Metodologías de Compresión, Restauración y Reconstrucción de imagen:

Compresión de Imágenes y Vídeo

Vicente González Ruiz
José Jesús Fernández Rodríguez

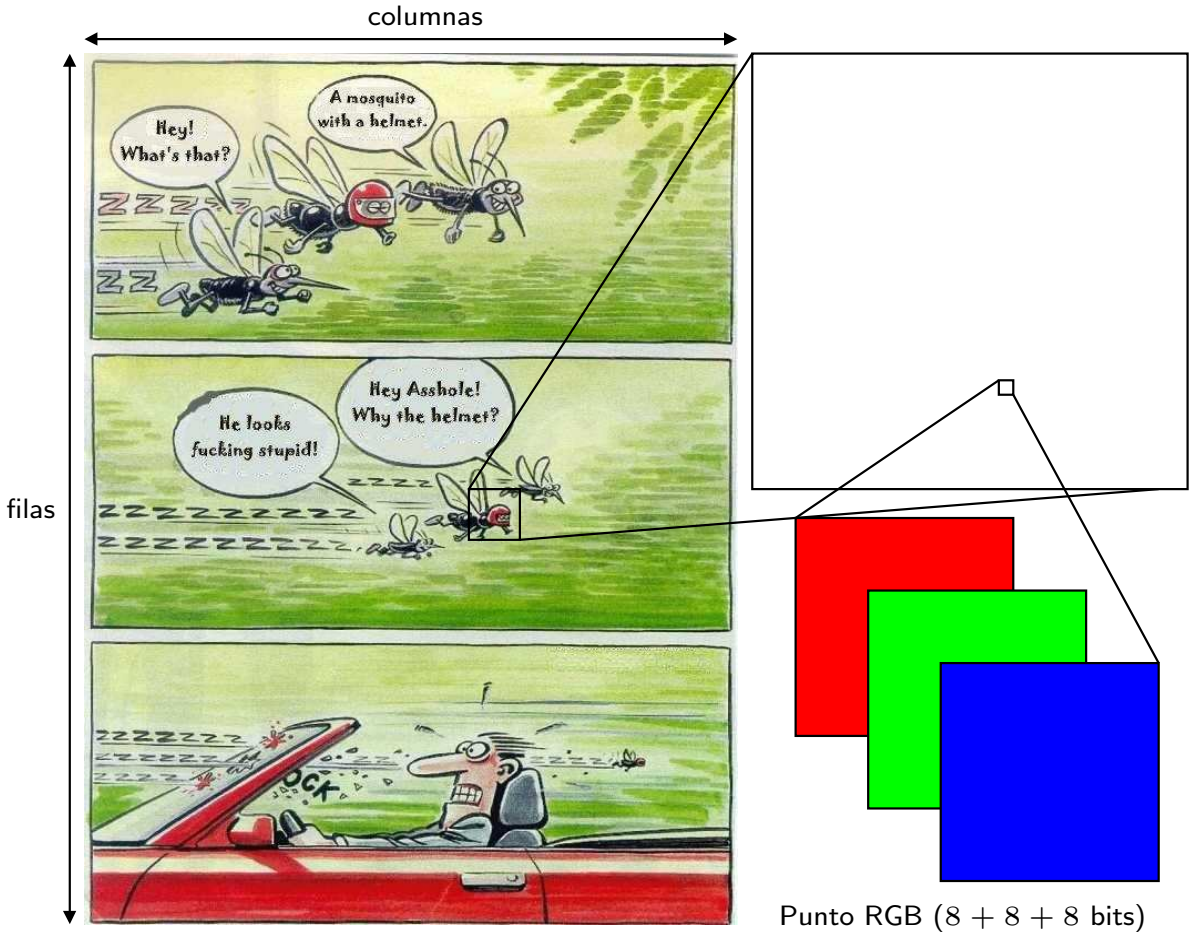
Contenidos

Capítulo 1

La compresión de imágenes y vídeo

1.1. ¿Qué es una imagen?

- En el contexto digital, una imagen es una matriz discreta de puntos que poseen un determinado número de componentes, cada una con una precisión concreta.



- El número de bits I que una imagen PCM ocupa está determinado por:

$$I = X \times Y \times C \times B$$

donde X es el ancho de la imagen, Y el alto, C el número de componentes y B el número de bits de cada componente. Por ejemplo, una imagen típica de televisión necesita:

$$625 \times 468 \times 3 \times 8 = 7.020.000 \text{ bits}$$

1.2. ¿Qué es un vídeo?

- Es una secuencia de imágenes que representada a una determinada velocidad proporciona sensación de movimiento.*
- El número de bits V que un vídeo digital ocupa está determinado por I y por el número N de imágenes que lo componen:

$$V = I \times N$$

Por ejemplo, un segundo de vídeo con calidad TV (25 imágenes/segundo) ocupa:

$$7.020.000 \ \frac{\text{bits}}{\text{imagen}} \times 25 \ \frac{\text{imágenes}}{\text{segundo}} = 175.500.000 \text{ bits}$$

*Nótese que esta definición no es exáctamente igual a la que haríamos de una señal de vídeo que podría contener información adicional como las señales de sincronismo para controlar el display.

1.3. ¿Por qué comprimir ?

- Las imágenes (y por supuesto las secuencias de éstas) se comprimen para reducir el número de bits que son necesarios para representarlas.
- La compresión de imágenes se lleva a cabo normalmente cuando se almacenan o cuando se transmiten. En el primer caso se ahorra espacio de almacenamiento y en el segundo ancho de banda.
- ¿Qué ocurre en la práctica?:
  1. Las imágenes en color se comprimen de forma reversible a una tasa promedio de 2:1*, y de forma irreversible (aunque visualmente sin pérdidas) a una tasa promedio de 20:1.
  2. Los vídeos en color se comprimen de forma irreversible (aunque visualmente sin pérdidas) a una tasa típica de 100:1.

*Esto quiere decir que el formato PCM no es tan redundante como pudiera parecer.

1.4. ¿Qué tipos de compresores existen?

- A la hora de comprimir imágenes podemos escoger entre los siguientes tipos de compresores:

  1. **Compresores de texto:** Se utilizan para comprimir cualquier tipo de fuente de información, no sólo las imágenes.
  2. **Compresores de imágenes:** Están específicamente diseñados para comprimir imágenes.
  3. **Compresores de vídeo:** Son una generalización de los anteriores donde se conoce de antemano que las imágenes pertenecen a una misma secuencia de vídeo.

1.5. ¿Cuáles son sus principales posibilidades?

- **Lossless/lossy:** Los compresores de texto son todos lossless, lo que significa que después de la descompresión se recuperan los datos originales (bit a bit). En la otra posibilidad (lossy) sólo se recupera una parte de la información. Los compresores de imágenes y de vídeo suelen ser lossy.

- **Tasa de compresión:** Debido a la posibilidad de eliminar la información visualmente menos relevante, los compresores de imágenes y de vídeo alcanzan tasas de compresión muy superiores a las obtenidas por los compresores de texto.

- **Accesibilidad a los datos:** Otro aspecto importante sobre el uso de un compresor/descompresor (codec) es la forma en que deben acceder a los datos comprimidos y expandidos. Veremos que esto constituye un factor determinante en determinadas aplicaciones.

1.6. Escalabilidad

Cuando descomprimimos secuencias de imágenes, hablaremos de las siguientes formas de acceder a los datos:

1. **Por niveles de resolución espacial (escalabilidad espacial):** Si sub-muestreamos (adecuadamente, sin provocar aliasing) una imagen, podemos representarla con un nivel de resolución inferior al que originalmente tenía.

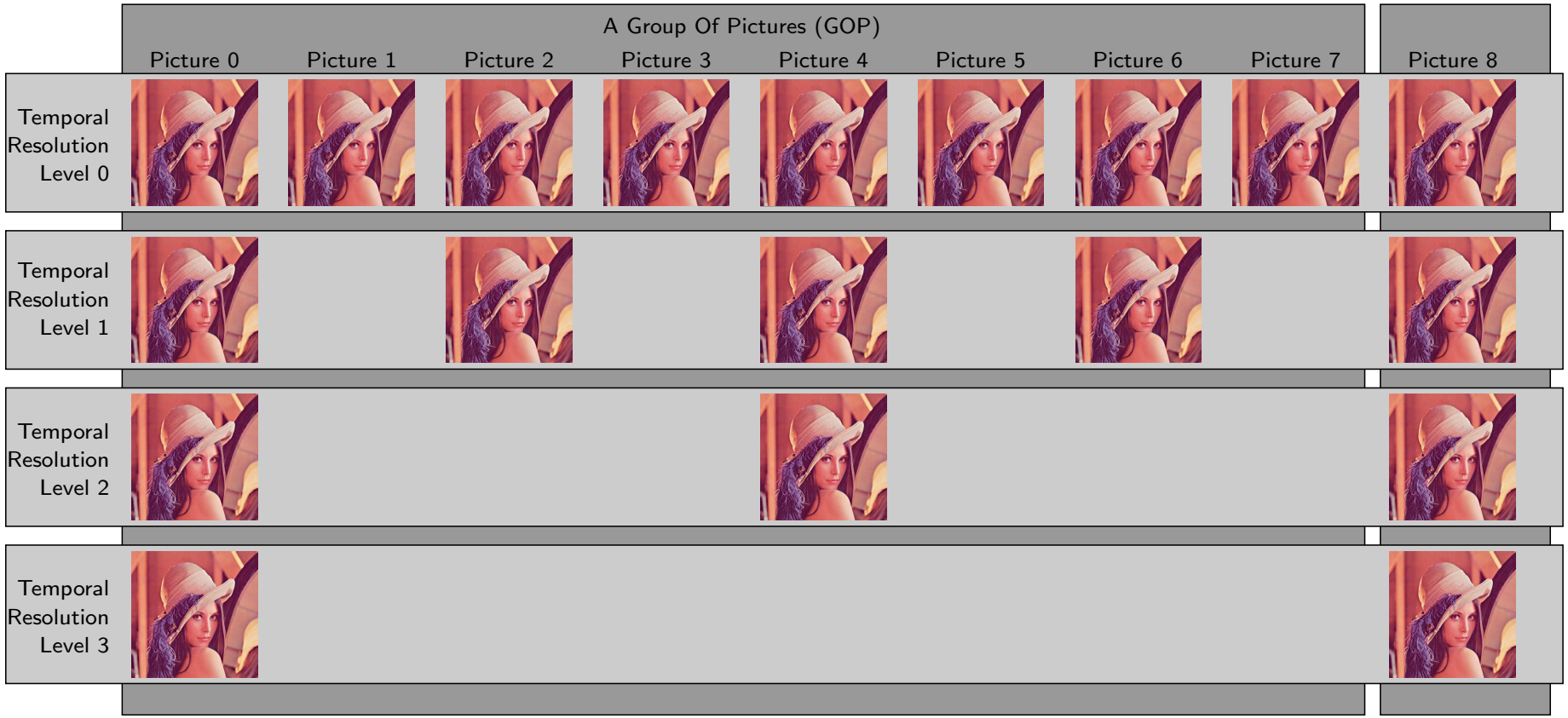
2. **Por niveles de calidad (escalabilidad en calidad o distorsión):** La mayoría de los compresores de imágenes se comportan como filtros de forma que sólo algunas componentes de frecuencia son obtenidas tras descomprimir (normalmente las más importantes visualmente hablando).

3. **Por niveles de resolución temporal (escalabilidad temporal):** En las secuencias de imágenes, podemos seleccionar qué imágenes van a ser mostradas.

4. **Por regiones de interés/objetos (escalabilidad por ROI o objetos):** Finalmente, en una imagen podemos descomprimir con mayor calidad una región o un determinado objeto.

Escalabilidad espacial: ejemplo

Escalabilidad temporal: ejemplo



Escalabilidad en calidad: ejemplo

Escalabilidad por ROI/objeto: ejemplo

Parte I

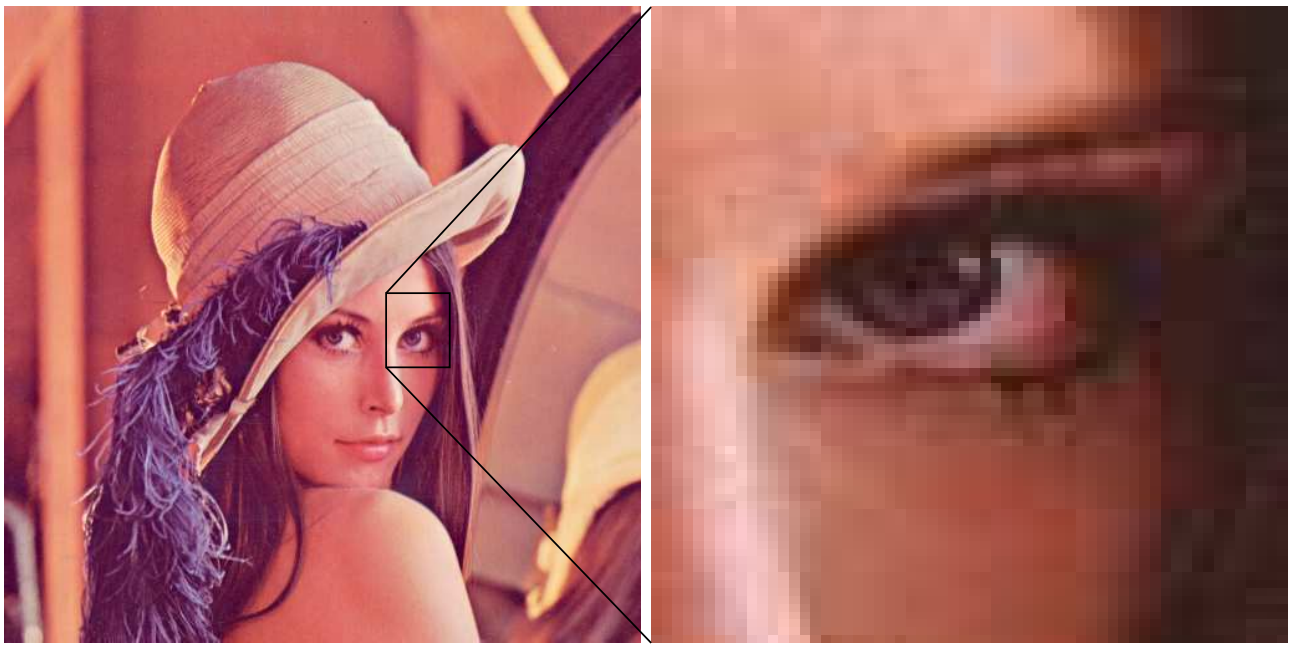
Compresión de texto

Capítulo 2

Fundamentos

2.1. Las fuentes de redundancia

- Los compresores de texto eliminan la **redundancia estadística** en las secuencias de datos. Esta es causada porque los diferentes símbolos que las forman generalmente no son equiprobables y además poseen algún tipo de correlación.



- Los compresores de texto eliminan la redundancia estadística y por tanto son totalmente **reversibles**.

2.2. Símbolos, series y cadenas

- Los compresores de texto procesan las secuencias de datos por trozos. Dependiendo del tipo de compresor hablaremos de:
  - **Símbolos:** típicamente bytes. Patrones de bits de longitud fija.
  - **Series de símbolos:** de longitud variable. Todos los símbolos deben ser iguales.
  - **Cadenas de símbolos:** de longitud variable. Los símbolos pueden ser cualesquiera.

Capítulo 3

Compresión de series (RLE)

La compresión de series

- RLE (Run Length Encoding).
- Elimina la redundancia debida a la repetición de símbolos.
- La compresión se produce cuando la codificación de la serie ocupa menos datos que la serie en sí.
- Existen muchas versiones de compresores RLE que se diferencian en el tamaño del alfabeto fuente y en la longitud mínima y máxima que pueden tener las series (runs).

Capítulo 4

RLE básico

- El tamaño del alfabeto fuente es 256 y la longitud mínima de la serie es 1.

4.1. Compresor

1. Mientras existan símbolos por codificar:
   - *a*) Sea s el siguiente símbolo.
   - *b*) Leer los siguientes n símbolos consecutivos iguales a s.
   - *c*) Emitir un par ns.

4.2. Descompresor

1. Mientras existan pares ns que descodificar:

   a) Escribir n símbolos iguales a s.

4.3. Ejemplo de compresión

La secuencia de símbolos:

aaaabbbbbaaaaaabbbbbbbccccccc

se codificarían como:

4a 5b 6a 7b 6c

Capítulo 5

RLE binario

- Cuando sólo existen 2 símbolos diferentes no es necesario indicar el símbolo porque cuando una serie acaba es porque comienza otra con el símbolo alternativo.

5.1. Compresor

1. Sea $s \leftarrow 0$.
2. Mientras existan bits por codificar:
   a) Leer los siguientes n bits consecutivos iguales a s.
   b) Escribir n.
   c) $s \leftarrow (s + 1)$ módulo 2.

5.2. Descompresor

1. Sea $s \leftarrow 0$.
2. Mientras existan items n que descodificar:
   a) Escribir n bits iguales a s.
   b) $s \leftarrow (s + 1)$ módulo 2.

5.3. Ejemplo de compresión

La secuencia de símbolos:

000011111000000111111000000

se codificarían como:

4 5 6 7 6

5.3 Ejemplo de compresión

Capítulo 6

RLE con tamaño de serie mínimo

- A continuación se presenta un codec que considera que una serie tiene un tamaño mínimo de 2 símbolos y un tamaño máximo de 255+2.

- En muchas situaciones, donde el tamaño las series de sólo un símbolo (que producen una expansión cuando se les aplica el algoritmo básico), el uso de un RLE con tamaño de serie mínimo conseguirá mejores ratios de compresión.
- Puede encontarse una implementación de un compresor RLE con tamaño de serie mínimo en el Apéndice 39.54.

Capítulo 7

Compresión de cadenas

La compresión de cadenas

- La correlación estadística provoca que los símbolos no se agrupen de cualquier forma. Esto provoca que el número de cadenas que encontramos en una secuencia de símbolos es generalmente menor que el número de cadenas posibles r^l, donde r es el tamaño del alfabeto y l la longitud de las cadenas medida en símbolos.
- Un ejemplo claro los tenemos en los diccionarios de la lengua. Las palabras (cadenas) pueden ser localizados indicando su posición dentro del diccionario. Si dicha referencia ocupa menos bits que la palabra en sí, estaremos comprimiendo.

Capítulo 8

LZ77 (Lempel y Ziv, 1977)

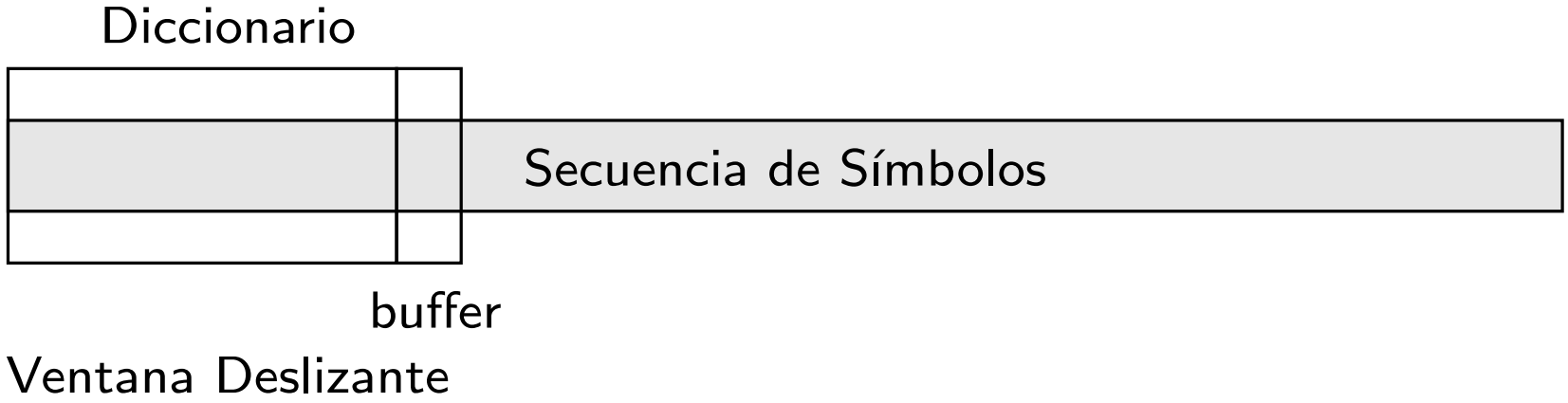
8.1. Antecedentes

- En 1977, Jacob Ziv y Abraham Lempel idearon una técnica de codificación, conocida como LZ77, que explota esta redundancia [25]. LZ77 es la base de compresores de datos tan famosos como gzip, zip y arj.

8.2. Compresor

- Un compresor LZ77 trabaja sustituyendo las cadenas que encuentra a la entrada por la referencia donde la cadena aparece en un diccionario. Si dicha referencia ocupa menos bits que la cadena en sí, se produce una compresión de datos.

- La secuencia de símbolos a codificar se procesa de forma secuencial. Para ello se define una ventana deslizante que consta de dos partes: (1) un diccionario y (2) un "buffer". En el diccionario encontramos las cadenas que ya han sido codificadas y en el buffer, aquellas que van a codificarse.



- El compresor emite un código de compresión formado por una terna de enteros ijk, donde i es la posición de la cadena dentro del diccionario, j es la longitud de la cadena y k es el siguiente símbolo que no forma parte de la cadena. A continuación la ventana deslizante se desplaza $k+1$ posiciones hacia la dirección del fin de la secuencia de símbolos.

- Una descripción del algoritmo de compresion podría ser:

  1. Sea I el tamaño del diccionario y J el tamaño del buffer, ambos medidos en símbolos.
  2. Introducir los primeros J símbolos en el buffer.
  3. Mientras existan símbolos por codificar:
     a) Sea i la posición de los primeros j símbolos del buffer en el diccionario y k el símbolo que hace que j no pueda ser mas grande.
     b) Escribir ijk.
     c) Introducir los siguientes $j+1$ símbolos en el buffer.

8.3. Ejemplo de compresión

	dicc	*buffer*		salida	comentario
		abab	cbababaaaaaa	0 0 a	diccionario vacío
	a	babc	bababaaaaaa	0 0 b	b no encontrado
	ab	abcb	ababaaaaaa	2 2 c	ab encontrado
a	babc	baba	baaaaaa	0 3 a	bab encontrado
ababc	baba	baaa	aaa	0 2 a	ba encontrado
ababcbab	abaa	aaaa		2 3 a	aaa encontrado
	0123				

8.4. Descompresor

- El descompresor utiliza en todo instante un diccionario y un *buffer* idénticos a los del compresor. Su trabajo consiste en emitir los j símbolos extraídos a partir de la posición i del diccionario y concatenar el símbolo k. Con cada código descodificado, la ventana deslizante se deslaza $j+1$ posiciones hacia el fin de la secuencia de símbolos.

- Una descripción del descompresor sería:

  1. Sea I el tamaño del diccionario y J el tamaño del buffer, ambos medidos en símbolos.
  2. Mientras existan ternas ijk por leer:
     a) Escribir los j símbolos extraídos a partir de la posición i del diccionario.
     b) Escribir k.
     c) Introducir todos los símbolos descodificados en el buffer.

8.5. Ejemplo de descompresión

entrada	salida		dicc	*buffer*
0 0 a	a			a
0 0 b	b		a	b
2 2 c	abc		ab	abc
0 3 a	baba	a	babc	baba
0 2 a	baa	ababc	baba	baa
2 3 a	aaaa	ababcbabab	abaa	aaaa
			0123	

8.6. LZSS (LZ77, Storer and Szymanski)

- El algoritmo original propuesto por Lempel y Ziv genera siempre un código de compresión ijk donde k no se codifica (aparece igual en la entrada). Nótese que este problema es especialmente acusado al comienzo de la compresión, cuando la cadena almacenada en el buffer de anticipación comienza por un símbolo que no aparece en el diccionario.

- J. A. Storer y T. G. Szymanski minimizaron este problema añadiendo un código de un bit que sirve de prefijo [18]. Así, si este bit es 1 el descompresor sabe que se trata de un símbolo nuevo y que sólo viene una k. Si es un 0, va a llegar un código ij.

- Puede encontrarse una implementación del algoritmo LZSS en el Apéndice 39.19.

Capítulo 9

LZ78 (Lempel y Ziv, 1978)

9.1. LZ78 vs LZ77

- En 1978, Ziv y Lempel publicaron un nuevo método de compresión de datos que ha tenido también mucho éxito, el LZ78 [26]. Lo realmente interesante de este algoritmo es la representación recursiva que utiliza el diccionario. Este está compuesto por pares wk donde w es una posición dentro del diccionario y k es un símbolo sin codificar.

- El compresor genera códigos wk cada vez que inserta una nueva entrada en el diccionario. De esta manera el descompresor consigue reconstruir una copia exacta del diccionario que usó el compresor en cualquier instante.

- Aunque los códigos wk utilizan más bits que los símbolos originales, el compresor sólo envía un código al descompresor cuando una nueva cadena wk es insertada en el diccionario.

- En realidad, wk representa a la cadena formada por la concatenación de la "cadena" w con el símbolo k. Es importante darse cuenta de que w es físicamente un número entero positivo, aunque representa

a una cadena porque apunta a una dirección dentro del diccionario donde hay otra $w'k'$. Llamaremos a este proceso de concatenación "string(w)", que finaliza cuando $w = 0$ (la entrada de índice 0 en el diccionario, la cadena vacía).

- La principal ventaja de LZ78 frente a LZ77 radica en que la longitud máxima de la cadena encontrada ahora no está limitada por el tamaño del look-ahead buffer, sino por el tamaño del diccionario que suele ser mucho mayor.

9.2. Compresor

1. Sea $w \leftarrow 0$.
2. Mientras existan símbolos por codificar:
   - *a*) $k \leftarrow$ siguiente símbolo de entrada.
   - *b*) Si wk existe en el diccionario, entonces:
     - 1) $w \leftarrow$ dirección de wk en el diccionario.
   - *c*) Si no:
     - 1) Escribir wk a la salida.
     - 2) Insertar wk en el diccionario.
     - 3) $w \leftarrow 0$.

9.3. Ejemplo de compresión

Entrada	Salida	Comentario
a	0a	$D[1] \leftarrow$ a
b	0b	$D[2] \leftarrow$ b
a		$D[1] =$ a
b	1b	$D[3] \leftarrow$ ab
c	0c	$D[4] \leftarrow$ c
b		$D[2] =$ b
a	2a	$D[5] \leftarrow$ ba
b		$D[2] =$ b
a		$D[5] =$ ba
b	5b	$D[6] \leftarrow$ bab
a		$D[1] \leftarrow$ a
a	1a	$D[7] \leftarrow$ aa
a		$D[1] \leftarrow$ a
a		$D[7] \leftarrow$ aa
a	7a	$D[8] \leftarrow$ aaa
a		$D[1] =$ a
a		$D[7] =$ aa
a		$D[8] =$ aaa
a	8a	$D[9] \leftarrow$ aaaa

Dirección	w	k	Cadena
1	0	a	a
2	0	b	b
3	1	b	ab
4	0	c	c
5	2	a	ba
6	5	b	bab
7	1	a	aa
8	7	a	aaa
9	8	a	aaaa

9.4. Descompresor

1. Mientras existan códigos por descodificar:
   a) Leer un código wk.
   b) Escribir string(w) a la salida.
   c) Escribir k a la salida.
   d) Insertar wk en el diccionario.

9.5. Ejemplo de descompresión

Entrada	Salida	Comentario
0a	a	$D[1] \leftarrow$ a
0b	b	$D[2] \leftarrow$ b
1b	ab	$D[3] \leftarrow$ ab
0c	c	$D[4] \leftarrow$ c
2a	ba	$D[5] \leftarrow$ ba
5b	bab	$D[6] \leftarrow$ bab
1a	aa	$D[7] \leftarrow$ aa
7a	aaa	$D[8] \leftarrow$ aaa
8a	aaaa	$D[9] \leftarrow$ aaaa

Dirección	w	k	Cadena
1	0	a	a
2	0	b	b
3	1	b	ab
4	0	c	c
5	2	a	ba
6	5	b	bab
7	1	a	aa
8	7	a	aaa
9	8	a	aaaa

Capítulo 10

LZW (LZ78, Welch)

10.1. LZW vs LZ78

- En 1984, Terry A. Welch propuso una modificación del algoritmo LZ78 para hacerlo un poco más eficiente, evitando escribir los símbolos k (sin codificar) [24]. El codec LZW (Lempel Ziv Welch) se utiliza en el formato GIF (Graphics Interchange Format) [4] de compresión de imágenes y en el compresor *compress* de los sistemas UNIX [21].
- El diccionario usado por LZW es casi igual al de LZ78, excepto que inicialmente no está vacío porque al comienzo se guardan todos los posibles símbolos (también llamados en el contexto del LZW, raíces). De esta forma, el símbolo k puede formar ahora parte de la cadena string(w).

10.2. Compresor

1. $w \leftarrow$ primer símbolo de entrada.
2. Mientras existan símbolos por codificar:
   - *a*) $k \leftarrow$ siguiente símbolo de entrada.
   - *b*) Si wk está en el diccionario, entonces:
     1) $w \leftarrow$ dirección de wk en el diccionario.
   - *c*) Si no:
     1) Escribir $\leftarrow w$ a la salida.
     2) Insertar wk en el diccionario.
     3) $w \leftarrow k$.

- La entrada 256 del diccionario (y el código de compresión 256) se utilizan para que el compresor le indique al descompresor distintas acciones. Esto se hace enviando el código 256 y a continuación un número que indique, por ejemplo:

1. Incrementar el número de bits utilizados para codificar w, en función del tamaño del diccionario.
2. Vaciar el diccionario porque está polucionado con cadenas que ahora no se están encontrado en la secuencia de símbolos de entrada. El diccionario también puede vaciarse porque se ha llenado.
3. El final de la descodificación.

- Puede encontrarse una implementación del algoritmo LZW en el Apéndice 39.20.

10.3. Ejemplo de compresión

Entrada	w	k	Salida	Comentario
ab	97	b	97	$D[257] \leftarrow$ ab
a	98	a	98	$D[258] \leftarrow$ ba
b	97	b		w $= 257$
c	257	c	257	$D[259] \leftarrow$ abc
b	99	b	99	$D[260] \leftarrow$ cb
a	98	a		w $= 258$
b	258	b	258	$D[261] \leftarrow$ bab
a	98	a		w $= 258$
b	258	b		w $= 261$
a	261	a	261	$D[262] \leftarrow$ baba
a	97	a	97	$D[263] \leftarrow$ aa
a	97	a		w $= 263$
a	263	a	263	$D[264] \leftarrow$ aaa
a	97	a		w $= 263$

Dirección	Cadena	w	k
0		0	NULL
⋮		⋮	⋮
97		0	a
98		0	b
99		0	c
⋮		⋮	⋮
256	reservado	ESC	
257	ab	97	b
258	ba	98	a
259	abc	257	c
260	cb	99	b
261	bab	258	b
262	baba	261	a
263	aa	97	a
264	aaa	263	a

10.4. Descompresor

1. $prev\_w \leftarrow$ primer código de entrada.
2. Escribir $prev\_w$ a la salida.
3. $k \leftarrow prev\_w$.
4. Mientras existan códigos de entrada:
   a) $w \leftarrow$ siguiente código de entrada.
   b) Si w está en el diccionario, entonces:
      1) Escribir string(w) a la salida.
   c) Si no:
      1) Escribir string(w) a la salida.
      2) Escribir k a la salida.
   d) $k \leftarrow$ primer símbolo emitido en la salida anterior.
   e) Insertar wk en el diccionario.
   f) $prev\_w \leftarrow w$.

10.5. Ejemplo de descompresión

Entrada	$code$	w	k	Salida	Comentario
97	97			a	Iniciación
98	98	97	b	b	$code < 257$, $D[257] \leftarrow$ ab
257	257	98	a	ab	$code < 258$, $D[258] \leftarrow$ ba
99	99	257	c	c	$code < 259$, $D[259] \leftarrow$ abc
258	258	99	b	ba	$code < 260$, $D[260] \leftarrow$ cb
261	261	258	b	bab	$code = 261$, $D[261] \leftarrow$ bab
97	97	261	a	a	$code < 262$, $D[262] \leftarrow$ baba
263	263	97	a	aa	$code = 263$, $D[263] \leftarrow$ aa

Capítulo 11

Compresión entrópica

11.1. Sub-optimalidad de los algoritmos greedy

- Los compresores de cadenas basados en diccionarios son muy rápidos y eficientes. Sin embargo, por tratarse de algoritmos "greedy", siempre padecen de una cierta pérdida de compresión que es sacrificada en pro de la velocidad de cómputo.

- Para entender esto valga el siguiente ejemplo. Supongamos que estamos usando LZSS para comprimir la cadena "Hola caracola" y el en diccionario existen las cadenas "Hol", "a c", "aracola" y " caracola". Utilizando una técnica greedy, sin tener en cuenta los símbolos futuros, se generarían los siguientes códigos (suponiendo 12 bits para el índice i y 4 bits para la longitud de la cadena j):

cadena	longitud
"Hol"	17 bits
"a c"	17 bits
"aracola"	17 bits

Sin embargo, también podría haberse producido la compresión:

cadena	longitud
“Hol”	17 bits
“a”	9 bits
“ caracola”	17 bits

que es 6 bits más corta.

11.2. Bit de dato y bit de información

- Aunque se trata de conceptos profundamente relacionados, existe una sutil diferencia entre dato e información:

  Los datos son la representación de la información.

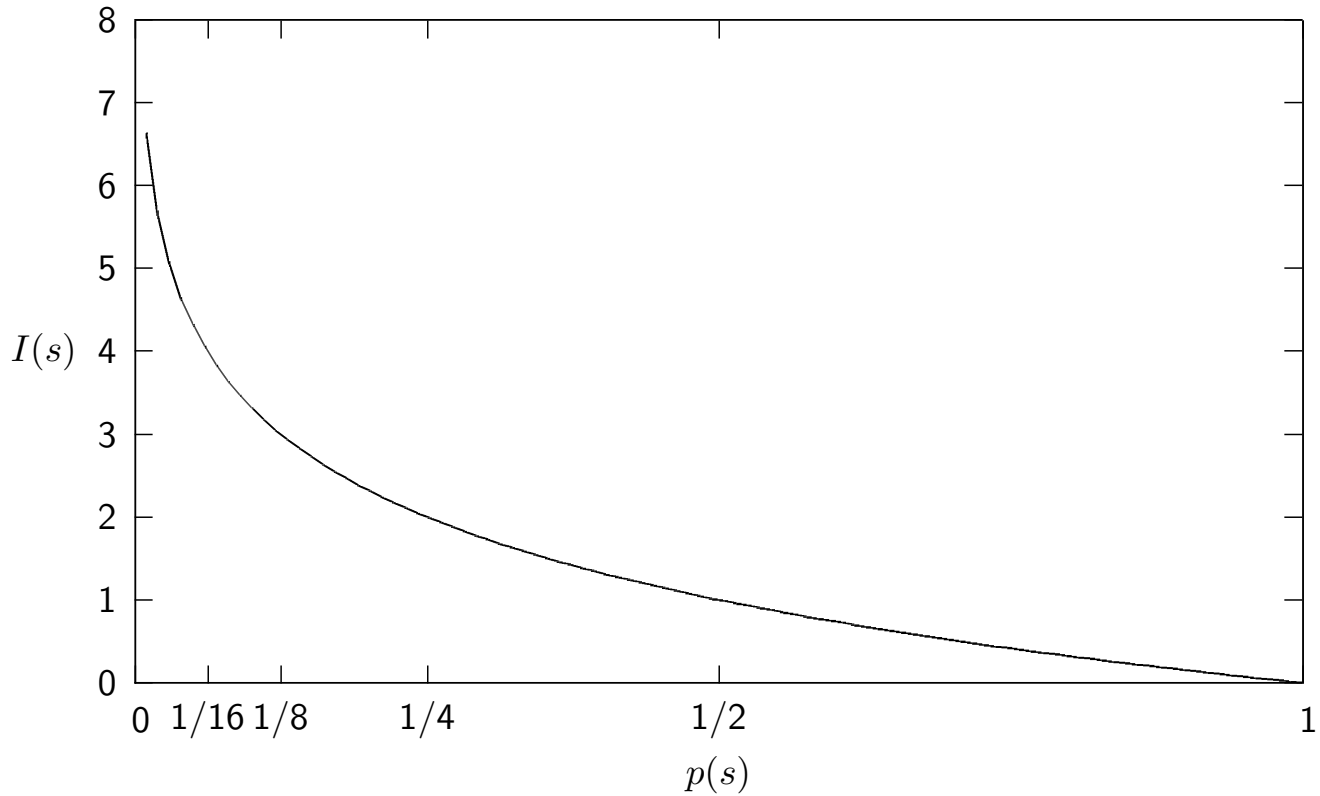
- Una consecuencia directa de este hecho es que la misma información puede ser representada de muchas formas, unas más compactas que otras. Esto explica la existencia de los compresores de datos.

- Por definición, un bit de de datos transporta un bit de información si y sólo si representa la ocurrencia de un evento equiprobable, es decir, si la probabilidad de que dicho evento sea verdadero es igual a la de que sea falso. En cualquier otro caso, el bit de datos representará una cantidad diferente de bits de información. Por definición, esta cantidad para el s-ésimo símbolo del alfabeto es

$$I(s) = -\log_2 p(s) \tag{11.1}$$

bits de información, donde $p(s)$ es la probabilidad de ocurrencia del símbolo s.

- Nótese que si la probabilidad de un símbolo es muy baja, entonces el número de bits de información representados es muy alto y viceversa. Gráficamente el número de bits de información asociados a un símbolo en función de su probabilidad es:



11.3. Entropía de una fuente de información

- La entropía es una medida de la cantidad de bits de información que una fuente de información proporciona en promedio, con cada símbolo generado. Por definición, la entropía $H(S)$ de una fuente S se calcula como

$$H(S) = \frac{1}{N} \sum_{s=1}^{N} p(s) \times I(s) \tag{11.2}$$

  donde N es el tamaño del alfabeto fuente (número de símbolos diferentes). Puede encontrarse una implementación del cálculo de la entropía en el Apéndice 39.13.

- La entropía se mide en bits de información por símbolo.

- La finalidad de la compresión estadística es la de encontrar una codificación tal que el número de bits de datos en promedio coincida con la entropía de la fuente.

11.4. Un codificador entrópico universal

- El objetivo de todos los codificadores entrópicos es generar un número de bits de código promedio que coincida con la entropía. Esto puede conseguirse si conseguimos representar cada símbolo con tantos bits de código como bits de información transporta.

Compresión de un símbolo

1. Mientras el símbolo no este determinado sin incertidumbre (por el descodificador):

   a) Realizar una afirmación acerca del símbolo que permita al descodificador reducir la incertidumbre sobre él. Intentar que dicha afirmación tenga las mismas posibilidades de ser cierta que falsa.

   b) Emitir un bit de código que indique el resultado de dicha afirmación.

Descompresión de un símbolo

1. Mientras el símbolo no este determinado sin incertidumbre:
   a) Realizar la misma afirmación que el codificador.
   b) Recibir un bit de código que indique el resultado de dicha afirmación.

- Si conseguimos que todas las afirmaciones sean equiprobables estaremos generando una codificación 100 % eficiente.

Ejemplo

Supongamos el alfabeto del lenguaje castellano. Nosotros sabemos que los símbolos de una palabra no se unen en cualquier orden. Por ejemplo, si tenemos que comprimir la palabra "preciosa" podríamos usar un diccionario para calcular el número de letras que pueden seguir a un determinado prefijo, ya conocido.

Siguiendo con el ejemplo, la primera letra nos costaría 5 bits el codificarla (recuérdese que el alfabeto español tiene menos de 32 letras y más de 16). Conociendo que se trata de una "p", sólo existen 9 posibilidades: "a", "e", "i", "l", "n", "o", "r", "s" y "u". Por tanto, para codificar la siguiente letra, la "r", necesitaríamos 4 bits.

El resto del proceso se sintetiza en la siguiente tabla:

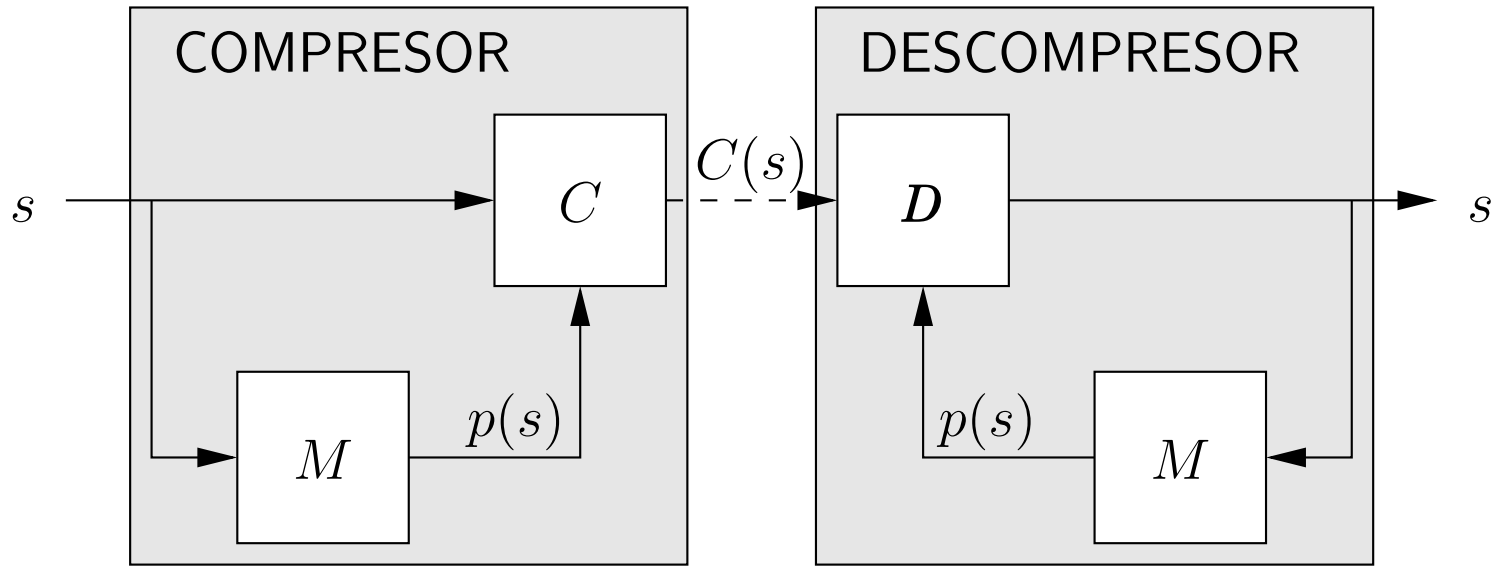
Intentos	p	r	e	c	i	o	s	a
1	*	a	a	a	a	a	s	a
2		e	e	b	e	n		i
3		i	i	c	i	o		o
4		l	o	d	l	p		
5		n	u	f	o	s		
6		o		g	u	t		
7		r		h				
8		s		i				
9		u		j				
10				l				
11				m				
12				n				
13				ñ				
14				o				
15				p				
16				r				
17				s				
18				t				
19				v				
20				z				
bits	5	4	3	5	3	3	0	2

Total de bits emitidos: 25. Total de bits codificados: $8 \times 5 = 40$.

11.4 Un codificador entrópico universal

11.5. Compresión basada en modelos probabilísticos

- El compresor consta de un codificador y de un modelo que le indica qué información es la que es necesario codificar. Dicha información puede considerarse como aquella que el modelo no es capáz de predecir.
- El descompresor contiene un modelo idéntico y un descodificador:



11.5 Compresión basada en modelos probabilísticos

Capítulo 12

Codificación de Shannon-Fano

- A finales de los 40, Claude Shannon (Bell Labs) y R.M. Fano (MIT) desarrollaron un sistema de codificación reversible que utiliza la en-

tropía de la secuencia de símbolos para comprimirla. La idea es:

- Los códigos de compresión tendrán una longitud variable.
- Aquellos símbolos más probables recibiran códigos más cortos y viceversa.

12.1. Algoritmo de Shannon-Fano

1. Ordenar los símbolos atendiendo a sus probabilidades.
2. Dividir el conjunto de símbolos en dos subconjuntos de forma que la probabilidad de cada subconjunto sea lo más parecida posible. Asignar a un subconjunto un bit de datos y al otro, el contrario.
3. Aplicar recursivamente el paso anterior a ambos subcojuntos hasta que no sea posible dividir más.

12.2. Ejemplo de codificación

Sea el siguiente modelo probabilístico:

Símbolo	Recuento
A	15
B	7
C	6
D	6
E	5

Primera iteración:

Símbolo	Recuento	Bits
A	15	0
B	7	0
C	6	1
D	6	1
E	5	1

Segunda iteración:

Símbolo	Recuento	Bits
A	15	00
B	7	01
C	6	10
D	6	11
E	5	11

Tercera iteración:

Símbolo	Recuento	Bits
A	15	00
B	7	01
C	6	10
D	6	110
E	5	111

Capítulo 13

Codificación de Huffman

13.1. El Algoritmo de Huffman

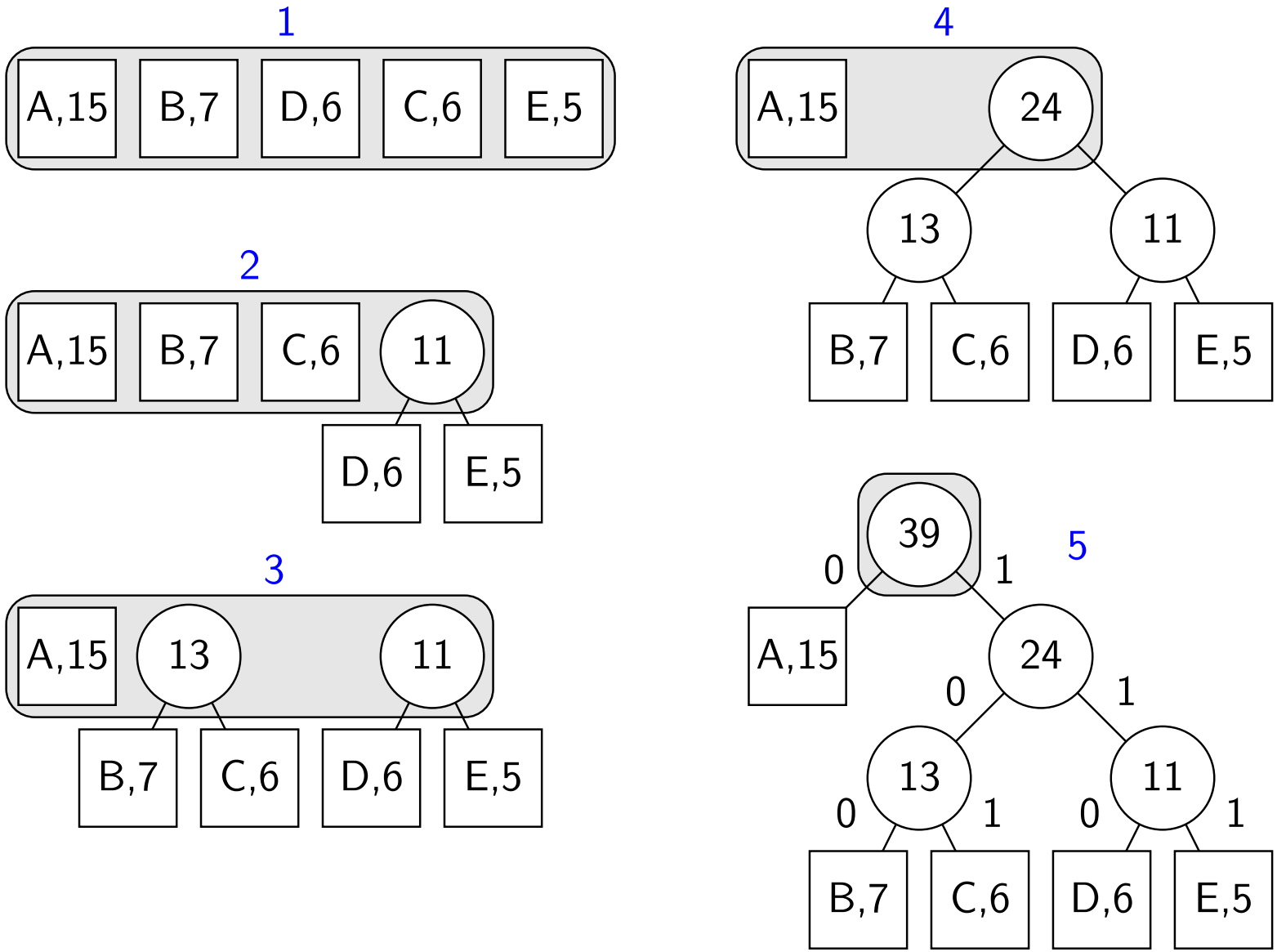
- Ideado por David A. Huffman en 1952 [10], es una de las codificaciones más utilizadas actualmente debido a su sencillez, buen rendimiento y facilidad de uso.
- Al igual que la codificación de Shannon-Fano, se basa en la idea de utilizar códigos de compresión de longitud variable de forma que aquellos símbolos más frecuentes se representan mediante los códigos más cortos y viceversa.
- El codificador crea los códigos mediante un árbol binario en el cual los símbolos están en las hojas y las ramas son etiquetadas usando los dígitos binarios 0 y 1. La distancia de un símbolo a la raíz del árbol define la longitud del código de Huffman asignado a dicho símbolo, y esto depende en última instancia de la probabilidad del símbolo. En concreto, el código asignado a un símbolo es aquel número binario que resulta de viajar desde la raíz hasta dicho símbolo.
- La codificación de Huffman es más eficiente que la de Shannon-Fano

porque el número de bits de datos asociado a cada código se aproxima más al número de bits de información.

13.2. Generación del árbol de Huffman

1. Crear una lista de árboles binarios, en la que cada árbol está formado por un único nodo y cada nodo contiene un símbolo y su probabilidad.
2. Mientras existan al menos 2 árboles en la lista:
   a) Extraer de la lista los 2 árboles con menor probabilidad.
   b) Insertar en la lista un nuevo árbol binario cuyas hojas son los árboles extraídos y cuya raíz es la suma de las probabilidades de estos.
3. Puede encontrarse una implementación del algoritmo de Huffman en el Apéndice 39.16.

13.3. Ejemplo


1
A,15
B,7
D,6
C,6
E,5
2
A,15
B,7
C,6
11
D,6
E,5
3
A,15
13
11
B,7
C,6
D,6
E,5
4
A,15
24
13
11
B,7
C,6
D,6
E,5
5
39
0
1
A,15
24
0
1
13
11
0
1
0
1
B,7
C,6
D,6
E,5

- Nótese que para diseñar el árbol de Huffman es necesario conocer la probabilidad de ocurrencia de todos los posibles símbolos de la secuencia. El modelo estadístico de la fuente es el elemento que proporciona dicha información.
- La descodificación se realiza conociendo el árbol de Huffman o lo que es lo mismo, el modelo estadístico utilizado por el codificador.

13.4. Comparación entre Shannon-Fano y Huffman

Símbolo	Recuento	Shannon-Fano bits/símbolo	S-F bits	Huffman bits/símbolo	Huffman bits
A	15	2	30	1	15
B	7	2	14	3	21
C	6	2	12	3	18
D	6	3	18	3	18
E	5	3	15	3	15
Total			89		87

13.5. Limitaciones del código de Huffman

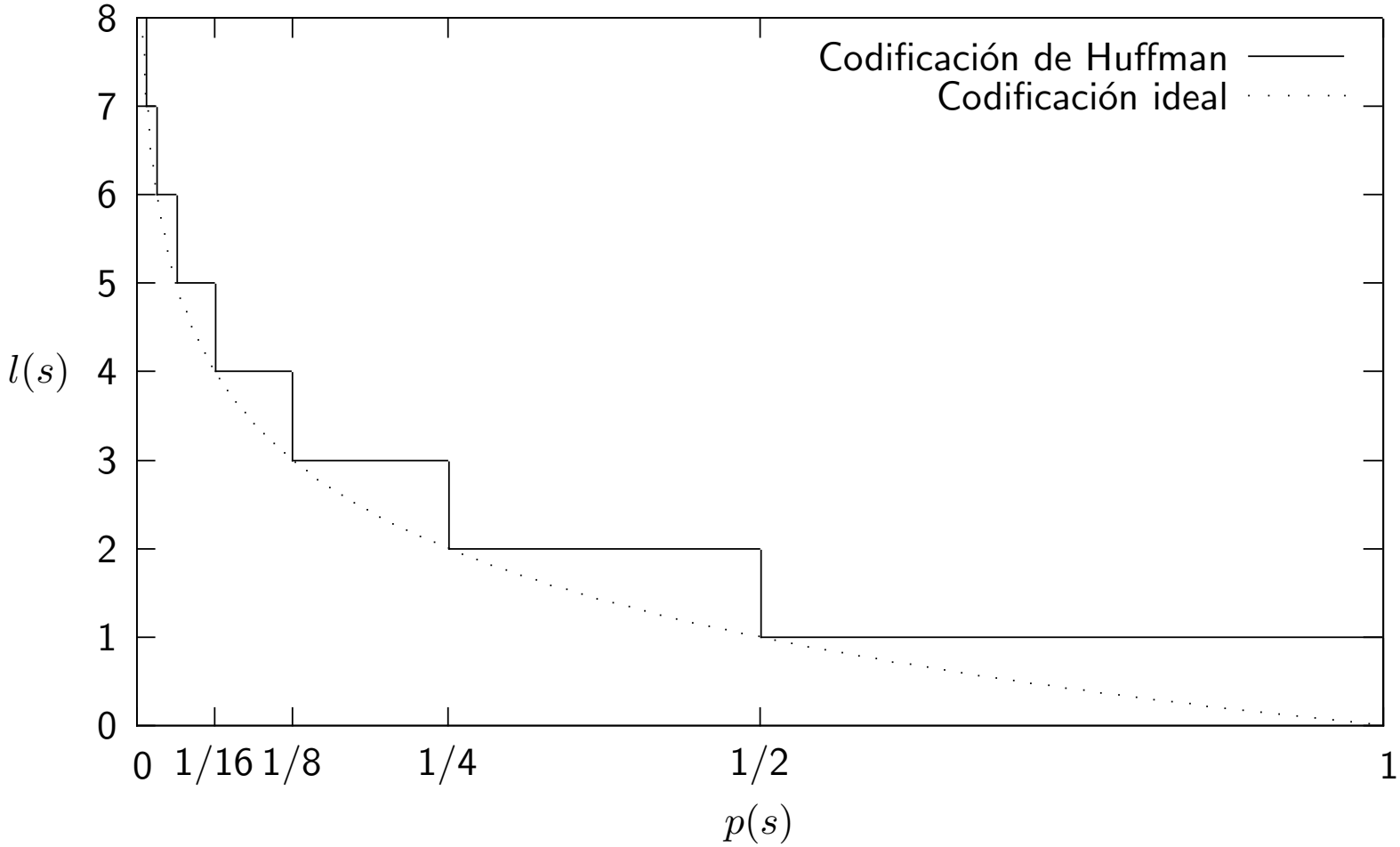
- Por desgracia, la codificación de Huffman asigna a cada símbolo un número entero de bits y por definición, el número de bits de información no tiene por qué serlo. En otras palabras, en la codificación de Huffman se cumple que

$$l\big(c(s)\big) = \lceil I(s) \rceil, \tag{13.1}$$

  donde $l\big(c(s)\big)$ es la longitud del código de compresión asignado al símbolo s.

- Esto provoca que con cada codificación de un símbolo, hasta casi un bit de datos de redundancia podría ser introducido. Así, para una secuencia completa el número de bits de redundancia puede ser importante.

- Este problema se agudiza cuando el número de símbolos es sólo 2. En este caso, la codificación de Huffman no cambia la representación original binaria (0 y 1), y la cantidad de redundancia introducida para codificar el símbolo más frecuente es importante. Gráficamente:



13.5 Limitaciones del código de Huffman

Capítulo 14

Codificación Aritmética

14.1. Los códigos aritméticos

- Un código aritmético solventa el problema del código de Huffman asignando a cada símbolo un número de bits de datos igual al número de bits de información que transporta, cumpliéndose que

$$l\big(c(s)\big) = I(s). \tag{14.1}$$

- En este sentido, la longitud media de un código aritmético coincide con la entropía de la fuente, medida en bits de datos por símbolo. Por tanto, se trata de un sistema de codificación óptimo.

14.2. Compresor (versión sencilla)

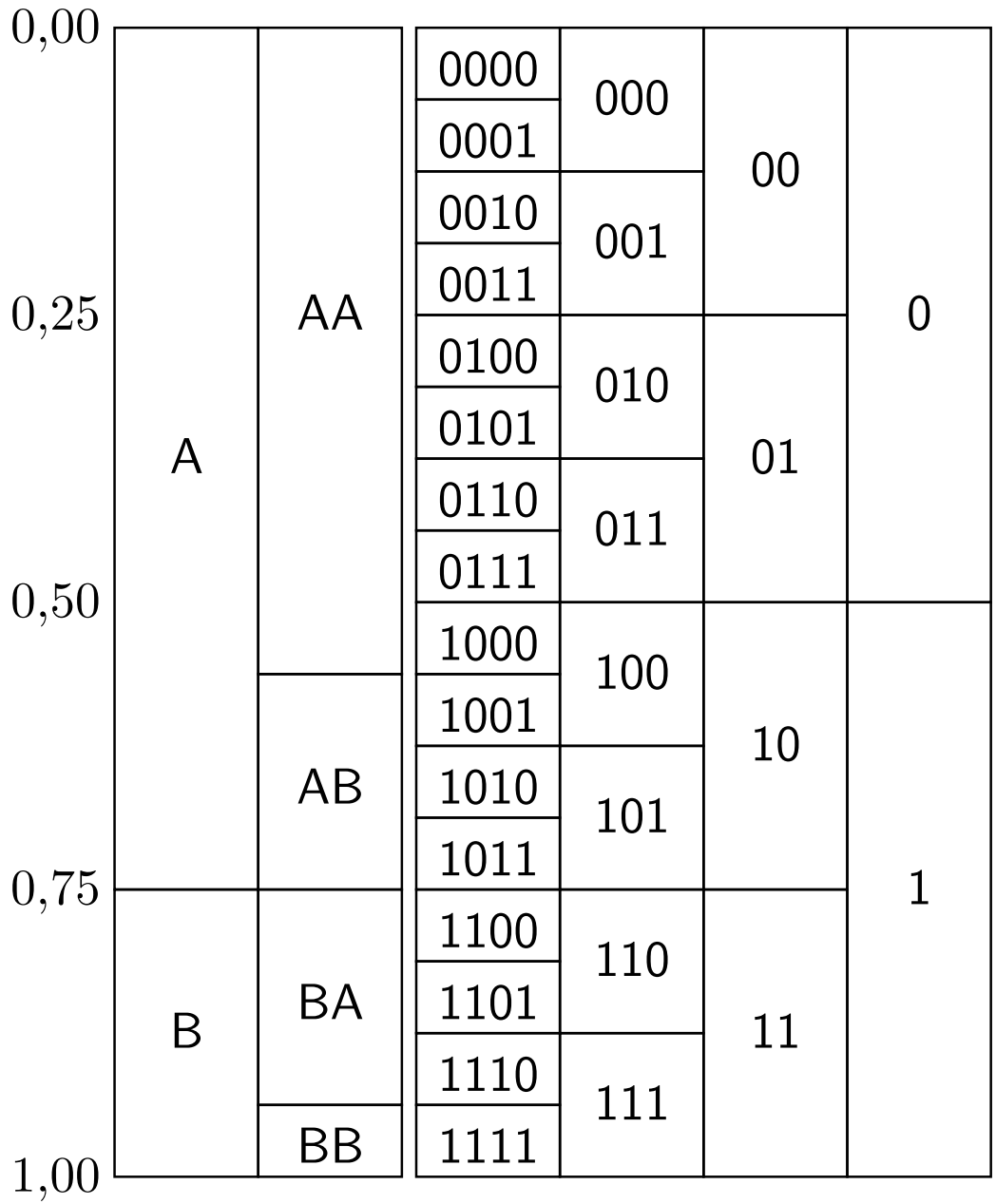
1. Sea $[L, H) \leftarrow [0{,}0, 1{,}0)$ el intervalo inicial.
2. Mientras existan símbolos de entrada:
   a) Dividir el intervalo $[L, H)$ en tantos sub-intervalos como símbolos diferentes existen en el alfabeto. El tamaño de cada sub-intervalo es proporcional a la probabilidad del símbolo asociado.
   b) Seleccionar de entre todos los sub-intervalos, el que corresponde al símbolo codificado en la iteración actual. Sea el intervalo elegido $[L', H')$.
   c) Hacer $[L, H) \leftarrow [L', H')$.
3. Emitir un número $x \in [L, H)$ como código aritmético. El número de cifras deberá permitir distinguir el intervalo final $[L, H)$ de cualquier otro.

14.3. Descompresor (versión sencilla)

1. Sea $[L, H) \leftarrow [0{,}0, 1{,}0)$ el intervalo inicial.
2. Mientras existan símbolos que descodificar:
   a) Dividir el intervalo $[L, H)$ en tantos sub-intervalos como símbolos diferentes existen en el alfabeto. El tamaño de cada sub-intervalo es proporcional a la probabilidad del símbolo asociado.
   b) Leer tantos bits del código aritmético x de entrada como sean necesarios para:
      1) Seleccionar $[L', H')$ al que el código aritmético x pertenece.
      2) Emitir el símbolo asociado a $[L', H')$.
      3) Hacer $[L, H) \leftarrow [L', H')$.

14.4. Ejemplo de compresión

Imaginemos que en una secuencia binaria (con alfabeto $\{\mathrm{A}, \mathrm{B}\}$) tenemos que $p(\mathrm{A}) = 3/4$ y $p(\mathrm{B}) = 1/4$. Calcúlese el código aritmético asociado a las secuencias A, B, AA, AB, BA y BB.



14.4 Ejemplo de compresión

14.5. Ejemplo de descompresión

(Ver el ejemplo del compresor, pero desde el punto de vista del descompresor). El proceso consiste básicamente en leer tantos bits de entrada como sean necesarios para determinar un símbolo, y repetir este proceso hasta que no haya más bits de código aritmético.

14.6. Transmisión incremental

- En la práctica el codificador envía los bits del código aritmético en el instante en que estos son conocidos.

  Por ejemplo, cuando se codifica la B, inmediatamente se transmite un 1 porque cualquier secuencia de símbolos que comienza por B se codifica inicialmente con un 1. A continuación se expande el intervalo $[0{,}50, 1{,}00)$ (porque se ha transmitido un 1) al intervalo $[0, 1)$ y el proceso se repite con el siguiente símbolo de entrada.

  Esto se hace para no perder precisión numérica a la hora de representar (usando un número finito de bits) los límites del intervalo.

- Nótese que la transmisión de aquellos bits más significativos de L y H en cuanto son conocidos implican el desplazamiento a la izquierda del resto de bits y la inclusión de bits iguales a 0 por la derecha. El resultado de dicho desplazamiento es una ampliación automática del intervalo seleccionado.

- Puede encontrarse una implementación de un codificador aritmético en el Apéndice 39.3.

14.7. Codificador (versión real)

1. Sean L y H los límites del intervalo de codificación. Suponiendo 16 bits de precisión para estos registros, inicialmente $L \leftarrow 0000$ y $H \leftarrow FFFF$.

2. Mientras existan símbolos a la entrada:

   a) Sea s el siguiente símbolo a codificar y $p(s)$ su probabilidad. Sea $P_L(s)$ la probabilidad acumulada hasta el símbolo $s-1$ y $P_H(s)$ la probabilidad acumulada hasta el símbolo s. Más formalmente,

$$P_L(s) = \sum_{i=0}^{i=s-1} p(i)$$

   y

$$P_H(s) = P_L(s) + p(s).$$

   Seleccionar el siguiente intervalo de codificación haciendo

$$H \leftarrow L + P_H \times R - 1$$

y

$$L \leftarrow L + P_L \times R,$$

donde

$$R = H - L + 1$$

es el rango (tamaño) del intervalo de codificación.

3. Ejecutar en un lazo los siguientes pasos:

   a) Si $H < 8000$ (nótese que siempre se cumple que $L < H$) emitir un 0.

   b) Si $L \geq 8000$ emitir un 1 y restar a L y H el offset 8000 para que no se produzca un overflow: $L \leftarrow L - 8000$ y $H \leftarrow H - 8000$.

   c) Si $4000 \leq L < H < C000$ (los dos MSb's de L son 01 y los de H son 10). En esta situación no podemos ejecutar la transmisión incremental y podríamos caer en un problema de underflow (los registros L y H podrían almacenar el mismo número). Para evitar esto, esplazaremos ambos registros haciendo $L \leftarrow L - 4000$ y $H \leftarrow H - 4000$ (nótese que esto es equivalente a haber restado

el valor 8000 después de la expansión) y anotamos el número de ocasiones que permanecemos en esta situación. La próxima vez que transmitamos un bit (debido al proceso de transmisión incremental), transmitiremos además tantos bits contrarios a este como dicho número indique.*

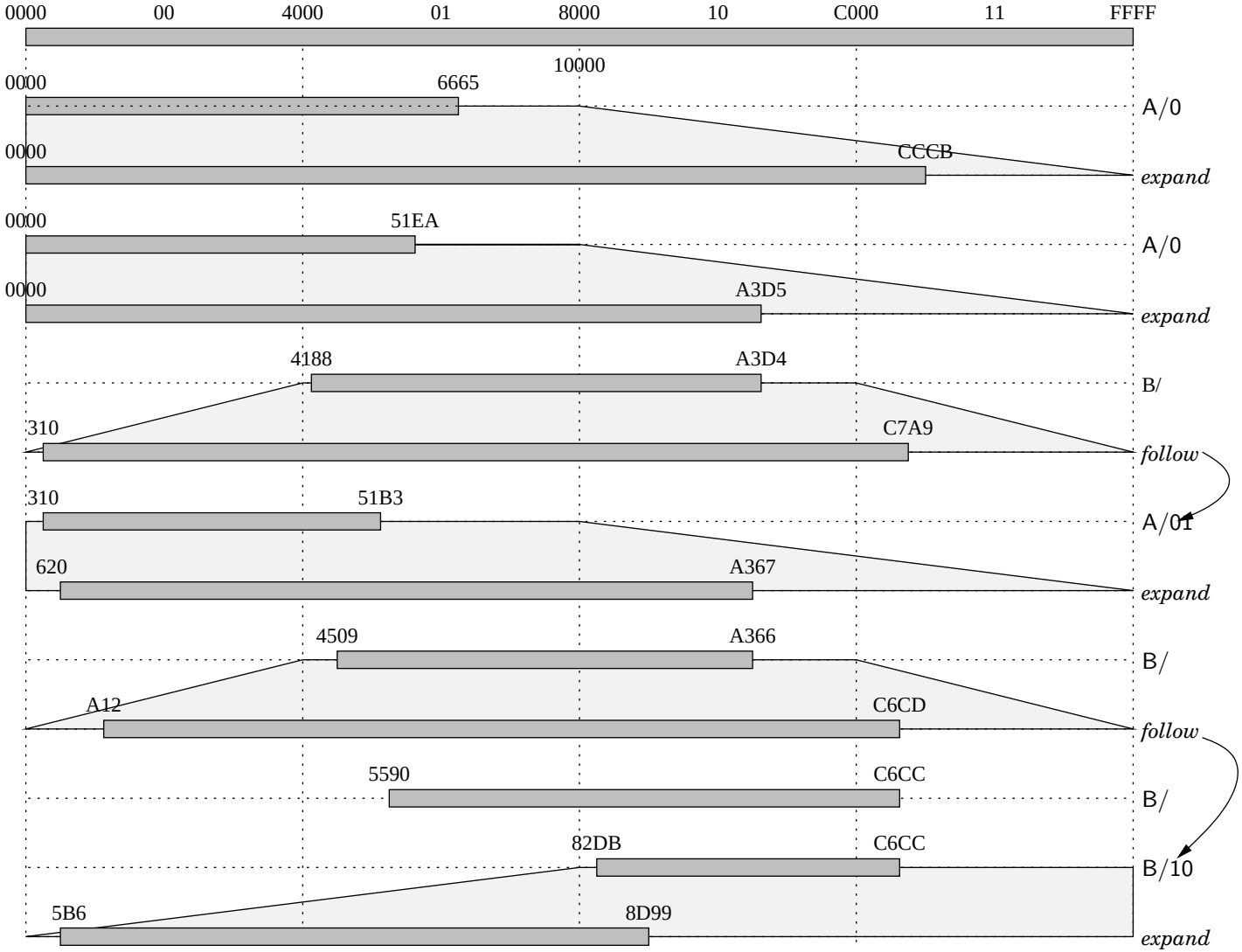
d) En cualquier otro caso, salir del lazo.

4. Expandir el intervalo de codificación: $L \leftarrow 2 \times L$ y $H \leftarrow 2 \times H + 1$.

*Esto produce el efecto equivalente a transmitir los bit "decimales" que dejamos de transmitir temporalmente debido a que los MSb's de L y H no coincidían.

14.8. Ejemplo de codificación (versión real)

Codificación de la secuencia AABABBB donde $p(\mathrm{A}) = 0{,}4$.



14.9. Descodificador (versión real)

1. Sean $L \leftarrow 0000$, $H \leftarrow FFFF$ y $V \leftarrow$ los primeros 16 bits de entrada.
2. Mientras existan símbolos que descodificar:
   a) Encontrar el símbolo cuyo intervalo contiene a V. Este es el símbolo descodificado.
   b) Calcular el siguiente intervalo de forma análoga al paso 2.a del codificador.
   c) Ejecutar en un lazo los siguientes pasos:
      1) Si el MSb de L y H son iguales a 1, hacer $L \leftarrow L - 8000$, $H \leftarrow H - 8000$ y $V \leftarrow V - 8000$.
      2) Si $4000 \leq L < H < C000$ (situación de posible underflow), hacer $L \leftarrow L - 4000$, $H \leftarrow H - 4000$ y $V \leftarrow V - 4000$.
      3) En cualquier otro caso, salir del lazo.
      4) Expandir el intervalo de codificación, igual que en el paso 4 del codificador.

5) Insertar como LSb el siguiente bit de código aritmético en V y desplazar el resto de bits hacia posiciones más significativas.

Capítulo 15

Modelos probabilísticos

15.1. Modelos estáticos

- Son los modelos probabilísticos más sencillos que existen porque asignan una probabilidad constante (a lo largo del tiempo) a los símbolos de la fuente.
- El uso de un modelo estático permite además precalcular los códigos de longitud variable, lo que simplifica el proceso de codificación.
- A pesar de su simpleza son muy utilizados porque normalmente se conoce de antemano la probabilidad de los símbolos (ejemplos: JPEG, MPEG, MP3, etc.).
- Un ejemplo de codificación estática lo podemos encontrar en el Apéndice 39.16.

15.2. Modelos adaptativos

- Cuando las probabilidades de los símbolos no son conocidas a priori, los modelos probabilísticos adaptativos pueden calcular sus probabilidades en "tiempo de ejecución".
- En general, las tasas de compresión conseguidas utilizando modelos adaptativos son superiores a las de los modelos estáticos.
- Ejemplos de modelos adaptativos están en los Apéndices 39.23, 39.35, 39.46, 39.41 y 39.45.

Compresor

1. Asignar la misma probabilidad a todos los símbolos.
2. Mientras existan símbolos que codificar:
   a) Codificar el siguiente símbolo.
   b) Actualizar (incrementando) su probabilidad.

Descompresor

1. Asignar la misma probabilidad a todos los símbolos.
2. Mientras existan símbolos que descodificar:
   a) Descodificar el siguiente símbolo.
   b) Actualizar (incrementando) su probabilidad.

15.3. Modelos inicialmente vacíos

- Una mejora que podemos hacer es contemplar la posibilidad de que transcurra bastante tiempo hasta que un símbolo es utilizado por primera vez. Cuando esto es así, el repartir inicialmente el espacio de probabilidades entre unos pocos símbolos en lugar de todos los posibles puede ayudar a incrementar las probabilidades de los símbolos que realmente están ocurriendo y reducir así la tasa de bits.

- Para hacer esto se incorpora al modelo un símbolo especial llamado ESC(ape) que nunca va a ser generado por la fuente, sino por el codificador. De esta forma, cuando el codificador encuentra un símbolo que nunca antes había aparecido, lo incorpora a su modelo y envía un símbolo ESC hasta el descodificador para indicarle que va a recibir un símbolo nuevo.

- Un ejemplo de un modelo probabilístico inicialmente vacío puede encontrarse en el Apéndice 39.41.

Compresor

1. Asignar a ESC la máxima probabilidad $1{,}0$ y $0{,}0$ al resto de símbolos.
2. Mientras existan símbolos que codificar:
   a) $s \leftarrow$ siguiente símbolo.
   b) Si s ya ha aparecido antes, entonces:
      1) Codificar s.
   c) Si no:
      1) Codificar un ESC.
      2) Enviar s (sin codificar).
      3) Añadir s al modelo.
   d) Actualizar $p(s)$.

Descompresor

1. Asignar a ESC la máxima probabilidad $1{,}0$ y $0{,}0$ al resto de símbolos.
2. Mientras existan símbolos que descodificar:
   a) Descodificar s.
   b) Si $s =$ ESC, entonces:
      1) Recibir un nuevo s (que no está descodificado).
      2) Añadir s al modelo.
3. Actualizar $p(s)$.

15.4. Modelos con memoria

- En muchos casos la probabilidad de un símbolo depende de los símbolos vecinos (también llamados contexto). Cuando esto ocurre se dice que el modelo posee memoria (recuerda el contexto) [3].
- La ventaja de tener en cuenta el contexto es que las probabilidades pueden calcularse con mayor precisión (asignando probabilidades más altas) y por tanto se decrementa el número de bits necesarios para representar las secuencias comprimidas.
- En los Apéndices 39.46, 39.41 y 39.45 encontraremos ejemplos de modelos con memoria.

Compresor

Sea $C[i]$ la secuencia de los i últimos símbolos codificados y sea $p(s|C[i])$ la probabilidad de que ocurra el símbolo s tras $C[i]$. Sea k el orden de predicción (el número máximo de símbolos recordados).

1. Crear un modelo vacío (excepto por el símbolo ESC) para cada posible contexto donde $i \geq 0$ y un modelo no vacío para $i = -1$.
2. Mientras existan símbolos que codificar:
   - *a*) $s \leftarrow$ siguiente símbolo.
   - *b*) $i \leftarrow k$ (excepto en la primera iteración donde $i \leftarrow 0$).
   - *c*) Mientras $p(s|C[i]) = 0$ (s es la primera vez que aparece tras $c[i]$):
     1) Codificar ESC según $p(\text{ESC}|C[i])$.
     2) Añadir s al contexto para $C[i]$.
     3) $i \leftarrow i - 1$.

d) Codificar s según $p(s|C[i])$. Todos los símbolos que estaban en contextos de orden mayor que i deben ser excluidos del contexto actual ya que es seguro que s no es ninguno de ellos.

e) Actualizar $p(s|C[i])$.

Ejemplo

- $r = 256$ es el tamaño del alfabeto fuente.
- El contexto $C[-1]$ tiene asociado un modelo probabilístico $M[C[-1]]$ de orden 0 no adaptativo y no vacío, en el que además aparece el símbolo especial EOF (End Of File). Por tanto, este modelo contiene $r + 1$ símbolos y su contenido es

$$M[C[-1]] = \{0, 1\ 1, 1\ \cdots\ \texttt{a}, 1\ \texttt{b}, 1\ \cdots\ 255, 1\ \text{ESC}, 1\ \text{EOF}, 1\},$$

  donde cada par a, b denota el símbolo a y su probabilidad b, medida como un número de ocurrencias.
- El contexto $C[0]$ tiene asociado un modelo probabilístico de orden 0 adaptativo e inicialmente vacío, que contiene sólo el símbolo ESC. Su contenido es, por tanto

$$M[C[0]] = \{\text{ESC}, 1\}.$$

- En este ejemplo (por simplicidad), el orden máximo de predicción es $k = 1$ (se recuerda a lo sumo el símbolo anterior). Por tanto, existen otros r contextos de orden 0 adaptativos e inicialmente vacíos, iguales al anteriormente descrito y que contienen sólo el símbolo ESC.
- Durante la codificación del símbolo a ocurre lo siguiente:
  1. $s \leftarrow$ a.
  2. $i \leftarrow 0$.
  3. $p(\text{a}|C[0]) = 0$ (el contexto contiene sólo el ESC).
  4. Codificamos un ESC aunque no se genera ningún bit de código porque su probabilidad es 1.
  5. Añadimos a al modelo $M[C[0]]$.
  6. Hacemos $i \leftarrow -1$.
  7. $p(\text{a}|C[-1]) \neq 0$.
  8. Codificamos a según $p(\text{a}|C[-1]) = 1/(r+1)$.
- Durante la codificación del segundo símbolo (b):

1. $s \leftarrow \mathtt{b}$.
2. $i \leftarrow 1$.
3. $p(\mathtt{b}|C[1]) = 0$ porque $C[1] = \mathtt{a}$ y $M[\mathtt{a}]$ contiene inicialmente sólo el símbolo ESC.
4. Codificamos un ESC y no se genera ningún bit de código porque su probabilidad es 1.
5. Añadimos $\mathtt{b}$ al modelo $M(\mathtt{a})$.
6. Hacemos $i \leftarrow 0$.
7. $p(\mathtt{b}|C[0]) = 0$ porque $M[C[0]]$ contiene sólo el símbolo ESC y el símbolo a.
8. Codificamos un ESC, pero ahora se generan $1/2$ bits de datos. Se incrementa su recuento.
9. Añadimos $\mathtt{b}$ al modelo $M[C[0]]$. Ahora $M[C[0]] = \{\mathrm{ESC}, 4\ \mathtt{a}, 1\ \mathtt{b}, 1\}$.
10. Hacemos $i \leftarrow -1$.
11. $p(\mathtt{b}|C[-1]) \neq 0$.

12. Codificamos b según $p(\mathtt{b}|C[-1]) = 1/r$. Nótese que debe excluirse el símbolo a para calcular la probabilidad del símbolo b porque el símbolo a ya aparece en el contexto $M[C[0]]$.

Entrada	Salida	Prob. de la Salida	Contextos Afectados
a	$c_{M[C[0]]}(\text{ESC})c_{M[C[-1]]}(\text{a})$	$1 \cdot 1 \cdot 1/(r+1)$	$M[C[0]] = \{\text{ESC}, 1\ \text{a}, 1\}$
b	$c_{M[\text{a}]}(\text{ESC})c_{M[C[0]]}(\text{ESC})c_{M[C[-1]]}(\text{b})$	$1 \cdot 1/2 \cdot 1/r$	$M[\text{a}] = \{\text{ESC}, 1\ \text{b}, 1\}$, $M[c[0]] = \{\text{ESC}, 1\ \text{a}, 1\ \text{b}, 1\}$
a	$c_{M[\text{b}]}(\text{ESC})c_{M[C[0]]}(\text{a})$	$1 \cdot 1/3$	$M[\text{b}] = \{\text{ESC}, 1\ \text{a}, 1\}$, $M[C[0]] = \{\text{ESC}, 1\ \text{a}, 2\ \text{b}, 1\}$
b	$c_{M[\text{a}]}(\text{b})$	$1/2$	$M[C[\text{a}]] = \{\text{ESC}, 1\ \text{b}, 2\}$
c	$c_{M[\text{b}]}(\text{ESC})c_{M[C[0]]}(\text{ESC})c_{M[C[-1]]}(\text{c})$	$1/2 \cdot 1/4 \cdot 1/(r-1)$	$M[\text{b}] = \{\text{ESC}, 1\ \text{a}, 1\ \text{c}, 1\}$, $M[C[0]] = \{\text{ESC}, 1\ \text{a}, 2\ \text{b}, 1\ \text{c}, 1\}$
b	$c_{M[\text{c}]}(\text{ESC})c_{M[C[0]]}(\text{b})$	$1 \cdot 1/5$	$M[\text{c}] = \{\text{ESC}, 1\ \text{b}, 1\}$, $M[C[0]] = \{\text{ESC}, 1\ \text{a}, 2\ \text{b}, 2\ \text{c}, 1\}$
a	$c_{M[\text{b}]}(\text{a})$	$1/3$	$M[\text{b}] = \{\text{ESC}, 1\ \text{a}, 2\ \text{c}, 1\}$
b	$c_{M[\text{a}]}(\text{b})$	$2/3$	$M[\text{a}] = \{\text{ESC}, 1\ \text{b}, 3\}$
a	$c_{M[\text{b}]}(\text{a})$	$2/4$	$M[\text{b}] = \{\text{ESC}, 1\ \text{a}, 3\ \text{c}, 1\}$
b	$c_{M[\text{a}]}(\text{b})$	$3/4$	$M[\text{a}] = \{\text{ESC}, 1\ \text{b}, 4\}$
a	$c_{M[\text{b}]}(\text{a})$	$3/5$	$M[\text{b}] = \{\text{ESC}, 1\ \text{a}, 4\ \text{c}, 1\}$
a	$c_{M[\text{a}]}(\text{ESC})c_{M[C[0]]}(\text{a})$	$1/5 \cdot 2/4$	$M[\text{a}] = \{\text{ESC}, 1\ \text{a}, 1\}\ \text{b}, 4$, $M[C[0]] = \{\text{ESC}, 1\ \text{a}, 3\ \text{b}, 2\ \text{c}, 1\}$
a	$c_{M[\text{a}]}(\text{a})$	$1/6$	$M[\text{a}] = \{\text{ESC}, 1\ \text{a}, 2\}\ \text{b}, 4$
a	$c_{M[\text{a}]}(\text{a})$	$2/7$	$M[\text{a}] = \{\text{ESC}, 1\ \text{a}, 3\ \text{b}, 4\}$
a	$c_{M[\text{a}]}(\text{a})$	$3/8$	$M[\text{a}] = \{\text{ESC}, 1\ \text{a}, 4\ \text{b}, 4\}$
a	$c_{M[\text{a}]}(\text{a})$	$4/9$	$M[\text{a}] = \{\text{ESC}, 1\ \text{a}, 5\ \text{b}, 4\}$
a	$c_{M[\text{a}]}(\text{a})$	$5/10$	$M[\text{a}] = \{\text{ESC}, 1\ \text{a}, 6\ \text{b}, 4\}$

Descompresor

1. Idem al paso 1 del compresor.
2. Mientras existan símbolos que descodificar:
   - *a*) $i \leftarrow k$.
   - *b*) Mientras el símbolo descodificado sea ESC:
     - 1) $i \leftarrow i - 1$.
   - *c*) Descodificar s según $p(s|c[i])$.
   - *d*) Actualizar $p(s|c[i])$.
   - *e*) Mientras $i < k$:
     - 1) $i \leftarrow i + 1$.
     - 2) Añadir s al contexto para $c[i]$.

Capítulo 16

Transformada mover-al-frente

La transformación mover-al-frente

- Existen muchas situaciones donde el proceso de codificación entrópico es más sencillo si los símbolos poseen una probabilidad de ocurrencia que decrece con el índice del símbolo en el alfabeto fuente.
- La transformación mover-al-frente (move-to-front: MTF, en inglés) puede ser aplicada a una secuencia de texto para conseguir el anterior efecto.
- El Apéndice 39.50 contiene una implementación de este sistema de codificación.

16.1. Transformada directa

1. Crear una lista $L[]$ con todos los símbolos del alfabeto.
2. Mientras existan símbolos que codificar:
   a) Sea s el siguiente símbolo de entrada.
   b) Buscar s en $L[]$ y emitir como código su posición en dicha lista.
   c) Mover s al frente de $L[]$ haciendo que $L[0] \leftarrow s$. El resto de símbolos son desplazados una posición hacia posiciones de índice superior.

16.2. Transformada inversa

1. Crear una lista $L[]$ con todos los símbolos del alfabeto (igual al paso 1 del codificador).

2. Mientras existan símbolos que descodificar:

   a) Sea c el siguiente código de entrada.

   b) Emitir como símbolo $s \leftarrow L[c]$.

   c) Idem al paso 2.c del codificador.

16.3. Ejemplo de transformación

lista	entrada	salida
$\cdots$abc$\cdots$	b	b
b$\cdots$ac$\cdots$	a	b
ab$\cdots$c$\cdots$	a	0
ab$\cdots$c$\cdots$	a	0
ab$\cdots$c$\cdots$	a	0
ab$\cdots$c$\cdots$	a	0
ab$\cdots$c$\cdots$	a	0
ab$\cdots$c$\cdots$	b	1
ba$\cdots$c$\cdots$	b	0
ba$\cdots$c$\cdots$	a	1
ab$\cdots$c$\cdots$	b	1
ba$\cdots$c$\cdots$	a	1
ab$\cdots$c$\cdots$	a	0
ab$\cdots$c$\cdots$	c	c
cab$\cdots\cdots$	a	1
acb$\cdots\cdots$	a	0

Capítulo 17

Codificación unaria

La codificación unaria

- La codificación unaria es un sistema de codificación de longitud variable donde el número de bits que se le asignan a los símbolos es directamente proporcional a su índice en el alfabeto.
- En concreto, la codificación unaria dedica al símbolo de índice s, s bits. s bits iguales a 1 (o a 0) y un bit igual a 0 (o a 1). Ejemplos:

s	Código
0	0
1	10
2	110
3	1110
4	11110
5	111110
6	1111110
7	11111110
8	111111110

- Como puede desprenderse de la Ecuación 11.1 a codificación unaria es óptima si

$$p(s) = 2^{-(s+1)} \tag{17.1}$$

donde $s = 0, 1, \cdots$.

- En el Apéndice 39.55 aparece una implementación de un codec unario.

Capítulo 18

Codificación de Rice

La codificación de Rice

- Al igual que la codificación unaria, la codificación de Rice se utiliza para codificar secuencias de símbolos con probabilidades de ocurrencia inversamente proporcional a sus índices en el alfabeto fuente [17].
- La codificación de Rice es óptima cuando

$$p(s) = 2^{-\left(\lfloor \frac{s+1}{2^k} \rfloor + 1\right)} \tag{18.1}$$

  donde $s = 0, 1, \cdots$ y $k = 0, 1, \cdots$.
- Como puede verse la codificación de Rice es una generalización de la codificación unaria, siendo equivalente a ésta si $k = 0$.
- k define la "pendiente" de la distribución geométrica que rige las probabilidades de los símbolos. Si k aumenta, la pendiente es menor.

18.1. Codificador

1. Sea $m \leftarrow 2^k$.
2. Emitir $\lfloor s/m \rfloor$* usando un código unario (de $\lfloor s/m \rfloor + 1$ bits).
3. Emitir los k bits menos significativos de s usando un código binario.

- En el Apéndice 39.53 se puede encontrar una implementación de la codificación de Rice.

*$\lfloor \cdot \rfloor$ denota "al mayor entero menos grande que".

18.2. Ejemplo de codificación

Supongamos que $k = 1$ y $s = 7$. Entonces $m \leftarrow 2$. Emitimos $\lfloor s/m \rfloor = 3$ como un número unario (de 4 bits) generando a la salida 1110 y seguidamente emitimos el bit de menos peso de 7 que es 1. Por lo tanto, el código asociado al símbolo 7, para $k = 1$ es el 11101.

18.3. Descodificador

1. Sea s el número de unos consecutivos a la entrada.
2. Sea x los siguientes $k+1$ bits de entrada.
3. $s \leftarrow s \times 2^k + x$.

18.4. Ejemplo de descodificación

Descodifiquemos el código 11101 para $k = 1$. En la entrada tenemos 3 unos consecutivos, con lo que $s \leftarrow 3$. Ahora leemos el siguiente 0 y el siguiente 1. Efectuamos la operación $s \leftarrow 3 \times 2^1 + 1 = 6 + 1 = 7$.

Capítulo 19

Codificación de Golomb

La codificación de Golomb

- En 1966, S. W. Golomb diseñó un algoritmo de codificación equivalente a la codificación de Huffman en eficiencia de compresión, pero que puede ser calculada más rápidamente.
- La codificación de Golomb es una generalización de la codificación de Rice que permite acomodar un mayor número de distribuciones de probabilidad de los símbolos. Dichas distribuciones siguen siendo distribuciones geométricas en las que la probabilidad de ocurrencia de un símbolo es inversamente proporcional al valor de su índice en el alfabeto fuente [8].
- La codificación de Golomb es óptima si se cumple que

$$p(s) = 2^{-\left(\lfloor \frac{s+1}{m} \rfloor + 1\right)} \tag{19.1}$$

donde $= 0, 1, \cdots$ y $m = 0, 1, \cdots$.

- De forma parecida a la codificación de Rice (que se obtiene cuando $m = 2^k$), m define la pendiente de la distribución geométrica de probabilidad.
- En la siguiente tabla se muestran los códigos de Golomb (Rice y unario) para los primeros símbolos del alfabeto fuente:

Golomb Rice	$m=1$ $k=0$	$m=2$ $k=1$	$m=3$	$m=4$ $k=2$	$m=5$	$m=6$	$m=7$	$m=8$ $k=3$
$s=0$	0	00	00	000	000	000	000	0000
1	10	01	010	001	001	001	0010	0001
2	110	100	011	010	010	0100	0011	0010
3	1110	101	100	011	0110	0101	0100	0011
4	11110	1100	1010	1000	0111	0110	0101	0100
5	1^50	1101	1011	1001	1000	0111	0110	0101
6	1^60	11100	1100	1010	1001	1000	0111	0110
7	1^70	11101	11010	1011	1010	1001	1000	0111
8	1^80	111100	11011	11000	10110	10100	10010	10000
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

18.4 Ejemplo de descodificación

- En el Apéndice 39.14 es posible encontrar una implementación de la codificación de Golomb.

19.1. Codificador

1. Sea $k \leftarrow \lceil \log_2(m) \rceil$*.
2. Sea Sea $r \leftarrow s \bmod m$ (el resto de la división entera).
3. Sea $t \leftarrow 2^k - m$.
4. Emitir $(s \operatorname{div} m)$ usando un código unario.
5. Si $r < t$:
   a) Emitir los $k-1$ bits menos significativos de r usando un código binario.
6. Si no:
   a) $r \leftarrow r + t$.
   b) Emitir los k bits menos significativos de r usando un código binario.

*$\lfloor \cdot \rfloor$ denota "al menor entero más grande que".

19.2. Ejemplo de codificación

Supongamos que $m = 7$ y $s = 8$. Entonces $k \leftarrow 3$, $r \leftarrow 1$ y $t \leftarrow 8 - 7 = 1$. Emitimos $\lfloor s/m \rfloor = 1$ como un número unario (de 2 bits) generando a la salida 10. Como no se cumple que $1 < 1$, calculamos $r \leftarrow 1 + 1 = 2$ y emitimos el número 2 usando un código binario de 3 bits. Por tanto, el código emitido es 10010.

19.3. Descodificador

1. Sea $k \leftarrow \lceil \log_2(m) \rceil$ y $t \leftarrow 2^k - m$.
2. Sea $s \leftarrow$ el número de unos consecutivos de entrada.
3. Sea $x \leftarrow$ los siguientes $k-1$ bits de entrada.
4. Si $x < t$:
   a) $s \leftarrow s \times m + x$.
5. Si no:
   a) $x \leftarrow x \times 2 +$ siguiente bit de entrada.
   b) $s \leftarrow s \times m + x - t$.

19.4. Ejemplo de descodificación

Descodifiquemos el código 10010 para $m = 7$ ($k \leftarrow 3$ y $t \leftarrow 1$). En la entrada tenemos sólo un 1 uno consecutivos, con lo que $s \leftarrow 1$. Ahora leemos los siguientes 2 bits en $x \leftarrow 1$. Como $2 > 1$, calculamos $x \leftarrow x \times 2 + 0 = 2$. Finalmente, $s \leftarrow 1 \times 7 + 2 - 1 = 8$.

19.4 Ejemplo de descodificación

Capítulo 20

La transformada de texto basada en predicción

- Si analizamos cuidadosamente la MTF veremos que en muchas ocasiones colocar el símbolo transformado en la cabeza de la lista de

símbolos es una apuesta demasiado fuerte para un símbolo que haya podido ocurrir, por ejemplo, una sola vez.

- Una forma de solucionar este problema consiste en mantener junto a cada símbolo de la lista, una cantidad que nos indique su probabilidad de ocurrencia. Así, la lista estará ordenada por la probabilidad del símbolo.
- Si utilizamos además un modelo probabilístico basado en el contexto obtenemos la transformada basada en predicción o PBT (Prediction Based Transform).
- El Apéndice 39.51 contiene una implementación de la PBT.

20.1. Codificador de orden 0

1. Idéntico al paso 1 del codificador MTF. Además, cada nodo de la lista contemplará un recuento que indica el número de veces que se ha usado el símbolo.

2. Mientras existan símbolos que codificar:

   a) Sea s el siguiente símbolo de entrada.

   b) Buscar s en $L[]$ y emitir como código su posición en dicha lista.

   c) Incrementar el recuento de s.

   d) Ordenar la lista en función del recuento de s. Puesto que la lista está siempre ordenada, este paso puede hacerse en general en un tiempo inferior a $\log_2(r)$ donde r es el número de símbolos en la lista. s se coloca a la cabeza de todos los símbolos que tienen un recuento igual que él.

20.2. Ejemplo de codificación

lista					entrada	salida
$\cdots$	a, 0	b, 0	c, 0	$\cdots$	b	b
b, 1	$\cdots$	a, 0	c, 0	$\cdots$	a	b
a, 1	b, 1	$\cdots$	c, 0	$\cdots$	a	0
a, 2	b, 1	$\cdots$	c, 0	$\cdots$	a	0
a, 3	b, 1	$\cdots$	c, 0	$\cdots$	a	0
a, 4	b, 1	$\cdots$	c, 0	$\cdots$	a	0
a, 5	b, 1	$\cdots$	c, 0	$\cdots$	a	0
a, 6	b, 1	$\cdots$	c, 0	$\cdots$	b	1
a, 6	b, 2	$\cdots$	c, 0	$\cdots$	b	1
a, 6	b, 3	$\cdots$	c, 0	$\cdots$	a	0
a, 7	b, 3	$\cdots$	c, 0	$\cdots$	b	1
a, 7	b, 4	$\cdots$	c, 0	$\cdots$	a	0
a, 8	b, 4	$\cdots$	c, 0	$\cdots$	a	0
a, 9	b, 4	$\cdots$	c, 0	$\cdots$	c	c
a, 9	b, 4	c, 1	$\cdots$	$\cdots$	a	0
a,10	b, 4	c, 1	$\cdots$	$\cdots$	a	0
a,11	b, 4	c, 1	$\cdots$	$\cdots$	b	1

20.3. Descodificador de orden 0

1. Idéntico al paso 1 del codificador PBT d eorden 0.
2. Mientras existan símbolos que descodificar:
   a) Sea c el siguiente código de entrada:
   b) Emitir como símbolo $s \leftarrow L[c]$.
   c) Idem al paso 2.d del codificador.

20.4. Codificador de orden N

1. Sea $c[i]$ el contexto actual de orden i y sea $L[c[i]]$ la lista de predicción asociada al contexto $c[i]$ o lo que es lo mismo, la lista de símbolos que alguna vez han sucedido a la cadena $c[i]$.
2. Sea $i \leftarrow k$ el orden de predicción.
3. Sea $H \leftarrow \varnothing$ la lista de símbolos diferentes de predicción probados.
4. Mientras $s \notin L[c[i]]$:
   a) $H \leftarrow H + L[c[i]]$, acumulando aquellos símbolos que son probados por primera vez. Esto significa que H no almacenará más de una vez el mismo símbolo en ningún momento.
   b) Actualizar s en $L[c[i]]$ según el método PBT de orden 0.
   c) $i \leftarrow i - 1$.
5. Sea e la posición de s en $L[c[i]]$ (el error de predicción).

6. $e \leftarrow e + size(H)$ (el número de símbolos en H). Así, sumamos todos los errores de predicción para contextos superiores.

7. Emitir el código correspondiente al error de predicción e.

8. Actualizar s en $L[c[i]]$ según el método PBT de orden 0.

20.5. Ejemplo de codificación

Orden máximo de predicción igual a uno.

entrada	salida	contextos afectados
a	a	$L[] = \{\mathtt{a}, 1 \cdots\}$
b	b	$L[\mathtt{a}] = \{\mathtt{b}, 1\}, L[] = \{\mathtt{b}, 1\ \mathtt{a}, 1 \cdots\}$
a	1	$L[\mathtt{b}] = \{\mathtt{a}, 1\}, L[] = \{\mathtt{a}, 2\ \mathtt{b}, 1 \cdots\}$
b	0	$L[\mathtt{a}] = \{\mathtt{b}, 2\}$
c	c	$L[\mathtt{b}] = \{\mathtt{c}, 1\ \mathtt{a}, 1\}, L[] = \{\mathtt{a}, 2\ \mathtt{c}, 1\ \mathtt{b}, 1 \cdots\}$
b	2	$L[\mathtt{c}] = \{\mathtt{b}, 1\}, L[] = \{\mathtt{b}, 2\ \mathtt{a}, 2\ \mathtt{c}, 1 \cdots\}$
a	1	$L[\mathtt{b}] = \{\mathtt{a}, 2\ \mathtt{c}, 1\}$
b	0	$L[\mathtt{a}] = \{\mathtt{b}, 3\}$
a	0	$L[\mathtt{b}] = \{\mathtt{a}, 3\ \mathtt{c}, 1\}$
b	0	$L[\mathtt{a}] = \{\mathtt{b}, 4\}$
a	0	$L[\mathtt{b}] = \{\mathtt{a}, 4\ \mathtt{c}, 1\}$
a	1	$L[\mathtt{a}] = \{\mathtt{b}, 4\ \mathtt{a}, 1\}, L[] = \{\mathtt{a}, 3\ \mathtt{b}, 2\ \mathtt{c}, 1 \cdots\}$
a	1	$L[\mathtt{a}] = \{\mathtt{b}, 4\ \mathtt{a}, 2\}$
a	1	$L[\mathtt{a}] = \{\mathtt{b}, 4\ \mathtt{a}, 3\}$
a	0	$L[\mathtt{a}] = \{\mathtt{a}, 4\ \mathtt{b}, 4\}$
a	0	$L[\mathtt{a}] = \{\mathtt{a}, 5\ \mathtt{b}, 4\}$
a	0	$L[\mathtt{a}] = \{\mathtt{a}, 6\ \mathtt{b}, 4\}$

Capítulo 21

La transformada de Burrows-Wheeler

21.1. El orden de los símbolos es importante

- La BWT (Burrows-Wheeler Transform) es un algoritmo de ordenación de cadenas reversible. La salida está formada por los mismos símbolos que tenemos a la entrada, pero en un orden diferente. Es precisamente dicho orden el que posibilita su compresión eficiente utilizando técnicas RLE.
- En la BWT la secuencia de símbolos se procesa por bloques, siendo la tasa de la compresión proporcional al tamaño de los mismos.
- En el Apéndice 39.6 encontramos una implementación de la BWT.

21.2. Transformada directa

Para un tamaño de bloque N prefijado de antemano, el algoritmo de la transformada BWT realiza los siguientes pasos:

1. Leer la secuencia N de símbolos.
2. Construir una matriz cuadrada de lado igual al tamaño del bloque N, donde la primera fila es la secuencia original, la segunda es la secuencia desplazada un símbolo a la izquierda de forma cíclica, etc.
3. Ordenar lexicográficamente la matriz por filas. Este es el paso pesado de algoritmo y se ejecuta en un tiempo proporcional a $N \times \log_2(N)$.
4. Buscar en la columna $N-1$ (la de más a la derecha) la fila en la que se encuentra el primer símbolo de la secuencia original. Sea este valor i.
5. La transformada BWT de la secuencia de entrada está formada por el contenido de la columna $N-1$ (la última) y el índice i.

21.3. Ejemplo de codificación

Matriz inicial

<a>babcbababaaaaaaa	0
babcbababaaaaaaa<a>	1
abcbababaaaaaaa<a>b	2
bcbababaaaaaaa<a>ba	3
cbababaaaaaaa<a>bab	4
bababaaaaaaa<a>babc	5
ababaaaaaaa<a>babcb	6
babaaaaaaa<a>babcba	7
abaaaaaaa<a>babcbab	8
baaaaaaa<a>babcbaba	9
aaaaaaa<a>babcbabab	10
aaaaaa<a>babcbababa	11
aaaaa<a>babcbababaa	12
aaaa<a>babcbabababaa	13
aaa<a>babcbababaaaa	14
aa<a>babcbabababaaaaa	15
a<a>babcbabababaaaaaa	16

Matriz ordenada

aaaaaaa<a>babcbabab	10	0	
aaaaaa<a>babcbababa	11	1	
aaaaa<a>babcbabababa	12	2	
aaaa<a>babcbababaaa	13	3	
aaa<a>babcbababaaaa	14	4	
aa<a>babcbabababaaaaa	15	5	
a<a>babcbababaaaaaa	16	6	
abaaaaaaa<a>babcbab	8	7	
ababaaaaaaa<a>babcb	6	8	
<a>babcbabababaaaaaa	0	9	
abcbababaaaaaaa<a>b	2	10	
baaaaaaa<a>babcbaba	9	11	
babaaaaaaa<a>babcba	7	12	
bababaaaaaaa<a>babc	5	13	
babcbababaaaaaaa<a>	1	14	$\leftarrow i$
bcbababaaaaaaa<a>ba	3	15	
cbababaaaaaaa<a>bab	4	16	

21.4. Transformada inversa

1. Sea $L[]$ la secuencia transformada. $L[]$ se ordena lexicográficamente en un tiempo proporcional a $N \times \log_2(N)$. Sea la secuencia ordenada $O[]$, usando el mismo algoritmo de ordenación usado en la transformada directa.

2. Calcular el vector de transformación $T[]$ de forma que si $O[j] = L[l]$ (donde l es el primer carácter de $L[]$ que recorrido secuencialmente comenzando desde el índice 0 cumple dicha condición), entonces $T[j] = l$. El carácter $L[l]$ sólo puede usarse una vez de forma que todos los elementos de $T[]$ son diferentes.

3. Sea $k \leftarrow i$ (i es el índice pasado como entrada).

4. Ejecutar N veces:

   a) Emitir el símbolo almacenado en $L[k]$.

   b) Hacer $k \leftarrow T[k]$.

21.5. Ejemplo de descodificación

Siguiendo con nuestro ejemplo, la secuencia transformada ordenada ($O[]$) es igual a aaaaaaaaaaabbbbbc. $O[0] =$ a es el mismo símbolo que $L[1]$ y por lo tanto $T[0] = 1$. $O[1] =$ a es el símbolo $L[2]$ lo que indica que $T[1] = 2$, etc. El vector de transformación completo es 1, 2, 3, 4, 5, 6, 9, 11, 12, 14, 15, 0, 7, 8, 10, 16 y 13.

Una vez calculado $T[]$ ejecutamos el lazo del paso 4. Emitimos $L[14] =$ a y $k \leftarrow 10$. Emitimos $L[10] =$ b y $k \leftarrow 15$. Emitimos $L[15] =$ a y $k \leftarrow 16$, etc. Finalmente la secuencia original es reconstruida.

Capítulo 22

Un ejemplo real: `gzip`

Acerca de `gzip`

- `gzip` es usado en los sistemas operativos Unix. Es muy rápido, especialmente cuando descomprime.
- Se basa en dos algoritmos diferentes: LZ77 [25] y la Codificación de Huffman [10].
- Como la mayoría de los compresores de texto, tratan las secuencias de símbolos *on-the-fly*. Esto significa que sólo necesita ver, en un instante dado, una pequeña parte de la secuencia para generar los códigos de compresión adecuados.
- La Web de `gzip` es `http://www.gzip.org/`.

22.1. Compresión de "akiyo"

- Para comprimir una secuencia de imágenes usando `gzip` debemos tratarlas como un flujo de símbolos, lo que puede conseguirse si consideramos cada componente como un carácter de 8 bits. Vamos a realizarlo de 3 formas distintas:

  1. Compresión del fichero `akiyo_352x288x300_30Hz.raw`:
     - Estructura del fichero:
       ```
       rgb = unsigned char[3];
       image = rgb[352][288];
       sequence = image[300];
       ```
     - Comando de compresión:
       ```
       gzip -9 akiyo_352x288x300_30Hz.raw
       ```
     - Tasa de compresión alcanzada: $1{,}70$.*
     - Nótese que para mostrar un pixel del vídeo, sólo es necesario descomprimir un elemento `rgb`.

*Resultado de dividir el tamaño del fichero original entre el tamaño del fichero comprimido. En concreto: $91.238.400/53.747.338$.

2. Compresión del fichero `akiyo_352x288x300_30Hz.raw`:
   - Estructura del fichero:
     ```
     R = unsigned char[352][288];
     G = unsigned char[352][288];
     B = unsigned char[352][288];
     image = {R,G,B};
     sequence = image[300];
     ```
   - Comando de compresión:
     ```
     gzip -9 akiyo_352x288x300_30Hz.raw
     ```
   - Tasa de compresión: $1{,}43$*, inferior a la del caso anterior. Esto es consecuencia de no se está explotando adecuadamente la correlación entre las componentes de color `R`, `G` y `B` de cada imagen (véase el apéndice 34).
   - En este caso, para mostrar un pixel, es necesario haber descomprimido un elemento `image`, lo que puede suponer un consumo excesivo de memoria para algunas aplicaciones.

*91.238.400/63.910.449.

3. Compresión de cada imagen de la secuencia {akiyo000, akiyo001, ..., akiyo299} de forma independiente:
   - Estructura de los ficheros:
     ```
     rgb = unsigned char[3];
     image = rgb[352][288];
     ```
   - Comandos:
     ```
     eyuvtoppm --width 352 --height 288 \
       akiyo_352x288x300_30Hz.raw > tmp_file
     split -d -a 3 -b $[352*288*3] tmp_file akiyo
     gzip -9 akiyo???
     ```
   - Tasa de compresión alcanzada: $1{,}70^*$.

- Nótese que, en los dos primeros casos, la secuencia comprimida no posee ningún tipo de escalabilidad. Esto significa que para acceder a una imagen, por ejemplo, hay que descomprimir necesariamente todas las anteriores. Sin embargo, en el tercero, sí que existe escalabilidad temporal porque podemos acceder a cada imagen de forma independiente.

*$91.238.400/53.673.243$

- Finalmente, esta última forma de compresión también es la más efectiva porque el final de una imagen no está estadísticamente correlacionado con el comienzo de la siguiente imagen.

Parte II

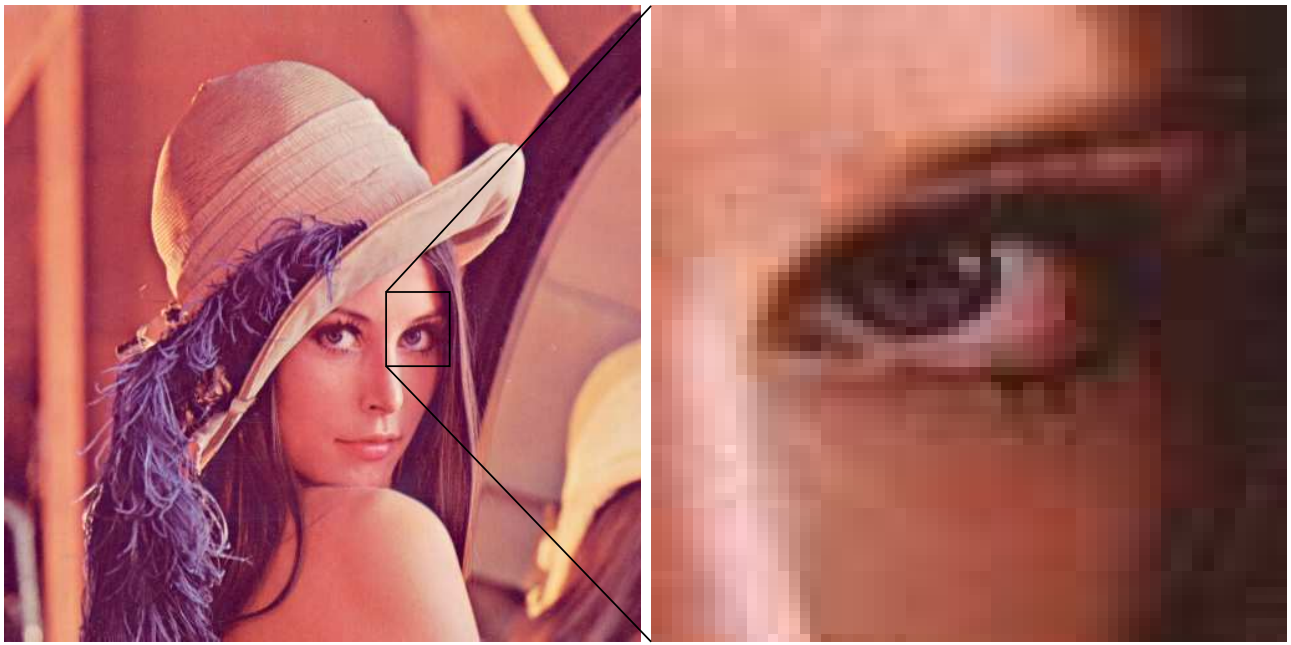
Compresión de imágenes

Capítulo 23

La compresión de imágenes

23.1. Fundamentos

- Los compresores de imágenes explotan la **redundancia espacial** que existe en las imágenes. Nótese que la redundancia espacial genera también redundancia estadística y por eso, los compresores de texto pueden comprimir las imágenes.



- Dependiendo de si el compresor codifica toda la información de la imagen o no, hablaremos de compresores lossy y lossless.

Capítulo 24

PNG (Portable Network Graphics)

El formato de imagen PNG

- URL: `http://www.libpng.org/pub/png`.
- Es un formato de imagen que nació con la idea de reemplazar el formato GIF (Graphics Interchange Format), protegida por una patente sobre el LZW [24], y para extender la funcionalidad del formato TIFF (Tagged Image File Format).
- Es libre, es decir, no hay que pagar nada a nadie por usarlo.
- Utiliza una implementación del algoritmo LZ77 [25] llamado "deflate".
- Es lossless.

- Soporta:

  1. Imágenes indexadas (con paleta) con hasta 256 colores.
  2. Compresión y descompresión on-the-fly (streamability). Esto significa que las imágenes no tienen que almacenarse completamente en memoria para comprimirse o descomprimirse.
  3. Visualización progresiva (escalabilidad en calidad), usando un número reducido de niveles.
  4. Imágenes en color con hasta 48 bpp (16 bits por componente).
  5. Imágenes en tonos de gris con hasta 16 bpp.
  6. Transparencias (alpha channel).
  7. Información sobre el *gamma* de la imagen para garantizar su presentación adecuada en cualquier display.

24.1. Compresión de “akiyo”

- El compresor PNG permite comprimir imágenes independientes, no secuencias de estas. Por tanto, utilizaremos el siguiente script:

```
for i in akiyo???
do
  echo $i
  pnmtopng $i > $i.png
done
```

para comprimir las imágenes, donde cada imagen `akiyo???` es una imagen PPM (Portable Pixel Map), con la estructura:

```
rgb = unsigned char[3];
image = rgb[352][288];
PPM_image = {
  "P6\n",         /* Magic number */
  "352 288\n",    /* Columns and rows of the image */
  "255\n",        /* Maximun level for the RGB components
  image           /* The image data */
}
```

- La tasa de compresión alcanzada (cabeceras incluidas) ha sido: $2{,}25$ bps $(91.238.400/40.516.570)$, que resulta bastante buena tratándose de una compresión sin pérdidas.
- Dado que el compresor se aplica de forma independiente a cada imagen, el descompresor tiene acceso a cada una de ellas también de forma independiente. Por tanto, este codec posee escalabilidad temporal porque nos permite acceder a la imagen que nosotros deseamos sin necesidad de descomprimir las demás.

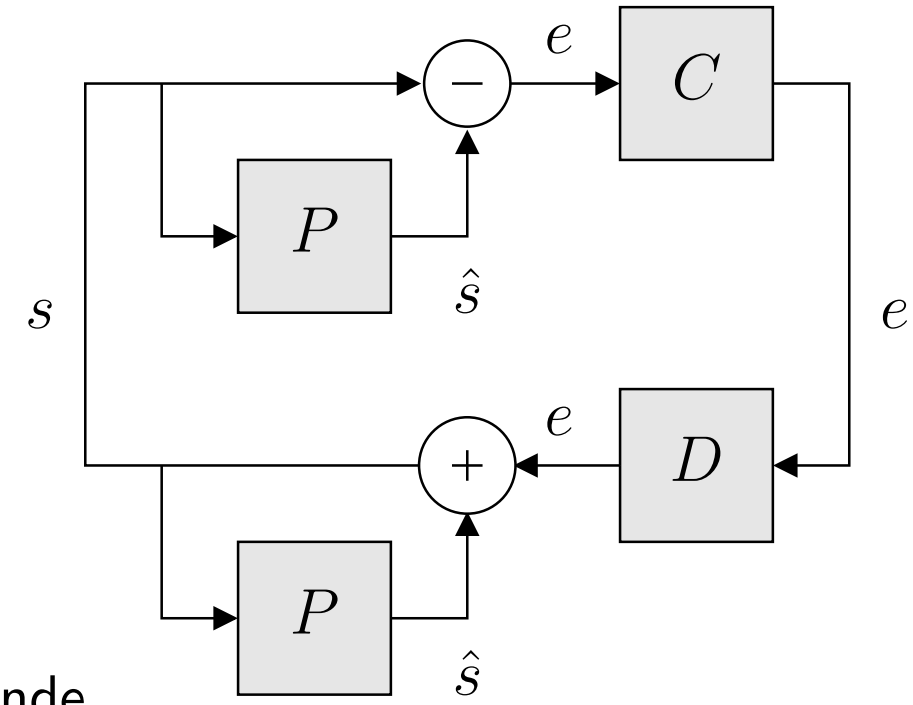
Capítulo 25

Lossless JPEG

Lossless JPEG

- Permite comprimir imágenes de forma totalmente reversible.
- Basado en un predictor espacial y un codificador de longitud variable (codificación de Huffman o aritmética).
- Implementación en `ftp://ftp.cs.cornell.edu/pub/multimed/ljpg.tar.Z`.

- Su diagrama de bloques es



P = Predictor

C = Compresor

D = Descompresor

s = Punto a predecir

e = Error de predicción

$\hat{s}$ = Predicción

donde

Contexto de Predicción

a	b
c	$\hat{s}$

Predictores

P_0	$\hat{s} \leftarrow 0$
P_1	$\hat{s} \leftarrow a$
P_2	$\hat{s} \leftarrow b$
P_3	$\hat{s} \leftarrow c$
P_4	$\hat{s} \leftarrow a + b - c$
P_5	$\hat{s} \leftarrow a + (b - c)/2$
P_6	$\hat{s} \leftarrow b + (a - c)/2$
P_7	$\hat{s} \leftarrow (b + c)/2$

25.1. Codificador

1. Generar $\hat{s}$.
2. Calcular el error de predicción $e \leftarrow s - \hat{s}$.
3. Codificar e.

25.2. Descodificador

1. Generar $\hat{s}$ (idéntico al paso 1 del compresor).
2. Descodificar e.
3. Calcular el valor del punto $s \leftarrow e + \hat{s}$.

25.3. Compresor de Huffman

- El compresor de Huffman utilizado en LS-JPEG utiliza un modelo probabilístico estático (ver tablas a continuación):

  1. Buscar e en $DIFF$ y seleccionar $SSSS$.
  2. Codificar $SSSS$ según el código base.
  3. Si $e > 0$ entonces:
     a) Codificar e usando un número binario de $SSSS$ bits. El bit más significativo de e va a ser siempre 1.
  4. Si no:
     a) Codificar $e - 1$ usando un número binario de $SSSS$ bits. El bit más significativo de e va a ser siempre 0.

Categorías

$SSSS$	$DIFF$
0	0
1	-1,1
2	-3,-2,2,3
3	-7,$\cdots$,-4,4,$\cdots$,7
4	-15,$\cdots$,-8,8,$\cdots$,15
5	-31,$\cdots$,-16,16,$\cdots$,31
6	-63,$\cdots$,-32,32,$\cdots$,63
7	-127,$\cdots$,-64,64,$\cdots$,127
8	-255,$\cdots$,-128,128,$\cdots$,255
9	-511,$\cdots$,-256,256,$\cdots$,511
10	-1023,$\cdots$,-512,512,$\cdots$,1023
11	-2047,$\cdots$,-1024,1024,$\cdots$,2043
12	-4095,$\cdots$,-2048,2048,$\cdots$,4095
13	-8191,$\cdots$,-4096,4096,$\cdots$,8191
14	-16383,$\cdots$,-8192,8192,$\cdots$,16383
15	-32767,$\cdots$,-16384,16384,$\cdots$,32767
16	-32768

Códigos de Huffman

$SSSS$	Longitud	Código base
0	3	00
1	4	010
2	5	011
3	5	100
4	7	101
5	8	110
6	10	1110
7	5	11110
8	6	111110
9	7	1111110
10	8	11111110
11	9	111111110
12	10	1111111110
13	11	11111111110
14	12	111111111110
15	-	1111111111110
16	-	11111111111110

25.4. Ejemplo de codificación

Por ejemplo, si $e = 5$, $SSSS = 3$. Emitimos 100 y a continuación 101 usando 3 bits. Si $e = -9$, $SSSS = 4$. Emitimos 101 seguido de los 4 bits menos significativos de -10 que son 0110.

25.5. Descompresor de Huffman

- El algoritmo del descompresor entrópico en LS-JPEG consiste en:

  1. Descodificar la categoría $SSSS$ usando el código base.
  2. $e \leftarrow$ siguientes $SSSS$.
  3. Ahora descodificamos la magnitud. Sea $x \leftarrow$ siguiente bit de entrada.
  4. Si $x \neq 0$ entonces:
     a) $e \leftarrow 2^{SSSS-1} +$ siguientes $(SSSS-1)$ bits.
  5. Si no:
     a) $e \leftarrow (-1)$ AND $(2^{SSSS-1} +$siguientes $(SSSS-1)$ bits$+1)$.

25.6. Ejemplo de descompresión

Por ejemplo, si el código es 100101, los 3 primeros bits indican que se trata de la categoría $SSSS = 3$. Leemos el siguiente bit de entrada ($x \leftarrow 1$) y como es distinto de cero, llegamos a que $e \leftarrow 2^2+$ siguientes dos bits de entrada $(01) = 101 = 5$.

Cuando el código es 1010110, los 3 primeros bits indican que $SSSS = 4$. Leemos el siguiente bit de entrada y vemos que es 0. Por tanto (usando 8 bits de precisión) $e \leftarrow$ 11111111 AND (0000+110+1) = 11110111 $= -9$ (los 4 bits más significativos de 0111 se suponen 1).

Capítulo 26

LOCO-I (JPEG-LS)

- LOCO-I [23] (*LOw COmplexity, context-based, lossless Image compression algorithm*) usa un codificador de Rice como *codec* entrópico y un modelo espacial basado en el contexto.

25.6 Ejemplo de descompresión

- Por ahora, HP sólo distribuye el codec en código ejecutable. Este puede ser encontrado en:
  `http://www.hpl.hp.com/loco/locodown.htm`
- LOCO-I tiene la ventaja de ser extraordinariamente rápido, tanto comprimiendo como descomprimiendo.

26.1. Codificador

1. **Inicialización de los contextos de predicción**:

   a) Sea Q el contexto actual. LOCO-I considera un máximo de 1094 contextos distintos.

   b) Sea $N[Q]$ el número de ocurrencias de cada contexto. Inicialmente $N[Q] \leftarrow 0 \; \forall Q$.

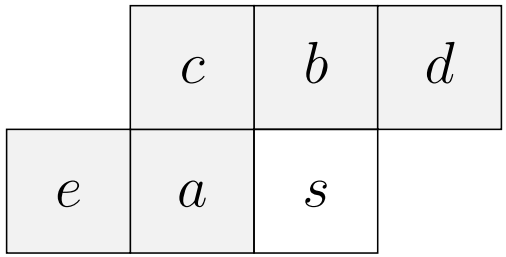
   c) Sea $B[Q]$ el error de predicción acumulado en cada contexto. Inicialmente $B[Q] \leftarrow 0 \; \forall Q$.

   d) Sea $A[Q]$ la suma de los valores absolutos de los errores de predicción para cada contexto. Inicialmente $A[Q] \leftarrow 0 \; \forall Q$.

   e) Sea $C[Q]$ los valores de cancelación del *bias*. El *bias* es un valor que sumado a la predicción espacial provoca que su media sea 0. Inicialmente $C[Q] \leftarrow 0 \; \forall Q$.

2. **Determinación del contexto de predicción** Q:

   a) Calcular el gradiente local. Para ello se efectúan las 4 diferencias (ver la siguiente figura):



$$\begin{aligned} g_1 &\leftarrow d - a \\ g_2 &\leftarrow a - c \\ g_3 &\leftarrow c - b \\ g_4 &\leftarrow b - e \end{aligned}$$

b) Cuantificar los gradientes según:

$$q_i \leftarrow \begin{cases} 0 & \text{si } g_i = 0 \\ 1 & \text{si } 1 \leq |g_i| \leq 2 \\ 2 & \text{si } 3 \leq |g_i| \leq 6 \\ 3 & \text{si } 7 \leq |g_i| \leq 14 \\ 4 & \text{en otro caso} \end{cases}$$

para $i = 1, 2, 3$ y

$$q_4 \leftarrow \begin{cases} 0 & \text{si } |g_4| < 0 \\ 1 & \text{si } 5 \leq g_4 \\ 2 & \text{en otro caso.} \end{cases}$$

3. **Cálculo del error de predicción** $M(e)$:

   a) Construir la predicción inicial:

$$\hat{s} \leftarrow \begin{cases} \min(a, b) & \text{si } c \geq \max(a, b) \\ \max(a, b) & \text{si } c \leq \min(a, b) \\ a + b - c & \text{en otro caso.} \end{cases}$$

   b) Cancelar el *bias*:

$$\hat{s} \leftarrow \begin{cases} \hat{s} + C[Q] & \text{si } g_1 > 0 \\ \hat{s} - C[Q] & \text{en otro caso.} \end{cases}$$

   c) Calcular el error de predicción:

$$e \leftarrow (s - \hat{s}) \bmod \beta,$$

donde β es el número de bits por punto. Esto provoca que el error de predicción sea proyectado desde el intervalo $[-\alpha+1, \alpha-1]$ al intervalo $[-\alpha/2, \alpha/2-1]$ donde $\alpha = 2^\beta$ es el tamaño del alfabeto.

d) Barajar los errores de predicción negativos y positivos generando una distribución de probabilidades que es decreciente con el índice del error. Esto se realiza según el siguiente mapeo:

$$M(e) \leftarrow \begin{cases} 2e & \text{si } e \geq 0 \\ 2|e| - 1 & \text{en otro caso.} \end{cases}$$

Tras dicho mapeo, los errores de predicción se ordenan según: 0, -1, $+1$, -2, $+2$, $\cdots$, $2^\beta - 1$.

4. **Codificación entrópica de $M(e)$ en el contexto Q:**

a) Se emite un código de Rice que codifica el símbolo $M(e)$ para $k = \lceil \log_2(A[Q]) \rceil$.

5. **Actualización del contexto Q:**

a) $B[Q] \leftarrow B[Q] + e$.

b) $A[Q] \leftarrow A[Q] + |e|$.

c) Si $N[Q] =$ RESET, entonces: (donde $64 \leq$ RESET ≤ 256)

   1) $A[Q] \leftarrow A[Q]/2$.
   2) $B[Q] \leftarrow B[Q]/2$.
   3) $H[Q] \leftarrow N[Q]/2$.

d) $N[Q] \leftarrow N[Q] + 1$.

e) La actualización del valor para la cancelación del *bias* es algo más compleja. Si el error de predicción inicial en el contexto Q no tiene media 0, el nivel de compresión decae severamente porque las medias de la distribución de Laplace real y la modelada no coinciden. Para evitar esto, $C[Q]$ almacena un valor proporcional a $B[Q]/N[Q]$ que es sumado a la predicción inicial para cancelar el *bias*.

   Además, $C[Q]$ es el encargado de solucionar otro problema derivado del barajamiento producido por $M(e)$ debido al cual,

se tiende a asignar un código más corto al error negativo que al respectivo error positivo.

Con todo esto, el algoritmo para la actualización de $C[Q]$ es el siguiente [12]:

1) Si $B[A] \leq -N[Q]$, entonces:
   - *a'* $B[Q] \leftarrow B[Q] + N[Q]$.
   - *b'* Si $C[Q] > -128$, entonces:
     - $C[Q] \leftarrow C[Q] - 1$.
   - *c'* Si $B[Q] \leq -N[Q]$, entonces:
     - $B[Q] \leftarrow -N[Q] + 1$.
2) Si no:
   - *a'* Si $B[Q] > 0$, entonces:
     - $B[Q] \leftarrow B[Q] - N[Q]$.
     - Si $C[Q] < 127$, entonces:
       - $\star$ $C[Q] \leftarrow C[Q] + 1$.
     - Si $B[Q] > 0$, entonces:
       - $\star$ $B[Q] \leftarrow 0$.

26.2. Descodificador

El descodificador es muy simétrico y recupera los símbolos s usando el siguiente algoritmo:

1. **Inicialización de los contextos de predicción** (ídem al paso 1 del compresor).
2. **Determinación del contexto** Q: (ídem al paso 2 del compresor).
3. **Descodificación de** $M(e)$: usando un descodificador de Rice con $k = \lceil \log_2(A[Q]) \rceil$.
4. **Determinación de** s:
   a) Computar la predicción inicial $\hat{s}$ como en el paso 3.a del compresor.
   b) Añadir el *bias* sobre $\hat{s}$:

$$\hat{s} \leftarrow \begin{cases} \hat{s} - C[Q] & \text{si } g_1 > 0 \\ \hat{s} + C[Q] & \text{en otro caso.} \end{cases}$$

c) Calcular el mapeo inverso generando la distribución de Laplace original:

$$e \leftarrow M^{-1}(M(e)) = \begin{cases} -M(e)/2 - 1 & \text{si } M(e) \text{ es impar} \\ M(e)/2 & \text{en otro caso.} \end{cases}$$

d) Calcular el símbolo s como:

$$s \leftarrow (e + \hat{s}) \bmod \beta.$$

5. **Actualización de** Q: (ídem al paso 5 del compresor).

26.3. El modo run-mode

LOCO-I usa un codificador de Rice que puede ser muy redundante cuando se generan distribuciones de probabilidad muy angostas, donde la probabilidad del error más frecuente (el 0) es mayor que $0{,}5$. Para evitar esto, LOCO-I tiene un modo especial de funcionamiento llamado *run-mode* en el que entra cuando $a = b = c = d$ (el contexto es constante) y provoca que $[q_1, q_2, q_3] \leftarrow [0, 0, 0]$. Mientras se permanece en este modo, no se emite ningún código.

El modo *run-mode* es abandonado si se encuentra un símbolo $s \neq a$ o si se ha alcanzado el final de una línea. Esto provoca que el número de símbolos r procesados en *run-mode* (la longitud de la serie) sea codificado. En este caso, LOCO-I emplea una tabla precalculada para encontrar el parámetro k.

Capítulo 27

El estándar JPEG (ISO/IEC 10918-1) [11]

JPEG

- Creado en 1992 por la ISO.
- Imágenes en color (RGB) y en tonos de gris.
- Calidad de compresión seleccionable $(0 - 100)$, aunque no el ratio de compresión [22].
- Reconstrucciones casi sin distorsión a ratios de 1 bpp para imágenes en color.
- Basado en la cuantificación de los coeficientes DCT (Discrete Cosine Transform), aplicada por bloques de 8×8 puntos.
- Permite descomprimir el fichero en pasadas, lo que significa cierto nivel de escalabilidad en calidad.
- Soporta también una codificación lossless (basada en una técnica DPCM: Differential Pulse Code Modulation) y jerárquica (basada en una pirámide de diferencias).

27.1. Compresor

Para una imagen RGB, el algoritmo básico consiste en:

1. Convertir la imagen al dominio YCbCr.
2. Submuestrear la crominancia.
3. Desplazar cada componente al rango $[-128, 127]$.
4. Para cada componente (Y, Cb y Cr):
   a) Aplicar la (8×8)-DCT a cada componente.
   b) Cuantificar los coeficientes DCT.
   c) Codificar entrópicamente los coeficientes cuantificados.

27.2. RGB → YCbCr

- El dominio RGB es generalmente más redundante que el YCbCr (redundancia espectral o redundancia inter-componente) y por esto los algoritmos de compresión rara vez se aplican directamente sobre el dominio RGB.
- Para realizar esto, primero se aplica una transformación que elimine correlación y por lo tanto decremente la entropía total de las 3 bandas.

- JPEG utiliza una aproximación de la transformación de color [14]

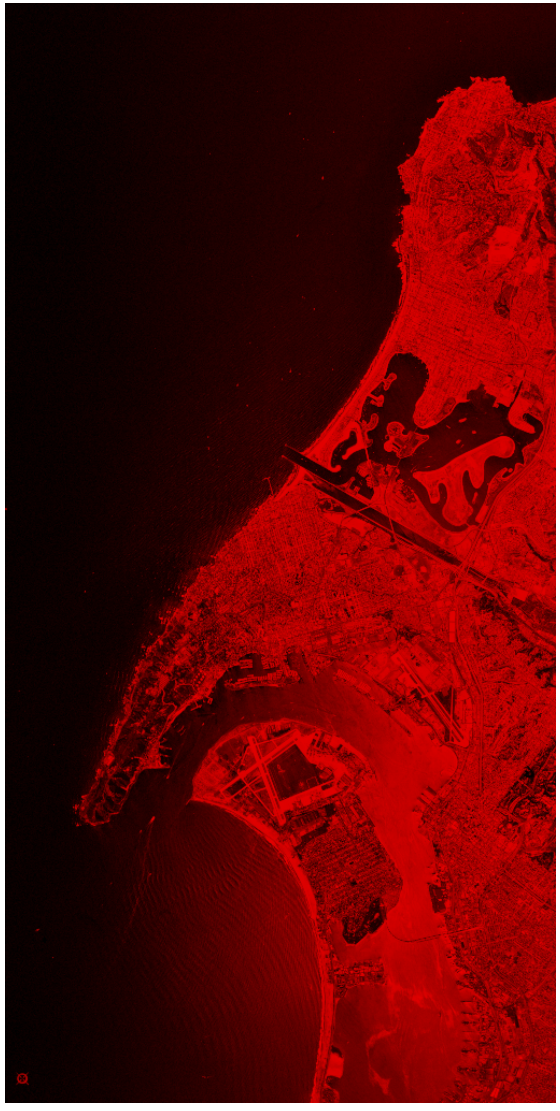
$$\begin{pmatrix} \mathsf{Y} \\ \mathsf{Cb} \\ \mathsf{Cr} \end{pmatrix} = \begin{pmatrix} 0,299 & 0,587 & 0,144 \\ -0,1687 & -0,3313 & 0,5 \\ 0,5 & -0,4187 & -0,0813 \end{pmatrix} \begin{pmatrix} \mathsf{R} \\ \mathsf{G} \\ \mathsf{B} \end{pmatrix} \quad (27.1)$$

que cuando el rango de los valores de entrada es $[0, 255]$ y se utiliza aritmética entera, queda como

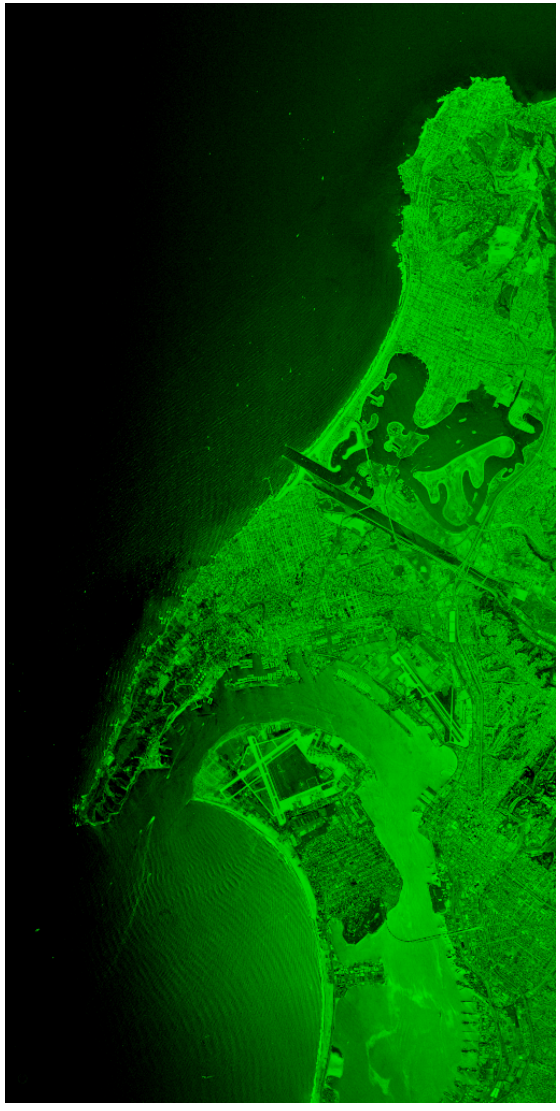
$$\begin{pmatrix} 256 \times \mathsf{Y} \\ 256 \times (\mathsf{Cb} - 128) \\ 256 \times (\mathsf{Cr} - 128) \end{pmatrix} = \begin{pmatrix} 77 & 150 & 29 \\ -44 & -67 & 131 \\ 131 & -110 & -21 \end{pmatrix} \begin{pmatrix} \mathsf{R} \\ \mathsf{G} \\ \mathsf{B} \end{pmatrix} \quad (27.2)$$

Ejemplo: San Diego (RGB)

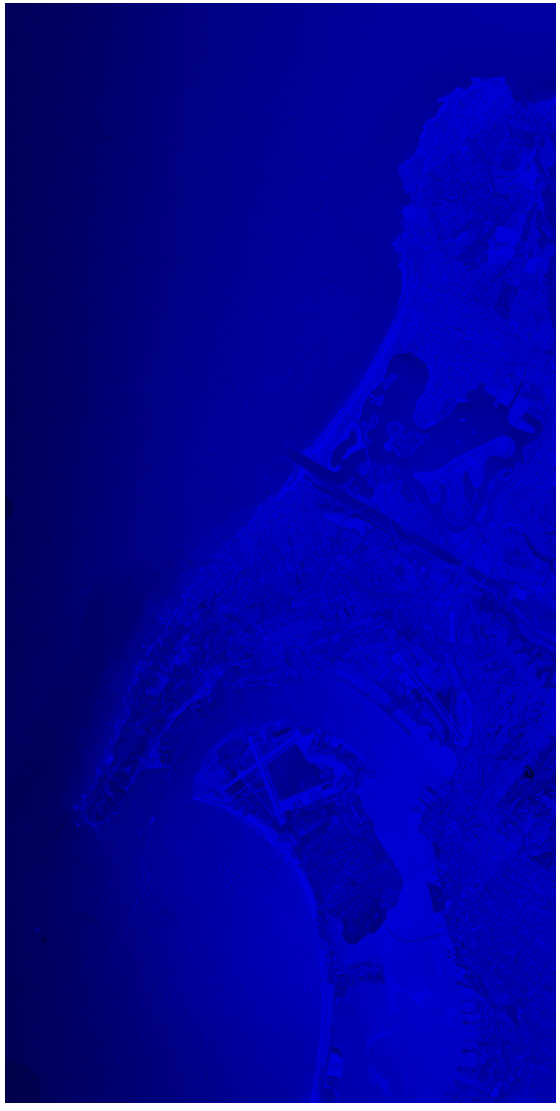
R, 7,51 bpp



G, 6,82 bpp



B, 7,04 bpp

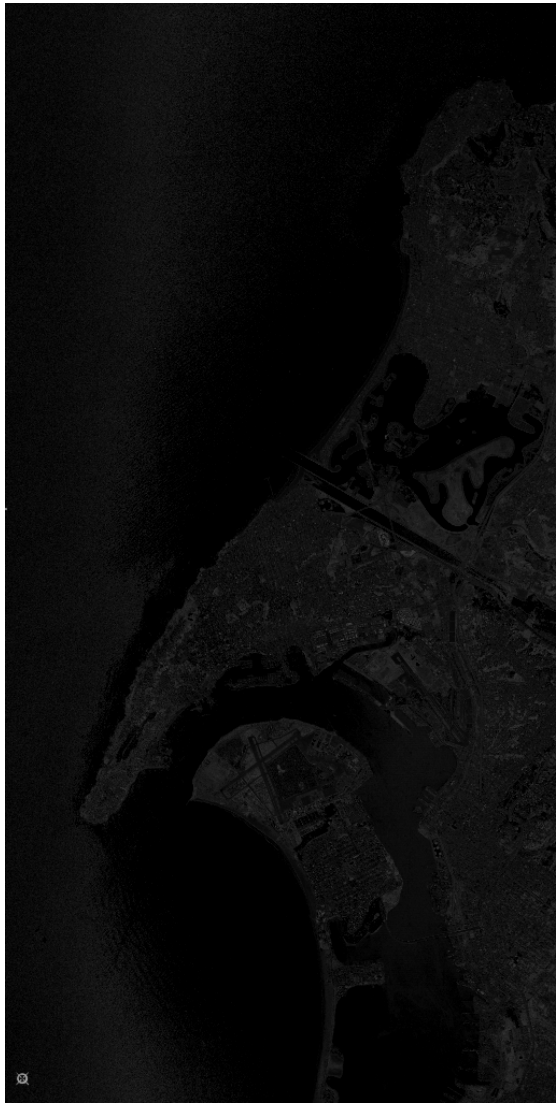


Total = 21,37 bpp

Y, 7,42 bpp

Cb, 6,86 bpp

Cr, 4,51 bpp



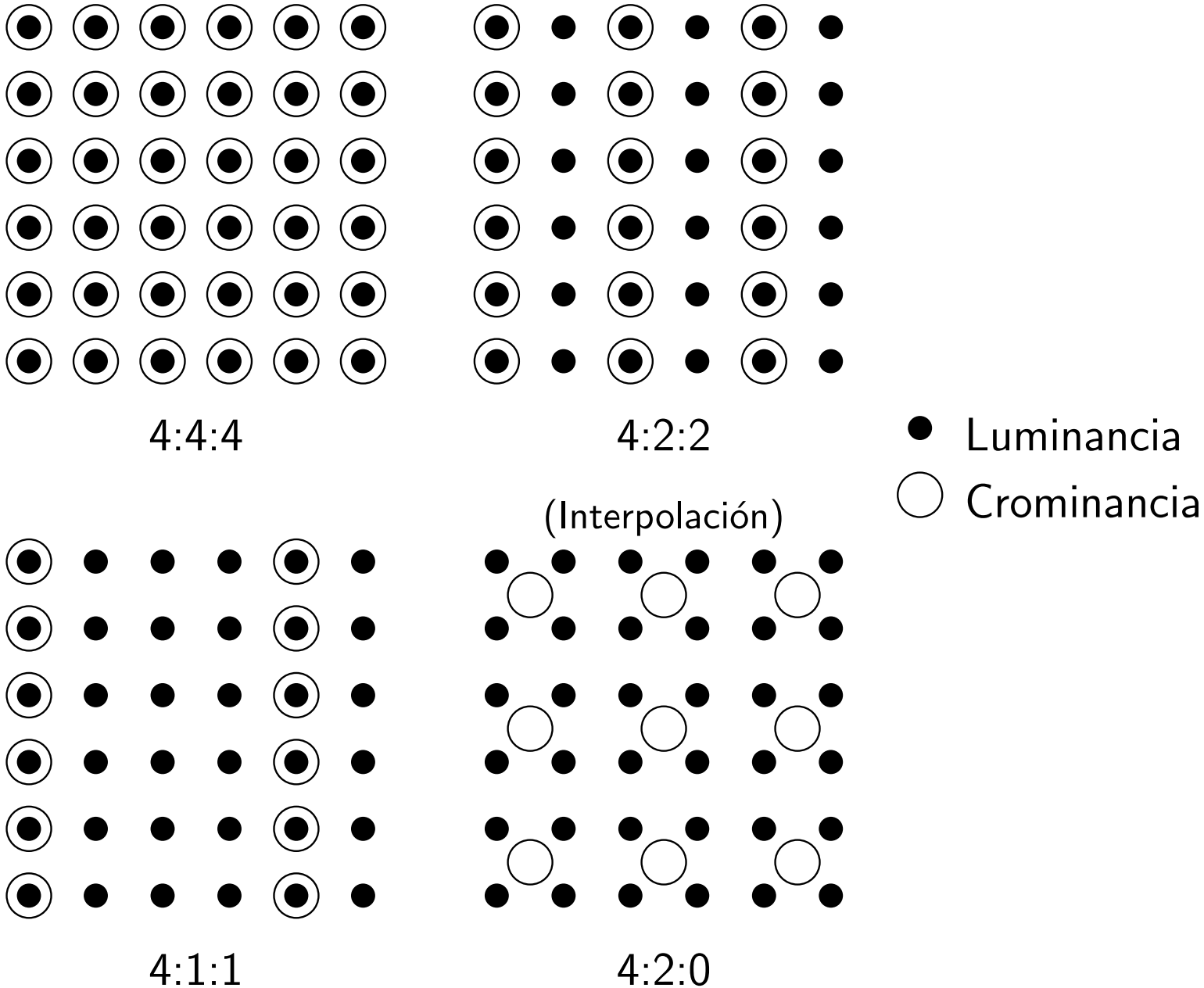
Total = 18,79 bpp

27.3. Submuestreo de la crominancia

- El sistema visual humano es más sensible a la pérdida de información en la luma que en el croma. Por eso las bandas Cb y Cr se submuestrean en una de estas 4 formas:

  1. **Formato 4:4:4.** Las tres componentes se muestrean con la misma frecuencia. Es la que debería aplicarse en aplicaciones *lossless*.
  2. **Formato 4:2:2.** Cada dos muestras de Y (sólo) en horizontal se toma una para Cb y otra para Cr.
  3. **Formato 4:1:1.** Cada cuatro muestras de Y en horizontal se toma una para Cb y otra para Cr.
  4. **Formato 4:2:0.** Cada dos muestras de Y en horizontal y en vertical se toma una para Cb y otra para Cr, pero en los puntos intermedios.

- Ejemplos:



27.3 Submuestreo de la crominancia

27.4. $[0, 255] \rightarrow [-128, 127]$

- Cada banda (Y, Cb y Cr en el caso de las imágenes en color o sólo Y en el caso de las imágenes en blanco y negro) se chequea para que sus puntos no sean siempre positivos. Si esto ocurre, se resta 128 a cada punto de esa banda.
- Esto se hace para decrementar la precisión artimética necesaria para calcular la DCT.

27.5. La 2D-DCT por bloques de 8×8 puntos

- Cada componente se procesa por bloques de 8×8 puntos y se le calcula la 2D-DCT (Discrete Cosine Transform). Esta transformada puede calcularse a partir de la DCT gracias a que la 2D-DCT es separable. La DCT se calcula como

$$\text{DCT}[u] = \frac{\sqrt{2}}{\sqrt{N}} K(u) \sum_{n=0}^{N-1} x[n] \cos \frac{(2n+1)\pi u}{2n} \tag{27.3}$$

y su transformada inversa como

$$x[n] = \frac{\sqrt{2}}{\sqrt{N}} \sum_{u=0}^{N-1} K(u)\text{DCT}[u] \cos \frac{(2n+1)\pi u}{2n} \tag{27.4}$$

donde N es el número de puntos a transformar y

$$K(u) = \begin{cases} \frac{1}{\sqrt{2}} & \text{si } u = 0 \\ 1 & \text{si } u > 0. \end{cases}$$

27.6. Características de la DCT

- **Es rápida:**
  Existe un algoritmo rápido de cálculo que se ejecuta en un tiempo proporcional a $N \log_2 N$.

- **Es casi óptima:**
  Es muy aproximada a la KLT (Karhunen-Lòeve Transform), una transformada óptima desde el punto de vista de su capacidad para acumular potencia espectral pero que tiene una complejidad de N^2. Como consecuencia, la DCT acumula en muy pocos coeficientes la mayor parte de la energía de la señal.

- **Es idempotente:**
  O lo que es lo mismo, la distancia euclídea es invariante en el dominio transformado. Esto permite decir que *aquellos coeficientes más grandes en valor absoluto son los que más minimizan el error cuadrático medio (MSE) de la reconstrucción*.

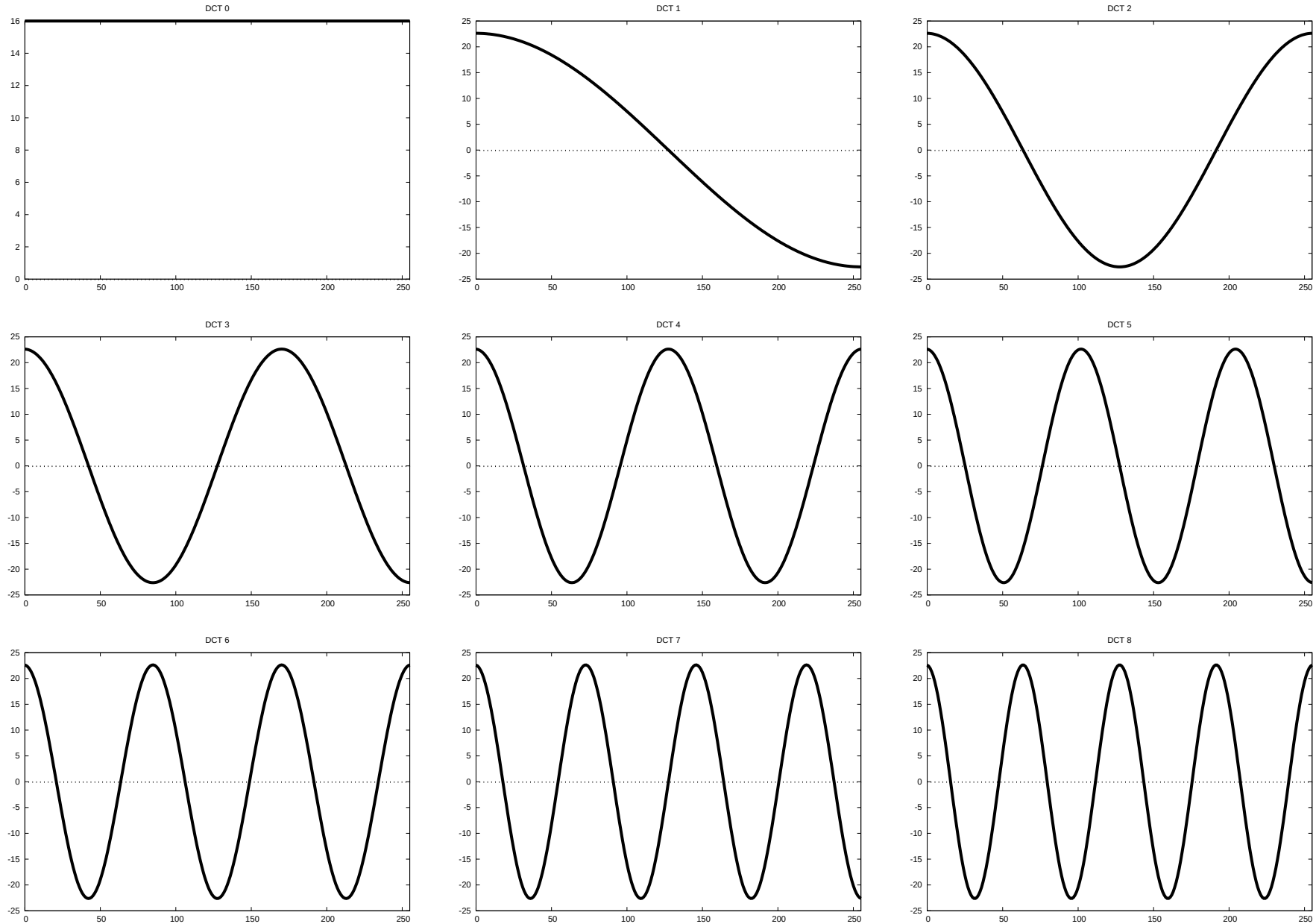
27.7. Ventajas de la 2D-DCT por bloques

- **Un menor coste computacional total:**
  Es más rápido calcular $(\frac{N}{8} \times \frac{N}{8})$ 2D-DCT's de 8×8 puntos que una 2D-DCT de $N \times N$ puntos ya que incluso la complejidad del algoritmo rápido es superior a la lineal.

- **Posibilidad de operación "in-line":**
  Este es un factor determinante cuando las imágenes son muy grandes. El compresor y el descompresor pueden trabajar por bloques de 8×8 puntos independientemente del tamaño de la imagen.

27.8. Funciones base DCT

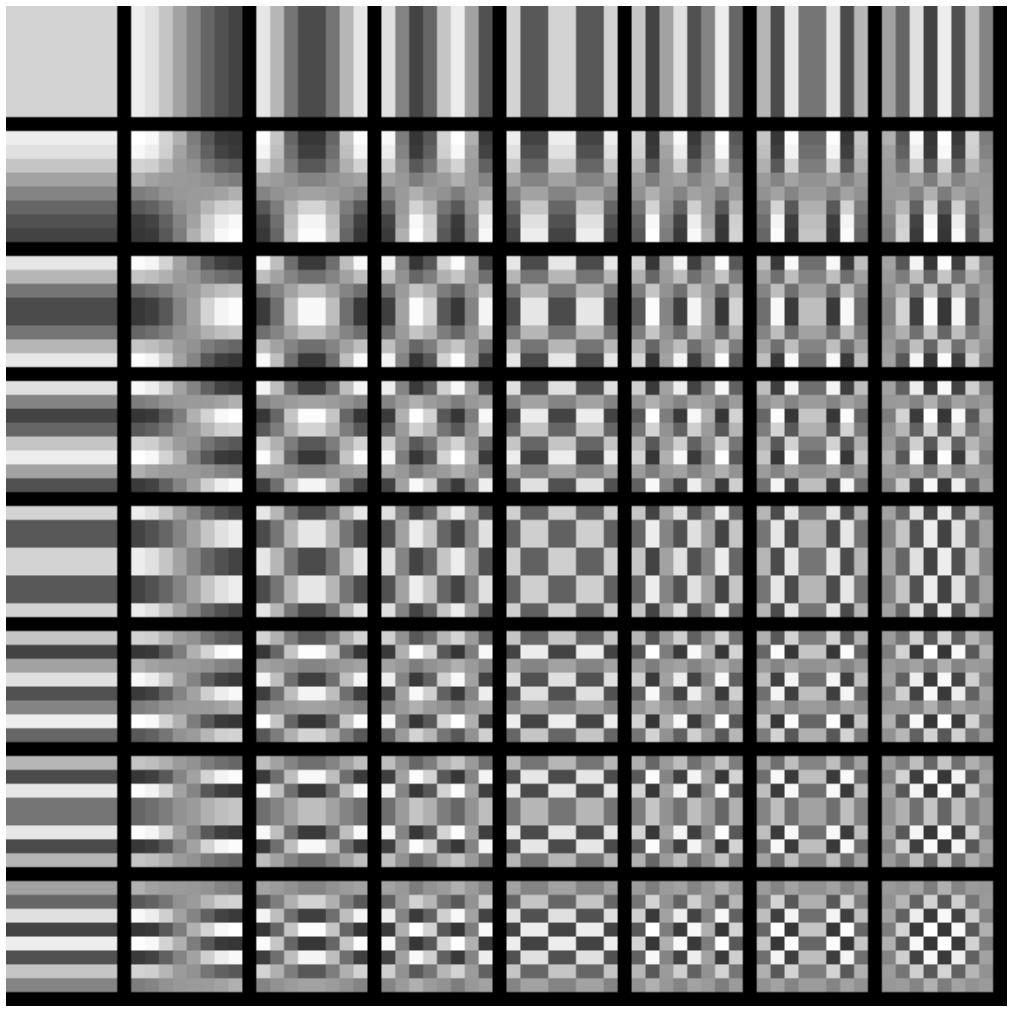
- Es interesante conocer la forma de las funciones base de la DCT porque así podemos hacernos una idea de la correlación espacial explotada y del aspecto de las reconstrucciones.
- Las primeras 9 funciones base de la DCT son:



27.8 Funciones base DCT

27.9. Funciones base 2D-DCT de 8×8 puntos

- Cada coeficiente de la 2D-DCT de un bloque de 8×8 puntos representa el peso que tienen cada uno de estos 64 patrones a la reconstrucción del bloque.



27.10. Cuantificación escalar

- El principal objetivo de la 2D-DCT es la compactación espectral. Cuando esto se consigue, muy pocos coeficientes acumulan la mayor parte de la energía.
- Como los coeficientes están muy descorrelacionados se utiliza cuantificación escalar, en lugar de cuantificación vectorial.
- El resultado de la cuantificación es un gran número de coeficientes cuantificados muy próximos o iguales a 0. Su distribución estadística es una Laplaciana.

- La fase de cuantificación es muy importante porque es la principal fuente de pérdida de información. El JPEG estudió el proceso y propuso una matriz de cuantificación para la luminancia y otra para la crominancia.

Luminancia

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

Cromiancia

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

- Como puede verse, se da más peso a las bajas frecuencias que a las altas*.

*Aunque en el caso de la luminancia el coeficiente DC no es el más importante.

- Aritméticamente, el proceso de cuantificación consiste en dividir el valor de los coeficientes 2D-DCT entre los coeficientes de cuantificación y redondear al entero más cercano, es decir

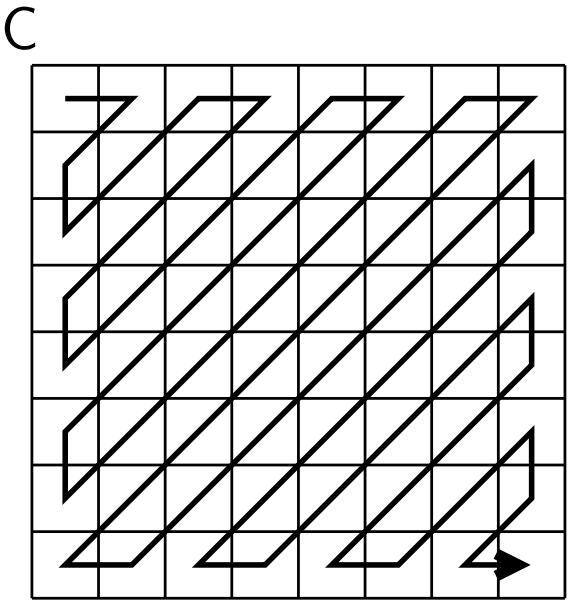
$$\text{2D-DCT}'[u,v] = \text{round}\left(\frac{\text{2D-DCT}[u,v]}{\text{Z}[u,v]}\right) \tag{27.5}$$

  donde $\text{Z}[\cdot,\cdot]$ es la matriz de cuantificación.

- El estándar prevee el uso de matrices de cuantificación diseñadas a medida, pero en este caso el compresor debe indicárselas al descompresor.

27.11. Codificación entrópica

- JPEG utiliza una mezcla entre DPCM, RLE y Huffman para codificar los coeficientes cuantificados.
- Algoritmo:
  1. Restar al coeficiente DC el valor del coeficiente DC del bloque anterior (DPCM). Objetivo: decrementar el rango dinámico del coeficiente DC.
  2. Recorrer los coeficientes en zig-zag tal y como se muestra a continuación:



para encontrar el punto a partir del cual todos los coeficientes son 0. Este evento se señaliza con un símbolo especial llamado EOB (End Of Block).

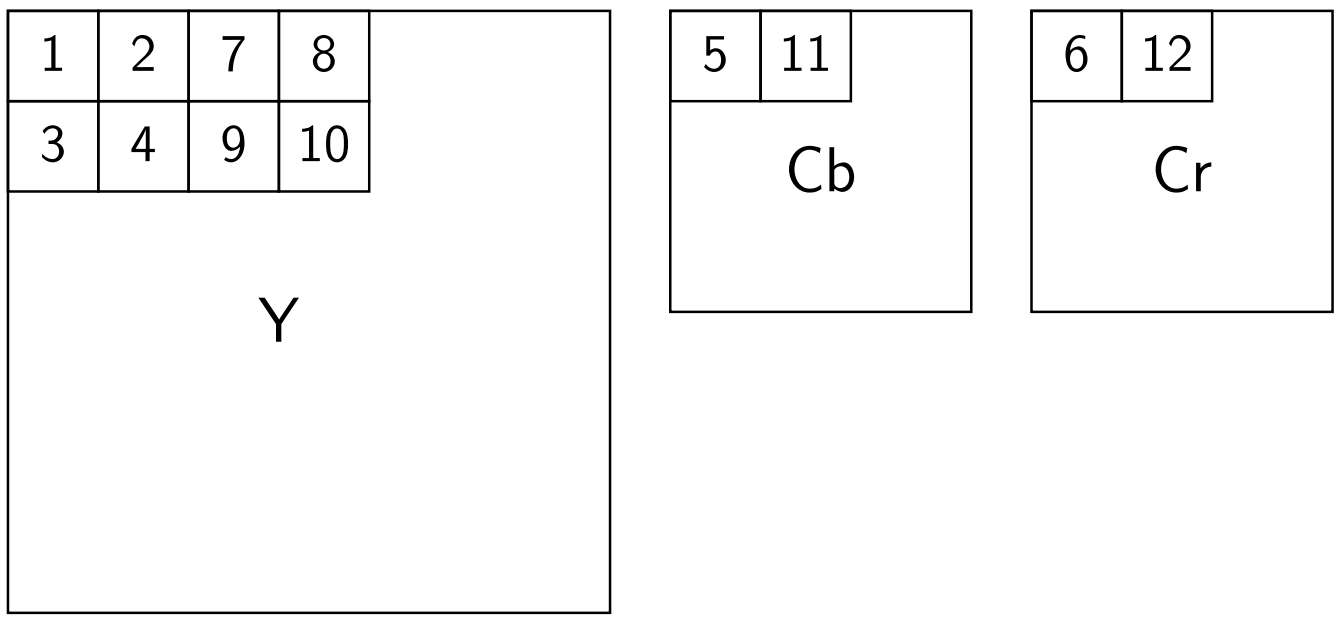
3. Codificar las series de coeficientes nulos (RLE) usando un código de longitud variable (Huffman) estático.**

**El JPEG propuso también el uso de una codificación aritmética pero debido al aumento de la carga computacional de esta etapa y a la ganancia marginal en compresión (aproximadamente un 10 % en promedio) rara vez se utiliza.

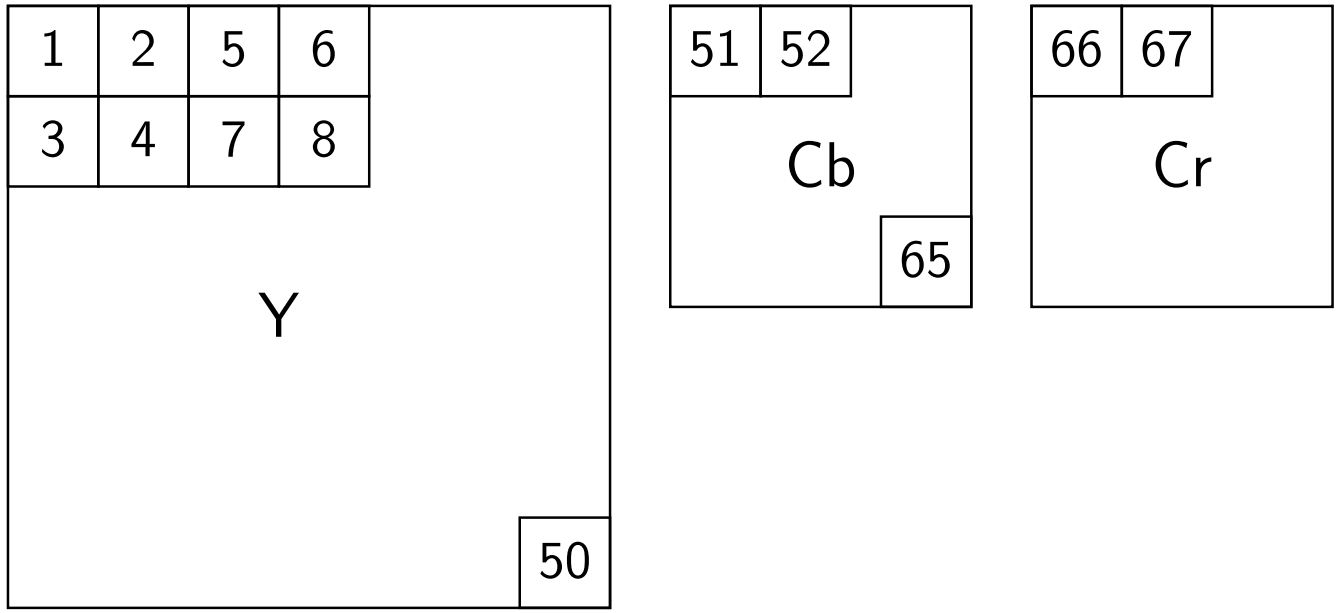
27.12. Entrelazamiento de las componentes de color

- Típicamente, JPEG utiliza un submuestreo 4:2:0 (véase la Sección 27.3).
- El entrelazamiento se utiliza para poder reconstruir una imagen en color por filas, conforme vamos leyendo el fichero comprimido.

1. Con entrelazamiento:

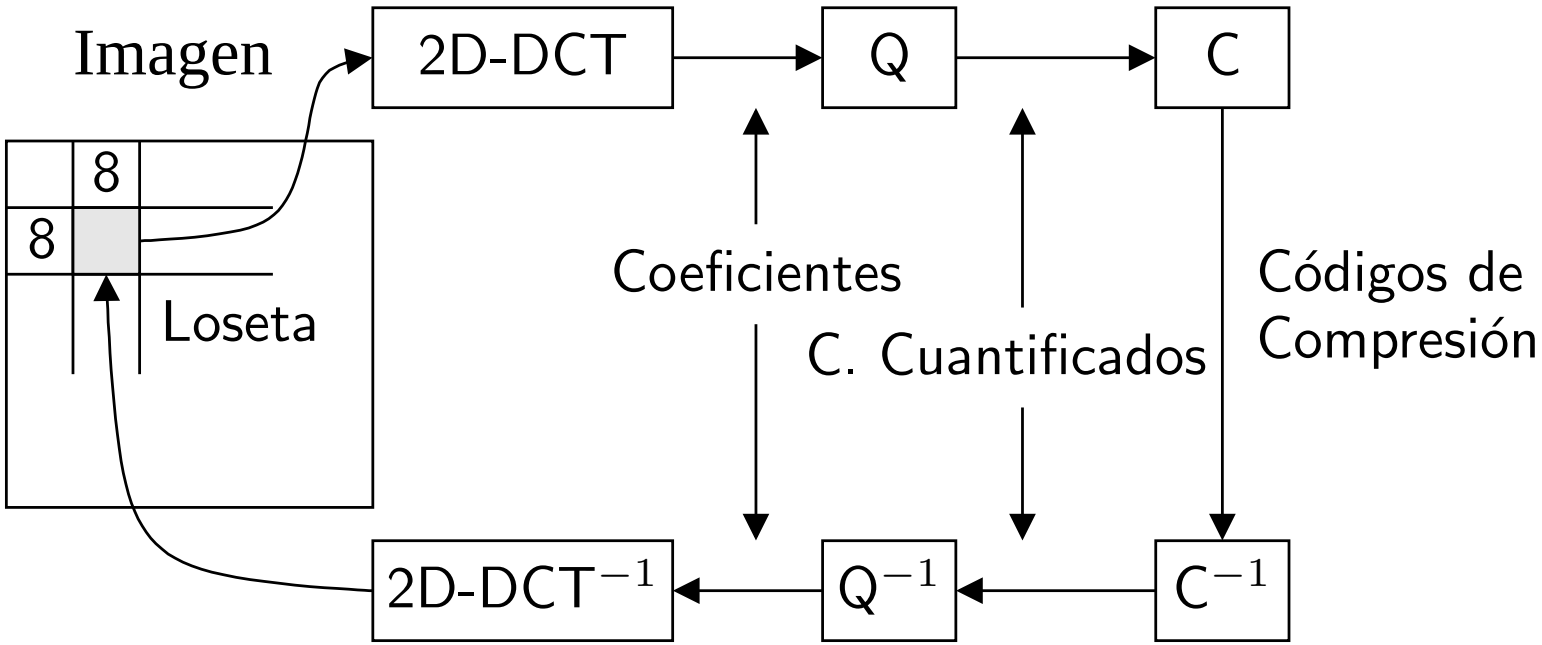


2. Sin entrelazamiento:



27.13. Ejemplo de compresión

- Vamos a realizar un ejemplo del proceso descrito



utilizando un ejemplo (una imagen en tonos de gris).

Cálculo de la 2D-DCT a un bloque

- El resultado de aplicar la 2D-DCT a un bloque de luminancia de la imagen "lena" es:

79	75	79	82	82	86	94	94
76	78	76	82	83	86	85	94
72	75	67	78	80	78	74	82
74	76	75	75	86	80	81	79
73	70	75	67	78	78	79	85
69	63	68	69	75	78	82	80
76	76	71	71	67	79	80	83
72	77	78	69	75	75	78	78

$\Leftrightarrow$

619	-29	8	2	1	-3	0	1
22	-6	-4	0	7	0	-2	-3
11	0	5	-4	-3	4	0	-3
2	-10	5	0	0	7	3	2
6	2	-1	-1	-2	0	0	8
1	2	1	2	0	2	-2	-2
-8	-2	-4	1	2	1	-1	1
-3	1	5	-2	1	-1	1	-3

- Nótese que la mayor parte de la energía se concentra en y a los arrededores del coeficiente DC.

Cuantificación

619	-29	8	2	1	-3	0	1
22	-6	-4	0	7	0	-2	-3
11	0	5	-4	-3	4	0	-3
2	-10	5	0	0	7	3	2
6	2	-1	-1	-2	0	0	8
1	2	1	2	0	2	-2	-2
-8	-2	-4	1	2	1	-1	1
-3	1	5	-2	1	-1	1	-3

div

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

=

39	-3	1	0	0	0	0	0
2	-1	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

- Nótese que la mayor párte de la energía de alta frecuencia se ha perdido.

Generación del EOB

- La matriz

39	-3	1	0	0	0	0	0
2	-1	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

  se recorre en zig-zag y se calcula el EOB.

- El resultado es

39 -3 2 1 -1 1 0 0 0 0 0 -1 EOB

27.14. Codificación entrópica de las series

- **Codificación del coeficiente DC.** Restar el coeficiente DC del bloque anterior para eliminar la componente continua. En nuestro ejemplo: $39 - 34 = 5$. Esta diferencia se codifica como en LS-JPEG (véase la Tabla Categorias de la Sección 25.3). El resultado es: 100101.

- **Codificación de los coeficientes AC.** La codificación depende del número de coeficientes AC nulos que los preceden (los coeficientes nulos no se codifican). Por ejemplo, el primer coeficiente AC es -3 y no lo antecede ningún 0. Por ser -3 sabemos (mirando en la Tabla Categorías de la Sección 25.3) que la categoría asignada es la 2. Miramos ahora en la Tabla Códigos AC propuesta por JPEG (para la luminancia) y vemos que los dos primeros bits de este código son 01. Los otros dos bits se calculan de la misma forma que para LS-JPEG. La tabla de códigos AC es muy extensa. Aquí sólo se indica la sección inicial que por suerte contiene la entrada "serie 0/categoría 2".

Códigos AC para JPEG (Luminancia)

Serie/Categoría	Longitud	Código base
0/0	4	1010 (=EOB)
0/1	3	00
0/2	4	01
0/3	6	100
:	:	:
15/10	26	1111 1111 1111 11110

El bit-stream completo generado para la loseta de nuestro ejemplo es:

100101 0100 0110 001 000 001 11110100 1010

Total 35 bits. Esto representa una tasa de compresión de $0{,}55$ bpp o de aproximadamente $15:1$.

27.15. Transmisión progresiva

- Se utiliza cuando se transmiten imágenes JPEG sobre enlaces lentos.
- Existen 3 versiones:
  1. **Selección espectral progresiva:**
     - Consiste en entrelazar los coeficientes de los distintos bloques en función de su frecuencia, transmitiendo primero los de menor frecuencia.
     - Proporciona a lo sumo 64 scans (pasadas).
  2. **Selección por planos de bits:**
     - Consiste en entrelazar los planos de bits de los coeficientes de los distintos bloques, transmitiendo primero los de mayor peso.
     - Proporciona a lo sumo 11 scans.
  3. **Una mezcla de las dos anteriores:**

- 64 u 11 pasadas pueden ser insuficientes en aquellos casos donde el tiempo de transmisión sea muy largo y la potencia computacional del dispositivo de visualización sea alta. Los coeficientes pueden transmitirse por planos de bits, pero seleccionándolos además por bandas de frecuencia (hasta 704 scans).

27.16. El algoritmo jerárquico

- Se basa en construir una pirámide diferencial y aplicar a cada imagen de la pirámide uno de los tres métodos anteriores.
- Algoritmo de construcción de la pirámide diferencial consiste en:
  1. Submuestrear (filtrando previamente) la imagen en un factor de 2 en cada dimensión.
  2. Interpolar la imagen submuestreada en un factor de 2 en cada dimensión.
  3. Restar ambas imágenes, obteniendo la base de la pirámide (altas frecuencias). Nótese que sumando la imagen diferencia y la imagen submuestreada (e interpolada) podemos recuperar la imagen original.
  4. Repetir el proceso con la imagen submuestreada tantas veces como se desee (o sea posible).

27.17. Calidad de JPEG vs factor de compresión

- A continuación mostramos una comparativa visual de la compresión de la imagen "lena" en color (512×512 puntos, 24 bits/punto) mediante el algoritmo JPEG (lossy) no progresivo.
- Como medida objetiva de la distorsión generada se va a utilizar el PSNR (véase el Apéndice 35).

Lena 512×512 RGB original

Lena a 1,0 bpp (32,85 dB)

Lena a 0,5 bpp (30,91 dB)

Lena a 0,4 bpp (30,09 dB)

Lena a 0,3 bpp (28,97 dB)

Lena a 0,2 bpp (26,64 dB)

Lena a 0,1 bpp (21,29 dB)

27.18. Compresión de "akiyo"

- JPEG no puede comprimir secuencias de imágenes, pero puede ser llamado iterativamente para generar un resultado semejante. Al igual que con PNG, comprimiremos la secuencia de imágenes PPM {`akiyo000`, `akiyo001`, ..., `akiyo299`}, a un $75\,\%$ de calidad, con el script:

```
for i in akiyo???
do
  pnmtojpeg -quality=75 $i > $i.jpg
done
```

- La tasa promedio de compresión (cabeceras incluidas) ha sido: $25{,}12$ $(91.238.400/3.631.831)$.

- Este método de compresión (también conocido por M-JPEG (Motion-JPEG)) posee una latencia de una imagen (lo que es ideal para aplicaciones interactivas) y permite la escalabilidad temporal.

- En el Apéndice 39.56 se presenta un fichero `Makefile` que muestra

cómo invocar al programa *FFMPEG* (`ffmpeg.mplayerhq.hu`) para generar una secuencia de vídeo en formato M-JPEG.

- Acceda al directorio `vids` que encontrará a la misma altura que este documento en el sistema de ficheros y visualice los vídeos `*MJPEG*`.
- En Linux se recomienda usar el programa *MPlayer* (`www.mplayerhq.hu`). Para lanzarlo ejecutar, por ejemplo:

```
mplayer -loop 0 -fs archivo_video
```

- Bajo Windows se recomienda usar el programa *VLC media player* (`www.videolan.org`).

Capítulo 28

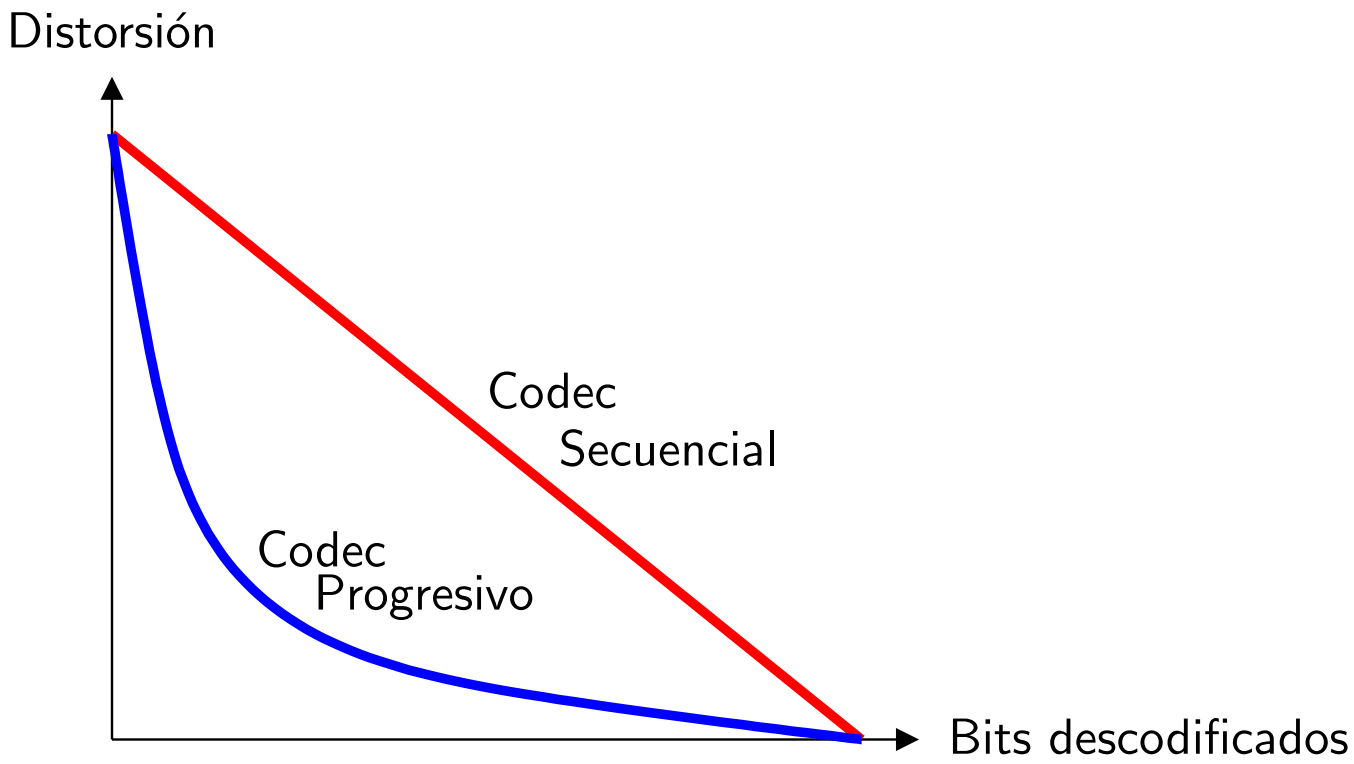
El estándar JPEG 2000 (ISO/IEC 15444-1) [20]

Particularidades del JPEG2000

- **Lossy y lossless:** El estándar permite usar un camino reversible o un camino irreversible, en función de las necesidades del usuario. La ventaja del irreversible (lossy) es que los PSNR's son superiores a tasas de bits bajas.
- **Escalabilidad en distorsión:** El descompresor selecciona la calidad de la imagen que se descomprime, hasta llegar al caso lossless.
- **Escalabilidad espacial:** El descompresor selecciona el nivel de resolución y el área a descomprimir (ROI).
- **Gran variedad de progresiones:** el compresor ordena los paquetes para determinar el tipo de escalabilidad al descomprimir secuencialmente la imagen.
- **Tolerancia a errores:** la descompresión de cada paquete no influye en la de los demás.

28.1. ¿Qué es la codificación progresiva?

- Además, debido a la gran cantidad de datos que se generan, la transmisión de las imágenes, incluso en un formato comprimido, necesitan un cierto tiempo. Cuando es posible reconstruir la imagen en el receptor con un sub-conjunto del stream de datos comprimidos, hablaremos de codificación progresiva de imágenes.



28.2. La transformada wavelet discreta (diádica)

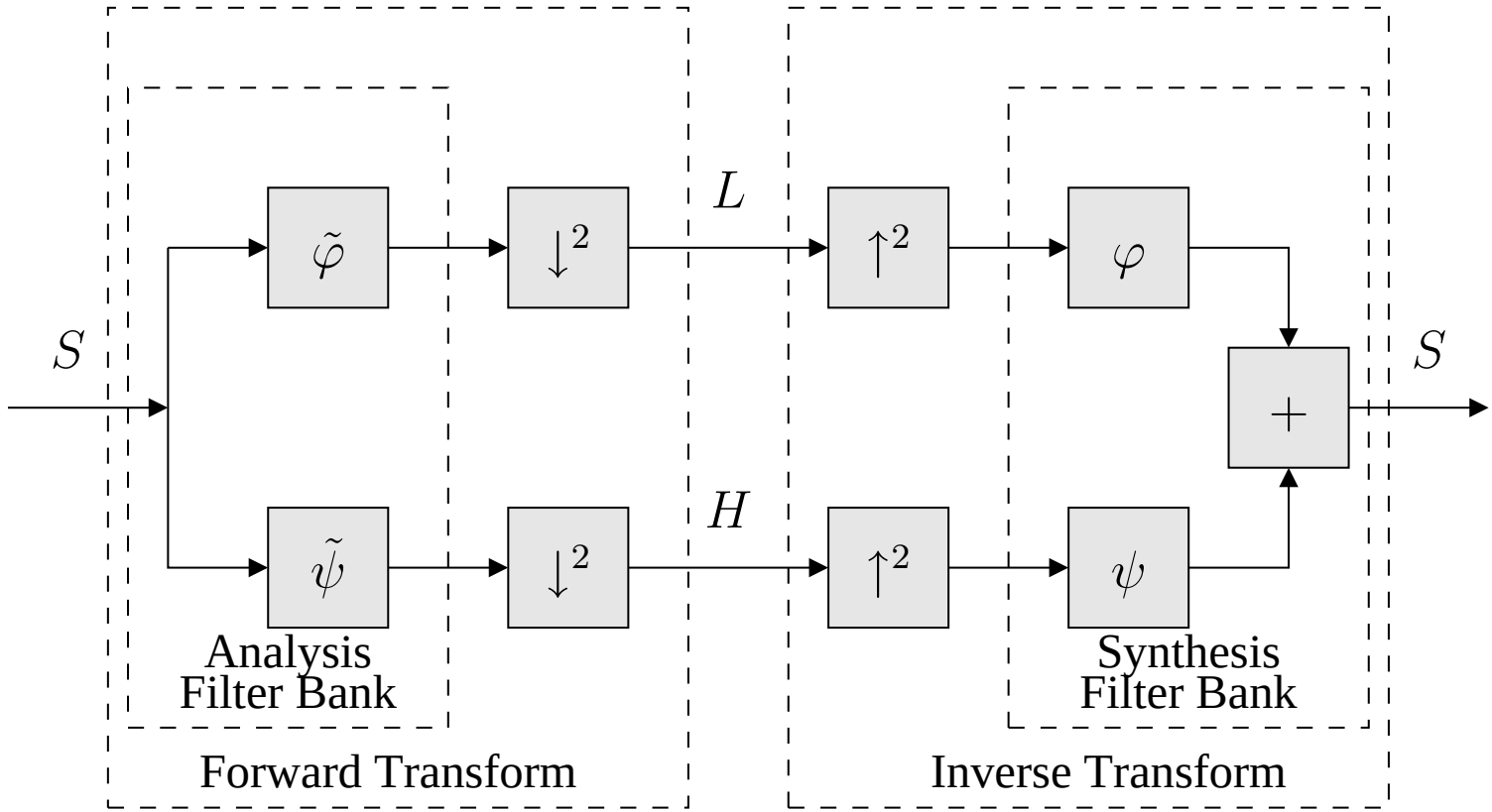
- La DWT (Discrete Wavelet Transform) es una transformada que se utiliza en la mayoría de los sistemas de codificación progresivos que existen actualmente.
- Permite:
  1. Concentrar en unos pocos coeficientes wavelet la mayor parte de la energía de la señal (generalmente mejor que la DCT).
  2. Encontrar una representación multiresolución de la imagen, típicamente por potencias de dos.
- Gracias a esto, mediante la DWT podemos reconstruir progresivamente una imagen maximizando la cantidad de energía transmitida (enviados los coeficientes más grandes, primero).

28.3. La 1D-DWT (algoritmo intuitivo)

- La 1D-DWT utiliza la correlación espacial para poder encontrar una representación lo más compacta posible.
- El algoritmo básicamente consiste en*:
  1. Encontrar una versión submuestreada de la señal.
  2. Realizar una predicción con ella.
  3. Restar la predicción a la señal.
  4. Almacenar las diferencias (que se pueden submuestrear).
- En otras palabras, en cada iteración descomponemos una señal S en dos bandas. Una de baja frecuencia a la que denotaremos por L y otra de alta frecuencia a la que denotaremos H. En H está lo que le hace falta a L para ser S.

*Aquí se expone sólo una iteración

28.4. La 1D-DWT (cálculo mediante bancos de filtros)



Siendo (fase de síntesis)

$$S = (\uparrow^2 (L) * \varphi) + (\uparrow^2 (H) * \psi)$$

donde (fase de análisis)

$$\begin{aligned} L &= \downarrow^2 (S * \tilde{\varphi}) \\ H &= \downarrow^2 (S * \tilde{\psi}). \end{aligned}$$

- $\tilde{\varphi}$ y $\tilde{\psi}$ son filtros (de análisis) espejo (sus funciones de transferencia son simétricas), lo que implica que L y H contienen la misma información que S. φ y ψ son los correspondientes filtros de síntesis, también simétricos entre sí.** En este caso se habla de un PR (Perfect Reconstruction) QMF (Quadrature Mirror Filter) bank. Cuando esto ocurre se dice que la DWT es biorthogonal.

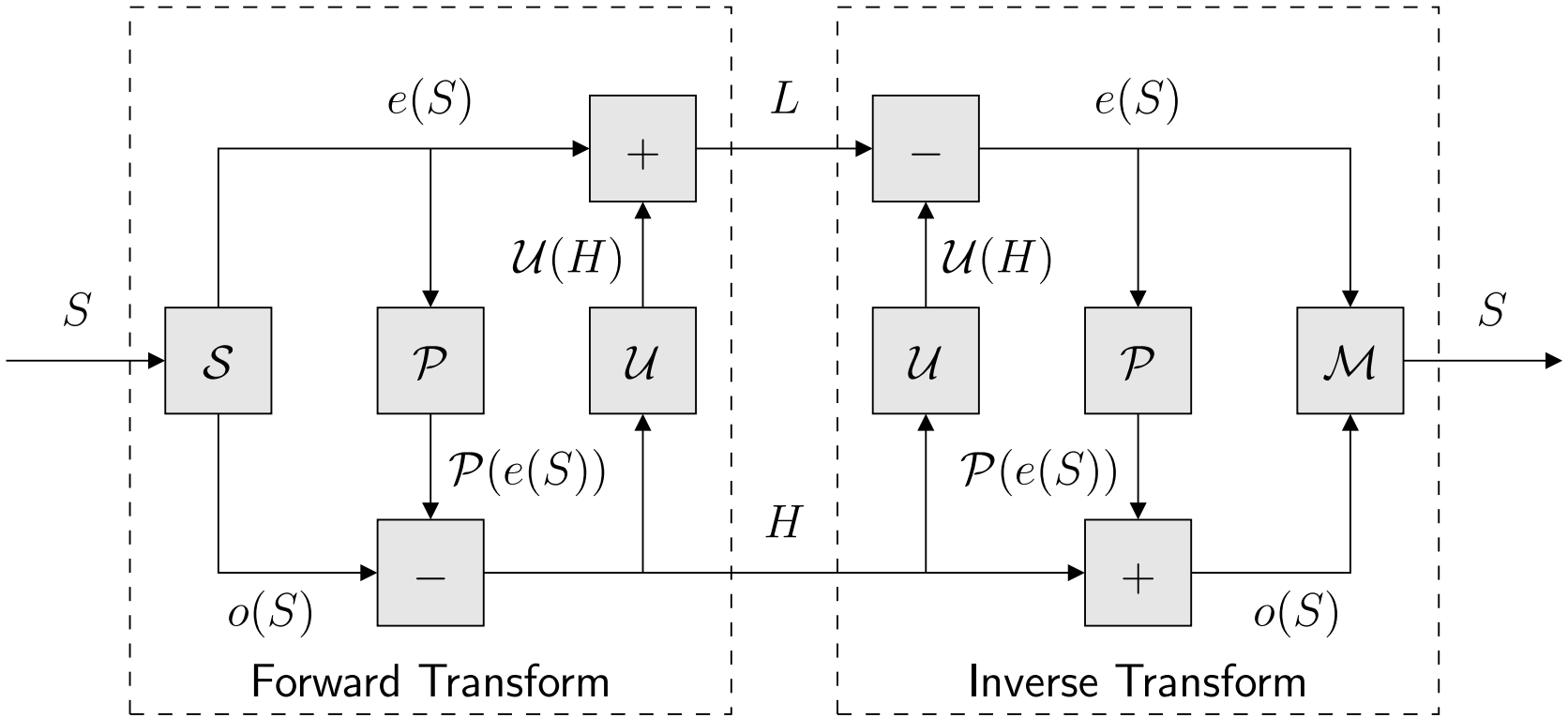
- En la Teoría Wavelet, a φ suele llamarse función escala (*scale function*) y a $\tilde{\psi}$ función madre (*mother function*) o wavelet.

- $\downarrow^2$ y $\uparrow^2$ denotan a las operaciones de submuestreo y sobremuestreo, en un factor de 2, respectivamente. En el Apéndice 36 puede encontrarse una definición más formal de estos operadores.

**Desde un punto de vista matemático, las funciones $\varphi(t)$ y $\psi(t)$ sería ortogonales entre sí y ortogonales con respecto a las funciones duales $\tilde{\varphi}(t)$ y $\tilde{\psi}(t)$.

- Finalmente, el operador $*$ denota a la convolución entre funciones (o señales).

28.5. La 1D-DWT (cálculo basado en Lifting [19])



With lifting, the forward 1-level dyadic DWT of S, $\mathcal{A}(\mathrm{S}) = \{L, H\}$, is calculated in three stages:

1. *Split ($\mathcal{S}$):* Decomposes the input sequence S into two sequences

$e(S) = \{S_{2i}\}$ (the even samples of S) and $o(S) = \{S_{2i+1}\}$ (the odd samples of S).

2. *Predict ($\mathcal{P}$):* Using one of these sequences (tipically $e(S)$), this stage generates a prediction for the other ($o(S)$). Lets $\mathcal{P}(e(S))$ the prediction sequence of samples. After that, the high-frequency subband H is calculated substracting both sequences:

$$H_i = S_{2i+1} - \mathcal{P}(e(S))_i. \tag{28.1}$$

Como consecuencia de que S generalmente es el resultado de muestrear una función suave y contínua, es fácil encontrar una predicción precisa (*accurate prediction*) para las muestras impares a partir de las pares usando un filtro de interpolación. Si la predicción es adecuada, esta operación reduce la entropía de la señal, es decir, necesitaremos menos bits de información para representar H que para representar $o(S)$. Como ambas señales tienen la misma longitud, esto permite realizar una compresión lossless de la señal más eficiente. Además, la energía de H tiende a ser menor que la energía de $o(S)$. Por tanto, para encontrar una representación aproximada de la señal

S, es más eficiente utilizar $\{e(S), \mathcal{Q}_{\mathcal{N}}(\mathcal{H})\}$ que $\{e(S), \mathcal{Q}_N(o(S))\}$, donde $\mathcal{Q}_{\mathcal{N}}(\cdot)$ es el scalar quantization operator.[***]

3. *Update ($\mathcal{U}$):* Finalmente, una combinación lineal de los errores de predicción alrrededor en la muestra i, $\mathcal{U}(H)_i$, es sumada a cada S_{2i} para reducir el aliasing provocado en la primera etapa ($\mathcal{S}$):

$$L_i = S_{2i} + \{\mathcal{U}(H)\}_i. \tag{28.2}$$

   De esta manera, si la cuantificación aplicada a la banda H es tan severa que no se utiliza ninguna cantidad de información de la misma en una reconstrucción parcial, dicha reconstrucción al menos no poseerá los efectos provocados por el aliasing (generalmente desagradables a la vista y al oído) que ocurren cuando usamos una downsampled signal que no ha sido convenientemente filtrada.

The inverse transform, $\mathcal{S}(\{L, H\}) = S$, is also computed in three stages. Basically, these steps undo the operations that have been performed at the analisys stage:

[***] See Appendix 38 to get a formal definition of this operator.

1. *Undo Update:* Given L and H we can recover the even samples by substracting the update information:

$$S_{2i} = L_i - \{\mathcal{U}(H)\}_i. \tag{28.3}$$

2. *Undo Predict:* We can reconstruct the odd samples by adding the errors and the prediction information:

$$S_{2i+1} = H_i + \{\mathcal{P}(e(S))\}_i. \tag{28.4}$$

3. *Merge ($\mathcal{M}$):* Now that we have the even and odd samples we simply have to zip them together to recover the original signal.

28.6. La N-levels 1D-DWT

La aplicación recursiva ς iteraciones de la 1-level DWT aplicada a la banda L genera lo que se conoce como ς-level dyadic DWT, ς-level octave-band decomposition or ς-level pyramid decomposition of S [5]. Esta transformada $\mathcal{A}^{\varsigma}(S) = \{L^{\varsigma-1}, \cup H^{s}\}$, donde $s = 0, 1, \cdots, \varsigma - 1$. descompone S en $\varsigma + 1$ bandas de frecuencia. Dicha recursión queda definida por

$$\begin{aligned} \{L^{s}, H^{s}\} &= \mathcal{A}(S^{s}) \\ S^{s+1} &= L^{s}. \end{aligned} \tag{28.5}$$

De (28.5) se desprende que s representa el nivel de resolución de S y que

$$S = L^{-1}. \tag{28.6}$$

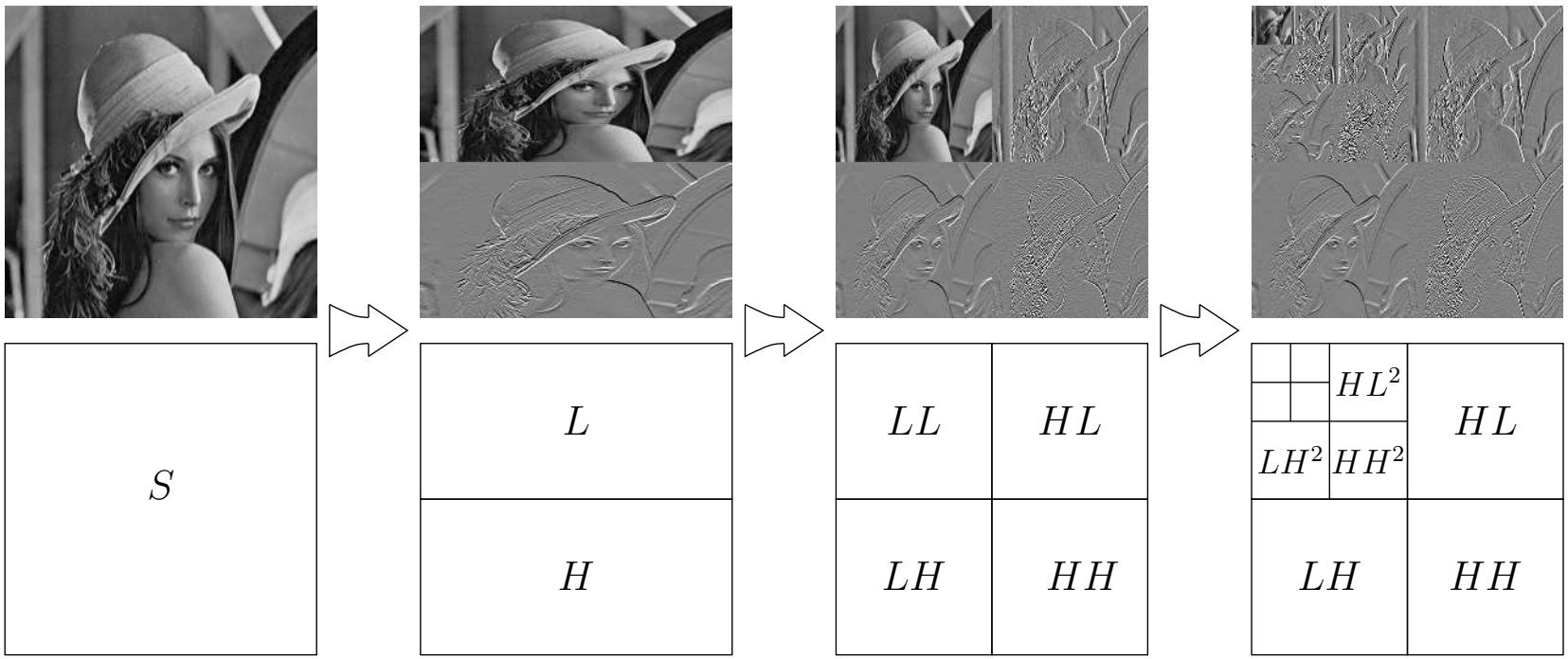
La fase de síntesis (transformada inversa) de S, $\mathcal{S}^{\varsigma}(S)$, reconstruye la señal original a partir de sus bandas de frecuencia. Por tanto se tiene que $\mathcal{S}^{\varsigma}(\{L^{\varsigma-1}, \cup H^{s}\}) = S$, donde

$$\begin{aligned} L^{s} &= S^{s+1} \\ S^{s} &= \mathcal{S}(\{L^{s}, H^{s}\}). \end{aligned} \tag{28.7}$$

En el Apéndice 39.8 se muestra una implementación de la transformada wavelet discreta en una dimensión.

28.7. La 2D-DWT

- La DWT 1-dimensional es separable y por tanto, el caso bidimensional puede calcularse aplicando la transformada 1D sobre las filas y a continuación sobre las columnas (o viceversa).



- Como puede apreciarse, aún sobre el dominio transformado podemos localizar espacialmente a los coeficientes.

- En el Apéndice 39.11 se muestra una implementación de la Transformada Wavelet Discreta en 2 dimensiones.

28.8. La Transformada de Haar (2/2) [9]

- Presupone que las señales son funciones constantes.
- La banda L se calcula como (banco de filtros)

$$\{L_i\} = \left\{\frac{S_{2i} + S_{2i+1}}{2}\right\} \tag{28.8}$$

y la banda H (*prediction step*, usando Lifting) como

$$\{H_i\} = \{S_{2i+1} - S_{2i}\}. \tag{28.9}$$

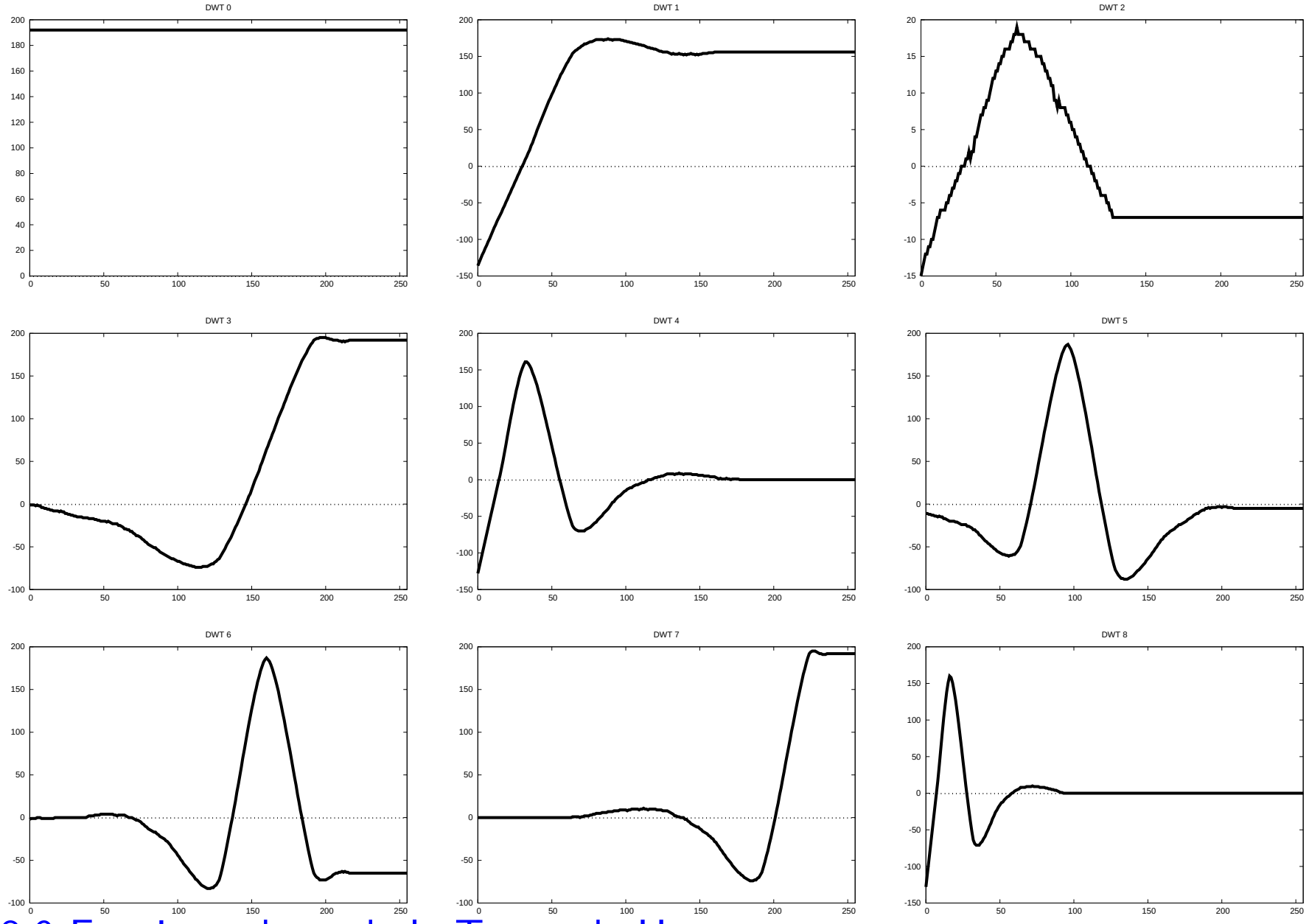
Si usamos Lifting (*update step*),

$$\{L_i\} = \left\{S_{2i} + \frac{H_i}{2}\right\}. \tag{28.10}$$

Como es posible observar, los coeficientes transformados serían todos cero si la señal fuera constante.

- En el Apéndice 39.15 se muestra una implementación de la Transformada de Haar.

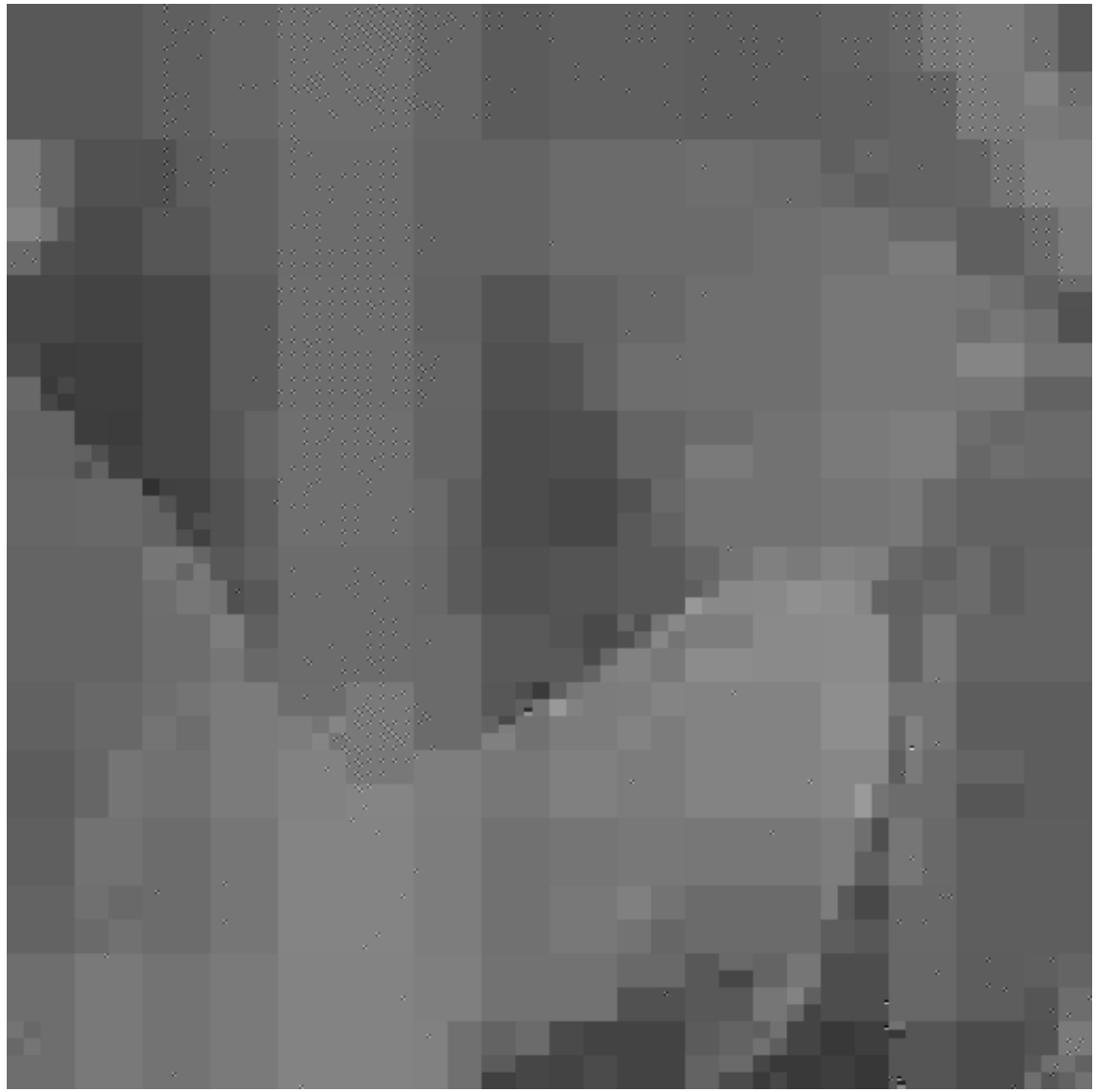
28.9. Funciones base de la Trans. de Haar



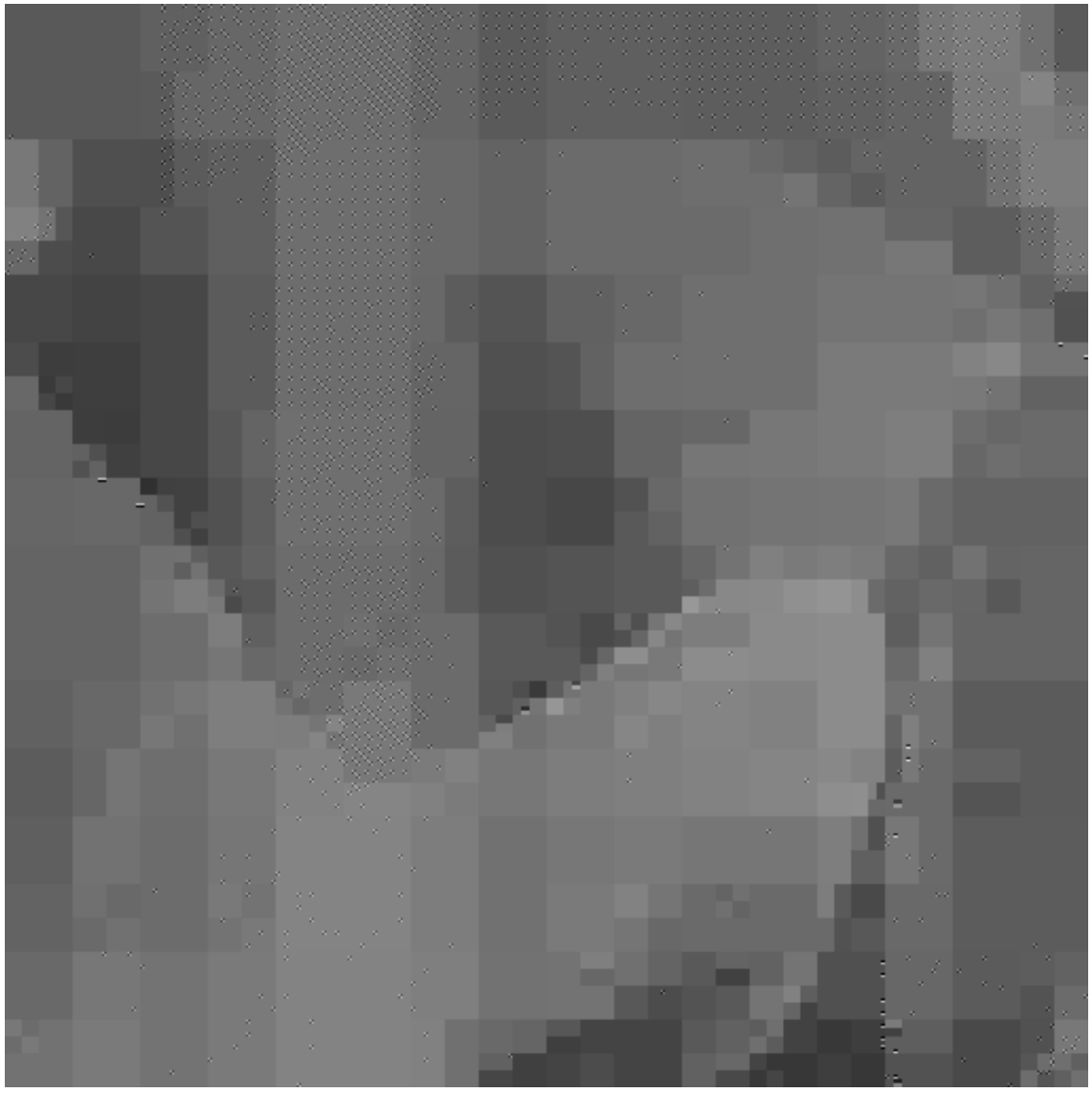
28.10. Transmisión progresiva de "lena" usando la Transformada de Haar

1. Realizamos la transformada directa a la imagen.
2. Eliminamos un determinado número de coeficientes, los de menos peso.
3. Realizamos la transformada inversa.

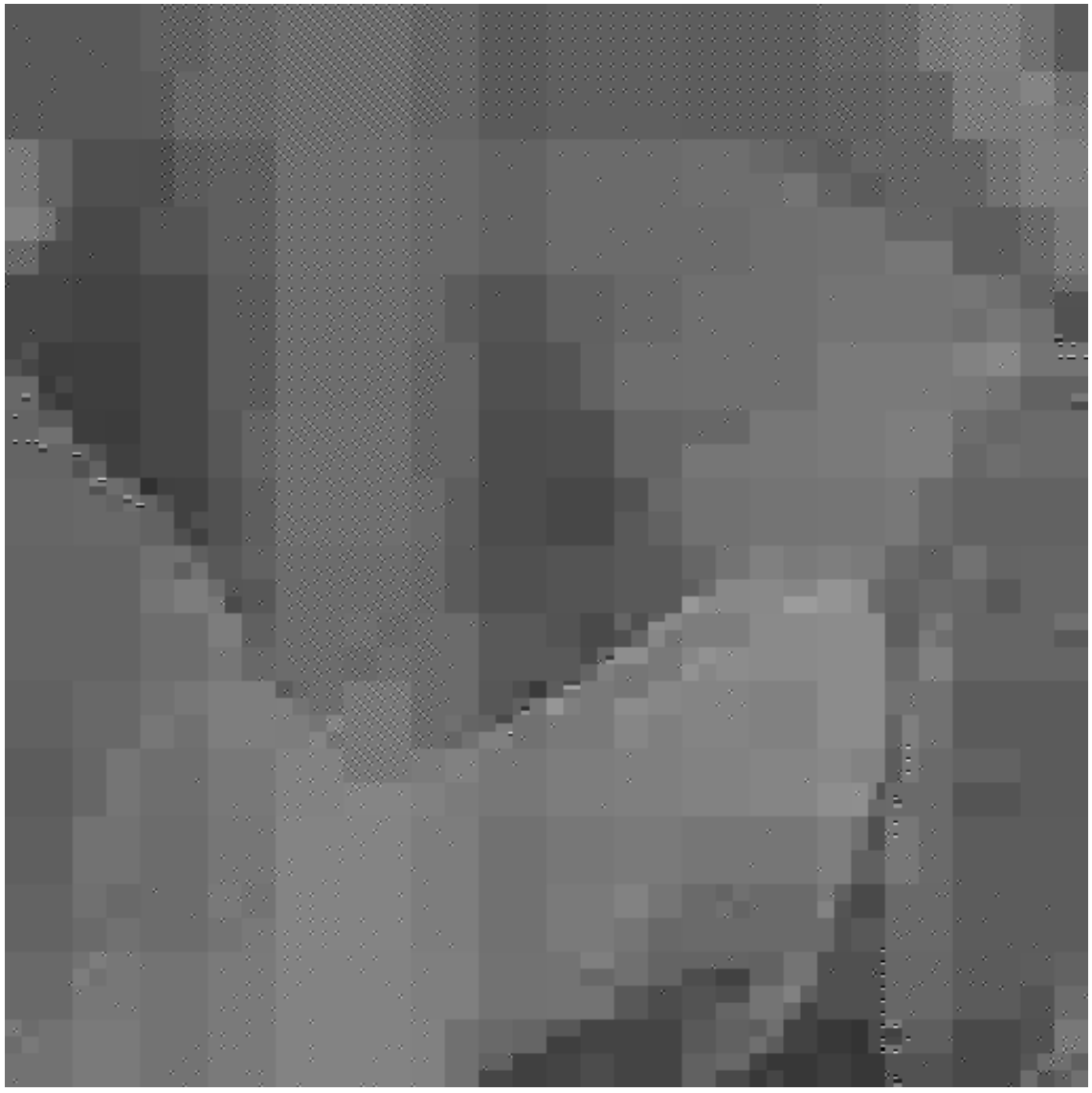
Reconstrucción de "lena". 1.000 coefs



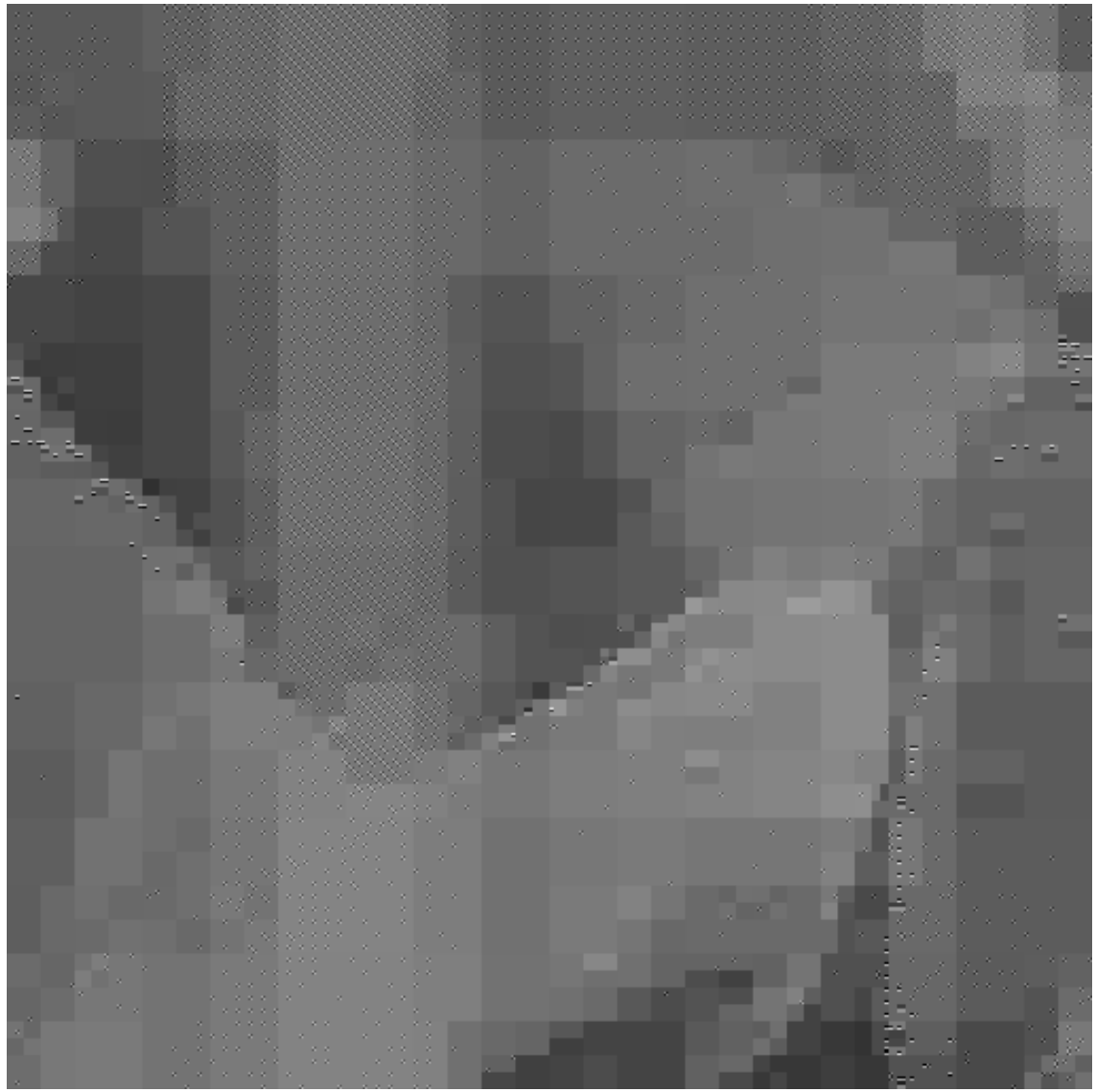
Reconstrucción de "lena". 2.000 coefs



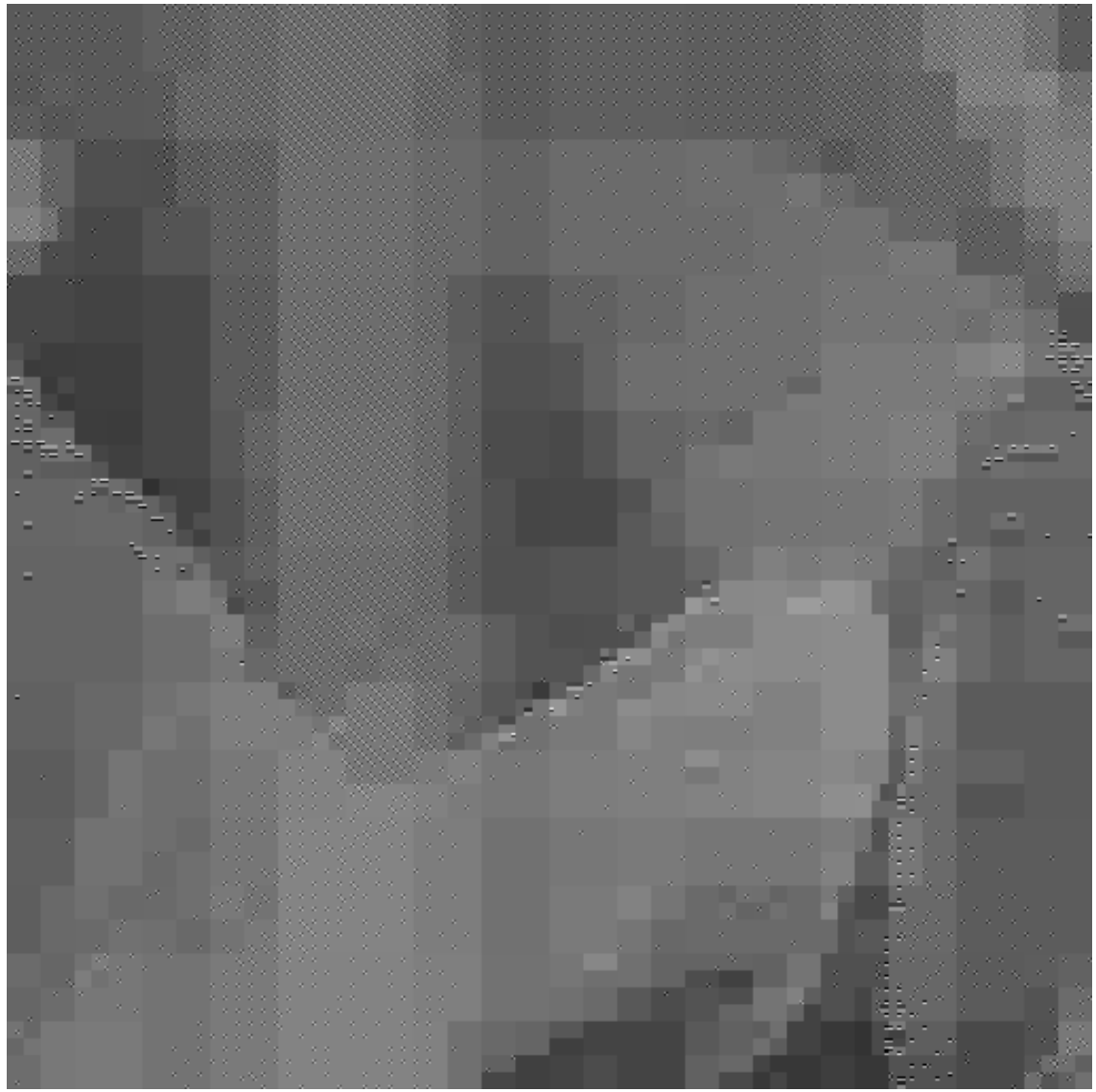
Reconstrucción de "lena". 3.000 coefs



Reconstrucción de "lena". 4.000 coefs



Reconstrucción de "lena". 5.000 coefs



28.11. La Transformada Lineal (Spline 5/3)

- Presupone que las señales están compuestas de líneas rectas [2, 19].
- La banda L se calcula (banco de filtros) como

$$\{L[i]\} = \{-\frac{1}{8}S_{2i-2} + \frac{1}{4}S_{2i-1} + \frac{3}{4}S_{2i} + \frac{1}{4}S_{2i+1} - \frac{1}{8}S_{2i+2}\} \quad (28.11)$$

y la banda H (*prediction step*) como

$$\{H_i\} = \left\{S_{2i+1} - \frac{S_{2i} + S_{2i+2}}{2}\right\}. \quad (28.12)$$

En el caso de usar Lifting, la subbanda de baja frecuencia puede también calcularse como

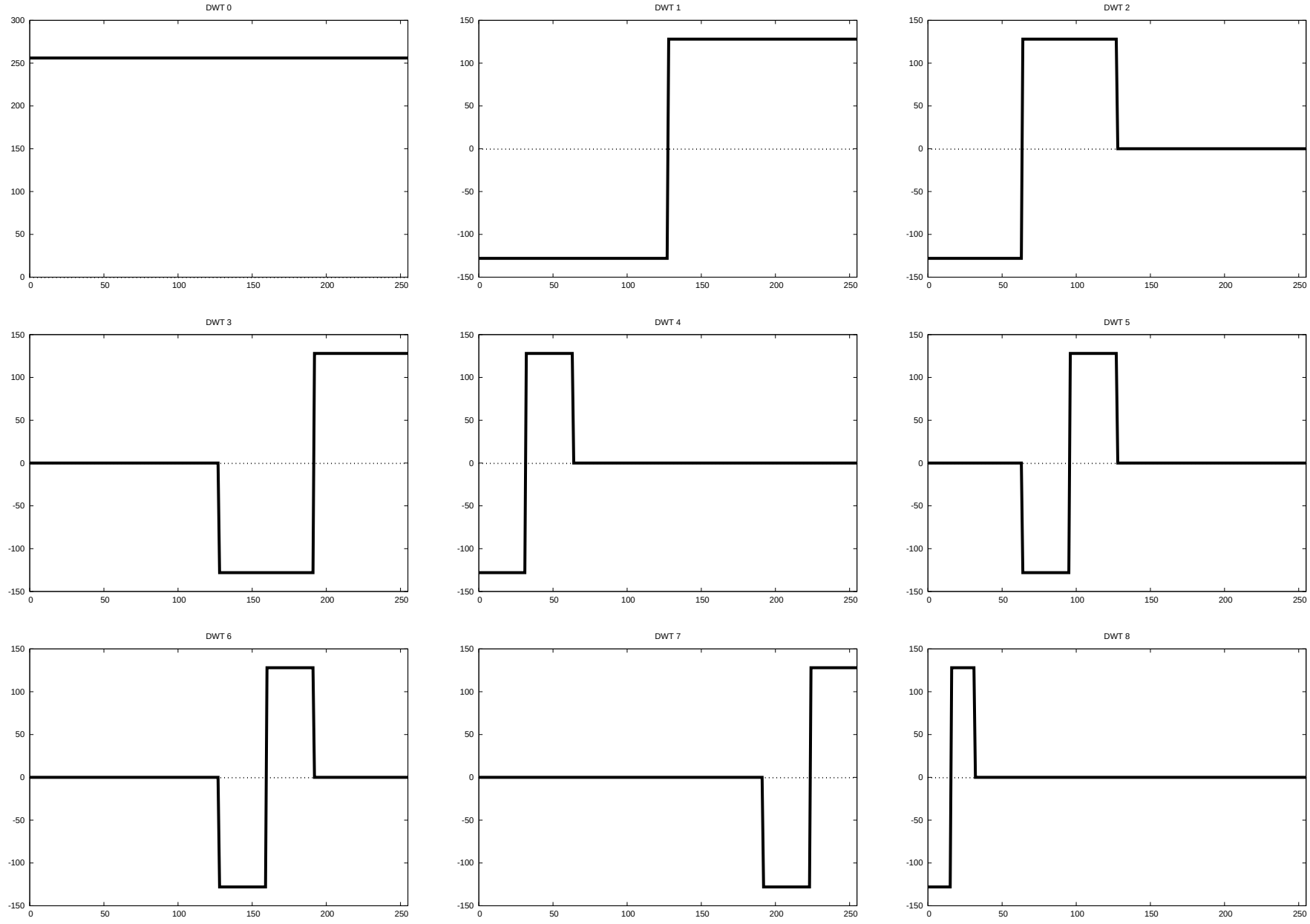
$$\{L_i\} = \left\{S_{2i} + \frac{H_{i-1} + H_i}{4}\right\}, \quad (28.13)$$

que como puede verse, es más eficiente que usar la Ecuación 28.11.

Ahora H_i almacena el error de predicción que separa a la muestra S_{2i+1} de ser el valor medio entre S_{2i} y S_{2i+2}. Esto provocará que las imágenes reconstruidas sean visiblemente más suaves que en el caso de la transformada de Haar.

- En el Apéndice 39.1 se muestra una implementación de la Transformada Lineal.

28.12. Funciones base de la Trans. Lineal



28.13. Transmisión progresiva de "lena" usando la Transformada Lineal

Véase la Sección 28.10.

Reconstrucción de "lena". 1.000 coefs

Reconstrucción de "lena". 2.000 coefs

28.13 Transmisión progresiva de "lena" usando la Transf. Lineal

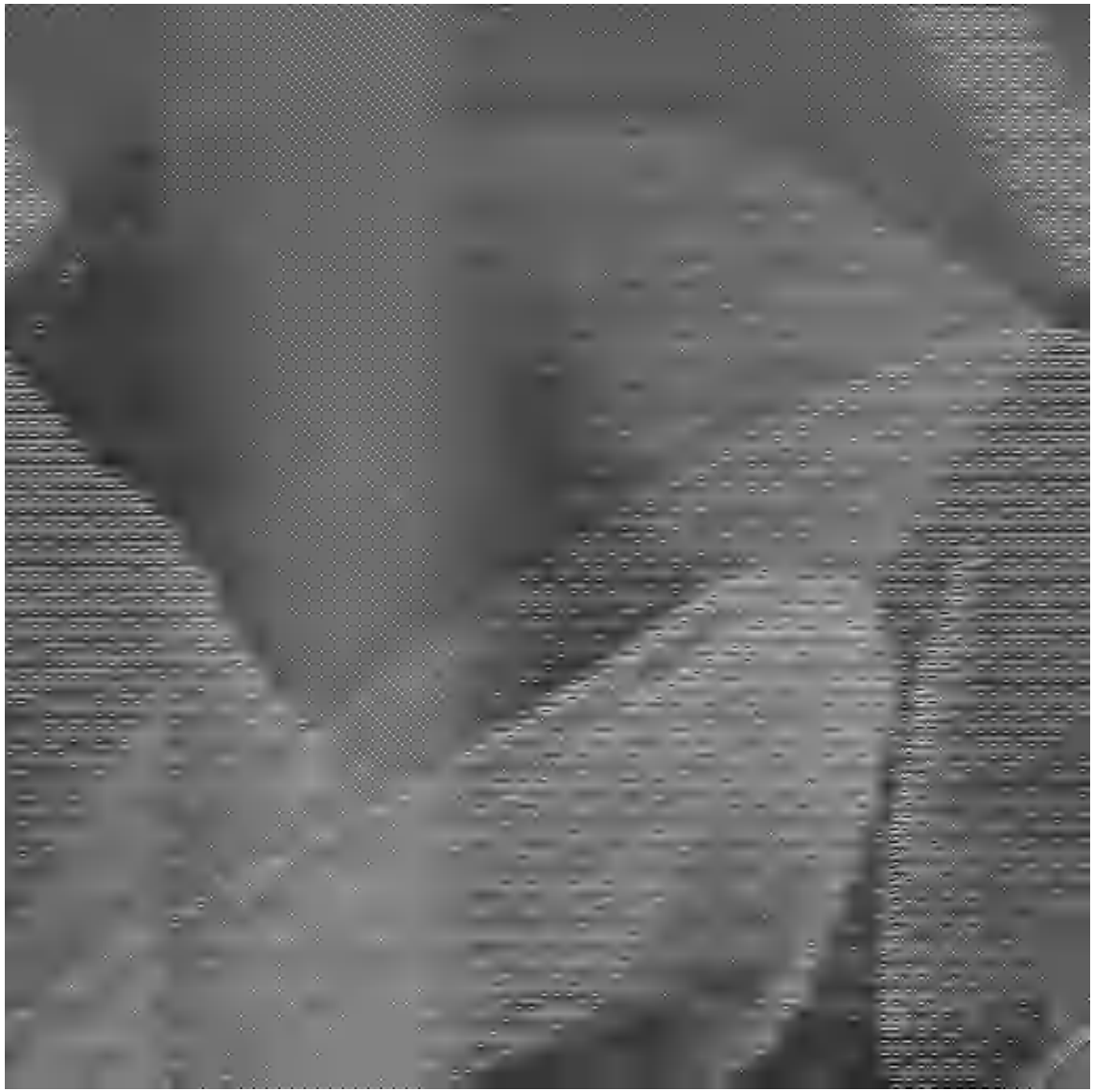
Reconstrucción de "lena". 3.000 coefs

28.13 Transmisión progresiva de "lena" usando la Transf. Lineal

Reconstrucción de "lena". 4.000 coefs

28.13 Transmisión progresiva de "lena" usando la Transf. Lineal

Reconstrucción de "lena". 5.000 coefs



28.13 Transmisión progresiva de "lena" usando la Transf. Lineal

28.14. La Transformada Spline 13/7 (cúbica)

- Realiza una interpolación para las muestras impares usando cuatro muestras pares[1, 19].
- La banda H se calcula como

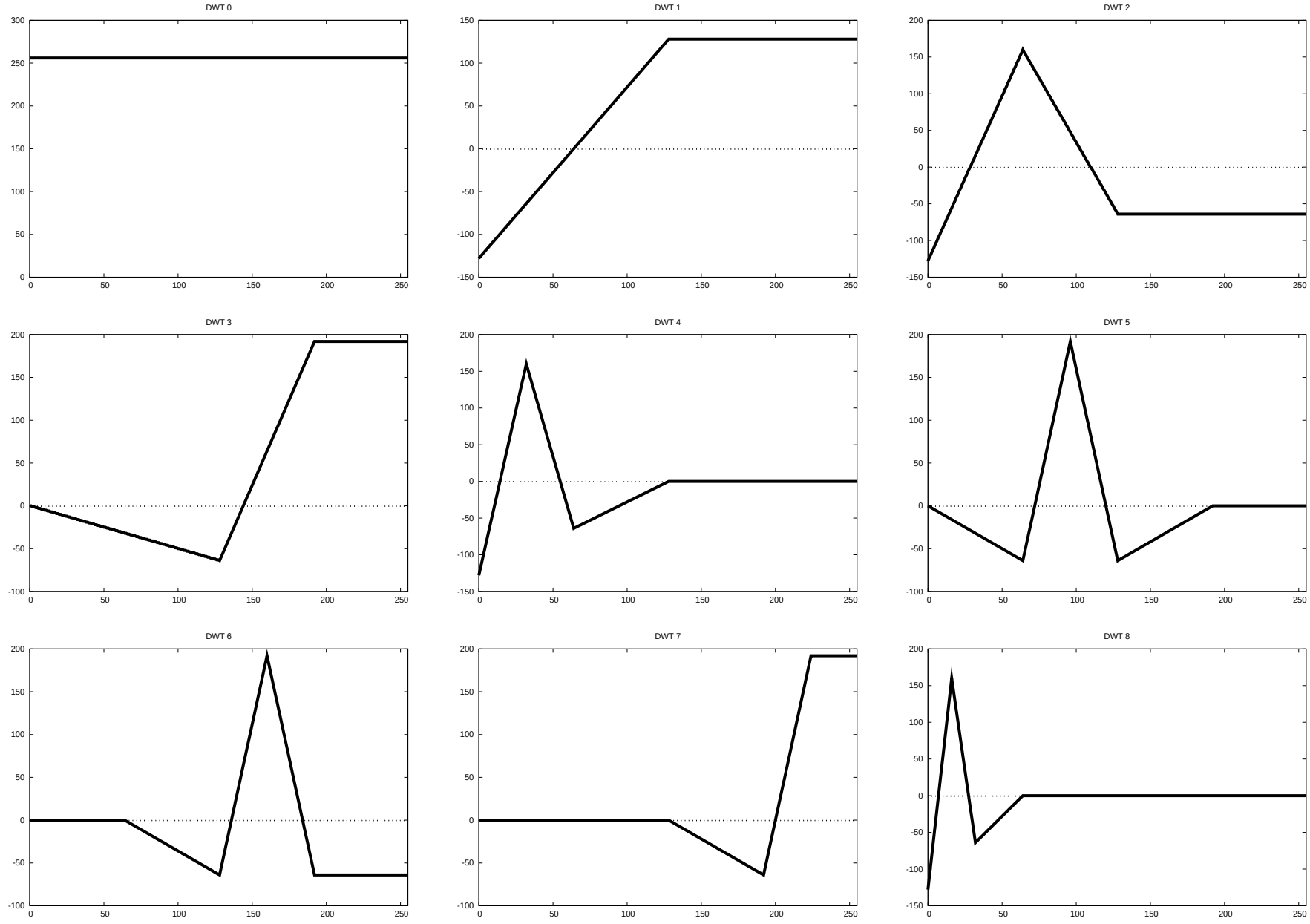
$$\{H_i\} = \left\{ S_{2i+1} - \left(\frac{9}{16}(S_{2i} + S_{2i+2}) - \frac{1}{16}(S_{2i-2} + S_{2i+4}) \right) \right\} \quad (28.14)$$

  y la banda L, usando Lifting, como

$$\{L_i\} = \left\{ S_{2i} + \frac{9}{32}(H_{i-1} + H_i) - \frac{1}{32}(H_{i-2} + H_{i+1}) \right\}. \quad (28.15)$$

- En el Apéndice 39.2 se muestra una implementación de la Transformada Cúbica.

28.15. Funciones base de la Trans. Cúbica



28.16. Transmisión progresiva de "lena" usando la Transf.ormadaCúbica

Véanse las Secciones 28.10 y 28.13.

Reconstrucción de "lena". 1.000 coefs

28.16 Transmisión progresiva de "lena" usando la Transf. Cúbica

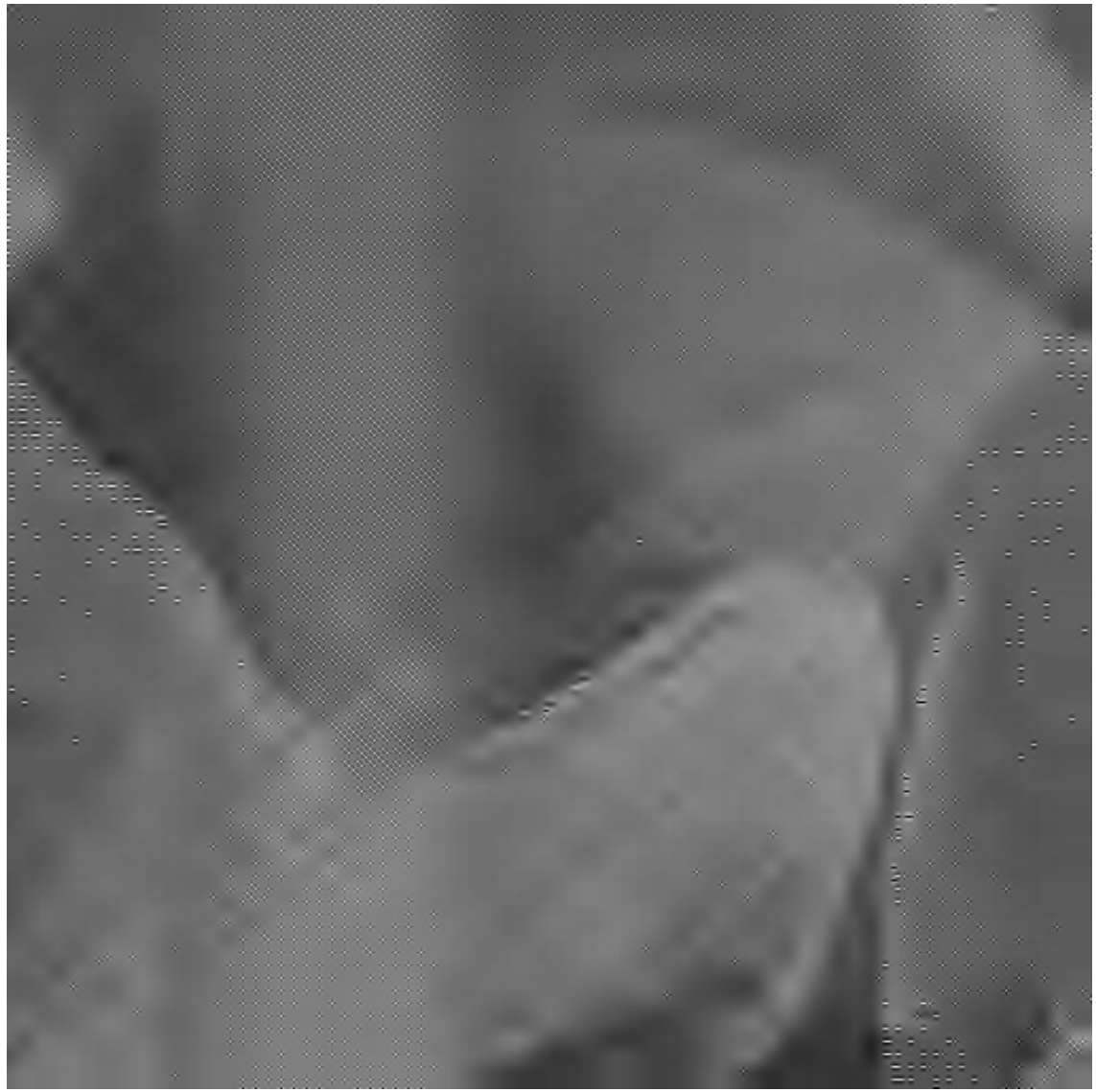
Reconstrucción de "lena". 2.000 coefs

28.16 Transmisión progresiva de "lena" usando la Transf. Cúbica

Reconstrucción de "lena". 3.000 coefs

28.16 Transmisión progresiva de "lena" usando la Transf. Cúbica

Reconstrucción de "lena". 4.000 coefs



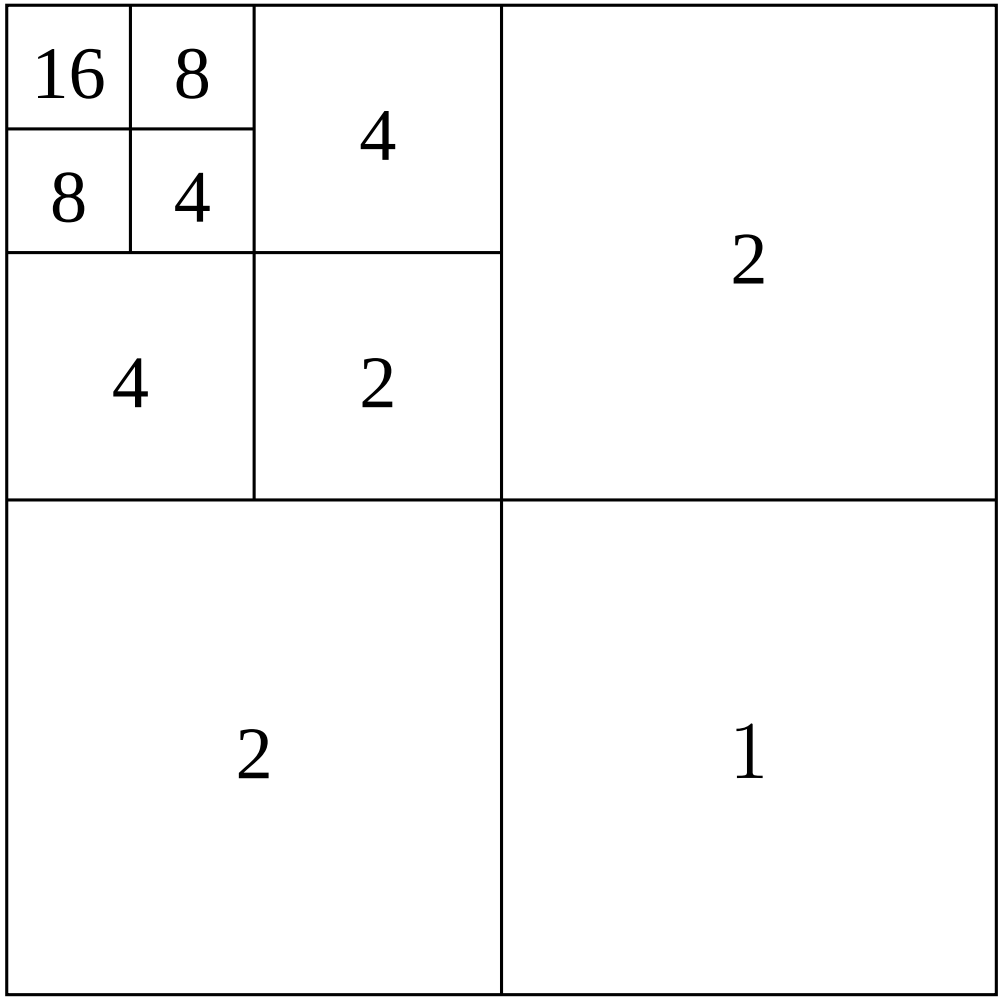
28.16 Transmisión progresiva de "lena" usando la Transf. Cúbica

Reconstrucción de "lena". 5.000 coefs

28.16 Transmisión progresiva de "lena" usando la Transf. Cúbica

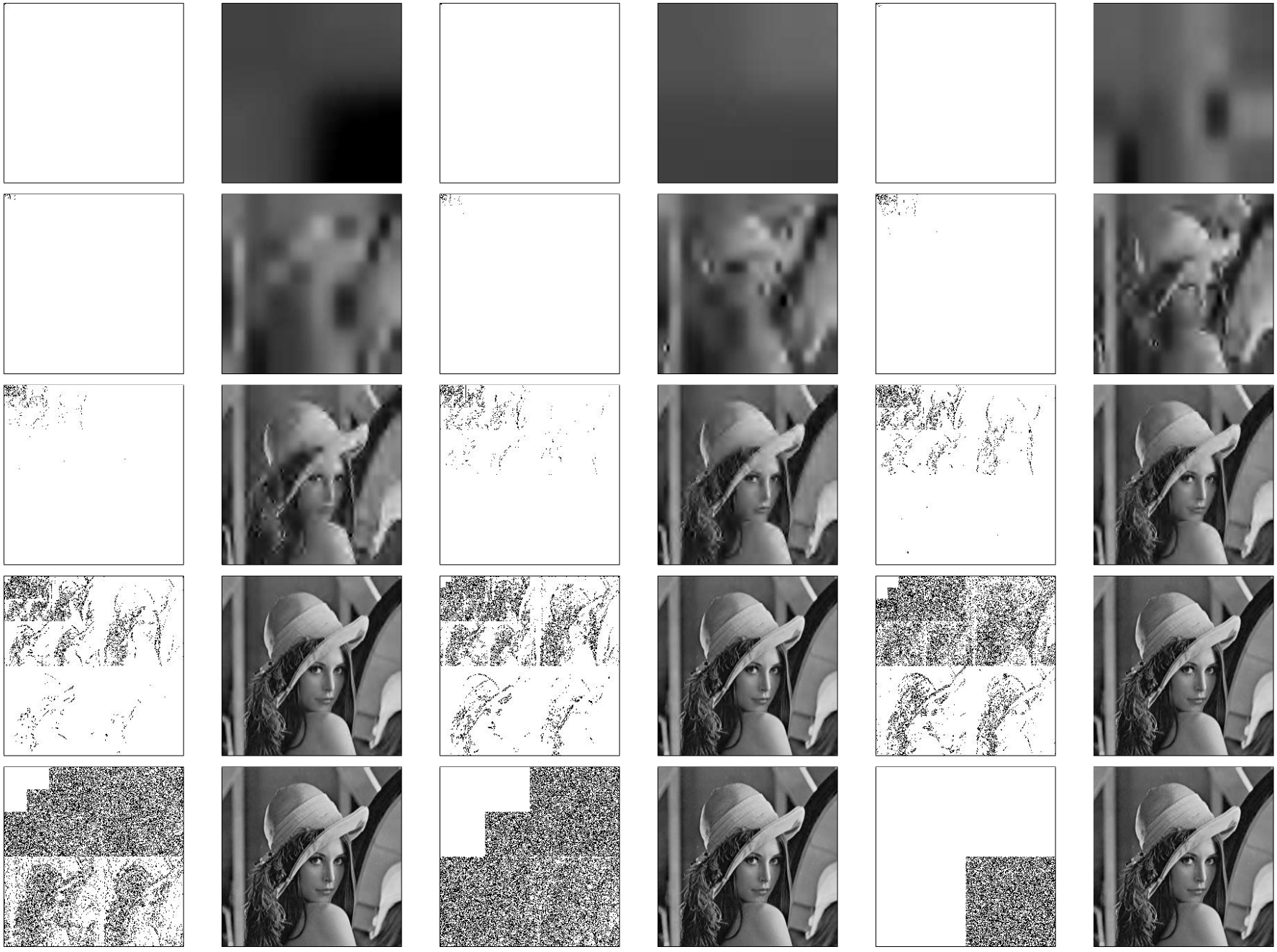
28.17. Transmisión de los coeficientes

- Para minimizar la distorsión es necesario concentrar la mayor cantidad de información en la mínima cantidad de bits.
- Una forma sencilla de hacer esto consiste en transmitir los coeficientes wavelet por planos de bits, comenzando por el más significativo.
- Además, cada banda de frecuencia no posee la misma importancia de cara a minimizar la distorsión.
- La importancia de una banda puede calcularse calculando la energía que aporta un coeficiente de esa banda a la imagen reconstruida.
- Una aproximación que normalmente funciona bien es utilizar los pesos:


16
8
8
4
4
4
2
4
2
2
1

28.18. Redundancia en el dominio wavelet (planos de bits)

- Para minimizar la cantidad de datos que se envían hay que explotar la correlación que existe en el dominio wavelet, dentro de cada plano de bits.
- Debido a los factores de ponderación de las bandas, los planos de bits de los coeficientes significativos (que son distinto de 0) se distribuyen formando una estructura piramidal que tiene su vértice superior en la esquina superior izquierda. Donde se localizan los coeficientes de menor frecuencia y con mayor magnitud.



28.18 Redundancia en el dominio wavelet (planos de bits)

- La principal fuente de redundancia en el espacio wavelet proviene de que si el coeficiente (i, j) no es significativo (su valor en valor absoluto es menor que 2^p donde p es el plano de bits transmitido) entonces todos sus descendientes $\mathcal{D}(i, j)$ no son significativos con una probabilidad muy alta, donde:

$$\begin{aligned}\mathcal{D}(i,j) \quad = \quad & \big\{(2i, 2j), (2i, 2j+1), \\ & (2i+1, 2j), (2i+1, 2j+1), \\ & \mathcal{D}(2i, 2j), \mathcal{D}(2i, 2j+1), \\ & \mathcal{D}(2i+1, 2j), \mathcal{D}(2i+1, 2j+1)\big\}\end{aligned} \tag{28.16}$$

  Decimos en este caso que tenemos un *zero-tree*. La ocurrencia de *zero-trees* es muy común en los planos de bits más significativos.

28.19. Un algoritmo básico de codificación progresivo

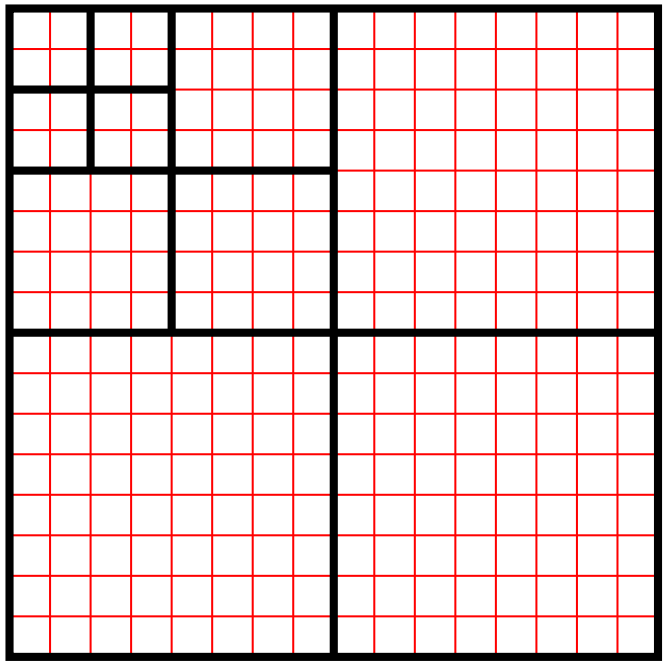
- La clave para diseñar compresores eficientes de planos de bits consiste en indicar dónde se encuentran los *zero-trees* usando un número de bits mínimo.
- Algoritmo (básico):
  1. Calcular la representación signo-magnitud de los coeficientes.
  2. Encontrar el plano de bits p más significativo.
  3. Codificar el plano de bits p usando *zero-trees* y emitir sus signos. Esto determina una lista inicial de coeficientes significativos.
  4. Mientras $p > 0$:
     a) $p \leftarrow p - 1$.
     b) Codificar el plano de bits p usando *zero-trees*, pero sólo tener en cuenta aquellos coeficientes que todavía no son significativos. Emitir sus signos.

c) Refinar el bit de peso p de aquellos coeficientes que eran significativos en planos superiores.

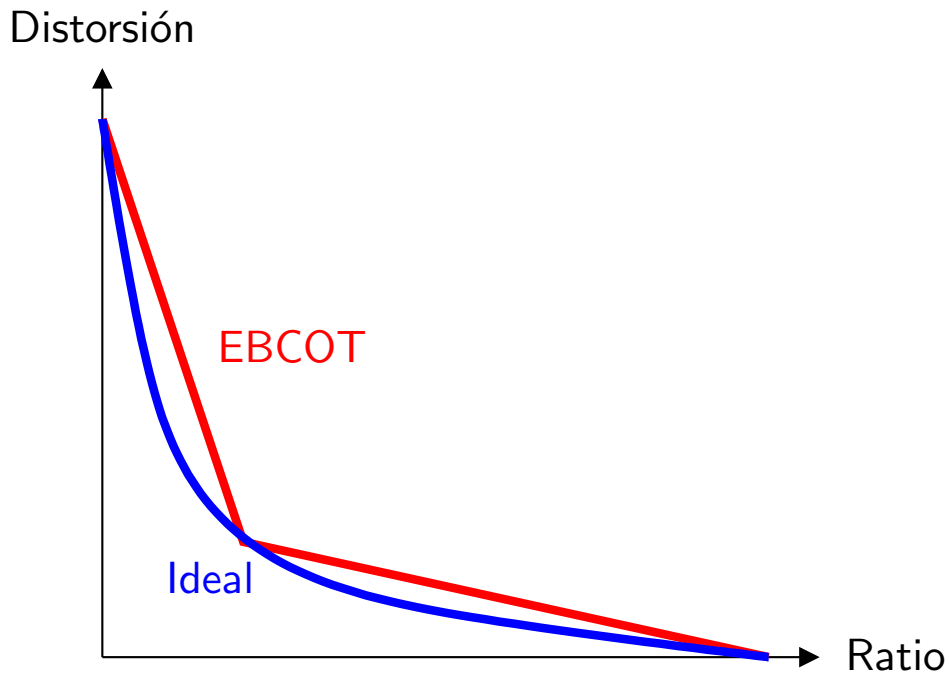
- Por desgracia este algoritmo posee un defecto: el descompresor necesita almacenar en memoria toda la imagen (en formato "wavelet") para poder descomprimirla. Esto, si la imagen es muy grande es un grave inconveniente.

28.20. EBCOT

- EBCOT: Embedded Block Coding with Optimal Truncation.
- Se basa en dividir la imagen wavelet en bloques cuadrados (típicamente de 32×32 o 64×64 coeficientes) llamados *code-blocks* y comprimir progresivamente (embedded) cada uno de forma independiente.



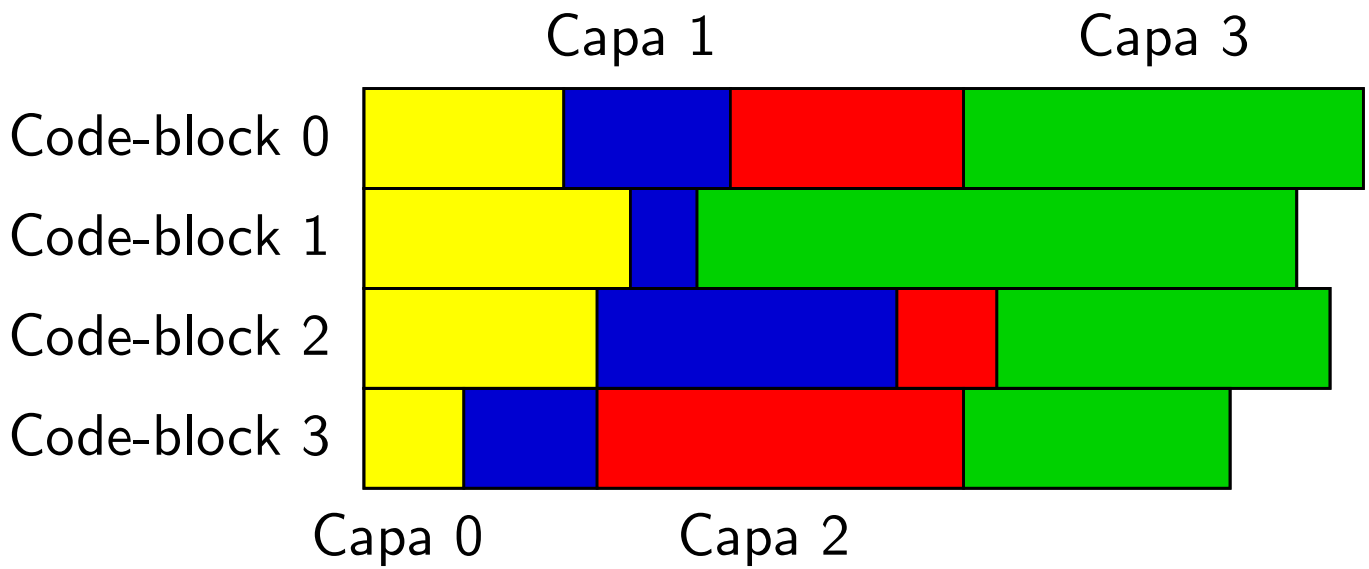
- Usando EBCOT no se explota la redundancia que existe entre los distintos code-blocks.[****]
- Cada bit-stream no es totalmente progresivo, aunque posee un gran cantidad de puntos de truncado óptimos[*****].



[****]Por ejemplo, no se elimina la redundancia padre-hijo presente en la jerarquía sub-banda.

[*****]En la figura se muestra un ejemplo en el que sólo existiría un punto de truncado óptimo.

- Una vez que los code-blocks se han comprimido, se seccionan y se reordenan los bit-streams (en capas o *layers*) para que se minimice la distorsión en el receptor dado un determinado volumen de bits.
- Así, el bit-stream se convierte en un packet-stream donde cada paquete contiene una cabecera que indica la contribución de ese code-block a esa *capa de calidad* seguido de los datos del correspondiente code-block truncados en un punto óptimo.

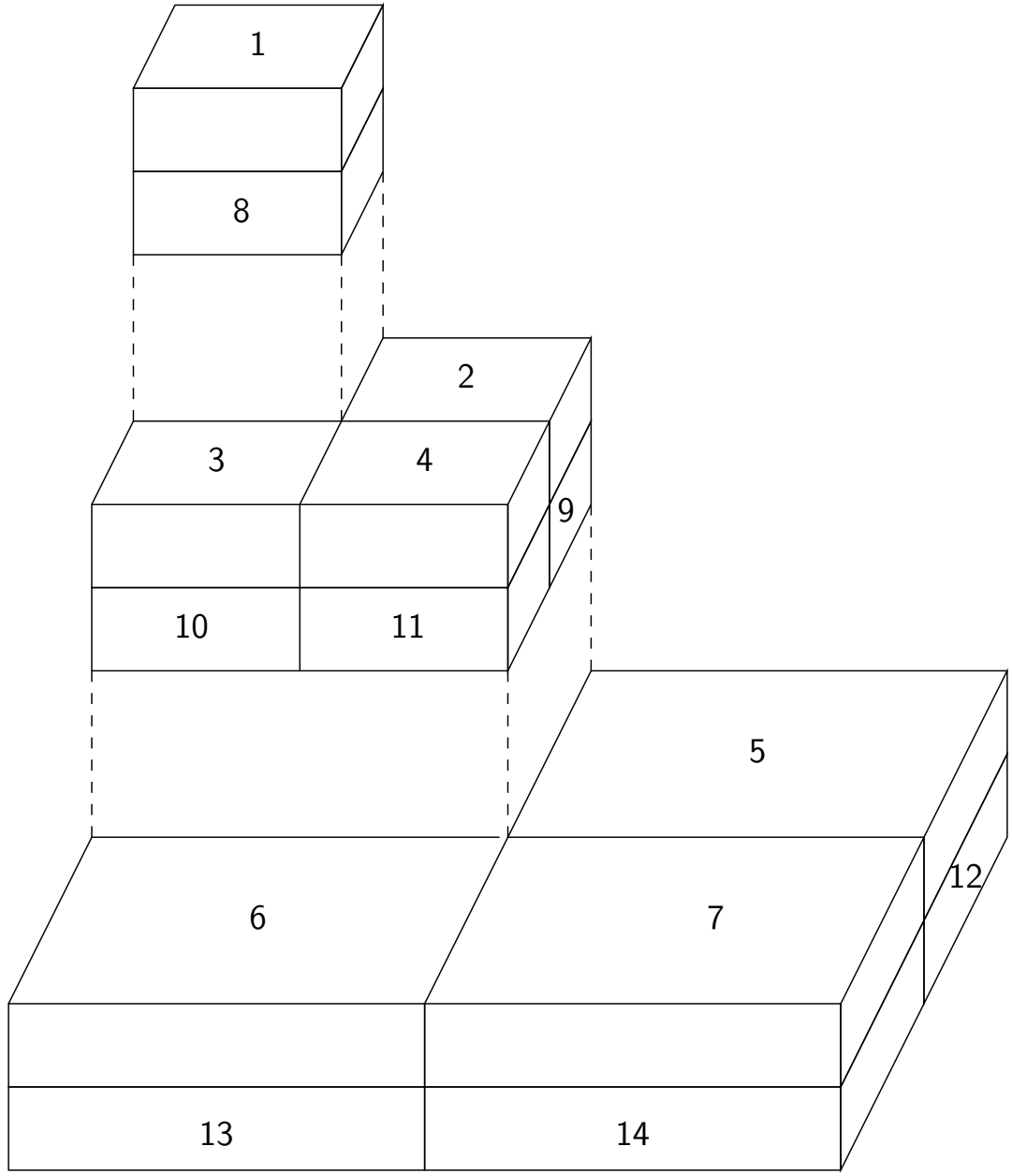


28.21. Progresiones y escalabilidad

- Utilizando EBCOT es posible ordenar la contribución de cada code-block a cada capa de calidad para descomprimir la imagen (1) por niveles de resolución (aprovechando las posibilidades de la 2D-DWT) o (2) por calidad (a la máxima resolución). En el caso (1) decimos que tenemos *escalabilidad espacial* y en el caso (2) que tenemos *escalabilidad en distorsión* (o en calidad, o SNR).

28.22. Progresión LR

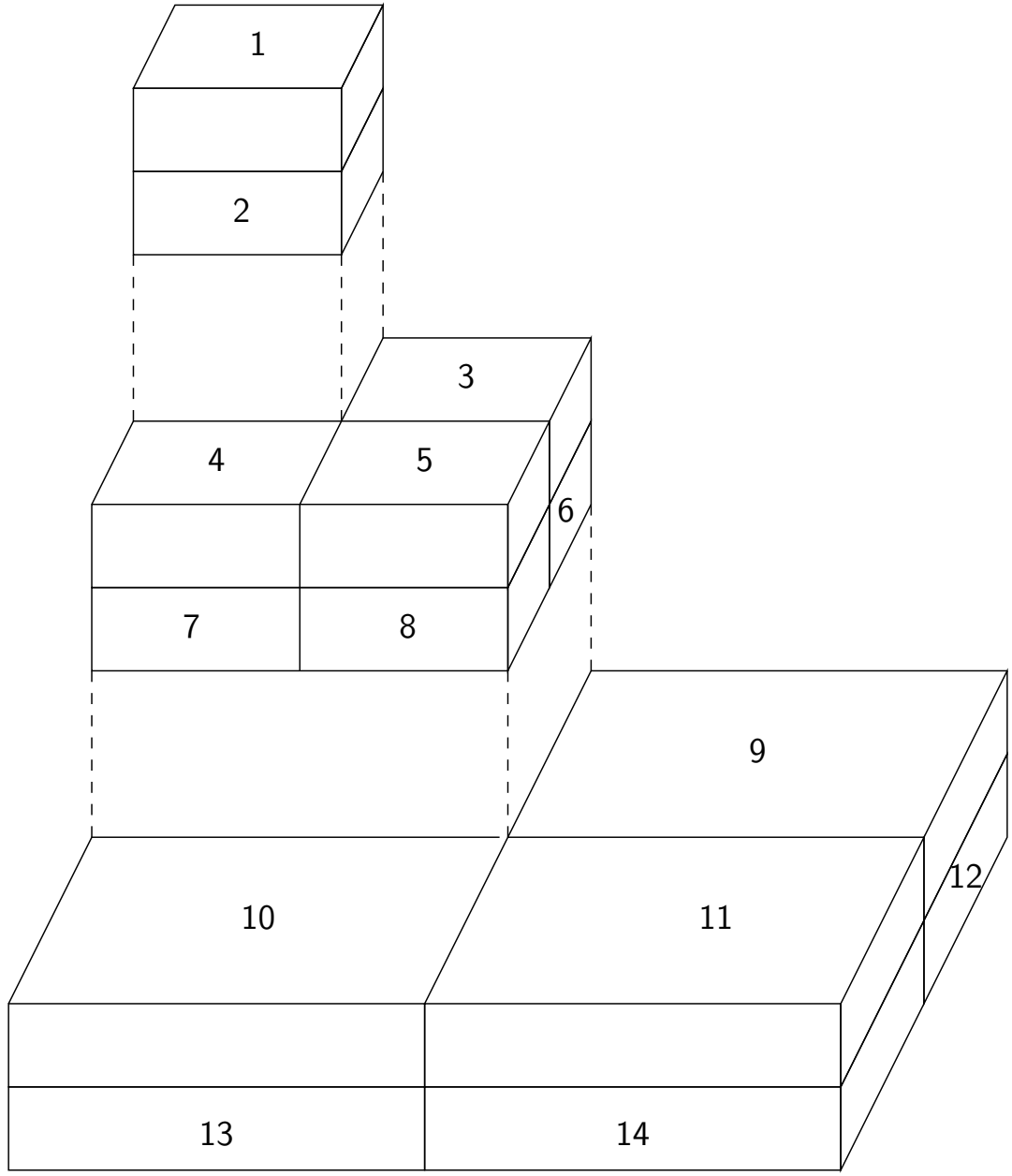
- La progresión (ordenación de los paquetes) que genera una reconstrucción por niveles de calidad también se llama LR (Layer-Resolution) y consiste en transmitir primero los planos de bits más significativos de todos los coeficientes.



28.22 Progresión LR

28.23. Progresión RL

- La progresión (ordenación de los paquetes) que genera una reconstrucción por niveles de resolución también se llama RL (Resolution-Layer) y consiste en transmitir primero los planos de bits de los coeficientes que generan los distintos niveles de resolución.



28.23 Progresión RL

28.24. El algoritmo JPEG2000

Para una imagen RGB, el algoritmo básico consiste en:

1. Level offset para cada componente (opcional).
2. Descorrelacionar las componentes (opcional).
3. Aplicar la 2D-DWT.
4. Cuantificar.
5. Definir ROI's.
6. Codificar entrópicamente.

28.25. Level offset

Depende de si se realiza una compresión reversible o irreversible. Se aplica independientemente a cada componente.

1. **Irreversible:** hacemos que todas las muestras $s[\boldsymbol{n}]$ cumplan que

$$-\frac{1}{2} \leq s[\boldsymbol{n}] \leq \frac{1}{2} \tag{28.17}$$

   donde $s[\boldsymbol{n}] = s[x, y]$ es el punto de coordenadas (x, y) de esa componente.

2. **Reversible:** hacemos que todas las muestras $s[\boldsymbol{n}]$ cumplan que

$$-2^{B-1} \leq s[\boldsymbol{n}] < 2^{B-1} \tag{28.18}$$

   donde B es el número de bits/componente.

28.26. Descorrelacionar las componentes

Se aplica sólo a imágenes en color (RGB) y depende de si la compresión es reversible o irreversible.

1. **Irreversible:**

$$\begin{aligned} \mathsf{Y} &= 0{,}299\mathsf{R} + 0{,}587\mathsf{G} + 0{,}144\mathsf{B} \\ \mathsf{Cb} &= \frac{0{,}5}{1-0{,}144}(\mathsf{B}-\mathsf{Y}) \\ \mathsf{Cr} &= \frac{0{,}5}{1-0{,}299}(\mathsf{R}-\mathsf{Y}) \end{aligned} \tag{28.19}$$

2. **Reversible:**

$$\begin{aligned} \mathsf{Y}' &= \left\lfloor \frac{\mathsf{R}+2\mathsf{G}+\mathsf{B}}{4} \right\rfloor \\ \mathsf{Db} &= \mathsf{B}-\mathsf{G} \\ \mathsf{Dr} &= \mathsf{R}-\mathsf{G} \end{aligned} \tag{28.20}$$

28.27. Aplicar la 2D-DWT

Se aplica de forma independiente a cada componente y depende de si la compresión es reversible o irreversible.

1. **Irreversible:**

$$
\begin{aligned}
L(z) = & \\
& 0{,}602949018236 \\
+ \quad & 0{,}266764118443(z^1 + z^{-1}) \\
- \quad & 0{,}078223266529(z^2 + z^{-2}) \\
- \quad & 0{,}016864118443(z^3 + z^{-3}) \\
+ \quad & 0{,}026748757411(z^4 + z^{-4})
\end{aligned}
\tag{28.21}
$$

$$
\begin{aligned}
H(z) = & \\
& 0{,}557543526229 \\
+ \quad & 0{,}295635881557(z^1 + z^{-1}) \\
- \quad & 0{,}028771763114(z^2 + z^{-2}) \\
- \quad & 0{,}045635881557(z^3 + z^{-3})
\end{aligned}
$$

2. **Reversible:** Es exactamente la transformada spline 5/3 (véase la Sección 28.11) que expresada mediante la Transformada Z queda como:

$$\begin{aligned} H(z) &= -\tfrac{1}{8}(z^2 + z^{-2}) + \tfrac{1}{4}(z^1 + z^{-1}) + \tfrac{3}{4} \\ L(z) &= -\tfrac{1}{2}(z^1 + z^{-1}) + 1 \end{aligned} \qquad (28.22)$$

28.28. Cuantificación

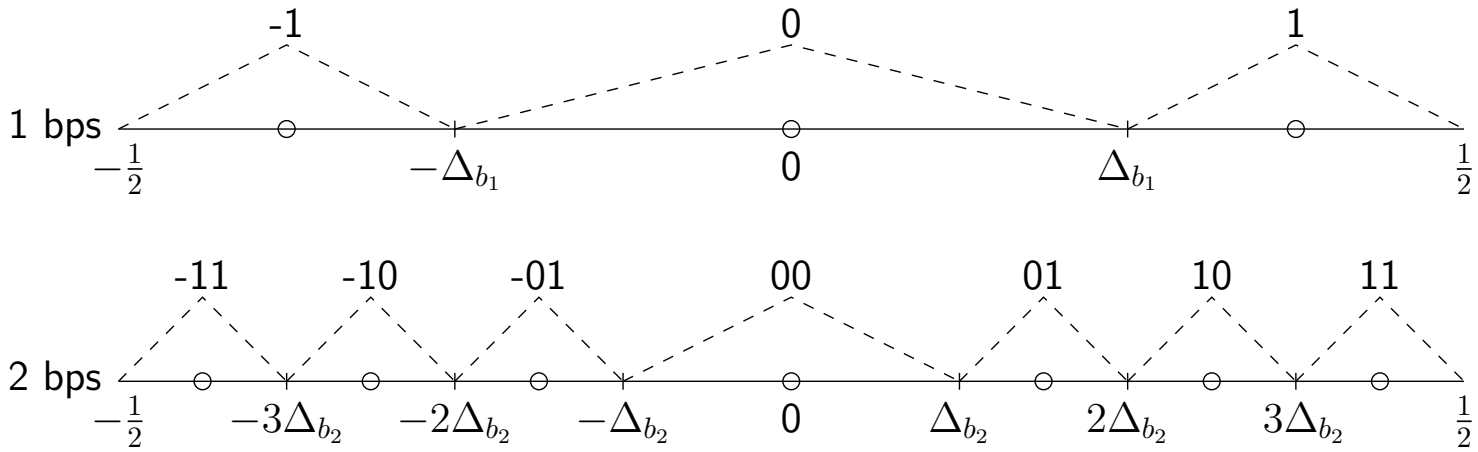
- La fase de cuantificación (que sólo se realiza cuando se está comprimiendo de forma irreversible) es la principal fuente de distorsión en la compresión.

  1. **Irreversible:** tiene como entrada los coeficientes wavelet que están en el rango $[-0{,}5, 0{,}5]$ y como salida una palabra binaria (o índice de cuantificación) de x bits que depende del nivel de compresión seleccionado. Esta consiste en

$$q_b = \text{sign}(y_b) \left\lfloor \frac{|y_b|}{\Delta_b} \right\rfloor \tag{28.23}$$

     donde y_b es un coeficiente wavelet de la banda de frecuencia b y Δ_b es el tamaño del paso del cuantificador progresivo con "dead-zone", es decir, el parámetro que define el nivel de compresión. Δ_b es distinto para cada banda porque el peso relativo de cada una de ellas es diferente (véase la Sección 28.17).

Un ejemplo para los dos primeros niveles de cuantificación:



2. **Reversible:** en este caso lo índices de cuantificación son idénticos a los coeficientes wavelet

$$q_b = y_b \tag{28.24}$$

tendiendo en cuenta los pesos de cada banda.

28.29. Regiones de interés

- Utilizando el particionamiento por code-blocks es posible definir con una cierta precisión una ROI.
- Sin embargo, cuando se necesita mayor resolución podemos multiplicar los coeficientes wavelet correspondientes por un valor que determina la importancia de la ROI.

28.30. Codificación entrópica

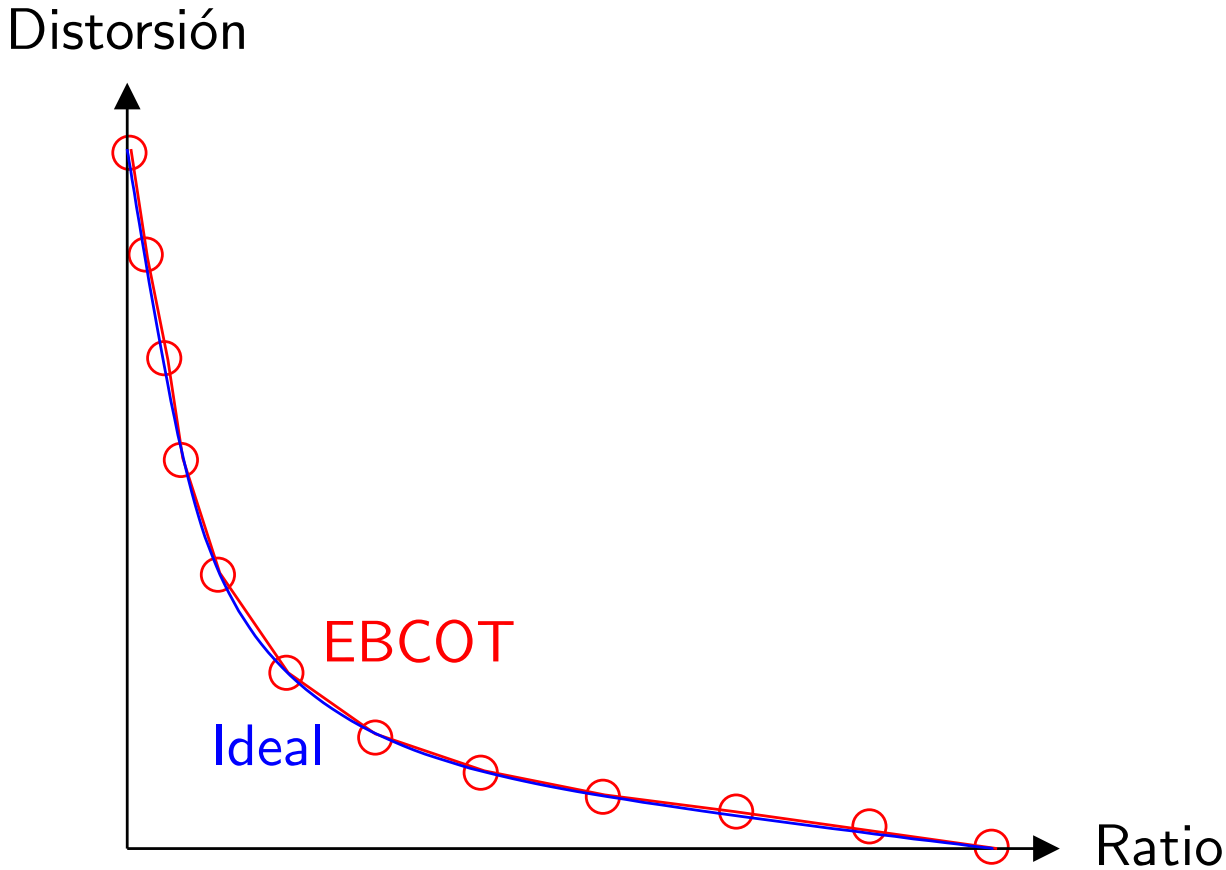
- Cada code-block se comprime independientemente utilizando codificación aritmética binaria (MQ coder) y un modelo espacial.
- En cada pasada se comprime un plano de bits y cada pasada tiene tres fases no progresivas:

  1. **Pasada de propagación de la significancia:** en ella se indican al descompresor qué coeficientes, de los que se espera que sean significativos en ese plano de bits, son realmente significativos. Con ellos también se envían los signos.
  2. **Pasada de refinamiento de la magnitud:** se refinan los coeficientes que ya se conocen significativos.
  3. **Pasada de limpieza general (clean-up):** se indican los coeficientes que son significativos y no han sido encontrados en la pasada 1.

- Si B es el número de bits/coeficiente, el número de puntos de truncado óptimos es

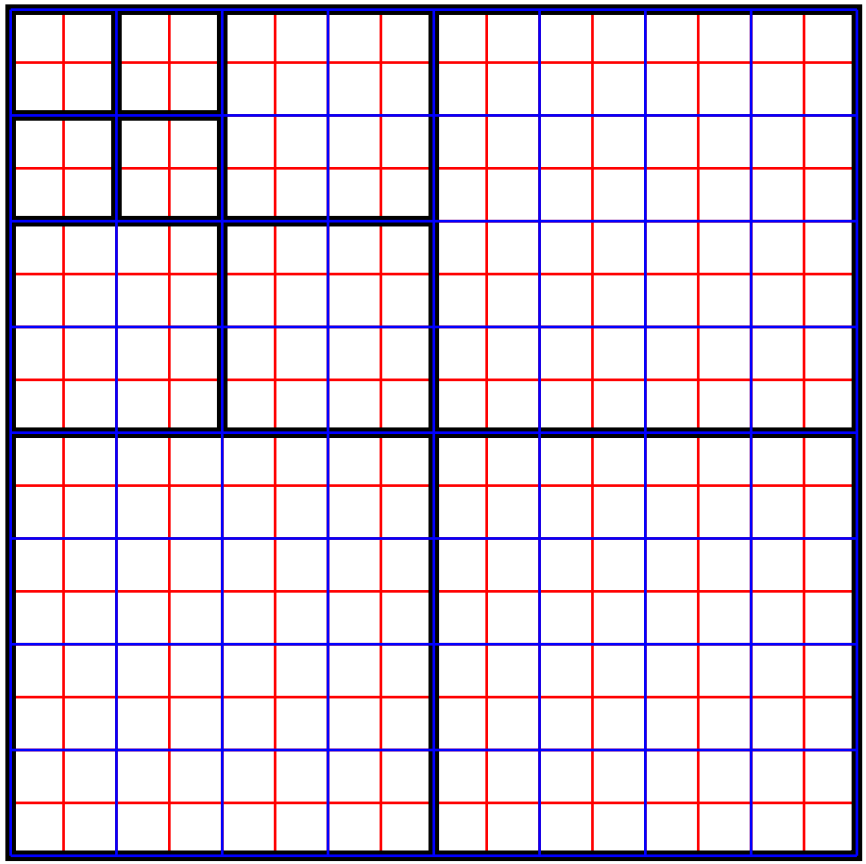
$$3B - 2 \tag{28.25}$$

- Nótese además que al comienzo de la transmisión grandes decrementos en la distorsión son esperados para cada nuevo punto de truncado óptimo.



28.31. Estructura real del packet-stream

- Si las imágenes son suficientemente grandes el número de paquetes (en principio el número de code-blocks que existen multiplicados por el número de capas de calidad) puede ser muy grande. Como cada paquete tiene asociado un tamaño ya que la contribución de ese code-block a esa capa de calidad es variable, la cantidad de índices en el fichero JPEG2000 puede ser considerable.
- Por esta razón se define una partición nueva en el dominio wavelet: el precinto.



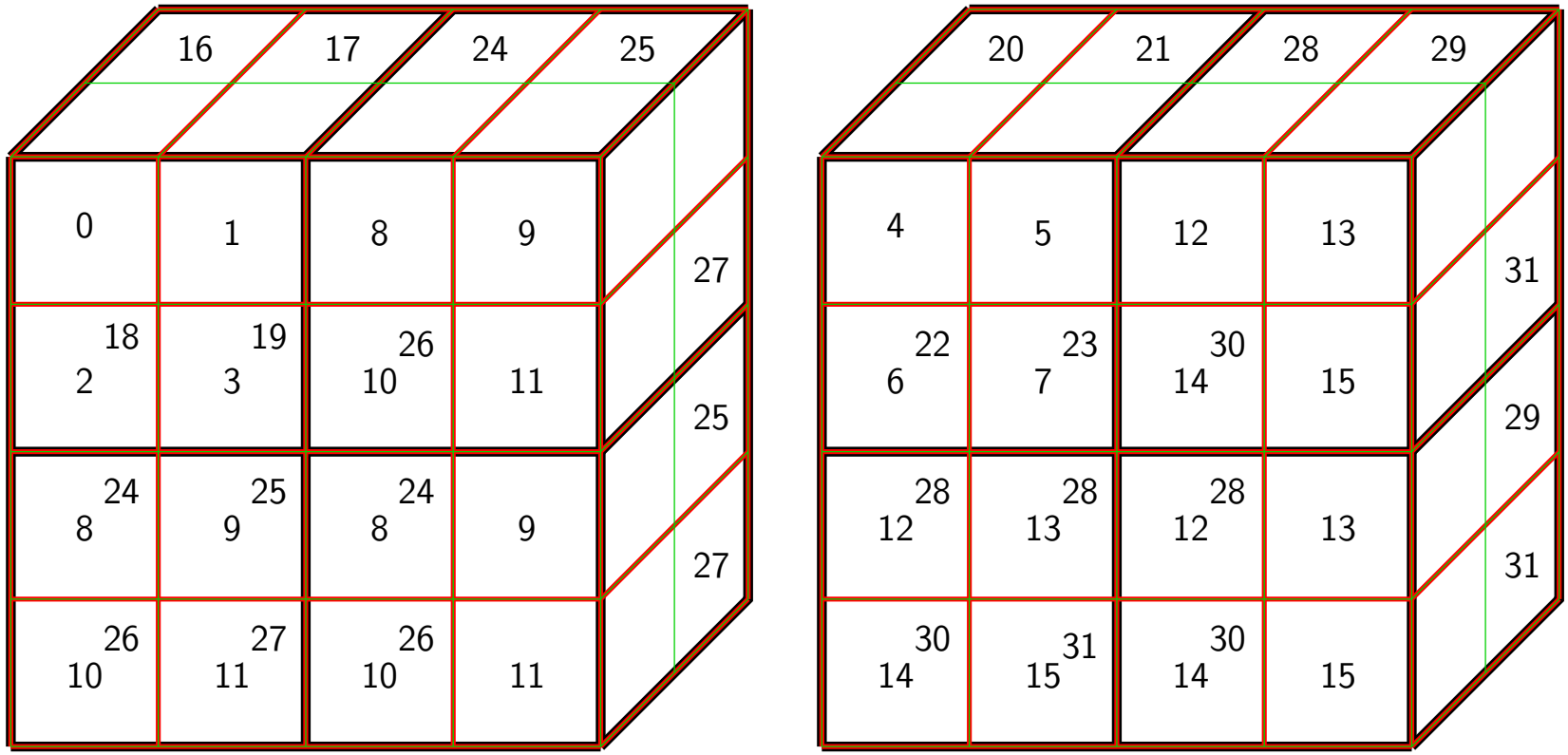
En este figura se ha supuesto que todos los precintos son del mismo tamaño. El estándar permite que los precintos de diferente nivel de resolución tengan tamaños diferentes.

- En JPEG2000, cada precinto genera exactamente un paquete, para cada capa de calidad.

28.32. Progresiones en JPEG2000

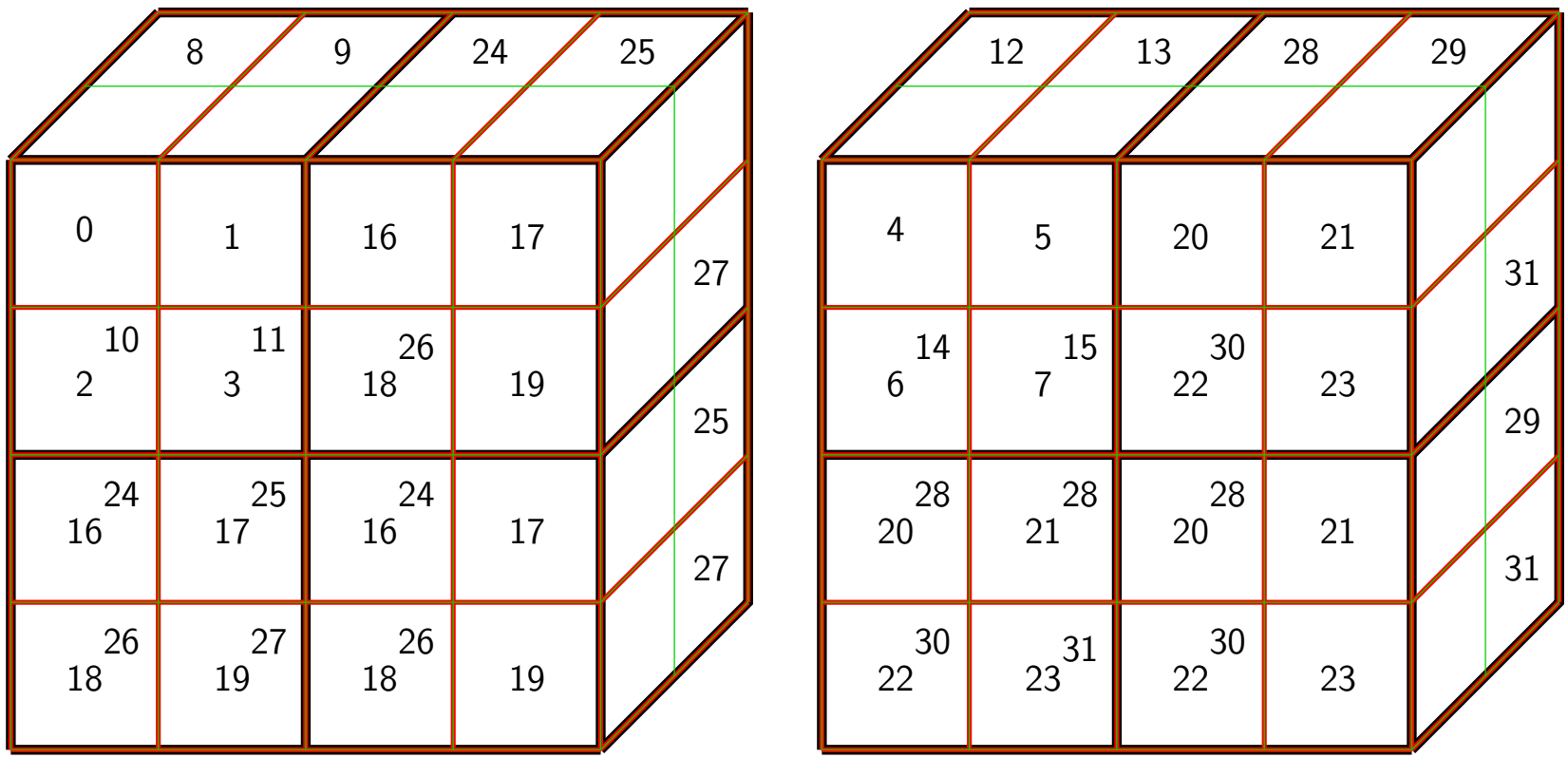
- En realidad, en JPEG2000 se genera un paquete distinto para cada precinto P (o posición en el dominio wavelet), componente C, nivel de resolución R y capa de calidad L. Dependiendo de cómo se ordenan los paquetes en el packet-stream tenemos las siguientes progresiones (existen más, pero estas son las más frecuentes):

1. **LRCP o progresión por calidad:** primero ordenamos los paquetes por capas de calidad, luego por niveles de resolución, luego por componente y finalmente por precinto. Un ejemplo:



Componente 0 Componente 1

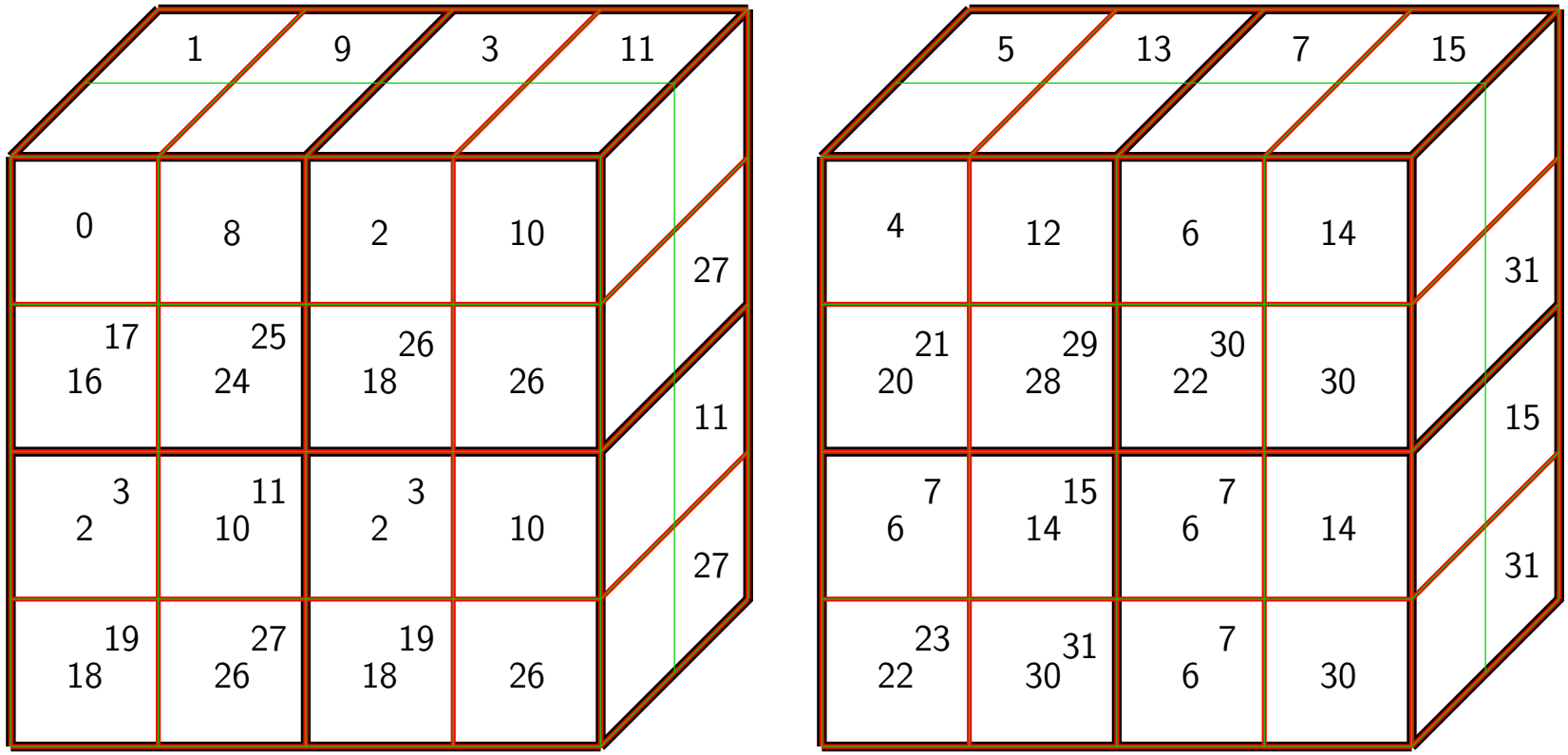
2. **RLCP o progresión por niveles de resolución:** primero ordenamos los paquetes por niveles de resolución, luego por capas de calidad, luego por componente y finalmente por precinto. Un ejemplo:



Componente 0

Componente 1

3. **PCRL o progresión secuencial:** primero ordenamos los paquetes por precinto, luego por componente, a continuación por resolución y finalmente por capa de calidad. Un ejemplo:



Componente 0 Componente 1

28.33. JPEG versus JPEG2000

- Vamos a comparar JPEG y JPEG 2000 para dos imágenes: “lena” y “cat”.
- Como medida objetiva de la distorsión generada se va a utilizar el PSNR (véase el Apéndice 35).

“Lena” a $0{,}1$ bpp

JPEG ($21{,}29$ dB)

JPEG2000 ($27{,}03$ dB)

“Lena” a $0{,}2$ bpp

JPEG ($26{,}64$ dB)

JPEG2000 ($29{,}22$ dB)

“Lena” a $0{,}3$ bpp

JPEG ($28{,}97$ dB)

JPEG2000 ($30{,}71$ dB)

“Lena” a $0{,}4$ bpp

JPEG ($30{,}09$ dB)

JPEG2000 ($31{,}58$ dB)

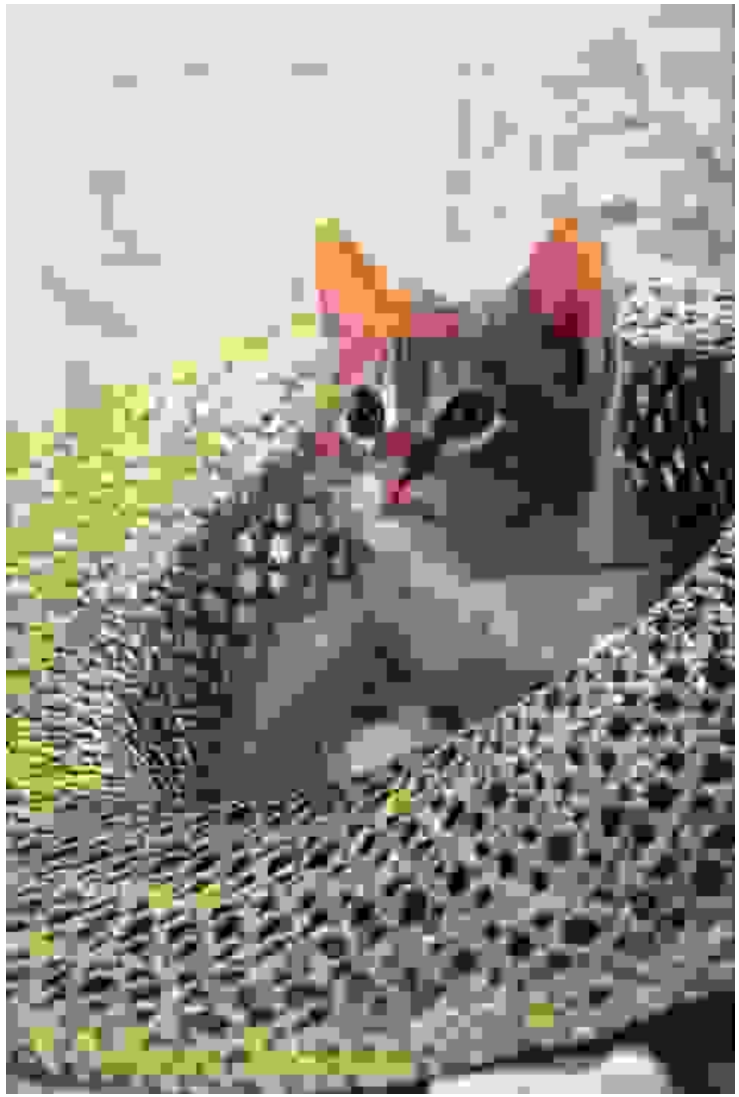
“Lena” a $0{,}5$ bpp

JPEG ($30{,}91$ dB)

JPEG2000 ($32{,}24$ dB)

"Cat" a 0,1 bpp

JPEG (17,79 dB)



JPEG2000 (23,33 dB)

"Cat" a 0,2 bpp

JPEG (23,59 dB)

JPEG2000 (25,97 dB)

"Cat" a 0,3 bpp

JPEG (25,63 dB)

JPEG2000 (27,60 dB)

"Cat" a 0,4 bpp

JPEG (27,10 dB)

JPEG2000 (28,97 dB)

"Cat" a 0,5 bpp

JPEG (28,23 dB)

JPEG2000 (30,08 dB)

28.34. Compresión de "akiyo"

- Al igual que JPEG, JPEG2000 no puede comprimir secuencias de imágenes, pero puede ser llamado iterativamente para generar un resultado semejante (M-JPEG2000). La ventaja de JPEG2000 radica en que además de alcanzar mejores tasas de compresión, podemos conseguir que cada imagen tenga el número de bits deseados.

- Para comprimir "akiyo", utilizaremos una versión comercial del estándar JPEG2000 llamada Kakadu (`http://www.kakadusoftware.com`). Al igual que con JPEG, usaremos una tasa de compresión de 25 (o $0{,}96$ bits/pixel). El comando usado para comprimir la secuencia ha sido:

```
for i in akiyo???
do
  kdu_compress -i $i -o $i.jp2 -rate 0.96
done
```

- De forma semejante a M-JPEG, M-JPEG2000 permite la escalabilidad temporal. Sin embargo, también disponemos del resto de escalabilidades: espacial, en calidad y ROI.
- Kakadu también dispone de una utilidad para comprimir directamente ficheros YUV usando M-JPEG2000. Véase el Apéndice 39.56 para ver el comando de llamada a esta utilidad.
- En el directorio `vids` encontrará diferentes secuencias de vídeo codificadas con M-JPEG2000. Busque los ficheros `*MJEG2000*`.

Parte III

Compresión de vídeo

Capítulo 29

La compresión de vídeo

Fundamentos

- Vídeos = Secuencias de imágenes.
- Los compresores de secuencias de imágenes eliminan tanto la **redundancia espacial** como la **temporal**. Esta última es consecuencia directa de que en la mayoría de las secuencias, las imágenes adyacentes en el tiempo son muy parecidas y por lo tanto, para codificar la siguiente (por ejemplo), es muy eficiente indicar sólo las diferencias con respecto a la actual.
- Debido a la cantidad de datos que se generan cuando se comprime una señal de vídeo, la inmensa mayoría de los **compresores** de secuencias de imágenes (también llamados compresores de vídeo) son **lossy**, porque las tasas de compresión son mucho mayores.

29.1. Motivación

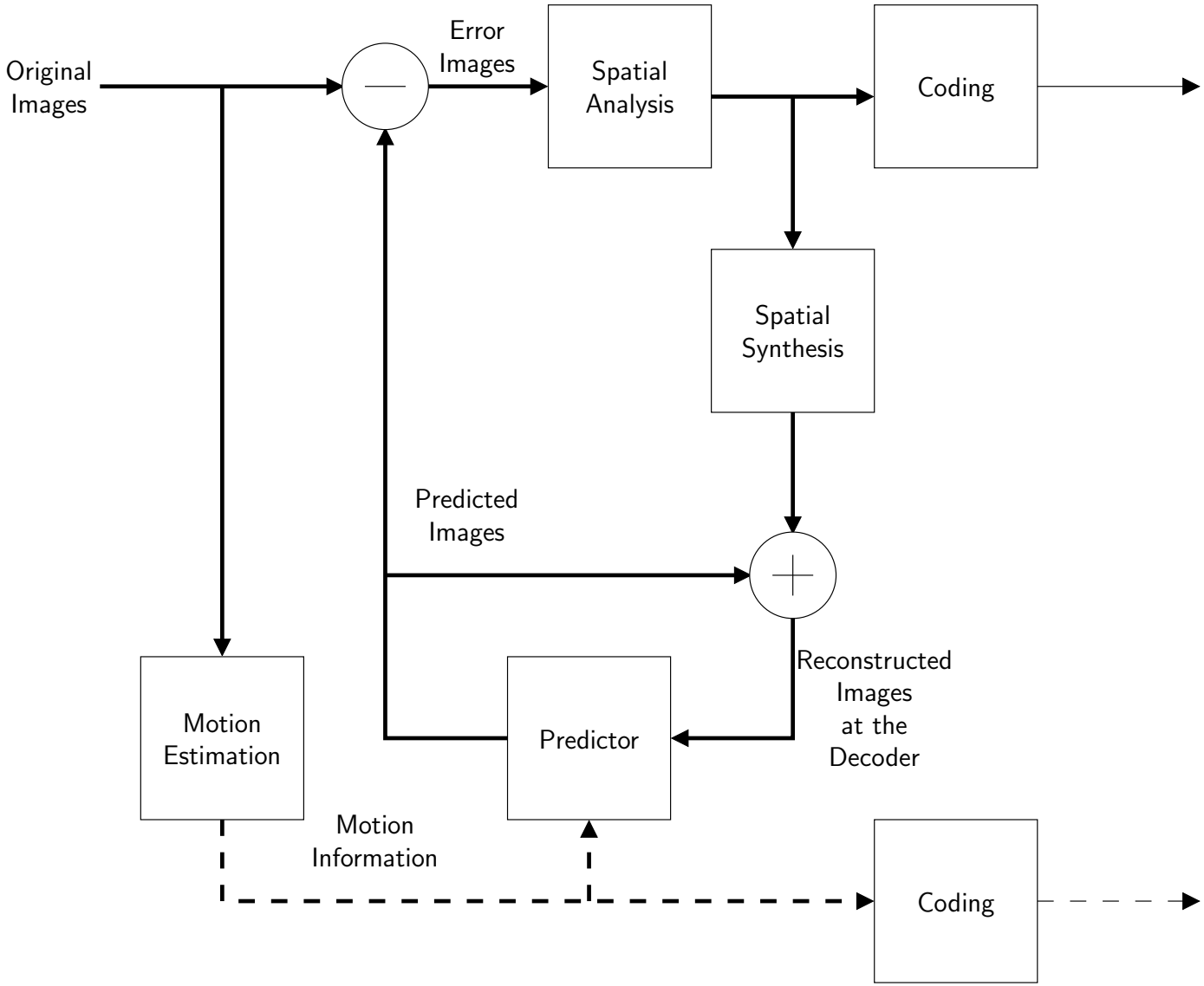
- Las secuencias de vídeo digital en formato PCM ocupan mucha memoria. Un segundo de vídeo en color, a una resolución de 640×480 puntos/imagen y 25 imágenes/segundo necesita:

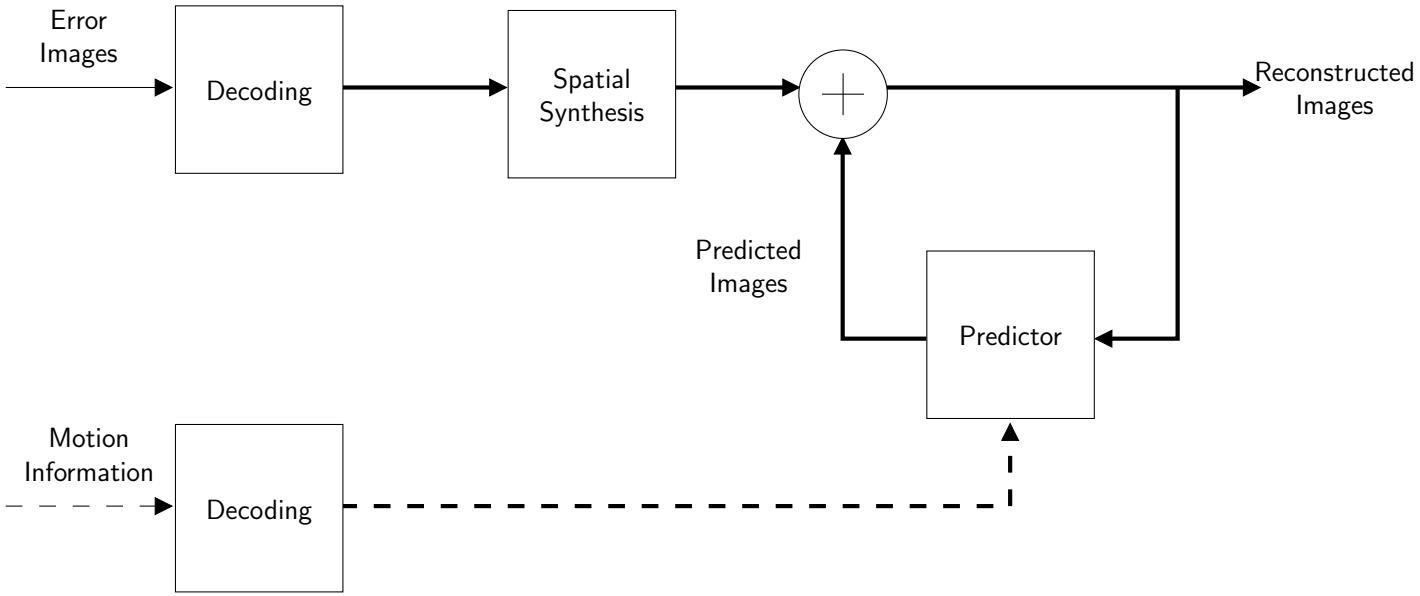
$$25\frac{\text{imágenes}}{\text{segundo}} \times 640 \cdot 480\frac{\text{puntos}}{\text{imagen}} \times 24\frac{\text{bits}}{\text{punto}} = 184.320.000\frac{\text{bits}}{\text{segundo}}$$

Esto significa que, por ejemplo, una hora de dicho vídeo ocupa:

$$184.320.000\frac{\text{bits}}{\text{segundo}} \times 3.600\frac{\text{segundos}}{\text{hora}} \times \frac{1\text{ G}}{1.024^3} \times \frac{1\text{ byte}}{8\text{ bits}} \approx 77\text{ Gbytes}$$

29.2. Closed Loop t+2D Coding

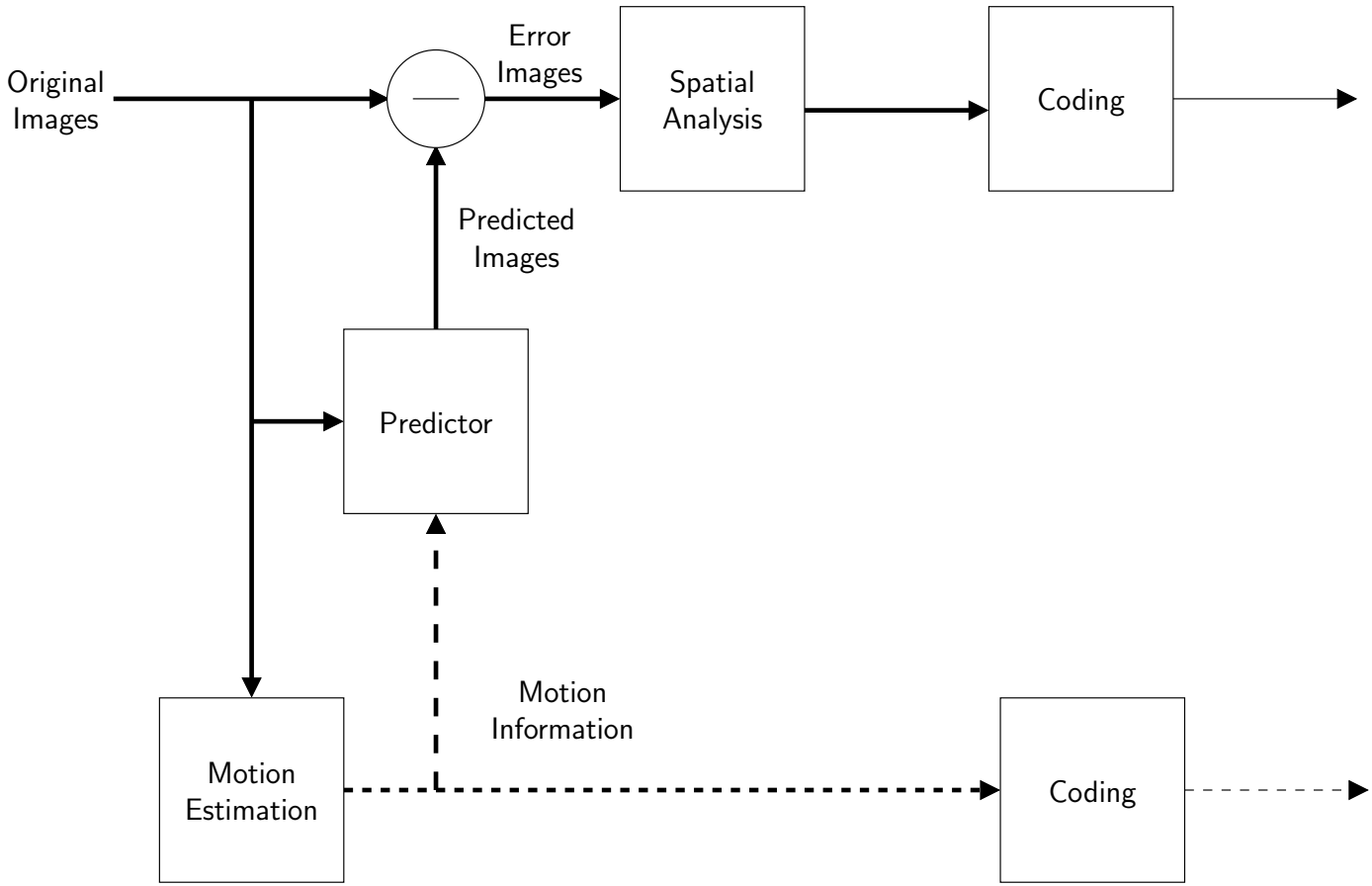


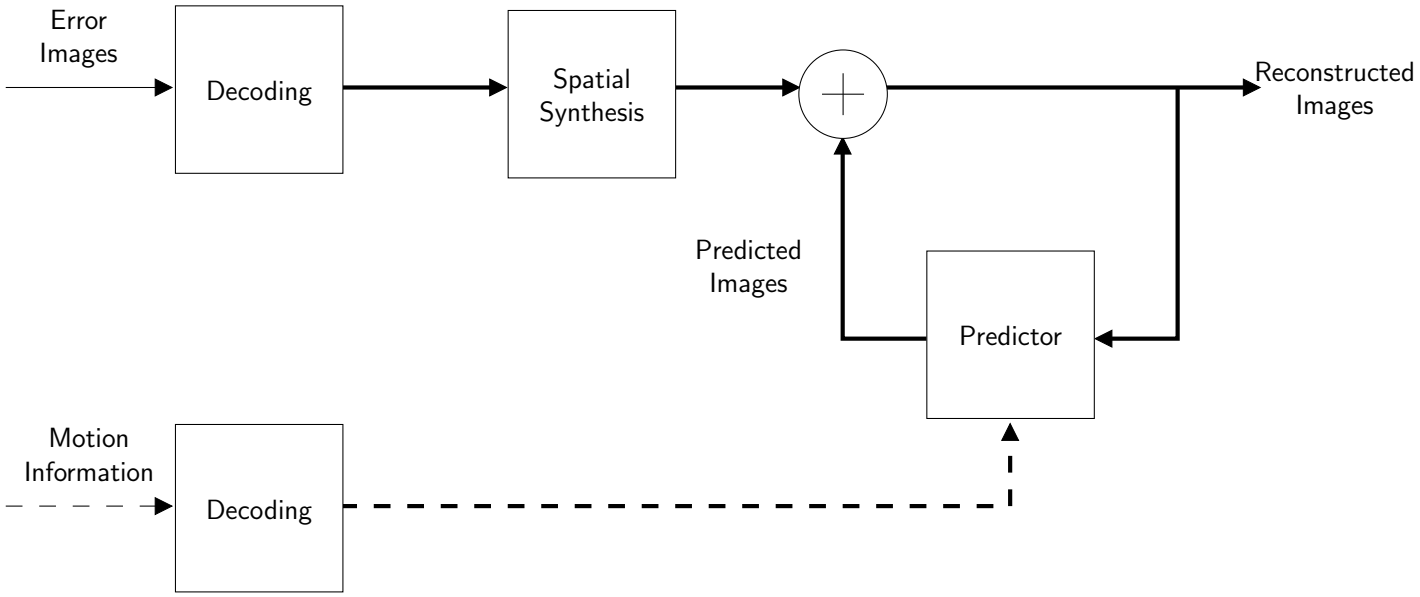


- La solución más eficiente cuando se conoce el bit-rate al que van a ser descomprimidas las imágenes.

29.3. Open Loop t+2D Coding

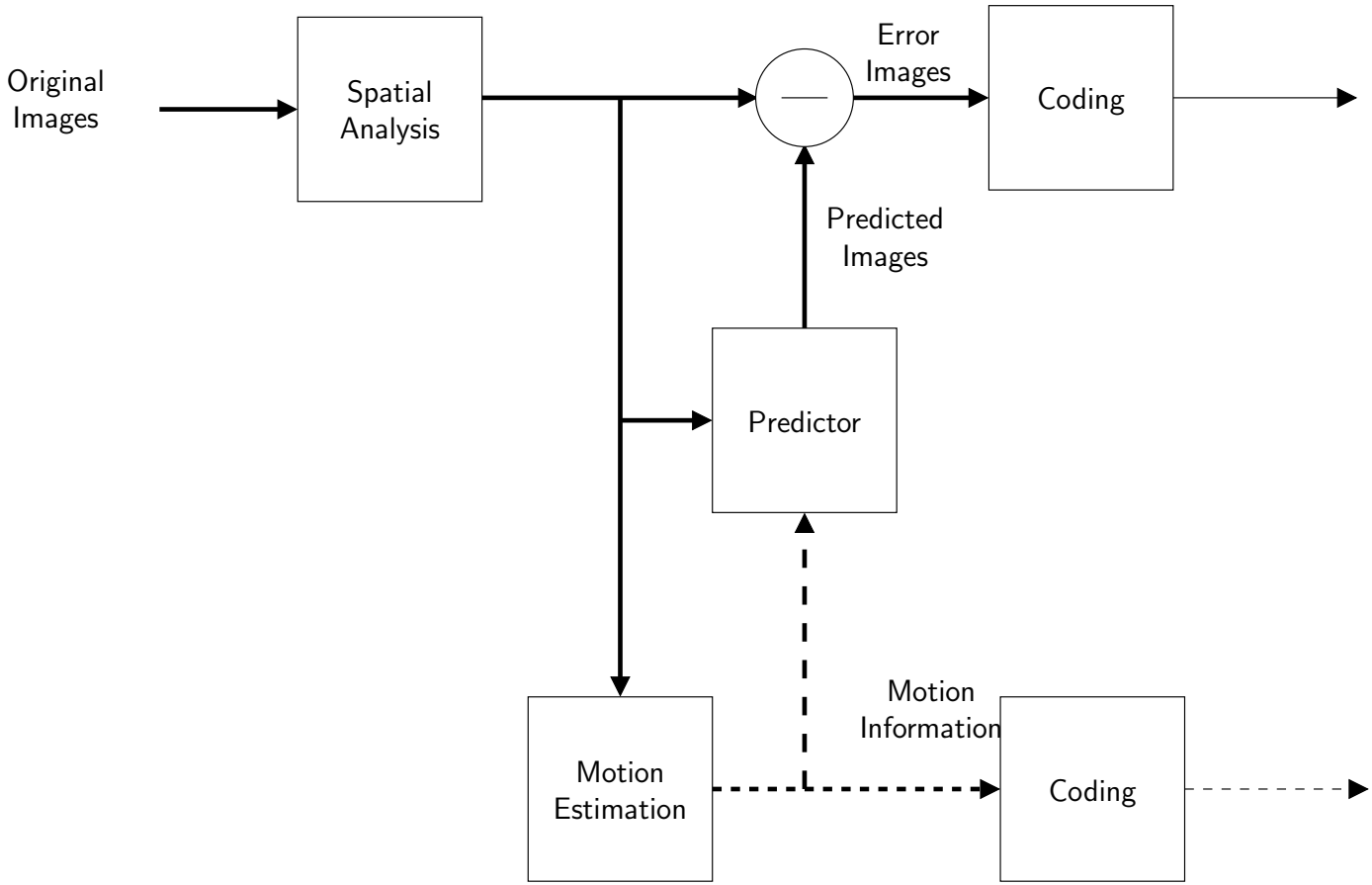
- Usado cuando no es conocido en tiempo de compresión la tasa de bits recuperada por el descompresor.

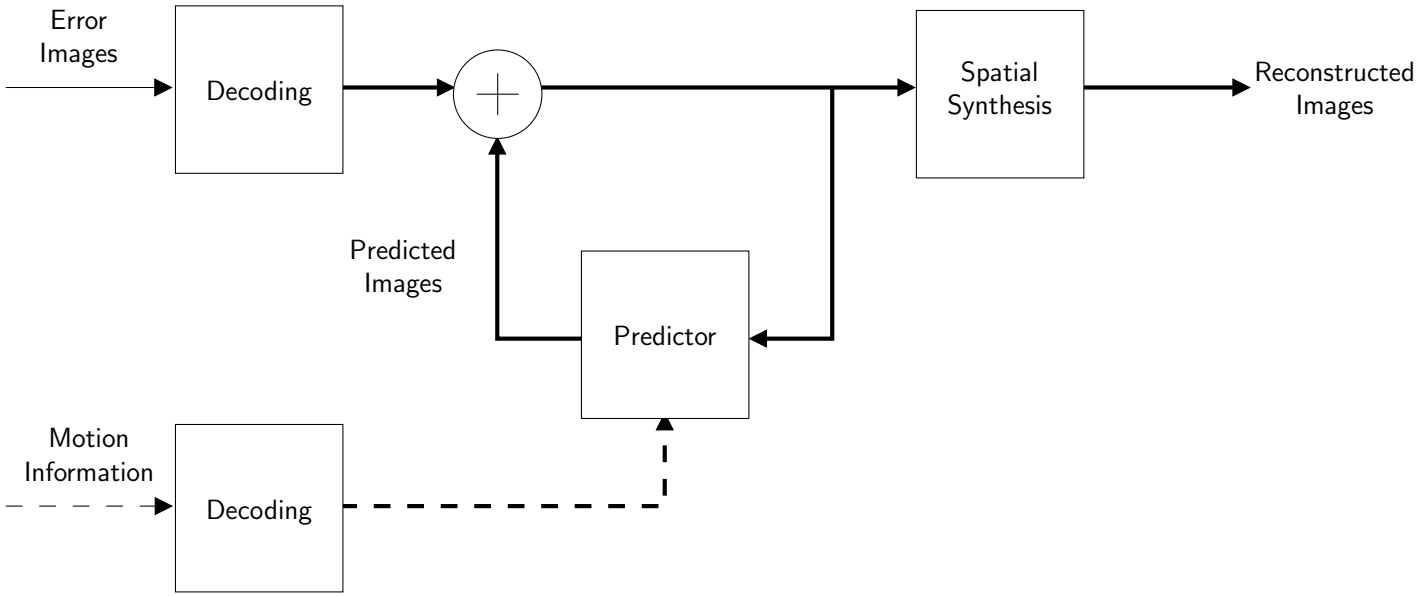




- Nótese que el descodificador es idéntico al de Closed Loop t+2D Coding.

29.4. Open Loop 2D+t Coding





- Similar a t+2D, pero la compensación del movimiento (resta de las imágenes de predicción) se realiza en el dominio transformado. Este hecho puede generar que algunas formas de escalabilidad en el descompresor (como la espacial) sea más eficiente de obtener que en el caso t+2D.

Capítulo 30

MPEG-1 (ISO/IEC 11172) [7]

30.1. Fundamentos

- MPEG-1 utiliza básicamente dos técnicas para eliminar la redundancia temporal y espacial:

  1. **Técnicas semejantes a las usadas en JPEG:** se utilizan para eliminar la redundancia espacial generada porque los puntos vecinos están correlacionados en valor.
  2. **Compensación de movimiento:** se emplea para eliminar la redundancia temporal. La compensación de movimiento permite generar, a partir de las imágenes ya codificadas, una imagen (compensada) que es muy semejane a la que se está intentando comprimir en ese momento. A continuación se resta esta imagen a la compensada y se obtiene una imagen residuo más comprimible con JPEG.

30.2. Bit-rate típico

- MPEG-1 (Moving Picture Experts Group) comprime una secuencia como la anterior sin una pérdida aparente de calidad a una tasa de bits típica igual a la de un reproductor de CD audio:

$$(16+16)\frac{\text{bits}}{\text{muestra}} \times 44.100\frac{\text{muestras}}{\text{segundo}} = 1.411.200\frac{\text{bits}}{\text{segundo}}$$

  Por tanto, la ganancia en compresión es aproximadamente (para este ejemplo en concreto) 180:1.

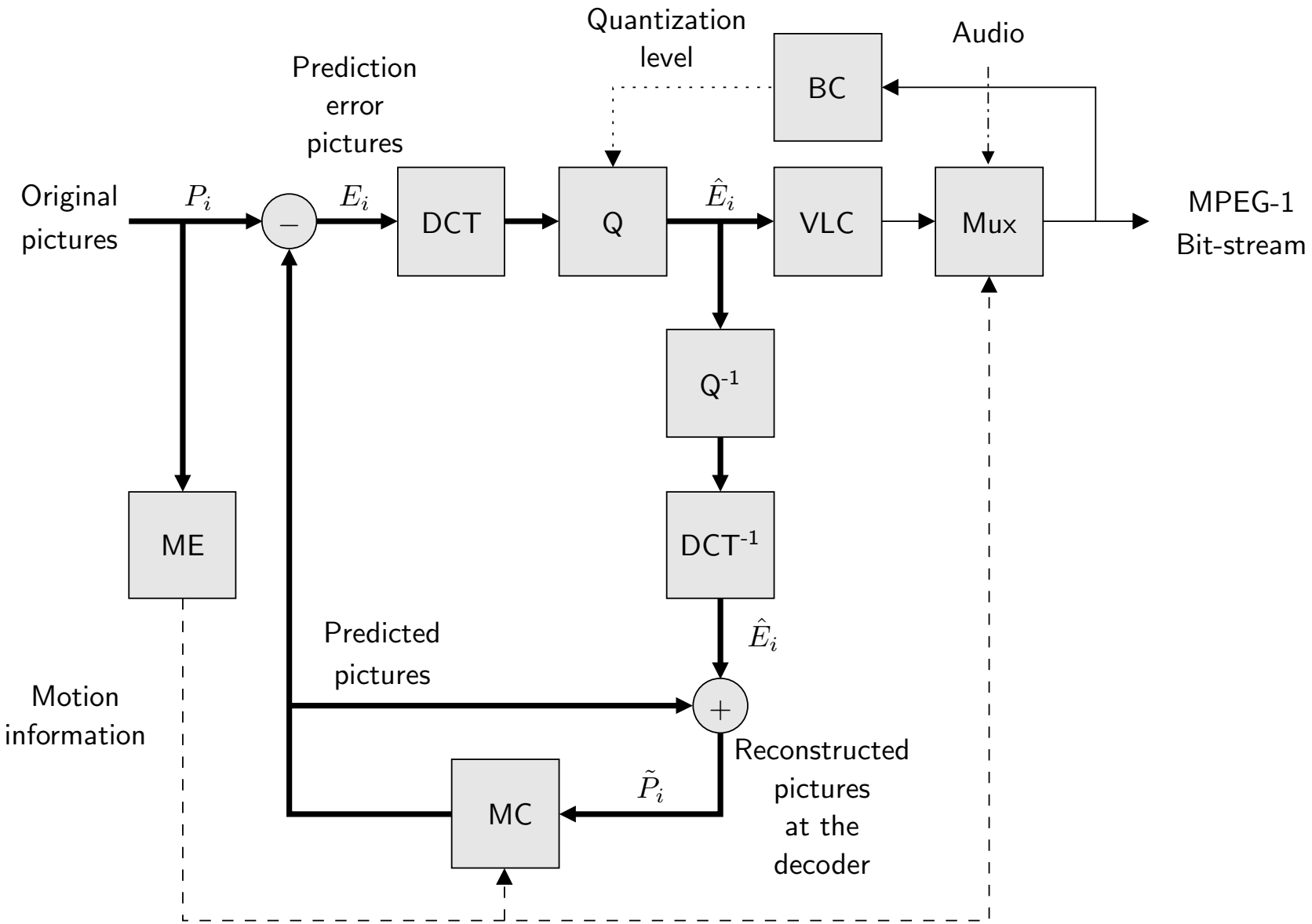
30.3. Posibilidades

- El estándar MPEG-1 (formalmente llamado "Coding of moving pictures and associated audio for digital storage media at up to about $1{,}5$ Mbit/s") [7] define cómo debe descomprimirse un bit-stream MPEG-1, no cómo debe comprimirse (aunque sí que aporta una propuesta).

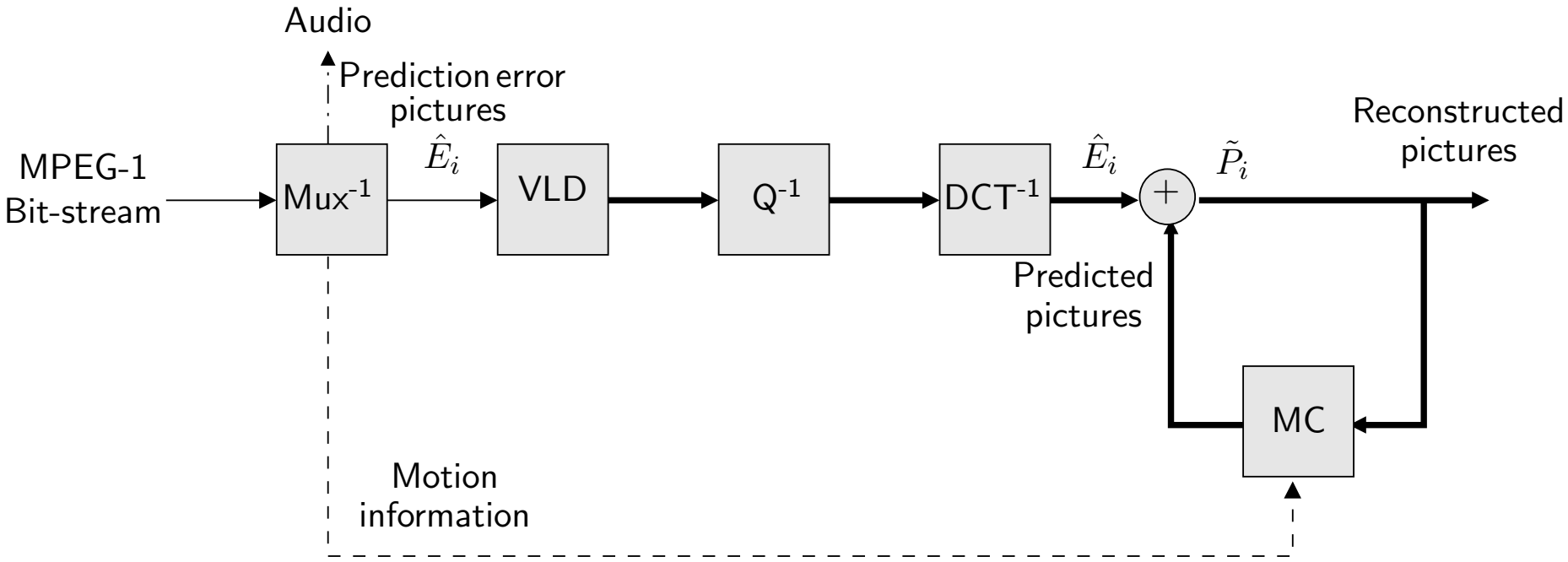
- Otros parámetros importantes del estándar son:

Non-interlazed video format	
Horizontal picture size	≤ 768 points (best 352)
Vertical picture size	≤ 576 points (best 288)
Picture-rate	$23{,}976$, 24, 25, $29{,}97$, 30, 50, $59{,}95$ y 60 Hz
Bit-rate	$\leq 1.856.000$ bps
Chrominance sub-sampling	4:2:0 (Y,Cb,Cr) (ver la Sección 27.3)

30.4. El compresor MPEG-1



30.5. El descompresor MPEG-1



- Nótese que el codec queda descrito por las siguientes recurrencias:

$$\begin{aligned}
&\tilde{P}_i = \mathrm{DCT}^{-1}\big(\mathrm{Q}^{-1}(\hat{E}_i)\big) + \mathrm{MC}(\tilde{P}_{i-1});\ \text{con } i > 0 \text{ donde} \\
&\hat{E}_i = \mathrm{Q}\big(\mathrm{DCT}\big(P_i - \mathrm{MC}(\tilde{P}_{i-1})\big)\big) \text{ siendo} \\
&\mathrm{MC}(\tilde{P}_0) = \mathbf{0}.
\end{aligned} \tag{30.1}$$

La imagen $\mathbf{0}$ representa una imagen donde todos sus puntos son 0.

El subíndice i, recorre la secuencia que no necesariamente tiene que estar ordenada en el tiempo.

30.6. Las etapas DCT, Q, BC y VLC

- MPEG-1 utiliza básicamente las mismas técnicas que JPEG para comprimir las imágenes residuo:
  - **DCT (Discrete Cosine Transform):** es la 2D-DCT y elimina la correlación espacial por bloques de 8×8 puntos.
  - **Q (Quantization):** básicamente elimina los planos de bits menos significativos de los coeficientes DCT. En ellos tiende a acumularse el ruido de las imágenes y además contienen información visualmente poco relevante.
  - **VLC (Variable Length Coding):** al igual que en JPEG, los coeficientes DCT cuantificados se recorren en zig-zag y se comprimen usando un código estático de Huffman.
- La cantidad de información eliminada por Q depende del bit-rate de salida y del deseado por el usuario. La etapa **BC (Bit-rate Control)** se encarga de que a la salida no se generen más datos de los permitidos.

30.7. La etapa ME

- Debido a su sencillez y eficiencia, la *estimación de movimiento* o ME (*Motion Estimation*) es la técnica más utilizada (no sólo en MPEG-1) para reducir la redundancia temporal en las secuencias de vídeo digitales.
- Sólo se realiza en el compresor porque, como veremos, es un proceso costoso en términos de operaciones artiméticas. Esto responde a la necesidad de "comprimir una vez, descomprimir muchas".

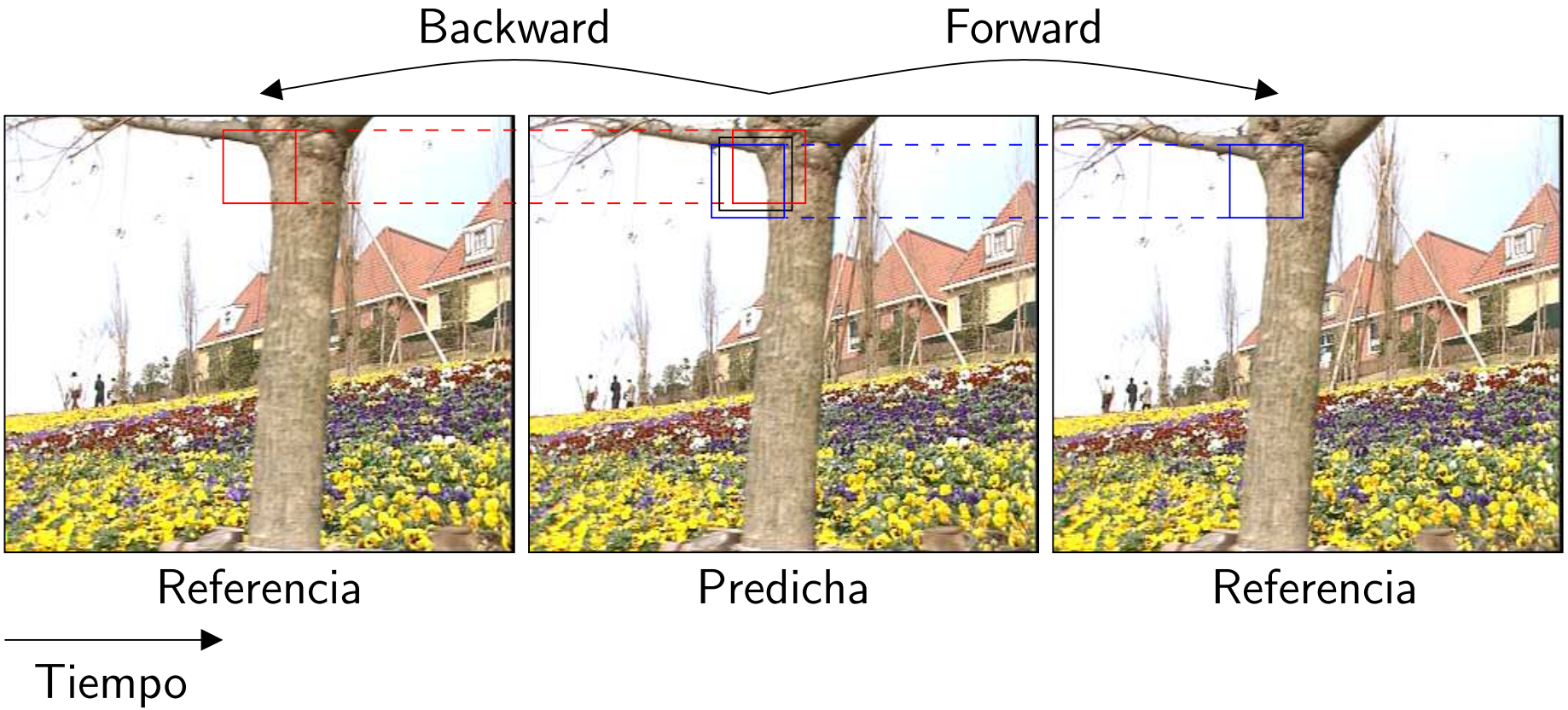
30.8. Imagenes de referencia, predicción y predicha

- ME se basa en que una imagen de la secuencia (*imagen predicha*) puede codificarse a partir de otra (*imagen de referencia*) con muy poco error si somos capaces de encontrar una proyección (*imagen predicción*) de la imagen de referencia que se asemeje mucho a la imagen predicha.

30.9. Estimación hacia delante, hacia detrás y bi-direccional

- Dependiendo de la posición relativa (en el tiempo) de la(s) imágen(es) de referencia y de la imagen predicha, hablaremos de:

  1. **Predicción hacia delante (forward):** cuando la imagen de referencia es posterior a la imagen predicha.
  2. **Predicción hacia detrás (backward):** cuando la imagen de referencia es anterior a la imagen predicha.
  3. **Predicción bi-direccional (bi-directional):** cuando la imagen predicha está entre dos imágenes de referencia.

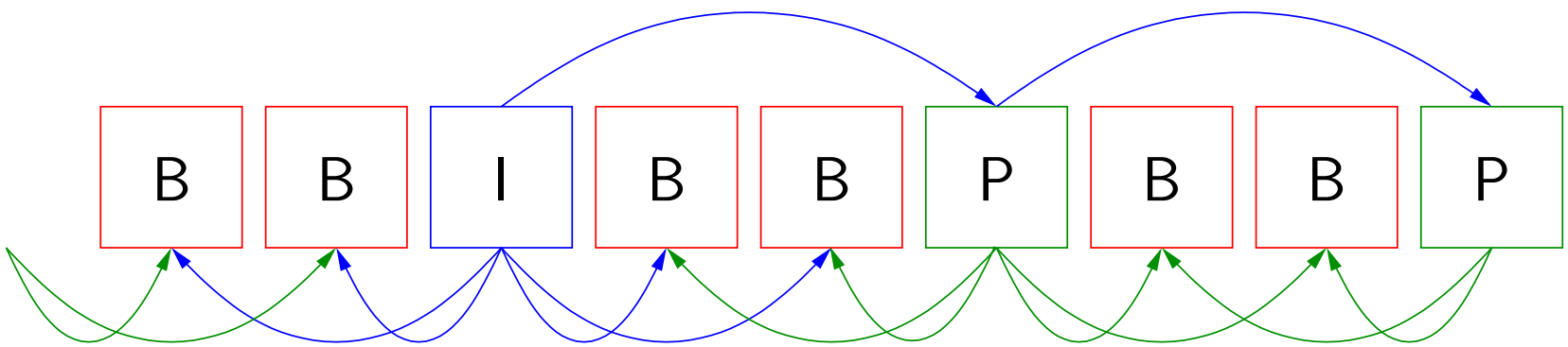


30.9 Estimación hacia delante, hacia detrás y bi-direccional

30.10. Imágenes I, P y B

- Dependiendo de la forma en que se estima una imagen, MPEG-1 distingue entre:

  1. **I-pictures (Intra-coded):** si la imagen no es estimada a partir de ninguna otra.
  2. **P-pictures (Predictive-coded):** si la imagen es estimada a partir de UNA imagen de predicción I o P, anteriores en el tiempo.
  3. **B-pictures (Bidirectionally predictive-coded):** cuando la imagen predicción se genera a partir de dos imágenes de referencia I o P, necesariamente una anterior y otra posterior. Las imágenes B nunca se toman como imágenes de referencia.



30.11. El GOP (Group Of Pictures)

- En MPEG-1, un GOP es un conjunto de imágenes que son continuas en el tiempo. Se cumple que:

  1. Todo GOP debe poseer al menos una imagen de tipo I. Estas imágenes suelen colocarse periódicamente en el tiempo (aproximadamente cada $0{,}5$ segundos), y generalmente es la primera del GOP.
  2. Su longitud no está limitado (excepto por el número de imágenes de la secuencia). Sin embargo, tamaños superiores a 16 no son frecuentes para facilitar el acceso aleatorio y el control de errores en el bit-stream.
  3. Su composición, en términos de imágenes I, P y B, puede cambiar a lo largo de la secuencia comprimida. El MPEG aconseja que ocurran 1 I-picture por cada 3 P-pictures y de 2 a 5 B-pictures por cada P-picture.

4. Puede existir más de una imagen I por GOP.
5. Algunos ejemplos de GOP*:

```
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
----------------------------> time
I
I P P
I B P B P
B B I B P B P
B B I B B P B B P B B P
B I B B B B P B I B B I I
I B B P B B P B B P B B P
B B I B B P B B P B B P B B P
```

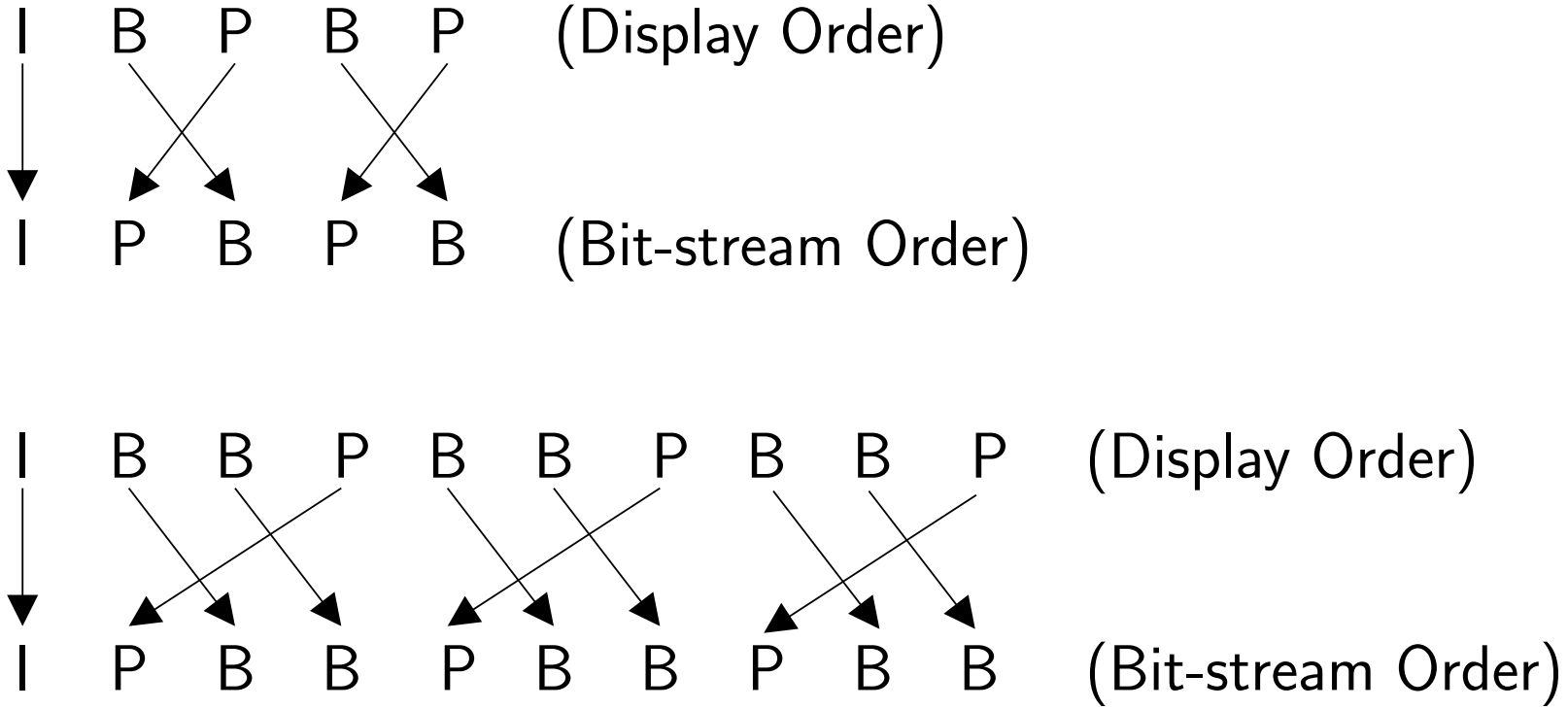
*Siempre en "display order".

30.12. Display (time) and bit-stream orders

- Cuando existen imágenes de tipo B en un GOP, en MPEG-1 se habla de dos tipos de ordenación de imágenes:

  1. **Display order:** es aquel en el que las imágenes necesitan ser mostradas en el visualizador. Ningún GOP en este orden puede comenzar en un P-picture o acabar en un B-picture.
  2. **Bit-stream order:** es aquel en el que las imágenes necesitan ser colocadas en el bit-stream. En este se cumple que todo GOP comienza por un I-picture.

- Estas ordenaciones son diferentes sólo cuando aparecen imágenes B. Esto es así porque como estas imágenes dependen de la siguiente I o P (en display order), dicha imágen I o P debe ser transmitida antes que la B.


I B P B P (Display Order)
I P B P B (Bit-stream Order)
I B B P B B P B B P (Display Order)
I P B B P B B P B B (Bit-stream Order)

30.13. GOP's abiertos y cerrados

- En MPEG-1 se habla de:

  1. **GOP cerrado (closed)** cuando puede descodificarse (y visualizarse) de forma independiente a cualquier otro GOP. En display order, los GOP's cerrados comienzan necesariamente por un I-picture.
  2. **GOP abierto (open)** cuando necesita el GOP anterior para descodificarse. Los GOP's abiertos comienzan siempre en un B-picture, en display order. Estos GOP's se utilizan rara vez porque dificultan el acceso aleatorio a las imágenes del vídeo.

     `I B B P B B P B B P B B P` (closed GOP)
     `B B I B B P B B P B B P B B P` (open GOP)

30.14. Estimación de movimiento basada en la búsqueda de bloques

- En MPEG-1 se utiliza una técnica de estimación de movimiento llamada *búsqueda de bloques* (block matching) que consiste en dividir la imagen de referencia** en bloques y en encontrar dónde se sitúan en la imagen predicha. MPEG-1 denota a los bloques *macrobloques* (*macroblocks*). Estos son siempre disjuntos y de un tamaño de 16×16 puntos.
- Asociado a cada macrobloque hay un vector de movimiento que indica a dónde se ha desplazado el macrobloque en la imagen predicha.
- Existen otros particionamientos de las imágenes por triángulos, objetos, puntos, ... Sin embargo estas técnicas son menos usadas por su mayor coste computacional.
- En la estimación de movimiento se puede utilizar una precisión de

**Nótese que también hubiera podido dividirse la imagen predicha.

1 punto o de $1/2$ de punto. En este caso, tanto el macrobloque de referencia como el predicho deben interpolarse según:

A	X_1	B
X_2	X_3	X_4
C	X_5	D

$$X_1 = (\mathsf{A} + \mathsf{B})/2$$

$$X_2 = (\mathsf{A} + \mathsf{C})/2$$

$$X_3 = (\mathsf{A} + \mathsf{B} + \mathsf{C} + \mathsf{D})/4$$

30.15. Matching criteria

- MPEG-1 propone para saber cómo de parecidos son dos macrobloques a y b, las siguientes medidas:

  1. **El error cuadrático medio:**

$$\frac{1}{16 \times 16} \sum_{i=1}^{16} \sum_{j=1}^{16} (a_{ij} - b_{ij})^2$$

  2. **El error absoluto medio:**

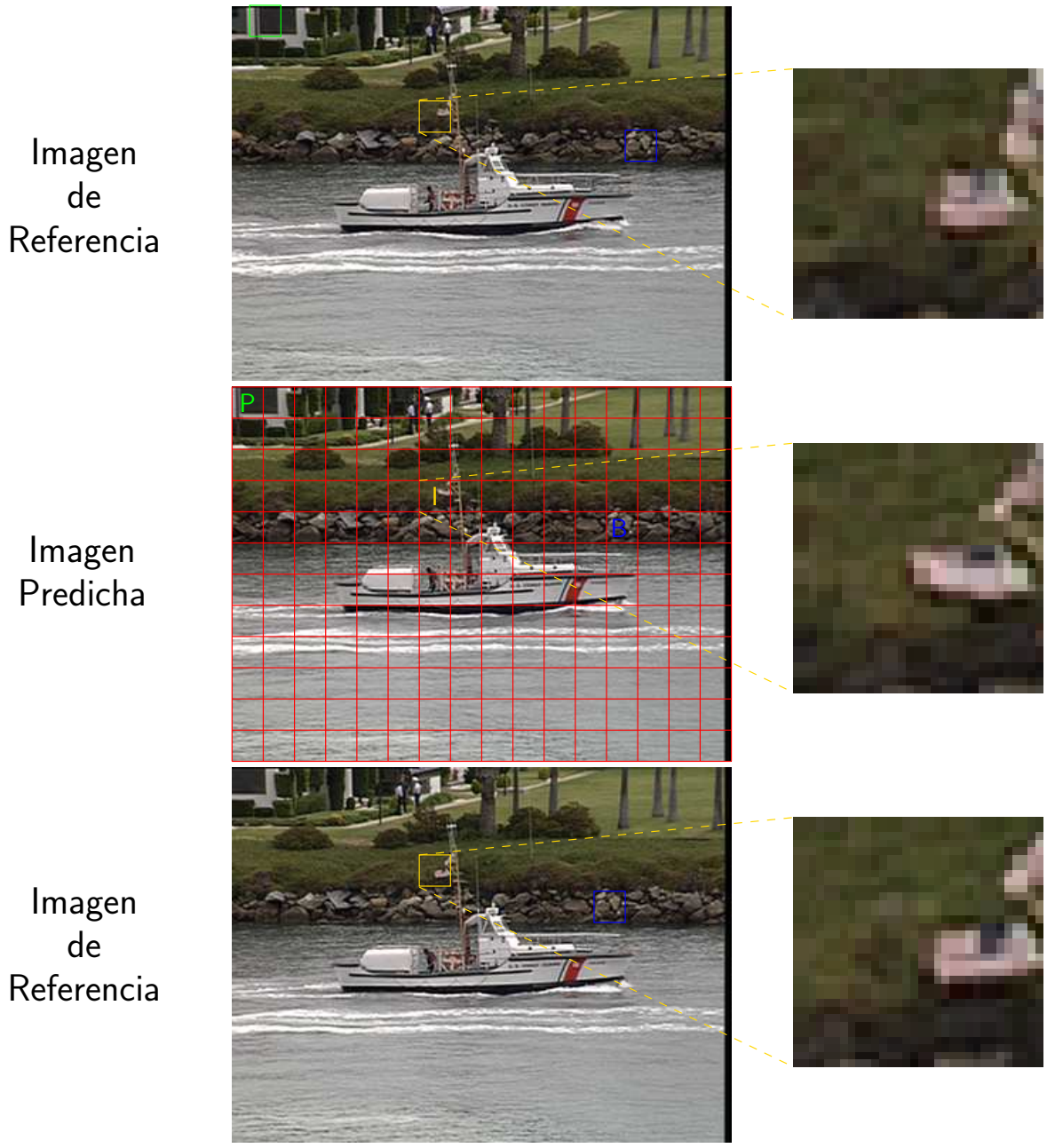
$$\frac{1}{16 \times 16} \sum_{i=1}^{16} \sum_{j=1}^{16} |a_{ij} - b_{ij}|$$

- Sin embargo, debe tenerse en cuenta que esto es únicamente usado por el compresor y éste no está estandarizado. Por tanto, cualquier otra medida (como la varianza o la entropía del macrobloque residuo) podría se utilizada.

30.16. Tipos de macrobloques

- Dependiendo del parecido que existe entre los bloques predicción y predichos, MPEG-1 distingue entre cuatro tipos de macrobloques:

  1. **Macrobloques I:** aquellos que no se predicen porque no existe una suficiente similitud entre bloques. Se encuentran en las imágenes I, P y B.
  2. **Macrobloques P:** son los que se predicen a partir de una imagen I o P, anterior o posterior. Se encuentran en las imágenes P y B.
  3. **Macrobloques B:** los que se predicen a partir de dos imágenes I o P, simultáneamente. Sólo se encuentran en las imágenes B. El macrobloque predicción se calcula como la media aritmética punto a punto de ambas predicciones (la backward y la forward).
  4. **Macrobloques S (Skipped):** cuando el error de predicción es muy pequeño, el compresor no envía residuo. Los macrobloques S pueden aparecer en las imágenes P y B.

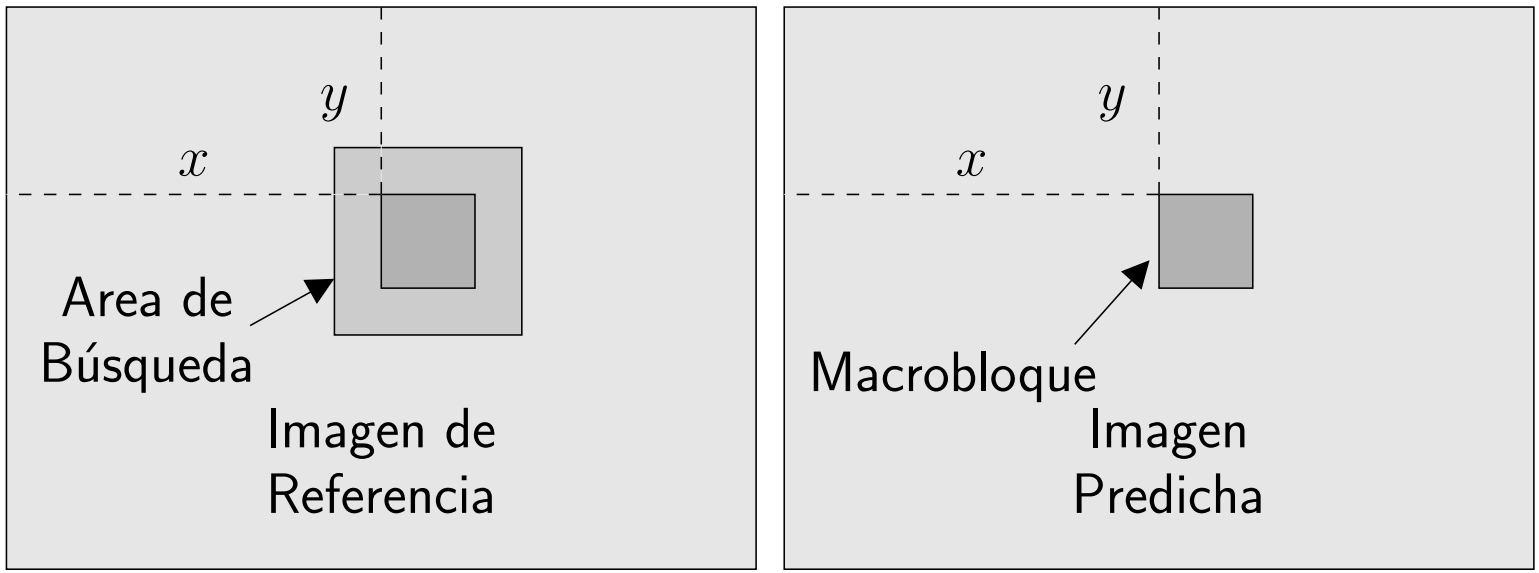


30.16 Tipos de macrobloques

30.17. Estrategias de búsqueda

- En MPEG-1 se proponen una serie de algoritmos de búsqueda de bloques que poseen diferentes demandas computacionales y resultados:

  1. **Full search (búsqueda exhaustiva):** consiste en buscar el macrobloque de referencia por todo el área de búsqueda y en quedarnos con la mejor coincidencia. Ventaja: optimalidad. Desventaja: muy costoso, especialmente cuando el área de búsqueda es grande.



2. **Logaritmic search:** es semejante a la búsqueda exhaustiva, excepto porque se realiza con versiones submuestreadas del macrobloque de referencia y del área de búsqueda. Inicialmente se utiliza una versión que involucra a muy pocos puntos y cuando se obtiene la mejor coincidencia, se duplica el muestreo y se busca en las proximidades de la última coincidencia. Este proceso continúa hasta alcanzar la resolución original o 1/2 puntos.
3. **Telescopic search:** consiste en usar los vectores de movimiento calculados para las imágenes vecinas (en el tiempo) para restringir el área de búsqueda. Esto suele funcionar bastante bien porque los campos de movimiento vecinos tienden a estar bastante correlacionados.

30.18. La etapa MC

- La *compensación de movimiento* o MC (*Motion Compensation*) es el proceso de generar la imagen predicción a partir de la información de movimiento (generalmente en forma de vectores de movimiento). Dicha imagen debería ser muy parecida a la imagen predicha.
- Para generar la predicción, en MPEG-1 se utilizan las imágenes de referencia que tiene el descompresor y los campos de movimiento que indican el desplazamiento relativo de cada macrobloque de la imagen predicha en las imágenes de referencia.

30.19. Codificación de los campos de movimiento

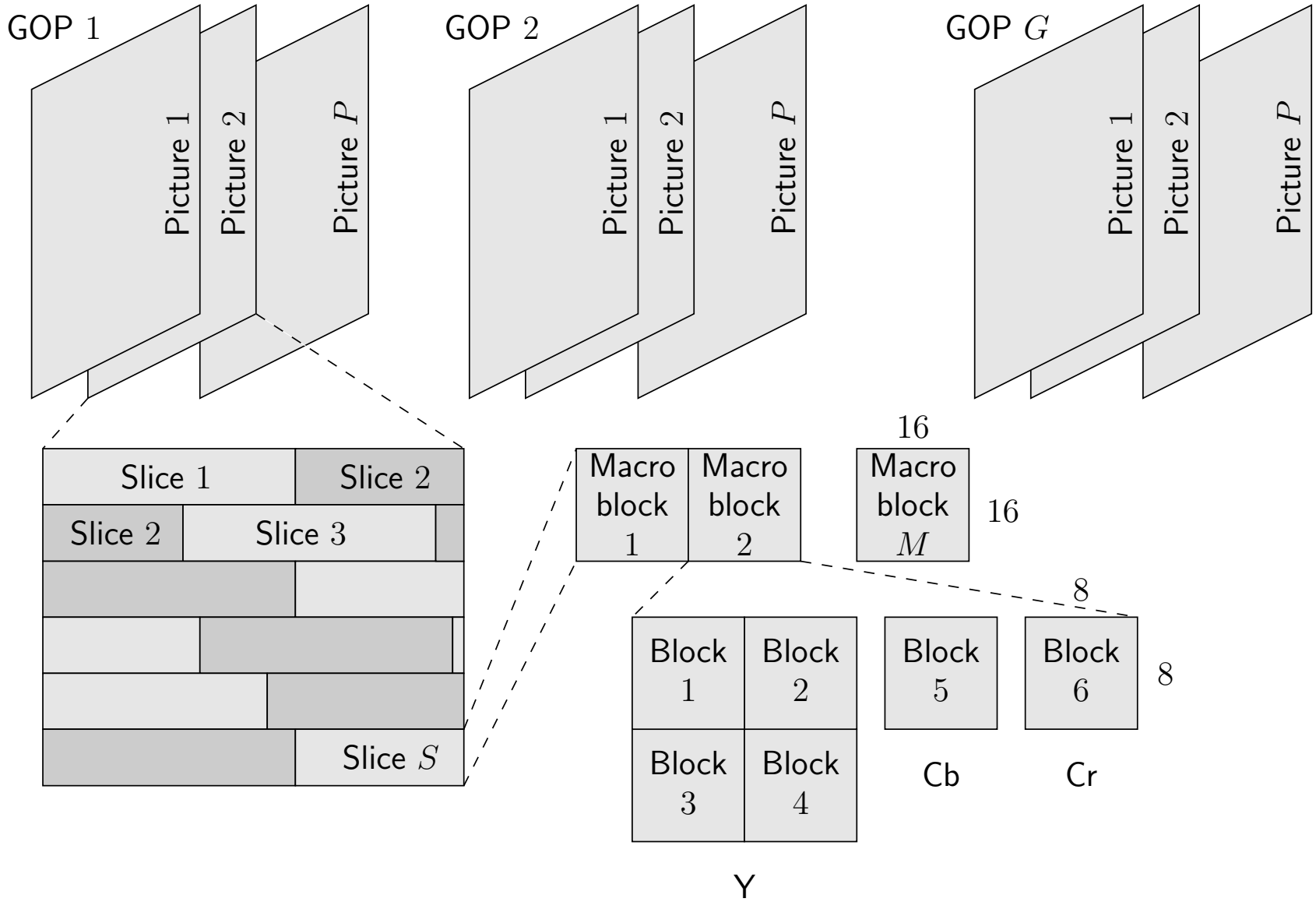
- MPEG-1 codifica los campos de movimiento (pares de coordenadas (x, y)) mediante un código de Huffman estático.
- La distribución de los valores x e y sigue una distribución de Laplace centrada en el valor 0. Esto es así porque en general los desplazamientos de los macrobloques entre imágenes adyacentes suelen ser bastante pequeños.
- El estándar no contempla desplazamientos superiores a 16 puntos:

Motion VLC code	offset
0000 0011 001	-16
0000 0011 011	-15
0000 0011 101	-14
0000 0011 111	-13

0000 0100 001	-12
0000 0100 011	-11
0000 0100 11	-10
0000 0101 01	-9
0000 0101 11	-8
0000 0111	-7
0000 1001	-6
0000 1011	-5
0000 111	-4
0001 1	-3
0011	-2
011	-1
1	0
010	1
0010	2
0001 0	3

0000 110	4
0000 1010	5
0000 1000	6
0000 0110	7
0000 0101 10	8
0000 0101 00	9
0000 0100 10	10
0000 0100 010	11
0000 0100 000	12
0000 0011 110	13
0000 0011 100	14
0000 0011 010	15
0000 0011 000	16

30.20. The MPEG-1's data partitioning



- Cada slice está identificado en el bit-stream con un código de comienzo. Así, cuando el descompresor "pierde el hilo" de la descodificación debido a un error en el bit-stream, se puede sincronizar en el siguiente slice. De esta manera, un error en un bit no afecta a un Picture o a un GOP completo, sino a un slice.
- El tamaño mínimo de un slice puede ser un macrobloque y el tamaño máximo el de un picture.

30.21. Visualización rápida

- MPEG-1 contempla un tipo especial de I-pictures llamadas D-pictures en las que sólo existen información sobre el coeficiente DC de las imágenes residuo. Estas imágenes se descomprimen muy rápidamente[***] y es posible realizar una visualización rápida hacia delante y hacia detrás de la secuencia comprimida.

- Las D-pictures son imágenes extra insertadas en el bit-stream y por tanto aumentan la tasa de bits. Por este motivo y porque a partir de una imagen I (con algo más de trabajo) se puede obtener el mismo resultado, su uso es bastante infrecuente.

[***] No hay que realizar la DCT^{-1} porque el coeficiente DC es la media del bloque.

30.22. Ejemplos de compresión

- Para ver los resultados que general el codec MPEG-1, se ha comprimido un conjunto de vídeos. Las tasas de compresión, aunque no son exáctas (por problemas del codec de vídeo) han sido: $1.200, 600, 300$ y 100 Kbps.
- Búsquese los ficheros `*MPEG-1*` en el directorio `vids`.

Capítulo 31

MPEG-2 (ISO/IEC 13818) [6]

Introducción

- Nombre exácto del estándar: "Generic Coding of Moving Pictures and Associated Audio".
- Disponible desde 1994 y masivamente utilizado en DVD's, televisión por satélite y TDT.
- Audio multicanal e imágenes progresivas y entrelazadas.
- Muy semejante a MPEG-1, pero que es capaz de trabajar con bit-rates de hasta 100 Mbps.
- Compatible con MPEG-1.

31.1. Motivación

Mejora a MPEG-1 en:

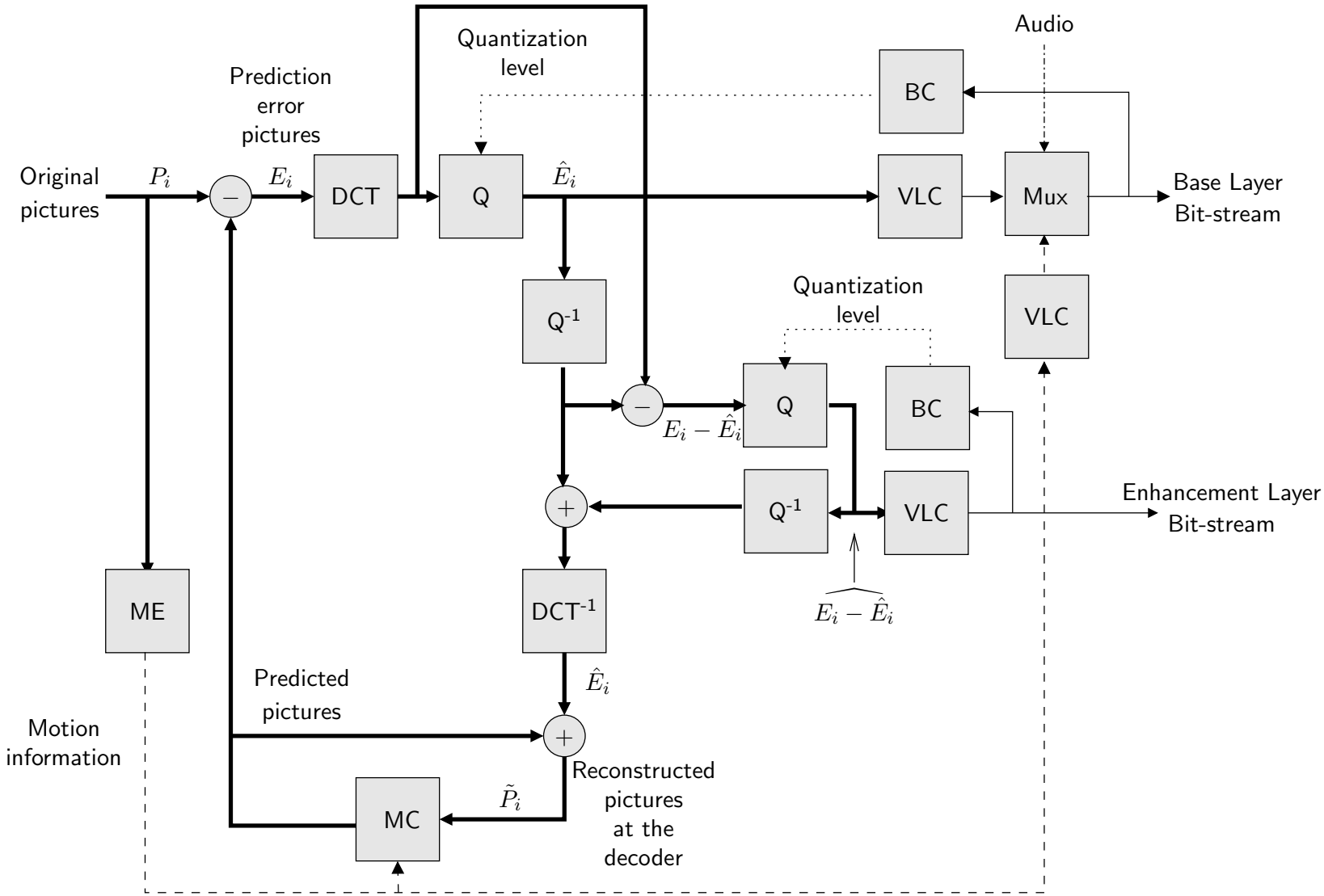
- Más resolución espacial (HDTV): 704x480, 1280x720 y 1920x1080.
- Más resistencia a errores de transmisión (Satélite y TDT).
- Soporte para vídeo entrelazado.

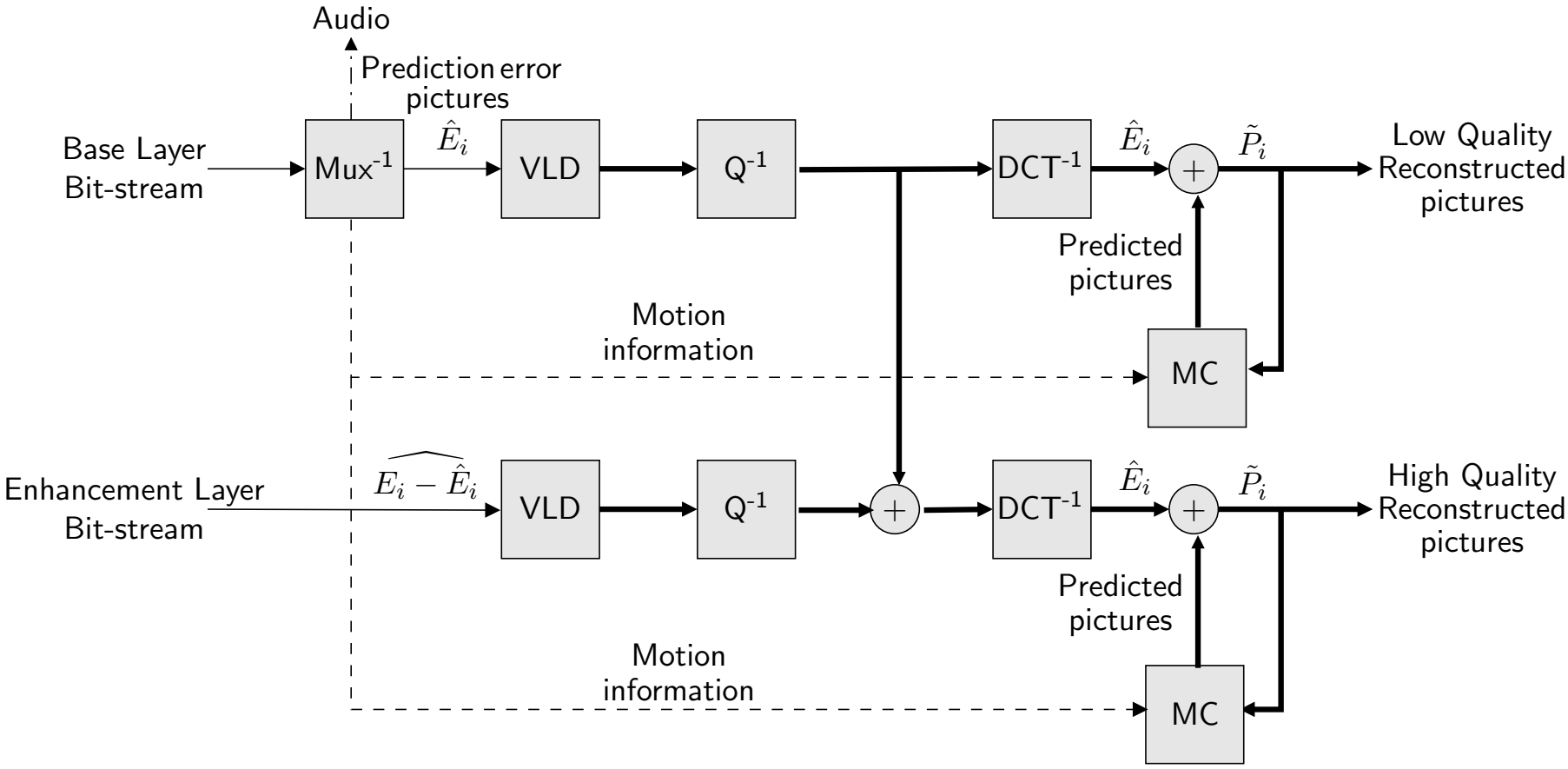
31.2. Bit-rates típicos

- MPEG-2 propone una serie de perfiles para codificar vídeo, y dentro de cada perfil, una serie de niveles (aunque esto no significa que pueda trabajar con otras resoluciones y tasas). Algunos ejemplos:

	Simple Profile	Main Profile	4:2:2 Profile	SNR Scal. Profile	Spatial Scal. Profile	High Profile
High Level		4:2:0 1920×1152				4:2:0 o 4:2:2 1920×1152 100 Mbps
High 1140 Level		4:2:0 1140×1152 60 Mbps			4:2:0 1140×1152 60 Mbps	4:2:0 o 4:2:2 1140×1152 80 Mbps
Main Level	4:2:0 720×576 15 Mbps	4:2:0 720×576 15 Mbps	4:2:2 720×576 50 Mbps	4:2:0 720×576 15 Mbps		4:2:0 o 4:2:2 720×576 20 Mbps
Low Level		4:2:0 532×288 4 Mbps		4:2:0 532×288 4 Mbps		

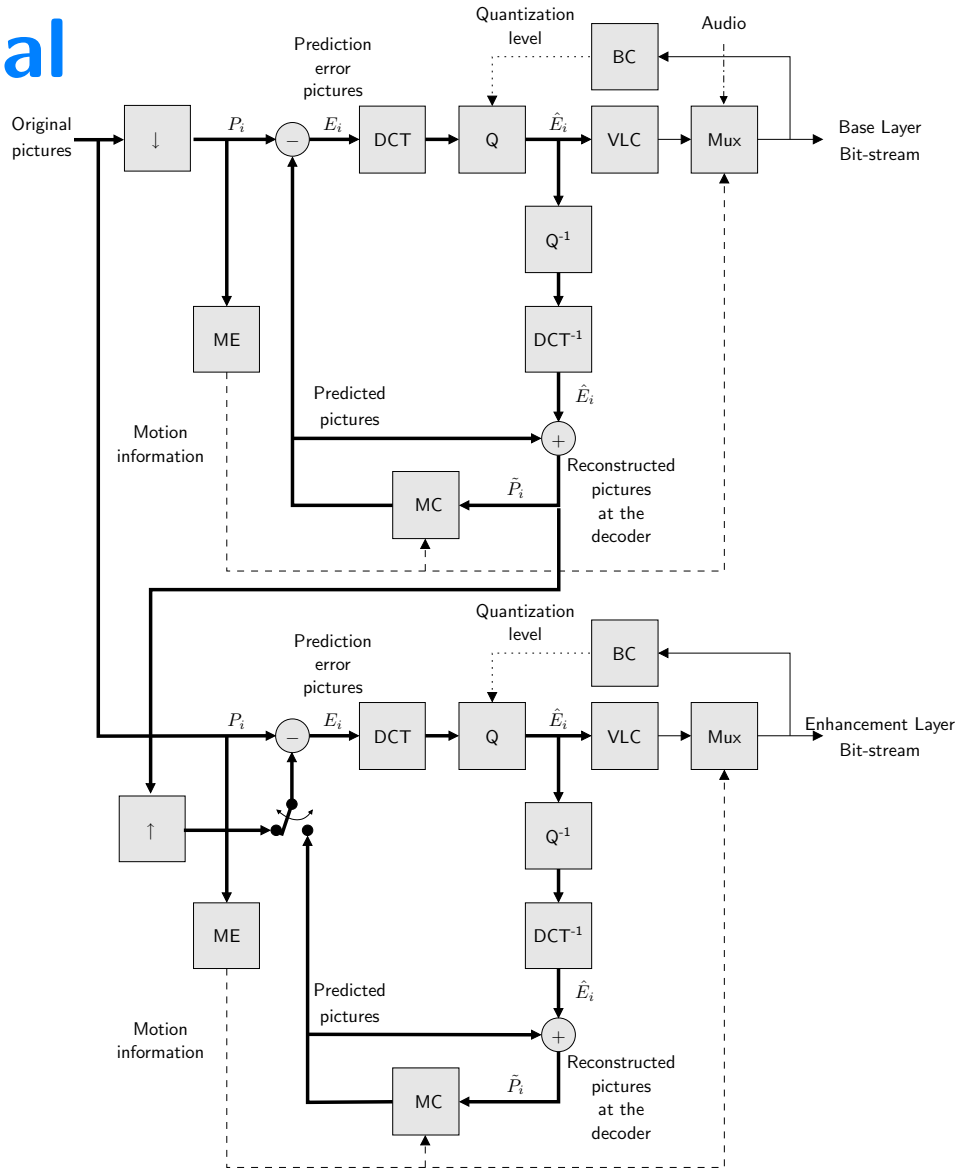
31.3. Codificación escalable en calidad

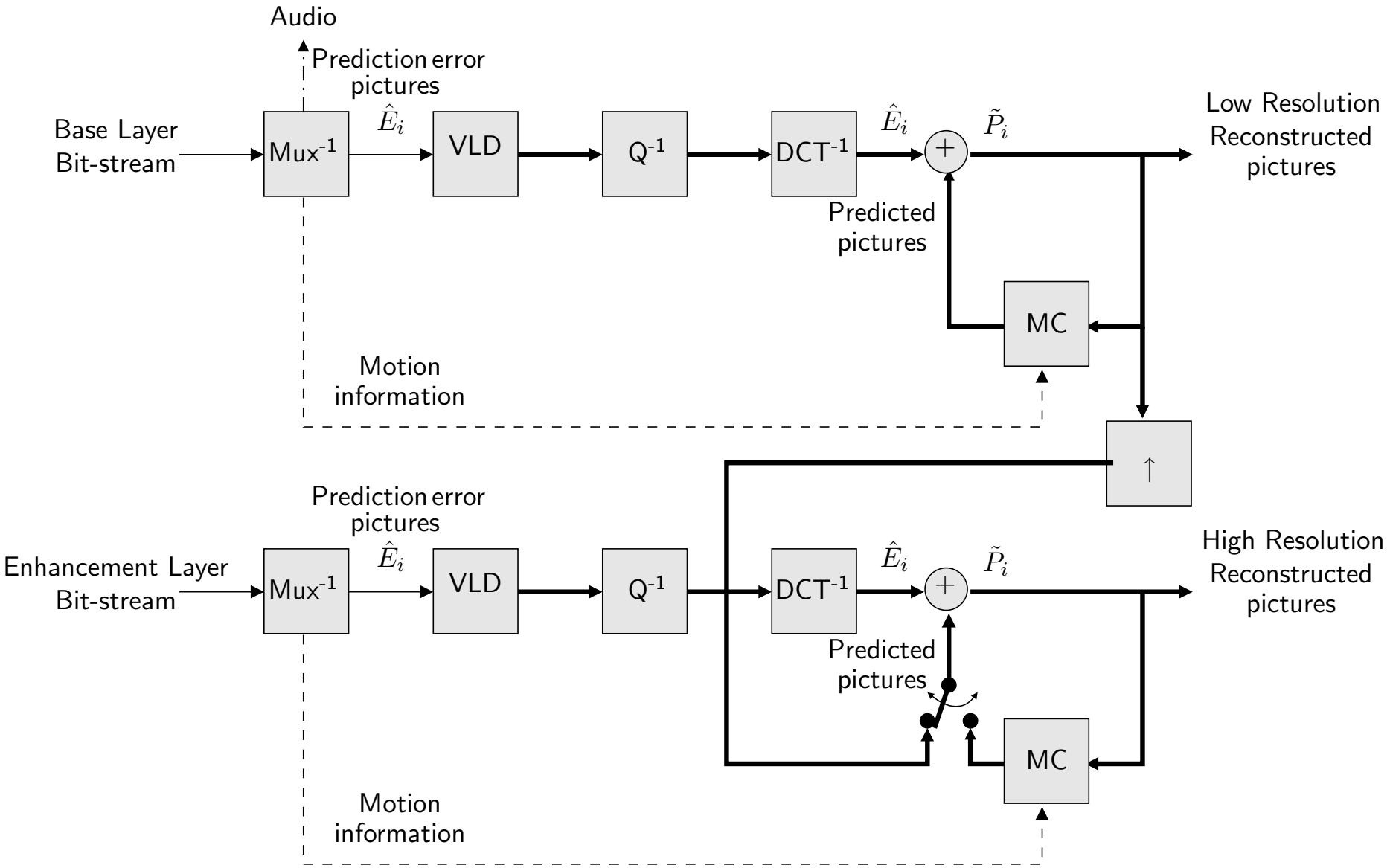




31.3 Codificación escalable en calidad

31.4. Codificación escalable en resolución espacial





31.4 Codificación escalable en resolución espacial

31.5. Codificación escalable en resolución temporal

- Consiste en acceder selectivamente a los distintos "pictures" en función de la posición temporal que ocupan en el GOP. Un ejemplo:

```
I B B B P B B B I B B B P ...
0 : : : : : : : 0 : : : : -> Base Layer (Only I-pics)
  : : : 1 : : :   : : : 1 -> Enhancement Layer (P-pics)
  : 2 :   : 2 :   : 2 :   -> Enhancement Layer (B-pics)
  3   3   3   3   3   3   -> Enhancement Layer (B-pics)
```

31.6. Ejemplos de compresión

- Para ver los resultados que general el codec MPEG-1, se ha comprimido un conjunto de vídeos. Las tasas de compresión, aunque no son exáctas (por problemas del codec de vídeo) han sido: $1.200, 600, 300$ y 100 Kbps.
- Búsquese los ficheros `*MPEG-2*` en el directorio `vids`.

Capítulo 32

MPEG-4 (ISO/IEC 14496) [15]

Introducción

- Nombre del estándar: "Coding of audio-visual objects".
- `http://www.chiariglione.org/mpeg`.
- Disponible desde 1999 y ampliamente utilizado desde la aparición de los DivX.
- Ideado para trabajar a tasas de bit muy bajas (hasta 5 Kbps [13]).
- Basado en las mismas ideas que MPEG-1 y MPEG-2, aunque mejoradas. También define, como MPEG-2, una serie de perfiles para trabajar.
- Un escenario audio-visual en MPEG-4 es un árbol jerárquico de media-objects (video-objects and audio-objects), cada uno de ellos codificado con técnicas que pueden ser diferentes.
- Preparado para trabajar tanto con contenidos naturales como sintéticos (2D y 3D) (a estos, por ejemplo, se les puede alterar su ángulo de visión, cambiar el tipo de voz – hombre/mujer –, etc.).

- Los video-objects se pueden procesar y descomprimir de forma independiente, y la composición de ellos en la escena se puede alterar.
- Incorpora un modo de compensación de movimiento basado en "sprites" donde unos video objects se mueven respecto a otros.
- Video-objects de forma arbitraria (no sólo rectangulares). En este caso se utiliza además un codificador de formas (shap coder) cuyo stream de salida se multiplexa con los streams de textura y audio.
- Uso de video-objects sintéticos (texture+meshes).
- Precisión sub-pixel 1/4 (6-tap filter).
- Macrobloques de 16x16, 16x8, 8x16 y 8x8, 8x4, 4x8 y 4x4.
- Deblocking filters usado para crear las predicciones.
- Entropy coding mejorada: CAVLC (Context Adaptive Variable Length Coding) (Golomb Coding) y CABAC (Context-Based Adaptive Arithmetic Coding) (Arithmetic Coding).

31.6 Ejemplos de compresión

32.1. Bit-rate típicos [16]

Profile	Typical visual session size	Max. number of objects	Max. bit-rate (Kbps)	Max. number of enhancement layers S/T
Simple L0	QCIF	1	64	N.A.
Simple L1	QCIF	4	64	N.A.
Simple L2	CIF	4	128	N.A.
Simple L3	CIF	4	348	N.A.
Adv. Real Time Simple L1	QCIF	4	64	N.A.
Adv. Real Time Simple L2	CIF	4	128	N.A.
Adv. Real Time Simple L3	CIF	4	348	N.A.
Adv. Real Time Simple L4	CIF	16	2000	N.A.
Simple Scalable L1	CIF	4	128	1
Simple Scalable L2	CIF	4	256	1
Core L1	QCIF	4	348	1

Core L2	CIF	16	2000	1
Advanced Core L1	QCIF	4	348	1
Advanced Core L2	CIF	16	2000	1
Core Scalable L1	CIF	4	768	1
Core Scalable L2	CIF	8	1500	1
Core Scalable L3	CCIR 601	16	4000	2
Main L2	CIF	16	2000	1
Main L3	CCIR 601	32	15000	1
Main L4	1920x1088	32	38400	1
Advanced Coding Efficiency L1	CIF	4	348	1
Advanced Coding Efficiency L2	CIF	16	2000	1
Advanced Coding Efficiency L3	CCIR 601	32	15000	1
Advanced Coding Efficiency L4	1920x1088	32	38000	1
N-Bit	CIF	16	2000	1
Advanced Simple L0	QCIF	1	128	1
Advanced Simple L1	QCIF	4	128	1

32.1 Bit-rate típicos [16]

Advanced Simple L2	CIF	4	348	1
Advanced Simple L3	CIF	4	768	1
Advanced Simple L4	CIF	4	3000	1
Advanced Simple L5	CCIR 601	4	8000	1
Fine Granularity Scalable L0	QCIF	1	128	1
Fine Granularity Scalable L1	QCIF	4	128	1
Fine Granularity Scalable L2	CIF	4	348	1
Fine Granularity Scalable L3	CIF	4	768	1
Fine Granularity Scalable L4	CIF	4	3000	1
Fine Granularity Scalable L5	CCIR 601	4	8000	1

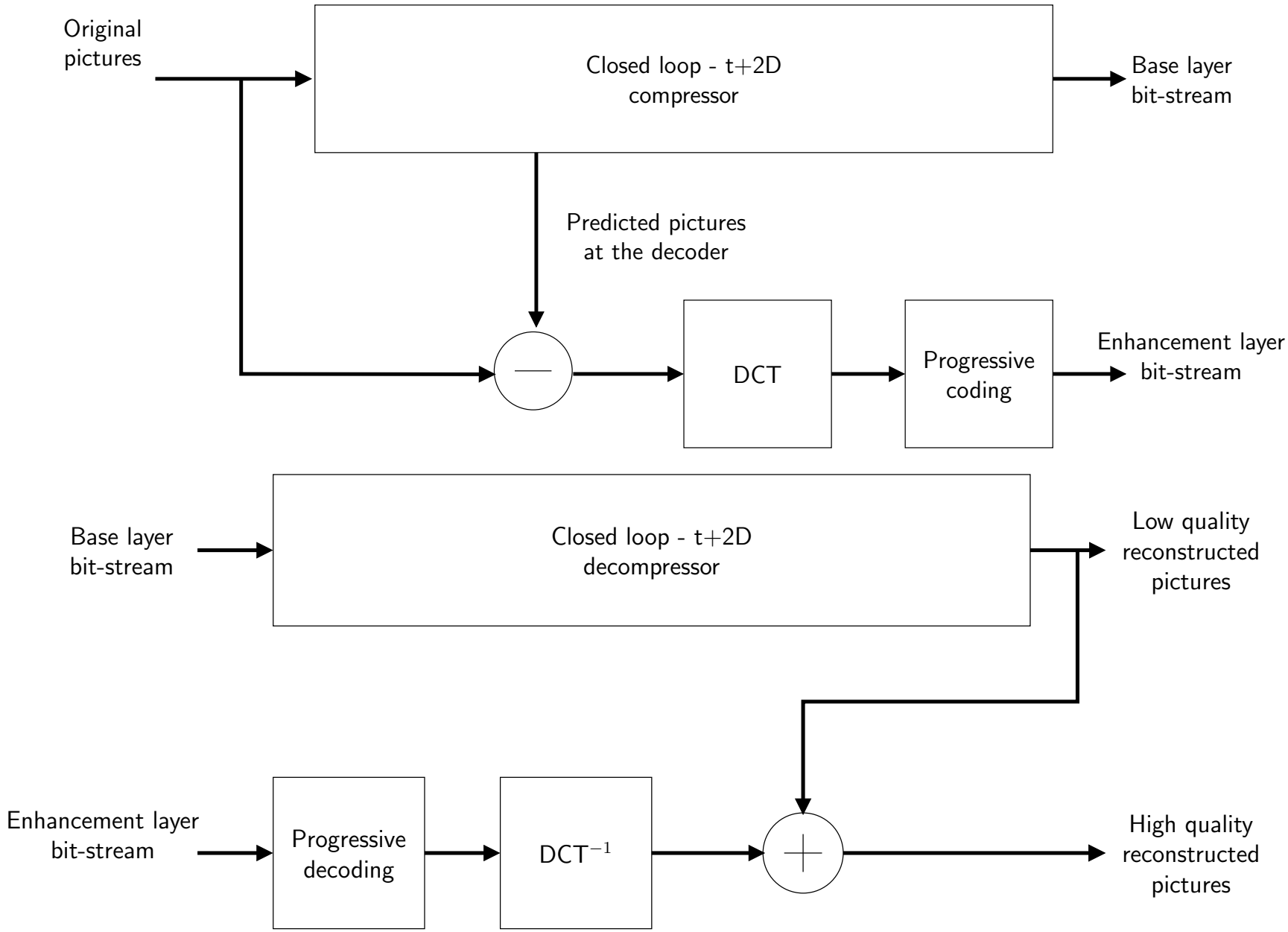
QCIF = 176x144

CIF = 352x288

CCIR 601 = 720x576

S/T = Spatial or Temporal

32.2. Fine Granularity Scalability



32.3. Ejemplos de compresión

- Para ver los resultados que general el codec MPEG-4, se ha comprimido un conjunto de vídeos. Las tasas de compresión, aunque no son exáctas (por problemas del codec de vídeo) han sido: $1.200, 600, 300$ y 100 Kbps.
- Búsquese los ficheros `*MPEG-4*` en el directorio `vids`.

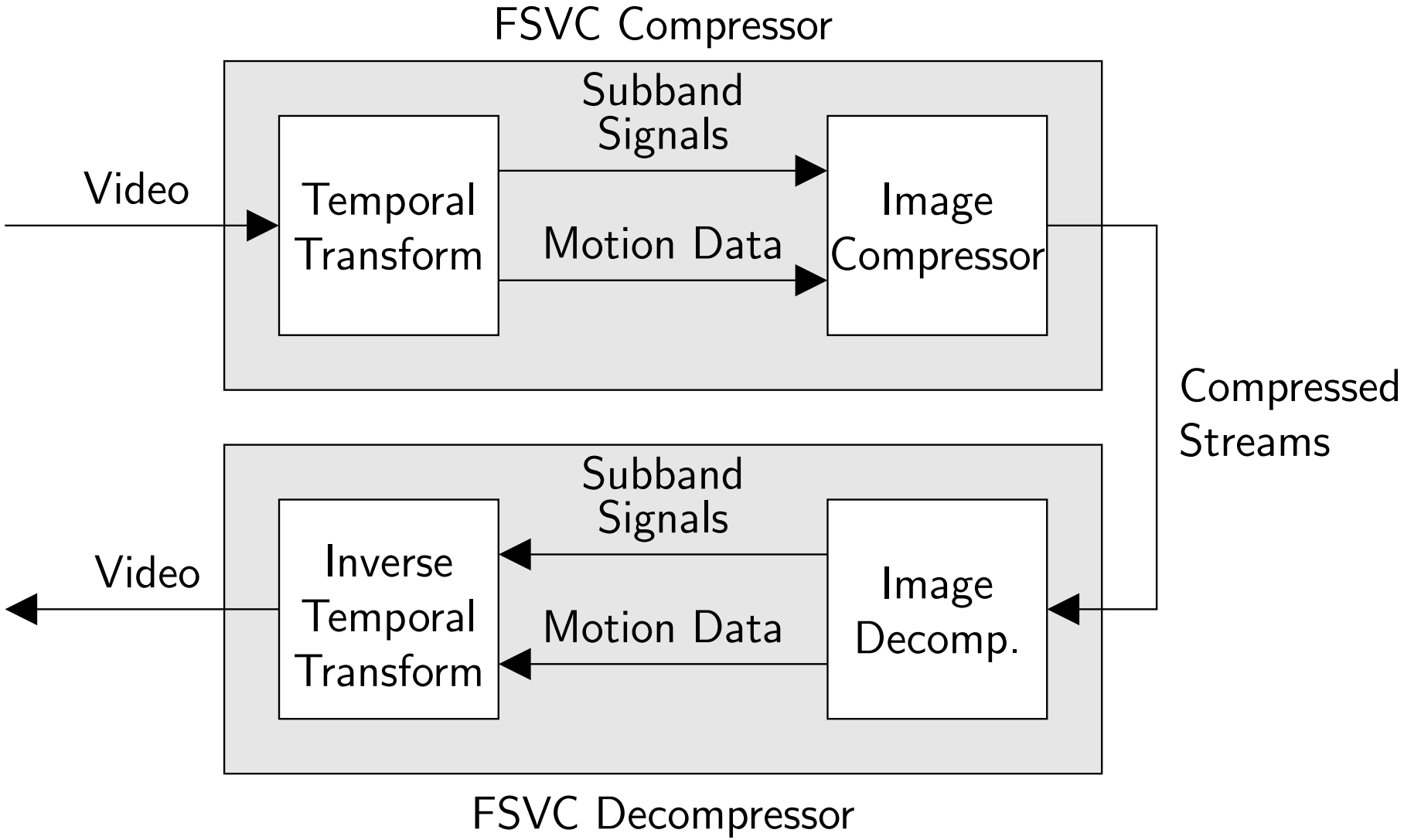
Capítulo 33

FSVC (Fully Scalable Video Coding)

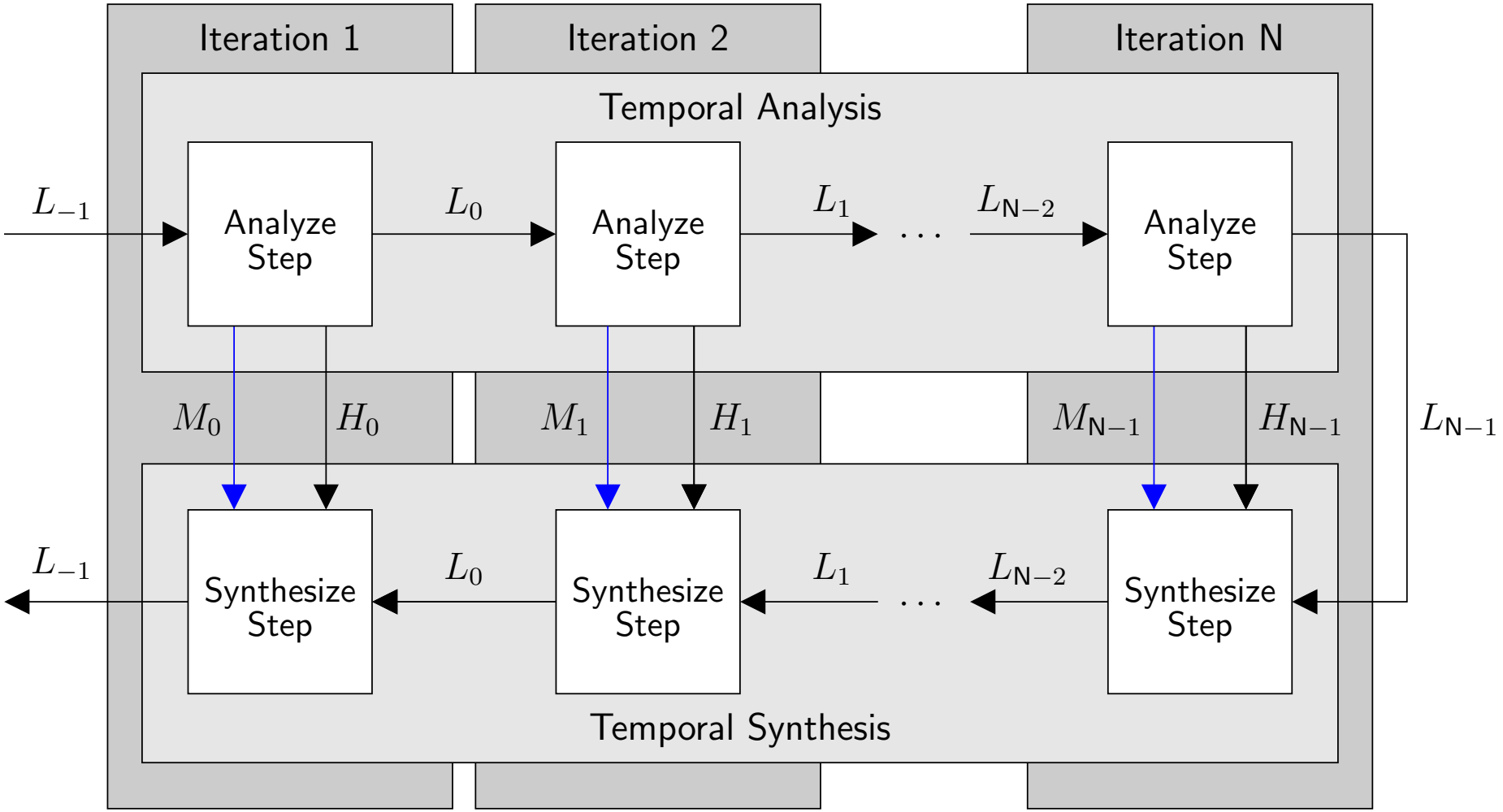
33.1. Antecedentes

- Técnica t+2D de lazo abierto.
- Desarrollada por el Grupo de Investigación "Supercomputación: Algoritmos" de la Universidad de Almería.
- Específicamente desarrollado para realizar streaming de vídeo, maximizando el ancho de banda instantáneo disponible.
- Basado en JPEG2000 y la transformada 5/3 (Lineal) aplicada sobre el dominio del tiempo (t-DWT).

33.2. Etapas básicas de FSVC

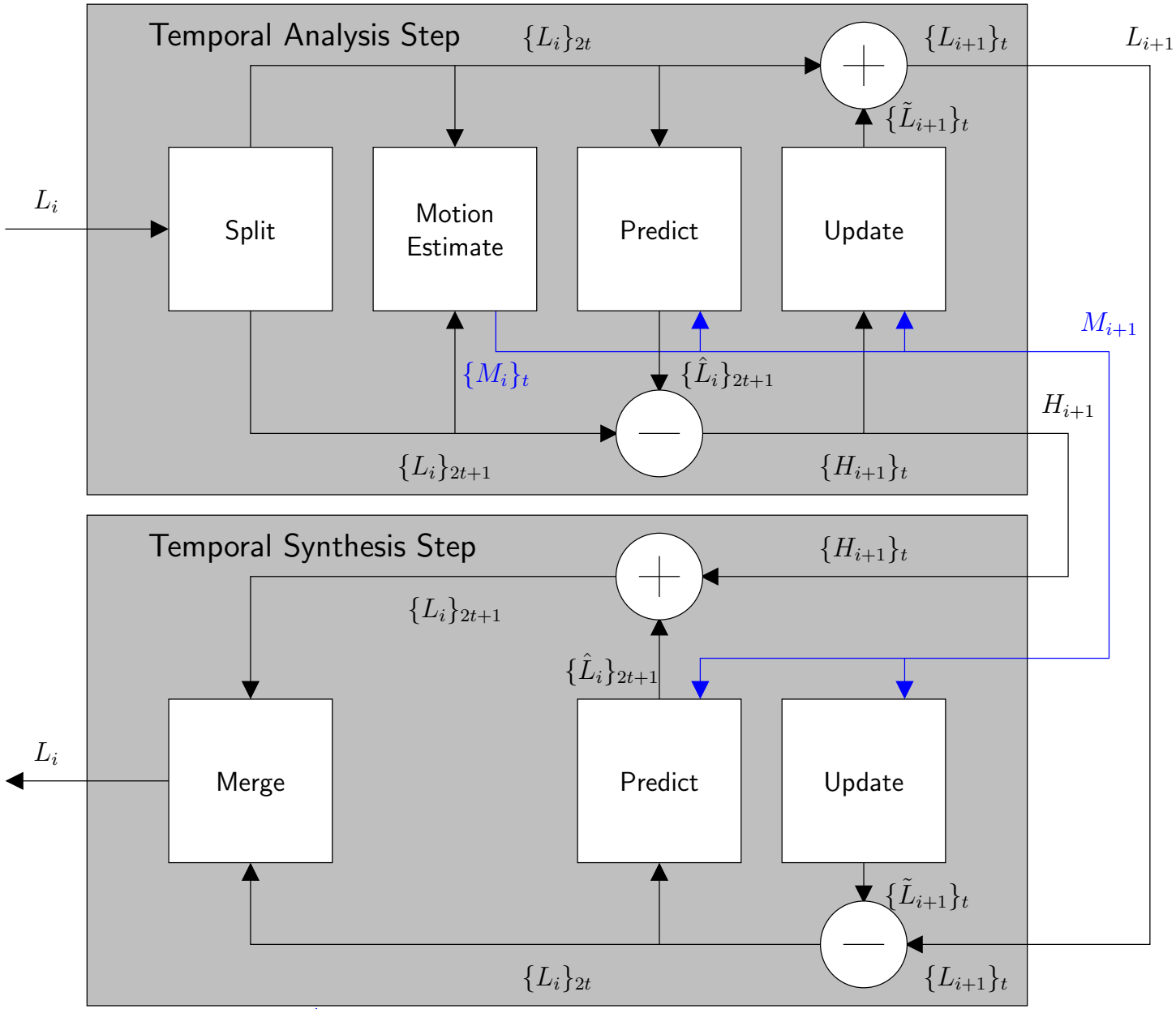


33.3. Temporal Analysis/Synthesis

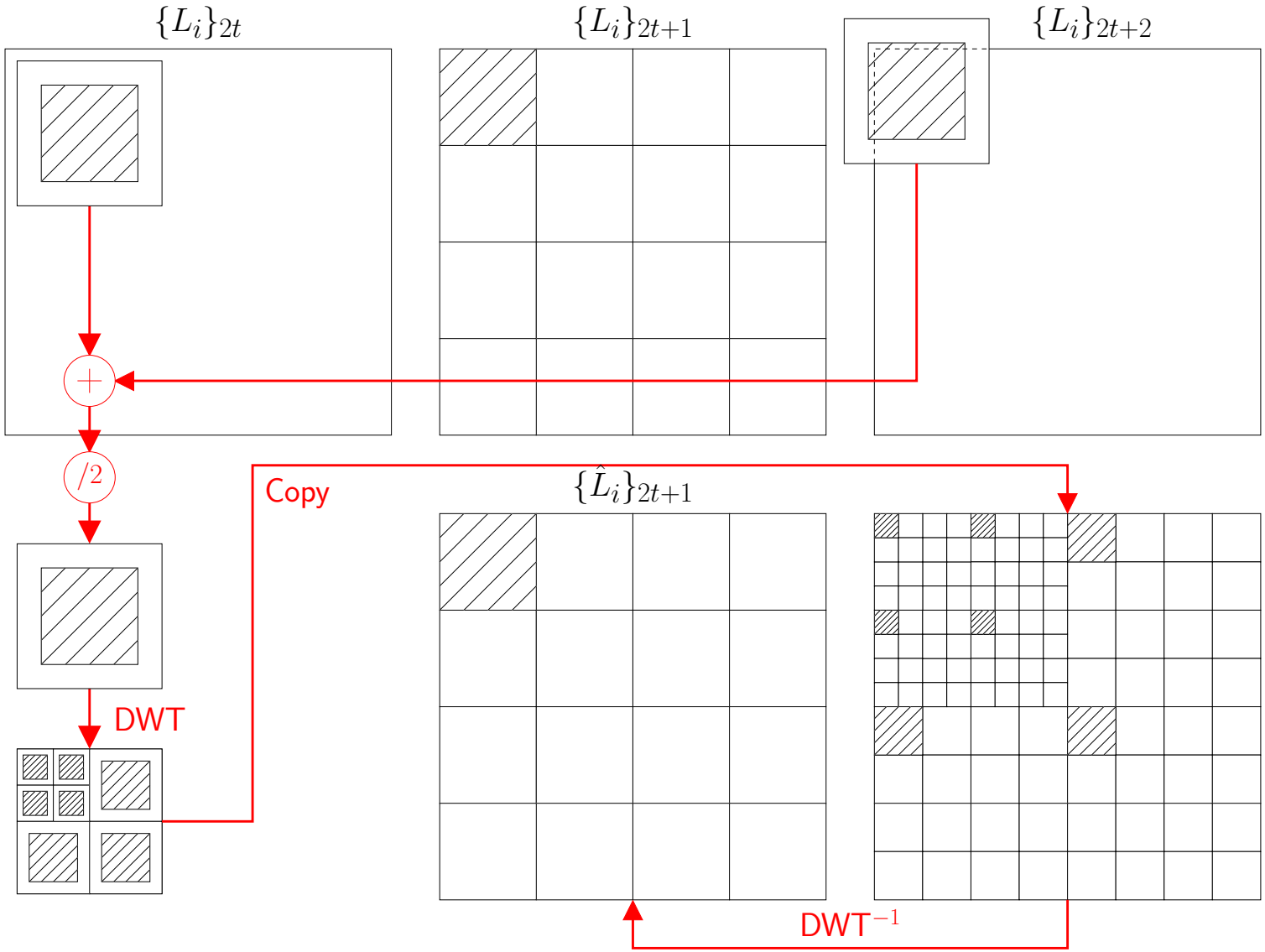


33.3 Temporal Analysis/Synthesis

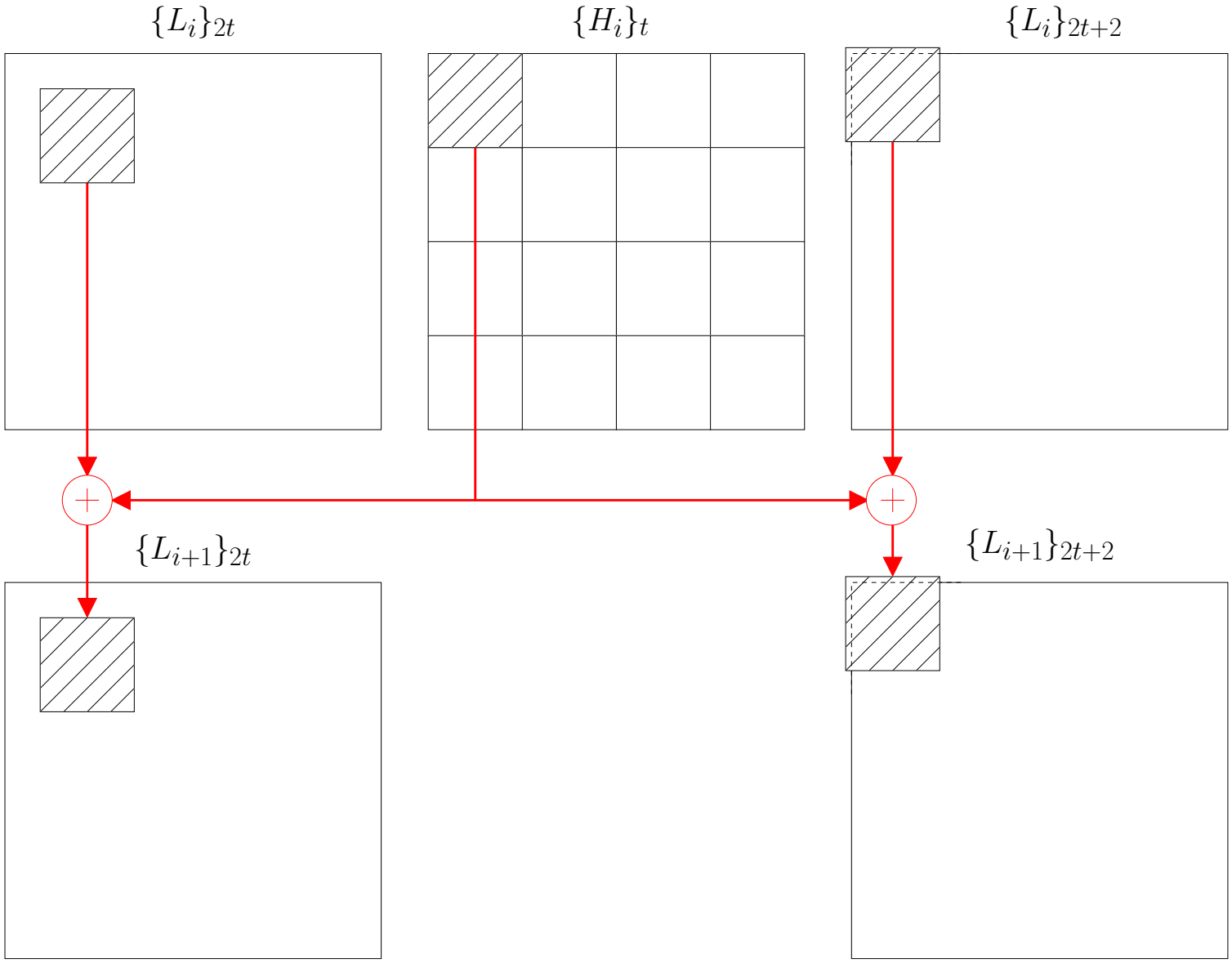
33.4. Temporal Analysis/Synthesis Step



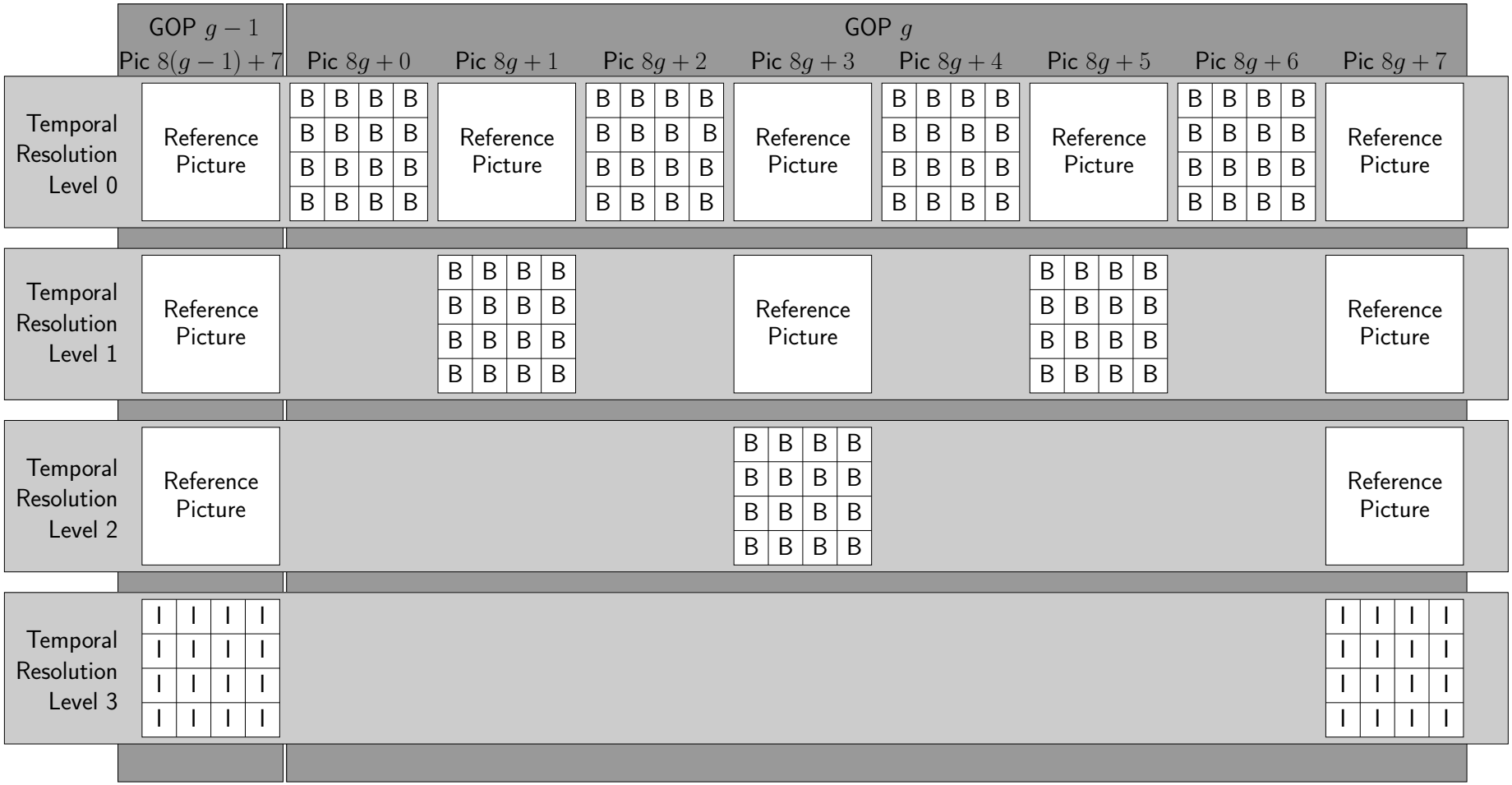
33.5. The prediction step



33.6. The update step

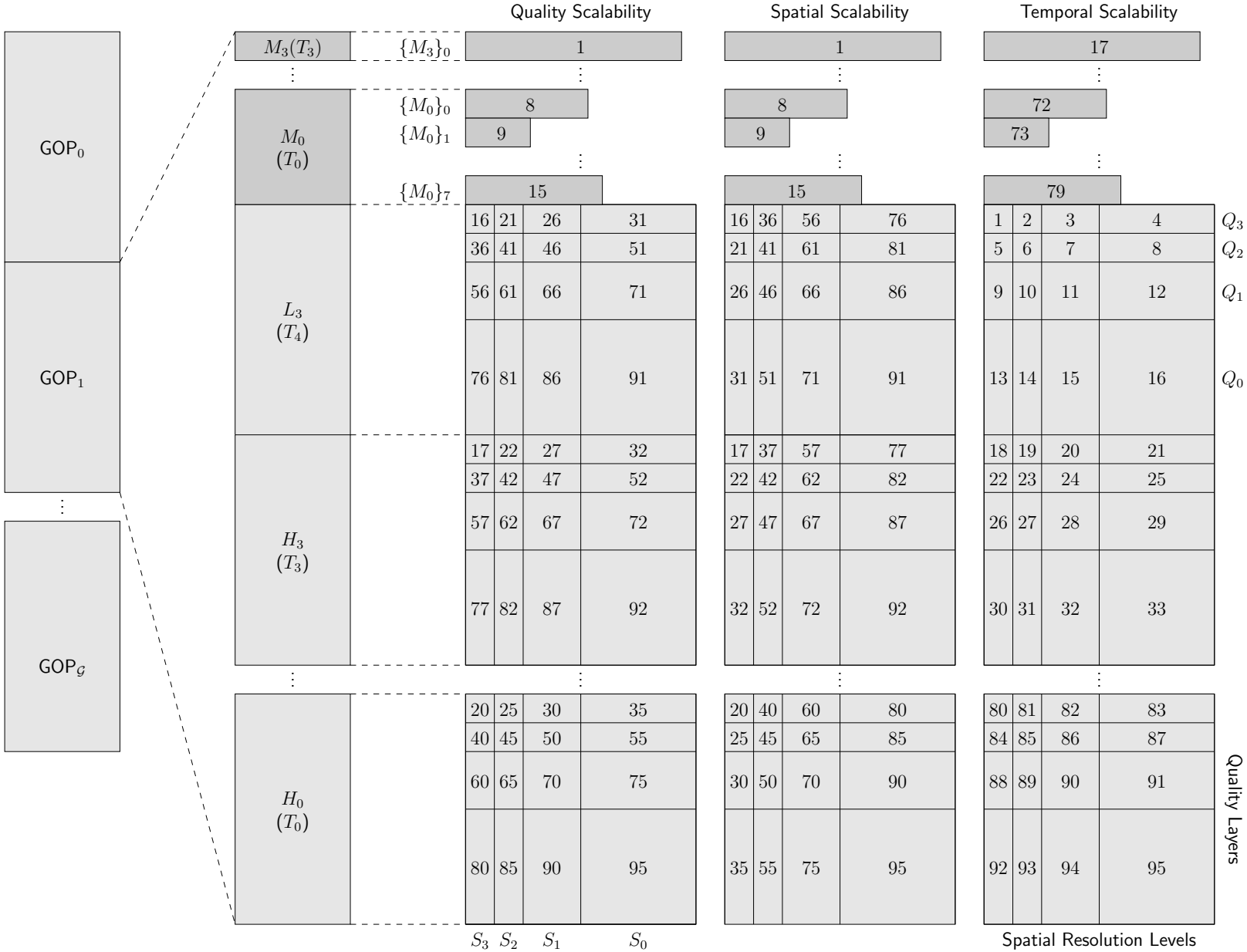


33.7. Motion Compensated Temporal Filtering



I = Intra coded block
B = Bidirectionaly predictive coded block

33.8. Stream organizations



33.9. Ejemplos de compresión

- Para ver los resultados que general el codec MPEG-1, se ha comprimido un conjunto de vídeos. Las tasas de compresión, aunque no son exáctas (por problemas del codec de vídeo) han sido: $1.200, 600, 300, 100$ y 50 Kbps.
- Búsquese los ficheros `*FSVC*` en el directorio `vids`.

33.10. Ejemplos de transmisión “en calidad”

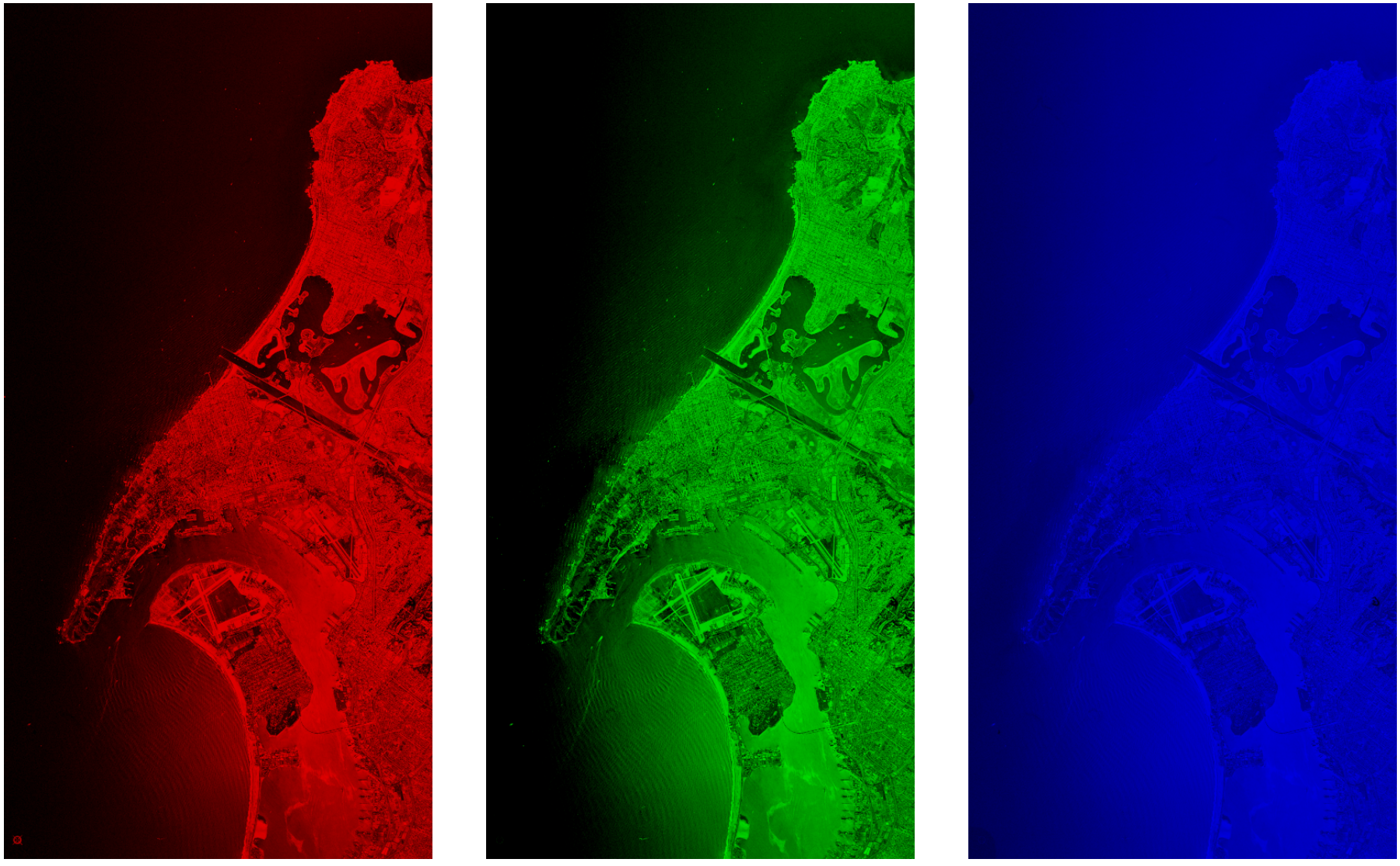
- La ventaja de usar un codec escalable en calidad es que las variaciones de ancho de banda durante la transmsión afecta a la calidad del vídeo reproducido y no a sus resoluciones espacial y temporal.
- En los siguientes experimentos se supone que el ancho de banda decae hasta 50 Kbps durante diferentes intervalos de tiempo de 1 segundo, aproximadamente.
- Búsquese los ficheros `*FSVC_1200kbps_trans*` en el directorio `vids`.

Apéndices

Apéndice 34

Correlación entre las componentes de color

34.1. El dominio RGB



Entropía total = 21,37 bpp

34.2. El dominio YCbCr

- Los algoritmos de compresión rara vez se aplican directamente sobre el dominio RGB. Por tal motivo, primero se aplica una transformación que elimine correlación y por lo tanto decremente la entropía total de las 3 bandas.
- Existen muchas transformaciones. Una de las más sencillas consiste en:

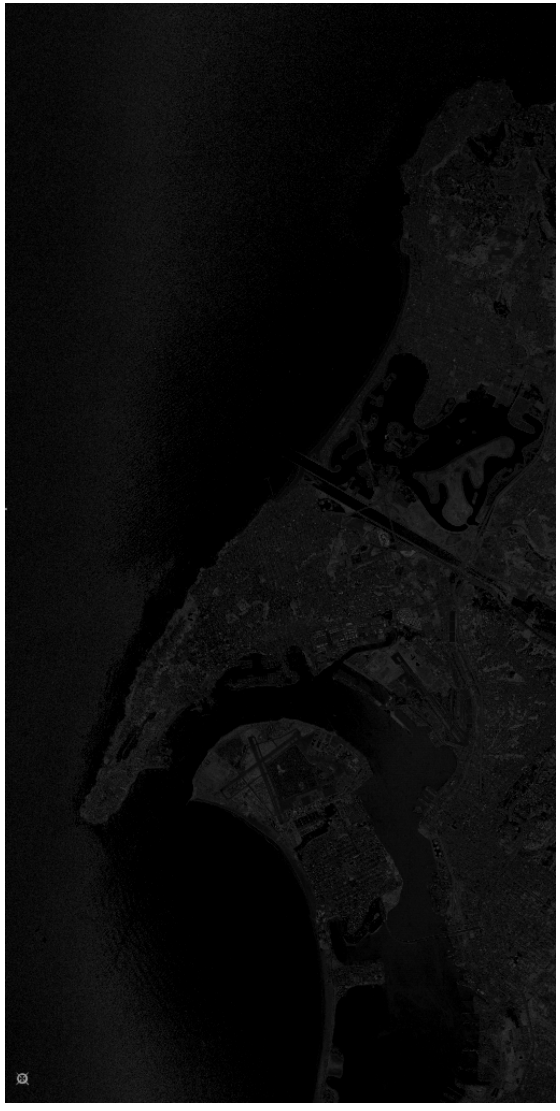
$$\begin{aligned} Y &= \frac{R + 2 \times G + B}{4} \\ Cb &= B - G \\ Cr &= R - G \end{aligned} \tag{34.1}$$

- A la componente Y se le llama luminancia (o luma) y a las componentes Cb y Cr crominancia (o croma).
- El sistema de visión humano es mucho menos sensible a la distorsión de la crominancia que a la de la luminancia. Esto es especialmente cierto en las altas frecuencias, donde la croma juega un papel secundario en la percepción de los detalles y las fronteras.

Y, 7,42 bpp

Cb, 6,86 bpp

Cr, 4,51 bpp



Total = 18,79 bpp

Apéndice 35

Definición del PSNR

35.1. Peak Signal-to-Noise Ratio

- Para comparar la calidad de las imágenes reconstruidas utilizando un compresor lossy se utilizan medidas de la distorsión provocada.
- Una de las medidas más comunes es la relación señal/ruido pico o PSNR (Peak Signal-to-Noise Ratio) que se define como

$$\text{PSNR[dB]} = 10 \log_{10} \frac{(2^b - 1)^2}{\text{MSE}} \tag{35.1}$$

donde b es la profundidad en bits inicial de los puntos y el MSE (Mean Squared Error) se calcula como

$$\text{MSE} = \frac{1}{N} \sum_{\boldsymbol{i}=1}^{N} (s[\boldsymbol{i}] - \hat{s}[\boldsymbol{i}])^2 \tag{35.2}$$

donde N es el número de puntos de las imágenes, s[$\boldsymbol{i}$] es el $\boldsymbol{i}$-ésimo ($\boldsymbol{i} = (x, y)$) punto de la imagen $s[\cdot]$ y $\hat{s}[\boldsymbol{i}]$ es el $\boldsymbol{i}$-ésimo punto de la imagen reconstruida (comprimida y descomprimida).

Apéndice 36

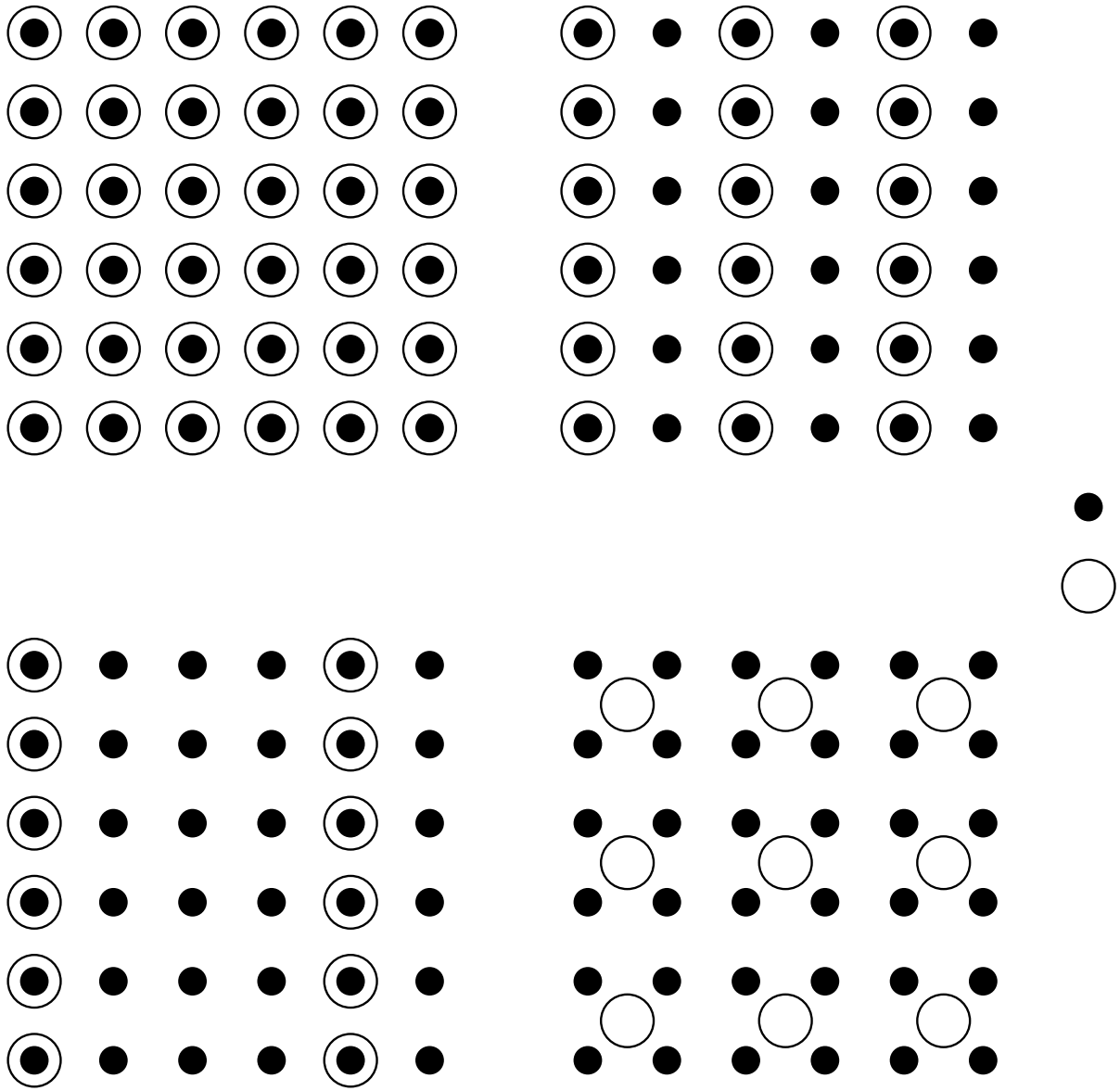
Submuestreo de la crominancia

36.1. Submuestro del dominio YCbCr

- El sistema visual humano es más sensible a la pérdida de información en la luma que en el croma. Por eso las bandas Cb y Cr se submuestrean en una de estas 4 formas:

  1. **Formato 4:4:4.** Las tres componentes se muestrean con la misma frecuencia. Es la que debería aplicarse en aplicaciones *lossless*.
  2. **Formato 4:2:2.** Cada dos muestras de Y (sólo) en horizontal se toma una para Cb y otra para Cr.
  3. **Formato 4:1:1.** Cada cuatro muestras de Y en horizontal se toma una para Cb y otra para Cr.
  4. **Formato 4:2:0.** Cada dos muestras de Y en horizontal y en vertical se toma una para Cb y otra para Cr.

- Ejemplos:



Apéndice 37

Downsampling and upsampling

A 2-fold downsampler, shown in figures as $\downarrow^2$, takes an input sequence I and produces a output sequence $\downarrow^2 (I)$ where

2 is referred to as the downsampling factor.

The inverse element 2-fold upsampler, despected in figures as $\uparrow^2$, for a given input sequence I produces the output $\uparrow^2 (I)$ where

$$(\uparrow^2 (I))_i = \begin{cases} I_{i/2} & \text{if } i \text{ is an integer multiple of 2} \\ 0 & \text{otherwise.} \end{cases}$$

Apéndice 38

The scalar quantization operator

A quantizer is a function that applied to a digital signal produces other digital signal of the same size, but where the number of possible sample val-

ues (representation levels) is reduced. This process generates an irreversible loss of information in the signal. The difference between the original samples S_i and its quantized version $\mathcal{Q}_N(S)$ is named the quantization error. A well designed quantizer should minminze this error signal.

When the quantization is scalar, this correspondence can be done using only an input sample for each output sample. A scalar quantizer with N representation levels is defined by

$$\mathcal{Q}_N(S_i) = \begin{cases} x_0 & \text{if } S_i < d_1 \\ x_1 & \text{if } d_1 \leq S_i < d_2 \\ \vdots & \vdots \\ x_{N-1} & \text{if } S_i > d_{N-1} \end{cases}$$

where

$$d_i = x_{i-1} + \frac{Z}{2}$$

are the decision levels and where

$$Z = x_{i+1} - x_i$$

is the quantization step size.

Apéndice 39

Códigos fuente

39.1. _5_3.h

```
template <typename TYPE>
class _5_3 {

 public:
  char *get_filter_name();
  void even_analyze(TYPE *signal, TYPE *low, TYPE *high, int n);
  void odd_analyze(TYPE *signal, TYPE *low, TYPE *high, int n);
  void even_synthesize(TYPE *signal, TYPE *low, TYPE *high, int n);
  void odd_synthesize(TYPE *signal, TYPE *low, TYPE *high, int n);

};

template <typename TYPE>
char *_5_3<TYPE>::get_filter_name() {
  return "5/3 (Lineal) Biorthogonal Perfect Reconstruction Filter Bank";
}

template <typename TYPE>
void _5_3<TYPE>::even_analyze(TYPE *signal, TYPE *low, TYPE *high, int n) {
  int i;
  for(i=0;i<n/2-1;i++) {
```

```
    int i2 = i<<1;
    //high[i] = signal[i2+1] - ((signal[i2]+signal[i2+2])>>1);
    high[i] = signal[i2+1] - (signal[i2]+signal[i2+2])/2;
  }
  high[i] = signal[n-1] - signal[n-2];

  //low[0] = signal[0] + ((high[0]+1)>>1);
  low[0] = signal[0] + high[0]/2;
  for(i=1;i<n/2;i++) {
    int i2 = i<<1;
    //low[i] = signal[i2] + ((high[i]+high[i-1]+2)>>2);
    low[i] = signal[i2] + (high[i]+high[i-1])/4;
  }
}

template <typename TYPE>
void _5_3<TYPE>::even_synthesize(TYPE *signal, TYPE *low, TYPE *high, int n)
  int i;
  //signal[0] = low[0] - ((high[0]+1)>>1);
  signal[0] = low[0] - high[0]/2;
  for(i=1;i<n/2;i++) {
    int i2 = i<<1;
    //signal[i2] = low[i] - ((high[i]+high[i-1]+2)>>2);
```

```
    signal[i2] = low[i] - (high[i]+high[i-1])/4;
  }

  for(i=0;i<n/2-1;i++) {
    int i2 = i<<1;
    //signal[i2+1] = high[i] + ((signal[i2]+signal[i2+2])>>1);
    signal[i2+1] = high[i] + (signal[i2]+signal[i2+2])/2;
  }
  signal[n-1] = high[i] + signal[n-2];
}

template <typename TYPE>
void _5_3<TYPE>::odd_analyze(TYPE *signal, TYPE *low, TYPE *high, int n) {
  int i;
  for(i=0;i<n/2;i++) {
    int i2 = i<<1;
    //high[i] = signal[i2+1] - ((signal[i2]+signal[i2+2])>>1);
    high[i] = signal[i2+1] - (signal[i2]+signal[i2+2])/2;
  }

  //low[0] = signal[0] + ((high[0]+1)>>1);
  low[0] = signal[0] + high[0]/2;
  for(i=1;i<n/2;i++) {
```

```
    int i2 = i<<1;
    //low[i] = signal[i2] + ((high[i]+high[i-1]+2)>>2);
    low[i] = signal[i2] + (high[i]+high[i-1])/4;
  }
  //low[i] = signal[n-1] + ((high[i-1]+1)>>1);
  low[i] = signal[n-1] + high[i-1]/2;
}

template <typename TYPE>
void _5_3<TYPE>::odd_synthesize(TYPE *signal, TYPE *low, TYPE *high, int n) {
  int i;
  //signal[0] = low[0] - ((high[0]+1)>>1);
  signal[0] = low[0] - high[0]/2;
  for(i=1;i<n/2;i++) {
    int i2 = i<<1;
    //signal[i2] = low[i] - ((high[i]+high[i-1]+2)>>2);
    signal[i2] = low[i] - (high[i]+high[i-1])/4;
  }
    //signal[n-1] = low[i] - ((high[i-1]+1)>>1);
  signal[n-1] = low[i] - high[i-1]/2;

  for(i=0;i<n/2;i++) {
    int i2 = i<<1;
```

```
    //signal[i2+1] = high[i] + ((signal[i2]+signal[i2+2])>>1);
    signal[i2+1] = high[i] + (signal[i2]+signal[i2+2])/2;
  }
}
```

39.2. _13_7.h

```
template <typename TYPE>
class _5_3 {

 public:
  char *get_filter_name();
  void even_analyze(TYPE *signal, TYPE *low, TYPE *high, int n);
  void odd_analyze(TYPE *signal, TYPE *low, TYPE *high, int n);
  void even_synthesize(TYPE *signal, TYPE *low, TYPE *high, int n);
  void odd_synthesize(TYPE *signal, TYPE *low, TYPE *high, int n);

};

template <typename TYPE>
char *_5_3<TYPE>::get_filter_name() {
  return "13/7 (Cuvic) Biorthogonal Perfect Reconstruction Filter Bank";
}

template <typename TYPE>
void _13_7<TYPE>::even_analyze(TYPE *s, TYPE *l, TYPE *h, int n) {
  int i;
  h[0] = s[1] - s[0];
```

```
if(n>2) {
  for(i=1;i<n/2-2;i++) {
    int i2 = i<<1;
    //h[i] = s[i2+1] - ((9*(s[i2]+s[i2+2]) - (s[i2-2]+s[i2+4])+8)>>4);
    h[i] = s[i2+1] - (9*(s[i2]+s[i2+2]) - (s[i2-2]+s[i2+4])/16);
  }
  //h[n/2-2] = s[n-3] - ((s[n-4]+s[n-2]+1)>>1);
  h[n/2-2] = s[n-3] - (s[n-4]+s[n-2])/2;
  h[n/2-1] = s[n-1] - s[n-2];
}

//l[0] = s[0] + ((h[0])>>1);
l[0] = s[0] + h[0]/2;

if(n>2) {
  //l[1] = s[2] + ((h[0]+h[1]+1)>>2);
  l[1] = s[2] + (h[0]+h[1])/4;
  for(i=2; i<n/2-1; i++) {
    int i2 = i<<1;
    //l[i] = s[i2] + ((-h[i-2]+9*(h[i-1]+h[i])-h[i+1]+16)>>5);
    l[i] = s[i2] + (-h[i-2]+9*(h[i-1]+h[i])-h[i+1])/32;
  }
```

```
    //l[n/2-1] = s[n-2] + ((h[n/2-2]+h[n/2-1]+1)>>2);
    l[n/2-1] = s[n-2] + (h[n/2-2]+h[n/2-1]+1)/4;
  }
}

template <typename TYPE>
void _13_7<TYPE>::odd_analyze(TYPE *s, TYPE *l, TYPE *h, int n) {
  int i;
  //h[0] = s[1] - ((s[0]+s[2]+1)>>1);
  h[0] = s[1] - (s[0]+s[2])/2;
  for(i=1;i<n/2-1;i++) {
    int i2 = i<<1;
    //h[i] = s[i2+1] - ((9*(s[i2]+s[i2+2]) - (s[i2-2]+s[i2+4])+8)>>4);
    h[i] = s[i2+1] - (9*(s[i2]+s[i2+2]) - (s[i2-2]+s[i2+4]))/16;
  }
  //h[n/2-1] = s[n-2] - ((s[n-3]+s[n-1]+1)>>1);
  h[n/2-1] = s[n-2] - (s[n-3]+s[n-1])/2;

  //l[0] = s[0] + (h[0]>>1);
  l[0] = s[0] + h[0]/2;
  //l[1] = s[2] + ((h[0]+h[1]+1)>>2);
  l[1] = s[2] + (h[0]+h[1])/4;
```

```
  for(i=2; i<n/2-1; i++) {
    int i2 = i<<1;
    //l[i] = s[i2] + ((-h[i-2]+9*(h[i-1]+h[i])-h[i+1]+16)>>5);
    l[i] = s[i2] + (-h[i-2]+9*(h[i-1]+h[i])-h[i+1])/32;
  }
  //l[n/2-1] = s[n-3] + ((h[n/2-2]+h[n/2-1]+1)>>2);
  l[n/2-1] = s[n-3] + (h[n/2-2]+h[n/2-1])/4;
  //l[n/2] = s[n-1] + (h[n/2-1]>>1);
  l[n/2] = s[n-1] + h[n/2-1]/2;
}

template <typename TYPE>
void _13_7<TYPE>::even_synthesize(TYPE *s, TYPE *l, TYPE *h, int n) {
  int i;
  //s[0] = l[0] - ((h[0])>>1);
  s[0] = l[0] - h[0]/2;

  if(n>2) {
    //s[2] = l[1] - ((h[0]+h[1]+1)>>2);
    s[2] = l[1] - (h[0]+h[1])/4;
    for(i=2; i<n/2-1; i++) {
      int i2 = i<<1;
      //s[i2] = l[i] - ((-h[i-2]+9*(h[i-1]+h[i])-h[i+1]+16)>>5);
```

```
      s[i2] = l[i] - (-h[i-2]+9*(h[i-1]+h[i])-h[i+1])/32;
    }
    //s[n-2] = l[n/2-1] - ((h[n/2-2]+h[n/2-1]+1)>>2);
    s[n-2] = l[n/2-1] - (h[n/2-2]+h[n/2-1])/4;
  }

  s[1] = h[0] + s[0];
  if(n>2) {
    for(i=1;i<n/2-2;i++) {
      int i2 = i<<1;
      //s[i2+1] = h[i] + ((9*(s[i2]+s[i2+2]) - (s[i2-2]+s[i2+4])+8)>>4);
      s[i2+1] = h[i] + (9*(s[i2]+s[i2+2]) - (s[i2-2]+s[i2+4]))/16;
    }
    //s[n-3] = h[n/2-2] + ((s[n-4]+s[n-2]+1)>>1);
    s[n-3] = h[n/2-2] + (s[n-4]+s[n-2])/2;
    s[n-1] = h[n/2-1] + s[n-2];
  }
}

template <typename TYPE>
void _13_7<TYPE>::odd_synthesize(TYPE *s, TYPE *l, TYPE *h, int n) {
  int i;
  //s[0] = l[0] - (h[0]>>1);
```

```
s[0] = l[0] - h[0]/2;
//s[2] = l[1] - ((h[0]+h[1]+1)>>2);
s[2] = l[1] - (h[0]+h[1])/4;

for(i=2; i<n/2-1; i++) {
  int i2 = i<<1;
  //s[i2] = l[i] - ((-h[i-2]+9*(h[i-1]+h[i])-h[i+1]+16)>>5);
  s[i2] = l[i] - (-h[i-2]+9*(h[i-1]+h[i])-h[i+1])/32;
}
//s[n-3] = l[n/2-1] - ((h[n/2-2]+h[n/2-1]+1)>>2);
s[n-3] = l[n/2-1] - (h[n/2-2]+h[n/2-1])/4;
//s[n-1] = l[n/2] - (h[n/2-1]>>1);
s[n-1] = l[n/2] - h[n/2-1]/2;

//s[1] = h[0] + ((s[0]+s[2]+1)>>1);
s[1] = h[0] + (s[0]+s[2])/2;
for(i=1;i<n/2-1;i++) {
  int i2 = i<<1;
  //s[i2+1] = h[i] + ((9*(s[i2]+s[i2+2]) - (s[i2-2]+s[i2+4])+8)>>4);
  s[i2+1] = h[i] + (9*(s[i2]+s[i2+2]) - (s[i2-2]+s[i2+4]))/16;
}
//s[n-2] = h[n/2-1] + ((s[n-3]+s[n-1]+1)>>1);
s[n-2] = h[n/2-1] + (s[n-3]+s[n-1])/2;
```

```
}
```

39.3. arithmetic_coding.c

```
/*
 * arithmetic_coding.c
 *
 * Un codificador artimético.
 *
 * Referencias:
 *
 * I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for
 * data compression," Commun. ACM, vol. 30, no. 6, pp. 520--540,
 * Jun. 1987.
 */

#include <stdio.h>
#include "bitio.h"
#include "vlc.h"

/*
 * Precisión aritmética. Este valor afecta al tamaño mínimo del
 * intervalo de codificación que es posible crear en una iteración del
 * proceso de transmisión incremental.
```

```
 */
#define BIT_ACCURACY 16

typedef long code_value;

/* Valores críticos en el intervalo de división. */
#define _0_99 (((long)1<<BIT_ACCURACY)-1) /* 0.999... */
#define _0_25 (_0_99/4+1)                 /* 0.25 */
#define _0_50 (_0_25*2)                   /* 0.50 */
#define _0_75 (_0_25*3)                   /* 0.75 */

/* Extremos del intervalo de codificación actual. */
static code_value low, high;

/* Bits del código artimético actualmente contemplados en la
   descodificación. */
static code_value value;

/* Número de bits (opuestos) emitidos tras el siguiente bit. */
static long bits_to_follow;

/* Inicializa el codificador. */
void init_encoder() {
```

```
  low = 0;
  high = _0_99;
  bits_to_follow = 0;
}

/* Inicializa el descodificador. */
void init_decoder() {
  int i;
  value = 0;
  for (i = 0; i<BIT_ACCURACY; i++) {
    value = 2*value;
    if(get_bit()) {
      value += 1;
    }
  }
  low = 0;
  high = _0_99;
}



/* Emite un bit y a continuación "bits_to_follow" bits contrarios. */
static void bit_plus_follow(int bit) {
  put_bit(bit);
```

```
  while (bits_to_follow>0) {
    put_bit(!bit);
    bits_to_follow -= 1;
  }
}

/* Codifica el índice "index" usando el espacio de recuentos
   acumulados "cum_freq". */
void encode_index(int index, unsigned short *cum_prob) {
  /* Tamaño del intervalo de codificación actual. */
  long range = (long)(high-low)+1;

  /* Seleccionamos el siguiente intervalo. */
  high = low + (range*cum_prob[index-1])/cum_prob[0]-1;
  low  = low + (range*cum_prob[index  ])/cum_prob[0];

  /* Lazo de transmisión incremental. Este lazo se ejecuta tantas
     veces como sea necesario para que "low" y "high" no coincidan en
     sus bits más significativos. */
  for (;;) {
    if (high<_0_50) {
      /* El bit más significativo de "low" y "high" es 0. Este bit
         puede ser transmitido. */
```

```
    bit_plus_follow(0);
  }
  else if (low>=_0_50) {
    /* El bit más significativo de "low" y "high" es 1. Este bit
       puede ser transmitido. */
    bit_plus_follow(1);

    /* Evitamos el desbordamiento de los registros. */
    low -= _0_50;
    high -= _0_50;
  }
  else if (low>=_0_25 && high<_0_75) {
    /* El bit más significativo de "low" y "high" no coinciden y por lo tan
    bits_to_follow += 1;

    /* Evitamos el desbordamiento. */
    low -= _0_25;
    high -= _0_25;
  }
  /* Ningún bit se "low" y "high" coinciden porque el intervalo de
     demasiado grande. */
  else break;
```

```
    /* Escalamos el intervalo de codificación. */
    low = 2*low;
    high = 2*high+1;
  }
}


/* Descodifica el siguiente índice usando el espacio de recuentos
   acumulados "cum_freq". */
int decode_index(unsigned short *cum_prob) {
  /* Símbolo descodificado. */
  int index;

  /* Tamaño del intervalo de codificación actual. */
  long range = (long)(high-low)+1;

  /* Recuento acumulado para "value". */
  int cum = (int)((((long)(value-low)+1)*cum_prob[0]-1)/range);

  /* Encontramos el símbolo. */
  for (index = 1; cum_prob[index]>cum; index++);

  /* Seleccionamos el intervalo de codificación asociado al símbolo
```

```
   descodificado, tal y como hizo el codificador. */
high = low + (range*cum_prob[index-1])/cum_prob[0]-1;
low  = low + (range*cum_prob[index  ])/cum_prob[0];

/* Recepción incremental. */
for (;;) {
  if (high<_0_50) {
    /* nothing */
  }
  else if (low>=_0_50) {
    /* Vamos a expandir la mitad superior del intervalo de
       codificación. Restamos el offset 0.5. */
    value -= _0_50;
    low -= _0_50;
    high -= _0_50;
  }
  else if (low>=_0_25 && high<_0_75) {
    /* Vamos a expandir la mitad media del intervalo de
       codificación. Restamos el offset 0.25. */
    value -= _0_25;
    low -= _0_25;
    high -= _0_25;
  }
```

```
    else break;

    /* Escalamos el intervalo de codificación. */
    low = 2*low;
    high = 2*high+1;

    /* Leémos el siguiente bit de código aritmético. */
    value = 2*value;
    if(get_bit()) {
      value += 1;
    }
  }
  return index;
}

/* Finaliza el codificador. */
void finish_encoder() {
  /* Transmitimos dos bits que seleccionan el cuarto que el
     intervalo de codificación actualmente contiene. */
  bits_to_follow += 1;
  if (low<_0_25) bit_plus_follow(0);
  else bit_plus_follow(1);
  flush();
```

```
}

/* Finaliza el descodificador. */
void finish_decoder() {
}
```

39.4. bitio.c

```
#include <stdio.h>
#include "bitio.h"

/* En realidad 2**bit_to_read, donde q bit_to_read va desde 0 hasta
   7. */
static int bit_to_read = 256;
static int input_byte = 0;
static int bit_to_write = 1;
static int output_byte = 0;
static int get_counter = 0;
static int put_counter = 0;

#define FEEDBACK 8192

int get_bit() {
  int bit;
  if(bit_to_read==256) {
    input_byte = getchar();
    bit_to_read = 1;
  }
  bit = input_byte & bit_to_read;
```

```
  bit_to_read <<= 1;
  get_counter++;
  if(!(get_counter%8192)) fprintf(stderr,".");
  return bit;
}

int get_bits(int number_of_bits_to_get) {
  int i;
  int s = (get_bit() ? 1:0);
  for(i=1; i<number_of_bits_to_get; i++) {
    s <<= 1;
    s |= (get_bit() ? 1:0);
  }
  return s;
}

void put_bit(int bit) {
  if(bit_to_write==256) {
    putchar(output_byte);
    bit_to_write = 1;
    output_byte = 0;
  }
  if(bit) output_byte |= bit_to_write;
```

```
  bit_to_write <<= 1;
  put_counter++;
  if(!(put_counter%8192)) fprintf(stderr,"o");
}

void put_bits(int bits, int number_of_bits_to_put) {
  int i;
  for(i=number_of_bits_to_put-1; i>=0; i--) {
    put_bit(bits & (1<<i));
  }
}

void flush() {
  if(bit_to_write!=0) putchar(output_byte);
}
```

39.5. bitio.h

```
int  get_bit ();
int  get_bits(int number_of_bits_to_get);
void put_bit (int bit);
void put_bits(int bits, int number_of_bits_to_put);
void flush   ();
```

39.6. bwt.c

```
//
//  BWT.CPP
//
//  Mark Nelson
//  March 8, 1996
//  http://web2.airmail.net/markn
//
// DESCRIPTION
// -----------
//
//  This program performs a Burrows-Wheeler transform on an input
//  file or stream, and sends the result to an output file or stream.
//
//  While this program can be compiled in 16 bit mode, it will suffer
//  greatly by virtue of the fact that it will need to drop its
//  block size tremendously.
//
//  This program takes two arguments: an input file and an output
//  file.  You can leave off one argument and send your output to
//  stdout.  Leave off two arguments and read your input from stdin
//  as well.
```

```
//
//  The output consists of a series of blocks that look like this:
//
//  long byte_count |  ...data... | long first | long last
//
//  The byte_count refers to the number of data bytes.  The data
//  itself is the "L" column from the sorted data.  "first" is the
//  index where the first character from the buffer appears in the
//  sorted output.  "last" is where the end-of-buffer special byte
//  appears in the output buffer.  These blocks are repeated until
//  I'm out of data.
//
//  This program accompanies my article "Data Compression with the
//  Burrows-Wheeler Transform."  There is one major deviation from
//  the text of the article in this implementation.  To simplify the
//  sorting, I append a special end-of-buffer character to the end
//  of the input buffer.  The end-of-buffer character isn't found
//  in the buffer, which means I no longer have to wrap around to
//  the start of the buffer when performing comparisons.  Instead,
//  I'm guaranteed that a memcmp() will terminate at or before the
//  last character in the buffer.
//
//  One problem, though.  Since I can handle any kind of binary input,
```

```
//  what character is guaranteed to never appear in the buffer?  None,
//  so instead I do a special hack and make sure I never *really*
//  look at that last position when comparing.  Instead, I only compare
//  until one or the other string gets to the end, then award the
//  comparison to whoever hit the end first.
//
//  This special character means the output has N+1 characters.  I just
//  output a '?' when I hit that special end-of-buffer character, but
//  I also have to pass along the information about the end-of-buffer
//  character's position to the decoder, so I append it to the end
//  of each data block.
//
//  The sorting for this routine is done via conventional qsort().
//  The STL capable version in BWT.CPP is neater, and in theory
//  should run faster.  The comparison function here is nearly
//  the same as that from BWT.CPP, but it isn't a template fn.  Also,
//  instead of passing pointers into the buffer, the qsort() compare
//  function gets indices.
//
// Build Instructions
// ------------------
//
//  Define the constant unix for UNIX or UNIX-like systems.  The
```

```
//  use of this constant turns off the code used to force the MS-DOS
//  file system into binary mode.  g++ already does this, your UNIX
//  C++ compiler might also.
//
//  Microsoft Visual C++ 2.x  : cl /W4 bwta.cpp
//  Borland C++ 4.5 32 bit    : bcc32 -w bwt.cpp
//  Microsoft Visual C++ 1.52 : cl /W4 bwt.cpp
//  Microsoft Visual C++ 2.1  : cl /W4 bwt.cpp
//  g++                       : g++ -o bwt bwt.cpp
//
// Typical Use
// -----------
//
//  rle < raw-file | bwt | mtf | rle | ari > compressed-file
//

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <fcntl.h>
#if !defined( unix )
#include <io.h>
```

```
#endif
#include <limits.h>

#if ( INT_MAX == 32767 )
#define BLOCK_SIZE 20000
#else
#define BLOCK_SIZE 10 //200000
#endif

//
// length has the number of bytes presently read into the buffer,
// buffer contains the data itself.  indices[] is an array of
// indices into the buffer.  indices[] is what gets sorted in
// order to sort all the strings in the buffer.
//
long length;
unsigned char buffer[ BLOCK_SIZE ];
int indices[ BLOCK_SIZE + 1 ];

//
// The logic in unbwt.cpp depends upon the strings having been
// sorted as unsigned characters.  Some versions of memcmp() sort
// using *signed* characters.  When this is the case, I compare
```

```
// using this special replacement version of memcmp().
//
int memcmp_signed;

int unsigned_memcmp( void *p1, void *p2, unsigned int i )
{
    unsigned char *pc1 = (unsigned char *) p1;
    unsigned char *pc2 = (unsigned char *) p2;
    while ( i-- ) {
        if ( *pc1 < *pc2 )
            return -1;
        else if ( *pc1++ > *pc2++ )
            return 1;
    }
    return 0;
}

//
// This is the special comparison function used when calling
// qsort() to sort the array of indices into the buffer. Remember that
// the character at buffer+length doesn't really exist, but it is assumed
// to be the special end-of-buffer character, which is bigger than
// any character found in the input buffer.  So I terminate the
```

```
// comparison at the end of the buffer.
//

int
#if defined( _MSC_VER )
_cdecl
#endif
bounded_compare( const unsigned int *i1,
                 const unsigned int *i2 )
{
    static int ticker = 0;
    if ( ( ticker++ % 4096 ) == 0 )
        fprintf( stderr, "." );
    unsigned int l1 = (unsigned int) ( length - *i1 );
    unsigned int l2 = (unsigned int) ( length - *i2 );
    int result;
    if ( memcmp_signed )
        result = unsigned_memcmp( buffer + *i1,
                                  buffer + *i2,
                                  l1 < l2 ? l1 : l2 );
    else
        result = memcmp( buffer + *i1,
                         buffer + *i2,
```

```
                        l1 < l2 ? l1 : l2 );
    if ( result == 0 )
        return l2 - l1;
    else
        return result;
};


main( int argc, char *argv[] )
{
    int debug = 0;
    if ( argc > 1 && strcmp( argv[ 1 ], "-d" ) == 0 ) {
        debug = 1;
        argv++;
        argc--;
    }
#ifdef _INFO_
    fprintf( stderr, "Performing BWT on " );
#endif
    if ( argc > 1 ) {
        freopen( argv[ 1 ], "rb", stdin );
        fprintf( stderr, "%s", argv[ 1 ] );
    } else
```

```
#ifdef _INFO_
      fprintf( stderr, "stdin" );
    fprintf( stderr, " to " );
#endif
    if ( argc > 2 ) {
        freopen( argv[ 2 ], "wb", stdout );
        fprintf( stderr, "%s", argv[ 2 ] );
    } else
#ifdef _INFO_
        fprintf( stderr, "stdout" );
    fprintf( stderr, "\n" );
#endif
#if !defined( unix )
    setmode( fileno( stdin ), O_BINARY );
    setmode( fileno( stdout ), O_BINARY );
#endif
    if ( memcmp( "\x070", "\x080", 1 ) < 0 ) {
        memcmp_signed = 0;
        fprintf( stderr, "memcmp() treats character data as unsigned\n" );
    } else {
        memcmp_signed = 1;
        fprintf( stderr, "memcmp() treats character data as signed\n" );
    }
```

```
//
// This is the start of the giant outer loop.  Each pass
// through the loop compresses up to BLOCK_SIZE characters.
// When an fread() operation finally reads in 0 characters,
// we break out of the loop and are done.
//
    for ( ; ; ) {
//
// After reading in the data into the buffer, I do some
// UI stuff, then write the length out to the output
// stream.
//
        length = fread( (char *) buffer, 1, BLOCK_SIZE, stdin );
        if ( length == 0 )
            break;
#ifdef _INFO_
        fprintf( stderr, "Performing BWT on %ld bytes\n", length );
#endif
        long l = length + 1;
        fwrite( (char *) &l, 1, sizeof( long ), stdout );
//
// Sorting the input strings is simply a matter of inserting
// the indices into the array, then calling qsort() with the
```

```
// special bounded_compare() function I wrote to work with
// these string types.  Note that I sort N+1 indices. The last index
// points one past the end of the buffer, which is where the
// imaginary end-of-buffer character resides.  Sort of.
//
        int i;
        for ( i = 0 ; i <= length ; i++ )
            indices[ i ] = i;
        qsort( indices,
               (int)( length + 1 ),
               sizeof( int ),
               ( int (*)(const void *, const void *) ) bounded_compare );
#ifdef _INFO_
        fprintf( stderr, "\n" );
#endif
//
// If the debug flag was turned on, I print out the sorted
// strings, along with their prefix characters.  This is
// not a very good idea when you are compressing a giant
// binary file, but it can be real helpful when debugging.
//
        if ( debug ) {
            for ( i = 0 ; i <= length ; i++ ) {
```

```
            fprintf( stderr, "%d : ", i );
            unsigned char prefix;
            if ( indices[ i ] == 0 )
                prefix = '?';
            else
                prefix = (unsigned char) buffer[ indices[ i ] - 1 ];
            if ( isprint( prefix ) )
                fprintf( stderr, "%c", prefix );
            else
                fprintf( stderr, "<%d>", prefix );
            fprintf( stderr, ": " );
            int stop = (int)( length - indices[ i ] );
            if ( stop > 30 )
                stop = 30;
            for ( int j = 0 ; j < stop ; j++ ) {
                if ( isprint( buffer[ indices[ i ] + j ] ) )
                    fprintf( stderr, "%c", buffer[ indices[ i ] + j ] );
                else
                    fprintf( stderr, "<%d>", buffer[ indices[ i ] + j ] )
            }
            fprintf( stderr, "\n" );
        }
    }
```

```
//
// Finally, I write out column L.  Column L consists of all
// the prefix characters to the sorted strings, in order.
// It's easy to get the prefix character, but I have to
// handle S0 with care, since its prefix character is the
// imaginary end-of-buffer character.  I also have to spot
// the positions in L of the end-of-buffer character and
// the first character, so I can write them out at the end
// for transmission to the output stream.
//
        long first;
        long last;
        for ( i = 0 ; i <= length ; i++ ) {
            if ( indices[ i ] == 1 )
                first = i;
            if ( indices[ i ] == 0 ) {
                last = i;
                fputc( '?', stdout );
            } else
                fputc( buffer[ indices[ i ] - 1 ], stdout );
        }
#ifdef _INFO_
        fprintf( stderr,
```

```
                    "first = %ld"
                    "  last = %ld\n",
                    first,
                    last );
#endif
        fwrite( (char *) &first, 1, sizeof( long ), stdout );
        fwrite( (char *) &last, 1, sizeof( long ), stdout );
    }
    return 0;
}
```

39.7. codec.h

```
void encode_stream(int argc, char *argv[]);
void decode_stream(int argc, char *argv[]);
```

39.8. dwt1d.h

```
/*
 * The 2D Discrete Wavelet Transform.
 * gse. 2007.
 */
template <typename TYPE, class FILTER>
class dwt1d: public FILTER, public mallok {

 public:
  dwt1d();
  ~dwt1d();
  void analyze(TYPE *signal, int x, int levels);
  void synthesize(TYPE *signal, int x, int levels);
  void set_max_line_size(int max_line_size);

 private:
  TYPE *line;
};

template <typename TYPE, class FILTER>
dwt1d<TYPE,FILTER>::dwt1d<TYPE,FILTER>() {
  line = (TYPE *)mallok::alloc_1d(1,sizeof(TYPE));
```

```
#if defined DEBUG
  if(!iine) {
    cerr << "dwt1d<TYPE,FILTER>::dwt1d<TYPE,FILTER>: out of memory for \"line
    abort();
  }
#endif
}

template <typename TYPE, class FILTER>
dwt1d<TYPE,FILTER>::~dwt1d<TYPE,FILTER>() {
  mallok::free_1d(line);
}

template <typename TYPE, class FILTER>
void dwt1d<TYPE,FILTER>::set_max_line_size(int max_line_size) {
  mallok::free_1d(line);
  line = (TYPE *)mallok::alloc_1d(max_line_size,sizeof(TYPE));
}

template <typename TYPE, class FILTER>
void dwt1d<TYPE,FILTER>::analyze(TYPE *signal, int x, int levels) {
  for(int lv=0;lv<levels;lv++) {
    int nx = x; x >>= 1;
```

```
    if(x == 0) x = 1;

    if(nx & 1) { /* N'umero impar de elementos */
      memcpy(line,signal,nx*sizeof(TYPE));
      odd_analyze(line,signal,signal+x+1,nx);
    } else { /* N'umero par de elementos */
      memcpy(line,signal,nx*sizeof(TYPE));
      even_analyze(line,signal,signal+x,nx);
    }
  }
}

template <typename TYPE, class FILTER>
void dwt1d<TYPE,FILTER>::synthesize(TYPE *signal, int x, int levels) {
  int nx = x>>levels;

  for(int lv = levels-1; lv>=0; lv--) {
    int mx = nx; nx = x>>lv;

    if(nx==0) nx = 1;

    if(nx & 1) { /* N'umero impar de elementos */
```

```
      memcpy(line,signal,nx*sizeof(TYPE));
      odd_synthesize(signal,line,line+mx+1,nx);
    } else { /* N'umero par de columas */
      memcpy(line,signal,nx*sizeof(TYPE));
      even_synthesize(signal,line,line+mx,nx);
    }
  }
}
```

39.9. dwt1d_image.cpp

```
#include <iostream>
using namespace std;
#include "mallok.h"
#include "Haar.h"
#include "_5_3.h"
#include "dwt1d.h"
//#define TYPE long
//#include "dwt2d.h"
#include "image.h"

#define Y 100
#define X 200
#define N 1000
#define L 3
#define TRAN _5_3
#define TYPE image

int main() {
  dwt1d < TYPE, TRAN < TYPE > > x;
  cout << x.TRAN < TYPE >::get_filter_name() << '\n';
  x.set_max_line_size(N);
```

```
  class mallok *m = new class mallok();
  TYPE *d = (TYPE *)m->alloc_1d(N, sizeof(TYPE));
  for(int n=0; n<N; n++) {
    d[n].create(Y,X);
    for(int y=0; y<Y; y++) {
      for(int x=0; x<X; x++) {
        d[n][y][x] = n*N*Y+y*Y+x;
      }
    }
  }

  x.analyze(d, N, L);
  x.synthesize(d, N, L);

}
```

39.10. dwt1d_rgb_image.cpp

```
#include <iostream>
using namespace std;
#include "mallok.h"
#include "Haar.h"
#include "_5_3.h"
#include "dwt1d.h"
//#define TYPE long
//#include "dwt2d.h"
#include "rgb_image.h"

#define Y 100
#define X 200
#define N 1000
#define L 3
#define TRAN _5_3
#define TYPE rgb_image

int main() {
  dwt1d < TYPE, TRAN < TYPE > > x;
  cout << x.TRAN < TYPE >::get_filter_name() << '\n';
  x.set_max_line_size(N);
```

```
  class mallok *m = new class mallok();
  TYPE *d = (TYPE *)m->alloc_1d(N, sizeof(TYPE));
  for(int n=0; n<N; n++) {
    d[n].create(Y,X);
    for(int y=0; y<Y; y++) {
      for(int x=0; x<X; x++) {
        d[n][0][y][x] = y;
        d[n][1][y][x] = y;
        d[n][2][y][x] = y;
      }
    }
  }

  x.analyze(d, N, L);
  x.synthesize(d, N, L);

}
```

39.11. dwt2d.h

```
/*
 * The 2D Discrete Wavelet Transform.
 * gse. 2007.
 */
template <typename TYPE, class FILTER>
class dwt2d: public FILTER, public mallok {

 public:
  dwt2d();
  ~dwt2d();
  void analyze(TYPE **signal, int y, int x, int levels);
  void synthesize(TYPE **signal, int y, int x, int levels);
  void set_max_line_size(int max_line_size);

 private:
  TYPE *in_line;
  TYPE *out_line;
};

template <typename TYPE, class FILTER>
dwt2d<TYPE,FILTER>::dwt2d<TYPE,FILTER>() {
```

```
  in_line = (TYPE *)mallok::alloc_1d(1,sizeof(TYPE));
#if defined DEBUG
  if(!in_line) {
    cerr << "dwt2d<TYPE,FILTER>::dwt2d<TYPE,FILTER>: out of memory for \"in_l
    abort();
  }
#endif
  out_line = (TYPE *)mallok::alloc_1d(1,sizeof(TYPE));
#if defined DEBUG
  if(!out_line) {
    cerr << "dwt2d<TYPE,FILTER>::dwt2d<TYPE,FILTER>: out of memory for \"out_
    abort();
  }
#endif
}

template <typename TYPE, class FILTER>
dwt2d<TYPE,FILTER>::~dwt2d<TYPE,FILTER>() {
  mallok::free_1d(out_line);
  mallok::free_1d(in_line);
}

template <typename TYPE, class FILTER>
```

```
void dwt2d<TYPE,FILTER>::set_max_line_size(int max_line_size) {
  mallok::free_1d(out_line);
  mallok::free_1d(in_line);
  in_line = (TYPE *)mallok::alloc_1d(max_line_size,sizeof(TYPE));
  out_line = (TYPE *)mallok::alloc_1d(max_line_size,sizeof(TYPE));
}

template <typename TYPE, class FILTER>
void dwt2d<TYPE,FILTER>::analyze(TYPE **signal, int y, int x, int levels) {
  for(int lv=0;lv<levels;lv++) {
    int nx = x; x >>= 1;
    int ny = y; y >>= 1;
    if(y == 0) y = 1; /* Nuevo */
    if(x == 0) x = 1; /* Nuevo */

    /* Transformamos las filas */
    if(nx & 1) { /* N'umero impar de columnas */
      for(int j=0;j<ny;j++) {
        memcpy(in_line,signal[j],nx*sizeof(TYPE));
        odd_analyze(in_line,signal[j],signal[j]+x+1,nx);
      }
    } else { /* N'umero par de columnas */
      for(int j=0;j<ny;j++) {
```

```
      memcpy(in_line,signal[j],nx*sizeof(TYPE));
      even_analyze(in_line,signal[j],signal[j]+x,nx);
    }
  }

  /* Transformamos las columnas */
  if(ny & 1) { /* N'umero impar de filas */
    for(int i=0;i<nx;i++) {
      for(int j=0;j<ny;j++) {
        in_line[j]=signal[j][i];
      }
      odd_analyze(in_line,out_line,out_line+y+1,ny);
      for(int j=0;j<ny;j++) {
        signal[j][i]=out_line[j];
      }
    }
  } else { /* N'umero par de filas */
    for(int i=0;i<nx;i++) {
      for(int j=0;j<ny;j++) {
        in_line[j]=signal[j][i];
      }
      even_analyze(in_line,out_line,out_line+y,ny);
      for(int j=0;j<ny;j++) {
```

```
            signal[j][i]=out_line[j];
          }
        }
      }
    }
  }

template <typename TYPE, class FILTER>
void dwt2d<TYPE,FILTER>::synthesize(TYPE **signal, int y, int x, int levels)
  int nx = x>>levels;
  int ny = y>>levels;

  for(int lv = levels-1; lv>=0; lv--) {
    int mx, my;
    mx = nx; nx=x>>lv;
    my = ny; ny=y>>lv;
    if(nx==0) nx=1; /* Nuevo */
    if(ny==0) ny=1; /* Nuevo */

    /* Transformamos las columnas */
    if(ny & 1) { /* N'umero de filas impar */
      for(int i=0;i<nx;i++) {
        for(int j=0;j<ny;j++) {
```

```
      in_line[j]=signal[j][i];
    }
    odd_synthesize(out_line,in_line,in_line+my+1,ny);
    for(int j=0;j<ny;j++) {
      signal[j][i]=out_line[j];
    }
  }
} else { /* N'umero de filas par */
  for(int i=0;i<nx;i++) {
    for(int j=0;j<ny;j++) {
      in_line[j]=signal[j][i];
    }
    even_synthesize(out_line,in_line,in_line+my,ny);
    for(int j=0;j<ny;j++) {
      signal[j][i]=out_line[j];
    }
  }
}

/* Transformamos las columnas (i) */
if(nx & 1) { /* N'umero impar de columnas */
  for(int j=0;j<ny;j++) {
    memcpy(in_line,signal[j],nx*sizeof(TYPE));
```

```
        odd_synthesize(signal[j],in_line,in_line+mx+1,nx);
      }
    } else { /* N'umero par de columas */
      for(int j=0;j<ny;j++) {
        memcpy(in_line,signal[j],nx*sizeof(TYPE));
        even_synthesize(signal[j],in_line,in_line+mx,nx);
      }
    }
  }
}
```

39.12. dwt2d_line.cpp

```
#include <iostream>
using namespace std;
#include "mallok.h"
#include "Haar.h"
#include "_5_3.h"
#include "dwt2d.h"
#include "line.h"

#define N 1000
#define L 3
#define TRAN _5_3
#define TYPE line

int main() {
  dwt2d < TYPE, TRAN < TYPE > > x;
  cout << x.TRAN < TYPE >::get_filter_name() << '\n';
  x.set_max_line_size(N);
  class mallok *m = new class mallok();
  TYPE **d = (TYPE **)m->alloc_2d(N, N, sizeof(TYPE));
  for(int y=0; y<N; y++) {
    for(int x=0; x<N; x++) {
```

```
      d[y][x].create(100);
      for(int z=0; z<100; z++) {
        d[y][x][z] = y*N*N+x*N+z;
      }
    }
  }

  x.analyze(d, N, N, L);
  x.synthesize(d, N, N, L);


}
```

39.13. entropy.c

```
/*
 * entropy.c
 *
 * Calcula la entropía de una secuencia de símbolos de 8 bits.
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*
 * Tamaño del alfabeto fuente.
 */
#define ALPHABET_SIZE 256

double entropy(unsigned long *count, int alphabet_size) {
  double entropy = 0.0;
  unsigned long total_count = 0;
  int i;
  for(i=0; i<alphabet_size; i++) {
```

```
    total_count += count[i];
  }

  for(i=0; i<alphabet_size; i++) {
    if(count[i]) {
      double prob = (double)count[i]/total_count;
      entropy += prob*log(prob)/log(2.0);
    }
  }
  return -entropy;
}

int main(int argc, char *argv[]) {
  unsigned long count[ALPHABET_SIZE];
  int i;
  for(i=0; i<ALPHABET_SIZE; i++) {
    count[i] = 0;
  }
  for(;;) {
    int x = getc(stdin);
    if(x==EOF) break;
    count[x]++;
  }
```

```
  for(i=0; i<ALPHABET_SIZE; i++) {
    fprintf(stderr, "%3d %u\n", i, count[i]);
  }
  fprintf(stderr, "Entropy = ");
  fflush(stderr);
  fprintf(stdout, "%f\n", entropy(count,256));
}
```

39.14. golomb.c

```
/*
 * unary.c
 *
 * Un codificador unario.
 *
 * Referencias:
 *
 * Golomb, S.W. (1966). , Run-length encodings. IEEE Transactions on
 * Information Theory, IT--12(3):399--401.
 *
 * Witten, Ian Moffat, Alistair Bell, Timothy. "Managing Gigabytes:
 * Compressing and Indexing Documents and Images." Second
 * Edition. Morgan Kaufmann Publishers, San Francisco CA. 1999 ISBN
 * 1-55860-570-3.
 *
 * David Salomon. "Data Compression",ISBN 0-387-95045-1.
 */

#include <stdio.h>
#include <math.h>
```

```
#include "bitio.h"
#include "vlc.h"

/* Inicializa el codificador. */
void init_encoder() {
}

/* Inicializa el descodificador. */
void init_decoder() {
}

/* Calcula el recuento del símbolo x. */
static int prob(unsigned short *cum_prob, int x) {
  return cum_prob[x-1] - cum_prob[x];
}
/* Estima la pendiente de la distribución de probabilidades de los
   símbolos. Se presupone que existen 256 símbolos en el alfabeto. */
int estimate_m(unsigned short *cum_prob) {
  int m;
  m = 255-(255.0*(cum_prob[0] - cum_prob[1]))/cum_prob[0];
  /* Debido a una limitación de "bitio", no podemos generar códigos
     unarios más largos de 32 bits (256/32=8). */
  if(m<8) m=8;
```

```
  return m;
}

/* Codifica el índice "index" usando el espacio de recuentos
   acumulados "cum_freq". */
void encode_index(int index, unsigned short *cum_prob) {
  int i, k, m, s ,t, r;
  m = estimate_m(cum_prob);
  k = ceil(log((float)m)/log(2.0));
  t = (1<<k)-m;
  s = index - 1;
  r = s % m;
  for(i=0; i<(s/m); i++) {
    put_bit(1);
  }
  put_bit(0);
  if(r<t) {
    put_bits(r, k-1);
  } else {
    r += t;
    put_bits(r, k);
  }
}
```

```
static int next_bit() {
  if(get_bit()) return 1;
  return 0;
}

/* Descodifica el siguiente índice. */
int decode_index(unsigned short *cum_prob) {
  int i, x, s, k, m, t;
  m = estimate_m(cum_prob);
  k = ceil(log((float)m)/log(2.0));
  t = (1<<k)-m;
  s = 0;
  while(get_bit()) {
    s++;
  }
  x = get_bits(k-1);
  if (x<t) {
    s = s*m + x;
  } else {
    x = x*2 + next_bit();
    s = s*m + x - t;
  }
```

```
  return s+1;
}

/* Finaliza el codificador. */
void finish_encoder() {
  flush();
}

/* Finaliza el descodificador. */
void finish_decoder() {
}
```

39.15. Haar.h

```
template <typename TYPE>
class Haar {

 public:
  char *get_filter_name();
  void even_analyze(TYPE *signal, TYPE *low, TYPE *high, int n);
  void odd_analyze(TYPE *signal, TYPE *low, TYPE *high, int n);
  void even_synthesize(TYPE *signal, TYPE *low, TYPE *high, int n);
  void odd_synthesize(TYPE *signal, TYPE *low, TYPE *high, int n);

};

template <typename TYPE>
char *Haar<TYPE>::get_filter_name() {
  return "2/1 (Haar) Biorthogonal Perfect Reconstruction Filter Bank";
}

template <typename TYPE>
void Haar<TYPE>::even_analyze(TYPE *signal, TYPE *low, TYPE *high, int n) {
  int i, k;
  for (i = k = 0; k < n; i++, k += 2) {
```

```
    high[i] = signal[k+1] - signal[k];
    low[i] = signal[k] + high[i]/2;
  }
}

template <typename TYPE>
void Haar<TYPE>::even_synthesize(TYPE *signal, TYPE *low, TYPE *high, int n)
  int i, k;
  for (i = k = 0; k < n; i++, k += 2) {
    signal[k] = low[i] - high[i]/2;
    signal[k+1] = signal[k] + high[i];
  }
}

template <typename TYPE>
void Haar<TYPE>::odd_analyze(TYPE *signal, TYPE *low, TYPE *high, int n) {
  int i, k;
  for (i = k = 0; k < (n-1); i++, k += 2) {
    high[i] = signal[k+1] - signal[k];
    low[i] = signal[k] + high[i]/2;
  }
  low[i] = signal[k];
}
```

```
template <typename TYPE>
void Haar<TYPE>::odd_synthesize(TYPE *signal, TYPE *low, TYPE *high, int n) {
  int i, k;
  for (i = k = 0; k < (n-1); i++, k += 2) {
    signal[k] = low[i] - high[i]/2;
    signal[k+1] = signal[k] + high[i];
  }
  signal[k] = low[i];
}
```

39.16. huff.c

```
/*
 * huff.c
 *
 * Un codificador de Huffman junto a un modelo probabilístico,
 * estático, de orden 0.
 *
 * Referencias:
 *
 * D. A. Huffman, Proceedings of the Institute of Radio Engineers,
 * Vol. 40, pp. 1098-1101. 1952.
 * M. Nelson and J.-L. Gailly, The Data Compression Book. 1995.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "bitio.h"
#include "main.h"

/*
```

```
 * The NODE structure is a node in the Huffman decoding tree.  It has a
 * count, which is its weight in the tree, and the node numbers of its
 * two children.
 */
typedef struct tree_node {
  unsigned int count;
  int child_0;
  int child_1;
} NODE;

/*
 * A Huffman tree is set up for decoding, not encoding. When encoding,
 * I first walk through the tree and build up a table of codes for
 * each symbol. The codes are stored in this CODE structure.
 */
typedef struct code {
  unsigned int code;
  int code_bits;
} CODE;

/*
 * Mantiene una copia de la entrada estándar. Recuérdese que este
 * programa funciona en dos pasadas. En la primera se calcula el
```

```
 * recuento de cada byte en el fichero a comprimir. En la segunda se
 * comprime. "tmp_file" almacena una copia de la entrada estándar en
 * un fichero temporal para poder recorrer el stream de entrada dos
 * veces.
 */
FILE *tmp_file;

/*
 * The special EOS symbol is 256, the first available symbol after all
 * of the possible bytes.  When decoding, reading this symbols
 * indicates that all of the data has been read in.
 */
#define END_OF_STREAM 256

/*
 * Comprime el stream de entrada.
 */
void compress(int argc, char *argv[]) {
  unsigned long counts[256];
  NODE nodes[514];
  CODE codes[257];
  int root_node;
  count_bytes(counts);
```

```
  scale_counts( counts, nodes );
  output_counts(nodes);
  root_node = build_tree( nodes );
  convert_tree_to_code( nodes, codes, 0, 0, root_node );
  compress_data(codes);
}

/*
 * Expande el stream de entrada.
 */
void expand(int argc, char *argv[]) {
  NODE nodes[514];
  int root_node;
  input_counts(nodes);
  root_node = build_tree( nodes );
  expand_data(nodes, root_node);
}

/*
 * This routine counts the frequency of occurence of every byte in
 * the input file.
 */
count_bytes(unsigned long *counts) {
```

```
  int c;
  char tmp_filename[80], number[3];
  sprintf(number,"%d", getpid()%1000);
  strcpy(tmp_filename, "/tmp/");
  strcat(tmp_filename, number);
  tmp_file = fopen(tmp_filename, "w+");
  if(!tmp_file) {
    fprintf(stderr, "huff: imposible abrir (%s)\n", tmp_filename);
    exit(1);
  }
  while ( (c = getchar()) != EOF ) {
    counts[ c ]++;
    putc(c, tmp_file);
  }
  fflush(tmp_file);
  rewind(tmp_file);
}

/*
 * In order to limit the size of my Huffman codes to 16 bits, I scale
 * my counts down so they fit in an unsigned char, and then store them
 * all as initial weights in my NODE array.  The only thing to be
 * careful of is to make sure that a node with a non-zero count doesn't
```

```
 * get scaled down to 0. Nodes with values of 0 don't get codes.
 */
scale_counts(unsigned long *counts, NODE *nodes) {
  unsigned long max_count;
  int i;
  max_count = 0;
  for ( i = 0 ; i < 256 ; i++ )
    if ( counts[ i ] > max_count )
      max_count = counts[ i ];
  if ( max_count == 0 ) {
    counts[ 0 ] = 1;
    max_count = 1;
  }
  max_count = max_count / 255;
  max_count = max_count + 1;
  for ( i = 0 ; i < 256 ; i++ ) {
    nodes[ i ].count = (unsigned int) ( counts[ i ] / max_count );
    if ( nodes[ i ].count == 0 && counts[ i ] != 0 )
      nodes[ i ].count = 1;
  }
  nodes[ END_OF_STREAM ].count = 1;
}
```

```
/*
 * Since the Huffman tree is built as a decoding tree, there is
 * no simple way to get the encoding values for each symbol out of
 * it. This routine recursively walks through the tree, adding the
 * child bits to each code until it gets to a leaf.  When it gets
 * to a leaf, it stores the code value in the CODE element, and
 * returns.
 */
convert_tree_to_code(NODE *nodes, CODE *codes,
                             unsigned int code_so_far, int bits, int node) {
  if ( node <= END_OF_STREAM ) {
    codes[ node ].code = code_so_far;
    codes[ node ].code_bits = bits;
    return;
  }
  code_so_far <<= 1;
  bits++;
  convert_tree_to_code( nodes, codes, code_so_far, bits,
                        nodes[ node ].child_0 );
  convert_tree_to_code( nodes, codes, code_so_far | 1,
                        bits, nodes[ node ].child_1 );
}
```

```
/*
 * Once the tree gets built, and the CODE table is built, compressing
 * the data is a breeze.  Each byte is read in, and its corresponding
 * Huffman code is sent out.
 */
compress_data(CODE *codes) {
  int c;
  while((c=getc(tmp_file))!=EOF) {
    bitio__put_bits(codes[ c ].code, codes[ c ].code_bits );
  }
  bitio__put_bits(codes[END_OF_STREAM].code, codes[END_OF_STREAM].code_bits )
  bitio__flush();
}

/*
 * In order for the compressor to build the same model, I have to store
 * the symbol counts in the compressed file so the expander can read
 * them in.  In order to save space, I don't save all 256 symbols
 * unconditionally.  The format used to store counts looks like this:
 *
 *  start, stop, counts, start, stop, counts, ... 0
 *
```

```
 * This means that I store runs of counts, until all the non-zero
 * counts have been stored.  At this time the list is terminated by
 * storing a start value of 0.  Note that at least 1 run of counts has
 * to be stored, so even if the first start value is 0, I read it in.
 * It also means that even in an empty file that has no counts, I have
 * to pass at least one count.
 *
 * In order to efficiently use this format, I have to identify runs of
 * non-zero counts.  Because of the format used, I don't want to stop a
 * run because of just one or two zeros in the count stream.  So I have
 * to sit in a loop looking for strings of three or more zero values in
 * a row.
 *
 * This is simple in concept, but it ends up being one of the most
 * complicated routines in the whole program.  A routine that just
 * writes out 256 values without attempting to optimize would be much
 * simpler, but would hurt compression quite a bit on small files.
 *
 */
output_counts(NODE *nodes) {
  int first;
  int last;
  int next;
```

```
int i;

first = 0;
while ( first < 255 && nodes[ first ].count == 0 )
  first++;
/*
 * Each time I hit the start of the loop, I assume that first is the
 * number for a run of non-zero values.  The rest of the loop is *
 * concerned with finding the value for last, which is the end of
 * the * run, and the value of next, which is the start of the next
 * run.  * At the end of the loop, I assign next to first, so it
 * starts in on * the next run.
 */
for ( ; first < 256 ; first = next ) {
  last = first + 1;
  for ( ; ; ) {
    for ( ; last < 256 ; last++ )
      if ( nodes[ last ].count == 0 )
        break;
    last--;
    for ( next = last + 1; next < 256 ; next++ )
      if ( nodes[ next ].count != 0 )
        break;
```

```
      if ( next > 255 )
        break;
      if ( ( next - last ) > 3 )
        break;
      last = next;
    };
    /*
     * Here is where I output first, last, and all the counts in
     * between.
     */
    putchar(first);
    putchar(last);
    for ( i = first ; i <= last ; i++ ) {
      putchar(nodes[i].count);
    }
  }
  putchar(0);
  fflush(stdout);
}


/*
 * When expanding, I have to read in the same set of counts.  This is
```

```
 * quite a bit easier that the process of writing them out, since no
 * decision making needs to be done.  All I do is read in first, check
 * to see if I am all done, and if not, read in last and a string of
 * counts.
 */
input_counts(NODE *nodes) {
  int first;
  int last;
  int i;
  int c;

  for ( i = 0 ; i < 256 ; i++ )
    nodes[ i ].count = 0;
  first = getchar();
  last = getchar();
  for ( ; ; ) {
    for ( i = first ; i <= last ; i++ ) {
      c = getc( stdin );
      nodes[ i ].count = (unsigned int)c;
    }
    first = getchar();
    if(first==0) break;
    last = getchar();
```

```
  }
  nodes[ END_OF_STREAM ].count = 1;
}

/*
 * Expanding compressed data is a little harder than the compression
 * phase.  As each new symbol is decoded, the tree is traversed,
 * starting at the root node, reading a bit in, and taking either the
 * child_0 or child_1 path.  Eventually, the tree winds down to a
 * leaf node, and the corresponding symbol is output.  If the symbol
 * is the END_OF_STREAM symbol, it doesn't get written out, and
 * instead the whole process terminates.
 */
expand_data(NODE *nodes, int root_node) {
  int node;

  for ( ; ; ) {
    node = root_node;
    do {
      if(bitio__get_bit()) {
        node = nodes[ node ].child_1;
      }
      else {
```

```
        node = nodes[ node ].child_0;
      }
    } while ( node > END_OF_STREAM );
    if ( node == END_OF_STREAM )
      break;
    putchar(node);
  }
}

/*
 * Building the Huffman tree is fairly simple.  All of the active nodes
 * are scanned in order to locate the two nodes with the minimum
 * weights. These two weights are added together and assigned to a new
 * node. The new node makes the two minimum nodes into its 0 child
 * and 1 child. The two minimum nodes are then marked as inactive.
 * This process repeats until their is only one node left, which is the
 * root node.  The tree is done, and the root node is passed back
 * to the calling routine.
 *
 * Node 513 is used here to arbitratily provide a node with a guaranteed
 * maximum value.  It starts off being min_1 and min_2.  After all active
 * nodes have been scanned, I can tell if there is only one active node
 * left by checking to see if min_1 is still 513.
```

```
 */
int build_tree( NODE *nodes ) {
  int next_free;
  int i;
  int min_1;
  int min_2;

  nodes[ 513 ].count = 0xffff;
  for ( next_free = END_OF_STREAM + 1 ; ; next_free++ ) {
    min_1 = 513;
    min_2 = 513;
    for ( i = 0 ; i < next_free ; i++ )
      if ( nodes[ i ].count != 0 ) {
        if ( nodes[ i ].count < nodes[ min_1 ].count ) {
          min_2 = min_1;
          min_1 = i;
        } else if ( nodes[ i ].count < nodes[ min_2 ].count )
          min_2 = i;
      }
    if ( min_2 == 513 )
      break;
    nodes[ next_free ].count = nodes[ min_1 ].count
      + nodes[ min_2 ].count;
```

```
    nodes[ min_1 ].count = 0;
    nodes[ min_2 ].count = 0;
    nodes[ next_free ].child_0 = min_1;
    nodes[ next_free ].child_1 = min_2;
  }
  next_free--;
  return next_free;
}
```

39.17. image.h

```
#include "line.h"

class image: public mallok {

 public:
  line *data;
  int Y;

 public:
  void create(int Y, int X) {
    data = (line *)mallok::alloc_1d(Y, sizeof(line));
    for(int y=0; y<Y; y++) {
      data[y].create(X);
    }
    this->Y = Y;
  }

  image(int Y, int X) {
    create(Y, X);
  }
```

```
void destroy() {
  for(int y=0; y<Y; y++) {
    data[y].destroy();
  }
  mallok::free_1d(data);
}

~image() {
  destroy();
}

line & operator[](int x) {
  return data[x];
}

void set_size(int Y, int X) {
  if(!data) {
    for(int y=0; y<Y; y++) {
      data[y].set_size(X);
    }
  }
  this->Y = Y;
}
```

```
  /* Sobrecarga del operador de asignación entre líneas. */
  image & operator=(const image & right) {
    set_size(right.Y, right.data[0].X);
    for(int y=0; y<Y; y++) {
      data[y] = right.data[y];
    }
    return *this;
}

  /*const*/ image operator+(const image & right) {
    image tmp(right.Y, right.data[0].X);
    for(int y=0; y<Y; y++) {
      tmp.data[y] = data[y] + right.data[y];
    }
    return tmp;
  }

  /*const*/ image operator-(const image & right) {
    image tmp(right.Y, right.data[0].X);
    for(int y=0; y<Y; y++) {
      tmp.data[y] = data[y] - right.data[y];
    }
```

```
    return tmp;
  }

  /*const*/ image operator/(const int val) {
    image tmp(Y, data[0].X);
    for(int y=0; y<Y; y++) {
      tmp.data[y] = data[y] / val;
    }
    return tmp;
  }
};
```

39.18. line.h

```
class line: public mallok {

 public:
  short *data;
  int X;

 private:

 public:
  void create(int X) {
    data = (short *)mallok::alloc_1d(X,sizeof(short));
    this->X = X;
  }

  void destroy() {
     mallok::free_1d(data);
  }

  line(int X) {
    create(X);
  }
```

```
  ~line() {
    destroy();
  }

  short & operator[](int x) {
    return data[x];
  }

  void set_size(int X) {
    if(!data) data = (short *)mallok::alloc_1d(X, sizeof(short));
    this->X = X;
  }

  /* Sobrecarga del operador de asignación entre líneas. */
  line & operator=(const line & right) {
    set_size(right.X);
    for(int x=0; x<X; x++) {
      data[x] = right.data[x];
    }
    return *this;
  }
```

```
/*const*/ line operator+(const line & right) {
  line tmp(right.X);
  for(int x=0; x<X; x++) {
    tmp.data[x] = data[x] + right.data[x];
  }
  return tmp;
}

/*const*/ line operator-(const line & right) {
  line tmp(right.X);
  for(int x=0; x<X; x++) {
    tmp.data[x] = data[x] - right.data[x];
  }
  return tmp;
}

/*const*/ line operator/(const int val) {
  line tmp(X);
  for(int x=0; x<X; x++) {
    tmp.data[x] = data[x] / val;
  }
  return tmp;
}
```

```
};
```

39.19. lzss.c

```
/*
 * lzss.c
 *
 * Storer & Szymanski implementation of the LZ77 algorithm.
 *
 * Referencias:
 *
 * J. Ziv and A. Lempel, IEEE Trans. IT-23, 337-343 (1977).
 * J. A. Storer and T. G. Szymanski, J. ACM, 29, 928-951 (1982).
 * T. C. Bell, IEEE Trans. COM-34, 1176-1182 (1986).
 * M. Nelson and J.-L. Gailly, The Data Compression Book. 1995.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "bitio.h"

/*
 * Tamaño del código que indican la posición de la cadena en el
```

```
 * diccionario. Determina el tamaño la ventana que contiene el texto
 * que se ha codificado anteriormente (diccionario) el y texto que va
 * a codificarse (look-ahead buffer).
 */
#define INDEX_SIZE 12
#define WINDOW_SIZE (1<<INDEX_SIZE)

/*
 * Tamaño del código que indica la longitud de la cadena
 * encontrada. Determina el tamaño del look-ahead buffer.
 */
#define LENGTH_SIZE 4
#define RAW_LOOK_AHEAD_SIZE (1<<LENGTH_SIZE)

/*
 * Tamaño mínimo de la cadena a codificar, medido en bytes. Se utiliza
 * para decidir si se envían códigos ijk o sólo símbolos
 * k. Típicamente MIN_ENCODED_STRING_SIZE valdrá 1, lo que signfica
 * que sólo si concatenamos al menos 2 símbolos utilizaremos un código
 * ijk.
 */
#define MIN_ENCODED_STRING_SIZE ((1 +INDEX_SIZE+LENGTH_SIZE)/9)
```

```
/*
 * Tamaño efectivo del look-ahead buffer. Puesto que en la práctica no
 * van a codificase cadenas de menos de 2 símbolos, es posible
 * reajustar el tamaño del look-ahead buffer sumándole 2. De esta
 * manera, cuando enviémos un código ijk donde j=0, en realidad
 * estaremos indicando una longitud real de 2. Nótese que esto también
 * provoca que el tamaño real del look-ahead buffer sea de 17 símbolos
 * cuando RAW_LOOK_AHEAD_BUFFER == 16.
*/
#define LOOK_AHEAD_SIZE (RAW_LOOK_AHEAD_SIZE + MIN_ENCODED_STRING_SIZE)

/*
 * Indice del nodo que apunta a la raíz del árbol binario.
 */
#define TREE_ROOT WINDOW_SIZE

/*
 * Código de compresión que indica el fin del stream de datos.
 */
#define END_OF_STREAM 0

/*
 * Indica que el nodo del árbol todavía no apunta a ninguna
```

```
 * cadena. Esto es lo mismo que codificar la cadena vacía.
 */
#define UNUSED 0

/*
 * Calcula el módulo del entero "a" usando INDEX_SIZE bits de
 * precisión.
 */
#define MOD_WINDOW(a) ((a) & (WINDOW_SIZE-1))

/*
 * Diccionario y look-ahead buffer. Cuando comparemos la cadena que
 * almacena el look-ahead buffer con la que está en la posición "p"
 * del diccionario, estaremos comparando dicha cadena con la que
 * comienza a partir de dicha dirección "p".
 */
unsigned char window[WINDOW_SIZE];

/*
 * Arbol binario de todas las cadenas que hay en la ventana ordenadas
 * lexicográficamente. Cada nodo contiene 3 índices que apuntan a una
 * posición en la ventana, y por lo tanto, especifican una cadena. La
 * posición 0 de la ventana . La cadena a la que apunta
```

```
* "smaller_child" es menor que la cadena a la que apunta "parent" y
* menos que la cadena a la que apunta "larger_child".
*
* Por motivos de eficiencia de cálculo, tree[WINDOW_SIZE] es un nodo
* especial que se utiliza para localizar la raíz del árbol. Este
* elemento no apunta a ninguna frase (como hacen el resto de nodos
* del árbol) Este nodo representa además al código END_OF_STREAM.
*
* Para comprender mejor cómo funciona este estructura, a continuación
* se presenta un ejemplo de cómo estaría el árbol cuando se ha
* introducido en él la cadena "ababcbababaaaaaaaa", siendo el tamaño
* del look-ahead buffer igual a 8.
*
* Window="ababcbababaaaaaaaa"
*
* ------------12345678
* cadena  1: "ababcbab"
* cadena  2: "babcbaba"
* cadena  3: "abcbabab"
* cadena  4: "bcbababa"
* cadena  5: "cbababaa"
* cadena  6: "bababaaa"
* cadena  7: "ababaaaa"
```

```
* cadena  8: "babaaaaa"
* cadena  9: "abaaaaaa"
* cadena 10: "baaaaaaa"
* cadena 11: "aaaaaaaa"
*
*
*                    nodo 4096
*          smaller +-----+-----+ larger
*                  |           |
*                          nodo 1
*                        "ababcbab"
*                      +-----+-----+
*                      |           |
*                    nodo 7       nodo 2
*                  "ababaaaa"   "babcbaba"
*                  +----+-+   +-----+-----+
*                  |      |   |           |
*                nodo 9     nodo 3      nodo 4
*              "abaaaaaa" "abcbabab"  "bcbababa"
*              +---+-+   +----+----+    +-+----+
*              |     |   |         |    |      |
*            nodo 11              nodo 6     nodo 5
*          "aaaaaaaa"           "babababaa" "cbababaa"
```

```
 *                              +---+---+
 *                              |       |
 *                            nodo 8
 *                           "babaaaaa"
 *                           +---+---+
 *                           |       |
 *                         nodo 10
 *                        "baaaaaaa"
 */
struct {
  int parent;
  int smaller_child;
  int larger_child;
} tree[WINDOW_SIZE + 1];

/*
 * Inicializa el árbol con el diccionario vacío.
 * Típicamente r=1. Así, tras ejecutar este código el árbol queda como:
 *
 *     nodo 4096
 *        +----+
 *             |
 *           nodo 1
```

```
 *        +----+----+
 *        |         |
 *        0         0
 */
void InitTree(int r) {
  tree[ TREE_ROOT ].larger_child = r;
  tree[ r ].parent = TREE_ROOT;
  tree[ r ].larger_child = UNUSED;
  tree[ r ].smaller_child = UNUSED;
}

/*
 * Antes:
 *                      parent
 *                  +-----+-----+
 *                  |           |
 *                 old
 *              +---+---+
 *              |       |
 *             new   "unused"
 *          +---+---+
 *          |       |
 *          A       B
```

```
 *
 * Después:
 *                      parent
 *                   +-----+-----+
 *                   |           |
 *                  new
 *              +---+---+
 *              |       |
 *              A       B
 */
void ContractNode(int old_node, int new_node) {
  tree[ new_node ].parent = tree[ old_node ].parent;
  if ( tree[ tree[ old_node ].parent ].larger_child == old_node )
    tree[ tree[ old_node ].parent ].larger_child = new_node;
  else
    tree[ tree[ old_node ].parent ].smaller_child = new_node;
  tree[ old_node ].parent = UNUSED;
}

/*
 * Antes:
 *                     parent            new
 *                  +-----+-----+     +---+---+
```

```
 *                  |           |    |       |
 *                 old
 *              +---+---+
 *              |       |
 *              A       B
 * Después:
 *                    parent
 *                  +-----+-----+
 *                  |           |
 *                 new
 *              +---+---+
 *              |       |
 *              A       B
 */
void ReplaceNode(int old_node, int new_node) {
  int parent;

  parent = tree[ old_node ].parent;
  if ( tree[ parent ].smaller_child == old_node )
    tree[ parent ].smaller_child = new_node;
  else
    tree[ parent ].larger_child = new_node;
  tree[ new_node ] = tree[ old_node ];
```

```
  tree[ tree[ new_node ].smaller_child ].parent = new_node;
  tree[ tree[ new_node ].larger_child ].parent = new_node;
  tree[ old_node ].parent = UNUSED;
}

/*
 * Localiza el siguiente nodo más pequeño que "node". Esto se hace
 * descendiendo primero por el hijo "smaller" de "node" y luego
 * bajando siempre por los hijos "larger", hasta llegar a una hoja del
 * árbol. Se asume que "node" tiene un hijo más pequeño.
 */
int FindNextNode(int node) {
  int next;

  next = tree[ node ].smaller_child;
  while ( tree[ next ].larger_child != UNUSED )
    next = tree[ next ].larger_child;
  return( next );
}

/*
 * Elimina un nodo del árbol.
 */
```

```
void DeleteString(int p) {
  int  replacement;
  /* Si el nodo a borrar no está en el árbol, no hacemos nada. */
  if ( tree[ p ].parent == UNUSED )
    return;
  /* Si el nodo a borrar sólo tiene un hijo, hacemos una contracción
     del árbol. */
  if ( tree[ p ].larger_child == UNUSED )
    ContractNode( p, tree[ p ].smaller_child );
  else if ( tree[ p ].smaller_child == UNUSED )
    ContractNode( p, tree[ p ].larger_child );
  /* Si el nodo a borrar tiene ambos descendientes. */
  else {
    /* Localizamos el siguiente nodo más pequeño que el nodo que
       intentamos borrar. */
    replacement = FindNextNode( p );
    /* Eliminaos el siguiente nodo más pequeño del árbol. Nótese que
       el nodo "replacemente" nunca va a tener los dos descendientes,
       lo que evita entrar en más de un nivel de recursión. */
    DeleteString( replacement );
    /* Sustituimos el nodo que estamos intentanbo borrar por el que
       acabamos de localizar y eliminar el árbol. */
    ReplaceNode( p, replacement );
```

```
  }
}

/*
 * Esta rutina inserta un nuevo nodo al árbol binario y encuentra
 * (retornando) la mejor ocurrencia de la cadena buscada.
 */
int AddString(int new_node, int *match_position) {
  int i;
  int test_node;
  int delta;
  int match_length;
  int *child;

  /* Estamos insertando el símbolo END_OF_STREAM? */
  if ( new_node == END_OF_STREAM )
    return 0;
  /* Accedemos a la raíz del árbol y de ahí al primer nodo con datos
     almacenado. */
  test_node = tree[ TREE_ROOT ].larger_child;
  /* La longitud de la cadena encontrada es todavía, 0. */
  match_length = 0;
  for ( ; ; ) {
```

```
/* "i" contiene el "match_length" actual. */
for ( i = 0 ; i < LOOK_AHEAD_SIZE ; i++ ) {
  /* "delta" < 1 si la cadena almacenada en "new_node" es menor
     que la de "test_node", "delta" = 0 si son iguales y "delta" >
     1 si la cadena en "nee_node" es mayor que la de
     "test_node". */
  delta = window[ MOD_WINDOW( new_node + i ) ] -
    window[ MOD_WINDOW( test_node + i ) ];
  if ( delta != 0 )
    break;
}
/* Si hemos encontrado una cadena más larga. */
if ( i >= match_length ) {
  match_length = i;
  *match_position = test_node;
  /* Si hemos encontrado todo el buffer de anticipación en el
     diccionario (ya no es posible buscar una cadena más
     larga). */
  if ( match_length >= LOOK_AHEAD_SIZE ) {
    /* "new_node" reemplaza a "test_node" para manterner el árbol
       lo más pequeño posible, sin afectar a la tasa de
       compresión. */
    ReplaceNode( test_node, new_node );
```

```
        return  match_length;
      }
    }
    /* Navegación binaria sobre el árbol. */
    if ( delta >= 0 )
      child = &tree[ test_node ].larger_child;
    else
      child = &tree[ test_node ].smaller_child;
    if ( *child == UNUSED ) {
      *child = new_node;
      tree[ new_node ].parent = test_node;
      tree[ new_node ].larger_child = UNUSED;
      tree[ new_node ].smaller_child = UNUSED;
      return match_length;
    }
    test_node = *child;
  }
}

/*
 * Realiza la compresión del stream.
 */
void compress(int argc, char *argv[]) {
```

```
int i;
int c;
int look_ahead_bytes;
int current_position;
int replace_count;
int match_length;
int match_position;

/* Carga el buffer de anticipación. */
current_position = 1;
for ( i = 0 ; i < LOOK_AHEAD_SIZE ; i++ ) {
  if ( ( c = getchar() ) == EOF )
    break;
  window[ current_position + i ] = (unsigned char) c;
}
look_ahead_bytes = i; /* look_ahead_bytes = 17, excepto al final de
                         la compresión. */

/* Inicializa el árbol de búsqueda binario. */
InitTree( current_position );

/* Longitud de la cadena encontrada. */
match_length = 0;
```

```
/* Posición de la cadena encontrada. */
match_position = 0;

/* Comienza la compresión. */
while ( look_ahead_bytes > 0 ) {
  if ( match_length > look_ahead_bytes )
    match_length = look_ahead_bytes;
  /* Decidimos si enviar una "k" o un código "ij". */
  if ( match_length <= MIN_ENCODED_STRING_SIZE ) {
    /* "k": Un-encoded output. */
    bitio__put_bit(1);
    bitio__put_bits(window[ current_position ], 8);
    replace_count = 1;
  } else {
    /* "ij": Encoded output. */
    bitio__put_bit(0);
    bitio__put_bits(match_position, INDEX_SIZE);
    bitio__put_bits((match_length - (MIN_ENCODED_STRING_SIZE + 1)),
                    LENGTH_SIZE );
    replace_count = match_length;
  }
```

```
    /* Insertamos "replace_count" símbolos en el buffer de
       anticipación y los eleiminados del diccionario.*/
    for ( i = 0 ; i < replace_count ; i++ ) {
      /* Eliminamos del árbol binario de búsqueda los símbolos que
         salen por la parte izquierda de la ventana deslizante. */
      DeleteString( MOD_WINDOW( current_position + LOOK_AHEAD_SIZE ) );
      /* Leemos los nuevos símbolos. */
      if ( ( c = getchar() ) == EOF )
        look_ahead_bytes--;
      else
        window[ MOD_WINDOW( current_position + LOOK_AHEAD_SIZE ) ]
          = (unsigned char) c;
      /* Actualizamos el "puntero" por el que vamos comprimiendo el
         stream de datos. No olvidemos que lo procesamos usando una
         cola circular. */
      current_position = MOD_WINDOW( current_position + 1 );
      /* Insertamos en el árbol binario de búsqueda los nuevos
         símbolos. */
      if ( look_ahead_bytes )
        match_length = AddString( current_position, &match_position );
    }
  };
  /* EOF alcanzado. */
```

```
  bitio__put_bit(0);
  bitio__put_bits(END_OF_STREAM, INDEX_SIZE);
  bitio__flush();
}

/*
 * Realiza la descompresión del stream.
 */
void expand(int argc, char *argv[]) {
  int i;
  int current_position;
  int c;
  int match_length;
  int match_position;

  current_position = 1;
  for ( ; ; ) {
    if (bitio__get_bit()) {
      /* Leído 1, un-encoded "k". */
      c = bitio__get_bits(8);
      putchar(c);
      window[ current_position ] = (unsigned char) c;
      current_position = MOD_WINDOW( current_position + 1 );
```

```
    } else {
      /* Leído 0, "ij" code. */
      match_position = bitio__get_bits(INDEX_SIZE); /* "i" */
      if ( match_position == END_OF_STREAM )
        break;
      match_length = bitio__get_bits(LENGTH_SIZE); /* "j" */
      match_length += MIN_ENCODED_STRING_SIZE;
      /* Copiamos a la salida "j" caracteres a partir de la posición
         "i" del diccionario. */
      for ( i = 0 ; i <= match_length ; i++ ) {
        c = window[ MOD_WINDOW( match_position + i ) ];
        putchar(c);
        window[ current_position ] = (unsigned char) c;
        current_position = MOD_WINDOW( current_position + 1 );
      }
    }
  }
}
```

39.20. lzw.c

```
/*
 * lzw15v.c
 *
 * The LZW algorithm.
 *
 * Referencias:
 *
 * T. A. Welch, IEEE Computer, Vol. 17, pp 8-19. 1984.
 * M. Nelson and J.-L. Gailly, The Data Compression Book. 1995.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bitio.h"

/*
 * Número máximo de bits de un código de compresión "w".
 */
#define MAX_CODE_SIZE_IN_BITS 15
```

```
/*
 * Valor máximo que puede tomar un código de compresión.
 */
#define MAX_CODE ((1<<MAX_CODE_SIZE_IN_BITS)-1)

/*
 * Define el tamaño del diccionario. Como se utiliza hashing y cuando
 * aparece una colisión se utiliza una búsqueda secuencial, interesa
 * utilizar un número primo suficientemente grande. A continuación se
 * muestran algunas sugerencias próximas a potencias de dos:
 * 4999, 8999, 17989, 35023, 69997, 139999.
 */
#define TABLE_SIZE                  35023L

/*
 * Indica el fin de la compresión.
 */
#define END_OF_STREAM               256

/*
 * Indica ...
 */
#define BUMP_CODE                   257
```

```
/*
 * Indica un vaciado del diccionario.
 */
#define FLUSH_CODE                  258

/*
 * Primera entrada en el diccionario que almacena una cadena.
 */
#define FIRST_CODE                  259

/*
 * Entrada vacía.
 */
#define UNUSED                      -1

/*
 * La siguiente estructura de datos declara (de forma estática) el
 * diccionario. Como puede apreciarse, se trata de una tabla de ternas
 * "code_value", "parent_code" y "character", donde cada terna
 * especifica una cadena diferente. El campo "code_value" es el código
 * de compresión asociado a la cadena "parent_code""character", es
 * decir, el índice de la cadena "parent_code""character" en el
```

```
 * diccionario. Recuerdese además que "code_value"s menores que 256
 * codifican símbolos (raíces).
 */
struct dictionary {
    int code_value;
    int parent_code;
    char k;
} dict[TABLE_SIZE];

/*
 * Contiene la cadena "w" descodificada.
 */
char decode_stack[TABLE_SIZE];

/*
 * Siguiente código insertado en el diccionario.
 */
unsigned int next_w;

/*
 * Tamaño actual del código de compresión.
 */
int current_code_bits;
```

```
/*
 * Código de compresión que incrementará el tamaño del código de
 * compresión.
 */
unsigned int next_bump_code;

/*
 * Inicializa el diccionario y otras variables globales.
 */
void InitializeDictionary()
{
    unsigned int i;

    for ( i = 0 ; i < TABLE_SIZE ; i++ )
        dict[ i ].code_value = UNUSED;
    next_w = FIRST_CODE;
    current_code_bits = 9;
    next_bump_code = 511;
}

/*
 * Busca en el diccionario la cadena "wk". Se utiliza una función hash
```

```
 * que depende "w" y de "k", que se relacionan mediante la operación
 * XOR. En caso de aparecer una colisión se busca, tantas veces como
 * sea necesario, en la entrada "index*2 mod TABLE_SIZE", donde
 * "index" el la posición esperrada de la cadena "wk" en el
 * diccionario.
 */
unsigned int find_child_node(int parent_code, int child_k) {
  unsigned int index;
  unsigned int offset;

  index = ( child_k << ( MAX_CODE_SIZE_IN_BITS - 8 ) ) ^ parent_code;
  if ( index == 0 )
    offset = 1;
  else
    offset = TABLE_SIZE - index;
  for ( ; ; ) {
    if ( dict[ index ].code_value == UNUSED )
      /* Entrada vacía. */
      return index;
    if ( dict[ index ].parent_code == parent_code &&
         dict[ index ].k == (char) child_k )
      /* Cadena encontrada. */
      return index;
```

```
    /* Colisión. */
    if ( (int) index >= offset )
      index -= offset;
    else
      index += TABLE_SIZE - offset;
  }
}

/*
 * This routine decodes a string from the dictionary, and stores it
 * in the decode_stack data structure.  It returns a count to the
 * calling program of how many characters were placed in the stack.
 */

unsigned int string(unsigned int count, unsigned int w) {
  while ( w > 255 ) {
    decode_stack[ count++ ] = dict[ w ].k;
    w = dict[ w ].parent_code;
  }
  decode_stack[ count++ ] = (char) w;
  return( count );
}
```

```
/*
 * The compressor is short and simple.  It reads in new symbols one
 * at a time from the input file.  It then  checks to see if the
 * combination of the current symbol and the current code are already
 * defined in the dictionary.  If they are not, they are added to the
 * dictionary, and we start over with a new one symbol code.  If they
 * are, the code for the combination of the code and character becomes
 * our new code.  Note that in this enhanced version of LZW, the
 * encoder needs to check the codes for boundary conditions.
 */

void compress(int argc, char *argv[]) {
  int k;
  int w;
  unsigned int index;

  InitializeDictionary();
  if ((w=getchar())==EOF)
    /* Fichero de entrada vacío! */
    w = END_OF_STREAM;
  while ((k=getchar())!=EOF) {
    /* Buscamos "wk" en el diccionario. */
    index = find_child_node(w, k);
```

```
if ( dict[ index ].code_value != - 1 )
  /* "wk" está en el diccionario. */

  /* w <- dirección de "wk". */
  w = dict[ index ].code_value;
else {
  /* "wk" no está en el diccionario. */

  /* Escribimos "w" a la salida. */
  bitio__put_bits(w, current_code_bits);

  /* Insertamos "wk" en el diccionario. */
  dict[ index ].code_value = next_w++;
  dict[ index ].parent_code = w;
  dict[ index ].k = (char) k;

  /* w <- k. */
  w = k;

  if (next_w > MAX_CODE) {
    /* Vaciamos el diccionario. */
    bitio__put_bits(FLUSH_CODE, current_code_bits);
```

```
        InitializeDictionary();
      } else if (next_w > next_bump_code) {
        /* Aumentamos el tamaño del código en 1 bit. */
        bitio__put_bits(BUMP_CODE, current_code_bits);
        current_code_bits++;
        next_bump_code <<= 1;
        next_bump_code |= 1;
      }
    }
  }
  /* EOS. */
  bitio__put_bits(w, current_code_bits);
  bitio__put_bits(END_OF_STREAM, current_code_bits);
  bitio__flush();
}

/*
 * The file expander operates much like the encoder.  It has to
 * read in codes, the convert the codes to a string of characters.
 * The only catch in the whole operation occurs when the encoder
 * encounters a CHAR+STRING+CHAR+STRING+CHAR sequence.  When this
 * occurs, the encoder outputs a code that is not presently defined
 * in the table.  This is handled as an exception.  All of the special
```

```
 * input codes are handled in various ways.
 */

void expand(int argc, char *argv[]) {
  unsigned int w;
  unsigned int prev_w;
  int k;
  unsigned int count;

  for ( ; ; ) {
    InitializeDictionary();
    /* prev_w <- primer código de entrada. */
    prev_w = bitio__get_bits(current_code_bits);
    if ( prev_w == END_OF_STREAM )
      return;
    /* Escribimos prev_w a la salida. */
    putchar(prev_w);
    /* k <- prev_w. */
    k = prev_w;
    for ( ; ; ) {
      /* w <- siguiente códido de entrada. */
      w = bitio__get_bits(current_code_bits);
      /* Mientras existan códigos de entrada. */
```

```
if ( w == END_OF_STREAM )
  return;
if ( w == FLUSH_CODE )
  break;
if ( w == BUMP_CODE ) {
  current_code_bits++;
  continue;
}
if ( w >= next_w ) {
  /* Si w no está en el diccionario. */
  /* Escribir string(w)+k a la salida. */
  decode_stack[ 0 ] = (char) k;
  count = string( 1, prev_w );
}
else
  /* Si w está en el diccionario. */
  /* Escribir string(w) a la salida. */
  count = string( 0, w );
/* k <- primer símbolo emitido en la salida anterior. */
k = decode_stack[ count - 1 ];
while ( count > 0 )
  putchar(decode_stack[--count]);
/* Insertar wk en el diccionario. */
```

```
        dict[ next_w ].parent_code = prev_w;
        dict[ next_w ].k = (char) k;
        next_w++;
        /* prev_w <- w. */
        prev_w = w;
      }
    }
}
```

39.21. main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "codec.h"

int main(int argc, char *argv[]) {
  clock_t ticks;
  if(argc<1) {
    fprintf(stderr,"%s: c|d < stdin > stdout\n");
    exit(1);
  }
  if(argv[1][0]=='c') {
    fprintf(stderr,"%s: encoding ...\n", argv[0]);
    ticks = clock();
    encode_stream(argc, argv);
    ticks = clock()-ticks;
  } else {
    fprintf(stderr,"%s: decoding ...\n", argv[0]);
    ticks = clock();
    decode_stream(argc, argv);
    ticks = clock()-ticks;
```

```
  }
  float time= (float)ticks/CLOCKS_PER_SEC;
  fprintf(stderr,"%s: run time = %f seconds\n", argv[0], time);
}
```

39.22. main.h

```
void encode(int argc, char *argv[]);
void decode(int argc, char *argv[]);
```

39.23. model_0.c

```
/*
 * model_0.c
 *
 * Un modelo probabilístico de orden 0.
 *
 * Referencias:
 *
 * Witten, Neal, and Cleary, CACM, 1987.
 * M. Nelson and J.-L. Gailly, The Data Compression Book. 1995.
 */

#include <stdio.h>
#include "vlc.h"
#include "codec.h"

/* Tamaño del alfabeto fuente. Los 256 caracteres posibles y el
   símbolo EOS (End Of Stream). */
#define ALPHA_SIZE 257

#include "model_0.h"
```

```
#include "model_0/find_symbols_and_indexes.h"
//#include "model_0/compute_cumulative_probs.h"
#include "model_0/init_model.h"
#include "model_0/scale_probs.h"
#include "model_0/test_if_scale.h"
#include "model_0/increment_prob_of_index.h"
#include "model_0/update_model.h"
#include "model_0/finish_model.h"
#include "model_0/encode_stream.h"
#include "model_0/decode_stream.h"
```

39.24. model_0.h

```
/* Código de compresión que indica el fin del stream de datos. Su
   posición dentro del alfabeto fuente será siempre al final del
   mismo. */
#define EOS (ALPHA_SIZE-1)

/* Máximo recuento acumulado permitido. Este valor afecta a la
   precisión del modelo probabilístico a la hora de calcular las
   probabilidades de los símbolos. */
#define MAX_CUM_COUNT 16383

/* Probabilidad (en forma de recuento) de los índices. Cada índice
   está asociado a un símbolo diferente, cumpliéndose que el índice 0
   no se pude usar para ningún símbolo aunque debe estar definido
   cumpliéndose siempre que el recuento para el índice 0 debe ser
   siempre 0 (este es un requerimiento del codificador aritmético que
   estamos usando). Por tanto, si existen ALPHA_SIZE símbolos
   diferentes, existen ALPHA_SIZE+1 índices distintos. Nótese además
   que el tipo de dato asociado se escoge en relación con el valor
   MAX_CUM_COUNT. */
static unsigned short prob[ALPHA_SIZE+1];
```

```
/* Recuentos acumulados de los índices. El codificador aritmético
   necesita que la entrada cum_prob[0] almacene el recuento acumlado
   de todos los símbolos. Nótese además que el tipo de dato asociado
   se escoge en relación con el valor MAX_CUM_COUNT. */
static unsigned short cum_prob[ALPHA_SIZE+1];

/* Símbolo codificado. */
static int symbol;

/* Indice del símbolo codificado. */
static int index;
```

39.25. model_0/compute_cumulative_probs.h

```
/* Dadas las probabilidades de los símbolos en "prob", calcula las
   probabilidades acumuladas en "cum_prob". */
void compute_cumulative_probs() {
  int i;
  int cum = 0;
  for(i=ALPHA_SIZE; i>=0; i--) {
    cum_prob[i] = cum;
    cum += prob[i];
  }
}
```

39.26. model_0/decode_stream.h

```
/* Descodifica el stream. */
void decode_stream(int argc, char *argv[]) {
  init_model();
  init_decoder();
  for(;;) {
    index = decode_index(cum_prob);
    symbol = find_symbol(index);
    if(symbol==EOS) break;
    //fprintf(stderr,"%d ",symbol);
    putchar(symbol);
    update_model();
  }
  finish_decoder();
  finish_model();
}
```

39.27. model_0/encode_stream.h

```
/* Codifica el stream. */
void encode_stream(int argc, char *argv[]) {
  init_model();
  init_encoder();
  for(;;) {
    symbol = getchar();
    //fprintf(stderr,"%d ",symbol);
    if(symbol==EOF) break;
    index = find_index(symbol);
    encode_index(index, cum_prob);
    update_model();
  }
  encode_index(find_index(EOS), cum_prob);
  finish_encoder();
  finish_model();
}
```

39.28. model_0/find_symbols_and_indexes.h

```
/* Convierte un símbolo en un índice. */
int find_index(int symbol) {
  return symbol+1;
}

/* Convierte un índice en un símbolo. */
int find_symbol(int index) {
  return index-1;
}
```

39.29. model_0/finish_model.h

```
/* Finaliza el modelo probabilístico. */
void finish_model() {
  fprintf(stderr,"\n");
}
```

39.30. model_0/increment_prob_of_index.h

```
/* Incrementa la probabilidad del símbolo asociado a "index". */
void increment_prob_of_index() {
  prob[index]++;
  while(index>0) {
    cum_prob[--index]++;
  }
}
```

39.31. model_0/init_model.h

```
/* Inicializa el modelo probabilistico. Todos los símbolos son,
   inicialmente, equiprobables. */
void init_model() {
  int i;
  for(i=0; i<ALPHA_SIZE; i++) {
    prob[find_index(i)] = 1;
  }
  prob[0] = 0;
#include "compute_cumulative_probs.h"
  compute_cumulative_probs();
}
```

39.32. model_0/scale_probs.h

```
/* Escala las probabilides de los símbolos dividiendo entre dos sus
   recuentos, redondeando al entero superior más cercano. */
void scale_probs() {
  int i;
  for (i = ALPHA_SIZE; i>=0; i--) {
    prob[i] = (prob[i]+1)/2;
  }
  fprintf(stderr,"S");
}
```

39.33. model_0/test_if_scale.h

```
/* Comprueba si hay que escalar los recuentos de los símbolos, y si es
   así, lo hace. */
void test_if_scale() {
  if (cum_prob[0]==MAX_CUM_COUNT) {
    scale_probs();
#include "compute_cumulative_probs.h"
    compute_cumulative_probs();
  }
}
```

39.34. model_0/update_model.h

```
/* Actualiza el modelo incrementando el recuento del símbolo
   asociado a "index". */
void update_model() {
  test_if_scale();
  increment_prob_of_index();
}
```

39.35. model_0s.c

```
/*
 * model_0s.c
 *
 * Un modelo probabilístico de orden 0, más eficiente.
 *
 * Referencias:
 *
 * model_0.c
 * Witten, Neal, and Cleary, Arithmetic Coding For Data Compression,
 *   CACM, 1987.
 * M. Nelson and J.-L. Gailly, The Data Compression Book. 1995.
 */
#include <stdio.h>
#include "vlc.h"
#include "codec.h"

/* Tamaño del alfabeto fuente. Los 256 caracteres posibles y el
   símbolo EOS (End Of Stream). */
#define ALPHA_SIZE 257

#include "model_0.h"
```

```
#include "model_0s.h"

#include "model_0s/find_symbols_and_indexes.h"
#include "model_0s/init_model.h"
#include "model_0/scale_probs.h"
#include "model_0/test_if_scale.h"
#include "model_0/increment_prob_of_index.h"
#include "model_0s/find_new_position_for.h"
#include "model_0s/update_model.h"
#include "model_0/finish_model.h"
#include "model_0/encode_stream.h"
#include "model_0/decode_stream.h"
```

39.36. model_0s.h

```
/* Traducción entre símbolos e índices. Los índices se mantienen
   ordenados por frecuencia de ocurrencia lo que minimiza el tiempo
   de descompresión. */
int _symbol_to_index[ALPHA_SIZE];
int _index_to_symbol[ALPHA_SIZE+1];
```

39.37. model_0s/find_new_position_for.h

```
/* Encuentra la nueva posición de un índice en función de su recuento
   y lo coloca ahí. Dicha posición es devuelta. */
int find_new_position_for(int index) {
  int i;
  for(i = index; prob[i]==prob[i-1]; i--);
  if (i<index) {
    int ch_i, ch_symbol;
    ch_i = _index_to_symbol[i];
    ch_symbol = _index_to_symbol[index];
    _index_to_symbol[i] = (unsigned char) ch_symbol;
    _index_to_symbol[index] = (unsigned char) ch_i;
    _symbol_to_index[ch_i] = index;
    _symbol_to_index[ch_symbol] = i;
  }
  return i;
}
```

39.38. model_0s/find_symbols_and_indexes.h

```
int find_index(int symbol) {
  return _symbol_to_index[symbol];
}

int find_symbol(int index) {
  return _index_to_symbol[index];
}
```

39.39. model_0s/init_model.h

```
#include "init_index_symbol_conversion.h"

void init_model() {
  init_index_symbol_conversion();
#include "../model_0/init_model.h"
  init_model();
}
```

39.40. model_0s/update_model.h

```
/* Actualiza el modelo probabilístico. */
void update_model() {
  test_if_scale();
  index = find_new_position_for(index);
  increment_prob_of_index();
}
```

39.41. model_1e.c

```
/*
 * model_1e.c
 *
 * Un modelo probabilístico de orden 1, inicialmente vacío.
 *
 * Referencias:
 *
 * model_0.c
 * model_0s.c
 * model_1s.c
 * Witten, Neal, and Cleary, CACM, 1987.
 * M. Nelson and J.-L. Gailly, The Data Compression Book. 1995.
 */

#include <stdio.h>
#include "vlc.h"
#include "codec.h"

#include "model_1e.h"

#include "model_0/find_symbols_and_indexes.h"
```

```
#define _symbol_to_index _symbol_to_index[context]
#define _index_to_symbol _index_to_symbol[context]
#define prob prob[context]
#define cum_prob cum_prob[context]
#include "model_1e/scale_probs.h"
#include "model_0/increment_prob_of_index.h"
#include "model_0/test_if_scale.h"
#include "model_0/update_model.h"
#undef _index_to_symbol
#undef _symbol_to_index
#undef cum_prob
#undef prob

#include "model_1e/init_model.h"
#include "model_0/finish_model.h"

void encode_stream() {
  context = ESC;
  init_model();
  init_encoder();
  for(;;) {
    symbol = getchar();
```

```
    if (symbol==EOF) break;
    index = find_index(symbol);
    if(prob[context][index]) {
      /* Si el símbolo ha aparecido antes en ese contexto, simplemente
         lo codificamos según la distribución de probabilidad de ese
         contexto. */
      encode(index, cum_prob[context]);
    } else {
      /* Enviamos un ESC, en el contexto correspondiente. */
      encode(ESC_index, cum_prob[context]);
      /* Enviamos el nuevo símbolo, en el contexto del ESC. */
      index = find_index(symbol);
      encode(index, cum_prob[ESC]);
    }
    update_model();
    context = symbol;
  }
  encode(EOS_index, cum_prob[context]);
  finish_encoder();
  finish_model();
}

void decode_stream() {
```

```
  context = ESC;
  init_model();
  init_decoder();
  for(;;) {
    index = decode(cum_prob[context]);
    if(index==EOS_index) break;
    symbol = find_symbol(index);
    if(symbol==ESC) {
      index = decode(cum_prob[ESC]);
      symbol = find_symbol(index);
    }
    putchar(symbol);
    update_model();
    context = symbol;
  }
  finish_decoder();
  finish_model();
}
```

39.42. model_1e.h

```
/* Tamaño del alfabeto fuente. 256 caracteres, ESC y EOS. */
#define ALPHA_SIZE 258

#include "model_1s.h"

/* Símbolo de escape que indica la aparición de un nuevo símbolo en el
   contexto. */
#define ESC/* ape */ (ALPHA_SIZE-2)
#define ESC_index (ESC+1)
#define EOS_index (EOS+1)
```

39.43. model_1e/init_model.h

```
void init_model() {
  int i;
  for(context=0; context<ALPHA_SIZE-1; context++) {
    /* Ahora, en cada posible contexto, sólo el ESC tiene un recuento
       diferente de cero. */
    for(i=0; i<ALPHA_SIZE+1; i++) {
      prob[context][i] = 0;
      cum_prob[context][i] = 0;
    }
    index = ESC_index;
    update_model();
    index = EOS_index;
    update_model();
  }
  /* Contexto del ESC. Todos los recuentos = 1. */
  for(i=0; i<ALPHA_SIZE; i++) {
    prob[ESC][find_index(i)] = 1;
  }
  context = ESC;
  prob[context][0] = 0;
```

```
#define _symbol_to_index _symbol_to_index[context]
#define _index_to_symbol _index_to_symbol[context]
#define prob prob[context]
#define cum_prob cum_prob[context]
#include "../model_0/compute_cumulative_probs.h"
  compute_cumulative_probs();
#undef _index_to_symbol
#undef _symbol_to_index
#undef cum_prob
#undef prob
}
```

39.44. model_1e/scale_probs.h

```
/* Escala los recuentos de los símbolos, esta vez redondeando al
   entero inferior más cercano. */
void scale_probs() {
  int i;
  for (i = ALPHA_SIZE; i>=0; i--) {
    prob[i] = prob[i]/2;
  }
  fprintf(stderr,"S");
}
```

39.45. model_1ee.c

```
/*
 * model_1e.c
 *
 * Un modelo probabilístico de orden 1, inicialmente vacío, con
 * exclusión de símbolos.
 *
 * Referencias:
 *
 * model_0.c
 * model_0s.c
 * model_1s.c
 * model_1e.c
 * Witten, Neal, and Cleary, CACM, 1987.
 * M. Nelson and J.-L. Gailly, The Data Compression Book. 1995.
 */

#include <stdio.h>
#include "vlc.h"
#include "codec.h"
#include "model_1e.h"
```

```
#include "model_0/find_symbols_and_indexes.h"

void compute_cumulative_probs(unsigned short *prob,
                              unsigned short *cum_prob) {
  int i;
  int cum = 0;
  for(i=ALPHA_SIZE; i>=0; i--) {
    cum_prob[i] = cum;
    cum += prob[i];
  }
}

#define _symbol_to_index _symbol_to_index[context]
#define _index_to_symbol _index_to_symbol[context]
#define prob prob[context]
#define cum_prob cum_prob[context]
#include "model_1e/scale_probs.h"
#include "model_0/increment_prob_of_index.h"
#include "model_0/test_if_scale.h"
#include "model_0/update_model.h"
#undef _index_to_symbol
#undef _symbol_to_index
#undef cum_prob
```

```
#undef prob

#include "model_1e/init_model.h"
#include "model_0/finish_model.h"

/* Tiene en cuenta los símbolos que han aparecido en "context" para
   calcular las probabilidades acumuladas en el contexto "ESC". */
void exclude_symbols(int context,
                     unsigned short *cum_prob) {
  unsigned short new_prob[ALPHA_SIZE+1];
  int i;
  for(i=0; i<ALPHA_SIZE+1; i++) {
    new_prob[i] = prob[ESC][i];
    if(prob[context][i]) {
      new_prob[i] = 0;
    }
  }
  compute_cumulative_probs(new_prob, cum_prob);
}

void encode_stream() {
  context = ESC;
  init_model();
```

```
init_encoder();
for(;;) {
  symbol = getchar();
  if (symbol==EOF) break;
  index = find_index(symbol);
  if(prob[context][index]) {
    encode(index, cum_prob[context]);
  } else {
    unsigned short new_cum_prob[ALPHA_SIZE+1];
    encode(ESC_index, cum_prob[context]);
    index = find_index(symbol);
    /* Calculamos un nuevo espacio de probabilidades, sabiendo que
       ninguno de los símbolos contemplados por el el contexto
       "context" es el símbolo a codificar "symbol". De esta forma,
       incrementamos la probabilidad del símbolo codificado. */
    exclude_symbols(context, new_cum_prob);
    encode(index, new_cum_prob);
  }
  update_model();
  context = symbol;
}
encode(EOS_index, cum_prob[context]);
finish_encoder();
```

```
  finish_model();
}

void decode_stream() {
  context = ESC;
  init_model();
  init_decoder();
  for(;;) {
    index = decode(cum_prob[context]);
    if(index==EOS_index) break;
    symbol = find_symbol(index);
    if(symbol==ESC) {
      unsigned short new_cum_prob[ALPHA_SIZE+1];
      exclude_symbols(context, new_cum_prob);
      index = decode(new_cum_prob);
      symbol = find_symbol(index);
    }
    putchar(symbol);
    update_model();
    context = symbol;
  }
  finish_decoder();
  finish_model();
```

```
}
```

39.46. model_1s.c

```
/*
 * model_1s.c
 *
 * Un modelo probabilístico de orden 1.
 *
 * Referencias:
 *
 * model_0.c
 * model_0s.c
 * Witten, Neal, and Cleary, Arithmetic Coding For Data Compression,
 *   CACM, 1987.
 * M. Nelson and J.-L. Gailly, The Data Compression Book. 1995.
 */

#include <stdio.h>
#include "vlc.h"
#include "codec.h"

/* Tamaño del alfabeto fuente. Los 256 caracteres posibles y el
   símbolo EOS (End Of Stream). */
#define ALPHA_SIZE 257
```

```
/* El código de model_1s.c es fundamentalmente el mismo que en
   model_0s.c, excepto que ahora se manejan ALPHA_SIZE-1 contextos
   (símbolos). */

#include "model_1s.h"

#define _symbol_to_index _symbol_to_index[context]
#define _index_to_symbol _index_to_symbol[context]
#define prob prob[context]
#define cum_prob cum_prob[context]

#include "model_0s/find_symbols_and_indexes.h"

void init_model() {
  /* Para cada posible contexto. */
  for(context=0; context<ALPHA_SIZE-1; context++) {
#include "model_0s/init_model.h"
    init_model();
  }
  context = 0;
}
```

```
#include "model_0/scale_probs.h"
#include "model_0s/find_new_position_for.h"
#include "model_0/increment_prob_of_index.h"
#include "model_0/test_if_scale.h"
#include "model_0/finish_model.h"

void update_model() {
#include "model_0s/update_model.h"
  update_model();
  context = symbol;
}

#include "model_0/encode_stream.h"
#include "model_0/decode_stream.h"

#undef _index_to_symbol
#undef _symbol_to_index
#undef cum_prob
#undef prob
```

39.47. model_1s.h

```
#define prob prob[ALPHA_SIZE-1]
#define cum_prob cum_prob[ALPHA_SIZE-1]
#define _symbol_to_index _symbol_to_index[ALPHA_SIZE-1]
#define _index_to_symbol _index_to_symbol[ALPHA_SIZE-1]
#include "model_0.h"
#include "model_0s.h"
#undef _index_to_symbol
#undef _symbol_to_index
#undef cum_prob
#undef prob

static int context;
```

39.48. mallok.h

```
/*
void  ** malloc__alloc_2d(int y, int x, int size);
void *** malloc__alloc_3d(int z, int y, int x, int size);
void      malloc__free_2d(void **h, int y);
void      malloc__free_3d(void ***h, int z, int y);

*/

class mallok {

 public:
  void *alloc_1d(int x, int size);
  void **alloc_2d(int y, int x, int size);
  void ***alloc_3d(int z, int y, int x, int size);

  void free_1d(void *h);
  void free_2d(void **h);
  void free_3d(void ***h);
};
```

39.49. mallok.cpp

```
#include <stdlib.h> /* malloc(), free(), abort() */
#include <string>
#include <sstream>
#if defined TEST | defined DEBUG
#include <iostream>
#endif
using namespace std;
#include "mallok.h"

void *mallok::alloc_1d(int x, int size) {
  void *h = malloc(x*size);
#if defined DEBUG
  cerr << "mallok::allok_1d: x=" << x << " size=" << size << '\n';
  cerr << "mallok::allok_1d: h=" << h << '\n';
#endif
#if defined TEST
  if(!h) {
    cerr << "mallok::allok_1d: out of memory (" << x << "," << size << ")";
    abort();
  }
```

```
#endif
  return h;
}

void mallok::free_1d(void *h) {
#if defined DEBUG
  cerr << "mallok::free_1d(" << h << ")\n";
#endif
  free(h);
}

void **mallok::alloc_2d(int y, int x, int size) {
  void **h = (void **)mallok::alloc_1d(y, sizeof(void *));
  h[0] = (void *)mallok::alloc_1d(y*x,size);
  int i;
  for(i=1; i<y; i++) {
    h[i] = (unsigned char *)h[i-1] + x*size;
  }
  return h;
}

void mallok::free_2d(void **h) {
  free_1d(h[0]);
```

```
  free_1d((void *)h);
}

void ***mallok::alloc_3d(int z, int y, int x, int size) {
  void ***h = (void ***)mallok::alloc_2d(z,y,sizeof(void *));
  h[0][0] = (void *)mallok::alloc_1d(z*y*x,size);
  int i;
  for(i=1; i<z*y; i++) {
    h[i/y][i%y] = (unsigned char *)h[0][i-1] + x*size;
  }
  return h;
}

void mallok::free_3d(void ***h) {
  free_1d(h[0][0]);
  free_2d((void **)h);
}

//#define _MAIN_

#ifdef _MAIN_
int main() {
  mallok m;
```

```
  m.alloc_1d(1,1);
}
#endif
```

39.50. mtf.c

```
/*
 * mtf.c
 *
 * Move-To-Front encoding.
 *
 * Referencias:
 *
 * "A Locally Adaptive Data Compression Scheme" by J. L. Bentley,
 * D. D. Sleator, R. E. Tarjan, V. K. Wei, Communications of the
 * ACM-Vol. 29, No. 4, 1986
 */

#include <stdio.h>
#include "codec.h"

unsigned char order[ 256 ];

void encode_stream(int argc, char *argv[]) {
  int i, c, j;
  for ( i = 0 ; i < 256 ; i++ )
    order[ i ] = (unsigned char) i;
```

```
  while ( ( c = getchar() ) >= 0 )  {
    //
    // Find the char, and output it
    //
    for ( i = 0 ; i < 256 ; i++ )
      if ( order[ i ] == ( c & 0xff ) )
        break;
    putchar( (char) i);
    //
    // Now shuffle the order array
    //
    for ( j = i ; j > 0 ; j-- )
      order[ j ] = order[ j - 1 ];
    order[ 0 ] = (unsigned char) c;
  }
}

void decode_stream(int argc, char *argv[]) {
  int i, j, c;
  for ( i = 0 ; i < 256 ; i++ )
    order[ i ] = (unsigned char) i;
  while ( ( i = getchar() ) >= 0 )  {
    //
```

```
    // Find the char
    //
    putchar( order[ i ] );
    c = order[ i ];
    //
    // Now shuffle the order array
    //
    for ( j = i ; j > 0 ; j-- )
      order[ j ] = order[ j - 1 ];
    order[ 0 ] = (unsigned char) c;
  }
}
```

39.51. pbt.c

```
/*
 * PPM text predictive coding.
 * gse. 1999.
 */

/*
  Respecto de la versi'on anterior, una ventana deslizante es definida con el
  objetivo de reducir el consumo de memoria.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "codec.h"

#define UCHAR  unsigned char
#define SYMBOL unsigned char
#define ORDER  unsigned char
#define CODE   unsigned char
```

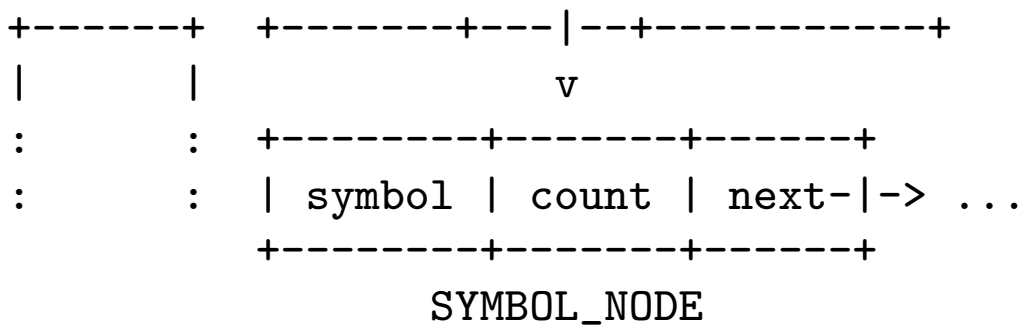
```
#ifdef _HASH_INFO_
int num_colisiones=0;
int num_tablas=0;
#endif

#ifdef _INFO_PREDICTION_
FILE *fo;
#endif

/*************************************************/
/* M A N E J O   D E   L A   T A B L A   H A S H */
/*************************************************/

/* La estructura de datos utilizada para manejar los contextos es la siguient
   Tenemos una tabla hash, en la que cada entrada es un puntero a un contexto
   Cada contexto se compone de 4 componentes: (1) el orden del contexto,
   (2) un puntero a una lista enlazada que almacena el contexto mediante
   pares s'imbolo/recuento y (3) el array de caracteres que almacenan el
   contexto. Gra'ficamente:

  hash_table       CONTEXT_TABLE
   +------+  +-------+-----+-----------+
   | ptr -|->| order | list | context[] |
```




```
   La tabla hash implementada permite la ocurrencia de colisiones. Cuando
   una ocurre (dos o m'as contextos comparten la misma posici'on dentro de
   la tabla) se realiza una b'usqueda secuencial a partir de la direcci'on
   de colisi'on. El n'umero de b'usquedas secuenciales m'aximo permitido se
   declara en la constante MAX_COLLISIONS. La tabla hash es est'atica, ya que
   no se permite que el n'umero de contextos aumente si la tabla est'a
   demasiado llena (esto ser'ia lo ideal, pero por ahora no se hace). El
   n'umero m'aximo de contextos permitidos es MAX_CONTEXTS. */

struct SYMBOL_NODE {
  UCHAR symbol;
  UCHAR count;
  struct SYMBOL_NODE *next;
};

typedef struct {
```

```
  UCHAR order;
#ifdef _1_
  unsigned long time;      /* Instante de la 'ultima actualizaci'on */
#endif
  struct SYMBOL_NODE *list;
} CONTEXT_TABLE;

CONTEXT_TABLE *hash_table[65537][32];

/* En esta tabla se indica si un s'imbolo ya ha sido visitado en contextos
   de orden superior durante la b'usqueda del s'imbolo a codificar. As'i,
   se acelera el proceso de b'usqueda. */
unsigned long file_position=0;
unsigned long visited[256];
#ifdef _1_
#define WINDOW_SIZE 65536
#endif

#ifdef _MEMORY_INFO_
int num_contextos=0;
long memory_used=0;
#define MALLOC(a) malloc(a); memory_used += a;
#else
```

```
#define MALLOC(a) malloc(a)
#endif

/* Inicializa la tabla del contexto de orden 0 */
void initialize_0_order_context_table() {
  int i;
  struct SYMBOL_NODE *node;
  hash_table[0][0]=MALLOC(sizeof(CONTEXT_TABLE));
  hash_table[0][0]->order=0;
  node=hash_table[0][0]->list=MALLOC(sizeof(struct SYMBOL_NODE));
  for(i=1;i<256;i++) {
    node->next=MALLOC(sizeof(struct SYMBOL_NODE));
    node=node->next;
    node->symbol=(unsigned char)(i/*+32*/);
    node->count=0;
  }
}

#ifdef _1_
void free_list(struct SYMBOL_NODE *s) {
  struct SYMBOL_NODE *old;
  while(s!=NULL) {
    old=s;
```

```
    s=s->next;
    free(old);
  }
}
#endif

inline void create_context(int table, int pos, char *context, UCHAR order) {
  hash_table[table][pos]=MALLOC(sizeof(CONTEXT_TABLE)+order);
  strncpy((char *)hash_table[table][pos]+sizeof(CONTEXT_TABLE),context,order)
  hash_table[table][pos]->list=NULL;
  hash_table[table][pos]->order=order;
#ifdef _1_
  hash_table[table][pos]->time=file_position;
#endif
}

/* Busca el contexto "context" de orden "order" en la tabla hash. Si lo
   encuentra devuelve su posici'on mediante un puntero, y sino, se crea
   una entrada en la tabla y la estructura CONTEXT_TABLE asociada al
   contexto. Ahora si el contexto no ha sido usado al menos WINDOW_SIZE
   s'imbolos, el contexto se libera y se reutiliza. */
CONTEXT_TABLE *locate(char *context, UCHAR order) {
  int table;
```

```
  unsigned long addr;
  for(table=0;table<32;table++) {
    int i;
    addr=0;
    for(i=0;i<order;i++) addr ^= context[i]<<(i*(table+1));
    addr %= 65537;
    if(hash_table[addr][table]==NULL) {
      create_context(addr,table,context,order);            /* Creamos un marc
#ifdef _HASH_INFO_
       num_contextos++;
       if(num_tablas<table) num_tablas=table;
#endif
      return hash_table[addr][table];           /* Retornamos su direcci'on *
    }
    if(hash_table[addr][table]->order!=order) {
#ifdef _HASH_INFO_
      num_colisiones++;
#endif
      continue;
    }
    if(strncmp((char *)hash_table[addr][table]+sizeof(CONTEXT_TABLE),context,
#ifdef _HASH_INFO_
      num_colisiones++;
```

```
#endif
      continue;
    }
    return hash_table[addr][table];
  }
  fprintf(stderr,"tpc-ppm: tablas hash llenas\n");
  exit(1);
}

/*****************************************************************/
/* M A N E J O   D E   L A S   L I S T A S   D E   C O N T E X T O S */
/*****************************************************************/

/* Escala un contexto dividiendo todos los recuentos entre 2. */
void scale_context(CONTEXT_TABLE *p) {
  struct SYMBOL_NODE *s=p->list;
  while(s!=NULL) {
    s->count >>= 1;
    s=s->next;
  }
}

/* M'aximo recuento acumulado permitido para un s'imbolo. Este valor es
```

```
   importante para el nivel entr'opico de salida alcanzable debido a que
   controla la frecuencia de escalados en los contextos. */
#define MAX_COUNT     255

UCHAR search_update_context(UCHAR symbol, char *context, UCHAR order) {
  CONTEXT_TABLE *context_table=locate(context,order);
  UCHAR code=0;
  int found=0;
  struct SYMBOL_NODE *s,*r;
  if(context_table->list==NULL) { /* Lista vac'ia ? */
    context_table->list=MALLOC(sizeof(struct SYMBOL_NODE));
    context_table->list->symbol=symbol;
    context_table->list->count=0;
    context_table->list->next=NULL;
  } else {
    s=context_table->list;
#ifdef _INFO_PREDICTION_
    putc(s->symbol,fo);
#endif
    while(s!=NULL) {
      if(visited[s->symbol]!=file_position) {
        visited[s->symbol]=file_position;
        if(s->symbol==symbol) { found=1; break; }
```

```
        code++;
      }
      r=s; s=s->next;
    }
    if(found) {
      if(s->count==MAX_COUNT) {
#ifdef _SCALING_INFO_
        fprintf(stderr,"S-%d ",order);
#endif
        scale_context(context_table);
      }
#ifndef _NO_COUNTS_
      s->count++;
#endif
      if(s!=context_table->list) { /* Si "s" no es el primer nodo de la lista
        struct SYMBOL_NODE *h,*g;
        if(r->count<=s->count) { /* Si la lista no est'a ordenada */
          h=context_table->list;
          while(h->count>s->count) { g=h; h=h->next; }
          if(h!=context_table->list) { /* Si la nueva posici'on no es la cabe
            g->next=s;
          } else { /* pero si s'i lo es */
            context_table->list=s;
```

```
        }
        r->next=s->next;
        s->next=h;
      }
    }
  } else { /* Si el nodo no est'a en la lista, lo a~nadimos por el final */
    struct SYMBOL_NODE *h,*g;
    s=MALLOC(sizeof(struct SYMBOL_NODE));
    s->symbol=symbol;
    s->count=0;
    h=context_table->list;
    while(h->count) {
      g=h; h=h->next;
      if(h==NULL) break;
    }
    if(h!=context_table->list) {
      g->next=s;
      s->next=h;
    } else {
      s->next=h;
      context_table->list=s;
    }
  }
```

```
  }
  return code;
}

/* Creates a new context "c" with the symbol "symbol". */
void CreateContext(CONTEXT_TABLE *c, UCHAR symbol) {
  c->list=MALLOC(sizeof(struct SYMBOL_NODE));
  c->list->symbol=symbol;
  c->list->count=0;
  c->list->next=NULL;
}

/* Update the symbol's count of a symbol in a context table. The symbol must
   exits in the context table.
*/
void UpdateSymbol(CONTEXT_TABLE *ct, SYMBOL symbol) {
  struct SYMBOL_NODE *s=ct->list,*r;
  while(symbol!=s->symbol) {r=s; s=s->next;} /* Searching */
  if(s->count==MAX_COUNT) /*ScaleContext*/scale_context(ct);
  s->count++;
  if(s!=ct->list) { /* If the updated symbol isn't on the top of the list */
    struct SYMBOL_NODE *h,*g;
    if(r->count<=s->count) { /* If the list isn't sorted */
```

```
      h=ct->list; /* Looking for the new position in the list */
      while(h->count>s->count) { g=h; h=h->next; }
      if(h!=ct->list) g->next=s; else ct->list=s;
      r->next=s->next;
      s->next=h;
    }
  }
}

/* Add a new symbol to a context table.
*/
void AddSymbolToContext(CONTEXT_TABLE *ct, SYMBOL symbol) {
  if(ct->list==NULL) CreateContext(ct,symbol);
  else { /* Insert the symbol at the begin of the 0-count sub-list */
    struct SYMBOL_NODE *s=ct->list,*r,*new;
    if(s->count) {
      while((s!=NULL) && (s->count)) {
        r=s; s=s->next;
      }
      new=MALLOC(sizeof(struct SYMBOL_NODE));
      new->symbol=symbol;
      new->count=0;
      new->next=s;
```

```
      r->next=new;
    } else {
      new=MALLOC(sizeof(struct SYMBOL_NODE));
      new->symbol=symbol;
      new->count=0;
      new->next=s;
      ct->list=new;
    }
  }
}

/* Retorna el s'imbolo colocado en la posici'on "code" en la tabla de
   contexto "ct". Si no se encuentra, devuelve -1. En cualquier caso,
   "code" se decrementa en cada b'usqueda.
*/
int FindSymbol(int *code, CONTEXT_TABLE *ct) {
  SYMBOL symbol;
  int found=0;
  struct SYMBOL_NODE *s=ct->list;
  while(s!=NULL) { /* loops inside of a context table */
    if(visited[s->symbol]!=file_position) {
      if((*code)==0) {symbol=s->symbol; found=1; break;}
      visited[s->symbol]=file_position;
```

```
      (*code)--;
    }
    s=s->next;
  }
  if(found) return symbol; else return -1;
}

#ifdef _INFO_
char filtro(char ch) {
  if(ch<32) ch='.';
  return ch;
}

void print_context_table(char *context, UCHAR order) {
  CONTEXT_TABLE *ct=locate(context,order);
  struct SYMBOL_NODE *s=ct->list;
  int c=0;
  while(s!=NULL) {
    fprintf(stderr,"%c,%d ",filtro(s->symbol),s->count);
    s=s->next;
    if(c>8) break;
    c++;
  }
```

```
}
#endif

int i;
int symbol;        /* S'imbolo a codificar */
int code;          /* Error de predicci'on */
int order;         /* Orden actual de predicci'on */
ORDER max_order;     /* M'aximo orden permitido */
SYMBOL *context;     /* Contexto actual */

void encode_stream(int argc, char *argv[]) {
  max_order=atoi(argv[2]);
  fprintf(stderr,"tpc-ppm: prediction order: %d\n",max_order);
  context=MALLOC(sizeof(UCHAR)*max_order);
  if(!context) {
    fprintf(stderr,".");
    fprintf(stderr,"tpc-ppm: sin memoria para el vector contexto\n");
    exit(1);
  }

#ifdef _INFO_PREDICTION_
  /* Output error-prediction file */
  fo=fopen("error","wb");
```

```
  if(!fo) {
    fprintf(stderr,"Unable to open \"error\"\n");
    exit(-1);
  }
#endif

  /* Inicializamos el vector contexto */
  for(i=max_order-1;i>=0;i--) {
    context[i]=getchar();
    putchar(context[i]);
    file_position++;
  }

  initialize_0_order_context_table();

  fprintf(stderr,"tpc-ppm: coding ...\n");
  symbol=getchar(); file_position++;
  while(symbol!=EOF) {
    code=0;
    order=max_order;
    for(;;) {
      code += search_update_context(symbol,context,order);
      if(visited[symbol]==file_position) break;
```

```
      order--;
    }
    putchar(code);

#ifdef _INFO_
    {
      int i;
      fprintf(stderr,"%3d(%c) %3d %2d |",symbol,filtro(symbol),code,order);
      for(i=max_order-1;i>=0;i--) {
        fprintf(stderr,"%c",filtro(context[i]));
      }
      fprintf(stderr,"|");
      for(i=0;i<=max_order;i++) {
        print_context_table(context,i);
        fprintf(stderr,"|");
      }
      fprintf(stderr,"\n");
    }
#endif
#ifdef _MEMORY_INFO_
    {
      int counter;
      if(!(counter++%256)) {
```

```
        fprintf(stderr,"tpc-ppm: memoria: %ld\n",memory_used);
      }
    }
#endif
#ifdef _MEMORY_INFO_
    {
      int counter;
      if(!(counter++%256)) {
        fprintf(stderr,"tpc-ppm: contextos: %d\n",num_contextos);
      }
    }
#endif
    /* Actualizamos la cadena con el nuevo contexto */
    for(i=max_order-1;i>0;i--) context[i]=context[i-1];
    context[0]=symbol;
    symbol=getchar();
    file_position++;
    if(!file_position) memset(visited,1,256*4);
  }

#ifdef _HASH_INFO_
  fprintf(stderr,"tpc-ppm: n'unmero de contextos: %d\n",num_contextos);
  fprintf(stderr,"pmi: N'umero de tablas hash usadas: %d\n",num_tablas);
```

```
  fprintf(stderr,"pmi: N'umero de colisiones: %d\n",num_colisiones);
#endif
#ifdef _MEMORY_INFO_
  fprintf(stderr,"tpc-ppc: memoria consumida: %ld bytes\n",memory_used);
#endif
  fprintf(stderr,"tpc-ppm: done\n");

}

void decode_stream(int argc, char *argv[]) {
  max_order=atoi(argv[2]);
  fprintf(stderr,"tpc-ppm: prediction order: %d\n",max_order);
  context=MALLOC(sizeof(UCHAR)*max_order);
  if(!context) {
    fprintf(stderr,".");
    fprintf(stderr,"tpc-ppm: sin memoria para el vector contexto\n");
    exit(1);
  }

  /* Inicializamos el vector contexto */
  for(i=max_order-1;i>=0;i--) {
    context[i]=getchar();
    putchar(context[i]);
```

```
    file_position++;
  }

  initialize_0_order_context_table();

  fprintf(stderr,"tpc-ppm: decoding ...\n");
  code=getchar(); file_position++;
  while(code!=EOF) {
    order=max_order;
    for(;;) {
      CONTEXT_TABLE *ct=locate(context,order);
      symbol=FindSymbol(&code,ct);
      if(symbol!=-1) {
        UpdateSymbol(ct,symbol);
        break;
      }
      order--;
    }
    for(i=order+1;i<=max_order;i++) {
      CONTEXT_TABLE *ct=locate(context,i);
      AddSymbolToContext(ct,symbol);
    }
    putchar(symbol);
```

```
#ifdef _INFO_
    {
      int i;
      fprintf(stderr,"%3d(%c) %3d %2d |",symbol,filtro(symbol),code,order);
      for(i=0;i<=max_order;i++) {
        print_context_table(context,i);
        fprintf(stderr,"|");
      }
      for(i=max_order-1;i>=0;i--) {
        fprintf(stderr,"%c",filtro(context[i]));
      }
      fprintf(stderr,"|\n");
    }
#endif
    /* Actualizamos la cadena con el nuevo contexto */
    for(i=max_order-1;i>0;i--) context[i]=context[i-1];
    context[0]=symbol;
    code=getchar();
    file_position++;
    if(!file_position) memset(visited,1,256*4);
  }
#ifdef _HASH_INFO_
  fprintf(stderr,"tpc-ppm: n'unmero de contextos: %d\n",num_contextos);
```

```
  fprintf(stderr,"pmi: N'umero de tablas hash usadas: %d\n",num_tablas);
  fprintf(stderr,"pmi: N'umero de colisiones: %d\n",num_colisiones);
#endif
#ifdef _MEMORY_INFO_
  fprintf(stderr,"tpc-ppc: memoria consumida: %ld bytes\n",memory_used);
#endif
  fprintf(stderr,"tpc-ppm: done\n");
}
```

39.52. rgb_image.h

```
#include "image.h"

class rgb_image: public mallok {

 public:
  image *component;
  //image *R;
  //image *G;
  //image *B;

 public:
  void create(int Y, int X) {
    component = (image *)mallok::alloc_1d(3, sizeof(image));
    for (int c=0; c<3; c++) {
      component[c].create(Y, X);
    }

    /*
    R = (image *)mallok::alloc_1d(1, sizeof(image));
    G = (image *)mallok::alloc_1d(1, sizeof(image));
    B = (image *)mallok::alloc_1d(1, sizeof(image));
```

```
    R->create(Y, X);
    G->create(Y, X);
    B->create(Y, X);
    */
  }

  rgb_image(int Y, int X) {
    create(Y, X);
  }

  void destroy() {
    for (int c=0; c<3; c++) {
      component[c].destroy();
    }
    /*
    R->destroy();
    G->destroy();
    B->destroy();
    */
  }

  ~rgb_image() {
    destroy();
```

```
}

image & operator[](int x) {
  return component[x];
}

void set_size(int Y, int X) {
  for (int c=0; c<3; c++) {
    component[c].set_size(Y, X);
  }
  /*
  R->set_size(Y, X);
  G->set_size(Y, X);
  B->set_size(Y, X);
  */
}

/* Sobrecarga del operador de asignación entre líneas. */
rgb_image & operator=(const rgb_image & right) {
  set_size(right.component[0].Y, right.component[0].data[0].X);
  for (int c=0; c<3; c++) {
    component[c] = right.component[c];
  }
```

```
  /*
  set_size(right.R->Y, right.R->data[0].X);
  *R = *right.R;
  *G = *right.G;
  *B = *right.B;
  */
  return *this;
}

/*const*/ rgb_image operator+(const rgb_image & right) {
  rgb_image tmp(right.component[0].Y, right.component[0].data[0].X);
  for (int c=0; c<3; c++) {
    tmp.component[c] = component[c] + right.component[c];
  }
  /*
  rgb_image tmp(right.R->Y, right.R->data[0].X);
  *tmp.R = *R + *right.R;
  *tmp.G = *G + *right.G;
  *tmp.B = *B + *right.B;
  */
  return tmp;
}
```

```
/*const*/ rgb_image operator-(const rgb_image & right) {
  /*
    rgb_image tmp(right.R->Y, right.R->data[0].X);
  *tmp.R = *R - *right.R;
  *tmp.G = *G - *right.G;
  *tmp.B = *B - *right.B;
  */
  rgb_image tmp(right.component[0].Y, right.component[0].data[0].X);
  for (int c=0; c<3; c++) {
    tmp.component[c] = component[c] + right.component[c];
  }
  return tmp;
}

/*const*/ rgb_image operator/(const int val) {
  /*
  rgb_image tmp(R->Y, R->data[0].X);
  *tmp.R = *R / val;
  *tmp.G = *G / val;
  *tmp.B = *B / val;
  */
  rgb_image tmp(component[0].Y, component[0].data[0].X);
  for (int c=0; c<3; c++) {
```

```
      tmp.component[c] = component[c] / val;
    }
    return tmp;
  }
};
```

39.53. rice.c

```
/*
 * unary.c
 *
 * Un codificador de Rice.
 *
 * Referencias:
 *
 * R. F. Rice, "Some Practical Universal Noiseless Coding Techniques,
 * " Jet Propulsion Laboratory, Pasadena, California, JPL Publication
 * 79--22, Mar. 1979.
 *
 */

#include <stdio.h>
#include "bitio.h"
#include "vlc.h"

/* Inicializa el codificador. */
void init_encoder() {
}
```

```
/* Inicializa el descodificador. */
void init_decoder() {
}

/* Calcula el recuento del símbolo x. */
static int prob(unsigned short *cum_prob, int x) {
  return cum_prob[x-1] - cum_prob[x];
}
/* Estima la pendiente de la distribución de probabilidades de los
   símbolos. Se presupone que existen 256 símbolos en el alfabeto. */
int estimate_k(unsigned short *cum_prob) {
  int k = 0;
  int i = 1;
  while(prob(cum_prob, i+1) > prob(cum_prob, i)/2) {
    /* Si la probabilidad del símbolo "i+1" es mayor o igual que la
       mitad del símbolo "i", dejamos de incrementar la "k". */
    i++;
    k++;
    if(k>7) break;
  }
  /* Debido a una limitación de "bitio", no podemos generar códigos
     unarios más largos de 32 bits (256/32=8=2^3). */
  if(k<3) k=3;
```

```
  return k;
}

/* Codifica el índice "index" usando el espacio de recuentos
   acumulados "cum_freq". */
void encode_index(int index, unsigned short *cum_prob) {
  int i, k, m, s;
  k = estimate_k(cum_prob);
  m = 1<<k;
  s = index - 1;
  for(i=0; i<(s/m); i++) {
    put_bit(1);
  }
  put_bit(0);
  put_bits(s, k);
}

/* Descodifica el siguiente índice. */
int decode_index(unsigned short *cum_prob) {
  int i, x = 0, s, k;
  k = estimate_k(cum_prob);
  s = 0;
  while(get_bit()) {
```

```
    s++;
  }
  if (k) {
    x = get_bits(k);
    s = (s<<k) + x;
  }
  return s+1;
}

/* Finaliza el codificador. */
void finish_encoder() {
  flush();
}

/* Finaliza el descodificador. */
void finish_decoder() {
}
```

39.54. rle.c

```
/*
 * rle.cpp
 *
 * Run Length Encoding.
 *
 * Codifica una secuencia de símbolos usando una codificación de
 * series. El tamaño del alfabeto es 256 y el tamaño del código usado
 * para especificar la longitud de la serie es de 8 bits. La longitud
 * mínima d ela serie es 2. Ejemplos:
 *
 * input output
 * ----- ------
 * ab    ab
 * aab   aa0b
 * aaab  aa1b
 * aaaab aa2b
 *
 * Usos típicos:
 *
 * rle < raw-file > compressed-file
 * rle < raw-file | bwt | mtf | rle | ari > compressed-file
```

```
 *
 * Referencias:
 *
 * M. Nelson and J.-L. Gailly, The Data Compression Book. 1995.
 */

#include <stdio.h>

void read_and_encode_run(int *symbol, int prev_symbol) {
  if(*symbol == prev_symbol) {
    int length = 0;
    *symbol = getchar();

    while((*symbol != EOF) && (length < 255)) {
      if(*symbol == prev_symbol) {
        *symbol = getchar();
        length++;
      }
      else break;
    }
    putchar(length);
    if((length != 255) && (*symbol != EOF)) {
      putchar(*symbol);
```

```
    }
  }
}

void encode_stream() {
  int prev_symbol = 0;
  int symbol = getchar();
  while (symbol != EOF) {
    putchar(symbol);
    read_and_encode_run(&symbol, prev_symbol);
    prev_symbol = symbol;
    symbol = getchar();
  }
}

void read_and_decode_run(int symbol, int prev_symbol) {
  if(symbol == prev_symbol) {
    int length = getchar();
    while(length-- > 0) {
      putchar(symbol);
    }
  }
}
```

```
void decode_stream() {
  int prev_symbol = 0;
  int symbol = getchar();;
  while(symbol != EOF)  {
    putchar(symbol);
    read_and_decode_run(symbol, prev_symbol);
    prev_symbol = symbol;
    symbol = getchar();
  }
}
```

39.55. unary.c

```
/*
 * unary.c
 *
 * Un codificador unario.
 *
 * Referencias:
 *
 * Khalid Sayood, Data Compression, 3rd ed, Morgan Kaufmann.
 * K. R. Rao, Principles of Digital Video Coding.
 */

#include <stdio.h>
#include "bitio.h"
#include "vlc.h"

/* Inicializa el codificador. */
void init_encoder() {
}

/* Inicializa el descodificador. */
```

```
void init_decoder() {
}

/* Codifica el índice "index" usando el espacio de recuentos
   acumulados "cum_freq". */
void encode_index(int index, unsigned short *cum_prob) {
  int i, s;
  s = index - 1;
  for(i=0; i<s; i++) {
    put_bit(1);
  }
  put_bit(0);
}

/* Descodifica el siguiente índice . */
int decode_index(unsigned short *cum_prob) {
  int s = 0;
  while(get_bit()) {
    s++;
  }
  return s+1;
}
```

```
/* Finaliza el codificador. */
void finish_encoder() {
  flush();
}

/* Finaliza el descodificador. */
void finish_decoder() {
}
```

39.56. vids/Makefile

```
# Ver ffmpeg -formats

VIDS := $(wildcard *.yuv)

###############
# Originales #
###############
AVIS = $(VIDS:%.yuv=%.avi)
%.avi: %.yuv
        ffmpeg -vcodec copy -r 30 -s 352x288 -i $*.yuv $*.avi

#########
# MJPEG #
#########

# 3631831 es el número de bytes generado en el experimento
# "jpeg/compresión de akiyo".
# 3631831/10.0*8 = 2905464
AVIS += $(VIDS:%.yuv=%_MJPEG_2905kbps.avi)
%_MJPEG_2905kbps.avi: %.yuv
```

```
	ffmpeg -pass 1 -vcodec mjpeg -r 30 -s 352x288 -b 2905464 \
	-i $*.yuv $*_MJPEG_2905kbps.avi
	ffmpeg -y -pass 2 -vcodec mjpeg -r 30 -s 352x288 -b 2905464 \
	-i $*.yuv $*_MJPEG_2905kbps.avi

###############
# M-JPEG2000 #
###############
AVIS += $(VIDS:%.yuv=%_MJPEG2000_2905kbps.avi)
%_MJPEG2000_2905kbps.avi: %.yuv
	echo -n "" > $*.vix          # Borramos el fichero.
	echo "vix" >> $*.vix        # Magic numbers.
	echo ">VIDEO<" >> $*.vix    # Se trata de un vídeo:
	echo "30.0 0" >> $*.vix     # Frame-rate y número de imágenes.
	echo ">COLOUR<" >> $*.vix   # El espacio de color es:
	echo "YCbCr" >> $*.vix      # YCbCr.
	echo ">IMAGE<" >> $*.vix    # Las imágenes son:
	echo "unsigned char 8 little-endian" >> $*.vix # Precisión y endian
	echo "352 288 3" >> $*.vix  # x_size y_size components
	echo "1 1" >> $*.vix        # Y sub-sampling
	echo "2 2" >> $*.vix        # Cb sub-sampling
	echo "2 2" >> $*.vix        # Cr sub-sampling
	cat $*.yuv >> $*.vix        # Pegamos el fichero .YUV al fichero .VI
```

```
	kdu_v_compress -i $*.vix -o $*.mj2 -rate 0.96 # Comprimimos
	kdu_v_expand -i $*.mj2 -o $*.vix # Descomprimimos
	rm -f $*.mj2                     # Borramos el fichero .mj2
	dd if=$*.vix bs=101 skip=1 of=tmp.yuv count=3000000 # Eliminamos la c
	ffmpeg -vcodec copy -r 30 -s 352x288 \
	-i tmp.yuv $*_MJPEG2000_2905kbps.avi # YUV -> AVI
	rm -f tmp.yuv                    # Borramos el fichero temporal

##########
# MPEG-1 #
##########

# 1200kbps
#AVIS = $(VIDS:/home/data/videos/%.yuv=%_MPEG-1_1200kbps.avi)
AVIS += $(VIDS:%.yuv=%_MPEG-1_1200kbps.avi)
%_MPEG-1_1200kbps.avi: %.yuv
	ffmpeg -pass 1 -vcodec mpeg1video -r 30 -s 352x288 -b 1200000 \
	-i $*.yuv $*_MPEG-1_1200kbps.avi
	ffmpeg -y -pass 2 -vcodec mpeg1video -r 30 -s 352x288 -b 1200000 \
	-i $*.yuv $*_MPEG-1_1200kbps.avi

# 600kbps
AVIS += $(VIDS:%.yuv=%_MPEG-1_600kbps.avi)
```

```
%_MPEG-1_600kbps.avi: %.yuv
        ffmpeg -pass 1 -vcodec mpeg1video -r 30 -s 352x288 -b 600000 \
        -i $*.yuv $*_MPEG-1_600kbps.avi
        ffmpeg -y -pass 2 -vcodec mpeg1video -r 30 -s 352x288 -b 600000 \
        -i $*.yuv $*_MPEG-1_600kbps.avi

# 300kbps
AVIS += $(VIDS:%.yuv=%_MPEG-1_300kbps.avi)
%_MPEG-1_300kbps.avi: %.yuv
        ffmpeg -pass 1 -vcodec mpeg1video -r 30 -s 352x288 -b 300000 \
        -i $*.yuv $*_MPEG-1_300kbps.avi
        ffmpeg -y -pass 2 -vcodec mpeg1video -r 30 -s 352x288 -b 300000 \
        -i $*.yuv $*_MPEG-1_300kbps.avi

# 100kbps
AVIS += $(VIDS:%.yuv=%_MPEG-1_100kbps.avi)
%_MPEG-1_100kbps.avi: %.yuv
        ffmpeg -pass 1 -vcodec mpeg1video -r 30 -s 352x288 -b 100000 \
        -i $*.yuv $*_MPEG-1_100kbps.avi
        ffmpeg -y -pass 2 -vcodec mpeg1video -r 30 -s 352x288 -b 100000 \
        -i $*.yuv $*_MPEG-1_100kbps.avi

##########
```

```
# MPEG-2 #
##########

# 1200kbps
#AVIS = $(VIDS:/home/data/videos/%.yuv=%_MPEG-2_1200kbps.avi)
AVIS += $(VIDS:%.yuv=%_MPEG-2_1200kbps.avi)
%_MPEG-2_1200kbps.avi: %.yuv
        ffmpeg -pass 1 -vcodec mpeg2video -r 30 -s 352x288 -b 1200000 \
        -i $*.yuv $*_MPEG-2_1200kbps.avi
        ffmpeg -y -pass 2 -vcodec mpeg2video -r 30 -s 352x288 -b 1200000 \
        -i $*.yuv $*_MPEG-2_1200kbps.avi

# 600kbps
AVIS += $(VIDS:%.yuv=%_MPEG-2_600kbps.avi)
%_MPEG-2_600kbps.avi: %.yuv
        ffmpeg -pass 1 -vcodec mpeg2video -r 30 -s 352x288 -b 600000 \
        -i $*.yuv $*_MPEG-2_600kbps.avi
        ffmpeg -y -pass 2 -vcodec mpeg2video -r 30 -s 352x288 -b 600000 \
        -i $*.yuv $*_MPEG-2_600kbps.avi

# 300kbps
AVIS += $(VIDS:%.yuv=%_MPEG-2_300kbps.avi)
%_MPEG-2_300kbps.avi: %.yuv
```

```
        ffmpeg -pass 1 -vcodec mpeg2video -r 30 -s 352x288 -b 300000 \
        -i $*.yuv $*_MPEG-2_300kbps.avi
        ffmpeg -y -pass 2 -vcodec mpeg2video -r 30 -s 352x288 -b 300000 \
        -i $*.yuv $*_MPEG-2_300kbps.avi

# 100kbps
AVIS += $(VIDS:%.yuv=%_MPEG-2_100kbps.avi)
%_MPEG-2_100kbps.avi: %.yuv
        ffmpeg -pass 1 -vcodec mpeg2video -r 30 -s 352x288 -b 100000 \
        -i $*.yuv $*_MPEG-2_100kbps.avi
        ffmpeg -y -pass 2 -vcodec mpeg2video -r 30 -s 352x288 -b 100000 \
        -i $*.yuv $*_MPEG-2_100kbps.avi

##########
# MPEG-4 #
##########

# 1200kbps
#AVIS = $(VIDS:/home/data/videos/%.yuv=%_MPEG-4_1200kbps.avi)
AVIS += $(VIDS:%.yuv=%_MPEG-4_1200kbps.avi)
%_MPEG-4_1200kbps.avi: %.yuv
        ffmpeg -pass 1 -vcodec mpeg4 -r 30 -s 352x288 -b 1200000 \
        -i $*.yuv $*_MPEG-4_1200kbps.avi
```

```
        ffmpeg -y -pass 2 -vcodec mpeg4 -r 30 -s 352x288 -b 1200000 \
        -i $*.yuv $*_MPEG-4_1200kbps.avi


# 600kbps
AVIS += $(VIDS:%.yuv=%_MPEG-4_600kbps.avi)
%_MPEG-4_600kbps.avi: %.yuv
        ffmpeg -pass 1 -vcodec mpeg4 -r 30 -s 352x288 -b 600000 \
        -i $*.yuv $*_MPEG-4_600kbps.avi
        ffmpeg -y -pass 2 -vcodec mpeg4 -r 30 -s 352x288 -b 600000 \
        -i $*.yuv $*_MPEG-4_600kbps.avi

# 300kbps
AVIS += $(VIDS:%.yuv=%_MPEG-4_300kbps.avi)
%_MPEG-4_300kbps.avi: %.yuv
        ffmpeg -pass 1 -vcodec mpeg4 -r 30 -s 352x288 -b 300000 \
        -i $*.yuv $*_MPEG-4_300kbps.avi
        ffmpeg -y -pass 2 -vcodec mpeg4 -r 30 -s 352x288 -b 300000 \
        -i $*.yuv $*_MPEG-4_300kbps.avi

# 100kbps
AVIS += $(VIDS:%.yuv=%_MPEG-4_100kbps.avi)
%_MPEG-4_100kbps.avi: %.yuv
```

```
	ffmpeg -pass 1 -vcodec mpeg4 -r 30 -s 352x288 -b 100000 \
	-i $*.yuv $*_MPEG-4_100kbps.avi
	ffmpeg -y -pass 2 -vcodec mpeg4 -r 30 -s 352x288 -b 100000 \
	-i $*.yuv $*_MPEG-4_100kbps.avi

#######
# FLV #
#######

# 1200kbps
AVIS += $(VIDS:%.yuv=%_FLV_1200kbps.avi)
%_FLV_1200kbps.avi: %.yuv
	ffmpeg -pass 1 -vcodec flv -r 30 -s 352x288 -b 1200000 \
	-i $*.yuv $*_FLV_1200kbps.avi
	ffmpeg -y -pass 2 -vcodec flv -r 30 -s 352x288 -b 1200000 \
	-i $*.yuv $*_FLV_1200kbps.avi


# 600kbps
AVIS += $(VIDS:%.yuv=%_FLV_600kbps.avi)
%_FLV_600kbps.avi: %.yuv
	ffmpeg -pass 1 -vcodec flv -r 30 -s 352x288 -b 600000 \
	-i $*.yuv $*_FLV_600kbps.avi
```

```
        ffmpeg -y -pass 2 -vcodec flv -r 30 -s 352x288 -b 600000 \
        -i $*.yuv $*_FLV_600kbps.avi

# 300kbps
AVIS += $(VIDS:%.yuv=%_FLV_300kbps.avi)
%_FLV_300kbps.avi: %.yuv
        ffmpeg -pass 1 -vcodec flv -r 30 -s 352x288 -b 300000 \
        -i $*.yuv $*_FLV_300kbps.avi
        ffmpeg -y -pass 2 -vcodec flv -r 30 -s 352x288 -b 300000 \
        -i $*.yuv $*_FLV_300kbps.avi

# 100kbps
AVIS += $(VIDS:%.yuv=%_FLV_100kbps.avi)
%_FLV_100kbps.avi: %.yuv
        ffmpeg -pass 1 -vcodec flv -r 30 -s 352x288 -b 100000 \
        -i $*.yuv $*_FLV_100kbps.avi
        ffmpeg -y -pass 2 -vcodec flv -r 30 -s 352x288 -b 100000 \
        -i $*.yuv $*_FLV_100kbps.avi

########
# H263 #
########
```

```
# 1200kbps
AVIS += $(VIDS:%.yuv=%_H263_1200kbps.avi)
%_H263_1200kbps.avi: %.yuv
        ffmpeg -pass 1 -vcodec h263 -r 30 -s 352x288 -b 1200000 \
        -i $*.yuv $*_H263_1200kbps.avi
        ffmpeg -y -pass 2 -vcodec h263 -r 30 -s 352x288 -b 1200000 \
        -i $*.yuv $*_H263_1200kbps.avi

# 100kbps
AVIS += $(VIDS:%.yuv=%_H263_100kbps.avi)
%_H263_100kbps.avi: %.yuv
        ffmpeg -pass 1 -vcodec h263 -r 30 -s 352x288 -b 100000 \
         -i $*.yuv $*_H263_100kbps.avi
        ffmpeg -y -pass 2 -vcodec h263 -r 30 -s 352x288 -b 100000 \
        -i $*.yuv $*_H263_100kbps.avi

########
# H264 #
########

# 1000kbps
AVIS += $(VIDS:%.yuv=%_H264_1000kbps.avi)
%_H264_1000kbps.avi: %.yuv
```

```
	mencoder $*.avi -o $*_H264_1000kbps.avi -ovc x264 -x264encopts bitrat
	mencoder $*.avi -o $*_H264_1000kbps.avi -ovc x264 -x264encopts bitrat

# 100kbps
AVIS += $(VIDS:%.yuv=%_H264_100kbps.avi)
%_H264_100kbps.avi: %.yuv
	mencoder $*.avi -o $*_H264_100kbps.avi -ovc x264 -x264encopts bitrate
	mencoder $*.avi -o $*_H264_100kbps.avi -ovc x264 -x264encopts bitrate

# 50kbps
AVIS += $(VIDS:%.yuv=%_H264_50kbps.avi)
%_H264_50kbps.avi: %.yuv
	mencoder $*.avi -o $*_H264_50kbps.avi -ovc x264 -x264encopts bitrate=
	mencoder $*.avi -o $*_H264_50kbps.avi -ovc x264 -x264encopts bitrate=

# 25kbps
AVIS += $(VIDS:%.yuv=%_H264_25kbps.avi)
%_H264_25kbps.avi: %.yuv
	mencoder $*.avi -o $*_H264_25kbps.avi -ovc x264 -x264encopts bitrate=
	mencoder $*.avi -o $*_H264_25kbps.avi -ovc x264 -x264encopts bitrate=

########
# FSVC #
```

```
########

# 1200Kbps
AVIS += $(VIDS:%.yuv=%_FSVC_1200kbps.avi)
%_FSVC_1200kbps.avi: %.yuv
        ./run_fsvc $* 1200 5
        ffmpeg -vcodec copy -r 30 -s 352x288 \
        -i tmp.yuv $*_FSVC_1200kbps.avi # YUV -> AVI
        rm -f tmp.yuv

# 600Kbps
AVIS += $(VIDS:%.yuv=%_FSVC_600kbps.avi)
%_FSVC_600kbps.avi: %.yuv
        ./run_fsvc $* 600 5
        ffmpeg -vcodec copy -r 30 -s 352x288 \
        -i tmp.yuv $*_FSVC_600kbps.avi # YUV -> AVI
        rm -f tmp.yuv

# 300Kbps
AVIS += $(VIDS:%.yuv=%_FSVC_300kbps.avi)
%_FSVC_300kbps.avi: %.yuv
        ./run_fsvc $* 300 5
        ffmpeg -vcodec copy -r 30 -s 352x288 \
```

```
        -i tmp.yuv $*_FSVC_300kbps.avi # YUV -> AVI
        rm -f tmp.yuv

# 100Kbps
AVIS += $(VIDS:%.yuv=%_FSVC_100kbps.avi)
%_FSVC_100kbps.avi: %.yuv
        ./run_fsvc $* 100 6
        ffmpeg -vcodec copy -r 30 -s 352x288 \
        -i tmp.yuv $*_FSVC_100kbps.avi # YUV -> AVI
        rm -f tmp.yuv

# 50Kbps
AVIS += $(VIDS:%.yuv=%_FSVC_50kbps.avi)
%_FSVC_50kbps.avi: %.yuv
        ./run_fsvc $* 50 7
        ffmpeg -vcodec copy -r 30 -s 352x288 \
        -i tmp.yuv $*_FSVC_50kbps.avi # YUV -> AVI
        rm -f tmp.yuv

# Transmisión
AVIS += $(VIDS:%.yuv=%_FSVC_1200kbps_trans.avi)
%_FSVC_1200kbps_trans.avi: %.yuv
        ./run_fsvc_trans $* 1200 5
```

```
	ffmpeg -vcodec copy -r 30 -s 352x288 \
	-i tmp.yuv $*_FSVC_1200kbps_trans.avi # YUV -> AVI
	rm -f tmp.yuv

################
# Makefile.tex #
################

Makefile.tex:		Makefile
	echo "\\\footnotesize" > Makefile.tex
	echo -n "\\\begin" >> Makefile.tex
	echo "{verbatim}" >> Makefile.tex
	awk '{gsub(/\t/,"        ");print}' < Makefile >> Makefile.tex
	echo -n "\\\end" >> Makefile.tex
	echo "{verbatim}" >> Makefile.tex
	echo "\\\normalsize" >> Makefile.tex

all:	$(AVIS) Makefile.tex

objetives:
	@echo $(AVIS) Makefile.tex

clean:
```

```
rm -f *.avi *.log *.tex *.vix
```

39.57. vlc.h

```
void   init_encoder();
void   init_decoder();
void   encode_index(int index, unsigned short *cum_counts);
int    decode_index(unsigned short *cum_counts);
void finish_encoder();
void finish_decoder();
```

Bibliografía

[1] M. D. Adams. *Reversible Wavelet Transforms and Their Applications to Embedded Image Compression*. PhD thesis, B.A.Sc. University of Waterloo, 1998.

[2] M. D. Adams and F. Kossentini. Reversible Integer-to-Integer Wavelet Transforms for Image Compression: Performance Evaluation and Analysis. *IEEE Trans. Image Process.*, 9(6):1010–1024, 2000.

[3] J. G. Cleary and I. H. Witten. Data Compression using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, 4(32):396–402, 1984.

[4] Compuserve Incorporated. *Graphics Interchange Format (GIF) Specification*, June 1987.

[5] Pamela C. Cosman, Robert M. Gray, and Martin Vetterli. Vector Quantization of Image Subbands: A survey. *IEEE Transactions on Signal Processing*, 5(2):202 – 225, February 1996.

[6] International Organization for Standardization (ISO). Iso/iec tr 13818-5 (reference software). http://www.iso.ch/iso/en/ittf/PubliclyAvailableStandards, 1997.

[7] International Organization for Standardization (ISO). Iso/iec tr 11172-5 (reference software). http://www.iso.ch/iso/en/ittf/PubliclyAvailableStandards, 1998.

[8] S.W. Golomb. Run-Length Encodings. *IEEE Transactions on Information Theory*, 12:399–401, 1966.

[9] A. Haar. Zur Theorie der orthogolanen Funktionen-Systeme. *Mathematische Annalen*, 69:331–371, 1910.

[10] D. A. Huffman. A Method for the Construction of Minimum Redundancy Codes. *Proceedings of the Institute of Radio Engineers*, 40:1098–1101, 1952.

[11] The Joint Photographic Experts Group (JPEG). *Recommendation T.81: Digital Compression and Coding of Continuous-tone Still Images*. International Telecommunication Union (ITU), September 1992.

[12] The Joint Photographic Experts Group (JPEG). *FCD 14495, Lossless and Near-Lossless Coding of Continuous Tone Still Images (JPEG-LS)*. The International Standards Organization (ISO)/The International Telegraph and Telephone Consultative Committee (CCITT), July 1997.

[13] ISO/IEC JTC1/SC29/WG11. Coding of moving pictures and adio. Technical report, International Organisation for Standardisation, http://www.chiariglione.org/mpeg/standards/mpeg-4, 2002.

[14] A.M. Marcos. *Compresión de imágenes. Norma JPEG*. Editorial Ciencia 3, 1999.

[15] F. Pereira and T. Ebrahimi. *The MPEG-4 Book*. Pearson Education, 2002.

[16] Fernando Pereira and Paulo Nunes. Levels for mpeg-4 visual profiles. Technical report, MPEG-4 Industry Forum, http://www.m4if.org/resources/profiles, 2002.

[17] R. F. Rice. Some Practical Universal Noiseless Coding Techniques. Technical Report 79/22, Jet Propulsion Laboratory, 1979.

[18] J. A. Storer and T. G. Szymanski. Data Compression via Textual Substitution. *Journal of the ACM*, 29(4):928–951, 1982.

[19] W. Sweldens and P. Schröder. *Building Your Own Wavelets at Home*.

[20] D. S. Taubman and M. W. Marcellin. *JPEG2000. Image Compression Fundamentals, Standards and Practice*. Kluwer Academic Publishers, 2002.

[21] S. W. Thomas, J. McKie, S. Davies, K. Turkowski, J. A. Woods, and J. Orost. *compress* 4.1. `ftp://wuarchive.wustl.edu/packages/compression/compress-4.1.tar`.

[22] G. K. Wallace. "the jpeg still picture compression standard". *Communications of the ACM*, 34(4):30 – 44, April 1991.

[23] M. J. Weinberger, G. Seroussi, and G. Sapiro. LOCO-I: A Low Complexity, Context-Based, Lossless Image Compression Algorithm. In *Data Compression Conference (DCC)*, pages 140–149, 1996.

[24] T. A. Welch. A Technique for High-Performance Data Compression. *IEEE Computer*, 17:8–19, 1984.

[25] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.

[26] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Trans. on Inf. Theory*, 24(5):530–536, 1978.