Chapter 3
Digital Image Processing

CS 3570

OBJECTIVES FOR CHAPTER 3

- Know the important file types for digital image data.
- Understand the difference between fixed-length and variable-length encoding schemes.
- Understand the application of LZW compression, Huffman encoding, and JPEG compression.
- Understand the luminance and chrominance downsampling.
- Understand the indexed color algorithms.
- Understand the dithering algorithms.
- Understand pixel point processing in digital image processing.
- Understand how convolutions are applied in filtering for enlarging, reducing, or sharpening images.
- Understand resampling and interpolation.

Digital image type

- If you take a picture with a digital camera or scan a photograph with a digital scanner, you'll have a choice of file types in which to save the image.
- You'll also have a choice of file types when you save an image in an image processing, paint, or drawing program.
- Not all color modes can be accommodated by all file types, and some file types require that the image be compressed while some do not.

Digital image type – bitmap - 1

File Suffix	File Type	Characteristics
.bmp	Windows bitmap	1 to 24-bit color depth, 32-bit if alpha channel is used. Can use lossless RLE or no compression. RGB or indexed color.
.gif	Graphics Interchange Format	Used on the web. Allows 256 RGB colors. Can be used for simple animations. Uses LZW compression. Originally proprietary to CompuServe.

- Both support ***transparency***, but BMP supported ***alpha channel***.
- GIF files can be saved in an interlaced format that allows ***progressive download*** of web images. In ***progressive download***, a low-resolution version of an image is downloaded first, and the image gradually comes into focus as the rest of the data is downloaded.

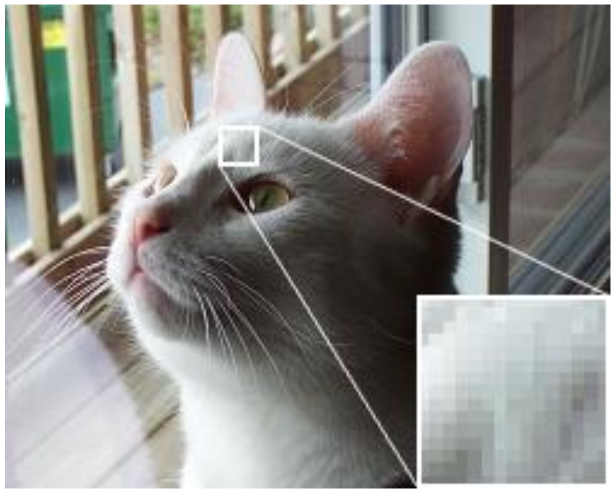
Digital image type – bitmap - 2

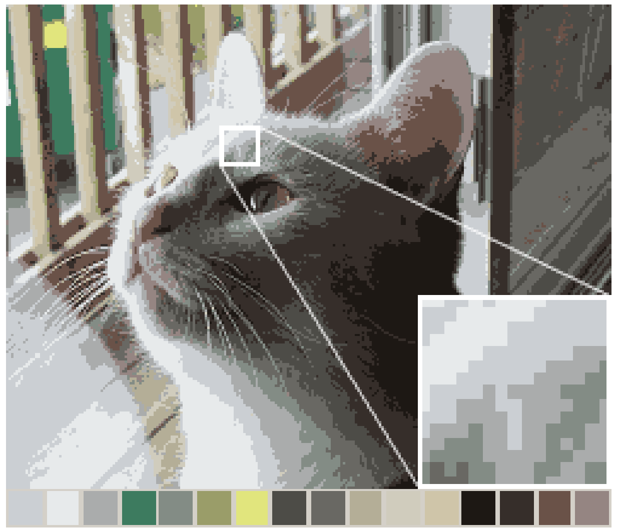
File Suffix	File Type	Characteristics
.jpeg or *.jpg*	Joint Photographic Experts Group	For continuous tone pictures. Lossy compression. Level of compression can be specified.
.png	Portable Network Graphics	Designed as an alternative to *.gif* files. Compressed with lossless method. 1 to 64-bit color with transparency channel.

- GIF and JPG files are also widely used on the web.
- Bitmap files in indexed color mode generally use 8 bits (or more) per pixel to store an index into a color table, called a ***palette.***
- Like BMP files, PNG files allow the use of the ***alpha channel***.
- The bits in the alpha channel indicate the level of transparency of each pixel.

Color Quantization - 1

- The process of reducing the number of colors in an image file is called ***color quantization***.
- In image processing programs, the color mode associated with color quantization is called ***indexed color***.
- Indexed color is a technique to manage digital images' colors in a limited fashion, in order to save computer memory and file storage.


24-bit RGB Color


Using 16 colors

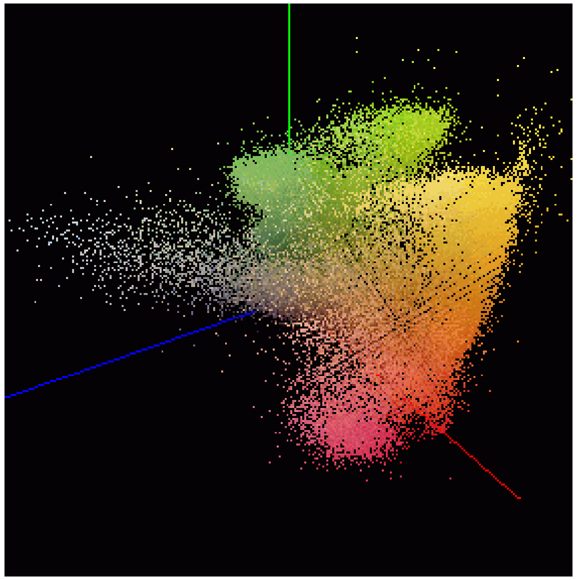
Color Quantization - 2

- Color quantization begins with an image file stored with a bit depth of *n* and reduces the bit depth to *b*.
- The number of colors representable in the original file is 2^n, and the number of colors representable in the adjusted file will be 2^b.
- As an example, let's assume your image is initially in RGB mode with 24-bit color, and you want to reduce it to 8-bit color.

Process of Color Quantization - 1

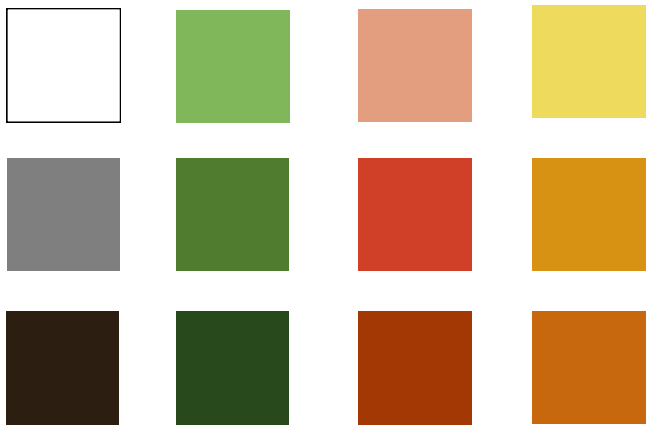
- The process of color quantization involves three steps

1. The actual range and number of colors used in your picture must be determined. If your image is stored initially in RGB mode with 24-bit color, then there are 2^{24} = 16,777,216 possible colors. The question is, ***which of these colors appear in the picture?***



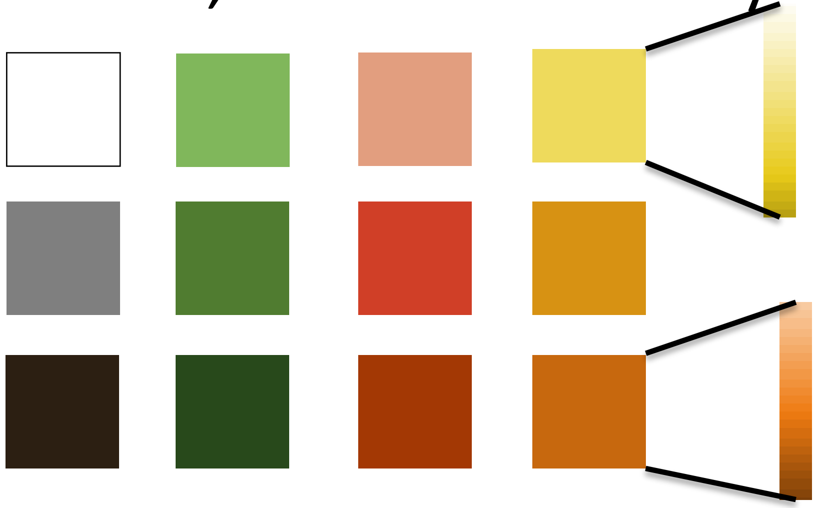
Process of Color Quantization – 2

2. The second step entails choosing 2^b colors to represent those that actually appear in the picture. For our example, the adjusted picture would be limited to $2^8 = 256$ colors



Process of Color Quantization – 3

3. To map the colors in the original picture to the colors chosen for the reduced bit-depth picture. The *b* bits that represent each pixel then become an index into a color table that has 2^b entries, where each entry is *n* bits long. In our example, the table would have 256 entries, where each entry is 24 bits long.



Popularity algorithm

- One simple way to achieve a reduction from a bit depth of *n* to a bit depth of *b*.

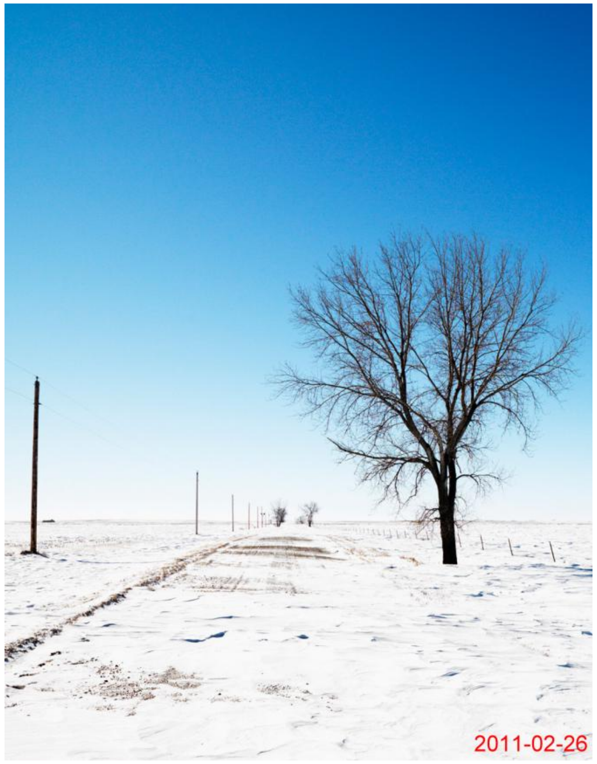
1. The 2^b colors that appear most often in the picture are chosen for the reduced-bit depth picture
2. To map one of the original colors to the more limited palette is by finding the color that is most similar using the minimum mean squared distance.

-> min $(R-r_i)^2+ (G-g_i)^2+(B-b_i)^2$

R,G,B: original color
$r_{i,}g_{i,}b_{i:}$ quantized color

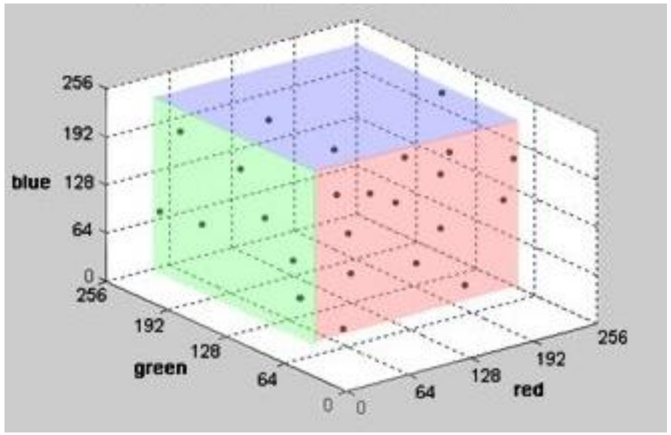
Popularity algorithm - Disadvantage

- Disadvantage: it completely throws out colors that appear infrequently
  - Example:
    A picture with one dramatic spot of red in a field of white snow, trees, and sky may lose the red spot entirely, completely changing the desired effect



Uniform partitioning Algorithm – 1

1. Finding the smallest "box" that contains all the colors appearing in the image
2. Divide the subspace containing the existing colors into 2^b blocks of equal size



Smallest box in RGB 3D space

Uniform partitioning for index color

- Example: the color space partitioned uniformly giving eight values of red, eight of green, and four of blue.

Range of Reds in the Original Image (decimal values)	Range of Red in the Original Image (binary values)	Index to Which They Map in the Color Table
0–31	00000000–00011111	0
32–63	00100000–00111111	1
64–95	01000000–01011111	2
96–127	01100000–01111111	3
128–159	10000000–10011111	4
160–191	10100000–10111111	5
192–223	11000000–11011111	6
224–255	11100000–11111111	7

Uniform partitioning Algorithm - disadvantage

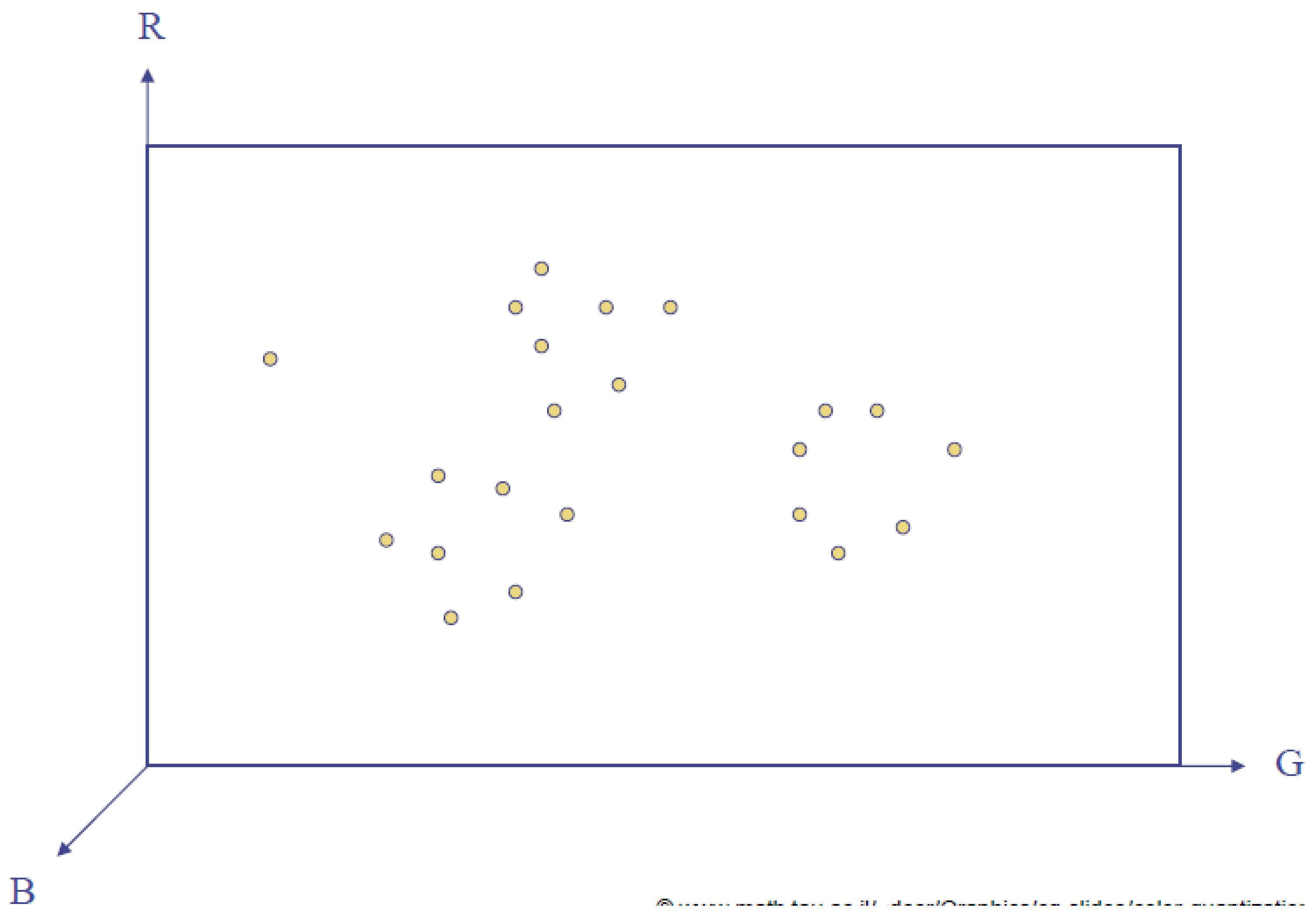
- It does not account for the fact that the equal-sized partitions of the color space may not be equally populated. There may be many colors in one partition, and only a few in another. All the colors in a heavily populated partition will be converted to a single color, and smooth transitions of color will be lost in the image.

Median-cut algorithm - 1

- The ***median-cut algorithm*** is superior to uniform partitioning in that it does a better balancing of the number of colors in a partition.

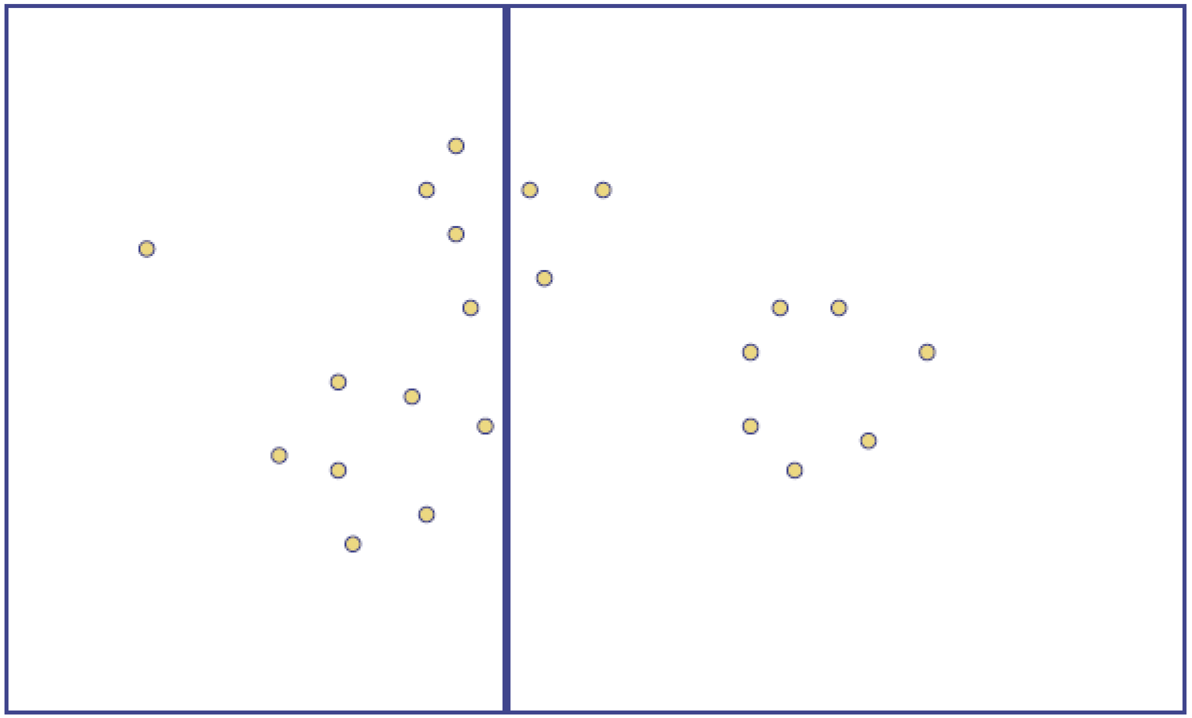
1. To reduce the RGB color space to the smallest block containing all the colors in the image.
2. The algorithm proceeds by a stepwise partitioning where at each step some sub-block containing $2^n = c$ colors is divided in half along its longest dimension such that $c/2$ of the existing colors from the image are in one half and $c/2$ are in the other.

Median-cut algorithm - 2



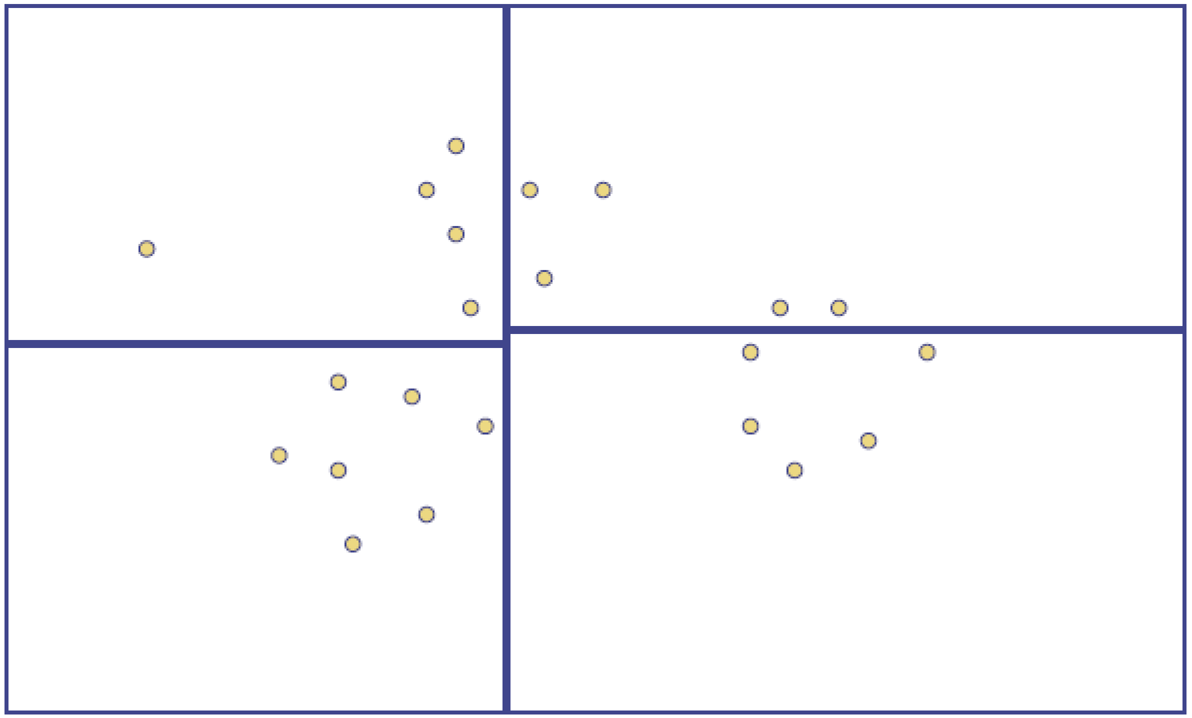
[illegible]

Median-cut algorithm - 3



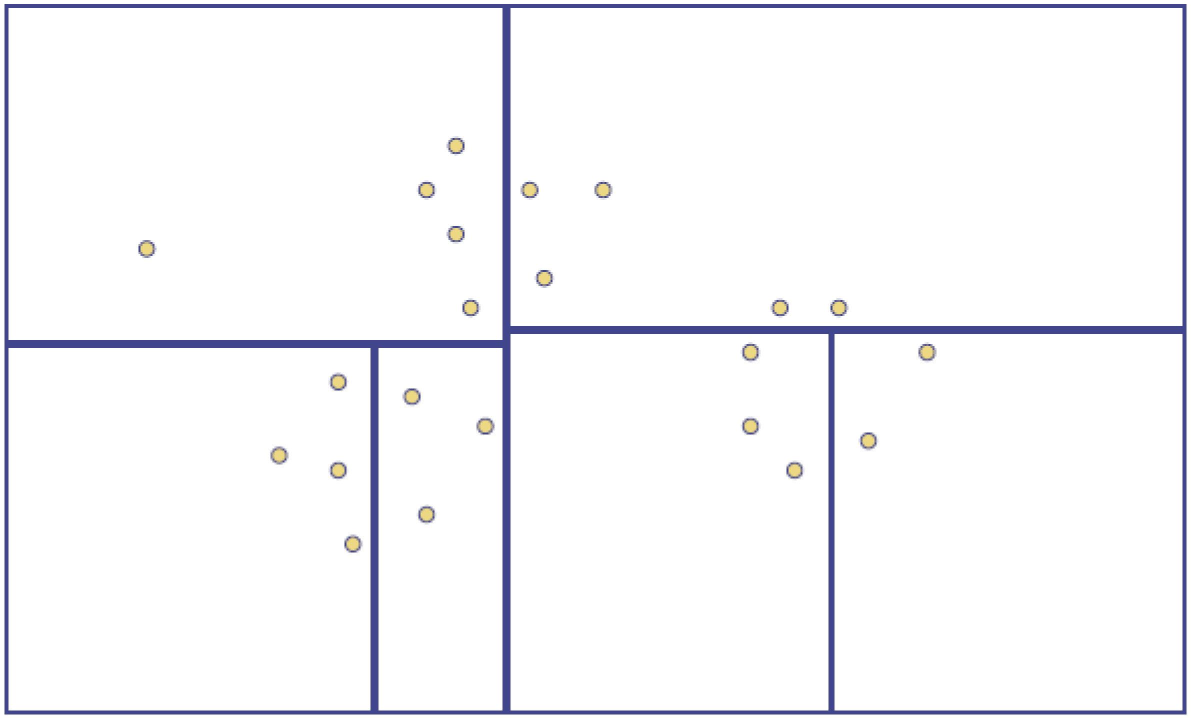
©www.math.tau.ac.il/~dcor/Graphics/cg-slides/color_quantization

Median-cut algorithm - 4



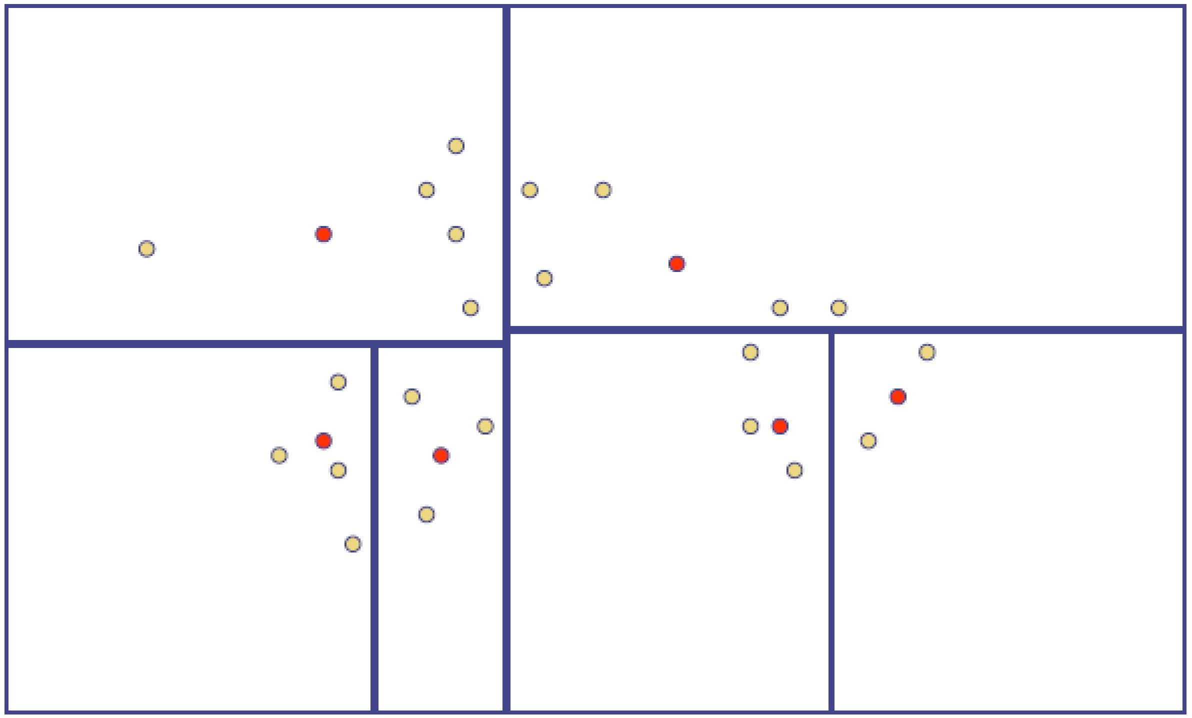
[illegible]

Median-cut algorithm - 5



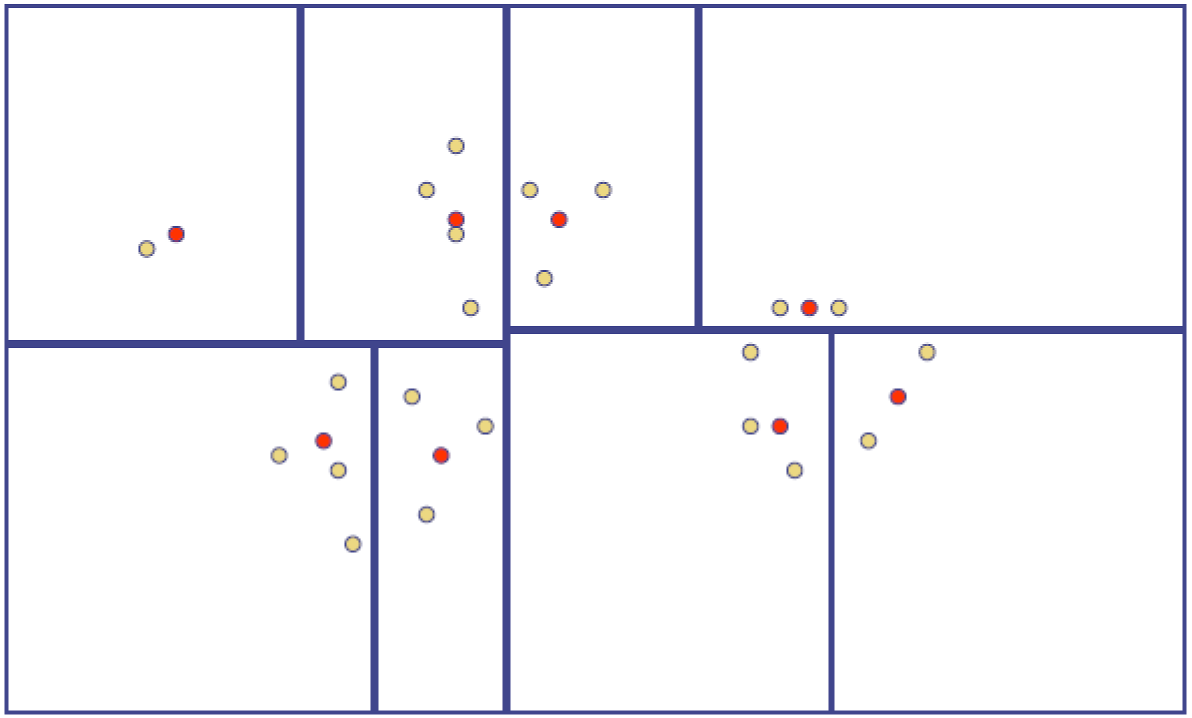
[illegible]

Median-cut algorithm - 6



[illegible]

Median-cut algorithm - 7



© www.math.tau.ac.il/~dcor/Graphics/cg-slides/color_quantization

Median-cut algorithm - 8

Median-cut Algorithm

```
Color_quantization(Image, n){
    For each pixel in Image with color C, map C in RGB space;
    B = {RGB space};
    While (n-- > 0) {
        L = Heaviest (B);
        Split L into L1 and L2;
        Remove L from B, and add L1 and L2 instead;
    }
    For all boxes in B do
        assign a representative (color centroid);
    For each pixel in Image do
        map to one of the representatives;
}
```

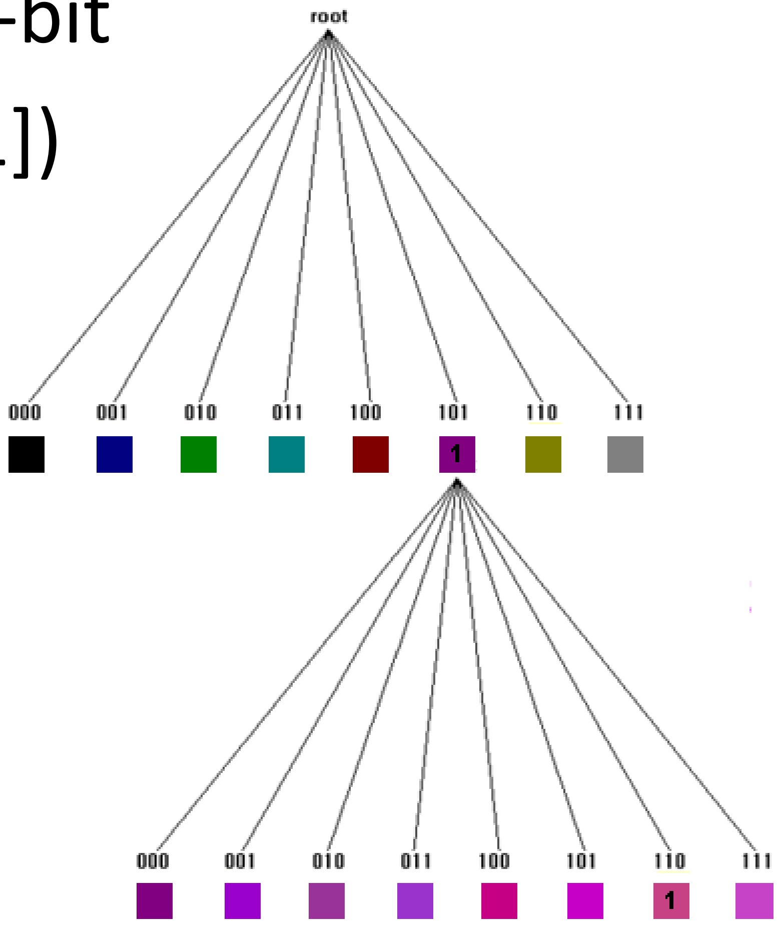
Octree algorithm - 1

- The ***octree algorithm*** is similar to the median-cut algorithm in that it partitions the color space with attention to color population

1. determining the colors to use in the reduced-bit depth image
2. converting the original image to fewer colors on the basis of the chosen color table

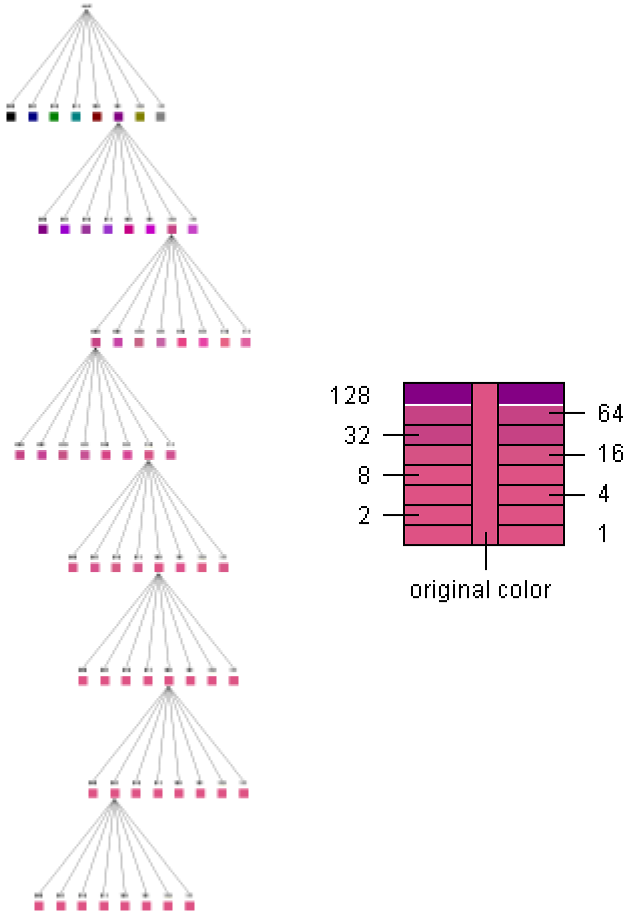
Octree algorithm - 2

- Example: reduce 24-bit to 8-bit
- Insert first pixel([220 81 131]) into octree
    - R: 11011100
    - G: 01010001
    - B: 10000011



Octree algorithm - 3

- The nodes are recorded with the times they are visited
- The lower the level, the more similar to the original color

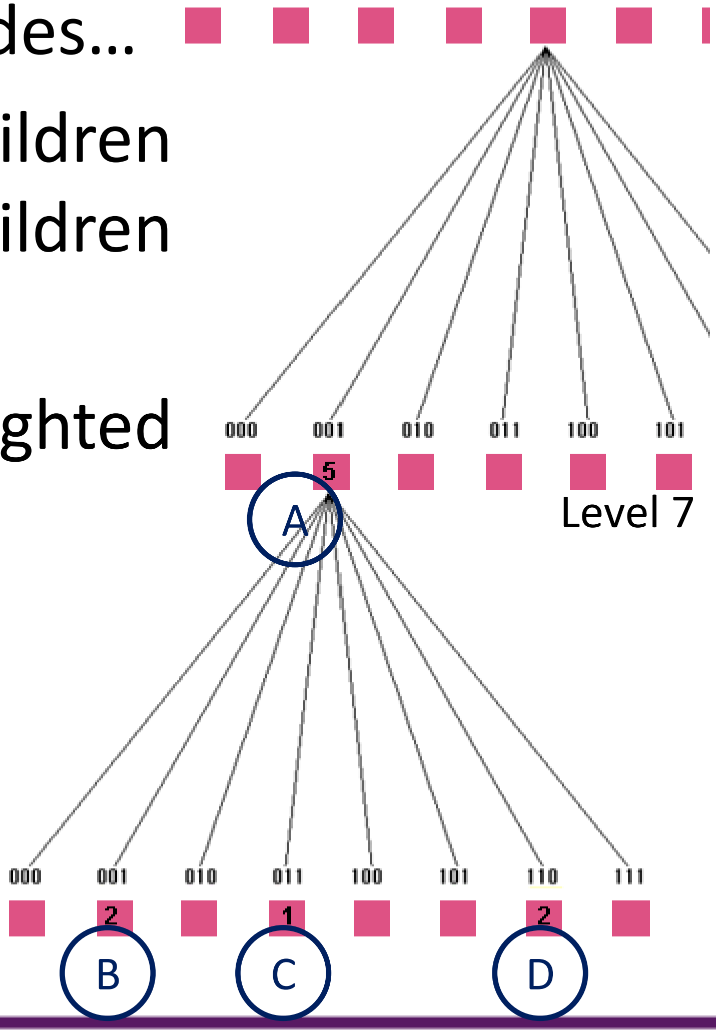


Octree algorithm - 4

- This yields a maximum of 8^8 leaf nodes. (Note that $8^8 = 2^{24}$) However, only 2^b leaf nodes are actually created. If processing a pixel results in the creation of a leaf node $2^b + 1$, some nodes need to be combined
  - A reducible node
    - ✓one that has at least two children
    - ✓must be found at the lowest possible level

Octree algorithm - 5

- If we have more than 256 leaf nodes…
- Branch '001' at level 7 has 3 children nodes. => Simply delete its children nodes.
- Then we can compute the weighted average color of BCD for A



Octree algorithm - Advantage

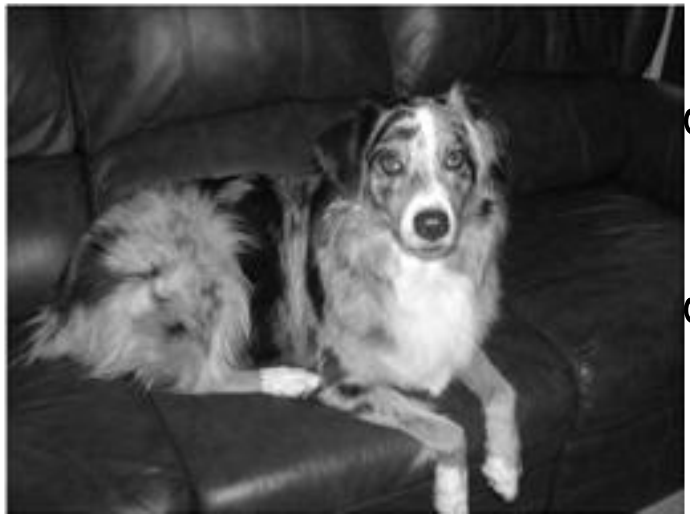
- Use the image’s original colors if possible.
- Average similar colors when this is not possible.
- It is also efficient in its implementation, since the tree never grows beyond a depth of eight.

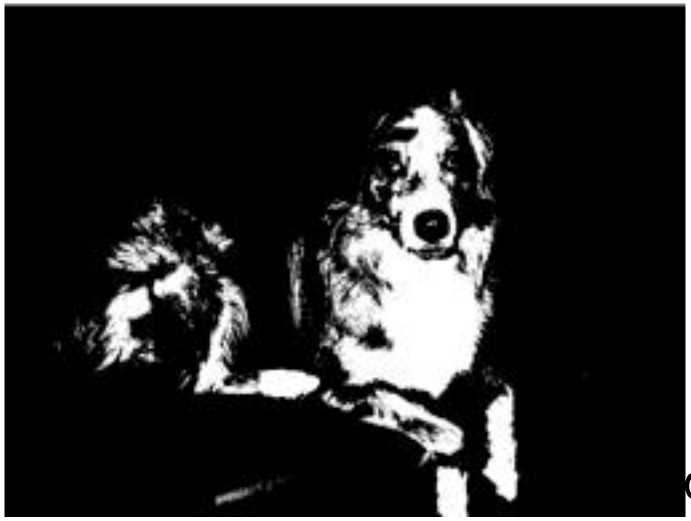
Dithering - 1

- ***Dithering*** is a technique for simulating colors that are unavailable in a palette by using available colors that are blended by the eye so that they look like the desired colors
- Dithering is helpful when you change an image from RGB mode to indexed color because it makes it possible to reduce the bit depth of the image, and thus the file size, without greatly changing the appearance of the original image

Thresholding

- If you have a grayscale image that uses eight bits per pixel and decide to reduce it to a black and white bitmap that uses one bit per pixel.
- A sensible algorithm to accomplish this would change pixel values less than 128 to black and values greater than or equal to 128 to white.
- This is called *thresholding.*


Origin image


After thresholding

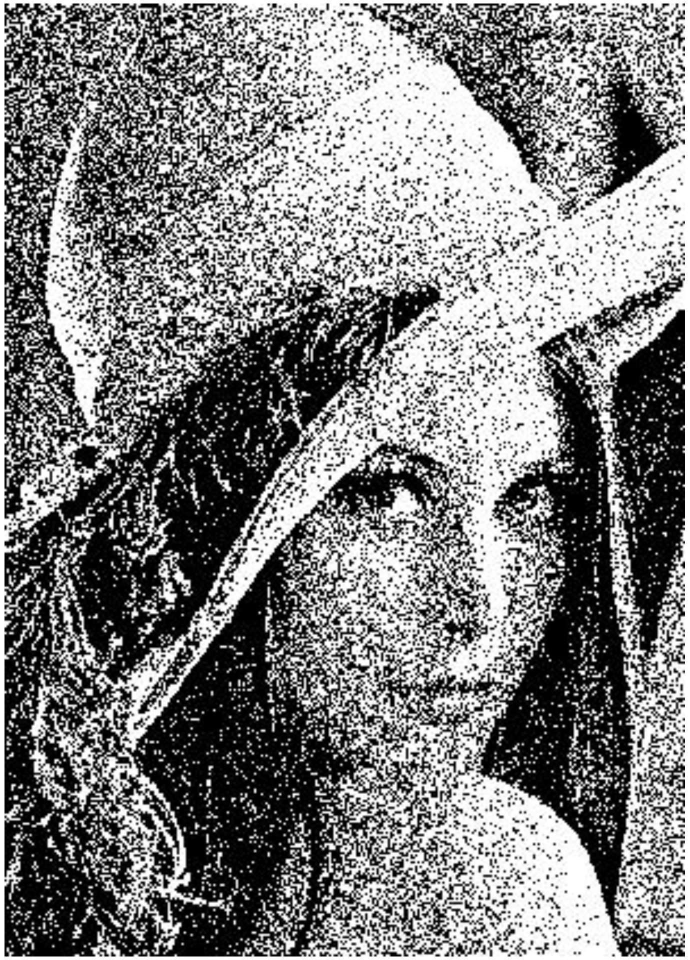
Noise dithering - 1

- Also called ***random dithering***

1. Generate a random number from 0 to 255
2. If the pixel's color value is greater than the number then it is white, otherwise black
3. Repeat step 2 for each pixel in the image

- Crude and "noisy"

Noise dithering - 2

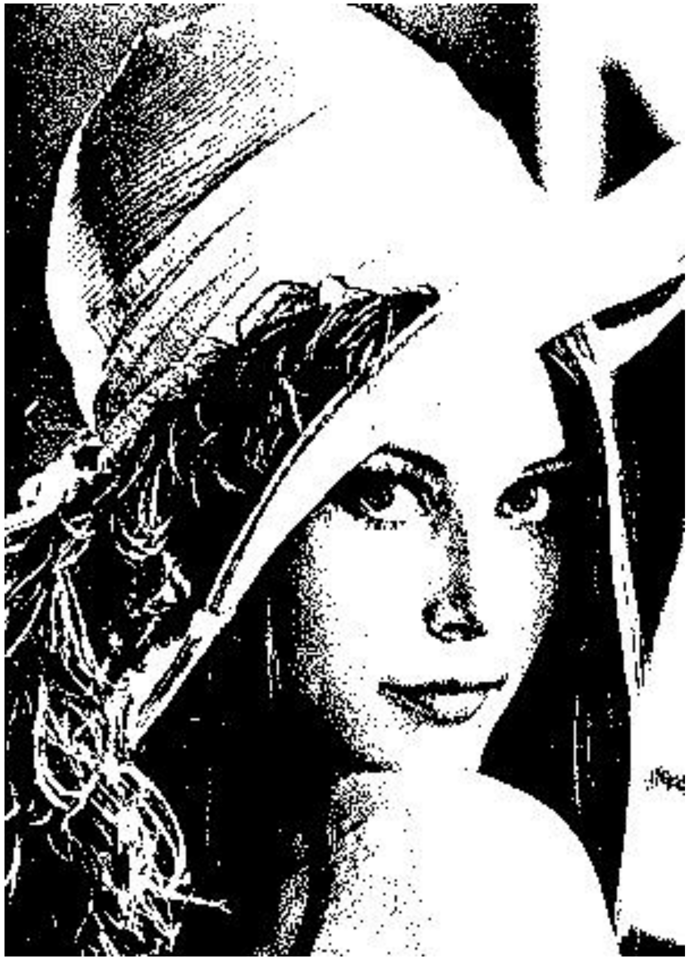


Average dithering - 1

1. Calculate an average pixel value
2. If each pixel is above this then white, else black
3. Repeat step 2 for each pixel in the image

- Crude and “contrasty”

Average dithering - 2



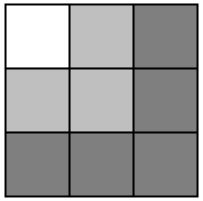
Pattern dithering - 1

- Also called ***ordered dithering***, ***Bayer method***

1. Break the image into small blocks
2. Define a *threshold matrix*
   - Use a different threshold for each pixel of the block
   - Compare each pixel to its own threshold

- Widely used by the printing industry - rare in multimedia

Image Block

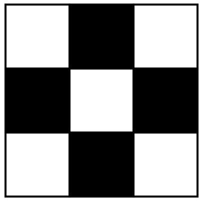


$$\begin{bmatrix} 255 & 192 & 128 \\ 192 & 192 & 128 \\ 128 & 128 & 128 \end{bmatrix}$$

Threshold matrix

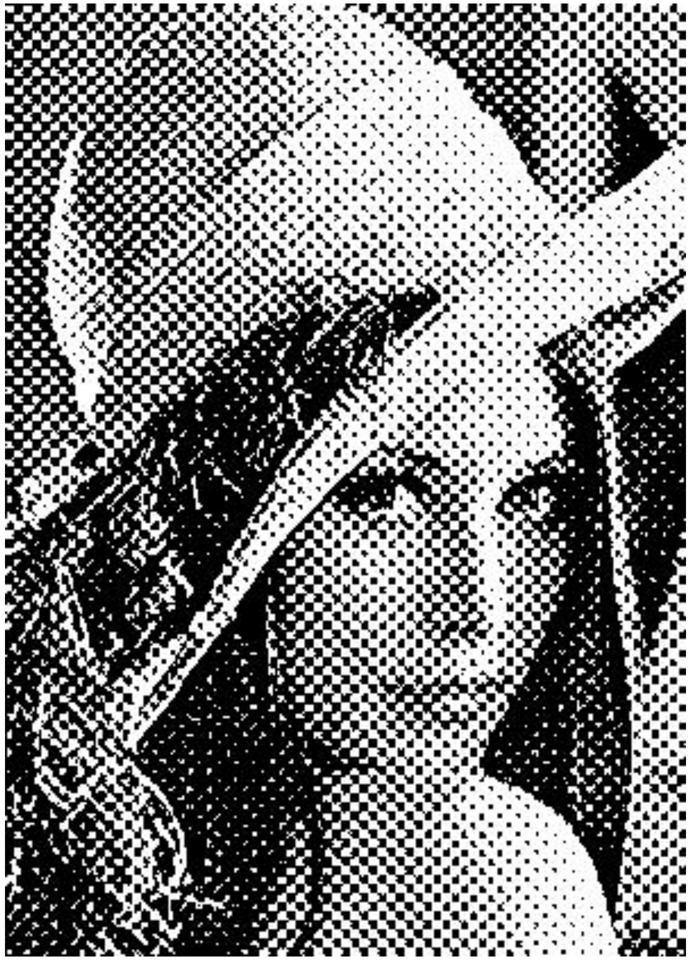
$$\begin{bmatrix} 200 & 250 & 100 \\ 220 & 150 & 200 \\ 10 & 150 & 50 \end{bmatrix}$$

Result



$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

Pattern dithering - 2



Error Diffusion Dithering - 1

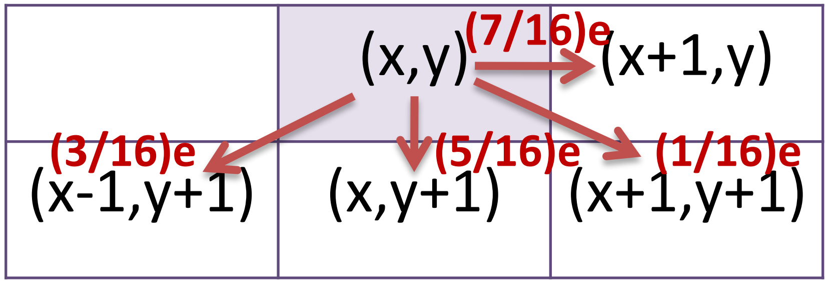
- Also called ***Floyd-Steinberg algorithm***
- Disperses the error, or difference between a pixel's original value and the color (or grayscale) value available

Error Diffusion Dithering - 2

1. Define the mask, for example:

	p	7
3	5	1

2. For each pixel p
    1. Define e: if p<128, e = p; otherwise e = p-255
    2. Change the value of neighbor pixel

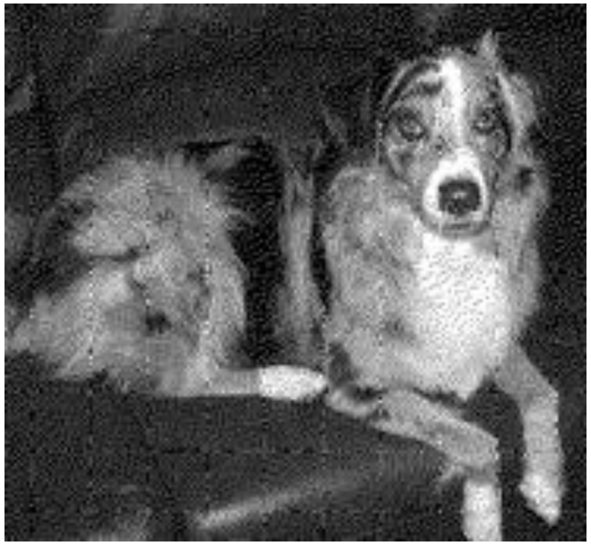


p(x+1,y) = p(x+1,y)+(7/16)e
p(x-1,y+1) = p(x-1,y+1)+(3/16)e
p(x,y+1) = p(x,y+1)+(5/16)e
p(x+1,y+1) = p(x+1,y+1)+(1/16)e

- When the mask is moved to the right by one pixel, the next step will operate on a pixel that has possibly been changed in a previous step

Error Diffusion Dithering - 3

- After the error has been distributed over the whole image, the pixels are processed a second time. This time for each pixel, if the pixel value is less than 128, it is changed to a 0 in the dithered image. Otherwise it is changed to a 1.



After dithering

Image transform - 1

- You want to create an effect on an image
- You want your image to be as clear and detailed as possible
- you want to provoke a certain mood or alter colors and shading that affect the aesthetics of the image
- you want to change the focus or emphasis

→**An image transform**:
a process of changing the color or grayscale values of image pixels

Image transform - 2

- Image transforms can be divided into two types
  - ***Pixel point processing***
    a pixel value is changed based only on its original value, without reference to surrounding pixels
  - ***Spatial filtering***
    changes a pixel's value based on the values of neighboring pixels

Histogram - 1

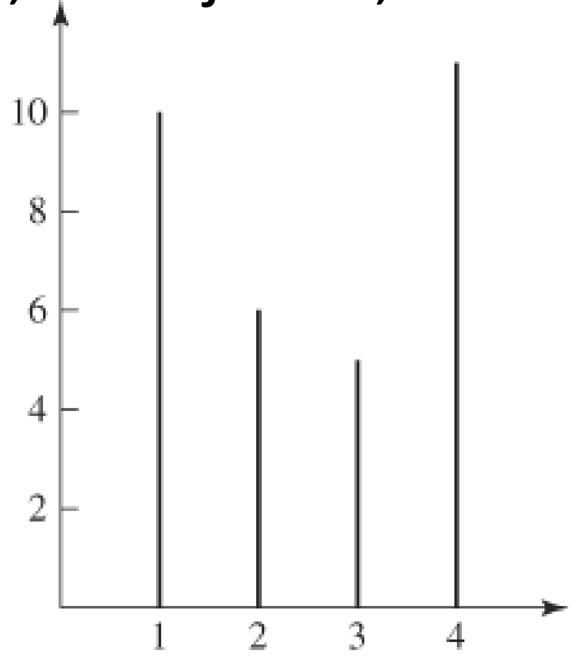
- A ***histogram*** is a discrete function that describes frequency distribution; that is, it maps a range of discrete values to the number of instances of each value in a group of numbers.
- Let v_i be the number of instances of value i in the set of numbers. Let *min* be the minimum allowable value of i, and let *max* be the maximum. Then the ***histogram function*** is defined as

$$h(i) = v_i \text{ for } min \le i \le max.$$

Histogram - 2

- Simple histogram

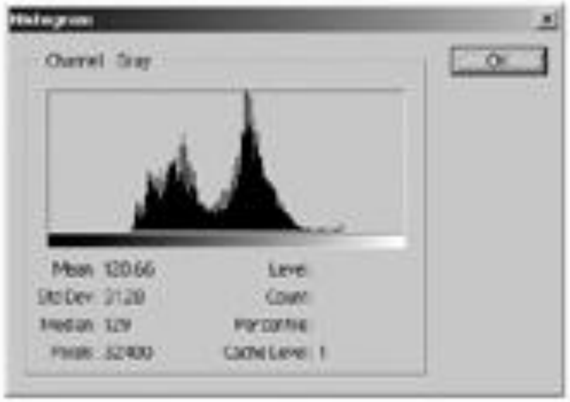
  a group of 32 students identified by class (1 for freshman, 2 for sophomore, 3 for junior, and 4 for senior)



- There are ten freshmen, six sophomores, five juniors, and 11 seniors

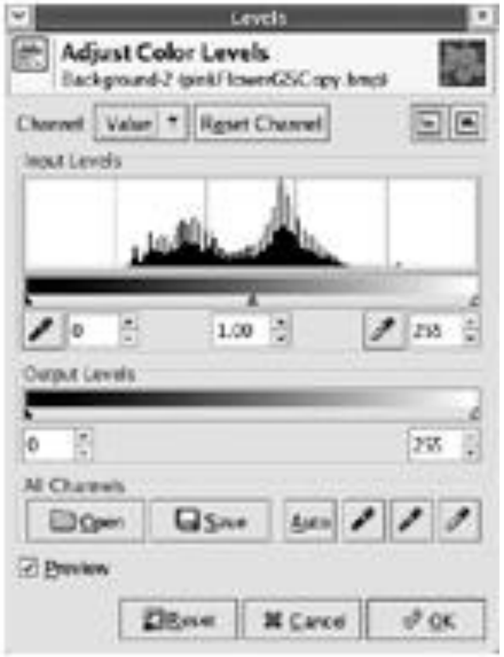
Histogram - 3

- An image histogram maps pixel values to the number of instances of each value in the image



Channel: Gray

Mean: 120.66
Std Dev: 21.20
Median: 129
Pixels: 32400
Level:
Count:
Percentile:
Cache Level: 1



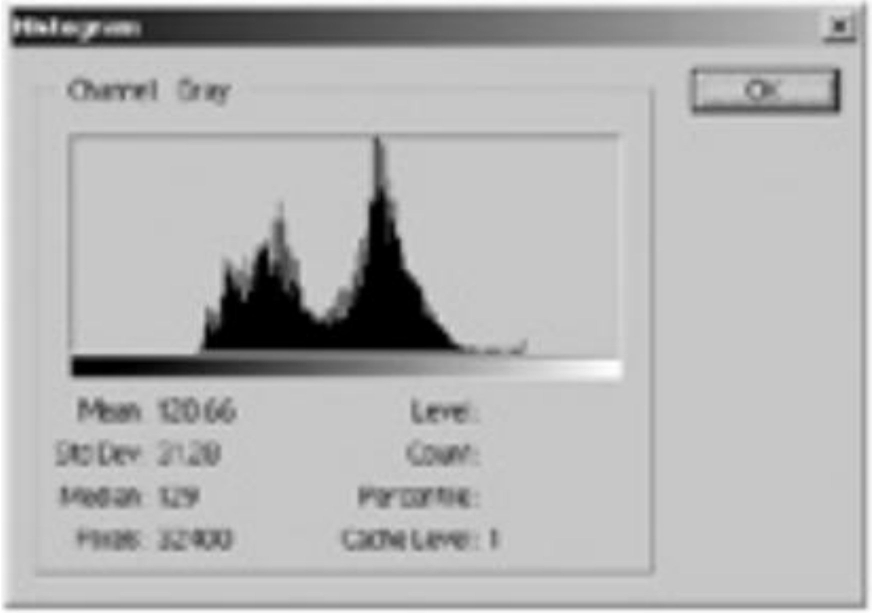
Channel: Value

Input Levels: 0 1.00 255

Output Levels: 0 255

Histogram - 4

- **Mean or average**
  - $\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$
- **Median**
  - A ***median*** is a value *x* such that at most half of the values in the sample population are less than *x* and at most half are greater
- **Standard deviation**
  - $\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2}$



```
   Mean: 120.66            Level:
Std Dev: 21.20             Count:
 Median: 129          Percentile:
 Pixels: 32400      Cache Level: 1
```

A large standard deviation implies that most of the pixel values are relatively far from the average, so the values are pretty well spread out over the possible range

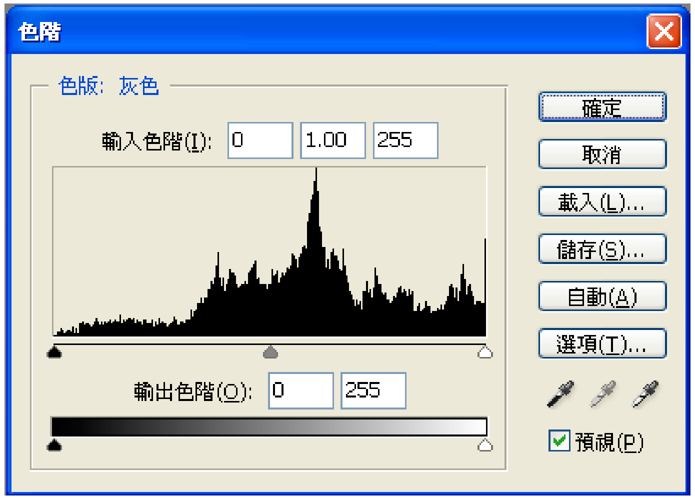
Luminance histogram

- Some scanners or image processing programs give you access to a ***luminance histogram*** (also called a ***luminosity histogram***) corresponding to a color image

$$L = 0.299R+0.587G+0.114B$$

- Among the three color channels, the human eye is most sensitive to green and least sensitive to blue

Histogram - 5



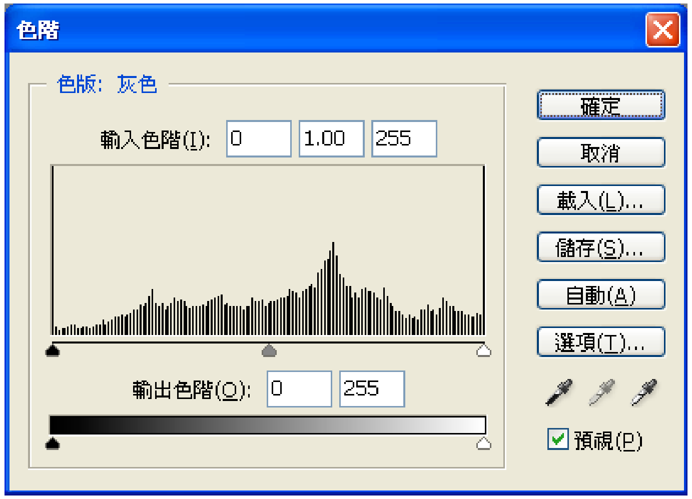
色版: 灰色

輸入色階(I): 0 1.00 255

輸出色階(O): 0 255

☑ 預視(P)

A histogram that can be manipulated to adjust brightness and contrast (from Photoshop)



色版: 灰色

輸入色階(I): 0 1.00 255

輸出色階(O): 0 255

☑ 預視(P)

Histogram of image after contrast adjustment (from Photoshop)

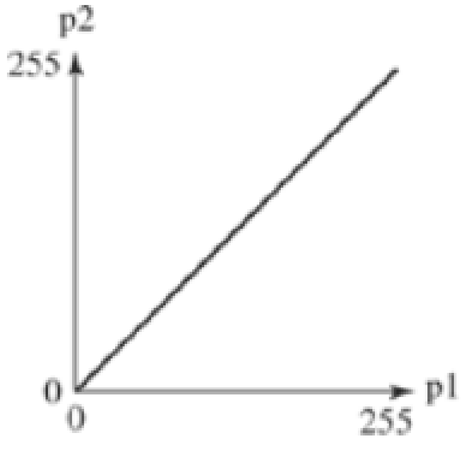
Transform function

- In programs like Photoshop and GIMP, the Curves feature allows you to think of the changes you make to pixel values as a transform function
- We define a ***transform*** as a function that changes pixel values

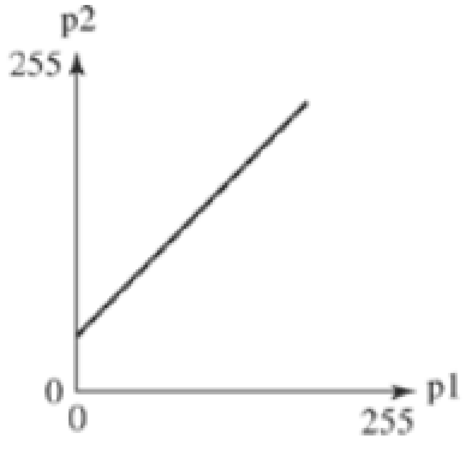
$$g(x, y) = T(f(x, y)) \text{ , } p_2=T(p_1)$$

  - $f(x, y)$ is the pixel value at that position (x, y) in the original image. Abbreviate $f(x,y)$ as p_1
  - T is the transformation function
  - $g(x, y)$ is the transformed pixel value. Abbreviate $f(x,y)$ as p_2

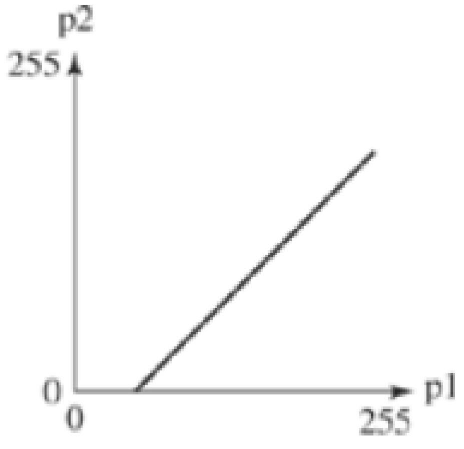
Curves - 1



(a)



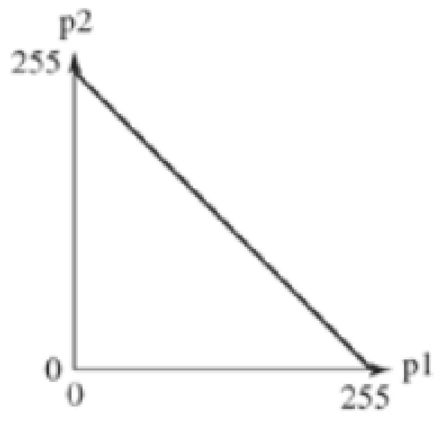
(b)



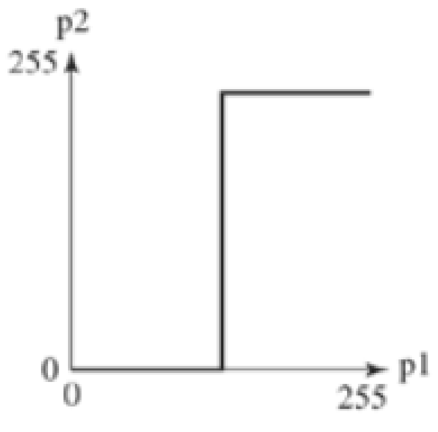
(c)

a) The transform doesn’t change the pixel values. The output equals the input.
b) The transform lightens all the pixels in the image by a constant amount.
c) The transform darkens all the pixels in the image by a constant amount.

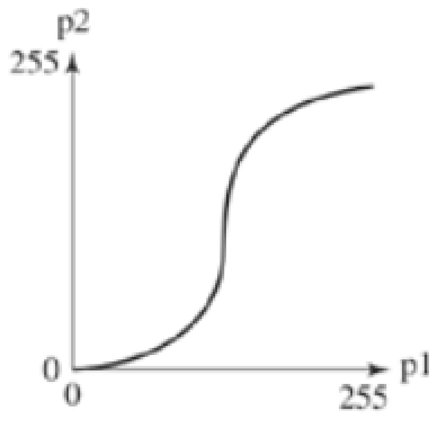
Curves - 2



(d)



(e)

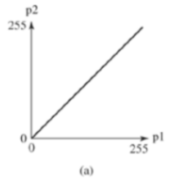


(f)

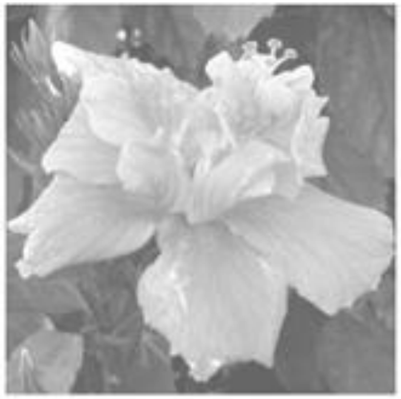
d) The transform inverts the image, reversing dark pixels for light ones.
e) The transform is a threshold function, which makes all the pixels either black or white. A pixel with a value below 128 becomes black, and all the rest become white.
f) The transform increases contrast. Darks become darker and lights become lighter.

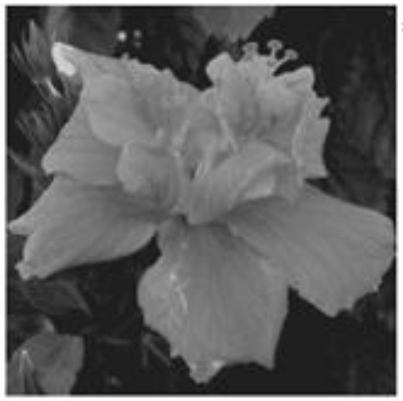
Apply curves

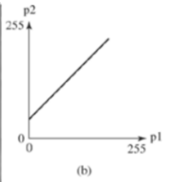
- Adjusting contrast and brightness with curves function

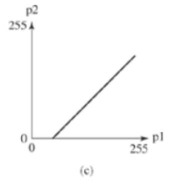


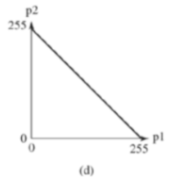
(a) Original image

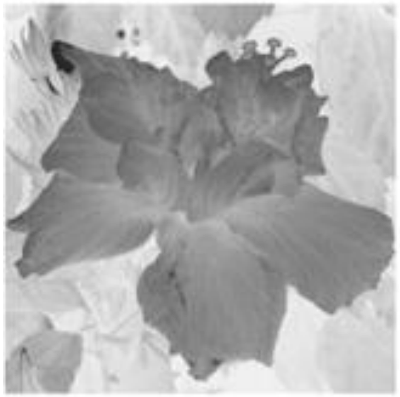

(b) Lighten

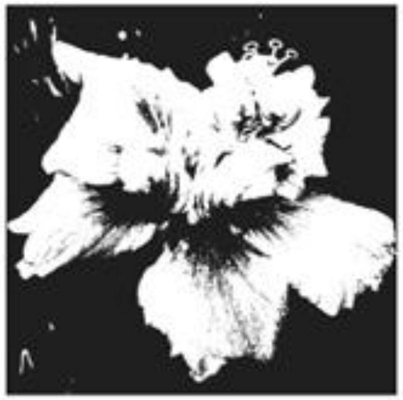

(c) Darken

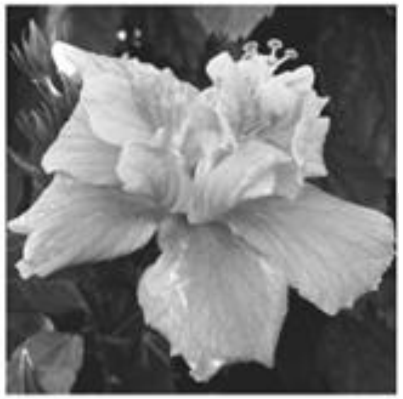


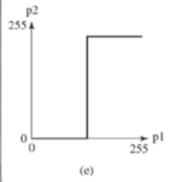


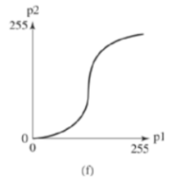



(d) Invert


(e) Threshold


(f) Increase contrast



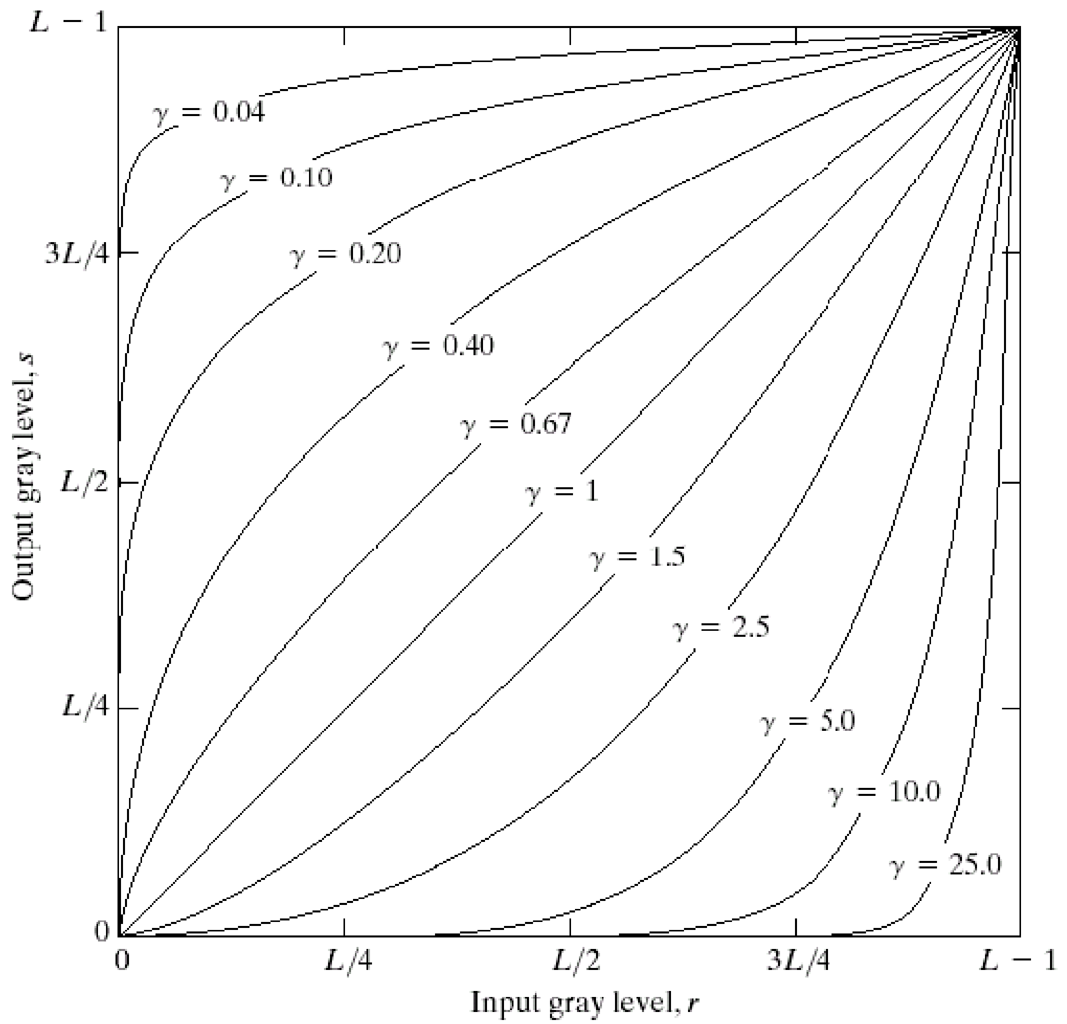


Gamma transform - 1

- The ***gamma value*** γ is an exponent that defines a nonlinear relationship between the input level *r* and the output level *s* for a pixel transform function.

$$s = r^{\gamma},\ 0 \le s \le 1$$

Gamma transform - 2



Gamma transform - 3

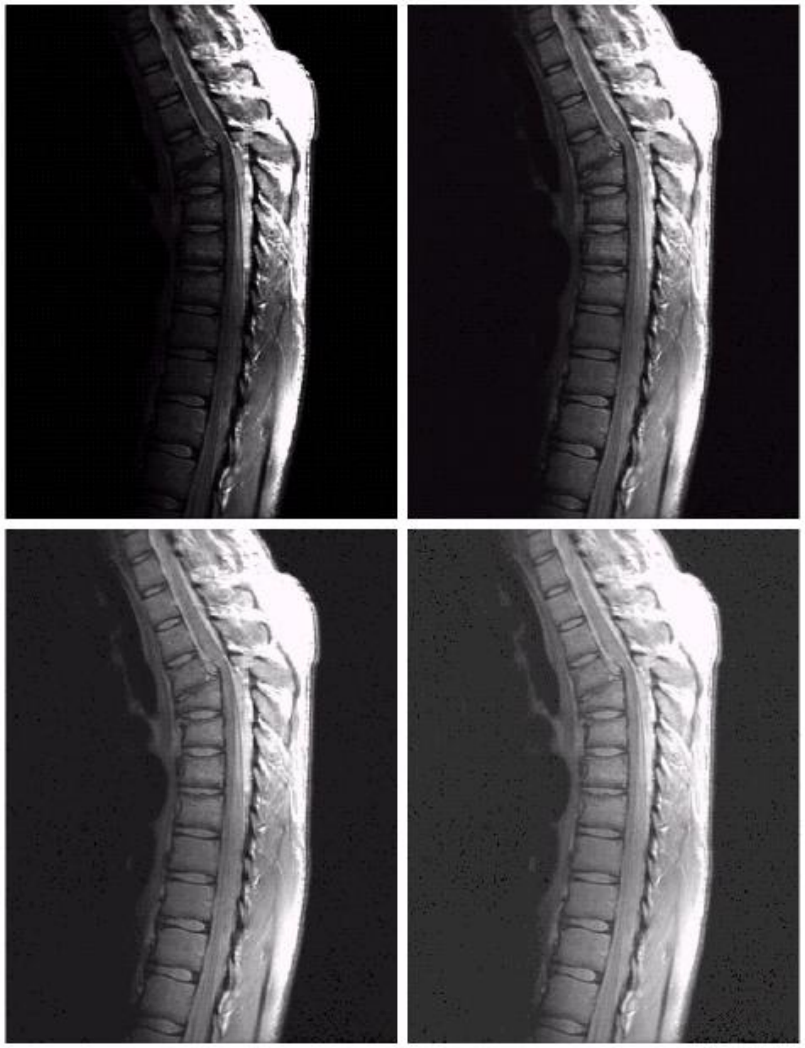
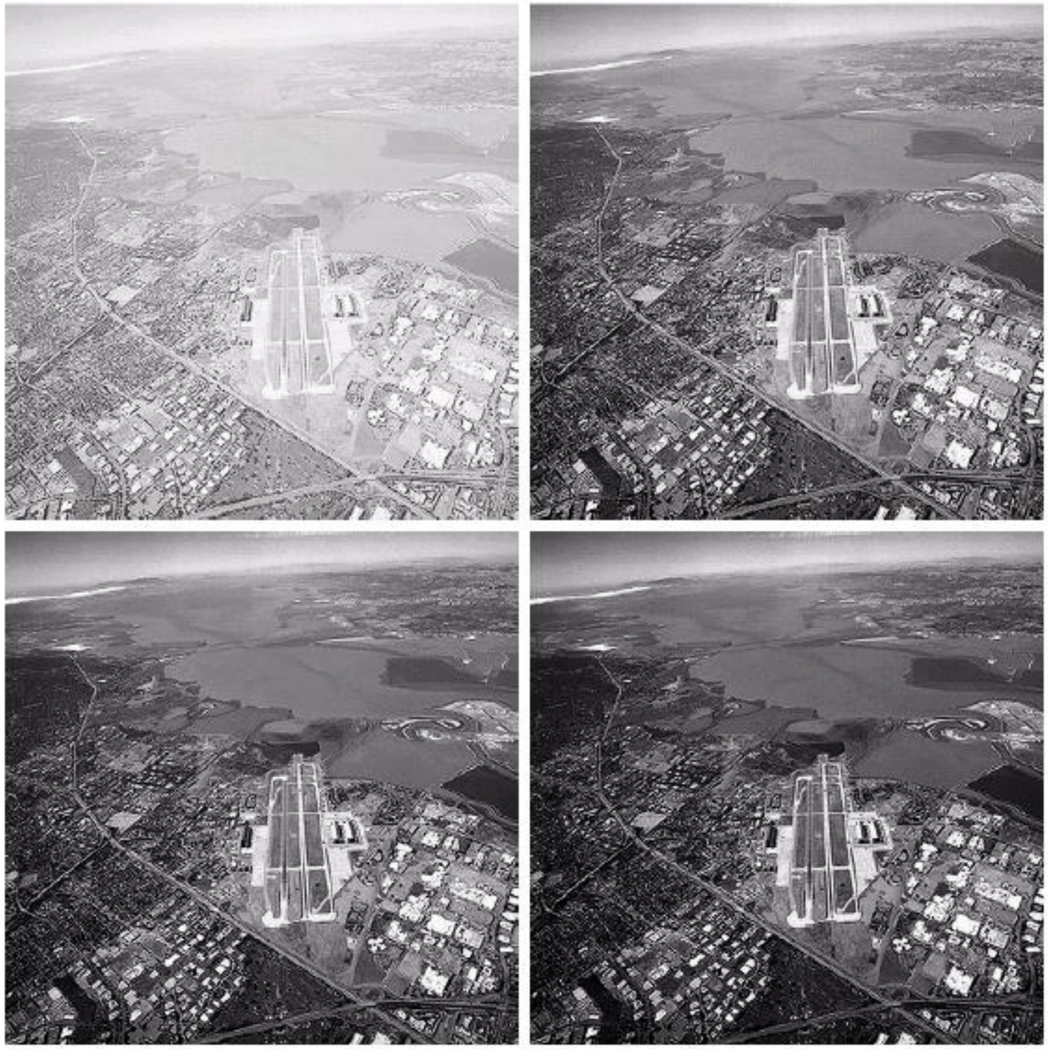

a b
c d

FIGURE 3.8 (a) Magnetic resonance (MR) image of a fractured human spine. (b)–(d) Results of applying the transformation in Eq. (3.2-3) with $c = 1$ and $\gamma = 0.6, 0.4$, and 0.3, respectively. (Original image for this example courtesy of Dr. David R. Pickens, Department of Radiology and Radiological Sciences, Vanderbilt University Medical Center.)

Gamma transform - 4

a	b
c	d

FIGURE 3.9 (a) Aerial image. (b)–(d) Results of applying the transformation in Eq. (3.2-3) with $c = 1$ and $\gamma = 3.0, 4.0,$ and 5.0, respectively. (Original image for this example courtesy of NASA.)

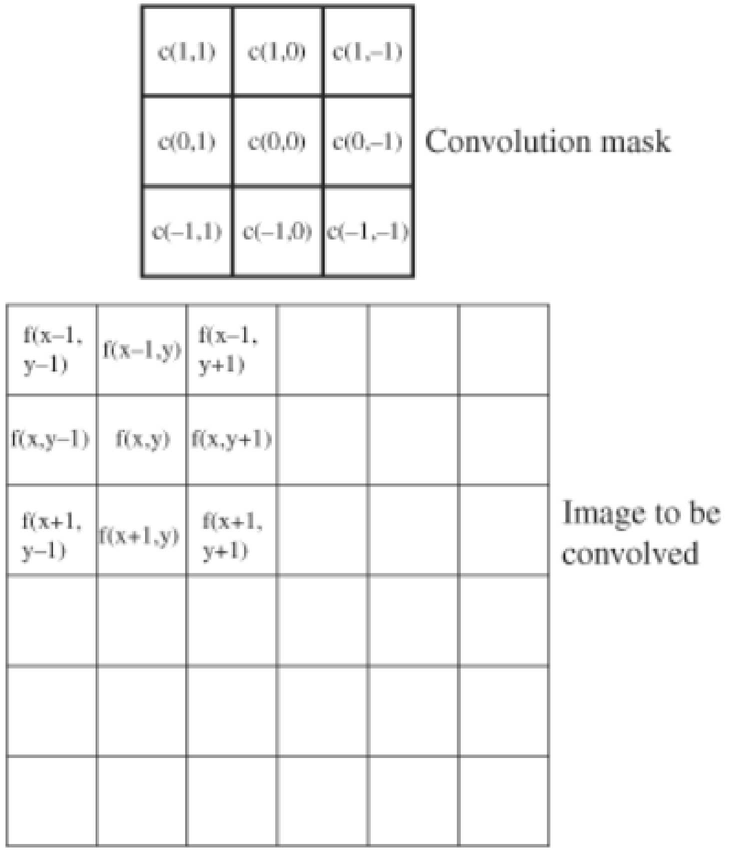


Filter

- A ***filter*** is an operation performed on digital image data to sharpen, smooth, or enhance some feature, or in some other way modify the image
- ***Filtering in the frequency domain*** is performed on image data that is represented in terms of its frequency components
- ***Filtering in the spatial domain*** is performed on image data in the form of the pixel's color values

Convolution - 1

- Spatial filtering is done by a mathematical operation called ***convolution***, where each output pixel is computed as a weighted sum of neighboring input pixels
- Convolution is based on a matrix of coefficients called a ***convolution mask***. The mask is also sometimes called a ***filter***.



1. Apply convolution mask to upper left corner of image.
2. Move mask to the right one pixel and apply again.
3. Continue applying mask to all pixels, moving left to right and top to bottom across image.

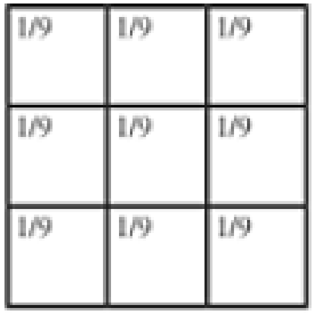
Convolution - 2

- Let $f(x, y)$ be an $M \times N$ image and $c(v, w)$ be an $m \times n$ mask. Then the equation for a linear convolution is

$$\mathrm{f}(x,y) = \sum_{v=-i}^{i} \sum_{w=-j}^{j} c(v,w) f(x-v, y-w)$$

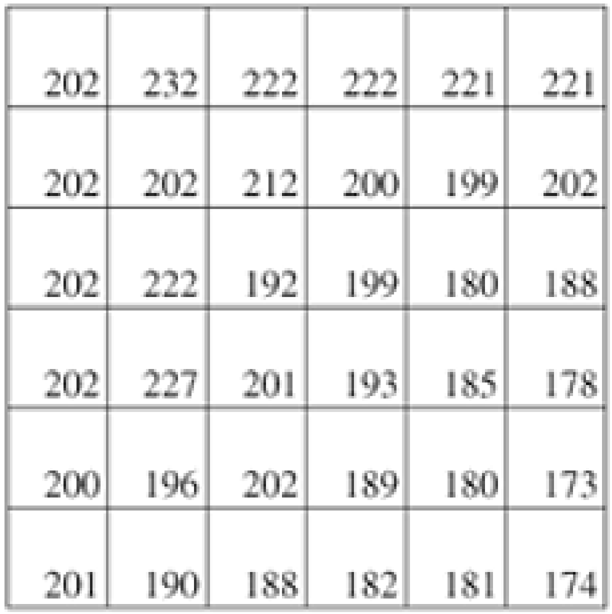
where $i = (m - 1)/2$ and $j = (n - 1)/2$. Assume m and n are odd. This equation is applied to each pixel $f(x, y)$ of an image, for $0 \le x \le M - 1$ and $0 \le y \le N - 1$. (If $x - v < 0$, $x - v \ge M$, $y - w < 0$, or $y - w \ge N$, then $f(x, y)$ is undefined. These are edge cases, discussed below.)

Convolution for averaging pixels



1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

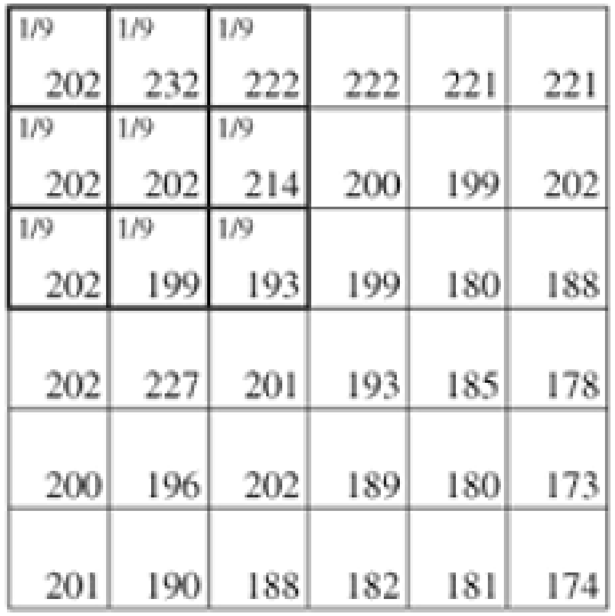
Convolution mask



202	232	222	222	221	221
202	202	212	200	199	202
202	222	192	199	180	188
202	227	201	193	185	178
200	196	202	189	180	173
201	190	188	182	181	174

Image to be convolved

Move the convolution mask over an area of pixels in the original image.

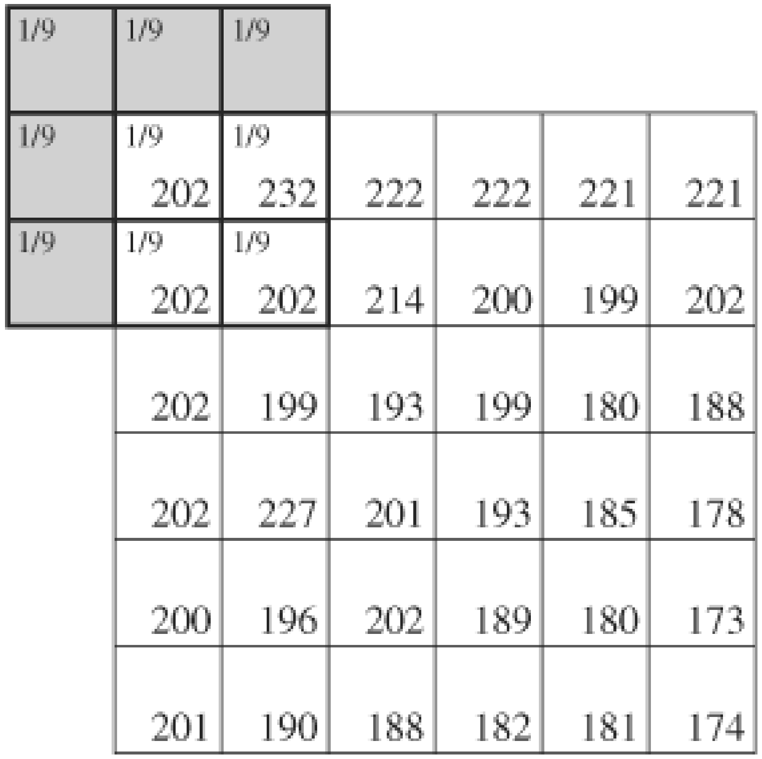


1/9 202	1/9 232	1/9 222	222	221	221
1/9 202	1/9 202	1/9 214	200	199	202
1/9 202	1/9 199	1/9 193	199	180	188
202	227	201	193	185	178
200	196	202	189	180	173
201	190	188	182	181	174

This mask will compute an average of the pixels in the neighborhood. The value of the center pixel will become

1/9*202+1/9*232+1/9*222+1/9*202+1/9*202+
1/9*214+1/9*202+1/9*199+1/9*193

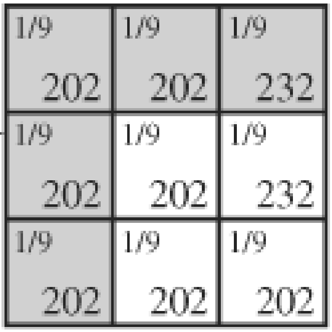
Handling edges in convolution



1/9	1/9	1/9				
1/9	1/9 202	1/9 232	222	222	221	221
1/9	1/9 202	1/9 202	214	200	199	202
	202	199	193	199	180	188
	202	227	201	193	185	178
	200	196	202	189	180	173
	201	190	188	182	181	174

Four ways to convolve pixels at the edge of an image:

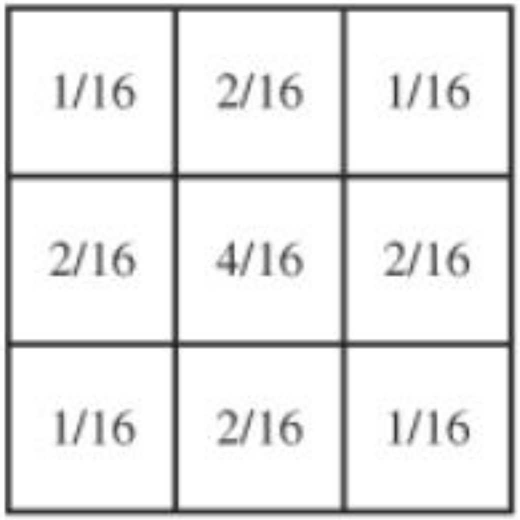
1. Assume that there are zero-valued pixels around the edges. These would be under the portion of the mask shaded in gray;
   or
2. Replicate the values from the edges, as shown;
   or
3. Use only the portion of the mask covering the image and change the weights appropriately for that step;
   or
4. Don't do convolution on the pixels at the edges.



1/9 202	1/9 202	1/9 232
1/9 202	1/9 202	1/9 232
1/9 202	1/9 202	1/9 202

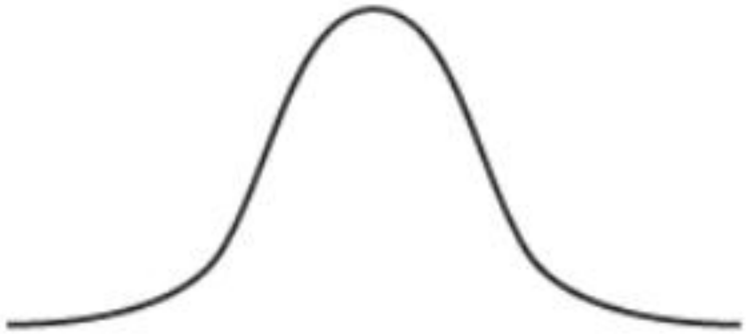
Gaussian blur

- An alternative for smoothing is to use a ***Gaussian blur***, where the coefficients in the convolution mask get smaller as you move away from the center of the mask



1/16	2/16	1/16
2/16	4/16	2/16
1/16	2/16	1/16

Gaussian convolution mask



Gaussian bell curve

An edge-detection filter

- The filter detects the edge, making that edge white while everything else is black

1	1	1
0	0	0
–1	–1	–1

Mask

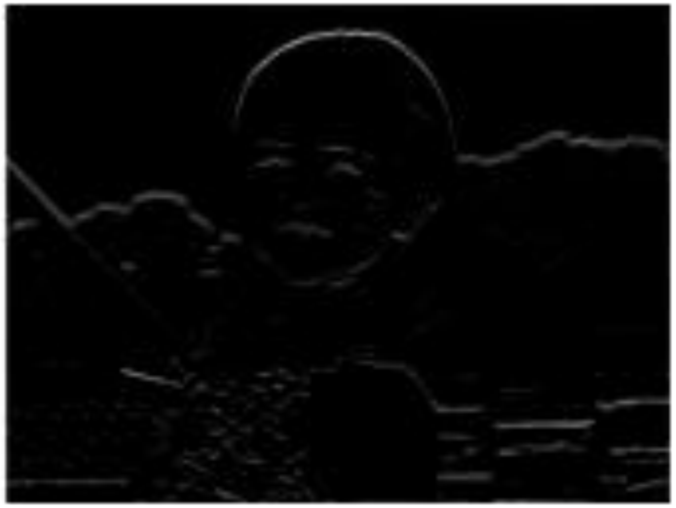
255	255	255	255
255	255	255	255
255	255	255	255
0	0	0	0
0	0	0	0
0	0	0	0

Block **a**

0	0	0	0
0	0	0	0
255	255	255	255
255	255	255	255
0	0	0	0
0	0	0	0

Block **b**

The mask above applied to block **a** of pixels yields block **b**.



Unsharp mask

- The name is misleading because this filter actually sharpens images
- The pixel values in the original image are doubled, and the blurred version of the image is subtracted from this

$$\begin{array}{|c|c|c|}\hline & & \\ \hline & 2 & \\ \hline & & \\ \hline\end{array} - \begin{array}{|c|c|c|}\hline & 1 & \\ \hline 1 & -3 & 1 \\ \hline & 1 & \\ \hline\end{array} = \begin{array}{|c|c|c|}\hline & -1 & \\ \hline -1 & 5 & -1 \\ \hline & -1 & \\ \hline\end{array}$$

2*original − Blur mask = Unsharp mask

Original image

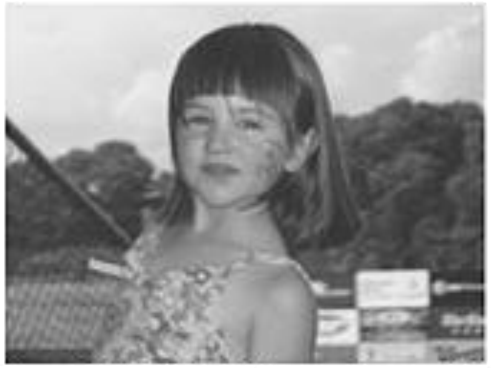


Image with blur filter applied

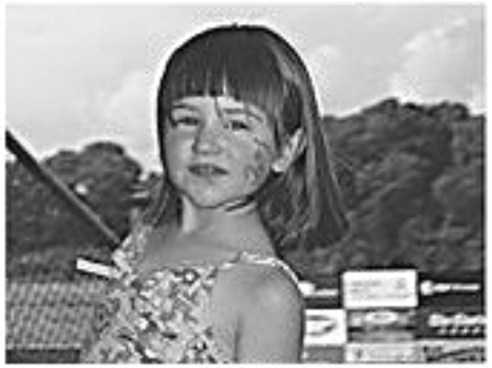


Image with unsharp mask applied

Resampling - 1

- ***Resampling*** is a process of changing the total number of pixels in a digital image
- To understand these, you need to understand the relationship between resolution, the size of an image in pixels, and print size
- Here are four scenarios where you might want to change the print size, resolution, or total pixel dimensions of a digital image. See if you can tell which require resampling:

Scenarios - 1

1) You scanned in an 8×10 inch photograph at a high resolution (300 pixels per inch, abbreviated *ppi*). You realize that you don’t need this resolution since your printer can’t print in that much detail anyway. You decide to decrease the resolution, but you don’t want to change the size of the photograph when it is printed out. If you want your image processing program to change the image size from 8×10 inches and 300 ppi to 8×10 inches and 200 ppi, does the image have to be resampled?

Scenarios - 2

2) You scanned in a 4×5 inch image at a resolution of 72 ppi, and it has been imported into your image processing program with these dimensions. You're going to display it on a computer monitor that has 90 ppi, and you don't want the image to be any smaller than 4×5 on the display. Does the image have to be resampled?

Scenarios - 3

3) You scanned in an 8×10 inch photograph at 200 ppi, and it has been imported into your image processing program with these dimensions. You want to print it out at a size of 4×5 inches. Does the image have to be resampled?

Scenarios - 4

4) You click on an image on your computer display to zoom in closer. Does the image have to be resampled?

Resampling - 2

- The key point to understand is that resampling is required whenever the number of pixels in a digital image is changed
- If the resolution—the ppi—is not changed but the print size *is*, resampling is necessary
- Resampling is also required if the print size is not changed but the resolution is

Scenarios - 1

The image has to be resampled in this case. If you have 300 ppi and an image that is 8 $\times$ 10 inches, you have a total of 300 * 8 * 10 = 24,000 pixels. You want 200 * 8 * 10 = 16,000 pixels. Some pixels have to be discarded, which is called ***downsampling***

Scenarios - 2

The image has to be resampled. The 4 $\times$ 5 image scanned at 72 ppi has pixel dimensions of 288 $\times$ 360. A computer display that can fit 90 pixels in every inch (in both the horizontal and vertical directions) will display this image at a size of 3.2 $\times$ 4 inches. Retaining a size of at least 4 $\times$ 5 inches on the computer display requires ***upsampling***, a process of inserting additional samples into an image.

Scenarios - 3

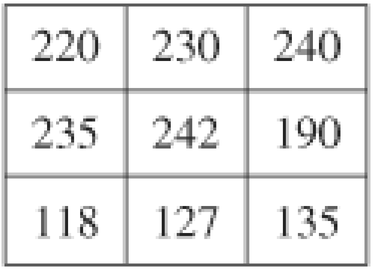
- The image doesn't have to be, although you can re-sample if you choose to.
- Without downsampling, ppi will be greater.
- With downsampling, ppi remains the same.

Scenarios - 4

- When you zoom in, the image is being upsampled, but only for display purposes. When you zoom out, the image is being downsampled.

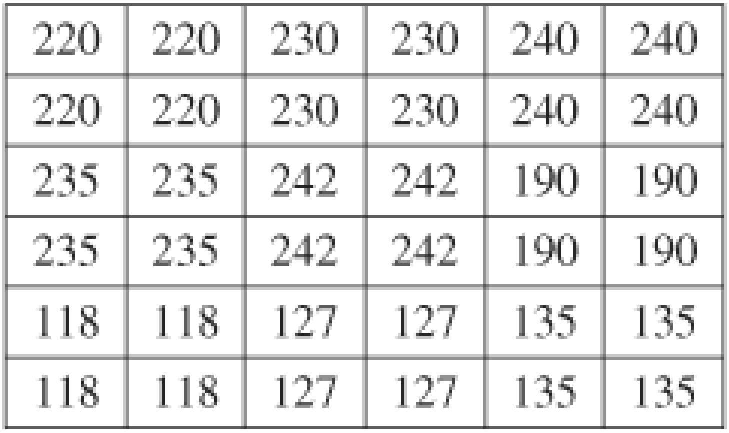
Replication - 1

- The simplest method for upsampling is ***replication***, a process of inserting pixels and giving them the color value of a neighboring pre-existing pixel
- Replication works only if you are enlarging an image by an integer factor



220	230	240
235	242	190
118	127	135

Original pixels



220	220	230	230	240	240
220	220	230	230	240	240
235	235	242	242	190	190
235	235	242	242	190	190
118	118	127	127	135	135
118	118	127	127	135	135

Pixels after replication

Replication - 2

- Since the new pixel values are copied from neighboring pixels, replication causes blockiness in the resampled image
- Of course there is no harm done to the file, since the pixel values are upsampled only for display purposes. The values stored in the image file don't change

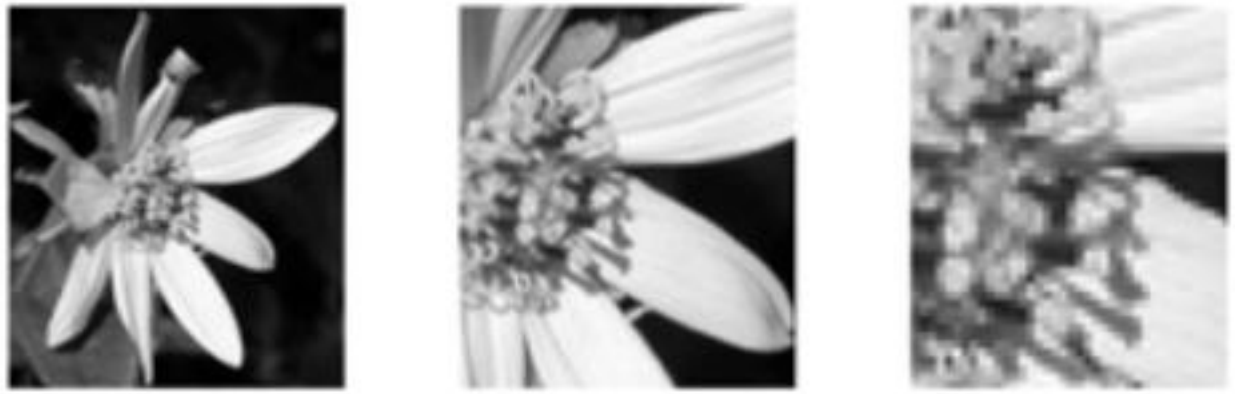


Image resampled as you zoom in

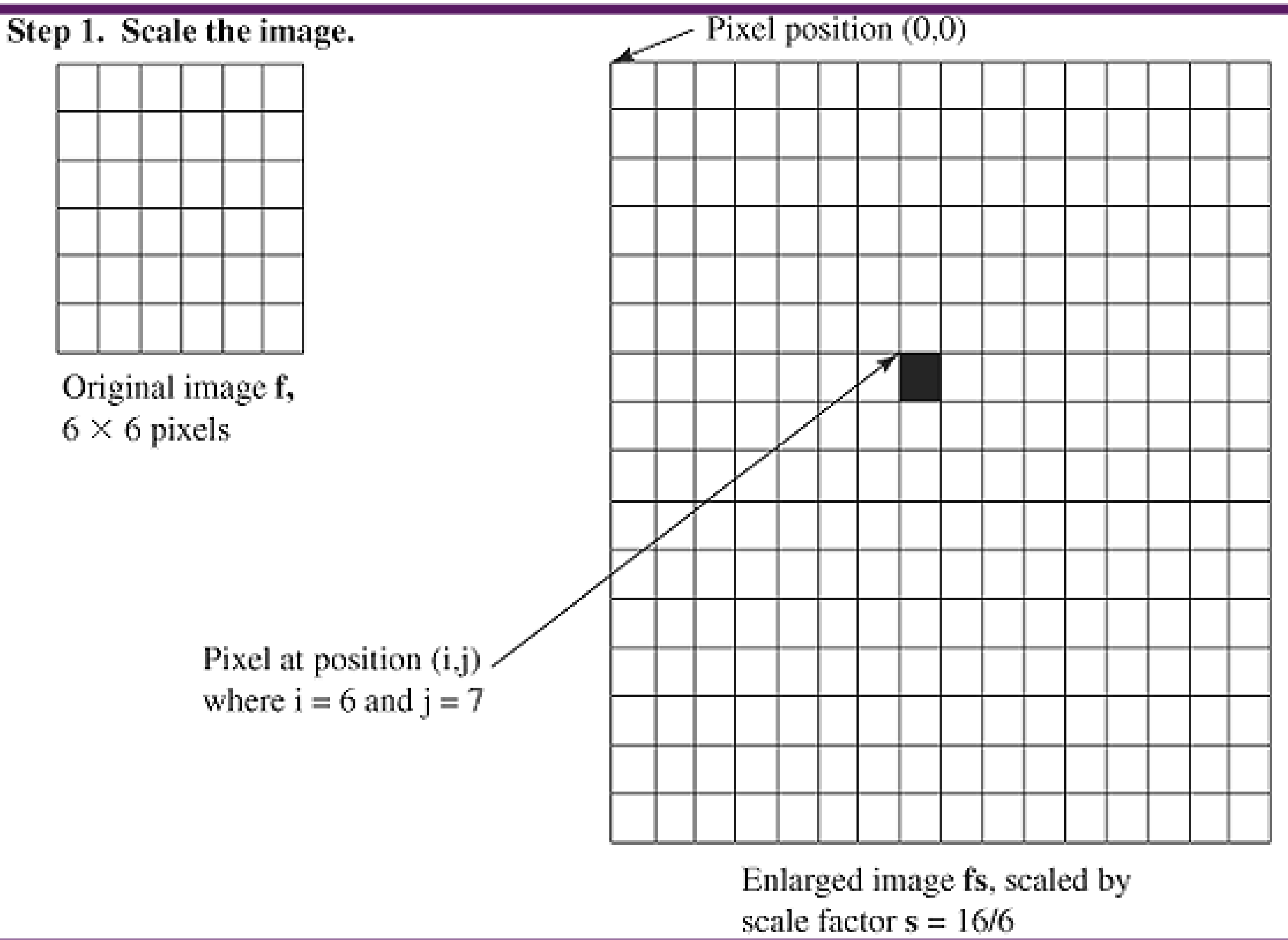
Row-column deletion

- The simplest method of downsampling is ***row-column deletion***, the inverse of replication
- Row-column deletion throws away information about the image, so you obviously lose detail

Interpolation

- There are interpolation methods for resampling that give better results than simple replication or discarding of pixels
- ***Interpolation*** is a process of estimating the color of a pixel based on the colors of neighboring pixels
  - Nearest neighbor
  - Bilinear
  - Bicubic

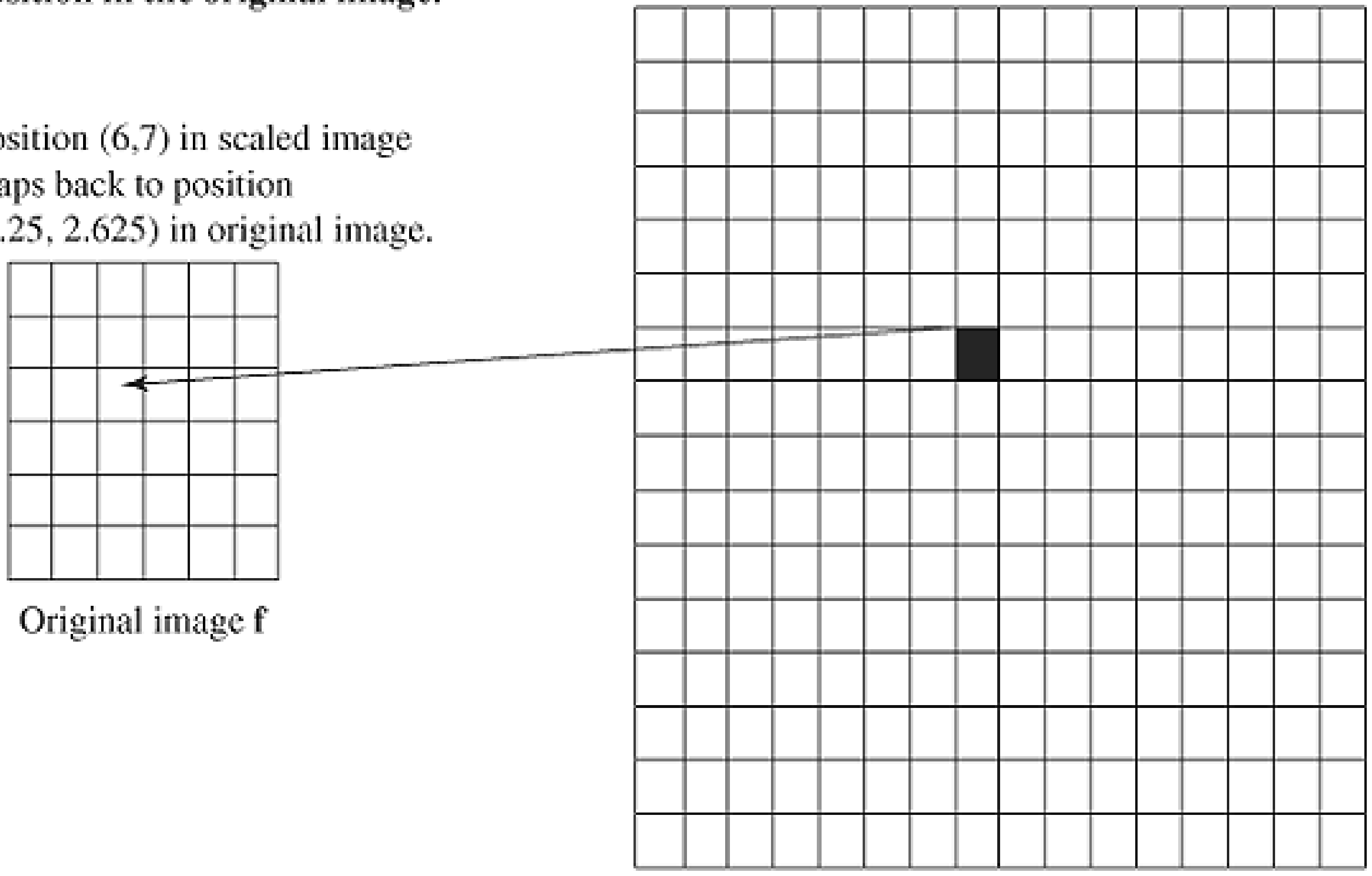
The first two steps in resampling - 1


Step 1. Scale the image.
Pixel position (0,0)
Original image f, 6 × 6 pixels
Pixel at position (i,j) where i = 6 and j = 7
Enlarged image fs, scaled by scale factor s = 16/6

The first two steps in resampling - 2

Step 2. Map each pixel in the scaled image back to a position in the original image.

Position (6,7) in scaled image maps back to position (2.25, 2.625) in original image.



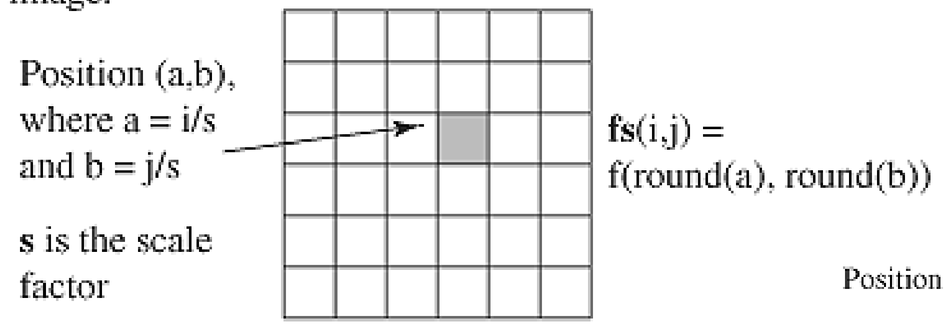
Original image f

Scaled image fs

Nearest neighbor interpolation

- ***Nearest neighbor interpolation*** simply rounds down to find one close pixel whose value is used for ***fs***(*i, j*)

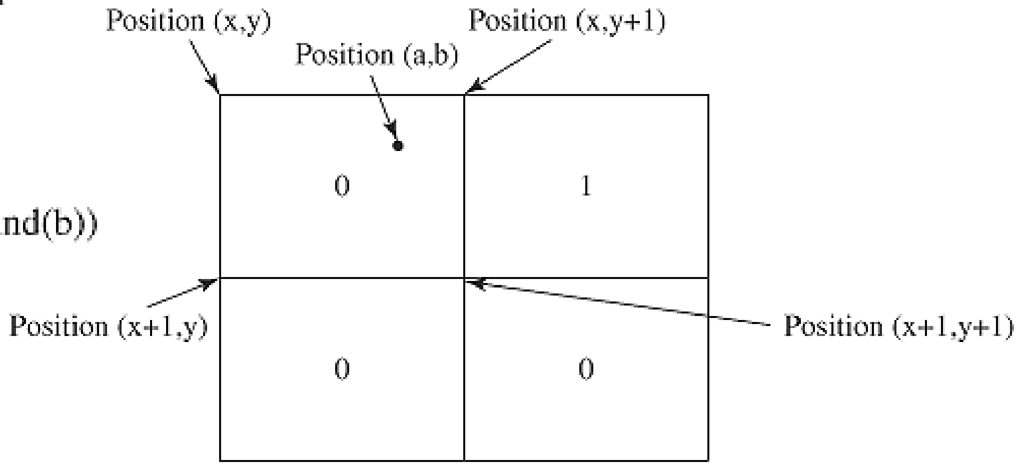
The **nearest neighbor algorithm** assigns to **fs**(i,j) the color value f(round(a), round(b)) from the original image.



Position (a,b), where a = i/s and b = j/s
s is the scale factor
fs(i,j) = f(round(a), round(b))

The nearest neighbor is marked in gray.

Nearest neighbor interpolation



Position with coordinates closest to both a and b gets a 1 in the convolution kernel.

Bilinear interpolation

- ***Bilinear interpolation*** uses four neighbors and makes ***fs***(*i, j*) a weighted sum of their color values. The contribution of each pixel toward the color of ***fs***(*i, j*) is a function of how close the pixel's coordinates are to (*a, b*).

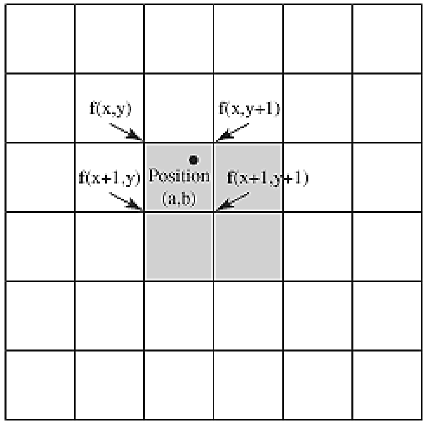
Bilinear interpolation uses an average color value of the four pixels surrounding position (a,b) in the original image. Each neighbor's contribution to the color is based on how close it is to (a,b). Let

x = floor(a)
y = floor(b)

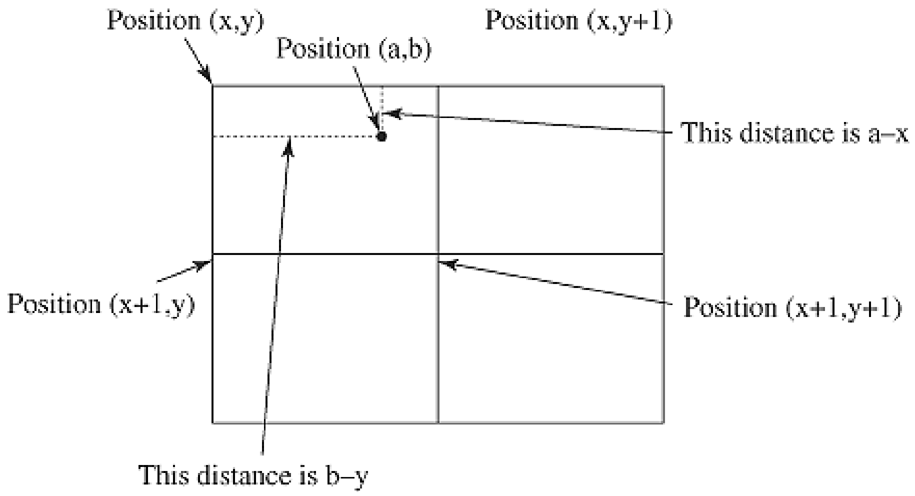
Then the pixels surrounding position (a,b) are

f(x, y)
f(x+1, y)
f(x+1, y+1)
f(x, y+1)

Neighborhood is shown in gray.



Bilinear interpolation



The color of the pixel in image **fs** is a weighted average of the four neighboring pixels. Weights come from each pixel's proximity to (a,b).

Bilinear interpolation

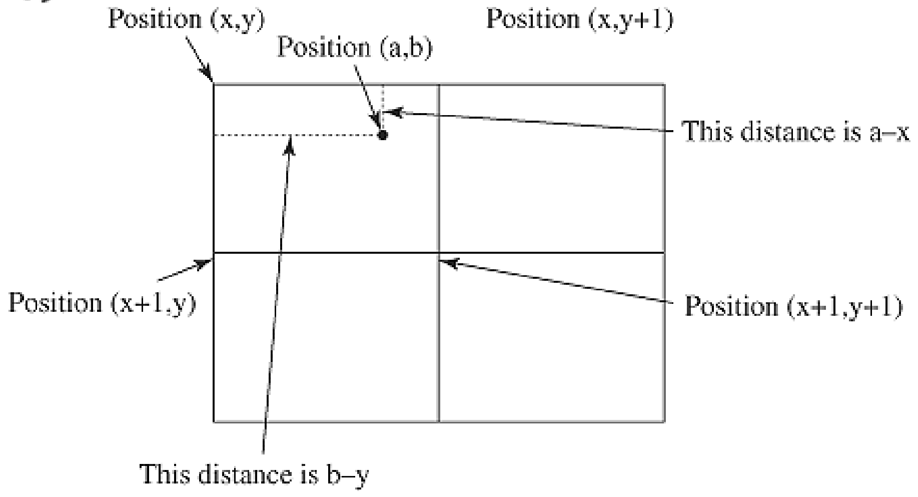
- 2 x 2 convolution mask H

$$t(m,n) = a - (x+m), 0 \leq m \leq 1, 0 \leq n \leq 1$$

$$u(m,n) = b - (y+n), 0 \leq m \leq 1, 0 \leq n \leq 1$$

$$H(m,n) = (1-|t|)(1-|u|)$$

Bilinear interpolation

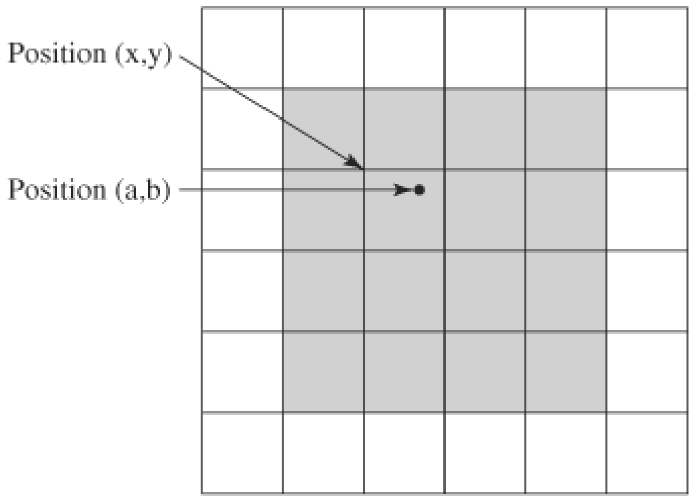


The color of the pixel in image **fs** is a weighted average of the four neighboring pixels. Weights come from each pixel's proximity to (a,b).

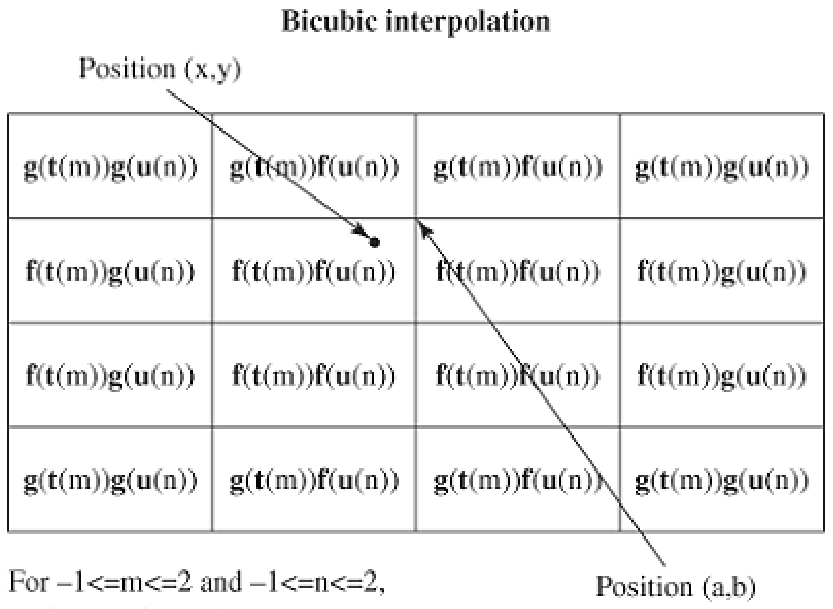
Bicubic interpolation

- ***Bicubic interpolation*** uses a neighborhood of sixteen pixels to determine the value of ***fs***(*i, j*)

Bicubic interpolation uses an "average" color value of the 16 pixels surrounding position (a,b) in the original image. The weight of each neighbor's contribution is based on a cubic equation that accounts for how close each neighbor is.



Neighbors are shaded in gray. The neighborhood of (a,b) extends from x–1 to x+2 in the vertical direction and from y–1 to y+2 in the horizontal direction.



g(**t**(m))g(**u**(n))	g(**t**(m))f(**u**(n))	g(**t**(m))f(**u**(n))	g(**t**(m))g(**u**(n))
f(**t**(m))g(**u**(n))	f(**t**(m))f(**u**(n))	f(**t**(m))f(**u**(n))	f(**t**(m))g(**u**(n))
f(**t**(m))g(**u**(n))	f(**t**(m))f(**u**(n))	f(**t**(m))f(**u**(n))	f(**t**(m))g(**u**(n))
g(**t**(m))g(**u**(n))	g(**t**(m))f(**u**(n))	g(**t**(m))f(**u**(n))	g(**t**(m))g(**u**(n))

For $-1 <= m <= 2$ and $-1 <= n <= 2$,
$\mathbf{t}(m) = a - (x + m)$
$\mathbf{u}(n) = b - (y + n)$

4x4 convolution mask

$f(\mathbf{t}(m)) = 1 - 2\mathbf{t}(m)^2 + |\mathbf{t}(m)|^3$
$f(\mathbf{u}(n)) = 1 - 2\mathbf{u}(n)^2 + |\mathbf{u}(n)|^3$
$g(\mathbf{t}(m)) = 4 - 8|\mathbf{t}(m)| + 5\mathbf{t}(m)^2 - |\mathbf{t}(m)|^3$
$g(\mathbf{u}(n)) = 4 - 8|\mathbf{u}(n)| + 5\mathbf{u}(n)^2 - |\mathbf{u}(n)|^3$

LZW compression

- ***LZW compression*** is a method that is applicable to both text and image compression. It is a lossless compression.
- The method is commonly applied to GIF and TIFF image files
- The LZW algorithm is based on the observation that sequences of color in an image file (or sequences of characters in a text file) are often repeated

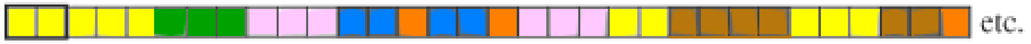
LZW compression - example

- The code table is initialized to contain all the individual colors existing in the image.
- If the pixel sequence is already in the code table, the window is successively expanded by one pixel until finally a color sequence not in the table is under the window.
- The full code table does not have to be stored with the compressed file; only the original colors in the image is needed.

Initial color table: **Space for more codes:**

Code	0	1	2	3	4	5	6											etc.
Color	🟦	🟩	🟫	🟧	🟪	🟨	⬛											

Step 1:

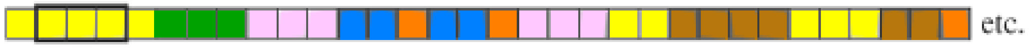


Window is over first sequence of colors not already in table. Output code for one yellow pixel, and put the new sequence in the table.

Compressed file so far is **5**.

Step 2:

Code	0	1	2	3	4	5	6	7										etc.
Color	🟦	🟩	🟫	🟧	🟪	🟨	⬛	🟨🟨										

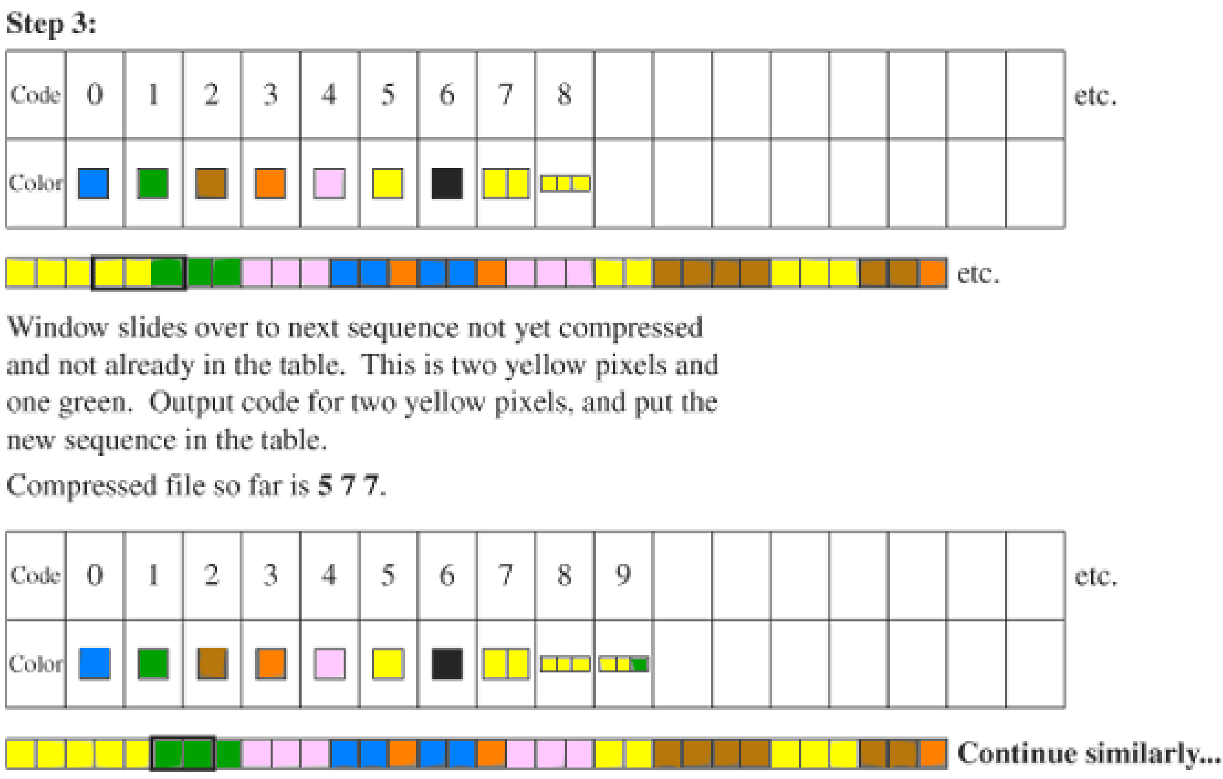


Window slides over to next sequence not yet compressed and not already in the table. This is three yellow pixels in a row. Output code for two yellow pixels, and put the new sequence in the table.

Compressed file so far is **5 7**.

LZW compression - example

- The code table is initialized to contain all the individual colors existing in the image.
- If the pixel sequence is already in the code table, the window is successively expanded by one pixel until finally a color sequence not in the table is under the window.
- The full code table does not have to be stored with the compressed file; only the original colors in the image is needed.


Step 3:

Code	0	1	2	3	4	5	6	7	8								
Color	🟦	🟩	🟫	🟧	🟪	🟨	⬛	🟨🟨	🟨🟨🟨								

Window slides over to next sequence not yet compressed and not already in the table. This is two yellow pixels and one green. Output code for two yellow pixels, and put the new sequence in the table.

Compressed file so far is **5 7 7**.

Code	0	1	2	3	4	5	6	7	8	9							
Color	🟦	🟩	🟫	🟧	🟪	🟨	⬛	🟨🟨	🟨🟨🟨	🟨🟨🟩							

LZW compression algorithm

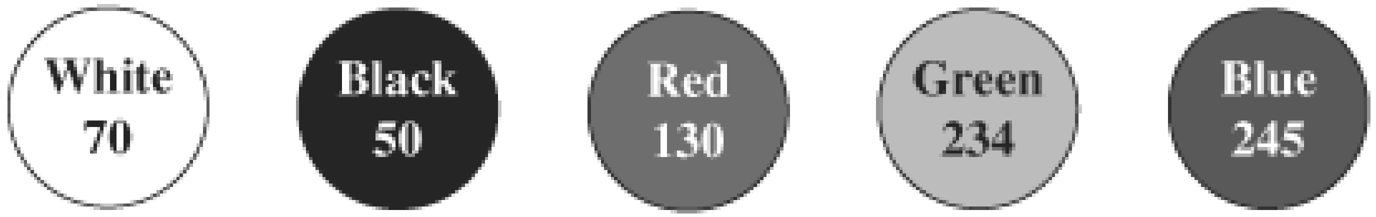
```
/*Input: A bitmap image.
Output: A table of the individual colors in the image
and a compressed version of the file.
Note that + is concatenation.*/
{
 initialize table to contain the individual colors in bitmap
 pixelString = first pixel value
 while there are still pixels to process {
  pixel = next pixel value
  stringSoFar = pixelString + pixel
  if stringSoFar is in the table then
     pixelString = stringSoFar
  else {
     output the code for pixelString
     add stringSoFar to the table
     pixelString = pixel
  }
 }
 output the code for pixelString
}
```

Huffman encoding

- ***Huffman encoding*** is another lossless compression algorithm that is used on bitmap image files.
- It differs from LZW in that it is a ***variable-length encoding*** scheme; that is, not all color codes use the same number of bits.
- The Huffman encoding algorithm requires two passes:
  1) determining the codes for the colors
  2) compressing the image file by replacing each color with its code

Huffman encoding - example

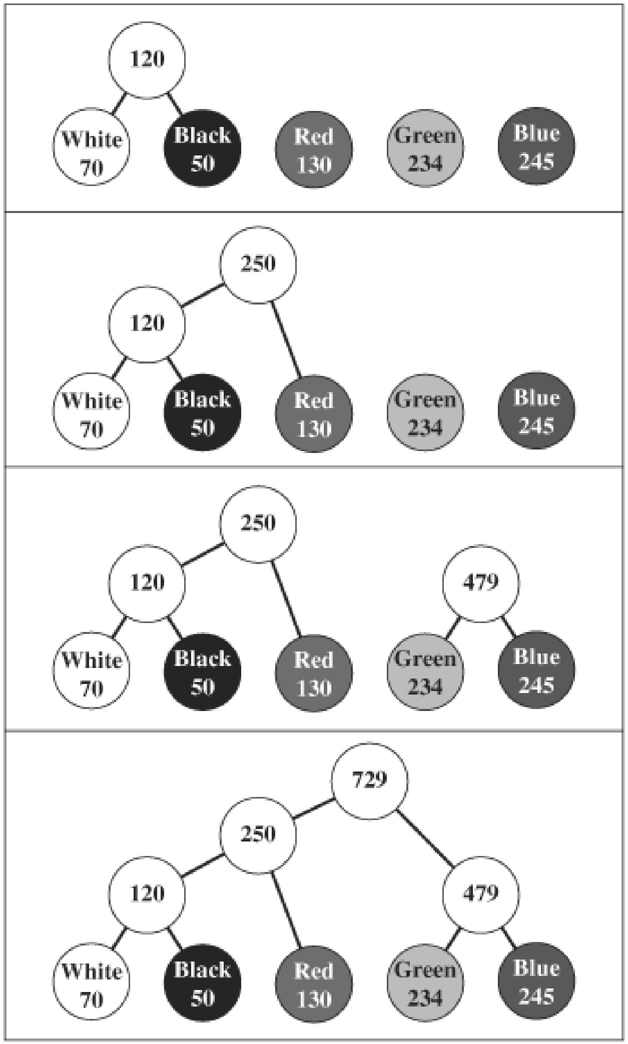
- The image file has only 729 pixels in it, with the following colors and the corresponding frequencies
  - White 70
  - Black 50
  - Red 130
  - Green 234
  - Blue 245



- A node is created for each of the colors in the image, with the frequency of that color's appearance stored in the node

Huffman encoding - example

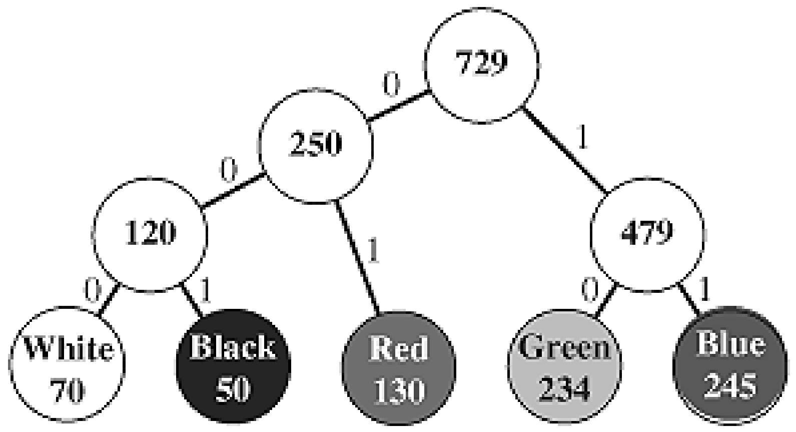
- Now the two nodes with the smallest value for *freq* are joined such that they are the children of a common parent node, and the parent node's *freq* value is set to the sum of the *freq* values in the children nodes
- This node-combining process repeats until you arrive at the creation of a root node



Huffman encoding - example

- Once the tree has been created, the branches are labeled with 0s on the left and 1s on the right

Label branches with 0s on the left and 1s on the right.



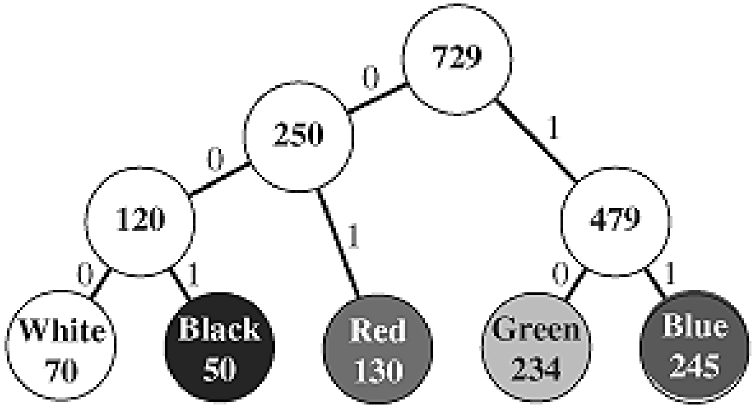
For each leaf node, traverse tree from root to leaf node and gather code for the color associated with the leaf node.

White	000
Black	001
Red	01
Green	10
Blue	11

Note that not all codes are the same number of bits, and no code is a prefix of any other code.

Huffman encoding - example

- After the codes have been created, the image file can be compressed using these codes
- Input: wwwkkwwbgr (k:black)
- Encoding: 000000000001001000000111001



White	000
Black	001
Red	01
Green	10
Blue	11

Huffman encoding

- By combining least-valued nodes first and creating the tree from the bottom up, the algorithm ensures that the colors that appear least frequently in the image have the longest codes
- Also, because the codes are created from the tree data structure, no code can be a prefix of another code
- Huffman encoding is useful as a step in JPEG compression, as we will see in the next section.

JPEG Compression

- JPEG is a lossy compression method
- Image processing programs allow you to choose the JPEG compression rate
- The main disadvantage to JPEG compression is that it takes longer for the encoding and decoding than other algorithms require

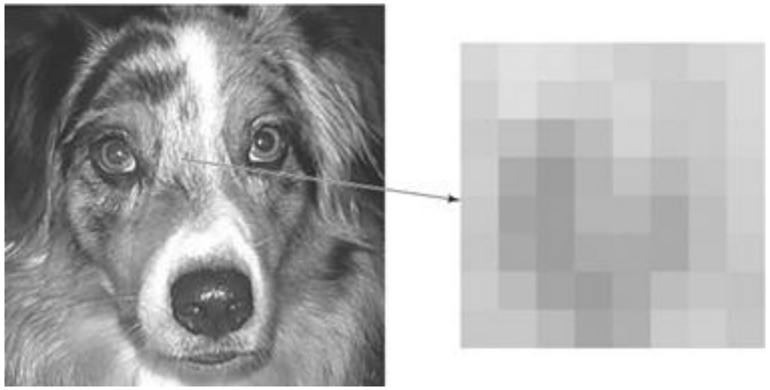
JPEG Compression – step1

- **Step1 : Divide the image into 8 × 8 pixel blocks and convert RGB to a luminance/chrominance color model**
- The image is divided into 8 × 8 pixel blocks to make it computationally more manageable for the next steps. Converting color to a luminance/chrominance model makes it possible to remove some of the chrominance information, to which the human eye is less sensitive, without significant loss of quality in the image.

JPEG Compression – step2

- **Step 2: Shift values by –128 and transform from the spatial to the frequency domain**
- On an intuitive level, shifting the values by –128 is like looking at the image function as a waveform that cycles through positive and negative values. This step is a preparation for representing the function in terms of its frequency components. Transforming from the spatial to the frequency domain makes it possible to remove high frequency components. High frequency components are present if color values go up and down quickly in a small space. These small changes are barely perceptible in most people's vision, so removing them does not compromise image quality significantly.

JPEG Compression – step2



Grayscale Values for 8 × 8 Pixel Area							
222	231	229	224	216	213	220	224
216	229	217	215	221	210	209	223
211	202	283	198	218	207	209	221
214	180	164	188	203	193	205	217
209	171	166	190	190	178	199	215
206	177	166	179	180	178	199	210
212	197	173	166	179	198	206	203
208	208	195	174	184	210	214	206

Pixel Values for Image in Figure 3.49 Shifted by –128							
94	103	101	96	88	85	92	96
88	101	89	87	93	82	81	95
83	74	55	70	90	79	81	93
86	52	36	60	75	65	77	89
81	43	38	62	62	50	71	87
78	49	38	51	52	50	71	82
84	69	45	38	51	70	78	75
80	80	67	46	56	82	86	78

JPEG Compression – step2

DCT of an 8 × 8 Pixel Area							
585.7500	–24.5397	59.5959	21.0853	25.7500	–2.2393	–8.9907	1.8239
78.1982	12.4534	–32.6034	–19.4953	10.7193	–10.5910	–5.1086	–0.5523
57.1373	24.829	–7.5355	–13.3367	–45.0612	–10.0027	4.9142	–2.4993
–11.8655	6.9798	3.8993	–14.4061	8.5967	12.9151	–0.3122	–0.1844
5.2500	–1.7212	–1.0824	–3.2106	1.2500	9.3595	2.6131	1.1199
–5.9658	–4.0865	7.6451	13.0616	–1.1927	1.1782	–1.0733	–0.5631
–1.2074	–5.7729	–2.0858	–1.9347	1.6173	2.6671	–0.4645	0.6144
0.6362	–1.4059	–0.719	1.6339	–0.1438	0.2755	–0.0268	–0.2255

JPEG Compression – step3

- **Step 3: Quantize the frequency values**
- Quantization involves dividing each frequency coefficient by an integer and rounding off. The coefficients for high-frequency components are typically small, so they often round down to 0—which means, in effect, that they are thrown away

JPEG Compression – step3

Quantization Table							
8	6	6	7	6	5	8	7
7	7	9	9	8	10	12	20
13	12	11	11	12	25	18	19
15	20	29	26	31	30	29	26
28	28	32	36	46	39	32	34
44	35	28	28	40	55	41	44
48	49	52	52	52	31	39	57
61	56	50	60	46	51	52	50

Quantized DCT Values							
73	–4	10	3	4	0	–1	0
11	2	–4	–2	1	–1	0	0
4	2	–1	–1	–4	0	0	0
–1	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

JPEG Compression – step4

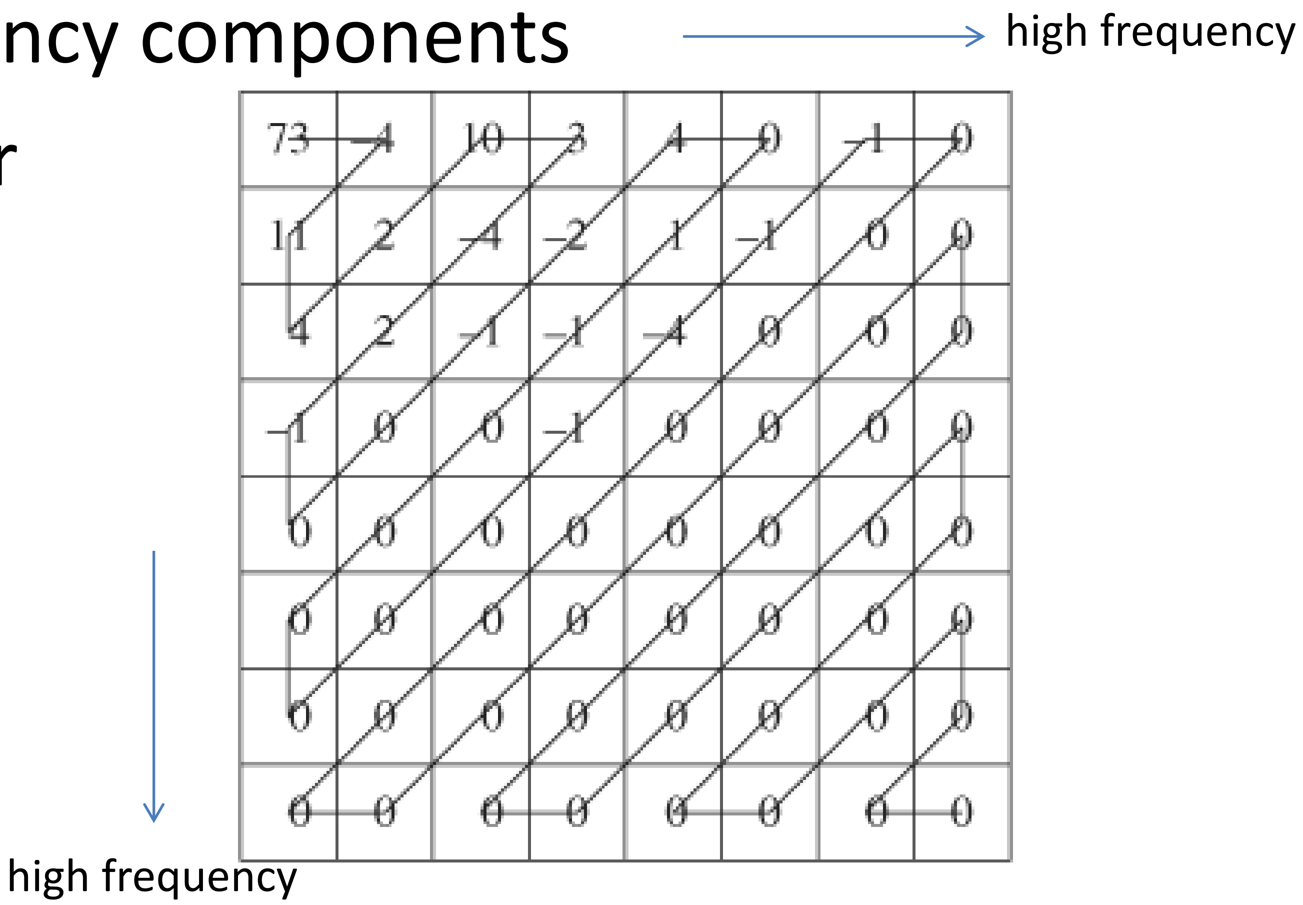
- **Step 4: Apply DPCM to the block**
- ***DPCM*** is the abbreviation for ***differential pulse code modulation***. In this context, DCPM is simply storing the difference between the first value in the previous 8×8 block and the first value in the current block. Since the difference is generally smaller than the actual value, this step adds to the compression.

JPEG Compression – step5

- **Step 5: Arrange the values in a zigzag order and do run-length encoding.**
- The zigzag reordering sorts the values from low-frequency to high-frequency components. The high-frequency coefficients are grouped together at the end. If many of them round to zero after quantization, run-length encoding is even more effective.

JPEG Compression – step5

- Quantized DCT values rearranged from low- to high-frequency components
- Zigzag order



73	–4	10	3	4	0	–1	0
11	2	–4	–2	1	–1	0	0
4	2	–1	–1	–4	0	0	0
–1	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

JPEG Compression – step6

- **Step 6: Do entropy encoding.**
- Additional compression can be achieved with some kind of entropy encoding.
- In JPEG, Huffman coding is used.

JPEG Compression Algorithm

```
algorithm jpeg
/*Input: A bitmap image in RGB mode.
Output: The same image, compressed.*/
{
  Divide image into 8 × 8 pixel blocks
  Convert image to a luminance/chrominance model such as YCbCr (optional)
  Shift pixel values by subtracting 128
  Use discrete cosine transform to transform the pixel data from the spatial domain
  to the frequency domain
  Quantize frequency values
  Store DC value (upper left corner) as the difference between current DC value and
  DC from previous block
  Arrange the block in a zigzag order
  Do run-length encoding
  Do entropy encoding (e.g., Huffman)
}
```

RGB → YC_bC_r

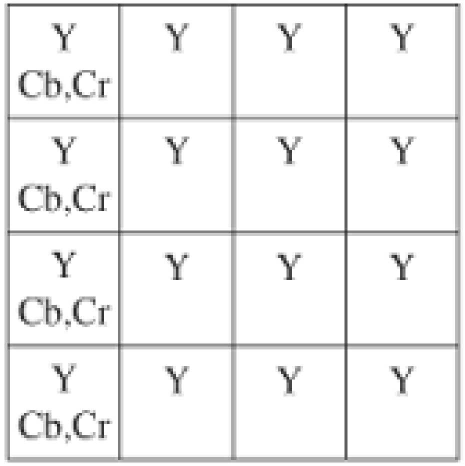
- YCbCr color model represents color in terms of one luminance component, Y, and two chrominance components, Cb and Cr.
- The human eye is more sensitive to changes in light (*i.e.*, luminance) than in color (*i.e.*, chrominance).

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} 0.257 & 0.504 & 0.098 \\ -0.148 & -0.291 & 0.439 \\ 0.439 & -0.368 & -0.071 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

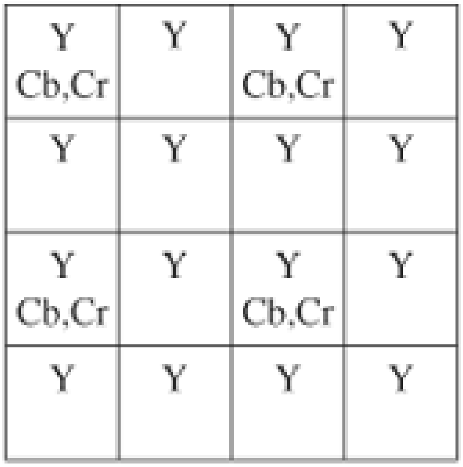
$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.164 & 0 & 1.596 \\ 1.164 & -0.392 & 0.439 \\ 1.164 & 2.017 & 0 \end{bmatrix} \left[\begin{bmatrix} Y-16 \\ C_b-128 \\ C_r-128 \end{bmatrix} \right]$$

Chrominance subsampling

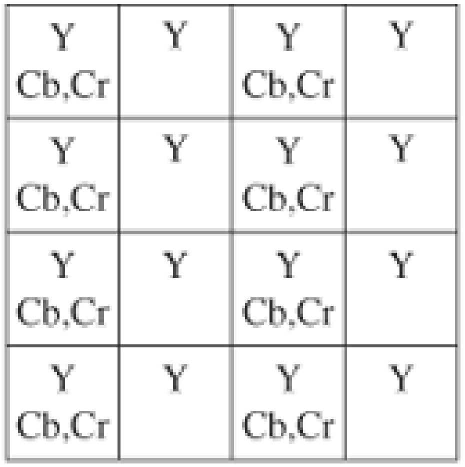
- luminance/chrominance subsampling is represented in the form *a*:*b*:*c*
- For each pair of four-pixel-wide rows, *a* is the number of Y samples in both rows, *b* and c are the numbers of C_b (C_r) samples in the 1st and 2nd rows, respectively.



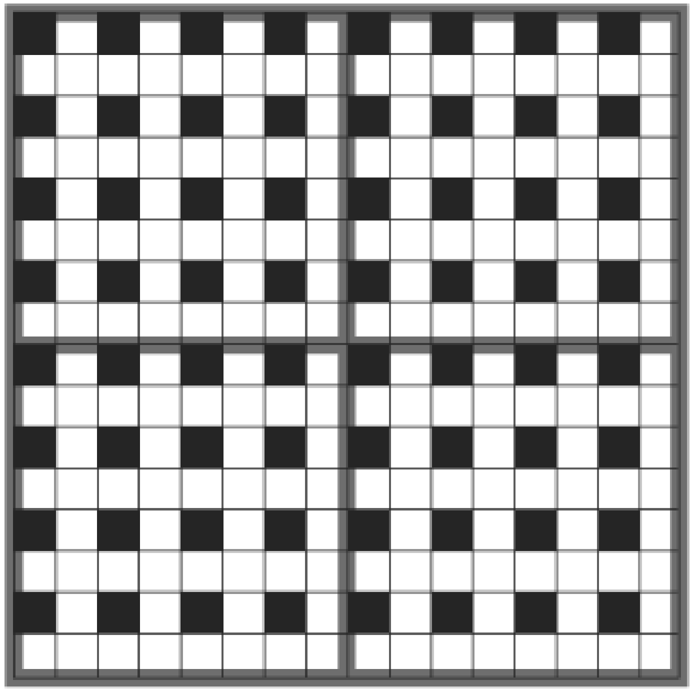
4:1:1



4:2:0

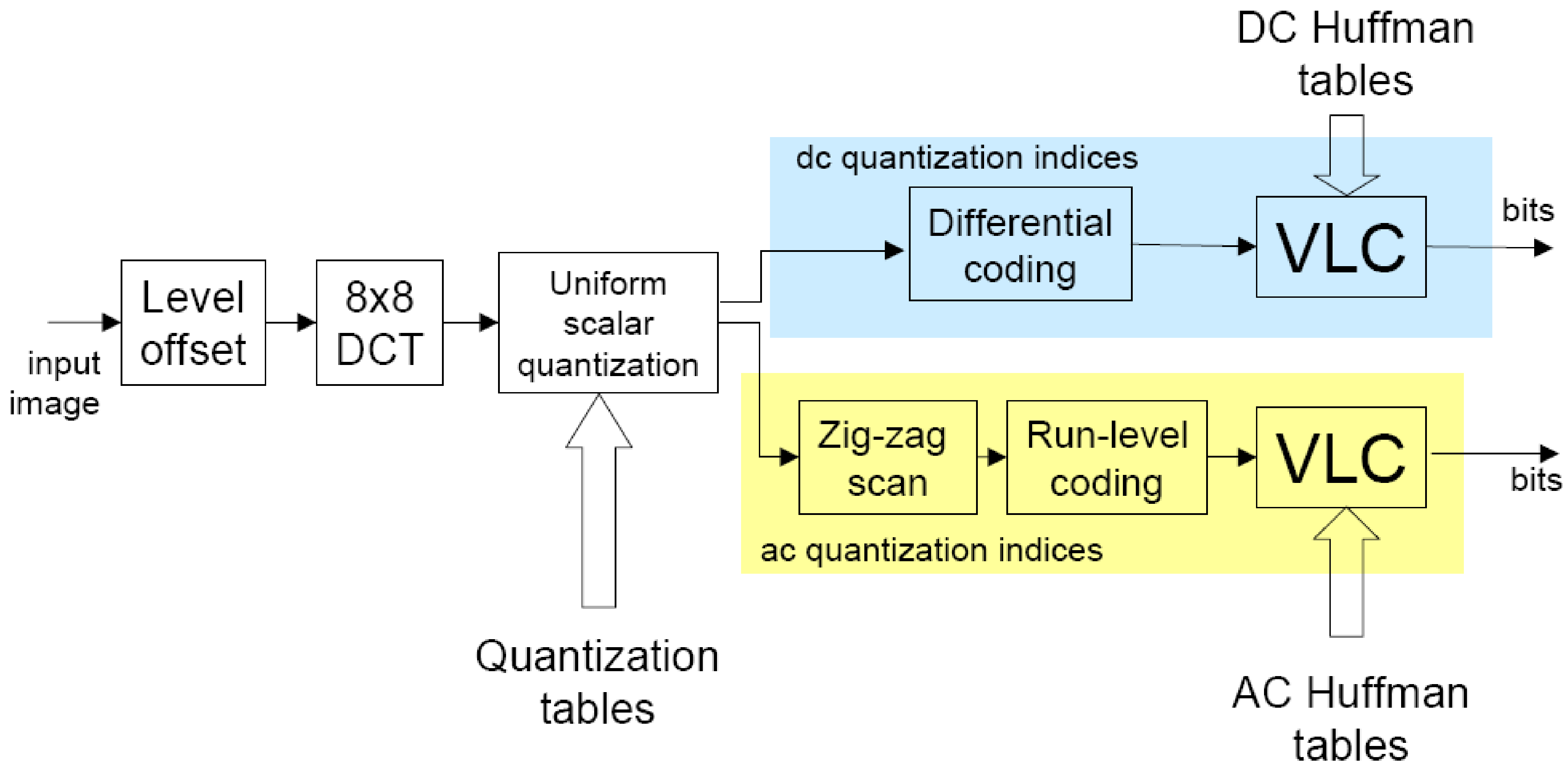

Y Cb,Cr | Y | Y Cb,Cr | Y
Y Cb,Cr | Y | Y Cb,Cr | Y
Y Cb,Cr | Y | Y Cb,Cr | Y
Y Cb,Cr | Y | Y Cb,Cr | Y
4:2:2

4:2:0 Chroma subsampling



With 4:2:0 YCbCr chrominance subsampling,
a 16 × 16 macroblock yields:
–four 8 × 8 blocks of Y values
–one 8 × 8 block of Cb values
–one 8 × 8 block of Cr values
Each black location represents one Cb
and one Cr sample taken for a block
of four pixels. Y samples are taken
at all locations.

JPEG Image Compression



8X8 DCT Bases

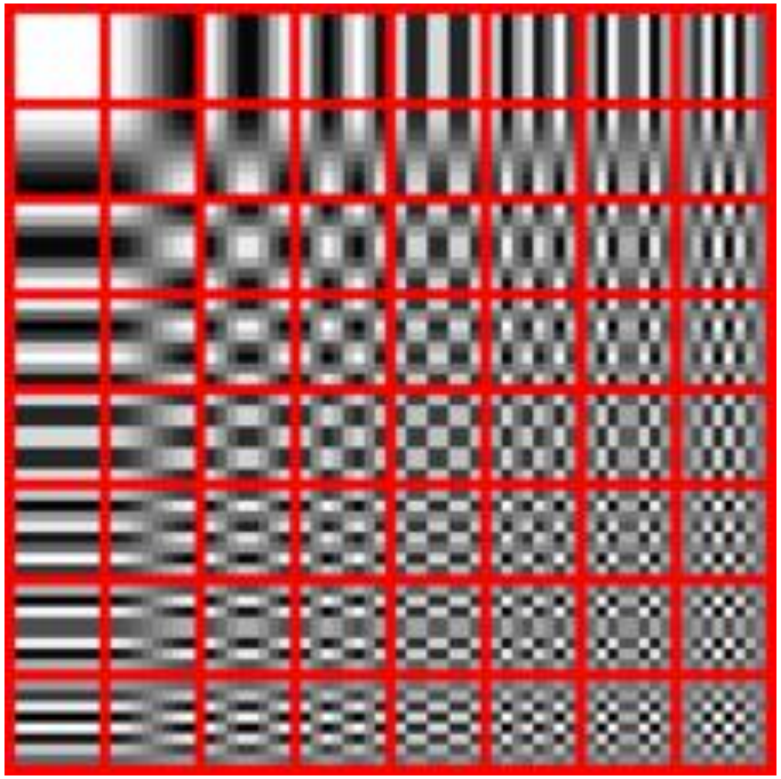
$$G_{u,v} = \sum_{x=0}^{7}\sum_{y=0}^{7} \alpha(u)\alpha(v) g_{x,y} \cos\left[\frac{\pi}{8}\left(x+\frac{1}{2}\right)u\right] \cos\left[\frac{\pi}{8}\left(y+\frac{1}{2}\right)v\right]$$

DCT coefficients

Image

$\rightarrow u$

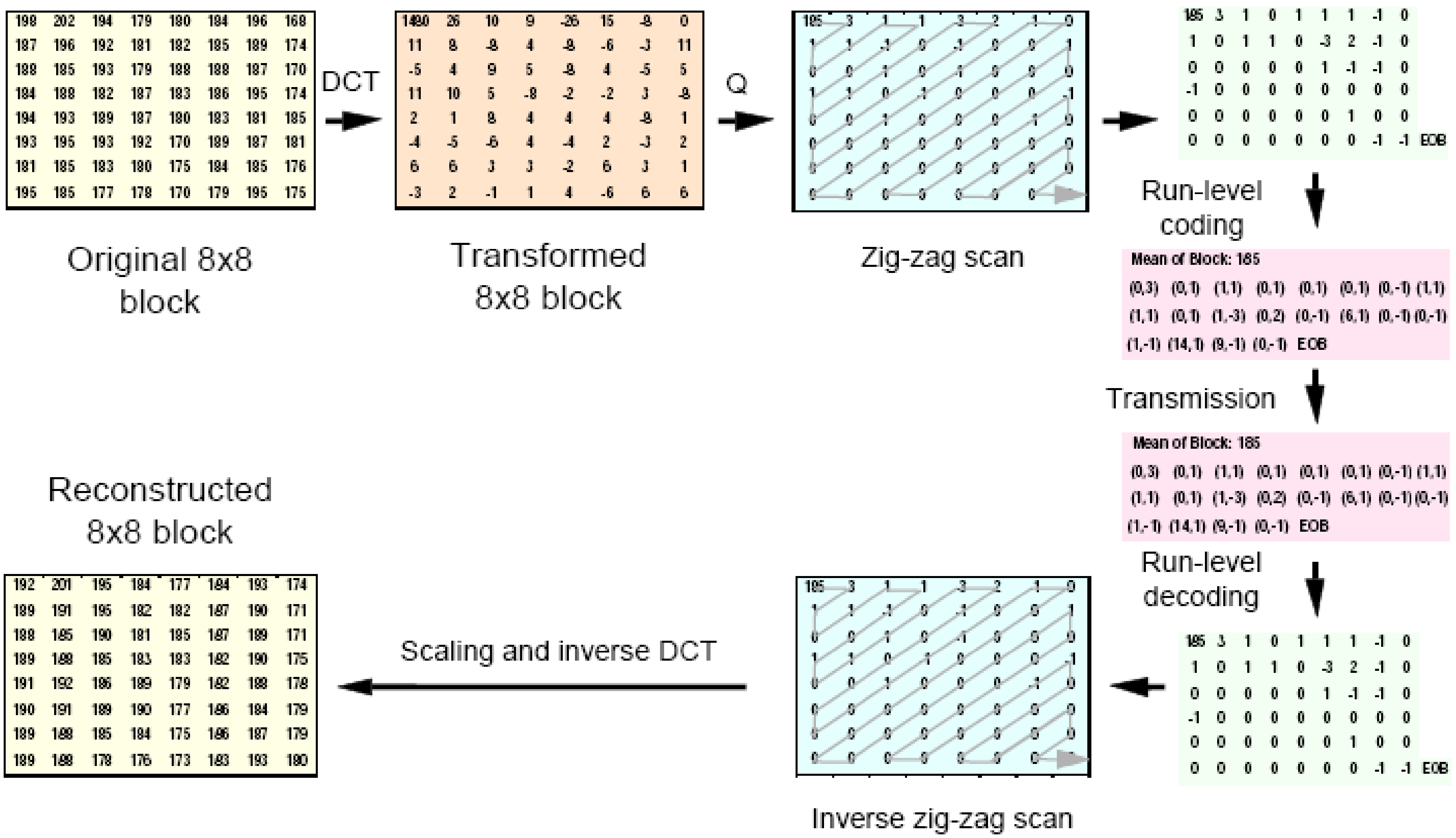
$\downarrow v$



Inverse DCT

$$f_{x,y} = \sum_{u=0}^{7}\sum_{v=0}^{7} \alpha(u)\alpha(v) F_{u,v} \cos\left[\frac{\pi}{8}\left(x+\frac{1}{2}\right)u\right] \cos\left[\frac{\pi}{8}\left(y+\frac{1}{2}\right)v\right]$$

Transform Coding



Original 8x8 block

198	202	194	179	180	184	196	168
187	196	192	181	182	185	189	174
188	185	193	179	188	188	187	170
184	188	182	187	183	186	195	174
194	193	189	187	180	183	181	185
193	195	193	192	170	189	187	181
181	185	183	180	175	184	185	176
195	185	177	178	170	179	195	175

Transformed 8x8 block

1480	26	10	9	-26	15	-8	0
11	8	-8	4	-8	-6	-3	11
-5	4	9	5	-8	4	-5	5
11	10	5	-8	-2	-2	3	-8
2	1	8	4	4	4	-8	1
-4	-5	-6	4	-4	2	-3	2
6	6	3	3	-2	6	3	1
-3	2	-1	1	4	-6	6	6

Zig-zag scan

185	3	1	1	-3	2	1	0
1	1	-1	0	-1	0	0	1
0	0	1	0	1	0	0	0
1	1	0	-1	0	0	0	-1
0	0	1	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

185	3	1	0	1	1	1	-1	0	
1	0	1	1	0	-3	2	-1	0	
0	0	0	0	0	1	-1	-1	0	
-1	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	1	0	0	
0	0	0	0	0	0	0	-1	-1	EOB

Mean of Block: 185

(0,3) (0,1) (1,1) (0,1) (0,1) (0,1) (0,-1) (1,1)
(1,1) (0,1) (1,-3) (0,2) (0,-1) (6,1) (0,-1) (0,-1)
(1,-1) (14,1) (9,-1) (0,-1) EOB

Mean of Block: 185

(0,3) (0,1) (1,1) (0,1) (0,1) (0,1) (0,-1) (1,1)
(1,1) (0,1) (1,-3) (0,2) (0,-1) (6,1) (0,-1) (0,-1)
(1,-1) (14,1) (9,-1) (0,-1) EOB

185	3	1	0	1	1	1	-1	0	
1	0	1	1	0	-3	2	-1	0	
0	0	0	0	0	1	-1	-1	0	
-1	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	1	0	0	
0	0	0	0	0	0	0	-1	-1	EOB

Inverse zig-zag scan

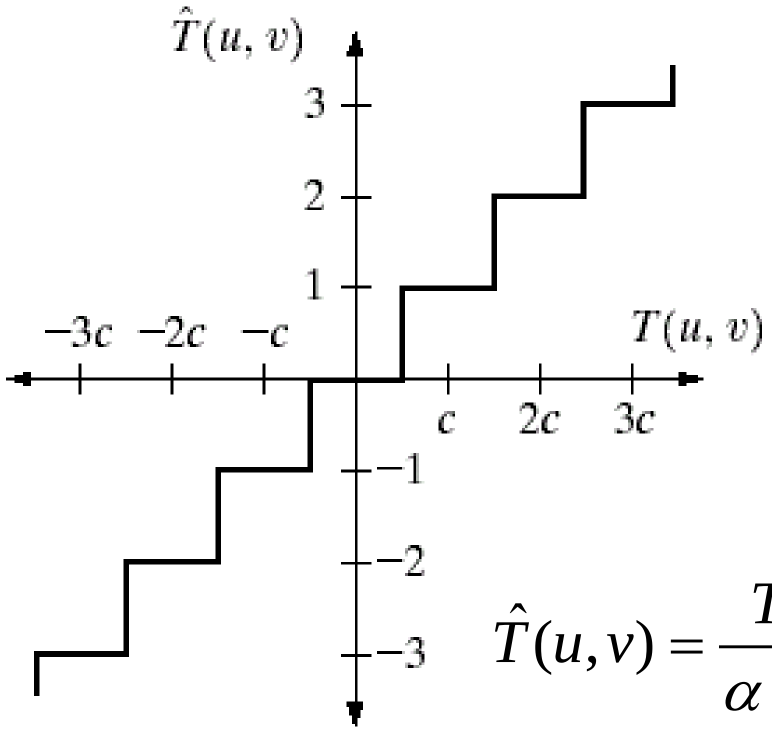
185	3	1	1	-3	2	1	0
1	1	-1	0	-1	0	0	1
0	0	1	0	-1	0	0	0
1	1	0	1	0	0	0	-1
0	0	1	0	0	0	-1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Reconstructed 8x8 block

192	201	195	184	177	184	193	174
189	191	195	182	182	187	190	171
188	185	190	181	185	187	189	171
189	188	185	183	183	182	190	175
191	192	186	189	179	182	188	178
190	191	189	190	177	186	184	179
189	188	185	184	175	186	187	179
189	188	178	176	173	183	193	180

Threshold Coding Using Normalization Matrix

Threshold coding quantization curve



$$\hat{T}(u,v) = \frac{T(u,v)}{\alpha\, Z(u,v)}$$

Normalization matrix Z

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

Examples of Transform Coding

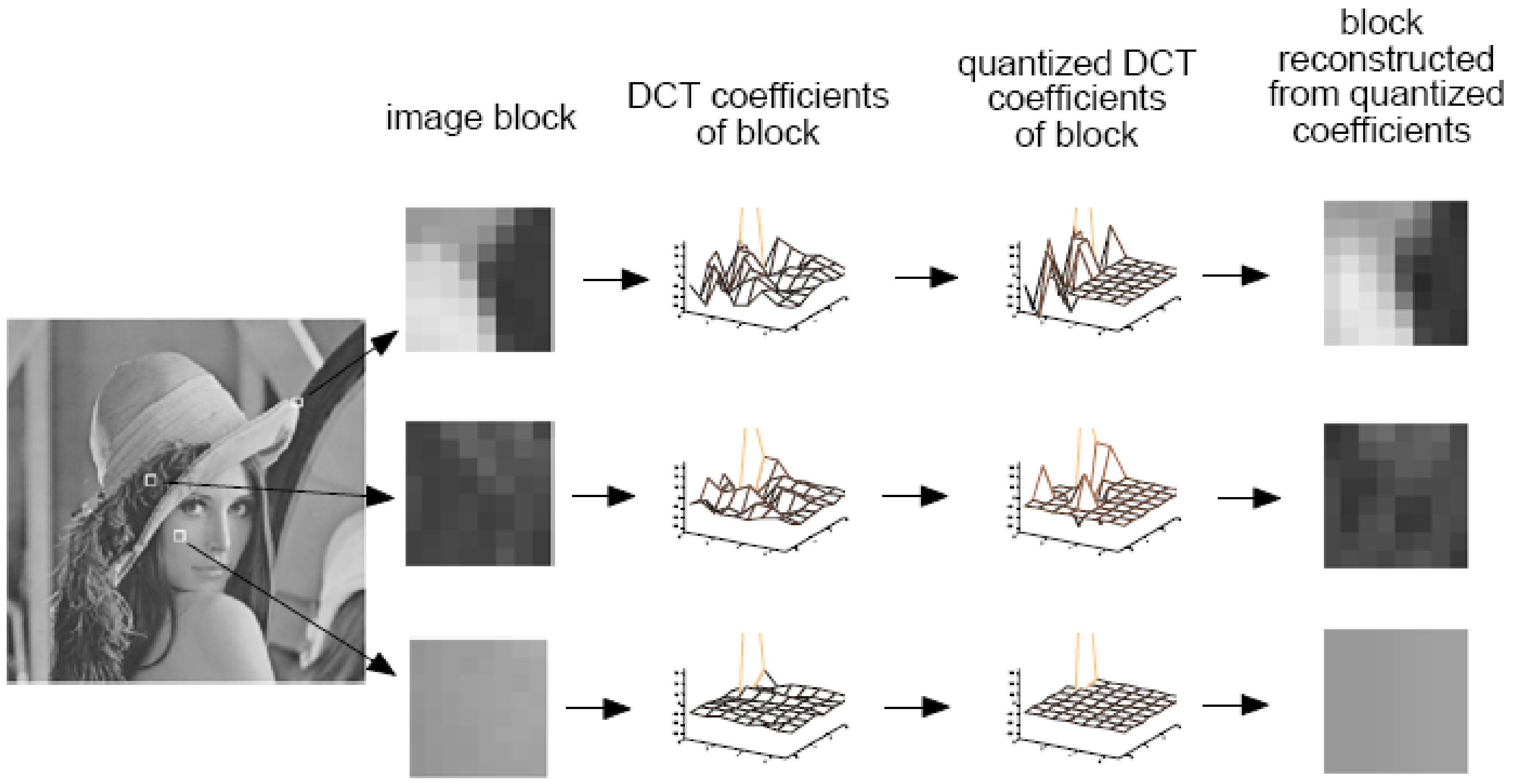
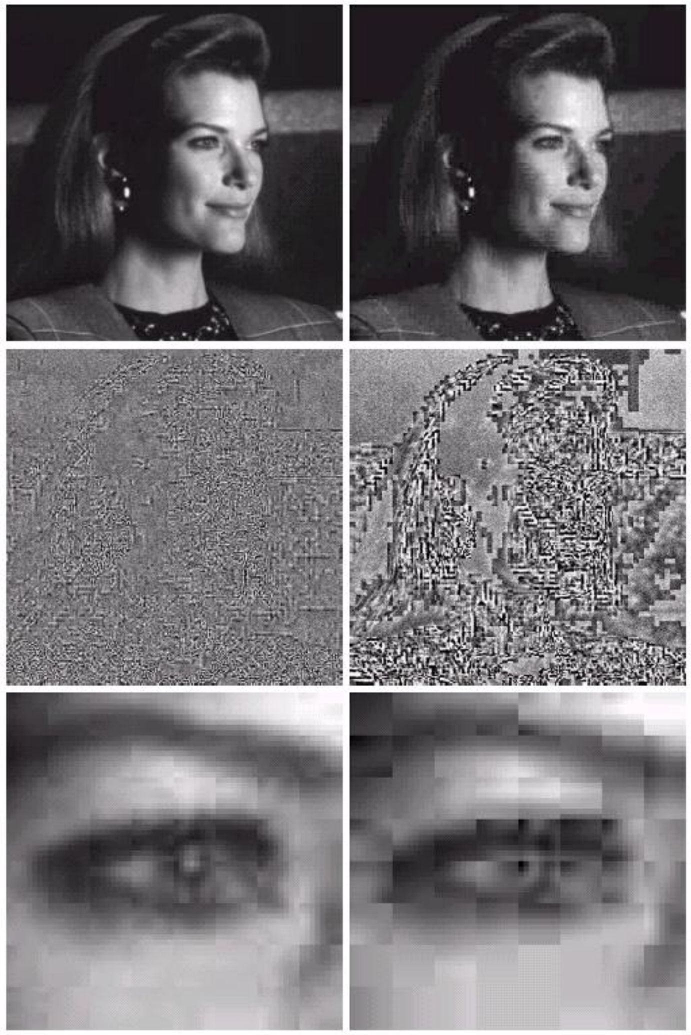


image block
DCT coefficients of block
quantized DCT coefficients of block
block reconstructed from quantized coefficients

Image Compression Error (Distortion)



Original Image

Compression Error

Reconstructed image

Quantization using Z 4Z

Quantization Tables in JPEG

- Luminance

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	36	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

- Chrominance

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