**Chapter 2. Image Compression Techniques**

1. **Introduction**

The aim of this chapter is to introduce the techniques used in image compression. Although some of these techniques are well known, it is important to consider how they are used in image compression. This chapter concentrates on the practical problems of using these techniques in image compression and suggests improvements that can be made to them, when they are adopted to be used on images.

There are three basic parts to an image compressor, as shown in figure 2.1, and these are:

1. The Transform. This is a spatial transform applied to the pixel intensities of the image. It decorrelates the image data and makes it easier to compress the image.
2. Quantisation. This reduces the range of values that the transform coefficients can take, hence reducing the amount of information that is needed to describe them. Although this compresses the coefficients, it also has the unfortunate effect of introducing errors into the data.
3. Lossless Coding. This increases the compression of the quantised coefficients by reducing their statistical and spacial redundancy. These techniques are usually error sensitive since they use variable length coding methods.



Figure 2-1 Block Diagram of the general image compression steps.

In this chapter each part of the image compression method will be discussed in detail, explaining how existing techniques can be used in image compression.

1. **Transforms**

The first stage in an image compressor is the transform. This transforms the image data into a domain where it is easier to compress. A transform operates on an image’s pixel intensities and converts them into a set of transform coefficients. Natural images (which are the most common images to be compressed) have a lot of spatial correlation between pixel intensities, and these correlations can be exploited by the transform. This is achieved by mapping similar large scale changes in the data onto single transform coefficients. This type of mapping causes the transformed image to become highly decorrelated and standard compression techniques can then be used to further compress the transform coefficients.

The general form of a spatial intensity transform used on the image data is shown below (equation 2.1) :

 (2.1)

where  is the transform coefficient,

 is the image pixel intensity array,

and  is the transform function.

This equation shows that the transform coefficients are the sum of the effects of the transform on the pixel intensities, over the whole section of the image to be transformed. This definition of a transform does not specify, how it is to be applied to different images, that could vary in pixel size (they could be as large as 1024x800 pixels in size, or larger). The application of the transform to different images will be dealt with in the next section.

1. **Applications of Transforms to Images**

The transform is rarely applied to the whole of the image. As the area of the image that the transform is applied to increases, the number of calculations required to implement it generally increases proportionally to the *Area3/2* (as is found from a standard DCT without fast implementations). This suggests that to keep the number of calculations small (and manageable), the area that the transform is applied to should be as small as possible. However, the decorrelation effects on the transform improve, when a larger area of the image is considered, and this in turn improves the compression performance.

In a real system a compromise is established between the compression and the speed of the transform. The effects of decorrelation are not linearly proportional to the area used so it is not possible to theoretically determine the best area to apply the transform to, it has to be done using practical results.

The image is broken into a patchwork of blocks and the transform is applied to each block separately. Each block then has a set of transform coefficients, that describe it. Although it has been stated that images are highly correlated, this is only true over local areas of the image. There may be little or no correlation between distant sections (~100 pixels) of the image. Applying the transform to image blocks exploits the local similarity of the image without losing the benefits of decorrelation in the transform coefficients.

Transforming image blocks also introduces a blocking artefact effect, which can be a major problem. Since the coefficients that describe one block are not related to those describing the surrounding blocks, it is possible for discontinuities to occur along the block edge of compressed images. Blocking artefact is only visible at higher compression rates, in most systems, but can severely reduce the visual quality of a compressor, even if the rate distortion performance is still acceptable.

The blocks that the image is broken into do not have to be a fixed size or shape, but they are generally non-overlapping. Work has been done on overlapping blocks, such as “Lapped Orthogonal Transforms” [7] but this is not considered in this work. Overlapping blocks duplicate information, but reduces the blocking artefact and error. Whether this has a sufficient rate distortion pay off has yet to be seen.

1. **Lossless Transforms**

Transforms are generally formed in pairs so that the image can be reconstructed by applying the inverse transform. If the transforms are applied to an image without compression, then there are two possibilities, as shown in figure 2-2:

1. The transform is perfectly reconstructing and there is no error in the recovered image. This type of transform is often quite complex, since no data is lost and it can take a long time to compute.
2. The transform is “lossy” and information is lost on either the forward or inverse transform stage. These transforms can be useful, since they are very easy to calculate and often represent the image using a low number of transform coefficients. They produce a moderate compression without any further processing.



Figure 2-2 Block Diagram of transform loop.

If the image is passed through the transform loop (figure 2-2) multiple times, then theoretically either lossless or lossy transforms should produce the same recovered error each time. However since lossy transforms often favour a variable block size method, it is not always the case that reapplying a codec to a recovered image (coded with the same codec) will produce the same result. This means that if you code an image many times it may never converge to a fixed result. This means cascade problems are more likely to occur with lossy transforms rather than with lossless transforms. This problem is cause for great concern in broadcast compression as images can be compressed many time after they are originally captured.

1. **Orthogonal Transforms**

Transforms can either be orthogonal, orthonormal or non-orthogonal [62]. Orthogonal transforms have no correlation between transform coefficients, they are decorrelated, while non orthogonal transforms result in coefficients correlated to some degree. Orthonormal transforms are orthogonal transforms, which have an inner product of 1 (equation 2.4). An orthogonal transform has several advantages over the others that are useful in image compression:

1. The coefficients of an orthogonal transform are completely independent, making them easier to compress since it is not necessary to allow for interdependencies between coefficients.
2. The MSE of a recovered image using an orthogonal transform can be directly calculated from the compressed transform coefficients, without needing to apply the inverse transform.
3. The coefficients of an orthogonal transform are straight forward to calculate from the image data.

The general form of an orthogonal transform is shown in equations 2.2 and 2.3. The defining property of the transform is shown in equation 2.4. In equation 2.4 the transform is usually normalised to make the existence condition unitary, and this is then known as an orthonormal transform. Any other transform that does not obey these set of equations is considered to be non-orthogonal.

 (2.2)

 (2.3)

 (2.4)

where,

i and j are spatial pixel indices (locations),

k and l are indices in transform space,

 is the transform coefficient,

 is the image pixel array,

 and  are the forward and reverse transforms.

1. **Gram-Schmidt Orthogonalisation**

Gram-Schmidt [62] Orthogonalisation is used to make a set of non-orthogonal functions, orthogonal with respect to each other. If there exists a set of non-orthogonal functions  that can be used to describe an image, then using this method it is possible to find a corresponding set of orthogonal functions  that describes the same image. In the resulting orthogonal functions there is no correlation between the transform coefficients. The method works by assuming that one function in the set  is unique and then all the other orthogonal functions can be derived from this assumption. It is common in image compression to have a function  that describes the average pixel intensity of an image block, or grey scale level. This grey scale level is normally used as the basis for Gram-Schmidt orthogonalisation.

The general principle of the Gram-Schmidt orthogonalisation is shown in equation 2.5 and is applied to progressively higher order values of *n* and *m*.

 (2.5)

The initial starting point has

 (2.6)

and then the next highest coefficient can be calculated.

 (2.7)

This progression continues until all of the functions in the set are orthogonised. It can be seen that the order in which the orthogonalisation occurs can change the actual functions that are produced, and this is a problem with the Gram-Schmidt orthogonal method. In image compression the functions are orthogonalised in order of importance, starting at the greyscale level. It can be seen that the functions generated by this method are not orthonormal but this can be simply remedied by making the derived orthogonal functions obey equation 2.4.

**2.2.5 Mean Squared Error Calculations for Orthogonal Transforms**

The property of the orthogonal transform shown in equation 2.8 makes it possible to show, how the MSE of the recovered image can be calculated from the compressed transform coefficients. A compressed coefficient can be at one extreme truncated completely, and at the other extreme it may only suffer a slight loss of accuracy. Equations 2.8 to 2.11 show, how the MSE is related to the transform coefficients.

 (2.8)

where is the image pixel array

is the quantised image pixel array.

and N is the number of pixels in .

Substituting equation 2.3 in equation 2.8 gives:

 (2.9)

Most of these sums are zero over because the functions are orthogonal (equation 2.4). Only when  and  are the sums non zero and so it is possible to reduce equation 2.9 to equation 2.10.

 (2.10)

where is the quantised transform coefficient

and  (2.11)

Orthogonal transforms are usually normalised such that , which means that the MSE in the recovered image is the same as the MSE in the compressed coefficients.

1. **Quantisation**

The transform stage spatially compacts the image, but does not always produce compression. Quantisation is a large subject, which is discussed in detail for example, by Gray [38]. Quantisation relevant to image coding is discussed in this section and simplified to allow a general rule for the quantisation of transform image coefficients to be developed. The quantisation stage is, where most of the image compression is achieved. Before quantisation a transform coefficient may take an infinite range of values, limited only by the accuracy of the medium it is stored in. After quantisation the transform coefficient will be represented by a number of discrete values. This could be represented by:

 (2.12)

where *q* is the quantisation function,

*c* is the transform coefficient,

and .

The actual implementation of the quantisation function can be done in several ways. The common methods are discussed in this section and the effects of quantisation are discussed in relation to image compression.

**2.3.1 Linear Quantisation**

Linear quantisation is the most basic form of quantisation. The transform coefficients are divided by a quantisation step and the result is converted to an integer, by truncation of the decimal point (equation 2.13).

 (2.13)

where *qi*is the quantisation step,

*ci* is the transform coefficient,

and cq is the integer quantised coefficient.

The transform data is limited based on the 8 bit pixel intensities of standard images. This allows a quantisation step to be chosen, which limits the number of quantised states available, hence compressing the coefficient to a desired number of bits. However, it is not possible to control compression in this way, since a real system losslessly codes the quantised coefficients and this operation is not well defined.

It can be seen that equation 2.13 is not ideal and it can be improved to a more effective form

 (2.14)

This means that the transform coefficients are mapped to the quantisation values that cause the least error as shown in figure 2-3. This effectively forms quantisation ‘bins’ , inside which all values are mapped to the same quantisation step. Bin boundaries are defined by the rounding effect of equation 2.14.



Figure 2-3. Diagram showing the mapping of transform coefficients onto quantisation values.

The choice of *q* can vary since some transform coefficients are more important than others and, as a result, a quantisation table *(q(k,l))* is usually formed, providing different *q* values for each transform coefficient. How to chose *q* is quite difficult since there is no simple relationship between the compression and the value *q* takes. It is possible to obtain some simple guide to the effects of *q* by considering equation 2.14. The transform coefficient *c* can be considered to be formed from two parts, an integer part and a rational remainder as in equation 2.15 :

 (2.15)

where *c* is a coefficient (unquantised),

*n* is an integer,

*q* is the quantisation step,

and *x* is a rational number in the range.

The quantised coefficient is

 (2.16)

and therefore substituting this in equation 2.8 for one pixel gives a Squared Error (SE)

 (2.17)

Equations 2.15-2.17 only consider one data point, but in reality there are numerous occurrences with a range of *x* values. However the range of *x* is limited to  and so it is possible to calculate the error of a known source by using equation 2.18.

 (2.18)

where *q* is the quantisation step,

*ISE* is the integral squared error,

*f(x)* is the probability density function over *x* space.

Equation 2.18 cannot be evaluated, unless *f(x)* is defined and since *f(x)* is different for every data source, it is not possible to use it as a predictor. It is possible to use it as an estimate of the SE by making *f(x)* flat. This allows the integrals to be solved, as shown in equation 2.19 :

 (2.19)

This is the squared error caused by quantising one coefficient of a transform. To convert this to the MSE caused in a image block, it is necessary to sum the effects of all the transform coefficients and divide by the size of the image block :

 (2.20)

where *N* is the number of pixels in the image block.

q is the quantisation step,

and *k*, *l* are indices to the quantisation step relating to the coefficients *c*(*k*,*l*).

In most cases the MSE caused by quantisation is less than the estimated, because the probability density function (PDF) of the data is highly correlated around *c=0,* as shown in figure 2-4, and this means that the probability density function for *f(x)* takes the form shown in figure 2-5. The function is this shape since it is describing values of x, which are remainders of the actual transfer coefficient. The net effect of *f(x)* is that the predicted MSE is a maximum limit on the error and not an exact prediction.



Figure 2-4. Stylised Diagram of PDF for a transform coefficient.



Figure 2-5. Stylised Diagram of typical *f(x)* function caused by PDF of the coefficient.

**2.3.2 Non-Uniform Quantisation**

Non-Uniform quantisation uses a more complex model to decide, what each quantisation value should be. The quantiser is implemented by defining ‘data bins’ and all transform coefficients inside a bin are quantised to the same value. The effect of such a quantiser is shown in figure 2-6.



Figure 2-6. Diagram of the ‘data bins’ in an Non-Uniform quantiser.

It can be seen that the size of the data bins in figure 2-6 changes over the range of coefficient values and this gives a higher accuracy to the approximation of the transform coefficients, where it is most appropriate.

The process of choosing the appropriate bin size is known as Lloyd-Max [21, 22] quantisation, which works by trying to minimise the error between the quantised coefficients and the original coefficients for a fixed number of quantisation values. The bins are calculated by using the probability density function (PDF) of the data as this indicates, where the ‘bin density’ should increase.

There are two methods that can be used to calculate the bin position using Lloyd-Max quantisation:

1. Obtain the true PDF from the data and derive the bins directly from this.
2. Iterate the bin positions, and slowly converge to a stable set of bins.

Since most image compression applications cannot form an accurate PDF function for the data (only an experimental histogram), the second method is often preferred.

There are several steps in Lloyd-Max quantisation:

1. Provide a set of initial ‘seed’ quantisation values, such as the values produced by linear quantisation.
2. Find the quantisation values, which produce the smallest error in the coefficient after quantisation, for each coefficient in the data set.
3. Partition the coefficients by grouping them with coefficient that used the same quantisation value.
4. Average all the coefficients in each partition to produce a new quantisation value for that partition.
5. Repeat 2-5, until the partitions become stable.
6. Find the coefficient boundary, which marks each partition and mark these as the bin edges.

Lloyd-Max quantisation produces a set of quantisation bins, which minimises the error in each quantised coefficients. Unfortunately, it does this at the expense of a reduced data entropy and as a result quantised coefficients are not substantially improved by the application of a lossless coding stage (since the quantised coefficients already have little statistical redundancy, that can be exploited by lossless coding). This can be a benefit, since lossless coding adds to computation, but it has been shown that linear quantisation with a lossless coding stage is more efficient in rate distortion terms, than Lloyd-Max quantisation [21-22].

An improvement that can be made to linear quantisation is to keep the bin boundaries fixed, but average the coefficients inside the bin to provide a better quantisation value. This improves the error performance without changing the quantised data’s statistics, and it is like applying a single pass of the Lloyd-Max method.

**2.3.3 Embedded Quantisation**

Embedded quantisation [39] is a method used to quantise coefficients so that they can be decoded to any error. This method of quantisation is common in wavelet compressors, where it is most suited.

The basis of embedded quantisation is to apply many quantisation steps to a single coefficient and produce a stream of symbols to describe that coefficient. If the stream is stopped at any point the data received should describe the coefficient as accurately as possible.

Consider a set of quantisation steps

 (2.21)

where .

These quantisation steps can be recursively applied to a particular transform coefficient to accurately describe it, as shown in equation 2.22.

 (2.22)

where  is the residual coefficient,

and .

It can be seen that iterating equation 2.22 produces two pieces of information, a quantised coefficient that can be used to refine the coefficients representation and a residual coefficient which can be quantised further. It is important to space *qi* and *qi+1* apart, so that they are different enough to produce quantised coefficients but similar enough to give a good resolution.

Generally in wavelet compression  and , so that a binary masking operation can be used for quantisation. This is very easy and fast to implement on most processors and this usually compensates for the more complex quantisation method. Another method adopted by wavelet compressors is to only pass the sign of the quantised value once. In the above scheme this means that the absolute value of the coefficient is used in equation 2.22 and the sign is only transmitted once for each coefficient.

**2.3.4 Quantisation Scalability**

Although there are three types of quantisers, only the linear quantiser will be used in this work, because it is easy it implement, efficient after lossless coding and more importantly it is scaleable. When the quantiser is applied to image blocks of different sizes, it is important that the quantiser has the same effect. Since the compression caused by the lossless coder is not well defined, the error alone must be used to ensure that the quantisation is scaled correctly.

If the MSE caused by quantisation is to be constant at different block sizes, it follows from equation 2.20 that

 (2.23)

where  is a constant,

*q* is the quantisation step

and N is the number of pixels in the block.

This is an upper-limit on the error, since it is calculated for the worst case and so the quantisation is generally scaled as shown in equation 2.24.

 (2.24)

where *q* is the quantisation step,

 is the quantisation factor, which can be derived by experiment.

**2.4 Lossless Coding**

Lossless coding aims to reduce the redundancy of a set of data, by exploiting its statistics. Theoretically this coding method should compress a data source without introducing any new errors into the data. There are three methods that can be used for lossless coding:

1. Entropy Coding. It is possible to reduce the information required to store data to a theoretical minimum, by exploiting the ‘blind’ statistics of the data, without considering the order in which it is received. This is usually achieved with a Huffman coder[8], but arithmetic coders[9] can also be useful.
2. Pre-processing. The data can be processed before being compressed by an entropy coder to improve the overall compression.
3. High Order Modelling. This exploits the interrelation between data, and assumes that there is some correlation between consecutive data symbols.

In lossless coding it is useful to refer to the input as data symbols to be compressed and the output from the lossless coder as compressed symbols. The data symbols are usually quantised transform coefficients in image compression, but they can be anything, provided the coder has a knowledge of their statistics.

1. **Entropy Coding**

This is the most effective method of lossless coding and is nearly always present in image compression. Entropy is the average minimum number of bits that a data symbol stream can be compressed into, when each symbol is considered in isolation, based on its statistics. It can be calculated directly (equation 2.25), but it is sometimes not possible to reach the theoretical minimum due to the implementation of the entropy coder.

 (2.25)

where *PDD(x)* is the probability density distribution of symbol *x.*

There are two different approaches to entropy coding, Huffman coding [8] and arithmetic coding [9]. Huffman compression a more common and a more robust method, but cannot compress data to less than one bit/symbol. Arithmetic coding is less controllable and does not compress well at higher bits/symbol, but it can reduce its entropy below one bit/symbol. For this reason arithmetic coders are not often used, unless the average entropy of the source is expected to drop below one bit / symbol.

**2.4.1.1 Huffman Coding**

Huffman compression [8] is designed to reduce the entropy of a data source close to the theoretical minimum described in equation 2.25. The Huffman coder does this by representing common data symbols with short compressed symbols and rare data symbols with long compressed symbols. The average effect of this method is to reduce the redundancy of each compressed symbol to a minimum.

A Huffman coder determines the compressed symbols by forming a data tree from the original data symbols and their associated probabilities. The tree is formed by applying the following rules until it is complete.

1. Link the two unlinked data symbols with the lowest probability to form a new data symbol with probability equal to the sum of the previous two symbols.
2. Continue to link the active data symbols with the smallest probability until a complete data tree is formed.

Consider the example shown in table 2-1. There are four symbols, which can be described by the binary data symbols shown, and have an average bits/symbol coding rate of 2. If the four symbols are Huffman coded, then they produce the data tree shown in figure 2-7 and the compressed binary symbols shown in table 2-1. The average bits/symbol for the compressed binary symbols is 1.3, which is a lossless compression of 1.5:1 .

Symbol Probability Binary Binary Compressed

Symbol Symbol

A 0.07 00 111

B 0.03 01 110

C 0.1 10 10

D 0.8 11 0

Table 2-1. Table showing four symbols compressed using a Huffman coder.



Figure 2-7. Diagram of data tree form from data in table 2-1. Symbols E, F and G are temporary node used by the Huffman coder.

The example shown in table 2-1 shows the major failing of the Huffman coder. The entropy of the source given is 1.0, but the Huffman coder is only able to reduce the bit/symbol coding rate to 1.3. If the entropy of the data source is often close to or below 1 bit/symbol, then the Huffman coder does not perform optimally, and needs to be improved. Pre-processing can increase the entropy of the data symbols, but reduce the number of compressed symbols necessary to describe the data. This allows the Huffman coder to be able to cope, and produces an average compression less than one bit/symbol.

**2.4.1.2 Arithmetic Coding**

Arithmetic coding [9] works by treating a stream of data symbols as a whole and does not replace individual data symbols with compressed versions. The coder is always implemented in binary and to avoid confusion it will be explained by reference to binary numbers.

An arithmetic coder takes an upper and lower limit, and defines a *range* between these *upper* and *lower* limits to be equivalent to a symbol with the probability of 1.0 . Symbols are encoded by modifying the *range* of the arithmetic coder and sending symbols to reconstruct this range information at the decoder. The operation of an arithmetic coder can be demonstrated by using the data previously used in table 2-1. The data can be represented as probabilities in the arithmetic coder as shown in figure 2-8(b) . It can be seen that the symbol probabilities stack to form a continuous range of probabilities between 0.0 and 1.0. This give a range of probabilities that represent each symbol.



Figure 2-8. Diagrams showing the limits of an arithmetic coder.

The *upper* limit of the arithmetic coder is initially set to a value which corresponds to a probability of 1.0 but with infinite precision. This can be represented in binary as an infinite number of bits set to 1, since there is no fixed position for the decimal point. The *lower* limit is set to zero with infinite precision, again using an infinite number of binary bits set to 0. The only problem with this assumption is that computers work with fixed length numbers, commonly 4 to 8 bytes long, so it is impossible to represent the infinite precision. To over come this the limits are set as large as possible and their infinite length is simulated, by the coding algorithm.

The operation of encoding a symbol by the coder requires the *range* to be reduced in the following way:







where  is the higher probability range of the symbol,

and  is the lower probability range of the symbol.

This can be demonstrated by encoding the symbols DDCABB as shown in figure 2-9. The *range* of the binary limits is constantly modified. It can be shown that when the most significant bits (MSB’s) of the limits become equal they are not changed again by further operations and can be removed for transmission as the encoded data. When the MSB is removed for transmission the whole limit is shifted left by one. A binary 1 is then added to the *upper* limit of the coder and a binary 0 is added to the *lower* limit of the coder, to simulate the infinite precession of the limits .



Figure 2-9. Diagrams showing the operation of an arithmetic coder.

An arithmetic coder only transmits bits when the MSB bits of the limits are the same, and this is not true for every symbol.A symbol that is quite common has a high probability, the *range* will not be greatly affected, and it is less likely that bits will be transmitted. A rare symbol has a low probability, causes a drastic change in the *range* (which requires a lot of information to encode), and will usually cause bits to be transmitted.

The method described above works, but it can reach a state where the coder is effectively ‘stuck’. The upper and lower limits can come so close together that the range can no longer be altered and no bits can be shifted out to increase the precision. This problem is known as underflow and is demonstrated in figure 2-10(a). The *upper* and *lower* limits are different by 1 and no bits can be shifted out.



Figure 2-10. Diagram of the under flow condition

There is a simple solution to underflow, and this is to prevent to coder from ever coming close to this state, by constantly checking for underflow and correcting it. Each time the coder tests to see if the MSB needs to be shifted out and transmitted, underflow is checked for at the same time. If no MSB are shifted out then the second MSB needs to be checked. If the second MSB of the *upper* limit is 0 and the second MSB of the *lower* limit is 1 then there is a danger of under flow as shown in figure 2-10(b). To remove the danger of underflow the second MSB is removed and the other bits are shifted left to fill the gap, as shown in figure 2-10(c). When the next MSB is transmitted, any underflow bits that have been accumulated are also transmitted at the same time.

1. **Pre-processing**

There are three basic methods of pre-processing for data source compression:

1. Delta coding. This removes linear correlation in a data source from symbol to symbol.
2. Combining data sources. This removes correlation between similar data sources.
3. Run length coding. This also removes linear correlation in the data source, but is more suitable, when large runs of the same symbol occur in the data stream.

If correlation does not exist in the data sources then applying pre-processing can increase the information required to store the data, reducing the efficiency of the system.

**2.4.2.1 Delta Coding**

This method is the simplest form of pre-processing and it is described by equation 2.26 :

 (2.26)

where is the data symbol at i,

and is the data coded symbol.

The original data item in the source is considered to be unchanged since no previous symbol exists. It is common to split the data source into sections, which are highly correlated. In image compression this usually means taking separate ‘lines’ of coefficient symbols out of the image.

The effect of delta coding is to give a set of source symbols that have a greater dynamic range , but the entropy of the data source is reduced by the process. In 8 bit greyscale images the intensity can vary from 0🡪 255 but after delta coding the range of the symbols can be -255 🡪 255 but very few symbols have these extreme ranges so the entropy of the encoded source is lower than the original.

**2.4.2.2 Combining Data Sources**

This method is effective when two or more data sources are correlated with respect to each other. A common example of this is a motion vector, where the x and y direction vectors are connected in some way. Data sources are combined by creating a new and larger data source that describes both the original data sources. Figure 2-11 illustrates the example of simple motion vector prediction.



Figure 2-11. Diagram of motion vector ranges for a simple motion predictor.



Figure 2-12. Probability density functions for individual x and y components (a) and combined data source(b).

The x component of the motion vector has a range and similarly for the y component. This gives two PDFs for the x and y motion components respectively, as shown in figure 2-12(a). Each of these components have *2n+1* individual data symbols, and by combining these sources to describe both motion components allows  data symbols in the new source. Although more individual symbols have been created in the new source, it is likely that the entropy will be less on average than the two individual motion components, provided a high correlation exists between them.

Another reason for combining the two original data sources is the limit imposed by Huffman coding. Since the minimum entropy for the individual sources are both 1 bit/symbol combining the two effectively reduces this limit to 0.5 bits/symbol (in the original format).

**2.4.2.3 Run Length Coding**

Run length coding is effective on data sources that have linear runs of the same symbol inside a data stream. The run length coder works by counting the number of occurrences of the same symbol and then forming a new symbol which describes the run length and the run symbol type. This is demonstrated in figure 2-13.



Figure 2-13. Diagram illustrating the action of a run length coder.

The entropy of the new symbol stream, which contains the run length codes is less than the original, providing there are sufficient runs to make the method viable. Run length coding has a similar effect to arithmetic coding without the complication of the arithmetic coder.

**2.4.3 Higher Order Modelling.**

Higher order modelling incorporates both the ideas of entropy compression of the data stream and the correlation between adjacent symbols in a data stream. It does this by using multiple probability density distrubtions (PDDs). Entropy compressors remove all redundancy from a data stream by using a PDD generated for the data, regardless of the order that it is coded in. Higher order encoders have multiple probability density functions, which are used in the context of a particular symbol or set of symbols. Given a particular symbol, the higher order encoder knows the probability of any symbol in the data source occuring after that given symbol.

The principle of a higher order encoder can be illustrated with the simple example shown in figure 2-14. There are three symbols which occur in the probability shown over a large set of the data.



Figure 2-14. Probability density Distrubtions(PDD) for a data set, showing different PDDs in context of different symbols.

The statistical PDD does not show anything about the layout of a typical data stream, but if the probability of A, B and C occurring in the context A, B or C are shown, the structure of the stream becomes clearer. It is possible to construct a typical data stream , as shown in figure 2-15.



Figure 2-15. Typical stream of data for the PDD’s shown in figure 2-14.

The multiple PDD tables can go a stage further, rather than having a PDD table to be used in context of a particular data symbol (a first order encoder) it is possible to have a PDD table for each combination of previous symbols (second order encoder or more). Although it would be ideal to have a very large context of previous symbols to work with it is not very practical computationally. The PDD tables require a lot of memory to store and this can soon become a problem. In practice a context of a single previous symbol works very effectively.

In this work higher order encoders are used in conjunction with the arithmetic coder since it can produce entropies below one bit/symbol, and sources which benefit from higher order encoders often have an entropy less than one bit/symbol.

**2.5 Summary**

The processes used in this work are primarily linear quantisation and Huffman coding, although most of the other processes described are used in one area or another. The only real development in this chapter was the understanding behind scaling the quantisation steps effectively.