# **Chapter 2**

## **Image Representation and Display**

**In Matlab**

Images maybe considered as matrices whose elements are pixel values of the image.

1. **Reading and Displaying Images**

**Grayscale Images**: Reading a grayscale image called ‘fourier.bmp’

*>> imshow(fourier.bmp’)*



Fig. 1 Image display in Matlab

We can also save the image as a matrix by

*>> fou=imread(‘fourier.bmp’);*

Now *fou* is a matrix of with pixel values. To display the image

*>>figure, imshow(fou)*

where figure creates a window on the screen for displaying the image or a graph, and *imshow(fou)* shows the image in the window.

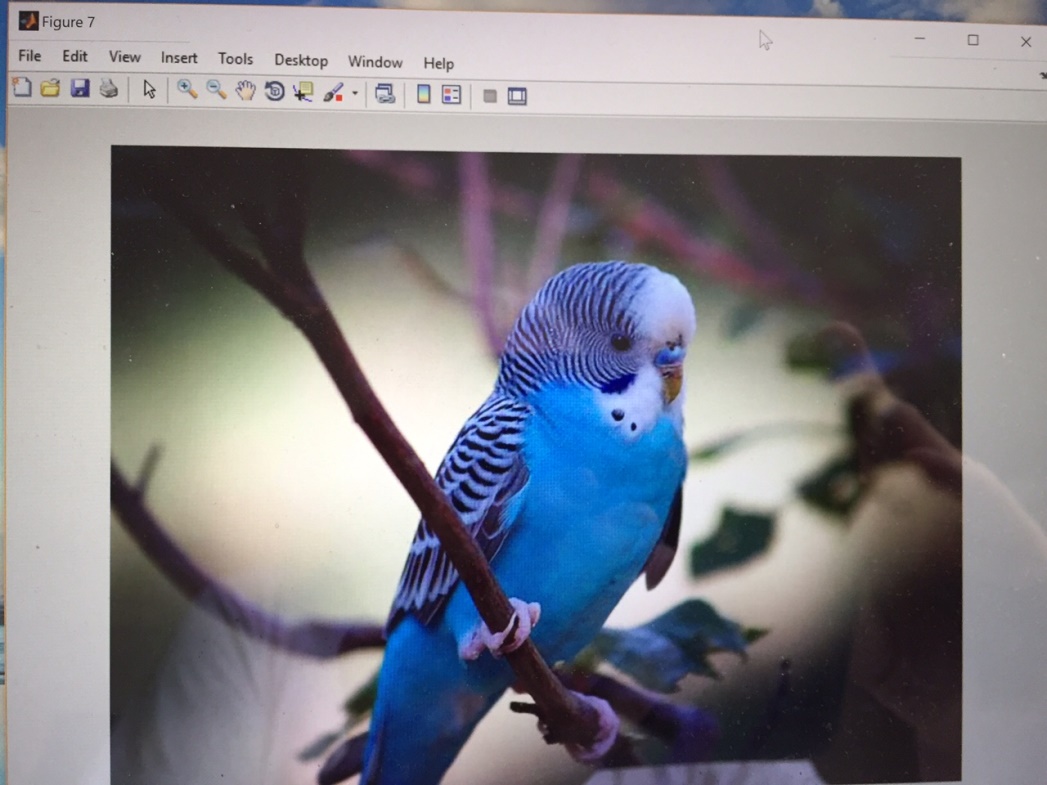


Fig. 2 Displaying the image in a frame

**Color Images**: The 24 bit color images are displayed the same way as grayscale images.

*>> bird=imread(‘bird.jpg’);*

*>> figure, imshow(bird)*

*>> size (bird)*

*ans =*

*525 700 3*

returns number of rows, 525, columns, 700, and bands (colors, that is RGB)

*>> bird(100, 200,2)*

*ans =*

*106*

gives the pixel value at x=100, y=200 and band 2 (i.e. green) which is 106. To get all three values, i.e. red, green and blue at pixel x=100, y=200:

*>> impixel(bird,200,100)*

*ans =*

*106 106 98*

To make the image darker by a factor of 2, i.e. multiply every pixel value by 2:

*>> M=imread('Bird.jpg');*

*>> imshow(M/2)*

Similarly we can multiply every pixel by a factor of 1.5 and add 50 to each pixel:

*>> M1=1.5\*M+50;*

**Detailed image information** about the image can be obtained as:

*>> imfinfo('fourier.bmp')*

*ans =*

*Filename: 'C:\Users\mtarokh\Desktop\AllFiles 01.20.2018\CS559\Fall 18\fourier.bmp'*

*FileModDate: '29-Jul-2017 18:09:20'*

*FileSize: 262962*

*Format: 'bmp'*

*FormatVersion: 'Version 3 (Microsoft Windows 3.x)'*

*Width: 512*

*Height: 512*

*BitDepth: 8*

*ColorType: 'indexed'*

*FormatSignature: 'BM'*

*NumColormapEntries: 191*

*Colormap: [191x3 double]*

*RedMask: []*

*GreenMask: []*

*BlueMask: []*

*ImageDataOffset: 818*

*BitmapHeaderSize: 40*

*NumPlanes: 1*

*CompressionType: 'none'*

*BitmapSize: 262144*

*HorzResolution: 2925*

*VertResolution: 2925*

*NumColorsUsed: 191*

Much of the above information is not useful for our purpose now, but we can see the size of the image (512 by 512), the size of image in bytes (262962 bytes), and number of bits per pixel (BitDepth =8).

For a color image we get:

*>> imfinfo('Bird.jpg')*

*ans =*

*Filename: 'C:\Users\mtarokh\Desktop\AllFiles 01.20.2018\CS559\Fall 18\Bird.jpg'*

*FileModDate: '17-Jul-2016 20:04:08'*

*FileSize: 37019*

*Format: 'jpg'*

*FormatVersion: ''*

*Width: 700*

*Height: 525*

*BitDepth: 24*

*ColorType: 'truecolor'*

*FormatSignature: ''*

*NumberOfSamples: 3*

*CodingMethod: 'Huffman'*

*CodingProcess: 'Sequential'*

*Comment: {}*

Note that the BitDepth=24 (i.e. 3 byte per color).

**Data Type**: Elements in an array can have different numeric data types, as follows:

|  |  |  |
| --- | --- | --- |
| Data type | Description | Range |
| logical | Boolean | 0 or 1 |
| int8 | 8 bit integer | -128 to 127 |
| uint8 | 8-bit unsigned integer | 0 to 255 |
| int16 | 16 bit integer | -32768 to 32767 |
| uint16 | 16-bit unsigned integer | 0 to 65535 |
| double or float | Double precision real umber | Machine specific |

Use class to find the data type of an image:

*>> class(M)*

*ans =*

*uint8*

1. **Image File Formats**

**Vector Versus Raster Images:** Vector format is a collection of lines or vectors. Raster format is a collect of values for the graylevel or other intensities of each pixel. The vector format has the advantage that the image can be magnified without losing sharpness. The disadvantage is that it is not very good for representation of natural scenes in which lines maybe few. The standard vector format is Adobe Postscript. Nearly all images captured by digital means (cameras and scanners) are stored as raster format.

**Simple Image format:**

* Header – image height, width, and a signature or “magic number” a sequence of bytes designed to identify the image format.
* Data part

header

Width, height, ..

Data

# Magic number

File formats generally fall into three categories:

**Device-specilized formats**: designed for specific hardeware, and suffer from lack of portability.

**Software specialized formats**: designed to be used with a particular program

* PCX and Windows bitmap formats found on PCs
* MacPaint used on Apple computers.

**Intechange formats**: designed to

* facilitate the exchange of image data between users
* usable with different hardware and software.

**Image compression** is standard features of interchange formats, some of which are:

* GIF (Graphic Interchange Format): came about for images on WWW.
* JFIF (JPEG File Intechnage Format): also appeared with WWW, uses special compression called JPEG to be discussed later.
* TIFF (Tagged Image File Format): Does not require information to be stored in fixed locations in an image file, instead it uses special strings or codes to identify a particular data items.
* PNG (Portable Network Graphics): is a replacement for GIF for legal reasons.
* PGM (Portable Gray Format): a popular format for grayscale image on Unix systems.
* FITS (Flexible Image Transport Format): allows large amount of information to be placed in the header.

**The Potable Formats Family**

**PGM, PBM, PPM formats:**

This family consists of PGM (Portable Grayscale Format), PBM (Portable Bitmap), PPM (Portable Pixmap). These formats are identified by two character signature, e.g. P1, P2

|  |  |  |
| --- | --- | --- |
| Signature | Image type | Storage type |
| P1 | binary | ASCII |
| P2 | grayscale | ASCII |
| P3 | RGB | ASCII |
| P4 | binary | raw byte |
| P5 | grayscale | raw byte |
| P6 | RGB | raw byte |

Signature of various portable formats

Raw image file contains minimally processed data from the image sensor of a digital camera.

Fig shows a very simple 7 by 7 grayscale image reprsentation as ASCII and raw PGM files (the raw format may have unprintable characters).

The advantage of ASCII format is that pixel values can be easily examined and changed using a text editor.

P2

# a simple PGM image

7 7 255

120 120 120 120 120 120 120

120 120 120 33 120 120 120

120 120 120 33 120 120 120

120 33 33 33 33 33 120

120 120 120 33 120 120 120

120 120 120 33 120 120 120

120 120 120 120 120 120 120

signature

comments

width

height

max gray level

However, the raw format is much more compact, and takes about one fourth space of the ASCII format.

P5

# a simple PGM file

7 7 255

xxxxxxxxxx ! xxxxxx ! xxxx !!!!! xxxx ! xxxxxx ! xxxxxxxxxx

(a)

**Examples of PGM (Portable Grayscale Format)**

The PGM and PPM formats (both ASCII and binary versions) have an additional parameter for the maximum value (numbers of grey between black and white) after the X and Y dimensions and before the actual pixel data. Black is 0 and max value is white. There is a newline character at the end of each line.

**P2**

**# Shows the word "FEEP"**

**24 7**

**15**

**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 3 3 3 3 0 0 7 7 7 7 0 0 11 11 11 11 0 0 15 15 15 15 0**

**0 3 0 0 0 0 0 7 0 0 0 0 0 11 0 0 0 0 0 15 0 0 15 0**

**0 3 3 3 0 0 0 7 7 7 0 0 0 11 11 11 0 0 0 15 15 15 15 0**

**0 3 0 0 0 0 0 7 0 0 0 0 0 11 0 0 0 0 0 15 0 0 0 0**

**0 3 0 0 0 0 0 7 7 7 7 0 0 11 11 11 11 0 0 15 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**

Example (magnified)



**PBM (Portable Bitmap) Example**

A simple example of the PBM format is as follows (there is a newline character at the end of each line):

**P1**

**# This is an example bitmap of the letter "J"**

**6 10**

**0 0 0 0 1 0**

**0 0 0 0 1 0**

**0 0 0 0 1 0**

**0 0 0 0 1 0**

**0 0 0 0 1 0**

**0 0 0 0 1 0**

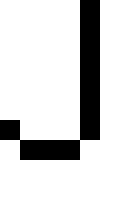
**1 0 0 0 1 0**

**0 1 1 1 0 0**

**0 0 0 0 0 0**

**0 0 0 0 0 0**

The string P1 identifies the file format. The hash sign introduces a comment. The next two numbers give the width and the height. Then follows the matrix with the pixel values (in the monochrome case here, only zeros and ones).

Magnified image:[](http://en.wikipedia.org/wiki/File:Example_of_ASCII-art_turned_into_a_bitmap_scale20.pbm.png)

The P4 binary format of the same image represents each pixel with a single bit, packing 8 pixels per byte, with the first pixel as the most significant bit. Extra bits are added at the end of each row to fill a whole byte**.**

**PPM (Portable Pixmap**) **Example**

This is an example of a color RGB image stored in PPM format. There is a newline character at the end of each line.

**P3**

**# The P3 means colors are in ASCII, then 3 columns and 2 rows,**

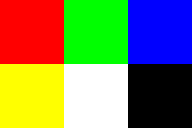
**# then 255 for max color, then RGB triplets**

**3 2**

**255**

**255 0 0 0 255 0 0 0 255**

**255 255 0 255 255 255 0 0 0**

The image (magnified): [](http://en.wikipedia.org/wiki/File:Tiny6pixel.png)

The P6 binary format of the same image represents each color component of each pixel with one byte (thus three bytes per pixel) in the order red, green, then blue. The file is smaller, but the color information is not readable by humans.

The PPM format is not compressed, and thus requires more space and bandwidth than a compressed format would. For example, a 192x128 PNG ([Portable Network Graphics](http://en.wikipedia.org/wiki/Portable_Network_Graphics)) image has a file size of 166 bytes. When converted to a 192x128 PPM image, the file size is 73,848 bytes. The PPM format is generally an intermediate format used for image work before converting to a more efficient format, for example the PNG format, without any loss of information in the intermediate step.

**The PNG (Portable Network Graphics) format**

* One of the newer formats, motivated by legal problems of patented compression algorithm in GIF.
* Supports gray scasle up to 16 bits/pixel, and RGB with up to 16 bits per band, or 48 bits/pixel.
* Also supports transparency value
* The data stream is compressed, and is lossless

A PNG file consists of an 8 byte signature, followed by a series of “chunks”. Each chunk consists of

* a 32 bit integer giving number of byte in the chunk’s data field
* a four byte code to indicate chunk type
* a 32 bit cyclic redundancy check (CRC) for the chuck to test data validity.

Table 2.3 below shows examples of chunk type.

|  |  |
| --- | --- |
| Chunk type | Usage |
| IHDR | Image header |
| IDAT | Image data |
| IEND | End of image file |
| PLTE | Color palette |
| gAMA | Gamma correction |
| pHYs | Pixel physical dimensions |
| teXt | Texual comment |
| tIME | Time of last modification |

Table 2.3 Chunk types.

Every PNG file must have a chunk type IHDR (header), one or more IDAT (data), and must end with an IEND chunk.

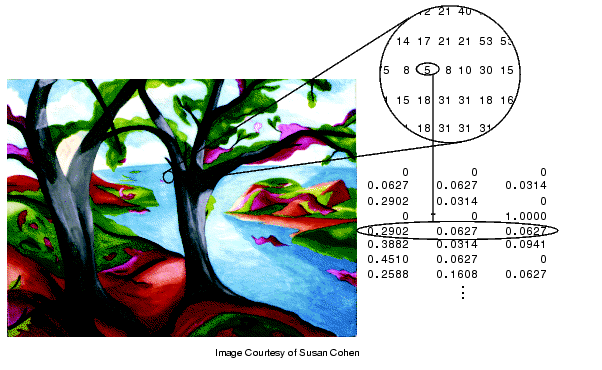
**Writing Image Files:**

*>> imwrite (M, ‘filename.abc’)*

Where M in the image matrix that is stored with the filename as an image file format abc (gif, jpg, tif, bmp). For example an image matrix M representing a horse, can be written to a PNG as

*>> imwrite(M, ‘horse.png’)*

Indexed image and color map: see <http://matlab.izmiran.ru/help/toolbox/images/intro5.html>



An indexed image can be written by including its colormap as well:

*>> imwrite(M, map, ‘filename.abc’)*

1. **Spatial Resolution and Quantization**

The function imresize changes the resolution.

*>> M2=imresize(M,1/2);imshow(M2)*

Reduces the size of the image rows and columns by a factor of 2. So

*>> size(M)*

*ans =*

*525 700 3*

*>> size(M2)*

*ans =*

*263 350 3*

If the argument of the imresize is more than 1, the pixels will be repeated. For example

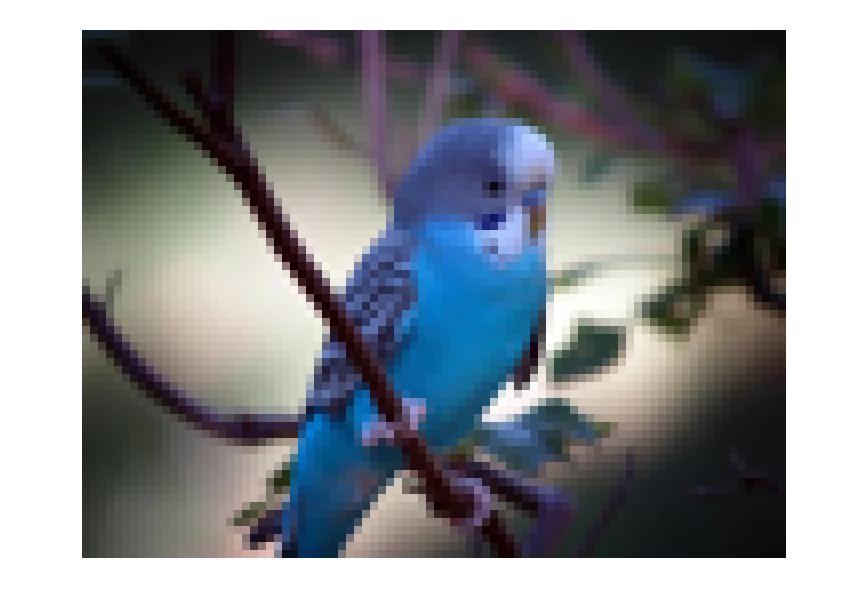
*>>imshow(imresize(M,2)*

Enlarges the image by 4 (twice rows and twice columns), and each pixel is repeated twice in each row and column.

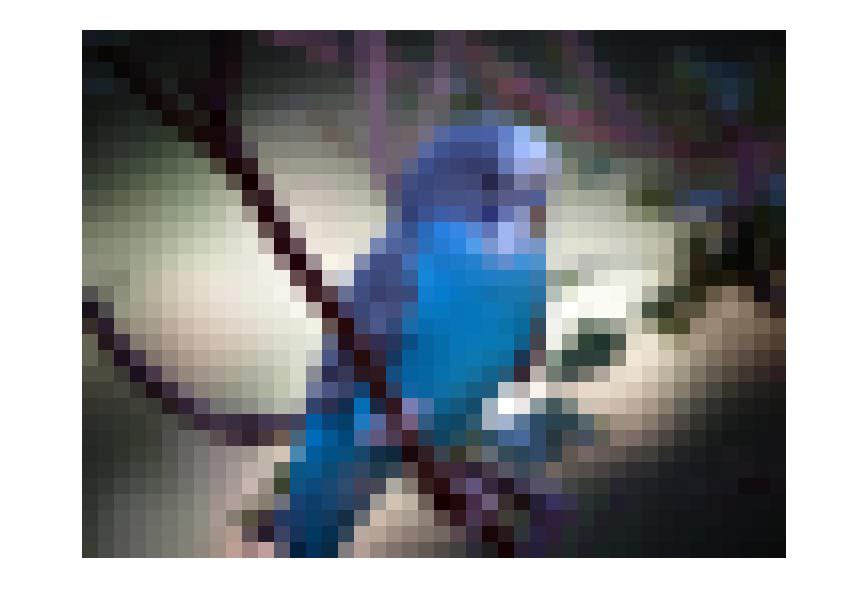
If we need to keep the appearance of the image the same with the reduced resolution, we need to increase the dimension of a pixel. For example if in a 200 x 200 image each pixel is 0.1 mm, then the image will show as 20 x 20 mm on the monitor. If we want to reduce the resolution to 100 x 100 but we want to show the reduced resolution the same 20 x 20 mm, we do the following:

*>> M2=imresize(imresize(M,1/2), 2, ‘nearest’);*

In the following images, Bird.jpg is first resized by 1/4 (Fig. 1) and again by another 1/2 so a total of 1/8.



(a) 1/4 resolution (132 x 175) (b) 1/8 resolution (66 x 88)



(c ) 1/16 resolution (33 x 44).

Fig. 3 Resolution and image size.

**Image Enlargement**

Sometime we want to increase the image resolution. This can be done by simply scaling the image (see below). However, this will reduce the quality of the image. In order to increase the resolution with little change in the quality of the image, bilinear interpolation and similar methods can be used.

**Scaling**: To enlarge or shrink an image by a factor of m, we have the following. Note that m > 1 is for enlarging and m < 1 is for shrinking.

* When the image is enlarged (m >1), then each pixel in the input image becomes an mm block in the enlarged image, and the gray level or color of all pixels in this block become the same as the pixel in the input image.
* When the image is to be shrunk, then we sample every m-th pixel in the input image to get the value of the output pixel.

**Bilinear Interpolation**

The method in Chapter 2 by copying values for the additional pixels to be inserted is very crude and results in pixelated image. A far better method is bilinear interpolation. Let the four pixels be at ***(x1,y1), (x1,y2), (x2,y1)*** and ***(x2,y2)*** and suppose we want to fill in pixels between them to enlarge the image. Let the pixel to be filled in be at ***(x,y).*** Then we first find the value of the pixel at ***(x1,y)*** by linear interpolation as



Similarly we find the value of the pixel at ***(x2,y)*** as



***(x1,y1) (x,y1) (x2,y1)***

***(x,y)***

***(x1,y2) (x,y2) (x2,y2)***

Now ***f(x,y)*** is computed as linear combination of ***f(x,y1)*** and ***f(x2,y)***as



**Interpolation Based on Areas**

In this method the value of the pixel at (x,y) is determined by the areas rather than distances in bilinear interpolation. Let A1 be the area enclosed bt points ***(x1,y1), (x,y1), (x,y)*** and ***(x1,y***). Similarly define ***A2, A3*** and ***A4***.

***(x1,y1) (x,y1) (x2,y1)***

***A1*** ***(x,y) A2***

***A3 A4***

***(x1,y2) (x,y2) (x2,y2)***



Where ***A = A1+A2+A3+A4***.

**Quantization** refers to the number of graylevels (or each of R, G, B) used to represent the image. Most images have 256 levels which is more than enough for human vision. We can reduce the number of levels. For example if a L=255 levels is to be reduced to L=4, in this case

Origin values 0utput value

0-63 0

64-127 1

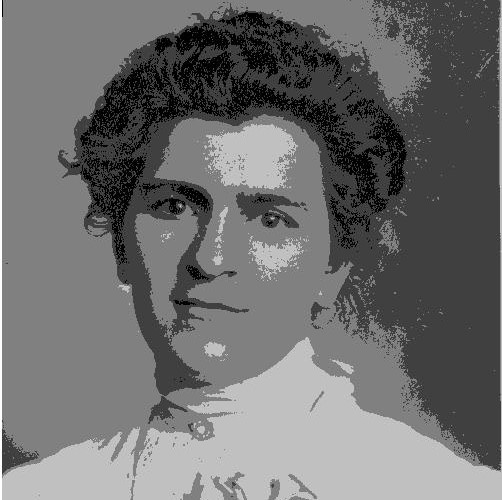
128-191 2

192-255 3

*>> QuantFourier=(Fou/64)\*64;*

*>> imshow(Quant.Fourier)*

Note that *Fou* is of type unit8 (values between 0 and 255), so dividing the values between values by 64, we will obtain values 0, 1, 2, 3. 4.Then multiplying by 64 the values will become 0, 64, 128 , 192 and 255. The result of the above operation is shown in the pictures below.

Original L=256 Quantized L=4

Fig. 4 The result of quantization

If we want to specify the number of graylevels 0,1,2, …, n then we use

>> Quant.n.Fourier =grayslice(Fou,n);

If n is small (say 4), the image produced above will show as black since it produce 0,1,2,3 which are close to zero. In order to scale it to 0, 64, 128 , 192, 255 we use *gray* which will produce evenly space graylevels between 0 and 255. For example

*>> Quant.n.Fourier=grayslice(Fou,16);*

*>> imshow(Quant.n.Fourier, gray(16))*



Fig, 5 Image quantized with L=16

1. **Printing of Images**

* Most printers produce a binary output, i.e. they either print a black dot or blank (white).
* Newspaper photographs simulate a grayscale by printing tiny black dots of varying size. The human visual system has a tendency to average brightness over small areas so that black dots and white background merge and are perceived as shades of gray.
* This process is called halftoning, and several methods of implementing it are available, as follows.

**Patterning**: In this method each pixel is replaced by a pattern taken from a binary font. An example of this font made up of 3 x 3 matrix of pixels is shown below for representing 10 graylevel values, from 0 to 9. In this case the width and height of the image is increased by a factor of 3.

0 1 2 3 4

5 6 7 8 9

Fig. 6 Halftoning via patterning

**Dithering**: This is accomplished by thresholding the image against a dither matrix. These matrices are square with dimension of power of 2. For seeing better details, the image is enlarged by a factor of where *m* is the dither matrix size. The elements of the matrix are thresholds. The first two dither matrices for an 8 bit grayscale image are

The matrix is laid on the image. The output pixel value becomes white (255) if the input pixel values is greater than the matrix element, or becomes black (0) if it is less than the matrix element.

**Dithering by Error Diffusion** : We start selecting a threshold, typically 128 for an 8-bit image. If f(x,y) > 128, the pixel at (x,y) in the output is set to white otherwise it is set to black. For pixels that are close to 128, the error will be large, and in order to reduce the effect, this error is spread or diffused to the neighboring pixels. A method for diffusing is shown below, where the error is spread among 4 neighbors that are ahead of the pixel, assuming top to bottom and left to right traversal. The values shown in Fig. are the factors by which the error is multiplied, the result is added to the corresponding pixel value

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

7/16

1/16

3/16

5/16

(x-1,y) (x, y+1) (x+1, y+1)

Fig 7. Error diffusion factors.

The algorithm by Floyd-Steinberg for error diffusion is

threshold = (black + white)/2

for ( all x and y) {

if (f(x,y) < threshold) {

g(x,y) = black // g(x,y) is the output image

error = f(x,y) – black }

else {

g(x,y) = white

error = f(x,y) – white

}

f(x+1,y) = f(x+1, y) + 7\*error/16

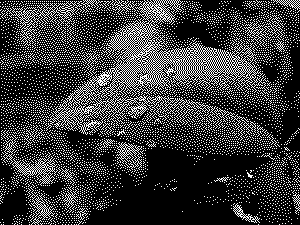
f(x-1, y+1) = f(x-1, y+1) + 3\*error/16

f(x, y+1) = f(x, y+1) + 5\*error/16

f(x+1,y+1) = f(x+1, y+1) + error/16

}

Examples of Dithering (magnified)

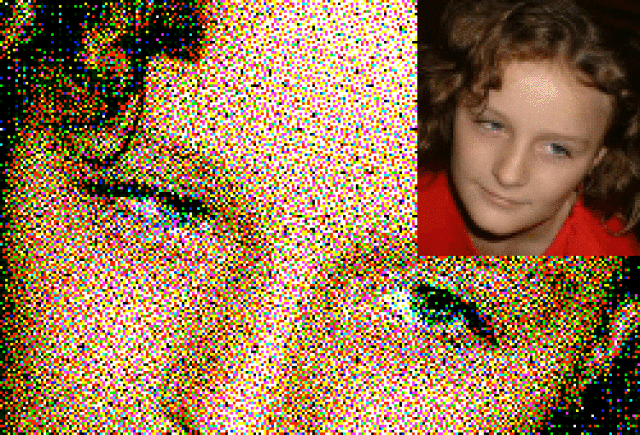
 

Fig8. Examples of dithering for grayscale and color images.

*Matlab function for Floyd-Steinberg dithering:*

*function dith=FloyedSteinberg(image, k)*

*height=size(image,1);*

*width= size(image,2);*

*ed=[0 0 0 7 0;0 3 5 1 0; 0 0 0 0 0]/16;*

*dith=unit8(zeros(height,width));*

*pict=zeros(height+4,width+4);*

*pict(3:height+2, 3:width+2)=image;*

*for x=3:height+2,*

*for y=3:width+2,*

*quant=floor(255/(k-1))\*floor(pict(x,y)\*k/256);*

*dith(x-2,y-2)=quant;*

*e=pict(x,y)-quant;*

*pict=(x:x+2, y-2:y+2)=pict(x:x+2, y-2:y+2)+e+ed;*

*endfor*

*endfor*

*endfunction*

The CMYK color model is used for printing of color images. The color printer has four inks – cyan, magenta, yellow and black. The output produced is binary, i.e. either one of the four colors is printed or absent of any point. One of the methods of halftoning is used for each color.

See also <https://webstyleguide.com/wsg2/graphics/dither.html>