3

Video Coding Concepts

3.1 INTRODUCTION

compress vb.: to squeeze together or compact into less space; condense

compress noun: the act of compression or the condition of being compressed

Compression is the process of compacting data into a smaller number of bits. Video compression (video coding) is the process of compacting or condensing a digital video sequence into a smaller number of bits. 'Raw' or uncompressed digital video typically requires a large bitrate (approximately 216 Mbits for 1 second of uncompressed TV-quality video, see Chapter 2) and compression is necessary for practical storage and transmission of digital video.

Compression involves a complementary pair of systems, a compressor (encoder) and a decompressor (decoder). The encoder converts the source data into a compressed form (occupying a reduced number of bits) prior to transmission or storage and the decoder converts the compressed form back into a representation of the original video data. The encoder/decoder pair is often described as a *CODEC* (en*CO*der/ *DEC*oder) (Figure 3.1).

Data compression is achieved by removing *redundancy*, i.e. components that are not necessary for faithful reproduction of the data. Many types of data contain *statistical* redundancy and can be effectively compressed using *lossless* compression, so that the reconstructed data at the output of the decoder is a perfect copy of the original data. Unfortunately, lossless compression of image and video information gives only a moderate amount of compression. The best that can be achieved with current lossless image compression standards such as JPEG-LS [1] is a compression ratio of around 3–4 times. *Lossy* compression is necessary to achieve higher compression. In a lossy compression system, the decompressed data is not identical to the source data and much higher compression ratios can be achieved at the expense of a loss of visual quality. Lossy video compression systems are based on the principle of removing *subjective* redundancy, elements of the image or video sequence that can be removed without significantly affecting the viewer's perception of visual quality.

H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia.
Iain E. G. Richardson. ISBN: 0-470-84837-5

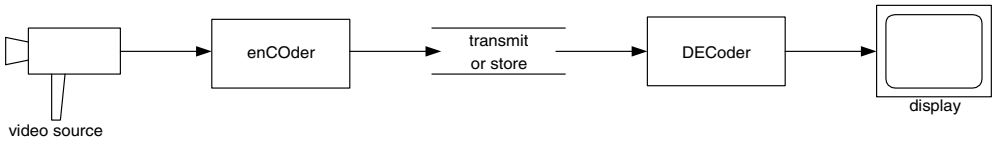

Figure 3.1 Encoder/decoder

Figure 3.2 Spatial and temporal correlation in a video sequence

Most video coding methods exploit both *temporal* and *spatial* redundancy to achieve compression. In the temporal domain, there is usually a high correlation (similarity) between frames of video that were captured at around the same time. Temporally adjacent frames (successive frames in time order) are often highly correlated, especially if the temporal sampling rate (the frame rate) is high. In the spatial domain, there is usually a high correlation between pixels (samples) that are close to each other, i.e. the values of neighbouring samples are often very similar (Figure 3.2).

The H.264 and MPEG-4 Visual standards (described in detail in Chapters 5 and 6) share a number of common features. Both standards assume a CODEC 'model' that uses block-based motion compensation, transform, quantisation and entropy coding. In this chapter we examine the main components of this model, starting with the temporal model (motion estimation and compensation) and continuing with image transforms, quantisation, predictive coding and entropy coding. The chapter concludes with a 'walk-through' of the basic model, following through the process of encoding and decoding a block of image samples.

3.2 VIDEO CODEC

A video CODEC (Figure 3.3) encodes a source image or video sequence into a compressed form and decodes this to produce a copy or approximation of the source sequence. If the

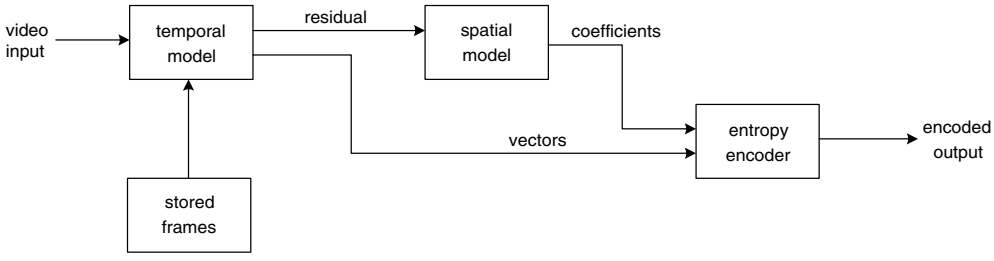

Figure 3.3 Video encoder block diagram

decoded video sequence is identical to the original, then the coding process is lossless; if the decoded sequence differs from the original, the process is lossy.

The CODEC represents the original video sequence by a *model* (an efficient coded representation that can be used to reconstruct an approximation of the video data). Ideally, the model should represent the sequence using as few bits as possible and with as high a fidelity as possible. These two goals (compression efficiency and high quality) are usually conflicting, because a lower compressed bit rate typically produces reduced image quality at the decoder. This tradeoff between bit rate and quality (the rate-distortion trade off) is discussed further in Chapter 7.

A video encoder (Figure 3.3) consists of three main functional units: a *temporal model*, a *spatial model* and an *entropy encoder*. The input to the temporal model is an uncompressed video sequence. The temporal model attempts to reduce temporal redundancy by exploiting the similarities between neighbouring video frames, usually by constructing a prediction of the current video frame. In MPEG-4 Visual and H.264, the prediction is formed from one or more previous or future frames and is improved by compensating for differences between the frames (motion compensated prediction). The output of the temporal model is a residual frame (created by subtracting the prediction from the actual current frame) and a set of model parameters, typically a set of motion vectors describing how the motion was compensated.

The residual frame forms the input to the spatial model which makes use of similarities between neighbouring samples in the residual frame to reduce spatial redundancy. In MPEG-4 Visual and H.264 this is achieved by applying a transform to the residual samples and quantizing the results. The transform converts the samples into another domain in which they are represented by transform coefficients. The coefficients are quantised to remove insignificant values, leaving a small number of significant coefficients that provide a more compact representation of the residual frame. The output of the spatial model is a set of quantised transform coefficients.

The parameters of the temporal model (typically motion vectors) and the spatial model (coefficients) are compressed by the entropy encoder. This removes statistical redundancy in the data (for example, representing commonly-occurring vectors and coefficients by short binary codes) and produces a compressed bit stream or file that may be transmitted and/or stored. A compressed sequence consists of coded motion vector parameters, coded residual coefficients and header information.

The video decoder reconstructs a video frame from the compressed bit stream. The coefficients and motion vectors are decoded by an entropy decoder after which the spatial

model is decoded to reconstruct a version of the residual frame. The decoder uses the motion vector parameters, together with one or more previously decoded frames, to create a prediction of the current frame and the frame itself is reconstructed by adding the residual frame to this prediction.

3.3 TEMPORAL MODEL

The goal of the temporal model is to reduce redundancy between transmitted frames by forming a predicted frame and subtracting this from the current frame. The output of this process is a residual (difference) frame and the more accurate the prediction process, the less energy is contained in the residual frame. The residual frame is encoded and sent to the decoder which re-creates the predicted frame, adds the decoded residual and reconstructs the current frame. The predicted frame is created from one or more past or future frames ('reference frames'). The accuracy of the prediction can usually be improved by compensating for motion between the reference frame(s) and the current frame.

3.3.1 Prediction from the Previous Video Frame

The simplest method of temporal prediction is to use the previous frame as the predictor for the current frame. Two successive frames from a video sequence are shown in Figure 3.4 and Figure 3.5. Frame 1 is used as a predictor for frame 2 and the residual formed by subtracting the predictor (frame 1) from the current frame (frame 2) is shown in Figure 3.6. In this image, mid-grey represents a difference of zero and light or dark greys correspond to positive and negative differences respectively. The obvious problem with this simple prediction is that a lot of energy remains in the residual frame (indicated by the light and dark areas) and this means that there is still a significant amount of information to compress after temporal prediction. Much of the residual energy is due to object movements between the two frames and a better prediction may be formed by *compensating* for motion between the two frames.

3.3.2 Changes due to Motion

Changes between video frames may be caused by object motion (rigid object motion, for example a moving car, and deformable object motion, for example a moving arm), camera motion (panning, tilt, zoom, rotation), uncovered regions (for example, a portion of the scene background uncovered by a moving object) and lighting changes. With the exception of uncovered regions and lighting changes, these differences correspond to pixel movements between frames. It is possible to estimate the trajectory of each pixel between successive video frames, producing a field of pixel trajectories known as the *optical flow* (optic flow) [2]. Figure 3.7 shows the optical flow field for the frames of Figure 3.4 and Figure 3.5. The complete field contains a flow vector for every pixel position but for clarity, the field is sub-sampled so that only the vector for every 2nd pixel is shown.

If the optical flow field is accurately known, it should be possible to form an accurate prediction of most of the pixels of the current frame by moving each pixel from the

Figure 3.4 Frame 1

Figure 3.5 Frame 2

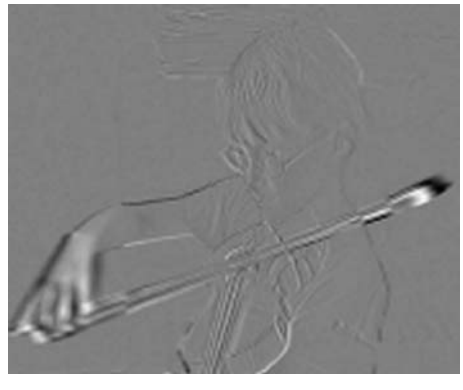

Figure 3.6 Difference

reference frame along its optical flow vector. However, this is not a practical method of motion compensation for several reasons. An accurate calculation of optical flow is very computationally intensive (the more accurate methods use an iterative procedure for every pixel) and it would be necessary to send the optical flow vector for every pixel to the decoder

Figure 3.7 Optical flow

in order for the decoder to re-create the prediction frame (resulting in a large amount of transmitted data and negating the advantage of a small residual).

3.3.3 Block-based Motion Estimation and Compensation

A practical and widely-used method of motion compensation is to compensate for movement of rectangular sections or 'blocks' of the current frame. The following procedure is carried out for each block of $M \times N$ samples in the current frame:

1. Search an area in the reference frame (past or future frame, previously coded and transmitted) to find a 'matching' $M \times N$-sample region. This is carried out by comparing the $M \times N$ block in the current frame with some or all of the possible $M \times N$ regions in the search area (usually a region centred on the current block position) and finding the region that gives the 'best' match. A popular matching criterion is the energy in the residual formed by subtracting the candidate region from the current $M \times N$ block, so that the candidate region that minimises the residual energy is chosen as the best match. This process of finding the best match is known as *motion estimation*.
2. The chosen candidate region becomes the predictor for the current $M \times N$ block and is subtracted from the current block to form a residual $M \times N$ block (*motion compensation*).
3. The residual block is encoded and transmitted and the offset between the current block and the position of the candidate region (*motion vector*) is also transmitted.

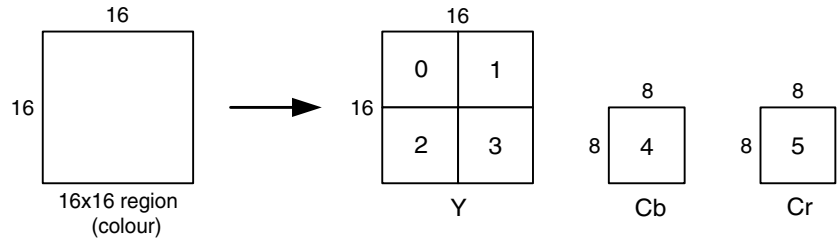

Figure 3.8 Macroblock (4:2:0)

The decoder uses the received motion vector to re-create the predictor region and decodes the residual block, adds it to the predictor and reconstructs a version of the original block.

Block-based motion compensation is popular for a number of reasons. It is relatively straightforward and computationally tractable, it fits well with rectangular video frames and with block-based image transforms (e.g. the Discrete Cosine Transform, see later) and it provides a reasonably effective temporal model for many video sequences. There are however a number of disadvantages, for example 'real' objects rarely have neat edges that match rectangular boundaries, objects often move by a fractional number of pixel positions between frames and many types of object motion are hard to compensate for using block-based methods (e.g. deformable objects, rotation and warping, complex motion such as a cloud of smoke). Despite these disadvantages, block-based motion compensation is the basis of the temporal model used by all current video coding standards.

3.3.4 Motion Compensated Prediction of a Macroblock

The *macroblock*, corresponding to a 16 × 16-pixel region of a frame, is the basic unit for motion compensated prediction in a number of important visual coding standards including MPEG-1, MPEG-2, MPEG-4 Visual, H.261, H.263 and H.264. For source video material in 4:2:0 format (see Chapter 2), a macroblock is organised as shown in Figure 3.8. A 16 × 16-pixel region of the source frame is represented by 256 luminance samples (arranged in four 8 × 8-sample blocks), 64 blue chrominance samples (one 8 × 8 block) and 64 red chrominance samples (8 × 8), giving a total of six 8 × 8 blocks. An MPEG-4 Visual or H.264 CODEC processes each video frame in units of a macroblock.

Motion Estimation

Motion estimation of a macroblock involves finding a 16 × 16-sample region in a reference frame that closely matches the current macroblock. The reference frame is a previously-encoded frame from the sequence and may be before or after the current frame in display order. An area in the reference frame centred on the current macroblock position (the search area) is searched and the 16 × 16 region within the search area that minimises a matching criterion is chosen as the 'best match' (Figure 3.9).

Motion Compensation

The selected 'best' matching region in the reference frame is subtracted from the current macroblock to produce a residual macroblock (luminance and chrominance) that is encoded

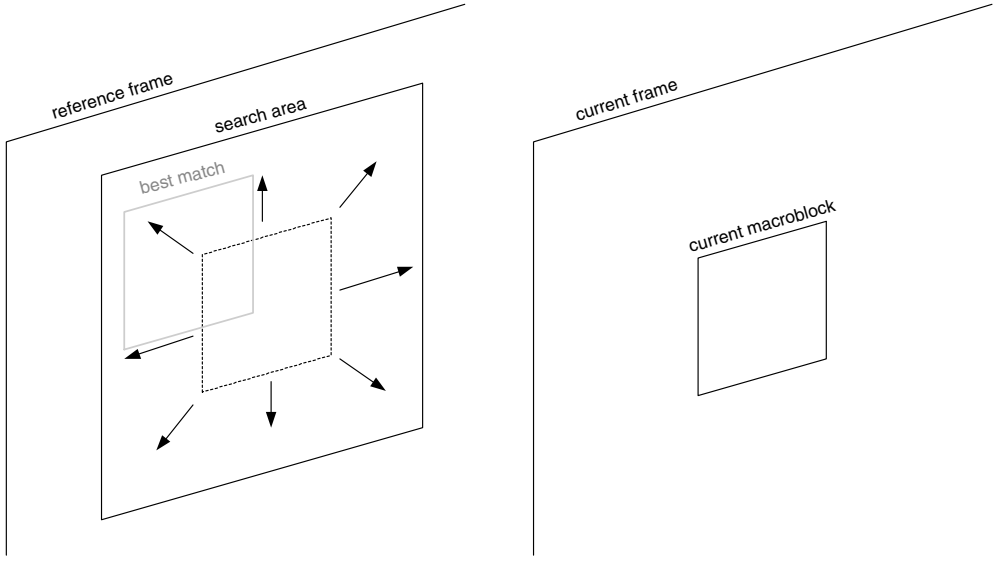

Figure 3.9 Motion estimation

and transmitted together with a motion vector describing the position of the best matching region (relative to the current macroblock position). Within the encoder, the residual is encoded and decoded and added to the matching region to form a reconstructed macroblock which is stored as a reference for further motion-compensated prediction. It is necessary to use a decoded residual to reconstruct the macroblock in order to ensure that encoder and decoder use an identical reference frame for motion compensation.

There are many variations on the basic motion estimation and compensation process. The reference frame may be a previous frame (in temporal order), a future frame or a combination of predictions from two or more previously encoded frames. If a future frame is chosen as the reference, it is necessary to encode this frame before the current frame (i.e. frames must be encoded out of order). Where there is a significant change between the reference and current frames (for example, a scene change), it may be more efficient to encode the macroblock without motion compensation and so an encoder may choose *intra* mode (encoding without motion compensation) or *inter* mode (encoding with motion compensated prediction) for each macroblock. Moving objects in a video scene rarely follow 'neat' 16×16-pixel boundaries and so it may be more efficient to use a variable block size for motion estimation and compensation. Objects may move by a fractional number of pixels between frames (e.g. 2.78 pixels rather than 2.0 pixels in the horizontal direction) and a better prediction may be formed by interpolating the reference frame to sub-pixel positions before searching these positions for the best match.

3.3.5 Motion Compensation Block Size

Two successive frames of a video sequence are shown in Figure 3.10 and Figure 3.11. Frame 1 is subtracted from frame 2 without motion compensation to produce a residual frame

Figure 3.10 Frame 1

Figure 3.11 Frame 2

Figure 3.12 Residual (no motion compensation)

Figure 3.13 Residual (16 × 16 block size)

Figure 3.14 Residual (8 × 8 block size)

(Figure 3. 12). The energy in the residual is reduced by motion compensating each 16 × 16 macroblock (Figure 3.13). Motion compensating each 8 × 8 block (instead of each 16 × 16 macroblock) reduces the residual energy further (Figure 3.14) and motion compensating each 4 × 4 block gives the smallest residual energy of all (Figure 3.15). These examples show that smaller motion compensation block sizes can produce better motion compensation results. However, a smaller block size leads to increased complexity (more search operations must be carried out) and an increase in the number of motion vectors that need to be transmitted. Sending each motion vector requires bits to be sent and the extra overhead for vectors may outweigh the benefit of reduced residual energy. An effective compromise is to adapt the block size to the picture characteristics, for example choosing a large block size in flat, homogeneous regions of a frame and choosing a small block size around areas of high detail and complex

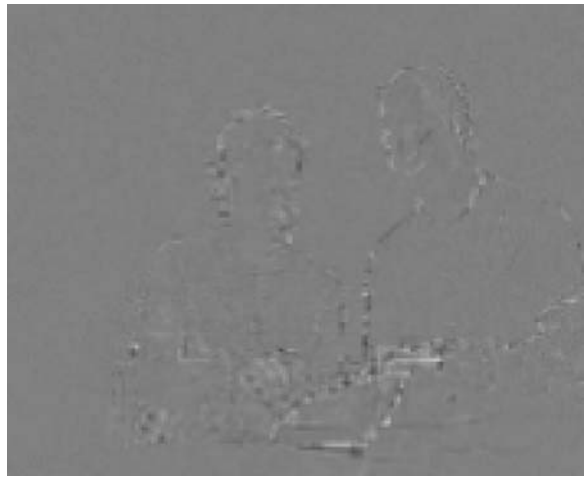

Figure 3.15 Residual (4 × 4 block size)

motion. H.264 uses an adaptive motion compensation block size (Tree Structured motion compensation, described in Chapter 6.

3.3.6 Sub-pixel Motion Compensation

Figure 3.16 shows a close-up view of part of a reference frame. In some cases a better motion compensated prediction may be formed by predicting from interpolated sample positions in the reference frame. In Figure 3.17, the reference region luma samples are interpolated to half-samples positions and it may be possible to find a better match for the current macroblock by searching the interpolated samples. 'Sub-pixel' motion estimation and compensation[1] involves searching sub-sample interpolated positions as well as integer-sample positions, choosing the position that gives the best match (i.e. minimises the residual energy) and using the integer- or sub-sample values at this position for motion compensated prediction. Figure 3.18 shows the concept of a 'quarter-pixel' motion estimation. In the first stage, motion estimation finds the best match on the integer sample grid (circles). The encoder searches the half-sample positions immediately next to this best match (squares) to see whether the match can be improved and if required, the quarter-sample positions next to the best half-sample position (triangles) are then searched. The final match (at an integer, half- or quarter-sample position) is subtracted from the current block or macroblock.

The residual in Figure 3.19 is produced using a block size of 4 × 4 samples using half-sample interpolation and has lower residual energy than Figure 3.15. This approach may be extended further by interpolation onto a quarter-sample grid to give a still smaller residual (Figure 3.20). In general, 'finer' interpolation provides better motion compensation performance (a smaller residual) at the expense of increased complexity. The performance gain tends to diminish as the interpolation steps increase. Half-sample interpolation gives a significant gain over integer-sample motion compensation, quarter-sample interpolation gives a moderate further improvement, eighth-sample interpolation gives a small further improvement again and so on.

[1] The terms 'sub-pixel', 'half-pixel' and 'quarter-pixel' are widely used in this context although in fact the process is usually applied to luma and chroma samples, not pixels.

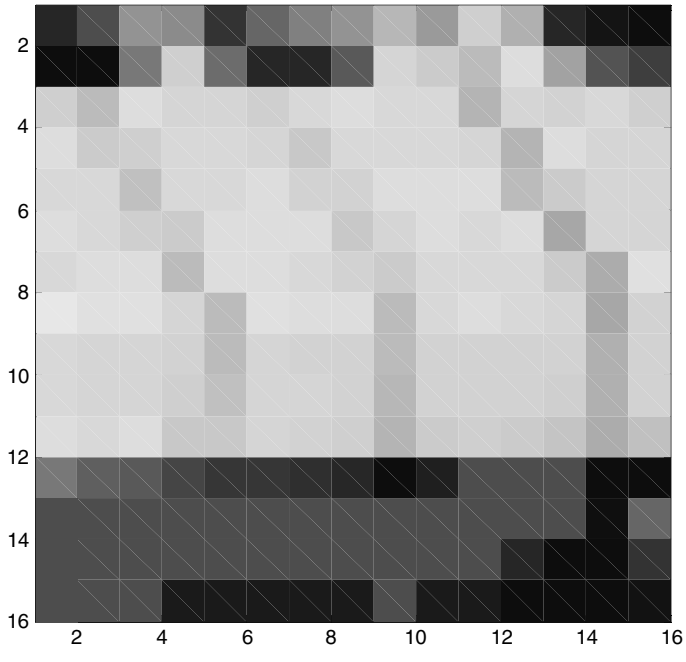

Figure 3.16 Close-up of reference region

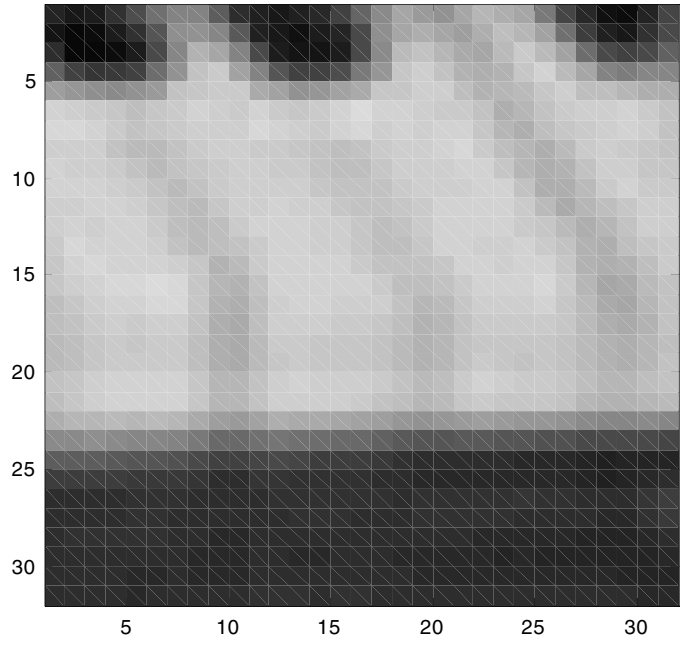

Figure 3.17 Reference region interpolated to half-pixel positions

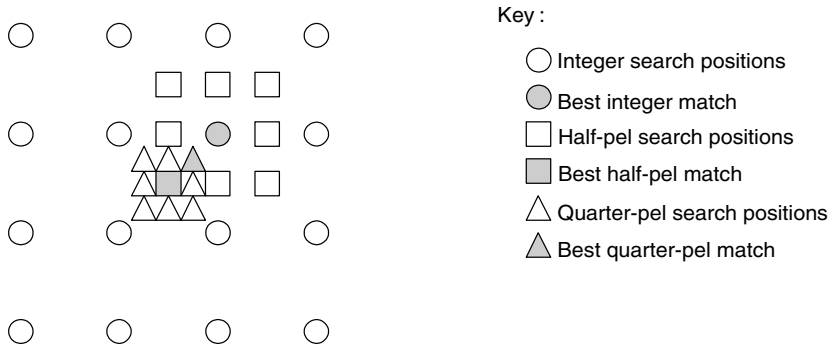

Figure 3.18 Integer, half-pixel and quarter-pixel motion estimation

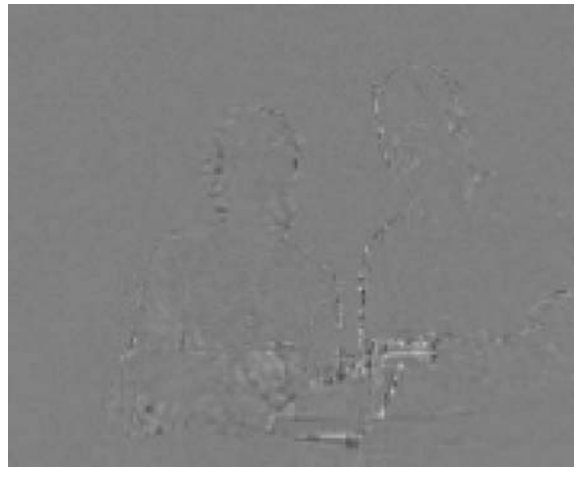

Figure 3.19 Residual (4 × 4 blocks, half-pixel compensation)

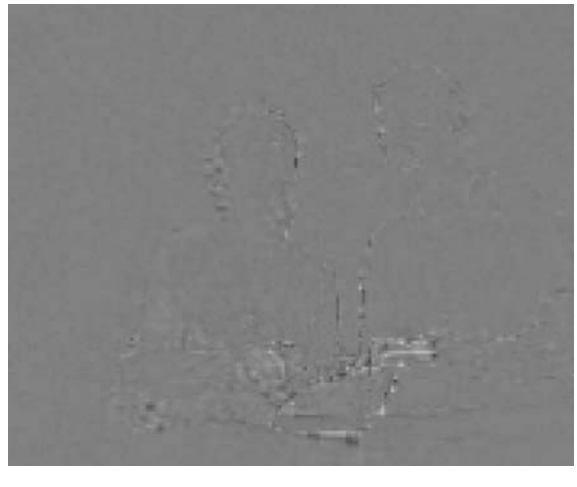

Figure 3.20 Residual (4 × 4 blocks, quarter-pixel compensation)

Table 3.1 SAE of residual frame after motion compensation (16 × 16 block size)

Sequence	No motion compensation	Integer-pel	Half-pel	Quarter-pel
'Violin', QCIF	171945	153475	128320	113744
'Grasses', QCIF	248316	245784	228952	215585
'Carphone', QCIF	102418	73952	56492	47780

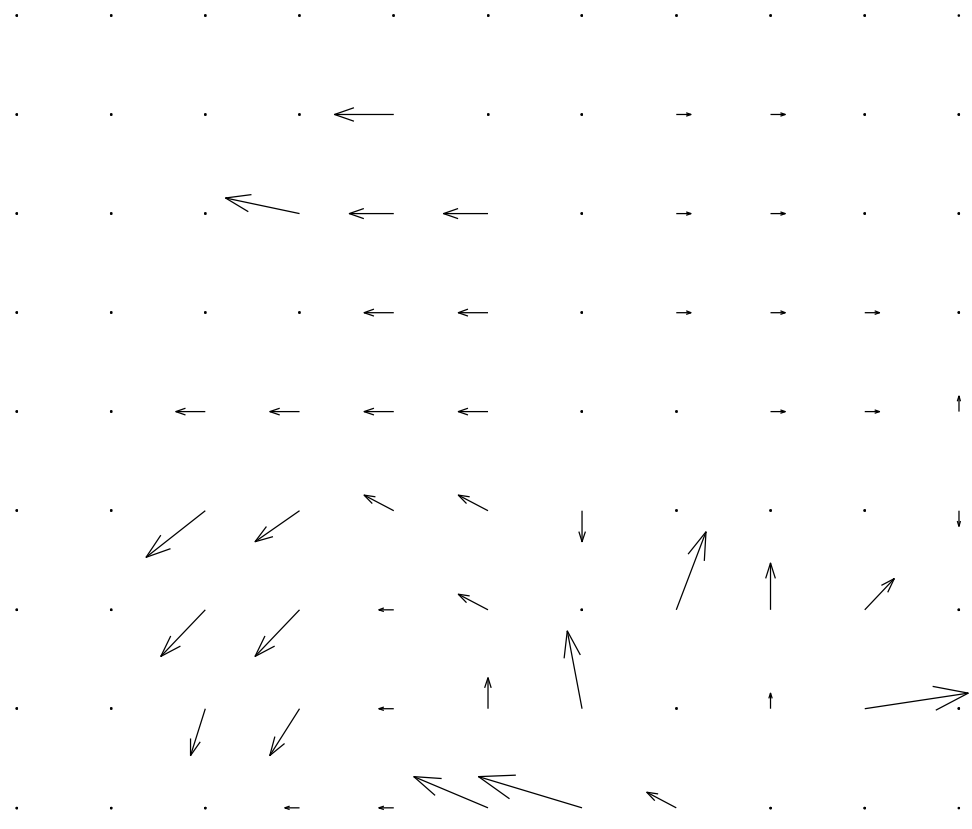

Figure 3.21 Motion vector map (16 × 16 blocks, integer vectors)

Some examples of the performance achieved by sub-pixel motion estimation and compensation are given in Table 3.1. A motion-compensated reference frame (the previous frame in the sequence) is subtracted from the current frame and the energy of the residual (approximated by the Sum of Absolute Errors, SAE) is listed in the table. A lower SAE indicates better motion compensation performance. In each case, sub-pixel motion compensation gives improved performance compared with integer-sample compensation. The improvement from integer to half-sample is more significant than the further improvement from half- to quarter-sample. The sequence 'Grasses' has highly complex motion and is particularly difficult to motion-compensate, hence the large SAE; 'Violin' and 'Carphone' are less complex and motion compensation produces smaller SAE values.

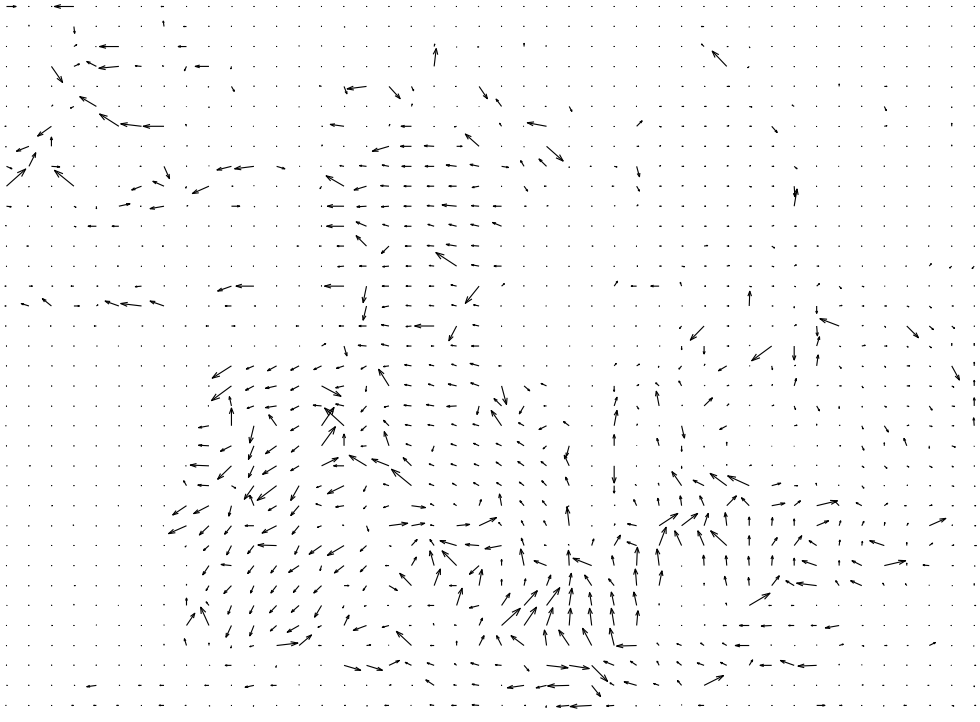

Figure 3.22 Motion vector map (4 × 4 blocks, quarter-pixel vectors)

Searching for matching 4 × 4 blocks with quarter-sample interpolation is considerably more complex than searching for 16 × 16 blocks with no interpolation. In addition to the extra complexity, there is a coding penalty since the vector for every block must be encoded and transmitted to the receiver in order to reconstruct the image correctly. As the block size is reduced, the number of vectors that have to be transmitted increases. More bits are required to represent half- or quarter-sample vectors because the fractional part of the vector (e.g. 0.25, 0.5) must be encoded as well as the integer part. Figure 3.21 plots the integer motion vectors that are required to be transmitted along with the residual of Figure 3.13. The motion vectors required for the residual of Figure 3.20 (4 × 4 block size) are plotted in Figure 3.22, in which there are 16 times as many vectors, each represented by two fractional numbers DX and DY with quarter-pixel accuracy. There is therefore a tradeoff in compression efficiency associated with more complex motion compensation schemes, since more accurate motion compensation requires more bits to encode the vector field but fewer bits to encode the residual whereas less accurate motion compensation requires fewer bits for the vector field but more bits for the residual.

3.3.7 Region-based Motion Compensation

Moving objects in a 'natural' video scene are rarely aligned neatly along block boundaries but are likely to be irregular shaped, to be located at arbitrary positions and (in some cases) to change shape between frames. This problem is illustrated by Figure 3.23, in which the

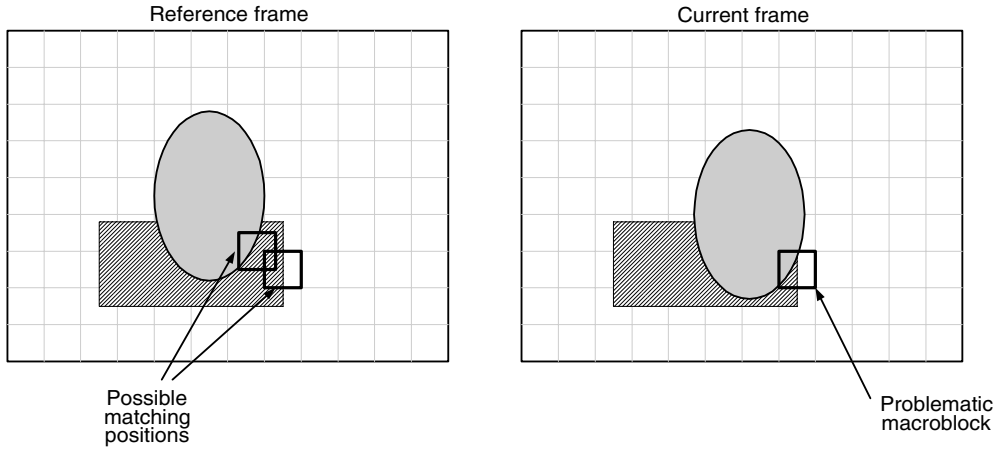

Figure 3.23 Motion compensation of arbitrary-shaped moving objects

oval-shaped object is moving and the rectangular object is static. It is difficult to find a good match in the reference frame for the highlighted macroblock, because it covers part of the moving object and part of the static object. Neither of the two matching positions shown in the reference frame are ideal.

It may be possible to achieve better performance by motion compensating arbitrary regions of the picture (region-based motion compensation). For example, if we only attempt to motion-compensate pixel positions *inside* the oval object then we can find a good match in the reference frame. There are however a number of practical difficulties that need to be overcome in order to use region-based motion compensation, including identifying the region boundaries accurately and consistently, (*segmentation*) signalling (encoding) the contour of the boundary to the decoder and encoding the residual after motion compensation. MPEG-4 Visual includes a number of tools that support region-based compensation and coding and these are described in Chapter 5.

3.4 IMAGE MODEL

A natural video image consists of a grid of sample values. Natural images are often difficult to compress in their original form because of the high correlation between neighbouring image samples. Figure 3.24 shows the two-dimensional autocorrelation function of a natural video image (Figure 3.4) in which the height of the graph at each position indicates the similarity between the original image and a spatially-shifted copy of itself. The peak at the centre of the figure corresponds to zero shift. As the spatially-shifted copy is moved away from the original image in any direction, the function drops off as shown in the figure, with the gradual slope indicating that image samples within a local neighbourhood are highly correlated.

A motion-compensated residual image such as Figure 3.20 has an autocorrelation function (Figure 3.25) that drops off rapidly as the spatial shift increases, indicating that neighbouring samples are weakly correlated. Efficient motion compensation reduces local correlation in the residual making it easier to compress than the original video frame. The function of the *image*

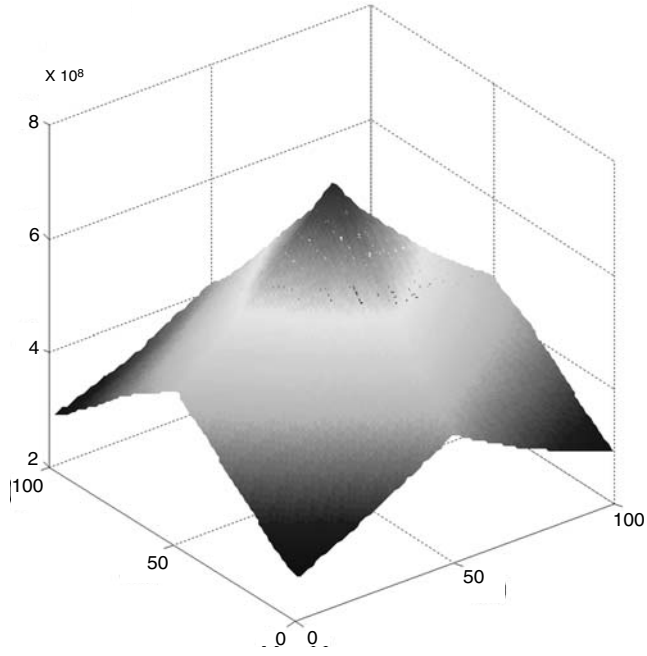

Figure 3.24 2D autocorrelation function of image

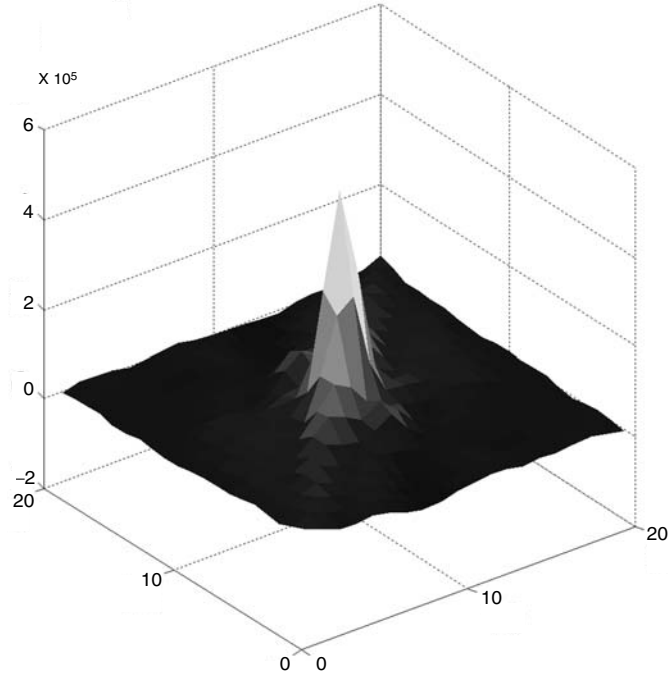

Figure 3.25 2D autocorrelation function of residual

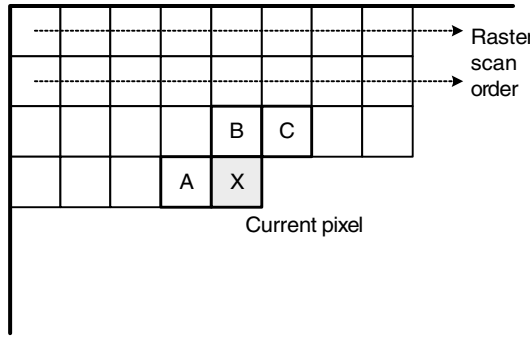

Figure 3.26 Spatial prediction (DPCM)

model is to decorrelate image or residual data further and to convert it into a form that can be efficiently compressed using an entropy coder. Practical image models typically have three main components, transformation (decorrelates and compacts the data), quantisation (reduces the precision of the transformed data) and reordering (arranges the data to group together significant values).

3.4.1 Predictive Image Coding

Motion compensation is an example of predictive coding in which an encoder creates a prediction of a region of the current frame based on a previous (or future) frame and subtracts this prediction from the current region to form a residual. If the prediction is successful, the energy in the residual is lower than in the original frame and the residual can be represented with fewer bits.

In a similar way, a prediction of an image sample or region may be formed from previously-transmitted samples in the same image or frame. Predictive coding was used as the basis for early image compression algorithms and is an important component of H.264 Intra coding (applied in the transform domain, see Chapter 6). Spatial prediction is sometimes described as 'Differential Pulse Code Modulation' (DPCM), a term borrowed from a method of differentially encoding PCM samples in telecommunication systems.

Figure 3.26 shows a pixel X that is to be encoded. If the frame is processed in raster order, then pixels A, B and C (neighbouring pixels in the current and previous rows) are available in both the encoder and the decoder (since these should already have been decoded before X). The encoder forms a prediction for X based on some combination of previously-coded pixels, subtracts this prediction from X and encodes the residual (the result of the subtraction). The decoder forms the same prediction and adds the decoded residual to reconstruct the pixel.

Example

Encoder prediction $P(X) = (2A + B + C)/4$
Residual $R(X) = X - P(X)$ is encoded and transmitted.

Decoder decodes R(X) and forms the same prediction: $P(X) = (2A + B + C)/4$
Reconstructed pixel $X = R(X) + P(X)$

If the encoding process is lossy (e.g. if the residual is quantised – see section 3.4.3) then the decoded pixels A′, B′ and C′ may not be identical to the original A, B and C (due to losses during encoding) and so the above process could lead to a cumulative mismatch (or 'drift') between the encoder and decoder. In this case, the encoder should itself decode the residual R′(X) and reconstruct each pixel.

The encoder uses decoded pixels A′, B′ and C′ to form the prediction, i.e. $P(X) = (2A' + B' + C')/4$ in the above example. In this way, both encoder and decoder use the same prediction P(X) and drift is avoided.

The compression efficiency of this approach depends on the accuracy of the prediction P(X). If the prediction is accurate (P(X) is a close approximation of X) then the residual energy will be small. However, it is usually not possible to choose a predictor that works well for all areas of a complex image and better performance may be obtained by adapting the predictor depending on the local statistics of the image (for example, using different predictors for areas of flat texture, strong vertical texture, strong horizontal texture, etc.). It is necessary for the encoder to indicate the choice of predictor to the decoder and so there is a tradeoff between efficient prediction and the extra bits required to signal the choice of predictor.

3.4.2 Transform Coding

3.4.2.1 Overview

The purpose of the transform stage in an image or video CODEC is to convert image or motion-compensated residual data into another domain (the transform domain). The choice of transform depends on a number of criteria:

1. Data in the transform domain should be decorrelated (separated into components with minimal inter-dependence) and compact (most of the energy in the transformed data should be concentrated into a small number of values).
2. The transform should be reversible.
3. The transform should be computationally tractable (low memory requirement, achievable using limited-precision arithmetic, low number of arithmetic operations, etc.).

Many transforms have been proposed for image and video compression and the most popular transforms tend to fall into two categories: block-based and image-based. Examples of block-based transforms include the Karhunen–Loeve Transform (KLT), Singular Value Decomposition (SVD) and the ever-popular Discrete Cosine Transform (DCT) [3]. Each of these operate on blocks of $N \times N$ image or residual samples and hence the image is processed in units of a block. Block transforms have low memory requirements and are well-suited to compression of block-based motion compensation residuals but tend to suffer from artefacts at block edges ('blockiness'). Image-based transforms operate on an entire image or frame (or a large section of the image known as a 'tile'). The most popular image transform is the Discrete Wavelet Transform (DWT or just 'wavelet'). Image transforms such as the DWT have been shown to out-perform block transforms for still image compression but they tend to have higher memory requirements (because the whole image or tile is processed as a unit) and

do not 'fit' well with block-based motion compensation. The DCT and the DWT both feature in MPEG-4 Visual (and a variant of the DCT is incorporated in H.264) and are discussed further in the following sections.

3.4.2.2 DCT

The Discrete Cosine Transform (DCT) operates on **X**, a block of $N \times N$ samples (typically image samples or residual values after prediction) and creates **Y**, an $N \times N$ block of coefficients. The action of the DCT (and its inverse, the IDCT) can be described in terms of a transform matrix **A**. The forward DCT (FDCT) of an $N \times N$ sample block is given by:

$$\mathbf{Y} = \mathbf{A}\mathbf{X}\mathbf{A}^{\mathrm{T}} \tag{3.1}$$

and the inverse DCT (IDCT) by:

$$\mathbf{X} = \mathbf{A}^{\mathrm{T}}\mathbf{Y}\mathbf{A} \tag{3.2}$$

where **X** is a matrix of samples, **Y** is a matrix of coefficients and **A** is an $N \times N$ transform matrix. The elements of **A** are:

$$A_{ij} = C_i \cos\frac{(2j+1)i\pi}{2N} \qquad \text{where } C_i = \sqrt{\frac{1}{N}}\ (i=0), \qquad C_i = \sqrt{\frac{2}{N}}\ (i>0) \tag{3.3}$$

Equation 3.1 and equation 3.2 may be written in summation form:

$$Y_{xy} = C_x C_y \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} X_{ij} \cos\frac{(2j+1)y\pi}{2N} \cos\frac{(2i+1)x\pi}{2N} \tag{3.4}$$

$$X_{ij} = \sum_{x=0}^{N-1}\sum_{y=0}^{N-1} C_x C_y Y_{xy} \cos\frac{(2j+1)y\pi}{2N} \cos\frac{(2i+1)x\pi}{2N} \tag{3.5}$$

Example: $N = 4$

The transform matrix **A** for a 4×4 DCT is:

$$\mathbf{A} = \begin{bmatrix} \frac{1}{2}\cos(0) & \frac{1}{2}\cos(0) & \frac{1}{2}\cos(0) & \frac{1}{2}\cos(0) \\ \sqrt{\frac{1}{2}}\cos\left(\frac{\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{5\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{7\pi}{8}\right) \\ \sqrt{\frac{1}{2}}\cos\left(\frac{2\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{6\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{10\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{14\pi}{8}\right) \\ \sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{9\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{15\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{21\pi}{8}\right) \end{bmatrix} \tag{3.6}$$

The cosine function is symmetrical and repeats after 2π radians and hence $\mathbf{A}$ can be simplified to:

$$\mathbf{A} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \sqrt{\frac{1}{2}}\cos\left(\frac{\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) & -\sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) & -\sqrt{\frac{1}{2}}\cos\left(\frac{\pi}{8}\right) \\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ \sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) & -\sqrt{\frac{1}{2}}\cos\left(\frac{\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{\pi}{8}\right) & -\sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) \end{bmatrix} \tag{3.7}$$

or

$$\mathbf{A} = \begin{bmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & b & c \end{bmatrix} \quad \text{where } a = \frac{1}{2},\ b = \sqrt{\frac{1}{2}}\cos\left(\frac{\pi}{8}\right),\ c = \sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) \tag{3.8}$$

Evaluating the cosines gives:

$$\mathbf{A} = \begin{bmatrix} 0.5 & 0.5 & 0.5 & 0.5 \\ 0.653 & 0.271 & 0.271 & -0.653 \\ 0.5 & -0.5 & -0.5 & 0.5 \\ 0.271 & -0.653 & -0.653 & 0.271 \end{bmatrix}$$

The output of a two-dimensional FDCT is a set of $N \times N$ coefficients representing the image block data in the DCT domain and these coefficients can be considered as 'weights' of a set of standard *basis patterns*. The basis patterns for the 4×4 and 8×8 DCTs are shown in Figure 3.27 and Figure 3.28 respectively and are composed of combinations of horizontal and vertical cosine functions. Any image block may be reconstructed by combining all $N \times N$ basis patterns, with each basis multiplied by the appropriate weighting factor (coefficient).

Example 1 Calculating the DCT of a 4×4 block

$\mathbf{X}$ is 4×4 block of samples from an image:

	$j=0$	1	2	3
$i=0$	5	11	8	10
1	9	8	4	12
2	1	10	11	4
3	19	6	15	7

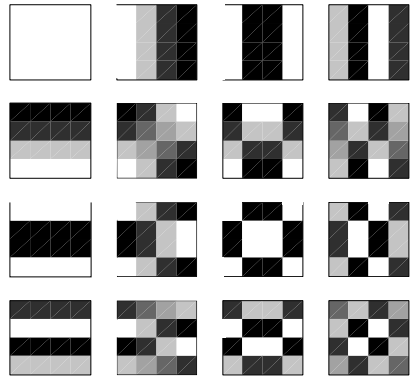

Figure 3.27 4 × 4 DCT basis patterns

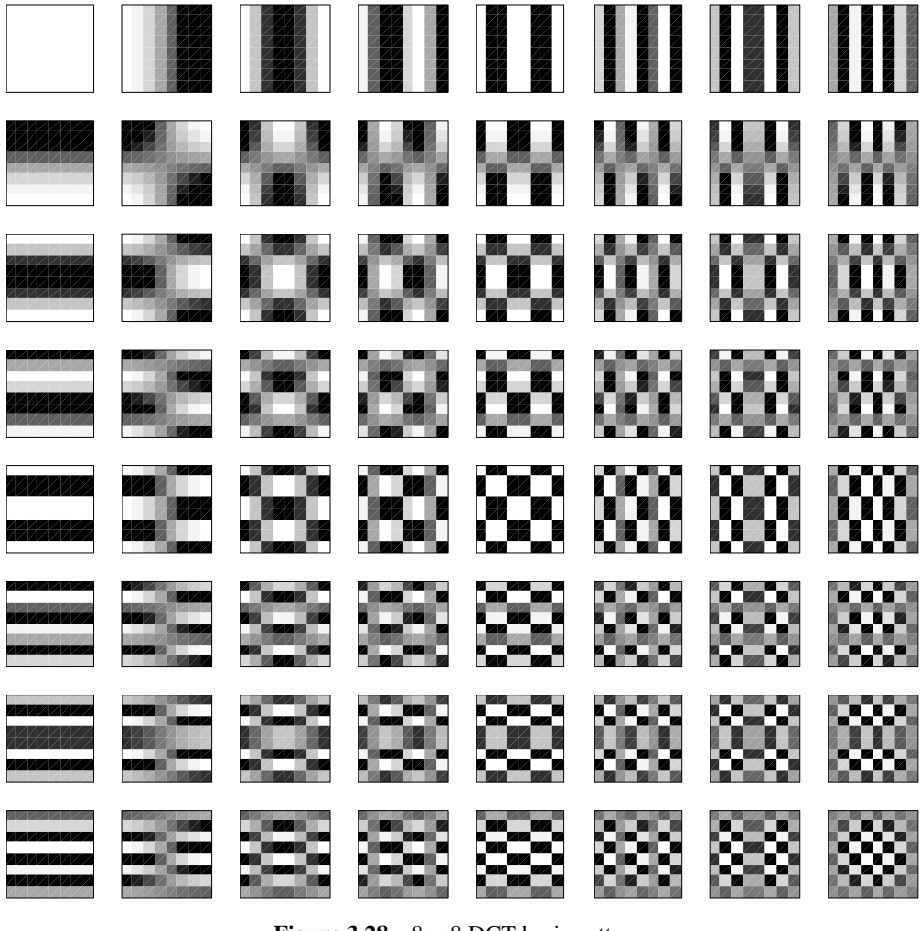

Figure 3.28 8 × 8 DCT basis patterns

The Forward DCT of $\mathbf{X}$ is given by: $\mathbf{Y} = \mathbf{AXA}^{\mathrm{T}}$. The first matrix multiplication, $\mathbf{Y}' = \mathbf{AX}$, corresponds to calculating the one-dimensional DCT of each *column* of $\mathbf{X}$. For example, Y'_{00} is calculated as follows:

$$Y'_{00} = A_{00}X_{00} + A_{01}X_{10} + A_{02}X_{20} + A_{03}X_{30} = (0.5 * 5) + (0.5 * 9) + (0.5 * 1) + (0.5 * 19) = 17.0$$

The complete result of the column calculations is:

$$\mathbf{Y}' = \mathbf{AX} = \begin{bmatrix} 17 & 17.5 & 19 & 16.5 \\ -6.981 & 2.725 & -6.467 & 4.125 \\ 7 & -0.5 & 4 & 0.5 \\ -9.015 & 2.660 & 2.679 & -4.414 \end{bmatrix}$$

Carrying out the second matrix multiplication, $\mathbf{Y} = \mathbf{Y}'\mathbf{A}^{\mathrm{T}}$, is equivalent to carrying out a 1-D DCT on each *row* of $\mathbf{Y}'$:

$$\mathbf{Y} = \mathbf{AXA}^{\mathrm{T}} = \begin{bmatrix} 35.0 & -0.079 & -1.5 & 1.115 \\ -3.299 & -4.768 & 0.443 & -9.010 \\ 5.5 & 3.029 & 2.0 & 4.699 \\ -4.045 & -3.010 & -9.384 & -1.232 \end{bmatrix}$$

(Note: the order of the row and column calculations does not affect the final result).

Example 2 Image block and DCT coefficients

Figure 3.29 shows an image with a 4×4 block selected and Figure 3.30 shows the block in close-up, together with the DCT coefficients. The advantage of representing the block in the DCT domain is not immediately obvious since there is no reduction in the amount of data; instead of 16 pixel values, we need to store 16 DCT coefficients. The usefulness of the DCT becomes clear when the block is reconstructed from a subset of the coefficients.

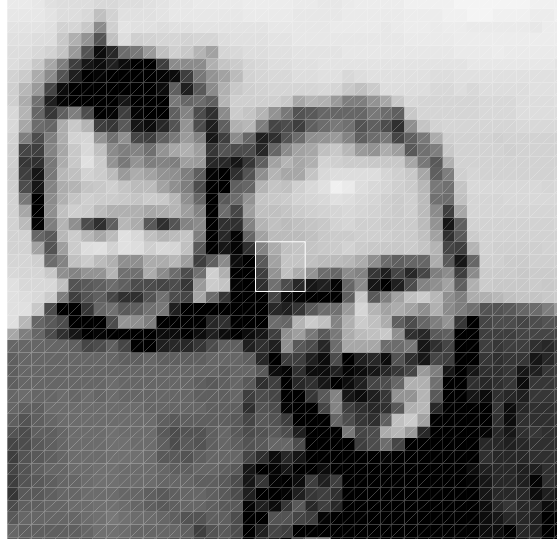

Figure 3.29 Image section showing 4×4 block

Figure 3.30 Close-up of 4 × 4 block; DCT coefficients

Setting all the coefficients to zero except the most significant (coefficient 0,0, described as the 'DC' coefficient) and performing the IDCT gives the output block shown in Figure 3.31(a), the mean of the original pixel values. Calculating the IDCT of the two most significant coefficients gives the block shown in Figure 3.31(b). Adding more coefficients before calculating the IDCT produces a progressively more accurate reconstruction of the original block and by the time five coefficients are included (Figure 3.31(d)), the reconstructed block is a reasonably close match to the original. Hence it is possible to reconstruct an approximate copy of the block from a subset of the 16 DCT coefficients. Removing the coefficients with insignificant magnitudes (for example by quantisation, see Section 3.4.3) enables image data to be represented with a reduced number of coefficient values at the expense of some loss of quality.

3.4.2.3 Wavelet

The popular 'wavelet transform' (widely used in image compression is based on sets of filters with coefficients that are equivalent to discrete wavelet functions [4]. The basic operation of a discrete wavelet transform is as follows, applied to a discrete signal containing N samples. A pair of filters are applied to the signal to decompose it into a low frequency band (L) and a high frequency band (H). Each band is subsampled by a factor of two, so that the two frequency bands each contain $N/2$ samples. With the correct choice of filters, this operation is reversible.

This approach may be extended to apply to a two-dimensional signal such as an intensity image (Figure 3.32). Each row of a 2D image is filtered with a low-pass and a high-pass filter (L_x and H_x) and the output of each filter is down-sampled by a factor of two to produce the intermediate images L and H. L is the original image low-pass filtered and downsampled in the x-direction and H is the original image high-pass filtered and downsampled in the x-direction. Next, each column of these new images is filtered with low- and high-pass filters (L_y and H_y) and down-sampled by a factor of two to produce four sub-images (LL, LH, HL and HH). These four 'sub-band' images can be combined to create an output image with the same number of samples as the original (Figure 3.33). 'LL' is the original image, low-pass filtered in horizontal and vertical directions and subsampled by a factor of 2. 'HL' is high-pass filtered in the vertical direction and contains residual vertical frequencies, 'LH' is high-pass filtered in the horizontal direction and contains residual horizontal frequencies and 'HH' is high-pass filtered in both horizontal and vertical directions. Between them, the four subband

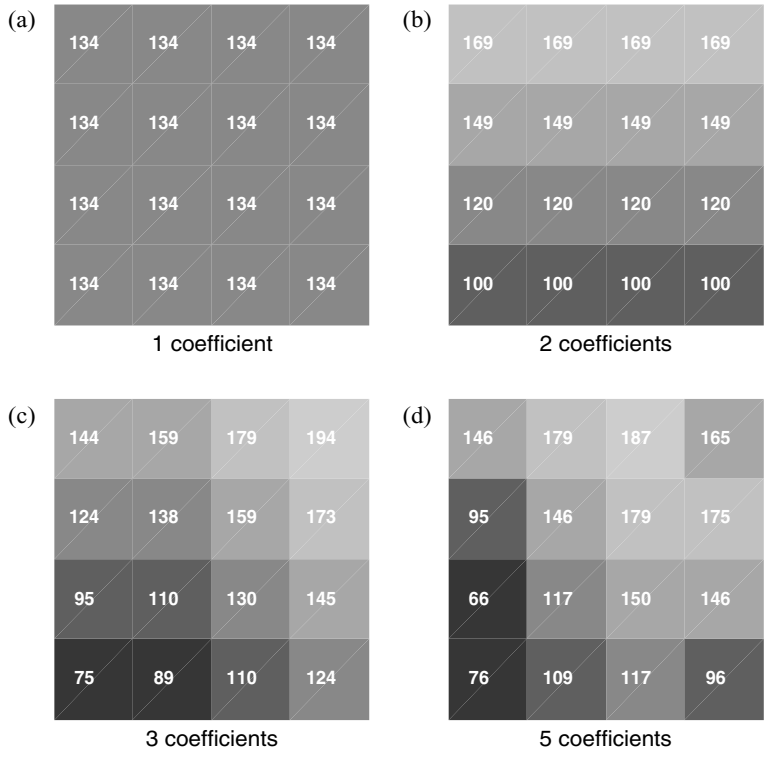

Figure 3.31 Block reconstructed from (a) one, (b) two, (c) three, (d) five coefficients

images contain all of the information present in the original image but the sparse nature of the LH, HL and HH subbands makes them amenable to compression.

In an image compression application, the two-dimensional wavelet decomposition described above is applied again to the 'LL' image, forming four new subband images. The resulting low-pass image (always the top-left subband image) is iteratively filtered to create a tree of subband images. Figure 3.34 shows the result of two stages of this decomposition and Figure 3.35 shows the result of five stages of decomposition. Many of the samples (coefficients) in the higher-frequency subband images are close to zero (near-black) and it is possible to achieve compression by removing these insignificant coefficients prior to transmission. At the decoder, the original image is reconstructed by repeated up-sampling, filtering and addition (reversing the order of operations shown in Figure 3.32).

3.4.3 Quantisation

A quantiser maps a signal with a range of values X to a quantised signal with a reduced range of values Y. It should be possible to represent the quantised signal with fewer bits than the original since the range of possible values is smaller. A *scalar quantiser* maps one sample of the input signal to one quantised output value and a *vector quantiser* maps a group of input samples (a 'vector') to a group of quantised values.

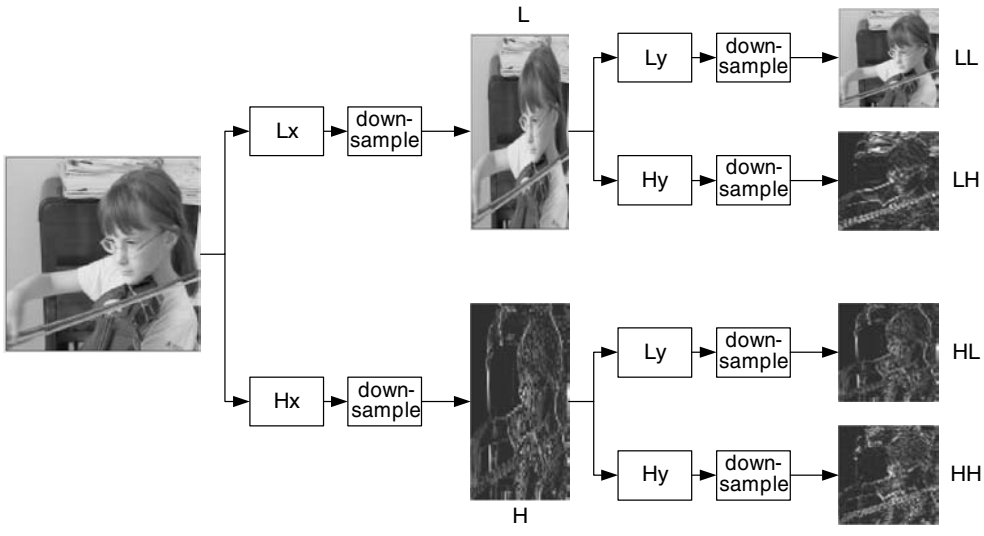

Figure 3.32 Two-dimensional wavelet decomposition process

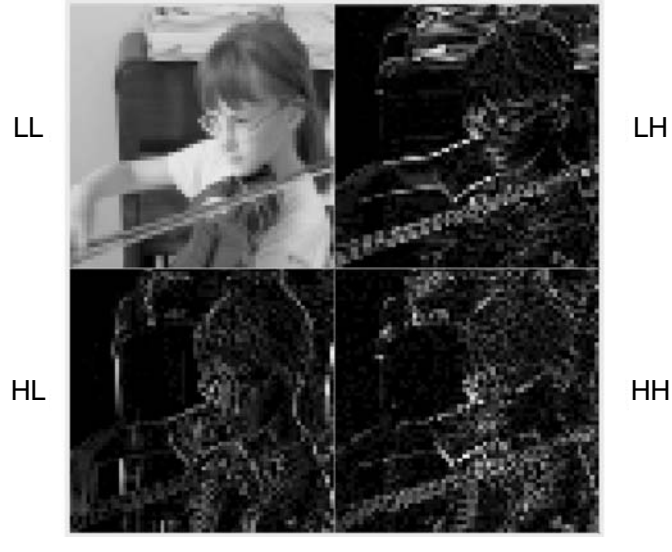

Figure 3.33 Image after one level of decomposition

3.4.3.1 Scalar Quantisation

A simple example of scalar quantisation is the process of rounding a fractional number to the nearest integer, i.e. the mapping is from R to Z. The process is lossy (not reversible) since it is not possible to determine the exact value of the original fractional number from the rounded integer.

Figure 3.34 Two-stage wavelet decomposition of image

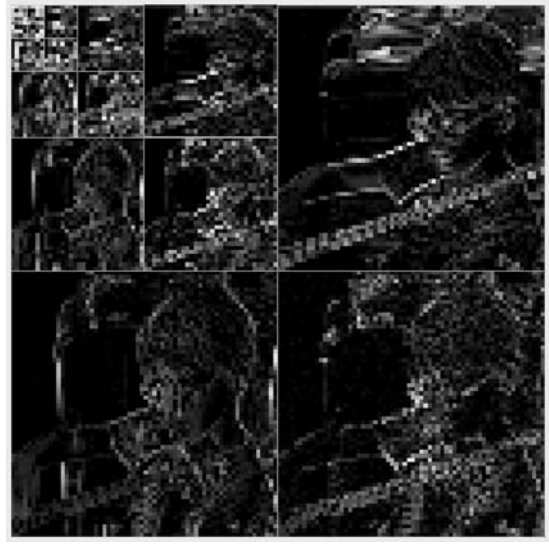

Figure 3.35 Five-stage wavelet decomposition of image

A more general example of a uniform quantiser is:

$$FQ = round\left(\frac{X}{QP}\right)$$
$$Y = FQ.QP \tag{3.9}$$

where QP is a quantisation 'step size'. The quantised output levels are spaced at uniform intervals of QP (as shown in the following example).

Example $Y = QP.round(X/QP)$

	Y			
X	$QP = 1$	$QP = 2$	$QP = 3$	$QP = 5$
−4	−4	−4	−3	−5
−3	−3	−2	−3	−5
−2	−2	−2	−3	0
−1	−1	0	0	0
0	0	0	0	0
1	1	0	0	0
2	2	2	3	0
3	3	2	3	5
4	4	4	3	5
5	5	4	6	5
6	6	6	6	5
7	7	6	6	5
8	8	8	9	10
9	9	8	9	10
10	10	10	9	10
11	11	10	12	10
.....				

Figure 3.36 shows two examples of scalar quantisers, a linear quantiser (with a linear mapping between input and output values) and a nonlinear quantiser that has a 'dead zone' about zero (in which small-valued inputs are mapped to zero).

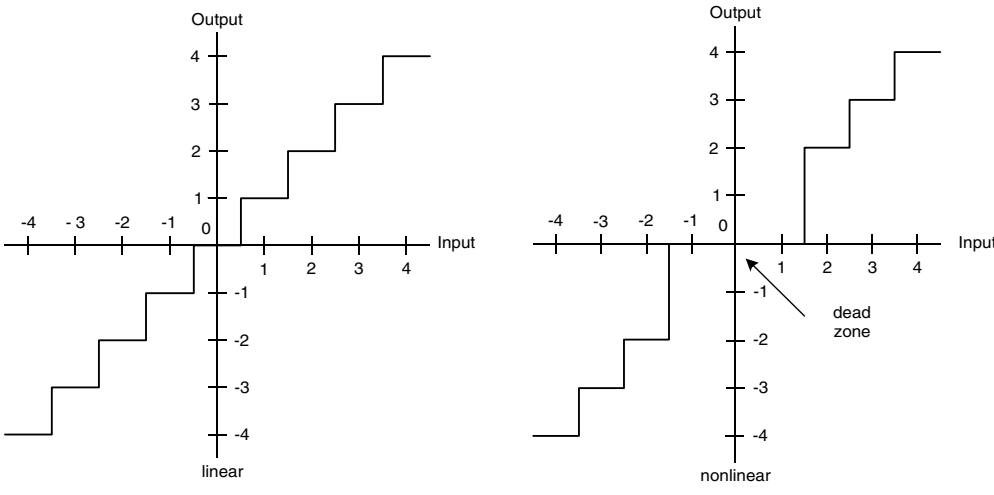

Figure 3.36 Scalar quantisers: linear; nonlinear with dead zone

In image and video compression CODECs, the quantisation operation is usually made up of two parts: a forward quantiser FQ in the encoder and an 'inverse quantiser' or (IQ) in the decoder (in fact quantization is not reversible and so a more accurate term is 'scaler' or 'rescaler'). A critical parameter is the *step size QP* between successive re-scaled values. If the step size is large, the range of quantised values is small and can therefore be efficiently represented (highly compressed) during transmission, but the re-scaled values are a crude approximation to the original signal. If the step size is small, the re-scaled values match the original signal more closely but the larger range of quantised values reduces compression efficiency.

Quantisation may be used to reduce the precision of image data after applying a transform such as the DCT or wavelet transform removing remove insignificant values such as near-zero DCT or wavelet coefficients. The forward quantiser in an image or video encoder is designed to map insignificant coefficient values to zero whilst retaining a reduced number of significant, nonzero coefficients. The output of a forward quantiser is typically a 'sparse' array of quantised coefficients, mainly containing zeros.

3.4.3.2 Vector Quantisation

A vector quantiser maps a set of input data (such as a block of image samples) to a single value (codeword) and, at the decoder, each codeword maps to an approximation to the original set of input data (a 'vector'). The set of vectors are stored at the encoder and decoder in a codebook. A typical application of vector quantisation to image compression [5] is as follows:

1. Partition the original image into regions (e.g. $M \times N$ pixel blocks).
2. Choose a vector from the codebook that matches the current region as closely as possible.
3. Transmit an index that identifies the chosen vector to the decoder.
4. At the decoder, reconstruct an approximate copy of the region using the selected vector.

A basic system is illustrated in Figure 3.37. Here, quantisation is applied in the spatial domain (i.e. groups of image samples are quantised as vectors) but it could equally be applied to

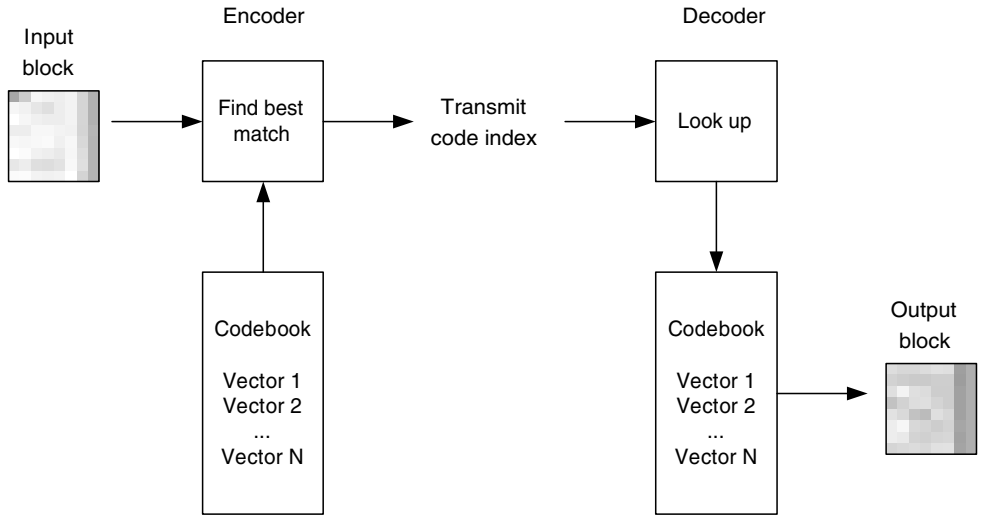

Figure 3.37 Vector quantisation

motion compensated and/or transformed data. Key issues in vector quantiser design include the design of the codebook and efficient searching of the codebook to find the optimal vector.

3.4.4 Reordering and Zero Encoding

Quantised transform coefficients are required to be encoded as compactly as possible prior to storage and transmission. In a transform-based image or video encoder, the output of the quantiser is a sparse array containing a few nonzero coefficients and a large number of zero-valued coefficients. Reordering (to group together nonzero coefficients) and efficient representation of zero coefficients are applied prior to entropy encoding. These processes are described for the DCT and wavelet transform.

3.4.4.1 DCT

Coefficient Distribution

The significant DCT coefficients of a block of image or residual samples are typically the 'low frequency' positions around the DC (0,0) coefficient. Figure 3.38 plots the probability of nonzero DCT coefficients at each position in an 8 × 8 block in a QCIF residual *frame* (Figure 3.6). The nonzero DCT coefficients are clustered around the top-left (DC) coefficient and the distribution is roughly symmetrical in the horizontal and vertical directions. For a residual *field* (Figure 3.39), Figure 3.40 plots the probability of nonzero DCT coefficients; here, the coefficients are clustered around the DC position but are 'skewed', i.e. more nonzero

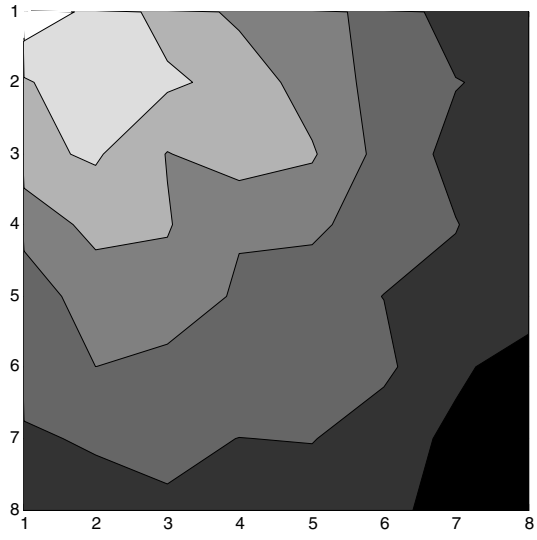

Figure 3.38 8×8 DCT coefficient distribution (frame)

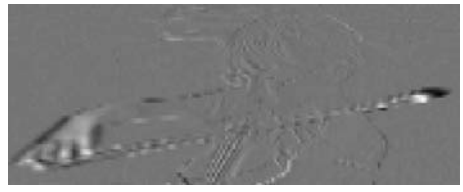

Figure 3.39 Residual field picture

coefficients occur along the left-hand edge of the plot. This is because the field picture has a stronger high-frequency component in the vertical axis (due to the subsampling in the vertical direction) resulting in larger DCT coefficients corresponding to vertical frequencies (refer to Figure 3.27).

Scan

After quantisation, the DCT coefficients for a block are reordered to group together nonzero coefficients, enabling efficient representation of the remaining zero-valued quantised coefficients. The optimum reordering path (scan order) depends on the distribution of nonzero DCT coefficients. For a typical frame block with a distribution similar to Figure 3.38, a suitable scan order is a zigzag starting from the DC (top-left) coefficient. Starting with the DC coefficient, each quantised coefficient is copied into a one-dimensional array in the order shown in Figure 3.41. Nonzero coefficients tend to be grouped together at the start of the reordered array, followed by long sequences of zeros.

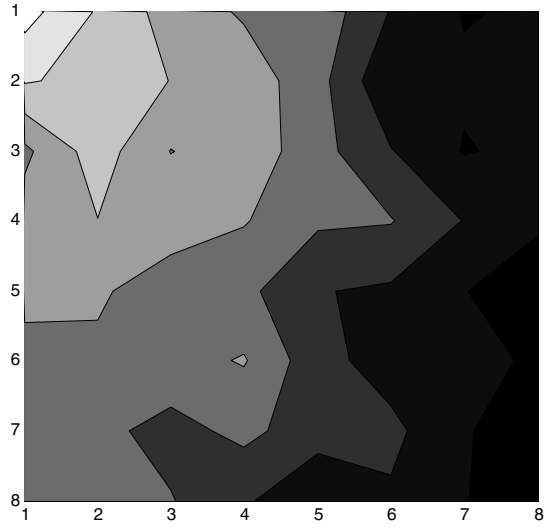

Figure 3.40 8 × 8 DCT coefficient distribution (field)

The zig-zag scan may not be ideal for a field block because of the skewed coefficient distribution (Figure 3.40) and a modified scan order such as Figure 3.42 may be more effective, in which coefficients on the left-hand side of the block are scanned before those on the right-hand side.

Run-Level Encoding

The output of the reordering process is an array that typically contains one or more clusters of nonzero coefficients near the start, followed by strings of zero coefficients. The large number of zero values may be encoded to represent them more compactly, for example by representing the array as a series of (run, level) pairs where *run* indicates the number of zeros preceding a nonzero coefficient and *level* indicates the magnitude of the nonzero coefficient.

Example

Input array: 16,0,0,−3,5,6,0,0,0,0,−7, . . .
Output values: (0,16),(2,−3),(0,5),(0,6),(4,−7). . .

Each of these output values (a run-level pair) is encoded as a separate symbol by the entropy encoder.

Higher-frequency DCT coefficients are very often quantised to zero and so a reordered block will usually end in a run of zeros. A special case is required to indicate the final nonzero coefficient in a block. In so-called 'Two-dimensional' run-level encoding is used, each run-level pair is encoded as above and a separate code symbol, '*last*', indicates the end of the nonzero values. If 'Three-dimensional' run-level encoding is used, each symbol encodes

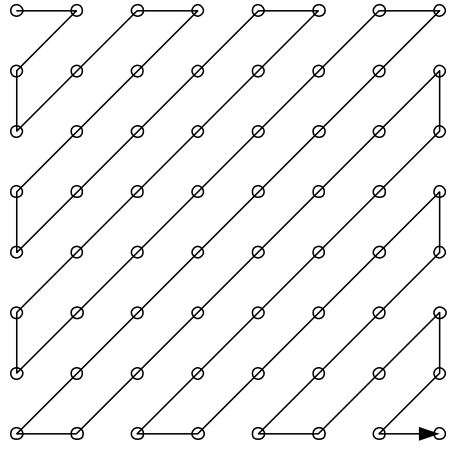

Figure 3.41 Zigzag scan order (frame block)

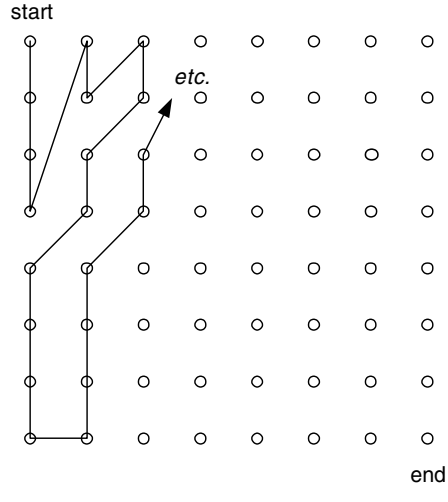

Figure 3.42 Zigzag scan order (field block)

three quantities, *run*, *level* and *last*. In the example above, if –7 is the final nonzero coefficient, the 3D values are:

$$(0, 16, 0), (2, -3, 0), (0, 5, 0), (0, 6, 0), (4, -7, 1)$$

The 1 in the final code indicates that this is the last nonzero coefficient in the block.

3.4.4.2 Wavelet

Coefficient Distribution

Figure 3.35 shows a typical distribution of 2D wavelet coefficients. Many coefficients in higher sub-bands (towards the bottom-right of the figure) are near zero and may be quantised

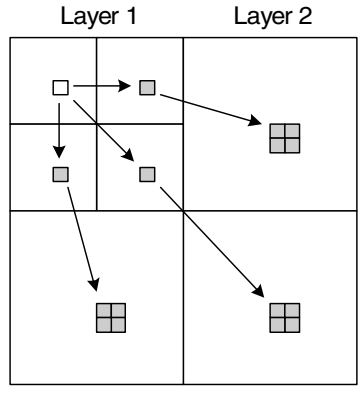

Figure 3.43 Wavelet coefficient and 'children'

to zero without significant loss of image quality. Nonzero coefficients tend to correspond to structures in the image; for example, the violin bow appears as a clear horizontal structure in all the horizontal and diagonal subbands. When a coefficient in a lower-frequency subband is nonzero, there is a strong probability that coefficients in the corresponding position in higher-frequency subbands will also be nonzero. We may consider a 'tree' of nonzero quantised coefficients, starting with a 'root' in a low-frequency subband. Figure 3.43 illustrates this concept. A single coefficient in the LL band of layer 1 has one corresponding coefficient in each of the other bands of layer 1 (i.e. these four coefficients correspond to the same region in the original image). The layer 1 coefficient position maps to four corresponding child coefficient positions in each subband at layer 2 (recall that the layer 2 subbands have twice the horizontal and vertical resolution of the layer 1 subbands).

Zerotree Encoding

It is desirable to encode the nonzero wavelet coefficients as compactly as possible prior to entropy coding [6]. An efficient way of achieving this is to encode each tree of nonzero coefficients starting from the lowest (root) level of the decomposition. A coefficient at the lowest layer is encoded, followed by its child coefficients at the next higher layer, and so on. The encoding process continues until the tree reaches a zero-valued coefficient. Further children of a zero valued coefficient are likely to be zero themselves and so the remaining children are represented by a single code that identifies a tree of zeros (*zerotree*). The decoder reconstructs the coefficient map starting from the root of each tree; nonzero coefficients are decoded and reconstructed and when a zerotree code is reached, all remaining 'children' are set to zero. This is the basis of the *embedded zero tree* (EZW) method of encoding wavelet coefficients. An extra possibility is included in the encoding process, where a zero coefficient may be followed by (a) a zero tree (as before) or (b) a nonzero child coefficient. Case (b) does not occur very often but reconstructed image quality is slightly improved by catering for the occasional occurrences of case (b).

3.5 ENTROPY CODER

The entropy encoder converts a series of symbols representing elements of the video sequence into a compressed bitstream suitable for transmission or storage. Input symbols may include quantised transform coefficients (run-level or zerotree encoded as described in Section 3.4.4), motion vectors (an x and y displacement vector for each motion-compensated block, with integer or sub-pixel resolution), markers (codes that indicate a resynchronisation point in the sequence), headers (macroblock headers, picture headers, sequence headers, etc.) and supplementary information ('side' information that is not essential for correct decoding). In this section we discuss methods of predictive pre-coding (to exploit correlation in local regions of the coded frame) followed by two widely-used entropy coding techniques, 'modified Huffman' variable length codes and arithmetic coding.

3.5.1 Predictive Coding

Certain symbols are highly correlated in local regions of the picture. For example, the average or DC value of neighbouring intra-coded blocks of pixels may be very similar; neighbouring motion vectors may have similar x and y displacements and so on. Coding efficiency may be improved by predicting elements of the current block or macroblock from previously-encoded data and encoding the difference between the prediction and the actual value.

The motion vector for a block or macroblock indicates the offset to a prediction reference in a previously-encoded frame. Vectors for neighbouring blocks or macroblocks are often correlated because object motion may extend across large regions of a frame. This is especially true for small block sizes (e.g. 4×4 block vectors, see Figure 3.22) and/or large moving objects. Compression of the motion vector field may be improved by predicting each motion vector from previously-encoded vectors. A simple prediction for the vector of the current macroblock X is the horizontally adjacent macroblock A (Figure 3.44), alternatively three or more previously-coded vectors may be used to predict the vector at macroblock X (e.g. A, B and C in Figure 3.44). The difference between the predicted and actual motion vector (Motion Vector Difference or MVD) is encoded and transmitted.

The quantisation parameter or quantiser step size controls the tradeoff between compression efficiency and image quality. In a real-time video CODEC it may be necessary to modify the quantisation within an encoded frame (for example to alter the compression ratio in order to match the coded bit rate to a transmission channel rate). It is usually

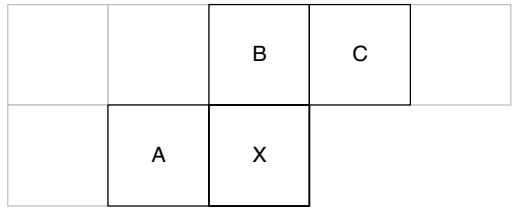

Figure 3.44 Motion vector prediction candidates

sufficient (and desirable) to change the parameter only by a small amount between successive coded macroblocks. The modified quantisation parameter must be signalled to the decoder and instead of sending a new quantisation parameter value, it may be preferable to send a delta or difference value (e.g. ± 1 or ± 2) indicating the change required. Fewer bits are required to encode a small delta value than to encode a completely new quantisation parameter.

3.5.2 Variable-length Coding

A variable-length encoder maps input symbols to a series of codewords (variable length codes or VLCs). Each symbol maps to a codeword and codewords may have varying length but must each contain an integral number of bits. Frequently-occurring symbols are represented with short VLCs whilst less common symbols are represented with long VLCs. Over a sufficiently large number of encoded symbols this leads to compression of the data.

3.5.2.1 Huffman Coding

Huffman coding assigns a VLC to each symbol based on the probability of occurrence of different symbols. According to the original scheme proposed by Huffman in 1952 [7], it is necessary to calculate the probability of occurrence of each symbol and to construct a set of variable length codewords. This process will be illustrated by two examples.

Example 1: **Huffman coding, sequence 1 motion vectors**

The motion vector difference data (MVD) for a video sequence ('sequence 1') is required to be encoded. Table 3.2 lists the probabilities of the most commonly-occurring motion vectors in the encoded sequence and their *information content*, $\log_2(1/p)$. To achieve optimum compression, each value should be represented with exactly $\log_2(1/p)$ bits. '0' is the most common value and the probability drops for larger motion vectors (this distribution is representative of a sequence containing moderate motion).

Table 3.2 Probability of occurrence of motion vectors in sequence 1

Vector	Probability p	log2$(1/p)$
−2	0.1	3.32
−1	0.2	2.32
0	0.4	1.32
1	0.2	2.32
2	0.1	3.32

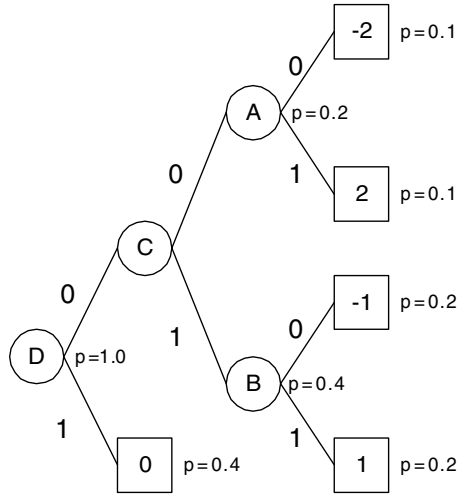

Figure 3.45 Generating the Huffman code tree: sequence 1 motion vectors

1. Generating the Huffman Code Tree

To generate a Huffman code table for this set of data, the following iterative procedure is carried out:

1. Order the list of data in increasing order of probability.
2. Combine the two lowest-probability data items into a 'node' and assign the joint probability of the data items to this node.
3. Re-order the remaining data items and node(s) in increasing order of probability and repeat step 2.

The procedure is repeated until there is a single 'root' node that contains all other nodes and data items listed 'beneath' it. This procedure is illustrated in Figure 3.45.

Original list:	The data items are shown as square boxes. Vectors (−2) and (+2) have the lowest probability and these are the first candidates for merging to form node 'A'.
Stage 1:	The newly-created node 'A' (shown as a circle) has a probability of 0.2 (from the combined probabilities of (−2) and (2)). There are now three items with probability 0.2. Choose vectors (−1) and (1) and merge to form node 'B'.
Stage 2:	A now has the lowest probability (0.2) followed by B and the vector 0; choose A and B as the next candidates for merging (to form 'C').
Stage 3:	Node C and vector (0) are merged to form 'D'.
Final tree:	The data items have all been incorporated into a binary 'tree' containing five data values and four nodes. Each data item is a 'leaf' of the tree.

2. Encoding

Each 'leaf' of the binary tree is mapped to a variable-length code. To find this code, the tree is traversed from the root node (D in this case) to the leaf (data item). For every branch, a 0 or 1

Table 3.3 Huffman codes for sequence 1 motion vectors

Vector	Code	Bits (actual)	Bits (ideal)
0	1	1	1.32
1	011	3	2.32
−1	010	3	2.32
2	001	3	3.32
−2	000	3	3.32

Table 3.4 Probability of occurrence of motion vectors in sequence 2

Vector	Probability	$\log_2(1/p)$
−2	0.02	5.64
−1	0.07	3.84
0	0.8	0.32
1	0.08	3.64
2	0.03	5.06

is appended to the code, 0 for an upper branch, 1 for a lower branch (shown in the final tree of Figure 3.45), giving the following set of codes (Table 3.3).

Encoding is achieved by transmitting the appropriate code for each data item. Note that once the tree has been generated, the codes may be stored in a look-up table.

High probability data items are assigned short codes (e.g. 1 bit for the most common vector '0'). However, the vectors (−2, 2, −1, 1) are each assigned three-bit codes (despite the fact that –1 and 1 have higher probabilities than –2 and 2). The lengths of the Huffman codes (each an integral number of bits) do not match the ideal lengths given by $\log_2(1/p)$. No code contains any other code as a prefix which means that, reading from the left-hand bit, each code is uniquely decodable.

For example, the series of vectors (1, 0, −2) would be transmitted as the binary sequence 0111000.

3. Decoding

In order to decode the data, the decoder must have a local copy of the Huffman code tree (or look-up table). This may be achieved by transmitting the look-up table itself or by sending the list of data and probabilities prior to sending the coded data. Each uniquely-decodeable code is converted back to the original data, for example:

011 is decoded as (1)

1 is decoded as (0)

000 is decoded as (−2).

Example 2: Huffman coding, sequence 2 motion vectors

Repeating the process described above for a second sequence with a different distribution of motion vector probabilities gives a different result. The probabilities are listed in Table 3.4 and note that the zero vector is much more likely to occur in this example (representative of a sequence with little movement).

Table 3.5 Huffman codes for sequence 2 motion vectors

Vector	Code	Bits (actual)	Bits (ideal)
0	1	1	0.32
1	01	2	3.64
−1	001	3	3.84
2	0001	4	5.06
−2	0000	4	5.64

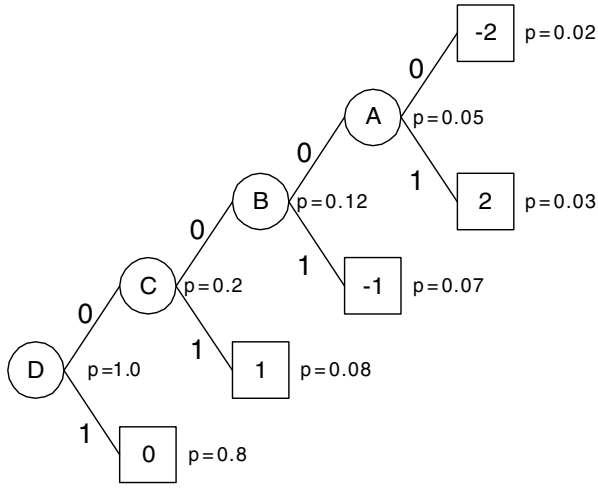

Figure 3.46 Huffman tree for sequence 2 motion vectors

The corresponding Huffman tree is given in Figure 3.46. The 'shape' of the tree has changed (because of the distribution of probabilities) and this gives a different set of Huffman codes (shown in Table 3.5). There are still four nodes in the tree, one less than the number of data items (five), as is always the case with Huffman coding.

If the probability distributions are accurate, Huffman coding provides a relatively compact representation of the original data. In these examples, the frequently occurring (0) vector is represented efficiently as a single bit. However, to achieve optimum compression, a separate code table is required for each of the two sequences because of their different probability distributions. The loss of potential compression efficiency due to the requirement for integral-length codes is very clear for vector '0' in sequence 2, since the optimum number of bits (information content) is 0.32 but the best that can be achieved with Huffman coding is 1 bit.

3.5.2.2 Pre-calculated Huffman-based Coding

The Huffman coding process has two disadvantages for a practical video CODEC. First, the decoder must use the same codeword set as the encoder. Transmitting the information contained in the probability table to the decoder would add extra overhead and reduce compression

Table 3.6 MPEG-4 Visual Transform Coefficient (TCOEF) VLCs (partial, all codes <9 bits)

Last	Run	Level	Code
0	0	1	10s
0	1	1	110s
0	2	1	1110s
0	0	2	1111s
1	0	1	0111s
0	3	1	01101s
0	4	1	01100s
0	5	1	01011s
0	0	3	010101s
0	1	2	010100s
0	6	1	010011s
0	7	1	010010s
0	8	1	010001s
0	9	1	010000s
1	1	1	001111s
1	2	1	001110s
1	3	1	001101s
1	4	1	001100s
0	0	4	0010111s
0	10	1	0010110s
0	11	1	0010101s
0	12	1	0010100s
1	5	1	0010011s
1	6	1	0010010s
1	7	1	0010001s
1	8	1	0010000s
ESCAPE			0000011s
...	...	...	...

efficiency, particularly for shorter video sequences. Second, the probability table for a large video sequence (required to generate the Huffman tree) cannot be calculated until after the video data is encoded which may introduce an unacceptable delay into the encoding process. For these reasons, recent image and video coding standards define sets of codewords based on the probability distributions of 'generic' video material. The following two examples of pre-calculated VLC tables are taken from MPEG-4 Visual (Simple Profile).

Transform Coefficients (TCOEF)

MPEG-4 Visual uses 3D coding of quantised coefficients in which each codeword represents a combination of (run, level, last). A total of 102 specific combinations of (run, level, last) have VLCs assigned to them and 26 of these codes are shown in Table 3.6.

A further 76 VLCs are defined, each up to 13 bits long. The last bit of each codeword is the sign bit 's', indicating the sign of the decoded coefficient (0 = positive, 1 = negative). Any (run, level, last) combination that is not listed in the table is coded using an escape sequence, a special ESCAPE code (0000011) followed by a 13-bit fixed length code describing the values of run, level and last.

Table 3.7 MPEG4 Motion Vector Difference (MVD) VLCs

MVD	Code
0	1
+0.5	010
−0.5	011
+1	0010
−1	0011
+1.5	00010
−1.5	00011
+2	0000110
−2	0000111
+2.5	00001010
−2.5	00001011
+3	00001000
−3	00001001
+3.5	00000110
−3.5	00000111
...	...

Some of the codes shown in Table 3.6 are represented in 'tree' form in Figure 3.47. A codeword containing a run of more than eight zeros is not valid, hence any codeword starting with 000000000... indicates an error in the bitstream (or possibly a start code, which begins with a long sequence of zeros, occurring at an unexpected position in the sequence). All other sequences of bits can be decoded as valid codes. Note that the smallest codes are allocated to short runs and small levels (e.g. code '10' represents a run of 0 and a level of ± 1), since these occur most frequently.

Motion Vector Difference (MVD)

Differentially coded motion vectors (MVD) are each encoded as a pair of VLCs, one for the x-component and one for the y-component. Part of the table of VLCs is shown in Table 3.7. A further 49 codes (8–13 bits long) are not shown here. Note that the shortest codes represent small motion vector differences (e.g. MVD = 0 is represented by a single bit code '1').

These code tables are clearly similar to 'true' Huffman codes since each symbol is assigned a unique codeword, common symbols are assigned shorter codewords and, within a table, no codeword is the prefix of any other codeword. The main differences from 'true' Huffman coding are (1) the codewords are pre-calculated based on 'generic' probability distributions and (b) in the case of TCOEF, only 102 commonly-occurring symbols have defined codewords with any other symbol encoded using a fixed-length code.

3.5.2.3 Other Variable-length Codes

As well as Huffman and Huffman-based codes, a number of other families of VLCs are of potential interest in video coding applications. One serious disadvantage of Huffman-based codes for transmission of coded data is that they are sensitive to transmission errors. An

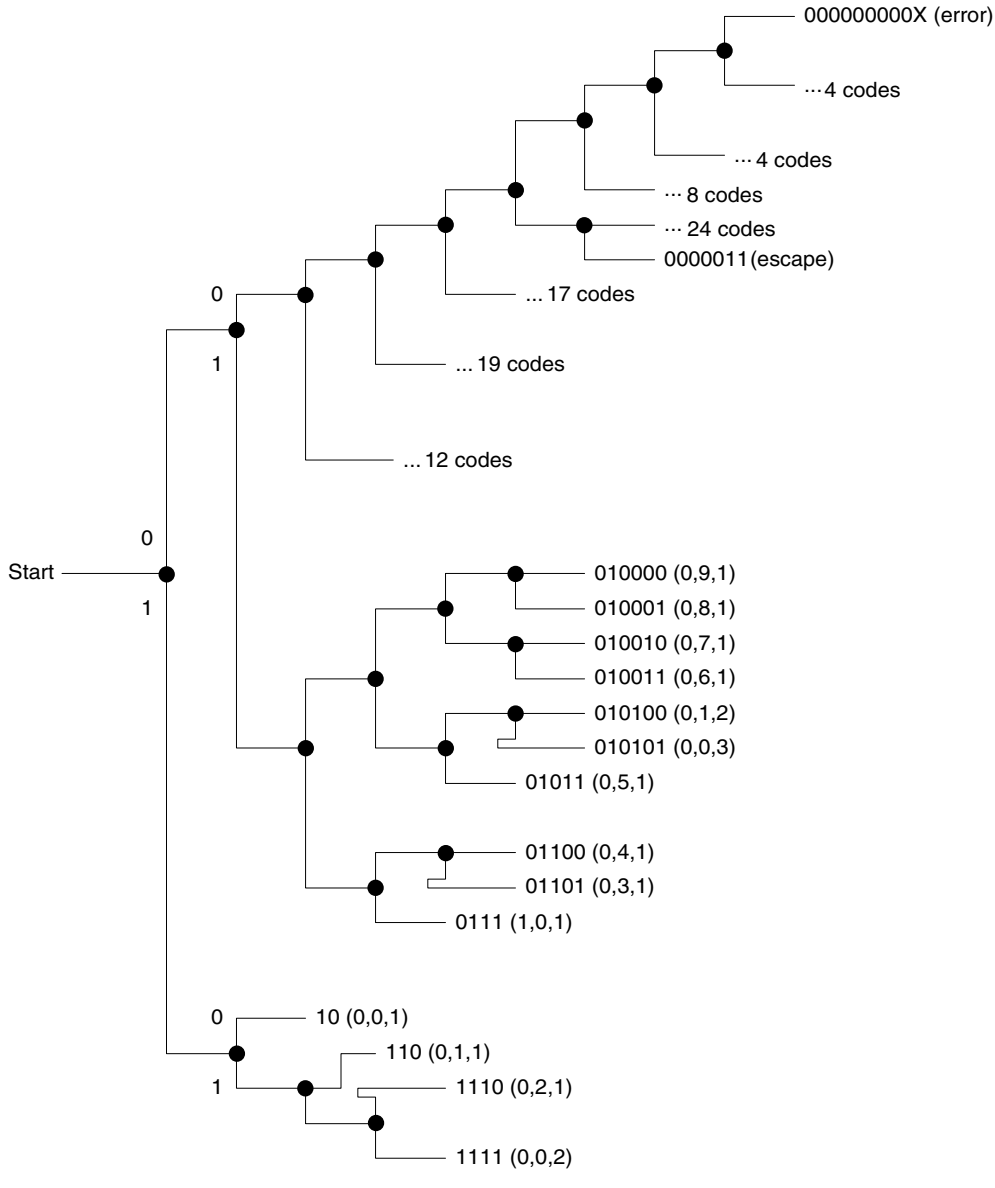

Figure 3.47 MPEG4 TCOEF VLCs (partial)

error in a sequence of VLCs may cause a decoder to lose synchronisation and fail to decode subsequent codes correctly, leading to spreading or propagation of an error in a decoded sequence. Reversible VLCs (RVLCs) that can be successfully decoded in either a forward or a backward direction can improve decoding performance when errors occur (see Section 5.3). A drawback of pre-defined code tables (such as Table 3.6 and Table 3.7) is that both encoder and decoder must store the table in some form. An alternative approach is to use codes that

can be generated automatically ('on the fly') if the input symbol is known. Exponential Golomb codes (Exp-Golomb) fall into this category and are described in Chapter 6.

3.5.3 Arithmetic Coding

The variable length coding schemes described in Section 3.5.2 share the fundamental disadvantage that assigning a codeword containing an integral number of bits to each symbol is sub-optimal, since the optimal number of bits for a symbol depends on the information content and is usually a fractional number. Compression efficiency of variable length codes is particularly poor for symbols with probabilities greater than 0.5 as the best that can be achieved is to represent these symbols with a single-bit code.

Arithmetic coding provides a practical alternative to Huffman coding that can more closely approach theoretical maximum compression ratios [8]. An arithmetic encoder converts a sequence of data symbols into a single fractional number and can approach the optimal fractional number of bits required to represent each symbol.

Example

Table 3.8 lists the five motion vector values (−2, −1, 0, 1, 2) and their probabilities from Example 1 in Section 3.5.2.1. Each vector is assigned a *sub-range* within the range 0.0 to 1.0, depending on its probability of occurrence. In this example, (−2) has a probability of 0.1 and is given the subrange 0–0.1 (i.e. the first 10% of the total range 0 to 1.0). (−1) has a probability of 0.2 and is given the next 20% of the total range, i.e. the subrange 0.1−0.3. After assigning a sub-range to each vector, the total range 0–1.0 has been divided amongst the data symbols (the vectors) according to their probabilities (Figure 3.48).

Table 3.8 Motion vectors, sequence 1: probabilities and sub-ranges

Vector	Probability	$\log_2(1/P)$	Sub-range
−2	0.1	3.32	0–0.1
−1	0.2	2.32	0.1–0.3
0	0.4	1.32	0.3–0.7
1	0.2	2.32	0.7–0.9
2	0.1	3.32	0.9–1.0

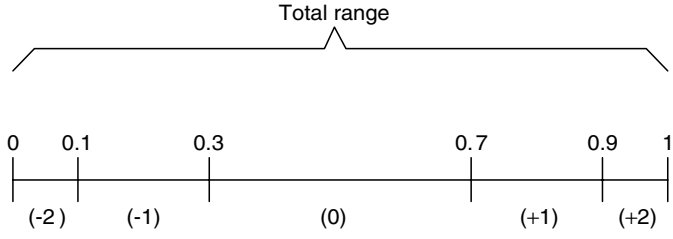

Figure 3.48 Sub-range example

Encoding Procedure for Vector Sequence (0, −1, 0, 2):

Encoding procedure	Range (L → H)	Symbol (L → H)	Sub-range	Notes
1. Set the initial range.	0 → 1.0			
2. For the first data symbol, find the corresponding sub-range (Low to High).		(0)	0.3 → 0.7	
3. Set the new range (1) to this sub-range.	0.3 → 0.7			
4. For the next data symbol, find the sub-range L to H.		(−1)	0.1 → 0.3	This is the sub-range within the interval 0–1
5. Set the new range (2) to this sub-range within the previous range.	0.34 → 0.42			0.34 is 10% of the range 0.42 is 30% of the range
6. Find the next sub-range.		(0)	0.3 → 0.7	
7. Set the new range (3) within the previous range.	0.364→0.396			0.364 is 30% of the range; 0.396 is 70% of the range
8. Find the next sub-range.		(2)	0.9 → 1.0	
9. Set the new range (4) within the previous range.	0.3928→0.396			0.3928 is 90% of the range; 0.396 is 100% of the range

Each time a symbol is encoded, the range (L to H) becomes progressively smaller. At the end of the encoding process (four steps in this example), we are left with a final range (L to H). The entire sequence of data symbols can be represented by transmitting any fractional number that lies within this final range. In the example above, we could send any number in the range 0.3928 to 0.396: for example, 0.394. Figure 3.49 shows how the initial range (0 to 1) is progressively partitioned into smaller ranges as each data symbol is processed. After encoding the first symbol (vector 0), the new range is (0.3, 0.7). The next symbol (vector –1) selects the sub-range (0.34, 0.42) which becomes the new range, and so on. The final symbol (vector +2) selects the sub-range (0.3928, 0.396) and the number 0.394 (falling within this range) is transmitted. 0.394 can be represented as a fixed-point fractional number using nine bits, so our data sequence (0, −1, 0, 2) is compressed to a nine-bit quantity.

Decoding Procedure

Decoding procedure	Range	Sub-range	Decoded symbol
1. Set the initial range.	0 → 1		
2. Find the sub-range in which the received number falls. This indicates the first data symbol.		0.3 → 0.7	(0)

(*cont.*)

Decoding procedure	Range	Sub-range	Decoded symbol
3. Set the new range (1) to this sub-range.	0.3 → 0.7		
4. Find the sub-range of *the new range* in which the received number falls. This indicates the second data symbol.		0.34 → 0.42	(−1)
5. Set the new range (2) to this sub-range within the previous range.	0.34 → 0.42		
6. Find the sub-range in which the received number falls and decode the third data symbol.		0.364 → 0.396	(0)
7. Set the new range (3) to this sub-range within the previous range.	0.364 → 0.396		
8. Find the sub-range in which the received number falls and decode the fourth data symbol.		0.3928→ 0.396	

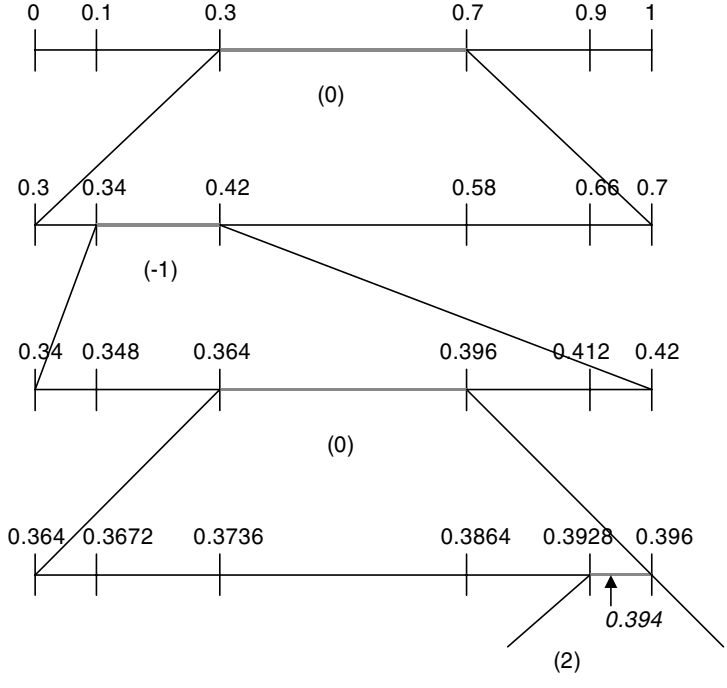

Figure 3.49 Arithmetic coding example

The principal advantage of arithmetic coding is that the transmitted number (0.394 in this case, which may be represented as a fixed-point number with sufficient accuracy using nine bits) is not constrained to an integral number of bits for each transmitted data symbol. To achieve optimal compression, the sequence of data symbols should be represented with:

$$\log_2(1/P_0) + \log_2(1/P_{-1}) + \log_2(1/P_0) + \log_2(1/P_2)\text{bits} = 8.28\text{bits}$$

In this example, arithmetic coding achieves nine bits, which is close to optimum. A scheme using an integral number of bits for each data symbol (such as Huffman coding) is unlikely to come so close to the optimum number of bits and, in general, arithmetic coding can out-perform Huffman coding.

3.5.3.1 Context-based Arithmetic Coding

Successful entropy coding depends on accurate models of symbol probability. Context-based Arithmetic Encoding (CAE) uses local spatial and/or temporal characteristics to estimate the probability of a symbol to be encoded. CAE is used in the JBIG standard for bi-level image compression [9] and has been adopted for coding binary shape 'masks' in MPEG-4 Visual (see Chapter 5) and entropy coding in the Main Profile of H.264 (see Chapter 6).

3.6 THE HYBRID DPCM/DCT VIDEO CODEC MODEL

The major video coding standards released since the early 1990s have been based on the same generic design (or model) of a video CODEC that incorporates a motion estimation and compensation front end (sometimes described as DPCM), a transform stage and an entropy encoder. The model is often described as a hybrid DPCM/DCT CODEC. Any CODEC that is compatible with H.261, H.263, MPEG-1, MPEG-2, MPEG-4 Visual and H.264 has to implement a similar set of basic coding and decoding functions (although there are many differences of detail between the standards and between implementations).

Figure 3.50 and Figure 3.51 show a generic DPCM/DCT hybrid encoder and decoder. In the encoder, video frame n (F_n) is processed to produce a coded (compressed) bitstream and in the decoder, the compressed bitstream (shown at the right of the figure) is decoded to produce a reconstructed video frame F'_n. not usually identical to the source frame. The figures have been deliberately drawn to highlight the common elements within encoder and decoder. Most of the functions of the decoder are actually contained within the encoder (the reason for this will be explained below).

Encoder Data Flow

There are two main data flow paths in the encoder, left to right (encoding) and right to left (reconstruction). The encoding flow is as follows:

1. An input video frame F_n is presented for encoding and is processed in units of a macroblock (corresponding to a 16 × 16 luma region and associated chroma samples).
2. F_n is compared with a *reference* frame, for example the previous encoded frame (F'_{n-1}). A motion estimation function finds a 16 × 16 region in F'_{n-1} (or a sub-sample interpolated

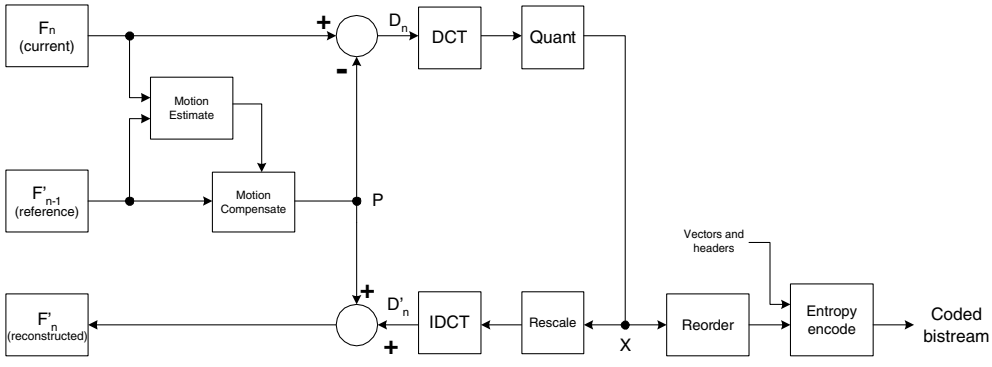

Figure 3.50 DPCM/DCT video encoder

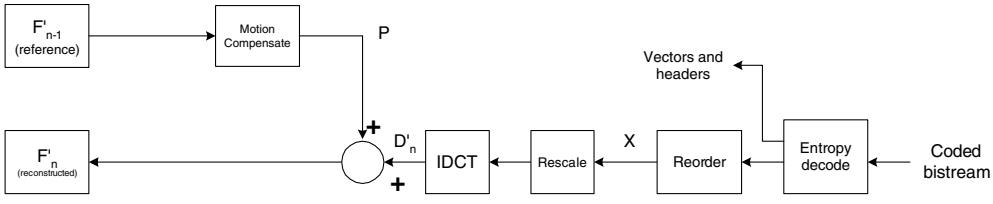

Figure 3.51 DPCM/DCT video decoder

version of F'_{n-1}) that 'matches' the current macroblock in F_n (i.e. is similar according to some matching criteria). The offset between the current macroblock position and the chosen reference region is a motion vector MV.

3. Based on the chosen motion vector MV, a motion compensated prediction P is generated (the 16×16 region selected by the motion estimator).
4. P is subtracted from the current macroblock to produce a residual or difference macroblock D.
5. D is transformed using the DCT. Typically, D is split into 8×8 or 4×4 sub-blocks and each sub-block is transformed separately.
6. Each sub-block is quantised (X).
7. The DCT coefficients of each sub-block are reordered and run-level coded.
8. Finally, the coefficients, motion vector and associated header information for each macroblock are entropy encoded to produce the compressed bitstream.

The reconstruction data flow is as follows:

1. Each quantised macroblock X is rescaled and inverse transformed to produce a decoded residual D′. Note that the nonreversible quantisation process means that D′ is not identical to D (i.e. distortion has been introduced).
2. The motion compensated prediction P is added to the residual D′ to produce a reconstructed macroblock and the reconstructed macroblocks are saved to produce reconstructed frame F'_n.

After encoding a complete frame, the reconstructed frame F'_n may be used as a reference frame for the next encoded frame F_{n+1} .

Decoder Data Flow

1. A compressed bitstream is entropy decoded to extract coefficients, motion vector and header for each macroblock.
2. Run-level coding and reordering are reversed to produce a quantised, transformed macroblock X.
3. X is rescaled and inverse transformed to produce a decoded residual D′.
4. The decoded motion vector is used to locate a 16 × 16 region in the decoder's copy of the previous (reference) frame F'_{n-1}. This region becomes the motion compensated prediction P.
5. P is added to D′ to produce a reconstructed macroblock. The reconstructed macroblocks are saved to produce decoded frame F'_n.

After a complete frame is decoded, F'_n is ready to be displayed and may also be stored as a reference frame for the next decoded frame F'_{n+1}.

It is clear from the figures and from the above explanation that the encoder includes a decoding path (rescale, IDCT, reconstruct). This is necessary to ensure that the encoder and decoder use identical reference frames F'_{n-1} for motion compensated prediction.

Example

A 25-Hz video sequence in CIF format (352 × 288 luminance samples and 176 × 144 red/blue chrominance samples per frame) is encoded and decoded using a DPCM/DCT CODEC. Figure 3.52 shows a CIF (video frame (F_n) that is to be encoded and Figure 3.53 shows the reconstructed previous frame F'_{n-1}. Note that F'_{n-1} has been encoded and decoded and shows some distortion. The difference between F_n and F'_{n-1} *without* motion compensation (Figure 3.54) clearly still contains significant energy, especially around the edges of moving areas.

Motion estimation is carried out with a 16 × 16 luma block size and half-sample accuracy, producing the set of vectors shown in Figure 3.55 (superimposed on the current frame for clarity). Many of the vectors are zero (shown as dots) which means that the best match for the current macroblock is in the same position in the reference frame. Around moving areas, the vectors tend to point in the direction *from which* blocks have moved (e.g. the man on the left is walking to the left; the vectors therefore point to the *right*, i.e. where he has come from). Some of the vectors do not appear to correspond to 'real' movement (e.g. on the surface of the table) but indicate simply that the best match is not at the same position in the reference frame. 'Noisy' vectors like these often occur in homogeneous regions of the picture, where there are no clear object features in the reference frame.

The motion-compensated reference frame (Figure 3.56) is the reference frame 'reorganized' according to the motion vectors. For example, note that the walking person (2nd left) has been moved to the left to provide a better match for the same person in the current frame and that the hand of the left-most person has been moved down to provide an improved match. Subtracting the motion compensated reference frame from the current frame gives the motion-compensated residual in Figure 3.57 in which the energy has clearly been reduced, particularly around moving areas.

Figure 3.52 Input frame F_n

Figure 3.53 Reconstructed reference frame F'_{n-1}

Figure 3.54 Residual $F_n - F'_{n-1}$ (no motion compensation)

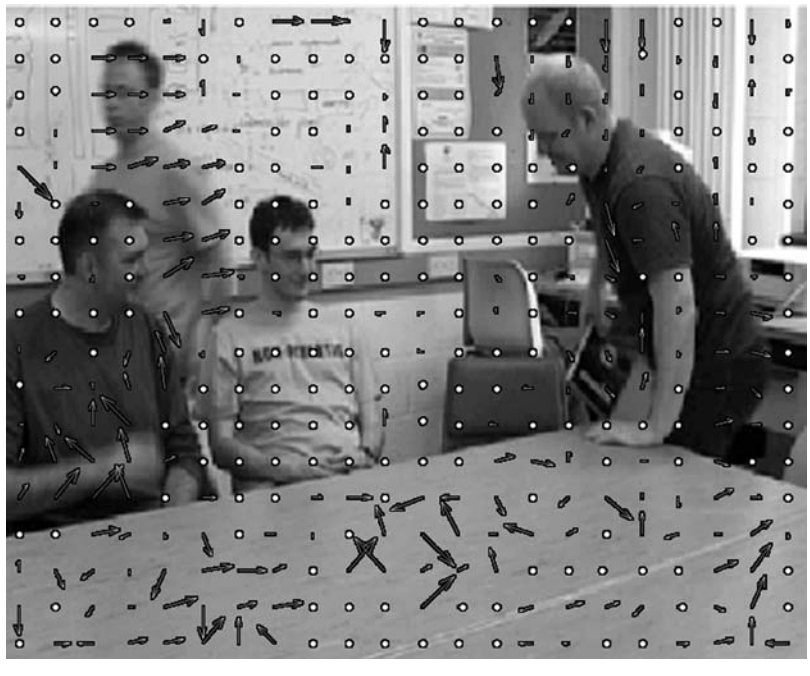

Figure 3.55 16 × 16 motion vectors (superimposed on frame)

Figure 3.56 Motion compensated reference frame

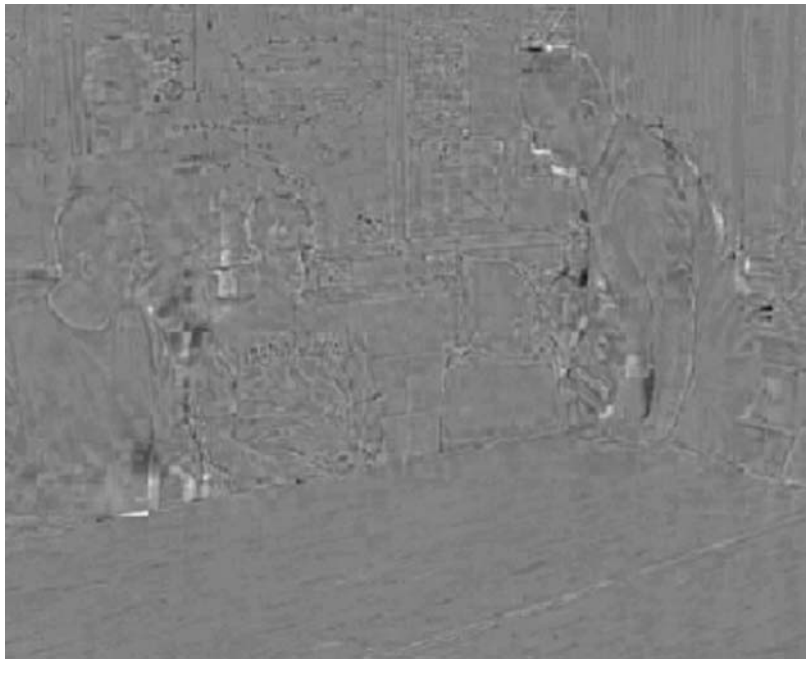

Figure 3.57 Motion compensated residual frame

Table 3.9 Residual luminance samples (upper-right 8 × 8 block)

−4	−4	−1	0	1	1	0	−2
1	2	3	2	−1	−3	−6	−3
6	6	4	−4	−9	−5	−6	−5
10	8	−1	−4	−6	−1	2	4
7	9	−5	−9	−3	0	8	13
0	3	−9	−12	−8	−9	−4	1
−1	4	−9	−13	−8	−16	−18	−13
14	13	−1	−6	3	−5	−12	−7

Figure 3.58 Original macroblock (luminance)

Figure 3.58 shows a macroblock from the original frame (taken from around the head of the figure on the right) and Figure 3.59 the luminance residual after motion compensation. Applying a 2D DCT to the top-right 8 × 8 block of luminance samples (Table 3.9) produces the DCT coefficients listed in Table 3.10. The magnitude of each coefficient is plotted in Figure 3.60; note that the larger coefficients are clustered around the top-left (DC) coefficient.

A simple forward quantiser is applied:

$$Qcoeff = round(coeff/Qstep)$$

where *Qstep* is the quantiser step size, 12 in this example. Small-valued coefficients become zero in the quantised block (Table 3.11) and the nonzero outputs are clustered around the top-left (DC) coefficient.

The quantised block is reordered in a zigzag scan (starting at the top-left) to produce a linear array:

$$-1, 2, 1, -1, -1, 2, 0, -1, 1, -1, 2, -1, -1, 0, 0, -1, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 1, 0, \ldots$$

Table 3.10 DCT coefficients

−13.50	20.47	20.20	2.14	−0.50	−10.48	−3.50	−0.62
10.93	−11.58	−10.29	−5.17	−2.96	10.44	4.96	−1.26
−8.75	9.22	−17.19	2.26	3.83	−2.45	1.77	1.89
−7.10	−17.54	1.24	−0.91	0.47	−0.37	−3.55	0.88
19.00	−7.20	4.08	5.31	0.50	0.18	−0.61	0.40
−13.06	3.12	−2.04	−0.17	−1.19	1.57	−0.08	−0.51
1.73	−0.69	1.77	0.78	−1.86	1.47	1.19	0.42
−1.99	−0.05	1.24	−0.48	−1.86	−1.17	−0.21	0.92

Table 3.11 Quantized coefficients

−1	2	2	0	0	−1	0	0
1	−1	−1	0	0	1	0	0
−1	1	−1	0	0	0	0	0
−1	−1	0	0	0	0	0	0
2	−1	0	0	0	0	0	0
−1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

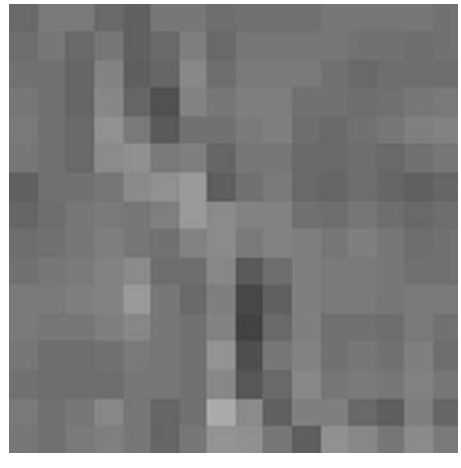

Figure 3.59 Residual macroblock (luminance)

This array is processed to produce a series of (zero run, level) pairs:

(0, −1)(0, 2)(0, 1)(0, −1)(0, −1)(0, 2)(1, −1)(0, 1)(0, −1)(0, 2)(0, −1)(0, −1)(2, −1)
(3, −1)(0, −1)(5, 1)(EOB)

'EOB' (End Of Block) indicates that the remainder of the coefficients are zero.

Each (run, level) pair is encoded as a VLC. Using the MPEG-4 Visual TCOEF table (Table 3.6), the VLCs shown in Table 3.12 are produced.

Table 3.12 Variable length coding example

Run, Level, Last	VLC (including sign bit)
(0,−1, 0)	101
(0,2, 0)	11100
(0,1, 0)	100
(0,−1, 0)	101
(0,−1, 0)	101
(0,2, 0)	11100
(1,−1, 0)	1101
...	...
(5,1, 1)	00100110

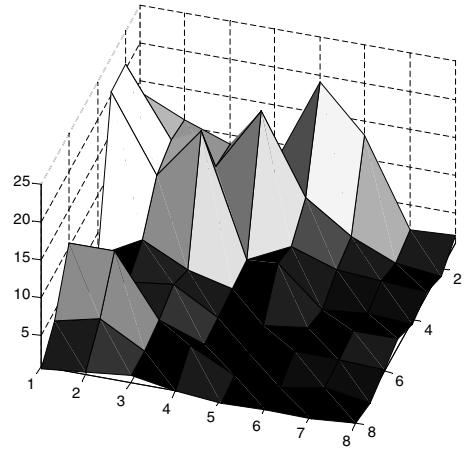

Figure 3.60 DCT coefficient magnitudes (top-right 8 × 8 block)

The final VLC signals that LAST = 1, indicating that this is the end of the block. The motion vector for this macroblock is (0, 1) (i.e. the vector points downwards). The predicted vector (based on neighbouring macroblocks) is (0,0) and so the motion vector difference values are MVDx = 0, MVDy = +1. Using the MPEG4 MVD table (Table 3.7), these are coded as (1) and (0010) respectively.

The macroblock is transmitted as a series of VLCs, including a macroblock header, motion vector difference (X and Y) and transform coefficients (TCOEF) for each 8 × 8 block.

At the decoder, the VLC sequence is decoded to extract header parameters, MVDx and MVDy and (run,level) pairs for each block. The 64-element array of reordered coefficients is reconstructed by inserting (run) zeros before every (level). The array is then ordered to produce an 8 × 8 block (identical to Table 3.11). The quantised coefficients are rescaled using:

$$Rcoeff = Qstep.Qcoeff$$

(where $Qstep = 12$ as before) to produce the block of coefficients shown in Table 3.13. Note that the block is significantly different from the original DCT coefficients (Table 3.10) due to

Table 3.13 Rescaled coefficients

−12	24	24	0	0	−12	0	0
12	−12	−12	0	0	12	0	0
−12	12	−12	0	0	0	0	0
−12	−12	0	0	0	0	0	0
24	−12	0	0	0	0	0	0
−12	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Table 3.14 Decoded residual luminance samples

−3	−3	−1	1	−1	−1	−1	−3
5	3	2	0	−3	−4	−5	−6
9	6	1	−3	−5	−6	−5	−4
9	8	1	−4	−1	1	4	10
7	8	−1	−6	−1	2	5	14
2	3	−8	−15	−11	−11	−11	−2
2	5	−7	−17	−13	−16	−20	−11
12	16	3	−6	−1	−6	−11	−3

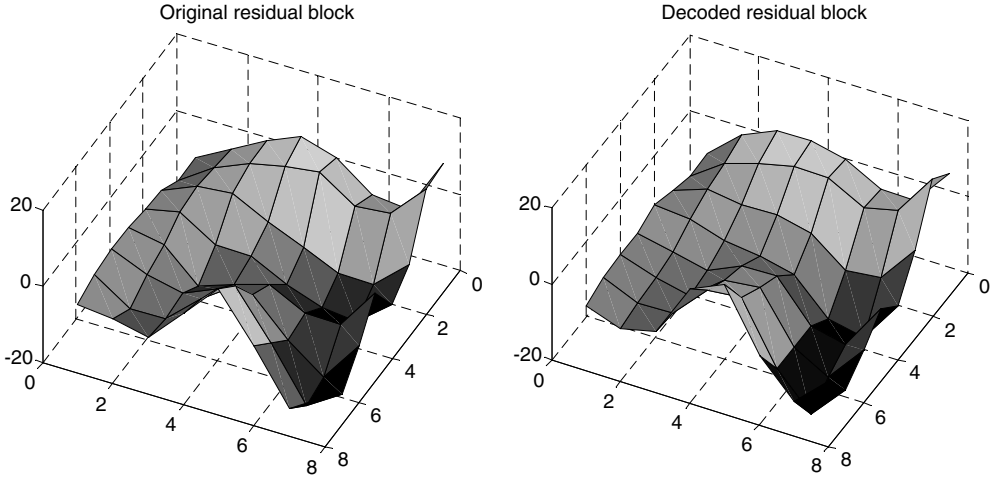

Figure 3.61 Comparison of original and decoded residual blocks

the quantisation process. An Inverse DCT is applied to create a decoded residual block (Table 3.14) which is similar but not identical to the original residual block (Table 3.9). The original and decoded residual blocks are plotted side by side in Figure 3.61 and it is clear that the decoded block has less high-frequency variation because of the loss of high-frequency DCT coefficients through quantisation.

Figure 3.62 Decoded frame F'_n

The decoder forms its own predicted motion vector based on previously decoded vectors and recreates the original motion vector (0, 1). Using this vector, together with its own copy of the previously decoded frame F'_{n-1}, the decoder reconstructs the macroblock. The complete decoded frame is shown in Figure 3.62. Because of the quantisation process, some distortion has been introduced, for example around detailed areas such as the faces and the equations on the whiteboard and there are some obvious edges along 8×8 block boundaries. The complete sequence was compressed by around 300 times (i.e. the coded sequence occupies less than 1/300 the size of the uncompressed video) and so significant compression was achieved at the expense of relatively poor image quality.

3.7 CONCLUSIONS

The video coding tools described in this chapter, motion compensated prediction, transform coding, quantisation and entropy coding, form the basis of the reliable and effective coding model that has dominated the field of video compression for over 10 years. This coding model is at the heart of the two standards described in this book. The technical details of the standards are dealt with in Chapters 5 and 6 but first Chapter 4 introduces the standards themselves.

3.8 REFERENCES

1. ISO/IEC 14495-1:2000 Information technology – lossless and near-lossless compression of continuous-tone still images: Baseline, (JPEG-LS).
2. B. Horn and B. G. Schunk, Determining optical flow, *Artificial Intelligence*, **17**, 185–203, 1981.
3. K. R. Rao and P. Yip, *Discrete Cosine Transform*, Academic Press, 1990.
4. S. Mallat, *A Wavelet Tour of Signal Processing*, Academic Press, 1999.
5. N. Nasrabadi and R. King, Image coding using vector quantisation: a review, *IEEE Trans. Commun*, **36** (8), August 1988.
6. W. A. Pearlman, Trends of tree-based, set-partitioned compression techniques in still and moving image systems, *Proc. International Picture Coding Symposium*, Seoul, April 2001.
7. D. Huffman, A method for the construction of minimum redundancy codes, *Proc. of the IRE*, **40**, pp. 1098–1101, 1952.
8. I. Witten, R. Neal and J. Cleary, Arithmetic coding for data compression, *Communications of the ACM*, **30** (6), June 1987.
9. ITU-T Recommendation, Information technology – coded representation of picture and audio information – progressive bi-level image compression, T.82 (JBIG).