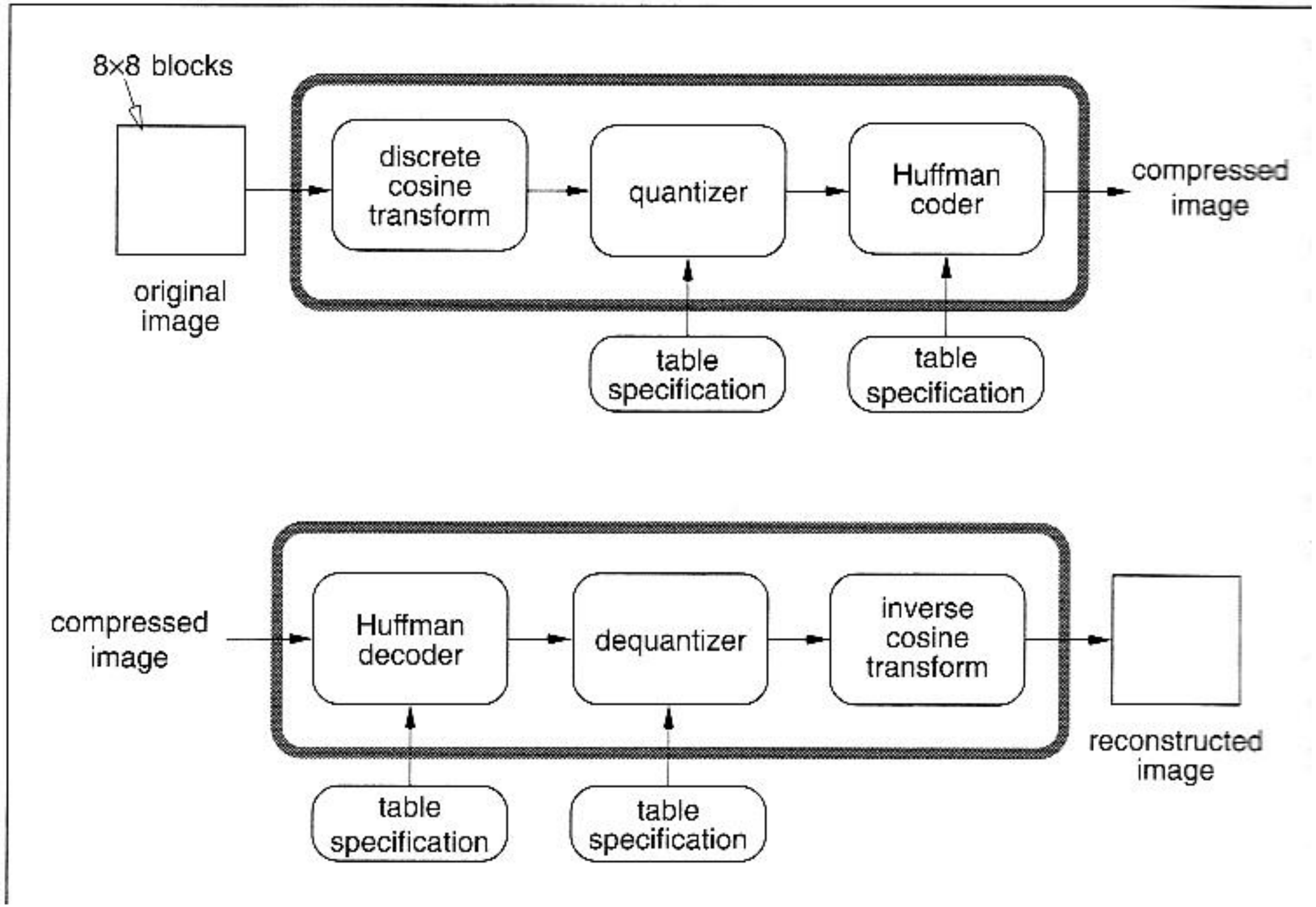
Wavelet Transform

CS6025 Data Encoding

Yizong Cheng

2-5-15

JPEG Encoding and Decoding



Discrete Cosine Transform

$$G_{ij} = \frac{1}{4} C_i C_j \sum_{x=0}^{7} \sum_{y=0}^{7} p_{xy} \cos\left(\frac{(2x+1)i\pi}{16}\right) \cos\left(\frac{(2y+1)j\pi}{16}\right),$$

$$\text{where } C_f = \begin{cases} \frac{1}{\sqrt{2}}, & f = 0, \\ 1, & f > 0, \end{cases} \quad \text{and } 0 \le i, j \le 7.$$

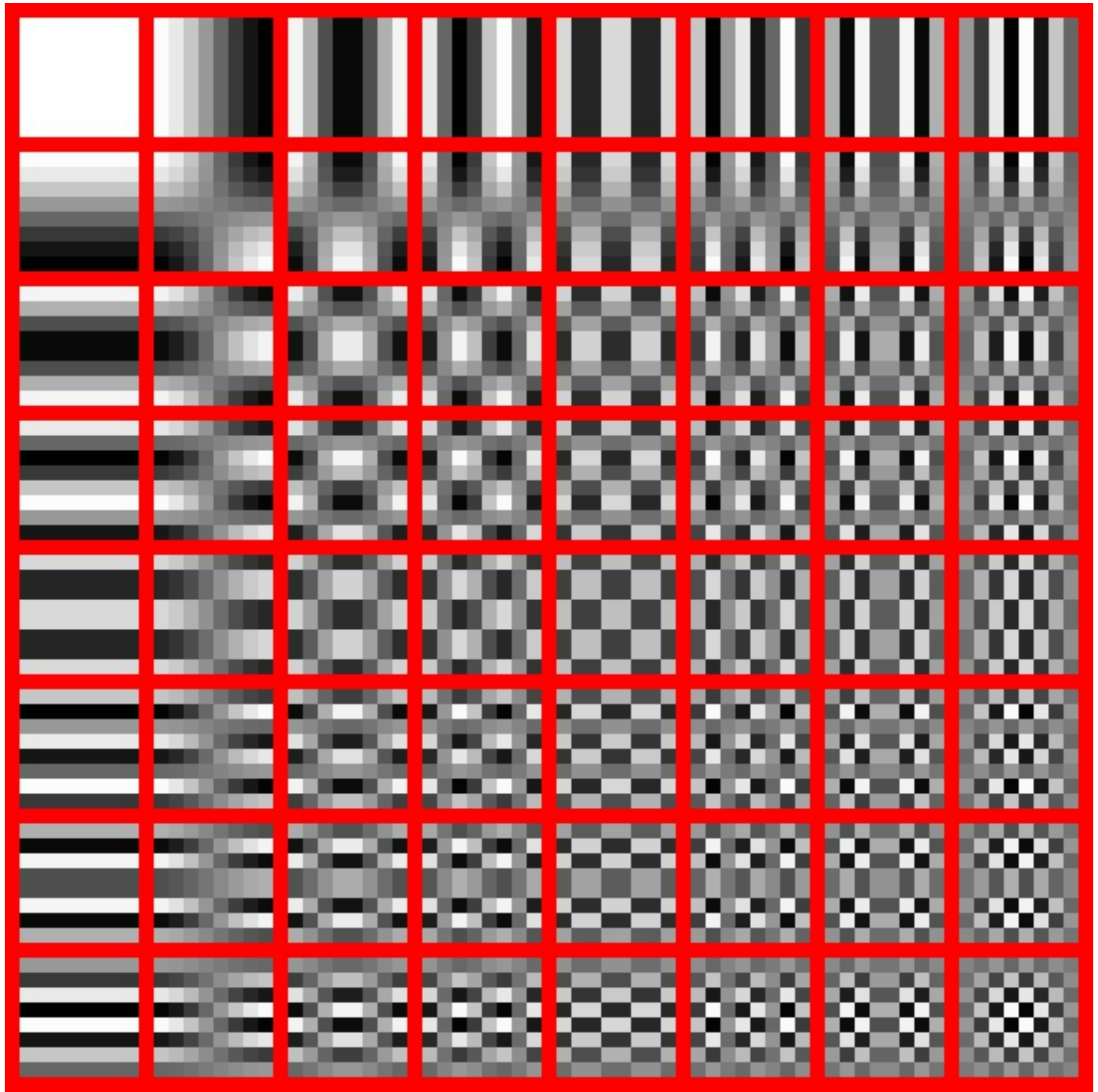
Inverse DCT

$$p_{xy} = \frac{1}{4} \sum_{i=0}^{7} \sum_{j=0}^{7} C_i C_j G_{ij} \cos\left(\frac{(2x+1)i\pi}{16}\right) \cos\left(\frac{(2y+1)j\pi}{16}\right)$$

$$\text{where } C_f = \begin{cases} \frac{1}{\sqrt{2}}, & f = 0; \\ 1, & f > 0. \end{cases}$$

DCT Components

wikipedia



Practical DCT

- components can be pre-calculated.
- amounts to three 8x8 matrices multiplied together.
- VLSI chips may accelerate transform.
- Purpose of DCT: The eye is more sensitive to low frequency components.
- G_{00} is the mean, or DC component
- Others are AC components.

JPEG Quantization

- Each of the 64 frequency components in a data unit is divided by a separate quantization coefficient (QC) and rounded to the nearest integer.
- These 64 integers are encoded using both run length and self-delimiting coding.

Quantization Coefficients

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

Luminance

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

Chrominance

A Typical Quantization Result

1118	2	0	0	0	0	0	0
0	0	0	0	0	0	0	0
−2	0	0	−1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Zigzag Order of Components

(0,0)	(0,1)	(1,0)	(2,0)	(1,1)	(0,2)	(0,3)	(1,2)
(2,1)	(3,0)	(4,0)	(3,1)	(2,2)	(1,3)	(0,4)	(0,5)
(1,4)	(2,3)	(3,2)	(4,1)	(5,0)	(6,0)	(5,1)	(4,2)
(3,3)	(2,4)	(1,5)	(0,6)	(0,7)	(1,6)	(2,5)	(3,4)
(4,3)	(5,2)	(6,1)	(7,0)	(7,1)	(6,2)	(5,3)	(4,4)
(3,5)	(2,6)	(1,7)	(2,7)	(3,6)	(4,5)	(5,4)	(6,3)
(7,2)	(7,3)	(6,4)	(5,5)	(4,6)	(3,7)	(4,7)	(5,6)
(6,5)	(7,4)	(7,5)	(6,6)	(5,7)	(6,7)	(7,6)	(7,7)

Coding DC Components

- 1118 = row 11 column 930
- 111111111110 01110100010
- 11 1’s, a zero, 11-bit binary 930

0:	0										0
1:	-1	1									10
2:	-3	-2	2	3							110
3:	-7	-6	-5	-4	4	5	6	7			1110
4:	-15	-14	...	-9	-8	8	9	10	...	15	11110
5:	-31	-30	-29	...	-17	-16	16	17	...	31	111110
6:	-63	-62	-61	...	-33	-32	32	33	...	63	1111110
7:	-127	-126	-125	...	-65	-64	64	65	...	127	11111110
⋮				⋮							
14:	-16383	-16382	-16381	...	-8193	-8192	8192	8193	...	16383	111111111111110
15:	-32767	-32766	-32765	...	-16385	-16384	16384	16385	...	32767	1111111111111110
16:	32768										1111111111111111

Differences of DC Components

- After the first data unit, DC components are represented by their differences to the previous DC components.
- A difference of -4 is at row 3 column 3 and is encoded as 1110 011.

Coding AC Components

- For each nonzero component x, find the number Z of consecutive zeros preceding x.
- Find x resides at row R and column C.
- C is written as an R-bit number, concatenated by the code for (Z,R) in the following table.

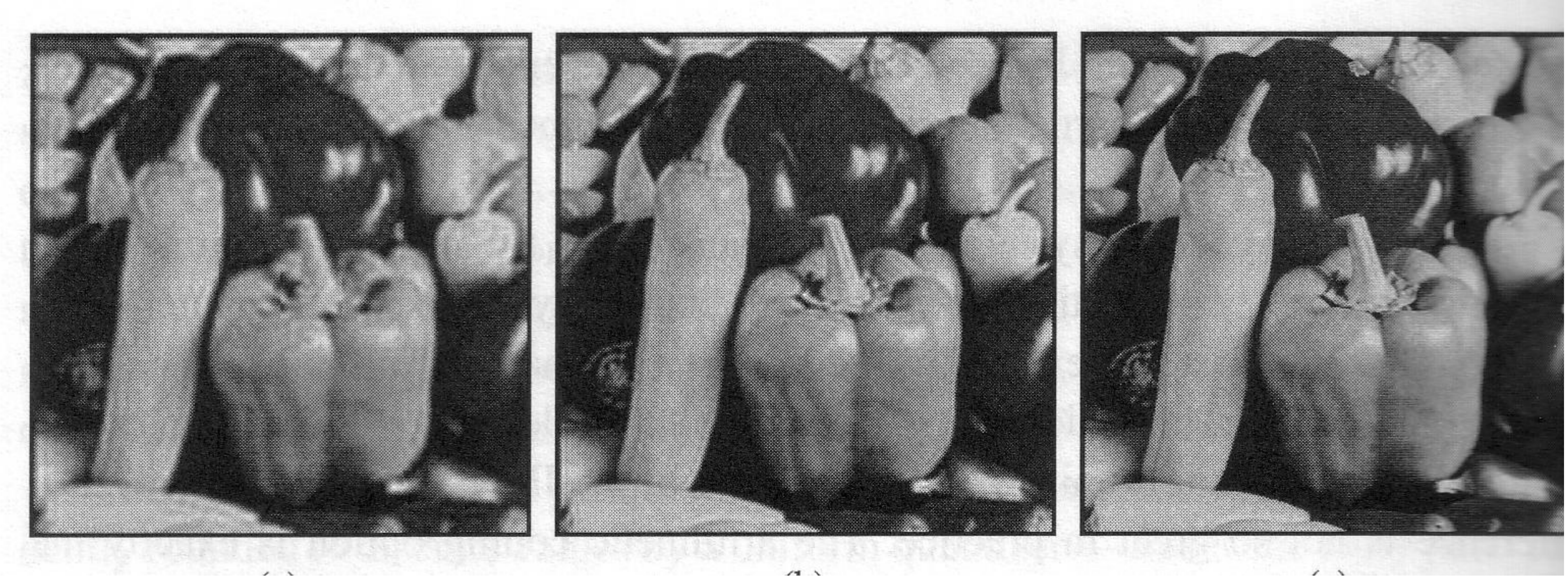
The (Z, R) Table

Z	R: 1 6	2 7	3 8	4 9	5 A
0	00 1111000	01 11111000	100 1111110110	1011 1111111110000010	11010 1111111110000011
1	1100 1111111110000100	11011 1111111110000101	11110001 1111111110000110	111110110 1111111110000111	11111110110 1111111110001000
2	11100 111111110001010	11111001 111111110001011	1111110111 111111110001100	111111110100 111111110001101	111111110001001 111111110001110
3	111010 1111111110010001	111110111 1111111110010010	111111110101 1111111110010011	1111111110001111 1111111110010100	1111111110010000 1111111110010101
4	111011 1111111110011001	1111111000 1111111110011010	1111111110010110 1111111110011011	1111111110010111 1111111110011100	1111111110011000 1111111110011101
5	1111010 1111111110100001	11111110111 1111111110100010	1111111110011110 1111111110100011	1111111110011111 1111111110100100	1111111110100000 1111111110100101
6	1111011 1111111110101001	111111110110 1111111110101010	1111111110100110 1111111110101011	1111111110100111 1111111110101100	1111111110101000 1111111110101101
7	11111010 1111111110110001	111111110111 1111111110110010	1111111110101110 1111111110110011	1111111110101111 1111111110110100	1111111110110000 1111111110110101
8	111111000 1111111110111001	111111111000000 1111111110111010	1111111110110110 1111111110111011	1111111110110111 1111111110111100	1111111110111000 1111111110111101

JFIF

- Graphics file format for JPEG
- divided into marker segments
- The first 2 bytes in each segment is the marker ffxx with xx from c0 to fe
- begins with marker ffd8 and ends with ffd9
- Segments specifying quantization and Huffman code.
- Scans ffda with concatenated codewords for symbols that are zero runs-size followed by size many bits.

JPEG with .1, .2, and 1 bit/pixel



Winter.jpg 800x600 105,542 bytes (1.76 bits per pixel)

Complexity of DCT

- Discrete cosine transform can be decomposed into that applied on rows followed by that applied on columns.
- Each is a linear transform representable by the matrix M_{ij} = $\cos((2j+1)i\ \pi/N)$.
- DCT on NxN block requires $2N^3$ multiplications and additions and is very time-consuming.
- JPEG uses DCT only on 8x8 blocks.

Matrix Multiplication

- To complete a 1D DCT transform, the following for loops are needed.
- for (i=0;i<N;i++){ y[i]=0;
-   for (j=0;j<N;j++) y[i]+=x[j]*M[i][j];
- }
- This requires N^2 multiplications.
- This has to be done for every column and thus the complexity of N^3.

M with k non-zeros each row

- What happens if M has a lot of zeros?
- Suppose there are only k non-zeros each row. The number of multiplications for 1D transform will be reduced to kN.
- We can even reformulate the double for loop so that the number of iterations is reduced to kN.
- The over all complexity is $O(N^2)$, not $O(N^3)$

The Quadrature Mirror Filter

$h_0 \;\; h_1 \;\; h_2 \;\; h_3 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \ldots 0 \;\; 0$

$0 \;\; 0 \;\; h_0 \;\; h_1 \;\; h_2 \;\; h_3 \;\; 0 \;\; 0 \ldots 0 \;\; 0$

…

$h_2 \;\; h_3 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \ldots h_0 \;\; h_1$

$h_1 \; –h_0 \; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \ldots h_3 \; –h_2$

…

$0 \;\; 0 \;\; \ldots \;\; 0 \;\; h_3 \; –h_2 \;\; h_1 \; –h_0 \; 0 \;\; 0$

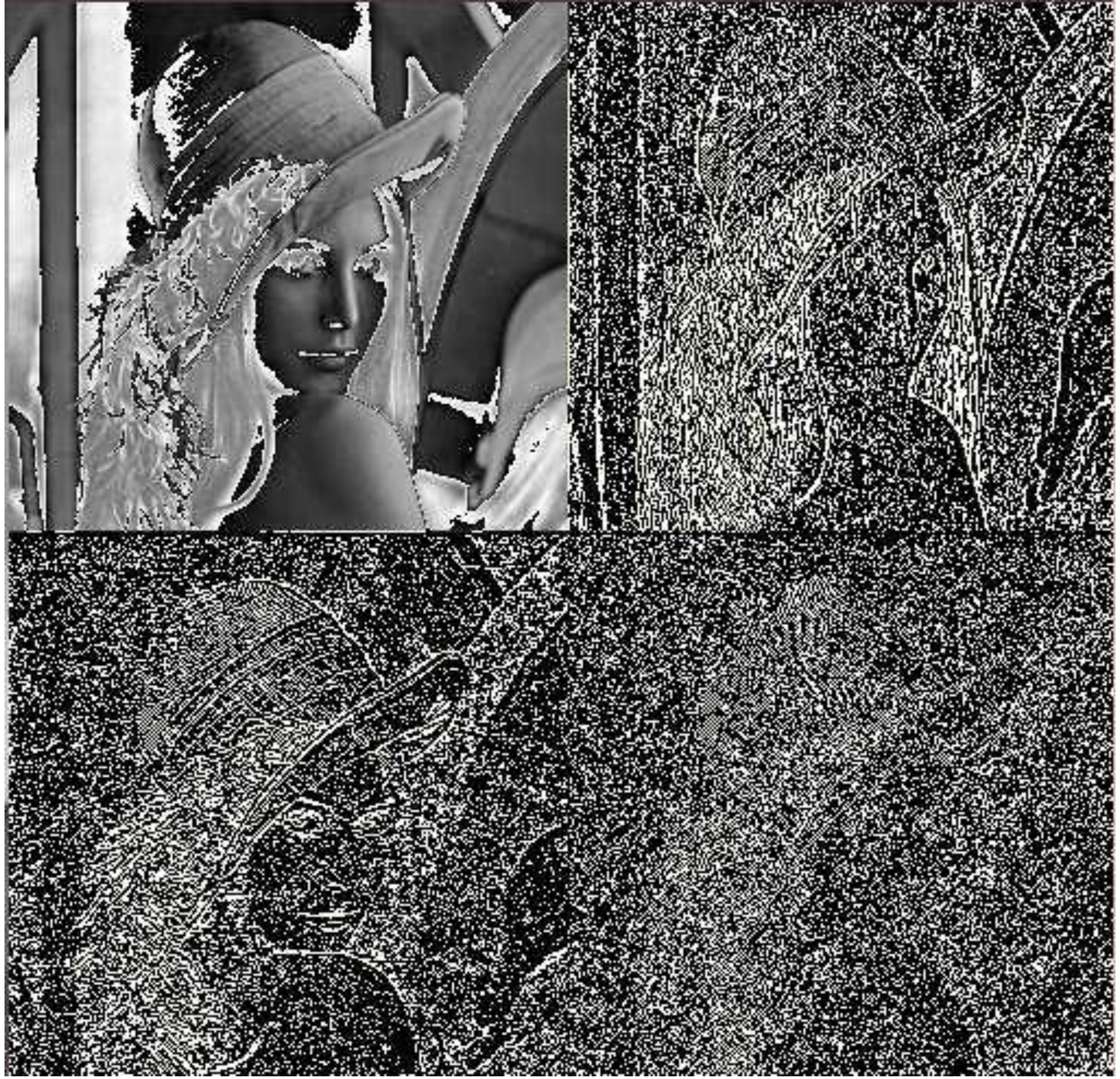
$0 \;\; 0 \;\; \ldots \;\; 0 \;\; 0 \;\; 0 \;\; h_3 \;\; -h_2 \; h_1 \;\; -h_0$

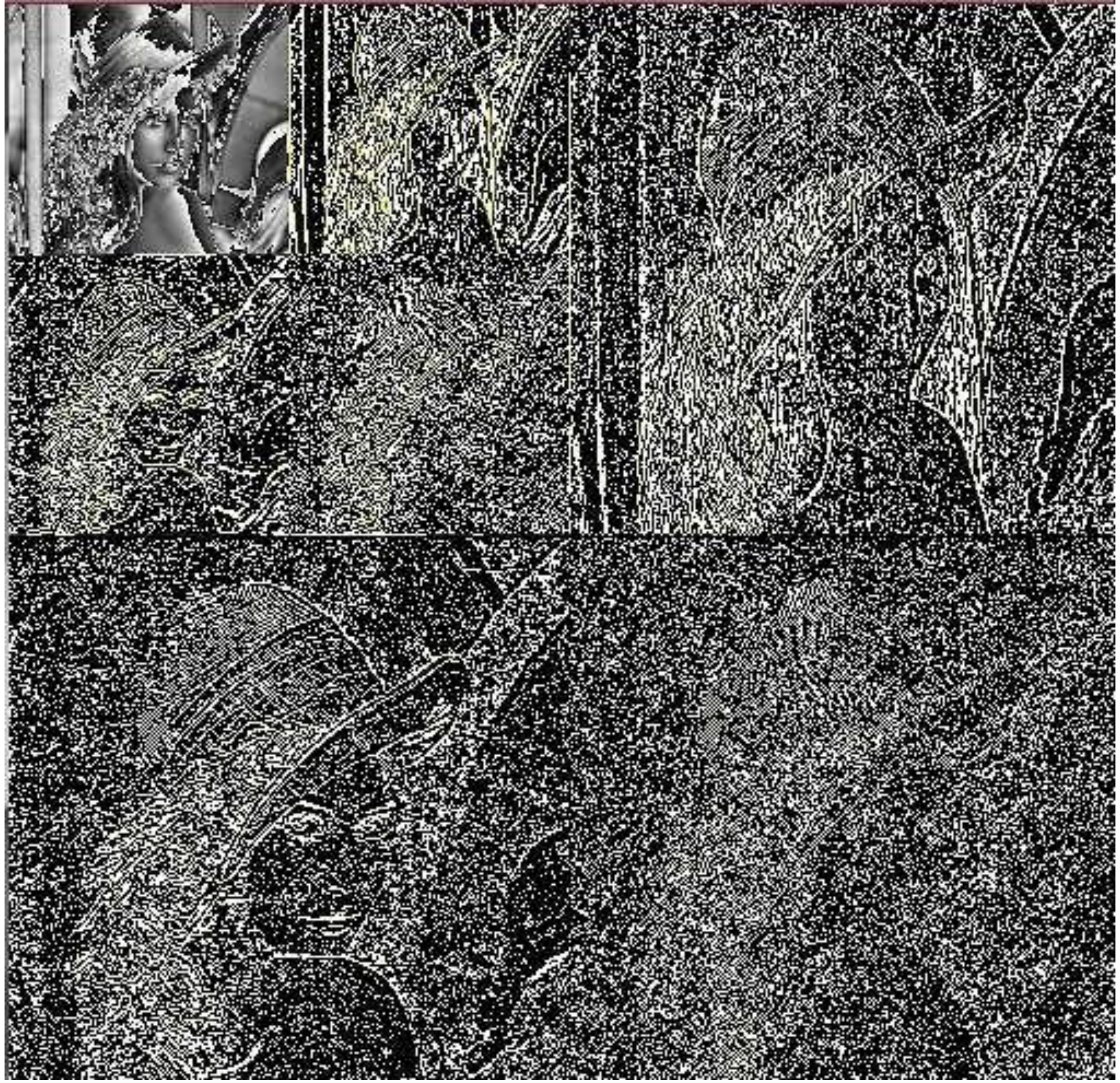
Orthonormal QMF

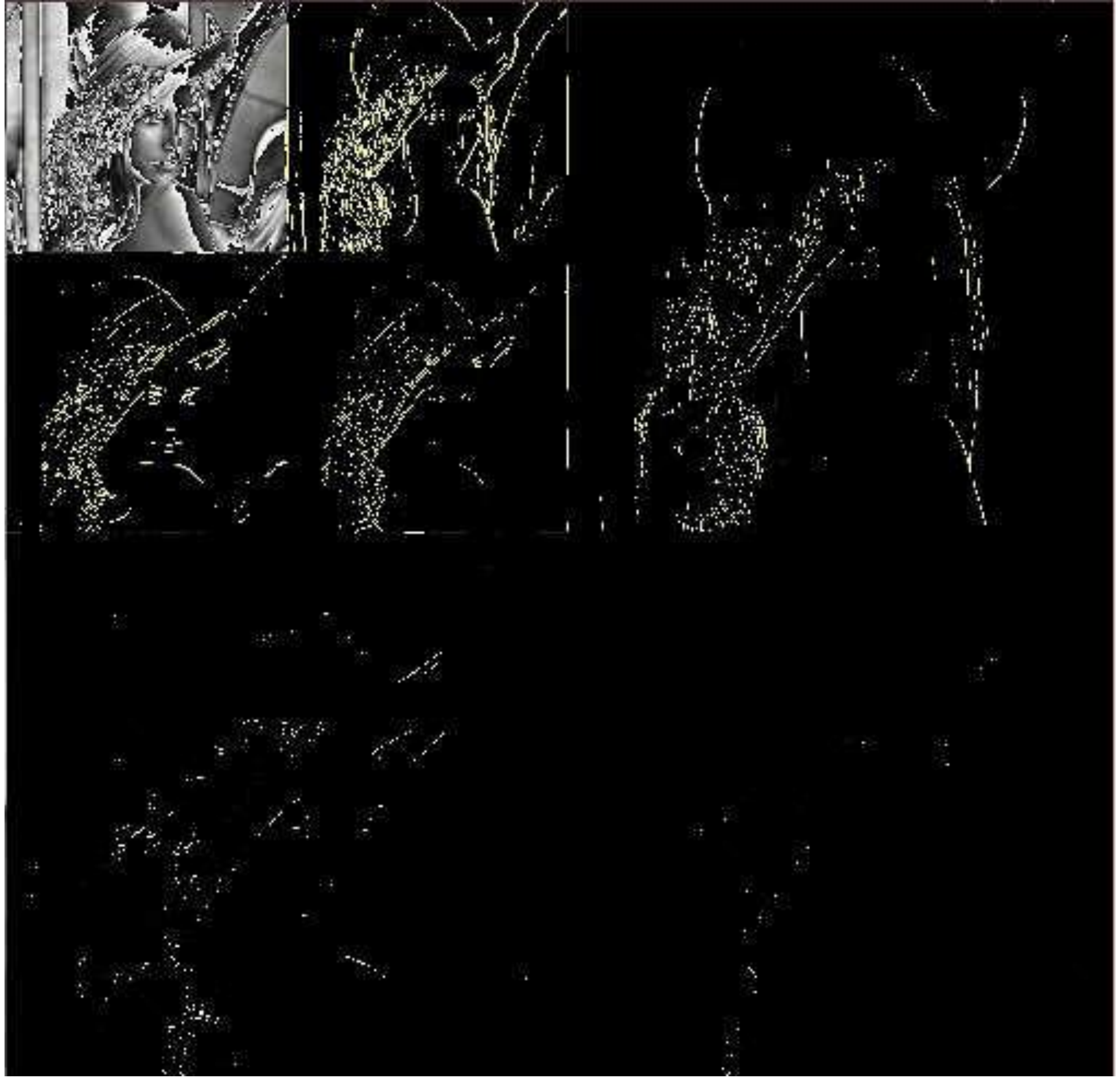
- M is orthonomal so $M^{-1}=M^{T}$.
- To be normalized, $h_0^2+h_1^2+h_2^2+h_3^2=1$.
- Innerproducts of columns are zero.
- Innerproducts of rows $h_0h_2+h_1h_3=0$.
- Wave should be smooth (Daubechies):
- $h_0-h_1+h_2-h_3=0$
- $4h_0-3h_1+2h_2-h_3=0$
- $h_0=0.483$, $h_1=0.837$, $h_2=0.224$, $h_3=-0.129$.

Wavelets with Integer Arithmetic

- Approximates spline wavelets.
- Can be implemented using integer arithmetic.
- Lossless and reversible for digital images.
- mandatory for JPEG2000
- Indexing requires symmetric extension of the arrays.



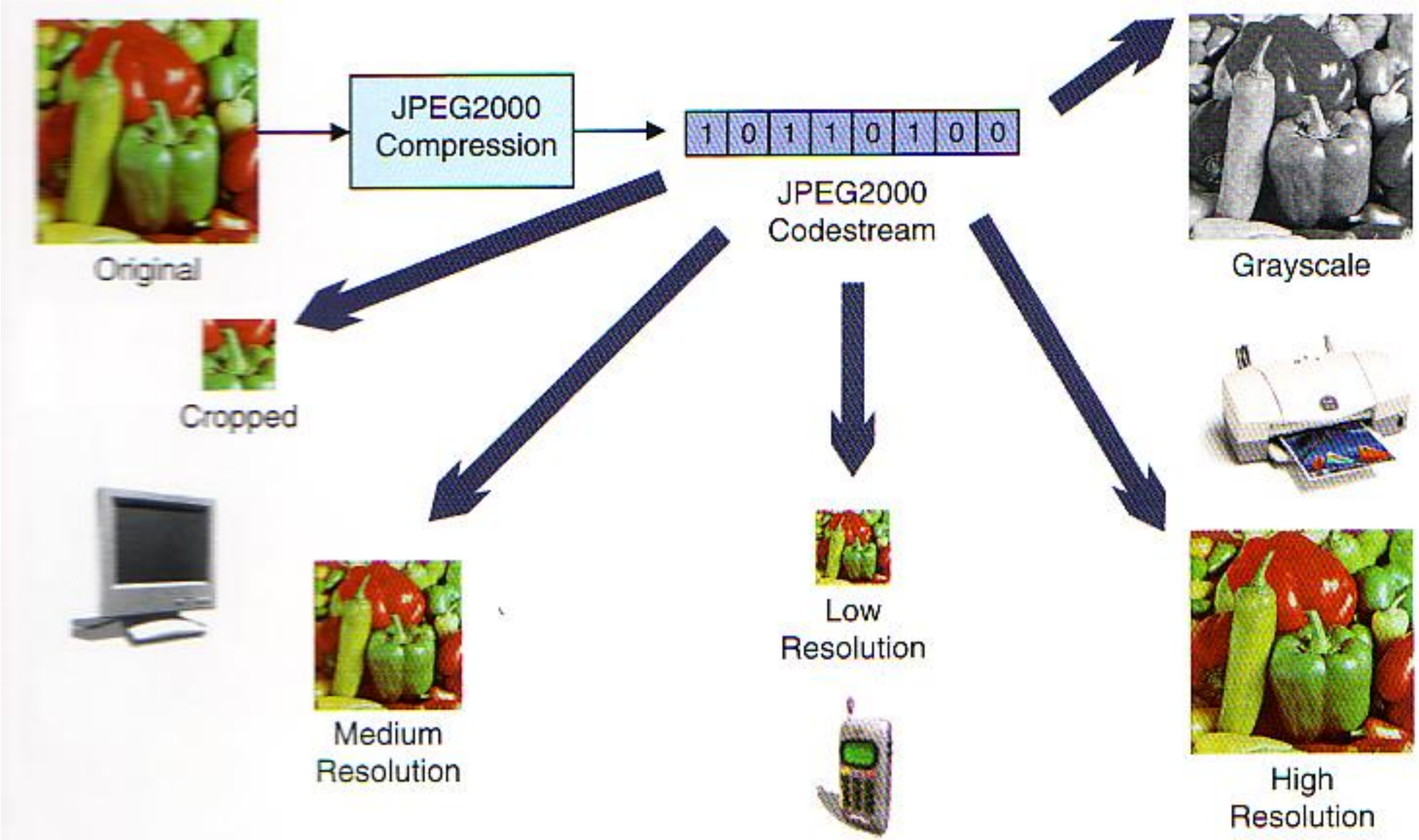




JPEG2000

- Single codestream for lossy and lossless compression of various resolutions.
- Locate, extract, and decode only the bytes of the desired image product.
- Uses wavelet transform.
- The minimum decoder supports the (9,7) wavelet transform and the (5,3) wavelet transform.

JPEG2000 Functionality



JPEG at 0.088 and 0.155 bpp

JPEG2000 at 0.088/0.155 bpp

(5,3) Wavelet Transform

$$C(2i+1) = P(2i+1) - \left\lfloor \frac{P(2i)+P(2i+2)}{2} \right\rfloor, \quad \text{for } k-1 \le 2i+1 < m+1,$$
$$C(2i) = P(2i) + \left\lfloor \frac{C(2i-1)+C(2i+1)+2}{4} \right\rfloor, \quad \text{for } k \le 2i < m+1.$$

$$\begin{array}{ccc} \xleftarrow{\quad l \quad} & & \xrightarrow{\quad r \quad} \\ \ldots\text{DEFGFEDCB} & |\underset{\underset{k}{\uparrow}}{\text{A}}\text{BCDEF}\underset{\underset{m}{\uparrow}}{\text{G}}| & \text{FEDCBABCD}\ldots \end{array}$$

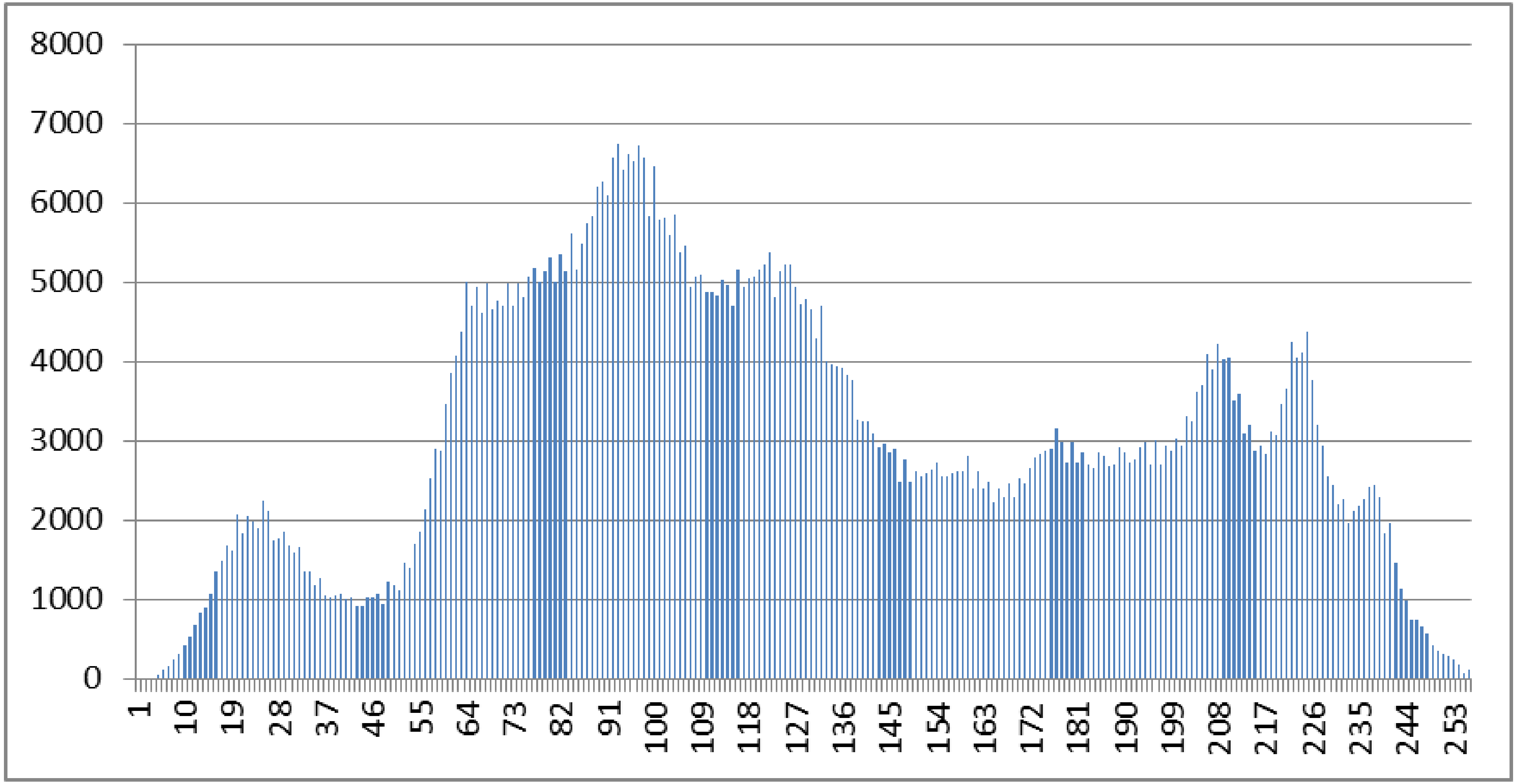
Figure 8.70: Extending a Row of Pixels.

(5,3) Transform on Rows

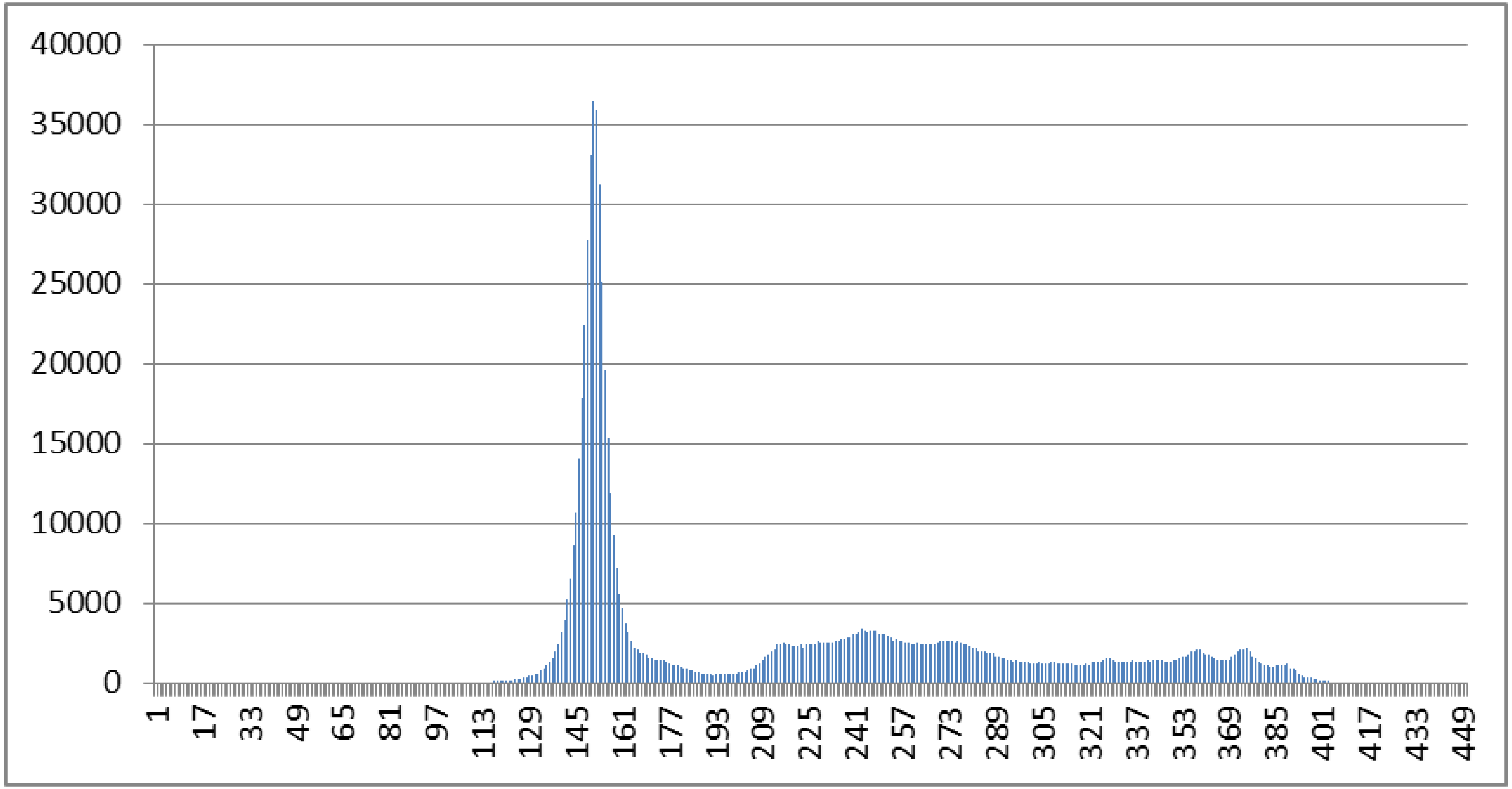
```
void rowTransform(){
   int halfWidth = width / 2;
   for (int k = 0; k < 3; k++)
    for (int i = 0; i < height; i++){
     for (int j = 0; j < halfWidth - 1; j++)
       coeffs[i][2 * j + 1][k] = raw[i][2 * j + 1][k] -
        (raw[i][2 * j][k] + raw[i][2 * j + 2][k]) / 2;
     coeffs[i][width - 1][k] =
       raw[i][width - 1][k] - raw[i][width - 2][k];
     for (int j = 1; j < halfWidth; j++)
       coeffs[i][2 * j][k] = raw[i][2 * j][k] +
   (coeffs[i][2 * j - 1][k] + coeffs[i][2 * j + 1][k] + 2) / 4;
     coeffs[i][0][k] = raw[i][0][k] + (coeffs[i][1][k] + 1) / 2;
   }
 }
```

LenaRGB.bmp 512x512, 24-bit

Lena Pixel Values Entropy = 7.75



Lena Row (5,3) Wavelet entropy = 7.17



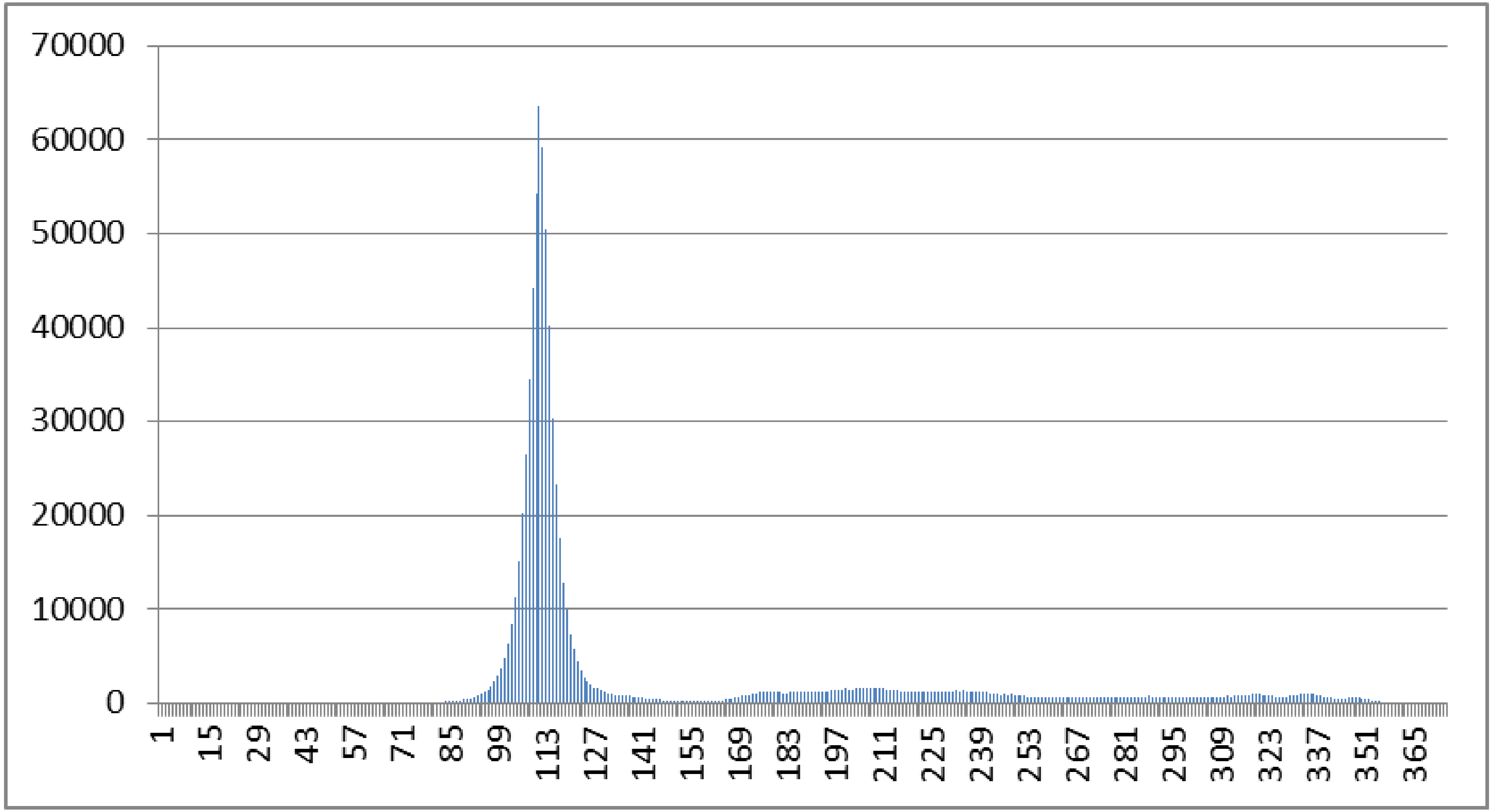
Inverse of Row Transform

```
void reverseRowTransform(){
   int halfWidth = width / 2;
   for (int k = 0; k < 3; k++)
    for (int i = 0; i < height; i++){
     raw[i][0][k] = coeffs[i][0][k] - (coeffs[i][1][k] + 1) / 2;
     for (int j = 1; j < halfWidth; j++)
       raw[i][2 * j][k] = coeffs[i][2 * j][k] -
   (coeffs[i][2 * j - 1][k] + coeffs[i][2 * j + 1][k] + 2) / 4;
     raw[i][width - 1][k] =
       coeffs[i][width - 1][k] + raw[i][width - 2][k];
     for (int j = 0; j < halfWidth - 1; j++)
       raw[i][2 * j + 1][k] = coeffs[i][2 * j + 1][k] +
        (raw[i][2 * j][k] + raw[i][2 * j + 2][k]) / 2;
   }
 }
```

(5,3) Transform on Columns

```
void columnTransform(){
   int halfHeight = height / 2;
   for (int k = 0; k < 3; k++)
    for (int j = 0; j < width; j++){
     for (int i = 0; i < halfHeight - 1; i++)
       raw[2 * i + 1][j][k] = coeffs[2 * i + 1][j][k] -
        (coeffs[2 * i][j][k] + coeffs[2 * i + 2][j][k]) / 2;
     raw[height - 1][j][k] = coeffs[height - 1][j][k] - coeffs[height - 2][j][k];
     for (int i = 1; i < halfHeight; i++)
       raw[2 * i][j][k] = coeffs[2 * i][j][k] +
        (raw[2 * i - 1][j][k] + raw[2 * i + 1][j][k] + 2) / 4;
     raw[0][j][k] = coeffs[0][j][k] + (raw[1][j][k] + 1) / 2;
   }
 }
```

Lena Row and Column Entropy = 6.08



Shrinkage: LenaRGB with 71% zeros

```
void shrink(){
  int threshold = 12;
  for (int k = 0; k < 3; k++)
    for (int i = 0; i < height; i++)
     for (int j = 0; j < width; j++){
      if (i % 2 > 0 || j % 2 > 0){
        if (raw[i][j][k] > 0 && raw[i][j][k] <= threshold) raw[i][j][k] = 0;
        if (raw[i][j][k] < 0 && raw[i][j][k] >= -threshold) raw[i][j][k] = 0;
        if (raw[i][j][k] != 0) raw[i][j][k] = raw[i][j][k] / 8 + 128;
    }
    if (raw[i][j][k] < 0) raw[i][j][k] = 0;
    if (raw[i][j][k] > 255) raw[i][j][k] = 255;
  }
}
```

Lossy Inverse of Shrinkage

```
void unshrink(){
  for (int k = 0; k < 3; k++)
    for (int i = 0; i < height; i++)
     for (int j = 0; j < width; j++)
      if (i % 2 > 0 || j % 2 > 0)
         if (raw[i][j][k] > 0)
           raw[i][j][k] = (raw[i][j][k] - 128) * 8;
}
```

Homework 8: due 2-11-15

- Implement the inverse of H8A.columnTransform() as H8B.reverseColumnTransform() and make H8B the inverse of H8A.
- H8A and H8B are (5,3)-wavelet transform and its inverse of rows and columns of an image with RGB colors.
- Recover the original image from test8.bmp.