Intel AI Academy

IMAGE PROCESSING

LEGAL NOTICES AND DISCLAIMERS

APPLICATIONS OF IMAGE PROCESSING

Processing Type

__Enhancement__
Example: Changing saturation, contrast to enhance look

__Understanding__
Example: Object detection, image classification to understand image content

__Perturbation__
Example: Blurring, rotating images to produce desired effect

Example Applications

- Marketing
- Photography
- Self-driving cars
- Security

- Censoring
- Animations

IMAGE PROCESSING TOOLS

IDEs (Integrated Development Environments)	Jupyter Notebook*†		Matlab*
Languages	Python*†	C++	
Libraries	NumPy†	OpenCV†	

† = Taught in this course

Focus of this lesson

COMPUTER ARITHMETIC WITH NUMPY

If you do arithmetic with NumPy, overflow can occur:

Here, uint8 stores a max of 256: 300 is rounded down to 300 - 256 = 44.

To avoid this, we have to promote to a bigger type than our expected result, do arithmetic, then convert back to smaller type:

```
np.uint8([100]) + np.uint8([200]), 300 - 2**8, 300 % 2**8
```

```
(array([44], dtype=uint8), 44, 44)
```

Here, we promote uint8 to uint16, do arithmetic, convert back into uint8.

```
arr1, arr2 = np.uint8([100]), np.uint8([200])
np.clip(arr1.astype(np.uint16) + arr2.astype(np.uint16), 0, 255)
```

```
array([255], dtype=uint16)
```

COMPUTER ARITHMETIC WITH OPENCV

Or

- Can use openCV arithmetic, which handles overflow (numbers being added to quantities greater than 255), and clipping (automatically reducing numbers greater than 255 down to 255) for us.
- With OpenCV, adding 100+200 with a max of 255 gets rounded down to 255, as expected.

```
cv2.add(np.uint8([100]), np.uint8([200]))
```

```
array([[255]], dtype=uint8)
```

INTERPOLATION

Resizing an image requires reassigning pixels from the original image to the new image space.

Matplotlib* and OpenCV have the same interpolation methods.

The difference: Matplotlib applies interpolation at display time. OpenCV applies interpolation prior to displaying the images.

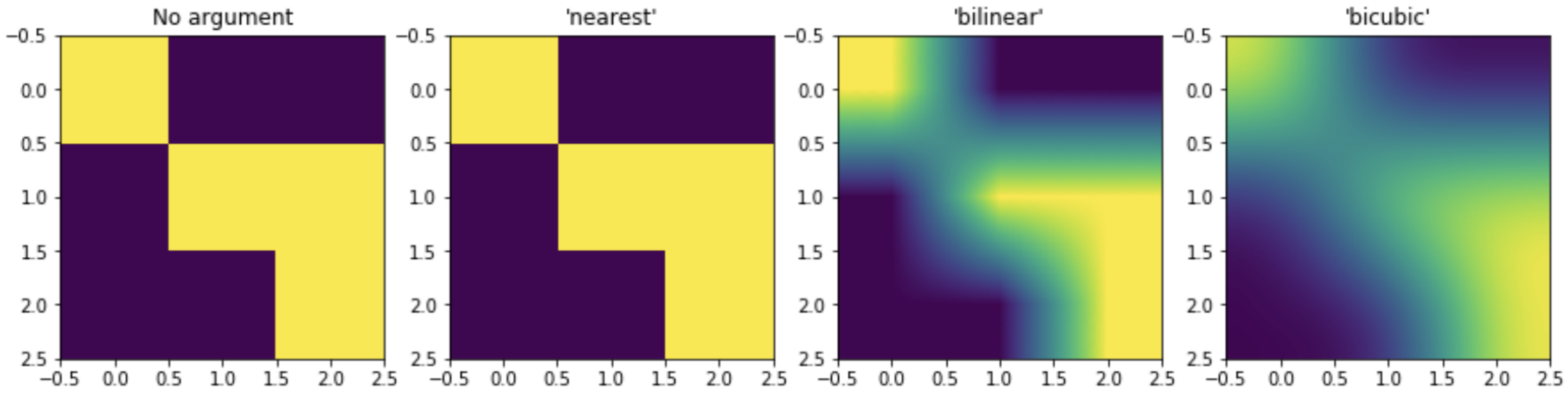
Interpolation is used to address forward projection problems. These problems are defined as:

1. Stretching the size of an image creates locations in the new image where pixels do not have values.
2. Shrinking the size of an image creates float pixel values that need to be assigned to an integer pixel value in the new image.

INTERPOLATION METHODS: IMAGES

```
img = np.eye(3)
img[1,2] = 1

print(img)

fig, axes = plt.subplots(1, 4, figsize=(15, 5))
axes[0].imshow(img)
axes[0].set_title("No argument")
methods = ['nearest', 'bilinear', 'bicubic']
for idx, method in enumerate(methods, 1):
    axes[idx].imshow(img, interpolation=method)
    axes[idx].set_title(f'{repr(method)}')
```

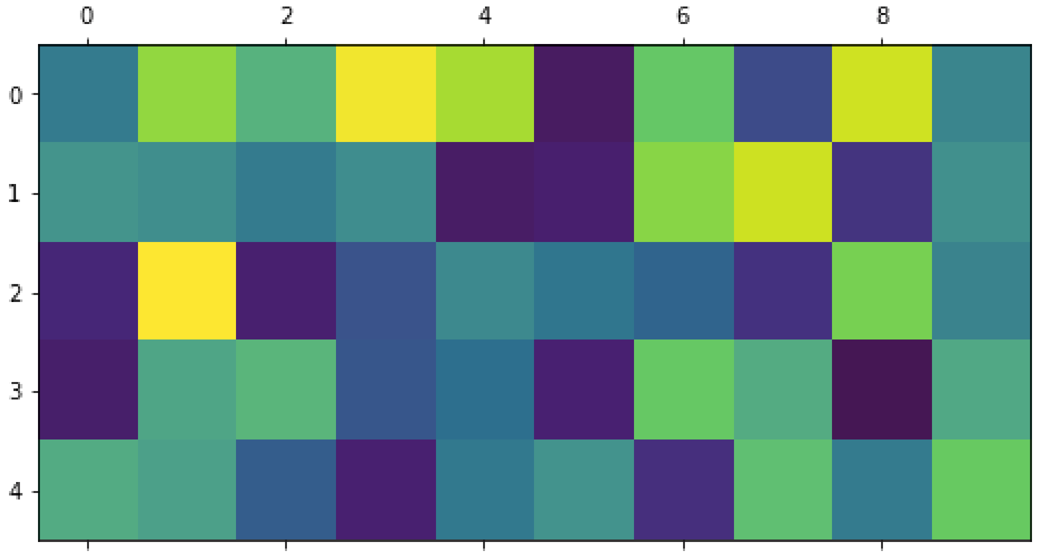
```
[[1. 0. 0.]
 [0. 1. 1.]
 [0. 0. 1.]]
```



NEAREST NEIGHBOR (CV2.INTER.NN)

```
img = np.random.randint(0, 256, size=(5, 10))
print(img)
# imshow with: origin='upper',
#              interpolation='nearest'
#              aspect='equal'
plt.matshow(img);
```

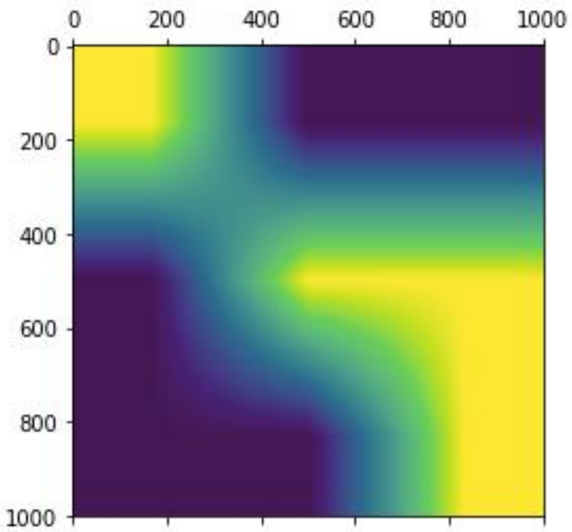
```
[[104 210 161 245 218  14 185  60 232 116]
 [130 126 106 124  16  23 206 231  41 127]
 [ 31 250  24  67 120 101  84  40 200 114]
 [ 20 148 166  70  94  26 187 155   4 151]
 [154 143  77  25 103 129  38 176 105 189]]
```



LINEAR (CV2.INTER_LINEAR)

```
img = np.eye(3)
img[1,2] = 1

print(img)

out = cv2.resize(img, (1000,1000), interpolation = cv2.INTER_LINEAR)
print(out.shape)
plt.matshow(out, interpolation=None)
```

```
[[ 1.  0.  0.]
 [ 0.  1.  1.]
 [ 0.  0.  1.]]
(1000, 1000)

<matplotlib.image.AxesImage at 0x11c65eb70>
```



INTERPOLATION METHODS: DATA

```
img = np.eye(3)
for meth in [cv2.INTER_LINEAR,
             cv2.INTER_NEAREST,
             cv2.INTER_CUBIC]:
    print(cv2.resize(img, (4,4), interpolation = meth))
#cv2.resize(img, (4,4), interpolation = )
#cv2.resize(img, (4,4), interpolation = cv2.INTER_CUBIC)
```

```
[[ 1.          0.375       0.          0.        ]
 [ 0.375       0.53125     0.390625    0.        ]
 [ 0.          0.390625    0.53125     0.375     ]
 [ 0.          0.          0.375       1.        ]]
[[ 1.  1.  0.  0.]
 [ 1.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
[[ 1.15385866  0.33241868 -0.17154694  0.00515199]
 [ 0.33241868  0.70369101  0.48258901 -0.17154694]
 [-0.17154694  0.48258901  0.70369101  0.33241868]
 [ 0.00515199 -0.17154694  0.33241868  1.15385866]]
```

OTHER INTERPOLATION IN OPENCV

spline16	quadric	laczos	hermite	kaiser
mitchell	sinc	hamming	bessel	hanning
guassian	spline36	catron	area	

You can play with these, but we will not get into them.

DISPLAY WITH MATPLOTLIB*: IMSHOW

Imshow arguments

- **(m,n):**
  Rows, columns: NOT x,y coordinates

Can (m,n,3) and (m,n,4)

Values uint8 or float in [0,1]

(m,n,3) = RGB

(m,n,4) = RGBA

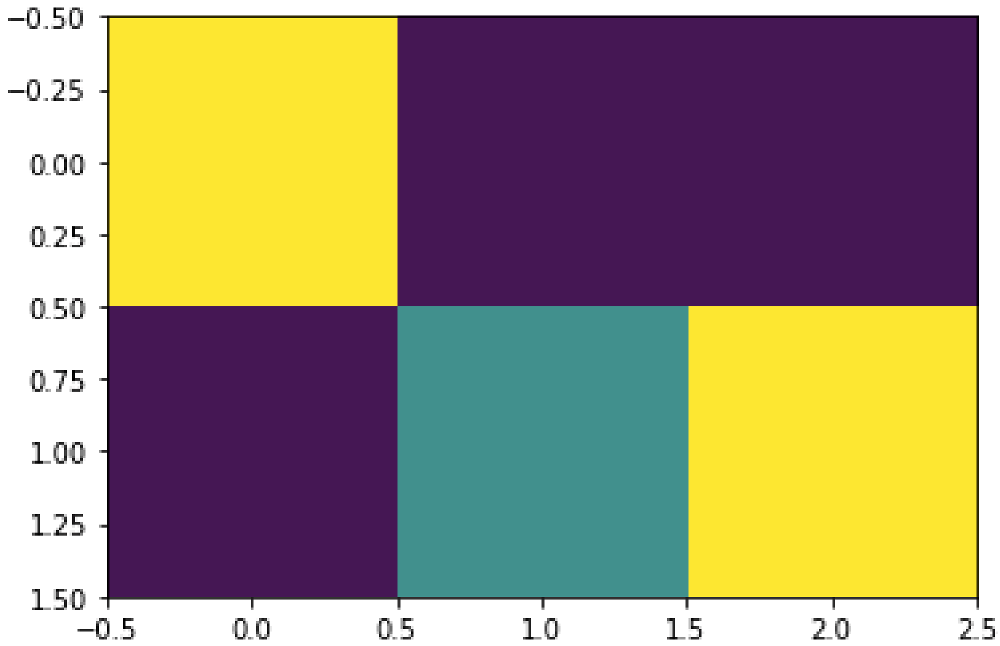
A → alpha

- **"cmap"**
  Color map: can be RGB, grayscale, and so on
  See upcoming slide for examples

- **"vmin", "vmax"**
  Used to change brightness

```
R,G,B = np.eye(3)*255

img = np.array([[R, G, B],
                [(0*(R+G+B)), (.5*(R+G+B)), (R+G+B)]])
img.min(), img.max()
plt.imshow(img[:,:,0])
```

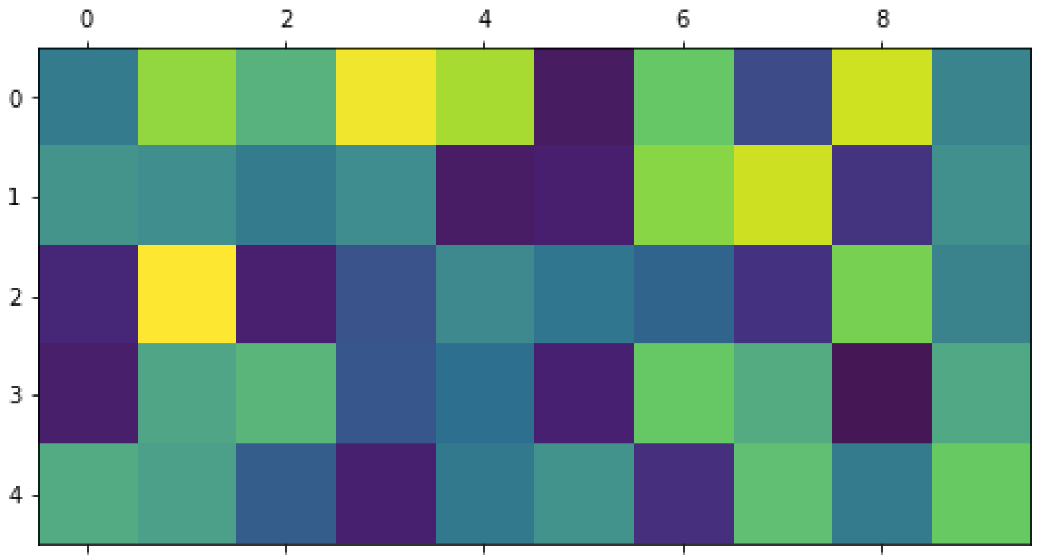
```
<matplotlib.image.AxesImage at 0x121088668>
```



DISPLAY OF 2D ARRAYS WITH MATPLOTLIB*: MATSHOW

```
img = np.random.randint(0, 256, size=(5, 10))
print(img)
# imshow with: origin='upper',
#              interpolation='nearest'
#              aspect='equal'
plt.matshow(img);
```

```
[[104 210 161 245 218  14 185  60 232 116]
 [130 126 106 124  16  23 206 231  41 127]
 [ 31 250  24  67 120 101  84  40 200 114]
 [ 20 148 166  70  94  26 187 155   4 151]
 [154 143  77  25 103 129  38 176 105 189]]
```



DISPLAY WITH MATPLOTLIB*: VMIN AND VMAX

vmin (minimum brightness) defaults to min data

vmax (maximum brightness) defaults to max of data

Scalar, optional, default: None

Used to normalize

Tip: To prevent scaling (0,255) for a restricted range (say 100,200), pass in vmin=0, vmax=255.

DISPLAY WITH MATPLOTLIB*: INTERPOLATION

Interpolation = 'none' and *interpolation = 'nearest'*

- Are equivalent when converting a figure to an image file (that is, PNG)

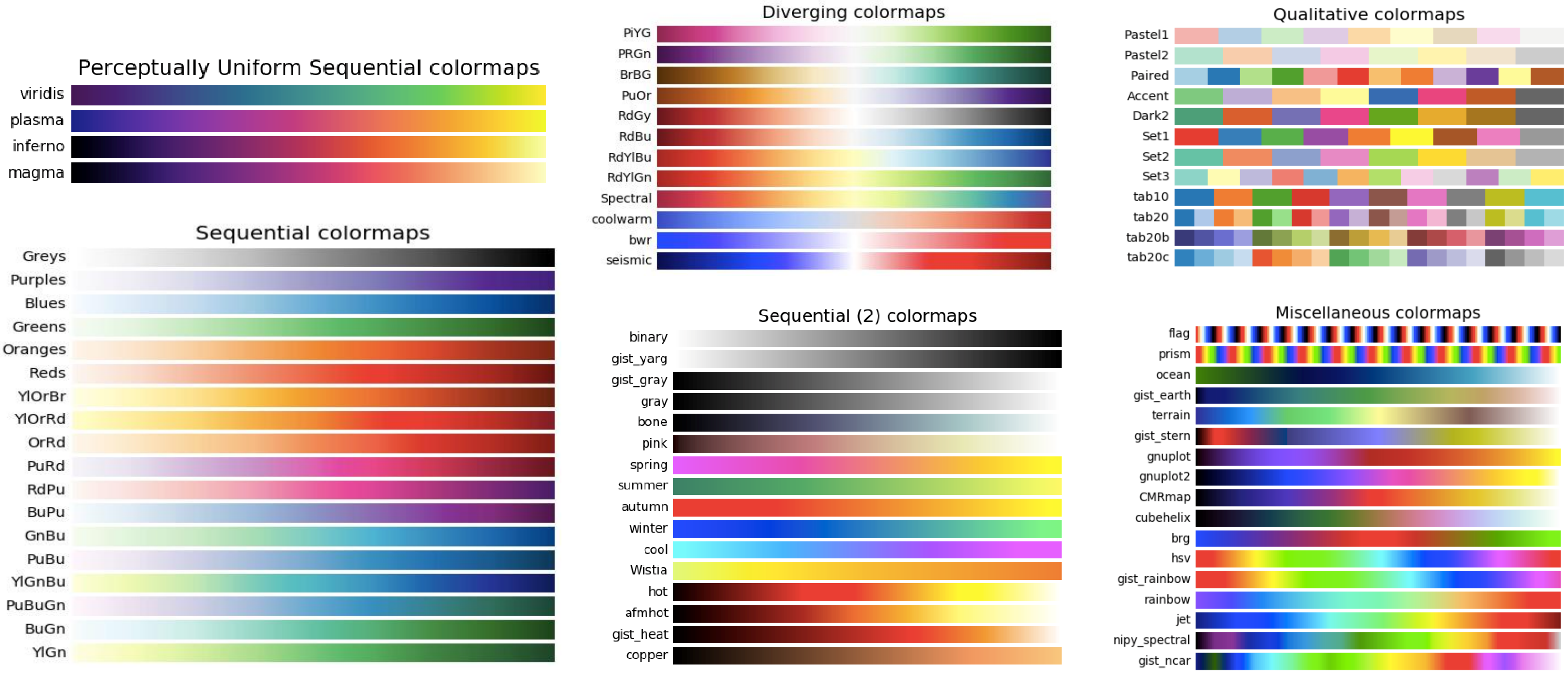
Interpolation = 'none' and *interpolation = 'nearest'*

- Behave quite differently when converting a figure to a vector graphics file (that is, PDF)

In our examples, *interpolation = 'none'* works well when a big image is scaled down.

While *interpolation = 'nearest'* works well when a small image is scaled up.

DISPLAY WITH MATPLOTLIB*: CMAP



COLOR SPACES

COLOR SPACES: INTRODUCTION

All images are represented by pixels with values in color spaces. Must represent an image in a color space before applying any processing techniques.

Examples of color spaces:

- Grayscale (black and white)
- Color
  - RGB (red, green, blue)
  - HSV (hue, saturation, value)
  - CMYK (cyan, magenta, yellow, black)

COLOR SPACES: GRAYSCALE

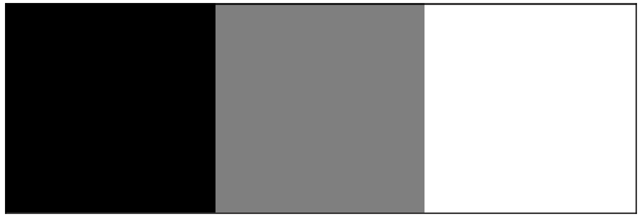
Grayscale to RGB conversion assumes all components of images are equal.

RGB to grayscale conversion uses

$$Y = (0.299)R + (0.587)G + (0.114)B$$

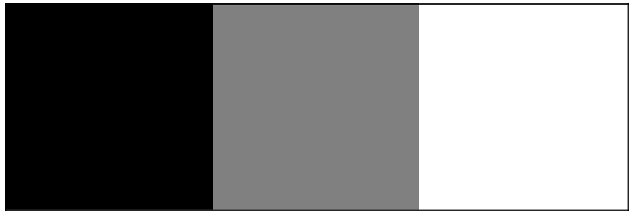
```
grays = np.array([[0, 127, 255]])
print(grays)
plt.tick_params(labelleft='off', labelbottom='off', left='off', bottom='off')
plt.gca().imshow(grays, cmap='gray');
```

```
[[  0 127 255]]
```



```
grays = np.array([[0, .5, 1.0]])
print(grays)
plt.tick_params(labelleft='off', labelbottom='off', left='off', bottom='off')
plt.gca().imshow(grays, cmap='gray');
```

```
[[ 0.   0.5  1. ]]
```



COLOR SPACES: RGB (RED, GREEN, BLUE)

RGB (red, green, blue)

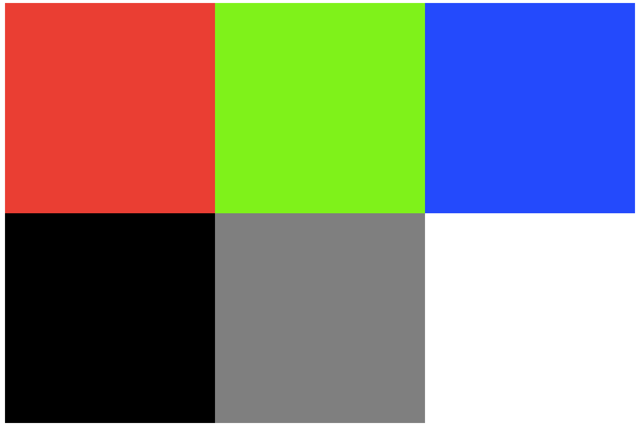
8-bit images 0-255

16-bit images 0-65,536

Floating point 0.0-1.0

```
R,G,B = np.eye(3)*255

img = np.array([[R, G, B],
               [(0*(R+G+B)), (.5*(R+G+B)), (R+G+B)]], dtype=np.uint8)

for idx, c in enumerate(['R', 'G', 'B']):
    print(c, '\n', img[:,:,idx])

my_show(plt.gca(), img, interpolation=None)
```

```
R
 [[255   0   0]
 [  0 127 255]]
G
 [[  0 255   0]
 [  0 127 255]]
B
 [[  0   0 255]
 [  0 127 255]]
```



COLOR SPACES: HSV (HUE, SATURATION, VALUE)

Hue is normally 0-360

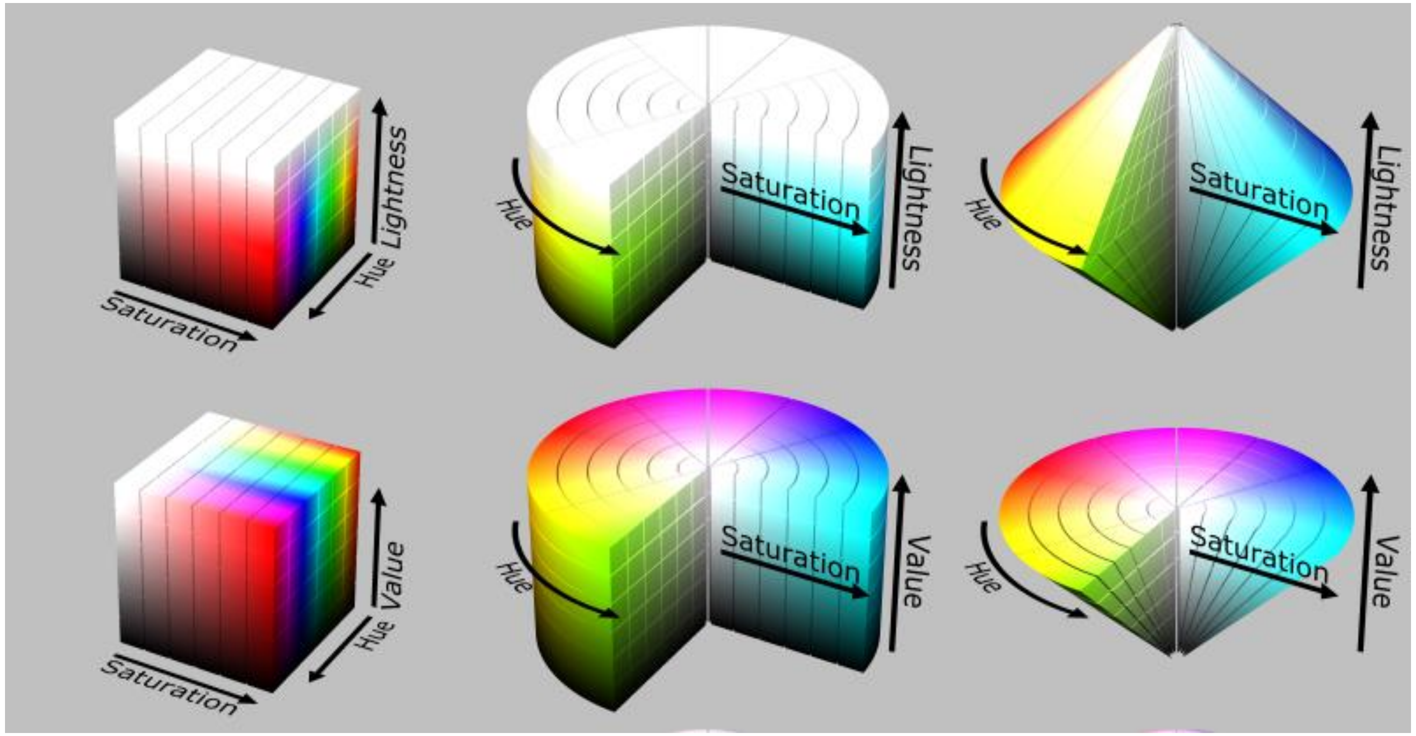
- Hue is walking around a color wheel
- OpenCV uses 0-179 to fit in 8 bit

Saturation (s) and value (v) are ranges of light to dark (intensity)

- Trade off black/white
- OpenCV uses 0-255 for both s and v

In cone models

- s is white; distance from cone axis
- v is black; distance from cone point



COLOR CONVERSION IN OPENCV

RGB is 8- or 16-bit colors.

Each pixel is represented by three values (one for each channel).

These values are pixel intensities

000 = black

111 = white

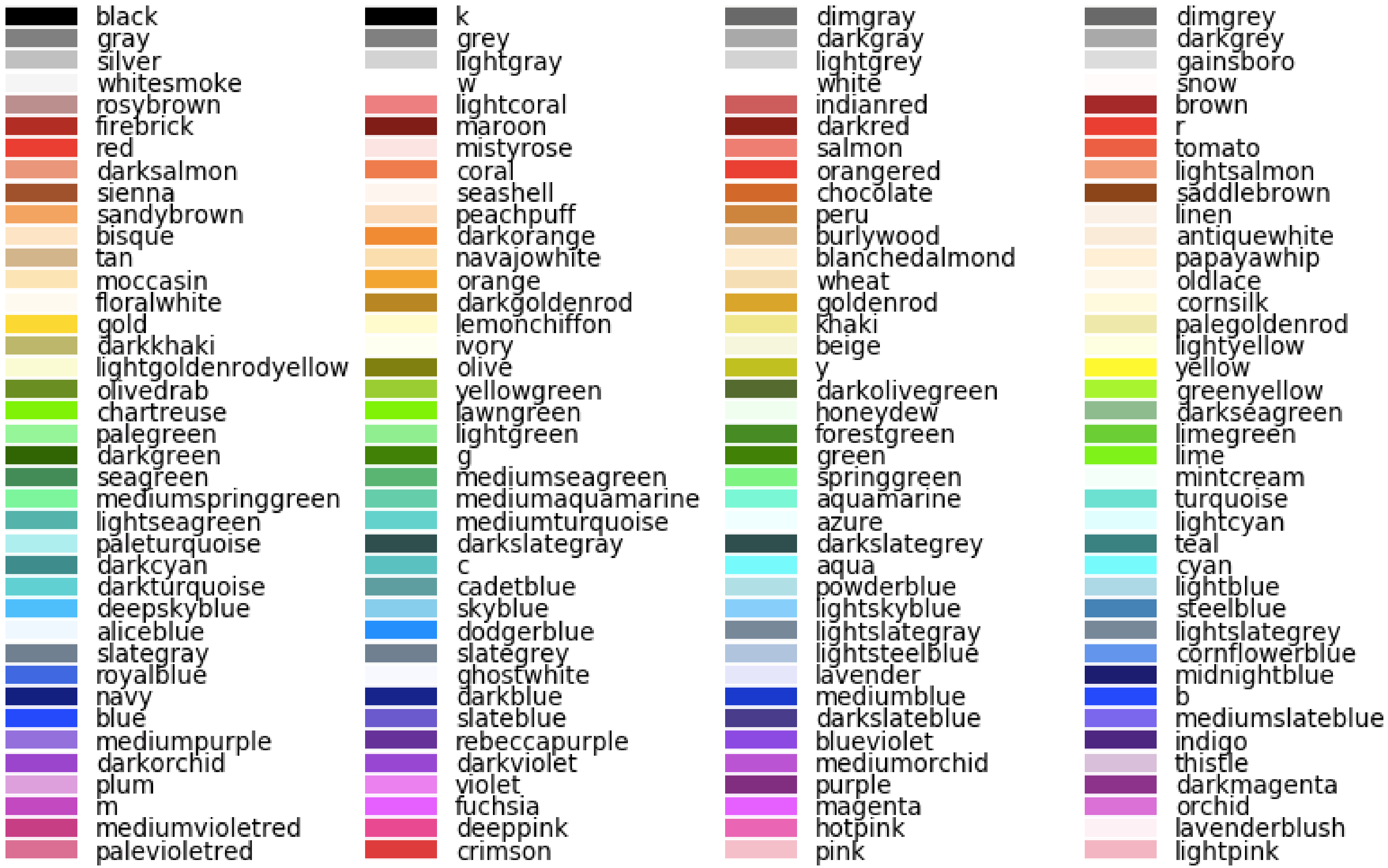
COLOR CONVERSION IN OPENCV

When converted to grayscale, all RGB are equal.

When converted from RGB (or BGR) to grayscale:

$$Y = (0.299)R + (0.587)G + (0.114)B$$

COLOR IN MATPLOTLIB*



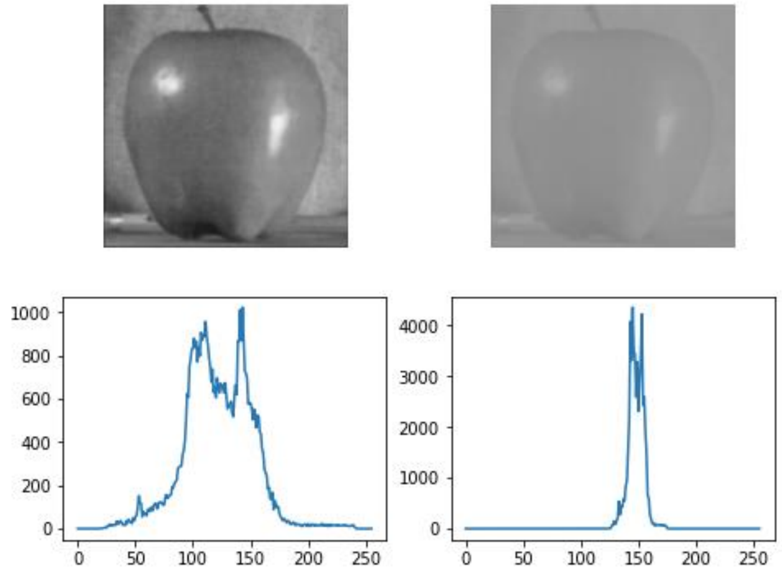
HISTOGRAMS

HISTOGRAMS: INTRODUCTION AND GRAYSCALE EXAMPLE

Histograms are a way of showing the intensity of color across an image.

Abnormal histograms can be used to detect poor image exposure, saturation, and contrast.

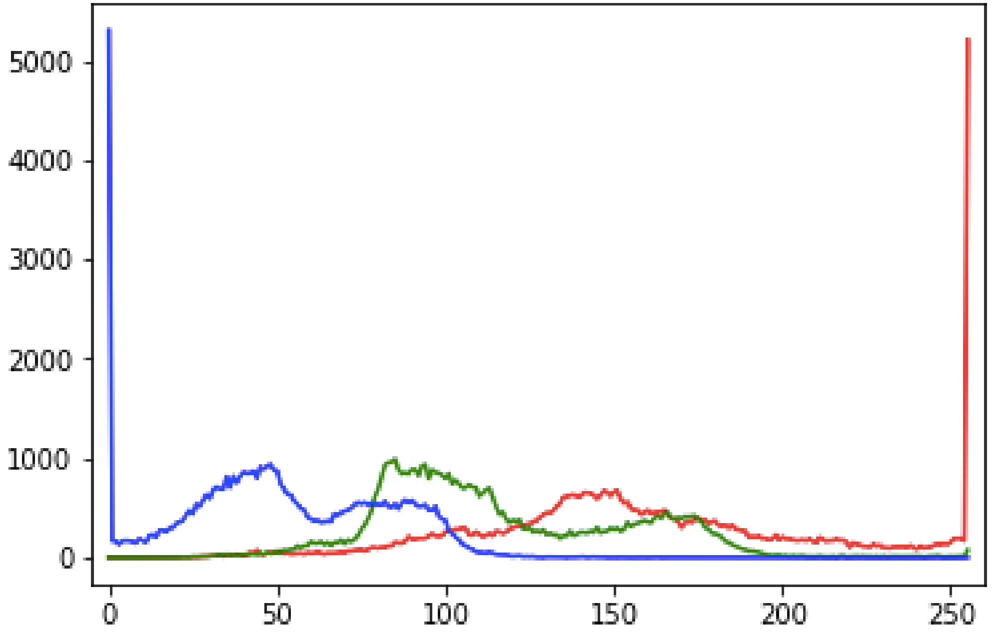
<u>Example:</u>



The image on the left has poor exposure, which can be detected via the histogram.

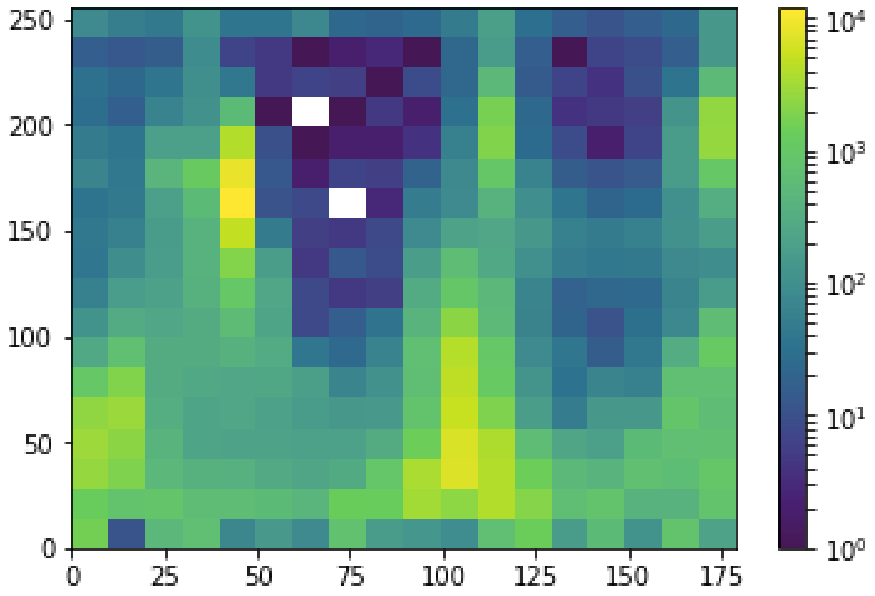
HISTOGRAM EXAMPLES: RGB

```python
for idx, color in enumerate(['r', 'g', 'b']):
    hist = cv2.calcHist([apple],[idx],None,[256],[0,256])
    plt.plot(hist, color = color)
    plt.xlim([-5,260])

# lots of red.  not too surprising.
```

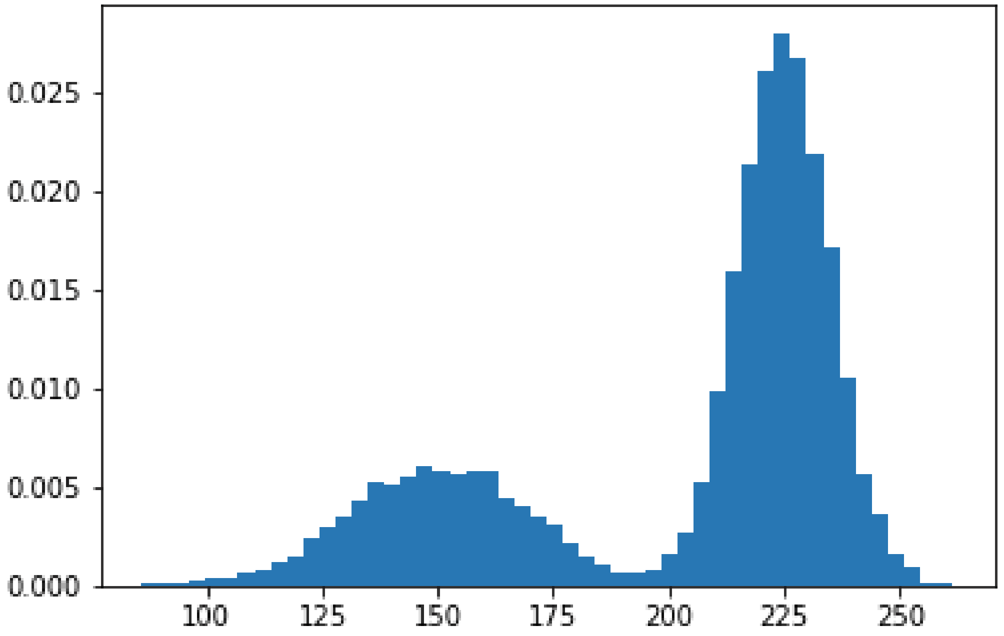


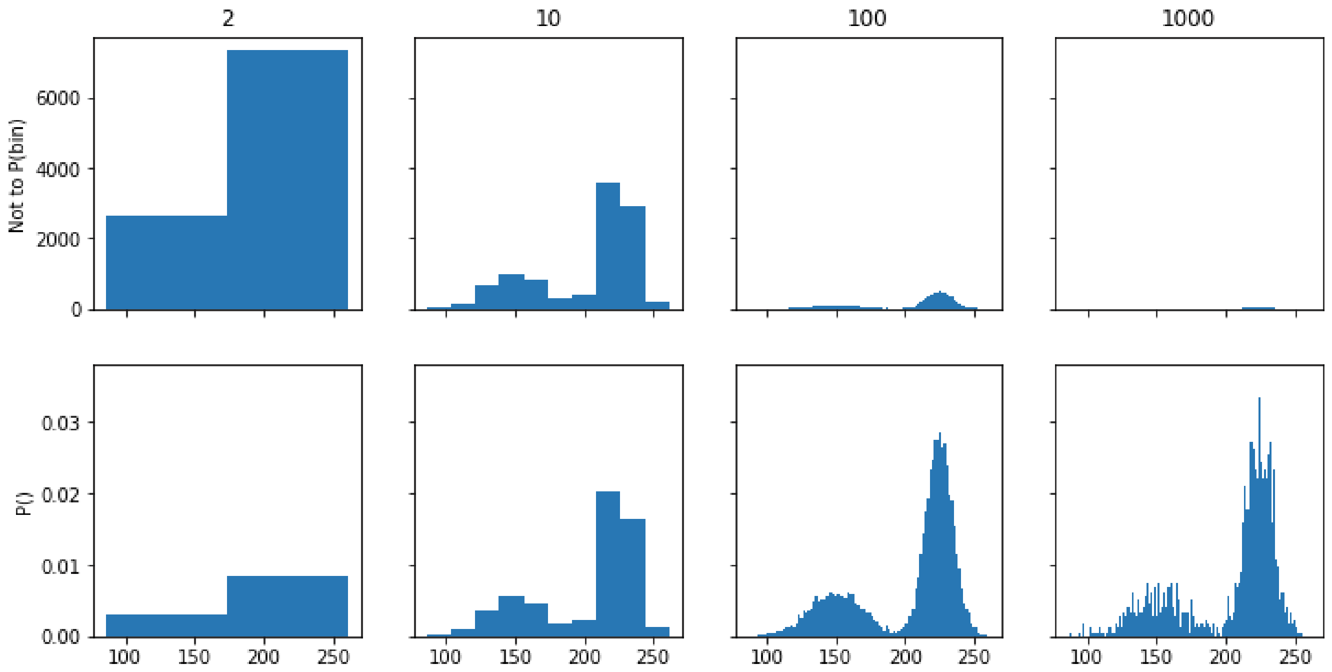
2D HISTOGRAMS

```
from matplotlib.colors import LogNorm as mpl_LogNorm

h,s,v = cv2.split(messi_hsv)
#_,_,_,cb_img = plt.hist2d(h.ravel(),s.ravel(),18)
_,_,_,cb_img = plt.hist2d(h.ravel(),s.ravel(),18, norm=mpl_LogNorm());
plt.colorbar(cb_img);
```



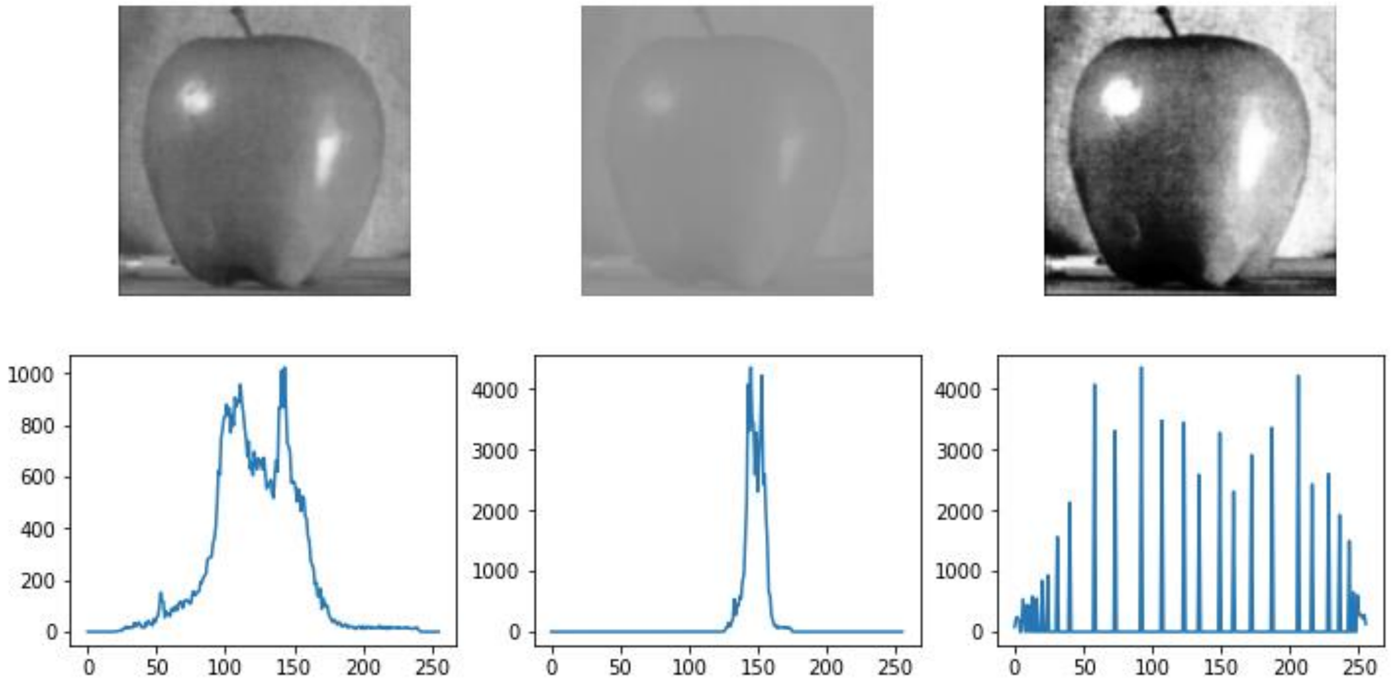
HISTOGRAMS: BIN WIDTHS





HISTOGRAM EQUALIZATION

Histogram equalization: Technique to adjust contrast to evenly spread y-values from an original distribution to a new distribution.

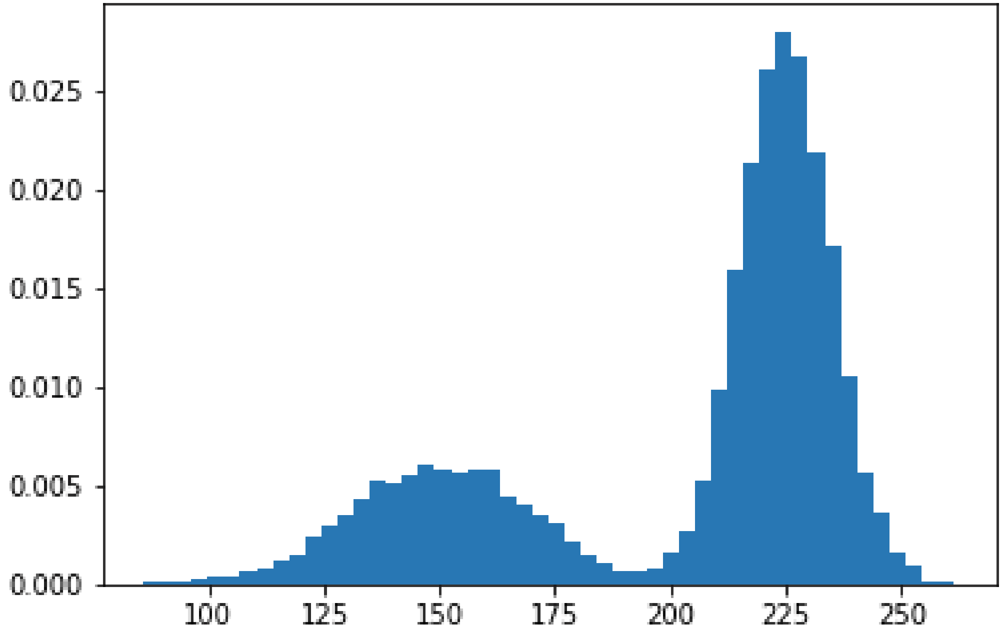


Usually increases global contrast of an image.

HISTOGRAM EQUALIZATION: PROBABILITY INTEGRAL TRANSFORM

Recall our original distribution:

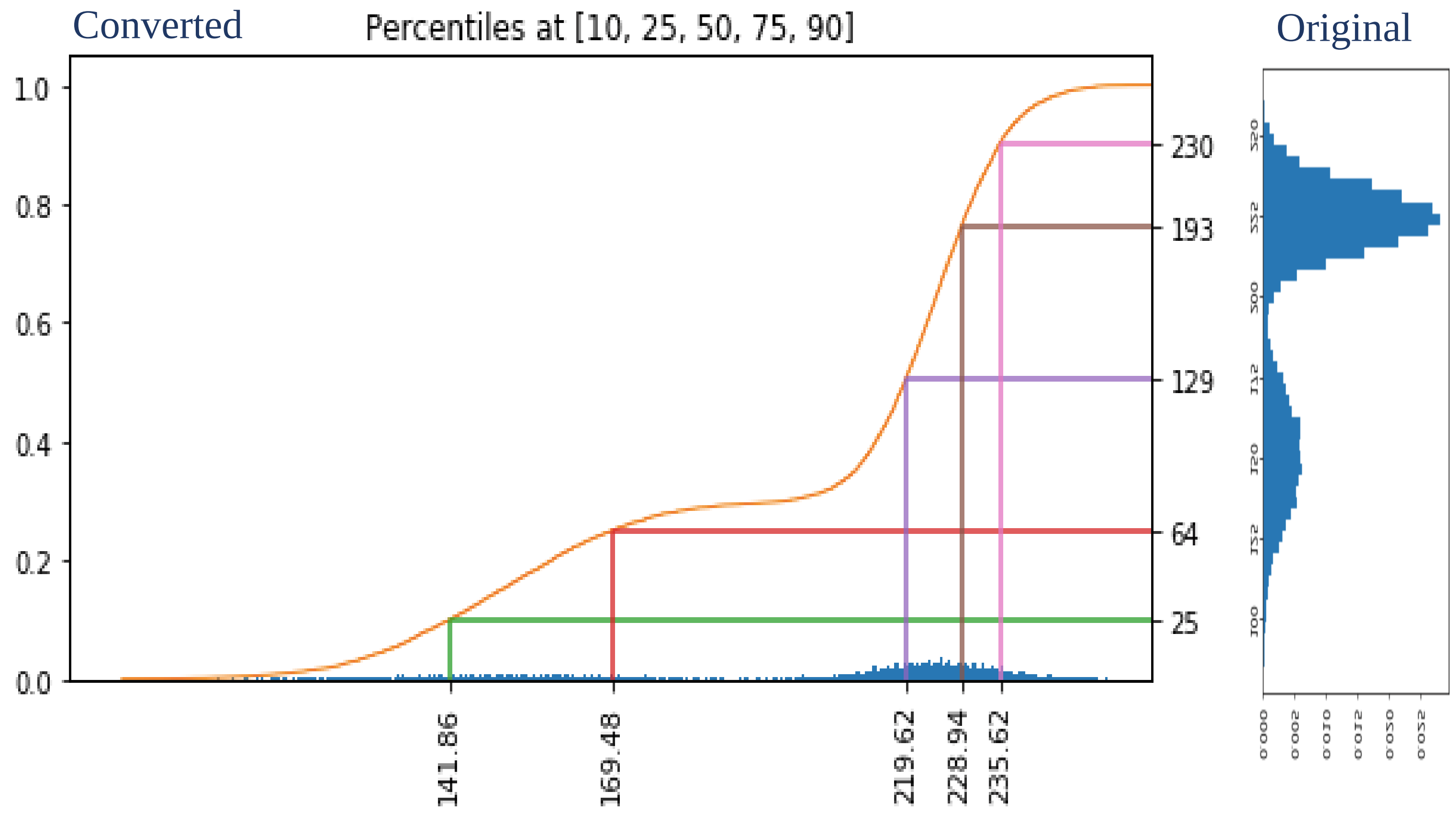
- Bimodal peaks
- Want to flatten this effect to get a uniform distribution



Probability integral transform allows us to convert an arbitrary distribution to a uniform distribution

HISTOGRAM EQUALIZATION: PROBABILITY INTEGRAL TRANSFORM

Covert an event from an arbitrary to a uniform distribution.



HISTOGRAM EQUALIZATION: PROBABILITY INTEGRAL TRANSFORM

Our original is a distribution of pixel intensities.

We will use percentiles from both the arbitrary distribution and the uniform distribution to do this conversion.

We need to know, for our input intensity x, what percentile it falls in to.

We compute the cumulative (running total) of the probabilities of the values less than x.

- Once we know that, we have an output value between [0,1] or U(0,1)

We can map U(0,1) to U(0,255) to get uniformly scaled pixel intensities.

- Simple linear scaling for (0,1) to (0, 255)

THRESHOLDING

FOREGROUND/BACKGROUND

Only part of the image has *important* data; we call these regions of interest (ROI).

In this case, we are interested in the foreground, but NOT the background of the logo image.

THRESHOLDING APPLICATION

In the case when the background has distinctively different color than the foreground, we can use thresholding to separate them.

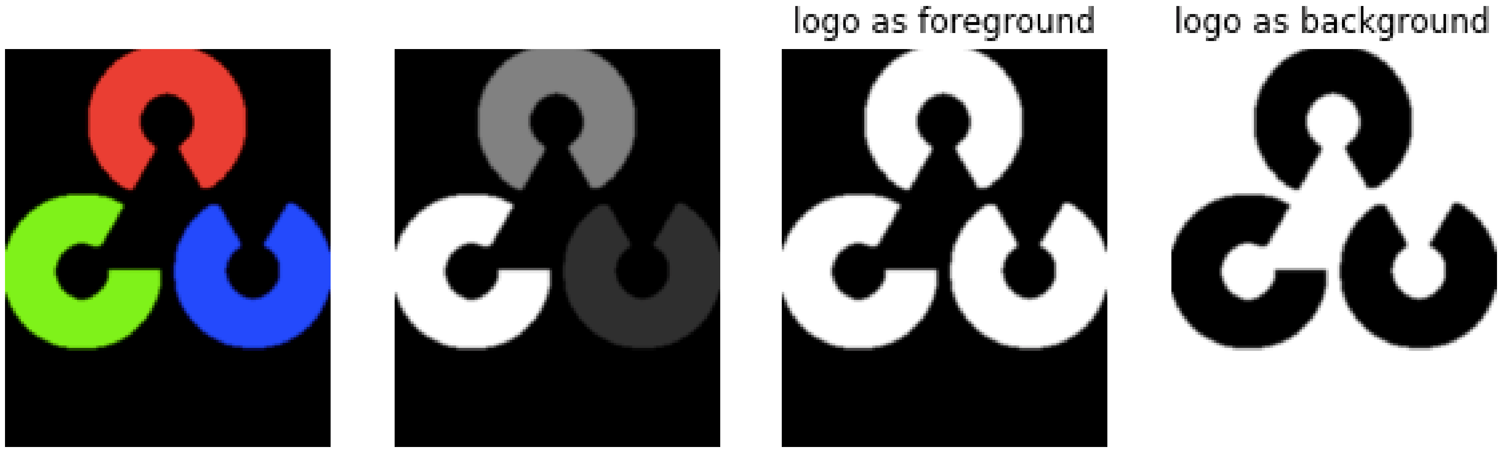
Example application:

The Application of Threshold Methods for Image Segmentation in Oasis Vegetation Extraction

Image source: https://upload.wikimedia.org/wikipedia/commons/2/23/Gfp-texas-big-bend-national-park-plants-on-the-desert-horizon.jpg

OPENCV LOGO EXAMPLE

```
logo_gray = cv2.cvtColor(logo,cv2.COLOR_RGB2GRAY)
ret, mask = cv2.threshold(logo_gray, 10, 255, cv2.THRESH_BINARY)
mask_inv  = cv2.bitwise_not(mask)

fig, axes = plt.subplots(1,4,figsize=(10,4))

my_show(axes[0], logo)
my_gshow(axes[1], logo_gray)
my_gshow(axes[2], mask, title='logo as foreground')
my_gshow(axes[3], mask_inv, title='logo as background')
```



THRESHOLDING: FOREGROUND VERSUS BACKGROUND

We need to build and interpret an image as foreground and background.

We need to make a *final* decision about points.

We will create a mask to denote regions of interest (foreground versus background).

THRESHOLDING: SIMPLE METHODS

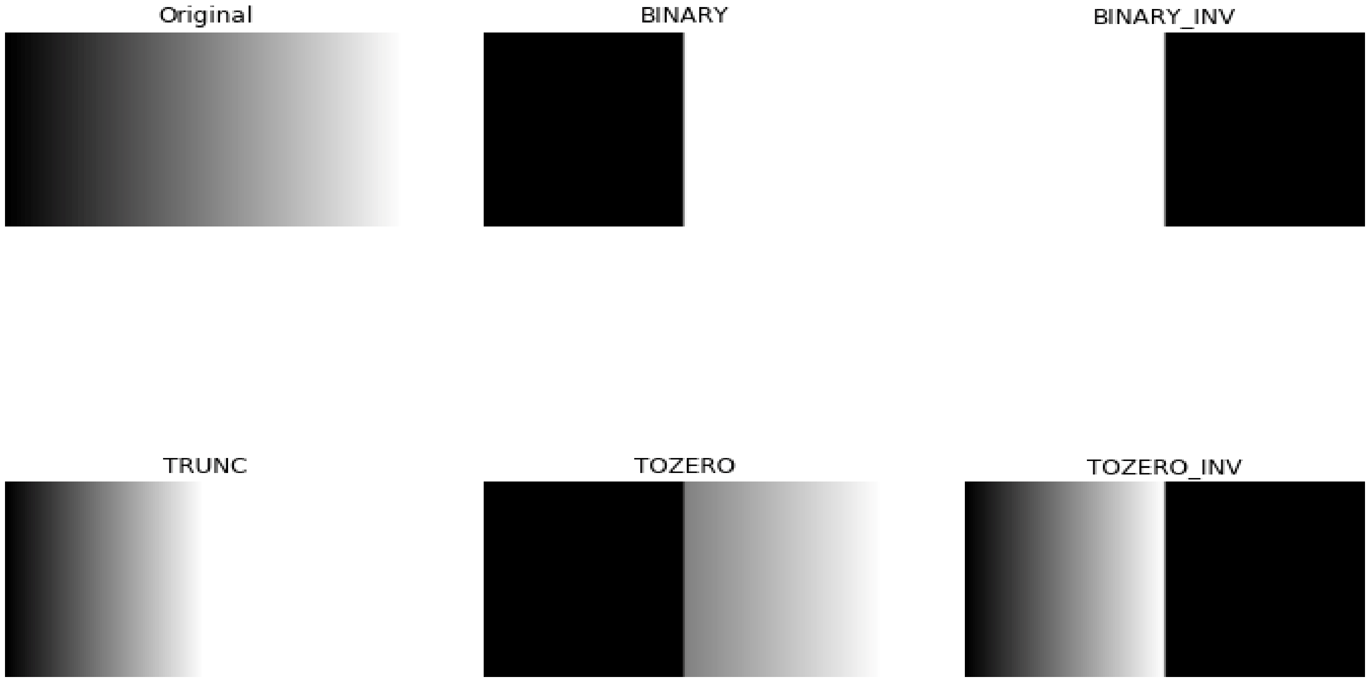
cv2.threshold( ) methods:

```
methods = [cv2.THRESH_BINARY, cv2.THRESH_BINARY_INV, cv2.THRESH_TRUNC,
           cv2.THRESH_TOZERO, cv2.THRESH_TOZERO_INV]
titles = ['BINARY','BINARY_INV','TRUNC','TOZERO','TOZERO_INV']
```

These methods are global / independent of neighbors.

THRESHOLDING: SIMPLE METHODS

```
fig, axes = plt.subplots(2,3,figsize=(12,9))
axes = axes.flat

ax = next(axes)
my_show(ax, gradient, cmap='gray')
ax.set_title("Original")

for m, t, ax in zip(methods, titles, axes):
    # src, midpoint, > map to, method
    _, res = cv2.threshold(gradient,127,255,m)
    my_show(ax, res, cmap='gray')
    ax.set_title(t)
```



THRESHOLDING: NEIGHBORHOOD METHODS

cvAdaptiveThreshold( ) is used when there is uneven illumination.

To calculate threshold for neighborhood:

- Calculate neighborhood means, using equal weights or weighted-averages based on Gaussian windows/filters.
- Subtract neighborhood mean from each point in the neighborhood.

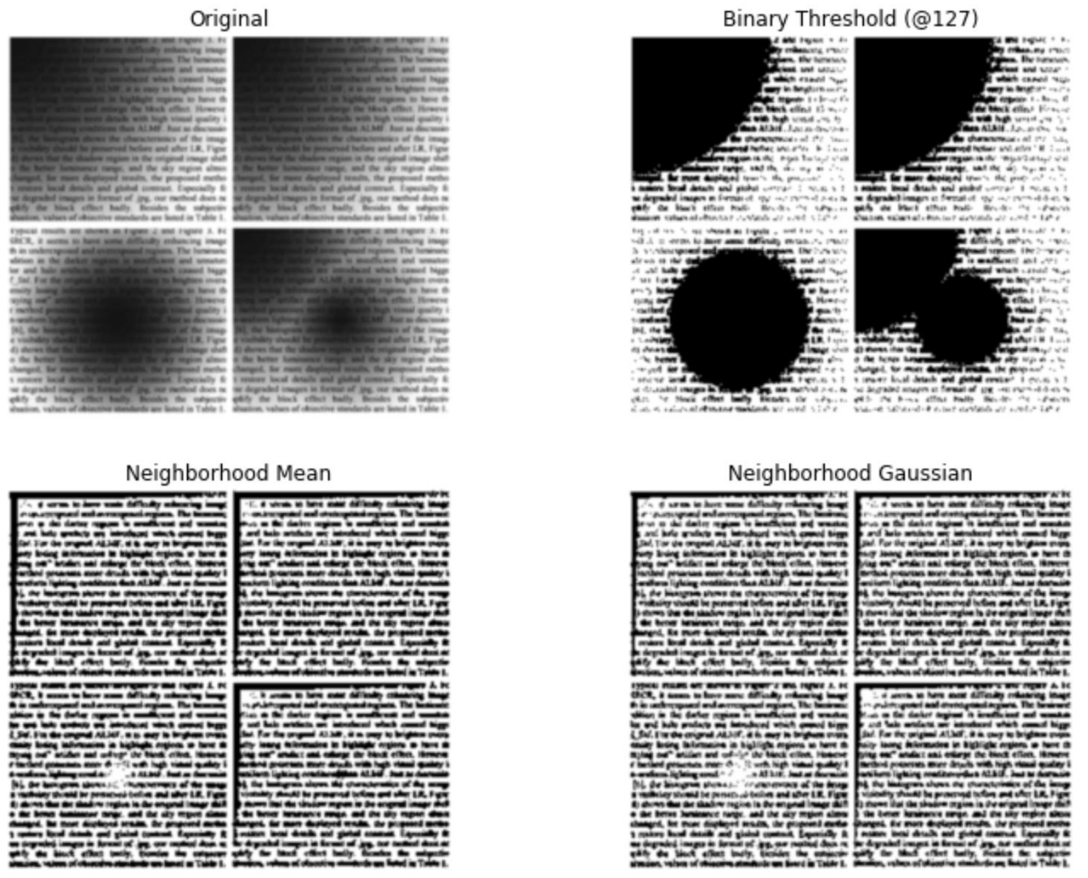
THRESHOLDING: NEIGHBORHOOD METHODS

Code to plot neighborhood thresholding:

```
fig, axes = plt.subplots(2,2,figsize=(12,9))
axes = axes.flat

uneven = cv2.imread(img_dir+'data/uneven-illumination.png',0)
ax = next(axes)
my_show(ax, uneven, cmap='gray')
ax.set_title("Original")

_, bin_th = cv2.threshold(uneven,127,255,cv2.THRESH_BINARY)
ax = next(axes)
my_show(ax, bin_th, cmap='gray')
ax.set_title("Binary Threshold (@127)")

methods = [cv2.ADAPTIVE_THRESH_MEAN_C, cv2.ADAPTIVE_THRESH_GAUSSIAN_C]
titles  = ["Neighborhood Mean", "Neighborhood Gaussian"]

for m, t, ax in zip(methods, titles, axes):
    # src, midpoint, > map to, method
    res = cv2.adaptiveThreshold(uneven,255,m,
                                cv2.THRESH_BINARY, # or BIN_INV
                                11, # neighborhood size (11x11),
                                2) # value subtracted from mean/gauss sum before comparison
    my_show(ax, res, cmap='gray')
    ax.set_title(t)
```

THRESHOLDING: NEIGHBORHOOD METHODS

Neighborhood methods illustration:



THRESHOLDING: OTSU'S METHOD

In Otsu's method, we don't *know* the classes:

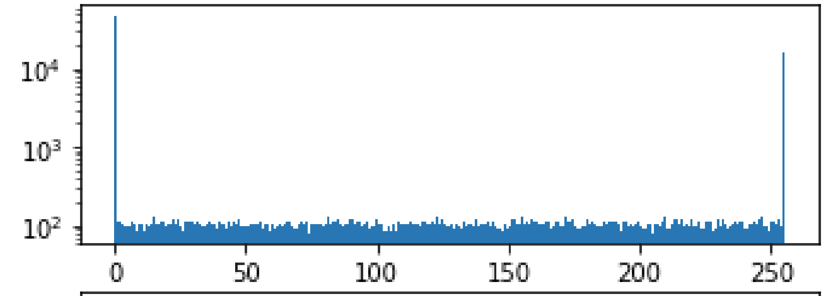
- We try every break point (for example, say 127, or an empirical mean or median) and assume it is the separator.
- The data is in two classes and we can compute the ratio of the *within* class variance and the *between* class variance.
- Only one break point *maximizes* the ratio of within class variance to between class variance.

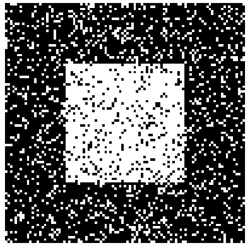
  This is our desired threshold.

Generally, Otsu works well with a large region of interest (compared to background) and a strong bimodal distribution.

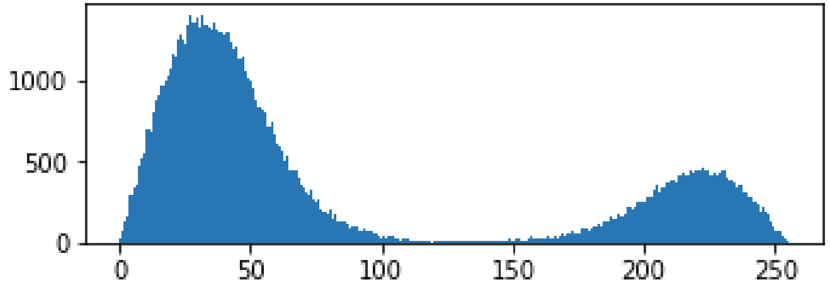
THRESHOLDING: OTSU'S METHOD

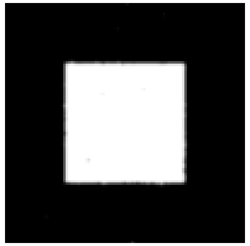
Optimal threshold unclear
(OpenCV finds 116)





Optimal threshold = 126





GEOMETRIC TRANSFORMS

GEOMETRIC TRANSFORMS: RESIZING

$x,y \rightarrow \quad f_x(x,y), f_y(x,y)$

```
# zoom in (make bigger)
res = cv2.resize(messi, None, fx=2, fy=2, interpolation = cv2.INTER_AREA)
plt.figure(figsize=size_me(res)) # slight use of matlab api to avoid plt.subplots(1,1,...) call
my_show(plt.gca(), res);
```


```
# zoom out (make smaller)
height, width = messi.shape[:2]
res = cv2.resize(messi, (int(.5*width), int(.5*height)), interpolation=cv2.INTER_LINEAR)

plt.figure(figsize=size_me(res))
my_show(plt.gca(), res)
```

Decreasing resolution (throwing out data) requires *decimation*

Increasing resolution (creating data from *nothing*) requires interpolation

MATRIX GLOSSARY

T	Translation
R	Rotation
A	Affine (arbitrary)
S	Scaling
H	Homography (prospective, projective)

TRANSLATION

What you would see in trig class:

$$p_{new} \leftarrow p_{old} + t$$

$$\begin{bmatrix} x \\ y \end{bmatrix} \leftarrow \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} x_{shift} \\ y_{shift} \end{bmatrix}$$

$$t_{trig} = \begin{bmatrix} x_{\text{shift}} \\ y_{\text{shift}} \end{bmatrix}$$

OpenCV affine matrix:

$$T_{ocv} = \begin{bmatrix} 1 & 0 & x_{\text{shift}} \\ 0 & 1 & y_{\text{shift}} \end{bmatrix}$$

TRANSLATION

The OpenCV affine matrix is applied to an *augmented* p

- p with a constant 1 tacked on to it.

$$T_{ocv}\hat{p} \rightarrow p_{new}$$

$$\begin{bmatrix} 1 & 0 & x_{\text{shift}} \\ 0 & 1 & y_{\text{shift}} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x + 0 + x_{shift} \\ 0 + y + y_{shift} \end{bmatrix} = \begin{bmatrix} x + x_{shift} \\ y + y_{shift} \end{bmatrix}$$

TRANSLATION - CODE

```
messi_gray = my_read_g(img_dir+'data/messi.jpg')

height, width = messi_gray.shape   # r,c -> x,y

x_shift, y_shift = 100, 50

M = np.array([[1, 0, x_shift],
              [0, 1, y_shift]], dtype=np.float32)
res = cv2.warpAffine(messi_gray, M, (width, height)) # arguments in x,y terms

my_show(plt.gca(), res, cmap='gray')
```

RIGID (ROTATION AND TRANSLATION)

Again, in OpenCV, this is applied to an augmented point:

$$R_{ocv}\hat{p} \rightarrow p_{new}$$

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \rightarrow p_{new}$$

RIGID (ROTATION AND TRANSLATION)

```
M = cv2.getRotationMatrix2D((width/2,height/2), 45, 1)
res = cv2.warpAffine(messi_gray, M, (width, height)) # args in xy terms (not rc terms)
my_show(plt.gca(), res, cmap='gray')
```

SCALE (SIMILARITY TRANSFORM)

What you would see in trig class:

$$p_{new} \leftarrow S p_{old}$$

$$p_{new} \leftarrow \begin{bmatrix} x_{scale} & 0 \\ 0 & y_{scale} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$S_{trig} = \begin{bmatrix} x_{scale} & 0 \\ 0 & y_{scale} \end{bmatrix}$$

OpenCV affine matrix:

$$S_{ocv} = \begin{bmatrix} x_{scale} & 0 & 0 \\ 0 & y_{scale} & 0 \end{bmatrix}$$

SCALE (SIMILARITY TRANSFORM)

```
messi_gray = my_read_g(img_dir+'data/messi.jpg')

height, width = messi_gray.shape   # r,c -> x,y

scale_factor = 0.75

M = np.array([[scale_factor, 0, 0],
              [0, scale_factor, 0]], dtype=np.float32)
res = cv2.warpAffine(messi_gray, M, (int(width * scale_factor),
                                     int(height * scale_factor))) # arguments in x,y terms

plt.figure(figsize=size_me(res))
my_show(plt.gca(), res, cmap='gray')
```

SCALE

When combining scaling with other transformation:

When $x_{scale} = y_{scale} = 1$

We get a rigid transformation

- Also called isometry or a Euclidean transformation
- Preserves scale (distance)

When $x_{scale} = y_{scale}$

We have a similarity transformation

- Also called a *scaled-rotation*

PERSPECTIVE (PROJECTIVE OR HOMOGRAPHY)

In a perspective transform, we have an arbitrary 3x3 matrix, *P*, applied to an augmented point:

$$p_{new} \leftarrow normalize(P\hat{p})$$

$$A_{ocv} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

The first two columns represent the scale-rotation component.

The last column represents the shift component.

PERSPECTIVE (PROJECTIVE OR HOMOGRAPHY)

```
# perspective transform:  straight lines -> straight lines

# pts in r,c terms (:sadface:)
rows, cols = messi_gray.shape
pts1 = np.float32([[0,0],[rows-1,0], [0,cols-1], [rows-1, cols-1]])
pts2 = np.float32([[0,0],[300,0],[0,300],[250,250]])

M = cv2.getPerspectiveTransform(pts1,pts2)
res = cv2.warpPerspective(messi_gray, M, (cols, rows)) # r,c -> x,y terms

my_show(plt.gca(), res, cmap='gray')
```

GEOMETRIC TRANSFORMS

Transform Type	Degrees of Freedom	Attribute preserved
Translation	2	Orientation
Rigid	3	Lengths
Similarity	4	Angles
Affine	6	Parallelism
Perspective	8	Straight lines