Signal and Image Compression

Assume that we start with digital or analog signal.

Compression consists (roughly speaking) of three steps.

① **Representation of data**
- samples
- transform coefficients

② **Quantization**

Represent data with fixed number of bits

e.g., real-valued samples quantized into integers 0, ..., 255 and represented with 8 bits each

③ **Coding**

More efficiently represent (lossless) or approximate (lossy) information

i.e., instead of 8 bit codes for each sample, use a more clever coding scheme.

State of the Art

Speech Compression for Communications

① Telephone speech is sampled at 8kHz
Requirements: intelligibility, limited quality

② Quantize each sample to 8 bit number
$\Longrightarrow$ 64 Kb/s

③ Model-based coding using linear predictive codes. Speech modeled by linear filters excited by a white noise or pulse train. Parameters of filters are coded instead of speech samples.

Audio (Speech and Music]

① Sampled at 44.1kHz (CD quality)
high fidelity

② Quantized $\Longrightarrow$ 706 kb/s

③ Coding Perceptual coders based on local time-freq representation (local cosine basis) give best results.
CD quality @ $\approx$ 128 kb/s

Images

1. Pixelized image, 8 bits/pixel

2. Transform image (DCT jpeg or DWT) and quantize coefficients, 8 bits/pixel, very little loss due to quantization.

3. Entropy coding (Huffman codes, prefix codes) $\Longrightarrow$ lossless compression (1-2 bits/pixel)

   Nonlinear signal adapted approximations + coding $\Longrightarrow$ lossy compression (0.5 bits/pixel or less)

Video

1. Pixelized video stream
   Teleconferencing $\approx$ 20 Mb/s

2. Motion compensation/prediction frame-to-frame + transform

3. Entropy code and approximate $\Longrightarrow$ decent quality @ 128 kb/s.

Transform Coding

1. Compute transform of signal (DCT, DWT)
2. Quantize coefficients
3. Code (lossless or lossy)

Compression in Orthonormal Bases

Let $B = \{g_m\}_{m=0}^{N-1}$ be an o.n. basis

We model the class of signals to be coded as a random vector $Y(n)$.

$$Y = \sum_{m=0}^{N-1} A(m)\, g_m \quad , \quad A(m) = \langle Y, g_m \rangle$$

$A(m)$ are coefficients modeled as random variables. We code the "centered" coeff.

$$A(m) - E[A(m)]$$

and store the mean $E[A(m)]$.

In many cases $A(m)$ is modeled as zero-mean (e.g., wavelet coefficients or higher freq cosine coefficients).

Quantization

Each $A(m)$ is quantized to a variable $\tilde{A}(m)$.

Scalar quantizer:

$$A(m) \longmapsto Q[A(m)] = \tilde{A}(m)$$

Vector quantizer:

$$A(m), \ldots, A(m+l) \longmapsto Q[A(m), \ldots, A(m+l)] = \tilde{A}(m), \ldots, \tilde{A}(m+l)$$

Scalar Quantizers:
- simple to analiyze and implement
- treat coefficients independently

Vector Quantizers:
- more complex to analyze and implement
- takes advantages of dependencies/correlations between coefficients to provide better approximation.

$$\tilde{Y} = \sum_{m=0}^{N-1} \tilde{A}(m)\, g_m$$

Distortion Rate :

$$D = E\left[\| Y - \tilde{Y} \|^2\right] = \text{distortion}$$

$$= \sum_{m=0}^{N-1} E\left[\left(A(m) - \tilde{A}(m)\right)^2\right] \quad , \text{ since transform is orthogonal (Parseval's)}$$

Suppose that R_m bits are allocated to encoded the quantized bit $\tilde{A}(m)$.

For a given R_m, the quantizer is designed to minimize $E\left[\left(A(m) - \tilde{A}(m)\right)^2\right]$.

The distortion is a function of the total bit rate, and so we define

$D(R)$ = distortion at rate R
"distortion rate"

$$R = \sum_{m=0}^{N-1} R_m$$

For a given R, we would like to adjust the bit allocation to minimize $D(R)$.

How many bits R_m should we use to code $\tilde{A}(m)$?

Quantization

For the purposes of this discussion we focus on uniform quantizers

$$\tilde{A}(m) = Q[A(m)] = \begin{cases} \left\lfloor \frac{A(m)}{W} + 1 \right\rfloor, & A(m) > \frac{W}{2} \\ 0 & -\frac{W}{2} \leq A(m) \leq \frac{W}{2} \\ \left\lceil \frac{A(m)}{W} - 1 \right\rceil, & A(m) < -\frac{W}{2} \end{cases}$$

This is the simplest choice, but non-uniform quantizers can perform better. Ideally we want to quantize so that

$$E\left[\left| A(m) - Q[A(m)] \right|^2 \right]$$

is minimized. This can be done if $p(A)$ is known, but we won't concern ourselves with this issue here.

Entropy Coding

Let A be a random variable that takes values from a finite set of symbols $\{a_k\}_{k=1}^{K}$.

e.g., $\{a_k\}_{k=1}^{K} = \{0, \ldots, 255\}$

If $\log_2 K$ is an integer, then the K mbols can be coded with binary code of $\log_2 K$ bits each.

e . $K = 256 \Longrightarrow 8$ bit codes

Is this the most efficient scheme?

No. Ideally we want to assign shorter codes to frequently occuring symbols and longer codes to infrequent or unlikely symbols.

Prefix Code

Suppose there were four symbols $a_1, \ldots, a_4$ with probabilities

$$p(a_1) > p(a_2) > p(a_3) > p(a_4)$$

then we want to assign a short code to a_1 and a longer code to a_4 (instead of using 2bits each)

Std Codes

symbolc	code
a_1	00
a_2	01
a_3	10
a_4	11

Variable length codes

symbol	code
a_1	0
a_2	10
a_3	110
a_4	101

If $p(a_1) >> p(a_i)$, $i \neq 1$, then the variable length code is more compact since it uses just 1 bit to code the most probable symbol.

In order to be able to decode, we must insure that the code words are uniquely decodable from a bitstream.

<u>std codes</u>

$a_1 \; a_2 \; a_1 \; a_4 \; a_1$

00 01 00 11 00

just read and decode every two bits

<u>variable length codes</u>

$a_1 \; a_2 \; a_1 \; a_4 \; a_1$

0 10 0 101 0

<u>problem</u>: 1010 could be either $a_2 a_2$ or $a_4 a_1$

To insure $\overset{\text{unique}}{\vee}$ decodability, the variable length codes must satisfy the <u>prefix condition</u>:

No code word can be the prefix (beginning) of another codeword.

The prefix condition is violated above since a_2 is a prefix of a_4 code.

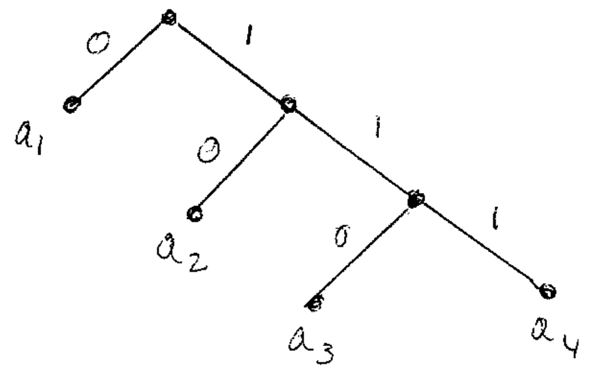
<u>Uniquely Decodable Codes</u>

symbol	code
a_1	0
a_2	10
a_3	110
a_4	111

$a_1 \; a_2 \; a_1 \; a_4 \; a_1$

0 10 0 111 0

Prefix Codes are characterized by binary trees:



We read a bit and follow the tree;

If bit = 0, then a_1
else if next bit = 0, then a_2
else if next, next bit = 0, then a_3
else a_4.

The average code length (assuming an iid sequence (memoryless source) with distribution of A) is

$$L = \sum_{k=1}^{K} l_k \, p(a_k)$$

where l_k is the code length for a_k.

__Theorem__ (Shannon)

$$L \geq H(A) = -\sum_{k=1}^{K} p(a_k) \log_2 p(a_k)$$

$H(A)$ is the entropy of the soure A.

ex. min entropy: $A = a_1$ w.p.1 $\left(\begin{array}{l} p(a_1) = 1 \\ p(a_i) = 0, i \neq 1 \end{array} \right)$

max entropy: $p(a_k) = \frac{1}{K}$

equally likely symbols

min entropy
$1 \cdot 0 = 0$

max entropy
$-\sum \frac{1}{k} \log_2 \frac{1}{k} = \log_2 k$

$$0 \leq H(A) \leq \log_2 K$$

__Huffman Codes__

The lower bound $H(A)$ can nearly be achieved by an optimal prefix code called a Huffman code.

Huffman Code Construction

Consider K symbols, enumerated according to decreasing probability

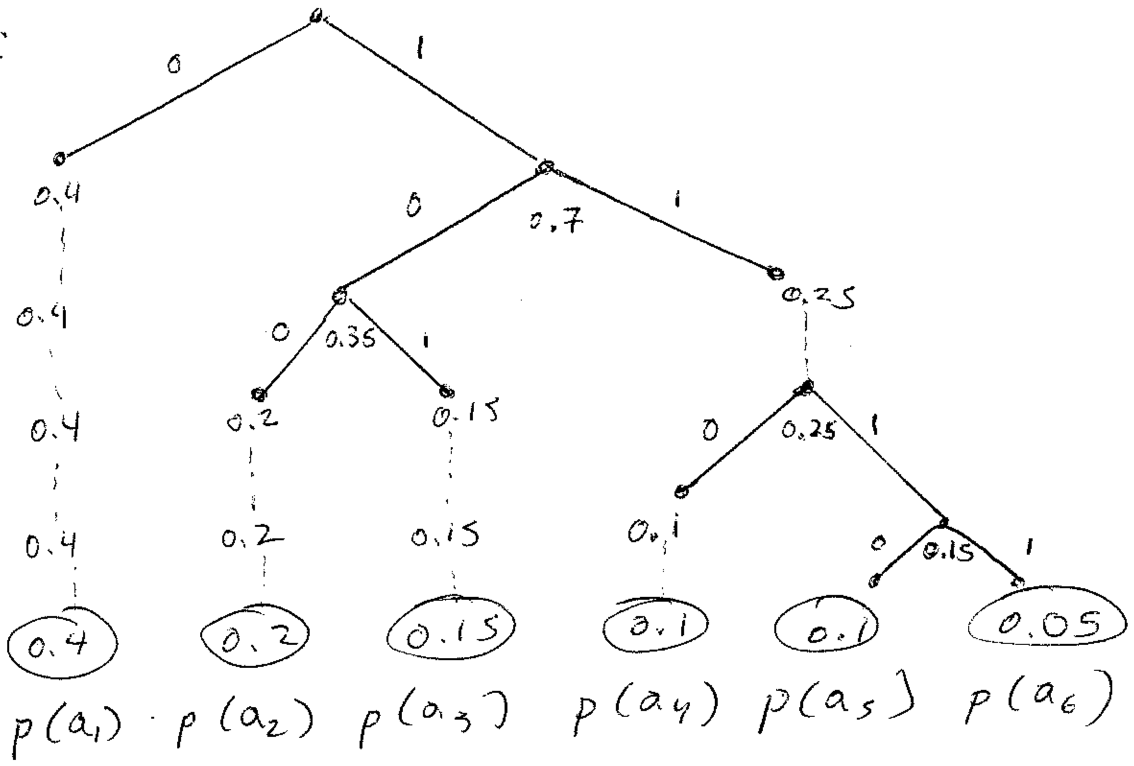
$$p(a_1) \geq p(a_2) \geq \cdots \geq p(a_K)$$

Aggregate two lowest probability symbols a_k and a_{k-1} into a single symbol $\{a_{k-1}, a_k\}$ with probability $P(\{a_{k-1}, a_k\}) = p(a_{k-1}) + p(a_k)$.

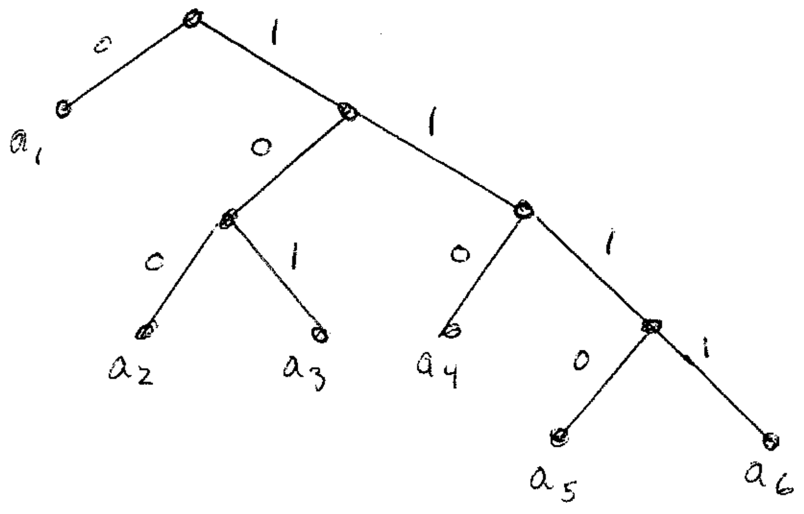
Then an optimal prefix code for K symbols is obtained by constructing the optimal coda for the $K-1$ symbols $\{a_1, a_2, \ldots, a_{k-2}, \{a_{k-1}, a_k\}\}$ and appending a 1 or 0 to the code word for the symbol $\{a_{k-1}, a_k\}$ to designate a_{k-1} or a_k resp.

Repeating this argument, again and again, gives us an algorithm for constructing the Huffman codes.

Ex.



Final Tree



$$H(A) = 2.2842 \text{ bits/symbol}$$

$$L = 0.4 \cdot 1 + 0.2 \cdot 3 + 0.15 \cdot 3 + 0.1 \cdot 3 + 0.1 \cdot 4 + 0.05 \cdot 4$$

$$= 2.35 \text{ bits/symbol}$$

Compression via the DWT

The Huffman codes are optimal under the assumption that the source is memoryless; i.e., the symbol sequence is i.i.d. (otherwise more sophisticated coding is required)

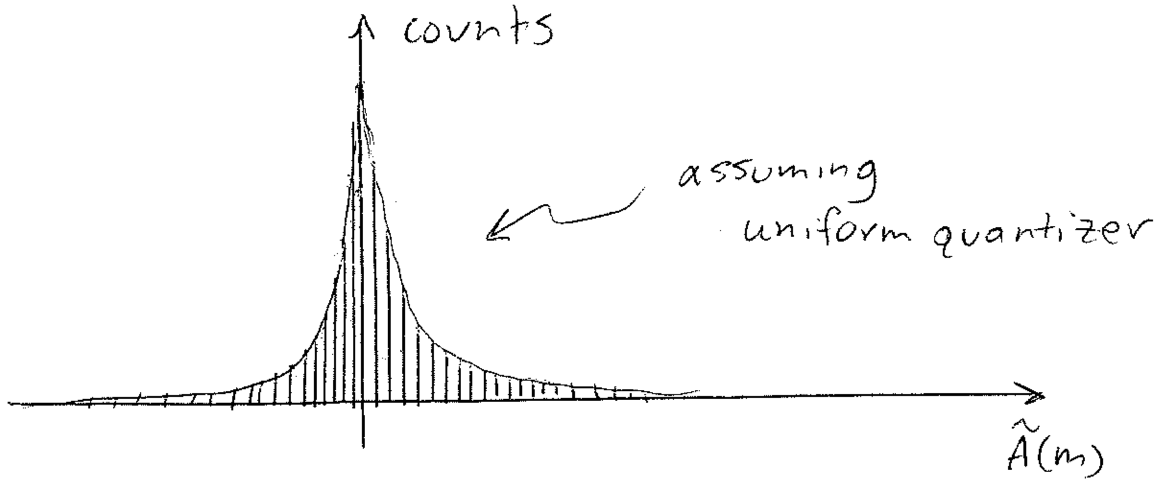
Consider a wavelet basis $B = \{g_m\}$

$$\tilde{Y} = \sum \tilde{A}(m)\, g_m$$

The memoryless assumption translates into assuming the wavelet and scaling coefficients (here, collectively denoted by $\{\tilde{A}(m)\}$) are independent random variables.

This is reasonable, at least as a first approximation, since the DWT tends to decorrelate signals. That is, the wavelet coefficients tend to be much less correlated than the original signal samples.

(Quantized) Wavelet Coefficient Distributions



- Most coefficients close to zero
- $p(\tilde{A}(m) \approx 0)$ large
- $p(|\tilde{A}(m)| > \Upsilon)$ small

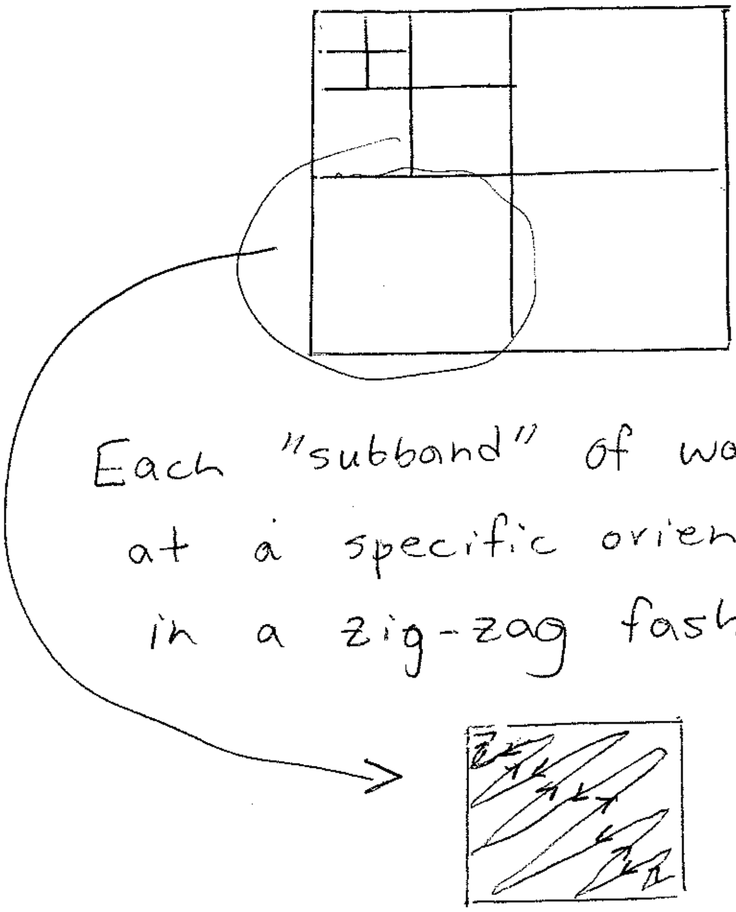
Construct Huffman codes accordingly.

Run-length Encoding

Long runs of zero-valued coefficients are often encountered. We can assign Huffman codes to such runs to achieve even greater compression.

Scanning for Runs

- Consider DWT image representation



Each "subband" of wavelet coefficients at a specific orientation is scanned in a zig-zag fash :

- We could assign special symbols to denote runs of say 2 to 32 zero-valued coefficients in succession, each of which has an associated probability of occurrence. These symbols are added to the set of quantized coefficient symbols to construct the Huffman code.

Aside: JPEG standard essentially uses the same set-up, except it employs the discrete cosine transform (DCT) on 8x8 subimages, the DCT of each subimage being treated similar to the wavelet subband images.

JPEG vs. DWT-based Coders

DWT-based coders typically outperform JPEG (better distortion rates and visual appearance).

One reason for this is that the wavelet basis functions are better suited to images than are cosines.

At very low bit rates, when many small coefficients are set to zero in quantization step, the distortions are less noticeable using wavelets.

The best wavelets are biorthogonal wavelets like the Daubechies 7,9 system.

Embedded Transform Coding

"Embedded" coders group bits in order of significance, by ordering wavelet coefficients according to magnitudes and sending the first bits of the largest coefficients first.

This is especially advantageous for high speed image transmission or fast image browsing since one can quickly obtain a coarse image approximation that is progressively enhanced as more bits arrive.

Embedded coders can also take advantage of prior information regarding the locations of large and small wavelet coefficients using so-called "zero-trees". (somewhat analogous to run-length encoding)

Embedding

Let's consider a general transform based representation

$$y = \sum_{m=0}^{N-1} a(m)\, g_m$$

where $a(m) = \langle y, g_m \rangle$.

The coefficients are partially ordered by grouping them into index sets S_K defined as

$$S_k = \{ m : 2^k \le |a(m)| < 2^{k+1} \}$$

The set S_k is coded with a binary "significance map"

$$b_k(m) = \begin{cases} 0, & \text{if } m \notin S_k \\ 1, & \text{if } m \in S_k \end{cases}$$

An embedded coder quantizes $a(m)$ uniformly with a quantization step size (width) $\Delta = 2^n$ that is progressively reduced.

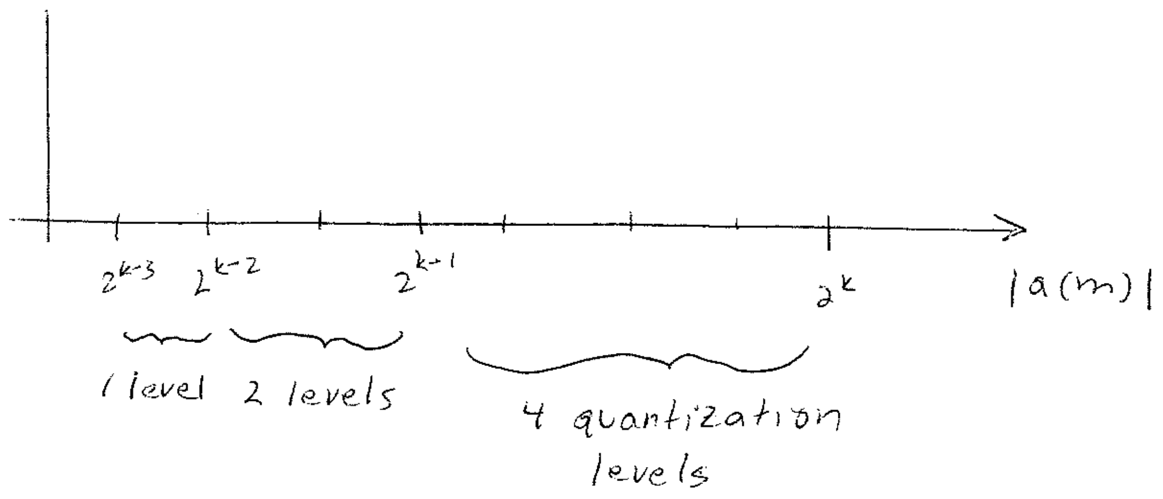
Suppose $m \in S_k$ with $k \geq n$.

The number $|a(m)|$ is quantized to a value between 2^k and 2^{k+1} using $k-n$ bits. The smaller n is, the finer the quantization. Also note that if $k = n$, then the entire set

$$\{a(m) : m \in S_k\}$$

is quantized to a single value.

Ex. $k-n = 3$



For $k < n$,

$\{a(m) : m \in S_k\}$ are quantized to zero.

One extra bit is added to denote the sign of each non-zero $Q(|a(m)|)$.

Embedded Coding Algorithm

① Initialization Store index n of 1st non-empty set S_n:

$$n = \left\lfloor \max_m \log_2 |a(m)| \right\rfloor$$

② Significance Map Store significance map $b_n(m)$ and sign of each $a(m) \in S_n$.

③ Quantization refinement Store nth bit of all $|a(m)| > 2^{n+1}$. These are coefficients in some set S_k for $k > n$, whose coordinates are already stored in one of the significance maps b_k for $k > n$. Their nth bit is stored in the order in which the position was recorded.*

④ Precision refinement

Decrease n by 1 and go to ②.

* No need to store nth bit of coefficients in set S_n since we know these are 1 by definition.

<u>Ex.</u> $\{a(m)\}_{m=0}^{4} = \{0, 5, 1, -3, 1.5\}$

<u>Initialization</u>: $n = \lfloor \log_2 5 \rfloor = 2$

$S_2 = \{1\}$

$b_2 = \{01000\} \xrightarrow{\text{store}} C = \{01000\}$

$\text{signs} = \{1\} \longrightarrow C = \{010001\} \xrightarrow{\text{recon}} \{0, 4, 0, 0, 0\}$

$\text{bits} = \{\}$

<u>$n = 1$</u>

$S_1 = \{3\}$

$b_1 = \{00010\} \longrightarrow C = \{01000100010\}$

$\text{signs} = \{0\} \longrightarrow C = \{010001000100\}$

$\text{bits} = \{\underset{\substack{\nearrow \\ n=1 \text{ bit} \\ \text{for } 5}}{0}\} \longrightarrow C = \{010010001000\} \xrightarrow{\text{recon}} \{0, 4, 0, -2, 0\}$

<u>$n = 0$</u>

$S_0 = \{2, 4\}$

$b_0 = \{00101\} \longrightarrow C = \{01001000100000101\}$

$\text{signs} = \{1, 1\} \longrightarrow C = \{0100100010000010111\}$

$\text{bits} = \{1, 1\} \longrightarrow C = \{010010001000001011111\} \xrightarrow{\text{recon}} \{0, 5, 1, -3, 1\}$

<u>$n = -1$</u>

$S_{-1} = \{\}$

$b_{-1} = \{00000\} \longrightarrow C = \{01001000100000101111100000\}$

$\text{signs} = \{\}$

$\text{bits} = \{0001\} \longrightarrow C = \{010010001000001011111000000001\}$

$\xrightarrow{\text{recon}} \{0, 5, 1, -3, 1.5\}$

The embedded coding algorithm can be stopped at any time, producing a code with a desired precision. If we stop at a value $n = \ell$, then all coefficients above 2^ℓ in magnitude are quantized uniformly with a quantization step 2^ℓ. The coefficients less than 2^ℓ in magnitude are all quantized to zero. That is, if $|a(m)| < 2^\ell$, then

$$Q_\ell[a(m)] = 0.$$

The total number of bits of the embedded code is

$$R = R_0 + R_1$$

R_0 = #bits needed for significance maps

R_1 = #bits needed to code amplitudes of quantized significant coefficients.

Using significance maps we only code the large coefficients, hence typically R_1 is less than the number of bits needed for a standard wavelet coding method.

However, the embedded coding adds the additional overhead of storing the significance maps.

However, we can exploit prior information regarding the positions of large and small coefficients to code the significance maps very efficiently.

The end result is an embedded code that is even more efficient than a standard wavelet coding method based on Huffman and run-length encoding.

One algorithm that accomplishes this and which is representative of the state of the art in image compression is the zero-tree algorithm introduced by Shapiro in 1993.

The Zero-Tree Embedded Wavelet Coder

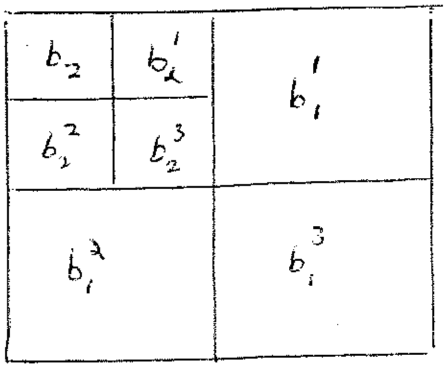
The key idea is to store the significance maps in a clever way known as "zero-trees".

Shapiro proposed using signed significance maps that store the location and sign of significant wavelet coefficients in a set $S_n = \{$ wavelet coefficients between 2^n and 2^{n+1} in magnitude

The significance maps of a set S_n have the same structures as the arrays of wavelet coefficients at each particular scale j and orientation $k=1,2,3$:

$$b_j^k(p,q) = \begin{cases} 1, & \text{if } 2^n \le d_j^k(p,q) < 2^{n+1} \\ -1, & \text{if } -2^{n+1} < d_j^k(p,q) \le -2^n \\ 0, & \text{otherwise.} \end{cases}$$

At the coarsest scale 2^J, we also compute a significance map b_J for the scaling coefficients.



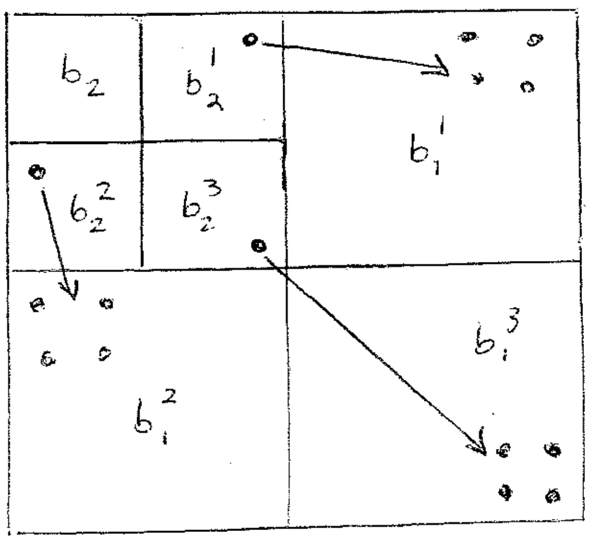
"Zero-trees" are special codes for representing multiple sets of significance maps at different scales in an efficient way. This is accomplished by associating each element of a significance map with its parent (at a coarser scale) and its children (at the next finer scale).

The parent of an element $b_j^k(p,q)$ is

$$b_{j+1}^k\left(\lfloor \tfrac{p}{2} \rfloor, \lfloor \tfrac{q}{2} \rfloor\right)$$

and its children are

$$b_{j-1}^k(2p,2q),\ b_{j-1}^k(2p+1,2q),\ b_{j-1}^k(2p,2q+1),\ b_{j-1}^k(2p+1,2q+1)$$



The key idea is the following:

$$\boxed{\text{If } b_j^k(m,n) \text{ is zero (insignificant), then it is very likely that it's children will also be zero (insignificant).}}$$

This reasoning is based on the fact that most natural images are piecewise smooth and in smooth areas the wavelet coefficients at several scales will all be insignificant. The wavelet coefficients at coarse scales are good predictors of the significance of coefficients at finer scales.

This motivates the use of special codes that signify an entire "subtree" of zeros in a significance map. These special zero-tree codes drastically reduce the number of bits required to represent significance maps.

More precisely, the zero-tree algorithm operates as follows.

If $b_j^k(p,q) = 0$ and if all its descendants are also 0, then we say that this coefficient belongs to a <u>zero-tree</u>. This implies that $d_j^k(p,q)$ and all of its descendants are insignificant (relative to a quantization level 2^n).

The position of all zeros in a zero-tree is completely determined by the scale, orientation, and position of the "<u>root</u>" node (coarsest element in the zero-tree), which is labeled with a symbol **R**. This encoding of the zero-tree is very effective if the root is at a very coarse scale.

If $b_j^k(p,q) = 0$, but one of its descendants is non-zero, then this coefficient is called an <u>isolated</u> zero and is represented with a symbol I.

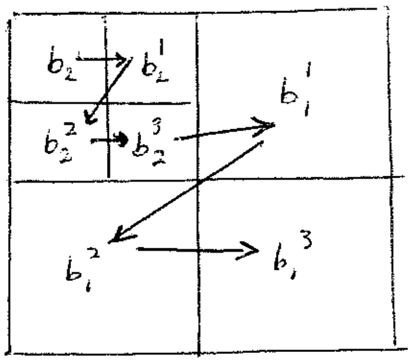
In the case of significant coefficients,

$b_j^k(p,q) = 1$ is represented by symbol P (<u>positive</u>)

$b_j^k(p,q) = -1$ is represented by symbol N (<u>negative</u>)

The significance map is then represented by a table consisting of the four symbols (R, I, P, N).

The table is scanned in a zig-zag fashion



and coded using Huffman codes (or arithemetic codes, another entropy code).

Ex.

63	-34	49	10	7	13	-12	7
-31	23	14	-13	3	4	6	-1
15	14	3	-12	5	-7	3	9
-9	-7	-14	8	4	-2	3	2
-5	9	-1	47	4	6	-2	2
3	0	-3	2	3	-2	0	4
2	-3	6	-4	3	6	3	6
5	11	5	6	0	3	-4	4

P	N	P	R	●	●	●	●
I	R	R	R	●	●	●	●
R	I	●	●	●	●	●	●
R	R	●	●	●	●	●	●
●	●	I	P	●	●	●	●
●	●	I	I	●	●	●	●
●	●	●	●	●	●	●	●
●	●	●	●	●	●	●	●

The left table is an array of 64 wavelet coefficients. The set S_5 corresponding to coefficients in $[32, 64)$ has a significance map whose zero-tree symbols are shown on the right.

The dots ● indicate positions within a zero-tree that do not require additional coding.

The symbol table at the right is scanned in a zig-zag fashion and coded with Huffman codes.

32y

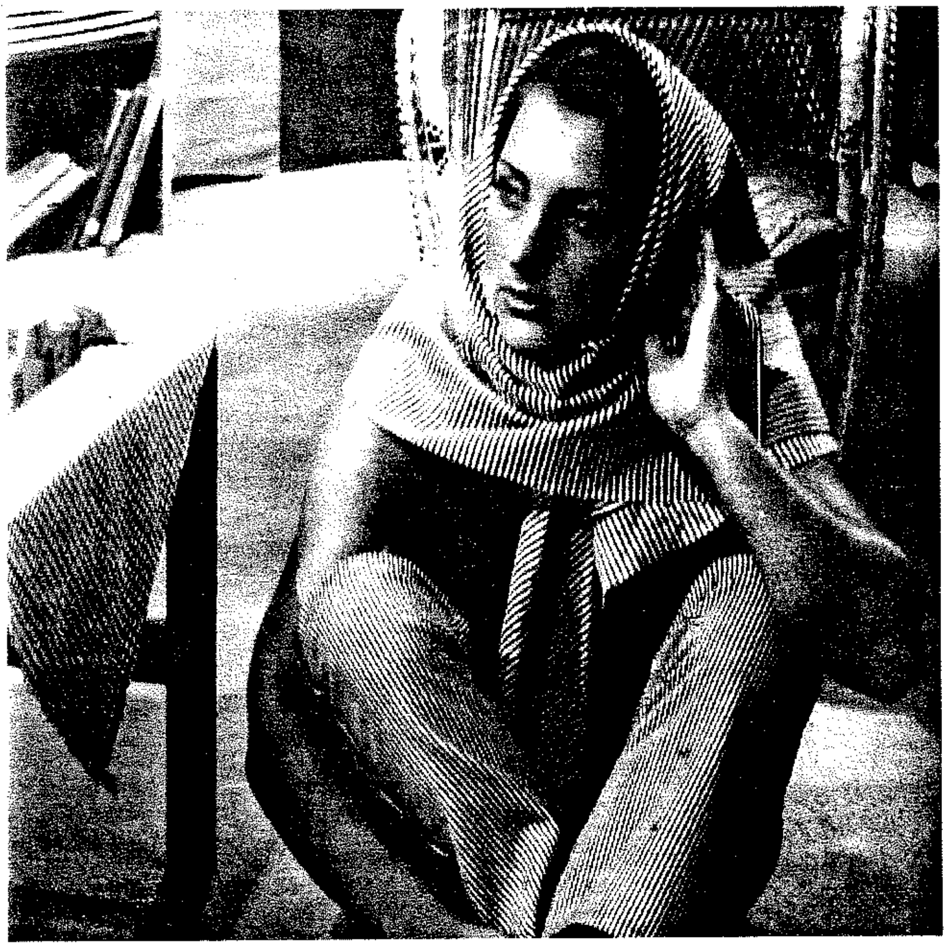

(A)

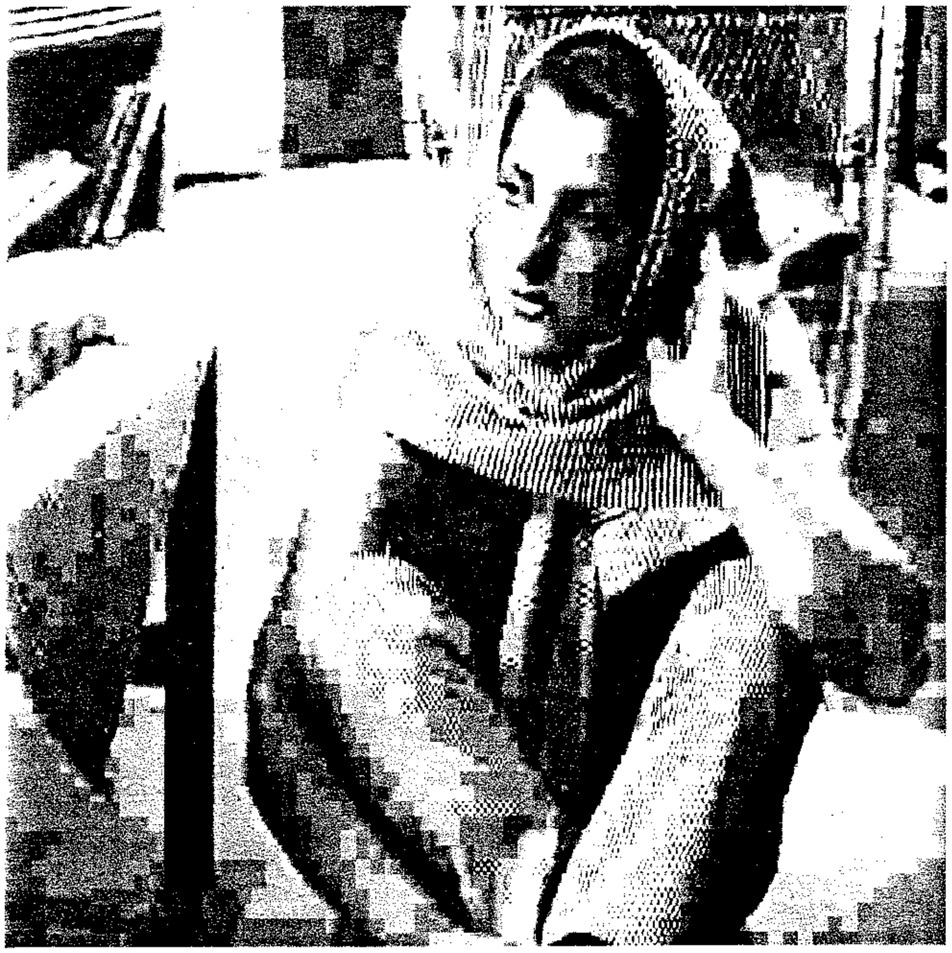
Figure 5.25 Comparison between a SPIHT-compressed and JPEG-compressed image at a compression ratio of 37:1. (A) Original image.

SPIHT = Set partioning in hierarchical trees

(similar to Shapiro's zero-tree coder)

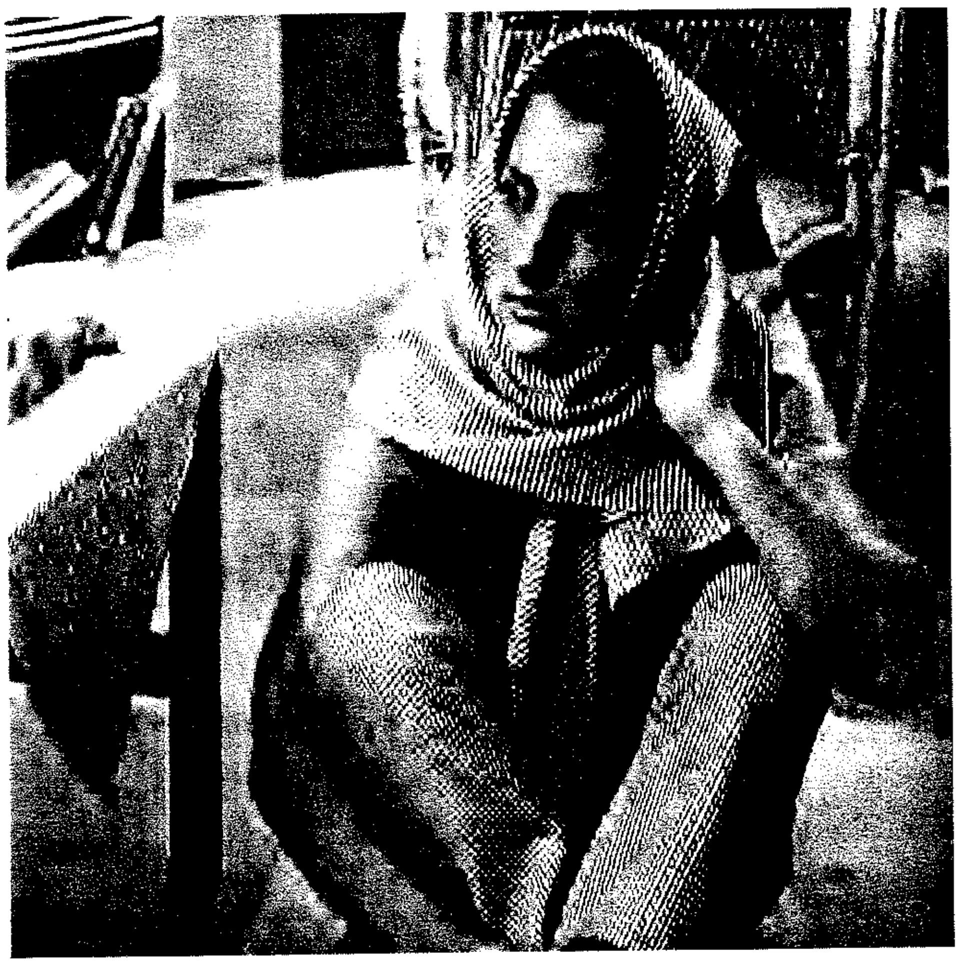
JPEG = Joint Photographic Experts Group

(current image compression standard)



(B)

Figure 5.25 (*continued*) (B) JPEG-compressed image.



(C)

Figure 5.25 (*continued*) (C) SPIHT-compressed image. Note the differences in the types of artifacts.