JPEG "Joint Photographic Experts Group"

JPEG is an image compression standard that was developed by the "Joint Photographic Experts Group". JPEG was formally accepted as an international standard in 1992. It employs a transform coding method using the DCT (Discrete Cosine Transform) to yield the spatial domain into the frequency domain.

Progressive Image compression

- Progressive compression is an attractive choice when compressed images are transmitted over a communications line and are decompressed and viewed in real time. When such an image is received and is decompressed, the decoder can very quickly display the entire image in a low-quality format, and improve the display quality as more and more of the image is being received and decompressed.
- Progressive image compression is like imagine that the encoder compresses the most important image information first, then compresses less important information and appends it to the compressed stream, and so on. This explains why all progressive image compression methods have a natural lossy option; simply stop compressing at a certain point.
- Progressive image compression, in connection with JPEG. JPEG uses the DCT to break the image up into its spatial frequency components, and it compresses the low-frequency components first. The decoder can therefore display these parts quickly, and it is these low-frequency parts that contain the principal image information. The high-frequency parts contain image details. Thus, JPEG encodes spatial frequency data progressively.

JPEG Compression Modes

The JPEG standard defined four compression modes: Hierarchical, Progressive, Sequential and lossless. Figure 0 shows the relationship of major JPEG compression modes and encoding processes.

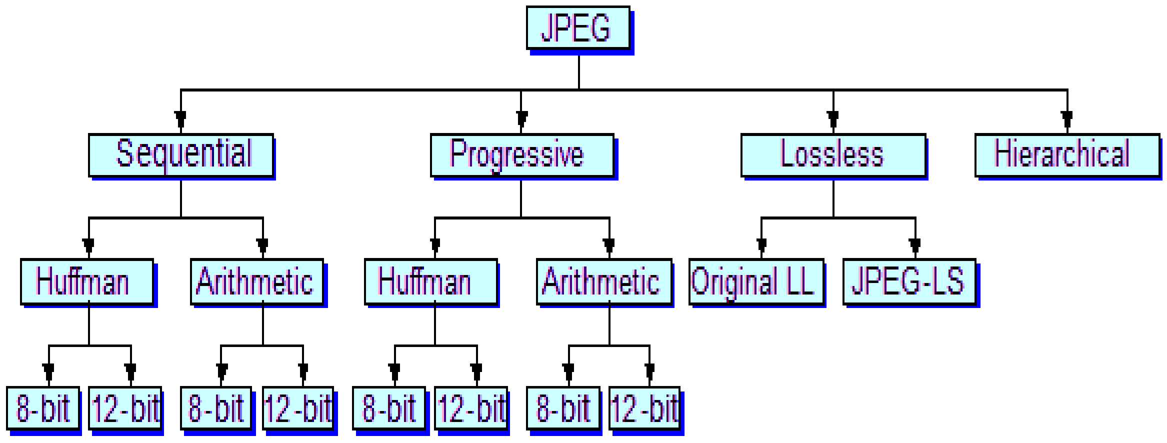
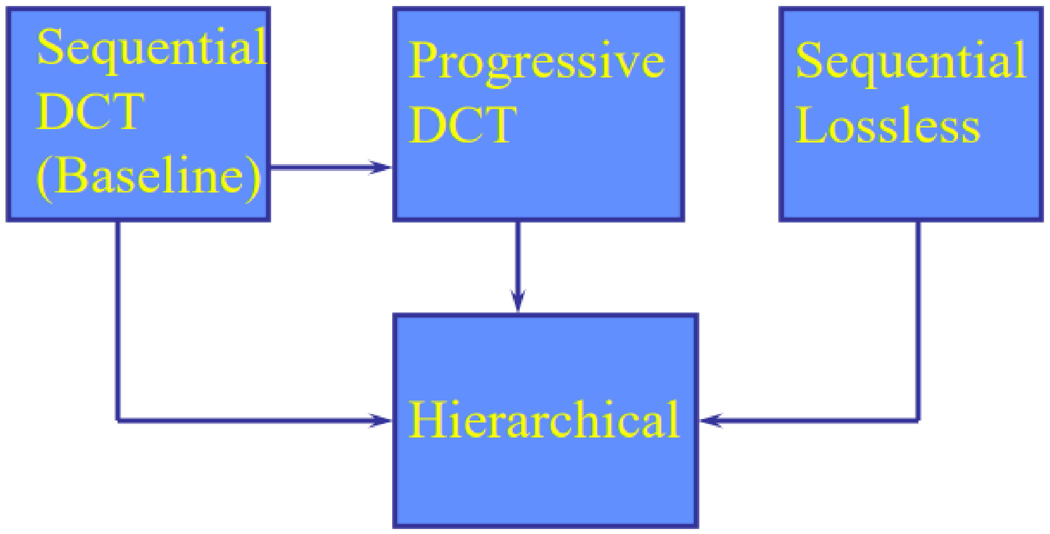


Figure . JPEG Operation Modes

JPEG 4 Compression Modes •

1. **Sequential DCT based (Lossy)**
2. **Progressive DCT based (Lossy)**
3. **Sequential lossless, DPCM based**
4. **Hierarchical**



1. **Sequential:** Sequential-mode images are encoded from top to bottom. Sequential mode supports sample data with 8 and 12 bits of precision.
   - Image components are compressed either individually or in groups (by interleaving).
   - One pass operation.
   - "Baseline System": A restricted mode, that must be included in any decoder.
   - Color Components Interleaving is done to save buffer size.

2. **Progressive**: In progressive JPEG images, components are encoded in multiple scans.

   A sequence of "scans", each codes a part of the quantized DCT coefficients data.

   • Two ways of doing this:

   – **Spectral selection:** coeff. are grouped into spectral bands, and lower-frequency bands sent first. Takes advantage of the "spectral" (spatial frequency spectrum) characteristics of the DCT coefficients: higher AC components provide detail information.

Scan 1: Encode DC and first few AC components, e.g., AC1, AC2.

Scan 2: Encode a few more AC components, e.g., AC3, AC4, AC5.

...

Scan k: Encode the last few ACs, e.g., AC61, AC62, AC63.

– **Successive Approximation:** Instead of gradually encoding spectral bands, all DCT coefficients are encoded simultaneously but with their most significant bits (MSBs) first.

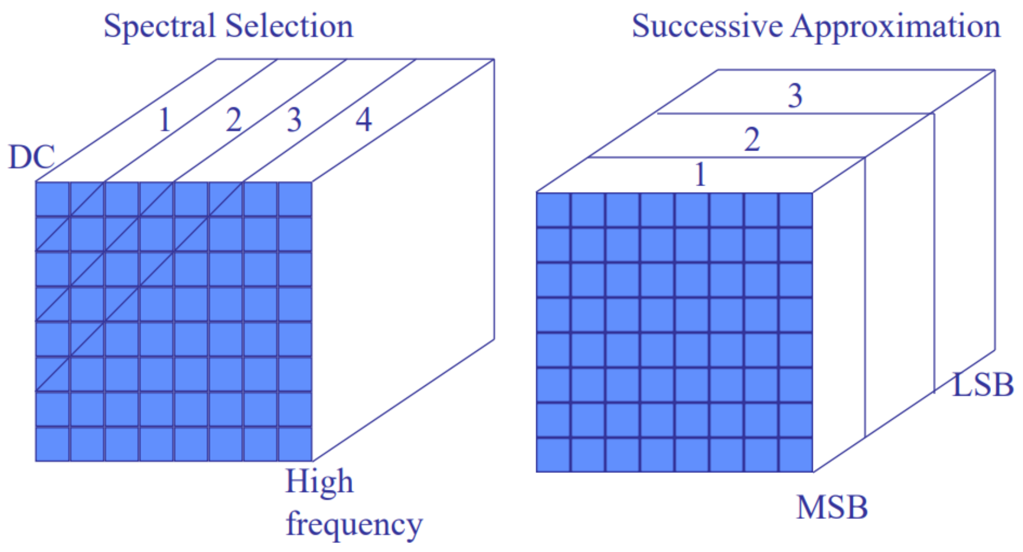
Scan 1: Encode the first few MSBs, e.g., Bits 7, 6, 5, 4.

Scan 2: Encode a few more less significant bits, e.g., Bit 3.

...

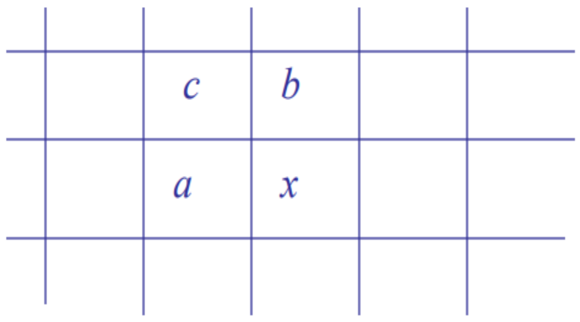
Scan m: Encode the least significant bit (LSB), Bit 0

- Note the top-left corner entry with the rather large magnitude. This is the DC coefficient (also called the constant component), which defines the basic hue for the entire block. (also called the alternating components)



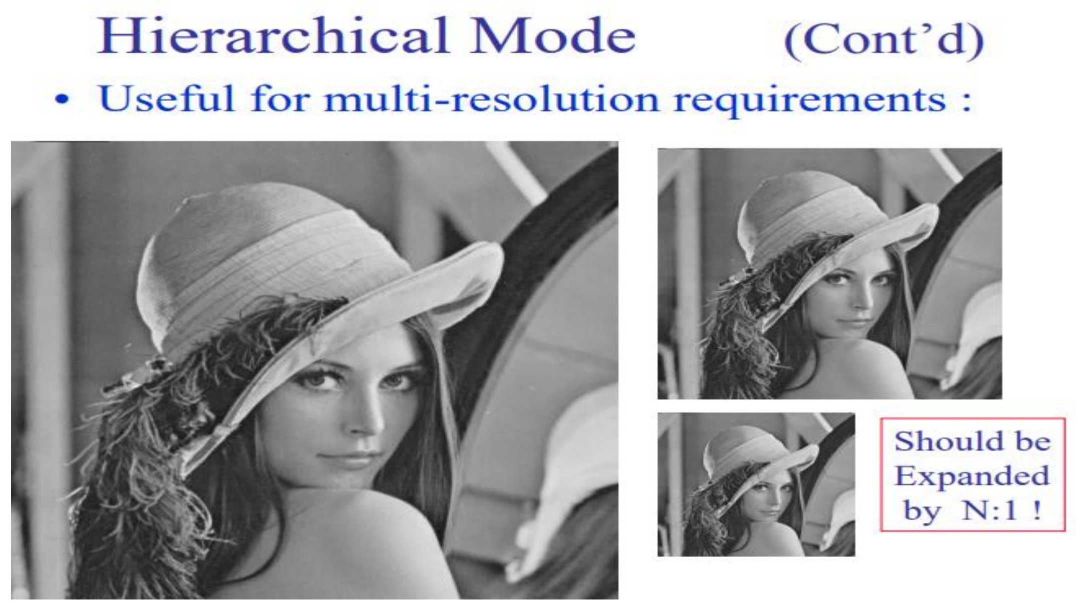
3. **Lossless:** is a class of data compression algorithms that allows the original data to be perfectly reconstructed from the compressed data

none, a, b, c, $a+b-c$, $a-(b-c)/2$, $b-(a-c)/2$, $(a+b)/2$



4. **Hierarchical:** JPEG is a super-progressive mode in which the image Is broken down into a number of subimages called frames. A frame is a collection of one or more scans. In hierarchical mode, the first frame creates a low-resolution version of image. The remaining frames refine the image by increasing the solution. JPEG is a sophisticated lossy/lossless compression method for color or grayscale still images (not videos). It does not handle bi-level (black and white)

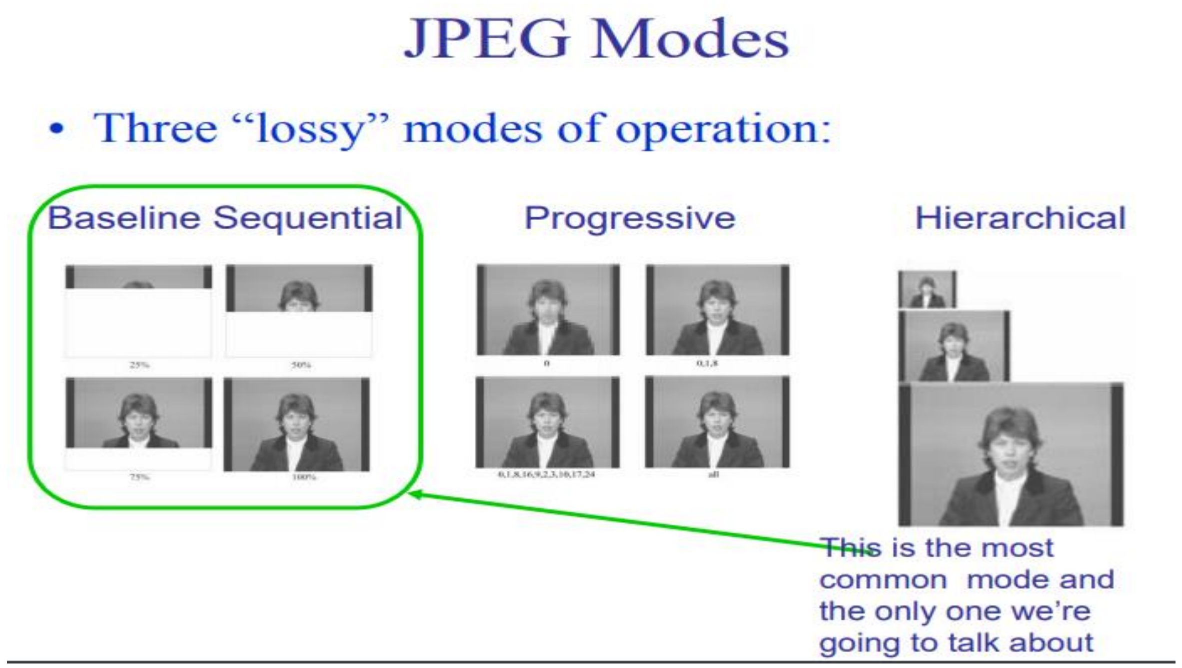
images very well. It also works best on continuous-tone images, where adjacent pixels have similar colors.



Hierarchical Mode (Cont'd)

- Useful for multi-resolution requirements :

Should be Expanded by N:1 !



JPEG Modes

- Three "lossy" modes of operation:

Baseline Sequential

Progressive

Hierarchical

This is the most common mode and the only one we're going to talk about

25%
50%
75%
100%
0
0,1,8
0,1,8,16,9,2,3,10,17,24
all

JPEG Compression:

- JPEG : Joint Photographic Experts Group
- The first international static image compression standard Published in 1992.
- The main reason for JPEG success is the quality of its output for relatively good compression ratio.
- JPEG is a lossy image compression method. It employs a transform coding method using the DCT (Discrete Cosine Transform).
- An image is a function of i and j (or conventionally x and y) in the spatial domain. The 2D DCT is used as one step in JPEG **in order to yield a frequency response** which is a **function F(u, v)** in the spatial frequency domain, indexed by two integers **u and v**.

Observations for JPEG Image Compression:

The effectiveness of the DCT transform coding method in JPEG relies on 3 major observations:

- **Observation 1:** Useful image contents change relatively slowly across the image, i.e., it is unusual for intensity values to vary widely several times in a small area, for example, within an 8×8 image block. much of the information in an image is repeated, hence "spatial redundancy".
- **Observation 2:** Psychophysical experiments suggest that humans are much less likely to notice the loss of very high spatial frequency components than the loss of lower frequency components. The spatial redundancy can be reduced by largely reducing the high spatial frequency contents.

- **Observation 3:** Visual acuity (accuracy in distinguishing closely spaced lines) is much greater for gray ("black and white") than for color.

Main Steps in JPEG Image Compression:

1. Transform RGB to YIQ or YUV and subsample color
2. Perform DCT on image blocks
3. Apply Quantization
4. Zigzag Ordering
5. DPCM on DC coefficients
6. RLE on AC coefficients
7. Perform entropy coding

The figure below shows the block diagram for JPEG encoder. If we reverse the arrows in the figure, we basically obtain a JPEG decoder.

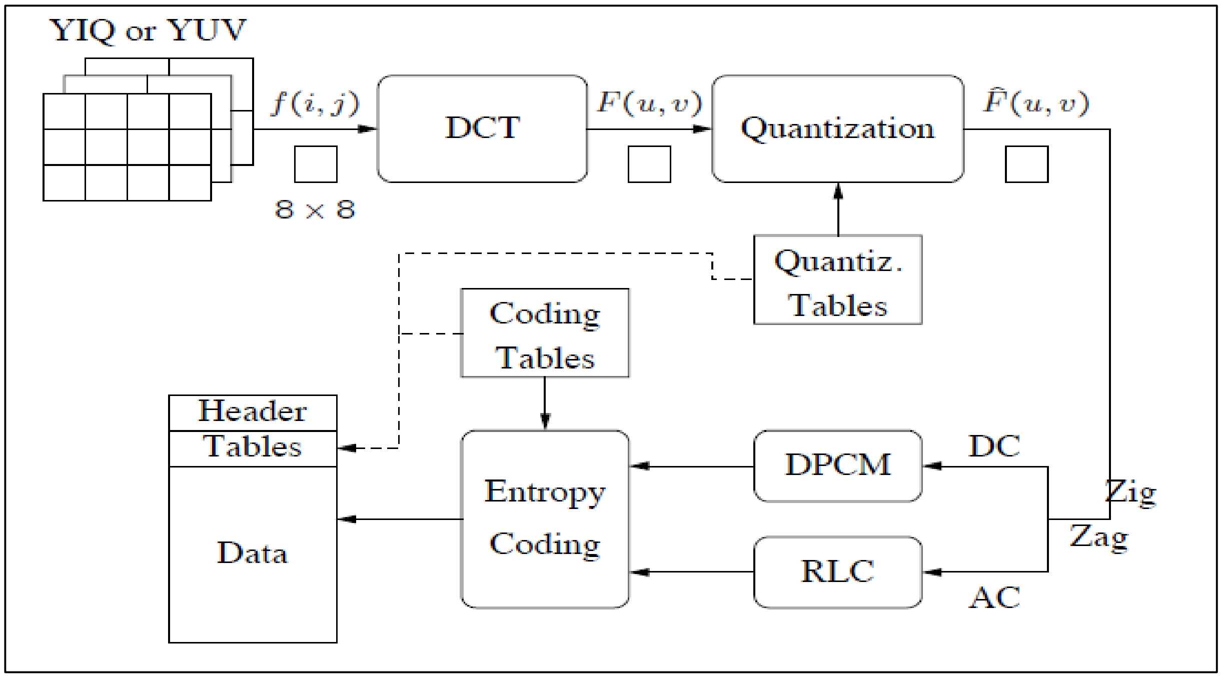


Figure (2) Block Diagram of JPEG Encoder

1. Optional Converting RGB to YUV

• YUV color mode stores color in terms of its luminance (brightness) and chrominance (hue).

• The human eye is less sensitive to chrominance than luminance.

• YUV is not required for JPEG compression, but it gives a better compression rate.

RGB vs. YUV

• It's simple arithmetic to convert RGB to YUV. The formula is based on the relative contributions that red, green, and blue make to the luminance and chrominance factors.

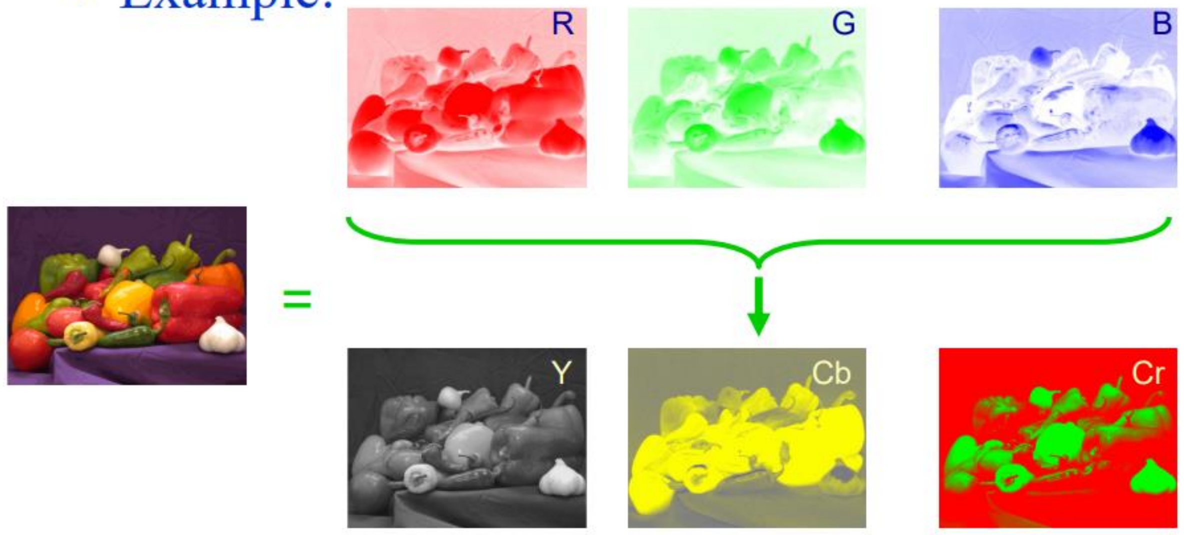
• There are several different formulas in use depending on the target monitor. For example:

$$Y = 0.299 * R + 0.587 * G + 0.114 * B$$

$$U = -0.1687 * R - 0.3313 * G + 0.5 * B + 128$$

$$V = 0.5 * R - 0.4187 * G - 0.813 * B + 128$$

• Example:



Remember: all JPEG process is operating on YCbCr color space !

2. DCT on image blocks

Each image is divided into 8 × 8 blocks each is called ***data unit***. The 2D DCT is applied to each block image $f(i, j)$, with output being the DCT coefficients $F(u, v)$ for each block. They represent the average pixel value and successive higher-frequency changes within the group. This prepares the image data for the crucial step of losing information. <u>***Since DCT involves the transcendental function cosine, it must involve some loss of information due to the limited precision of computer arithmetic.***</u> This means that even without the main lossy step (the quantization step), there will be some loss of image quality, but it is normally small. As an example, one such 8×8 8-bit subimage might be:

$$\begin{bmatrix} 52 & 55 & 61 & 66 & 70 & 61 & 64 & 73 \\ 63 & 59 & 55 & 90 & 109 & 85 & 69 & 72 \\ 62 & 59 & 68 & 113 & 144 & 104 & 66 & 73 \\ 63 & 58 & 71 & 122 & 154 & 106 & 70 & 69 \\ 67 & 61 & 68 & 104 & 126 & 88 & 68 & 70 \\ 79 & 65 & 60 & 70 & 77 & 68 & 58 & 75 \\ 85 & 71 & 64 & 59 & 55 & 61 & 65 & 83 \\ 87 & 79 & 69 & 68 & 65 & 76 & 78 & 94 \end{bmatrix}$$

Before computing the DCT of the 8×8 block, its values are shifted from a positive range to one centered on zero. For an 8-bit image, each entry in the original block falls in the range {\displaystyle [0,255]}. The midpoint of the range (in this case, the value 128) is subtracted from each entry to produce a data range that is centered on zero, so that the modified range is {\displaystyle [-128,127]}. This step reduces the dynamic range requirements in the DCT processing stage that follows. This step results in the following values:

$$g = \begin{array}{c} \overset{x}{\longrightarrow} \\ \begin{bmatrix} -76 & -73 & -67 & -62 & -58 & -67 & -64 & -55 \\ -65 & -69 & -73 & -38 & -19 & -43 & -59 & -56 \\ -66 & -69 & -60 & -15 & 16 & -24 & -62 & -55 \\ -65 & -70 & -57 & -6 & 26 & -22 & -58 & -59 \\ -61 & -67 & -60 & -24 & -2 & -40 & -60 & -58 \\ -49 & -63 & -68 & -58 & -51 & -60 & -70 & -53 \\ -43 & -57 & -64 & -69 & -73 & -67 & -63 & -45 \\ -41 & -49 & -59 & -60 & -63 & -52 & -50 & -34 \end{bmatrix} \end{array} \Big\downarrow y.$$

The next step is to take the two-dimensional DCT, which is given by:

The next step is to take the two-dimensional DCT, which is given by:

$$G_{u,v} = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7} g_{x,y}\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right]$$

where

- u is the horizontal spatial frequency, for the integers $0 \le u < 8$.
- v is the vertical spatial frequency, for the integers $0 \le v < 8$.
- $\alpha(u) = \begin{cases} \frac{1}{\sqrt{2}}, & \text{if } u = 0 \\ 1, & \text{otherwise} \end{cases}$ is a normalizing scale factor to make the transformation orthonormal
- $g_{x,y}$ is the pixel value at coordinates (x, y)
- $G_{u,v}$ is the DCT coefficient at coordinates (u, v).

If we perform this transformation on our matrix above, we get the following (rounded to the nearest two digits beyond the decimal point):

$$G = \begin{array}{c} \overset{u}{\longrightarrow} \\ \begin{bmatrix} -415.38 & -30.19 & -61.20 & 27.24 & 56.12 & -20.10 & -2.39 & 0.46 \\ 4.47 & -21.86 & -60.76 & 10.25 & 13.15 & -7.09 & -8.54 & 4.88 \\ -46.83 & 7.37 & 77.13 & -24.56 & -28.91 & 9.93 & 5.42 & -5.65 \\ -48.53 & 12.07 & 34.10 & -14.76 & -10.24 & 6.30 & 1.83 & 1.95 \\ 12.12 & -6.55 & -13.20 & -3.95 & -1.87 & 1.75 & -2.79 & 3.14 \\ -7.73 & 2.91 & 2.38 & -5.94 & -2.38 & 0.94 & 4.30 & 1.85 \\ -1.03 & 0.18 & 0.42 & -2.42 & -0.88 & -3.02 & 4.12 & -0.66 \\ -0.17 & 0.14 & -1.07 & -4.19 & -1.17 & -0.10 & 0.50 & 1.68 \end{bmatrix} \end{array} \Big\downarrow v.$$

3. Quantization

Each of the 64 frequency components in a data unit is divided by a separate number called its ***quantization coefficient*** (QC), and then rounded to an integer. This is where information is irretrievably lost. Large QCs cause more loss, so the high frequency components typically have larger QCs. Each of the 64 QCs is a JPEG parameter and can, in principle, be specified by the user. In practice, most JPEG implementations use the QC tables recommended by the JPEG standard for the ***luminance*** and ***chrominance*** image components.

$$\hat{F}(u,v) = round\left(\frac{F(u,v)}{Q(u,v)}\right)$$

Below is an example of the jpge compression for a smooth image block

Luminance Quantization Table

16,	11,	10,	16,	24,	40,	51,	61,
12,	12,	14,	19,	26,	58,	60,	55,
14,	13,	16,	24,	40,	57,	69,	56,
14,	17,	22,	29,	51,	87,	80,	62,
18,	22,	37,	56,	68,	109,	103,	77,
24,	35,	55,	64,	81,	104,	113,	92,
49,	64,	78,	87,	103,	121,	120,	101,
72,	92,	95,	98,	112,	100,	103,	99

Chrominance Quantization Table

17,	18,	24,	47,	99,	99,	99,	99,
18,	21,	26,	66,	99,	99,	99,	99,
24,	26,	56,	99,	99,	99,	99,	99,
47,	66,	99,	99,	99,	99,	99,	99,
99,	99,	99,	99,	99,	99,	99,	99,
99,	99,	99,	99,	99,	99,	99,	99,
99,	99,	99,	99,	99,	99,	99,	99,
99,	99,	99,	99,	99,	99,	99,	99

The human eye is good at seeing small differences in brightness over a relatively large area, but not so good at distinguishing the exact strength of a high frequency brightness variation. This allows one to greatly reduce the amount of information in the high frequency components. This is done by simply dividing each component in the frequency domain by a constant for that component, and then rounding to the nearest integer. This rounding operation is the only lossy operation in the whole process (other than chroma subsampling) if the DCT computation is performed with sufficiently high precision. As a result of this, it is typically the case that many of the higher frequency components are rounded to zero, and many of the rest become small positive or negative numbers, which take many fewer bits to represent.

The elements in the quantization matrix control the compression ratio, with larger values producing greater compression. A typical quantization matrix (for a quality of 50% as specified in the original JPEG Standard), is as follows:

$$Q = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}.$$

The quantized DCT coefficients are computed with

$$B_{j,k} = \mathrm{round}\left(\frac{G_{j,k}}{Q_{j,k}}\right) \text{ for } j = 0,1,2,\ldots,7; k = 0,1,2,\ldots,7$$

$$B_{j,k} = \mathrm{round}\left(\frac{G_{j,k}}{Q_{j,k}}\right) \text{ for } j = 0,1,2,\ldots,7; k = 0,1,2,\ldots,7$$

where G is the unquantized DCT coefficients; Q is the quantization matrix above; and B is the quantized DCT coefficients.

Using this quantization matrix with the DCT coefficient matrix from above results in:

$$B = \begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -3 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

For example, using −415 (the DC coefficient) and rounding to the nearest integer

$$\mathrm{round}\left(\frac{-415.37}{16}\right) = \mathrm{round}\,(-25.96) = -26.$$

Notice that most of the higher-frequency elements of the sub-block (i.e., those with an *x* or *y* spatial frequency greater than 4) are compressed into zero values.

An 8 × 8 block from the Y image of 'Lena'

$$f(i,j) = \begin{matrix}
200 & 202 & 189 & 188 & 189 & 175 & 175 & 175 \\
200 & 203 & 198 & 188 & 189 & 182 & 178 & 175 \\
203 & 200 & 200 & 195 & 200 & 187 & 185 & 175 \\
200 & 200 & 200 & 200 & 197 & 187 & 187 & 187 \\
200 & 205 & 200 & 200 & 195 & 188 & 187 & 175 \\
200 & 200 & 200 & 200 & 200 & 190 & 187 & 175 \\
205 & 200 & 199 & 200 & 191 & 187 & 187 & 175 \\
210 & 200 & 200 & 200 & 188 & 185 & 187 & 186
\end{matrix}$$

$$F(u,v) = \begin{matrix}
515 & 65 & -12 & 4 & 1 & 2 & -8 & 5 \\
-16 & 3 & 2 & 0 & 0 & -11 & -2 & 3 \\
-12 & 6 & 11 & -1 & 3 & 0 & 1 & -2 \\
-8 & 3 & -4 & 2 & -2 & -3 & -5 & -2 \\
0 & -2 & 7 & -5 & 4 & 0 & -1 & -4 \\
0 & -3 & -1 & 0 & 4 & 1 & -1 & 0 \\
3 & -2 & -3 & 3 & 3 & -1 & -1 & 3 \\
-2 & 5 & -2 & 4 & -2 & 2 & -3 & 0
\end{matrix}$$

$$\hat{F}(u,v) = \begin{matrix}
32 & 6 & -1 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{matrix}$$

$$\tilde{F}(u,v) = \begin{matrix}
512 & 66 & -10 & 0 & 0 & 0 & 0 & 0 \\
-12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-14 & 0 & 16 & 0 & 0 & 0 & 0 & 0 \\
-14 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{matrix}$$

$$\tilde{f}(i,j) = \begin{matrix}
199 & 196 & 191 & 186 & 182 & 178 & 177 & 176 \\
201 & 199 & 196 & 192 & 188 & 183 & 180 & 178 \\
203 & 203 & 202 & 200 & 195 & 189 & 183 & 180 \\
202 & 203 & 204 & 203 & 198 & 191 & 183 & 179 \\
200 & 201 & 202 & 201 & 196 & 189 & 182 & 177 \\
200 & 200 & 199 & 197 & 192 & 186 & 181 & 177 \\
204 & 202 & 199 & 195 & 190 & 186 & 183 & 181 \\
207 & 204 & 200 & 194 & 190 & 187 & 185 & 184
\end{matrix}$$

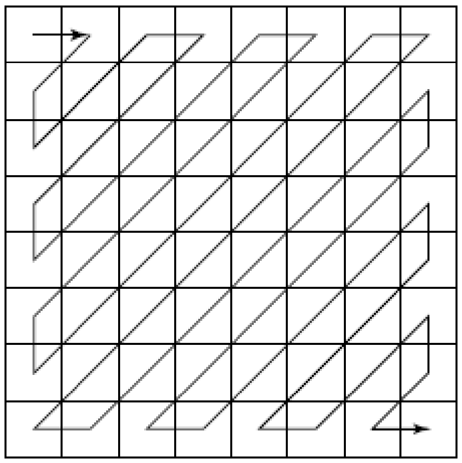
$$\epsilon(i,j) = f(i,j) - \tilde{f}(i,j) = \begin{matrix}
1 & 6 & -2 & 2 & 7 & -3 & -2 & -1 \\
-1 & 4 & 2 & -4 & 1 & -1 & -2 & -3 \\
0 & -3 & -2 & -5 & 5 & -2 & 2 & -5 \\
-2 & -3 & -4 & -3 & -1 & -4 & 4 & 8 \\
0 & 4 & -2 & -1 & -1 & -1 & 5 & -2 \\
0 & 0 & 1 & 3 & 8 & 4 & 6 & -2 \\
1 & -2 & 0 & 5 & 1 & 1 & 4 & -6 \\
3 & -4 & 0 & 6 & -2 & -2 & 2 & 2
\end{matrix}$$

Notes that changing the compression ratio simply by multiplicatively scaling the numbers in the *Q(u,v)* matrix. In fact, the *quality factor*, a user choice offered in every JPEG implementation, is essentially linearly tied to the scaling factor.

Now note how the $\boldsymbol{\epsilon(i,j) = f(i,j) - \tilde{f}(i,j)}$ is differ in the examples above, why? In the first example the pixel values in the block contain few high –spatial frequency changes. i.e. JPEG dose introduce more loss if the image has quickly changing details.

4. Zig-zag ordering and run-length encoding

The 64 quantized frequency coefficients (which are now integers) of each data unit are encoded using a combination of RLE and Huffman coding. To make it most likely to hit a long run of zeros: a *zig-zag scan* is used to turn the 8×8 matrix $\hat{F}(u,v)$ into a *64-vector*.



$$B = \begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -3 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

$$\begin{array}{rrrrrrr}
\mathbf{-26} \\
\mathbf{-3} & \mathbf{0} \\
\mathbf{-3} & \mathbf{-2} & \mathbf{-6} \\
\mathbf{2} & \mathbf{-4} & \mathbf{1} & \mathbf{-3} \\
\mathbf{1} & \mathbf{1} & \mathbf{5} & \mathbf{1} & \mathbf{2} \\
\mathbf{-1} & \mathbf{1} & \mathbf{-1} & \mathbf{2} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{-1} & \mathbf{-1} & \mathbf{0} & \mathbf{0}
\end{array}$$

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	
0	**0**	**0**	**0**	**0**	**0**		
0	**0**	**0**	**0**	**0**			
0	**0**	**0**	**0**				
0	**0**	**0**					
0	**0**						
0							

RLC aims to turn the $\hat{F}(u,v)$ values into sets *{#-zeros-to skip , next non-zero value}*.

From the above example

(32,6,-1,-1,0,-1,0,0,0,-1,0,0,1,0,0,……….,0)

First do not treat the DC component and the rest (AC component) will be

(0, 6)(0,-1)(0,-1)(1,-1)(3,-1)(2, 1)(0, 0)

A special pair (0,0) indicates the end of blocks after the last nonzero AC coefficient is reached.

1. DPCM on DC coefficients

DPCM *Differential Pulse Code Modulation* is a member differential encoding family. Differential Encoding methods calculates the difference between two consecutive data items, and encode the difference. It is depends on the concept that correlated values are generally similar, so their differences are small.

DC values reflects the average intensity of each block, but these coefficient in unlikely to change hardly within a short distance. This makes DPCM *Differential Pulse Code Modulation* an ideal scheme for coding the DC coefficients.

If the DC coefficients for the first 5 image blocks are

150, 155, 149, 152, 144

Then the DPCM would produce

```
150, 5, -6, 3, -8,
```

Assuming $d_i = DC_{i+1} - DC_i \;\; and \;\; d_0 = DC_0$

2. Huffman Coding of DC coefficients

Use DC as an example: each DPCM coded DC coefficient is represented by (SIZE, AMPLITUDE), where SIZE indicates how many bits are needed for representing the coefficient, and AMPLITUDE contains the actual bits. For the negative values, one's complement scheme is used. SIZE is Huffman coded since smaller SIZEs occur much more often. AMPLITUDE is not Huffman coded; its value can change widely so Huffman coding has no appreciable benefit.

In the example we're using, codes

150, 5, −6, 3, −8

Will be turned into

```
(8, 10010110), (3, 101), (3, 001), (2, 11), (4, 0111)
```

3. Huffman coding for AC coefficients

AC coefficients are run-length encoded (RLC). RLE pairs (Runlength, Value) are Huffman coded as with DC only on Value. So we get a triple: (Runlength, Size, Amplitude). However, Runlength, Size allocated 4-bits each (both will be one byte) and put into a single byte with is then Huffman coded. Again, Amplitude is not coded. So only Runlength an size are Huffman coded.

JPEG Image Decompression System

The JPEG decompression system is inverse of the JPEG compression system.

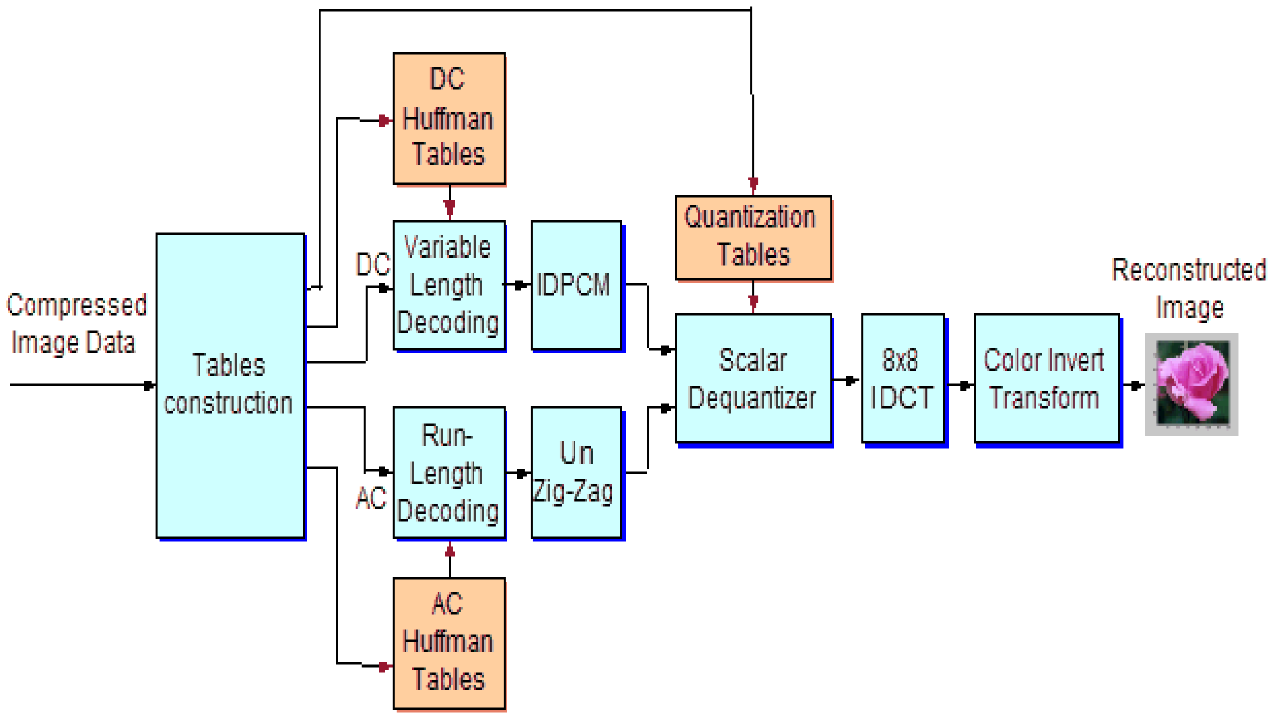


Figure 10. The JPEG decompression structure

As soon the code streams entered the decompression system, the all the received quantization and Huffman tables are reconstructed. The frame headers are also decoded to determine the size and precision of the image. The compressed stream for each 8×8 block is split into two parts. The DC code is decoded using the DC Huffman tables. The value output from DC decoder is, indeed, the difference between the DC value of the current and the previous 8×8 blocks. The IDPCM restores back the true DC value by adding the value obtained from the DC decoder with the DC value decoded from the previous block. The AC part is decoded using the AC Huffman tables to get the AC coefficients, which are organized in zig-zag order. Therefore, the unzigzag stage reorganizes the coefficients into 8×8 block. The dequantization stage performs the multiplications between the coefficients with the The IDCT performs the invert discrete cosine transform for each 8×8 block. Since the quantization generates quantization errors, the reconstructed block data is no longer identical to that of original image.

The data obtained at the IDCT output form the chrominance and luminance images, adding with the level offset and finally are converted into the RGB image before displaying on the screen.

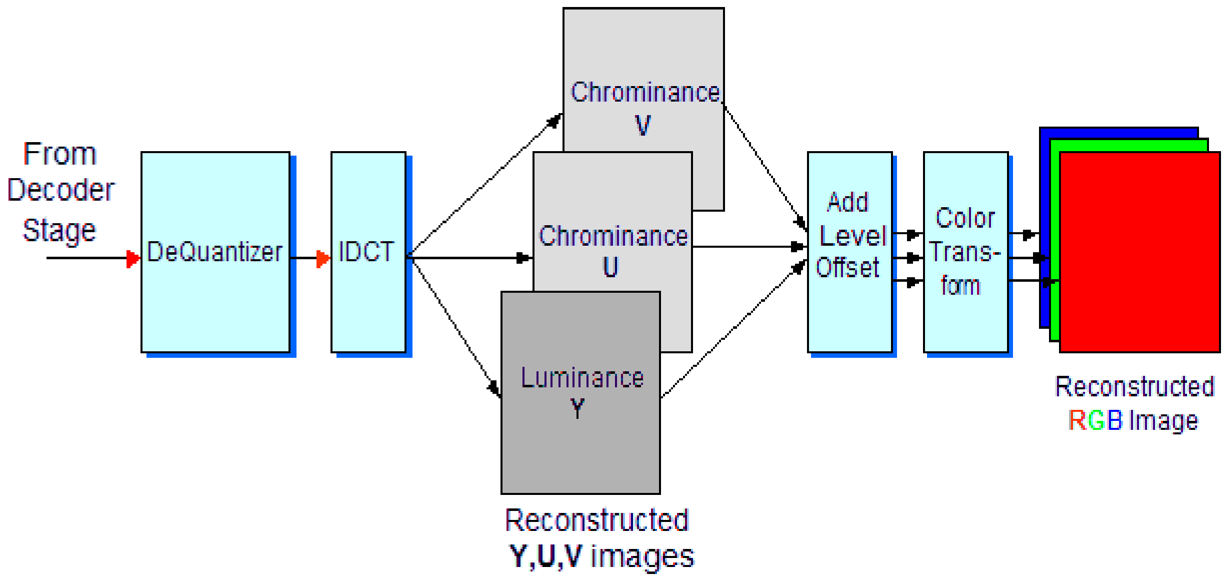


Figure 3. Color invert transformation at the decoder

The color invert transformation scheme is illustated in Figure 3.

JPEG-LS

The lossless mode of JPEG is inefficient and often is not even implemented. As a result, the ISO decided to develop a new standard for the ***lossless (or near-lossless)*** **compression of *continuous-tone images*.** The result became popularly known as JPEG-LS. This method is not simply an extension or a modification of JPEG. It is a new method, designed to be simple and fast**. *It does not use the DCT, does not use arithmetic coding, and uses quantization in a limited way, and only in its near-lossless option.***

JPEG-LS examines several of the previously seen neighbors of the current pixel, uses them as the *context* of the pixel, uses the context to predict the pixel and to select a probability distribution out of several such distributions, and uses that distribution to encode the prediction error with a special Golomb code. There is also a run mode, where the length of a run of identical pixels is encoded.

Vector Quantization

This is a generalization of the scalar quantization method. It is used for both image and sound compression. In practice, vector quantization is commonly used to compress data that has been digitized from an analog source, such as sampled sound and scanned images (drawings or photographs). Such data is called ***Digitally Sampled Analog Data (DSAD).*** It is a lossy compression method.

Vector quantization is based on two facts:

i. We know that compression methods that compress strings, rather than individual symbols, can, in principle, produce better results.
ii. Adjacent items in an image and in digitized sound are correlated. There is a good chance that the near neighbors of a pixel P will have the same values as

P or very similar values. Also, consecutive audio samples rarely differ by much.

The basic vector quantization procedure is illustrated in the following figure says that the encoder finds the closest code vector to the input vector and outputs the associated index. On the decoder side, exactly the same codebook is used. When the code index of the input vector is received, *a simple table lookup is performed* to determine the reconstruction vector.

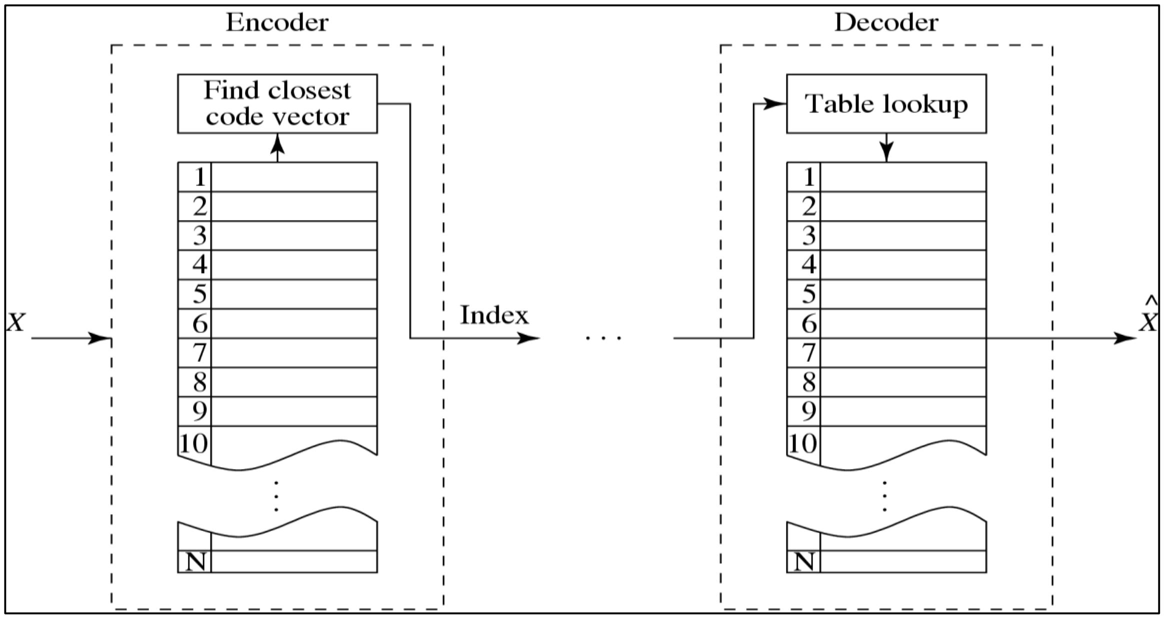


Figure (3) Basic vector quantization procedure

Now how to build this codebook, there are many ways, LBG algorithm is the basis of many vector quantization methods for the compression of images and sound.

Its main steps are the following:

<u>Step 0:</u> Select a threshold value ϵ and set k = 0 and $D^{(-1)} = \infty$. Start with an initial codebook with entries $C_i^{(k)}$(where k is currently zero, but will be incremented in each iteration). Denote the image blocks byB_i (these blocks are also called *training vectors*, since the algorithm uses them to find the best codebook entries).

Step 1: Pick up a codebook entry $C_i^{(k)}$. Find all the image blocks $B_m$that are closer to $C_i$than to any other C_j . Phrased more precisely; find the set of all B_m that satisfy

$$d(B_m, C_i) < d(B_m, C_j) for\ all\ \ i \neq j$$

This set (or *partition*) is denoted by $P_i^{(k)}$. Repeat for all values of *i*. It may happen that some partitions will be empty, and we deal with this problem below.

Step 2: Select an *i* and calculate the distortion $D_i^{(k)}$ between codebook entry $C_i^{(k)}$ and the set of training vectors (partition) $P_i^{(k)}$found for it in Step 1. Repeat for all *i*, then calculate the average $D^{(k)}$of all the $D_i^{(k)}$. A distortion $D_i^{(k)}$ for a certain *i* is calculated by computing the distances $d(C_i^{(k)}, B_m)$ for all the blocks $B_m$in partition $P_i^{(k)}$, then computing the average distance.

Step 3: If $\left(D^{(k-1)} - D^{(k)}\right)/D^{(k)} \ \leq \epsilon$ halt. The output of the algorithm is the last set of codebook entries $C_i^{(k)}$. This set can now be used to (lossy) compress the image with vector quantization. In the first iteration *k* is zero, so $D^{(k-1)} = D^{(-1)} = \infty > \epsilon$. This guarantees that the algorithm will not stop at the first iteration.

Step 4: Increment *k* by 1 and calculate new codebook entries $C_i^{(k)}$; each equals the average of the image blocks (training vectors) in partition $P_i^{(k-1)}$ that was computed in Step 1. (This is how the codebook entries are adapted to the particular image.) Go to Step 1.

Example:

Our example assumes an image consisting of 24 pixels, organized in the 12 blocks (each has 2 pixels that can be plotted on paper as 2D points)

$B1$ = (32, 32),

$B2$ = (60, 32),

$B3$ = (32, 50),

$B4$ = (60, 50),

$B5$ = (60, 150),

$B6$ = (70, 140),

$B7$ = (200, 210),

$B8$ = (200, 32),

$B9$ = (200, 40),

$B10$ = (200, 50),

$B11$ = (215, 50),

and $B12$ = (215, 35).

It is clear that the 12 points are concentrated in four regions. We select an initial codebook with the four entries

$C_1^{(0)}$= (70, 40),

$C_2^{(0)}$ = (60, 120),

$C_3^{(0)}$= (210, 200),

and $C_4^{(0)}$= (225, 50)

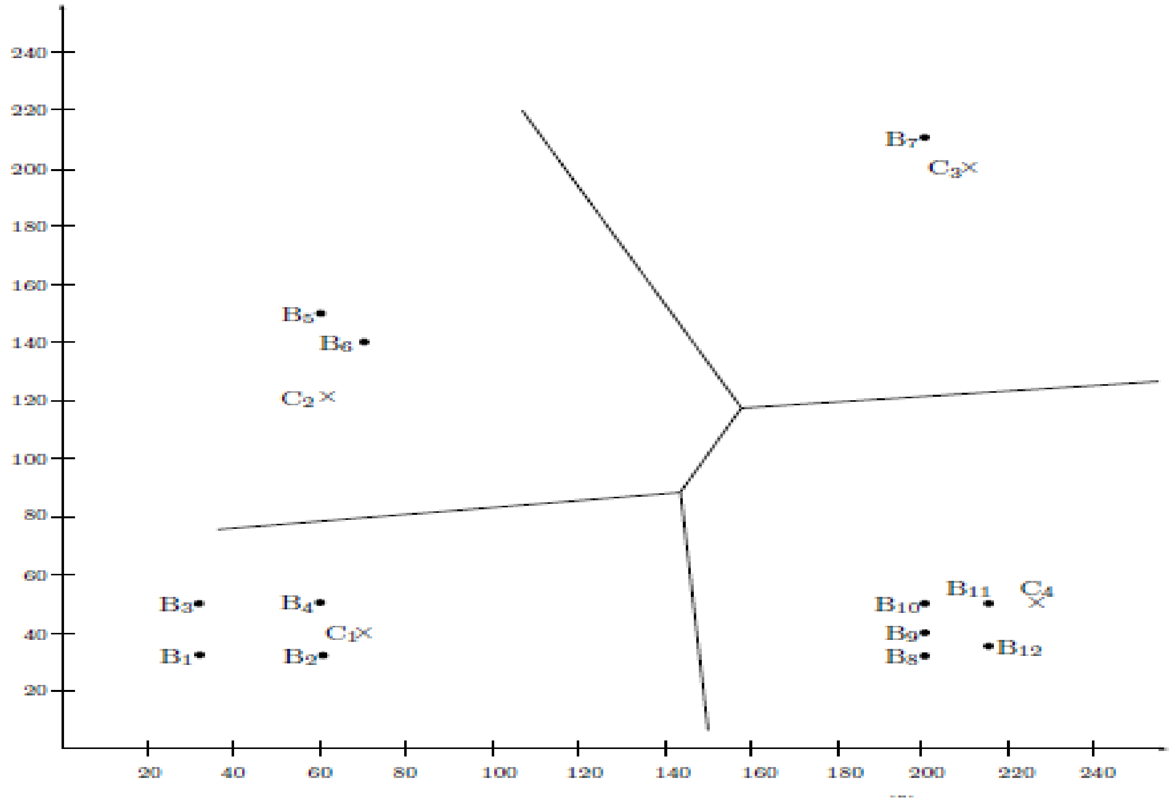
(shown as × in the diagram). These entries were selected more or less at random but we show later how the LBG algorithm selects them methodically, one by one. Because of the graphical nature of the data, it is easy to determine the four initial partitions. They are

$P_1^{(0)}$= (B1,B2,B3,B4),

$P_2^{(0)}$= = (B5,B6),

$P_3^{(0)}$= = (B7), and

$P_4^{(0)}$= (B8,B9,B10,B11,B12).



The table below to compute the $D_i^{(0)}$

$$\begin{array}{rrr}
\text{I:} & (70-32)^2+(40-32)^2 = 1508, & (70-60)^2+(40-32)^2 = 164, \\
 & (70-32)^2+(40-50)^2 = 1544, & (70-60)^2+(40-50)^2 = 200, \\
\text{II:} & (60-60)^2+(120-150)^2 = 900, & (60-70)^2+(120-140)^2 = 500, \\
\text{III:} & (210-200)^2+(200-210)^2 = 200, & \\
\text{IV:} & (225-200)^2+(50-32)^2 = 449, & (225-200)^2+(50-40)^2 = 725, \\
 & (225-200)^2+(50-50)^2 = 625, & (225-215)^2+(50-50)^2 = 100, \\
 & (225-215)^2+(50-35)^2 = 325. &
\end{array}$$

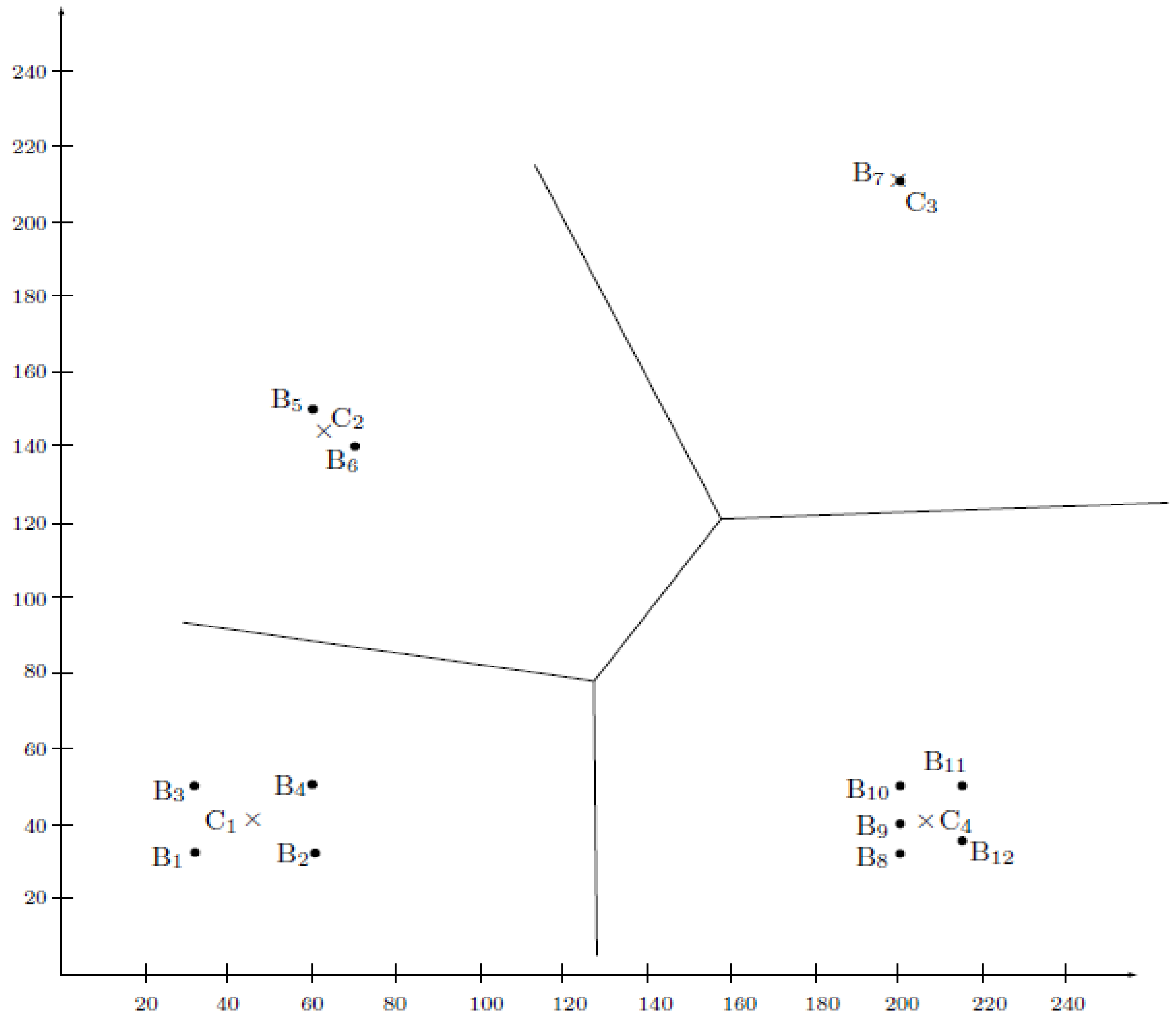
The Table above shows how the average distortion $D^{(0)}$is calculated for the first iteration (we use the Euclidean distance function). The result is

$$\begin{aligned}
D^{(0)} =&(1508+164+1544+200+900+500 \\
&+200+449+725+625+100+325)/12 \\
=&603.33.
\end{aligned}$$

Step 3 indicates no convergence, since $D(-1) = \infty$, so we increment k to 1 and calculate four new codebook entries $C(1)i$ (rounded to the nearest integer for simplicity)

$$\begin{aligned}
C_1^{(1)} &= (B_1+B_2+B_3+B_4)/4 = (46,41), \\
C_2^{(1)} &= (B_5+B_6)/2 = (65,145), \\
C_3^{(1)} &= B_7 = (200,210), \\
C_4^{(1)} &= (B_8+B_9+B_{10}+B_{11}+B_{12})/5 = (206,41).
\end{aligned}$$

They are shown in the Figure below.


240
220
200
180
160
140
120
100
80
60
40
20
20
40
60
80
100
120
140
160
180
200
220
240
B_7
C_3
B_5
C_2
B_6
B_3
B_4
C_1
B_1
B_2
B_{11}
B_{10}
B_9
C_4
B_8
B_{12}

Differential Lossless Compression

The principle is to compare each pixel *p* to a *reference pixel*, which is one of its previously-encoded immediate neighbors, and encode *p* in two parts: a prefix, which is the number of most-significant bits of *p* that are identical to those of the reference pixel, and a suffix, which is the remaining least-significant bits of *p*. For example, if the reference pixel is 10110010 and *p* is 10110100, then the prefix is 5, because the five most-significant bits of *p* are identical to those of the reference pixel, and the suffix is 00. Notice that the remaining three least-significant bits are 100 but the suffix does not have to include the 1.

The prefix part is Huffman coded and since we expect most suffixes to be small, it makes sense to write the suffix on the output stream un-coded.

So for the above example the code of *p* will be the five bits $\boxed{010|00}$.

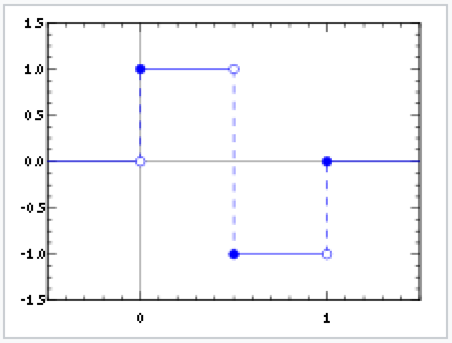
A simple black and white digitized image consists of a rectangular or square array of pixels (say 256 by 256). Each of these pixels is assigned a number that indicates how light or dark that particular pixel is in the image; usually there are 256 gray levels, with 0 corresponding to black (no brightness on your screen) and 255 to white (maximum brightness). This means that one pixel uses 8 bits; for the whole 256x256 image this gives 256 * 256 * 256 = 16777216 bits, which is 16777216/256 = 65536 bytes or 65536/1024 = 64 KB. Color images of course use a bit more memory and movies, with 18-25 images per second, use much more memory.

That is why it is very important to be able to compress images. In many cases, you can obtain images that are so close to the original image as to be virtually indistinguishable, but that are in fact stored on only a fraction of the space originally needed. For some applications, you may even be happy with a noticeable distortion if the image still looks pretty good, provided the memory savings are really huge.

There exist several types of algorithms that compress images. We shall illustrate one of them in this lab, called "subband filtering"; it is related to a mathematical concept called the "wavelet transform".

1D Haar Wavelet Transform: Recursive Definition

The Haar wavelet is also the simplest possible wavelet. The technical disadvantage of the Haar wavelet is that it is not continuous, and therefore not differentiable. This property can, however, be an advantage for the analysis of signals with sudden transitions, such as monitoring of tool failure in machines.



The Haar wavelet

Two iterations of the 2D Haar wavelet decomposition on the Lenna image. The original image is high-pass filtered, yielding the three detail coefficients subimages (top right: horizontal, bottom left: vertical, and bottom right: diagonal). It is then low-pass filtered and downscaled, yielding an approximation coefficients subimage (top left); the filtering process is repeated once again on this approximation image

Horizontal Differencing and Averaging

The basic principle is very simple. Imagine taking out just one horizontal line from the black and white image. The gray levels for the pixels in this line form a sequence of numbers between 0 and 255. In this sequence many values are very close to their neighbors - sudden jumps only occur if there was a sudden transition there in brightness (say from a light object to a dark wall) in the image. So a piece of this sequence could look like:

45 45 46 46 47 48 53 101 104 105 106 106 107 106 106 106.

If you are given any 2 numbers **a** and **b** then you completely characterize them by giving their average s = (a+b)/2 and their difference d = a-b. (Because a = s+d/2, b = s-d/2). The averages and differences for successive pairs in our sequence are:

s:	45	46	47.5	77	104.5	106	106.5	106
d:	0	0	-1	-48	-1	0	1	0

The sequence of differences has many more really small entries than large entries, and such sequences are easy to compress. Moreover, we can now easily make a small change to the **d**-sequence that would make it even more compressible. For instance, if we replace in the **d**-sequence every entry that is a 0 or 1 or -1 by 0:

d': 0 0 0 -48 0 0 0 0

then this **d'**-sequence is highly compressible.

If we now recomputed the original sequence (rounding off if necessary, so that we get integers) by a' = s+d'/2 , b' = s-d'/2, then we get:

45 45 46 46 47 47 53 101 104 104 106 106 106 106 106 106

which is very close to the original.

Vertical Differencing and Averaging

Since images are 2-dimensional, we have to do this averaging and differencing in two dimensions as well. We can first do it within every row, transforming our 256x256 image into two arrays (one of averages, one of differences) of 256x128 entries each; for each of these we can then do the same vertically, so that in the end we have four arrays of 128x128. Here is a simple example:

45	47	101	101
46	46	103	103
47	47	103	101
48	48	55	55

After averaging and differencing within every row:

averages		differences	
46	101	-2	0
46	103	0	0
47	102	0	2
48	55	0	0

After averaging and differencing in two directions:

	horizontal averages		horizontal differences	
vertical averages	46	102	-1	0
	47.5	78.5	0	1
vertical differences	0	-2	-2	0
	-1	47	0	2

In these four little arrays, the top left one corresponds to averaging in both directions; this array typically has sizeable entries for all pixels. The other three arrays typically have most of their entries very small, and can thus be highly compressed.

Averaging and Differencing

For simplicity to describe the Averaging and differencing process we take only the first row of an 8*8 matrix. This row is shown below. Because our matrix is 8*8 the process will involve three steps ($2^3=8$)

[**3** 5 **4** 8 **13** 7 **5** 3]

Step 1

For the first step we take the average of each pair of components in our original string and place the results in the first four positions of our new string. The remain four numbers are differences of the first element in each pair and its corresponding average e.g. 3-4=-1, 4-6=-2, these numbers are called detail coefficients. Our result of the first step therefore contains four averages and four detail coefficients (bold) as shown

[4 6 10 4 **-1 -2 3 1]**

Step 2

We then apply this same method to the first four components of our new string resulting in two new averages and their corresponding details coefficients. The remain four detail coefficients are simply carried directly down from our previous step. And the result for step two is as follow.

[5 7 -1 **3 -1 -2 3 1]**

Step 3

Performing the same averaging and differencing to the remaining pair of averages completes step three. The last six components have again been carried down from the previous step. We know have as our string. One row average in the first position followed by seven detail coefficient

[6 **-1 -1 3 -1 -2 3 1**]

Haar Wavelet Transform model

The easiest of all discrete wavelet transformations is the Discrete Haar Wavelet Transformation (HWT). Analysis of the Two-Dimensional HWT You can see why the wavelet transformation is well-suited for image compression. The two-dimensional HWT of the image has most of the energy conserved in the upper left-hand corner of the transform - the remaining three-quarters of the HWT consists primarily of values that are zero or near zero. The transformation is *local* as well - it turns out any element of the HWT is constructed from only four elements of the original input image. If we look at the HWT as a block matrix product, we can gain further insight about the transformation.

Suppose that the input image is square so we will drop the subscripts that indicate the dimension of the HWT matrix. If we use *H* to denote the top block of the HWT matrix and *G* to denote the bottom block of the HWT, we can express the transformation as:

$$B = WAW^T = \begin{bmatrix} H \\ G \end{bmatrix} A \begin{bmatrix} H \\ G \end{bmatrix}^T = \begin{bmatrix} H \\ G \end{bmatrix} A \begin{bmatrix} H^T \\ G^T \end{bmatrix} = \begin{bmatrix} HA \\ GA \end{bmatrix} \begin{bmatrix} H^T \\ G^T \end{bmatrix} = \begin{bmatrix} HAH^T & HAG^T \\ GAH^T & GAG^T \end{bmatrix}$$

We now see why there are four blocks in the wavelet transform. Let's look at each block individually. Note that the matrix H is constructed from the lowpass Haar filter and computes weighted averages while G computes weighted differences. The upper left-hand block is HAHT - HA averages columns of A and the rows of this product are averaged by multiplication with HT. Thus the upper left-hand corner is an approximation of the entire image. In fact, it can be shown that elements in the upper left-hand corner of the HWT can be constructed by computing weighted averages of each 2 x 2 block of the input matrix. Mathematically, the mapping is

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \rightarrow 2\cdot(a+b+c+d)/4$$

The upper right-hand block is HAGT - HA averages columns of A and the rows of this product are differenced by multiplication with GT. Thus the upper right-hand corner holds information about vertical in the image - large values indicate a large vertical change as we move across the image and small values indicate little vertical change. Mathematically, the mapping is

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \rightarrow 2\cdot(b+d-a-c)/4$$

The lower left-hand block is GAHT - GA differences columns of A and the rows of this product are averaged by multiplication with HT. Thus the lower left-hand corner holds information about horizontal in the image - large values indicate a large horizontal change as we move down the image and small values indicate little horizontal change. Mathematically, the mapping is

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \rightarrow 2\cdot(c+d-a-b)/4$$

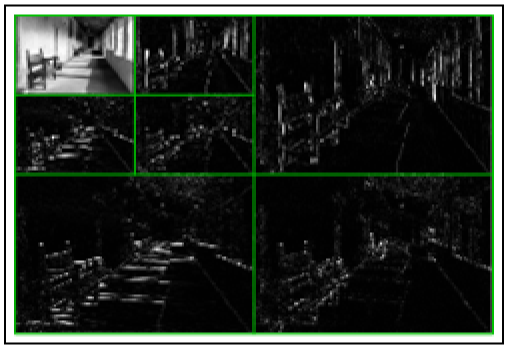
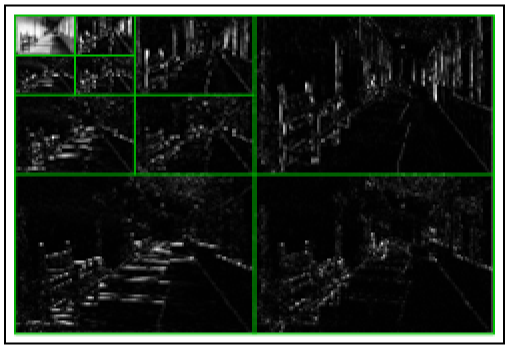
The lower right-hand block is differences across both columns and rows and the result is a bit harder to see. It turns out that this product measures changes along ± 45-degree lines. This is diagonal differences. Mathematically, the mapping is

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \rightarrow 2\cdot(b+c-a-d)/4$$

To summarize, the HWT of a digital image produces four blocks. The upper-left hand corner is an approximation or blur of the original image. The upper-right, lower-left, and lower-right blocks measure the differences in the vertical, horizontal, and diagonal directions, respectively.

Iterating the Process

If there is not much change in the image, the difference blocks are comprised of (near) zero values. If we apply quantization and convert near-zero values to zero, then the HWT of the image can be effectively coded and the storage space for the image can be drastically reduced. We can iterate the HWT and produce an even better result to pass to the coder. Suppose we compute the HWT of a digital image. Most of the high intensities are contained in the blur portion of the transformation. We can iterate and apply the HWT to the blur portion of the transform. So in the composite transformation, we replace the blur by itstransformation! The process is completely invertible - we apply the inverse HWT to the transform of the blur to obtain the blur. Then we apply the inverse HWT to obtain the original image. We can continue this process as often as we desire (and provided the dimensions of the data are divisible by suitable powers of two). The illustrations below show two iterations and three iterations of the HWT.

$$x = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

$$x = {}^{1}\!/_{\sqrt{2}} \begin{bmatrix} a+b+c+d & a-b+c-d \\ a+b-c-d & a-b-c+d \end{bmatrix}$$

Top left: **a+b+c+d** = 4-point average or 2-D low pass (L0-L0) filter.

Top right : **a-b+c-d** = average horizontal gradient or horizontal highpass and vertical lowpass (Hi-L0) filter.

Lower left : **a+b-c-d** = Average vertical gradient or horizontal lowpass and vertical high pass (L0-Hi) filter.

Lower right a**-b-c+d** =diagonal curvature or 2-D highpass (Hi-Hi) filter

To apply this transform to a complete image, we group the pixels into 2*2 blocks and apply (3) to each block. The result (after reordering)is shown in figure 1(b). to view the result sensibly we have grouped all the top left sub image in figure 1(b) and done the same for the components in the other 3 positions to form the corresponding other 3 sub images.

E.g. (1).

$$x = \begin{bmatrix} 12 & -2 \\ -2 & 0 \end{bmatrix}$$

$$x' = \begin{bmatrix} 4 & 6 \\ 6 & 8 \end{bmatrix}$$

E.g. (2).

$$x = \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$$

$$x' = \begin{bmatrix} 7 & -1 \\ -2 & 0 \end{bmatrix}$$

Video Compression MPEG: Motion Picture Expert Group

This is the standard designed by the Motion Picture Expert Group, including the updated versions MPEG-1, MPEG-2, MPEG-4, and MPEG-7.

Most popular standards include the ISO's MPEG-1 and the ITU's H.261.

It is built upon the three basic common analogue television standards: NTSC, PAL and SECAM.

There are two sizes of SIF (Source Input Format): SIF-525 (with NTSC video) and SIF-625 (for PAL video).

The H.261 standard uses CIF (Common Intermediate Format) and QCIF (Quarter Common Intermediate Format).

Video compression is based on two types of redundancies among the video data, namely spatial redundancy and temporal redundancy.

1. Spatial redundancy means the correlation among neighbouring pixels in each frame of image. This can be dealt with by the techniques for compressing still images.
2. Temporal redundancy means the similarity among neighbouring frames, since a video frame tends to be similar to its immediate neighbours.

We have studied the theory of encoding now let us see how this is applied in practice.

We need to compress video (and audio) in practice since:

1. Uncompressed video (and audio) data are huge. In HDTV, the bit rate easily exceeds 1 Gbps. -- big problems for storage and network communications. For example:

One of the formats defined for HDTV broadcasting within the United States is 1920 pixels horizontally by 1080 lines vertically, at 30 frames per second. If these numbers are all multiplied together, along with 8 bits for each of the three primary colors, the total data rate required would be approximately 1.5 Gb/sec. Because of the 6 MHz. channel bandwidth allocated, each channel will only support a data rate of 19.2 Mb/sec, which is further reduced to 18 Mb/sec by the fact that the channel must also support audio, transport, and ancillary data information. As can be seen, this restriction in data rate means that the original signal must be compressed by a figure of approximately 83:1. This number seems all the more impressive when it is realized that the intent is to deliver very high quality video to the end user, with as few visible artifacts as possible.

2. Lossy methods have to employed since the *compression ratio* of lossless methods (e.g., Huffman, Arithmetic, LZW) is not high enough for image and video compression, especially when distribution of pixel values is relatively flat.

The following compression types are commonly used in Video compression:

- Spatial Redundancy Removal - Intraframe coding (JPEG)
- Spatial and Temporal Redundancy Removal - Intraframe and Interframe coding (H.261, MPEG)

These are discussed in the following sections.

H. 261 Compression

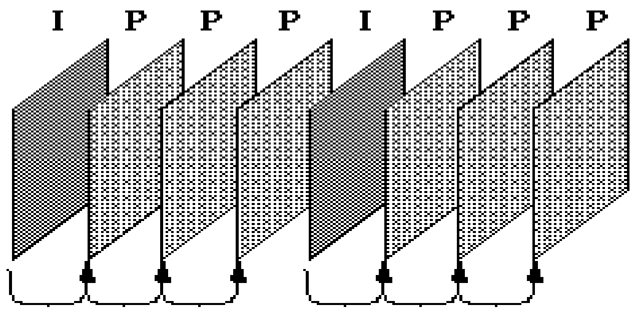
H. 261 Compression has been specifically designed for video telecommunication applications:

- Developed by CCITT in 1988-1990
- Meant for videoconferencing, video telephone applications over ISDN telephone lines.
- Baseline ISDN is 64 kbits/sec, and integral multiples (*p*x64)

Overview of H.261

The basic approach to H. 261 Compression is summarised as follows:

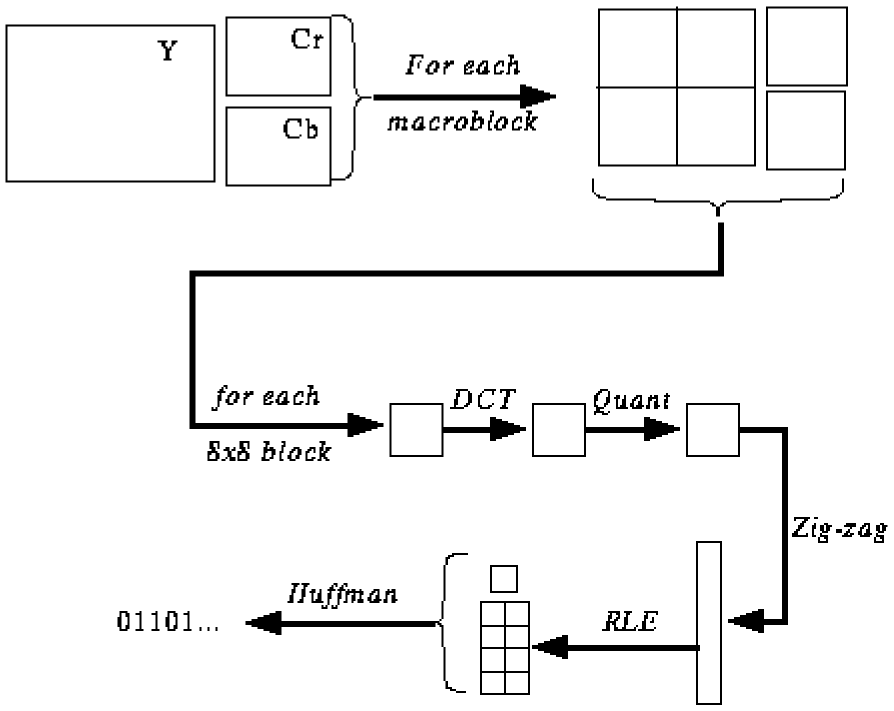
- Decoded Sequence



- Frame types are CCIR 601 CIF (352x288) and QCIF (176x144) images with 4:2:0 subsampling.
- Two frame types: Intraframes (*I-frames*) and Interframes (*P-frames*)
- I-frames use basically JPEG
- P-frames use ***pseudo-differences*** from previous frame (predicted), so frames depend on each other.
- I-frame provide us with an accessing point.

Intra Frame Coding

The term ***intra frame coding*** refers to the fact that the various lossless and lossy compression techniques are performed relative to information that is contained only within the current frame, and not relative to any other frame in the video sequence. In other words, no temporal processing is performed outside of the current picture or frame. This mode will be described first because it is simpler, and because non-intra coding techniques are extensions to these basics. Figure below shows a block diagram of a basic video encoder for intra frames only. It turns out that this block diagram is very similar to that of a JPEG still image video encoder, with only slight implementation detail differences.

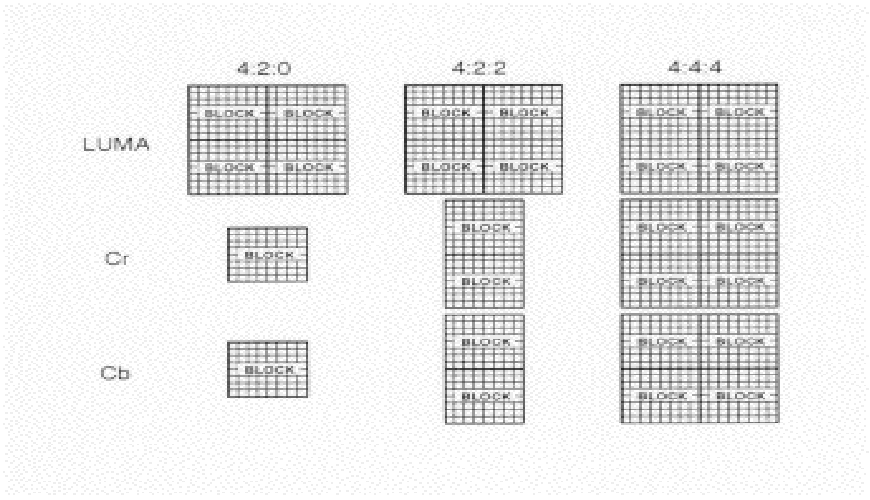


The potential ramifications of this similarity will be discussed later. The basic processing blocks shown are the video filter, discrete cosine transform, DCT coefficient quantizer, and run-length amplitude/variable length coder. These blocks are described individually in the sections below or have already been described in JPEG Compression.

This is a basic Intra Frame Coding Scheme is as follows:

- Macroblocks are 16x16 pixel areas on Y plane of original image.
  A ***macroblock*** usually consists of 4 Y blocks, 1 Cr block, and 1 Cb block.

  In the example HDTV data rate calculation shown previously, the pixels were represented as 8-bit values for each of the primary colors red, green, and blue. It turns out that while this may be good for high performance computer generated graphics, it is wasteful in most video compression applications. Research into the **Human Visual System (HVS)** has shown that the eye is most sensitive to changes in luminance, and less sensitive to variations in chrominance. Since absolute compression is the name of the game, it makes sense that MPEG should operate on a color space that can effectively take advantage of the eyes different sensitivity to luminance and chrominance information. As such, H/261 (and MPEG) uses the YCbCr color space to represent the data values instead of RGB, where Y is the luminance signal, Cb is the blue color difference signal, and Cr is the red color difference signal. A macroblock can be represented in several different manners when referring to the YCbCr color space. Figure below shows 3 formats known as 4:4:4, 4:2:2, and 4:2:0 video. 4:4:4 is full bandwidth YCbCr video, and each macroblock consists of 4 Y blocks, 4 Cb blocks, and 4 Cr blocks. Being full bandwidth, this format contains as much information as the data would if it were in the RGB color space. 4:2:2 contains half as much chrominance information as 4:4:4, and 4:2:0 contains one quarter of the chrominance information. Although MPEG-2 has provisions to handle the higher chrominance formats for professional applications, most consumer level products will use the normal 4:2:0 mode.



Macroblock Video Formats

Because of the efficient manner of luminance and chrominance representation, the 4:2:0 representation allows an immediate data reduction from 12 blocks/macroblock to 6 blocks/macroblock, or 2:1 compared to full bandwidth representations such as 4:4:4 or RGB. To generate this format without generating color aliases or artifacts requires that the chrominance signals be filtered.

The Macroblock is coded as follows:

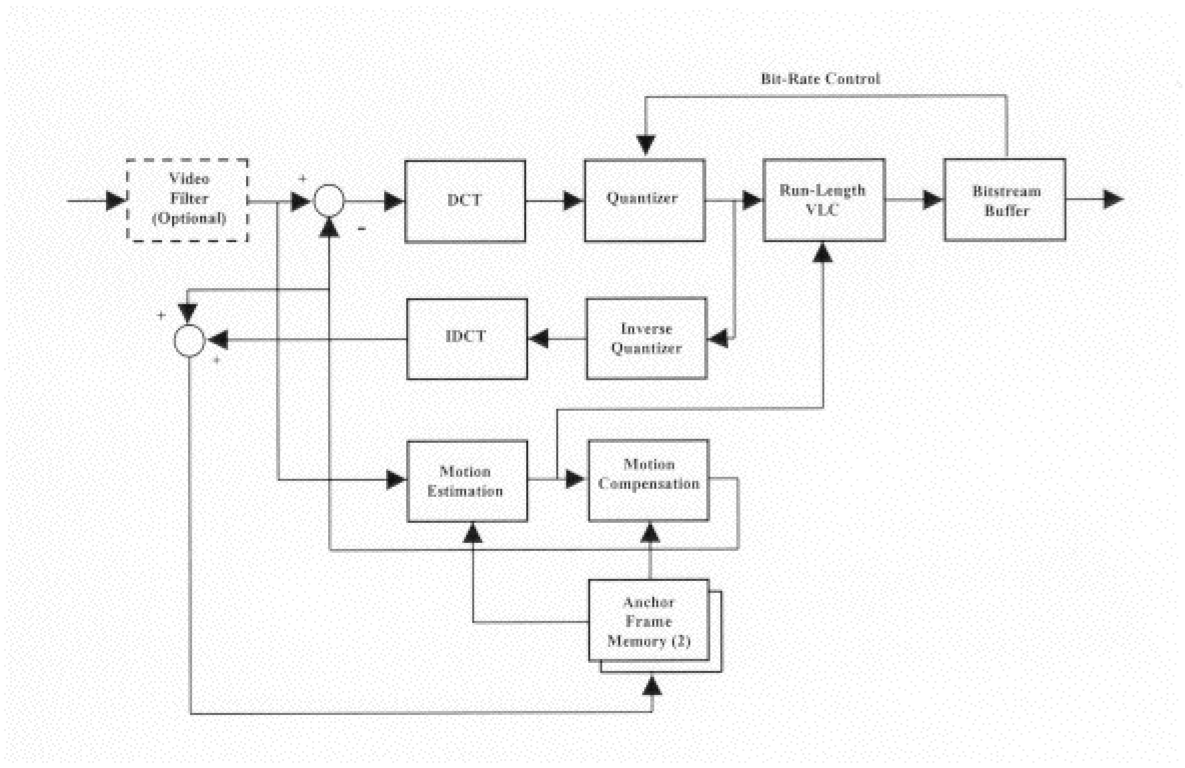
Addr	Type	Quant	Vector	CBP	b0	b1	...	b5

- Many macroblocks will be exact matches (or close enough). So send address of each block in image -> *Addr*
- Sometimes no good match can be found, so send INTRA block -> *Type*

  - Will want to vary the quantization to fine tune compression, so send quantization value -> *Quant*
  - Motion vector -> *vector*
  - Some blocks in macroblock will match well, others match poorly. So send bitmask indicating which blocks are present (Coded Block Pattern, or *CBP*).
  - Send the blocks (4 Y, 1 Cr, 1 Cb) as in JPEG.
- Quantization is by constant value for all DCT coefficients (i.e., no quantization table as in JPEG).

Inter-frame (P-frame) Coding

The previously discussed intra frame coding techniques were limited to processing the video signal on a spatial basis, relative only to information within the current video frame. Considerably more compression efficiency can be obtained however, if the inherent temporal, or time-based redundancies, are exploited as well. Anyone who has ever taken a reel of the old-style super-8 movie film and held it up to a light can certainly remember seeing that most consecutive frames within a sequence are very similar to the frames both before and after the frame of interest. Temporal processing to exploit this redundancy uses a technique known as block-based motion compensated prediction, using motion estimation. A block diagram of the basic encoder with extensions for non-intra frame coding techniques is given in Figure below. Of course, this encoder can also support intra frame coding as a subset.

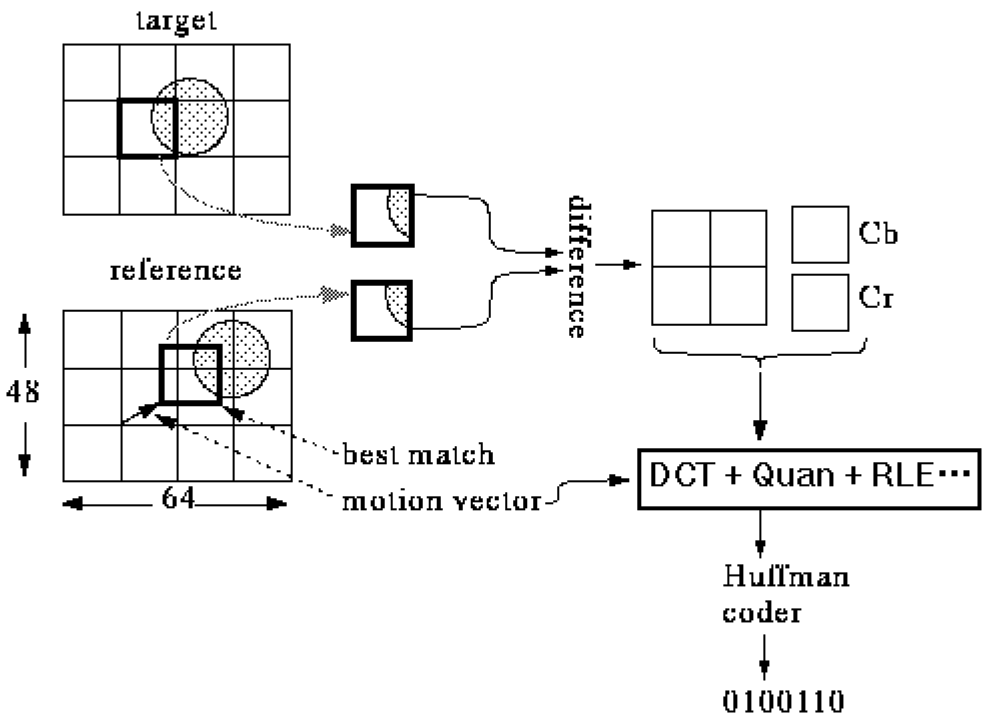


P-Frame Coding

Starting with an intra, or I frame, the encoder can forward predict a future frame. This is commonly referred to as a P frame, and it may also be predicted from other P frames, although only in a forward time manner. As an example, consider a group of pictures that lasts for 6 frames. In this case, the frame ordering is given as I,P,P,P,P,P,I,P,P,P,P,

Each P frame in this sequence is predicted from the frame immediately preceding it, whether it is an I frame or a P frame. As a reminder, I frames are coded spatially with no reference to any other frame in the sequence.

P-coding can be summarised as follows:



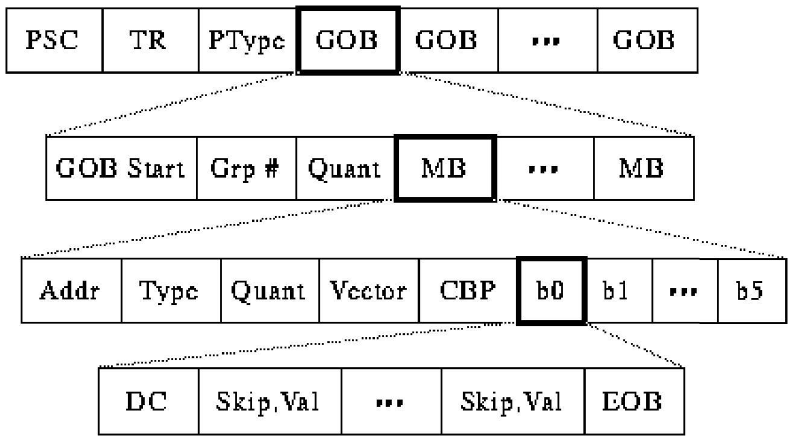
- An Coding Example (P-frame)
- Previous image is called *reference image*.
- Image to code is called *target image*.
- Actually, the difference is encoded.
- Subtle points:

1. Need to use decoded image as reference image, *not* original. Why?

2. We're using "Mean Absolute Difference" (MAD) to decide best block. Can also use "Mean Squared Error" (MSE) = sum(E*E)

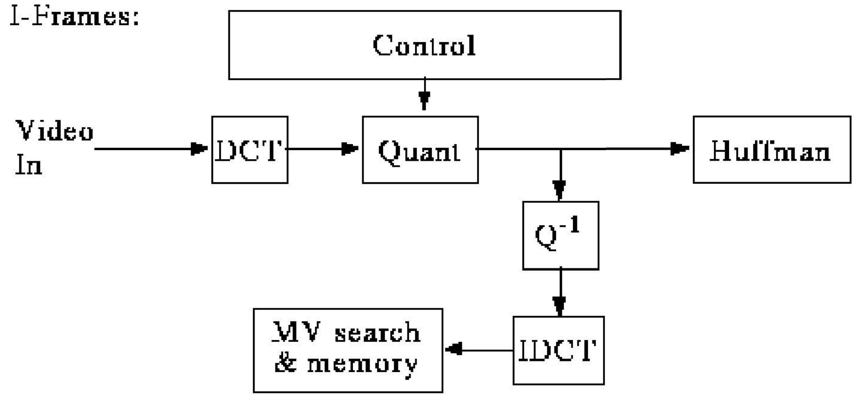
The H.261 Bitstream Structure

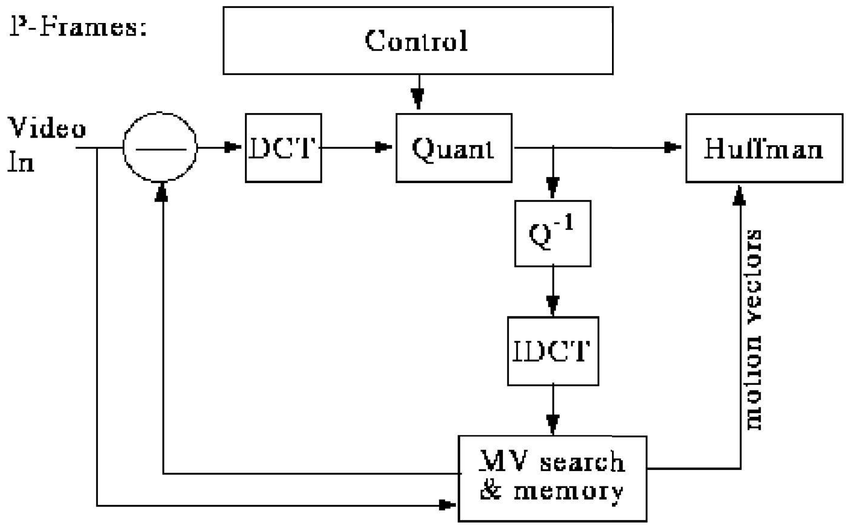
The H.261 Bitstream structure may be summarised as follows:



- Need to delineate boundaries between pictures, so send Picture Start Code -> *PSC*
- Need timestamp for picture (used later for audio synchronization), so send Temporal Reference -> *TR*
- Is this a P-frame or an I-frame? Send Picture Type -> *PType*
- Picture is divided into regions of 11x3 macroblocks called Groups of Blocks -> *GOB*
- Might want to skip whole groups, so send Group Number (*Grp #*)
- Might want to use one quantization value for whole group, so send Group Quantization Value -> *GQuant*
- Overall, bitstream is designed so we can skip data whenever possible while still unambiguous.

The overall H.261 Codec is summarised in Fig below.



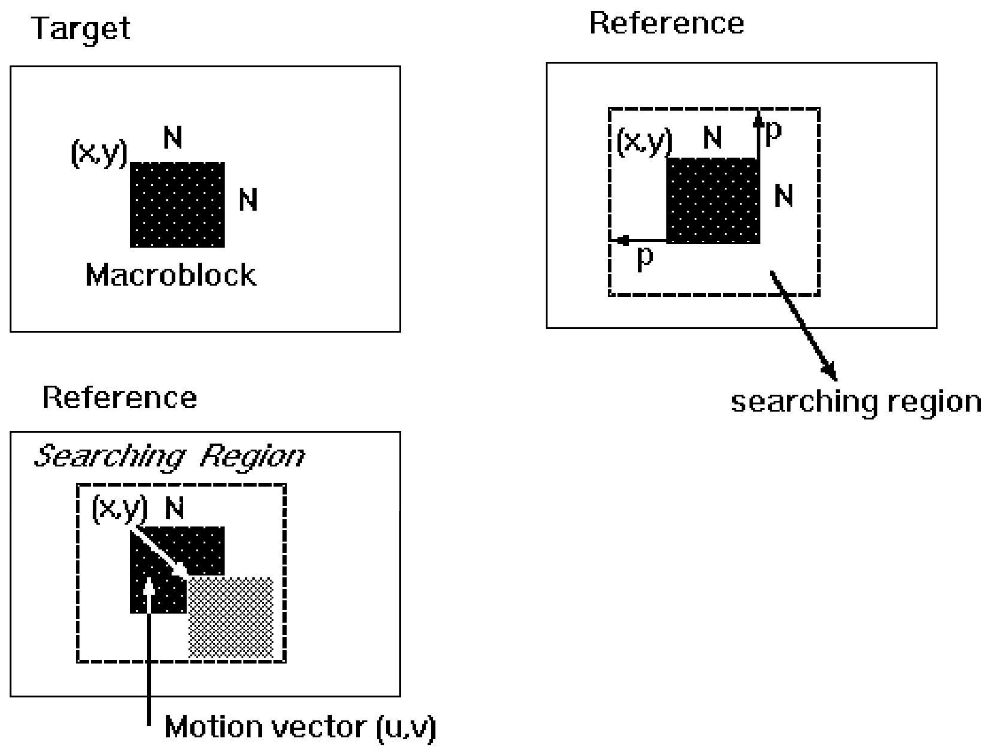


Hard Problems in H.261

There are however a few difficult problems in H.261:

- **Motion vector search**
- **Propagation of Errors**
- **Bit-rate Control**

Motion Vector Search



- *C*(*x*+*k*,*y*+*i*) - pixels in the macro block with upper left corner (*x*,*y*) in the Target.

  R(*X*+*i*+*k*,*y*+*j*+*l*) - pixels in the macro block with upper left corner (*x*+*i*,*y*+*j*) in the Reference.

  Cost function is:

$$MAE(i,j) = \frac{1}{N^2}\sum_{k=0}^{N-1}\sum_{l=0}^{N-1}\left| C(x+k, y+l) - R(x+i+k, y+j+l) \right|$$

Where MAE stands for *Mean Absolute Error*.

- Goal is to find a vector (*u*, *v*) such that MAE (*u*, *v*) is minimum

Full Search Method:

1. Search the whole $[-p, p]$ searching region.

2. Cost is:

$$\frac{IJF}{N^2}(2p+1)^2 \times N^2 \times 3$$

operations, assuming that each pixel comparison needs 3 operations (Subtraction, Absolute value, Addition).

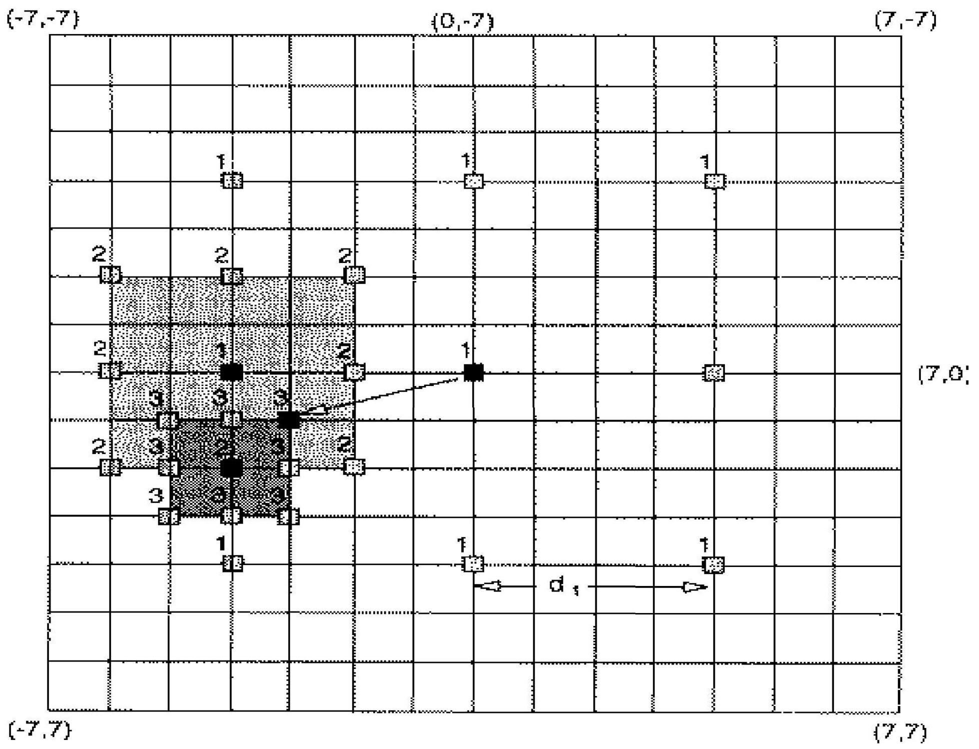
- **Two-Dimensional Logarithmic Search:**

Similar to binary search. MAE function is initially computed within a window of $[-p/2, p/2]$ at nine locations as shown in the figure.

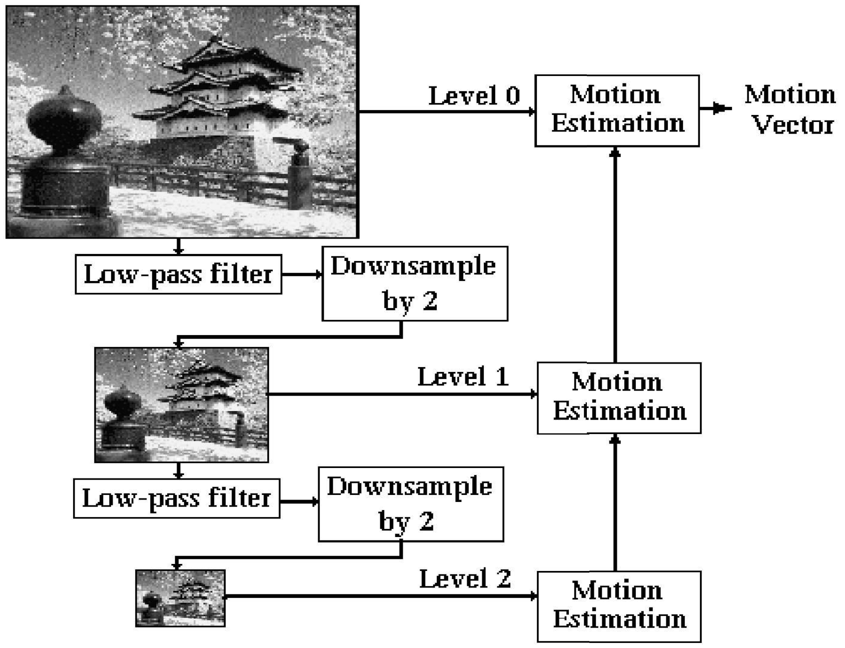
Repeat until the size of the search region is one pixel wide:

1. Find one of the nine locations that yields the minimum MAE.

2. Form a new searching region with half of the previous size and centered at the location found in step 1.



- **Hierarchical Motion Estimation:**



1. Form several low resolution version of the target and reference pictures
2. Find the best match motion vector in the lowest resolution version.
3. Modify the motion vector level by level when going up.

Performance comparison:

Search Method	Operation for 720x480 at 30 fps p = 15	p=7
Full Search	29.89 GOPS	6.99 GOPS
Logarithmic	1.02 GOPS	777.60 MOPS
Hierarchical	507.38 MOPS	398.52 MOPS

Propagation of Errors

- Send an I-frame every once in a while
- Make sure you use decoded frame for comparison

Bit-rate Control

- Simple feedback loop based on "buffer fullness"

  If buffer is too full, increase the quantization scale factor to reduce the data.

MPEG Compression

The acronym MPEG stands for Moving Picture Expert Group, which worked to generate the specifications under ISO, the International Organization for Standardization and IEC, the International Electrotechnical Commission. What is commonly referred to as "MPEG video" actually consists at the present time of two finalized standards, MPEG-11 and MPEG-22, with a third standard, MPEG-4, was finalized in 1998 for ***Very Low Bitrate Audio-Visual Coding***. The MPEG-1 and MPEG-2 standards are similar in basic concepts. They both are based on motion compensated block-based transform coding techniques, while MPEG-4 deviates from these more traditional approaches in its usage of software image construct descriptors, for target bit-rates in the very low range, < 64Kb/sec. Because MPEG-1 and MPEG-2 are finalized standards and are both presently being utilized in a large number of applications, this paper concentrates on compression techniques relating only to these two standards. Note that there is no reference to MPEG-3. This is because it was originally anticipated that this standard would refer to HDTV applications, but it was found that minor extensions to the MPEG-2 standard would suffice for this higher bit-rate, higher resolution application, so work on a separate MPEG-3 standard was abandoned.

The current thrust is MPEG-7 "Multimedia Content Description Interface" whose completion is scheduled for July 2001. Work on the new standard MPEG-21 "Multimedia Framework" has started in June 2000 and has already produced a Draft Technical Report and two Calls for Proposals.

MPEG-1 was finalized in 1991, and was originally optimized to work at video resolutions of 352x240 pixels at 30 frames/sec (NTSC based) or 352x288 pixels at

25 frames/sec (PAL based), commonly referred to as Source Input Format (SIF) video. It is often mistakenly thought that the MPEG-1 resolution is limited to the above sizes, but it in fact may go as high as 4095x4095 at 60 frames/sec. The bit-rate is optimized for applications of around 1.5 Mb/sec, but again can be used at higher rates if required. MPEG-1 is defined for progressive frames only, and has no direct provision for interlaced video applications, such as in broadcast television applications.

MPEG-2 was finalized in 1994, and addressed issues directly related to digital television broadcasting, such as the efficient coding of field-interlaced video and scalability. Also, the target bit-rate was raised to between 4 and 9 Mb/sec, resulting in potentially very high quality video. MPEG-2 consists of profiles and levels. The profile defines the bitstream scalability and the colorspace resolution, while the level defines the image resolution and the maximum bit-rate per profile. Probably the most common descriptor in use currently is Main Profile, Main Level (MP@ML) which refers to 720x480 resolution video at 30 frames/sec, at bit-rates up to 15 Mb/sec for NTSC video. Another example is the HDTV resolution of 1920x1080 pixels at 30 frame/sec, at a bit-rate of up to 80 Mb/sec. This is an example of the Main Profile, High Level (MP@HL) descriptor. A complete table of the various legal combinations can be found in reference2.

MPEG Video

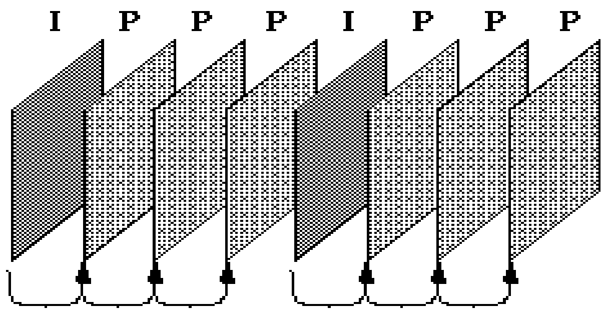
MPEG compression is essentially a attempts to over come some shortcomings of H.261 and JPEG:

Basic steps used in Video Compression

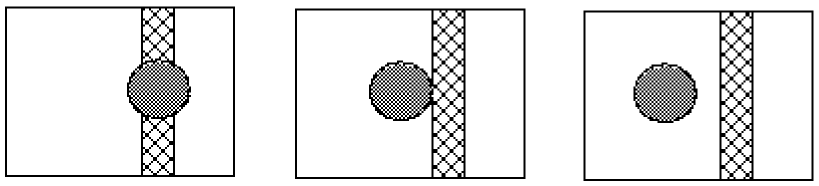
The Video Compression algorithm utilized in numerous standards (such as MPEG 1, 2 H.263) usually consists of the following steps:

1. **Motion Estimation**
2. **Motion Compensation and Image Subtraction**
3. **Discrete Cosine Transform**
4. **Quantization**
5. **Run Length Encoding**
6. **Entropy Coding – Huffman Coding**

- **Recall H.261 dependencies:**

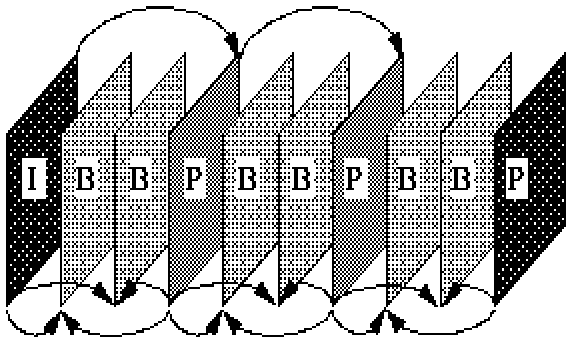


- The Problem here is that many macroblocks need information is **not** in the reference frame.
- For example:



- The ***MPEG solution*** is to add a third frame type which is a bidirectional frame, or *B-frame*
- B-frames search for macroblock in *past* and *future* frames.
- Typical pattern is IBBPBBPBB IBBPBBPBB IBBPBBPBB

Actual pattern is up to encoder, and need not be regular.

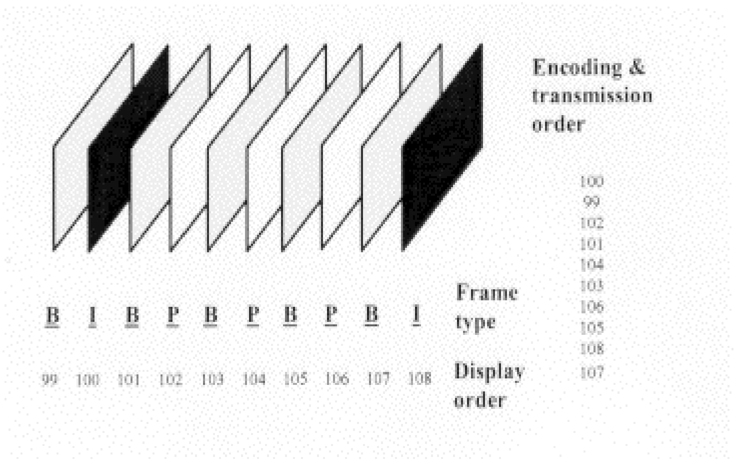


MPEG Video Layers

MPEG video is broken up into a hierarchy of layers to help with error handling, random search and editing, and synchronization, for example with an audio bitstream. From the top level, the first layer is known as the video sequence layer, and is any self-contained bitstream, for example a coded movie or advertisement. The second layer down is the group of pictures, which is composed of 1 or more groups of intra (I) frames and/or non-intra (P and/or B) pictures that will be defined later. Of course the third layer down is the picture layer itself, and the next layer beneath it is called the slice layer. Each slice is a contiguous sequence of raster ordered macroblocks, most often on a row basis in typical video applications, but not limited to this by the specification. Each slice consists of macroblocks, which are 16x16 arrays of luminance pixels, or picture data elements, with 2 8x8 arrays of associated chrominance pixels. The macroblocks can be further divided into distinct 8x8 blocks, for further processing such as transform coding. Each of these layers has its own unique 32 bit start code defined in the syntax to consist of 23 zero bits followed by a one, then followed by 8 bits for the actual start code. These start codes may have as many zero bits as desired preceding them.

B-Frames

The MPEG encoder also has the option of using forward/backward interpolated prediction. These frames are commonly referred to as bi-directional interpolated prediction frames, or B frames for short. As an example of the usage of I, P, and B frames, consider a group of pictures that lasts for 6 frames, and is given as I,B,P,B,P,B,I,B,P,B,P,B, As in the previous I and P only example, I frames are coded spatially only and the P frames are forward predicted based on previous I and P frames. The B frames however, are coded based on a forward prediction from a previous I or P frame, as well as a backward prediction from a succeeding I or P frame. As such, the example sequence is processed by the encoder such that the first B frame is predicted from the first I frame and first P frame, the second B frame is predicted from the second and third P frames, and the third B frame is predicted from the third P frame and the first I frame of the next group of pictures. From this example, it can be seen that backward prediction requires that the future frames that are to be used for backward prediction be encoded and transmitted first, out of order. This process is summarized in Figure below. There is no defined limit to the number of consecutive B frames that may be used in a group of pictures, and of course the optimal number is application dependent. Most broadcast quality applications however, have tended to use 2 consecutive B frames (I,B,B,P,B,B,P,) as the ideal trade-off between compression efficiency and video quality.



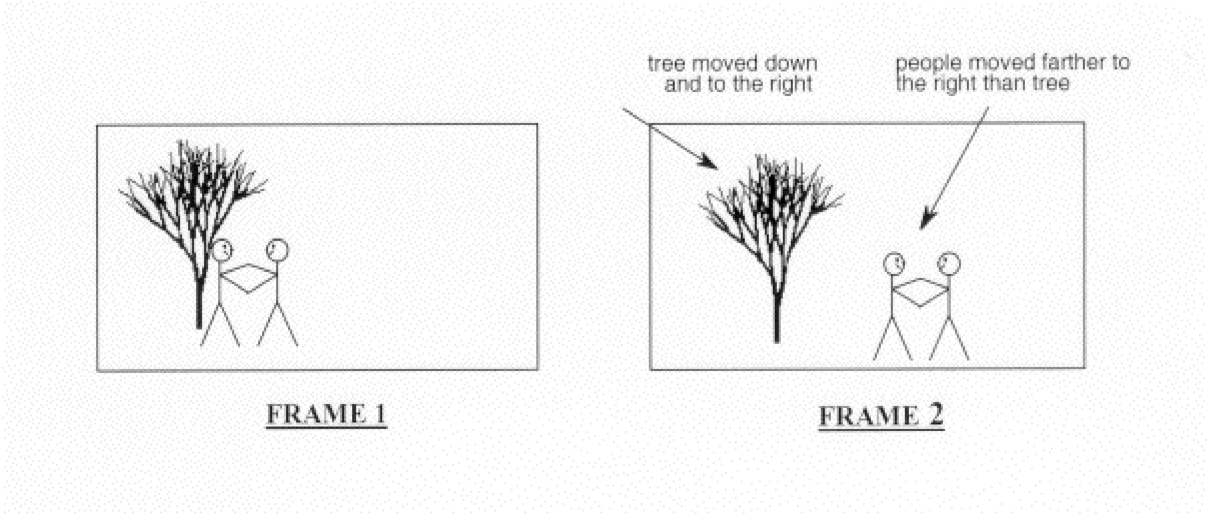
B-Frame Encoding

The main advantage of the usage of B frames is coding efficiency. In most cases, B frames will result in less bits being coded overall. Quality can also be improved in the case of moving objects that reveal hidden areas within a video sequence. Backward prediction in this case allows the encoder to make more intelligent decisions on how to encode the video within these areas. Also, since B frames are not used to predict future frames, errors generated will not be propagated further within the sequence.

One disadvantage is that the frame reconstruction memory buffers within the encoder and decoder must be doubled in size to accommodate the 2 anchor frames. This is almost never an issue for the relatively expensive encoder, and in these days of inexpensive DRAM it has become much less of an issue for the decoder as well. Another disadvantage is that there will necessarily be a delay throughout the system as the frames are delivered out of order as was shown in Figure ❑. Most one-way systems can tolerate these delays, as they are more objectionable in applications such as video conferencing systems.

Motion Estimation

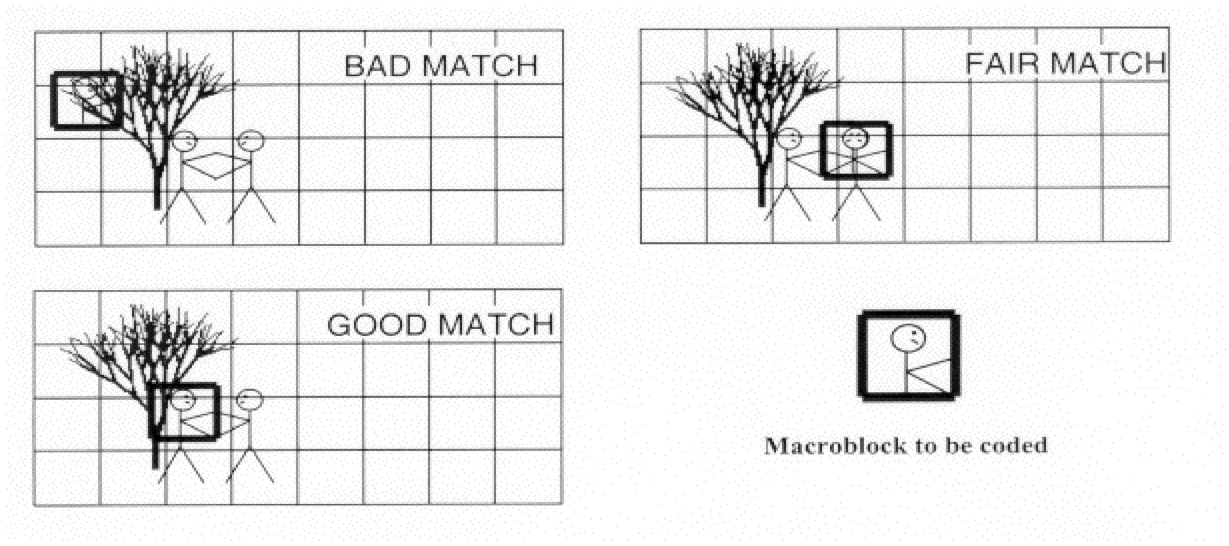
The temporal prediction technique used in MPEG video is based on motion estimation. The basic premise of motion estimation is that in most cases, consecutive video frames will be similar except for changes induced by objects moving within the frames. In the trivial case of zero motion between frames (and no other differences caused by noise, etc.), it is easy for the encoder to efficiently predict the current frame as a duplicate of the prediction frame. When this is done, the only information necessary to transmit to the decoder becomes the syntactic overhead necessary to reconstruct the picture from the original reference frame. When there is motion in the images, the situation is not as simple. Figure Below shows an example of a frame with 2 stick figures and a tree. The second half of this figure is an example of a possible next frame, where panning has resulted in the tree moving down and to the right, and the figures have moved farther to the right because of their own movement outside of the panning. The problem for motion estimation to solve is how to adequately represent the changes, or differences, between these two video frames.


tree moved down and to the right
people moved farther to the right than tree
FRAME 1
FRAME 2

Motion Estimation Example

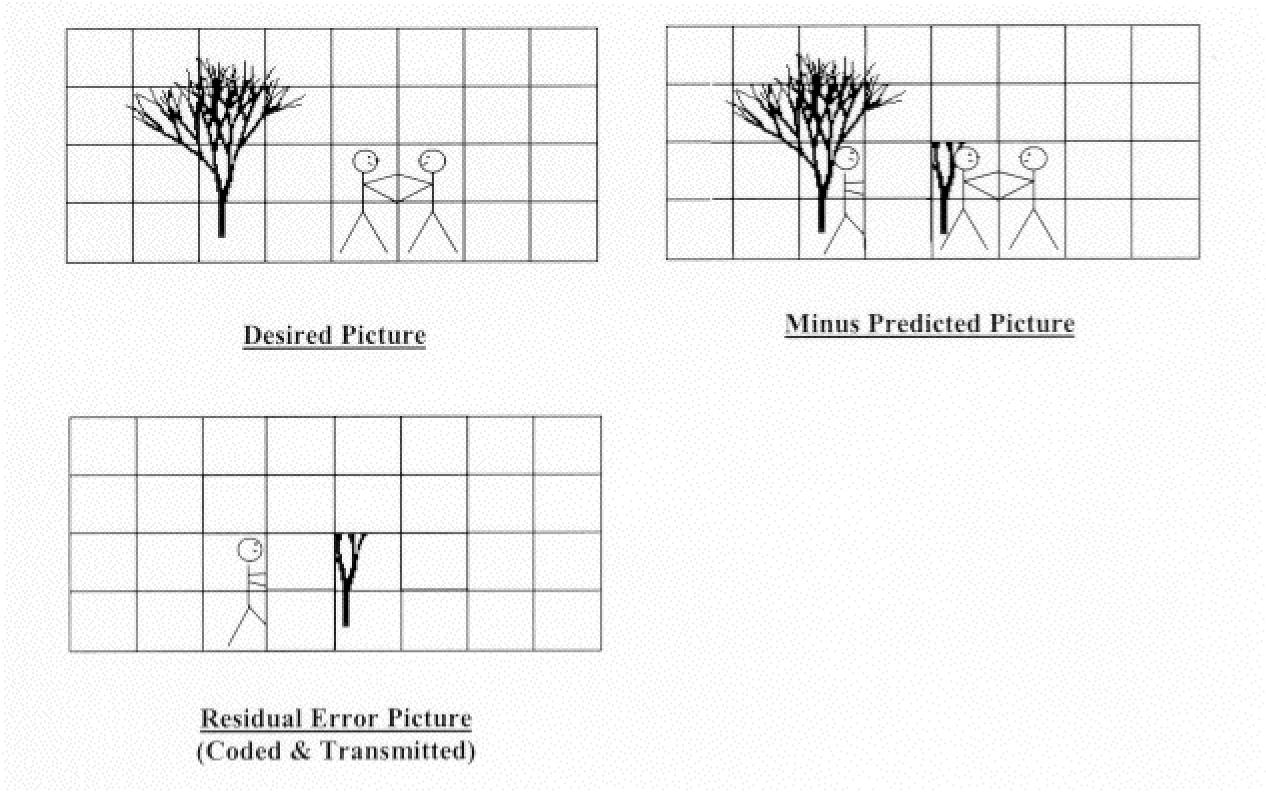
The way that motion estimation goes about solving this problem is that a comprehensive 2-dimensional spatial search is performed for each luminance macroblock. Motion estimation is not applied directly to chrominance in MPEG video, as it is assumed that the color motion can be adequately represented with the same motion information as the luminance. It should be noted at this point that MPEG does not define how this search should be performed. This is a detail that the system designer can choose to implement in one of many possible ways. This is similar to the bit-rate control algorithms discussed previously, in the respect that complexity vs. quality issues need to be addressed relative to the individual application. It is well known that a full, exhaustive search over a wide 2-dimensional area yields the best matching results in most cases, but this performance comes at an extreme computational cost to the encoder. As motion estimation usually is the most computationally expensive portion of the video encoder, some lower cost encoders might choose to limit the pixel search range, or use other techniques such as telescopic searches, usually at some cost to the video quality.

Figure 7.18 shows an example of a particular macroblock from Frame 2 of Figure 7.17, relative to various macroblocks of Frame 1. As can be seen, the top frame has a bad match with the macroblock to be coded. The middle frame has a fair match, as there is some commonality between the 2 macroblocks. The bottom frame has the best match, with only a slight error between the 2 macroblocks. Because a relatively good match has been found, the encoder assigns motion vectors to the macroblock, which indicate how far horizontally and vertically the macroblock must be moved so that a match is made. As such, each forward and backward predicted macroblock may contain 2 motion vectors, so true bidirectionally predicted macroblocks will utilize 4 motion vectors.



Motion Estimation Macroblock Example

Figure 7.19 shows how a potential predicted Frame 2 can be generated from Frame 1 by using motion estimation. In this figure, the predicted frame is subtracted from the desired frame, leaving a (hopefully) less complicated residual error frame that can then be encoded much more efficiently than before motion estimation. It can be seen that the more accurate the motion is estimated and matched, the more likely it will be that the residual error will approach zero, and the coding efficiency will be highest. Further coding efficiency is accomplished by taking advantage of the fact that motion vectors tend to be highly correlated between macroblocks. Because of this, the horizontal component is compared to the previously valid horizontal motion vector and only the difference is coded. This same difference is calculated for the vertical component before coding. These difference codes are then described with a variable length code for maximum compression efficiency.



Final Motion Estimation Prediction

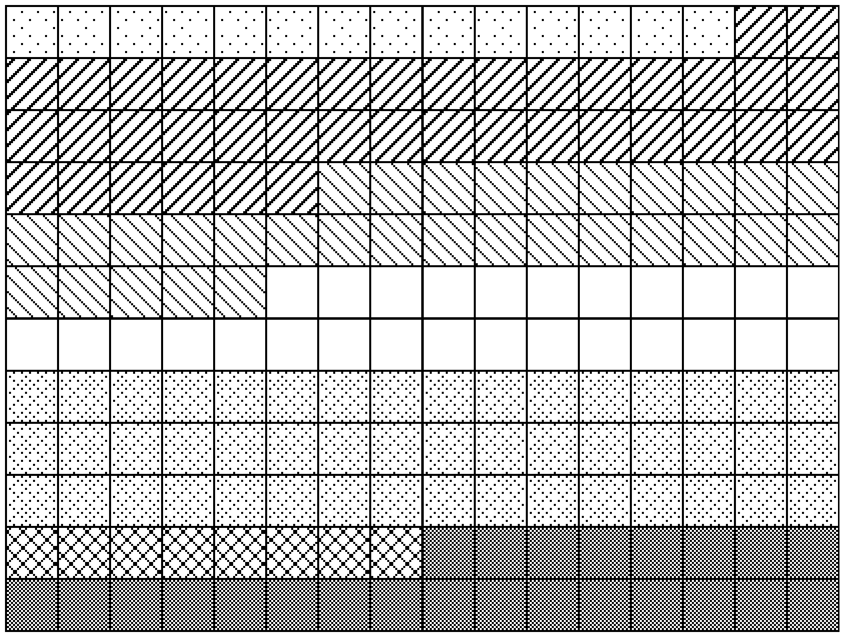
Of course not every macroblock search will result in an acceptable match. If the encoder decides that no acceptable match exists (again, the "acceptable" criterion is not MPEG defined, and is up to the system designer) then it has the option of coding that particular macroblock as an intra macroblock, even though it may be in a P or B frame. In this manner, high quality video is maintained at a slight cost to coding efficiency.

Coding of Predicted Frames: Coding Residual Errors

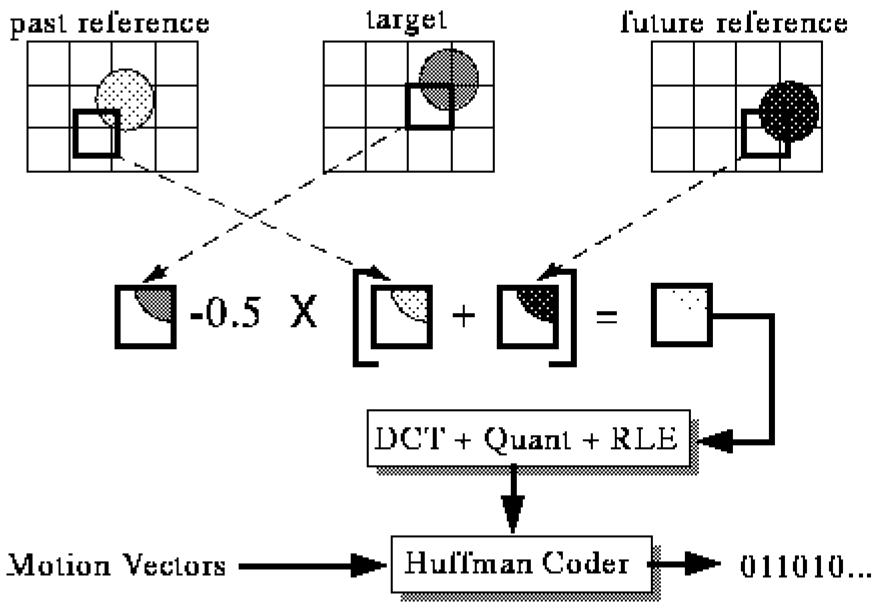
After a predicted frame is subtracted from its reference and the residual error frame is generated, this information is spatially coded as in I frames, by coding 8x8 blocks with the DCT, DCT coefficient quantization, run-length/amplitude coding, and bitstream buffering with rate control feedback. This process is basically the same with some minor differences, the main ones being in the DCT coefficient quantization. The default quantization matrix for non-intra frames is a flat matrix with a constant value of 16 for each of the 64 locations. This is very different from that of the default intra quantization matrix which is tailored for more quantization in direct proportion to higher spatial frequency content. As in the intra case, the encoder may choose to override this default, and utilize another matrix of choice during the encoding process, and download it via the encoded bitstream to the decoder on a picture basis. Also, the non-intra quantization step function contains a dead-zone around zero that is not present in the intra version. This helps eliminate any lone DCT coefficient quantization values that might reduce the run-length amplitude efficiency. Finally, the motion vectors for the residual block information are calculated as differential values and are coded with a variable length code according to their statistical likelihood of occurrence.

Differences from H.261

- Larger gaps between I and P frames, so expand motion vector search range.
- To get better encoding, allow motion vectors to be specified to fraction of a pixel (1/2 pixels).
- Bitstream syntax must allow random access, forward/backward play, etc.
- Added notion of *slice* for synchronization after loss/corrupt data. Example: picture with 7 slices:



- B frame macroblocks can specify *two* motion vectors (one to past and one to future), indicating result is to be averaged.



Compression performance of MPEG 1

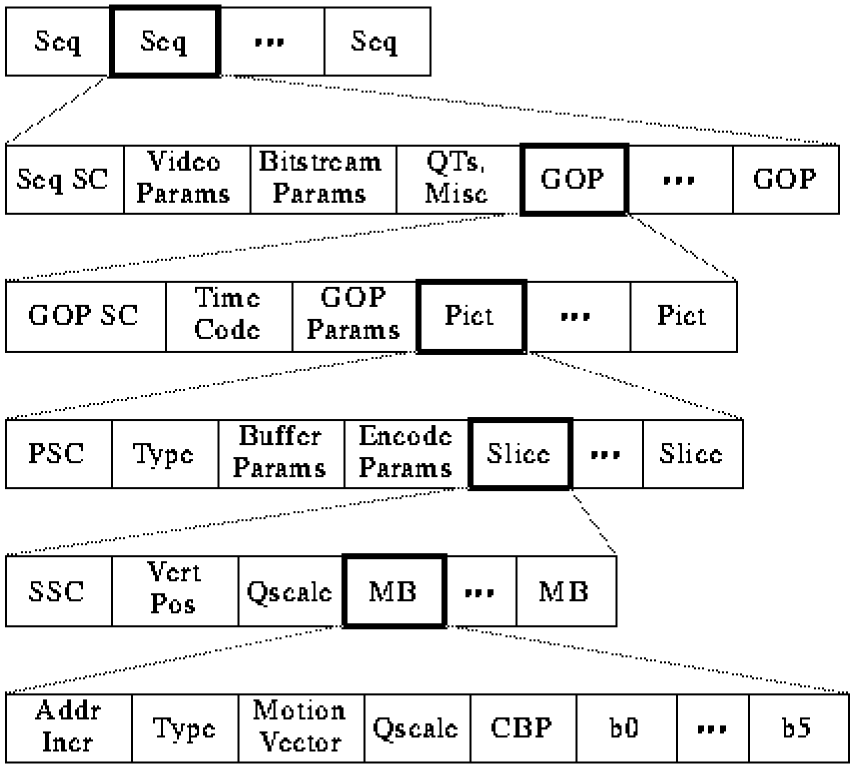
Type	Size	Compression

I	18 KB	7:1
P	6 KB	20:1
B	2.5 KB	50:1
Avg	4.8 KB	27:1

The MPEG Video Bitstream

The MPEG Video Bitstream is summarised as follows:

- Public domain tool **mpeg_stat** and **mpeg_bits** will analyze a bitstream.



- **Sequence Information**

1. Video Params include width, height, aspect ratio of pixels, picture rate.

2.Bitstream Params are bit rate, buffer size, and constrained parameters flag (means bitstream can be decoded by most hardware)

3. Two types of QTs: one for intra-coded blocks (I-frames) and one for inter-coded blocks (P-frames).

Group of Pictures (GOP) information

1. Time code: bit field with SMPTE time code (hours, minutes, seconds, frame).
2. GOP Params are bits describing structure of GOP. Is GOP closed? Does it have a dangling pointer broken?

- **Picture Information**

1. *Type*: I, P, or B-frame?
2. *Buffer Params* indicate how full decoder's buffer should be before starting decode.
3. *Encode Params* indicate whether half pixel motion vectors are used.

- **Slice information**

1. *Vert Pos*: what line does this slice start on?
2. *QScale*: How is the quantization table scaled in this slice?

- **Macroblock information**

1. *Addr Incr*: number of MBs to skip.

2. *Type*: Does this MB use a motion vector? What type?

3. *QScale*: How is the quantization table scaled in this MB?

4. *Coded Block Pattern (CBP)*: bitmap indicating which blocks are coded.

Decoding MPEG Video in Software

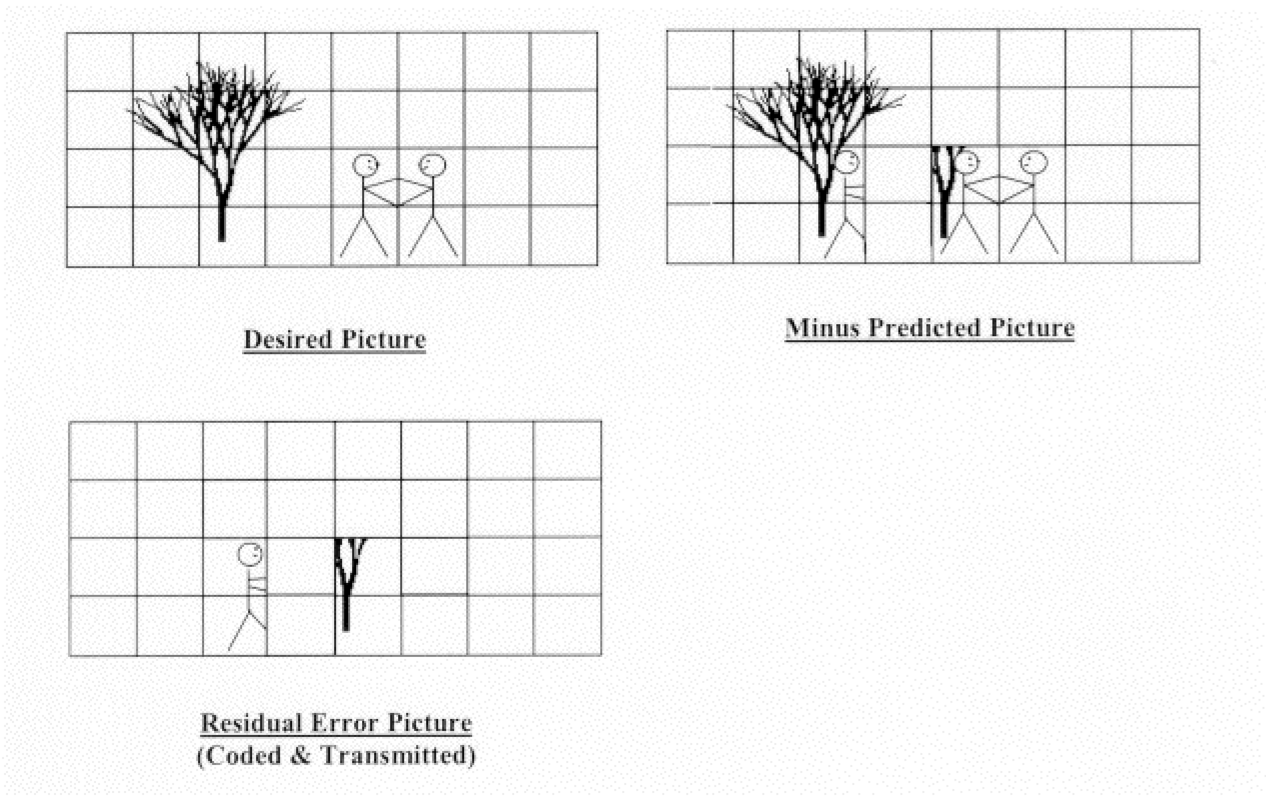
Software Decoder goals: portable, multiple display types

Breakdown of time

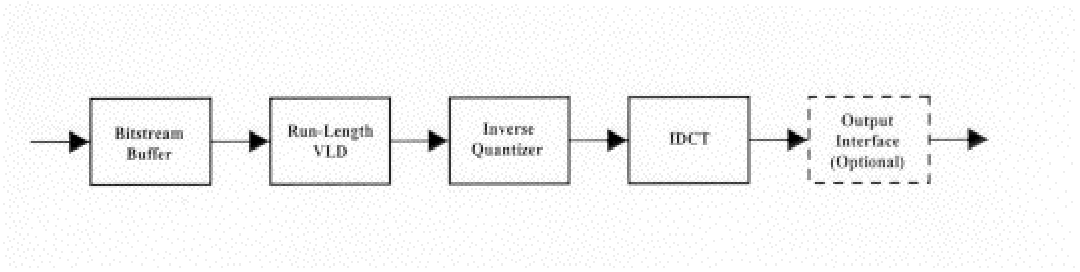
Function	% Time
Parsing Bitstream	17.4%
IDCT	14.2%
Reconstruction	31.5%
Dithering	24.5%
Misc. Arith.	9.9%
Other	2.7%

Intra Frame Decoding

To decode a bitstream generated from the encoder of Figure 7.20, it is necessary to reverse the order of the encoder processing. In this manner, an I frame decoder consists of an input bitstream buffer, a Variable Length Decoder (VLD), an inverse quantizer, an Inverse Discrete Cosine Transform (IDCT), and an output interface to the required environment (computer hard drive, video frame buffer, etc.). This decoder is shown in Figure.



Intra Frame Encoding



Intra Frame Decoding

The input bitstream buffer consists of memory that operates in the inverse fashion of the buffer in the encoder. For fixed bit-rate applications, the constant rate bitstream is buffered in the memory and read out at a variable rate depending on the coding efficiency of the macroblocks and frames to be decoded.

The VLD is probably the most computationally expensive portion of the decoder because it must operate on a bit-wise basis (VLD decoders need to look at every bit, because the boundaries between variable length codes are random and non-aligned) with table look-ups performed at speeds up to the input bit-rate. This is generally the only function in the receiver that is more complex to implement than its corresponding function within the encoder, because of the extensive high-speed bit-wise processingnecessary.

The inverse quantizer block multiplies the decoded coefficients by the corresponding values of the quantization matrix and the quantization scale factor. Clipping of the resulting coefficients is performed to the region 2048 to +2047, then an IDCT mismatch control is applied to prevent long term error propagation within the sequence.

The IDCT operation is given in Equation 2, and is seen to be similar to the DCT operation of Equation 1. As such, these two operations are very similar in implementation between encoder and decoder.

Non-Intra Frame Decoding

It was shown previously that the non-intra frame encoder built upon the basic building blocks of the intra frame encoder, with the addition of motion estimation and its associated support structures. This is also true of the non-intra frame decoder, as it contains the same core structure as the intra frame decoder with the addition of motion compensation support. Again, support for intra frame decoding is inherent in the structure, so I, P, and B frame decoding is possible. The decoder is shown in Figure 24.

MPEG-2, MPEG-3, and MPEG-4

MPEG-2 target applications

Level	size	Pixels/sec	bit-rate (Mbits)	Application
Low	352 x 240	3 M	4	consumer tape equiv.
Main	720 x 480	10 M	15	studio TV
High 1440	1440 x 1152	47 M	60	consumer HDTV
High	1920 x 1080	63 M	80	film production

Differences from MPEG-1

1. Search on fields, not just frames.

2. 4:2:2 and 4:4:4 macroblocks

3. Frame sizes as large as 16383 x 16383

4. Scalable modes: Temporal, Progressive,...

5. Non-linear macroblock quantization factor

6. A bunch of minor fixes (see MPEG FAQ for more details)

- MPEG-3: Originally for HDTV (1920 x 1080), got folded into MPEG-2

MPEG-4: Originally targeted at very low bit-rate communication (4.8 to 64 kb/sec). Now addressing video processing...

Identification Codes

The purpose of this article is to learn about some of the most common identification numbers and check digit algorithms involved in the verification of these identification numbers. We will not be covering all of the identification numbers as there are a lot many out there to be covered in this article. However, once you go through this article you will understand the most common algorithms involved.

Many products that we use have an identification number that may or may not have a bar-code. Some examples are books, electronics, grocery items, credit cards, money orders, driver’s license, etc. The identification number helps encode the information about the product. These numbers are usually separated by a space or a hyphen and each part holds specific information about the product. We will cover the most common ones in this article but before that we should have some basic understanding of how the identification number is verified.

Check Digits

A check digit is added to the identification number (usually the last digit). This digit is used to verify the identification number for its legitimacy. Check digit is added to the number to detect any errors made while typing the number into the system. The check digit is calculated with an algorithm. Some most common algorithms are mod9, mod10, and mod11. We will further notice that the mod10 algorithm is the most common and is used in most of the identification numbers.

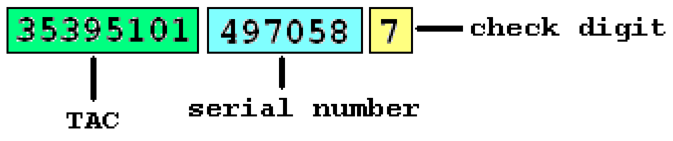
Now will discuss some of the common identification numbers used.

IMEI (International Mobile Equipment Identity)

The IMEI number is a unique 15 digit number used to identify mobile phones as well as some of the satellite phones. The IMEI number can be found on the box in which the phone was packed, inside the phone in the battery compartment, and you can even find the IMEI by typing *#06# or *#0000# on some phones. The IMEI number is used by the GSM network to identify a valid device.

Structure of IMEI

The IMEI number is a 15 digit number (14 digit plus the last digit which is the check digit). The IMEI contains the origin, model, and the serial number of the device plus the check digit for validation. The first eight digits, known as the **TAC (Type Allocation Code)**, hold the information about the origin and model. The next six digits are the **serial number** defined by the manufacturer. The last digit is the **check digit**. Figure 1.1 illustrates the structure of an IMEI number.



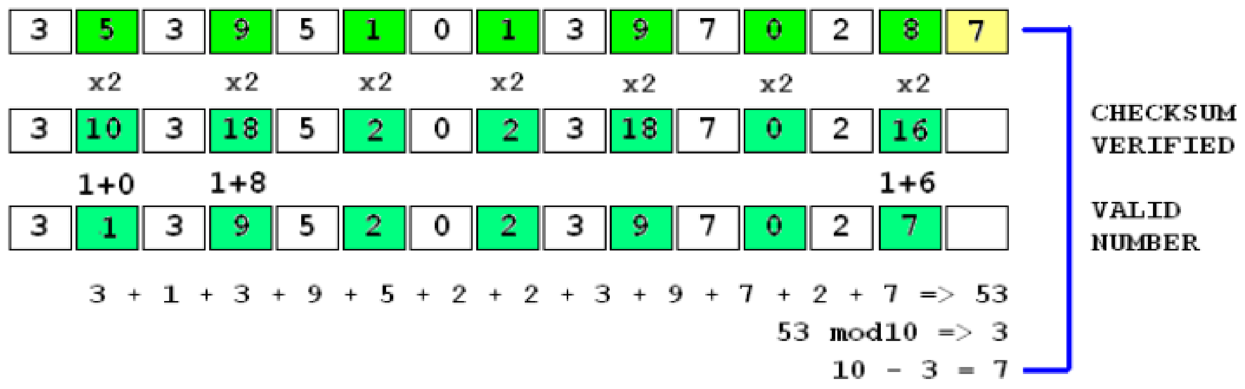
Calculation of the Check Digit

The check digit is calculated using the LUHN’s algorithm (mod10 algorithm). The LUHN’s algorithm was created by Hans Peter Luhn, a scientist at IBM. Here are the steps to calculate the check digit using mod10 algorithm:

1. Starting from the right, double every second digit.
2. Add the digits together if the doubling gives you a two digit number.
3. Now add the doubled digits with the digits that were not doubled.

4. Divide the sum by 10 and check if the remainder is zero. If the remainder is zero then that is the check digit. If the number is not zero, then subtract the remainder from 10. The resulting number will be the check digit.

Here is an illustration:



Summary

- IMEI is a unique 15 digit number assigned to mobile phones.
- It is used by GSM networks to verify the legitimacy of the device.
- First eight digits are known as TAC (Type Allocation Code), next six digits are the serial number, and the last digit is the check digit.
- IMEI number uses mod10 or LUHN's algorithm to verify the number.

Bank Card numbers

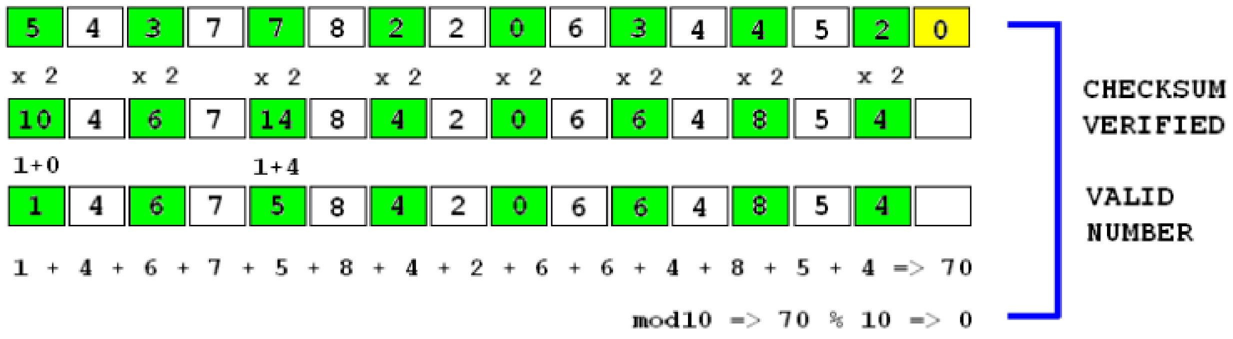
Bank card number are found on credit, debit, and other cards issued from the bank and some gift cards can also be verified with Luhn's algorithm. The first digit of the card number is the **Major Industry Identified (MII)**, which tells us which category of the entity issued the card. For example, if the number begins with 1 or 2, it's issued by Airlines industry. If the number begins with 4, 5, or 6, it is issued by the financial and banking industry. The first six digits of the card number (including the

MII) is known as the **Issuer Identification number (IIN)**. Examples - 4 stands for Visa, 51 or 55 stand for MasterCard, and 34 or 37 for American Express. The numbers left are issued by the bank and the last digit is the check digit.

Verifying the card number using mod10 algorithm

Mod10 algorithm is used to verify bank card numbers. We have already discussed this algorithm, so we will not go into the details again.

Here is an illustration:



Routing Numbers

Routing number is a nine digit bank code designed to facilitate the sorting, bundling, and shipment of paper checks back to the drawer's account. The RTN is also used by Federal Reserve Bank to process Fedwire fund transfers, and by the Automatic Clearing house to process direct deposits and other automatic transfers.

Routing number format

Routing number appears in two formats:) Fraction form and MICR (Magnetic ink character recognition) form. Both forms give the same information. Fraction form was used when MICR form was not invented, however it is still used as a backup.

Calculation of the Check Digit

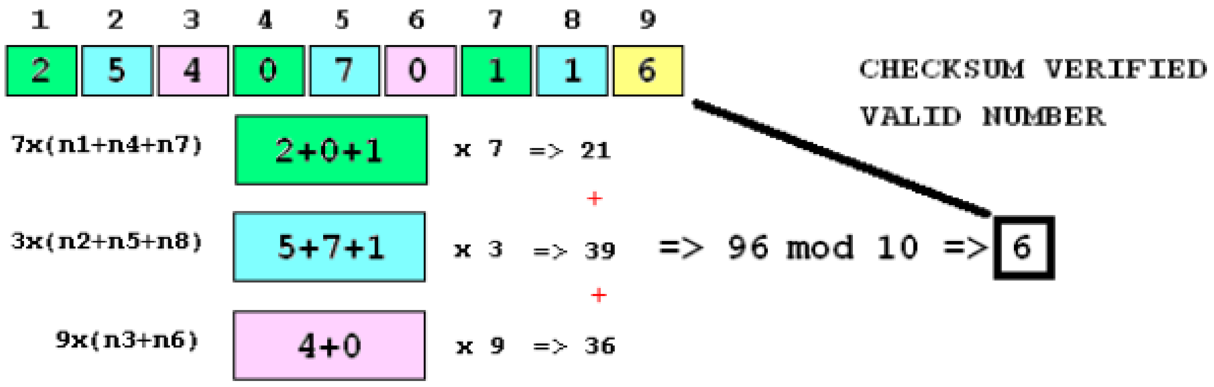
The check digit can be calculated by using this formula:

check digit (d9) = [7x(d1 + d4 + d7) + 3(d2 + d5+ d8) + 9(d3 + d6)] mod 10

Let's calculate the check digit for this routing number: 2540 7011 6. Here, the check digit is 6. According to the formula, check digit (d9) = 7x(2+0+1) + 3x(5+7+1) + 9x(4+0)

=> 7x3 + 3x13 + 9x4 => 21 + 39 + 36 => 96. Now, 96 mod 10 => 6. Hence 6 is the check digit.

Here is an illustration:



USPS money order number

USPS money order number

The US Postal office uses an identification number for postal orders. It’s an 11 digit number and the last number is the check digit as we have seen in other cases.

Calculation of the check digit

To calculate the check digit, add up the first 10 digits and the sum is divided by 9. The remainder is the check digit. Let's calculate the check digit for **84310325021**. Check digit => 8+4+2+1+0+3+2+5+0+2+1 => 28 mod 9 => 1. Hence 1 is the check digit.

International Standard Book Number (ISBN)

This is a unique number created by Gordon Foster in 1961. The 10 digit format was developed by ISO (International Organization for Standardization). An ISBN is assigned to each edition of a book.

- 10 digit is assigned before Jan 1, 2007 and 13 digits is assigned after that.
- Three parts:

1. the group identifier
2. the publisher code
3. the item number (title of the book).

- Separated by spaces or hyphen.
- Group Identifier: 1-5 digits (country, language).
- The Publisher code: The national ISBN agency assigns the publisher number.
- The publisher selects the item number.

Example: 9971-5-0210-0, 0-943396-04-2, 0-85131-041-9

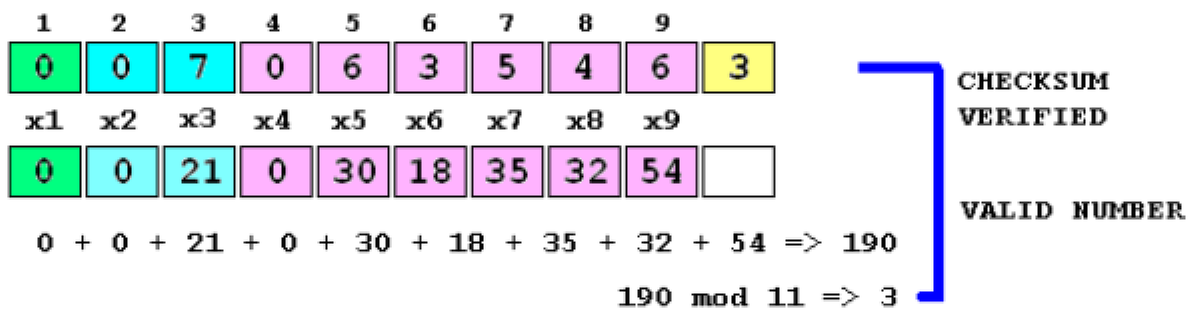
Calculation of the check digit

- **ISBN check digit (10 digits) - mod11 algorithm**

The last digit in an ISBN is the check digit, must range from 0 to 10. The ISBN uses a weighted system of checking. Each digit from left to right is assigned a weight from ten to one. Each digit is multiplied by its position weight and the resulting numbers are summed.

Let's calculate the check digit for **0-07-063546-3**.

Hence 3 is the check digit. Here is an illustration.

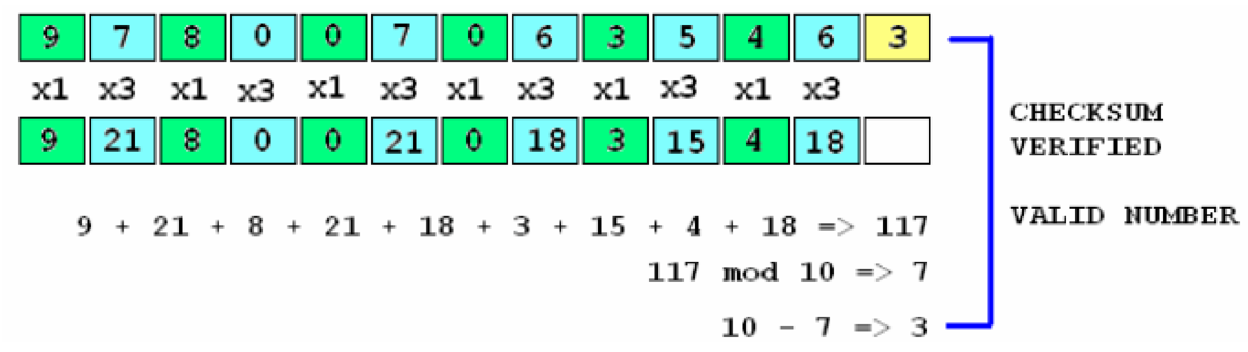


- **ISBN check digit (13 digits)**

Each digit, starting from the left to right, is multiplied by 1 or 3 alternatively. The sum of the products modulo 10 gives us either zero or a number between 1 to 9. Subtract the number from 10 and it gives us the checksum.

Hence 3 is the check digit.

Here is an illustration:



ISSN (International Standard Serial Number)

An ISSN is a unique eight number used to identify a print or electronic periodical publication. The code format is divided by a hyphen into a four digit number. The last number is the check digit as in the other codes that we have covered.

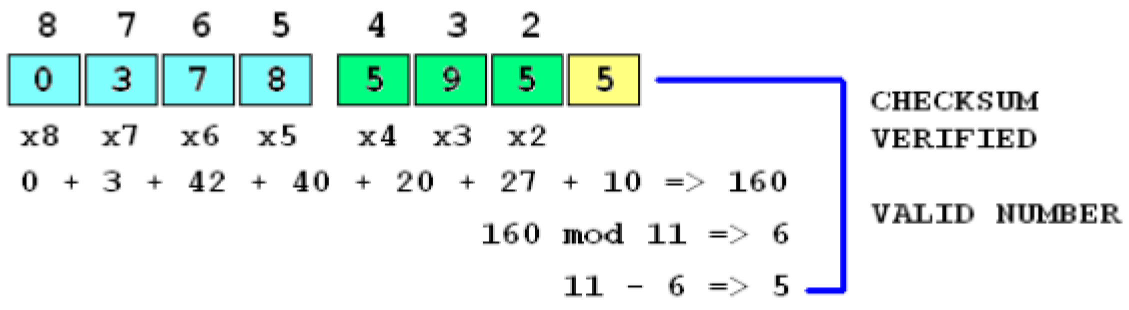
Calculation of the check digit

Starting from the left, each digit is multiplied by its position in the number. Add those numbers and the sum is divided by 11 (mod11). If the remainder is not zero, then the remainder is subtracted from 11 and that gives us the check digit. So for example, this number – **0378-5955**.

Leave out the last digit because we want to verify this digit: 0x8 + 3x7 + 7x6 + 8x5 + 5x4 + 9x3 + 5x2 = >160 % 11 = 6.

Now because the remainder is a non-zero digit, we will subtract it from 11 to get the check digit, so 11 - 6 => 5. Hence the check digit is 5.

Here is an illustration:



UPC and EAN

The UPC (Universal Product Code) is a barcode symbol and is used to track trade items in stores. The most common form of UPC is the UPC-A which consists of 12 digits which is unique for a trade item. It consists of a strip of black and white spaces which can be scanned. The area that can be scanned in a UPC-A follows this pattern:

SLLLLLLMRRRRRRE

Here S -> Start, M -> Middle and E -> End

L -> Left and R -> Right make the barcode unique. The last digit in the barcode is the check digit.

Calculation and Verification of the check digit

Verification: To verify the number, we can use this formula:

[3.d1 + 1.d2 + 3.d3 + 1.d4 + 3.d5 + 1.d6 + 3.d7 + 1.d8 + 3.d9 + 1.d10 + 3.d11 + 1.d12] mod10 = 0`````

Here d1, d2, d3...etc. are the digits. Starting from the left, we multiply the digits with 3 and 1 alternatively.

Example: 036000 291452

3x0 + 1x3 + 3x6 + 1x0 + 3x0 + 1x0 + 3x2 + 1x9 + 3x1 + 1x4 + 3x5 + 1x2

=> 0+3+18+0+0+0+9+3+4+15+2 => 60 => 60mod10 => 0.

Hence the number is verified:

Calculation: To calculate the check digit, we use the same formula but subtract the remainder from 10 to get the check digit.

Example: 036000 29145?

3x0 + 1x3 + 3x6 + 1x0 + 3x0 + 1x0 + 3x2 + 1x9 + 3x1 + 1x4 + 3x5 + x

=> 0+3+18+0+0+0+9+3+4+15+x => 58 => 58 mod10 => 8

10 - 8 => 2

Hence 2 is the check digit.

EAN

The EAN-13 (European Article Number) is a 13 digit barcode which is a superset of the UPC (12 digits), and is used worldwide for marking products sold at retail points of sale (POS). EAN also indicates the country in which the company who sells the product is based in.

Calculation of the check digit

Verification: To verify the number, multiply the digits with 1 or 3 with respect to the position they have in the digits, starting from the left.

Example: 8 901526 206056

1x8 + 3x9 + 1x0 + 3x1 + 1x5 + 3x2 + 1x6 + 3x2 + 1x0 + 3x6 + 1x0 + 3x5+ 1x6 => 8 + 27 + 3 + 5 + 6 + 6 + 6 + 18 + 15 + 6 => 100 mod10 => 0. Hence number is verified

Calculation: We use the same method as above, however we will omit the last digit from the calculation because that is the digit we want to find. Here if the remainder is a non-zero number then it is subtracted from 10.

Example: 8 901526 206056

1x8 + 3x9 + 1x0 + 3x1 + 1x5 + 3x2 + 1x6 + 3x2 + 1x0 + 3x6 + 1x0 + 3x5
=> 8 + 27 + 3 + 5 + 6 + 6 + 6 + 18 + 15 => 94 mod10 => 6. 10 - 4 = > 6.

Hence 6 is the check digit.

REF:

[1] "Data Compression: The Complete Reference", by David Salomon, Fourth Edition, 2007, Springer.

[2] "Fundamentals of Multimedia", by: Li and Drew, 2004, Prentic Hall.

https://www.ece.ucdavis.edu/cerl/reliablejpeg/compression/