Intel AI Academy

IMAGE TRANSFORMATIONS

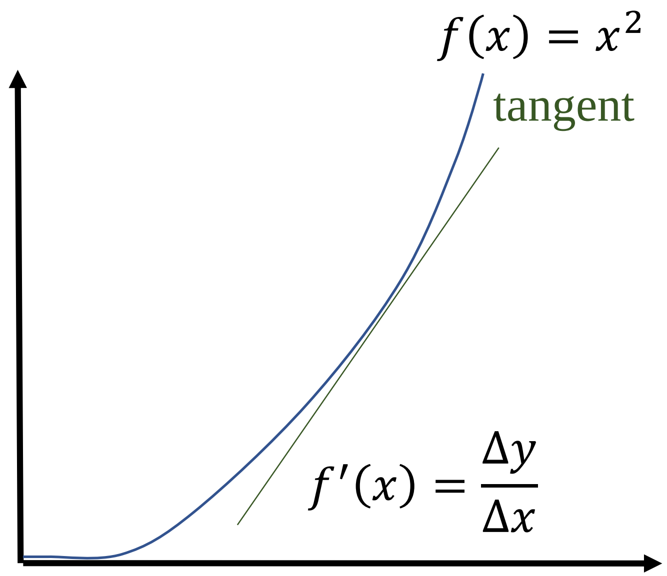
Part I

LEGAL NOTICES AND DISCLAIMERS

CALCULUS IN PIXEL SPACE: IMAGE DERIVATIVES

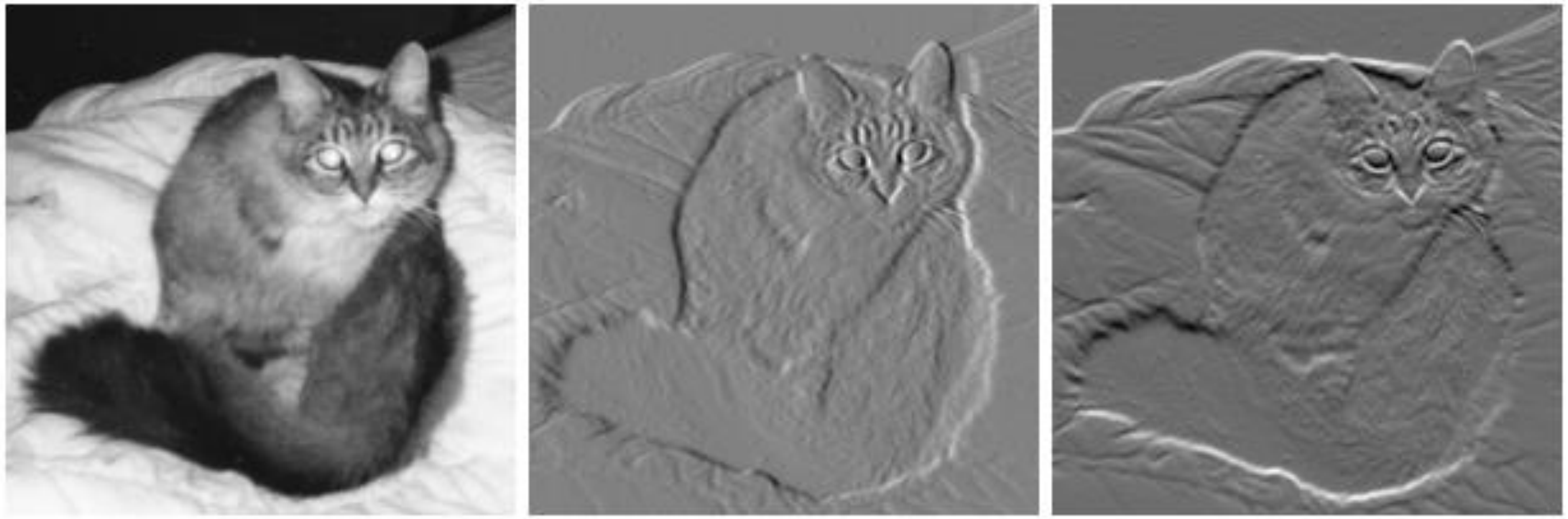
An image derivative represents the amount that an image's pixel values are changing at a given point.

Analogous to a derivative from calculus:



MOTIVATION FOR IMAGE DERIVATIVES

Image derivatives in x or y directions can detect features of images, especially edges:



Edges tend to correspond to changes in the intensity of pixels, which a derivative would capture.

Image source: https://upload.wikimedia.org/wikipedia/commons/6/67/Intensity_image_with_gradient_images.png

CALCULUS IN PIXEL SPACE: IMAGE DERIVATIVES

Image derivatives example:

Step | Code | Output

Generate box

```
from utilities import my_gshow

box = np.zeros((15, 15), dtype=np.int8)
box[4:11, 4:11] = 1.0
my_gshow(plt.gca(), box, interpolation=None)
```

Slice across middle

```
# here's a line across the middle ....
line = box[5:6, :] # 5:6 to keep 2D
line_p = np.diff(line) # line "prime" aka derivative

print(line)
print(line_p)
print(line_p.shape)
```

```
[[0 0 0 0 1 1 1 1 1 1 1 0 0 0 0]]
[[ 0  0  0  1  0  0  0  0  0  0 -1  0  0  0]]
(1, 14)
```

Plot slice, derivative

```
# note, derivative has values [-1, 0, 1] ...
# these get mapped to [0, 128, 255]
# (aka, black, gray, white) inside of imshow
fig,axes = plt.subplots(2, 1, figsize=(6,1), sharex=True)
my_gshow(axes[0], line, interpolation=None)
my_gshow(axes[1], line_p)
```

INTEGRAL IMAGES

Many applications

Fast calculation of Haar wavelets in face recognition

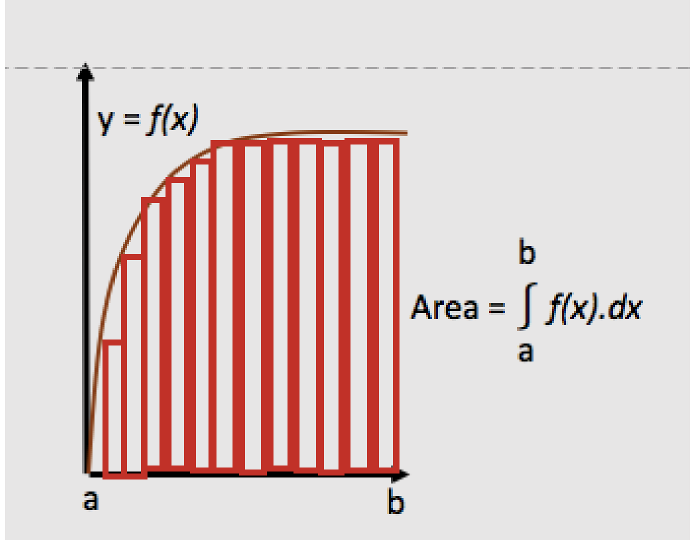
Precomputing can speed up application of multiple box filters

Can be used to approximate other (non-box) kernels

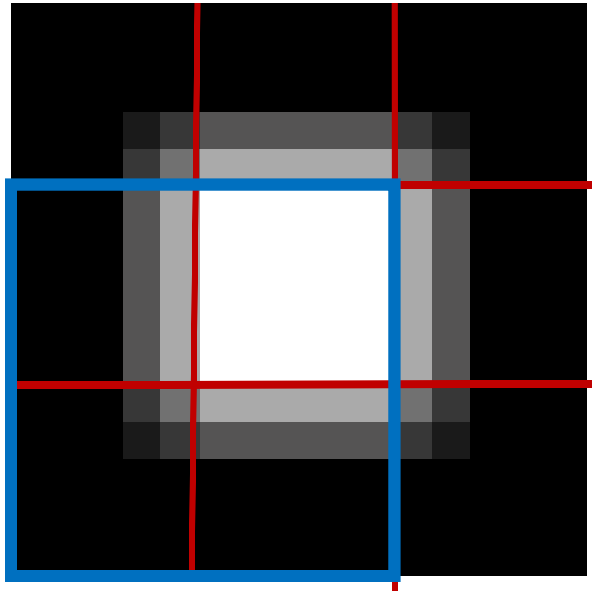
Method

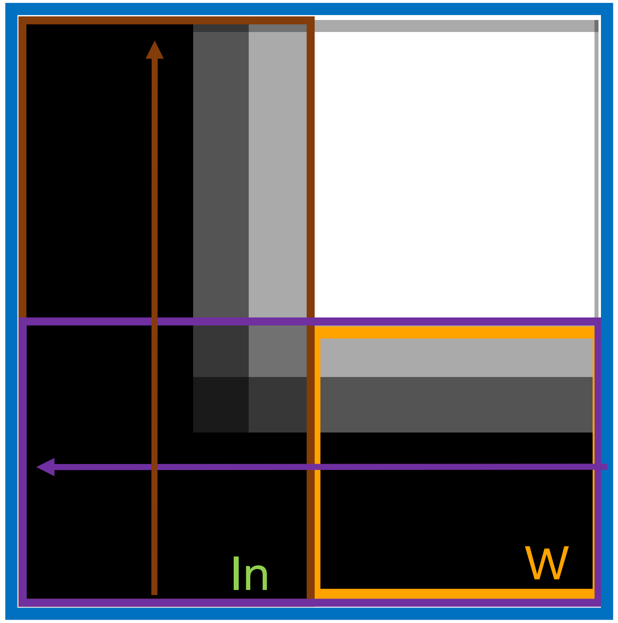
Summed area table is precalculated

- Pixel values from origin
- Recursive algorithm used



INTEGRAL IMAGES: FROM INTEGRAL TO AREA





Total (T; Entire Image Segment)

Row (R)

Out

Column (C)

$$\int_0^C f(x)dx$$

$$\int_a^b f(x)dx$$

$$\int_0^b - \int_0^a = \int_a^b$$

Input

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

$$W = T - C - R + In$$

Output

1	3	6	10
6	14	24	36
15	33	54	78
28	60	96	136

CONVOLUTIONS

DICE PROBABILITY

Probability of a 2 .... given by:

1. Taking all combinations of events.
2. Computing sums.
3. Returning counts of 2 events divided by total number of events.
4. Also, we know this is 1/6 * 1/6

```
d1, d2 = np.meshgrid(*[np.arange(1,7)]*2)
sums = d1 + d2
event_table = dict(zip(*np.unique(d1+d2, return_counts=True)))
print("{:.4f} {:.4f}".format(event_table[2] / sums.size, 1/6 * 1/6))
```

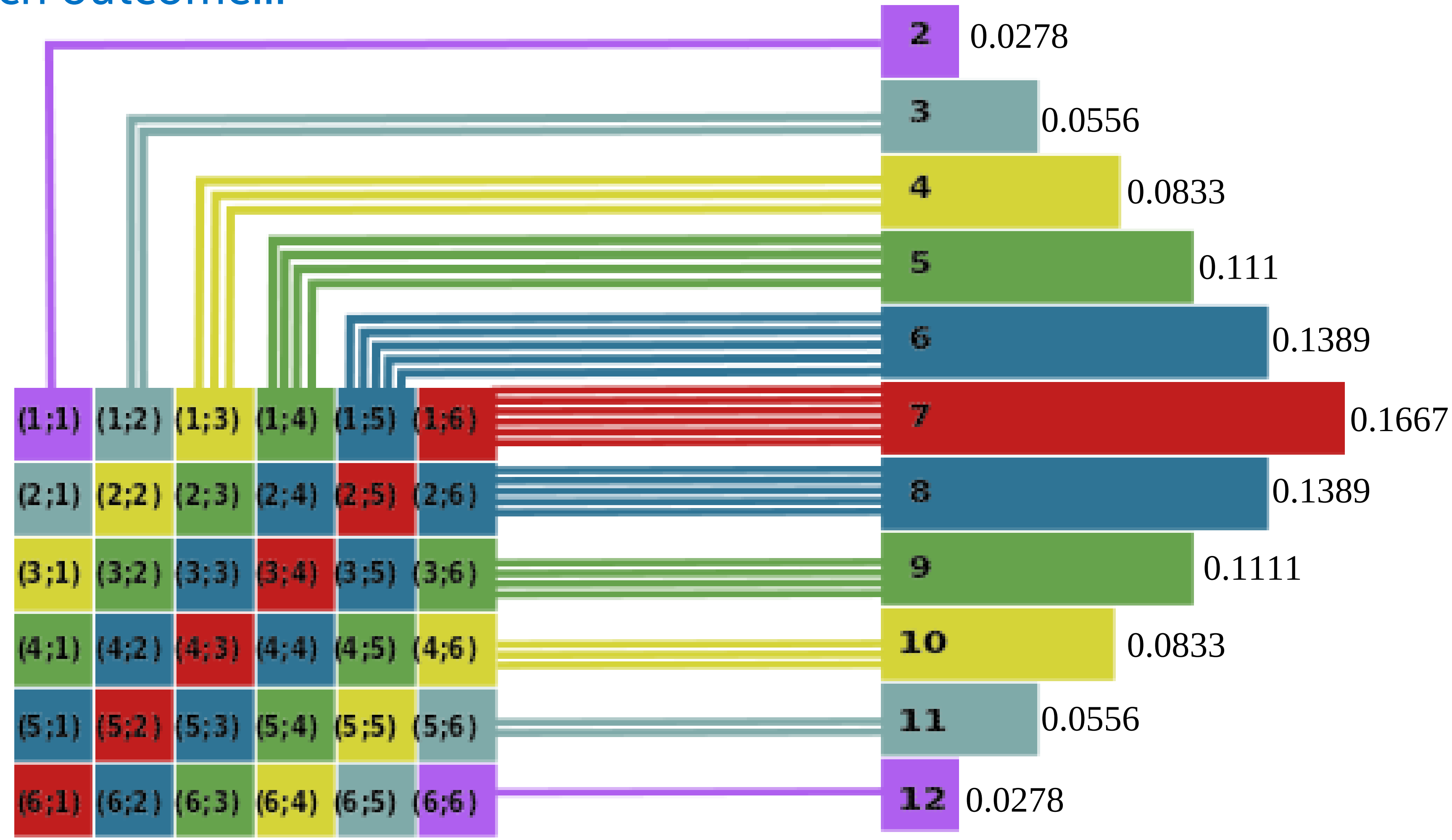
```
0.0278 0.0278
```

die = np.full(6, 1/6.0)print(die)

np.convolve(die, die, mode='full')

DICE PROBABILITY: CONVOLUTION

Probability of a each outcome…



Box graphic from: https://commons.wikimedia.org/wiki/Category:Dice_probability#/media/File:Twodice.svg
Dice graphic from: https://commons.wikimedia.org/wiki/Category:6-sided_dice#/media/File:6sided_dice.jpg

PROBABILITY AS A SLIDING WINDOW

In the dice example, to get, say, a total of 7:

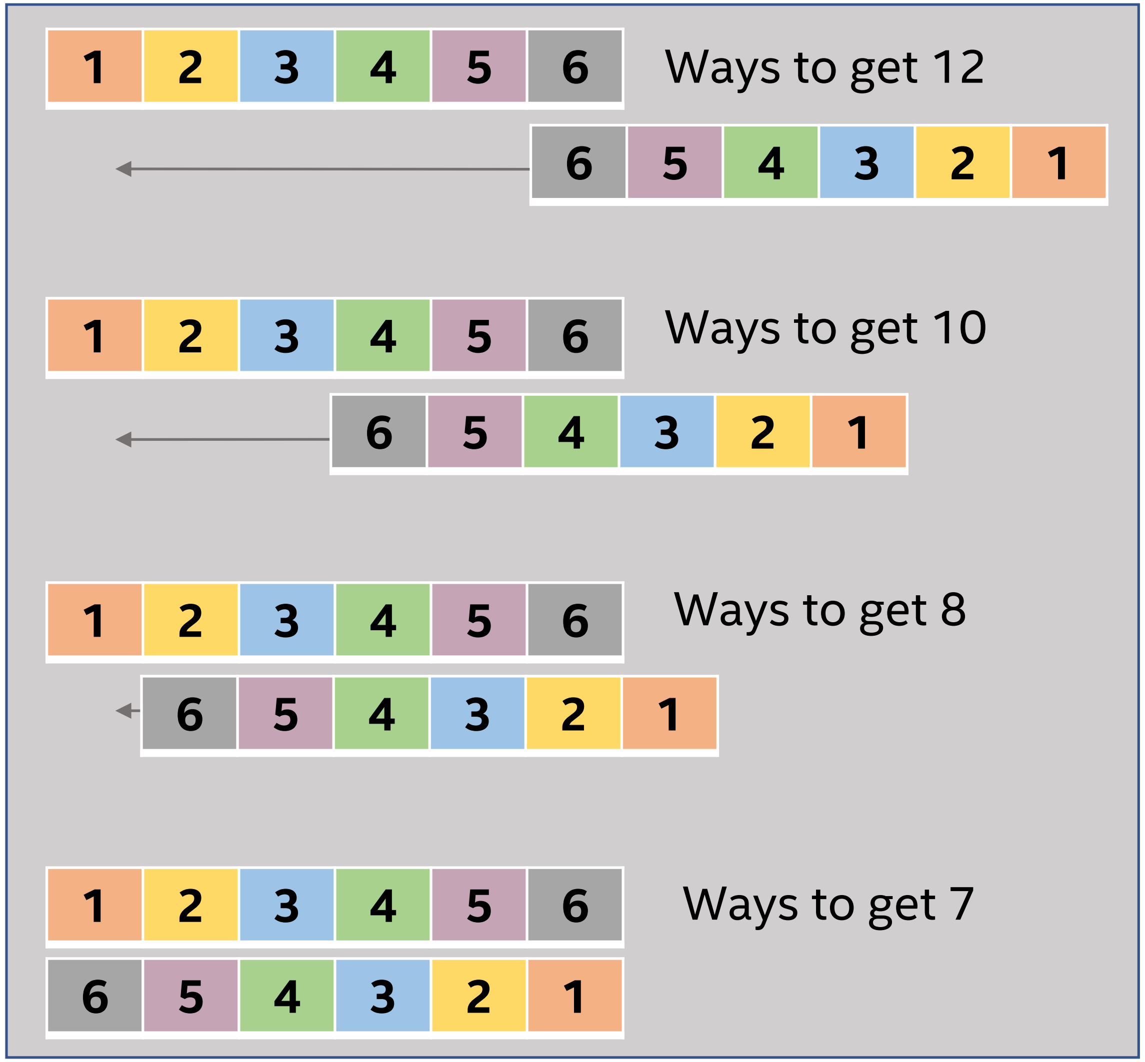
- We can fix a point 7 and slide the two arrays past (one in reverse order).
- And when they add up to seven, we take that sum-product.

$$P(s = T) = \sum_{i} P(i)\, P(T - i)$$

$$P(T) = \sum_{i+j=T} P(i)\, P(j)$$

P(s=2) = P_1(⚀)* P_2(⚀)

P(s=3) = P_1(⚀)* P_2(⚁) + P_1(⚁)* P_2(⚀)



PROBABILITY AS A CONVOLUTION

In general, the probability of sums of events is the convolution of the probabilities of the component events.

In general, in mathematics, a convolution is a function *h* produced by a function *g* "operating" on another function *f*. This is usually written:

$$f \otimes g = h$$

Here, the distribution of probabilities for two dice (*h*) is a convolution of the probability distribution over the first die (*f*) with the probability distribution over the second die (*g*).

PROBABILITY AS A CONVOLUTION

Dice probability is a convolution:

$$P(2) = P(1) * P(1)$$

$$P(3) = \sum_{r=1,2} P(r) * P(T - r) = P(1) * p(2) + P(2) * P(1)$$

Sum over all the right rolls (r) so that the events add up to our desired total T

Note: r + (T - r) = T

Could also write it like this:

$$P(3) = \sum_{r_1 + r,2 = 3} P(r_1) * P(r_2)$$

And in general like this:

$$P(T) = \sum_{r_1 + r,2 = T} P(r_1) * P(r_2)$$

Those sums are convolutions!

Here, we've convolved a discrete function with itself.

Just like we'll do with images, except images are 2D.

PROBABILITY AS A CONVOLUTION

Dice example with code:

Step	Code & Output
Generate dice probability distribution	see code 1
Convolve to get overall probabilities	see code 2

Generate dice probability distribution

```
# probability of a 2:
# P_twodice(Total) = sum_part P_onedie(part) * P_onedie(Total-part)
# P_twodice(2) = sum_part P_onedie(part) * P_onedie(2-part)
# [note, the only valid part here is part = 1 ... all others are "too big"]
die = np.full(6, 1/6.0)
print(die)
```

```
[0.1667 0.1667 0.1667 0.1667 0.1667 0.1667]
```

Convolve to get overall probabilities

```
# can think:  invert second, align, multiple and add
np.convolve(die, die, mode='full') # all probabilities in event space (pads outside with effective zeros)
```

```
array([0.0278, 0.0556, 0.0833, 0.1111, 0.1389, 0.1667, 0.1389, 0.1111,
       0.0833, 0.0556, 0.0278])
```

A GENERAL 2D CONVOLUTION

The usual formula looks like this:

$$f \otimes g[N] = \sum_{offsets} f[offset]g[N - offset] \qquad f \otimes g[N,M] = \sum_{\text{"right"}ij,kl} f[i,k]g[j,l]$$

But, as an idea, that is less than clear!

a pixel

$$img \otimes kern[R,C] = \sum_{\substack{pixel\_pair \\ in aligned \\ neighborhoods \\ (kern,R,C)}} img[imgr, imgc]kern[kernr, kernc]$$

result

img_r, img_c
$kern_r$, $kern_c$

given a kernel, and R, C…
this tells us the corresponding pixels in img and elements in kernel

2D CONVOLUTION

A combination of a sliding window as it moves over a 2D image

We align the kernel with a block of the image at an anchor point

- In the probability example, our anchor point was a Total

The resulting value at the anchor point is the dot-product of the aligned regions

- Dot-product means multiply elements pairwise and then sum

2D CONVOLUTION

The probability formula and the 2D convolution formula have these pieces:

- Multiply element wise
- Sum the products
- Align

The alignment determines the values we sum over and the values passed to the functions inside the sum:

- Total anchored the probabilities
- For a 2D convolution, a target pixel at R,C anchors the neighborhoods of the image and the kernel

MANUAL 2D CONVOLUTION WITH SCIPY

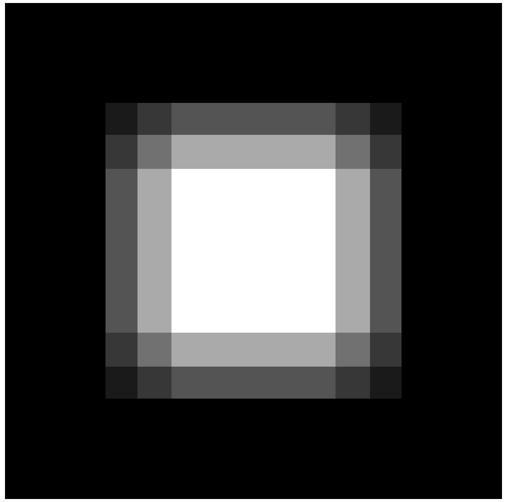
convolve2d(box, kernel, mode)

With mode='same' or mode='full' we have to pad

- Can use wrapping
- Symmetric
- Fill value

```
from scipy.signal import convolve2d as ss_convolve2d
box = np.zeros((15, 15), dtype=np.int8)
box[4:11, 4:11] = 1.0

size = 3 # adjust me!
kernel = np.full((size,size), 1/size**2)
out = ss_convolve2d(box, kernel, mode='same')
```



KERNEL

KERNEL METHODS

Kernel methods involve taking a convolution of a *kernel* - a small array - with an image to detect the presence of features at locations on an image.

Also called filtering methods.

Images are *filtered* using neighborhood operators.

Separable filtering:

- 2D can be applied as sequential 1D (first a horizontal filter, then a vertical filter).

LOW-PASS FILTERS

Linear methods:

- Box (mean)
- Gaussian blur (weighted sum)

Non-linear methods:

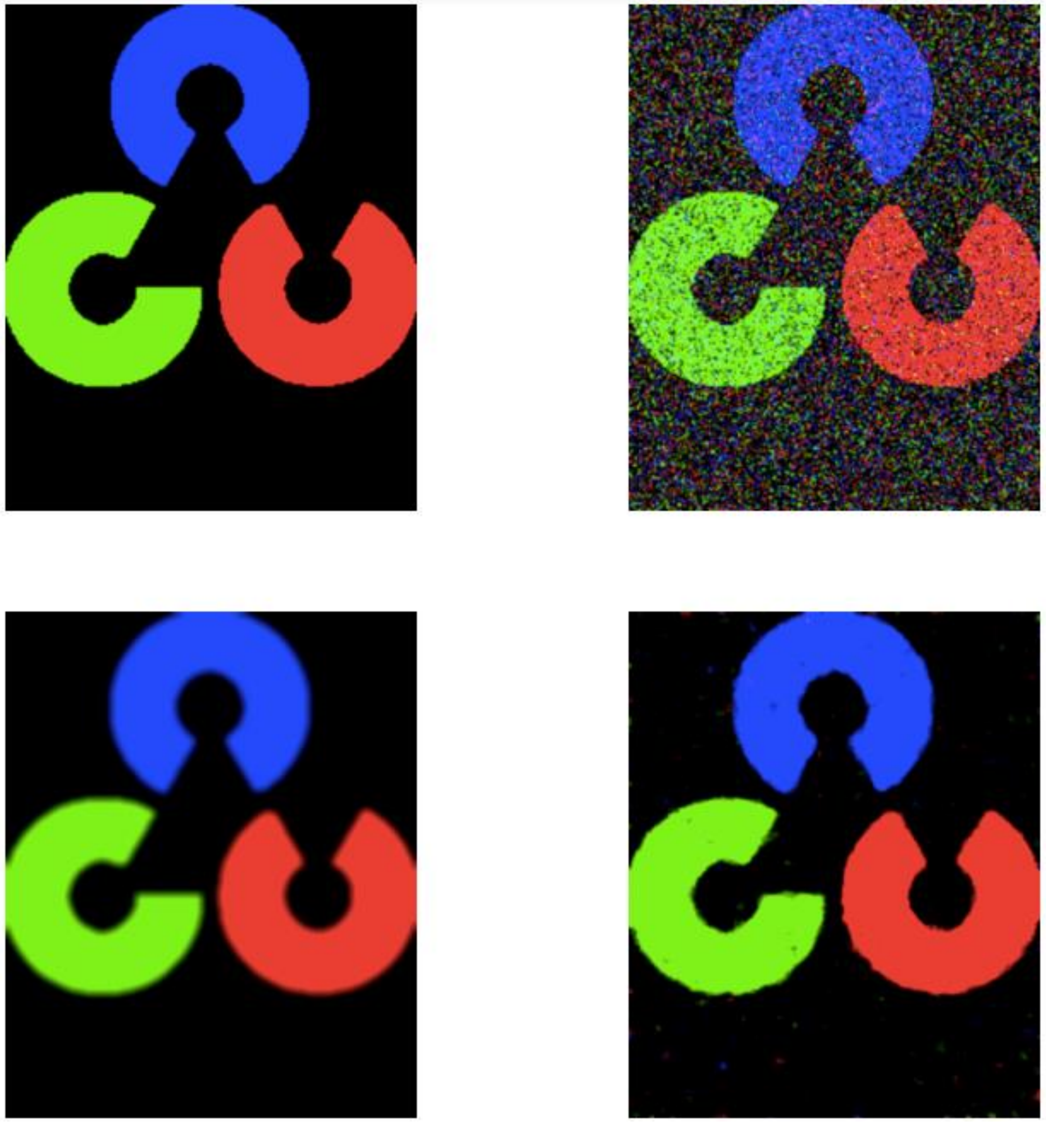
- Bilateral filter (combination of a Gaussian and a empirical similarity of the neighborhood to the center pixel)
- Median blur

BLURRING

```
gauss  = cv2.GaussianBlur(img, kernel_size, 5)


dots = np.random.randint(0,256,size=img.shape).astype(np.uint8)
dotted = np.where(np.random.uniform(size=img.shape) > .3, img, dots)
median = cv2.medianBlur(dotted, kernel_size[0])  # 5 --> 5x5 neighborhood

fig,axes = plt.subplots(2,2,figsize=(12,12))
my_show(axes[0,0], img)
my_show(axes[0,1], dotted)
my_show(axes[1,0], gauss)
my_show(axes[1,1], median)
```



BLURRING AND SMOOTHING USING BILATERAL FILTER

Bilateral smooths both intensities and colors

Edge preserving will produce a watercolor effect when repeated

Pixel distance -and- "color distance"

```
saved = cv2.bilateralFilter(dotted,9,100, 50)
my_show(plt.gca(), saved)
```

KERNEL APPLICATION DETAILS

Padding – border effects

Constant

Replicate the edge pixel value

Wrap around to other side of image

Mirror back toward center of image

MORPHOLOGY

MORPHOLOGY FUNDAMENTAL OPERATIONS

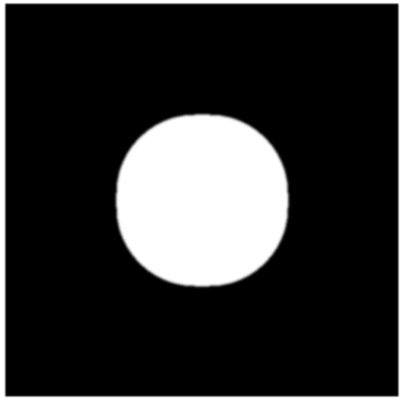
Morphology in image processing refers to turning each pixel of an image *on* or *off* depending on whether its neighborhood meets a criteria.

Fundamental examples: erosion and dilation

Original image

Erosion: Pixel is on if entire kernel-neighborhood is on

Dilation: Pixel is on if ANY kernel-neighborhood is on



MORPHOLOGY FUNDAMENTAL OPERATIONS

Binary image operations: formal definitions

Morphological operations

Dilation dilate(f,s) = c > 1

on if any in neighborhood are on

Erosion erode(f,s) = c = S

on if all in neighborhood are on

Majority maj(f,s) = c > S/2

on if most in neighborhood are on

s= structuring element f = binary image

S= size of structuring element

c = count in aligned neighborhood after multiplying f and s

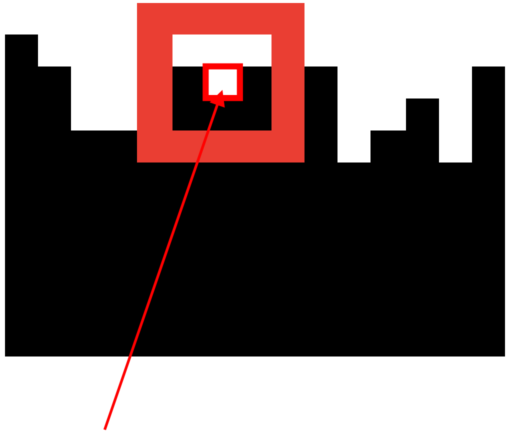
ERODE

Erode away the foreground (foreground is white)

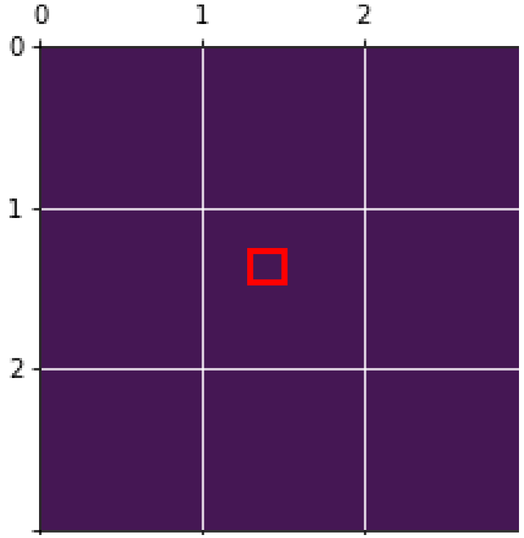
- Pixel is on if entire kernel-neighborhood is on
- So, inside is good, outside is off

- Borders of foreground: will be reduced
- More will become background

- Enhances background
- Removes noise in background
- Add noise in foreground



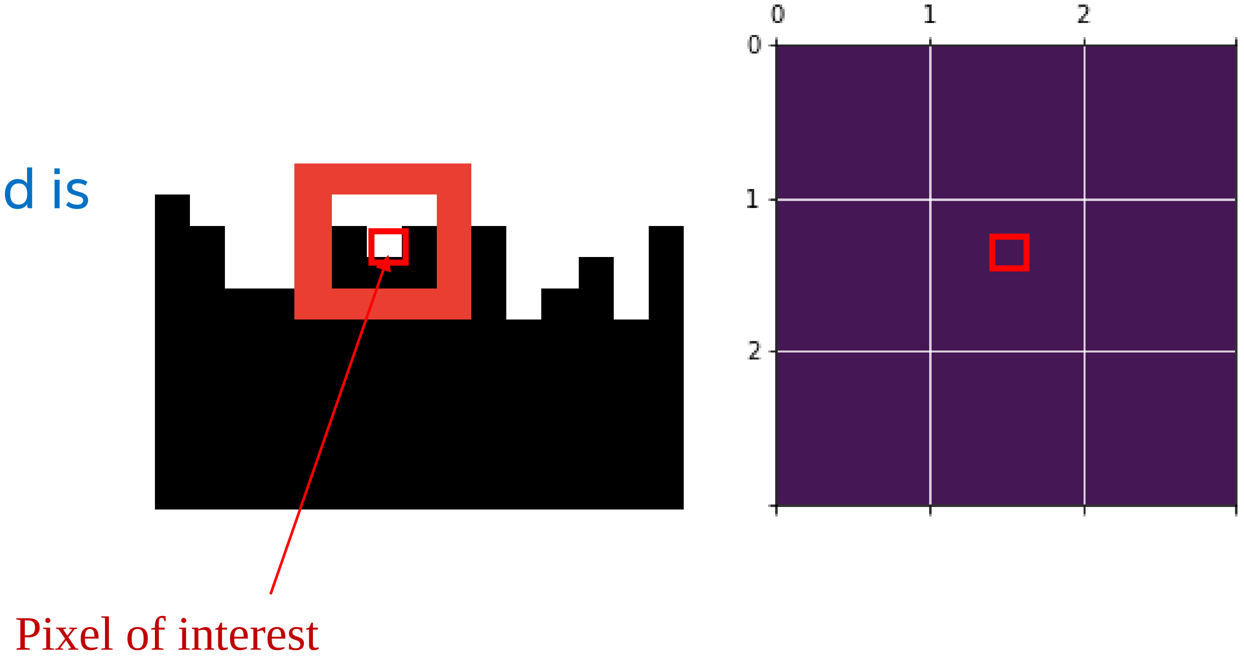
Pixel of interest



DILATE

Dilate adds to the foreground (white)

- Pixel is on if ANY kernel-neighborhood is on
- Inside - good; outside – off
- Border – expanded
- Enhances foreground
- Removes noise in foreground
- Adds noise in background



MORPHOLOGY: ADDITIONAL OPERATIONS

Binary image operations

Morphological operations

Opening open(f,s) = dilate(erode(f,s), s)

Closing close(f,s) = erode(dilate(f,s), s)

s= structuring element

f = binary image

S= size of structuring element

c = count

OTHER RELATIONSHIPS: OPEN

Opening: dilate(erode(img))

- Erode it, then dilate it
- Remove outside noise (false foreground); remove local peaks
- Count objects
- opening = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)

OTHER RELATIONSHIPS: CLOSE

Closing: erode(dilate(img))

- Remove inside noise (false background)
- Used as a step in connected-components analysis

  closing = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)

  Iterations of these are erode(erode(dilate(dilate())))

  - $e^i(d^{i(img)})$

  gradient = dilation - erosion

  - finds boundary

  gradient = cv2.morphologyEx(img, cv2.MORPH_GRADIENT, kernel

OTHER RELATIONSHIPS: TOPHAT/BLACKHAT

Isolate brighter/dimmer (tophat/blackhat) than their surroundings

Tophat: image – opening

```
tophat = cv2.morphologyEx(img, cv2.MORPH_TOPHAT, kernel)
```

Blackhat: closing – image

```
blackhat = cv2.morphologyEx(img, cv2.MORPH_BLACKHAT, kernel)
```

These can be related to other mathematical techniques:

- Max-pool in neural network layers is a dilation using a square structuring element followed by downsample (1/p).
- It is possible to learn the operations; for example, as implicit layers in the neural network.

IMAGE PYRAMIDS

IMAGE PYRAMIDS

A *stack* of images at different resolutions is an image pyramid.

- Use when you are unsure of object sizes in an image
  - Work with images of different resolutions and find object in each
- Uses *Gaussian* and *Laplacian* layers

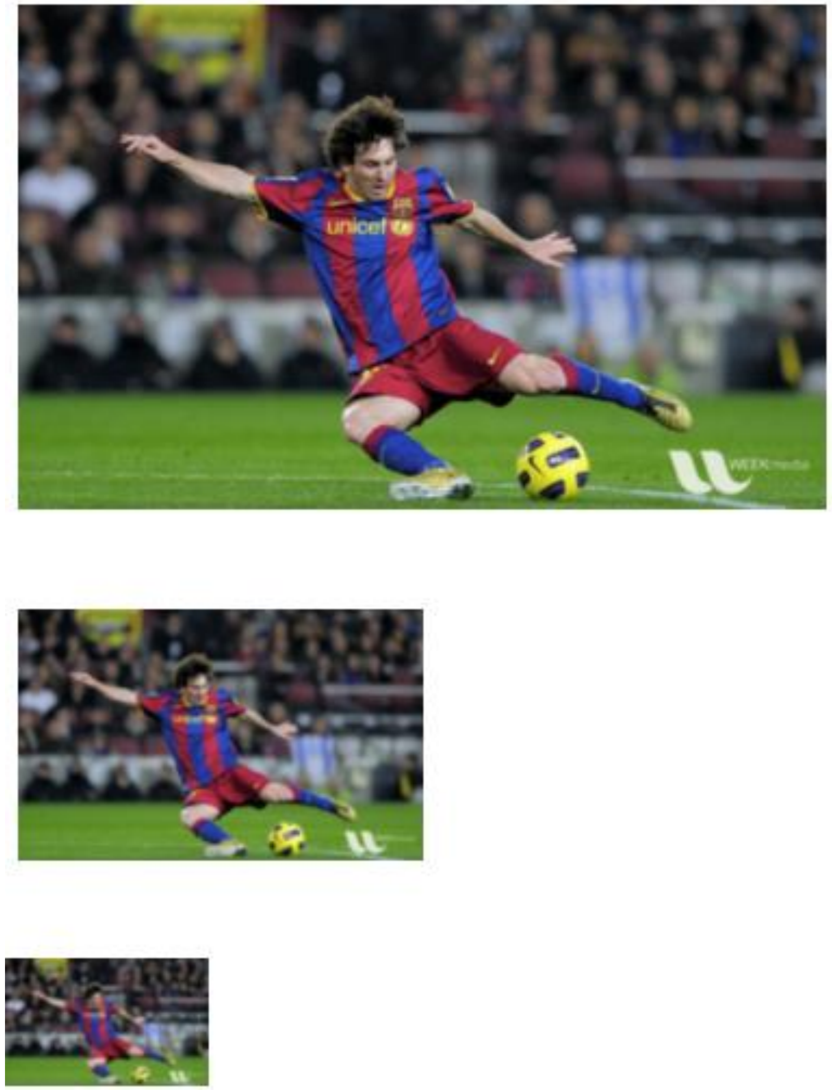
GAUSSIAN PYRAMID

Having multiple resolutions represented simultaneously.

Working with lower-resolution images allows for faster computations.

Each Gaussian level loses information:

- Create a complementary Laplacian Pyramid, which holds that information
- Bottom Gaussian level *plus* all Laplacian levels reconstructs the original image



PYRAMIDS

Gaussian Pyramid

$$G_0 = original\ image$$

$$G_1 = DWN(G_o \otimes \text{Gaus}_{5x5}) = PyrDwn(G_o)$$

$$G_2 = PyrDwn(G_1)$$

Laplacian Pyramid

$$L_o = G_o - UP(G_1) \otimes \text{Gaus}_{5x5} = G_o - PyrUp(G_1)$$

$$L_1 = G_1 - PyrUp(G_2)$$

EXAMPLE LAPLACIAN LEVEL

$$L_o = G_o - UP(G_1) \otimes \text{Gaus}_{5x5} = G_o - PyrUp(G_1))$$

```
laplacian_messi = messi - cv2.pyrUp(cv2.pyrDown(messi))
fig, ax = plt.subplots(1,1,figsize=size_me(laplacian_messi))
my_show(ax, laplacian_messi)
```

LAPLACIAN PYRAMID (PYRUP/PYRDOWN)

Power of 2 for biggest image sizes

- This makes halving/doubling work well
- Can also pad out to next power of 2

```
restored_1 = l_0 + cv2.pyrUp(g_1)
restored_2 = l_0 + cv2.pyrUp(l_1 + cv2.pyrUp(base))
fig, ax = plt.subplots(1,3,figsize=(12,4))
my_show(ax[0], g_0)
my_show(ax[1], restored_1)
my_show(ax[2], restored_2)
```

Expanding:

$$g_0 = l_0 + UP(g_1) = l_0 + UP\big(l_1 + UP(g2)\big) = l_0 + UP(l_1 + UP(base)))$$

USING IMAGE PYRAMIDS FOR BLENDING

Combining two images seamlessly (image stitching and compositions)

1. Decompose source images into Laplacian pyramid.
2. Create a Gaussian mask from the binary mask image.
3. Compute the sum of the two weighted pyramids to stitch the images together.

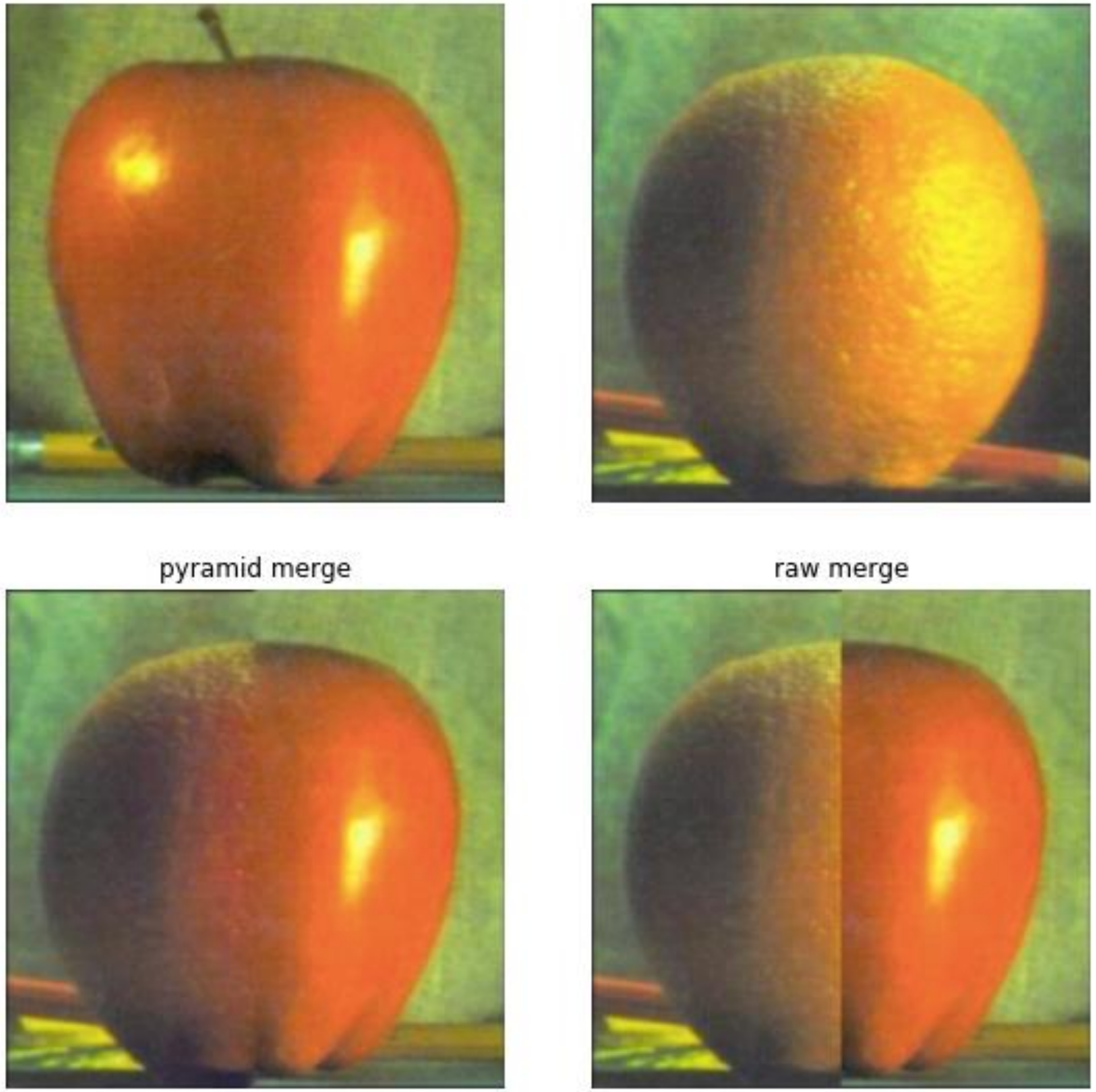


IMAGE GRADIENTS

EDGE DETECTORS

Finding stable features for matching

Matching human boundary detection

Sobel, Scharr, and Laplacian filters

```
laplacian = cv2.Laplacian(box, cv2.CV_64F, ksize=5)
sobel_x   = cv2.Sobel(box, cv2.CV_64F, 1, 0, ksize=5)
sobel_y   = cv2.Sobel(box, cv2.CV_64F, 0, 1, ksize=5)
sobel_xy  = cv2.Sobel(box, cv2.CV_64F, 1, 1, ksize=5)

fig, axes = plt.subplots(1,5,figsize=(12,3))
for ax, smoother in zip(axes.flat,
                        [box,laplacian, sobel_x, sobel_y, sobel_xy]):
    my_gshow(ax, smoother)
```

SOBEL

Most common differentiation operator

Approximates a derivative on discrete grid

- Actually a fit to polynomial

Used for kernels of any size

- Larger kernels are less sensitive to noise, and therefore more accurate

Combine Gaussian smoothing with differentiation

Higher order also (first, second, third, or mixed derivatives)

Sobel x

-1	0	1
-2	0	2
-1	0	1

Sobel y

1	2	1
0	0	0
-1	-2	-1

SCHARR

Scharr is a specific Sobel case used for computing 3x3

- As fast as Sobel, but more accurate for small kernel sizes

Especially useful when implementing common shape classifiers

- Need to collect shape information through histograms of gradient angles

First x- or y- image derivative

- Scharr(src, dst, ddepth, dx, dy, scale, delta, borderType)
- Sobel(src, dst, ddepth, dx, dy, cv_scharr, scale, delta, borderType)

Scharr x

-3	0	3
-10	0	10
-3	0	3

Scharr y

3	10	3
0	0	0
-3	10	3

LAPLACIAN

Laplacian function

$$Laplace(f) = \frac{\delta 2f}{\delta x2} + \frac{\delta 2f}{\delta y2}$$

Can be used to detect edges

Can use 8-bit or 32-bit source image

Often used for blob detection

Local peak and trough in an image will maximize and minimize Laplacian

Sum of second derivatives in x,y

- Works like a second-order Sobel derivative

MULTIPLE COLORS

Should you detect edges in color or grayscale?

- Typically we do edge detection in grayscale.
- If we want to do edge detection in color…
  - If you take the union of edges, you might thicken the edges
  - If you take the sum of gradients, you need to be careful about sign cancelation
  - Consider non-RGB color space

DISTANCE TRANSFORM

Once we have edges, we may need to find and group together pixels as *an object.*

One step in that process is to find the distances from a pixel to a boundary:

1. Invert an edge detector (non-edge is white)
2. Find distance from central points (now white) to nearest edge (now black)

```
dist = cv2.distanceTransform(sobel_xy, cv2.DIST_L2, 0) # different distances; 0 (precise), 3,5
my_gshow(plt.gca(), dist, interpolation=None)
```