Research report 128

IMAGE DATA COMPRESSION

M Todd, R Wilson

(RR128)

This report describes compression of the bandwidth required to represent digital images using a hierarchical anisotropic predictive coder. The basic coder algorithm is recursive binary nesting, which involves a recursive segmentation of the image into edge sharing quadrants. Each recursive segmentation quarters the spatial area of the image sections being considered and involves the predictive coding of the corner points of the new sections. The segmentation of a section can be halted at any level, and the remaining pixels within the section determined by an interpolation from the corner points. Because of the anisotropic nature of images and the importance of oriented lines and edges in visual physiology, an oriented interpolation is introduced at smaller spatial areas in those sections where a single oriented line or edge feature exists. Four quadrature filters are used to estimate the strength and orientation of single local line or edge features for each pixel in the image. A measure of consistent orientation is used within a section to determine whether it contains a single oriented feature. The quantised prediction errors and quantised section orientations are coded using arithmetic entropy coding, with a dynamic estimation of the probability distribution for each data stream. The results obtained compare favourably with those reported elsewhere. In particular, at low bit rates the reconstructed images generally have a 'painted' look to them.

This work has been supported by UK SERC and British Telecom.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

July 1988

Image Data Compression

M.Todd,R.Wilson

Department Of Computer Science
University Of Warwick
Coventry
CV4 7AL

Table of Contents

July 21, 1988

Introduction

In its raw form digital image data usually consists of a 2-dimensional rectangular array of N by M picture elements called *pixels*. Each pixel is a data sample representing the intensity or *luminance* and for colour images also the colour or *chrominance* of the image at the point in the image defined by the co-ordinates $\{x,y\}$ of the pixel. These images may be obtained from the real world, via TV/video cameras and digitisation equipment, they may be medical images produced by X-ray or NMR equipment, or computer produced graphics images.

The quality of these digital images will depend on the number of pixels used to represent them and the amount of information stored for each pixel. For monochrome images with approximately equivalent quality to TV transmissions an array of around 512 by 512 pixels is required with 8-bits of information for each pixel i.e. 256 different gray-levels. Thus $512 \times 512 \times 8 = 2{,}097{,}152$ bits are required to represent a single mono-chrome image. Video pictures require around 25 frames per second i.e. 52 million bits per second. The number of binary bits of information required to represent the image defines the *bandwidth* of the image. The total number of bits required by the image divided by the total number of pixels in the image is known as the bit rate (in bits per pixel) for the image, and is the most common way of defining its bandwidth.

Image data compression is the art of reducing the number of bits required to represent a digital image whilst still retaining the necessary quality or fidelity of the image. This fidelity may vary from being an exact representation of the original to a caricature depending on the application. For medical images it may not be acceptable to lose a single nuance of information from the original, whereas for video telephones it may only be necessary that the visual perception of the image remains roughly the same (i.e. the person at the other end of the phone can be recognized).

Data compression is possible because raw image data contains a large amount of *redundancy*. The redundancy comes from the fact that most images have a considerable degree of structure, and algorithms can be developed to use this structure to represent the image in a more compact form. There are two categories for techniques which reduce redundancy, the first being those which retain an exact representation of the original image by eliminating as much of the statistical redundancy as possible. This is known as *noiseless* coding, and restricts the amount of compression that can be achieved. The second permit errors between the compressed image and the original, which allows for a higher degree of compression, but which may introduce distortions into the image. By considering some of the findings about how the human visual system works the perceived distortions produced by these errors can be reduced. Thus a certain degree of distortion will be allowed between the coded image and the original, and there will be a trade-off between the compression factor that can be achieved and the degree of distortion of the image. The maximum acceptable distortion of the image will depend on the application of the coding scheme.

The *coder* algorithm takes the raw image data as input, and produces coded data streams for transmission or storage. Since the only input to the *decoder* algorithm will be these data streams, they must contain all the information necessary to rebuild the image, including any image

dependent control information used by the algorithm.

There are two main categories of coders/decoders.

(1) Predictive

These algorithms attempt to predict the pixel values from the pixels which have already been coded, and then code the differences between the pixel and its predicted value. Thus they only code new information, rather than information already known from the previously coded pixels. The most common form is DPCM.

(2) Transform

These algorithms [10],[11],[12] involve transforming the pixel data into a different domain to the original spatial domain. The aim is to find a domain which decorrelates the data. Each domain will be characterized by a set of orthogonal basis functions for the domain. The transforms to these domains are usually done first, then a coding algorithm is applied to the data in the new domain. At the decoder, the decoding algorithm is performed, then the inverse transform is performed.

The work described in this report is concerned with improving an existing algorithm *recursive binary nesting* (RBN) developed at the British Telecom Research Labs (BTRL). Although RBN is essentially a predictive system, it is spatially non-causal and can be considered therefore as representing an intermediate form of coder between traditional predictive and transform systems (see also [18],[19]).

1. Section 1 — Recursive Binary Nesting

1.1. Introduction

The *recursive binary nesting* algorithm was developed by BTRL[4] as a method of compressing single frame images for transmission. The algorithm attempts to take account of the structure in an image, and to code areas with a large degree of activity by using a high density of pixels and areas with little activity by using a lower density of pixels. Activity refers to rapid and varying changes in the image characteristics from pixel to pixel, which makes prediction difficult. Starting with the entire image, the algorithm recursively segments the image into smaller sections, and attempts to represent each section with a minimum amount of information. If the representation is not sufficiently good, then that section is segmented further. The advantage of this method is that it provides a simple and efficient method of having variable resolution within the image. Although the basic algorithm for RBN is very simple, in order to achieve the highest compression factors a number of more complicated (and computationally more expensive) techniques must be used.

1.2. Algorithm

The general algorithm as shown in fig 1.1 involves some form of prediction which attempts to predict the value of each pixel in a section of the image from the information known about that section. The predicted values are then compared to the real values for the pixels and a decision is made on the quality of the predictions. If the quality is sufficiently then no further information is sent about that section. If it is insufficient then the section is segmented into four new sections and extra information is extracted for each section. Each new section is considered in turn, with the extra information used to produce a better prediction for the pixels within that section. The algorithm continues recursively down each section until the quality of the whole image is acceptable.

If the segmentation process is restricted to splitting the section into four equal sized rectangular quadrants, then the information required to represent each section can be restricted to its four corner pixels, and the truncation of the algorithm can be coded by a single bit for each section processed, which codes the decision whether to segment it further or not. The pixels within a section are predicted by a linear interpolation from the four corner points.

A simple splitting criterion such as a maximum error for any pixel can then be used to determine whether the section will be segmented further. Fig 1.2 shows that if it is, then five new pixels are coded in order to provide the corner points to the new sections. However because neighbouring sections are edge sharing some of these five pixels (those on the old section boundaries) may already have been coded by the segmentation of adjacent sections, and need not be coded again.

A new pixel is coded by taking the prediction error from the interpolation, quantising this error into a set number of *quantisation buckets*, and entropy coding the result. The entropy coding is described in section 4, and involves a dynamic estimation of the probability distribution of the quantisation buckets.

1.3. Splitting Criteria

The decision whether to segment a section further or not depends on the quality of the interpolation from the corner points, and the acceptable error in the final image, so a set of criteria must be devised to determine this decision. Generally, the smaller the spatial area of the section being considered, the larger the acceptable error is, and any criteria should take this into account.

Maximum Error

If the error for any pixel is greater than a particular threshold then the section must be split. The threshold varies with the size of the section being predicted, since errors in smaller sections are less visible. The maximum error criterion was found to be a good method of preventing quality deteriorating below a given level. However it was inefficient in that it causes many sections to be split which are adequately represented, except for an isolated pixel. This criterion performs very badly (in terms of compression) if the image has noise added to it, since a single pixel which is altered by noise can cause the local area to be split down to the lowest level, with a large number of additional and unnecessary corner points being passed, simply in order to represent the "noise" better.

In [4] a method of clustering local errors is used to prevent isolated errors from forcing a split.

Mean Squared Error

The mean squared error for a section of the image of size n by m is calculated from the equation:-

$$mse = \frac{1}{(n-1)\times(m-1)} \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} e^2(i,j)$$

Where $e(i,j)$ is the error for pixel $\{i,j\}$. This criterion is more robust to noise, but is an unreliable measure of quality since it takes no account of the distribution of the errors within the section. Errors which are aligned in some way have a far more noticeable effect than errors which are evenly distributed.

1.4. Distortions

Blocking Effects

The nature of the algorithm gives rise to *blocking*, where distinct blocks can be seen in the image.

Blocking can be caused by the interpolation scheme used to reconstruct the image when different sized sections are next to each other (fig 1.3). Since a neighbouring section has been split into smaller sections, there are now extra pixels available on the edges of the current block. The edges now have a difference in sampling rate which causes "T-shaped" block effects. In order to eliminate these features the interpolation scheme must take into account any extra pixels on the edges of a block being reconstructed. This can be done by waiting until all the information about

the image is sent before attempting to interpolate the section, and using the pixels on the edges of the section as well as the four corner points to interpolate the image.

They are also caused by the fact that the sections of the image are all rectangular, and a simple interpolation scheme to reconstruct the image gives rise to *gradient discontinuities* at the boundaries of the sections. These occur where there is a sharp change of gradient in the image (but not a discontinuity in the actual pixel values), which is picked up and emphasised in the human visual cortex [14]. At low bit rates lines and edges become jagged as the splitting criteria allow truncation over larger areas. This effect comes from using the cartesian axis as the directions for the interpolation.

Breaking lines

This effect is similar to the blocking effect, but is produced when a thin line crosses several section boundaries. If the line is not quite distinct enough to avoid the problem, then some of the sections will be split further than others, and the reconstruction will section up the line, blurring over some parts of it but not others. Again the effect is small but very visible. It splits what was a distinct structure emphasised by the human visual cortex [14] into a less structured feature which does not trigger the same response in the cortex, and so a distinct distortion is perceived.

The Case for Orientation

The two distortions mentioned are caused by the structure of the coder being aligned with the cartesian axis, whereas the actual image is anisotropic. Since results from visual physiology suggest that the visual cortex contains mechanisms for detecting oriented lines and edges any errors which break these features cause an amplified effect on the perception on the image. In [1] it is shown that errors in the vicinity of such features have an effect which depends on there alignment with the orientation of the feature. Maximum distortion occurs when the error is aligned at 90 degrees to the feature. It would therefore be better to align the coder structure locally to the image.

In order to do this a way of segmenting the image by curved boundaries and performing curved interpolation is needed. One possibility would be to warp the axis of the image in such a way as to align the local axes of the image with the features of the image. This was attempted but never successfully implemented, the problems of discontinuities and the overhead of the feature orientation information was not solved. The possibility of using *tensor theory* to achieve a solution might be considered. A simpler solution is to keep the cartesian structure of the coder, but to use an oriented interpolation in sections of the image containing single features. This was successfully implemented.

2. Section 2 — The Use Of Local Orientation Within The RBN Algorithm

2.1. Introduction

The purpose of the orientation filters is to produce a description of the visually important features in the image. Features are considered to be distinct lines or edges, or areas of texture where the individual features are not distinct, but are part of a pattern of features. The orientation filters used in the feature detector are those described by Knutsson, Wilson and Granlund in [1],[2], and are designed to produce a local estimate of the strength and orientation of lines and edges in an image. They also produce an estimate of the strength of non-oriented features such as textures. The filters are described in more detail in [3]. This estimate, when suitably quantised, can be passed to the coder and used to produce an interpolation, within the general RBN framework, which is adaptive to local block orientation for any block satisfying a consistent orientation criterion (fig 2.1).

2.2. Double Angle

If a vector is used to represent the strength and orientation of line and edge features within an image, then there is a problem with the fact that the features in the image have an orientation which only has a range of π, whereas the vector has a range of $2 \times \pi$. The vectors could be forced into the correct range, but this leaves difficulties in any mathematical operations on the the vectors. If two equal strength vectors at x radians and $x+\pi-\delta$ radians (where δ is small) are averaged the result is a small vector at $x+\pi/2-\delta/2$ radians. However the two features represented by these vectors have almost identical orientations, so this vector representation is not a useful way of processing the orientations. The solution used in [3] is to have a vector whose angle is twice the orientation of the feature. Thus two vectors which differ in angle by π represent two features with orientations at right angles to each other.

2.3. Filter Definitions

In order to detect both lines and edges an *analytical filter pair* (quadrature filter pair) is required (fig 2.2). This is a pair of filters which add together to give zero over half of the frequency domain. This gives them the same response to both sine and cosine functions which means they respond to both lines and edges. The equations of these filters in the spatial-frequency domain are given in polar co-ordinate form:-

$$H_e(\rho,\theta) = j\ sgn(cos\,\theta)\ exp-\left[\frac{4ln(2)}{\ln^2 B}\ln^2\left[\frac{\rho}{\rho_c}\right]\right]\cos^2\theta$$

$$H_l(\rho,\theta) = exp-\left[\frac{4ln(2)}{\ln^2 B}\ln^2\left[\frac{\rho}{\rho_c}\right]\right]\cos^2\theta$$

The choice of the functions $H_e(\rho,\theta)$ and $H_l(\rho,\theta)$ is constrained by the need to (i) provide an unbiased interpolation of angles other than those of the filters, giving rise to the trigonometric angular functions and (ii) to provide a reasonable match to the radial spectral content of typical

images[3]. These are oriented line/edge detectors whose output will be a scalar quantity representing the amount of local energy in the orientation of the positive x-axis. In order to get a filter at an arbitrary orientation a rotation of the filter pair is introduced.

$$H_e^k(\rho,\theta) = H_e(\rho,\theta-\theta_k)$$

$$H_l^k(\rho,\theta) = H_l(\rho,\theta-\theta_k)$$

The filter pair H_e^k and H_l^k give an output which is a scalar quantity representing the amount of local energy in the orientation defined by θ_k. In order to get a per pixel vector of the energy, the outputs from several differently oriented filter pairs will be combined to estimate the orientation of the energy. Four filter pairs are used[3], each one being offset from the previous one by 45 degrees giving a table for θ_k:-

k	θ_k
1	0
2	45
3	90
4	135

The outputs of these four filter pairs are combined into a vector which represents the oriented local energy. If the spatial domain filters are h_e^k and h_l^k, and the image data is $g(x,y)$, then two quantities can be defined:-

$$s_k(x,y) = h_e^k(x,y) * g(x,y)$$

$$c_k(x,y) = h_l^k(x,y) * g(x,y)$$

These are the outputs from the four filter pairs. Now define:-

$$E_k(x,y) = \sqrt{s_k^2(x,y) + c_k^2(x,y)}$$

Then $E_k(x,y)$ is a scalar quantity which represents the magnitude of the energy in the orientation θ_k. The orientation vector $\alpha(x,y)$ for each pixel is defined by:-

$$\alpha_x(x,y) = \beta(x,y)\times\cos(2\xi(x,y)) = E_2(x,y) - E_4(x,y)$$

$$\alpha_y(x,y) = \beta(x,y)\times\sin(2\xi(x,y)) = E_1(x,y) - E_3(x,y)$$

$$\alpha(x,y) = \begin{bmatrix} \alpha_x(x,y) \\ \alpha_y(x,y) \end{bmatrix}$$

$$\gamma(x,y) = \sum_k E_k(x,y)$$

Where $\beta(x,y)$ is the magnitude of the orientation vector and $\xi(x,y)$ is the angle of orientation. $\gamma(x,y)$ is a measure of the total local activity, which includes both oriented and non-oriented activity. Note that the equations give $\xi(x,y)$ in the range 0 to π since the cos and sin equations have double angles. This is because the feature angle can only be specified in this range, there being no definable difference between a feature oriented at $\pi/2$ and one at $3\pi/2$. The X and Y components of the orientation vector with the feature angle rather than the double angle can be calculated from:-

X-component = $\beta(x,y)\cos(\xi(x,y))$

Y-component = $\beta(x,y)\sin(\xi(x,y))$

2.4. Implementation

The feature detector is implemented as a *filtering* operation, and its result is a per pixel measure of the activity. The filtering operation can be done in either the spatial domain by a convolution operation, or the spatial-frequency domain by a simple per pixel multiplication. The determining feature for the method to be used is the speed of computation. For the convolution in the spatial domain a filter of size n by m, must be designed. If the image is of size N by M pixels then the computation in the convolution operation is of order $n \times m \times N \times M$ multiplications (the associated additions, comparisons and loop indexing are ignored here as they are insignificant compared with the multiplications). For the spatial-frequency domain method the major computational burden is the transform from the spatial domain to the spatial-frequency domain and back. This is of the order $\log_2(X) \times N \times M$, where X is the maximum of N and M. From this consideration, the spatial-frequency domain is faster if $\log_2(X) < n \times m$.

For images of 512^2 or 256^2 pixels, $\log_2(X)$ is 9 and 8 respectively. The size of the spatial filter that would be required is at lest 8 by 8, so that the spatial-frequency domain is a factor of approximately 64/9 = 8 faster than the spatial domain. However the spatial convolution method is very adaptable to parallel processing techniques, and in the work of Knutsson et al [1],[2],[3] the spatial domain was used on the available parallel processing GOP processor. In this work, a SUN-3/160 with a 68881 floating point co-processor was used, so the spatial-frequency domain was chosen.

The filtering operation is performed in the Fourier domain, so the image data $g(x,y)$ is first transformed using the *fast Fourier transform*, which is a fast implementation of the Fourier transform equation, to give $G(u,v)$, which is a complex variable. Then the filtering operation is performed for each of the four filters (k=1..4), to give four complex filtered images $F_o^k(u,v)$.

$$F_o^k(u,v) = G(u,v) \times (H_e^k(u,v) + H_i^k(u,v))$$

The four filtered images $F_o^k(u,v)$ are then transformed back into the spatial domain using the inverse fast Fourier transform, to give $f_o^k(x,y)$, which are still complex images. Note that this required four inverse transforms, one for each image, and therefore requires considerable computational time. The complex magnitudes of these images are then taken, to give four scalar images $E_k(x,y)$.

$$E_k(x,y) = \sqrt{(Re[f_o^k(x,y)])^2 + (Im[f_o^k(x,y)])^2}$$

These are the $E_k(x,y)$ given in the theory in section 2.3, from which the parameters can be calculated. Fig 2.3 shows the general scheme of the implementation, and fig 2.4 to fig 2.7 show the envelopes of the four filters.

2.5. Oriented Interpolation

The simple linear operation employed in this scheme operates by calculating the intersection of a vector in the direction of the orientation of the section from the interior pixel being coded to the edge boundaries. At these intersections a value is obtained by linear interpolation from the two nearest pixels for the value at the point of intersection. Each interior pixel has two boundary intersections, and a linear interpolation from these two gives the predicted value for the pixel. In order to be a reasonable prediction the orientation of the features within a block must be consistent. This simple anisotropic interpolator will not work if a given block contains two oriented features at different angles.

2.6. Local Consistency

Consider a rectangular section of the image of size $n \times m$ pixels at a particular stage of the RBN splitting algorithm (fig 2.8). Each pixel $\{i,j\}$ in the section has an estimate for magnitude and orientation of any local line or edge features in the form of the vector $\alpha(i,j)$. Where the magnitude of $\alpha(i,j)$ represents the strength of the feature and the angle of $\alpha(i,j)$ is twice the orientation of the feature. An average value for the orientation vector $\hat{\alpha}$ of the section can be determined from the vector summation of $\alpha(i,j)$ within the section.

$$\hat{\alpha} = \frac{1}{(n-1)\times(m-1)} \sum_{i=0}^{n-1} \sum_{j-0}^{m-1} \alpha(i,j)$$

In order to determine if the section can be represented by an oriented interpolation the consistency of the local orientation within the section must be determined. One measure of consistency is to compare the magnitude of the average vector with the average magnitude of the individual vectors. The average magnitude of the vectors is given by

$$\beta = \frac{1}{(n-1)\times(m-1)} \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} |\alpha(i,j)|$$

The ratio of the two gives a coefficient c, which lies between 0 and 1, and which is a measure of the consistency of the individual orientations weighted by their magnitudes.

$$c = \frac{|\hat{\alpha}|}{\beta}$$

The more consistent the dominant vectors (i.e. those with relatively large magnitudes) are, the closer the value of c will be to 1, and a threshold can be determined to provide a decision on the consistency of orientation , and therefore the coding strategy for the section. Note that because of the double angle representation, vectors which cancel each other out, i.e. have equal magnitude but with 180 degree difference in angle are equivalent to equal strength features at right angles to each other.

In addition, β gives a measure of the total strength of the features within the section, which can be used to determine if there are any significant features within the section. If there are no strong features then the average orientation of any weak features (even if it is not very consistent) can be used to interpolate the section rather than aligning the interpolation along the cartesian axis. This gives an area which is close to a feature but does not include it, an "orientation" which tends to be aligned with the feature and which can give the image a "painted" look at low bit rates.

The value of $\hat{\alpha}$ for each section is quantised to a given number of different orientations. The quantised value can then be entropy coded.

2.7. Block Edges

Once a section satisfies the criteria for consistent orientation, it is interpolated along the direction defined by the average orientation vector $\hat{\alpha}$. However, the interpolation scheme requires all the pixels on the section boundaries to be known and so these pixels must be coded first. They are coded with a 1-dimensional RBN algorithm along each section edge. Since adjacent sections share edges, and these edges need only be coded once, approximately half the pixels will already have been coded. Even so, this is a major cost (in terms in bit rate) for the overall algorithm.

An alternative strategy for coding the edges of the section is to predict the edge pixels from any pixels already coded on the edge in the direction of the orientation vector (fig 2.9). This strategy has a problem in that for large areas of the image which have adjacent sections with acceptable orientation, any quantisation error in an edge at one end of the area will be used in the prediction right across the area, leading to a spreading of the error across the whole area. These linear errors are aligned with the image orientation, but are distinct from the actual image, and are a substantial distortion. In particular, at low bit rates they make the image look at if it has been badly painted with a large brush.

2.8. Interpolation

Once all the edge pixels of a section have been coded, the oriented interpolation of the interior pixels can be performed. The interpolation for each pixel involves finding the intersection between the orientation from that pixel and two of the section edges (fig 2.10). The two edge pixels nearest to the intercept are used to give a linear interpolation at the point of intercept (which will generally not fall exactly at a pixel co-ordinate). The two values at the points of intercept are then linearly interpolated to give the value at the pixel co-ordinate being coded.

3. Section 3 — Entropy Coding

3.1. Introduction

The function of the entropy coder within the overall system is to perform a noiseless coding on the data produced by the general RBN algorithm. The coding performs the elimination of the statistical redundancy in the data.

A data stream consists of a sequence of discrete data values. If each possible data value is assigned a unique symbol i, then the set of all possible symbols $\{i\}$ is the alphabet for the data stream. For a data stream with an alphabet of N symbols, a simple binary code can be devised for each symbol with $\lceil \log_2(N) \rceil$ bits per symbol.

Each symbol in the alphabet will have a particular probability $P(i)$ of occurring, and set of probabilities for all the symbols $\{P(i)\}$ is the probability distribution for the alphabet. If the set $\{P(i)\}$ is non-uniform then a compression over the simple binary code can be achieved by *entropy coding* the symbols. This is achieved by using more bits for the least probable symbols and fewer bits for the more probable symbols. The optimum number of bits for a symbol i with probability $P(i)$ is given by its *entropy*, which is defined as $-\log_2[P(i)]$.

A *codeword* is built up for the data stream by combining the codes for the symbols that occur in the stream. For Huffman coding this simply involves devising unique binary codes for each symbol with the number of bits in the code for a symbol i determined by $\log_2[P(i)]$. The codes for the symbols which occur in the data stream are then concatenated in the order in which they occur into a single codeword. For arithmetic coding the codeword is built up by combining the codes for each symbol into the codeword using an arithmetic operation.

In the building or decoding of the codeword the entire codeword may have to be stored until the last symbol in the data steam has been dealt with. It is more efficient if only the end of the codeword must be stored, so that the codeword only needs to be stored to a finite precision and the decoding can be started before the codeword is complete. For a concatenation coder such as a Huffman coder this is clearly possible. However, for arithmetic coding, where the codeword is built up by an arithmetic combination of all the symbols in the data stream, there is a problem with the precision (in bits) to which the codeword must be held. This is the *infinite precision problem*, since the precision required is undefinable if the number of symbols in the data stream is not known. A problem related to the infinite precision problem is the *carry-over problem*, where the codeword may contain an arbitrarily long sequence of 1's to which an addition may cause a carry through the length of the sequence. In order to do this the codeword would have to be held to the length of the maximum possible such sequence, which for an unknown number of symbols is undefined. A further problem is the *increasing precision problem*, which occurs when two finite precision probabilities are multiplied together. The result will generally require more bits precision than either of the original probabilities, and since arithmetic coding involves such a calculation for each event, the precision required will be constantly increasing.

Entropy Of Sequences

The entropy for an symbol i with a probability $P[i]$ is given by,

$$entropy(i) = -\log_2[P(i)]$$

The optimal number of bits required for a binary codeword to represent such a symbol is equal to its entropy. In addition the codeword must be unique to that symbol, such that given the codeword and the set $\{P(i)\}$ the symbol i can be uniquely determined. Since only an integer number of bits can be used in any practical codeword, the entropy for the symbol must be approximated by an integer, resulting in a loss of optimality. Huffman coding is a well known method of devising such a set of unique codewords, which are the best approximation possible given the integer constraint. However under certain conditions, the deviation from the optimal can be quite large.

Example

symbol	number of occurrences	$P(i)$	$\log_2[P(i)]$	Huffman codes	nr of bits N(i)
a	9000	0.9	0.152	0	1
b	500	0.05	4.323	10	2
c	500	0.05	4.323	11	2

Entropy per event = $\sum_i P(i) \times \log_2[P(i)] = 0.9*0.152+0.05*4.323+0.05*4.323 = 0.5691$

For the Huffman codes, the average number of bits per event is given by,

$$\sum_i N(i) \times P(i) = 0.9*1+0.05*2+0.05*2 = 1.1000$$

The entropy for a sequence of two events (i_1, i_2) is given by,

$$entropy(i_1, i_2) = -\log_2[P(i_1, i_2)]$$

Where $P(i_1, i_2) = P(i_1) \times P(i_2|i_1)$

and, $P(i_2|i_1)$ is the conditional probability of i_2 being the second event given that i_1 was the first.

If the events are independent then the equation reduces to,

$$P(i_1, i_2) = P(i_1) \times P(i_2)$$

If there are N possible symbols for each event, then there are N^2 possible combinations of two events in sequence. The probabilities $P(i_1, i_2)$ can be calculated for each sequence, and a set of

integer length codewords devised using the huffman coding method to represent each sequence. These codes lead to an average number of bits per event which is closer to the theoretic entropy than the single event codes, since the probabilities $P(i_1, i_2)$ are smaller than either $P(i_1)$ or $P(i_2)$, the approximation of $-\log_2[P(i_1, i_2)]$ to an integer contains a smaller percentage error. In addition the single event codes concatenate this percentage error for two events, whereas the double event codes have only a single percentage error for each pair of events.

Example (assuming events are independent)

symbol	$P(i_1) \times P(i_2) = P(j)$	$\log_2[P(i)]$	Huffman codes	N(j)
aa	0.9 * 0.9 = 0.81	0.304	0	1
ab	0.9 * 0.05 = 0.045	4.475	1100	4
ac	0.9 * 0.05 = 0.045	4.475	1101	4
ba	0.05 * 0.9 = 0.045	4.475	111	3
bb	0.05 * 0.05 = 0.0025	8.646	10010	5
bc	0.05 * 0.05 = 0.0025	8.646	10011	5
ca	0.05 * 0.9 = 0.045	4.475	101	3
cb	0.05 * 0.05 = 0.0025	8.646	10000	5
cc	0.05 * 0.05 = 0.0025	8.646	10001	5

The entropy per double event is now,

$$entropy(i_1, i_2) = \sum_j P(j) \times \log_2[P(j)] = 1.1382$$

Therefore the entropy per single event = $\frac{1.1382}{2} = 0.5691$ as before.

The average number of bits per double event is now,

$$\sum_j N(j) \times P(j) = 1.49$$

Therefore the average number of bits per symbol is now $\frac{1.49}{2} = 0.745$, which is closer to the entropy.

The entropy for a sequence of n events is,

$$entropy(i_1, \cdots, i_n) = -\log_2[P(i_1, \cdots, i_n)]$$

Where $P(i_1, \cdots, i_n) = P(i_1) \times P(i_2 | i_1) \times \cdots \times P(i_n | (i_1, \cdots, i_{n-1}))$

Again if the events are independent this reduces to

$$P(i_1, \cdots, i_n) = P(i_1) \times \cdots \times P(i_n)$$

If n is the length of the entire sequence of events to be coded, then the integer approximation of this entropy is the optimal number of bits required to code the sequence. In theory Huffman coding can be used to derive the unique codeword to represent the sequence. However in practice the use of the standard Huffman coding algorithm would involve computing the N^n possible $P(i_1, \cdots, i_n)$, which for large n is clearly impractical. What is required is an algorithm which computes a unique codeword of length $-\log_2[P(i_1, \cdots, P(i_n)]$ given the sequence $(i_1, \cdots, i_n)$ and $(P(i_1), \cdots, P(i_n))$. One simple algorithm is arithmetic coding. It is particularly efficient when the events are not independent.

3.2. Modeling Arithmetic Coding - The Number Line

The Arithmetic coding process can be viewed as a method of segmenting a section of the real number line between 0.0 and 1.0 into intervals representing each symbol, the size of each interval depending on the probability of the symbol. e.g.

for the symbols	a	b	c	d
with probabilities (P_i)	¼	½	⅛	⅛

To give the line segment shown in fig 3.1.

The starting point and length of each interval are represented by binary fractions, so

interval	start	start in binary	length	length in binary
a	0	0.0	¼	0.01
b	¼	0.01	½	0.1
c	¾	0.11	⅛	0.001
d	⅞	0.111	⅛	0.001

To code an event "b", 0.01 is selected for the codeword C with an associated interval size A of 0.1. Any point within the range C to $C+A$ (excluding $C+A$) can be used to represent the string "b". Within this interval a unique point can be chosen whose position can be defined with a number of bits given by given by $\lceil -\log_2(A) \rceil$. Since the length of the interval for symbol i is equal to the probability $P(i)$ the number of bits required for the symbol i is $\lceil -\log_2[P(i)] \rceil$ i.e. its entropy. For the case of symbol "b" this point is 0.1. Note that the zero before the decimal place is not included in the bits needed to define the point.

To code the next symbol, the b-interval is split into new intervals, again based on the probabilities of the symbols. The next line segment is shown in fig 3.2.

The starting points for each interval are now,

interval	start	length
ba	0.01	0.001
bb	0.011	0.01
bc	0.101	0.0001
bd	0.1011	0.0001

Thus the codeword for "bb" is 0.011 with interval size 0.01.

Again a point can be chosed within this interval which is defined by $\lceil -\log_2 \rceil$ of the interval size. The interval size is $P(i_1, i_2)$, and so the number of bits required is again the entropy. In the example, for string "bb",

$$-\log_2(A) = -\log_2[P(b,b)] = -\log_2[P(b) \times P(b \mid b)] = -\log_2[P(b) \times P(b)] = -\log_2(½ * ½) = -\log_2(0.01) = 2.$$

This means a point can be found within the interval 0.011 to 0.101 with 2 bits, which uniquely represents the string "bb". The point is 0.10.

This process is continued until the all n events have been coded. The interval size A is now equal to $P(i_1, \cdots, i_n)$, and so $-\log_2(A)$ is the entropy for the entire sequence. A point can now be chosen with $\lceil \log_2(A) \rceil$ bits which is unique to that sequence. The binary representation of this point is the final unique codeword for that particular sequence of n events.

To decode the codeword the line is segmented in the same manner as the coder, and the interval boundaries are compared with the codeword to see which interval it lies in. This interval will determine which symbol is decoded, and provided the probabilities are the same at the encoder and decoder, it will be the same symbol that was coded.

The advantage of this method over an extended Huffman coding algorithm is that only the probability for the entire sequence is being calculated, and the probabilities for all the other possible sequences are not needed. In addition, the algorithm works just as efficiently for events which are not independent, provided a separate statistics unit produces the same probabilities for both the encoder and decoder. For huffman coding of single events, this would require a set of huffman tables for each possible probability distribution.

3.3. Algorithmic View Of Model

The segmentation of the number line can be described by a set of equations on two registers, which will be treated as binary fractions of infinite precision (between 0 and 1). To initialise the algorithm, the interval size register A is set to 1.0, and the codeword register C is set to 0.0. $P(i)$ will represent the probability of the symbol i and $P_c(i)$ will represent the cumulative probability for symbol i,

$$P_c(i) = \sum_{j=1}^{j=i-1} P(j)$$

At each stage a single symbol is coded into the codeword, by splitting up the current interval. From the diagram of the general number line model (fig 3.3) equations can be derived for the new values of C and A when an event i is coded:-

$$C = C + P_c(i) * A;$$
$$A = A * P(i);$$

The value $P_c(i)*A$ is called the augend, and represents the step along the current segment to the start of the interval for the symbol being coded. If the $\{P(i)\}$ are conditional on previous symbols, then to achieve the entropy, $\{P(i)\}$ must be adjusted at each stage. After all the symbols have been coded, C will define a unique codeword for the data, and $-\log_2(A)$ will define the precision that C must be held to.

To decode the string, the algorithm is initialised with $A=1.0$ and C equal to the codeword. Then for each stage, the codeword C is examined to see which of the new intervals it lies in. This is done by finding the symbol i for which $P_c(i)*A \leq C < P_c(i-1)*A$, giving,

```
i = 0;
while(C < Pc(i) × A) i++;
    decode symbol i;
    C = C − Pc(i)*A;
    A = A*P(i);
```

Multi-symbol vs Binary Arithmetic Coding

A significant simplification of the algorithm is achieved if the events are restricted to a binary rather than multi-symbol alphabet[6]. i.e. There are only two possible events at each stage, the more probable symbol (MPS) and the less probable symbol (LPS), and a probability Q is assigned to the LPS and $1-Q$ to the MPS. Any multi-symbol alphabet can be broken down into a sequence of binary events for each code symbol, so no generality is lost in the binary arithmetic coder, by using a *binary decision tree.* A binary tree is devised with the same number of leaf nodes as there are symbols in the alphabet. The symbols are then assigned to the leaf nodes. The paths from each node to its two children are assigned the symbols MPS and LPS, and a probability is assigned to each path. The path from the root node to the leaf node associated with a symbol determines the binary events used to code it. Note that the probabilities for each of these binary events can be calculated from the probabilities for the symbols, as shown in fig 3.4. The table of symbols now has only two elements,

symbol	$P(i)$	$P_c(i)$
LPS	Q	0
MPS	$1-Q$	Q

and the coder algorithm is

```
if the event is MPS
    C = C+Q*A;
    A = A*(1-Q);
else {event is LPS}
    C = C+0;
    A = A*Q;
```

the decoder is

```
if C≥Q
    MPS decoded
    C = C-Q*A;
    A = A*(1-Q);
else
    LPS decoded
    C = C-0;
    A = A*(Q);
```

This is the general form of the binary arithmetic coder, with Q determined by a separate statistics unit, which provides the same value to both the coder and decoder.

Increasing Precision Problem

If the A and Q values are stored to a finite precision, then in general the result of the multiplication $A*Q$ will require twice this precision. In order to prevent this precision expanding indefinitely, the answer can be truncated to a fixed precision. In the line model this simply represents placing the boundary of the LPS/MPS segments in slightly the wrong place. But since the precision is the same at both encoder and decoder, they will both place the boundary in the same place, so the model is still valid. This increasing precision solution simply represents an approximation which reduces the compression slightly.

Finite Precision Problem

As the coding progresses through the string of symbols, A will get smaller and smaller, and only the last few digits will be active in the calculations. Thus although in theory infinite precision registers are required, in practice a window onto the last few binary digits is needed. The only problem with this is when a carry occurs in C , and propagates past the edge of the window. This is called the *carry over problem.*

Carry Over - A Buffering Problem

The window on the C-register into which Q is added on a MPS covers the end of the C-register bit stream and ensures that the bits before it are not required for the processing. However if a stream of successive 1's occur in the codeword, then there may be a problem with a carry which will propagate all the way through the stream. This tends to imply that the buffered bits from the C-register must be held to an arbitrary precision and compute the entire codeword before sending it. However two points can be made which clarify the nature of the problem, and lead to an easy solution[6].

1. Consider the number line shown in fig 3.5.

No further coding of symbols can cause C to be greater than or equal to $C+A$. Thus once a carry propagation occurs at any bit position which represents a value greater than A on the codeword, to get a second carry at that position a number greater than A would have to be added to the codeword. This can never occur, so once a carry has occurred at a particular bit position greater than A, there can never be a second carry at that position. Thus only a single carry must be accounted for.

2. In order to start decoding it is not necessarily to have the carry propagate to its natural end. The value it represents at a particular bit position must however not be lost and it must complete its propagation by the time the symbol that caused it is decoded. i.e. At each MPS $Q*A$ is added into the codeword in the encoder, and is looked for at the decoder to indicate the symbol that was coded. The carry propagation can never change the codeword in such a way as to effect the decoders decision about whether this is present for any symbol which precedes the symbol which caused the carry. So the carry need not be propagated until it reaches the first bit in the carry stream in the decoder. This leads to a solution to the carry-over problem, called bit-stuffing.

Bit Stuffing

The solution to the carry problem is to add an extra zero bit to the code stream after each sequence of 1's which could cause a problem. This 0 will absorb any subsequent carry in the coder and since from the first point there will never be a second carry, thus the problem has been solved for the encoder.

The decoder must know when a bit was stuffed and so the criterion to stuff a bit is a sequence of n 1's in a row. When such a sequence occurs, the next bit is assumed to be the stuffed bit, and if it is a 1, the carry is propagated in the decoder. Since from the second point the carry does not need to be propagated until the symbol that caused it is being decoded, and when it is propagated in this scheme it has not yet got to that symbol, the problem has been solved for the decoder.

This process adds an extra bit to the bit rate every time a sequence of n consecutive 1's appear out of the coder. Assuming that the {1,0} output is random (if it is not, the output is not fully compressed), a stuffed bit is expected every 2^n bits. In order to deal with bit stuffing in practice, the output from the Q-coder must be buffered to at least n bits before it is passed to the decoder.

3.4. Q-coder

Elimination Of The Multiply

In order to prevent A becoming arbitrarily small it can be scaled up (*renormalised*) to approximately 1 whenever it gets too low. Provided C and A are scaled in synchronization, this can be viewed as a rescaling of the entire line segment, which causes no alteration to the model. This method was developed at IBM for an application involving DPCM coding of images [8],[9].

The normalisation scheme chosen is a multiplication by 2 whenever the A register falls below 0.75 after coding a symbol, and continuing the multiplication until A is greater then or equal to 0.75. This gives us a value of A which is always in the range $1.5>A\geq 0.75$ before each symbol is coded. Since a multiply by 2 is a shift of one bit to the left, a single bit will be shifted out of our C register each time it is renormalised. These bits will be stored in a buffer, and when the buffer is full, it will be passed to the decoder together with any bit stuffing. Thus the decoder can start decoding before the coder has finished coding the entire string.

With A in the range $1.5>A\geq 0.75$ the approximation $A=1$ can be made and the equations in the algorithm simplified by eliminating the multiply, giving for the coder algorithm,

```
if event=MPS
     C = C+Q;
     A = A-Q;
else {event=LPS}
     A = Q;
```

and for the decoder,

```
if C≥Q
     MPS decoded
     C = C+Q;
     A = A-Q;
else
     LPS decoded
     A = Q;
```

Where $A*Q$ has been approximated to Q and $A*(1-Q)$ to $A-Q$. $A*(1-Q)$ is approximated to $A-Q$ rather than $1-Q$, since the lengths of the two new segments add up to the length of the previous one.

Probability Estimation

One of the advantages of arithmetic coding is its ability to change the probabilities between successive symbols without changing the basic algorithm or the complexity of the calculations. Thus substantial extra compression can be achieved for non-stationary data if the probabilities can be estimated at each stage. Since the estimated probability (Q_e) must be identical for both the encoder and decoder, the estimating technique must depend on data known to both the encoder and decoder, and it should estimate the probability for each event, so the estimation process must be reasonable fast. One method would be to count the previous symbols and estimate the probability from these events.

The Q-coder provides a simple solution to estimating Q_e which involves the ratio of MPS to LPS events, and which adjusts its estimate of Q_e every time a renormalization occurs[9]. It is based on having a fixed number N of possible probability values for the LPS event, denoted by Q_e^k. Each Q_e^k will be represented using a set number b of bits. This means that the minimum probability estimate that can be represented is 2^{-b}, and the N values will be spread across the range $Q_e^1 = 0.5$ to $Q_e^N = 2^{-b}$. Note that if the probability estimate becomes greater than 0.5, the definitions of LPS and MPS will be swapped round, so 0.5 is the maximum probability that needs to be represented.

If a LPS occurs A will be assigned Q_e, and since Q_e is less than 0.5, it will force a renormalization. The estimation process assumes that since the least probable event has occurred, the estimate of Q_e is too low and k is decremented by 1, taking the estimate Q_e from Q_e^k to Q_e^{k-1}. If a MPS occurs A will have Q_e subtracted from it. After a sequence of n successive MPS's, A will go below 0.75 and cause a renormalization. The process now assumes that since the more probable event has occurred, the estimate of Q_e is too high and k is incremented by 1, taking Q_e from Q_e^k to Q_e^{k+1}. Note that at Q_e^N a MPS can not increment k any further, so it is left as it is. However if a LPS occurs at Q_e^1 then the LPS and MPS events must be swapped round (i.e. {LPS=0,MPS=1} to {LPS=1,MPS=0}), and k remains the same. The value of Q_e is therefore being continuously updated, and it is possible to show that it reaches a dynamic equilibrium at approximately the correct probability, and will track this probability as it changes. An approximate proof of this result is given in [9].

This process can be viewed as a finite state machine, with each state representing a particular probability value Q_e^k. Thus a renormalization caused by a LPS will take it to a state with slightly greater Q_e, and a renormalization caused by a MPS will take it to a state with a slightly smaller Q_e. If a LPS occurs in the state with the largest Q_e value (which will be just under 0.5) then the definition of the LPS and MPS are swapped round. The set of values used for Q_e^k must be determined from experience to give a good distribution (so that Q_e can be estimated accurately) and a fast dynamic response to a changing probability distribution. These two factors oppose each other, and in order to improve the dynamic response for a given granularity of Q_e values, a rate variable could be used to determine how many states are jumped for each renormalization.

Multi-context Q-coder

For every binary event the Q-coder deals with, there must be an estimate Q_e of the events probability. A single Q_e could be used for all events the coder deals with, however this will not

achieve the highest compression rate if the binary events have different probabilities. The events are split up into different contexts, with each context having a separate estimate for Q_e. Clearly there should be different context for the prediction error and orientation data. Within multi-symbol events such as the prediction errors where a binary tree is used to convert is to binary events, each branch of the tree will generally have a different probability and so a separate context should be used for each branch. In addition the data statistics will vary with the recursion level of the RBN algorithm, so separate contexts can be used for the various levels of recursion.

In the finite state machine representation, this requires a separate machine for each context. In practice there is a separate pointer into the finite state tables for each context.

3.5. Arithmetic Coding Within RBN

The arithmetic entropy coding scheme is used to compress the data stream from the truncation addressing, prediction error and orientation information. Each is coded with a different context scheme.

3.6. Context Schemes

Prediction errors

The probability distribution of the prediction error is generally biased toward zero, and tails of towards larger errors. The binary decision tree used is shown in (fig 3.6) and is an attempt to minimise the computation involved in coding it.

Orientation

The data stream for the orientation information consisted of the quantised orientation vectors rather than any prediction errors, and thus a simple complete binary tree was used as the binary decision tree (fig 3.7). The number of orientations allowed by the quantisation scheme will determine the size of the tree.

4. Results

Photos 6.1 to 6.18 show the results of the work on three common images For the lower bit rates, the images were pre-filtered using an anisotropic low-pass filter controlled by the quantised average orientation $\hat{\alpha}$ for a window surrounding each pixel. For areas with no consistent orientation, an isotropic low-pass filter is used. No post-filtering is done on the images shown.

The characteristics of the local features in these three images are different, and since the coder is attempting to account for these features, the performance for the images is different. The "GIRL" (photo 6.3), which is the least complicated of the three, has large areas where there are either no substantial features or a single edge feature of a particular orientation. The areas with no substantial features are highly compressible with the simple RBN algorithm, whereas those areas with a single feature can be compressed with the oriented interpolation. Areas such as the feathers, hair and eyes contain more detail and require coding to a relatively small block size. The "BOATS" (photo 6.2) image contains a large number of line features in the masts and ropes which are consistently oriented, but require accurate coding of the section boundaries for an oriented interpolation to be used. The image has some texture in the sand below the boat, and areas of no substantial feature in the sky. The "BARBARA" (photo 6.1) image is the the most difficult to code. It has large regions of stripes on the trousers and shawl which have a high spatial frequency, and which have folds in the garments leading to abrupt changes in the orientation and frequency of the features. The oriented interpolation is suitable for coding such areas provided they do not include one of these folds. In addition, the stripes in the table cloth and the wicker chair contain two distinct orientations at all but the smallest block sizes. Since the orientation filters can only detect a single dominant orientation, these areas are particularly difficult to code. Photo 6.4-6.6 show the X and Y components of the feature orientations as defined in section 2.3.

Photos 6.7-6.10 show the results of coding "BARBARA" at various bit rates. At 0.36 bits per pixel (bpp) (photo 6.8) the stripes on her knee and the lower left corner of the shawl have started to merge into her arm. This is a failure to detect a feature boundary occurring in an oriented region. For areas where this boundary has the same orientation as the feature, such as the left hand edge of the shawl, this is not a problem however when the boundary is at right angles to the feature the oriented interpolation is no longer valid. An improvement to the coder would be to distinguish between these two cases. The right hand face of the tablecloth has been reduced from a cross pattern to a stripe pattern. At 0.23 bpp more of the stripes have merged with the arms, and the two parts of the tie are starting to merge. The wicker chair has become a single stripe rather then a cross pattern, but the most noticeable distortion (because of its significance to the viewer rather than the absolute error) is the loss of detail in the eyes, which make her appear to be squinting. At 0.16 bpp the most of the stripes are not present and there is some merging between the regions.

Photos 6.10-6.14 show the results of coding the "BOATS" image at various bit rates. At 0.31 bpp the texture of the sand has been smoothed out, and it now looks more like water. The ropes have been blurred slightly and at some of the edges their intensities gradually merge with the background. There are some breaks in the ropes. At 0.26 bpp the merging and breaking of the ropes has increased, and there is a large smear at the top of the central mast. At 0.13 bpp most of the features are smeared, any nearly all the ropes have disappeared, the large smear is still present at

the top of the mast.

Photos 6.15-6.18 show the results of coding the "GIRL" image at various bit rates. At 0.20 bpp the mouth is distorted at the right edge, and the whole image looks like it has been "painted". This is a feature of the oriented interpolation of local blocks. Any errors near distinct features are "painted" out in the same orientation as the feature. At 0.12 bpp the black band on the right of the picture has some smears in it, and the nose and mouth are slightly distorted. At 0.08 bpp the whole image is slightly blurred, and features such as the eyes, mouth and nose are smeared.

5. Discussion of Results

One possibility for improvement is in the coding of the section edges for consistently oriented sections. The method used ignores the fact that adjacent edges in a section, and indeed in its neighbouring sections will generally contain similar information. e.g. in "barbara" (photo 6.1), sections in the trousers will all have similar edge information, however each edge will cost a relatively large number of bits, since it has a sinusoidal variation. Thus if the edges are coded with no reference to each other, the coding of the edges in this area of the image will be costly. One method which was attempted was to use the known orientation for the block to attempt to predict the edge from an adjacent edge, however this lead to considerable distortions. An alternative to coding the edges using a 1-dimensional RBN algorithm is to use a 1-dimensional orthogonal transform, such as a discrete cosine transform (DCT). This may lead to a better method of relating the edges to each other.

The images have not been post-filtered, and the pre-filtering used is fairly simplistic. There is considerable scope for improving the pre- and post-filtering of the images.

REFERENCES

Orientation Filters

[1] Hans E. Knutsson, Roland Wilson and Goesta H. Granlund,
"Anisotropic Nonstationary Image Estimation and its Applications:
Part I-Restoration of Noisy Images.
IEEE Trans. COM-31 Nr.3 pp.388-397 (March 1983).

[2] Roland Wilson, Hans E. Knutsson and Goesta H. Granlund,
"Anisotropic Nonstationary Image Estimation and its Applications:
Part II-Predictive Image Coding.
IEEE Trans. COM-31 Nr.3 pp.398-406 (March 1983).

[3] Hans E. Knutsson,
"Filtering And Reconstruction In Image Processing"
Linkopings Studies In Science and Technology Dissertations Nr.28

Recursive Binary Nesting

[4] J. M. Beaumont,
"The RIBENA Algorithm - Recursive Predictive Still Picture Coding",
British Telecom Memo RT4343/88/14 (March 1988).

Arithmetic Coding

[5] J. J. Rissanen and Glen G. Langdon,
"Arithmetic Coding"
IBM J. Res. Dev. vol 23 pp.149-162 (1979)

[6] Glen G. Langdon,
"An introduction To Arithmetic Coding"
IBM J.Res.Develop. 28 pp.135-149 (March 1984).

[7] Jorma Rissanen, Glen G. Langdon,
"Universal Modeling And Coding"
IEEE Trans. IT-27 Nr 1 pp.12-23 (January 1981).

[8] Joan L. Mitchell, William B. Pennebaker,
"Software Implementation Of The Q-coder"
IBM Research Report,RC 12660 (April 1987).

[9] Joan L. Mitchell, William B. Pennebaker.
"Probability Estimation For the Q-coder"
IBM Research Report RC 12659 (April 1987).

Transform Coding

[10] William K. Pratt, Julius Kane and Harry C. Andrews,
"Hadamard Transform Image Coding"
Proc. IEEE Vol 57 Nr 1 pp.58-68 (January 1969).

[11] P. A. Wintz,
"Transform Picture Coding"
Proc. IEEE Vol 60 Nr 7 pp.809-820 (1972).

[12] Wen-Hsiung Chen and William K. Pratt,
"Scene Adaptive Coder"
IEEE Trans. COM-32 pp.225-232 (March 1984).

General

[13] J. Max,
"Quantizing for Minimum Distortion"
IRE Trans. IT-6 pp.7 (1960).

[14] David H. Hubel and Torsten N. Wiesel,
"Brain Mechanisms Of Vision"
Sci. American pp.130-144 (September 1979).

[15] Arun N. Netravali and John D. Limb,
"Picture Coding:A Review"
Proc IEEE Vol 68 pp.366-406 (March 1980).

[16] A. K. Jain,
"Image Data Compression:A Review"
Proc. IEEE Vol 69 Nr 3 (March 1981).

[17] Murat Kunt, Athanassios Ikonomopoulos and Michel Kocher,
"Second-Generation Image-Coding Techniques"
Proc. IEEE Vol 73 Nr 4 (April 1985).

[18] Roland G. Wilson,
"Quad-Tree Predictive Coding:A New Class Of Image Data

Compression Algorithms"
Proc IEEE Conf on A.S.S.P. San Diego,CA (March 1984).

[19] Edward H. Adleson and Eero Simoncelli,
"Orthogonal Pyramid Transforms For Image Coding"

fig 1.1

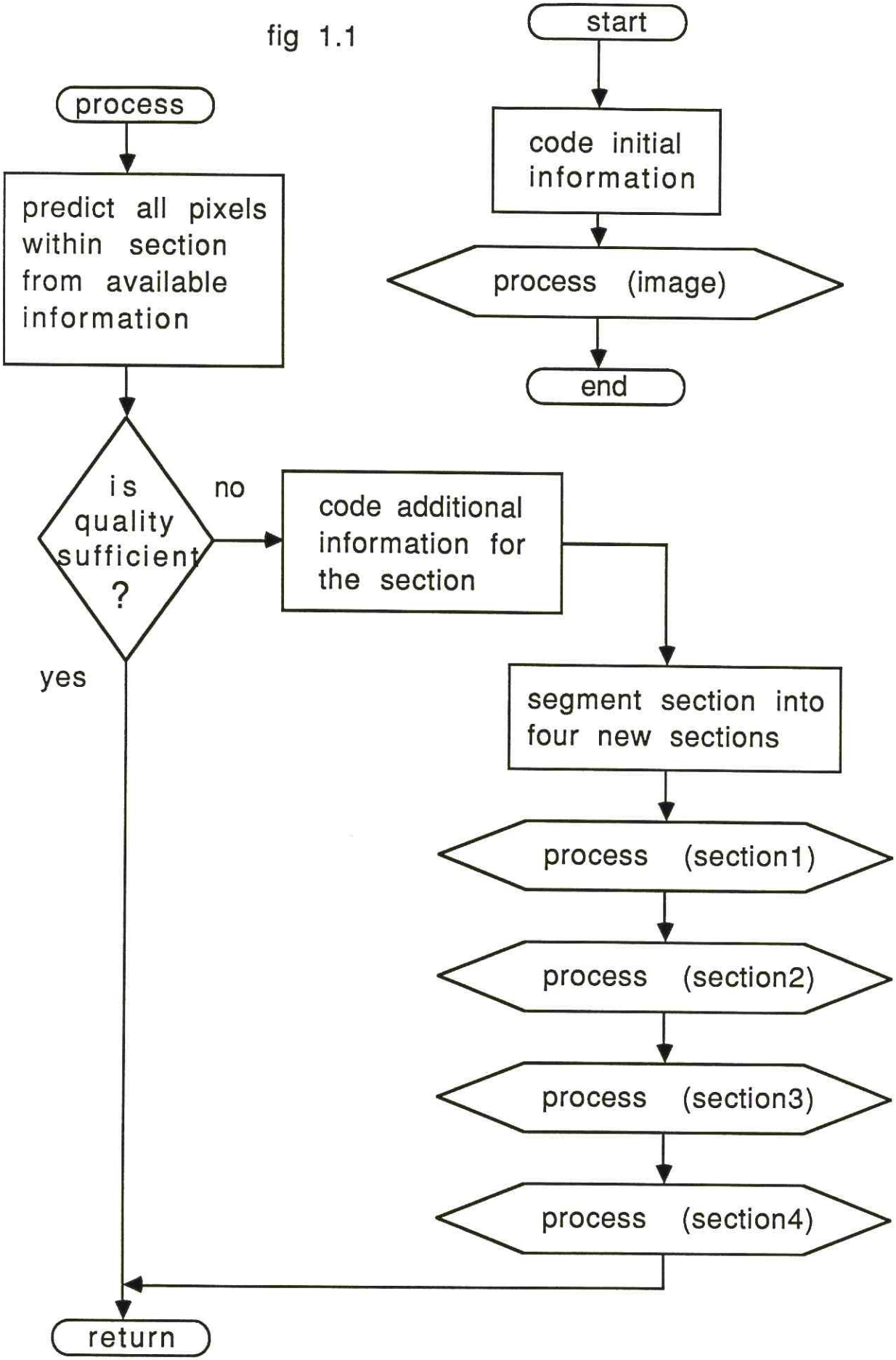
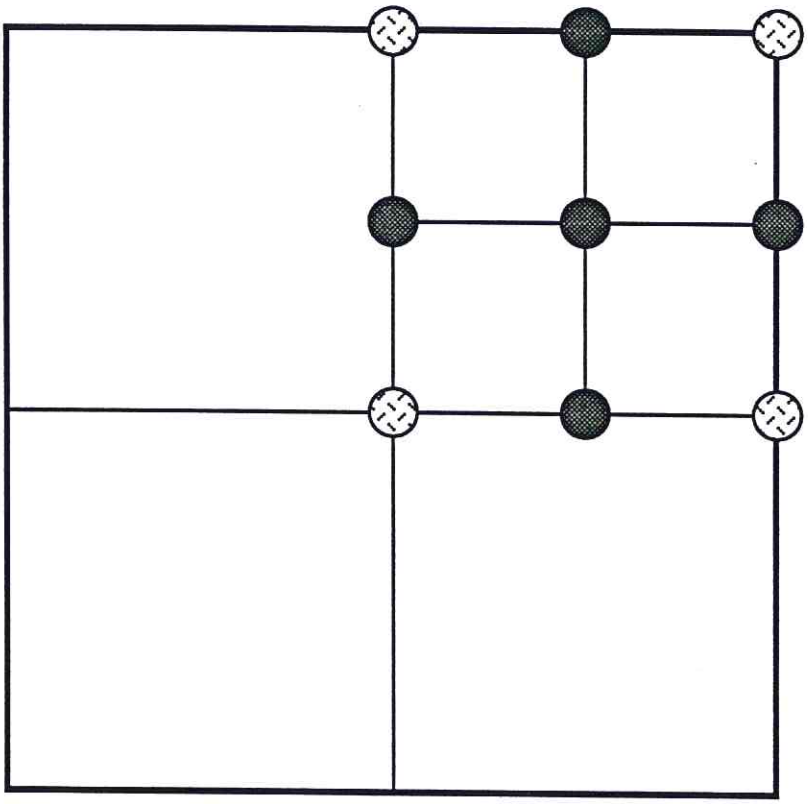

start
code initial information
process (image)
end
process
predict all pixels within section from available information
is quality sufficient ?
no
code additional information for the section
yes
segment section into four new sections
process (section1)
process (section2)
process (section3)
process (section4)
return

fig 1.2



- new pixels

- corner pixels

fig 1.3

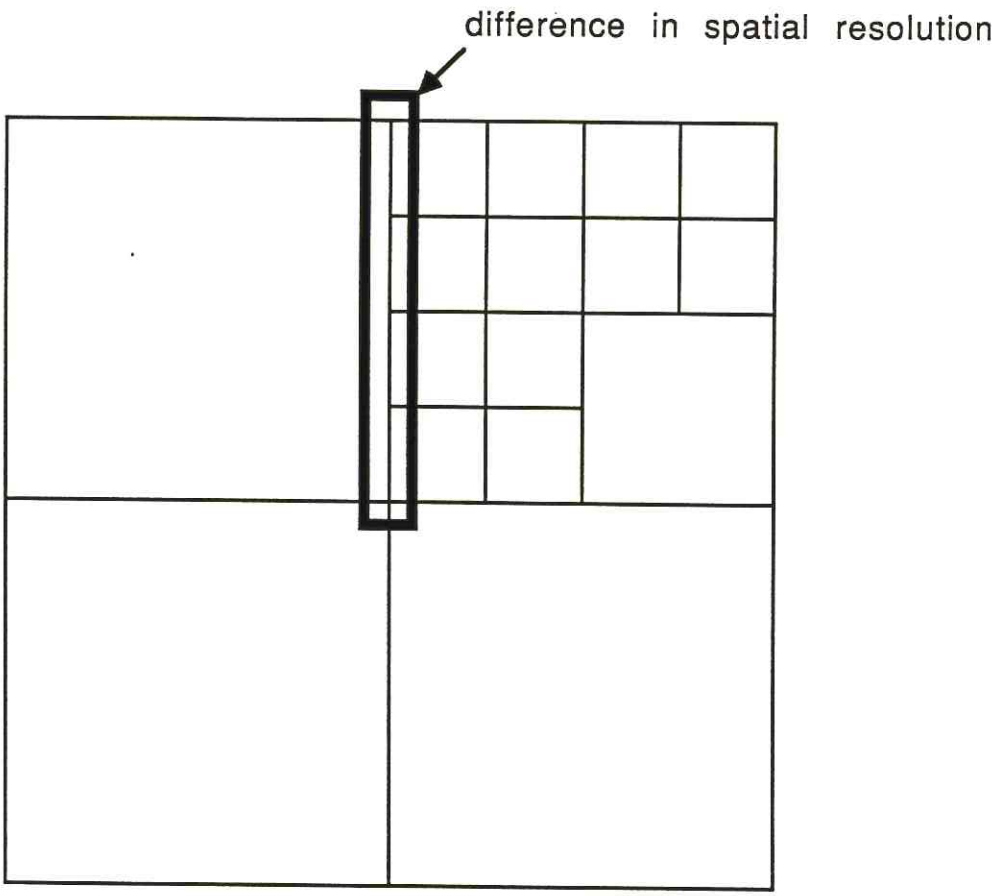


fig 2.1

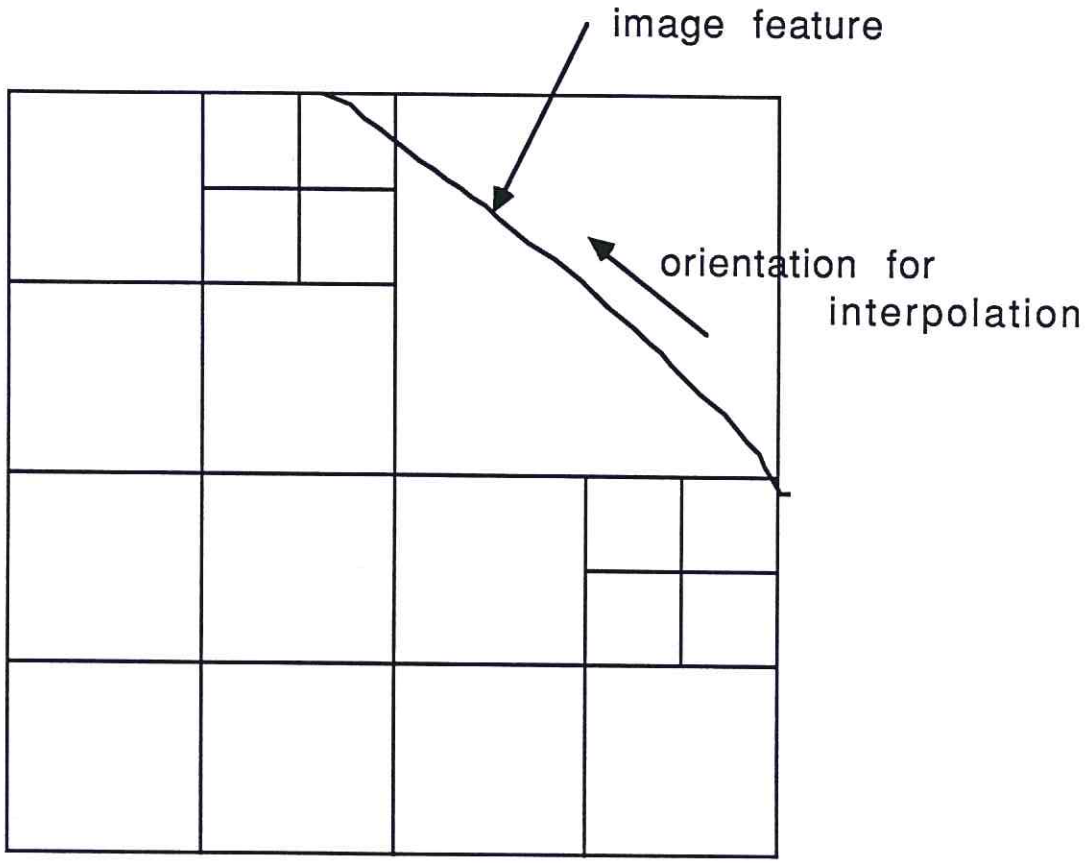
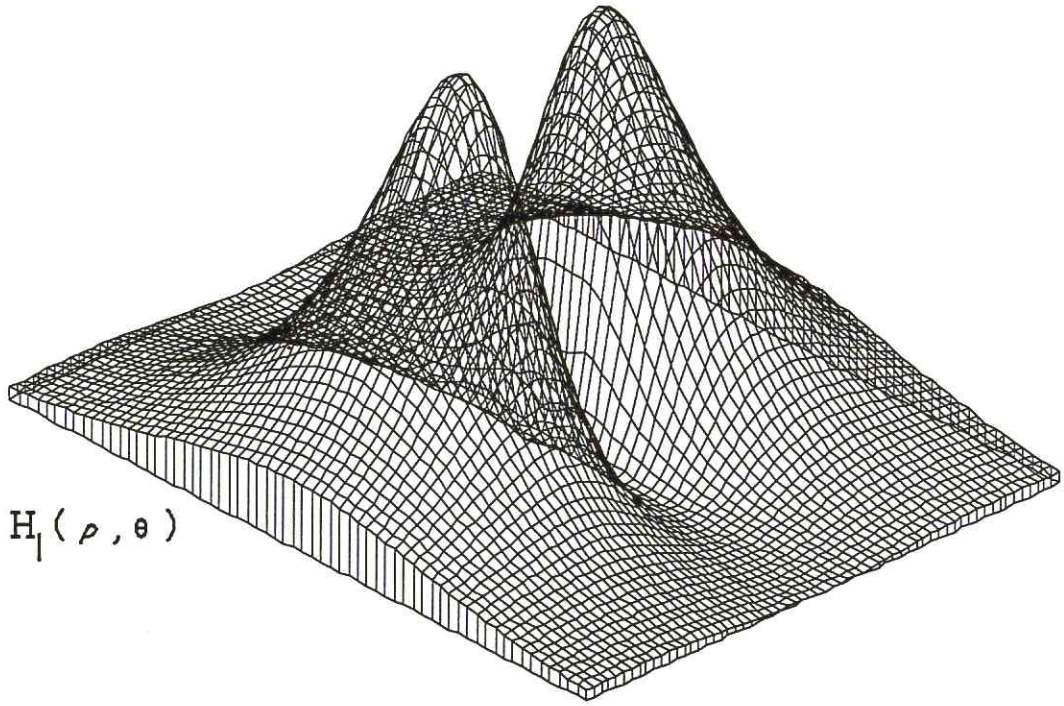
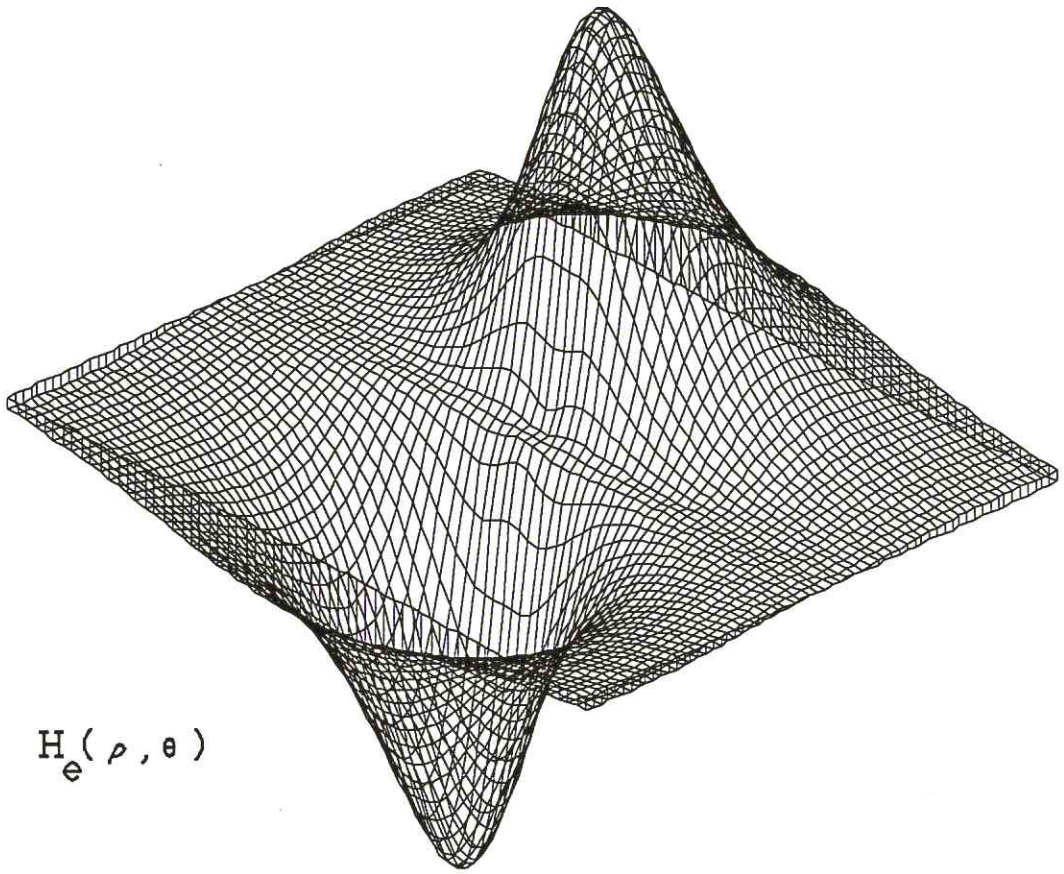


fig 2.2



$H_l(\rho, \theta)$



$H_e(\rho, \theta)$

fig 2.3

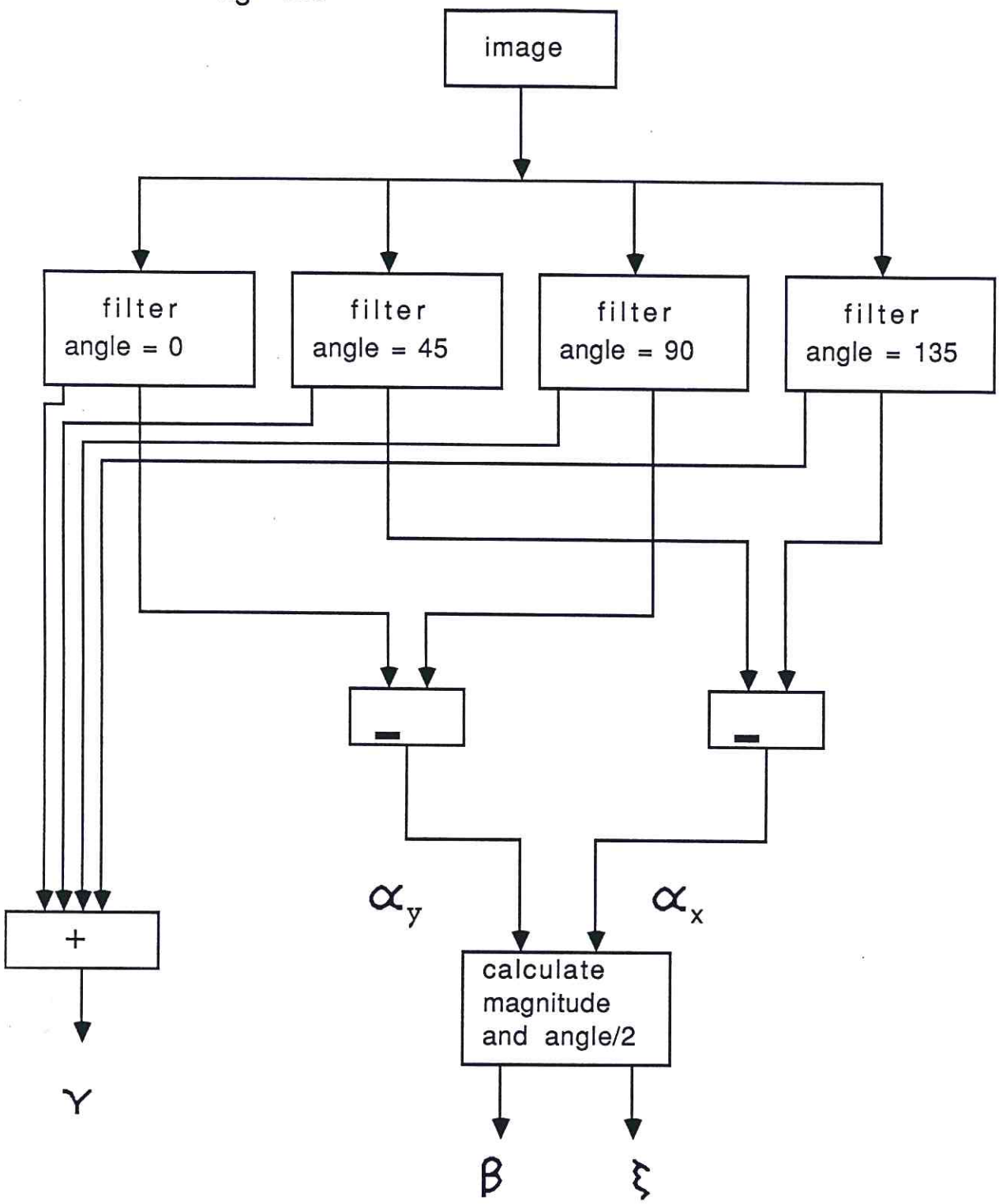
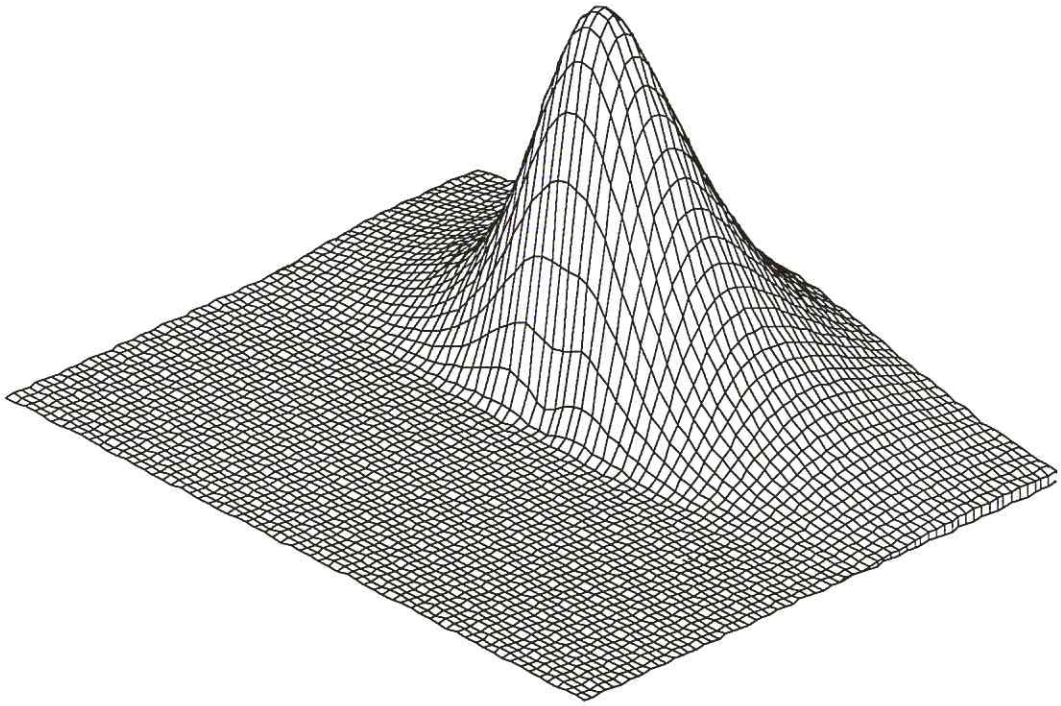


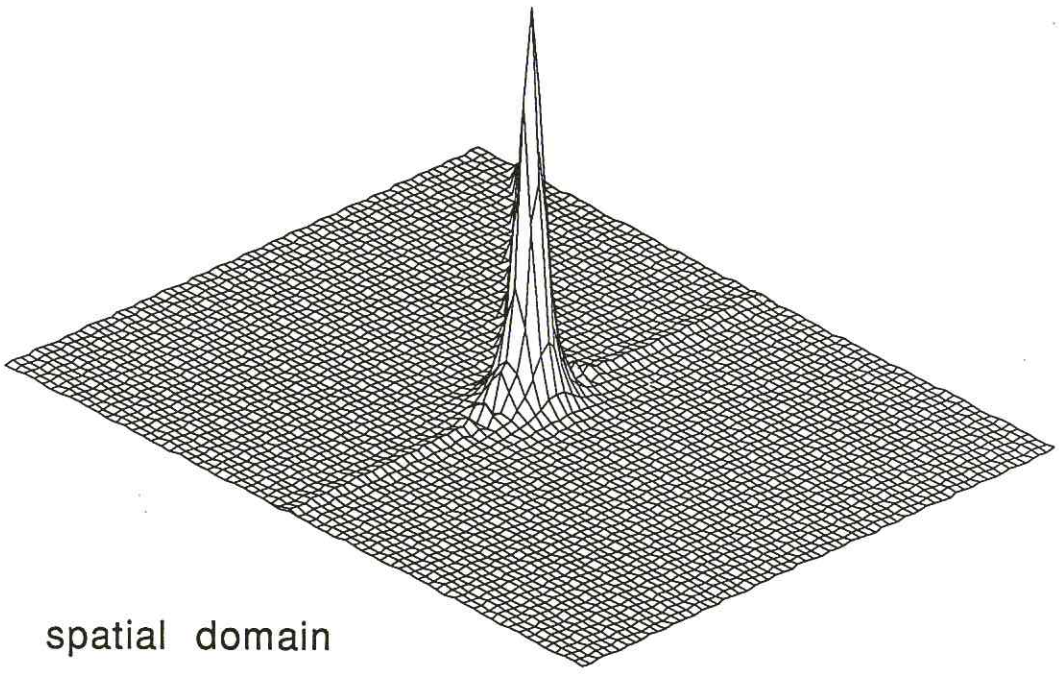
image
filter
angle = 0
filter
angle = 45
filter
angle = 90
filter
angle = 135
−
−
+
α_y
α_x
calculate
magnitude
and angle/2
γ
β
ξ

fig 2.4

filter angle = 0



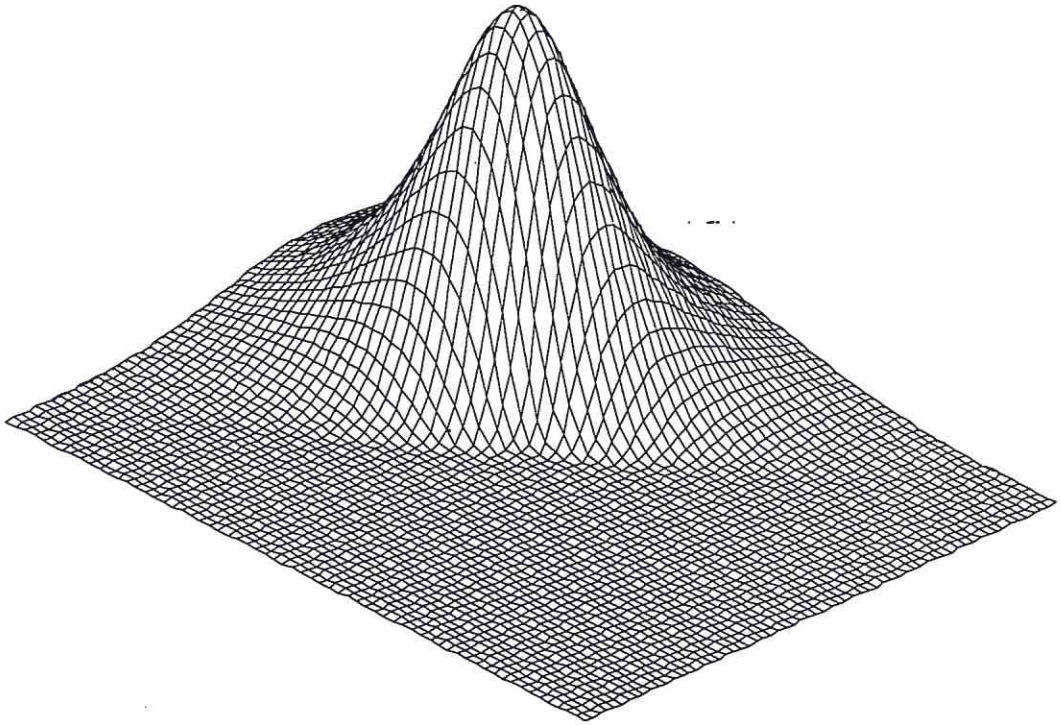
spatial-frequency domain



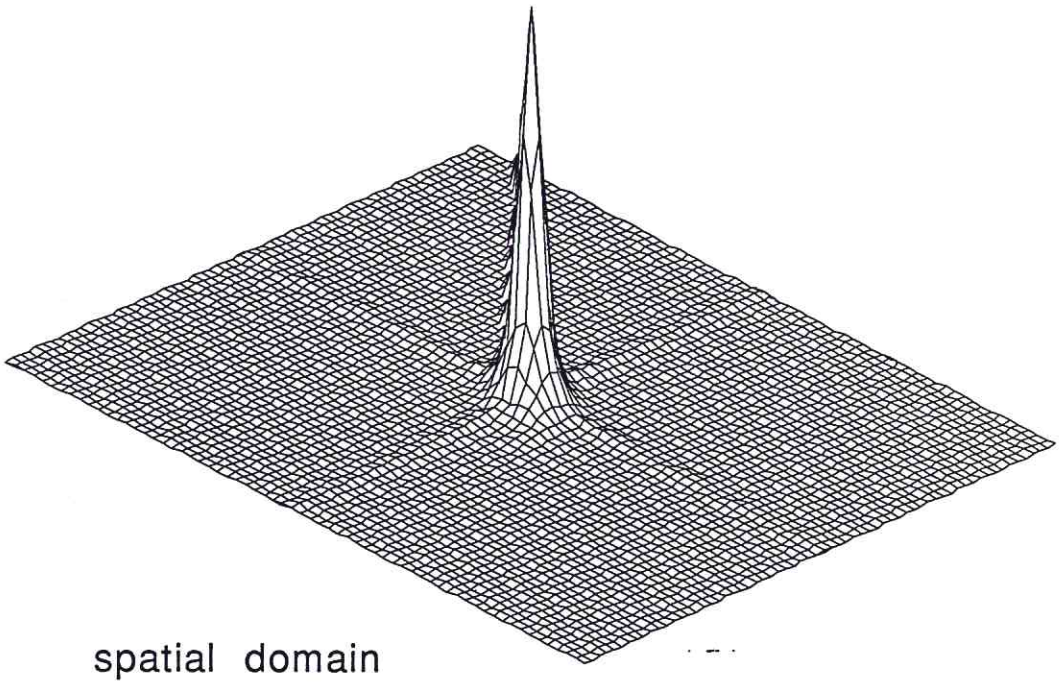
spatial domain

fig 2.5

filter angle = 45



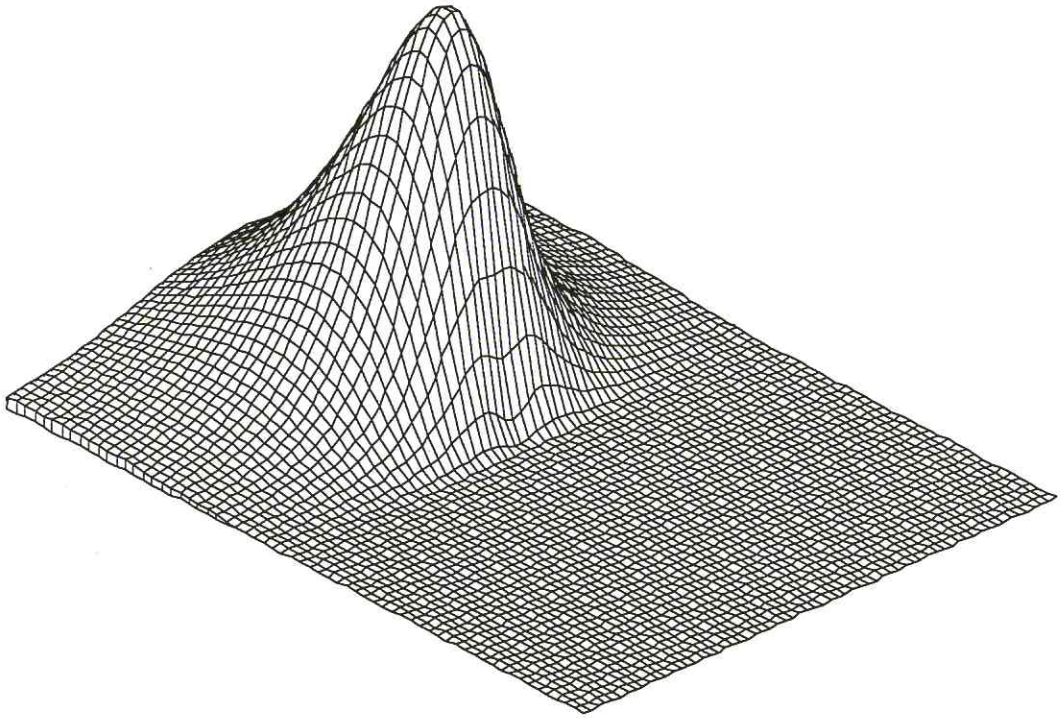
spatial-frequency domain



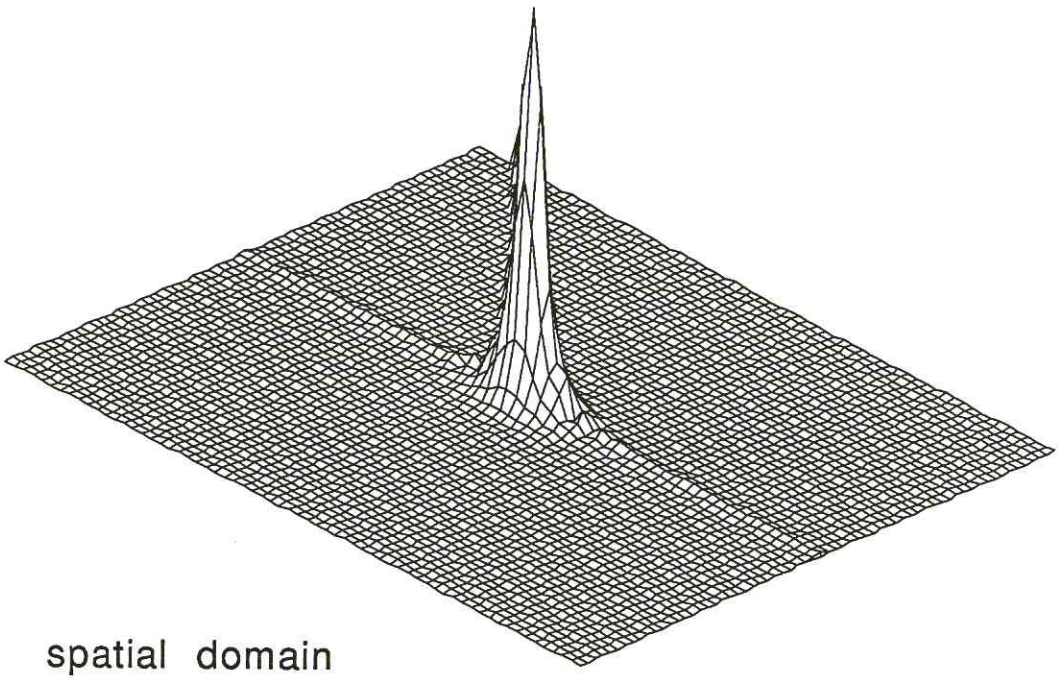
spatial domain

fig 2.6

filter angle = 90



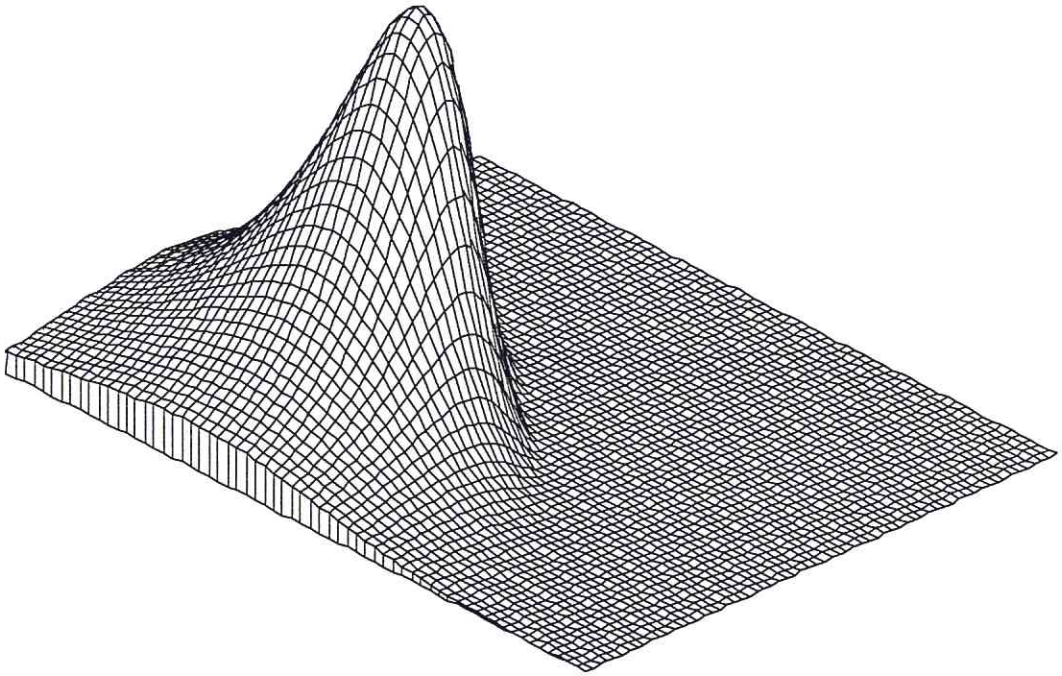
spatial-frequency domain



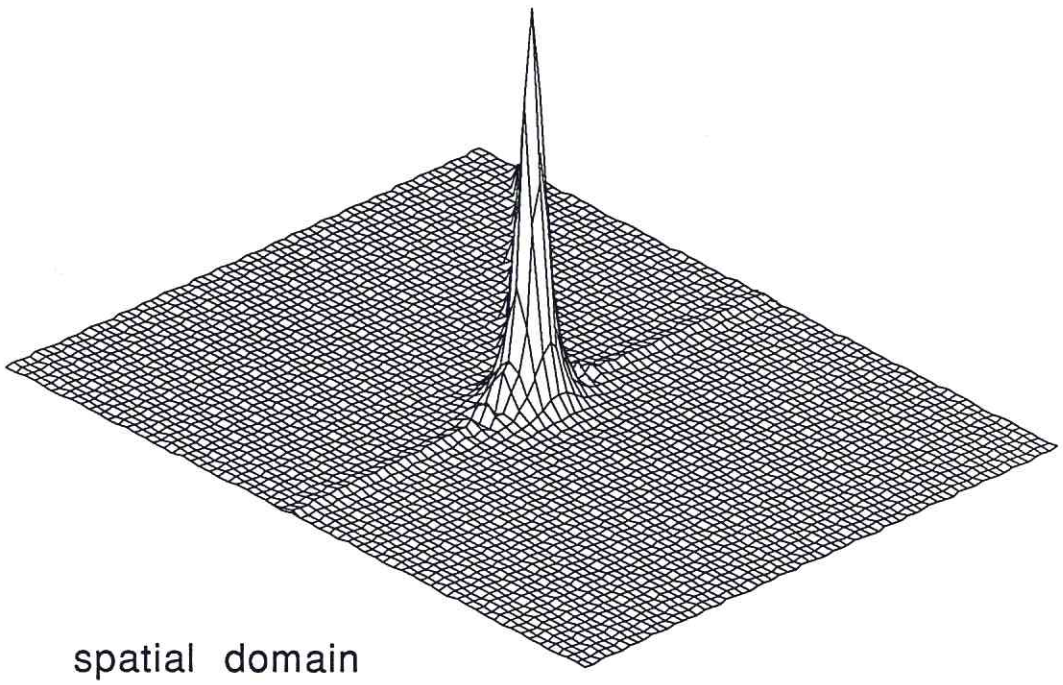
spatial domain

fig 2.7

filter angle = 135



spatial-frequency domain



spatial domain

fig 3.1

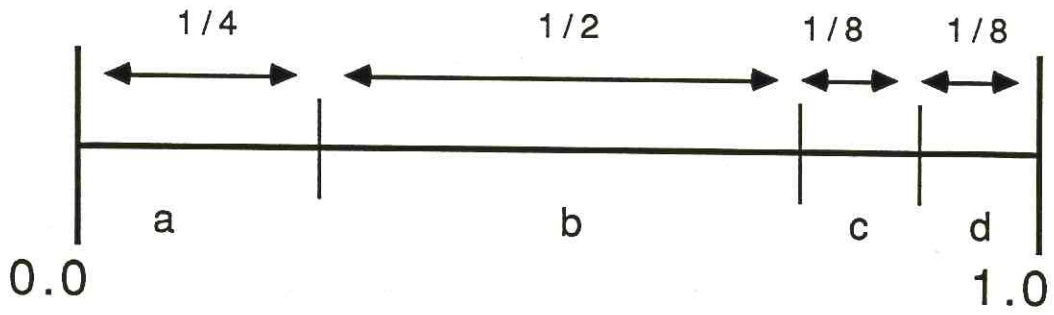


fig 3.2

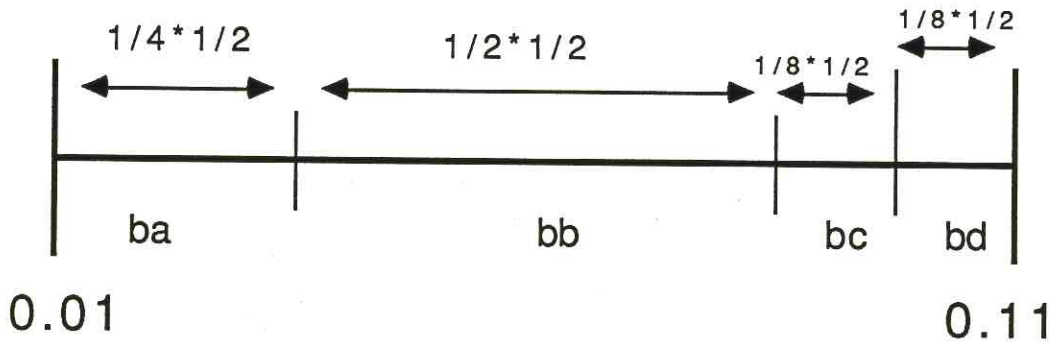


fig 3.3

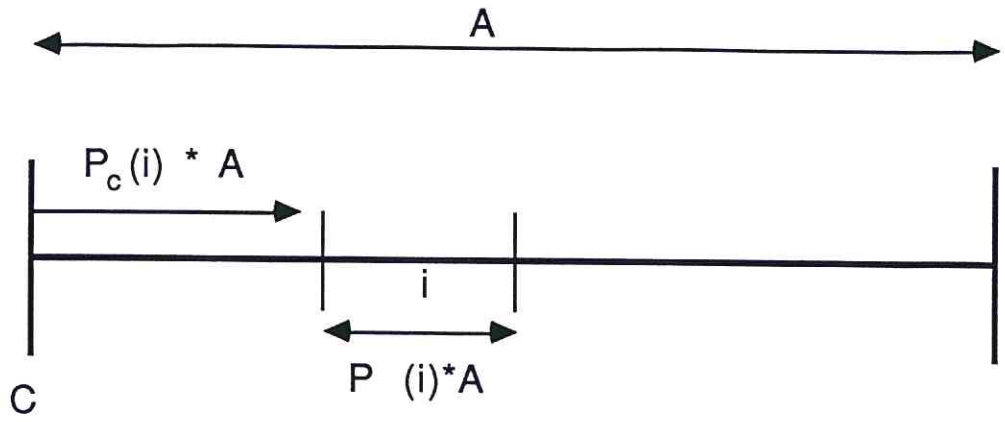


fig 3.4

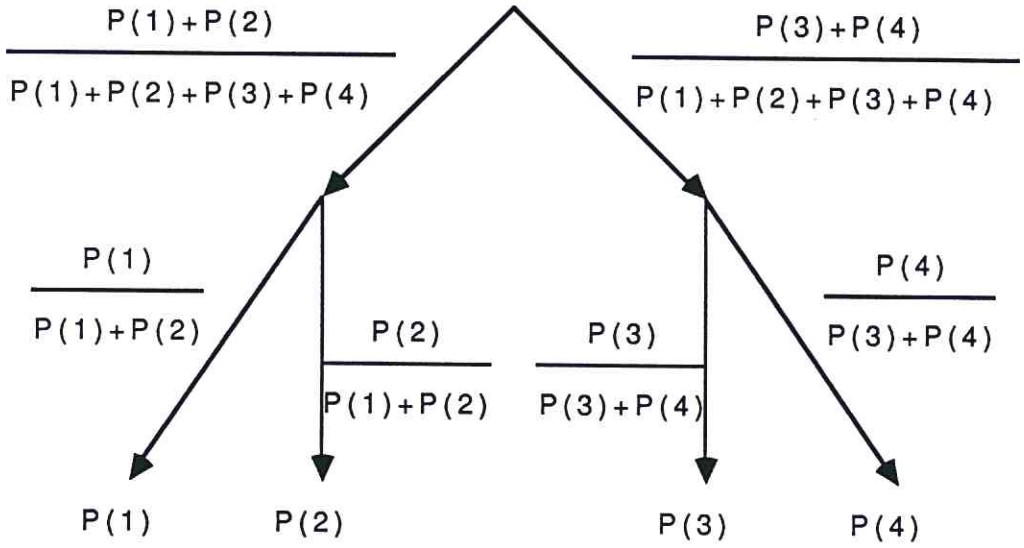


fig 3.5

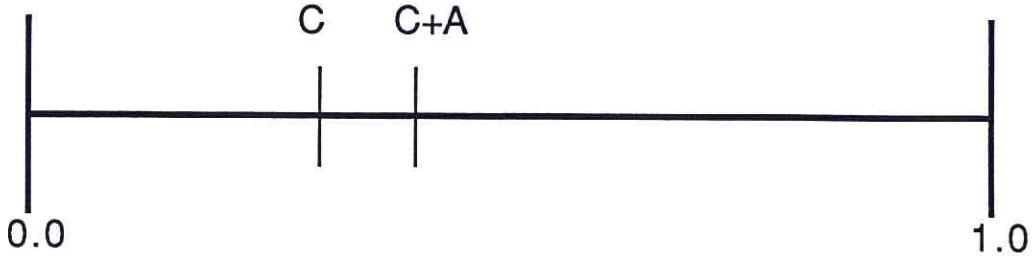


fig 3.6

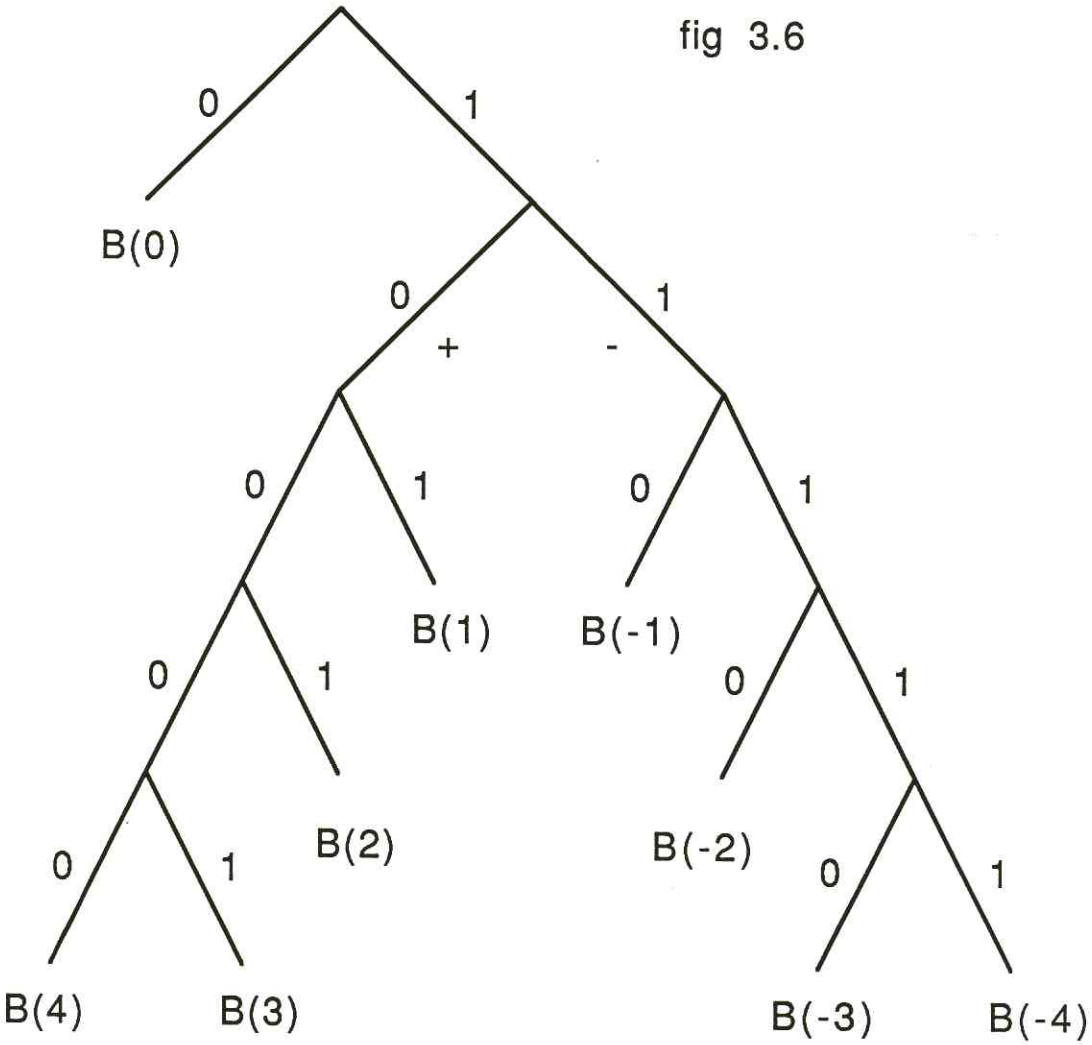


fig 3.7

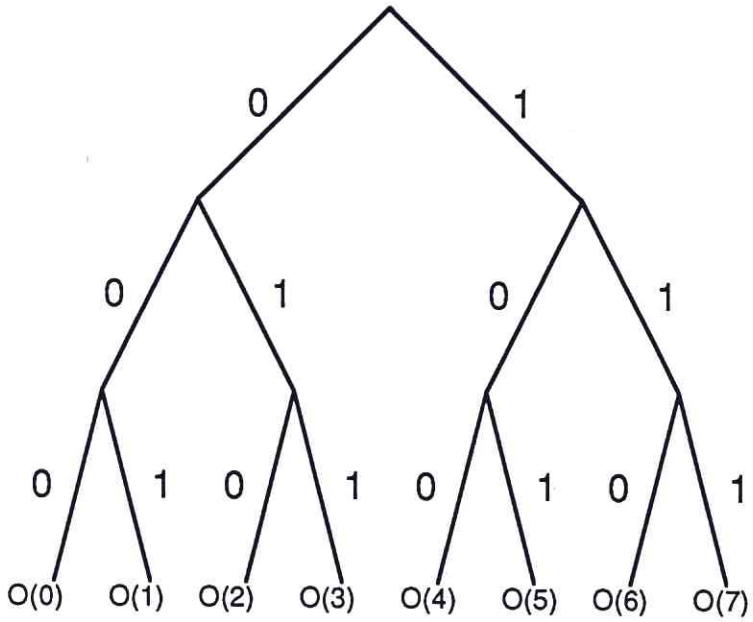


photo 6.1

BARBARA

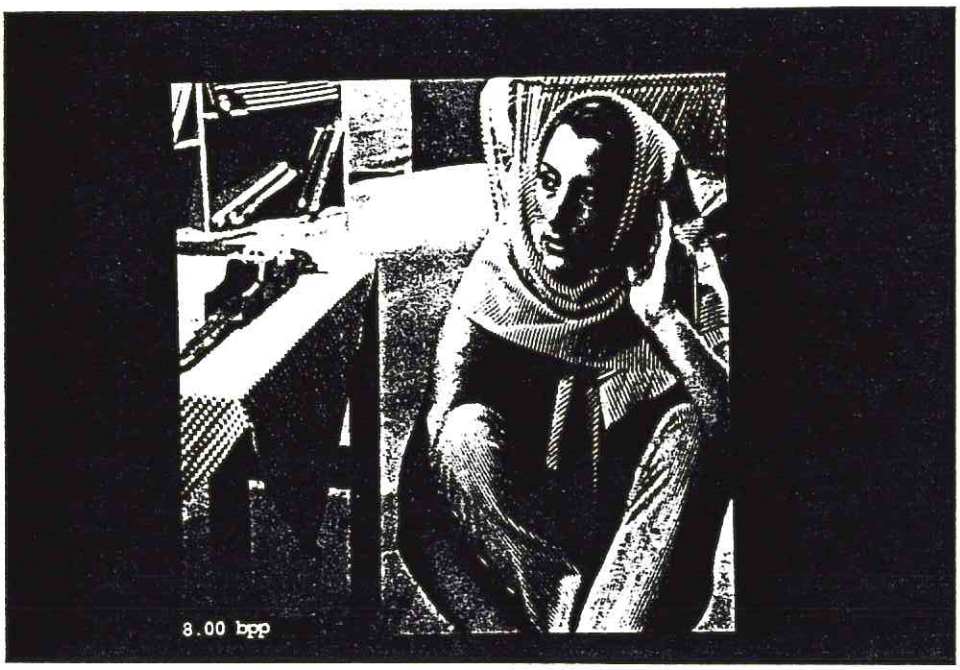


photo 6.2

BOATS

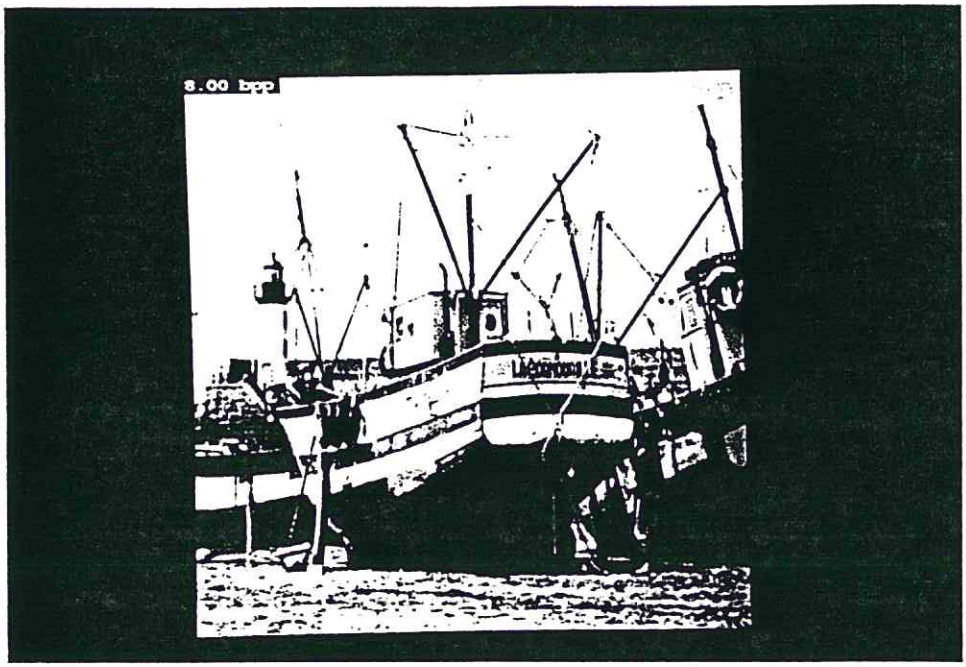
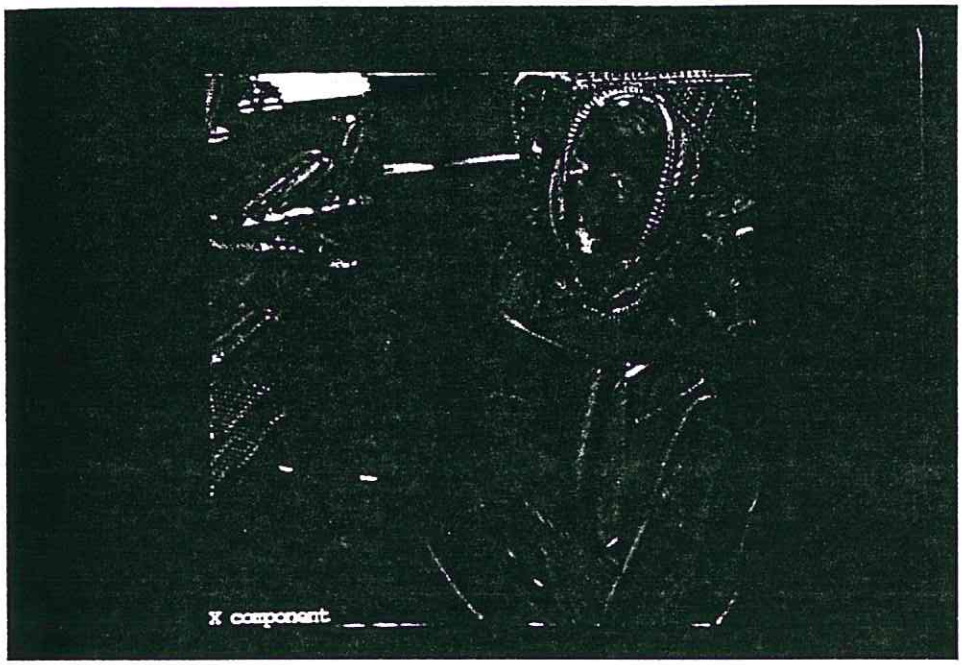


photo 6.3

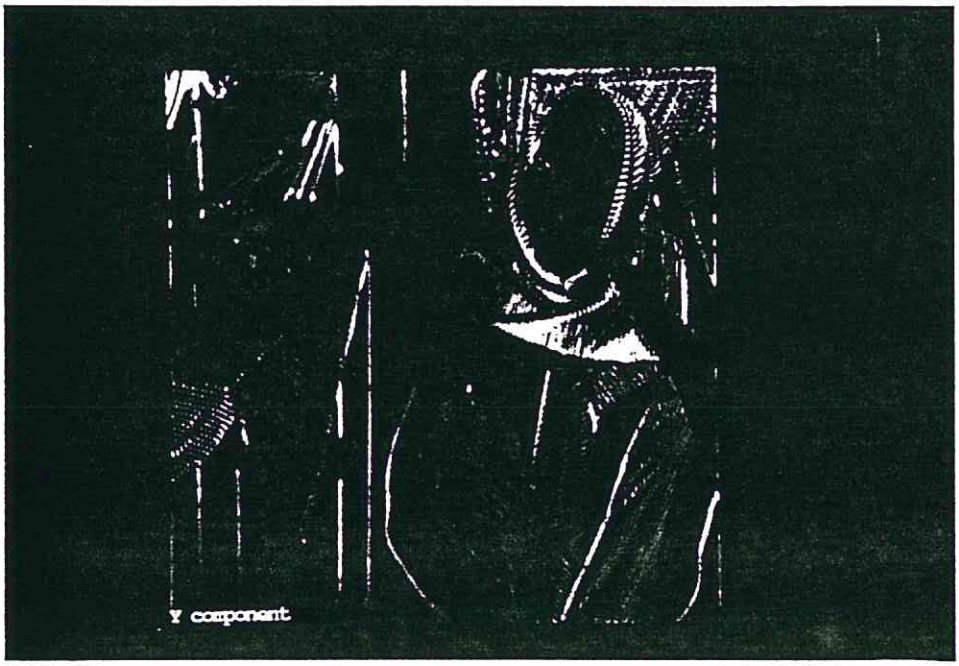
GIRL

photo 6.4

BARBARA



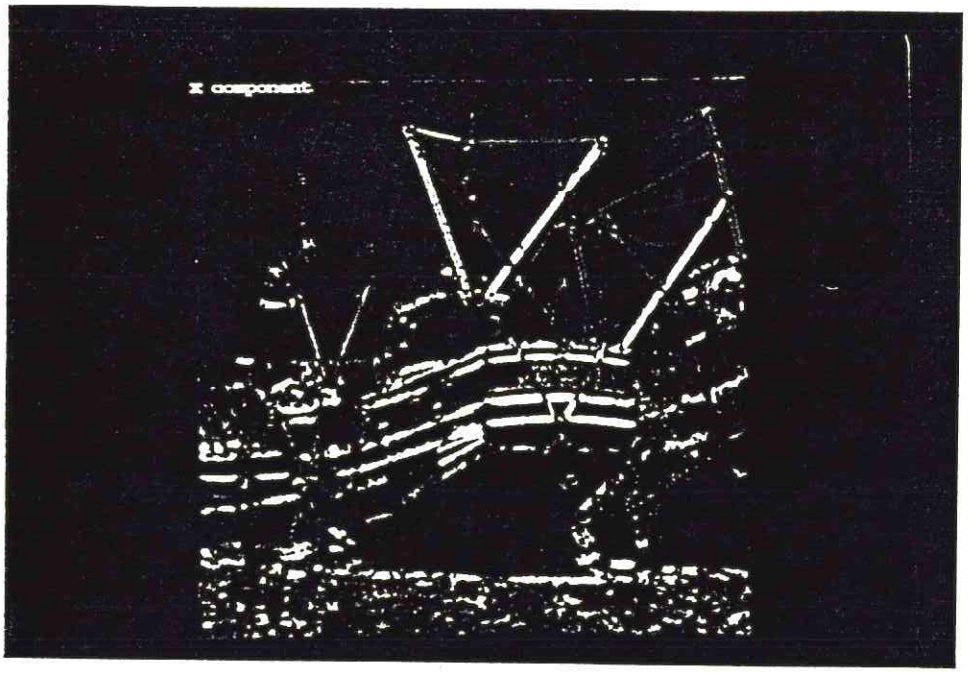
X-component



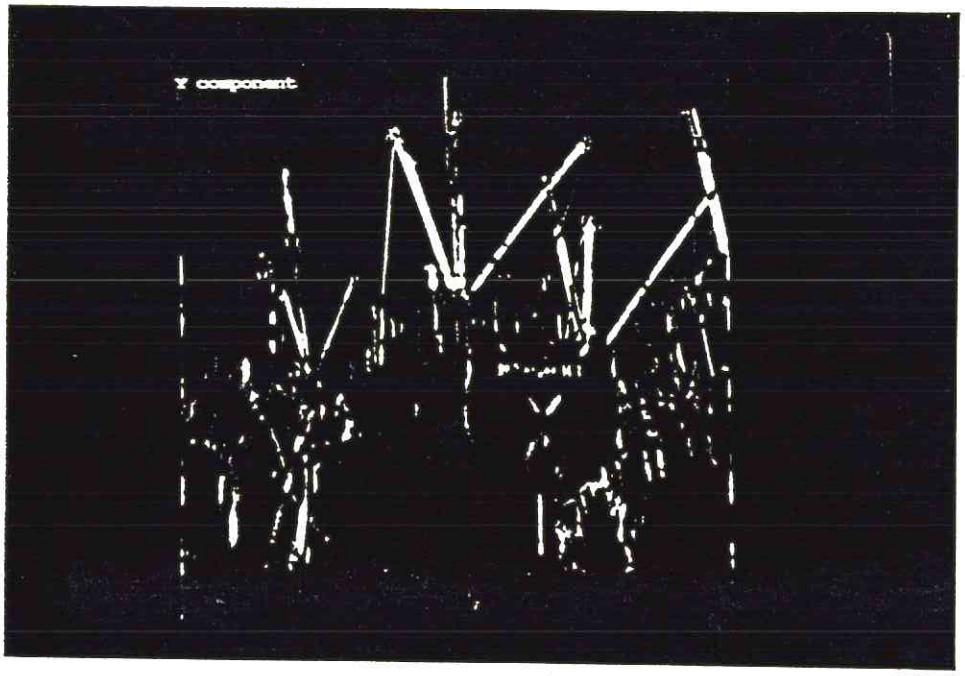
Y-component

photo 6.5

BOATS



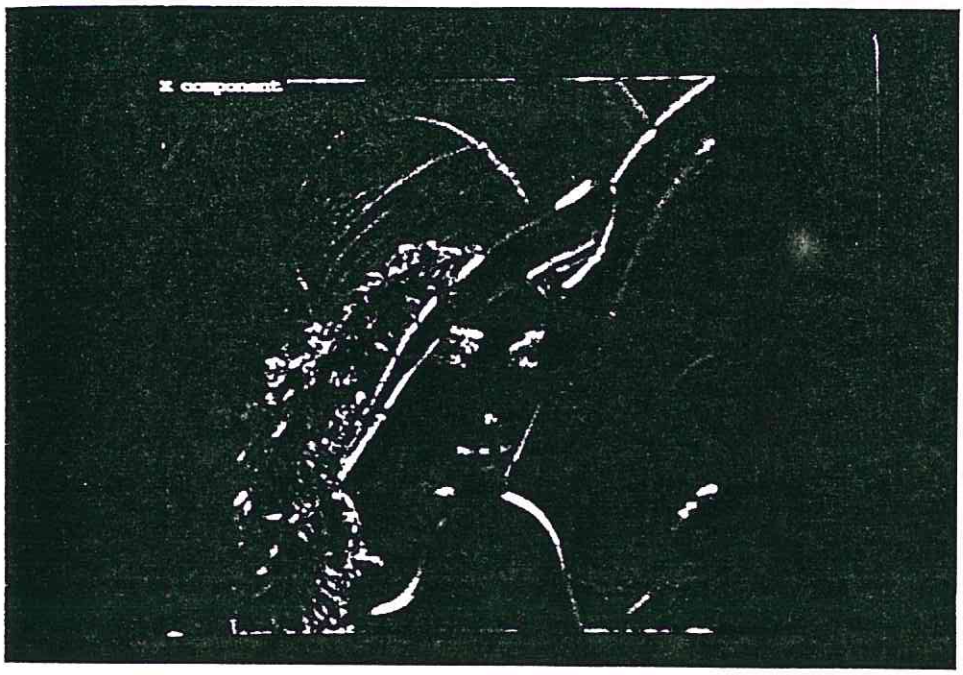
X-component



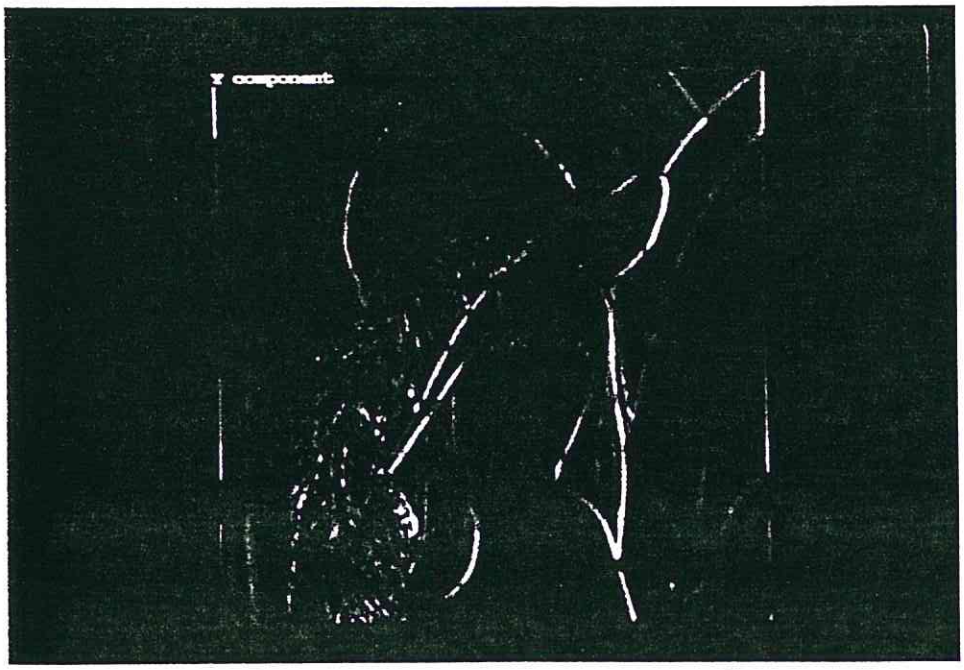
Y-component

photo 6.6

GIRL



X-component



Y-component

photo 6.7

BARBARA at 0.65 bpp

photo 6.8

BARBARA at 0.36 bpp

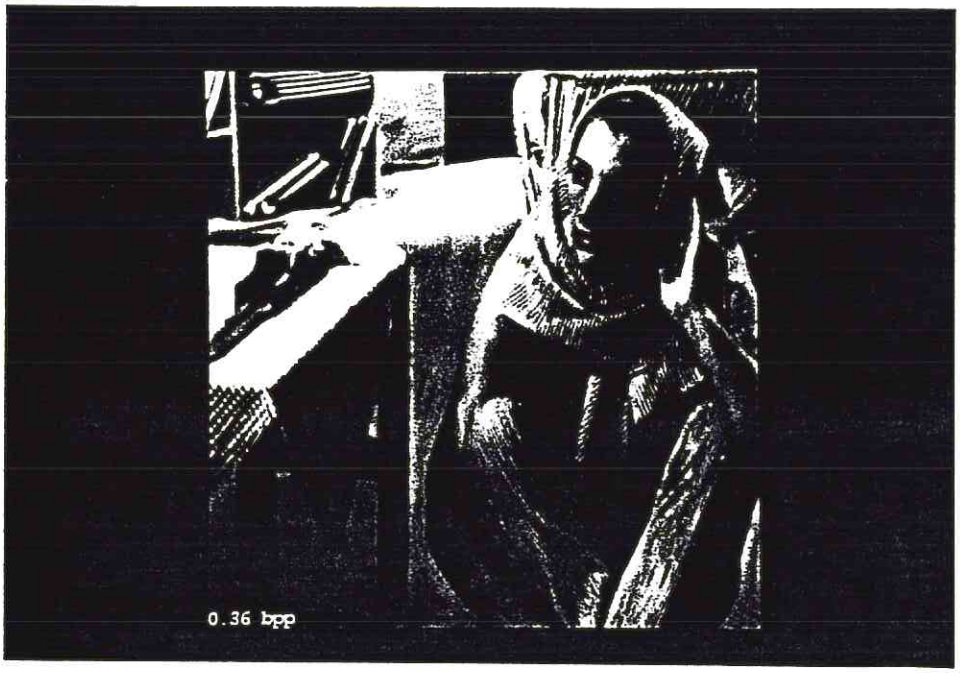


photo 6.9

BARBARA at 0.23 bpp

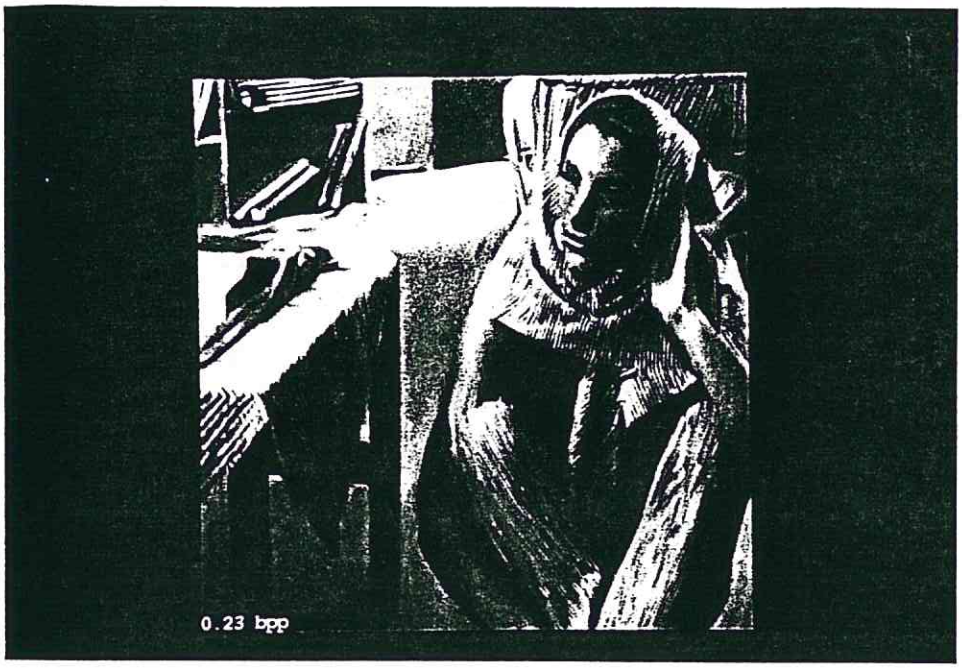


photo 6.10

BARBARA at 0.16 bpp

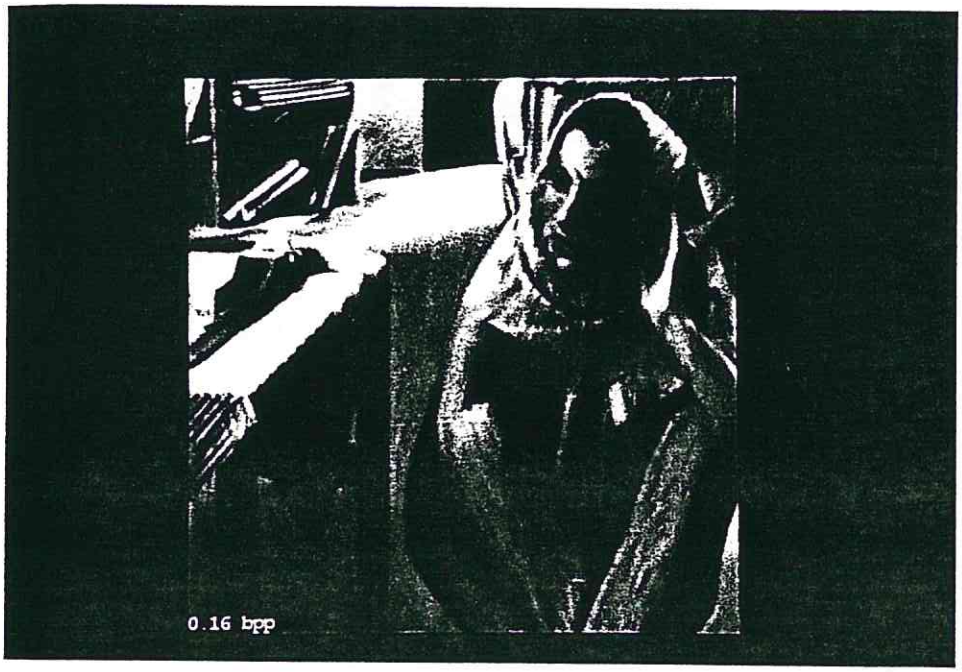


photo 6.11
BOATS at 0.43 bpp

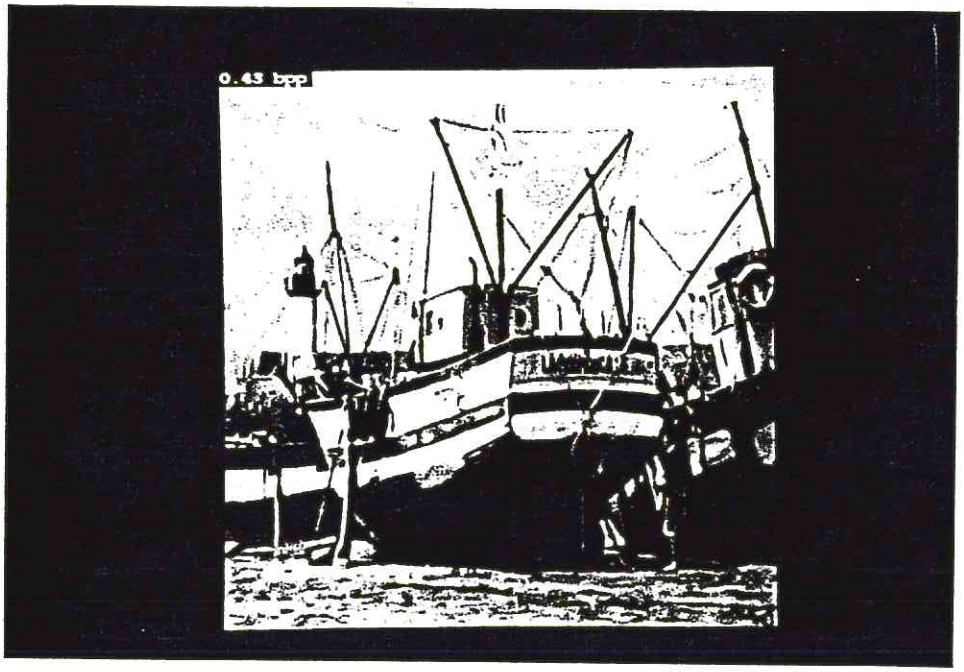


photo 6.12
BOATS at 0.31 bpp

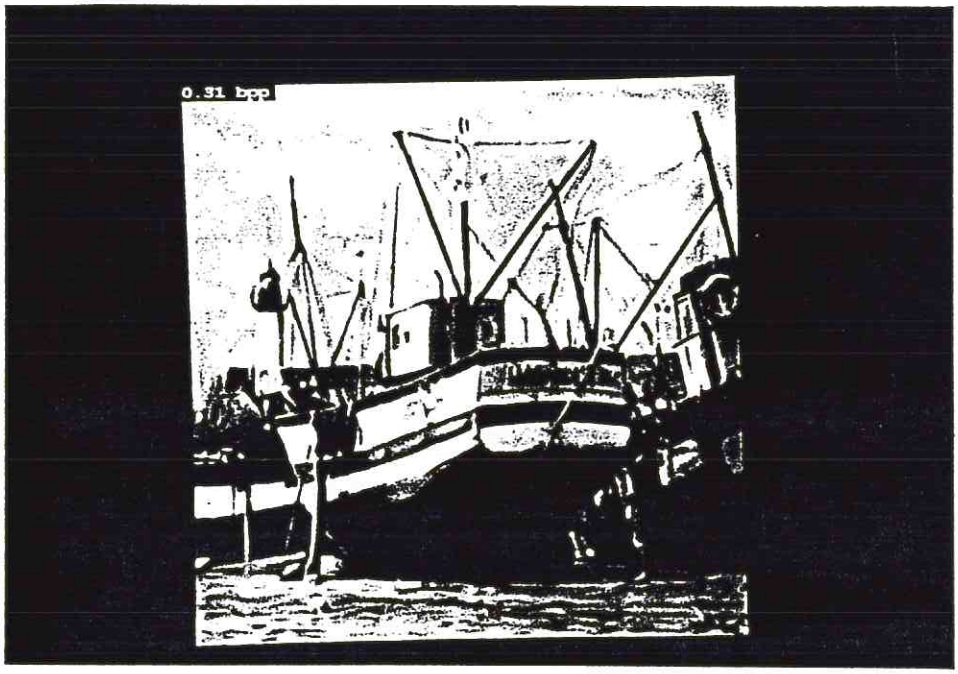


photo 6.13
BOATS at 0.26 bpp

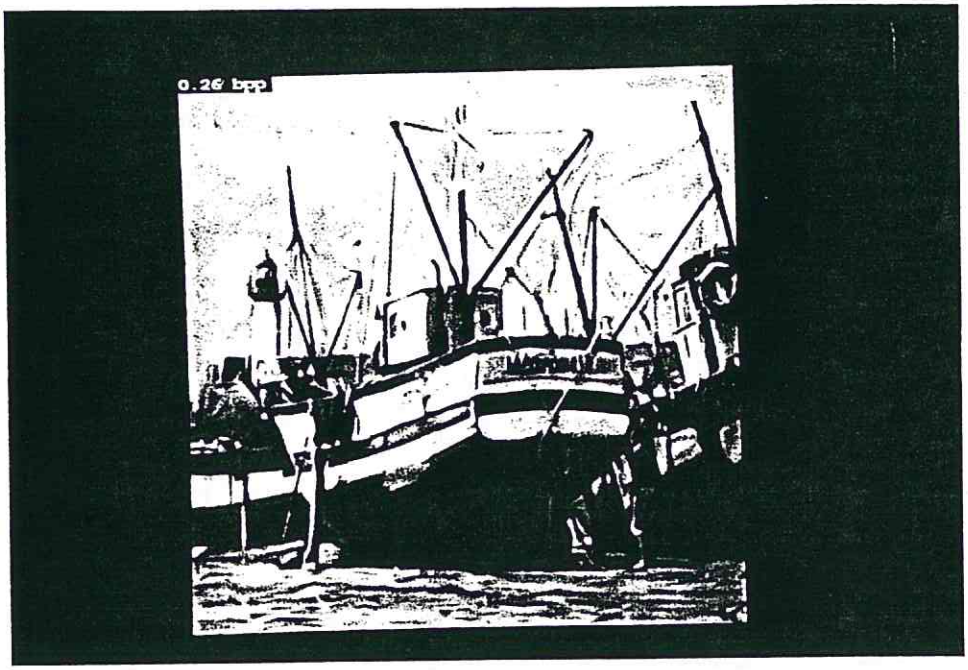


photo 6.14
BOATS at 0.13 bpp

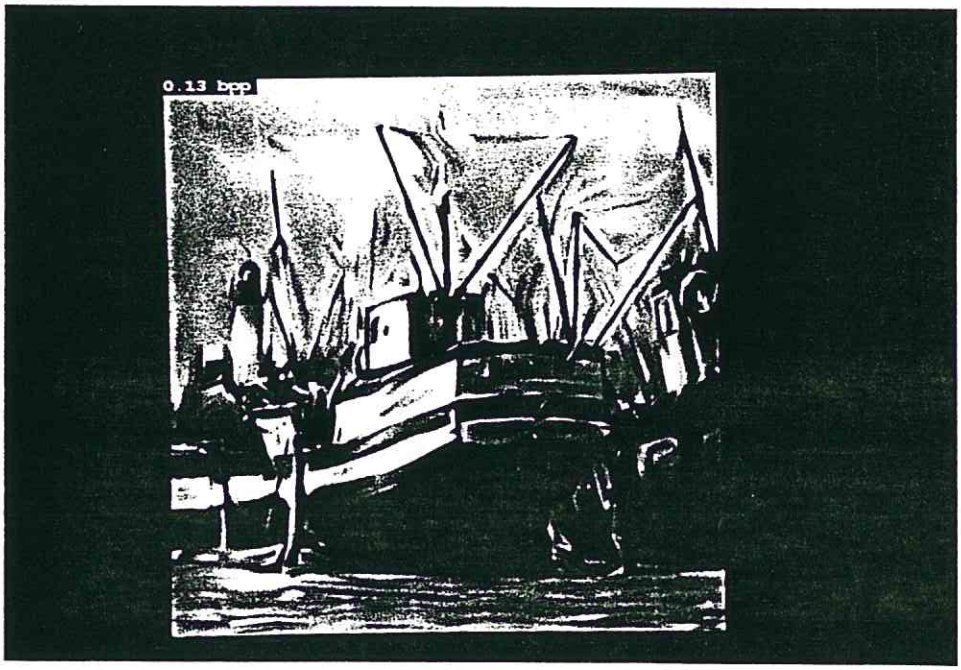


photo 6.15
GIRL at 0.29 bpp

photo 6.16
GIRL at 0.20 bpp

photo 6.17
GIRL at 0.12 bpp

photo 6.18
GIRL at 0.08 bpp