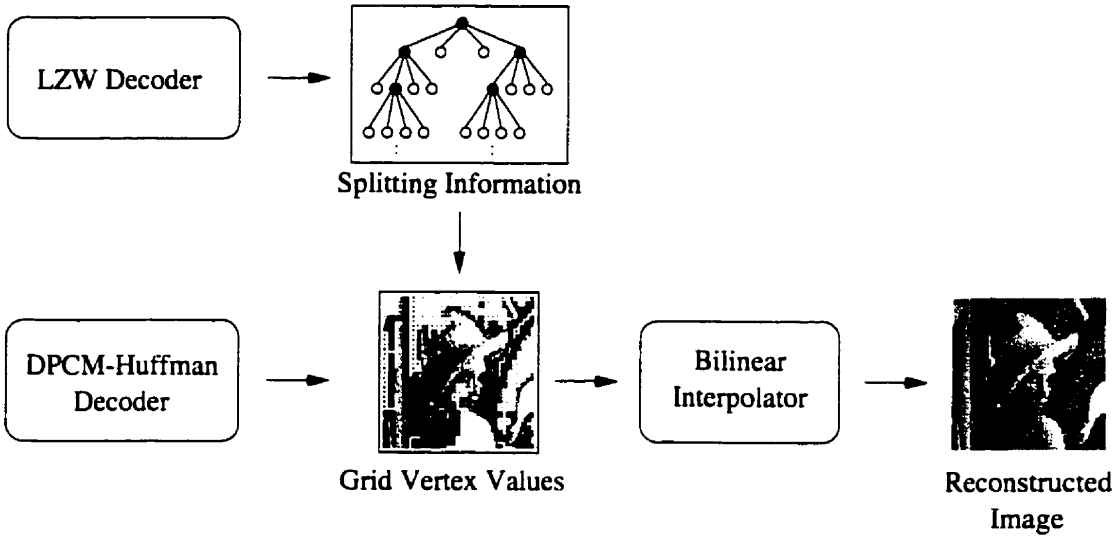


Figure 2.2: Block diagram of the image decoder.

the image using the bilinear interpolator, the decoder must first replicate the non-uniform sampling grid so as to have rectangles with known corner values to perform bilinear interpolation on. Since the structure of the sampling grid, including block sizes and positions, is contained in the splitting tree, the decoder must first decode the LZW coded data that represents the splitting tree. Then, by knowing the original image dimensions and how it was subdivided by the coder, the sampling grid can be reconstructed. With the sampling grid established, the decoder can Huffman-DPCM decode the vertex values and place them into their proper locations within the grid. Finally, with the intensity values retained by the coder in place, the decoder can reconstruct the image by traversing the splitting tree and filling in unknown intensity values within child blocks (i.e. blocks that have not been split) through bilinear interpolation [15].

The major advantages of this image codec system over other image coding techniques are

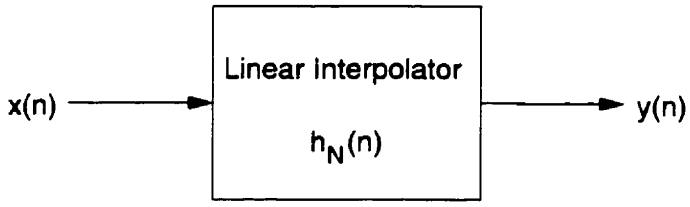


Figure 2.3: Linear interpolation system.

1. the absence (or near-absence) of floating point calculation requirements.

2. the possibility of high speed operation in both the coding and decoding stages.

3. the absence of edge effects and blockiness associated with some other image coding techniques.

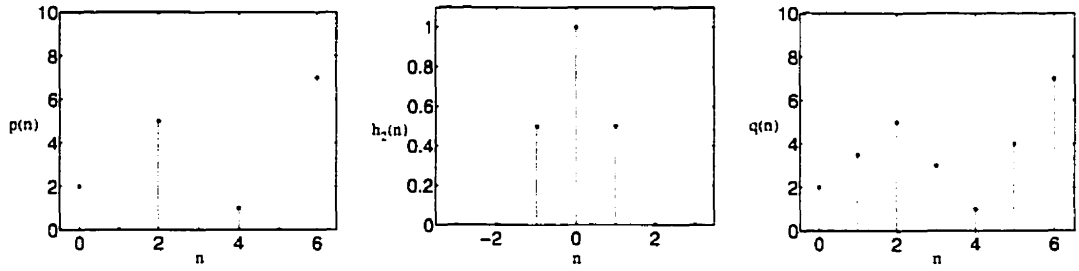
4. and the possibility for lossless coding of an image—depending on implementation and original image dimensions (see Section 3.2.2 for further details).

2.2 One-Dimensional Linear Interpolation

Linear interpolation in one dimension is a simple method that can be used to generate sample values given two known values and their indices (locations) [15]. Thus, given two known sample values (i.e. intensities), Z_0 and Z_N, separated by $N-1$ unknown samples, linear interpolation of a general point, Z_i is:

$$Z_i = \frac{Z_N - Z_0}{N} i + Z_0 \tag{2.1}$$

Now, if given a signal, where every Nth sample is known, Equation 2.1 can be used to piecewise linear interpolate the signal. In this situation, it is more convenient to think of the interpolation process as an input/output system. This is demonstrated in Figure 2.3, where $x(n)$ is an input signal, where every Nth sample is known and



(a) (b) (c)

Figure 2.4: Example of 1-D linear interpolation (a) Up-sampled signal $p(n)$. (b) Linear interpolating filter impulse response $h_2(n)$. (c) Linear interpolated signal $q(n)$.

all others are zero (which constitutes an up-sampled signal), and $y(n)$ is the piecewise linear interpolated output. The linear interpolating filter, $h_N(n)$, has a non-causal sawtooth impulse response of the form [14]:

$$h_N(n) = \begin{cases} 1 - \left|\frac{n}{N}\right| & \text{if } n < |N| \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

Thus, the interpolated signal, $y(n)$, is produced by convolving the input signal, $x(n)$, with the impulse response of the linear interpolating filter given in Equation 2.2. That is, the piecewise linear interpolated output of the filter is:

$$\begin{aligned} y(n) &= h_N(n) * x(n) \\ &= \sum_{i=-\infty}^{\infty} h_N(i)x(n-i) \end{aligned} \tag{2.3}$$

and since $h_N(i)$ has a limited region of support, i.e. $h_N(i) \neq 0$, for $-N < i < N$, then

$$y(n) = \sum_{i=-(N-1)}^{N-1} h_N(i)x(n-i) \tag{2.4}$$

For example, the sequence $p(n)$ in Figure 2.4(a) requires interpolation to fill in every second sample value (note the zero values at $n = 1$, 3, and 5). Using a linear interpolating filter with an impulse response $h_2(n)$ shown in Figure 2.4(b), the

missing samples are replaced by interpolated values. The resulting output, $q(n)$, is calculated using Equation 2.4 so that:

$$q(n) = h_2(n) * p(n) \tag{2.5}$$

and is shown in Figure 2.4(c).

It should be noted that for the same input sequence, Equations 2.1 and 2.4 are equivalent:

$$Z_n = y(n), \quad \text{for } 0 \leq n \leq N \tag{2.6}$$

For piecewise linear interpolation, the two calculation methods will have identical output within the input signal's domain.

2.2.1 Frequency Response of the Linear Interpolating Filter

With the impulse response of the linear interpolating filter defined in Equation 2.4, it is possible to calculate the frequency response of $h_N(n)$ and gain insight into how linear interpolation affects a signal in the frequency domain, especially an up-sampled signal. Taking the Fourier Transform of $h_N(n)$ yields:

$$\begin{aligned} H_N(e^{j\omega}) &= \sum_{k=-\infty}^{\infty} e^{-j\omega k} h_N(k) \\ &= \sum_{k=-(N-1)}^{N-1} \left(1 - \left|\frac{k}{N}\right|\right) e^{-j\omega k} \end{aligned} \tag{2.7}$$

Since $h_N(k)$ is symmetrical about $k = 0$, this can be further simplified so that:

$$H_N(e^{j\omega}) = 1 + 2\sum_{k=1}^{N-1} \left(1 - \frac{k}{N}\right) \cos k\omega \tag{2.8}$$

This shows that $H_N(e^{j\omega})$ is a zero-phase lowpass filter with its bandwidth inversely proportional to N. Figure 2.5 shows $H_N(e^{j\omega})$ for various values of N.

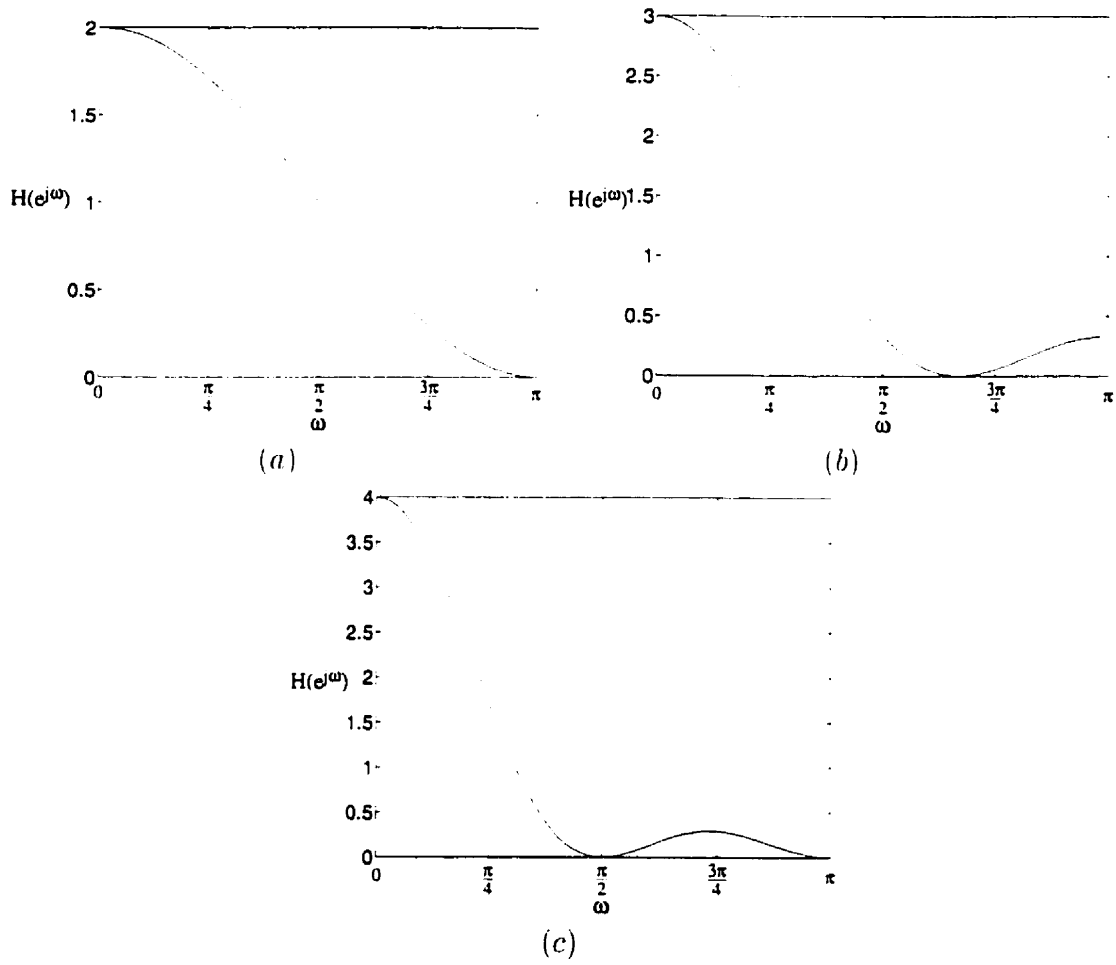


Figure 2.5: Frequency response of $h_N(n)$ for (a) $N = 2$, (b) $N = 3$ and (c) $N = 4$.

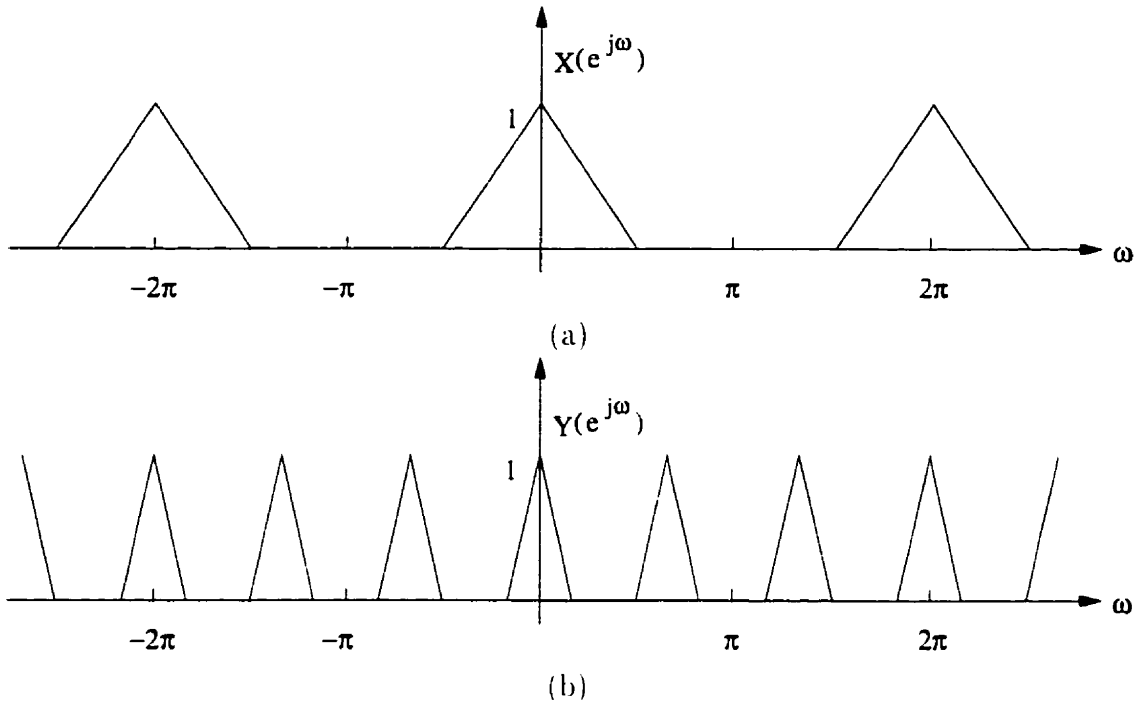


Figure 2.6: Frequency spectrum of (a) a lowpass signal and (b) the signal after up-sampling by a factor of $N = 3$.

2.2.1.1 Analysis of the Up-Sampling Process

Up-sampling by a factor N involves the insertion of $N - 1$ zero valued samples between each sample of the input signal. The input output relationship for the up-sampling process is

$$x(n) = \begin{cases} x\left(\frac{n}{N}\right) & \text{if } n = \text{mult of } N \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

The effect of this stretching in the time domain results in a compression in the frequency domain and is demonstrated in Figure 2.6 for $N = 3$. The figure shows that in the frequency domain the up-sampled signal is composed of $N - 1$ replicas of the original signal, in the same frequency range, shrunken in bandwidth by a factor of N. The transform domain relationship between input and output is quite simple and is

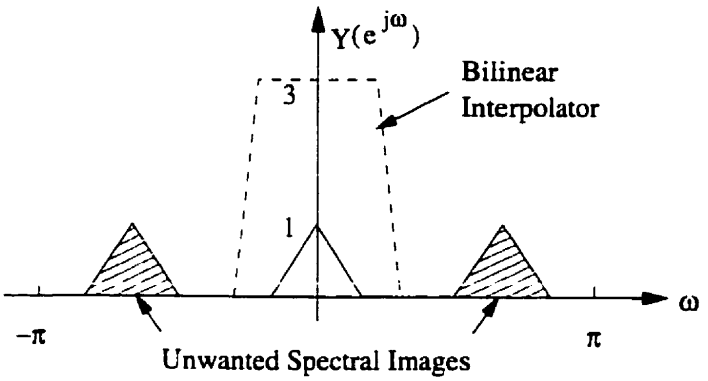


Figure 2.7: Removal of unwanted spectral images for $N = 3$.

given by [19]:

$$Y\left(e^{j\omega}\right) = X\left(e^{j\omega N}\right) \tag{2.10}$$

If the goal is to recover the original signal, the unwanted frequency replicas introduced by up-sampling must be removed. This is done by using a simple bandpass (or lowpass) filter centred around the desired replica that will reject the unwanted frequency replicas while retaining the one that lies within the bandwidth of the filter.

In this image data compression scheme the desired frequency replica lies on the origin so a lowpass filter is needed. However, as seen in Section 2.2.1, the linear interpolator acts as a lowpass filter that decreases in bandwidth with an increase in N. So the linear interpolator is an ideal choice for removing the unwanted spectral images from the up-sampled signal both for its frequency selectivity and especially its simplicity. For example, in Figure 2.7, the removal of unwanted spectral images by a bilinear interpolating filter is shown.

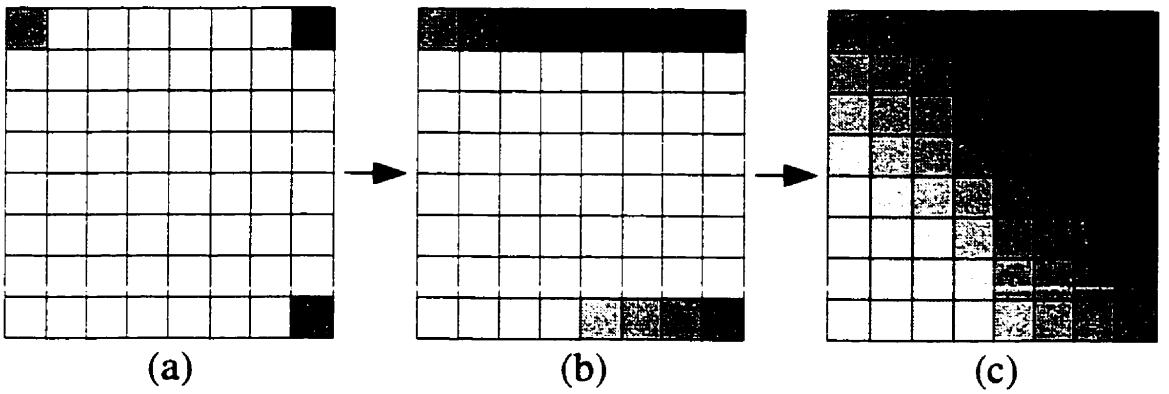


Figure 2.8: Two step, two dimensional bilinear interpolation [15]: (a) original block, (b) interpolation of top and bottom rows, (c) interpolation of each column.

2.3 Two-Dimensional Bilinear Interpolation

Bilinear interpolation is an operation that involves separate linear interpolations (defined in Equation 2.1), in both spatial directions of the 2-D domain [15]. Thus, given a 2-D signal $Z_{i,j}$, with $i, j \in [0 \ldots N]$, and known corner values, a bilinear interpolation is simply achieved by linear interpolating, first in the horizontal direction, between the pairs of values $Z_{0,0}$, $Z_{0,N}$ and $Z_{N,0}$, $Z_{N,N}$, and then by vertically interpolating the values between the pairs of points $Z_{0,j}$ - $Z_{N,j}$ with $j \in [0 \ldots N]$. Figure 2.8 demonstrates how bilinear interpolation can be performed using a series of 1-D linear interpolations. The two step process involves the linear interpolation of the top and bottom rows which is then followed by the interpolation of each column between the top and bottom rows. The two-step bilinear interpolation is equivalent if instead the left and right columns are linear interpolated first and *then* the rows between them are interpolated.

So, given a rectangular block of size $(N_1 + 1) \times (N_2 + 1)$ and the four corner values ($Z_{0,0}$, Z_{0,N_2}, $Z_{N_1,0}$, Z_{N_1,N_2}) are known, the interpolation of a generic point $Z_{i,j}$ can be calculated as follows [17]:

1. compute the value of the point $Z_{0,i}$ (which is a linear interpolation of $Z_{0,0}$ and Z_{0,N_2}):

$$Z_{0,i} = \frac{Z_{0,N_2} - Z_{0,0}}{N_2} i + Z_{0,0} \tag{2.11}$$

2. compute the value of the point $Z_{N_1,i}$ (linear interpolation of $Z_{N_1,0}$ and Z_{N_1,N_2}):

$$Z_{N_1,i} = \frac{Z_{N_1,N_2} - Z_{N_1,0}}{N_2} i + Z_{N_1,0} \tag{2.12}$$

3. compute the value of the point $Z_{i,j}$ (linear interpolation of $Z_{0,j}$ and $Z_{N_1,j}$ from equations 2.11 and 2.12):

$$\begin{aligned} Z_{i,j} &= \frac{Z_{N_1,j} - Z_{0,j}}{N_1} i + Z_{0,j} \\ &= \frac{(Z_{N_1,N_2} + Z_{0,0}) - (Z_{N_1,0} + Z_{0,N_2})}{N_1 N_2} ij + \\ &\quad \frac{(Z_{N_1,0} - Z_{0,0})}{N_1} i + \frac{(Z_{0,N_2} - Z_{0,0})}{N_2} j + Z_{0,0} \qquad (2.13) \\ &= Aij + Bj + Ci + D \qquad (2.14) \end{aligned}$$

Looking at equation 2.14, it should be noted that the bilinear interpolation function, although easy to compute using a series of linear interpolations, is not a simple first order interpolator but a second order 2-D spline function.

Now, if given a 2-D signal in which known sample values lie on a regular grid, that is every N_1th sample in the horizontal direction and every N_2th sample in the vertical direction are known, Equation 2.13 can be used to piecewise bilinear interpolate the signal. However, as in Section 2.2, it is useful to consider the bilinear interpolation process as a 2-D filter in order to examine its characteristics.

In order to create a bilinear interpolating filter for horizontal and vertical up-sampling factors of N_1 and N_2, respectively, a filtering operation with an impulse response, as in Equation 2.2, is performed in both the horizontal and vertical directions. These separate filtering operations can be combined into a single filter.

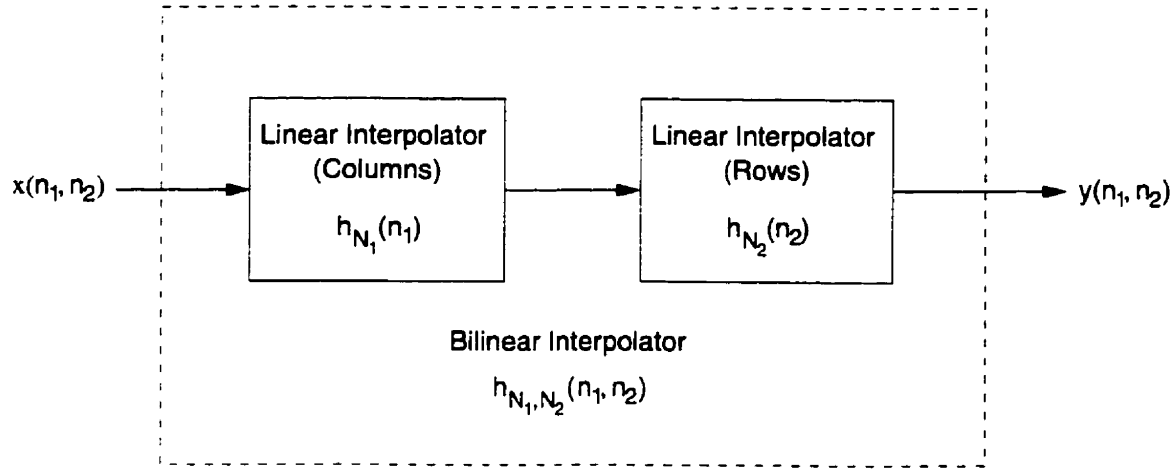


Figure 2.9: Signal-based bilinear interpolation system.

Figure 2.9 shows the input being interpolated by two different linear interpolators in each direction which is equivalent to a single bilinear interpolator.

The overall impulse response of the system in Figure 2.9 is the 2-D convolution of the two impulse responses, first converting $h_{N_1}(n_1)$ and $h_{N_2}(n_2)$ to the 2-D domain. That is,

$$\begin{aligned} h_{N_1,N_2}(n_1,n_2) &= h_{N_1}(n_1,n_2) * h_{N_2}(n_1,n_2) \\ &= (h_{N_1}(n_1)\cdot\delta(n_2)) * (\delta(n_1)\cdot h_{N_2}(n_2)) \\ &= h_{N_1}(n_1)\cdot h_{N_2}(n_2) \end{aligned} \tag{2.15}$$

and substituting in the definitions for $h_{N_1}(n_1)$ and $h_{N_2}(n_2)$ results in:

$$h_{N_1,N_2}(n_1,n_2) = \begin{cases} 1-\left|\frac{n_1}{N_1}\right|-\left|\frac{n_2}{N_2}\right|+\left|\frac{n_1 n_2}{N_1 N_2}\right| & \text{if } |n_1|<N_1 \text{ and } |n_2|<N_2 \\ 0 & \text{otherwise} \end{cases} \tag{2.16}$$

As an example, the impulse response of the bilinear interpolating filter with $N_1 = 5$ and $N_2 = 5$ is shown in Figure 2.10

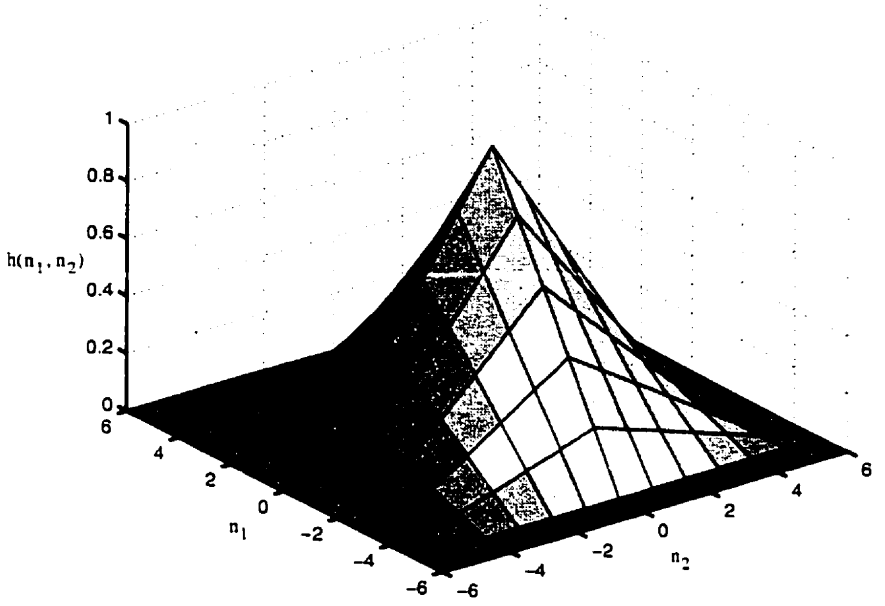


Figure 2.10: Impulse response of the bilinear interpolating filter $h_{5,5}$.

Thus, the result of filtering a two dimensional signal, $x(n_1, n_2)$, with the bilinear interpolating filter of Equation 2.16 is a 2-D convolution of the input signal with the impulse response of the filter:

$$\begin{aligned} y(n_1, n_2) &= h_{N_1,N_2}(n_1, n_2) * x(n_1, n_2) \\ &= \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} h_{N_1,N_2}(i, j) x(n_1 - i, n_2 - j) \end{aligned} \tag{2.17}$$

and since $h_{N_1,N_2}(i, j)$ has a limited region of support,

$$y(n_1, n_2) = \sum_{i=-(N_1-1)}^{N_1-1} \sum_{j=-(N_2-1)}^{N_2-1} h_{N_1,N_2}(i, j) x(n_1 - i, n_2 - j) \tag{2.18}$$

As with the 1-D case, for identical 2-D input sequences, Equations 2.13 and 2.18 will produce the same output:

$$Z_{n_1,n_2} = y(n_1, n_2) \quad \text{for } 0 \leq n_1 \leq N_1 \text{ and } 0 \leq n_2 \leq N_2 \tag{2.19}$$

As well, for piecewise bilinear interpolation, the two calculation methods will have identical output within the input signal's domain.

2.3.1 Frequency Response of the Bilinear Interpolating Filter

In order to determine the frequency response of the bilinear interpolating filter, the Fourier Transform could be applied to the impulse response described in Equation 2.18. However, since the row filter and column filter are cascaded, the overall frequency response of the system in Figure 2.9 is the product of the two individual frequency responses:

$$H_{N_1,N_2}(e^{j\omega_1}, e^{j\omega_2}) = H_{N_1}(e^{j\omega_1}, e^{j\omega_2}) \cdot H_{N_2}(e^{j\omega_1}, e^{j\omega_2}) \tag{2.20}$$

Knowing that $H_N(e^{j\omega_1}, e^{j\omega_2}) = H_N(e^{j\omega})$ and the definition of $H_{N_1}(e^{j\omega_1})$ from Equation 2.8, the frequency response of the bilinear interpolator becomes:

$$\begin{aligned} H_{N_1,N_2}(e^{j\omega_1}, e^{j\omega_2}) \quad = \quad & \left[1 + 2\sum_{k=1}^{N_1-1}\left(1 - \frac{k}{N_1}\right)\cos k\omega_1\right] \cdot \\ & \left[1 + 2\sum_{l=1}^{N_2-1}\left(1 - \frac{l}{N_2}\right)\cos l\omega_2\right] \end{aligned} \tag{2.21}$$

This is plotted for $N_1 = N_2 = 5$ in Figure 2.11.

From Section 2.2.1 it is known that the 1-D linear interpolator has a lowpass frequency response. Since the bilinear interpolating filter is a separable filter consisting of two 1-D linear interpolators, it is reasonable to expect $H_{N_1,N_2}(e^{j\omega_1}, e^{j\omega_2})$ to exhibit lowpass behavior as well. As can be seen from Figure 2.11, the bilinear interpolator is indeed a 2-D lowpass filter. With an increase in N_1, the bandwidth in the ω_1 direction decreases. Similarly, an increase in N_2 causes a decrease of the bandwidth in the ω_2 direction.

The lowpass frequency response of the filter is crucial in the removal of unwanted spectral images caused by the 2-D up-sampling of a compressed image. Similar to the 1-D case (see Section 2.2.1.1) the up-sampling of image data along each axis

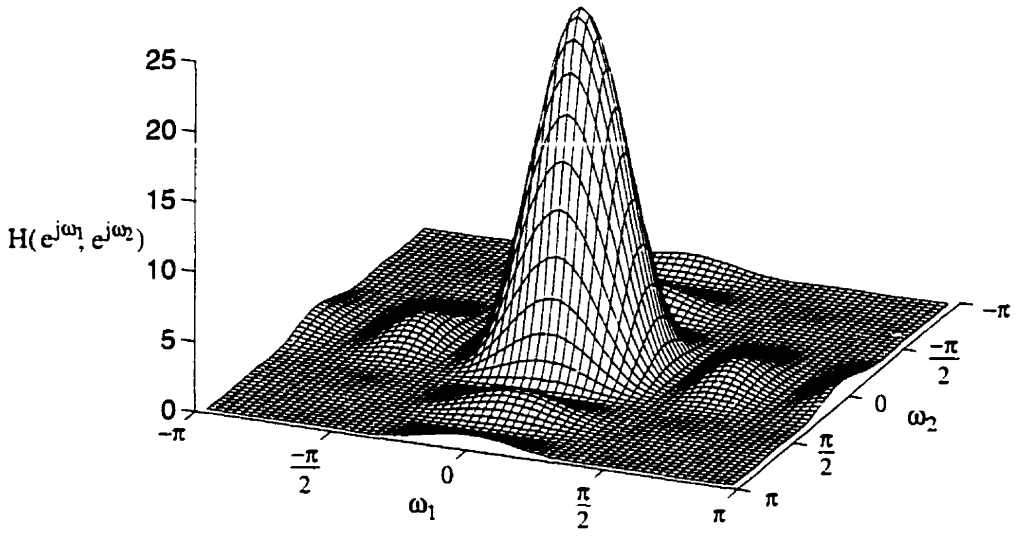


Figure 2.11: Magnitude frequency response of $h_{5,5}(n_1, n_2)$.

results in a frequency spectrum composed of $(N_1 - 1) \times (N_2 - 1)$ replicas of the original spectrum compressed by a factors N_1 along the ω_1 axis and by N_2 along the ω_2 axis. The bilinear interpolating filter, $H_{N_1,N_2}(e^{j\omega_1}, e^{j\omega_2})$, with its lowpass response removes the unwanted frequency replicas and retains the replica centred around the origin.

2.4 Bilinear Interpolation of Sub-Sampled Image Data

In order to demonstrate how image data can be compressed and reconstructed using bilinear interpolation, an image is first sub-sampled along a regular grid. This reduces the number of pixels in the entire image dramatically. Sub-sampling, in both directions, by a factor of N will reduce the information requirement by an order of

Figure 2.12: Original 8 bits/pixel 257×257 image of Lena.

N^2. Then by DPCM coding followed by entropy coding the remaining pixel values, the image data can be compressed further.

For example, in Figure 2.12 is a 257×257 image of Lena sampled at 8 bits/pixel. Sub-sampling this image by a factor of 2 in each direction removes approximately 3/4 of the pixels. The remaining samples in their proper locations that will be used by the decoder are shown in Figure 2.13. The sub-sampling operation immediately reduces the number of pixels from 66049 in the original image to 16641 pixels in the sub-sampled version. So the sub-sampling alone is able to code the image at about 2 bits/pixel. Then by using a DPCM-Huffman coder, the data requirements can be reduced further to 1.53 bits/pixel. Now, in order to see the effects of this data compression, the image must be decompressed and compared to the original. The decompression stage is heavily dependent on the bilinear interpolator to fill in the

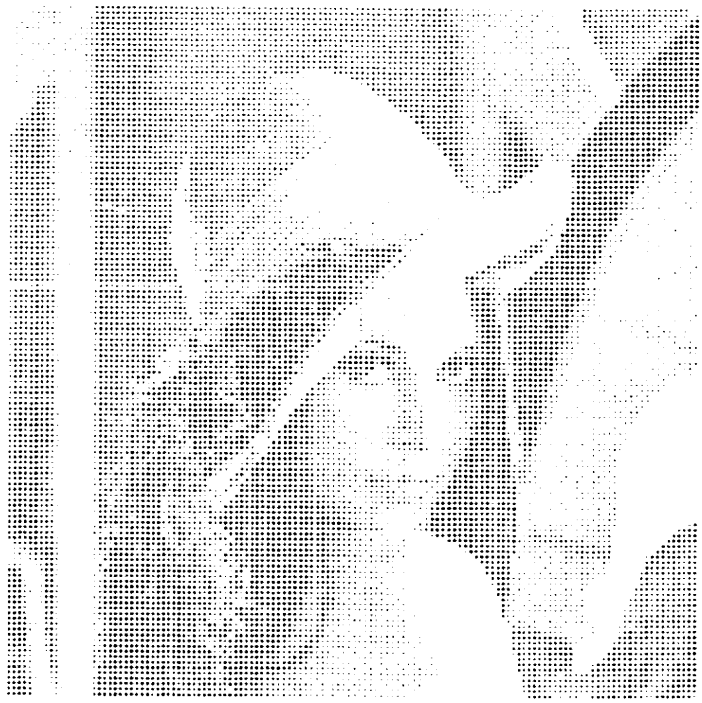


Figure 2.13: Image samples retained for bilinear interpolation.

missing pixel values resulting from the sub-sampling of the original image. First, in the decompression of the image, the image information must be Huffman-DPCM decoded. After that, the bilinear interpolator can be used to generate the reconstructed image. Figure 2.14 shows the bilinear interpolated resulting image which, when compared to the original image of Lena, has a PSNR of 28.1 dB.

2.4.1 Disadvantages of a Fixed Sampling Grid

For image coding, the disadvantage of using a fixed sampling grid, that is, a fixed sub-sampling factor, is due to the frequency content of the image being coded. In order to understand the distortion caused by this image coding method, the sub-sampling of the image data must be studied. Almost all of the distortion in the output image is due to this step—the information removal step. Since the image

Figure 2.14: Bilinear interpolated output image (PSNR = 28.1dB).

manipulation techniques are fully separable, most of the following discussion will deal with the 1-D operations, as they are easier to visualize.

The input/output relationship for the sub-sampling process is:

$$y(n) = x(Mn) \tag{2.22}$$

where the sub-sampling factor, M, is an integer. In the frequency domain, the sub-sampling operation is [19]:

$$Y(e^{j\omega}) = \frac{1}{M} \sum_{k=0}^{M-1} X\left(e^{\frac{j\omega - 2\pi k}{M}}\right) \tag{2.23}$$

Since the sub-sampling is a compression in the time domain, the same operation acts as an expansion (or stretching) in the frequency domain. This is confirmed in Equation 2.23. It shows that the output frequency spectra consists of the summation of shifted and expanded input spectra. This expansion of the frequency spectrum

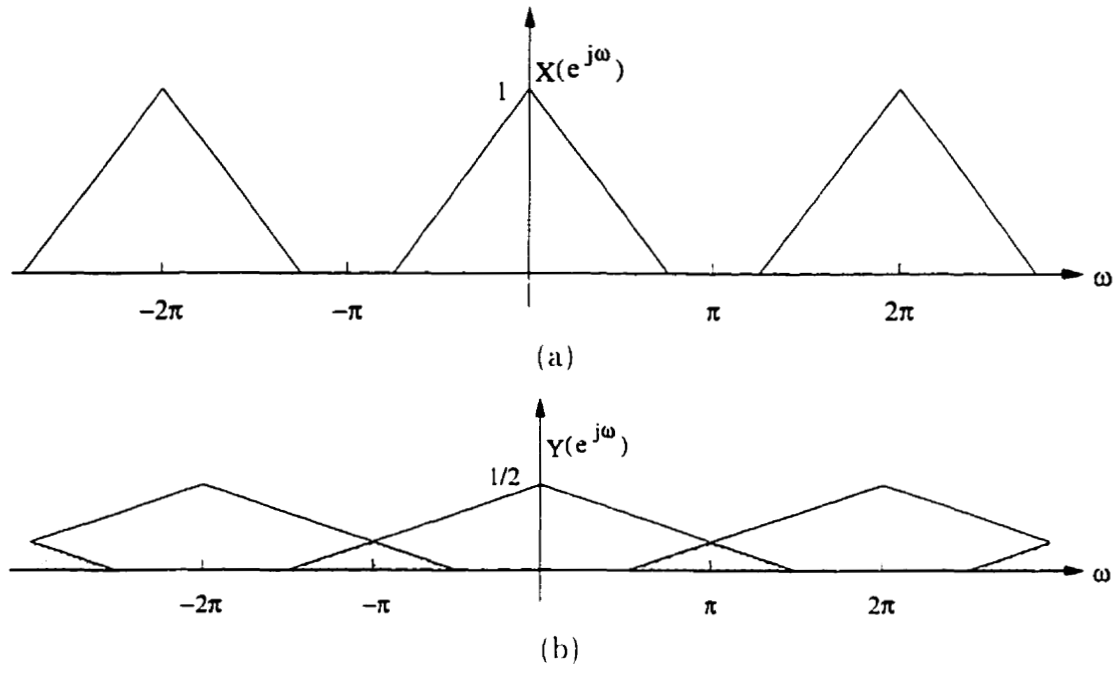


Figure 2.15: Aliasing due to a fixed sub-sampling factor (a) Original frequency spectrum (b) Aliasing due to sub-sampling by a factor of 2.

can cause overlapping in the frequency domain, or aliasing. Aliasing produces visible errors the spatial domain. Figure 2.15(a) shows an initial lowpass signal, $X(e^{j\omega})$. It is then sub-sampled by a factor of 2 producing the signal $Y(e^{j\omega})$ shown in Figure 2.15(b). The shaded overlapping regions indicated where aliasing occurs. Thus, the original signal can never be fully recovered due to the aliasing present.

To avoid aliasing when sub-sampling, the sub-sampling ratio must lie in the range:

$$0 < M \leq \frac{\pi}{\mathrm{BW}} \tag{2.24}$$

where BW is the bandwidth of the signal being sub-sampled and the sub-sampling ratio, M, is an integer. If this range is not satisfied, aliasing will occur and high frequency components will be lost. Since most natural images have a wide range of frequency components, this constraint on M is difficult to satisfy. The main problem

with using a fixed or uniform sampling grid is that the sub-sampling ratio must satisfy the relationship in Equation 2.24 over the entire image. So if, in general, an image consists of mostly low frequencies with a few sharp edges, a low sub-sampling factor must be used over the entire image to properly handle the small number of high frequency components caused by the sharp edges. This results in a low compression ratio for the image because a large number of samples must be retained.

In general, the sub-sampling of an image along a fixed grid causes problems due to the aliasing and loss of high frequency components because it must use a fixed sub-sampling factor over the entire image and thus, local variations in frequency components cannot be exploited. This removal of high frequency components by the fixed sub-sampling ratio results in the visible blurriness and some blockiness of the "Lena" output image, shown in Figure 2.14.

2.5 Adaptive Grid Generation

To avoid aliasing and loss of high frequency information due to sub-sampling the input image, a more general non-uniform sampling grid can be employed. The advantages of using such a grid are two-fold. First, by only removing samples in low frequency areas of the image, no loss of high frequency information occurs, that is sharp edges and textures are retained. The bilinear interpolator can easily fill in the missing low frequency information. Also, by reducing the sub-sampling factor for high frequency areas of the image, errors due to aliasing are reduced or eliminated.

This is the basis of the image coding method presented: a set of points are chosen at the crossings of a grid and then are transmitted or stored. The decoder can then reconstruct the image by bilinear interpolation of these points. The points chosen may belong to a uniform grid, however this does not take into account the

spatial variance in frequency components of the image. As a result, uniform areas will be better approximated than high frequency areas. If a non-uniform grid is chosen, larger grid elements can be used for more uniform areas while smaller grid elements can be used in high frequency portions of the image.

2.5.1 Grid Generation and Representation

A good way of generating and describing the grid of required sample points is with a hierarchical structure, such as a quaternary tree (quadtree) structure. The grid is generated by a recursive technique that examines the interpolation error within a grid element and determines if smaller grid elements—a lower sub-sampling factor—for that area is required. If the interpolation error is high the current block is split into four smaller blocks of approximately equal size. The process repeats until either the error is small enough or a preset minimum block dimensions are reached. This way, if the error tolerance is set to zero and the minimum block dimensions are set to 2 pixels, lossless coding of the image is possible.

The splitting of a "parent" block with the dimensions $(2^N + 1) \times (2^N + 1)$, results in four smaller "child" blocks of size $(2^{N-1} + 1) \times (2^{N-1} + 1)$. When a block is split, five new vertices are created and shared amongst the four new "child" blocks. It is this overlap that allows smooth transitions from one block to another to occur and helps to eliminate unwanted "blockiness" in the output image that other image block-coding methods suffer from.

The resulting 2-D grid, which is represented by the quadtree structure, is easily represented by a string of bits. Figure 2.16 shows a simple grid, the associated quadtree structure, and the string of bits that represent them both. The root tree node represents a block that is the size of the starting image. It is then subdivided

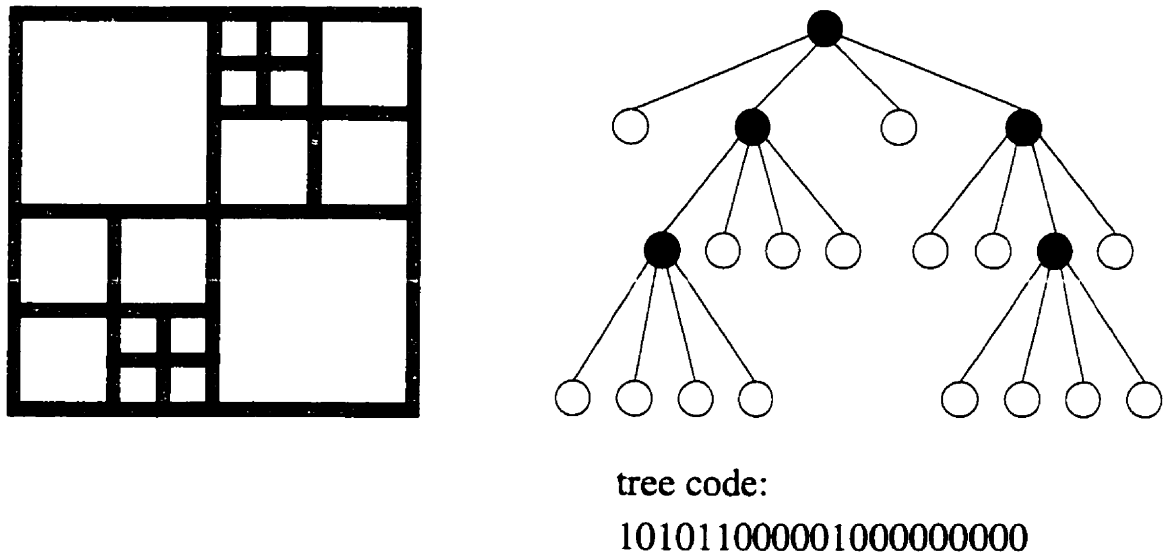


Figure 2.16: Two dimensional non-uniform grid and the representative tree structure [15].

into smaller and smaller blocks as needed. Each node or leaf (nodes with no children) in the tree is described by a single bit. A "1" indicates a split while a "0" indicates no split. However, the tree leaves that are of the minimum block size require no bits to represent them because they are never split. The decoder then must be informed of the minimum block size in order to properly reconstruct the quadtree.

When lossless coding of the image is the goal and the above splitting process is used there will be limitations on the dimensions of the original image. This is due to the inflexibility of the quad-splitting process when a minimum block dimension is reached. That is, when a block that has one side that is less than or equal to the minimum block dimension in length does not meet the error criterion it cannot be split into four because of the minimum block dimension limitation. Even if the minimum block dimension is set to 2 pixels, if a block size is $(2 \times K)$ where $K > 2$ it cannot be split further and brought down to the pixel level. The interpolation error then may not be zero and lossless coding may not possible. Even if the original

image has an aspect ratio (width to height) of 1:1 (i.e. it is square), this problem still will occur unless the image has the specific dimensions of $(2^N + 1) \times (2^N + 1)$. This happens because, although odd dimensions can be split exactly into equal parts, odd sized blocks may be split into even sizes which cannot be split again into equal overlapping parts. Only dimensions of $2^N + 1$ can be recursively split down to the pixel level—down to block sizes of 2×2. For example, the four blocks created from splitting a 23×23 block will all have sizes of 12×12. Recursive splitting of this block will then result in blocks of 3×2, 2×3, and 2×2 in size. Since interpolation must be done on the 3×2 and 2×3 sized blocks, some interpolation error may occur and lossless coding may not be possible. As the aspect ratio of the original image varies from 1:1, this effect will be magnified. Since the input image size is rarely $(2^N + 1) \times (2^N + 1)$, lossless coding cannot be guaranteed by setting the error tolerance to zero and having a minimum block dimension of 2.

This method is best used as a lossy image coder, so perfect reconstruction is not usually a requirement. The unsplittable blocks are usually quite small in one direction, depending on the minimum block dimension, and so, the error introduced by this is quite small. If lossless coding is important, a small modification can be made to the tree structure, so there are no limitations to the image dimensions or aspect ratio. However, this modification tends to have a negative impact on the compression performance. In Section 3.2, this splitting method modification is discussed in greater detail.

Chapter 3

Implementation of and Improvements to the Adaptive Interpolation Image Codec

In this chapter, some of the specific implementation issues concerning the adaptive interpolation image codec system will be discussed as well as some of the possible choices made during implementation. Also some extensions and improvements to the basic bilinear interpolation image coder will be given. Section 3.1 describes various algorithms for implementing the bilinear interpolator along with their advantages and disadvantages. Then, some specifics are given on the sub-sampling grid generation and its representation in Section 3.2. Section 3.3 then shows how the vertex values can be stored by using an 8-bit DPCM coding algorithm along with scalar quantization. After that, Section 3.4 contains a discussion into how the bilinear interpolation reconstruction error can be reduced without increasing the information requirements by using a method of least-squares. Furthermore, a way of optimizing the calculation and storage requirements of the least-squares method is also shown. Finally, some

results, including input/output examples, are discussed in Section 3.5.

3.1 Bilinear Interpolation

In Section 2.3 the bilinear interpolation function was identified along with the bilinear interpolating filter. In a practical image coding application the use of the 2-D filter is not feasible. The calculation of the filter coefficients would be extremely time consuming and even if calculated off-line, the storage requirements would be very large to take into account all possible block sizes. Also, the filter operation would require the use of large amounts of floating-point computations for the interpolation process. The introduction of the bilinear interpolating filter was only used to examine the frequency characteristics of bilinear interpolation, which was accomplished in Section 2.3.1.

The preferred method of bilinear interpolation in this image coding application is the two step method shown in Figure 2.8 which involves multiple one dimensional linear interpolations. It is equivalent to the one-step method shown in Equation 2.13, however it is less complex and requires fewer computations per pixel. The bilinear interpolation of a $(N_1 \times N_2)$ sized block requires at least $N_1 N_2 - 2$ linear interpolated values. The 1-D linear interpolation described in Equation 2.1 can be calculated directly for each pixel. This involves 2 additions, 1 multiplication, and 1 division and can be done by using either fixed- or floating-point arithmetic.

It is well known that divisions and multiplications, whether they are performed in floating- or fixed-point, are expensive either in time consumption or hardware complexity. By using an incremental algorithm, the need for multiplications can be eliminated and the number of divisions can be drastically reduced. This can be done

by noting that the linear interpolated value at $i+1$ is

$$\begin{aligned} Z_{i+1} &= m(i+1) + Z_0 = mi + Z_0 + m \\ &= Z_i + m \end{aligned} \tag{3.1}$$

where $m = \frac{Z_N - Z_0}{N}$. Thus, the value of the next pixel can be calculated by adding m to the current pixel value. So multiplication is not required at all, and only one divide for the entire row or column is needed—in order to calculate the constant difference m. Then, for each sample starting with Z_0, the next interpolated value is equal to the sum of the current value and the constant difference. This algorithm is often referred to as a Digital Differential Analyzer (DDA) [20]. Accuracy is important in this incremental algorithm because any error in the value of m will be amplified with each successive addition. So an accurate integer representation of m is not usually possible.

Before going further, a brief review of number formats might be required. There are many formats for representing numerical values—three of which are of use in this section. They are: floating-point, fixed-point, and integer number formats. Shown in Figure 3.1 are some various implementations of these number formats using a 32-bit word. The floating-point format can represent a wide range of values because it makes use of an exponent. However, because of the added exponent field and sign bit, operations on floating-point values are quite complex and require specialized hardware and/or additional time to compute. Conversely, the plain integer format is very simple and easy to perform computations on—all microprocessors support direct operations on this type. The main drawback is the smaller range of values that can be represented. Furthermore, the integer format cannot represent fractional values. In between the two number formats is the fixed-point format. In fact an integer is actually a specific fixed-point representation with no fractional part. A

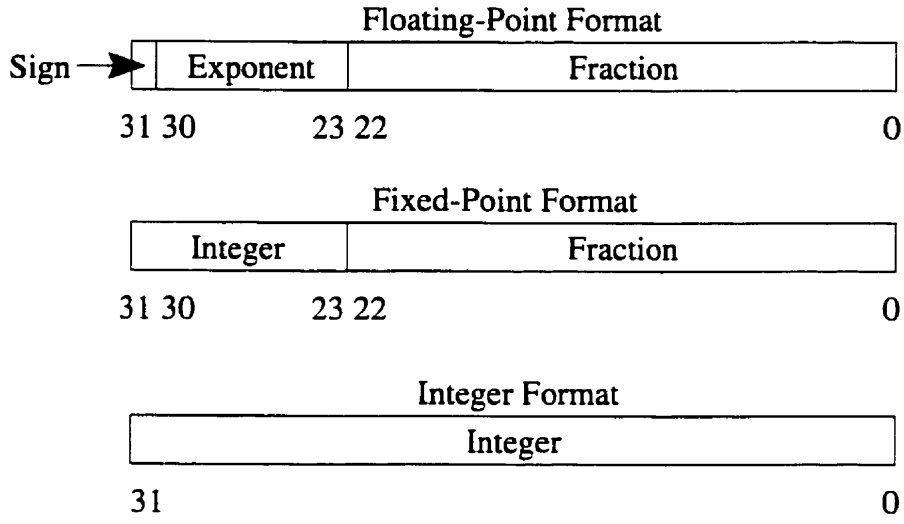


Figure 3.1: Formats of various number representations.

fixed-point format is assigned a pre-determined number of bits to the integer and fraction parts of a number. It has less range than an integer format because it has fewer bits to represent the integer part. However, it can represent fractional numbers. Furthermore, operations on fixed-point numbers can be performed almost as easily as integer formatted numbers. In fact, fixed-point numbers can be added together using integer operations and can be multiplied together using an integer multiplier combined with bit shifts. Actually, bit shifts are not required if multiplying an integer with a fixed-point value.

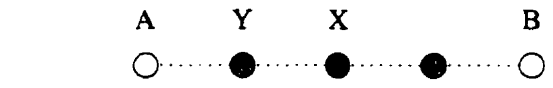
Above, in the description of the DDA, the constant value m is used to increment the pixel intensity across a row or column of pixels. Since m is calculated from the fraction $\frac{Z_N - Z_0}{N}$, an exact integer format representation is not generally possible. Using a floating-point value is the most flexible representation.

Implementation of a fixed-point DDA is possible and thus integer calculations can be used which should be somewhat faster than floating-point calculations. The fixed-point DDA uses fixed-point arithmetic to compute floating-point values. It

assigns a pre-determined number of bits to the integer part and to the fraction part of a number. To convert from an integer to a fixed-point value is done simply by multiplying the integer by 2^F, where F is the number of fraction bits. Conversion from a fixed-point value to an integer is then done by adding 2^{F-1} and dividing by 2^F. The multiplication and division by 2^F is actually performed by bit shifting the value F places to the left for multiplication or to the right for division. For a balance between precision and speed, the fixed-point DDA used hereafter uses 23 bits for the fraction and 9 bits for the integer. This way, values from -256 to 255 can be represented and can be computed using 32-bit integer arithmetic.

It is possible to completely eliminate divisions and multiplications by using Bresenham's incremental line drawing algorithm [20]. This method requires no multiplications or divisions and can be performed with integer arithmetic. Instead of calculating pixel locations between two arbitrary 2-D end-points, the Bresenham algorithm is used to calculate intensity values between two 1-D end-point values. As a line drawing method, the Bresenham algorithm steps along the major axis (the axis with the largest change) and for each step a decision is made whether or not to take a step along the minor axis. When used as a linear interpolator and conditions make the spatial dimension the minor axis, the method can be somewhat inefficient. This is because multiple intensity steps must be made before a spatial step can be performed. This makes the speed performance of the Bresenham algorithm very dependent on the end-point values. Another drawback to the Bresenham method is that a few extra calculations are required during setup. Furthermore, the decision to increment along the minor axis when stepping along the major axis requires a jump which can negatively impact the speed performance of the algorithm if implemented on a microprocessor.

Another bilinear interpolation method which does not require multiplications

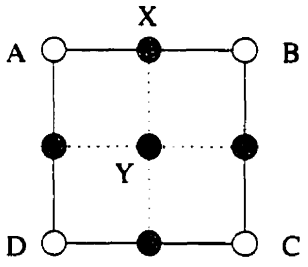


○ Starting vertices (A,B)

● Primary interpolated mid-point (e.g.: $X = \frac{A+B}{2}$)

● Secondary interpolated mid-point (e.g.: $Y = \frac{A+X}{2}$)

(a)



○ Starting vertices (A,B,C,D)

● Interpolated mid-points (e.g.: $X = \frac{A+B}{2}$)

● Interpolated central point (e.g.: $Y = \frac{A+B+C+D}{4}$)

(b)

Figure 3.2: Recursive interpolation: (a) 1-D linear interpolation (b) 2-D bilinear interpolation.

or divisions is based on a recursive interpolation implementation (derived from the recursive trilinear interpolation method introduced in [17]). This method calculates the midpoint pixel value by averaging the end-point pixel values together. This involves only an integer addition and a bit shift (which is a division by two). Then the midpoint is used as end-point value for the calculation of two new midpoint values lying on each side of the old midpoint. This process is repeated until all pixel values are filled in. The 1-D recursive linear interpolation method is illustrated in Figure 3.2(a). The value at point X is calculated from the end-points A and B. Then

the secondary mid-points, such as point Y, are calculated in the same manner (except point Y is calculated using the end-points A and X). Similar to the other linear interpolators, this recursive method can be used to implement a two-step bilinear interpolator by first interpolating in one direction followed by interpolations in the second direction.

It is possible to extend this recursive method to implement a one-step 2-D bilinear interpolator. This is shown in Figure 3.2(b). Using the four starting corner points, the interior midpoints and centre point can be calculated using additions and bit shifts. Then the interior midpoints and centre points of the interior blocks can be calculated all using simple integer arithmetic.

The advantage of using floating-point operations is the accuracy obtained by their use. When using integer division, the result is rounded, and in the two part bilinear interpolation, that will introduce an unwanted quantization step in between the two operations. Over large block sizes, the quantization effect is noticeable because the bilinear interpolated transition between the four corner points is not as smooth as when floating-point calculations are used throughout. Table 3.1 summarizes the Mean Squared Error (MSE) between the output of different bilinear interpolation methods and the actual one-step 2-D bilinear interpolator of Equation 2.13. The mean squared errors are averaged over 100 512×512 blocks with random corner values. Since the output of the various bilinear interpolators consists of 8-bit values (in order for a display device to properly show the pixels), another MSE measurement is needed: a comparison between the 8-bit quantized output of Equation 2.13 and the output of the other bilinear interpolators. This MSE measurement is also included in Table 3.1. From the table, the most accurate methods are the ones which employ floating-point operations and result in no error at all while the least accurate of the methods is the integer divide method. The precision of the fixed-point DDA calcula-

Table 3.1: Comparison between the average mean squared error of different bilinear interpolation methods over 100 random 512×512 blocks.

Interpolation Method	Average MSE	
	Method vs. Actual	Method vs. Quantized Actual
Floating Point Divide	0.083	0.0
Floating Point DDA	0.083	0.0
Fixed Point DDA	0.083	1.9×10^{-5}
Bresenham	0.14	0.20
Integer Divide	0.52	0.66
Recursive 2-D[a]	3.8	3.9
Recursive 1-D	5.3	5.3

[a]One-step interpolation

tions are almost equal to that of floating-point calculations. This is possible because the quantization that would normally happen between the first and second step of the bilinear interpolation process when using integer values, can be removed. The results of the first linear interpolation step can be temporarily stored using the fixed-point DDA output values, which are then used as input for the next interpolation step. It should be noted that the larger the block to be interpolated, the less accurate the methods tend to be. In practice though, block sizes as large as 512×512 are never used so the errors shown in Table 3.1 are more representative of a worst case scenario.

Based on the error performance of the bilinear interpolators, the floating-point methods work best, followed closely by the fixed-point DDA method. On average, when used in this image coder, interpolation error tends not to affect the image coding performance because the coder adapts to the interpolation error during the splitting process. If the coder meets the required error criterion by recursive splitting, the decoder will meet the error criterion as well. However, the performance of the overall coder/decoder system will be affected if the same interpolator is not used in both the coder and the coder. Operations on floating-point values are usually very complex

and can be time consuming, so it is desirable to avoid relying on them.

Another point to address is the order of operations in computing the two-step bilinear interpolation. As stated previously, the order of the two-step bilinear interpolation, whether it is rows or columns first, makes no difference in the output. However, from a hardware perspective, the speed of operation may be affected if a microprocessor with a data cache is employed. For example, if the pixels in each row of the image are stored in consecutive memory addresses linear interpolation along the rows will have better cache performance than column interpolation. The left and right columns should be interpolated first, followed by the row-wise interpolation. This will improve performance assuming there are more than two rows in the block (i.e. the number of row interpolations is greater than the number of column interpolations).

To illustrate the processing times required for the various interpolation algorithms, different sized blocks were bilinear interpolated using the floating-point division, floating-point DDA, fixed-point division, Bresenham, and the fixed-point DDA methods. Since each size of block contains different numbers of pixels, smaller blocks need to be interpolated more than once to keep the number of interpolated pixels constant. For each block size, 26214400 pixel values were interpolated. Table 3.2 shows the timing results results for 100 512×512 blocks, 6400 64×64 blocks, and 1638400 4×4 blocks. The processing times are based on a 166MHz Intel Pentium processor. From the table, it can be seen that the set up time for each method becomes more critical for smaller block sizes. The Bresenham method suffers severely due to its more complex set up requirements. The recursive methods need virtually no set up and so they perform well on small blocks while performing quite slowly on larger ones due to the large amount of recursion required. In this image codec, smaller blocks are more common than large ones, so the 4×4 block timings are the most important.

Table 3.2: Processing times of different block sizes (containing the same number of pixels) for various bilinear interpolation methods.

Interpolation Method	Time (seconds)		
	100 blocks (512×512)	6400 blocks (64×64)	1638400 blocks (4×4)
Actual (Equation 2.13)	29.49	29.24	25.99
Bresenham	8.94	13.33	191.15
Floating Point Divide	13.45	13.62	16.48
Integer Divide	10.31	10.62	15.94
Floating Point DDA	4.76	4.89	8.58
Recursive 1-D	11.22	10.87	8.55
Recursive 2-D	13.22	12.30	6.82
Fixed Point DDA	1.21	1.38	5.70

It is clear that the fixed-point DDA method requires the least amount of time to compute. Because of this and its low bilinear interpolation error, the fixed-point DDA interpolator becomes the best choice for implementing this image codec.

3.2 Grid Generation and Representation

3.2.1 Minimum and Maximum Block Dimensions

During the encoding process the decision whether or not to split a block is based on the block interpolation error, and the minimum and maximum block dimensions. These dimensions have to be carefully chosen based on compression and speed considerations. The maximum block dimension, D_{max}, is the starting block size of the splitting process. All blocks larger than D_{max} are unconditionally split and so interpolation and MSE calculations can be saved. The minimum block size, D_{min}, governs the maximum resolution of the reconstructed image. Usually, it is not useful to reach the pixel resolution (i.e. in order to have lossless image coding when

the interpolation error threshold is set to zero), for the data reducing performance of the interpolators becomes non-existent, making other lossless image compression techniques more attractive. However, it is sometimes useful to have a single algorithm provide lossless and lossy coding. Typical values for a medium-resolution natural image (512×512 for example) are $D_{max} = 32 \times 32$ and $D_{min} = 3 \times 3$. Choosing a larger starting size is generally useless, because almost all blocks of this size will need to be split, thus notably increasing the computational load [14].

3.2.2 Tree Structure

In Section 2.5.1 it was stated that lossless coding was only possible for images that have specific dimensions of $(2^N + 1) \times (2^N + 1)$. There is a couple of ways that this problem can be remedied if lossless image coding is required. However, these solutions do result in slightly larger output file sizes.

The first solution is to change the structure of the tree to a quaternary-binary tree. This would enable the image coder to split in half—horizontally or vertically—or to split into quarters. The representation of this type of tree would be more complex, as more than one bit would be required for each type of split. One type of representation could have "0" chosen for no split (a leaf node), "11" for a quad-split, "100" for a horizontal split, and "101" for a vertical split. With this structure, it is possible to split previously unsplittable blocks when one dimension of the block is too small to be split. Lossless image coding becomes possible since splitting can be performed down to the pixel level. Even if lossless coding is not a requirement, the tree may be optimized to produce less error as it may be possible that a horizontal or vertical split may be more advantageous than a quad-split (split into four). The disadvantage to this solution is that the tree code becomes more complex and harder

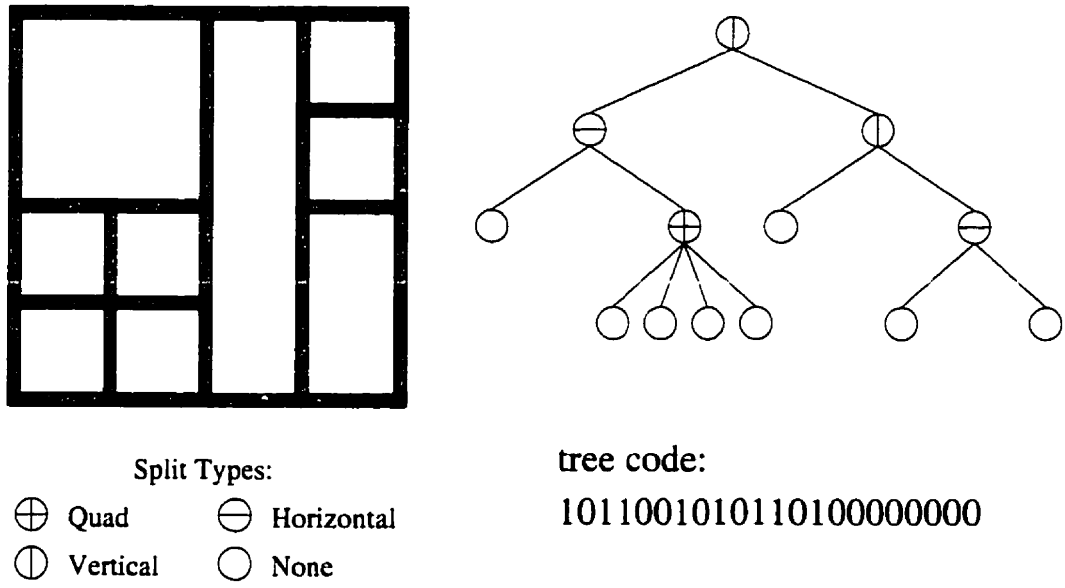


Figure 3.3: Quaternary-Binary tree structure.

to compress. In Figure 3.3, an example of a quaternary-binary tree is shown along with the resulting splitting diagram and bit string.

The second solution would have the same structure as described in Section 2.5.1, except the minimum block dimension parameter would be used differently. Blocks that can no longer be quad-split due to one side being less than or equal to the minimum block dimension still would have the option to split horizontally or vertically depending on which side still can be split. So a Quaternary-Binary Tree would be used but its representation still would require only 1 bit per node. Similar to the tree representation described in Section 2.5.1, the bits representing leaves that have both sides smaller than the minimum dimension can be omitted from storage assuming the decoder is informed of the minimum block size so that the tree can be properly reconstructed.

3.2.3 Compression of Tree Information

The quadtree (or quad-binary tree) structure is represented by a string of bits that indicate the type of node. It can be seen in the bit string representing the tree a (see Figure 2.16 for example) there are large numbers of repeated patterns and bits. This indicates that the tree code data can be compressed further by the use of a lossless data compression algorithm. The algorithm must be lossless as the image decoding depends on an uncorrupted tree in order to generate the sampling grid. Due to the small alphabet size of the bit string (2 symbols), an entropy coder would not perform well. So a data compression algorithm that uses repeated patterns in the string to reduce the data consumption would be more appropriate. The Lempel-Ziv Welch (LZW) compression algorithm uses a dictionary based approach for taking advantage of repetitive patterns within a data stream [18]. This is the lossless data compression algorithm that will be used for compressing the bit string which describes the quadtree. It will be shown that the average number of bits per tree node can be reduced to below 1 bits/node.

3.2.4 Interpolation Discontinuities

When the bilinear interpolation has to be performed on the global tree structure (for reconstruction of the image) some interpolation discontinuities may arise. This is due to the fact that a tree structure may arise in which some vertex locations will not match those of the neighboring blocks. Figure 3.4 is an example of such a case. It can be seen that an interpolation discontinuity will happen where point X lies. If the shaded block is interpolated only by vertices A, B, C, and D, an artifact may appear in the neighboring block, due to the presence of point X.

One solution is to accept small interpolation discontinuities and save on compu-

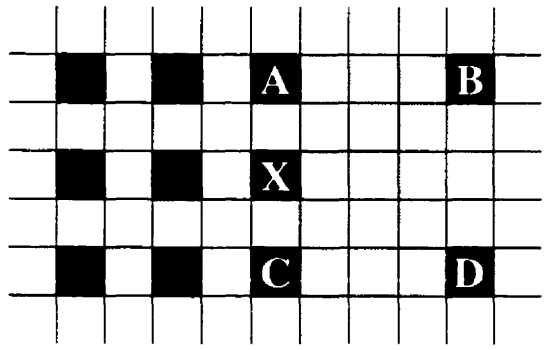


Figure 3.4: Example of a vertex orientation leading to interpolation discontinuities [14].

tation. The effects of the discontinuities can be reduced by the way the splitting tree is traversed when performing the bilinear interpolations. A top-down (or breadth-first) traversal is the best choice when compared to a recursive (or depth-first) traversal. The top-down approach will interpolate, in order, each size of block starting with the largest sized block. That way, in Figure 3.4, the shaded block would be interpolated first and then when interpolating the smaller blocks the edge between pixels A and D would be replaced by the values interpolated in the smaller blocks that contain pixel X. Some pixel values will be calculated twice, however a separate tree traversal is not necessary and the coder and decoder can be better matched.

An alternative solution via block size equalization (BSE) has been suggested in [14]. The discontinuity problem can be solved by applying a top-down iterative algorithm in the reconstruction phase: for each level of the tree, the midpoint pixel on each of the four block sides and the central point of each block are calculated, if they are not present, on the basis of the four block corners. The newly calculated pixels then serve as new block corners that are common with all neighboring blocks. This recursion can be performed so that all leaves in the splitting tree are on the same level (i.e. all the blocks have the same size). Bilinear interpolation is then performed

over all these blocks so that interpolation continuity is maintained. Although this does remove all interpolation discontinuities, the newly calculated 8-bit corner points, may be slightly quantized from a true midpoint calculation and these errors will propagate over the entire block. As stated in Section 3.1, small errors in bilinear interpolation are not a significant problem if the coder and decoder are matched. This solution, however, does cause a slight mismatch in the coder and decoder. It would be possible for the coder to implement this as well, however it would result in a larger computational load so it is most efficient to implement on the decoder only.

The resulting output images produced by block size equalization are of a higher visual quality and, on average have an overall lower MSE (see Section 3.5). The only disadvantage is that the decoding phase can be slowed dramatically if the block sizes are equalized down to the pixel level. It should also be noted that the decoder should not split beyond the minimum block dimensions used by the image coder as this usually results in a higher MSE and a greater computational load.

3.3 Vertex Representation

The samples at the grid vertices are required by the decoder as corner values for the bilinear interpolation operation. As mentioned in Section 2.4, the data required by the vertices can be reduced by using DPCM followed by a Huffman coder. Any correlation that exists between adjacent sample values is taken advantage of by the DPCM process. In DPCM, the error between the current sample and the value predicted by a weighted sum of previous samples is encoded. The error signal usually has a smaller dynamic range than the original signal and also has a probability distribution function with a smaller variance [1]. This decrease in the variance results in a corresponding decrease in the entropy of the signal. The Huffman entropy coder that

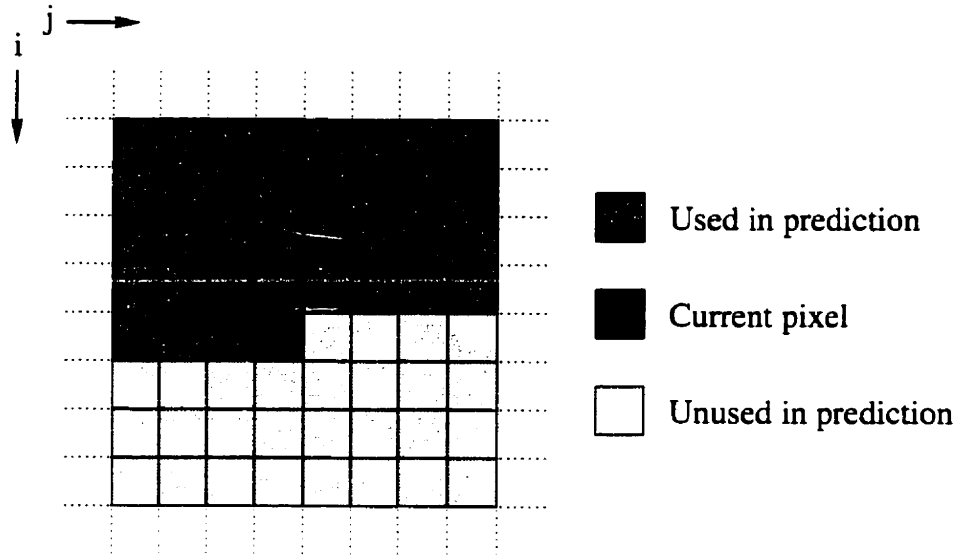


Figure 3.5: Prediction of $\hat{x}(i,j)$ on a uniform sampling grid.

follows, takes advantage of this lower entropy to encode the DPCM output using, on average, fewer bits per sample than the original vertices. The effectiveness of the DPCM process is dependent on the accuracy of the predictor. The more accurate the prediction process is, the lower the entropy of the error signal becomes.

Normally, when performing DPCM coding on a uniform sampling grid, a 2-D predictor will use previous samples from above the current sample and samples directly to the left of the current sample (see Figure 3.5)—assuming that the coding of the image is done from top to bottom and left to right. No other sample values can be used for the prediction because the decoder will only have access to previously decoded sample values. The predicted value, $\hat{x}(i,j)$, of the pixel $x(i,j)$ is calculated using the equation

$$\hat{x}(i,j) = \sum_{n=1}^{N} c_{0,n}\, x(i,j-n) + \sum_{m=1}^{N} \sum_{n=-N}^{N} c_{m,n}\, x(i-m,j-n) \qquad (3.2)$$

where $c_{m,n}$ are the prediction coefficients and N is the order of the predictor. Usually the $c_{m,n}$ coefficients are chosen so that $\sum c_{m,n} = 1$. In this image coding application,

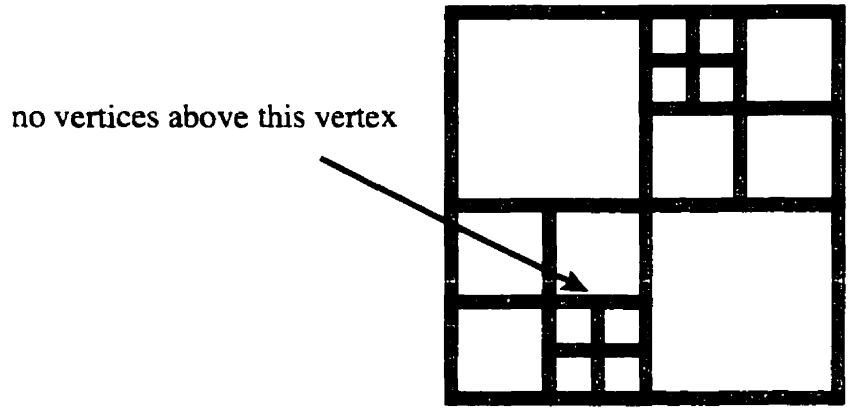


Figure 3.6: Vertex with no above neighboring vertices.

only the immediate samples surrounding the predicted sample are used, so $N = 1$. This simplifies Equation 3.2, which results in

$$\begin{aligned} \hat{x}(i,j) &= c_{0,1}x(i,j-1) + c_{1,-1}x(i-1,j+1) + \\ &\quad c_{1,0}x(i-1,j) + c_{1,1}x(i-1,j-1) \end{aligned} \tag{3.3}$$

A reasonable first order predictor will simply set $c_{0,1} = c_{1,-1} = c_{1,0} = c_{1,1} = \frac{1}{4}$, so that the predicted value is simply the average of the four previous surrounding values.

When DPCM coding values on a non-uniform grid, there may not always be vertices above and/or to the left of the current vertex, and the inter-sample spacing can vary as well. This is apparent in Figure 3.6 where the specified vertex has no neighboring vertex directly above it. To take into account the possibility of unavailable and varying distances between vertices, the prediction process is modified. Varying distances between vertices is handled by searching horizontally or vertically for the nearest vertex. A horizontal search is performed to find the closest vertex to the left while for the $x(i-1,j+1)$, $x(i-1,j)$, and $x(i-1,j-1)$ values required in Equation 3.3, vertical searching is performed. Since the prediction coefficients are all equal and the prediction is simply the average of the surrounding vertex values, having unavailable vertices is not too much of a problem. If a vertex is not available,

it is simply removed from the calculation of the average.

After the DPCM coding of the vertex values is performed, a Huffman entropy coder replaces the DPCM difference values with variable length codes. The Huffman codes used to represent each DPCM value are based on the probability of that value occurring within the data stream so, on average, the number of bits per vertex is reduced.

To further reduce the information required by the grid vertex values, the values can be quantized before DPCM coding [21]. That is, divide each value by some constant quantization factor, Q, and rounding the result. This reduces the number of possible values which in turn reduces the entropy of the data. During the reconstruction stage, the decoder simply multiplies the vertex values by Q to reverse the scaling performed by the coder—with some loss of precision.

An example of vertex quantization can be shown: starting with a vector containing the vertex values

$$\mathbf{v} = \begin{bmatrix} 123 & 8 & 125 & 121 & 3 & 7 & 100 & 255 & 237 & 232 \end{bmatrix}$$

where all the values are 8-bit integers, if the coder divides this vector by the constant $Q = 8$ and rounds each number to an integer value, the resulting vector will be

$$\mathbf{v}_Q = \begin{bmatrix} 15 & 1 & 15 & 15 & 0 & 0 & 12 & 31 & 29 & 29 \end{bmatrix}$$

As can be seen, the dynamic range of the values has been reduced which also results in some repeated values. Now, these numbers can be represented by 5-bit integers. Furthermore, the increased occurrence of certain values will make data compression easier when using the Huffman entropy coder. By increasing Q, the entropy can be decreased further, which will result in less data required for the vertex values after DPCM-Huffman coding. In the decoder, to obtain an approximation of the original

vertex values, multiplication by $Q = 8$ is performed that results in the vector

$$\tilde{\mathbf{v}} = \begin{bmatrix} 120 & 8 & 120 & 120 & 0 & 0 & 96 & 248 & 232 & 232 \end{bmatrix}$$

From this example, it can be seen that the reconstructed vector, $\tilde{\mathbf{v}}$, is similar to—but is not an exact replica of—the original: there is some quantization error present.

The larger Q is, the higher the quantization error will be. However, the quantization of the vertex values does not dramatically affect the reconstructed image quality since the interpolation is performed using 256 levels (for 8 bits/pixel) and re-creates intermediate values, thus avoiding visual artifacts such as the "onion ring" effect usually caused by quantization.

3.3.1 Improvement in Huffman Encoding of DPCM Data

DPCM coding itself does not reduce the bit rate of a data stream. In fact, the bits per symbol must actually increase when DPCM coding is performed. This is due to the increase in dynamic range that the differential operation produces. For example, when coding 8-bit unsigned data, such as pixel values, the original symbols can represent values from 0 to 255. When calculating the difference between any two of these 8-bit values, the new range of possible values is -255 to 255 and thus requires a signed 9-bit value to be properly represented. This extra bit is required to carry the sign information.

Traditionally, a 9-bit entropy coder is used in order to compress the 9-bit DPCM coded data. However, using unsigned 8-bit arithmetic it is possible to output 8-bit data from the DPCM coder while still ensuring proper reconstruction in the decoder [22]. The resulting 8-bit data will possibly have less entropy than a 9-bit data stream created from the same original signal or in the worst case the entropy will be the same. Furthermore, the entropy coder required for the 8-bit data stream

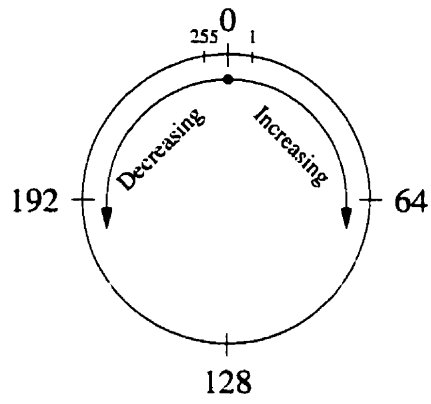
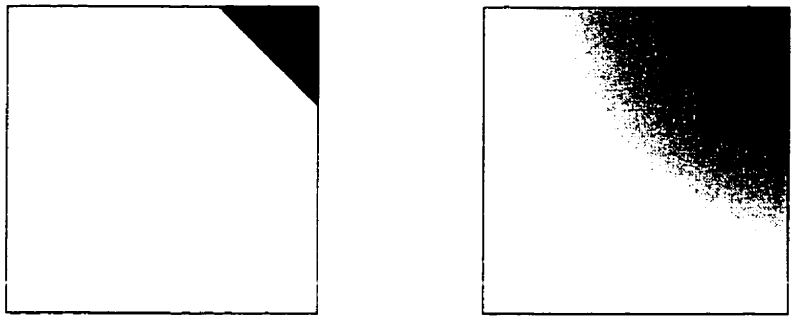


Figure 3.7: Unsigned 8-bit number circle.

will have to contend with less data throughput, use less memory, and be slightly less complex than a 9-bit entropy encoder.

By using the wrap-around nature of unsigned 8-bit arithmetic shown in Figure 3.7, an extra bit is not required to convey the sign information in the DPCM data. Thus, the addition of a negative number, $-x$, (i.e. subtraction by x) is equivalent to the addition of $2^N - x$ in an N-bit system. For example, if the current value being DPCM coded is 50 and given that the previous sample value is 65, the difference between the two is -15. The corresponding 8-bit DPCM output is 241. When decoding the value, the addition of 65 from the previous sample and 241 from the DPCM output will be 306 and will cause a wrap-around in the unsigned 8-bit number system resulting in a value of 50 which is the original value before DPCM coding.

Performing the unsigned 8-bit arithmetic is very easy when using a microprocessor that uses two's compliment representation for negative values. When calculating the difference between two 8-bit values, simply remove the top bit of the resulting 9-bit output value. This will be the 8-bit DPCM output value. Decoding is performed in exactly the same way: addition and truncation to 8-bits.



(a) (b)

Figure 3.8: Interpolation of a block crossed by a high contrast edge: (a) original block, (b) corresponding block after being reconstructed via bilinear interpolation [15].

3.4 Least-Squares Bilinear Interpolation (LSBI)

The major drawback to the bilinear interpolation discussed previously is the fact that values lying in between and within the grid points are not at all taken into account. In the presence of noise this may cause problems because the error related to a corner point chosen for the interpolation affects all the neighboring blocks. Furthermore, in the presence of high contrast edges, the value at one vertex of a uniform block may be entirely different from the other vertex values. This is illustrated in Figure 3.8 where the upper-right vertex belongs to a different object in the image. When the image is reconstructed via bilinear interpolation the contrasting corner value is spread across the entire block.

The above problem can be alleviated by the use of a least-squares bilinear interpolator. Instead of using the actual pixel values of the original image that lie on the sampling grid, the intensity values transmitted to the decoder for the block corners are computed to minimize the MSE of the overall block. After performing the quadtree segmentation, the information provided by the corners of small blocks adjacent to larger ones can be used to improve the interpolation which will result in

better reconstruction. The combination of the spatially adaptive splitting along with LSBI is denoted as the Adaptive Least-Squares Bilinear Interpolation (ALSBI) image coding algorithm [16].

The optimal solution in the MSE sense can be achieved by solving the equation from [15]:

$$\mathbf{Ax} \simeq \mathbf{y} \tag{3.4}$$

where $\mathbf{x}$ is the vector containing all the unknown corner values, $\mathbf{y}$ is the vector of the image data, and $\mathbf{A}$ is a rectangular matrix. Matrix $\mathbf{A}$ can be calculated from the coefficients of the bilinear interpolation for all the image pixels, as a function of their positions only. It can be shown that the vector $\mathbf{x}$ that minimizes the MSE between the approximate points, $\mathbf{Ax}$, and the original ones, $\mathbf{y}$, can be calculated from the following equation, which satisfies the least-squares approximation:

$$\mathbf{x} = \left(\mathbf{A}^{\mathrm{T}}\mathbf{A}\right)^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{y} \tag{3.5}$$

where T denotes matrix transposition (see Appendix A for a complete derivation). Even though $\mathbf{A}$ is a very sparse matrix, because only a small number of corner pixels contribute to the local approximation, for an image of size 512×512, for example, the computation and storage requirements become overbearing. To overcome this problem a suboptimal solution has been developed by [15] which allows the interpolation error to be noticeably reduced (under the same splitting conditions) and provides more accurate edge reconstruction.

This suboptimal solution is based on a local solution of the least-squares problem which is computed for each block. Thus, for each block the following equation is solved:

$$\mathbf{Bx} \simeq \mathbf{y} \tag{3.6}$$

where $\mathbf{x}$ is a vector of the four corner pixel values, $\mathbf{y}$ is the vector of the original block values, and $\mathbf{B}$ is analogous to the matrix $\mathbf{A}$ for the block being considered. The solution to this formula is identical to the global solution given in Equation 3.5 with $\mathbf{B}$ in place of $\mathbf{A}$:

$$\begin{aligned} \mathbf{x} &= \left(\mathbf{B}^T\mathbf{B}\right)^{-1}\mathbf{B}^T\mathbf{y} \\ &= \mathbf{S}\mathbf{y} \end{aligned} \tag{3.7}$$

The resulting interpolation scheme can still be optimal in the least-squares sense provided that the four least-squares optimized corner values are retained for each block. This is quite impractical because it leads to a large increase in the storage requirements since the four corner values are no longer shared with neighboring blocks. This also can heavily reduce the intensity continuity between adjacent blocks. In order to overcome these problems a suboptimal criterion, as suggested in [15], can be followed: the new value for each shared corner pixel is calculated by averaging the optimal values. Although this solution is quite simple, it does provide a higher quality reconstructed image.

A further improvement to the reconstructed quality can be made by using a weighted average calculation on the shared optimal corner values instead of a simple average. A corner pixel can be shared by up to four neighboring blocks. It quite possible that these neighboring blocks may contain different numbers of pixels due to varying areas covered by each block. This means that the optimized corner point for a larger block should be given more weight than a point from a smaller neighboring block because during the reconstruction interpolation, the corner point for the larger block will affect a larger area of the image. If this is done, the larger blocks will have less error than their smaller neighbors which results in an overall higher PSNR of the output image. Thus, a weighted average, where the weights are proportional to the

area covered by each block, should be used to compute the vertex values that are to be stored or transmitted to the decoder.

It should be noted that, when using this LSBI optimization, no extra calculations are required when decoding and no changes to the decoder are needed. All that is needed is the addition of a post-processing step to the encoder in order to compute the LSBI optimized vertices.

An alternative method for the ALSBI coder has been described in [16]. It calculates the inverse to a least-squares matrix formulation similar to Equation 3.4 via an iterative gradient descent algorithm (e.g. Gauss-Jacobi algorithm [2]). It further suggests that the LSBI should be performed during the splitting process. The quadtree is generated on a level-by-level (breadth first) basis. For each level, LSBI is performed and the MSE is calculated for all blocks. The blocks for which the MSE is above the threshold are further split. After each splitting iteration, the optimized corner vertices from the previous step remain unchanged only if they do not belong to a split block. The computational load is quite high because the LSBI must be computed before each decision to split is made. Also, large blocks which neighbor smaller blocks will not have all optimal corner pixel values since the least-squares calculation is done based on a smaller block size.

The disadvantage to using the LSBI method is the least-squares error minimization matrix calculation performed in Equation 3.7. As can be seen, the calculation of the optimization matrix, **S**, involves many matrix multiplications and a matrix inversion. These are very costly operations to compute—especially the matrix inversion—which can severely slow down the encoding process. To overcome this, it is possible to pre-calculate and store **S** for all possible block sizes. Then when the LSBI optimization is required, only one vector-matrix multiplication is required.

3.4.1 Reducing Coefficient Storage Requirements

The large number of calculations required for the LSBI approach make it somewhat unattractive, if high speed encoding is desired. Even if the optimization matrices are computed off-line, the memory storage requirements become quite large, as there are many possible block sizes—especially for images with non-unity aspect ratios. Given a $M \times N$ block, the matrix $\mathbf{S}_{M,N}$ will have 4 rows and MN columns for a total of $4MN$ elements. Using 4 byte floating point values, that results in a total of $16MN$ bytes. That means that a block of size 16×16, would require 4096 bytes of memory—and that is just for one possible block size! Given $D_{max} = 32 \times 32$, 495 $\mathbf{S}$ matrices are needed (the matrix $\mathbf{S}_{N,M}$ can be constructed by re-ordering the matrix $\mathbf{S}_{M,N}$ which halves the storage space). That means a grand total of 2313280 bytes (over 2.2MB) required for storage of the coefficients.

To reduce the calculation and possible memory requirements of the LSBI algorithm, it is useful to study exactly what the least-squares optimization matrix does. It can be thought of as four 2-D finite impulse response (FIR) filters (or convolution masks) having identical magnitude-frequency responses with differing phase responses, each one acting on a different corner.

Now, examining the 1-D case for a moment, the least-squares linear interpolation (LSLI) optimization is derived in the exact same manner as for the LSBI optimization. The only difference is the dimensions of the matrices. The vector $\mathbf{x}$ in Equation 3.6 contains the two end-point values while $\mathbf{y}$ is a vector of the original values. It follows that the least-squares optimization matrix contains 2 rows and as many columns as there are original values. The 1-D LSLI optimization matrix $\mathbf{S}$ can also be interpreted as a series of FIR filters that each output one of the optimized end-points. It is then possible to construct the corresponding 2-D LSBI optimization

filters via the same method that the bilinear interpolating filter was created from the linear interpolating filter: row filtering followed by column filtering. The two LSLI filters when combined with one another results in the four LSBI filters required for the 2-D optimization.

To illustrate this, the following example is presented. Given a 4×5 block, the LSLI optimization is computed for a linear interpolation across 4 elements and across 5 elements. This results in the optimization matrices:

$$\mathbf{S}_4 = \begin{bmatrix} \mathbf{s}_4^{\mathrm{T}} \\ \mathbf{s}_4^{\mathrm{TR}} \end{bmatrix} = \frac{1}{10} \begin{bmatrix} 7 & 4 & 1 & -2 \\ -2 & 1 & 4 & 7 \end{bmatrix}$$

$$\mathbf{S}_5 = \begin{bmatrix} \mathbf{s}_5^{\mathrm{T}} \\ \mathbf{s}_5^{\mathrm{TR}} \end{bmatrix} = \frac{1}{5} \begin{bmatrix} 3 & 2 & 1 & 0 & -1 \\ -1 & 0 & 1 & 2 & 3 \end{bmatrix}$$

where R indicates a reversal of the vector values. The first row of each matrix contains the weighting factors for the first end-point and the second row (which is a reversed version of the first row) is used to compute the optimized second end-point. Now, the four 2-D convolution masks required for LSBI can be created by multiplying together every combination of the four vectors $\mathbf{s}_4$, $\mathbf{s}_4^{\mathrm{R}}$, $\mathbf{s}_5$ and $\mathbf{s}_5^{\mathrm{R}}$. So, the 2-D filter required to optimize the upper left vertex value is simply created by row filtering with $\mathbf{s}_5$ and column filtering with $\mathbf{s}_4$. That is,

$$\mathbf{H1}_{4\times 5} = \mathbf{s}_4 \mathbf{s}_5^{\mathrm{T}} = \frac{1}{10} \cdot \frac{1}{5} \begin{bmatrix} 7 \\ 4 \\ 1 \\ -2 \end{bmatrix} \begin{bmatrix} 3 & 2 & 1 & 0 & -1 \end{bmatrix}$$

The other three required masks are created in a similar fashion. These calculations

result in the four LSBI optimization convolution masks:

$$\mathbf{H1}_{4\times5} = \frac{1}{50}\begin{bmatrix} 21 & 14 & 7 & 0 & -7 \\ 12 & 8 & 4 & 0 & -4 \\ 3 & 2 & 1 & 0 & -1 \\ -6 & -4 & -2 & 0 & 2 \end{bmatrix} \quad \mathbf{H2}_{4\times5} = \frac{1}{50}\begin{bmatrix} -7 & 0 & 7 & 14 & 21 \\ -4 & 0 & 4 & 8 & 12 \\ -1 & 0 & 1 & 2 & 3 \\ 2 & 0 & -2 & -4 & -6 \end{bmatrix}$$

$$\mathbf{H3}_{4\times5} = \frac{1}{50}\begin{bmatrix} -6 & -4 & -2 & 0 & 2 \\ 3 & 2 & 1 & 0 & -1 \\ 12 & 8 & 4 & 0 & -4 \\ 21 & 14 & 7 & 0 & -7 \end{bmatrix} \quad \mathbf{H4}_{4\times5} = \frac{1}{50}\begin{bmatrix} 2 & 0 & -2 & -4 & -6 \\ -1 & 0 & 1 & 2 & 3 \\ -4 & 0 & 4 & 8 & 12 \\ -7 & 0 & 7 & 14 & 21 \end{bmatrix} \quad (3.8)$$

which when put into a single matrix (one row for each mask) is identical to the $\mathbf{S}_{4\times5}$ matrix that would be calculated using the 2-D LSBI optimization described in Section 3.4. Due to the symmetry of the $\mathbf{s}$ and $\mathbf{s}^R$ vectors, the mask for any corner can be created from any other corner mask by way of flipping the mask values horizontally, vertically, or both.

By constructing the 2-D LSBI convolution masks from the 1-D LSLI filters, a large amount of space can be saved by only storing the short LSLI optimization filters. So with $D_{max} = 32 \times 32$, only 31 vectors that require 527 elements of storage, which when using 4 byte floating-point values, results in 2108 bytes. This amount is miniscule (over 1000 times less storage) when compared to the 2313280 bytes required for storing all the required 2-D LSBI arrays! In fact, by storing these values using 2 byte integer values with a 2 byte scaling factor for each vector would result in 1116 bytes of storage. Also, by doing this, floating-point operations are no longer required to perform the LSBI optimization.

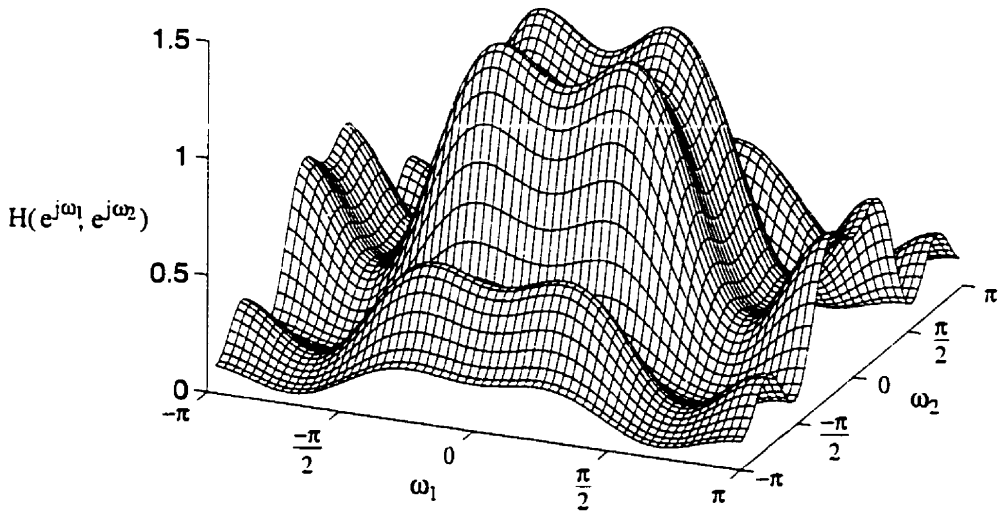


Figure 3.9: LSBI magnitude-frequency response for a 4×5 block.

3.4.2 Examination of the LSBI Frequency Characteristics

In Section 2.4.1 it was stated that in order to eliminate aliasing when sub-sampling the bandwidth of the signal must satisfy the inequality

$$\mathrm{BW} < \frac{\pi}{M} \tag{3.9}$$

where M is the sub-sampling factor. In order to reduce the effects of aliasing, the signal can be filtered to remove higher frequency components before sub-sampling. This is exactly what the least-squares optimization does. The optimization for each corner pixel is, in fact, a 2-D low-pass filter designed to remove frequency components above $\frac{\pi}{M}$ in each dimension which in turn reduces unwanted effects caused by aliasing. Figure 3.9 demonstrates this by showing the magnitude-frequency response of the $\mathbf{H1}_{4\times5}$ filter in Equation 3.8.

3.5 Results and Discussion

With an understanding of the basic image coding system, along with some possible modifications, various comparisons can be drawn. These comparisons are based on the storage requirements of the coded image (in bits/pixel) and the quality of the reconstructed image output by the decoder measured via PSNR.

In Section 2.4 an example of simple bilinear interpolation across a fixed grid was shown. The only required parameter was simply the grid spacing (or sub-sampling factor) in each direction. This parameter does not provide much control over the quality of the output image and for large sub-sampling factors the image can become unrecognizable. In the example, the image of Lena was coded at 1.53 bits/pixel and the output had a PSNR of 28.1 dB. This base result will be used to demonstrate the effect of the various modifications to the image coding method.

The spatially adaptive sub-sampling modification improves upon the basic method. Given the same image of Lena and using a minimum block size of 3×3, the adaptive coder can achieve a PSNR of 31.7 dB at the same bit rate of 1.53 bits/pixel. This improvement of 3 dB is caused by strategically located samples in areas where they are most needed. It is then possible to remove more information from the entire image without decreasing the quality. Figure 3.10(a) demonstrates that where high frequency components are located, more samples are allocated to that area. Areas such as edges and the feathers in her hat have large numbers of samples while more uniform areas such as the wall have fewer samples. From Figure 3.10(b), the output image after bilinear interpolation shows finer details than possible by simple fixed grid sub-sampling. In this example the splitting tree contained 15793 tree nodes (11845 of which were leaves) that were coded at 0.067 bits/node by the LZW coder. The compressed image also included 14300 vertices that were coded at 6.98 bits/vertex. This

(a) (b)

Figure 3.10: Spatially adaptive sub-sampling: (a) image samples retained, (b) reconstructed image (PSNR = 31.7 dB).

is somewhat less than the 16641 vertices required by the fixed 3×3 grid sub-sampling method.

In Section 3.2.4 the problem of discontinuities arose due to differing block sizes between neighboring blocks. After performing the block size equalization algorithm discussed in that section, the decoder produces a slightly higher quality image which is shown in Figure 3.11. The increase in PSNR is only 0.4 dB, however the effects of the visible discontinuities have been reduced and the output is more pleasing to the eye. The only drawback is that the decoding time in this example increased by a factor of 4. The time increase has been somewhat amplified due to the small minimum block size of 2×2.

In the compressed image, the vertices require the largest amount of storage. This is because they are stored in a lossless fashion which always requires more storage space than lossy coding. To remedy this, the vertices can be quantized as per Section 3.3. With a lower bits/vertex requirement, a higher resolution can be achieved at

Figure 3.11: Output image after block size equalization (PSNR = 32.1 dB).

the same overall bit rate. So coding the "Lena" image again, and using a fixed quantizer, the PSNR can be boosted to 34.8 dB at the same bit rate of 1.53 bits/pixel. The output of this is displayed in Figure 3.12. The higher quality of the reconstructed image using quantized vertices is because over 1.5 times the number of pixels are available for reconstruction when compared to Figures 3.10 and 3.11. With a higher number of samples, smaller block sizes can be utilized. Now areas like the top of Lena's hat are now distinct where as in Figures 3.10 and 3.11 it is blurred by a large block at that location.

The PSNR can be increased further by employing the least-squares optimization on the vertex values as described in Section 3.4. For comparison, the image of Lena is coded at 1.53 bits/pixel using the LSBI optimization with and with out quantizing the vertex values. The resulting reconstructed images are indeed of a higher quality with regards to PSNR compared to those images coded without using LSBI. The output image, shown in Figure 3.13(b), has a PSNR of 35.7 dB—almost a full decibel above the previous example (Figure 3.12). A 1 dB improvement is also seen

Figure 3.12: Output image using quantized vertices (PSNR = 34.8 dB).

(a) (b)

Figure 3.13: Output images resulting from ALSBI coding: (a) non-quantized vertices (PSNR = 33.2 dB), (b) quantized vertices (PSNR = 35.7 dB).

Table 3.3: Block sizes created by image coder ("Lena" coded at 1.53 bits/pixel).

Figures	Quantized Vertices	Block Sizes					
		33 × 33	17 × 17	9 × 9	5 × 5	3 × 3	2 × 2
2.14[a]	no	0	0	0	0	16384	0
3.11, 3.13(a)	no	3	35	255	1006	2664	7872
3.12, 3.13(b)	yes	1	34	178	1047	3381	14140

[a]Non-adaptive, fixed sampling grid method from Section 2.4

Table 3.4: Effect of vertex quantization and LSBI on PSNR ("Lena" coded at 1.53 bits/pixel).

Figure	Quantized Vertices	LSBI	PSNR (dB)
2.14[a]	no	no	28.1
3.11	no	no	32.1
3.13(a)	no	yes	33.2
3.12	yes	no	34.8
3.13(b)	yes	yes	35.7

[a]Non-adaptive, fixed sampling grid method from Section 2.4

by performing the LSBI optimization when coding the image using non-quantized vertices (Figure 3.13(a) vs. Figure 3.11).

Tables 3.3 and 3.4 show the effects of spatial adaptivity, vertex quantization, and LSBI on PSNR and block size. From the tables it can be seen that vertex quantization enables smaller block sizes to be realized using the same bit rate. This results in a higher PSNR. The PSNR is increased even further by performing the LSBI optimization on the vertices before quantization.

In order to look at how the compression method works over a range of bits/pixel, the coding parameters can be varied and the results observed. Due to the number of parameters (splitting error tolerance, minimum block size, vertex quantization factor, block size equalization, and LSBI optimization) the complete behavior can not

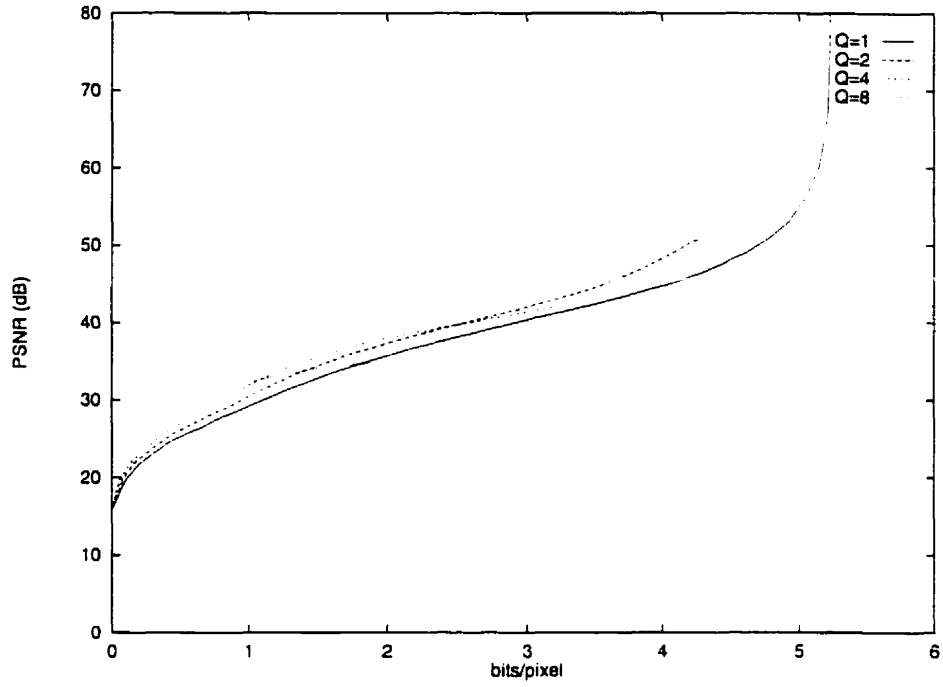


Figure 3.14: Effects of varying the splitting MSE threshold on bits/pixel and PSNR for various vertex quantization factors (Q).

be visualized on one graph. To maintain continuity for all the results, the same image of Lena is used to observe the effects of the image coder. The "Lena" image is a standard test image used for many image processing applications because it is a "natural" image and has a wide variety of textures distributed throughout.

Figure 3.14 shows the relationship between PSNR and bits/pixel as the splitting MSE threshold is varied. As the threshold is raised, output image quality as well as the bits/pixel are reduced. To illustrate the effect of vertex quantization, the graph contains traces corresponding to different quantization values ($Q = 1, 2, 4,$ and 8). As can be seen, when $Q = 1$, perfect reconstruction (PSNR $= \infty$) is possible. At lower bits/pixel it is advantageous to increase the quantization factor in order to increase the reconstructed image quality.

In Figure 3.15, the effects of the quantization factor are shown while holding

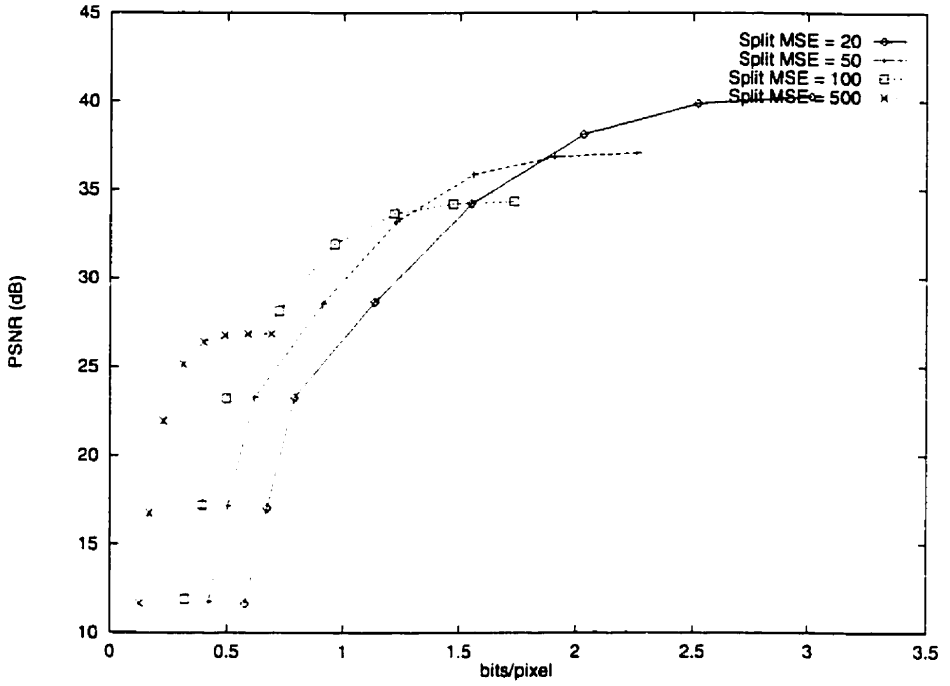


Figure 3.15: Effects of varying the vertex quantization factor on bits/pixel and PSNR for various splitting MSE threshold.

various splitting MSE thresholds constant. The points on each graph trace correspond to quantization values of 1, 2, 4, 8, 16, 32, 64, and 128. As the quantization factor is increased, more information is lost which results in a reduction of both the bits/pixel and the PSNR. However, the quality does not begin to drop sharply until $Q > 4$ which makes $Q = 4$ a good choice to provide a compromise between quality and compression.

The minimum block size (D_{min}) can also be changed to affect the PSNR-bits/pixel relationship. Figure 3.16 again shows the effects of sweeping the splitting MSE threshold for various minimum block sizes. As the minimum block size is raised, the output image quality is diminished. It can be seen, at higher splitting error thresholds, the behavior is identical for different values of D_{min} because the error threshold becomes so high that splitting stops before D_{min} is reached. Another interesting ef-

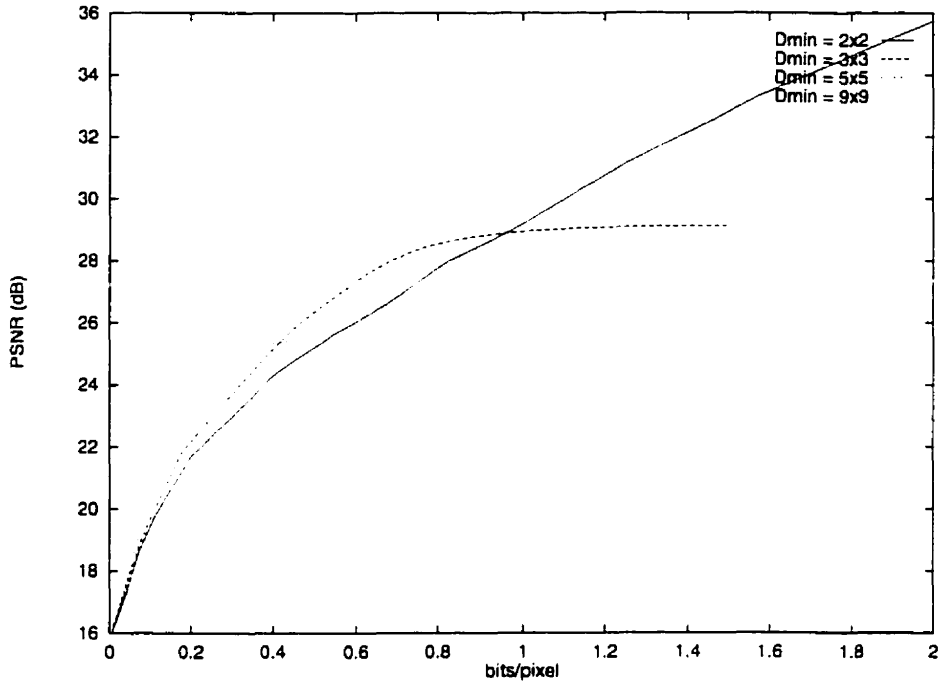


Figure 3.16: Effects of varying the minimum block size (D_{min}) on bits/pixel and PSNR for various splitting MSE threshold.

fect of increasing the minimum block size is that the available range of bits/pixel is reduced when the splitting error threshold is varied.

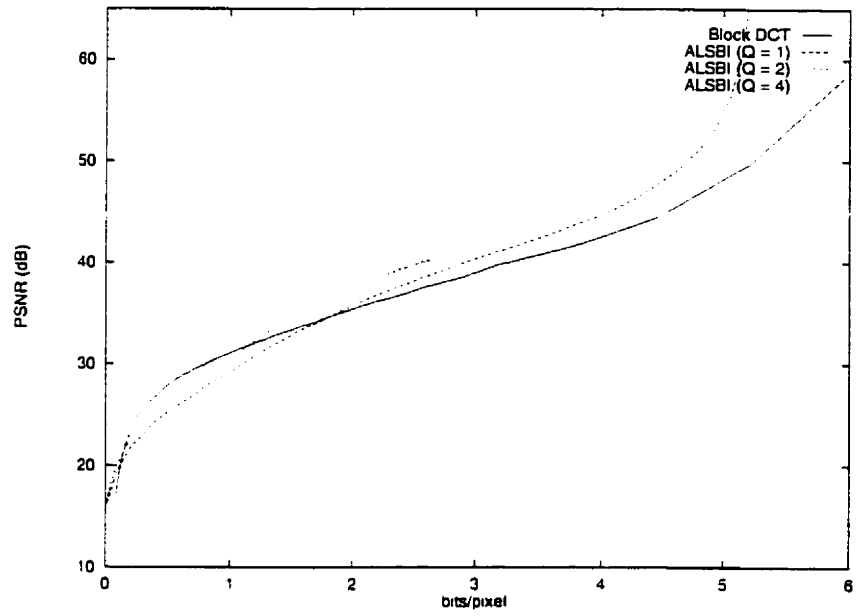
3.5.1 Comparison of Adaptive Interpolation and Block DCT Coding

To gain a better insight into the performance of the Adaptive Interpolation coder, a comparison can be drawn between it and a Block Discrete Cosine Transform (DCT) image coder which is a popular method for the coding of image data and has been used for many years [23]. The JPEG baseline image compression standard [13] makes extensive use of the Block DCT in order to decorrelate the input image data before entropy coding.

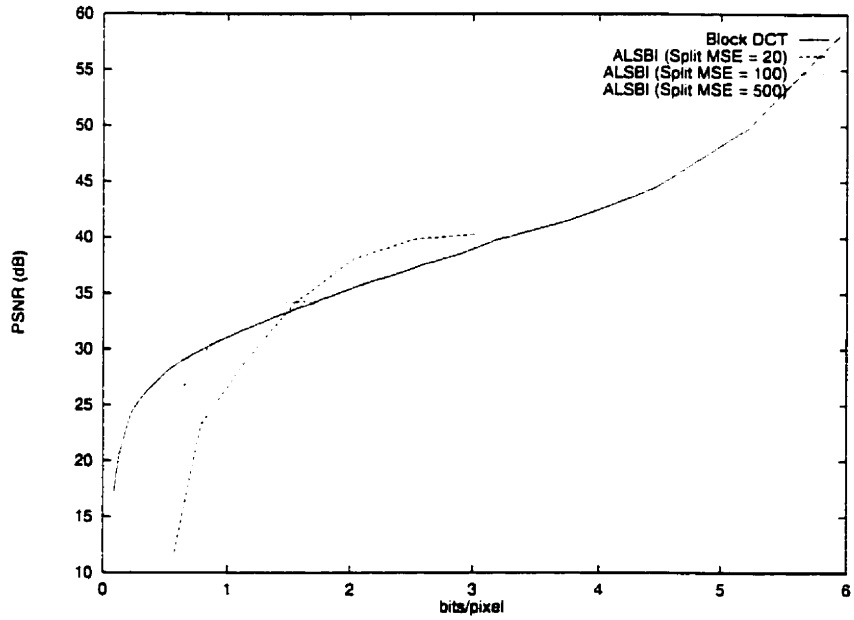
The Block DCT coder used in this comparison first starts by segmenting the input image into 8×8 blocks and performs the 2-D DCT on each block. Then the DCT coefficients are quantized using a quantization matrix scaled by a "quality" parameter. The base quantization matrix used is that of the *de facto* quantization matrix that is used in the JPEG image coding algorithm. After quantization, zig-zag run length coding is performed to remove the large number of zero coefficients resulting from quantization. Finally, the output of the run length coder is entropy coded using a Huffman coder.

Figure 3.17 shows how the ALSBI image coder performance compares to that of the Block DCT coder while (a) sweeping the splitting MSE tolerance and (b) sweeping the vertex quantization factor. From the figure it can be seen that there are certain places on the PSNR-bits/pixel graph ALSBI maintains a higher quality with the same number of bits/pixel. This occurs between 1 and 5 bits/pixel in Figure 3.17(a) and between 1 and 3 bits/pixel in Figure 3.17(b) for small values of Q (i.e. $Q = 1, 2, 4, 8$). Figure 3.17(a) also shows that at extremely low PSNRs, the ALSBI coder can represent the "Lena" image with slightly fewer bits/pixel than the Block DCT coder. However, this region of the PSNR-bits/pixel graph is not very useful in image coding due to the visually poor output image quality. To illustrate this point, Figure 3.18 shows two output images with PSNR ≈ 21 dB—one from the Block DCT coder and one from the ALSBI image coder. The ALSBI coder is able to code the image with slightly fewer bits/pixel than the Block DCT coder (0.106 bits/pixel vs. 0.130 bits/pixel).

From the Block DCT output image two types of distortion are visible. First the "blockiness" of the output is due to the discontinuities between adjacent 8×8 blocks. In order to achieve such low bits/pixel the DCT coefficients are heavily quantized which frequently leaves the DC coefficient the only non-zero coefficient. Thus, every



(a)



(b)

Figure 3.17: Comparison between the ALSBI image coder and a Block DCT image coder by (a) sweeping the splitting MSE threshold and (b) sweeping the vertex quantization factor.

(a) (b)

Figure 3.18: Output images at low PSNR (a) Block DCT (PSNR = 20.6 dB, 0.130 bits/pixel). (b) ALSBI (PSNR = 20.7 dB, 0.106 bits/pixel).

pixel value in the reconstructed 8×8 block is the quantized average of the input block pixel values. The quantization also leads to the second type of distortion visible, that is the quantized or limited number of greyscale values present.

In the ALSBI output image, greyscale quantization is not very apparent as the bilinear interpolation fills in intermediate values between the quantized vertex values. The most visible distortion is the blurring or smearing in areas of the image where few vertices are present. In areas containing smaller blocks and thus more vertices the quality increases dramatically.

The number of bits/pixel required for the image in Figure 3.18(b) can further be decreased by increasing the splitting MSE threshold. However, the plot in Figure 3.17(a) shows that the output image quality degrades rapidly when the number of bits/pixel is reduced in this region.

3.5.1.1 Comparison for Coding of Bi-level Images

It has been previously stated in Section 1.2.1, that JPEG (Block DCT) does not perform well on two-toned images [3]. Therefore, a comparison of Block DCT coding and of how the adaptive interpolation image codec performs on bi-level images.

Bi-level or black-and-white images are not used as frequently as greyscale or colour images, but they are extensively used in Fax (Facsimile) transmission and are sometimes used as "preview" images, in order to save storage space or transmission bandwidth. In its raw form, a bi-level image only requires 1 bit/pixel for representation, which can be reduced further by using an algorithm, such as the JBIG (Joint Bi-level Image experts Group) algorithm [3], that specializes in black-and-white images. However, this results in having to change the image codec for different image types.

Figure 3.20 shows a black-and-white version of the "Lena" image. In the image, pixels are only have the values 0 or 255. By coding and decoding this image multiple times, with the adaptive interpolation method, while changing the MSE splitting threshold, the plots for $Q = 1$ and $Q = 255$ in Figure 3.19 are created. In the same figure, the PSNR vs. bits/pixel relationship for the Block DCT coder, operating on the same image, is also shown. As can be seen from the figure, the Block DCT coder does not perform well on the two-toned image of Lena. Although the Adaptive Interpolator with $Q = 1$ does not perform much better, there is some improvement. When Q is given a maximum value of $Q = 255$ a remarkable improvement is seen.

To illustrate further, an output image coded at 1.4 bits/pixel by each codec is shown in Figure 3.21. The output of the Block DCT coder, Figure 3.21(a), shows more distortion than the output of the Adaptive Interpolation coder, Figure 3.21(b). In a PSNR comparison, the difference is considerable: over 18 dB. Visually, the main

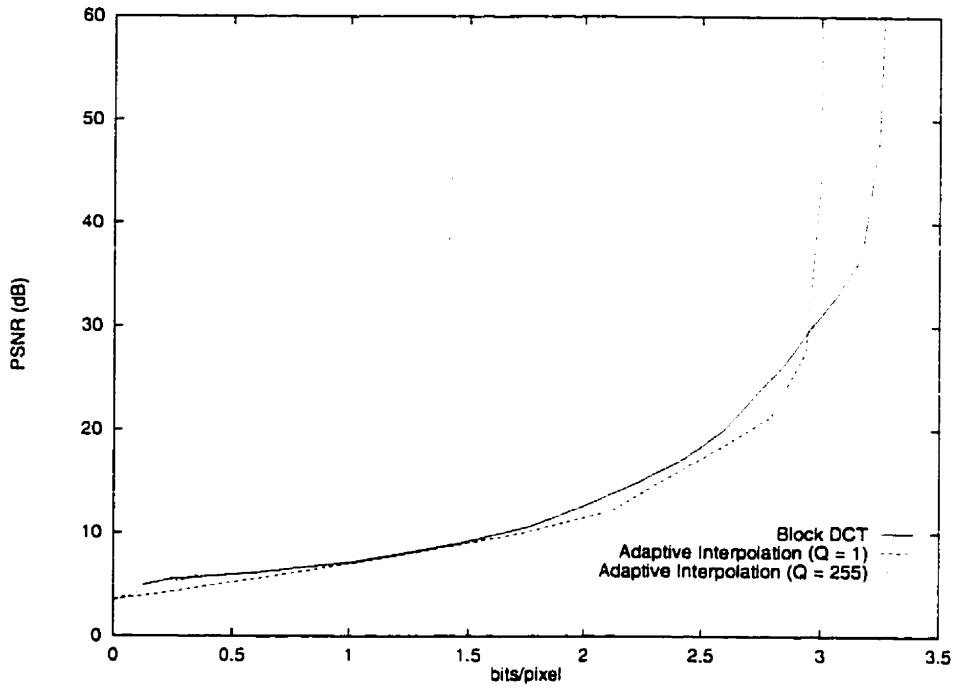


Figure 3.19: PSNR vs. bits/pixel comparisons for bi-level image coding with the Block DCT coder and the Adaptive Interpolation coder with $Q = 1$ and $Q = 255$.

Figure 3.20: Black-and-white image of Lena.

(a) (b)

Figure 3.21: Block DCT and Adaptive Interpolation black-and-white image coding at 1.4 bits/pixel: (a) Block DCT (PSNR = 8.8 dB) and (b) Adaptive Interpolation (PSNR = 27.6 dB).

difference is the "noise" seen over the entire Block DCT coded image. The Block DCT coder performs badly since it has difficulty with the large changes in pixel values that result in large numbers of high frequency components in the DCT.

It should be noted that neither codec system, Block DCT or Adaptive Interpolation, is able to code the black-and-white image of "Lena" at less than 1 bit/pixel very well. However, this example does show that the Adaptive Interpolation method does perform better on less "continuous toned" images than the Block DCT coder—black-and-white images being an extreme case of less "continuous toned" images.

Chapter 4

Adaptive Three-Dimensional Sampling and Interpolation for Image Sequence Coding

In Chapter 2 the basic theory behind a two dimensional image codec was discussed. It was shown how an image could be sub-sampled along a non-uniform grid and reconstructed via bilinear interpolation. Then in Chapter 3 some specific issues were discussed concerning the implementation of this adaptive interpolation coder/decoder system.

This chapter is an extension of the 2-D image coder system into three dimensions (two spatial dimensions and one temporal dimension) so that image sequence data compression can be achieved. The main difference between 2-D and 3-D coding is, obviously, the addition of the temporal dimension (or time axis). As for images, the domain of the two spatial dimensions is limited. However, the temporal domain can stretch to infinity (i.e. there can be no limit to the number of frames in an image sequence). The large number of frames in digitized video results in huge amounts of

data to be stored or transmitted. Usually, the volume of data is so vast that the raw or uncompressed storage of the video is impossible or very impractical. So in most cases, digitized video data is compressed before storage or transmission.

Since the image coding technique discussed in Chapter 2 treats the image as a two dimensional plane of data, the three dimensional volume of data in an image sequence can be coded using the same basic ideas. A general overview of the video codec is provided in Section 2.1. Then, trilinear interpolation, the most important aspect of the system, is detailed in Section 4.2—both in theory and implementation. Following that, in Section 4.3, the generation of the 3-D adaptive sampling grid is presented. The storage of the grid structure and the coding of the grid vertices will also be discussed. Then, analogous to Section 3.4, an improvement to the codec system, based on the optimization of the vertex values using a least-squares error method, is presented in Section 4.4. Finally, Section 4.5 contains various results achieved with this video coder and also includes a discussion regarding these results. In this chapter, special attention will be given to areas where the 3-D methods differ from their 2-D counterparts described previously.

4.1 Overview of the Video Codec System

The video codec system is quite similar to the image codec described in Section 2.1. It has the same order of operations except that the operations are performed on a 3-D volume of data instead of a 2-D plane. In Figure 4.1 the basic structure of the coder and decoder is shown: the video codec is quite similar to the image codec of Chapters 2 and 3. In the coder, the original sequence is recursively split spatially and/or temporally. The splitting information is stored in a tree structure that is compressed by a LZW-based coder. The remaining sample points lying on the

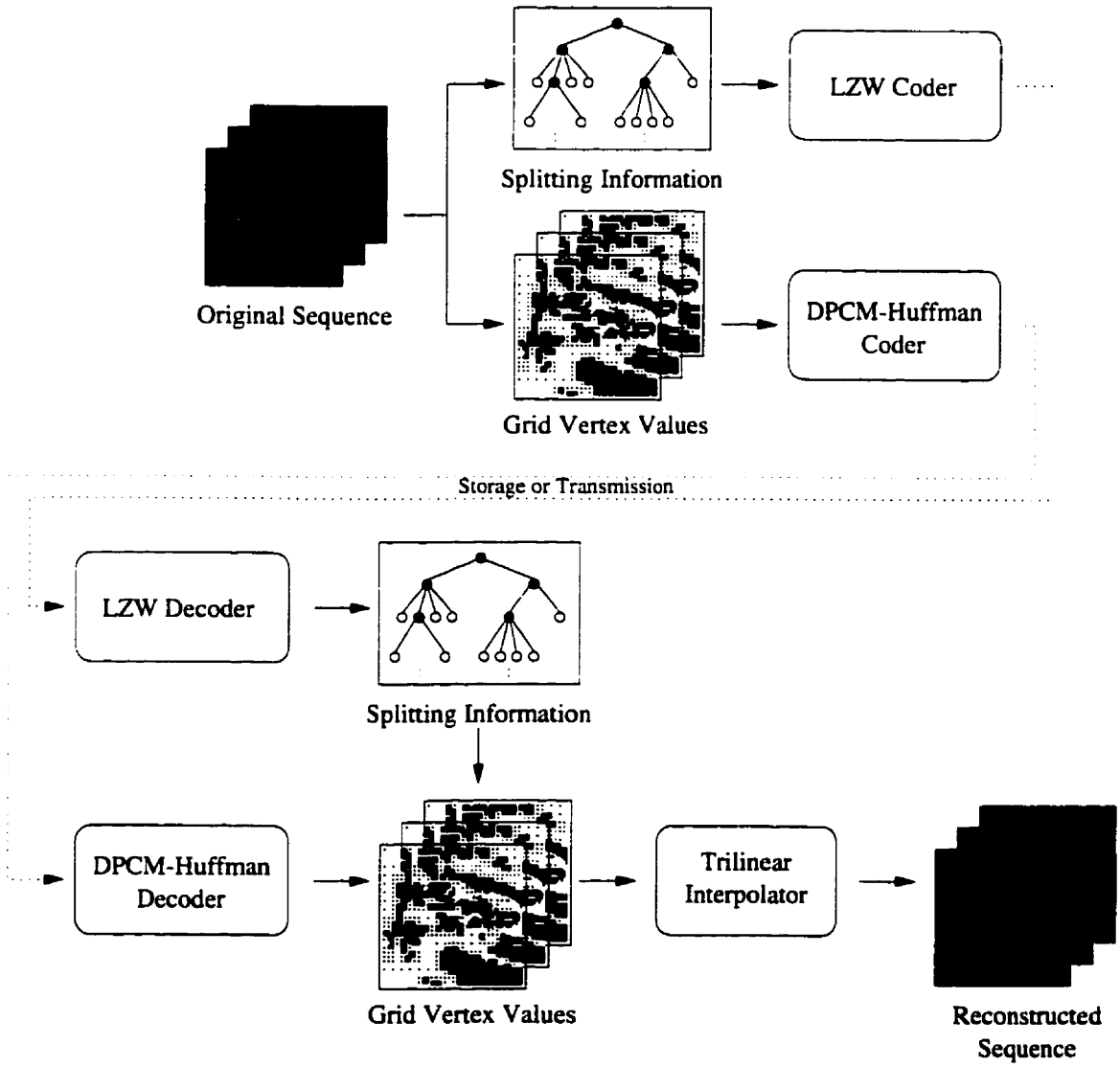


Figure 4.1: Block diagram of the image coder/decoder system.

splitting grid vertices are DPCM-Huffman coded. The splitting information and the grid vertices are then either stored, to be decoded later, or transmitted directly to the decoder. In the decoder, the process is basically reversed. The LZW coded splitting tree is decoded and then used to DPCM-Huffman decode the values at the grid vertices. Then it is simply a matter of interpolating the grid vertices to reconstruct the voxel values lying between grid points.

It is possible to simply perform 2-D coding on each frame of an image sequence (intra-frame coding), which uses spatial redundancy to achieve data compression. However, in an image sequence, there can be large amounts of temporal redundancy—especially for stationary objects or backgrounds within the sequence. A large amount of transmission bandwidth (or storage space) can be saved by exploiting these temporal similarities (inter-frame coding). So by treating the video data as a 3-D volume of data, inter- and intra-frame coding can be performed at the same time.

The advantages gained by using the previously described 2-D image coder can be translated to advantages for the 3-D video coder. These advantages are:

1. the absence (or near-absence) of floating point calculation requirements.
2. the possibility of high speed operation in both the coding and decoding stages.
3. the absence of edge effects and blockiness associated with some other video coding techniques.
4. and the possibility for lossless coding of an image sequence—depending on implementation and the spatial dimensions of the image sequence.

Furthermore, an advantage that this method has over some other video coding systems is that there is no need for computationally expensive motion estimation for the

utilization of temporal redundancies. Overall, these properties result in an attractive video coder/decoder system.

4.2 Three Dimensional Trilinear Interpolation

The video codec described in this chapter is based on the volumetric interpolation of a rectangular parallelepiped (a prism having all rectangular faces [24]) using eight corner points. The simplest interpolator that fulfills this requirement—one that is a first order interpolator in each dimension—is the trilinear interpolator. It is used both in the video coder and decoder.

Just as the bilinear interpolator can be constructed via multiple 1-D linear interpolations, so can the trilinear interpolator. Separate linear interpolations, as defined in Equation 2.1, need to be performed in both spatial dimensions as well as the temporal dimension. Thus, given a three dimensional signal, $Z_{i,j,k}$, with known corner values lying on a rectangular parallelepiped, trilinear interpolation can be achieved by bilinear interpolation of the front and back faces (which is performed by cascading multiple linear interpolations) and linear interpolating between those faces. This three step process involving interpolating in each of the three dimensions is shown in Figure 4.2.

To calculate the trilinear interpolation function, an analytical combination of the separate linear interpolations can be performed. Given an $(N_1 + 1) \times (N_2 + 1) \times (T + 1)$ block with the eight corner values known, the trilinear interpolation of a generic point $Z_{i,j,k}$ is calculated as such [17]:

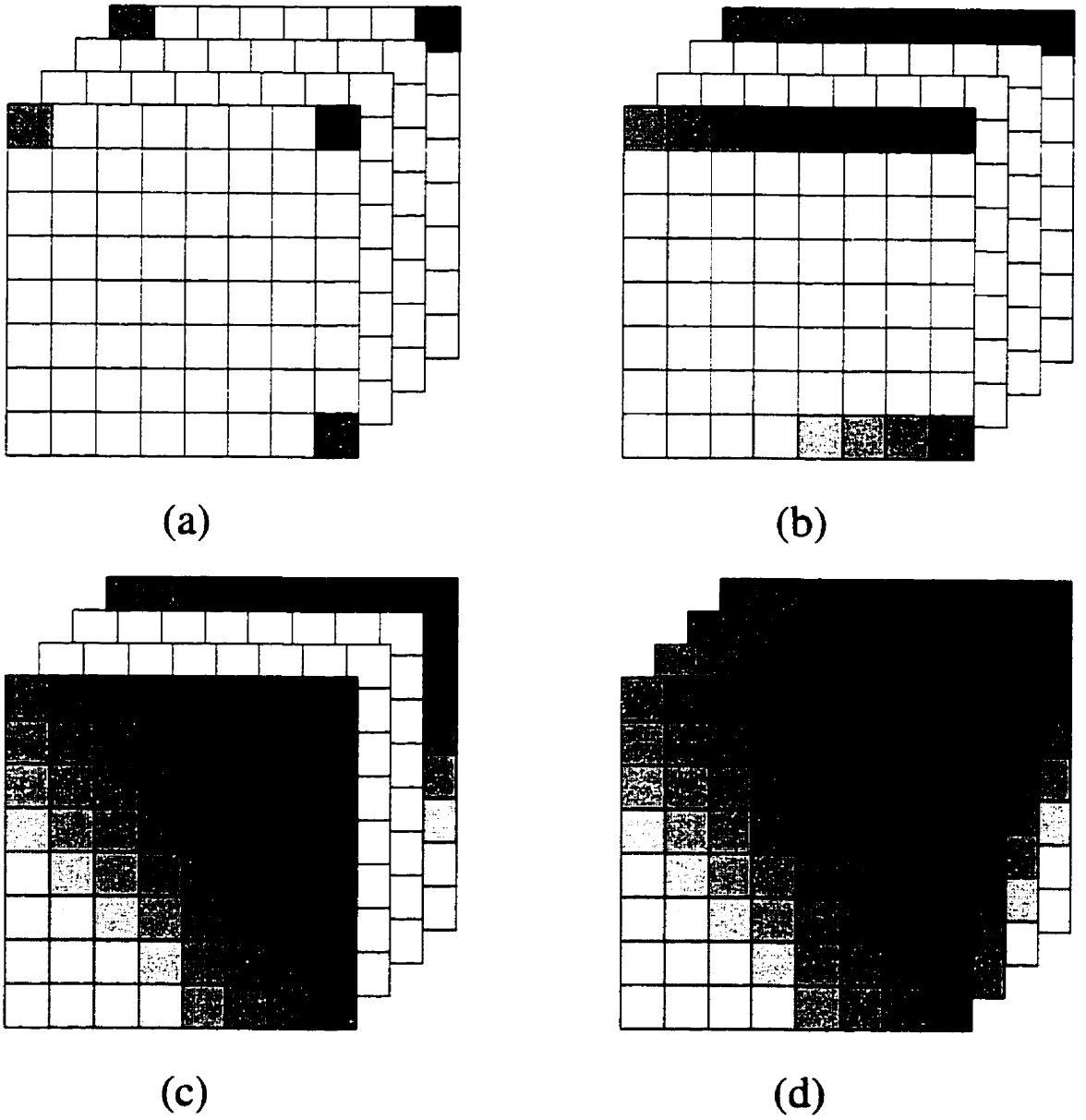


Figure 4.2: Three step, three dimensional trilinear interpolation: (a) original block, (b) interpolation of top and bottom rows on the front and back faces, (c) interpolation of each column on the front and back faces, and (d) interpolation between the front and back faces [17, 25, 26].

1. compute the value of the point $Z_{0,j,0}$ (which is a linear interpolation between $Z_{0,0,0}$ and $Z_{0,N_2,0}$):

$$Z_{0,j,0} = \frac{Z_{0,N_2,0} - Z_{0,0,0}}{N_2} j + Z_{0,0,0} \tag{4.1}$$

2. compute the value of the point $Z_{N_1,j,0}$ (linear interpolation between $Z_{N_1,0,0}$ and $Z_{N_1,N_2,0}$):

$$Z_{N_1,j,0} = \frac{Z_{N_1,N_2,0} - Z_{N_1,0,0}}{N_2} j + Z_{N_1,0,0} \tag{4.2}$$

3. compute the value of the point $Z_{i,j,0}$ (linear interpolation of $Z_{0,j,0}$ and $Z_{N_1,j,0}$ from equations 4.1 and 4.2):

$$\begin{aligned} Z_{i,j,0} &= \frac{Z_{N_1,j,0} - Z_{0,j,0}}{N_1} i + Z_{0,j,0} \\ &= \frac{(Z_{N_1,N_2,0} + Z_{0,0,0}) - (Z_{N_1,0,0} + Z_{0,N_2,0})}{N_1 N_2} ij \\ &\quad + \frac{(Z_{N_1,0,0} - Z_{0,0,0})}{N_1} i + \frac{(Z_{0,N_2,0} - Z_{0,0,0})}{N_2} j + Z_{0,0,0} \\ &= A_1 ij + A_2 j + A_3 i + A_4 \end{aligned} \tag{4.3}$$

4. repeat steps 1 to 3 to calculate the point $Z_{i,j,T}$:

$$\begin{aligned} Z_{i,j,T} &= \frac{Z_{N_1,j,T} - Z_{0,j,T}}{N_1} i + Z_{0,j,T} \\ &= \frac{(Z_{N_1,N_2,T} + Z_{0,0,T}) - (Z_{N_1,0,T} + Z_{0,N_2,T})}{N_1 N_2} ij \\ &\quad + \frac{(Z_{N_1,0,T} - Z_{0,0,T})}{N_1} i + \frac{(Z_{0,N_2,T} - Z_{0,0,T})}{N_2} j + Z_{0,0,T} \\ &= B_1 ij + B_2 j + B_3 i + B_4 \end{aligned} \tag{4.4}$$

5. calculate the value of $Z_{i,j,k}$ (which is a linear interpolation of $Z_{i,j,0}$ and $Z_{i,j,T}$:

$$Z_{i,j,k} = \frac{Z_{i,j,T} - Z_{i,j,0}}{T} k + Z_{i,j,0} \tag{4.5}$$

$$\begin{aligned} &= \frac{(B_1 - A_1)}{T} ijk + \frac{(B_2 - A_2)}{T} jk + \frac{(B_3 - A_3)}{T} ik \\ &\quad + \frac{(B_4 - A_4)}{T} k + A_1 ij + A_2 j + A_3 i + A_4 \end{aligned} \tag{4.6}$$

The trilinear interpolating function of Equation 4.6 was calculated by interpolation in the horizontal, vertical, and then the temporal directions. It can be shown that the order of the interpolations does not matter—Equation 4.6 will always be the result. Even though the trilinear interpolator is composed of multiple linear operations, it no longer a simple linear function but is a third order three dimensional spline function.

The trilinear interpolator can also be realized as a 3-D FIR filter. Using the linear filter defined in Equation 2.2 it is possible to perform horizontal, vertical, and then temporal linear interpolations. The impulse response of the trilinear interpolating filter is equal to the convolution of the three 1-D impulse responses $h_{N_1}(n_1)$, $h_{N_2}(n_2)$, and $h_T(t)$ in three dimensions. That is,

$$\begin{aligned} h_{N_1,N_2,T}(n_1,n_2,t) &= h_{N_1}(n_1,n_2,t) * h_{N_2}(n_1,n_2,t) * h_T(n_1,n_2,t) \\ &= [h_{N_1}(n_1)\cdot\delta(n_2)\cdot\delta(t)] * [h_{N_2}(n_2)\cdot\delta(n_1)\cdot\delta(t)] \\ &\quad * [h_T(t)\cdot\delta(n_1)\cdot\delta(n_2)] \\ &= h_{N_1}(n_1)\cdot h_{N_2}(n_2)\cdot h_T(t) \end{aligned} \tag{4.7}$$

and substituting the definitions for $h_{N_1}(n_1)$, $h_{N_2}(n_2)$, and $h_T(t)$ yields:

$$h_{N_1,N_2,T}(n_1,n_2,t) = \begin{cases} \left.\begin{array}{l} 1-\left|\frac{n_1}{N_1}\right|-\left|\frac{n_2}{N_2}\right|-\left|\frac{t}{T}\right|+ \\ \left|\frac{n_1 n_2}{N_1 N_2}\right|+\left|\frac{n_1 t}{N_1 T}\right|+\left|\frac{n_2 t}{N_2 T}\right|+ \\ \left|\frac{n_1 n_2 t}{N_1 N_2 T}\right| \end{array}\right\} & \text{if } |n_1|<N_1,\ |n_2|<N_2, \\ & \quad \text{and } |t|<T \\ 0 & \text{otherwise} \end{cases} \tag{4.8}$$

As can be seen, the impulse response of the trilinear interpolating filter, $h_{N_1,N_2,T}(n_1,n_2,t)$, has a region of support that is a $(N_1+1)\times(N_2+1)\times(T+1)$ rectangular parallelepiped.

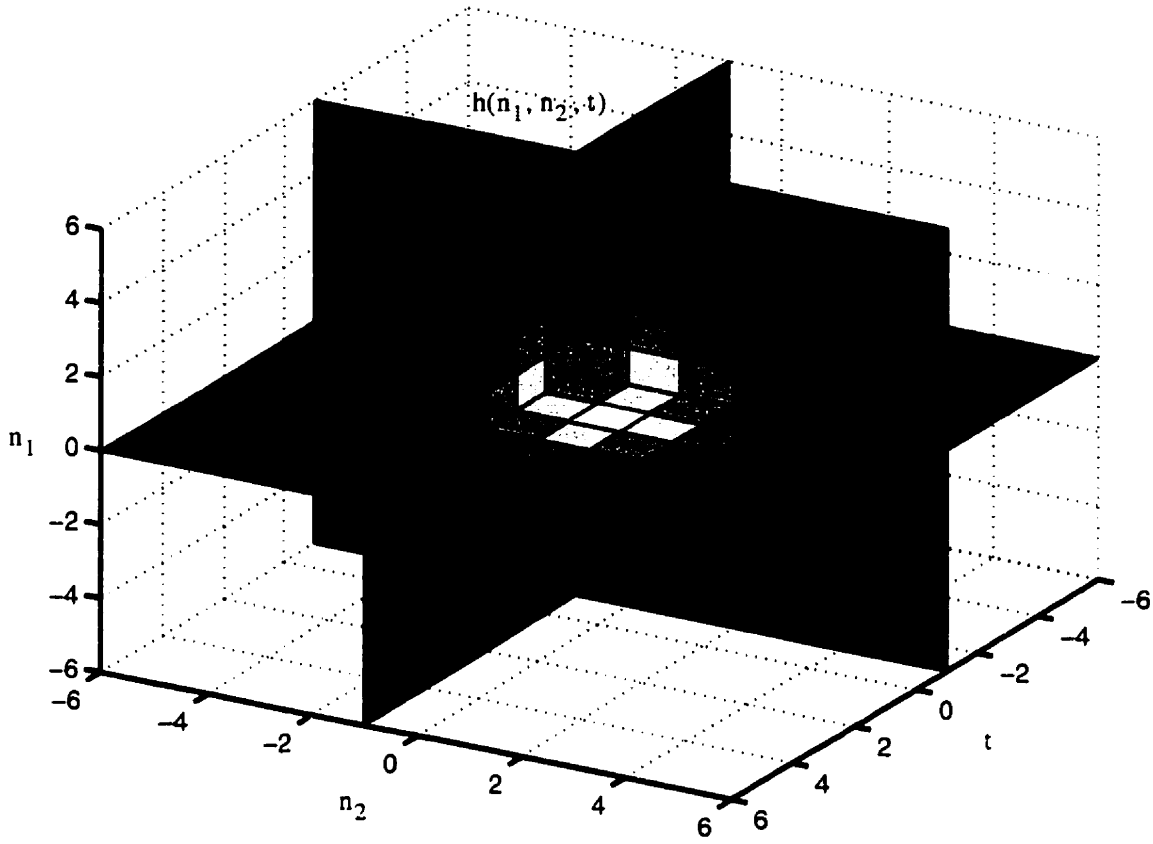


Figure 4.3: Impulse response of the trilinear interpolating filter $h_{4,4,4}$.

The visualization of 3-D signals can be somewhat difficult because four dimensions are required: each of the three independent variables and the one dependent variable requires an axis. This can be remedied by using three axis for the dependent variables and varying the colour (or greyscale) within the three dimensional space in order to represent different values of the dependent variable. This is demonstrated in Figure 4.3, where the impulse response $h_{4,4,4}$ is shown using lighter shades of grey representing larger values. In order to properly see the various shades within the 3-D volume, the figure shows the greyscale values lying on planar slices through the volume.

Given a 3-D input signal, $x(n_1, n_2, t)$, having been up-sampled by factors N_1,

N_2, and T in the vertical, horizontal, and temporal directions respectively, intermediate values can be interpolated via the trilinear interpolating filter. This is done by simply performing a 3-D convolution of the input signal with the impulse response of the filter:

$$\begin{aligned} y(n_1, n_2, t) &= h_{N_1,N_2,T}(n_1, n_2, t) * x(n_1, n_2, t) \\ &= \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} h_{N_1,N_2,T}(i, j, k) x(n_1 - i, n_2 - j, t - k) \end{aligned} \quad (4.9)$$

Furthermore, the limited region of support of $h_{N_1,N_2,T}(i, j, k)$ results in the simpler equation

$$y(n_1, n_2, t) = \sum_{i=-N_1+1}^{N_1-1} \sum_{j=-N_2+1}^{N_2-1} \sum_{k=-T+1}^{T-1} h_{N_1,N_2,T}(i, j, k) x(n_1 - i, n_2 - j, t - k) \quad (4.10)$$

Again, as with the 1-D and 2-D cases, Equations 4.6 and 4.10 will result in identical output given the same input. That is,

$$Z_{n_1,n_2,t} = y(n_1, n_2, t) \quad \text{for } 0 \leq n_1 \leq N_1,\ 0 \leq n_2 \leq N_2, \text{ and } 0 \leq t \leq T \quad (4.11)$$

Also, when using these two methods for piecewise trilinear interpolation of a 3-D upsampled signal, they will produce the same output within the input signal's domain.

4.2.1 Frequency Response

Using the equation for the impulse response of the trilinear interpolating filter (Equation 4.10), it is possible to calculate the three dimensional frequency response. However, as in Section 2.3.1, the frequency response can be more easily calculated by using the 1-D frequency response of the linear interpolating filter. Due to the trilinear interpolating filter having been created by filtering in the horizontal, vertical, and temporal directions separately, the 3-D frequency response is simply the product of

the 1-D responses:

$$\begin{aligned} H_{N_1,N_2,T}(e^{j\omega_1}, e^{j\omega_2}, e^{j\omega_3}) &= H_{N_1}(e^{j\omega_1}, e^{j\omega_2}, e^{j\omega_3}) \\ &\quad \cdot H_{N_2}(e^{j\omega_1}, e^{j\omega_2}, e^{j\omega_2}) \\ &\quad \cdot H_T(e^{j\omega_1}, e^{j\omega_2}, e^{j\omega_2}) \end{aligned} \tag{4.12}$$

Again, knowing that $H_N(e^{j\omega_1}, e^{j\omega_2}, e^{j\omega_2}) = H_N(e^{j\omega})$ and the definition of $H_N(e^{j\omega})$ from Equation 2.8, the frequency response of the trilinear interpolator is then

$$\begin{aligned} H_{N_1,N_2,T}(e^{j\omega_1}, e^{j\omega_2}, e^{j\omega_3}) &= \left[1 + 2\sum_{l=1}^{N_1-1}\left(1 - \frac{l}{N_1}\right)\cos l\omega_1\right] \\ &\quad \cdot \left[1 + 2\sum_{m=1}^{N_2-1}\left(1 - \frac{m}{N_2}\right)\cos m\omega_2\right] \\ &\quad \cdot \left[1 + 2\sum_{n=1}^{T-1}\left(1 - \frac{n}{T}\right)\cos n\omega_2\right] \end{aligned} \tag{4.13}$$

It may not be directly evident from Equation 4.13, but the trilinear interpolator is a 3-D lowpass filter. This follows from the fact that it is separable into three 1-D lowpass filters. The bandwidth in the ω_1, ω_2, and ω_3 directions can be adjusted by changing the values of N_1, N_2, and T respectively. Increasing these parameter values results in decreasing the bandwidth of the filter in their respective directions. Showing the frequency response of the filter graphically is somewhat difficult to do directly, so the volumetric slice method, as used to show the 3-D impulse response, is used to depict the magnitude frequency response. An example showing the frequency response, $H_{4,4,4}(e^{j\omega_1}, e^{j\omega_2}, e^{j\omega_3})$, of the trilinear interpolating filter $h_{4,4,4}$ is shown in Figure 4.4. The lowpass behavior is somewhat evident as shown by the bright centre lobe: the side lobes of the response are also visible. Comparing the 3-D magnitude frequency response with the 1-D response h_4 shown in Figure 2.5(c), the similarities (i.e. lowpass behavior and side lobes) between the two are evident.

The trilinear interpolator, having an easily adjustable lowpass behavior in

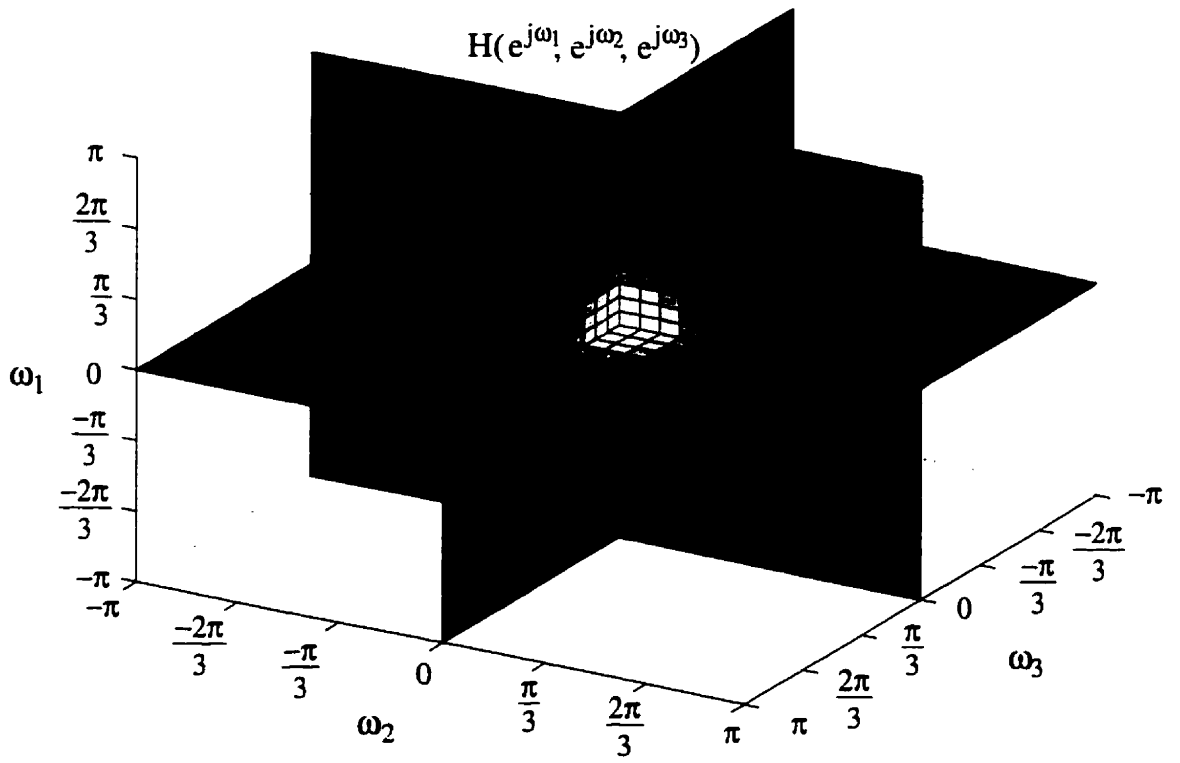


Figure 4.4: Magnitude frequency response of $h_{4,4,4}(n_1, n_2, t)$.

three dimensions, is useful for removing unwanted spectral images resulting from up-sampling a 3-D signal. Just as in Section 2.2.1.1, where it was shown that up-sampling by a factor N results in the same number of spectral images, up-sampling a 3-D signal by factors N_1, N_2, and T in the n_1, n_2, and t directions, respectively, results in N_1N_2T spectral images—where $N_1N_2T - 1$ of these images are unwanted. These unwanted spectral copies can be removed by the trilinear interpolator $h_{N_1,N_2,T}$ while the desired spectral component of the signal that lies near the origin can be retained—all because of the frequency characteristics of the filter.

4.2.2 Implementation

Implementation of the 3-D trilinear interpolator involves many of the same ideas that were presented in Section 3.1. However, there are some specific issues that are somewhat different from the 2-D implementation. The basic implementation is based on the trilinear interpolation function defined in Equation 4.6 instead of the trilinear interpolating filter of Equation 4.8. This is mainly due to the computational load required by the filter and the storage needs of the filter coefficients (or the computational needs of the coefficients if calculated as required). The filter, however, was useful in examining the frequency characteristics of the trilinear interpolator.

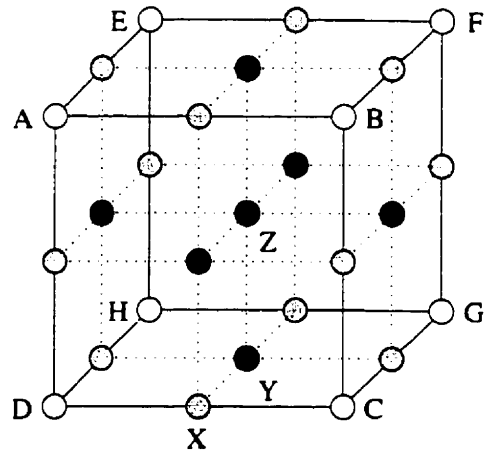
Trilinear interpolation can be performed directly by using the 3-D function in Equation 4.6. However, doing this results in a large number of calculations per voxel: for each volume element, 26 additions, 13 multiplications, and 13 divisions are required. To reduce the computational load, the calculation can be done (or approximated) by a number of other methods—mostly consisting of multi-step interpolations. The methods covered here are similar to the ones discussed in Chapter 3. The three step (as seen in Figure 4.2) interpolation methods examined are: The Bresenham,

floating point divide, integer divide, floating point DDA, fixed point DDA, and recursive 1-D methods. There is also a recursive 3-D trilinear interpolation method that will be examined.

The two step bilinear interpolation methods discussed in Chapter 3 can easily be extended into three dimensions in order to implement a trilinear interpolator. Individual linear interpolations are used to interpolate the top and bottom rows of the first and last frame. Then they are used again to interpolate between the top and bottom rows in the end frames. Finally, linear interpolations are used to calculate the values between the first and last frame within the volume. Using the three step method vastly reduces the number of calculations over the one step direct method (Equation 4.6). Depending on the implementation of the 1-D linear interpolator, the computational load can be reduced further. However, with a decrease in the computational complexity, the error between the resulting 3-D interpolation and actual trilinear interpolation can increase.

A recursive method shown in [17] and [25] can be also be used to perform the trilinear interpolation—similar to the one step recursive 2-D bilinear interpolator described in Section 3.1. By using only integer additions and bit shifts, the voxels within the 3-D volume can easily be calculated. The operation of the recursive 3-D interpolator is shown in Figure 4.5. After the centre and mid-points are calculated, eight new blocks are created to which the algorithm is applied again. This is recursively performed until the voxel level is reached.

As mentioned before, there can be some trade offs between accuracy and speed. In Section 3.1 the advantages and disadvantages of each linear interpolation implementation were discussed. As all the trilinear interpolation implementations (except the recursive 3-D method) make use of multiple 1-D interpolations these advantages and disadvantages still apply.



○ Starting vertices (A,B,C,D,E,F,G,H)

⊙ Interpolated segment mid-points (e.g.: $X = \frac{D+C}{2}$)

● Interpolated face's centre point (e.g.: $Y = \frac{C+D+G+H}{4}$)

● Interpolated block's central point (e.g.: $Z = \frac{A+B+C+D+E+F+G+H}{8}$)

Figure 4.5: Recursive 3-D trilinear interpolation [17, 25].

Table 4.1: Comparison between the average mean squared error of different trilinear interpolation methods over 100 random $64 \times 64 \times 64$ blocks.

Interpolation Method	Average MSE	
	Method vs. Actual	Method vs. Quantized Actual
Floating Point Divide	0.083	0.0
Floating Point DDA	0.083	0.0
Fixed Point DDA	0.083	6.9×10^{-7}
Bresenham	0.40	0.48
Integer Divide	0.68	0.74
Recursive 3-D[a]	2.8	2.8
Recursive 1-D	5.0	4.8

[a]One-step interpolation

To observe the accuracy of the various trilinear interpolation methods, 100 $64 \times 64 \times 64$ blocks having random corner values were interpolated and compared with the actual trilinear interpolated values generated by Equation 4.6. Table 4.1 shows the average MSE of the different trilinear interpolators. When using an interpolator, the output is quantized (to 8 bits/voxel in this discussion) in order to view on a display device. Also, the originating sequence is usually quantized and represented by a fixed number of bits/voxel. A slightly different error measurement uses the 8-bit output values of the trilinear interpolators and compares it to the 8-bit quantized output of Equation 4.6. The average MSE based on this measurement is also shown in Table 4.1.

The calculation speed of the trilinear interpolator is also important. The calculation times for the different interpolation methods of a fixed number of voxels contained in various block sizes are shown in Table 4.2. These processing times were based on a Pentium CPU running at 166MHz.

As can be seen from Tables 4.1 and 4.2, the different trilinear interpolation implementations perform relatively similar to the bilinear methods in MSE perfor-

Table 4.2: Processing times of different block sizes (containing the same number of voxels) for various trilinear interpolation methods.

Interpolation Method	Time (seconds)		
	100 blocks ($64 \times 64 \times 64$)	6400 blocks ($16 \times 16 \times 16$)	409600 blocks ($4 \times 4 \times 4$)
Actual (Equation 4.6)	70.80	68.28	60.02
Bresenham	12.37	36.84	199.59
Floating Point Divide	13.65	14.36	18.25
Integer Divide	10.99	12.06	17.06
Recursive 1-D	12.60	12.35	11.41
Recursive 3-D	13.83	11.44	6.14
Floating Point DDA	5.53	6.12	10.43
Fixed Point DDA	2.56	3.18	7.13

mance and speed of operation. Again, it can be seen that the most accurate methods are the floating point ones, followed closely by the fixed point DDA method. With regards to computation speed, it can be seen that some methods are dominated by their setup time versus their computation time. The Bresenham method is a prime example of this. For a large number of small blocks, its calculation time is very slow. The calculation times for the actual trilinear interpolator make it obvious why it is not the most desirable method: even though it is the most accurate method, it is also the slowest. Overall, the fixed point DDA is still the fastest method. It should be noted that the recursive 3-D method works slightly faster on smaller blocks. However, it also performs quite poorly in the MSE sense. These results show that the three step fixed point DDA is the best choice for implementing the trilinear interpolator.

4.3 Adaptive 3-D Sampling Grid

Given a three dimensional signal it is possible to reduce the number of samples representing the signal by simply sub-sampling the signal by a constant factor in each

direction. Then, when reconstructing the signal, the signal must be up-sampled by the same factors followed by trilinear interpolation to fill in values between the transmitted/stored sample values. This simple codec can significantly reduce the required number of bits/voxel. However, as with the 2-D case discussed in Section 2.4.1, using this 3-D fixed sampling grid results in frequency aliasing effects and the loss of high frequency components in both of the spatial directions and the temporal direction.

4.3.1 Non-Uniform Grid Generation

To overcome the problems associated with a fixed sampling grid, a non-uniform sampling grid can be used. With a non-uniform sampling grid, volumes that contain high frequency components can be assigned more voxels than volumes having low frequencies. In this way, the localized sub-sampling factors can be adjusted to reduce the amount of aliasing or to avoid aliasing altogether.

The generation of the adaptive non-uniform 3-D sampling grid is similar to the 2-D grid grid generation method described in Section 2.5. However, the addition of the temporal dimension requires some modifications and also allows some optimizations to be made. An image sequence, in general, is not constrained in the temporal direction (or at least the temporal length of the sequence is much larger than the spatial lengths) so when coding, it is not possible to look at the entire image sequence. Thus, the coder processes groups of frames (frame stacks) in order to accommodate the large temporal sizes. The non-uniform 3-D grid contains larger number of vertices in regions containing intense activity—spatial (edges, textures) or temporal (movement). The frame stack is first subdivided into blocks of maximum dimension: $D_{max} = (2^s + 1) \times (2^s + 1) \times (2^t + 1)$. Then, starting with a block of maximum dimensions with only eight vertices, the algorithm recursively subdivides

the volume in the spatial or temporal domain, guided by the calculation of an error measurement [17]. When a spatial split is made, four equi-sized sub-blocks are created with the dimensions $(2^{s-1}+1) \times (2^{s-1}+1) \times (2^t+1)$; consequently, ten new vertices are generated by the spatial split. A temporal split results in the generation of two equally sized sub-blocks having dimensions $(2^s+1) \times (2^s+1) \times (2^{t-1}+1)$ and creates four new vertices. Splitting in this fashion always results in vertices that are shared with neighboring blocks.

Starting with a block with dimensions $(2^s+1) \times (2^s+1) \times (2^t+1)$, it is possible to recursively split it down to the voxel level. That is, the final block sizes will all be $2 \times 2 \times 2$. Lossless coding is then possible, as there will be no error introduced by trilinear interpolation. Actually, in this case, no interpolation would even be necessary. In practice, the spatial dimensions of the original frame stack sequence will rarely fit the pattern $(2^s+1) \times (2^s+1)$. So cleaving the block into four identically sized sub-blocks is not possible and approximately equi-sized sub-blocks have to be acceptable. Splitting down to the voxel level is then not possible using the above splitting method because one length along its spatial dimension will reach its minimum size before the other dimension will. This is not a major problem, as there is less demand for lossless video coding than for lossless image coding. However, if lossless coding is a requirement, the splitting process can be modified to a method similar to the one described in Section 3.2.2 where spatial splits are performed by bisection rather than splitting into four parts.

By breaking the subdivision process into separate temporal and spatial splits, as opposed to splitting each block into eight sub-blocks, the computational requirements of the error evaluation function can be reduced significantly because a complete approximation of the block is not necessary. When deciding if a spatial split has to be performed, only one frame of the block has to be interpolated. To decide if a

temporal split is necessary, only the linear interpolations of the four pairs of corresponding vertices in the time direction are required. These valuable simplifications as suggested in [17] are justified by the following considerations:

1. Assuming there is no movement in the block, the first face can be considered to be representative of the entire block, so decisions for spatial splitting can be based entirely on it.

2. If the block is spatially smooth (i.e. there are no high frequencies in the spatial directions), the four corner vertices of each frame within the block can be considered representative of the entire frame, so the temporal splitting decision can be made based only on these vertices.

4.3.2 Grid Representation

In order to reconstruct the image sequence, the decoder must be able to recreate the splitting process performed by the coder. Thus, the splitting information must be represented in an efficient manner that is able to be stored or transmitted. The splitting hierarchy lends itself well to a tree structure. Starting with a node that represents the current frame stack, if a split is necessary within the current node, the node becomes a parent to multiple child nodes which represent the sub-blocks. The number of children depends on the type of split: four child nodes for a spatial split and two child nodes for a temporal split. This results in a quaternary-binary tree. After all the splitting is performed, the nodes that have not been split will have no children—they are leaf nodes of the tree—and it is the sub-blocks that are represented by these children that are to be interpolated when reconstruction occurs.

Just as in the 2-D image coder, the nodes in the tree can be described by a code that is a variable length string of bits. The entire tree can then be represented

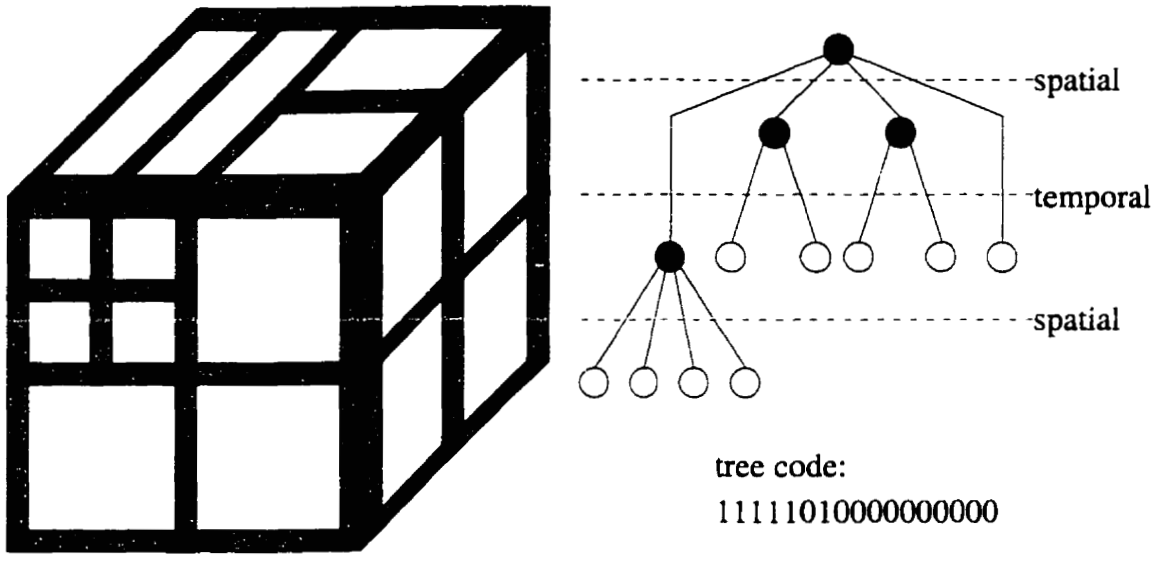


Figure 4.6: A three dimensional non-uniform grid and its representative tree structure [17, 25, 26, 27].

by a large string of bits made by the concatenation of the bits for each node through a traversal of the tree. The individual node bits can be chosen to reduce the length of the tree string. It can be seen that the leaf nodes are always the most abundant in the tree. Therefore, by representing the leaves by the shortest code, an overall savings in storage can be achieved. Figure 4.6 shows a subdivided block along with the quaternary-binary tree and the corresponding bit string that represents it. In the tree code, which was created through a depth-first traversal of the tree, a "0" indicates no split (a leaf node) while a "1" followed by a "1" or a "0" indicates a spatial or a temporal split respectively. Some of the bits in the tree code can be removed for further data savings. If a leaf size is equal or less than the minimum dimensions parameter, the "0" indicating that there is no split is not required because the decoder knows that the block cannot be split any further and does not need to be informed explicitly.

By observing the bit string representing the tree in Figure 4.6, it can be seen that there are many repeated patterns of digits. Thus, the entire tree code can be

compressed further by compressing the information stored in the bit string with the dictionary based LZW coder—exactly as was done to compress the tree in the bilinear interpolation based image coder. Again, a lossless information compressor must be used since the tree information is crucial to the reconstruction of the image sequence. The combination of the variable length node codes along with the LZW compressor results in a tree representation that is, on average, much less than 1 bits per tree node.

If the frame stack does not have spatial dimensions that are of the form $(2^s + 1) \times (2^s + 1)$, lossless coding of the sequence is not possible. Usually lossless coding is not a requirement for video coding as the amount of information can still be quite formidable. If lossy coding is not acceptable, the quaternary-binary tree can be replaced by a binary tree in which the split nodes can be either representing splits in the n_1, n_2, or t directions. In this way, final block sizes of 2×2 can always be realized and lossless coding is possible. Using this modification will result in a larger splitting tree since more splits will be necessary. Also, the number of bits required to represent each tree node will increase since there are more types of nodes: no split, horizontal split, vertical split, and temporal split. Thus, lossless coding of an image sequence is possible if a gain in the overall bits/voxel is acceptable.

4.3.3 Interpolation Discontinuities

When the reconstructing the image sequence across the whole grid some problems may arise. It is noted in [17], [25], and [27] that a tree structure may arise in which some vertex locations do not match with those of the neighboring blocks which can cause discontinuities in the interpolated output. This problem may adversely affect the visual quality of the image sequence if not handled correctly. In

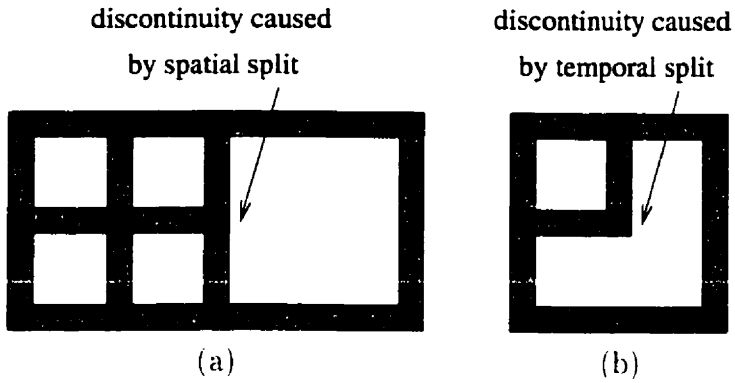


Figure 4.7: Examples of possible grid interpolation discontinuities: (a) discontinuity caused by a spatial split and (b) discontinuity caused by a temporal split [17, 25, 26, 27].

general, interpolation discontinuities between larger blocks are more visually apparent than discontinuities between small blocks. In Figure 4.7 two typical examples of interpolation discontinuity points.

When the global interpolation is performed, a breadth-first traversal of the tree is preferred because large blocks will be interpolated before smaller neighboring blocks. The higher detail along the edges of the smaller blocks will overwrite the low detailed edges of the neighboring large blocks. This results in a higher PSNR and can somewhat reduce the effects of interpolation discontinuities.

To completely remove the effects of interpolation discontinuities, the image sequence decoder can perform the block size equalization (BSE) procedure shown for the 2-D image decoder in Section 3.2.4. Before performing the global interpolation using the tree structure, the decoder must perform a top-down iterative algorithm that splits larger blocks and computes the necessary centre and mid-points via interpolation for the new sub-blocks. This can be done until all leaf nodes are at the same level in the tree which results in sub-blocks that are all approximately the same size. Then, when the global interpolation is performed, no interpolation discontinuities will

occur.

To remove all discontinuity points, block size equalization must ensure that all blocks are split down to the minimum block dimensions specified by the coder. This can result in a large computational load because of the potentially large number of blocks and voxels to be considered within a frame stack—much larger than for the 2-D BSE algorithm. So it is recommended that the equalization is only performed so far down the tree that too much processing time is not required. Since discontinuities occurring between large blocks are much more noticeable, the sub-optimal block size equalization procedure can be sufficient. If decoding time is very crucial and the small visual defects caused by interpolation discontinuities are acceptable, the block size equalization step can be skipped altogether.

4.3.4 Vertex Representation

The sample points lying on the grid vertices are required by the decoder as corner points for the trilinear interpolation process. Therefore, they must be stored and sent to the decoder along with the information describing the splitting tree. Since the splitting tree contains the structure of the image sequence and the vertices only contain intensities, the vertices can be coded in a lossy manner to increase the compression ratio with only minor effects to the reconstructed video quality.

The DPCM-Huffman coder described in Section 3.3 is used again to code the 3-D vertex values—with some modifications. The DPCM coder takes advantage of any correlation that exists between adjacent samples and attempts to reduce the dynamic range of the signal. This then can result in an entropy reduction that can be exploited by the Huffman entropy coder to reduce the overall information requirements of the vertex values further. The prediction performed by the DPCM

coder is crucial to its performance: the better the prediction is, the lower the output entropy of the signal can be. The 2-D DPCM coder uses samples from above and to the left of the current position in the image to generate a predicted value (see Figure 3.5). Only these samples can be used in prediction because all other values will be unavailable to the decoder. For 3-D DPCM coding, samples above and to the left, as well as samples from previous frames can be used in the prediction process. Thus, the prediction, $\hat{x}(i,j,k)$, of the value $x(i,j,k)$ is computed by the N-th order 3-D predictor:

$$\begin{aligned}\hat{x}(i,j,k) &= \sum_{n=1}^{N} c_{0,n,0}\, x(i,j-n,k) \\ &\quad + \sum_{m=1}^{N}\sum_{n=-N}^{N} c_{m,n,0}\, x(i-m,j-n,k) \\ &\quad + \sum_{p=1}^{N}\sum_{m=-N}^{N}\sum_{n=-N}^{N} c_{m,n,p}\, x(i-m,j-n,k-p)\end{aligned} \tag{4.14}$$

where $c_{m,n,p}$ are the prediction coefficients. For minimum prediction error, the prediction coefficients are chosen so that $\sum c_{m,n,p} = 1$.

In this video coder, a first order predictor is used because of the potential large number of calculations required for a higher order, more complex, predictor. This results in 13 prediction coefficients: 1 to the left of the current position, 3 above, and 9 in the previous frame. The simplest prediction coefficient selection is also made: all the coefficients are made equal to $\frac{1}{13}$. This makes the predicted value simply the average of its neighboring values.

There is one major problem with DPCM coding of the vertices on the non-uniform grid. Samples above, to the left, and on the previous frame will not always be available and may also be different distances away from the current sample. If a vertex does not exist immediately next to the current location, the nearest neighboring vertex in that direction is used instead. However, if no suitable replacement can be found,

that value can be omitted and the prediction coefficients can be adjusted (i.e. the value is simply removed from the calculation of the average surrounding vertices).

The above describes results in lossless coding of the vertices. The overall bit rate required for the vertices can be reduced further by first quantizing the vertex values before DPCM-Huffman coding. For moderate quantization factors the number of bits per vertex can be reduced dramatically while the effects of vertex quantization are not noticeable. Visual artifacts, such as the "onion ring" effect, that are usually noticeable when quantizing images or image sequences, are avoided when quantizing the vertex values because the trilinear interpolation performed by the decoder fills in intermediate values at the maximum precision provided by the interpolator and are not fixed at the same precision as output by the vertex quantizer.

4.4 Least-Squares Trilinear Interpolation (LSTI)

When coding an image sequence using the above method, voxel values that lie within a block to be interpolated are not taken into account during interpolation. This can cause an increase in the overall error of the reconstructed output for a number of reasons. If there is any noise present in the original image sequence, the noise imposed on a vertex, when interpolated, will be spread across all neighboring blocks. Furthermore, when a high contrast spatial or temporal edge cuts through a block and only affects one or two vertices, it results in the edge being smeared throughout the entire block.

The image sequence coder can bring about a higher PSNR in the decoded image sequence by performing an optimization that minimizes, in the least-squares sense, the trilinear interpolation error by modifying the corner values of each block to be interpolated. Furthermore, since the optimization is performed by the coder,

there is no change required to the decoder and no extra calculations are required in the decoding stage. This least-squares trilinear interpolation (LSTI) method is quite similar to the LSBI method described in Chapter 3, except a third dimension is added into the calculation.

In Section 3.4 the sub-optimal least-squares optimization that is performed on a per-block basis was shown to have the solution

$$\begin{aligned} \mathbf{x} &= \left(\mathbf{B}^{\mathrm{T}}\mathbf{B}\right)^{-1}\mathbf{B}^{\mathrm{T}}\mathbf{y} \\ &= \mathbf{S}\mathbf{y} \end{aligned} \tag{4.15}$$

where $\mathbf{x}$ is a vector containing the optimized corner values, $\mathbf{B}$ is the interpolation matrix, and $\mathbf{y}$ contains all the original values in the image block. The formulation of Equation 4.15 is not dimension specific. Therefore, the optimization is also equally valid for a three dimensional block of samples. This means that for LSTI, the vector $\mathbf{x}$ will contain the eight optimized corner points to be used by the decoder instead of the values found in the original image sequence.

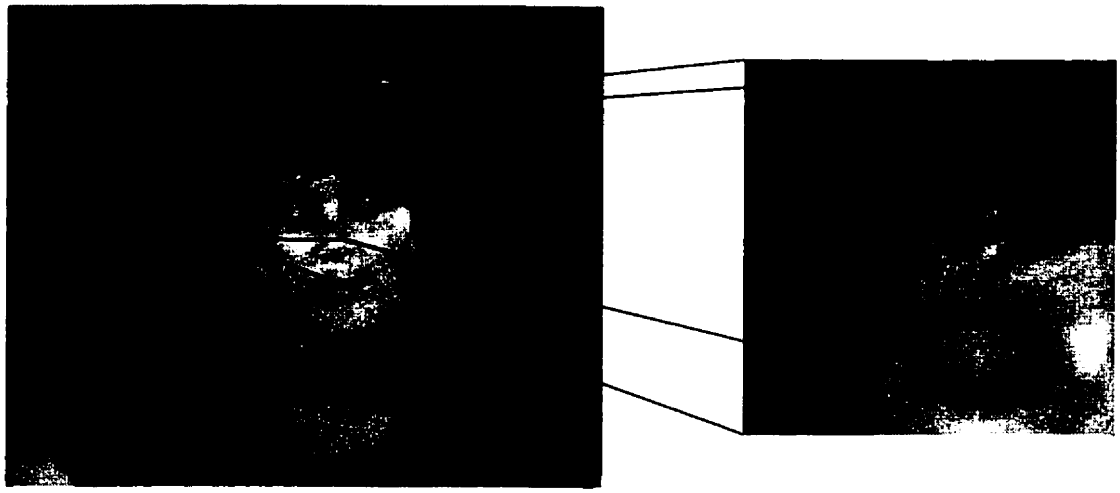
The LSTI method is optimal in the least-squared sense if all eight optimized corner points can be retained for each block. This is not desirable for image sequence data compression since it will result in a huge increase in the number of vertices to be transmitted or stored. As with the LSBI method, the LSTI optimized corner values can be averaged with the optimized values from neighboring blocks. This results in a higher quality reconstruction without increasing the amount of data storage or transmission bandwidth. For an even more accurate reconstruction, a weighted average based on the volume of the blocks that share common corners can be used to calculate the optimized output corner values. When this is done, for a given vertex location, the LSTI optimized corner value from a large block will be given a larger weight than the LSTI optimized value from a smaller neighboring block. Thus, the

actual value that is output will be closer to the optimized corner value calculated from the large block. After reconstruction, the larger blocks will have less interpolation error which results in an overall higher PSNR for the entire image sequence. This combination of LSTI and adaptive sub-sampling will be denoted as Adaptive Least-Squares Trilinear Interpolation (ALSTI).

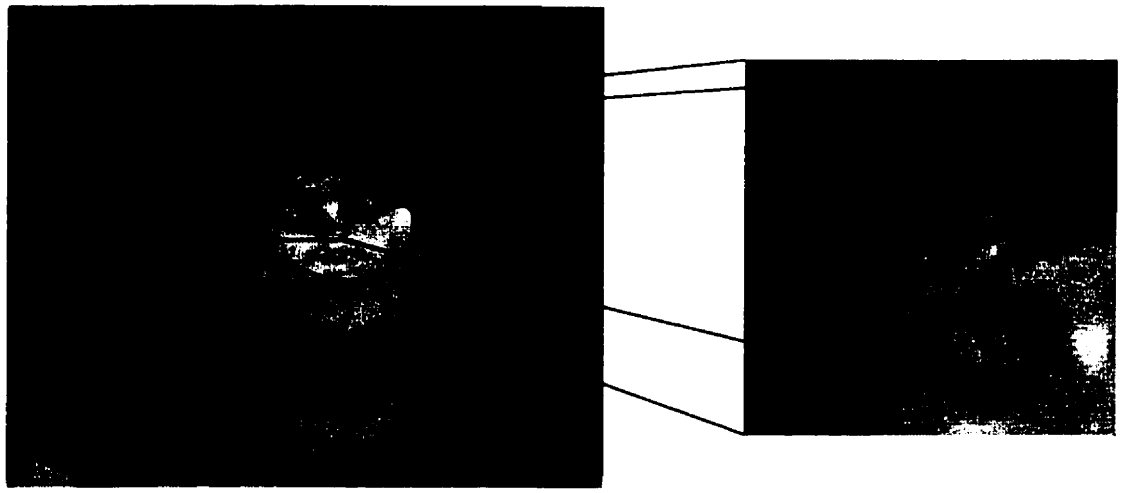
An example illustrating the differences between two reconstructed frames from the "Miss America" sequence, one with and one without LSTI optimization, is shown in Figure 4.8. The images show a point in the sequence where the high contrast edge between her hair and face moves through a block just above her left eye. When no LSTI optimization is performed, the output is as shown in Figure 4.8(a). However, after LSTI optimization is done, a higher quality reconstruction results, as seen by the frame in Figure 4.8(b). The reconstructed output not only improves visually but also in terms of PSNR. In this case there is over a 1 dB increase in PSNR when the LSTI optimization is used (35.97 dB with LSTI versus 34.76 dB without).

In order for the image sequence coder to perform the LSTI optimization, the coefficients in the **S** matrix must either be calculated during the coding stage or calculated off-line and read into memory. Calculating the coefficients as needed is very computationally expensive and not practical. If the **S** matrix coefficients are pre-calculated, a very large amount of storage space is required—a fact that is amplified even further because coefficients are required for three dimensions.

This storage problem can be alleviated by treating the LSTI optimization as multiple 1-D LSLI optimizations, just as was done for the LSBI optimization in Section 3.4.1. Then, the coefficients can be stored treated and stored as 1-D FIR filter coefficients. In fact, the coefficients required for the LSTI calculations are exactly the same as those needed for the LSBI optimization because of the separation into 1-D operations (i.e. the same 1-D FIR filters are used in LSLI, LSBI, and LSTI optimization).



(a)



(b)

Figure 4.8: Reduction of interpolation error using LSTI. (a) Without LSTI interpolation artifacts are introduced when a high contrast edge cuts through a block. (b) With LSTI the effect of the edge on the reconstructed output is diminished.

4.5 Results and Discussion

Now that the operation of the image sequence codec has been described, the effects of the various parameters on the reconstructed output can be shown and discussed. The quality of the reconstructed output is based on a PSNR comparison between the output and the original image sequence. A commonly used measurement of coding efficiency is the number of bits per second required to transmit the sequence from coder to decoder. This measurement can sometimes be misleading as it is dependent on the frame rate of the original sequence. Therefore the information compression ability of the video coder is the average number of bits per voxel required to code the image sequence. Given the frame rate, it is simple to convert between bits/voxel and bits/second.

As shown in this chapter, there are many parameters involved in this particular coding method of image sequence data. Each parameter affects the quality, compression ratio, or computational speed in different ways. These parameters and their effects will be discussed and output examples will be shown in the following sections.

4.5.1 Effects of Spatial and Temporal Block Sizes

The minimum spatial block size and spatial MSE splitting tolerance will, obviously, affect the spatial quality while the corresponding temporal parameters affect the output quality in the temporal direction. In general, a low MSE splitting tolerance (i.e. splitting will occur even for small interpolation errors across a block) will result in large numbers of small blocks to be interpolated by the decoder.

To show and example of how large temporal and large spatial block sizes (or large temporal or spatial interpolation error) affect the interpolated output, the "football" image sequence is coded first using a large error tolerance for splitting in the

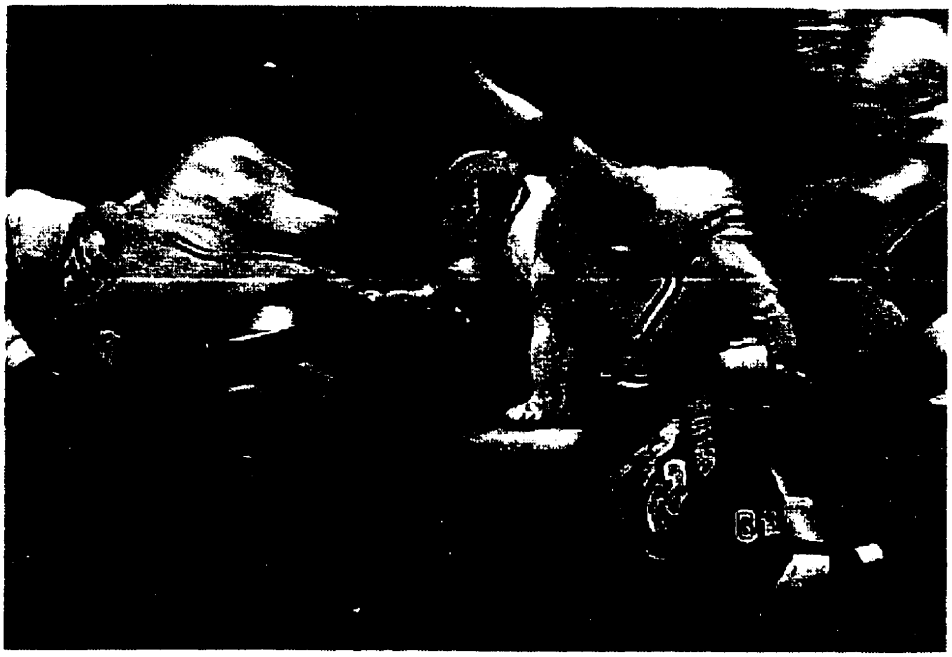


Figure 4.9: An original frame from the "football" image sequence.

spatial dimensions while keeping the temporal error tolerance small. The sequence is then coded again with low spatial error tolerance and high temporal error tolerance. Figure 4.9 shows an original frame of the "football" sequence. This sequence provides many challenges to a video codec system as it contains a large amount of motion as well as sharp edges (the lettering on the players' jerseys) and detailed textures (the grass). Figure 4.10 shows frames from the reconstructed "football" sequence when large spatial errors are allowed and large temporal errors are allowed. For a proper comparison, both sequences were coded at the same bit rate (1.69 bits/voxel).

As can be seen in Figure 4.10(a), the large spatial block sizes remove most of the high frequency information in each frame giving it a blurred appearance. However, it does contain large numbers of samples in the temporal direction. Even small amounts of motion can be seen from frame to frame. To contrast this case, Figure 4.10(b) shows how large temporal block sizes produce blurriness in the temporal direction

(a)

(b)

Figure 4.10: Example showing the effects of (a) large spatial and (b) temporal block sizes.

while small spatial block dimensions enable small details to be seen in each frame. The temporal blurriness is somewhat interesting as details from surrounding frames can be seen in the current frame giving it a somewhat cluttered appearance as moving objects fade in and out of view over a number of frames.

These are some extreme cases of large block sizes. The PSNR is quite low for both output sequences: 23.7 dB and 22.8 dB for the sequences represented by Figures 4.10(a) and 4.10(b) respectively. In practice, a balance between spatial and temporal error can produce a more pleasing output at a similar bit rate.

To illustrate the effects of the spatial and temporal MSE splitting thresholds, an image sequence can be coded and decoded multiple times with different threshold values. In Figure 4.11, the spatial MSE splitting threshold is swept across various values for each coding and decoding of the "western" sequence while holding the temporal splitting MSE constant. The "western" sequence has somewhat less motion and high frequency textures than the "football" sequence but is more complex than the "Miss America" sequence. This makes it a good sequence of average complexity to code and decode. Furthermore, it has spatial dimensions of 256×256 which somewhat eliminates the need for the modified splitting method described in Section 4.3.2.

The effects of the temporal MSE splitting threshold are shown in a similar manner: the temporal threshold is changed for each coding/decoding operation while the spatial threshold is held constant. For the "western" sequence, the resulting PSNR vs. bits/voxel curve for this operation is shown in Figure 4.12 for various spatial MSE threshold values.

It is easy to observe that as the splitting threshold (temporal or spatial) is increased, the quality of the image decreases along with a corresponding decrease in the number of bits/voxel. Both of these effects occur because when using a higher error tolerance, less splitting is performed resulting in larger blocks to be encoded.

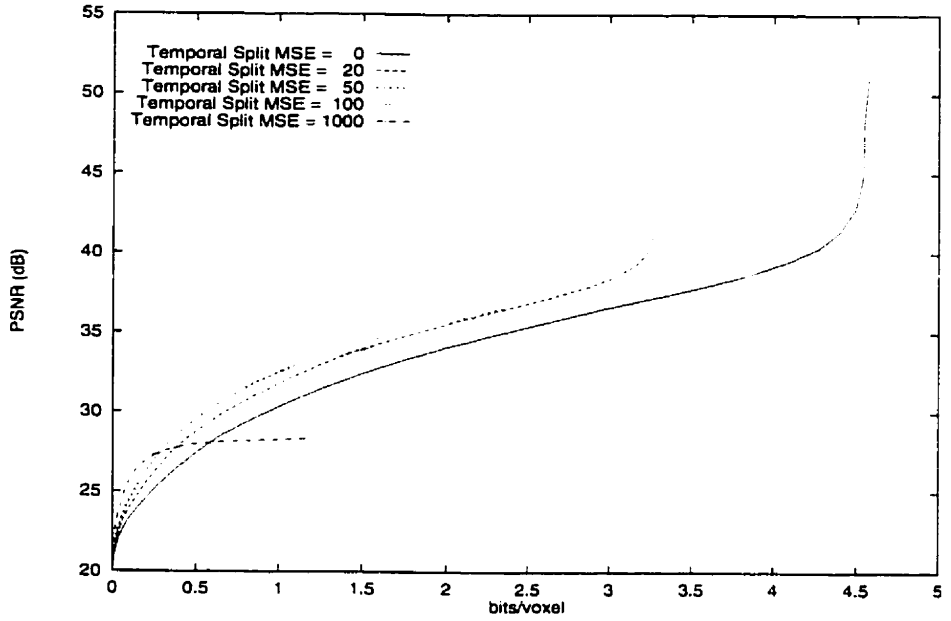


Figure 4.11: The "western" image sequence PSNR vs. bits/voxel during a sweep of the spatial MSE splitting threshold while the temporal splitting threshold is held constant.

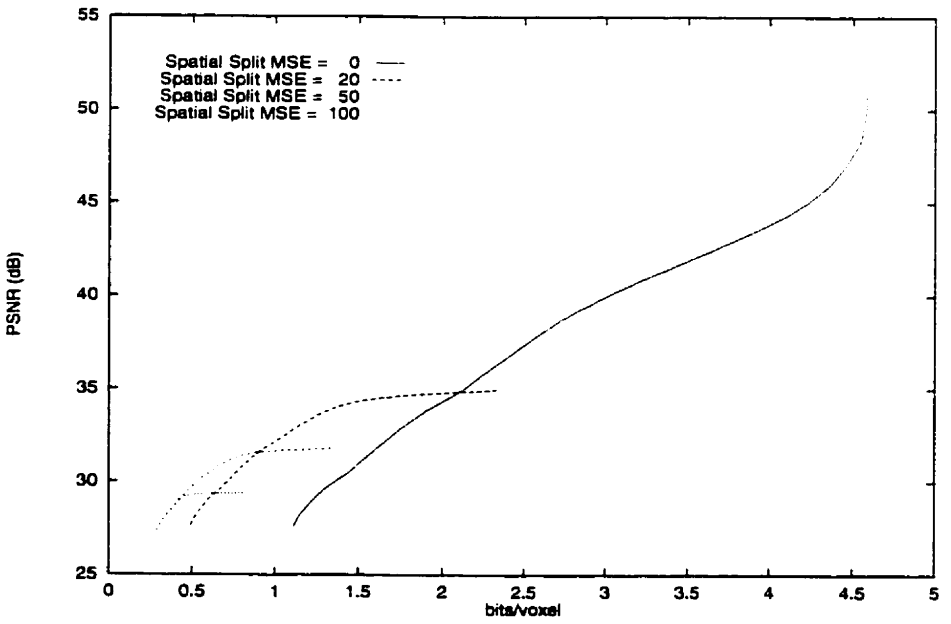


Figure 4.12: The "western" image sequence PSNR vs. bits/voxel during a sweep of the temporal MSE splitting threshold while the spatial splitting threshold is held constant.

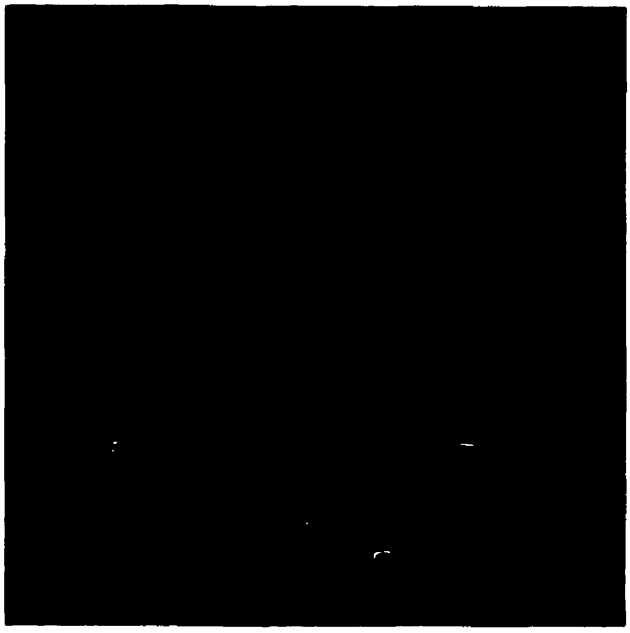
Larger blocks—and consequently, fewer blocks—generally result in a larger interpolation error and because there are fewer blocks, a lower bit rate can be achieved.

4.5.2 Effects of Interpolation Discontinuities

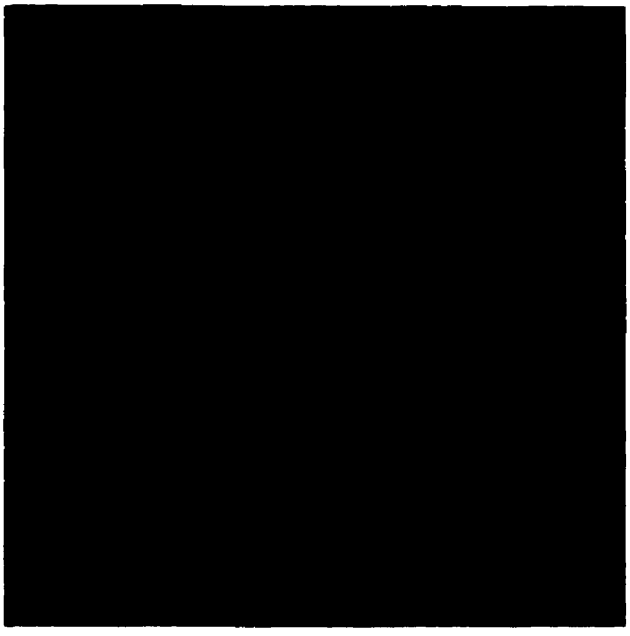
In Section 4.3.3 it was discussed how discontinuities in interpolation can arise at temporal and spatial block boundaries. It was also shown how the effects of these discontinuities can be minimized by block size equalization during the decoding process. This results in a higher quality output image sequence. For example, in Figure 4.13 two output frames are shown: one decoded normally while the other is decoded using BSE. From the output images, it is apparent that the block size equalization method improves the output quality; a gain of over 2 dB in PSNR is achieved.

In general, unless the minimum splitting dimensions have been globally reached, the BSE algorithm will improve the quality of the reconstruction. Figure 4.14 shows the results of BSE over a wide variety of combined temporal and spatial splitting thresholds when coding the "western" image sequence. As can be seen, when operating at the same bit rate, the PSNR is higher for the coder using BSE than the one that does not.

Since this algorithm can increase the memory and computational demands, the decoder can select various levels of recursion, or no recursion at all, for the equalization process. A lower level of recursion into the tree, until the minimum block size has been reached, will result in a higher quality output. Thus, if quick decoding is necessary, a trade off between quality and speed must be done. It is useful to note that this algorithm is done solely by the decoder. No change is required to the coder or the format of the compressed data.



(a)



(b)

Figure 4.13: Example showing the effects of block size equalization. Shown are reconstructed frames from the "western" sequence coded at 0.32 bits/voxel and decoded (a) without using and (b) using block size equalization. The resulting outputs have a PSNR of (a) 28.4 dB and (b) 30.8 dB.

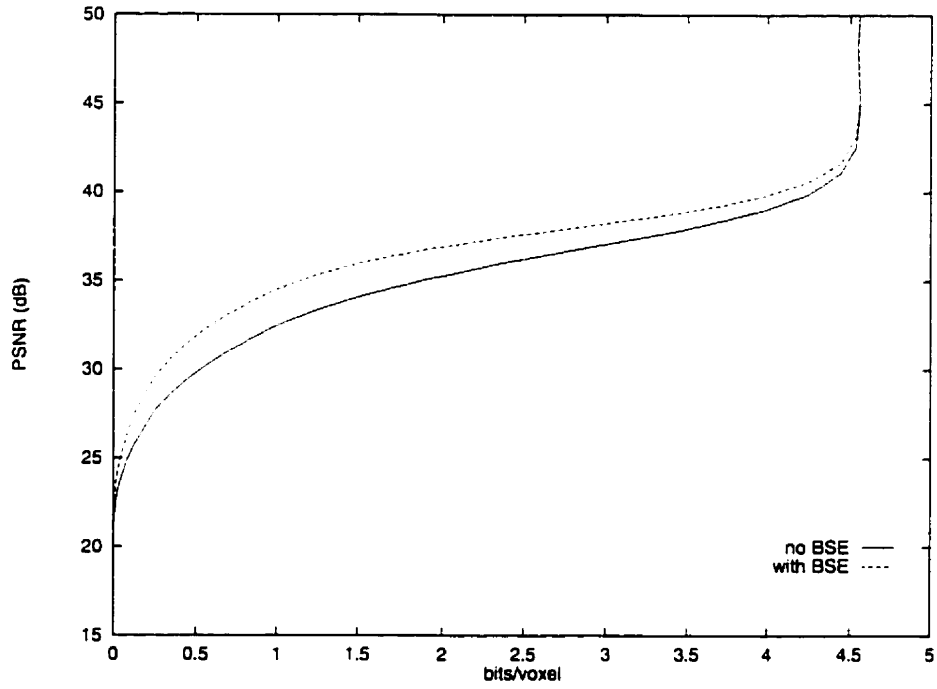


Figure 4.14: PSNR vs. bits/voxel during a sweep of the splitting MSE threshold for a BSE coder and a non-BSE coder operating on the "western" image sequence.

4.5.3 Effects of LSTI Optimization

Similar to the 2-D case, the least-squares trilinear interpolation optimization can be performed by the coder to result in a more accurate reconstruction by the decoder. This makes it possible to off-load some of the computations on to the coder which is desirable since fast decoding time is usually more often required than fast coding. Furthermore, no changes are needed to be made to the decoder or the bit stream format, enabling the decision to perform LSTI optimization entirely up to the coder and independent to the rest of the system.

As mentioned previously, the LSTI optimization does not drastically change the bits/voxel requirements of the image sequence: usually the changes are miniscule or even non-existent. Since the optimization is performed on a 3-D block of voxels, it improves both the spatial quality and the temporal quality of the reconstructed image sequence. It is difficult to show the temporal quality on a 2-D medium (i.e. this page), however the spatial quality improvement can be easily shown. One example showing the results of LSTI optimization has already been shown in Figure 4.8. To illustrate this further, Figure 4.15 contains an LSTI optimized output frame, the same frame number as shown in Figure 4.13 and it is coded at the same bit rate (0.32 bits/voxel). The resulting image sequence reconstruction has a PSNR of 29.4 dB—an increase of 1 decibel.

In general, by using the LSTI optimization, one can expect to improve the PSNR of the reconstructed output by around 1 dB. This can depend somewhat on the image sequence and cases of extreme splitting MSE thresholds. Figure 4.16 shows how LSTI optimization can provide an improvement over a wide range of splitting thresholds. For the "western" sequence, it can be seen that there is an improvement in PSNR over the entire range of bit rates when the LSTI optimization is performed

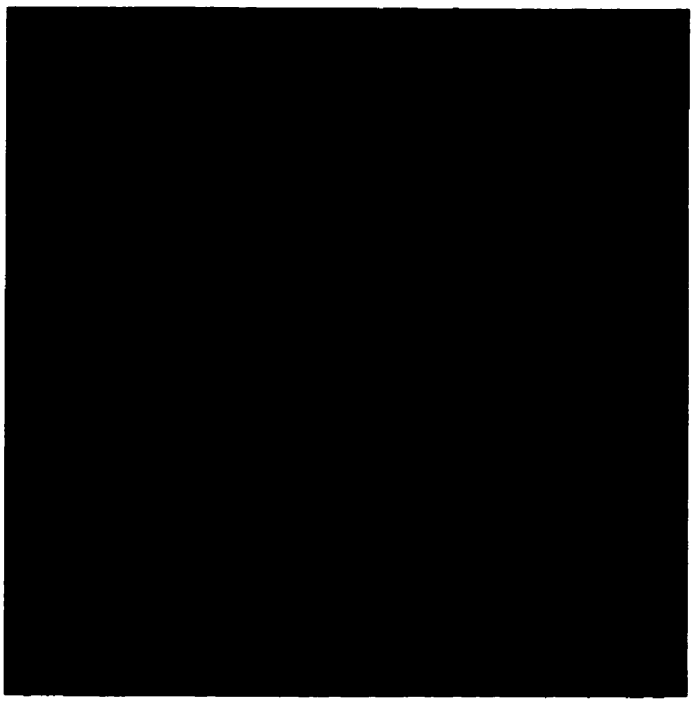


Figure 4.15: A reconstructed frame from the "western" image sequence, coded at 0.32 bits/voxel, showing the resulting improvements (over Figure 4.13(a)) from LSTI optimization. An improvement in PSNR of 1 dB (resulting in a PSNR of 29.4 dB) is achieved over non-LSTI coding.

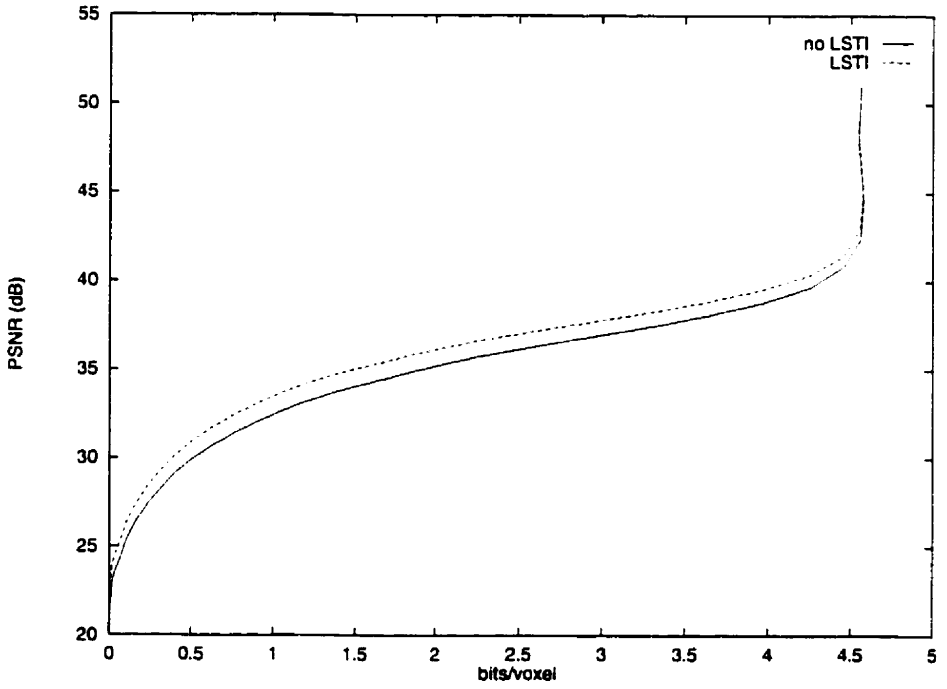


Figure 4.16: PSNR vs. bits/voxel for the "western" image sequence during a sweep of the splitting MSE threshold for a coder that uses LSTI optimization and one that does not.

by the coder. In fact, for most of the bits/voxel range, the increase is around 1 dB. With this improvement in mind and the fact that there are no required changes to the decoder or any significant increase in the bit rate, using an LSTI coder is a simple way to increase the reconstructed quality of the image sequence. The only disadvantage is the extra coding time required.

4.5.4 Effects of Frame Stack Size Selection

The selection of the frame stack size also has an effect on the PSNR vs. bits/voxel relationship. Temporal redundancies can be more easily exploited when a larger frame stack is used. For example, if a scene within an image sequence contains a stationary background with relatively little motion, very long blocks in the temporal direction can be created while still maintaining low interpolation error. At the same time, these temporally long blocks can have small spatial dimensions in order to reconstruct small spatial details in the static parts of the scene. A good example of this type of scene is in the "Miss America" sequence in which there is very little motion. The bits/voxel requirements of this type of image sequence can be significantly lowered by increasing the size of the frame stack.

The effects of frame stack size are shown in Figure 4.17 where the PSNR vs. bits/voxel is shown while the stack size is gradually changed from 2 to 33 frames while holding all other parameters constant. As the stack size is increased, the number of bits/voxel decreases while the PSNR only varies slightly. It should be noticed in the plot that the sequences with fewer motion components benefit more from an increased stack size and are more easily compressed.

The benefit of a large stack size is apparent. A very large frame stack would seem to be the best choice. This would provide the best compression performance,

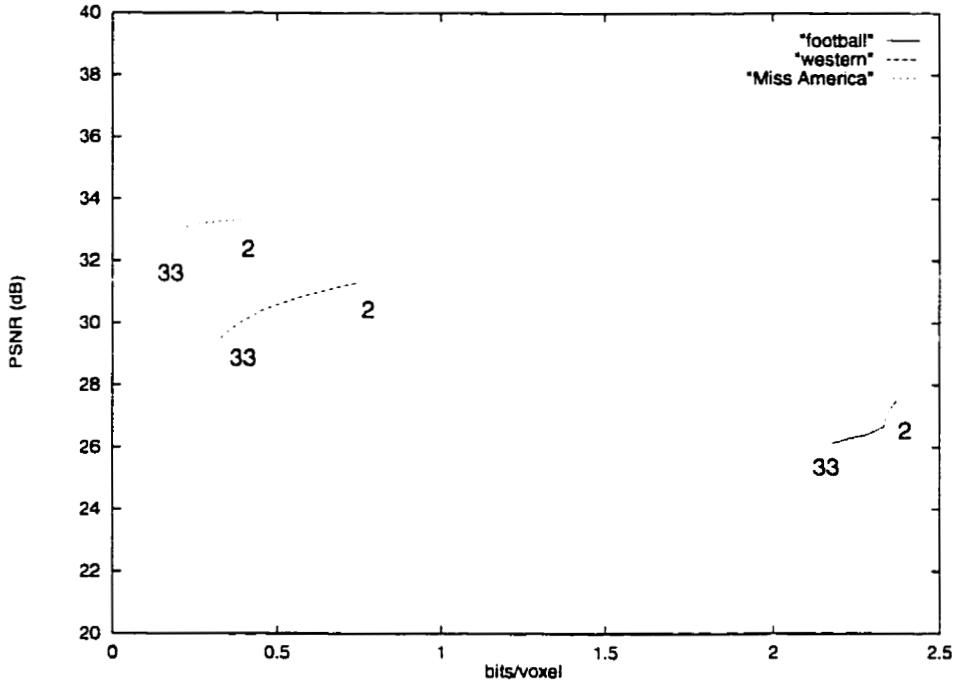


Figure 4.17: PSNR vs bits/voxel showing the effects of the frame stack size.

however it may not always be feasible to use such a large stack size. With an increased stack size, the memory requirements for both the coding and decoding algorithm can become quite large. Not only do the extra frames have to be stored in memory during the processing, but the splitting tree size will also need to be expanded. Therefore, a balance between memory usage and stack size must be made.

4.5.5 Effects of Vertex Quantization

In the compressed bit stream, the vertex values require the largest amount of information. Even with DPCM-Huffman coding, the number of bits/vertex can still be on the order of 5 to 7. Compounded with the possibility of a large number of vertices in an image sequence, it is easy to see why the vertex values take up the most information. As found in Section 3.3 for the 2-D case, one of the simplest solutions,

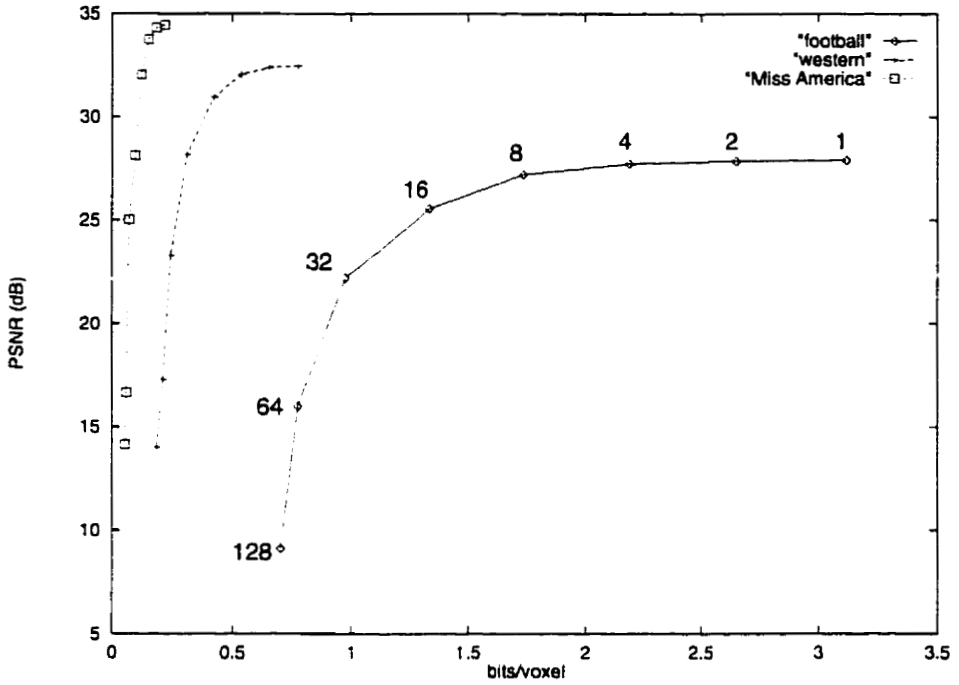


Figure 4.18: PSNR vs bits/voxel showing the effects of vertex quantization. Q is swept from 1 to 128.

which turns out to be one of the best solutions, is to quantize the vertex values before DPCM-Huffman coding. By simply dividing each vertex value by the constant factor Q and truncating the result, the bits/vertex can be reduced significantly depending on the value of Q. During the reconstruction stage, the decoder simply multiplies every vertex by Q before using as a corner value for trilinear interpolation. Furthermore, since the interpolation is performed using full arithmetic precision, the "onion ring" effect, commonly seen resulting from the quantization of image or video data, is avoided.

To see how various values of Q affect the coding and output quality of an image sequence, a plot illustrating this is shown in Figure 4.18. In the figure, the points on the graph correspond to quantization coefficient values of $Q = 1, 2, 4, 8, 16, 32, 64,$ and 128. From the plot, it can be seen that as Q is increased (i.e. the vertex values

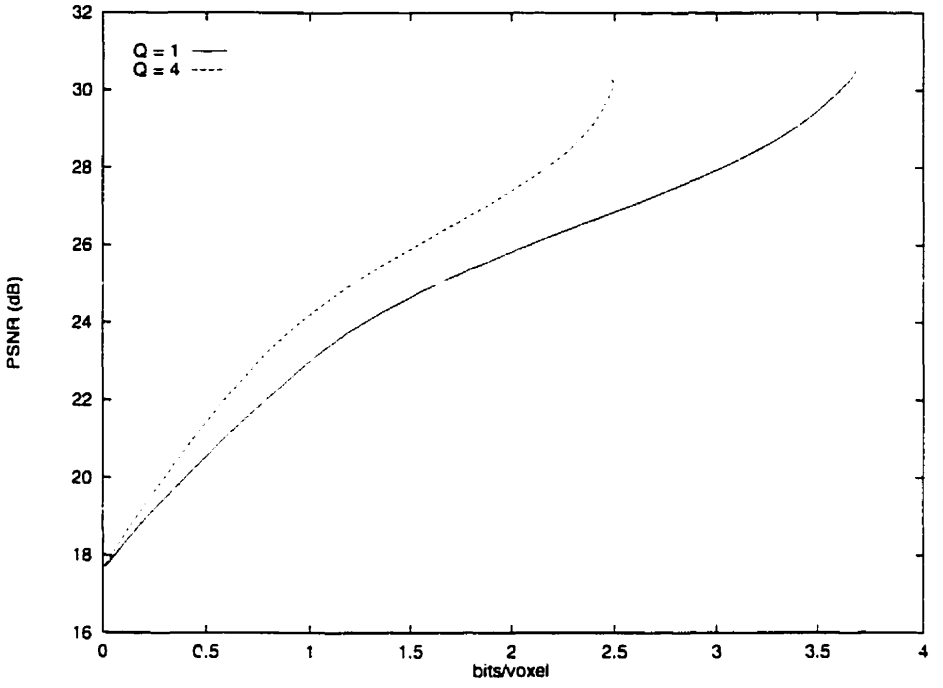


Figure 4.19: Spatial MSE splitting tolerance sweeps for the "football" image sequence using vertex quantization factors (Q) of 1 and 4.

are more heavily quantized), the number of bits/voxel is reduced. For changes in Q while it is small ($1 \leq Q \leq 8$), the resulting change in PSNR is negligible which is quite useful as the bits/voxel can be reduced with little effect on PSNR. It can also be seen from the figure that for image sequenced that have more vertices to code have the most to gain by vertex quantization. This is well illustrated by the PSNR vs. bits/voxel trace for the "football" sequence.

Since the number of bits/vertex can be reduced, a denser sampling grid can be employed to retain more vertices. Therefore, at the same bit rate, the quality of a reconstructed sequence can actually be increased by vertex quantization. This is shown for the "football" sequence in Figure 4.19 where the spatial MSE splitting tolerance is gradually changed with vertex quantization factors of 1 and 4. The improvement in PSNR can be seen over the entire range of splitting tolerances. For

example, at around 2.5 bits/voxel there is an improvement in PSNR of 3.3 dB for the coder using $Q = 4$. Thus, a radical improvement can be gained by quantizing the vertex values so that more vertices can be included at the same bit rate.

4.5.6 Arbitrary Spatial Dimensions

In Section 4.3.2, it was shown how the non-uniform sampling grid can be generated for a given 3-D block. It was also mentioned that if the spatial dimensions of the block are not of the form $(2^n + 1) \times (2^n + 1)$, fairly large block sizes can remain after subdivision that cannot be spatially spit further. This problem escalates when the aspect ratio of the image sequence deviates from 1 : 1.

The problem of arbitrary spatial dimensions can be alleviated by modifying the coder and decoder to allow horizontal or vertical splits when required. When a block has one side that cannot be subdivided further, an indication to split will be interpreted as a subdivision into two blocks along the axis that can support the split. This will result in a somewhat larger splitting tree that will increase the average number of bits/voxel of the compressed image sequence. However, it can result in a higher quality reconstruction.

In Figure 4.20 the effect of the arbitrary image sequence dimension allowance is shown for the coding and decoding of the "football" sequence. This sequence has spatial dimensions of 350 × 240, which yields an aspect ratio of 1.46 : 1. With large amounts of recursive splitting, blocks of size 4 × 2 (in the spatial dimensions) may result. These blocks can no longer be quad-split. A single vertical split would cleave this size of block into two blocks: 3 × 2 and 2 × 2. In the figure, the spatial and temporal MSE threshold is swept using the regular splitting method and the splitting method that allows for arbitrary spatial dimensions. The effects are also

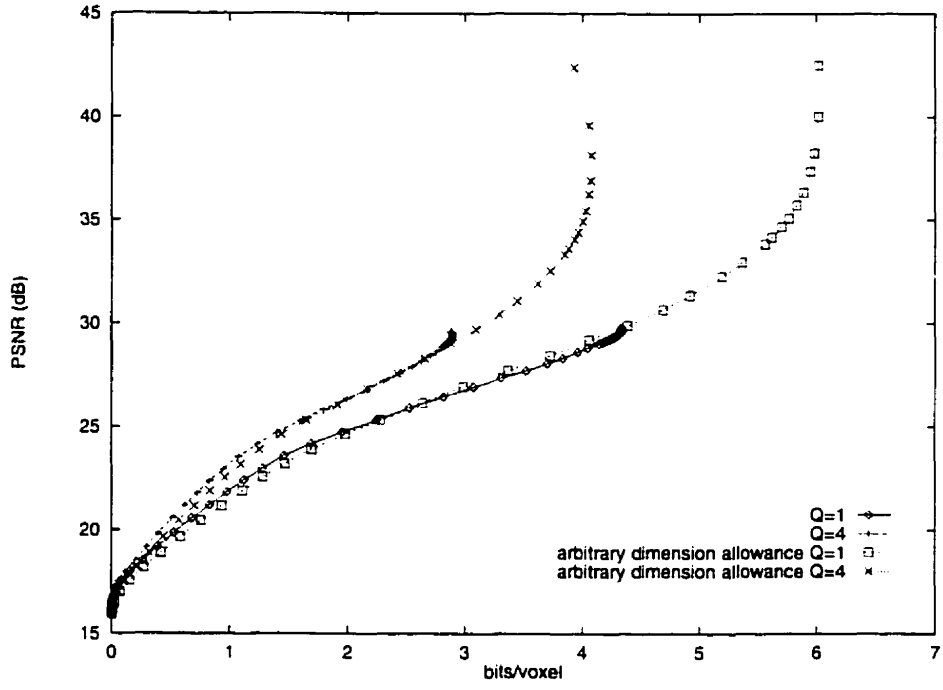


Figure 4.20: Combined spatial and temporal MSE splitting tolerance sweeps for the "football" image sequence using vertex quantization factors of $Q = 1$ and $Q = 4$, with and without arbitrary spatial dimensions allowance.

demonstrated for a vertex quantization factor of $Q = 4$. It can be seen that the coder which is designed to accommodate arbitrary spatial dimensions can code the image sequence with a wider range of bits/voxel and PSNR. At higher bit rates, higher quality reconstruction is possible since smaller blocks (more vertices) can be used. However, for low quality coding, the figure clearly shows that the arbitrary spatial dimensions allowance increases the bits/voxel slightly because more bits are required to represent the splitting tree.

A more visual example is portrayed in Figure 4.21. Shown is a reconstructed frame from (a) the regular coder and one from (b) the arbitrary dimensions coder. These sequences were coded with approximately the same number of bits/voxel: (a) at 2.68 bits/voxel and (b) at 2.67 bits/voxel. Normal splitting tends to result in small

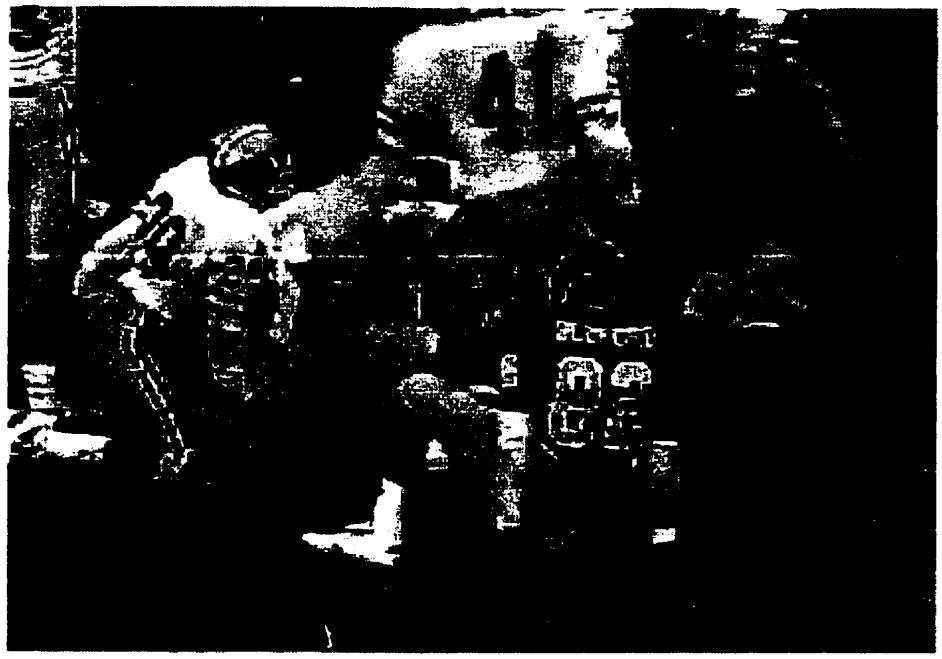
rectangular shaped blocks. This is quite noticeable when looking at the lettering on the players' jerseys or the stripes on the players' pants. This happens because the resolution in the horizontal direction is limited where the unsplittable 4×2 or 3×2 blocks are situated. In Figure 4.21(b) this problem is fixed by the arbitrary dimensions allowance. It is interesting to note that the increase in PSNR is negligible (0.08 dB) but the increase in visual quality is quite apparent.

Unless very low bit rate video coding is required, it is advantageous to implement the modified subdivision process that allows for arbitrary image dimensions. The codec system can then realize wider ranges of bit rates and qualities, and can perform lossless coding of an image sequence.

4.5.7 Comparison Between ALSTI and MPEG Video Coding

Now that the effects of all the coding parameters of the ALSTI coder have been explored, a proper comparison can be made with an existing video coder: the MPEG video coder. From the previous sections, it was seen that the ALSTI coder performs best by using a combination of LSTI, BSE, and a moderate quantization coefficient of $1 \leq Q \leq 8$. Furthermore, for sequences with arbitrary spatial dimensions, it is useful to include the arbitrary dimensions allowance modification to the splitting process. It was also shown that a large frame stack size is advantageous—if the memory requirements do not become too great. A general overview of the MPEG video codec was given in Section 1.2.2. In this section, the MPEG coder and decoder used are the "Berkeley MPEG-1 Video Encoder" [28] and the "Berkeley MPEG Player" [29] respectively.

By using the "football" and "Miss America" sequences, a good comparison between the ALSTI coder and the MPEG coder can be made since the two image



(a)

(b)

Figure 4.21: Example showing the effects of arbitrary dimensions allowance on the "football" image sequence: (a) Normal coding and (b) coding with allowances for arbitrary image dimensions.

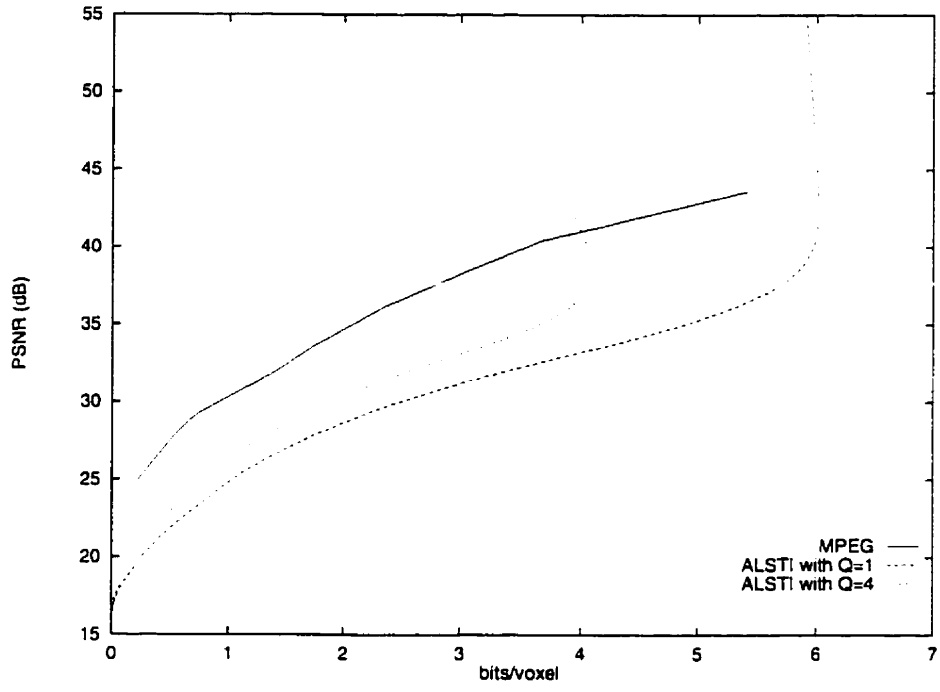


Figure 4.22: Comparison between ALSTI and MPEG video coding using the "football" image sequence.

sequences are of drastically different types: the "football" sequence contains high spatial frequencies and fast motion, while the "Miss America" sequence contains fairly simple slow moving objects. Therefore, best case and worst case comparisons can be drawn. Figures 4.22 and 4.23 show the ALSTI vs. MPEG comparisons for both the "football" and "Miss America" image sequences. In the figures, the ALSTI traces are a result of sweeping the combined spatial and temporal MSE splitting tolerances so that a wide range of vales can be seen. The PSNR vs. bits/voxel values for the MPEG codec are obtained by using various allowable combinations of I-, P-, and B-frames, as well as varying the Block DCT quantization factors. It should be noted that the highest bit rate shown for the MPEG coded sequences also represents the highest quality possible. This was achieved by using only I-frames and the lowest possible quantization values.

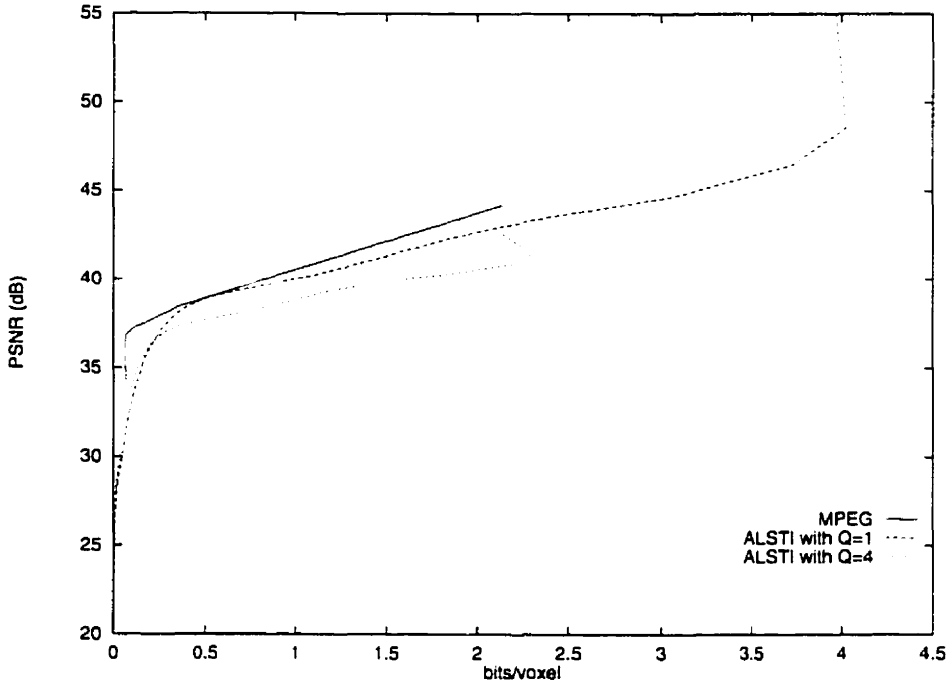


Figure 4.23: Comparison between ALSTI and MPEG video coding using the "Miss America" image sequence.

When examining Figure 4.22 it is clear that over its entire range of bits/voxel, the MPEG codec achieves superior PSNR values to the ALSTI codec with $Q = 1$. The ALSTI codec with $Q = 4$ is also outperformed by the MPEG codec, except in the region of high splitting; the ALSTI codec ($Q = 4$) can produce a higher quality output than MPEG. Furthermore, the MPEG codec cannot produce any higher PSNR than shown, where the ALSTI codec with $Q = 1$ can perform lossless coding of the image sequence to produce an infinite PSNR.

For the simpler "Miss America" image sequence, a more similar performance between the ALSTI and MPEG codecs can be seen. In fact, at around 0.5 bits/voxel, the performance is virtually identical. Furthermore, as seen with the "football" sequence, the ALSTI codec with $Q = 1$ can produce a higher quality output than is possible with the MPEG codec. Also, in the extremely low bit rate range, the sharp

drop in quality for the MPEG codec suggests that if the bit rate is reduced further, the ALSTI codec may achieve better coding performance.

Chapter 5

Conclusions and Further Research

5.1 Summary of Thesis

The work presented in this thesis has primarily dealt with the information compression of digital images and video. The methods described in detail were based on an adaptive sampling and interpolation scheme. In Chapter 1, the reasons behind digital image and video compression were given, along with the definitions of some common image and video coding performance metrics. Then, a commonly used image codec, JPEG, and a family of video codecs, MPEG, H.261, and H.263, were described, along with their advantages and shortcomings; it was these shortcomings that were the motivation behind the work presented in this thesis.

The adaptive interpolation algorithm for image coding was described in Chapter 2, starting with a detailed background of linear and bilinear interpolation. It was then shown how an image can be adaptively subdivided in to smaller sub-blocks, and how the size and position information of these sub-blocks can be efficiently stored in a quaternary tree structure.

In Chapter 3, issues concerning the implementation of the adaptive interpo-

lation image coder were discussed. Various bilinear interpolation methods were introduced and and compared. Based on interpolation accuracy and speed, it was found that the DDA implementation was the best choice. The efficient storage of the adaptive splitting tree, based on bit patterns and LZW compression, was discussed. It was also found that images having arbitrary dimensions could not be subdivided down to the pixel level so that lossless coding could not be performed. A solution, based on a modification to the tree structure, was given in order to alleviate this problem with only a minor degradation to the compression performance. In Chapter 3, it was also shown how the effects of interpolation discontinuities, arising from different sized blocks lying adjacent to one another, could be reduced by a block size equalization algorithm that is performed during the image reconstruction. The storage of the vertices lying on the adaptive sampling grid was also addressed and an example showing how vertex quantization can be used to further compress the information required for these vertices. The vertices were stored by a 2-D DPCM algorithm followed by Huffman coding. By using unsigned 8-bit arithmetic, it was shown how the DPCM-Huffman coding could be made more efficient. The Least-Squares Bilinear Interpolation optimization was then described. It was shown how this optimization could be performed by the coder to result in a higher quality reconstruction. The problem of LSBI coefficient storage was also solved by describing how the LSBI optimization is simply a series of 1-D LSLI operations. Finally, results of using the Adaptive Interpolation image coder were shown and a comparison between the ALSBI and Block DCT image coders was given. It was found that the ALSBI coder performed quite similarly to the Block DCT coder in most cases and outperformed the Block DCT coder in two-toned image compression.

The discussion in Chapter 4 was devoted to the description and implementation of a digital video codec based on an extension to the Adaptive Interpolation method

into three dimensions. Trilinear interpolation—the basis of this video coder—was discussed in detail. Just as in the 2-D case, it was found that, based on accuracy and speed, the best trilinear interpolation implementation was the DDA method. The generation and representation of the adaptive sampling grid was also described. For a 3-D video signal which has the added dimension of time, the concept of temporal splitting was introduced and the splitting tree was changed to a quaternary-binary tree where quad-splits were performed in the spatial dimensions and binary-splits were done in the temporal dimension. It was shown that this method of splitting can result in significant computational savings in the coder because the interpolation error over an entire block does not have to be evaluated; only the front and back faces need to be examined for spatial splitting, and for temporal splitting, only the four edges extending into the temporal direction need examining. It was also shown that if the spatial dimensions of the image sequence are not of the form $(2^n - 1) \times (2^n - 1)$, lossless coding of the sequence is not possible without modifications made to the splitting process to allow arbitrary spatial dimensions. The method for block size equalization was reviewed in order to reduce the effects of interpolation discontinuities in three dimensions. The 3-D DPCM-Huffman coding of the vertex values was described and vertex quantization was revisited. Then, the concept of Least-Squares Trilinear Interpolation was introduced and an example was shown into how it can be used by the video coder to increase the visual quality of a reconstructed image sequence. It was also shown that the LSTI coefficients required no more storage space than the 1-D LSLI, or 2-D LSBI coefficients. Finally, the effects of varying the ALSTI coding parameters were shown and discussed along with a comparison between ALSTI and MPEG video coding.

5.2 Conclusions

5.2.1 ALSBI Image Coding

In general the ALSBI image codec performs quite comparably to the Block DCT coder. Many of the disadvantages associated with Block DCT coding, such as blockiness, waviness or "ringing" around edges, and poor coding performance with images containing many high contrast edges and/or low numbers of greyscales, can be avoided at the same level of compression. Blockiness is avoided because the ALSBI coder is not, in the strict sense, a block-based coder. It usually uses many different block sizes where the blocks are not usually visible due to the sharing of common corner points and bilinear interpolation. The Block DCT coder has problems with edges, sometimes producing waviness in the output. This is also the reason behind its poor performance with images only having low numbers of greyscales (or two-toned images) as well as images containing many high contrast edges. The ALSBI coder avoids this problem by simply allocating more samples around edges, allowing for a higher quality reconstruction.

The ALSBI coder does not use any transform based coding. Thus, a (potentially) computationally intensive transform is avoided in both the coder and decoder. When compared to the DCT, this is not too much of an advantage, since the DCT has been highly optimized over the years [23]—especially for fixed sized blocks. In fact, the ALSBI image coder requires more computations for higher quality reconstructions than for low quality reconstructions because more splits and interpolation error calculations are required. Furthermore, optimizations for improving the output quality, such as the LSBI optimization performed by the coder and block size equalization performed by the decoder, require additional calculation time and system resources. However, for most images and output qualities, the ALSBI coder and decoder do not

require much more time to compute than their Block DCT counterparts.

Lossless image compression can also be performed by the ALSBI codec. This is a great advantage because the Block DCT coder cannot achieve lossless compression. In the JPEG standard, there does exist a lossless JPEG compression method [12]. However, it is a completely different algorithm and is not very popular. With the ALSBI codec, lossless image coding can be achieved without having to change the algorithm used.

Overall, the ALSBI coder avoids Block DCT artifacts, performs reasonably well on images having reduced numbers of greyscales, is comparable in computational complexity, and can achieve lossless image compression. This makes it a much more flexible image coder that can achieve approximately the same quality and compression as a Block DCT coder.

5.2.2 ALSTI Video Coding

The ALSTI video codec, being based on the same principals as the ALSBI image codec, also does not suffer from Block DCT artifacts in the spatial domain. So, unlike the MPEG codec at low bit rates, the ALSTI video coder will not exhibit blockiness, and "ringing" around edges and moving objects. Furthermore, the ALSTI video codec does not require any motion estimation, which is essential to MPEG video coding. This makes it quite attractive because of its low computational complexity. However, due to the recursive nature of the ALSTI method, higher quality reconstruction requires more calculations than for lower quality reconstruction: the codec is fastest on medium to low output qualities. Thus, as was seen when coding the highly complex "football" sequence, ALSTI with $Q = 4$ did perform better in PSNR at around 4 bits/voxel (see Figure 4.22), however this was encoded with a high degree

of splitting, which resulted in slower operation, especially for the coder.

Although the ALSTI codec can produce a higher PSNR than the MPEG video codec, it does so at the expense of higher bits/voxel. For the "football" image sequence, within the MPEG codec's range of bits/voxel, MPEG easily outperforms ALSTI by up to 5.5 dB. However, outside this bits/voxel range, ALSTI can encode the video sequence in a lossless manner—something the MPEG coder cannot do. For less complex image sequences, such as the "Miss America" sequence, the ALSTI codec rivals—but does not exceed—the performance of MPEG. Therefore, ALSTI would best seem suited for coding image sequences of low complexity—video conferencing, for example. Overall, the ALSTI codec is a more versatile codec which can produce wider ranges of bits/voxel and PSNR than MPEG. However, when operating at the same bit rate, MPEG usually will outperform ALSTI in PSNR.

5.3 Recommendations for Further Research

5.3.1 ALSBI Image Coding

In order to investigate improvements to the ALSBI image coder, a few suggested ideas can be given. It was shown in Section 3.5 that the vertex values require the largest amount of information to encode. If the number of bits/vertex could be reduced further, the overall bits/pixel required by the coded image could also be reduced. One course of investigation could be into the vector quantization of the vertex values or perhaps another type of error minimizing quantizer—one of which may lead to better results than the scalar quantization used in this thesis. It may also be possible to change the quantization level for different areas within the image. This could be used to increase the quality of important details while allowing unimportant areas

of the image to be coded with fewer bits. The bits/vertex may also be reduced by replacing the Huffman coder with a more optimized entropy coder, such as an arithmetic coder [2] or a Huffman coder with fixed output codes that are optimized for a large number of images (so that the Huffman decoding table does not have to be sent to the decoder). Another avenue of approach into reducing the storage requirements of the vertices would be the investigation into a lossy DPCM coding algorithm for coding of the vertex values.

Another interesting improvement would be to somehow combine the LSBI optimization along with vertex quantization. This may not decrease the bits/vertex, but it might increase the reconstructed image quality. Furthermore, the LSBI solution (shown in Equation 3.5) does not merely have to be applied to one block at a time. It is possible to use Equation 3.5 to optimize multiple blocks of varying sizes at once which would result in a lower interpolation error. It is even possible to optimize over the entire splitting grid. However, the computational costs of doing so would be impractical. So, it may be possible to perform the LSBI optimization over regions of the image instead of just optimizing individual blocks. This would reduce the reconstruction error further.

Splitting an image into rectangular blocks might not be the optimum subdivision method for image coding. It has been suggested in [30], that it is possible to subdivide images into triangular sections. This may lead to more efficient coding because fewer vertices are required to represent each triangular section. However, since fewer vertices are used, triangular subdivision might require a deeper level of recursion (i.e. more splits) to achieve the same amount of detail as rectangular subdivision.

The speed of the bilinear interpolator might also be increased by using a hardware-based interpolator. In computer graphics, bilinear interpolation is used in *Gouraud shading* of arbitrary polygons [20]. Since most personal computer graphics

accelerators include hardware support for Gouraud shading, they would also perform bilinear interpolation very quickly. For Adaptive Interpolation image coding, the polygons that require interpolation are simple rectangles. Some accelerator implementations may require that arbitrary polygons be converted to triangles. This is a trivial operation when the polygons are rectangles. So, by using a graphics accelerator to perform quick bilinear interpolations, the time required for both coding and decoding could be reduced.

Finally, it has been suggested in [16] and [14] that the ALSBI image compression scheme can be supplemented by applying a Block DCT coder to the image reconstruction error (i.e. the difference between the actual and ALSBI decoded image), which is also known as the residual error. In this way, the output image quality can be improved by the addition of the decoded residual values to the ALSBI decoded pixel values. This becomes somewhat similar to a sub-band decomposition scheme, where the ALSBI channel contains mostly low frequency components, while the Block DCT channel contains high frequencies. The only disadvantage is the large amount of data required to store the DCT coefficients. So, if an alternate method could be developed to store the residual error without requiring large amounts of data, the reconstructed image quality could be vastly improved.

5.3.2 ALSTI Video Coding

Most of the same recommendations given for the ALSBI image codec could also benefit the ALSTI video codec. The vertices lying on the 3-D non-uniform sampling grid also require the largest amount of information for their representation. Thus, a more efficient representation would greatly benefit the video codec. The LSTI optimization could also be expanded to 3-D regions of a frame stack instead of

individual blocks. The use of a graphics accelerator could also speed up the trilinear interpolation process. Since Gouraud shading only uses 2-D bilinear interpolation for 2-D polygons, the graphics accelerator could not perform trilinear interpolation directly. However, the first two steps of a three-step trilinear interpolation method (i.e. the bilinear interpolation of the front and back faces) could, at least, be performed in hardware. Alternately, by first performing linear interpolation of the four corner values in the temporal direction, each frame within the 3-D block could be separately bilinear interpolated.

To lower the bit rate and/or increase the output quality further, it may be possible to use a form of motion estimation with the ALSTI video coder. This would dramatically increase the computational load for the coder but the gains in quality and/or compression may be well worth it. A combination between motion estimation and ALSTI coding was attempted, during the research for this thesis, by subdividing a frame stack into rhombohedrons (parallelepipeds bounded by six congruent rhombuses [24]) having rectangular faces contained in the spatial planes. There was great difficulty in ensuring that the entire frame stack was covered by the rhombohedrons. Furthermore, the information required for the representation of the splitting process became quite large since a simple splitting tree was not enough to describe the rhombohedrons covering the frame stack. Although this approach did not work well, there still may be a method that can successfully combine ALSTI coding and motion estimation for increased compression and/or reconstructed quality.

References

[1] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing.* Addison Wesley. 1993.

[2] William H. Press. Saul A. Teukolsky. William T. Vetterling. and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press. second edition. 1997.

[3] comp.compression Frequently Asked Questions. Posted to comp.compression newsgroup. January 9. 1999.

[4] Denis Howe. The Free On-line Dictionary of Computing. http://foldoc.doc.ic.ac.uk. 1999.

[5] Hai-Ling Margaret Cheng. 3D Spatio-temporal Interpolation of Digital Image Sequences using Low-order 3D IIR Filters. Master's thesis. University of Calgary. 1997.

[6] Syed A. Rizvi. Nader Mohsenian. and Nasser M. Nasrabadi. Color image compression using neural network prediction of color components. In *Proceedings of SPIE.* volume 2664. pages 37–46. The International Society for Optical Engineering, 1996.

[7] International Business Machines Corporation. *Ultimedia Services Version 2 for AIX: Programmer's Guide and Reference*. 1996.

[8] Chad Fogg. MPEG-2 Frequently Asked Questions. http://bmrc.berkeley.edu/frame/research/mpeg/mpeg2faq.html. April 1996.

[9] MPEG-1 Frequently Asked Questions. http://bmrc.berkeley.edu/frame/research/mpeg/mpegfaq.html.

[10] Antonio Ortega and Kannan Ramchandran. Rate-Distortion Methods for Image and Video Compression. *IEEE Signal Processing Magazine*. 15(6):23-50. November 1998.

[11] Gary J. Sullivan and Thomas Wiegand. Rate-Distortion Optimization for Video Compression. *IEEE Signal Processing Magazine*. 15(6):74-90. November 1998.

[12] JPEG Frequently Asked Questions. Posted to comp.graphics.misc newsgroup. March 29, 1999.

[13] Gregory K. Wallace. The JPEG Still Picture Compression Standard. *IEEE Transactions on Consumer Electronics*. 38:xviii–xxxiv. February 1992.

[14] Francesco G. B. De Natale. Giuseppe S. Desoli. and Daniele D. Giusto. Adaptive Image Sampling and Interpolation for Data Compression. *Journal of Visual Communications and Image Representation*. 5(4):338–402. December 1994.

[15] F. G. B. De Natale, G. S. Desoli. and D. D. Giusto. Hierarchical Image Coding via MSE-Minimizing Bilinear Approximation. *Electronics Letters*. 27(22):2035–2037. October 1991.

[16] F. G. B. De Natale. G. S. Desoli. and D. D. Giusto. Adaptive Least-Squares Bilinear Interpolation (ALSBI): A New Approach to Image-Data Compression. *Electronics Letters*. 29(18):1638–1639. September 1993.

[17] Francesco G. B. De Natale. Giuseppe S. Desoli. and Daniele D. Giusto. Image Sequence Data Compression Through Adaptive Space-Temporal Interpolation. *Time-Varying Image Processing and Moving Object Recognition*. 1994.

[18] Steve Blackstock. LZW and GIF explained. Posted to comp.graphics newsgroup. April 21. 1989.

[19] P. P. Vaidyanathan. Quadrature Mirror Filter Banks. M-Band Extensions and Perfect-Reconstruction Techniques. *IEEE ASSP Magazine*. pages 4–20. July 1987.

[20] James D. Foley. Andries van Dam. Steven K. Feiner. John F. Hughes. and Richard L. Phillips. *Introduction to Computer Graphics*. Addison-Wesley Publishing Company. 1994.

[21] Correspondence with Francesco G. B. De Natale. July 1997.

[22] Conversations with Remi J. Gurski. April 1998.

[23] N. U. Ahmed and K. R. Rao. *Orthogonal Transforms for Digital Signal Processing*. Springer-Verlag. 1975.

[24] Eric W. Weisstein. *The CRC Concise Encyclopedia of Mathematics*. CRC Press. 1998.

[25] G. Desoli, F. De Natale, and Giusto D. High Compression Video Coding: a novel approach. *Image and Video Compression*. 2186:270–277, 1994.

[26] Francesco G. B. De Natale and Daniele D. Giusto. A Mesh-Interpolation Scheme for Very-Low Bitrate Coding of Video Sequences. *European Transactions on Telecommunications and Related Technologies*. 9(1):47–55. January-February 1998.

[27] F. G. B. De Natale, G. S. Desoli, and D. D. Giusto. Efficient interpolation scheme for image-sequence data compression. *Electronics Letters*. 30(1):20–21. January 1994.

[28] Plateau Research Group. Berkeley MPEG-1 Video Encoder Users Guide. http://bmrc.berkeley.edu/research/mpeg.

[29] Plateau Research Group. The Berkeley MPEG Player. http://bmrc.berkeley.edu/research/mpeg.

[30] Joceli Mayer. A Blending Model for Efficient Compression of Smooth Images. In James A. Storer and Martin Cohn, editors. *Data Compression Conference*. pages 228–237. IEEE Computer Society. March 1999.

Appendix A

Least-Squares Interpolation

A.1 Derivation of the Least-Squares Interpolation Equation

The purpose of interpolation is to approximate a large set of values from a smaller set of known values. In a coding application (1-D, 2-D, or 3-D) the large set of values to be approximated belongs to the original data set while the smaller set is given to the decoder in order to reconstruct the original set. Mathematically, this is equivalent to

$$\mathbf{A}\mathbf{x} \simeq \mathbf{y} \tag{A.1}$$

or

$$\mathbf{A}\mathbf{x} = \tilde{\mathbf{y}} \tag{A.2}$$

where $\mathbf{x}$ is the small set of data to be interpolated, $\mathbf{A}$ is the interpolation matrix, and $\mathbf{y}$ is a vector containing the original sample values or as in Equation A.2 $\tilde{\mathbf{y}}$ represents the approximation to the original values created through interpolation.

These equations can also be expressed as

$$\mathbf{Ax} = \mathbf{y} + \mathbf{e} \tag{A.3}$$

where $\mathbf{e}$ is the interpolation error (i.e. the error between the interpolated output values and the original values).

In order to minimize the magnitude of the interpolation error ($\|\mathbf{e}\|$) in the least-squares sense, the values in $\mathbf{x}$ can be modified by using the interpolation matrix and the original values contained in $\mathbf{y}$. First, the squared magnitude of the interpolation error vector must be calculated:

$$\begin{aligned} \|\mathbf{e}\|^2 &= \mathbf{e}^T\mathbf{e} \\ &= \left(\tilde{\mathbf{y}}^T - \mathbf{y}^T\right) \cdot (\tilde{\mathbf{y}} - \mathbf{y}) \\ \|\mathbf{e}\|^2 &= \tilde{\mathbf{y}}^T\tilde{\mathbf{y}} - \tilde{\mathbf{y}}^T\mathbf{y} - \mathbf{y}^T\tilde{\mathbf{y}} + \mathbf{y}^T\mathbf{y} \end{aligned} \tag{A.4}$$

Now, since $\mathbf{y}^T\tilde{\mathbf{y}} = \tilde{\mathbf{y}}^T\mathbf{y}$, Equation A.4 becomes

$$\|\mathbf{e}\|^2 = \tilde{\mathbf{y}}^T\tilde{\mathbf{y}} - 2\,\tilde{\mathbf{y}}^T\mathbf{y} - \mathbf{y}^T\mathbf{y} \tag{A.5}$$

Then, substitution of $\tilde{\mathbf{y}}$ with $\mathbf{Ax}$ (from Equation A.2) results in

$$\|\mathbf{e}\|^2 = \mathbf{x}^T\mathbf{A}^T\mathbf{Ax} - 2\,\mathbf{x}^T\mathbf{A}^T\mathbf{y} - \mathbf{y}^T\mathbf{y} \tag{A.6}$$

In order to calculate the optimal $\mathbf{x}$ vector that will minimize the interpolation error, the above equation must be minimized with respect to $\mathbf{x}$. This is accomplished by first taking the derivative of Equation A.6 with respect to $\mathbf{x}$:

$$\frac{d\left(\|\mathbf{e}\|^2\right)}{d\mathbf{x}} = 2\,\mathbf{A}^T\mathbf{Ax} - 2\,\mathbf{A}^T\mathbf{y} \tag{A.7}$$

At the minimum squared error magnitude, the derivative will be zero. Then the vector $\hat{\mathbf{x}}$ that results in the minimum squared error is found by solving the equation

$$2\,\mathbf{A}^T\mathbf{A}\hat{\mathbf{x}} - 2\,\mathbf{A}^T\mathbf{y} = \mathbf{0} \tag{A.8}$$

This results in a solution for the vector $\hat{\mathbf{x}}$ that will result in the minimum squared error magnitude—which will also result in the minimum mean squared error:

$$\hat{\mathbf{x}} = \left(\mathbf{A}^{\mathrm{T}}\mathbf{A}\right)^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{y} \tag{A.9}$$

It should be noted that the solution of Equation A.8 can also result in a maximum value because the first derivative of any function is equal to zero at maxima and minima. To ensure that $\hat{\mathbf{x}}$ results in a minimum squared error, the second derivative of Equation A.6 with respect to $\mathbf{x}$ is taken:

$$\frac{\mathrm{d}\left(\|\mathbf{e}\|^2\right)^2}{\mathrm{d}^2\mathbf{x}} = 2\,\mathbf{A}^{\mathrm{T}}\mathbf{A} \tag{A.10}$$

Now, since

$$\mathbf{A}_{i,j} \geq 0 \qquad \forall\, i, j \tag{A.11}$$

and the columns of $\mathbf{A}$ are non-orthogonal with each other, it follows that $\mathbf{A}^{\mathrm{T}}\mathbf{A} > \mathbf{0}$ and thus,

$$\frac{\mathrm{d}\left(\|\mathbf{e}\|^2\right)^2}{\mathrm{d}^2\mathbf{x}} > \mathbf{0} \tag{A.12}$$

which means that the squared error caused by interpolating $\hat{\mathbf{x}}$ will be a minimum—not a maximum.

The equation for $\hat{\mathbf{x}}$ (Equation A.9) is the result used in Sections 3.4 and 4.4 and is identical to the equation given in [15].

A.2 Minimum Least-Squares Interpolation Error

It is possible to calculate the resulting interpolation error when using $\mathbf{A}$ to interpolate the optimized vector $\hat{\mathbf{x}}$. This is done by using the definitions for $\hat{\mathbf{x}}$ and the squared error $\|\mathbf{e}\|^2$ given in Equations A.4 and A.9. Thus, the minimum squared

error resulting from the interpolation of $\hat{\mathbf{x}}$ by $\mathbf{A}$ can be simplified into a number of different forms:

$$\|\mathbf{e}\|^2_{\min} = \hat{\mathbf{x}}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}\hat{\mathbf{x}} - 2\,\mathbf{y}^{\mathrm{T}}\mathbf{A}\hat{\mathbf{x}} + \mathbf{y}^{\mathrm{T}}\mathbf{y} \tag{A.13}$$

$$\|\mathbf{e}\|^2_{\min} = \mathbf{y}^{\mathrm{T}}\left[\mathbf{1} - \mathbf{A}\left(\mathbf{A}^{\mathrm{T}}\mathbf{A}\right)^{-1}\mathbf{A}^{\mathrm{T}}\right]\mathbf{y} \tag{A.14}$$

$$\|\mathbf{e}\|^2_{\min} = \mathbf{y}^{\mathrm{T}}\mathbf{y} - \mathbf{y}^{\mathrm{T}}\mathbf{A}\hat{\mathbf{x}} \tag{A.15}$$