Intro Math Problem Solving
October 31

"Black and White" (grayscale) images

Saving Graphics in a File

Image Read, Show and Write

Working with UINT8 Data

Lighten a Dark Image

Repairing Damaged Images

Final Project

Next week, details of the final project will show up on Canvas. It will look something like a homework set, except somewhat longer and harder. If you look now, you will see just a brief description of the tasks.

The final project should be turned in by midnight, 12 December, our last class meeting day.

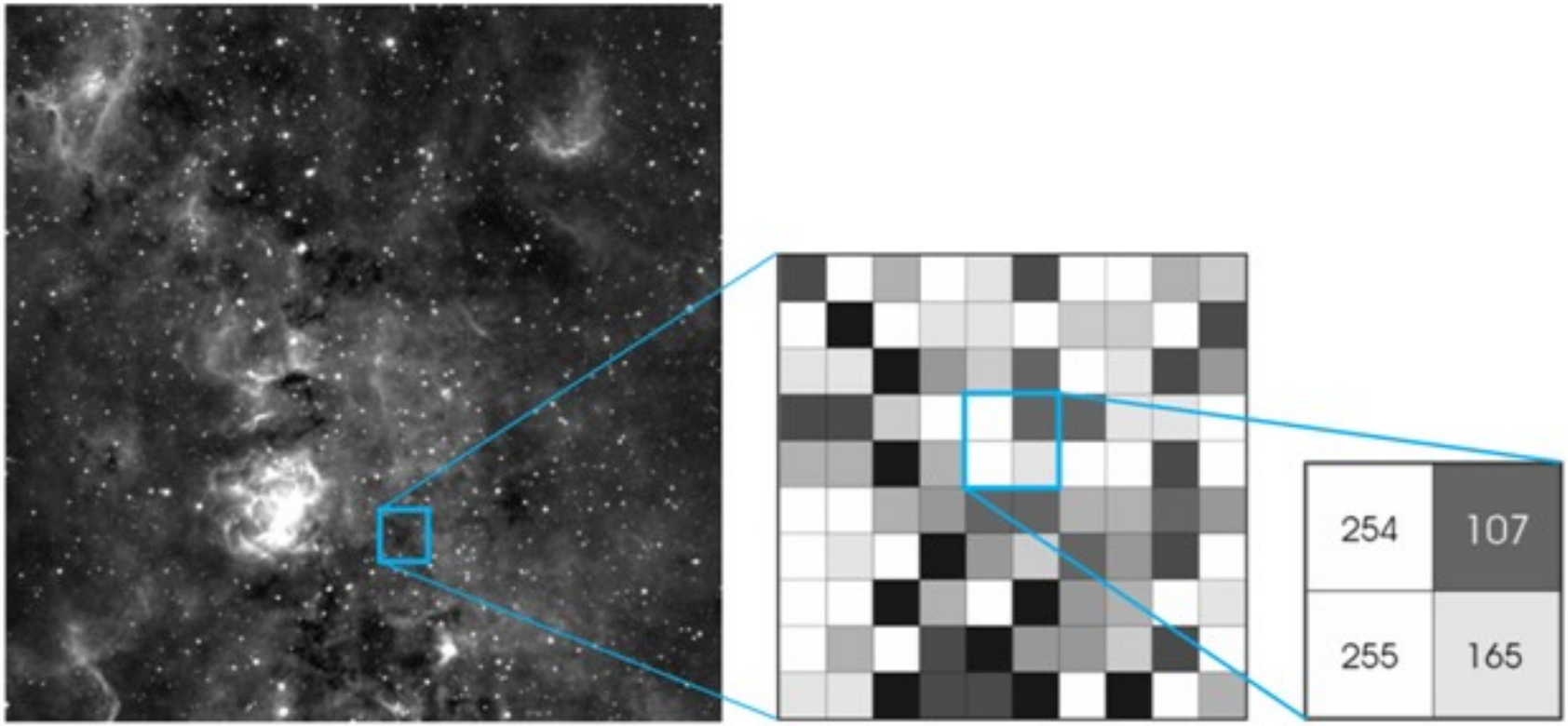
If you prefer to do a personalized final project, you must tell me so before November 16th (last class before Thanksgiving break).

Reference

Chapter 12, Section 4 of our textbook "Insight Through Computing" discusses the way black-and-white and color image data is stored and manipulated.

.

Object -> Table -> Numbers



Pictures as Arrays

When a camera takes a "black and white" picture of some figure, it is encoded as a 2D array, which might be called "A".

The entries of A are numbers. Typically,

```
  0     <=     A(i,j)     <=   255
(black)                       (white)
```

Values in between correspond to different levels of grayness. A "black and white" picture is more properly called a "grayscale" image.

Need to Compress Information

Each entry of the matrix A corresponds to a single "pixel" in the image (a square picture element of a single color).

A typical computer image might have dimension 480x640, but a good camera might create images of 4672x3104 cells, so a single picture may have 12 million pixels.

Therefore, most computer images use some kind of compression to reduce the amount of computer memory needed.

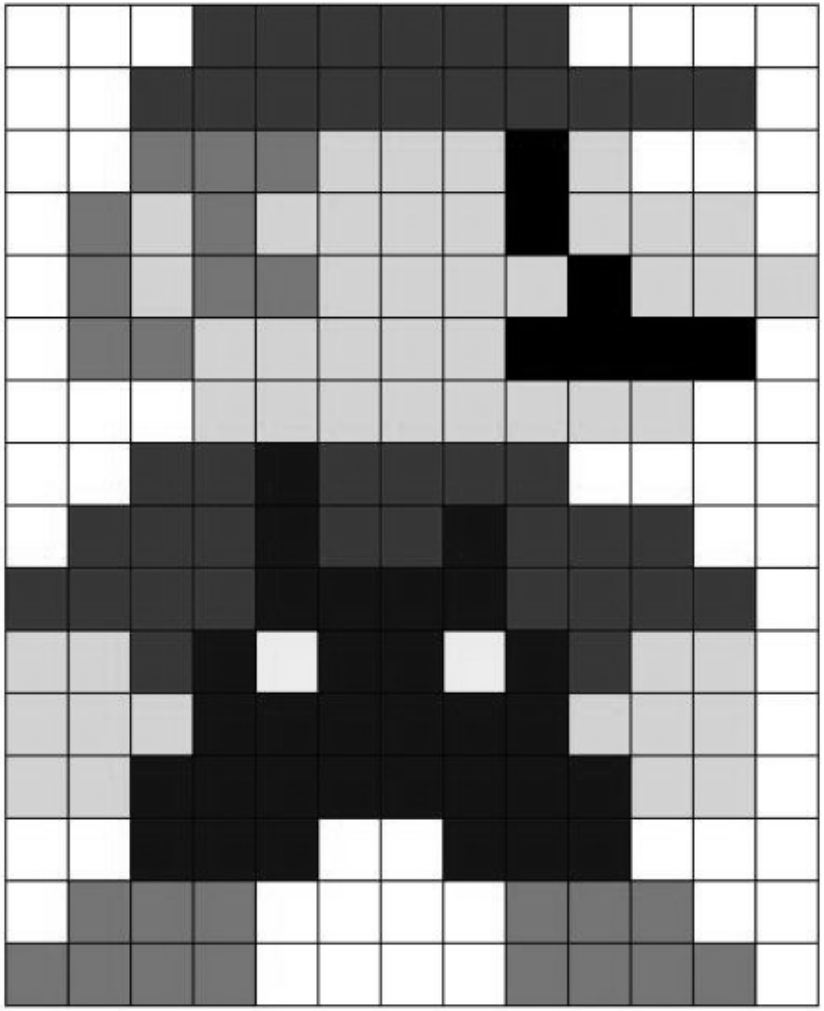
Images using Shades of Gray

Suppose we have a grid of pixels of various shades of gray.

It seems natural to try to store this information as a MATLAB matrix.

Your eye is often not a good judge of levels of grayness, but let's try to take a black-and-white Mario image, create the corresponding matrix, and see if we have recreated the image.

Portrait of Mario in Gray



Choose Numbers for Grays

We can guess we need about 7 shades of gray:

```
grays = floor ( linspace ( 0, 255, 7 ) )
```

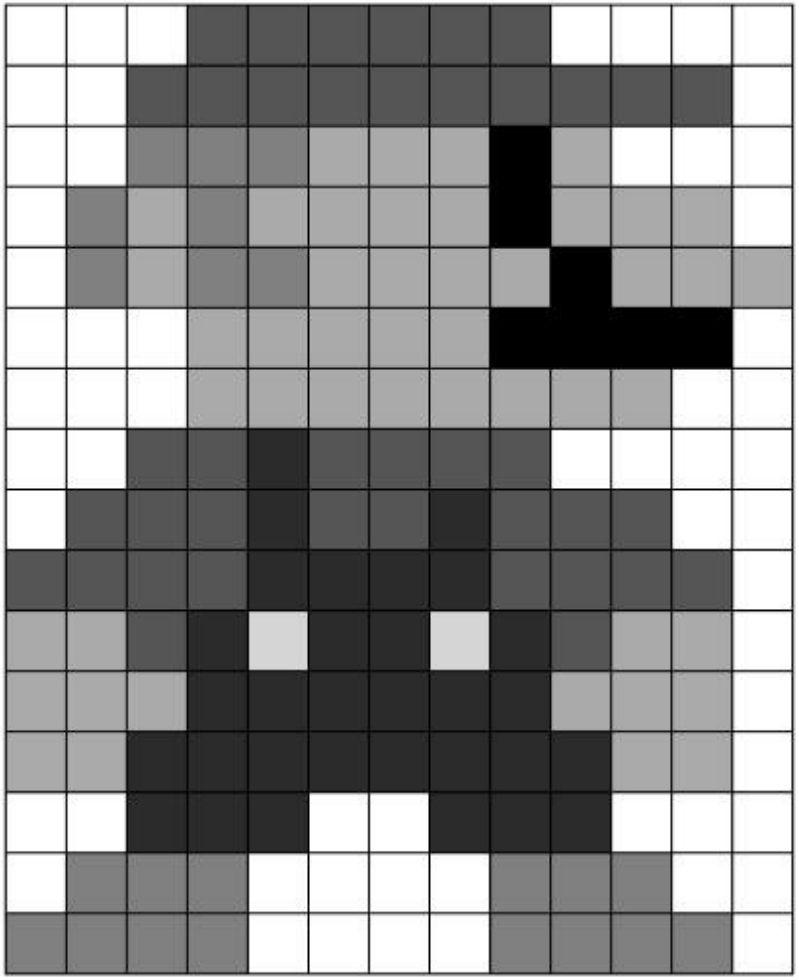
We can make a table with an index, name, and value.

	i	i/255	[R,G,B]
0: black	0	0.00	[0.00, 0.00, 0.00]
1: very dark	43	0.17	[0.17, 0.17, 0.17]
2: dark gray	85	0.33	[0.33, 0.33, 0.33]
3: half gray	127	0.50	[0.50, 0.50, 0.50]
4: light gray	170	0.67	[0.67, 0.67, 0.67]
5: very light	212	0.84	[0.84, 0.84, 0.84]
6: white	255	1.00	[1.00, 1.00, 1.00]

Our Color Specification is Simple

```
for i = 1 : m
  for j = 1 : n

    k = MARIO_GRAY(i,j);

    g = k / 6;
    color = [ g, g, g ];

    a = j - 1;
    b = j;
    c = m - i + 1;
    d = m - i;
    fill ( [ a, b, b, a ], [ c, c, d, d ], color );

  end
end
```

Our "Computed" Mario is Pretty Good



Modified Gray Scale

We chose evenly spaced gray levels.

grays = floor (linspace (0, 255, 7))

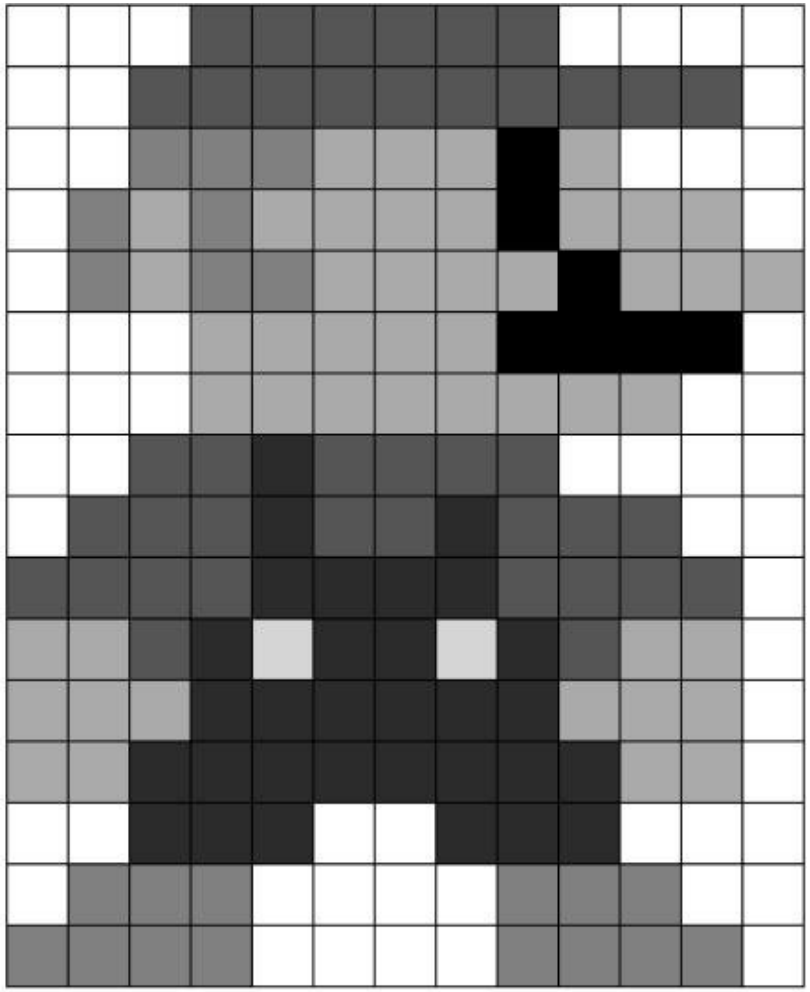
But the eye is better at distinguishing light grays than darks ones. Try a new scale using the square roots of old values:

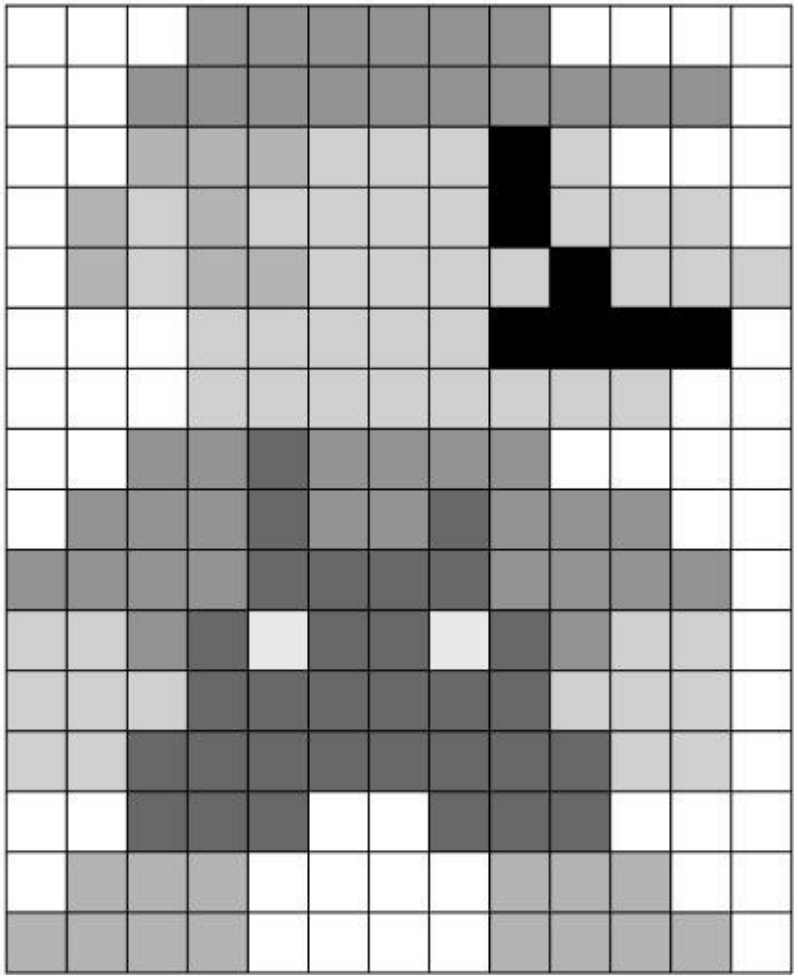
gray2 = sqrt (grays)

0.00 0.40 0.57 0.70 0.81 0.91 1.00

Now most of the grays have moved towards white, that is, they have gotten lighter.

Compare Linear and SQRT Grays





Convert RGB to Gray Images

Suppose we have an RGB image and we want to create a grayscale version automatically? For each colored pixel, we need to replace [R,G,B] by [g,g,g] where g is an appropriate gray value.

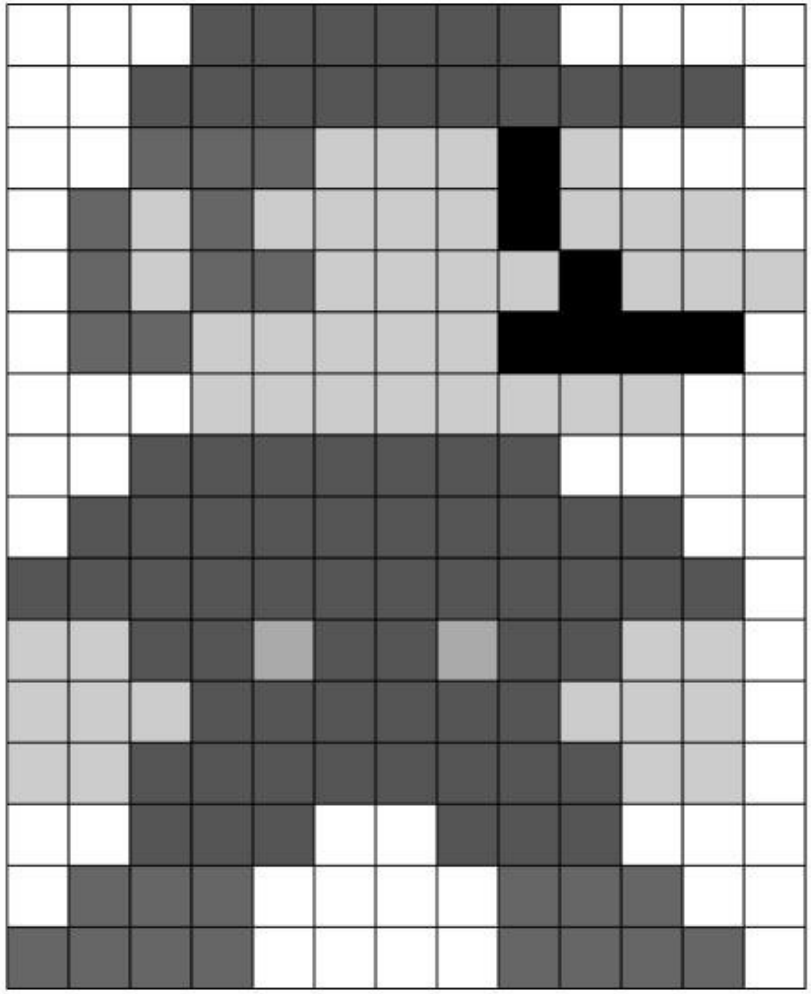
One choice is a **uniform** average of R, G, and B.

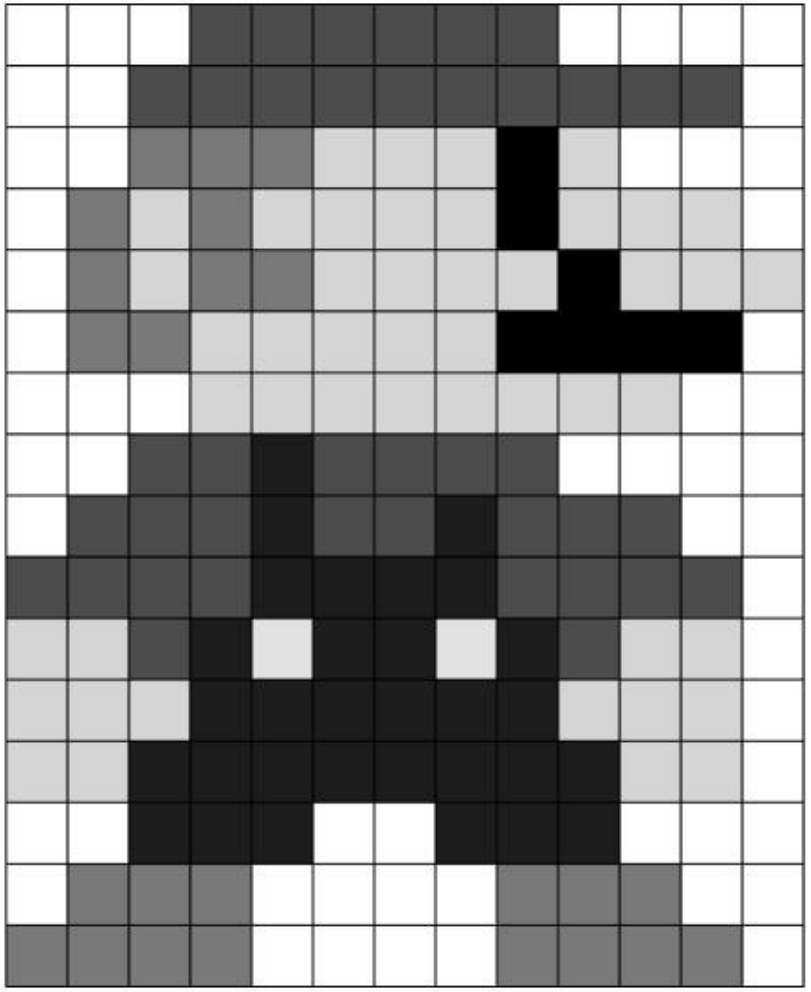
$g = (R+G+B)/3.$

Because of properties of the eye, a better choice is a **weighted average** (more green, less blue);

$g = 0.299*R + 0.587*G + 0.114*B.$

Uniform vs Weighted RGB Average





Saving Graphics in a File

Graphics Information

Most graphics information can be thought of as a 2D color image using an MxN array of pixels whose R, G and B values are stored.

In the interest of saving space, various methods can be used to compress the information, but it still can be understood in terms of a matrix of colors.

The R, G, B values are usually integers, mostly commonly between 0 and 255. (When we work with MATLAB, we often need to convert these to real numbers between 0 and 1.)

In a black-and-white (grayscale) image, only one color value is stored, or each pixel has equal R, G and B values.

There are many formats for saving graphics information to a file. We will look at one of the simplest, for grayscale images.

The PGM Format

PGM: Portable Grayscale Map (Text version)

Line #1: P2

Line #2: Width(N) Height(M)

Line #3: 255 (assuming 0-255 scale)

Line #3+1: row 1 values

…

Line #3+M: row M values

mario.pgm

```
P2
13 16
255
 255 255 255  85  85  85  85  85  85 255 255 255 255
 255 255  85  85  85  85  85  85  85  85  85  85 255
 255 255 127 127 127 170 170 170   0 170 255 255 255
 255 127 170 127 170 170 170 170   0 170 170 170 255
 255 127 170 127 127 170 170 170 170   0 170 170 170
 255 255 255 170 170 170 170 170   0   0   0   0 255
 255 255 255 170 170 170 170 170 170 170 170 255 255
 255 255  85  85  43  85  85  85  85 255 255 255 255
 255  85  85  85  43  85  85  43  85  85  85 255 255
  85  85  85  85  43  43  43  43  85  85  85  85 255
 170 170  85  43 212  43  43 212  43  85 170 170 255
 170 170 170  43  43  43  43  43  43 170 170 170 255
 170 170  43  43  43  43  43  43  43  43 170 170 255
 255 255  43  43  43 255 255  43  43  43 255 255 255
 255 127 127 127 255 255 255 255 127 127 127 255 255
 127 127 127 127 255 255 255 255 127 127 127 127 255
```

Advantages of File Storage

We created a Mario image with MATLAB, but to share it with a remote friend, we'd have to send the MATLAB program, and hope the friend had MATLAB installed.

A graphics file allows different users on different computer systems with different software to share graphics information.

The graphics information can also be created by one program, and then modified by another. For instance, a weather satellite might create an image of the ocean. The graphics data can be processed by a program that searches for evidence of hurricane formation. Multiple images like this could be joined to form an animation.

Many Image File Formats

While the simple PGM format is probably new to you, you know about JPEG/JPG, PNG, TIFF, BMP and GIF formats. These all represent different ideas for efficiently storing the MxN pixel color information.

PDF and PS/EPS are related, but more complicated formats that concentrate on storing text, or images represented as a sequence of line segments.

ImageMagick is one example of a program that allows you to display graphics information from most of these formats, or to convert it from one format to another.

MATLAB: Image Read and Write

We know how to create graphics images inside of MATLAB.

However, MATLAB has many tools for image processing; to apply them to photographs and images we created elsewhere, it's necessary to be able to import images, display them, modify them, and then export them again.

Let's begin with the command to import:

I = **imread** ('*filename*')

The imread() function can handle most common image file formats.

I = imread ('mario.pgm')

```
>> I = imread ( 'mario.pgm' )

I =

  16×13 uint8 matrix

   255   255   255    85    85    85    85    85    85   255   255   255   255
   255   255    85    85    85    85    85    85    85    85    85    85   255
   255   255   127   127   127   170   170   170     0   170   255   255   255
   255   127   170   127   170   170   170   170     0   170   170   170   255
   255   127   170   127   127   170   170   170   170     0   170   170   170
   255   255   255   170   170   170   170   170     0     0     0     0   255
   255   255   255   170   170   170   170   170   170   170   170   255   255
   255   255    85    85    43    85    85    85    85   255   255   255   255
   255    85    85    85    43    85    85    43    85    85    85   255   255
    85    85    85    85    43    43    43    43    85    85    85    85   255
   170   170    85    43   212    43    43   212    43    85   170   170   255
   170   170   170    43    43    43    43    43    43   170   170   170   255
   170   170    43    43    43    43    43    43    43    43   170   170   255
   255   255    43    43    43   255   255    43    43    43   255   255   255
   255   127   127   127   255   255   255   255   127   127   127   255   255
   127   127   127   127   255   255   255   255   127   127   127   127   255
```

IMREAD: 4 Facts

1) The command "I = imread ('filename')" looks for the named graphics file, and creates a corresponding matrix.
2) Because this is a grayscale image, we only see one RGB value; Color images will be more complicated.
3) We usually want a semicolon on this command, to avoid seeing all that data.
4) This is a "uint8" array, literally an array of "unsigned integers using 8 bits", or simply, integer values between 0 and 255. This fact has important consequences for us.

IMSHOW: Display an Image

Now that "I" is a matrix containing the grayscale image data, we can use MATLAB's imshow() command to view it;

```
imshow ( I )
```

Sadly, MATLAB displays our image exactly, namely a tiny image of 16 rows and 13 columns, which is almost impossible to see!

.

imshow (I) (after zooming in!)

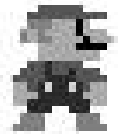


Image Blowup

To make our image visible, MATLAB's zoom function is not very helpful! (Try it!)

Instead, we can "blow up" our image by replacing each single pixel by a block of the same value:

```
                AABB
A B  -- x2 -->  AABB
C D             CCDD
                CCDD
```

We'll replace each pixel by a 32x32 block:

```
I32 = gray_blowup ( I, 32 );
```

imshow (I32)

gray_blowup.m

```
function A2 = gray_blowup ( A1, f )

  [ m, n ] = size ( A1 );
  A2 = uint8 ( zeros ( f*m, f*n ) );

  for i = 1 : m
    ilo = ( i - 1 ) * f + 1;
    ihi = i * f;
    for j = 1 : n
      jlo = ( j - 1 ) * f + 1;
      jhi = j * f;
      A2(ilo:ihi,jlo:jhi) = A1(i,j);  % <-  1 A1 pixel becomes FxF A2 block
    end
  end
```

Simple manipulations

Since I is a matrix, we can matrix operations on it:

Let's reverse video:

```
I2 = 255 – I:
```

And flip the picture from left to right:

```
I3(1:16,1:13) = I2(1:16,13:-1:1);
```

Use the "gray_blowup" routine to make the plot visible.

```
I4 = gray_blowup ( I3, 32 );
```

Reverse Video + Flip Left/Right

IMWRITE: Save New Image to File

To save a graphics image to a file, use the imwrite() function:

```
imwrite ( I3, 'mario.jpg' )
```

To save as a (text) PGM file, we have to say something like:

```
imwrite ( I3, 'mario.pgm', 'Encoding', 'ASCII' )
```

The last two inputs keep MATLAB from writing a compressed binary file that we can't easily type out.

Interactive alternative: any MATLAB image being displayed also has a menu

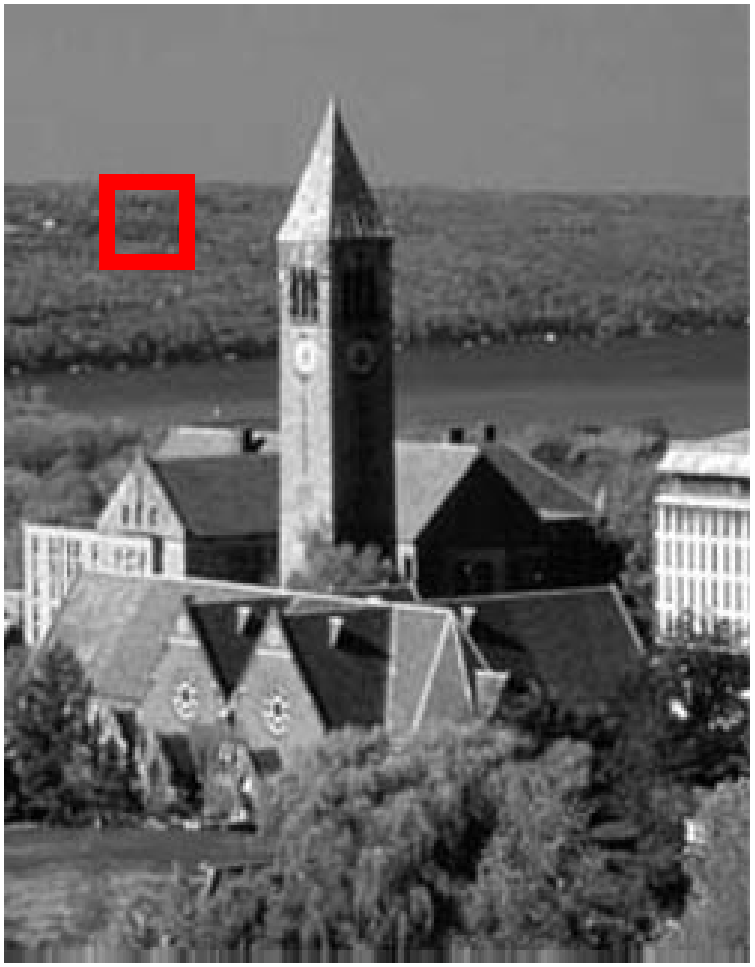
FILE / Save As / ...

that allows you to save the image in a variety of formats.

Working with UINT8 Data

318-by-250

49	55	58	59	57	53
60	67	71	72	72	70
102	108	111	111	112	112
157	167	169	167	165	164
196	205	208	207	205	205
199	208	212	214	213	216
190	192	193	195	195	197
174	169	165	163	162	161



Working with UINT8 Data

Up to now, the numeric data we have been working with has all been classified as what MATLAB calls type "double"; in other words, real numbers that are stored using 64 bits.

MATLAB's default graphics arrays are of a new type, "uint8" which we have seen consist of integers between 0 and 255.

We need to understand the differences in arithmetic associated with uint8 data.

We also need to know how to convert between uint8 and double data, and when this is necessary.

Some Operations Are OK

When A=imread('filename') is called to read a grayscale file, the matrix A it creates is typically of type uint8.

Copying A creates a new uint8 array:

```
B = A;
```

We can set entries to integers between 0 and 255:

```
A(5,4) = 77;  A(1:16,1) = 255; A(2:5,1:8) = 8;
```

We can use logical indices:

```
I = ( A >= 127 );
A(I) = 127;
```

But A cannot store real numbers, or integers outside the range of 0 to 255.

Integers Become Doubles

Working with data from the A matrix, some operations produce uint8 values, some produce doubles:

max(A) -> uint8 vector

mean(A) -> double vector

median(A) -> uint8 vector

sqrt(A) -> double vector

(we did this for our gray values earlier)

sum(A) -> double vector (why?)

(A(1,2) + A(1,3)) / 2 -> double value

RESHAPE: Matrix -> Vector

We will shortly be in a situation where we need to use MATLAB's histogram plotting command **hist(g)**. This command expects the input to be a vector (a 1D list), but we want to histogram all the data in a matrix G (a 2D table).

We use MATLAB's **reshape()** command:

```
a_new = reshape ( a_old, m_new, n_new);
```

which copies the data in the mxn object a_old into the m_new x n_new object a_new.

RESHAPE Examples

```
A = [ 11, 12, 13;
      21, 22, 23 ];          <- 2x3 matrix
b = reshape ( A, 1, 6 );   <- 1x6 row vector
b <- [ 11, 21, 12, 22, 13, 23 ];

C = reshape ( A, 3, 2 );  <- 3x2 matrix
C <- [ 11, 22;
       21, 13;
       12, 23];
```

Two Things to Watch out For

When we operate on numbers in a uint8 array, there are two things to worry about:

1) range: values might be less than 0 or greater than 255;

A(1,2) + A(1,3) could be more than 255

2) precision: values might not be integers;

(A(1,2) + A(1,3)) / 2

Range Issues

We may be able to deal with range issues by truncation. The uint8() function will take care of this:

Every result greater than 255 is replaced by 255:

A(1,1) = uint8 (A(1,2) + A(1,3));
255 <-- uint8 (170 + 158);

Every result less than 0 becomes 0:

A(8,9) = uint8 (A(3,3) – A(3,8));
0 <-- uint8 (127 – 170);

Precision Issues

The uint8() function will also take any value that is not an integer and convert it to the nearest integer in the range [0,255];

```
A(3,7) = uint8 ( ( A(2,7) + A(4,7) ) / 2 );
  44  <-- uint8 ( ( 49 + 40 ) / 2 );

A(4,8) = uint8 ( sqrt ( A(4,8) ) );
  9    <-- uint8 ( sqrt ( 89 ) );

A(1:13,8) = uint8 ( mean ( A(1:13,8) ) )
  74   <--  uint8 ( 73.752 );
```

Convert Back and Forth

When working with image data, sometimes it will be easier to make a "double" copy of the uint8 array, and then use any numerical operations we want, and at the end, convert the whole array back to uint8 format:

```
A = double ( I );  % <- copy I
  ...              % <- work on A
I = uint8 ( A );   % <- replace I with results
```

Lighten a Dark Image

Dark Image

Many photographs are taken in bad light, or have been badly developed.

Darkened images, in particular, are hard for the eye to interpret, whereas lighter shades are easier to "read".

For this photograph, we can display how the dark and light shades have been used by counting how many times each gray shade was used, between 0 and 255.

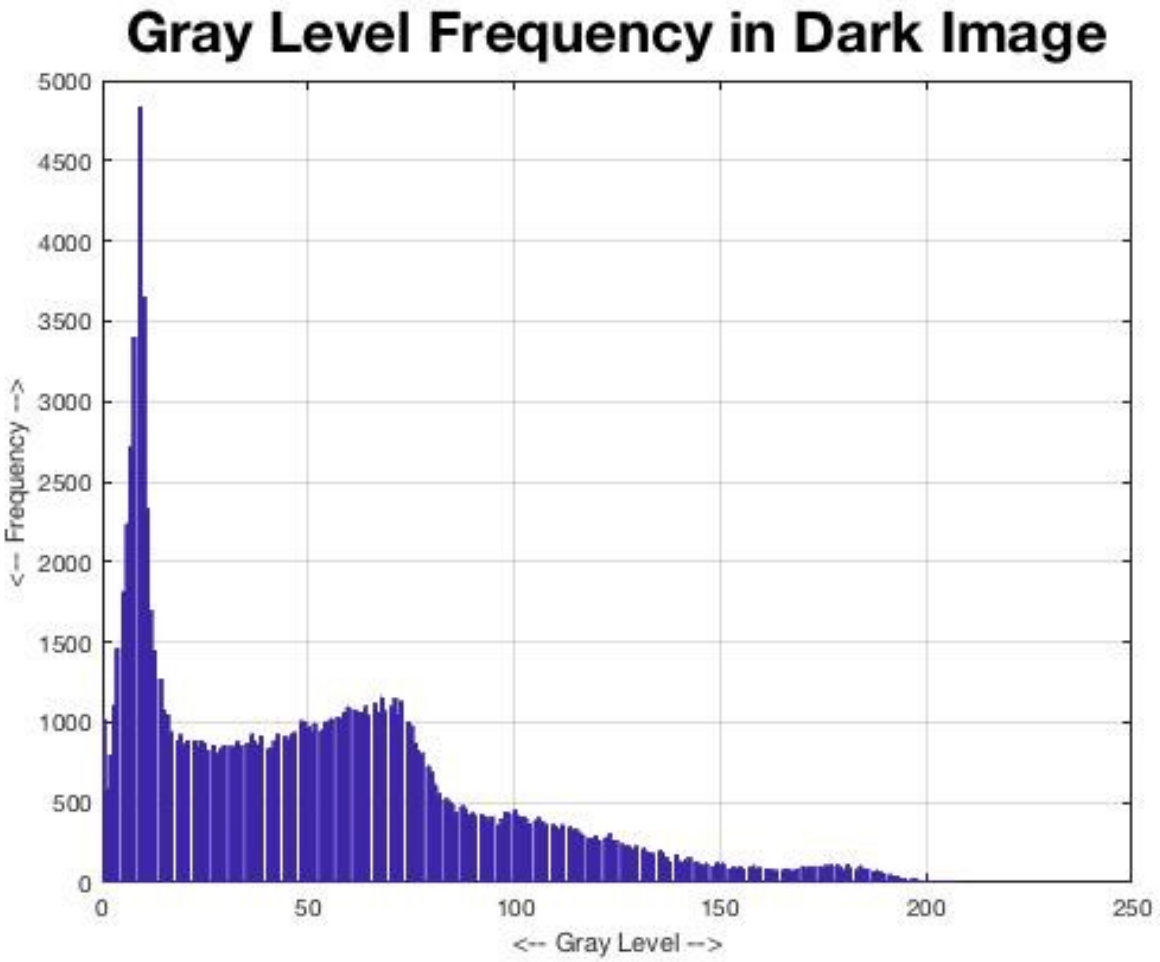
In other words, we want a bar plot or histogram.

Why So Dark?

A histogram of the gray levels will show how often each shade is used. However, MATLAB's hist() command will only accept "double" information, and it must be a vector:

```
I = imread ( 'snap.png' );
G = double ( I );                <- Convert to double
[ m, n ] = size ( G );
gvec = reshape ( G, m * n, 1 );  <- Column vector
hist ( gvec, 256 );                 <- histogram
```

Heavy use of Darkest Grays



Lighten the Levels

We are hardly using the lighter gray levels in the picture. Here are two ways we can try:

1) Double grays between 0 and 127;

   Grays about 127 become 255;

2) Gray -> sqrt(Gray/255) * Gray;

Doubling

```
I = imread ( 'snap.png' );
i1 = ( I < 128 );
i2 = ( 128 <= I );
I2(i1) = 2 * I(i1);
I2(i2) = 255;
imshow ( I2 );
```

Lightening by Doubling

Sqrt

```
I = imread ( 'snap.png' );
G = double ( I );
G = sqrt ( G / 255 ) * 255;
I3 = uint8 ( G );
imshow ( I3 );
```

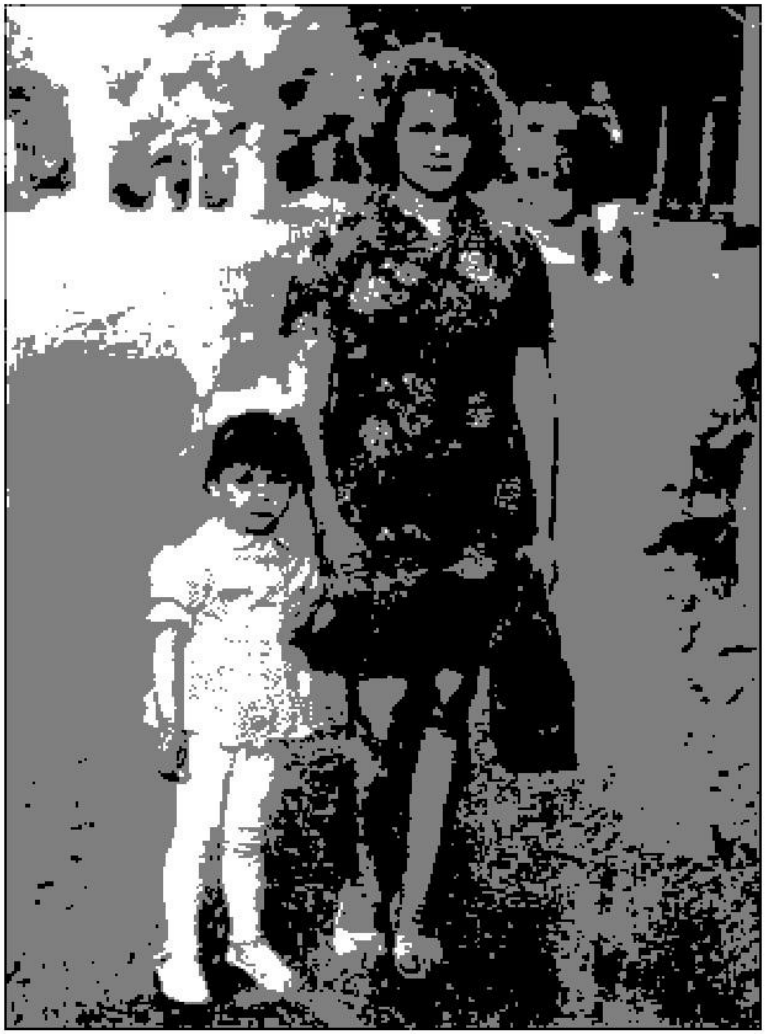
Lighten by Sqrt

Dark, Medium, Light

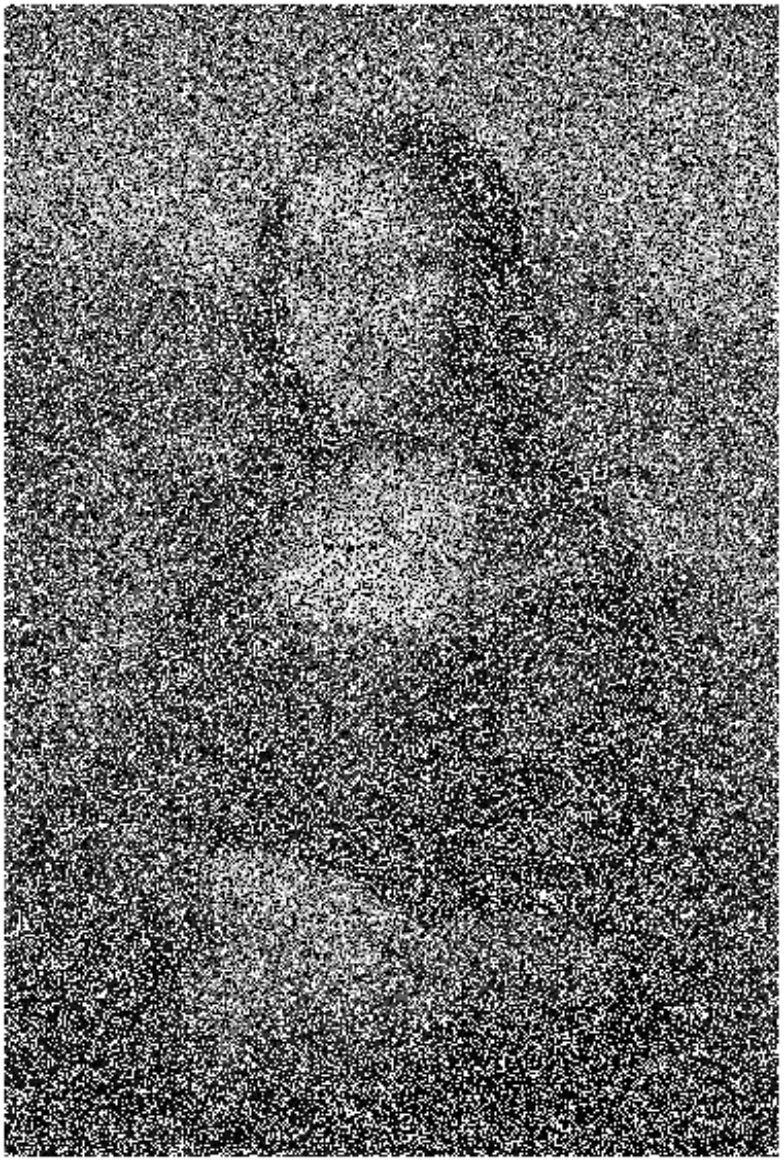
We can separate the image into dark, medium and light sectors:

```
I = imread ( 'snap.png' );
i1 = ( I < 25 );
i2 = (     25 <= I & I < 100 );
i3 = (                       100 <= I );
[m,n] = size ( I );
I2 = uint8 ( zeros ( m, n ) );
I2(i1) = 0;
I2(i2) = 127;
I2(i3) = 255;
imshow ( I2 );
```

3 shades: Black, Medium Gray, White



Repairing Damaged Images



Salt and Pepper Noise

A common kind of damage to photographs or graphics files involves "salt and pepper noise", in which the correct grays of many pixels have been lost, and replaced by random values, some of which may be light (white) or dark (pepper).

The damage can be so severe that the eye cannot make out the original image.

But often, there is enough information to recover an approximation of the original image, using a technique called "filtering".

Filtering

To filter out our noise, we assume that in the original image, the information changed smoothly; thus, in the gray scale matrix, we'd generally expect to see that neighboring matrix entries had similar values.

If a pixel was replaced by a random noise value, then we'd expect that this value would "stick out", and probably be noticeably larger or smaller than most of the neighboring values.

This suggests that we could automatically detect most such cases. But can we "repair" them?

Perhaps Two Noisy Pixels Here?

157	170	166	166	160	160	157	153
159	166	166	166	163	163	157	154
161	160	160	166	164	164	158	160
163	158	160	158	166	245	158	164
165	158	154	147	146	157	159	159
166	169	27	159	146	150	154	159
166	166	166	159	146	148	154	167
165	165	166	166	157	155	153	160

Automatic Repair Strategies

NW -- N - NE 164 164 158
W— P — E 166 245 158
SW— S -- SE 146 147 159

Mean-filter: replace each pixel by the mean of the 3x3 neighborhood. P -> 167.

Median-filter: replace each pixel by the medial of the 3x3 neighborhood. P -> 159.

Automatic Repair Strategies

NW -- N - NE	158 154 147
W— P — E	169 27 159
SW— S -- SE	166 166 159

Mean-filter: replace each pixel by the mean of the 3x3 neighborhood. P -> 145.

Median-filter: replace each pixel by the medial of the 3x3 neighborhood. P -> 159.

Mean versus Median

The properties of the median guarantee that it will pick an extreme value, and won't be much influenced by one. If most of the pixel values are correct, and close, it will pick one of those.

The mean will always include the influence of an extreme value, and the result may be larger or smaller than all the correct pixel values.

Mean vs Median for 5 values

Data	Mean	Median
1,2,3,4,5	3	3
0,8,8,8,8	6.4	8
3,5,5,7,1000	204	5
0,1,2,2,255	52	2

gray_medianfilter.m

```
function I2 = gray_medianfilter ( I1 )

  [ m, n ] = size ( I1 );
  P = double ( I1 );
  P2 = P;
  for i = 2 : m - 1
    for j = 2 : n - 1
      P2(i,j) = median ( ...                    <- replace by "mean" to get mean filter.
        [ P(i+1,j-1), P(i+1,j), P(i+1,j+1), ...
          P(i,  j-1), P(i,  j), P(i,  j+1), ...
          P(i-1,j-1), P(i-1,j), P(i-1,j+1) ] );
    end
  end
  I2 = uint8 ( P2 );

  return
end
```

Mean vs Median