- Previous Lecture:
  - Image processing
    - Add frame, grayscale

- Today’s Lecture:
  - More image processing
    - Mirror, vectorized code
    - Color → grayscale, uint8
    - “Noise” filtering
    - (Read in book: Edge-finding example)

- Announcements:
  - Discussion via Zoom; see Canvas for link
  - Project 4 due Mon 4/13
  - Consulting resumes today via Zoom, hours extended
  - Be sure to review—re-do—Prelim 1 now so that you have a firm foundation
Where did we leave off?

How to put a picture in a frame

Two approaches:

1. Ask every pixel whether it is covered by the frame
   - Easy to understand
2. Identify which subarrays are covered by the frame
   - More efficient; easy to vectorize
Pictures as matrices

Pixel: an element in a matrix (location corresponds to row, column index)

“Greyness”: a value in 0..255
A color picture is made up of RGB matrices → 3D array

<table>
<thead>
<tr>
<th>R</th>
<th>G</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>114</td>
<td>114</td>
<td>112</td>
</tr>
<tr>
<td>114</td>
<td>113</td>
<td>111</td>
</tr>
<tr>
<td>115</td>
<td>114</td>
<td>112</td>
</tr>
<tr>
<td>116</td>
<td>117</td>
<td>116</td>
</tr>
<tr>
<td>113</td>
<td>112</td>
<td>112</td>
</tr>
<tr>
<td>115</td>
<td>115</td>
<td>115</td>
</tr>
<tr>
<td>112</td>
<td>113</td>
<td>116</td>
</tr>
<tr>
<td>115</td>
<td>116</td>
<td>118</td>
</tr>
<tr>
<td>116</td>
<td>116</td>
<td>117</td>
</tr>
</tbody>
</table>

E.g., color image data is stored in a 3-d array $A$:

$$0 \leq A(i,j,1) \leq 255$$

$$0 \leq A(i,j,2) \leq 255$$

$$0 \leq A(i,j,3) \leq 255$$
Visualize a 3D array as a stack of “layers” which are 2D arrays

Beware the two different “3”s:

- `dims = size(A)  % [720, 1280, 3]`
- `length(dims) == 3  % A has 3 dimensions: rows, columns, layers`
- `dims(3) == 3  % A has 3 layers: red, green, blue`
Example: Mirror Image

1. Read *LawSchool.jpg* from memory and convert it into an array.
2. Manipulate the Array.
3. Convert the array to a jpg file and write it to memory.
Reading and writing jpg files

```matlab
% Read jpg image, uncompres to a 3D array A of type uint8
A = imread('LawSchool.jpg');

% Write 3D array B to memory as a jpg image
imwrite(B,'LawSchoolMirror.jpg')
```
% Store mirror image of A in array B

[nr, nc, np] = size(A);
for r = 1:nr
    for c = 1:nc
        B(r, c) = A(r, nc-c+1);
    end
end
end
%Store mirror image of A in array B

[nr,nc,np]= size(A);

for r = 1:nr
    for c = 1:nc
        for p = 1:np
            B(r,c,p) = A(r,nc-c+1,p);
        end
    end
end
end
Both fragments create a mirror image of A.
% Make mirror image of A -- the whole thing

A = imread('LawSchool.jpg');
[nr,nc,np]= size(A);

for r= 1:nr
    for c= 1:nc
        for p= 1:np
            B(r,c,p)= A(r,nc-c+1,p);
        end
    end
end

imshow(B)  % Show 3-d array data as an image
imwrite(B,'LawSchoolMirror.jpg')
% Make mirror image of A -- the whole thing

A = imread('LawSchool.jpg');
[nr,nc,np] = size(A);

B = zeros(nr,nc,np);   % zeros returns type double
B = uint8(B);          % Convert B to type uint8

for r = 1:nr
    for c = 1:nc
        for p = 1:np
            B(r,c,p) = A(r,nc-c+1,p);
        end
    end
end

imshow(B)   % Show 3-d array data as an image
imwrite(B,'LawSchoolMirror.jpg')
Vectorized code simplifies things...
Work with a whole column at a time

<table>
<thead>
<tr>
<th>A</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Column c in B is column nc-c+1 in A
Consider a single matrix (just one layer)

\[
[nr,nc,np] = \text{size}(A);
\]

\[
\text{for } c = 1:\text{nc} \\
\quad B(\text{all rows},c) = A(\text{all rows},nc-c+1);
\]

\text{end}
Consider a single matrix (just one layer)

```matlab
[nr,nc,np] = size(A);
for c = 1:nc
    B(1:nr,c) = A(1:nr,nc-c+1);
end
```
Consider a single matrix (just one layer)

\[
[nr, nc, np] = \text{size}(A);
\]

\[
\text{for } c = 1 : nc \\
\quad B(:, c) = A(:, nc - c + 1);
\]

\textbf{end}

The colon says “all indices in this dimension.” In this case it says “all rows.”
Now repeat for all layers

```
[nr,nc,np] = size(A);
for c = 1:nc
    B(:,c,1) = A(:,nc-c+1,1)
    B(:,c,2) = A(:,nc-c+1,2)
    B(:,c,3) = A(:,nc-c+1,3)
end
```
Vectorized code to create a mirror image

```matlab
A = imread('LawSchool.jpg')
[nr,nc,np] = size(A);
for c = 1:nc
    B(:,c,1) = A(:,nc-c+1,1)
    B(:,c,2) = A(:,nc-c+1,2)
    B(:,c,3) = A(:,nc-c+1,3)
end
imwrite(B, 'LawSchoolMirror.jpg')
```
Even more compact vectorized code to create a mirror image...

```matlab
for c = 1:nc
    B(:,c,1) = A(:,nc-c+1,1)
    B(:,c,2) = A(:,nc-c+1,2)
    B(:,c,3) = A(:,nc-c+1,3)
end

B = A(:,nc:-1:1,:)```

Example: color $\rightarrow$ black and white

Can “average” the three color values to get one gray value.
Converting from color (RGB) to grayscale
Averaging the RGB values to get a gray value

\[ .21R + .72G + .07B \]

\[ \frac{R}{3} + \frac{G}{3} + \frac{B}{3} \]
Averaging the RGB values to get a gray value

\[
\text{for } i=1:m \\
\quad \text{for } j=1:n \\
\quad \quad M(i,j) = 0.21 \times R(i,j) + 0.72 \times G(i,j) + 0.07 \times B(i,j) \\
\text{end} \\
\text{end}
\]
Averaging the RGB values to get a gray value

for i = 1:m
    for j = 1:n
        \[ M(i,j) = 0.21A(i,j,1) + 0.72A(i,j,2) + 0.07A(i,j,3) \]
    end
end
Averaging the RGB values to get a gray value

for i = 1:m
  for j = 1:n
    M(i,j) = 0.21*A(i,j,1) + 0.72*A(i,j,2) + 0.07*A(i,j,3)
  end
end

M = 0.21*A(:, ,1) + 0.72*A(:, ,2) + 0.07*A(:, ,3)
Computing in type \texttt{uint8}

- Respect the range \([0..255]\)
- Arithmetic on \texttt{uint8}'s results in \texttt{uint8}'s
- **Saturation** (also called “capped”)
  - \texttt{uint8}(90) + \texttt{uint8}(200) \rightarrow 255 \ (\text{type \texttt{uint8}})
  - \texttt{uint8}(90) - \texttt{uint8}(200) \rightarrow \_\_\_ \ (\text{type \texttt{uint8}})
- **Rounding** (not truncation)
  - \texttt{uint8}(32)/\texttt{uint8}(3) \rightarrow \underline{11} \ (\text{type \texttt{uint8}})
- Arithmetic between a \texttt{uint8} and a \texttt{double} results in a \texttt{uint8}
  - \texttt{uint8}(90) + 200 \rightarrow \underline{255} \ (\text{type \texttt{uint8}})
Here are 2 ways to calculate the average. Are gray value matrices \( g \) and \( h \) the same given uint8 image data \( A \)?

\[
\begin{align*}
\text{for } r &= 1:nr \\
&\quad \text{for } c = 1:nc \\
&\quad \quad g(r,c) = \frac{A(r,c,1)}{3} + \frac{A(r,c,2)}{3} + \ldots + \frac{A(r,c,3)}{3}; \\
&\quad \quad h(r,c) = \ldots \\
&\quad \quad \quad \left( \frac{A(r,c,1)+A(r,c,2)+A(r,c,3)}{3} \right); \\
\end{align*}
\]

A: yes  
B: not quite (rounding)  
C: no (saturation)
Application: median filtering

How can we remove noise?
Dirty pixels look out-of-place

150 149 152 153 152 155
151 150 153 154 153 156
153  2  3 156 155 158
154  2  1 157 156 159
156 154 158 159 158 161
157 156 159 160 159 162
How to fix “bad” pixels?

- Visit each pixel
- Replace with typical values from its neighborhood
  - How to choose “typical” value?
  - How big is the neighborhood?
- “Typical”: mean vs. median
  - Median better for rejecting noise, preserving edges
- Neighborhood: moving window of radius $r$
Using a radius-1 neighborhood

Before

After
Top-down design

- Visit each pixel
- Choose a new gray value equal to the median of the old gray values in the “neighborhood”

```matlab
[nr,nc] = size(A); % A is 2d array of image data
B = uint8(zeros(nr,nc));
for i = 1:nr
    for j = 1:nc
        C = neighborhood of pixel (i,j)
        B(i,j) = median of elements in C
    end
end
```
Replace $\times$ with the median of the values under the window.
Original:

Filtered:

Replace \( \times \) with the median of the values under the window.
Original:

\[
i = 1
\]
\[
j = 3
\]

Filtered:

Replace \( \times \) with the median of the values under the window.
Replace $\times$ with the median of the values under the window.
Replace \( \square \) with the median of the values under the window.
Replace the value marked with the median of the values under the window.
Original:

\[ i = nr \]
\[ j = nc \]

Filtered:

Replace the median of the values under the window.
Details at a pixel \((i,j)\) with a radius 1 “neighborhood”

\[ \begin{array}{ccc}
7 & 7 & 6 \\
7 & 0 & 6 \\
7 & 6 & 6 \\
\end{array} \]

\[ \begin{array}{ccc}
7 & 7 & 6 \\
7 & 6 & 6 \\
7 & 6 & 6 \\
\end{array} \]

Replace pixel \((i,j)\)
with median value

\[ \frac{7 + 7 + 6 + 7 + 6 + 6 + 6 + 6 + 6}{9} = 6 \]

% Get median value in a matrix xMat
\texttt{xVec} = \texttt{xMat}(:) % Convert matrix to vector
\texttt{medianVal} = \texttt{median(xVec)} % Use built-in function
Deal with boundary issues – moving window

\[
\begin{align*}
\text{nr} \times \text{nc} \text{ matrix } A \\
\begin{array}{cccc}
1 & \ldots & \cdots & \text{nc} \\
\vdots & & & \vdots \\
\text{nr} & & & 1 \\
\end{array}
\end{align*}
\]

% Get C, the radius r
% neighborhood of pixel (i,j)

\[
i_{\text{Min}} = i - r \\
i_{\text{Max}} = i + r \\
j_{\text{Min}} = j - r \\
j_{\text{Max}} = j + r \\
C = A(i_{\text{Min}}:i_{\text{Max}},j_{\text{Min}}:j_{\text{Max}})
\]
Deal with boundary issues – moving window

% Get C, the radius r
% neighborhood of pixel (i,j)
iMin = max(1, i-r)
iMax = min(nr, i+r)
jMin = max(1, j-r)
jMax = min(nc, j+r)
C = A(iMin:iMax, jMin:jMax)

See Insight §12.4 for complete code: MedianFilter.m
B = medianFilter(A, 3)
Mean Filter with radius 3
Mean Filter with radius 10
Mean filter fails because the mean does not capture representative values.

Mean-filtered values with radius 1 neighborhood:

\[
\begin{array}{cccccccc}
150 & 149 & 152 & 153 & 152 & 155 \\
151 & 150 & 153 & 154 & 153 & 156 \\
153 & 2 & 3 & 156 & 155 & 158 \\
154 & 2 & 1 & 157 & 156 & 159 \\
156 & 154 & 158 & 159 & 158 & 161 \\
157 & 156 & 159 & 160 & 159 & 162 \\
\end{array}
\]

Median-filtered values with radius 1 neighborhood:

\[
\begin{array}{cccc}
85 & 86 \\
87 & 88 \\
150 & 150 \\
153 & 154 \\
\end{array}
\]
Finding Edges: read example in Sec 12.4

Identify “sharp changes” in image data—a kind of outliers.

Subtracting \texttt{uint8} values correctly to prevent “underflow”

“Thresholding”—use a parameter to control the amount of details extracted from image