ECE5554 – Computer Vision
Lecture 2b – Filtering

Creed Jones, PhD

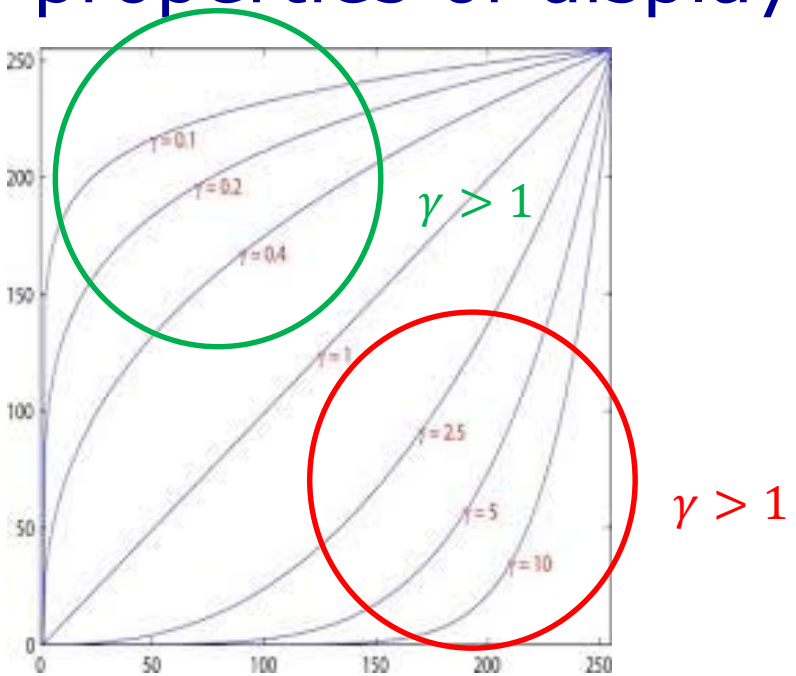
Today's Objectives

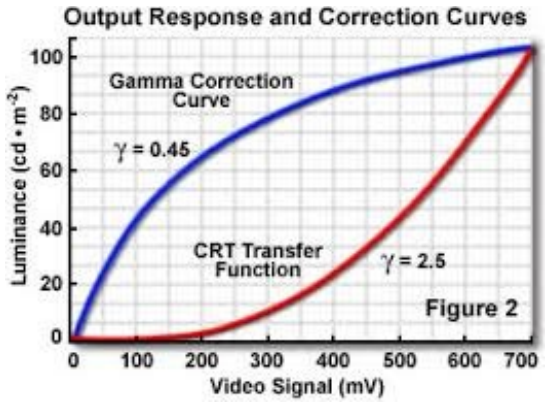
- Gamma Correction
- Image Arithmetic
- Filtering
- Kernels
- Smoothing
- Gaussian Smoothing
- Filtering in Color Images

Intensity transformations defined by a power expression $I_o = k(I_i)^\gamma$ are called *gamma correction*; they are often used to compensate for non-linear intensity properties of displays or printers

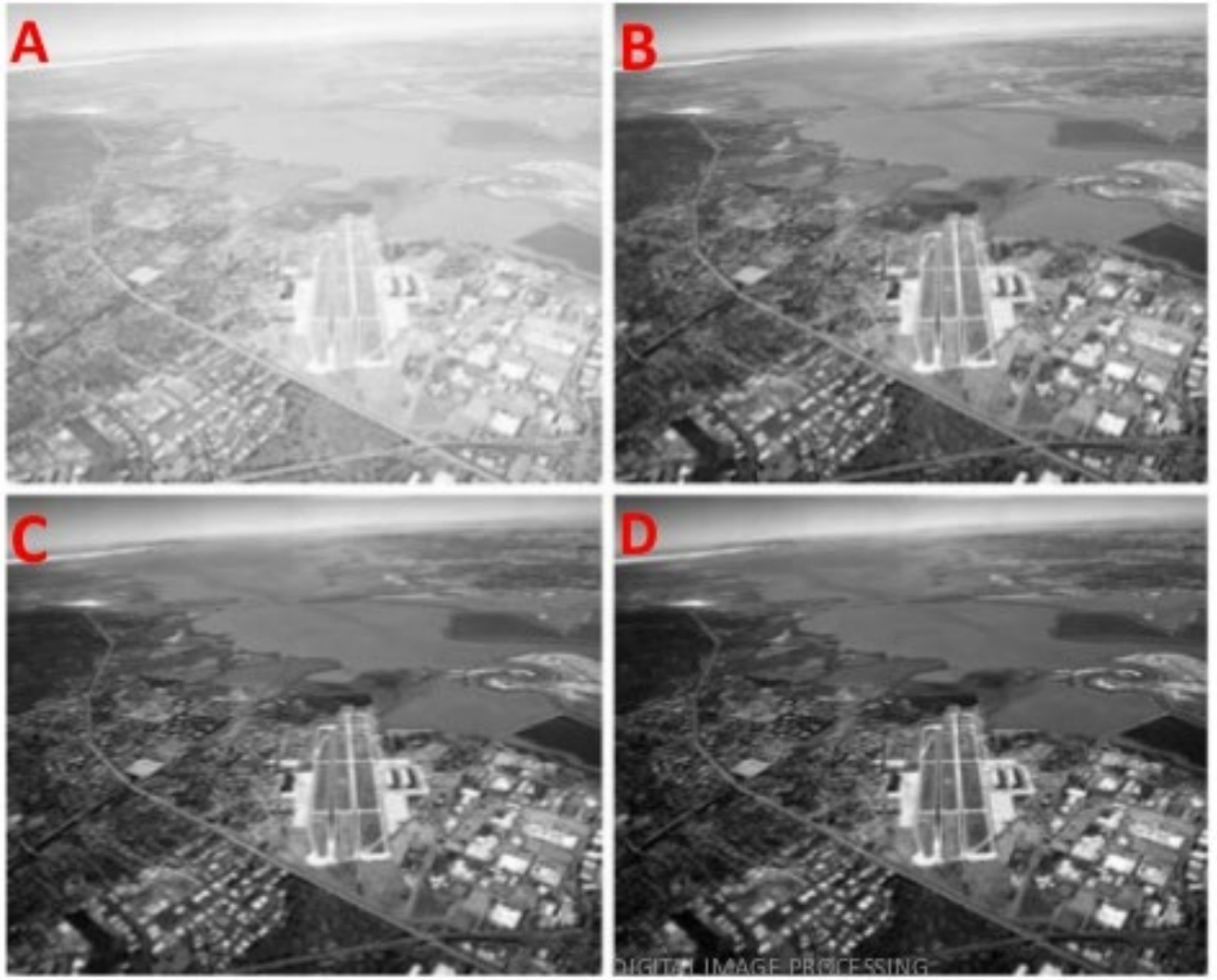
- If the display device has a non-linear mapping from input to displayed intensity, we often approximate it by a power curve
  - Old CRTs had $\gamma = 2.5$ or so
  - To compensate, images were transformed by a power transformation with $\gamma = 0.45$

- The resulting displayed image has apparent brightness approximately proportional to initial intensity values





Power Law Transformation

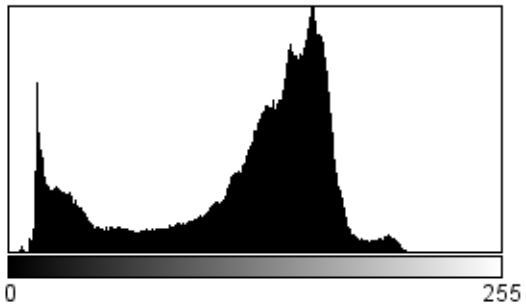


A : original image
For c=1
B : γ =3.0
C : γ =4.0
D : γ =5.0

DIGITAL IMAGE PROCESSING 35

$$I_o(c,r) = k[I_i(c,r)]^{\gamma}$$

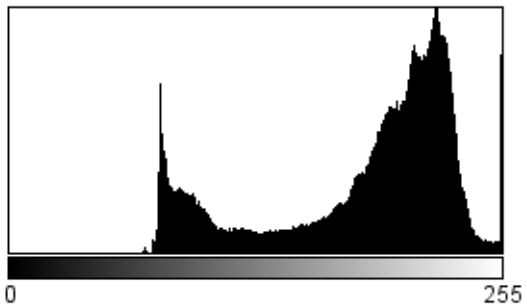
Considering the pixel values as scalars, we can define many useful single-image point operations as simple mathematical operations on each pixel value



Count: 414720
Mean: 120.027
StdDev: 49.188

Min: 3
Max: 220
Mode: 157 (7480)

- The operation add(p) represents adding the same scalar constant p to each pixel value
- $I_o(c,r) = I_I(c,r) + p$
- In this case, 64 is added to each intensity value



Count: 414720
Mean: 183.929
StdDev: 49.036

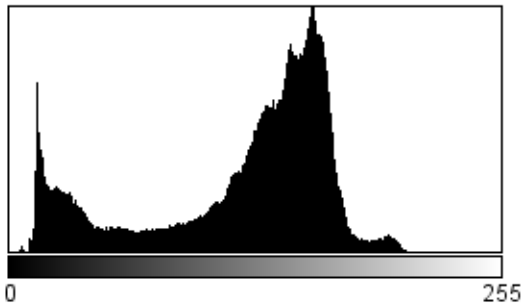
Min: 67
Max: 255
Mode: 221 (7480)

abs(), add(p), multiply(s), invert(), log(), sqrt()

- add(p) adds a scalar constant to each pixel
  - shifts the entire histogram by p
- abs() transforms negative to positive values
  - "folds" the histogram
- multiply(s) multiplies each pixel by a constant
- invert() flips the histogram (each intensity subtracted from 255)
- log() and sqrt() are examples of contrast stretching curves

- All of these can be implemented via LUT – because each pixel of a given intensity will transform to a common new intensity

Compute the square root of each pixel value and replace the original value; the result will be too dim to see without increasing contrast (I used histogram equalization)

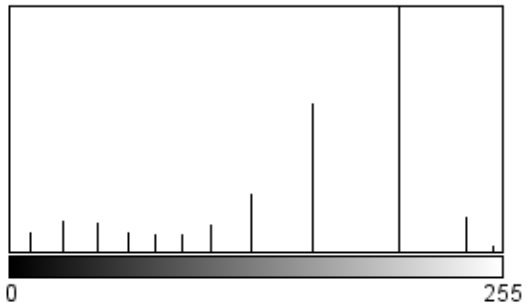


Count: 414720
Mean: 120.027
StdDev: 49.188

Min: 3
Max: 220
Mode: 157 (7480)

Count: 414720
Mean: 10.113
StdDev: 2.777

Min: 1
Max: 14
Mode: 12 (153380)



Count: 414720
Mean: 150.516
StdDev: 63.523

Min: 0
Max: 251
Mode: 202 (153380)

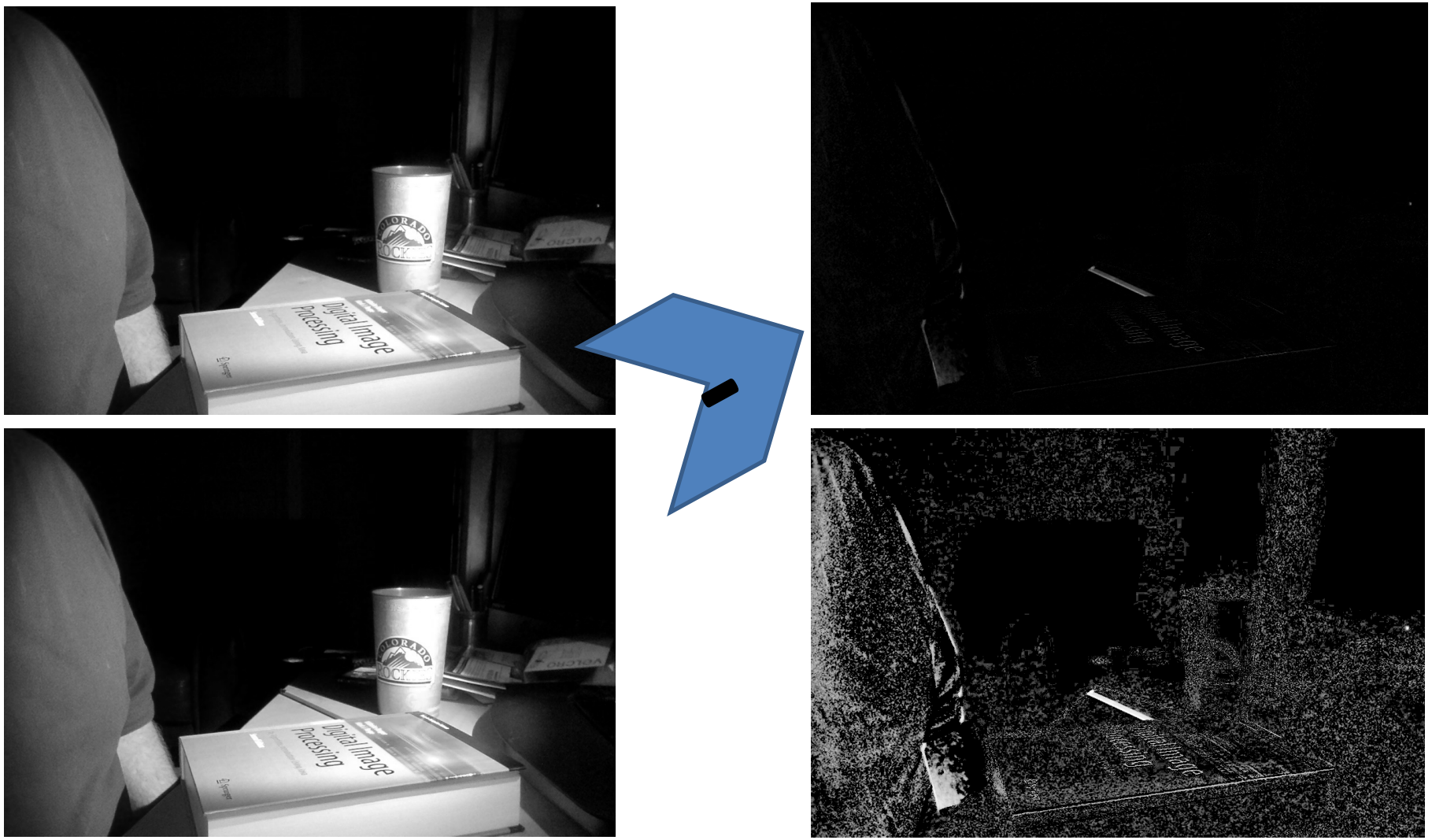
Multi-image point operations produce a result by combining pixels at the same locations in two (or more) source images:

$$I_o(c,r) = f(I_a(c,r), I_b(c,r))$$

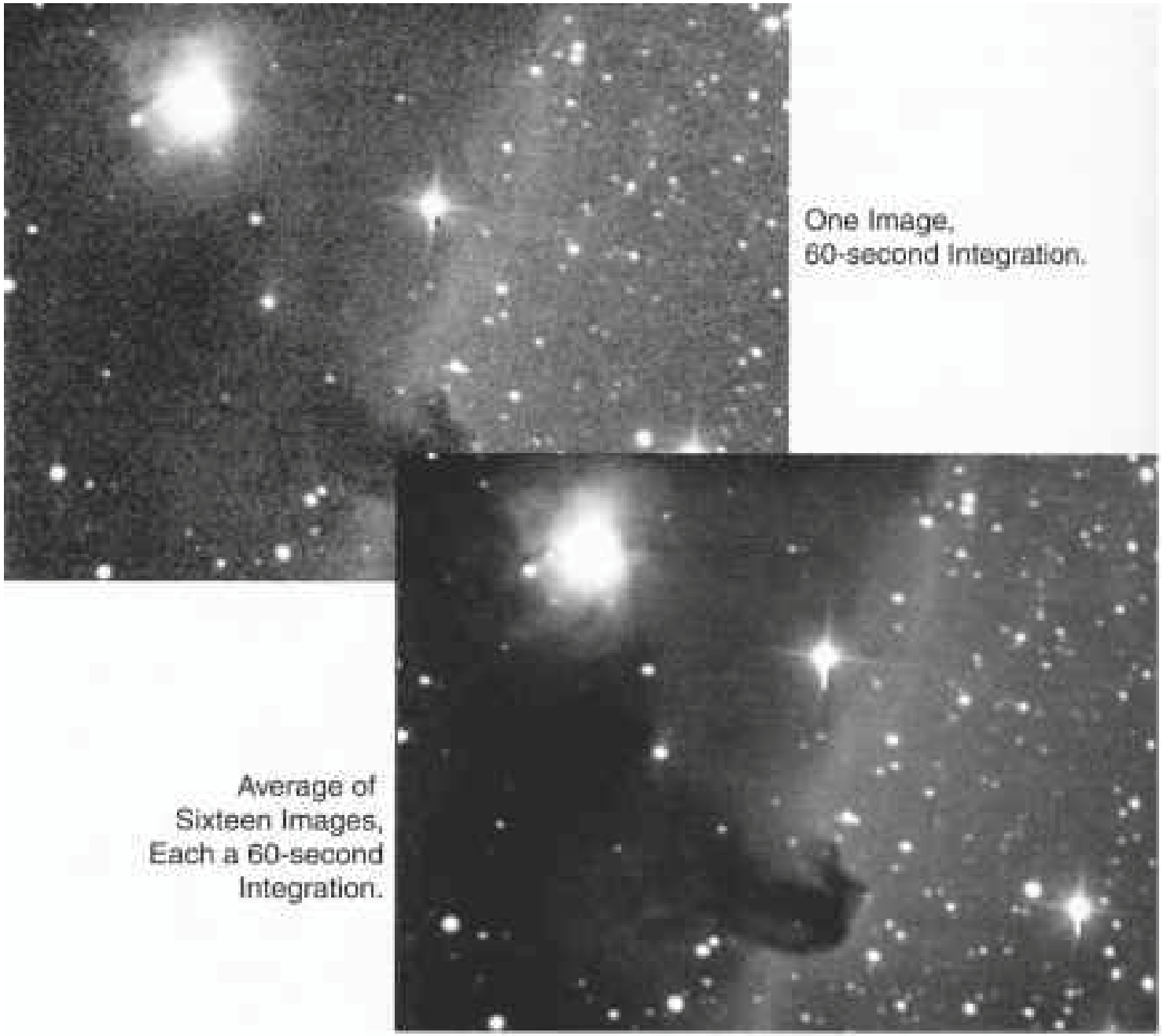
- add(I, J) will add the two images pixel by pixel
  - often will divide by two or otherwise scale the output
- subtract(I,J) or diff(I,J) will form the difference in the two images (order generally matters)

- Image subtraction (or differencing) will show areas of difference – <u>if the two images are precisely registered spatially, and if all other image characteristics are constant (illumination, focus, etc).</u>

Pixelwise subtraction will reveal differences between two images – BUT noise is enhanced and the registration must be excellent



Averaging two or more well-registered images is a common method for reducing noise; the noise is randomly distributed (assume $\mu = 0$) but the image content will be reinforced



Computing the pixel-wise max or min of two images can be used to reduce the effect of artifacts like glare, or to find occasional defects

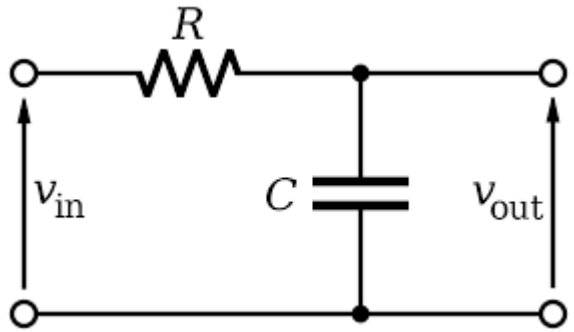
MAX

Other mathematical operations between pixel values of separate images have special uses but aren't as widely performed

- multiplying two images, pixel by pixel, is often done when one image is a synthetically-created "mask" or "weighting" image
- This can also be done by pixel-wise "and" and "or" operations

Image Filtering is the application of operations that change the spatial frequency characteristics of an image – generally to improve the "distinguishability" of important features

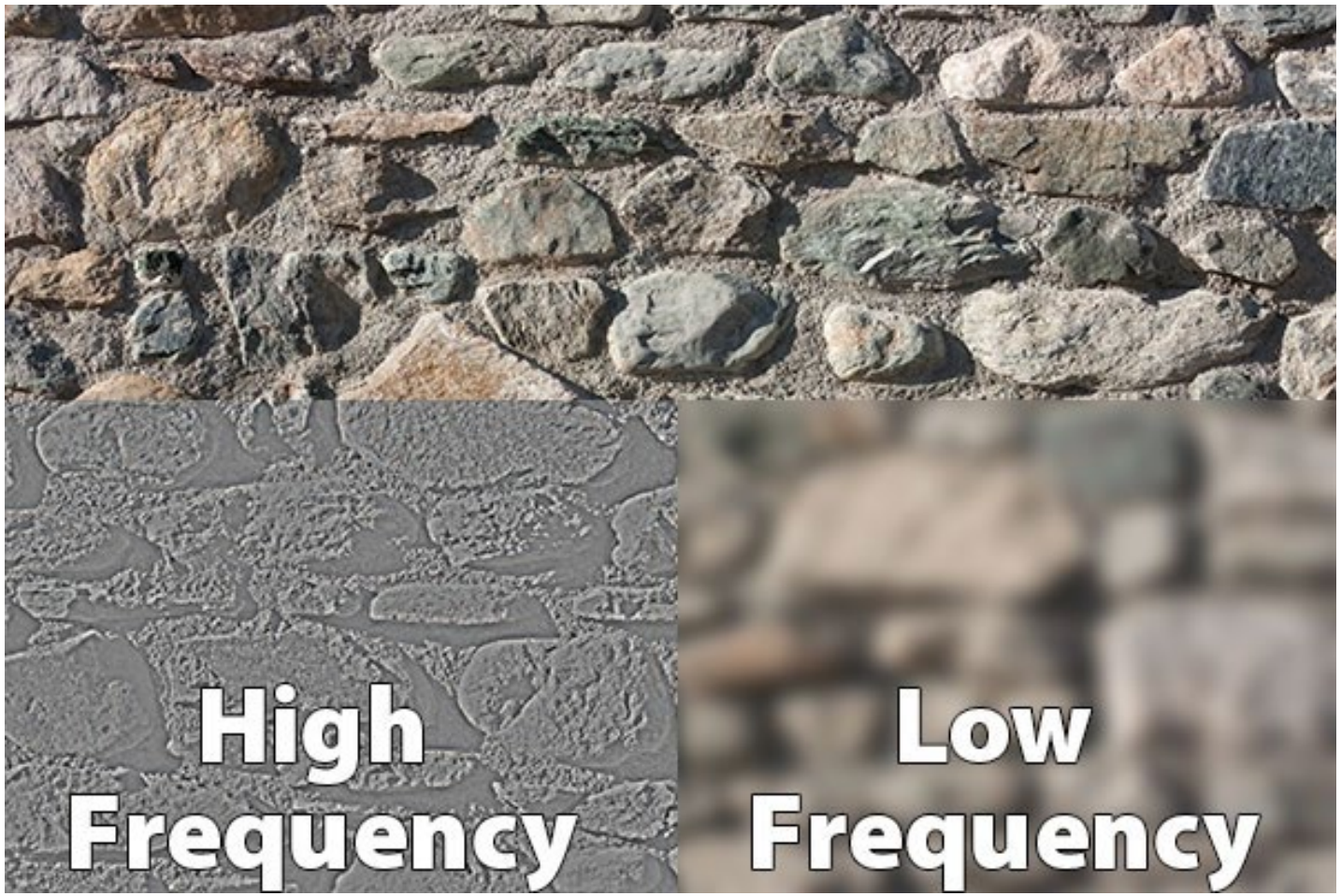
- Spatial frequency – how far apart (in pixels) are details
- Frequency-based filtering in EE involves running a signal through an element that suppresses some frequencies



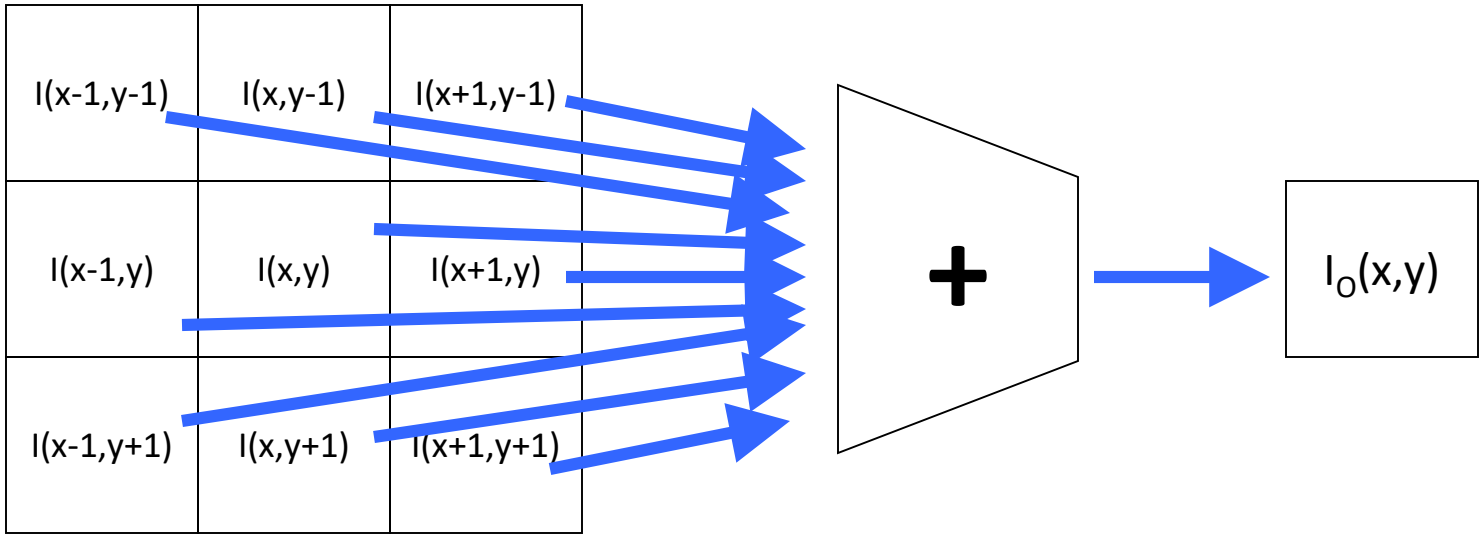
- It's the same in image filtering
  - But what do "frequency sensitive components" look like?

We will discuss frequency-domain processing images in detail, but for now think about image features that change rapidly as we move across the image, and those that change slowly

Most often, we are concerned with *spatial* frequencies, $\frac{\partial I}{\partial x}$ and $\frac{\partial I}{\partial y}$, rather than time-based frequencies



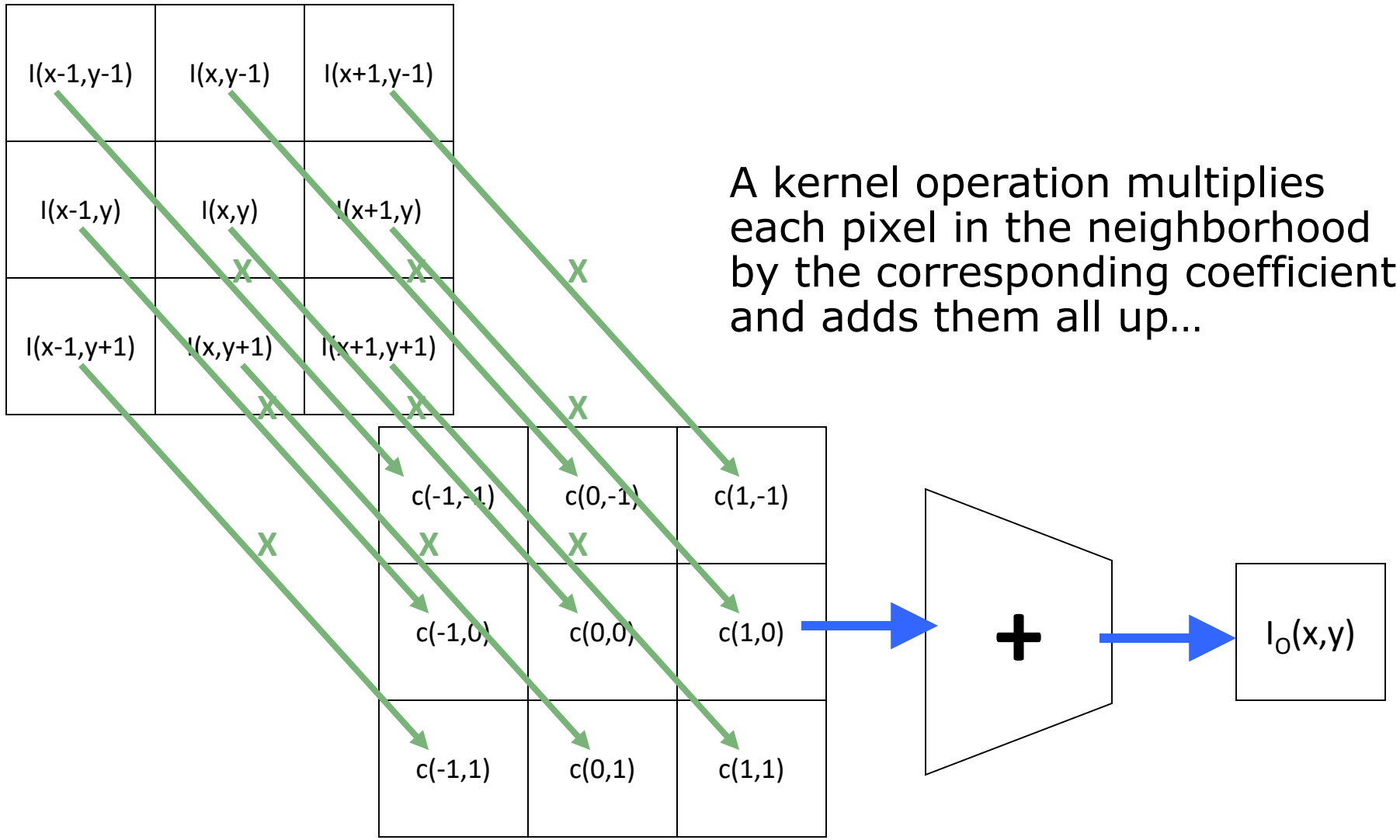
To implement filters, we define *Neighborhood Operations* that are calculated on pixel values in the vicinity ("neighborhood") of an image location to find its new value



A neighborhood operation computes the value of the pixel in the new image from value of the pixel *and its neighbors* in the old image

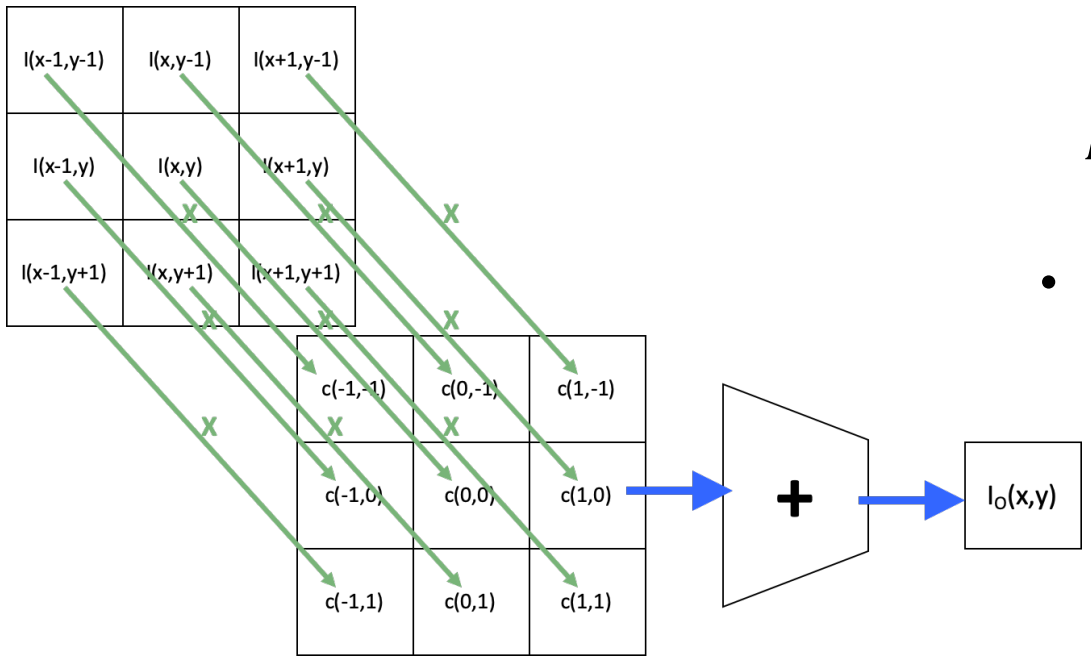
It's not possible to affect the spatial frequency characteristics of an image by operating only on individual pixels (point operations)

The basics of a Kernel operation



A kernel operation multiplies each pixel in the neighborhood by the corresponding coefficient and adds them all up…

Kernel operations are just an implementation of 2D convolution



- The two-dimensional convolution of a signal *f* with a kernel is defined as:

$$I_o(x,y) = \omega * f(x,y) = \sum_{i=-M}^{M} \sum_{j=-M}^{M} w(i,j)\, f(x-i, y-j)$$

- Most image processing kernels are symmetric so we don't usually worry about the usual inversion in index order for the kernels

Kernel operation characteristics

- Any kernel operation can be specified in terms of the kernel (the coefficients)
  - There are many kernels, with various properties
- Remember, you have to multiply each coefficient by each pixel in the neighborhood for every pixel in the output image
- Depending on the implementation, the intermediate calculations should be done as doubles to prevent overflow
  - usually have to scale the image somehow for display

The two simplest kernels achieve image smoothing or low-pass filtering

- These achieve *image smoothing* – every pixel is replaced by an average of the pixels in the 3x3 neighborhood
  - the one on the left is called a box filter
  - the one on the right weights the center pixels a little bit more – which seems right

$\frac{1}{9} \times$

1	1	1
1	1	1
1	1	1

$\frac{1}{16} \times$

1	2	1
2	4	2
1	2	1

a b

FIGURE 3.34 Two 3×3 smoothing (averaging) filter masks. The constant multiplier in front of each mask is equal to the sum of the values of its coefficients, as is required to compute an average.

Kernel sizes

- We often use 3x3 kernels for simplicity
  - also 5x5, 7x7, 15x15, 31x31, 51x51, whatever
- Here are two 5x5 smoothing masks:
  - For "equalized output", need to divide by 25 for the left one and 81 for the right one

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

1	2	3	2	1
2	4	6	4	2
3	6	9	6	3
2	4	6	4	2
1	2	3	2	1

```
# GaussFilter.py - a simple Gaussian filter demo - poorly coded in Python!!!
# Created on Thu Aug 29 15:17:05 2019
# @author: crjones4

import numpy as np
import cv2

# Load an image in grayscale
img = cv2.imread('C:\\Data\\spock.bmp', cv2.IMREAD_GRAYSCALE)
height = len(img)
width = len(img[0])

# create an output image
res = np.zeros((height, width, 1), dtype = "uint8")

# apply a kernel
kernel = [1, 4, 7, 4, 1, \
          4, 16, 26, 16, 4,\
          7, 26, 41, 26, 7,\
          4, 16, 26, 16, 4, \
          1, 4, 7, 4, 1]
ksize = 5
denominator = 273.0
```

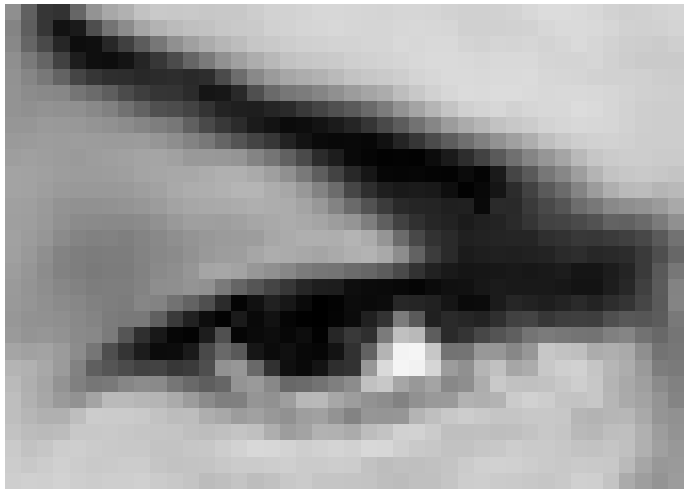
```
for r in range(height-ksize):
    for c in range(width-ksize):
        pixel = 0
        for y in range(ksize):
            for x in range(ksize):
                pixel += kernel[y*ksize+x]*img[r+y-ksize+1, c+x-ksize+1]
        res[r, c] = min(255, int(abs(pixel)/denominator));

# show image at 1/2 scale
cv2.namedWindow('INPUT', flags=cv2.WINDOW_NORMAL)
cv2.imshow('INPUT',img)
cv2.resizeWindow('INPUT', (int(width/2), int(height/2)))
cv2.namedWindow('RES', flags=cv2.WINDOW_NORMAL)
cv2.imshow('RES',res)
cv2.resizeWindow('RES', (int(width/2), int(height/2)))
cv2.waitKey(0)
cv2.destroyAllWindows()
```

```
# GaussFilter.py - a simple Gaussian filter demo - poorly coded in Python!!!
# Created on Thu Aug 29 15:17:05 2019
# @author: crjones4

import numpy as np
import cv2

# Load an image in grayscale
img = cv2.imread('C:\\Data\\spock.bmp', cv2.IMREAD_GRAYSCALE)
height = len(img)
width = len(img[0])

# create an output image
res = np.zeros((height, width, 1), dtype = "uint8")

# apply a kernel
kernel = [1, 4, 7, 4, 1, \
          4, 16, 26, 16, 4,\
          7, 26, 41, 26, 7,\
          4, 16, 26, 16, 4, \
          1, 4, 7, 4, 1]
ksize = 5
denominator = 273.0
```

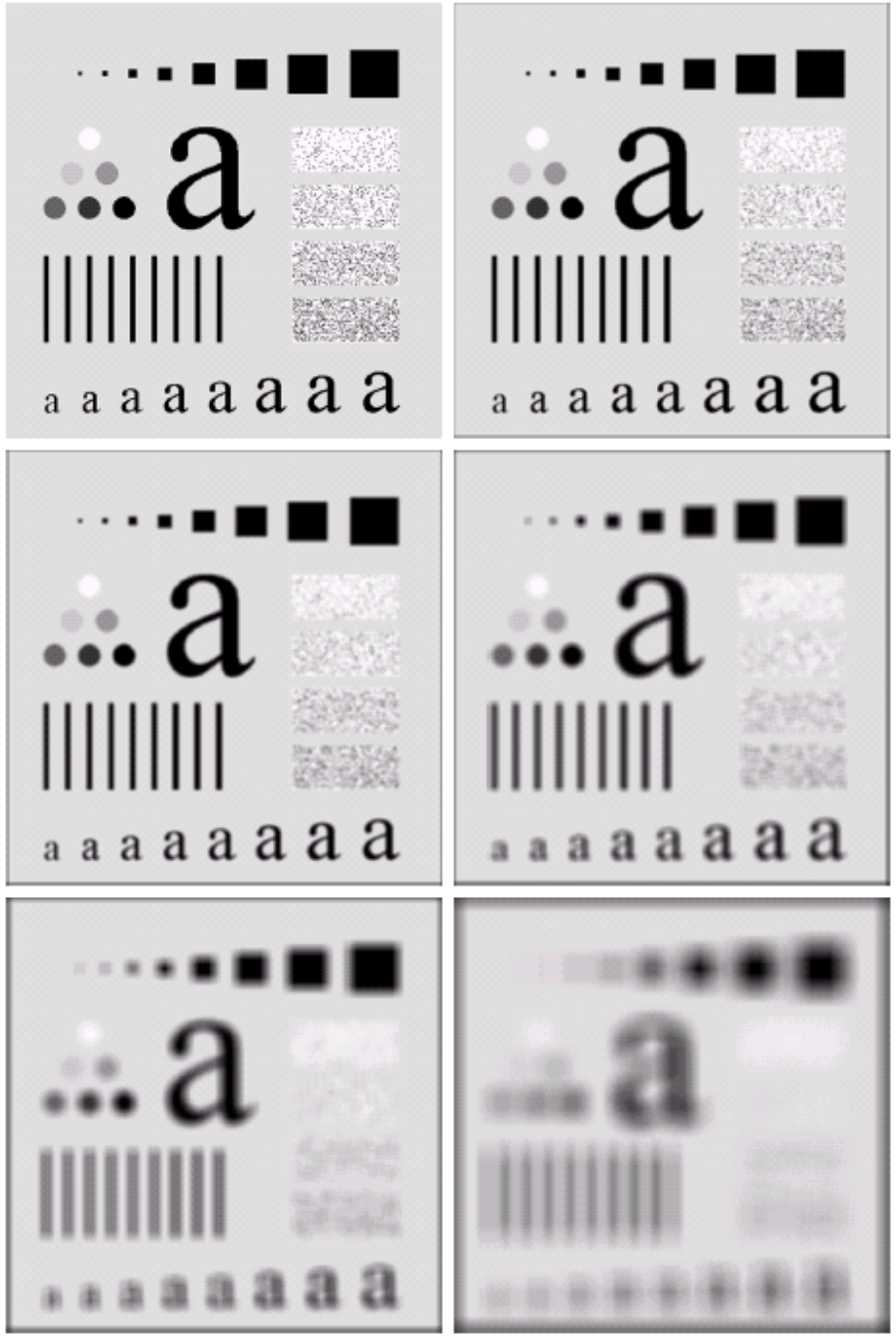
```
for r in range(height-ksize):
    for c in range(width-ksize):
        pixel = 0
        for y in range(ksize):
            for x in range(ksize):
                pixel += kernel[y*ksize+x]*img[r+y-ksize+1, c+x-ksize+1]
        res[r, c] = min(255, int(abs(pixel)/denominator));

# show image at 1/2 scale
cv2.namedWindow('INPUT', flags=cv2.WINDOW_NORMAL)
cv2.imshow('INPUT',img)
cv2.resizeWindow('INPUT', (int(width/2), int(height/2)))
cv2.namedWindow('RES', flags=cv2.WINDOW_NORMAL)
cv2.imshow('RES',res)
cv2.resizeWindow('RES', (int(width/2), int(height/2)))
cv2.waitKey(0)
cv2.destroyAllWindows()
```

```
# GaussFilter.py - a simple Gaussian filter demo - poorly coded in Python!!!
# Created on Thu Aug 29 15:17:05 2019
# @author: crjones4

import numpy as np
import cv2

# Load an image in grayscale
img = cv2.imread('C:\\Data\\spock.bmp', cv2.IMREAD_GRAYSCALE)
height = len(img)
width = len(img[0])

# create an output image
res = np.zeros((height, width, 1), dtype = "uint8")

# apply a kernel
kernel = [1, 4, 7, 4, 1, \
          4, 16, 26, 16, 4,\
          7, 26, 41, 26, 7,\
          4, 16, 26, 16, 4, \
          1, 4, 7, 4, 1]
ksize = 5
denominator = 273.0
```

```
for r in range(height-ksize):
    for c in range(width-ksize):
        pixel = 0
        for y in range(ksize):
            for x in range(ksize):
                pixel += kernel[y*ksize+x]*img[r+y-ksize+1, c+x-ksize+1]
        res[r, c] = min(255, int(abs(pixel)/denominator));

# show image at 1/2 scale
cv2.namedWindow('INPUT', flags=cv2.WINDOW_NORMAL)
cv2.imshow('INPUT',img)
cv2.resizeWindow('INPUT', (int(width/2), int(height/2)))
cv2.namedWindow('RES', flags=cv2.WINDOW_NORMAL)
cv2.imshow('RES',res)
cv2.resizeWindow('RES', (int(width/2), int(height/2)))
cv2.waitKey(0)
cv2.destroyAllWindows()
```



The effect of smoothing masks

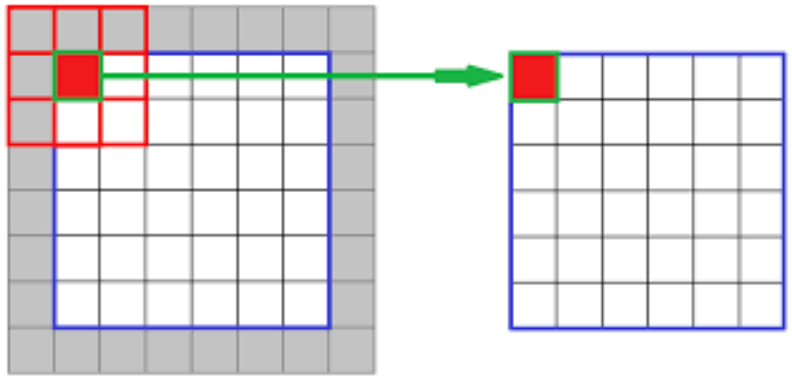


- This image of a *resolution test target* has been filtered by more and more dramatic smoothing filters
  - 3x3, 5x5, 9x9, 15x15 and 35x35
- Note the loss of crisp focus
- Note the dark margin

a b
c d
e f

FIGURE 3.35 (a) Original image, of size 500 × 500 pixels. (b)–(f) Results of smoothing with square averaging filter masks of sizes $n = 3, 5, 9, 15$, and 35, respectively. The black squares at the top are of sizes 3, 5, 9, 15, 25, 35, 45, and 55 pixels, respectively; their borders are 25 pixels apart. The letters at the bottom range in size from 10 to 24 points, in increments of 2 points; the large letter at the top is 60 points. The vertical bars are 5 pixels wide and 100 pixels high; their separation is 20 pixels. The diameter of the circles is 25 pixels, and their borders are 15 pixels apart; their gray levels range from 0% to 100% black in increments of 20%. The background of the image is 10% black. The noisy rectangles are of size 50 × 120 pixels.

Applying a kernel of size N results in the valid portion of the output image shrinking by $\frac{N-1}{2}$ on each side



We can either:

- ignore the outer margins in the original image (may show up dark)
- shrink the output image
- expand the input image by either duplicating the boundary cells or some other means

General equation for smoothing

$$Img_{out}(x,y) = \frac{\sum_{i=0}^{M-1}\sum_{j=0}^{M-1} Img_{in}\left(x+i-\frac{M-1}{2}, y+j-\frac{M-1}{2}\right)c(i,j)}{\sum_{i=0}^{M-1}\sum_{j=0}^{M-1} c(i,j)}$$

- The numerator is the weighted sum of the pixels in the neighborhood
- The denominator is the sum of the weights
  - so that the image average stays about the same
- This equation is valid for any arithmetic kernel function

Different types of noise can appear in images

- Random noise

- Salt-and-pepper noise

- Image artifacts

Often, noise comes from the electronic processing

Filtering random noise is usually done using a smoothing filter

- If we don't know anything else about the noise, the best bet is to use a general smoothing filter
- How big? Just big enough to do the job
  - Any smoothing operation removes some detail
- For equal-weighted kernels (all 1), smaller kernels have narrower smoothing effects
- If the weights are not equal, the important thing is how far from the center the weights "fall off" or get smaller

Low-pass filtering coefficients

- Equal-weighted smoothing kernels (all 1) are easy, but it seems "right" that the weights should be less at the edge than in the center
- So how do we compute the weights?
- One option is to mimic what happens in an optical defocus
  - When a lens is defocused
- Optical defocus makes a point source of light seem to have a Gaussian profile

Mask coefficients for Gaussian low-pass filtering

$$\frac{1}{1000}\times$$

.01965	.23941	1.0729	1.7690	1.0729	.23941	.01965
.23941	2.9166	13.071	21.551	13.071	2.9166	.23941
1.0729	13.071	58.581	96.585	58.581	13.071	1.0729
1.7690	21.551	96.585	159.24	96.585	21.551	1.7690
1.0729	13.071	58.581	96.585	58.581	13.071	1.0729
.23941	2.9166	13.071	21.551	13.071	2.9166	.23941
.01965	.23941	1.0729	1.7690	1.0729	.23941	.01965

(i_c, j_c)

- Here is a 7x7 Gaussian low-pass (or smoothing) filter kernel, with σ=1

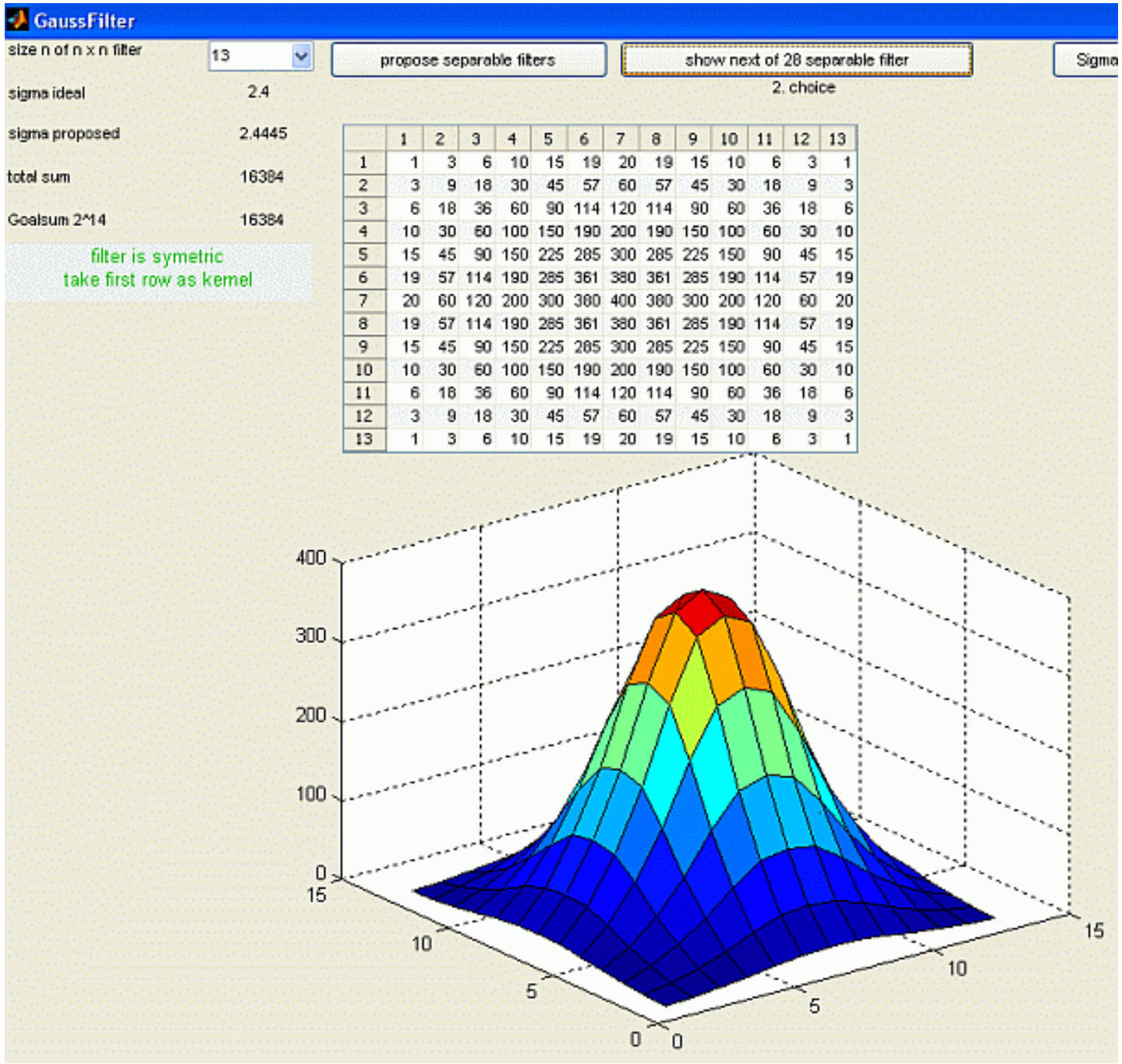
$$c(i,j) = Ke^{\frac{-\left((i-i_c)^2+(j-j_c)^2\right)}{2\sigma^2}}$$

Alternative formulation – integer coefficients

$$\frac{1}{273} \times$$

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

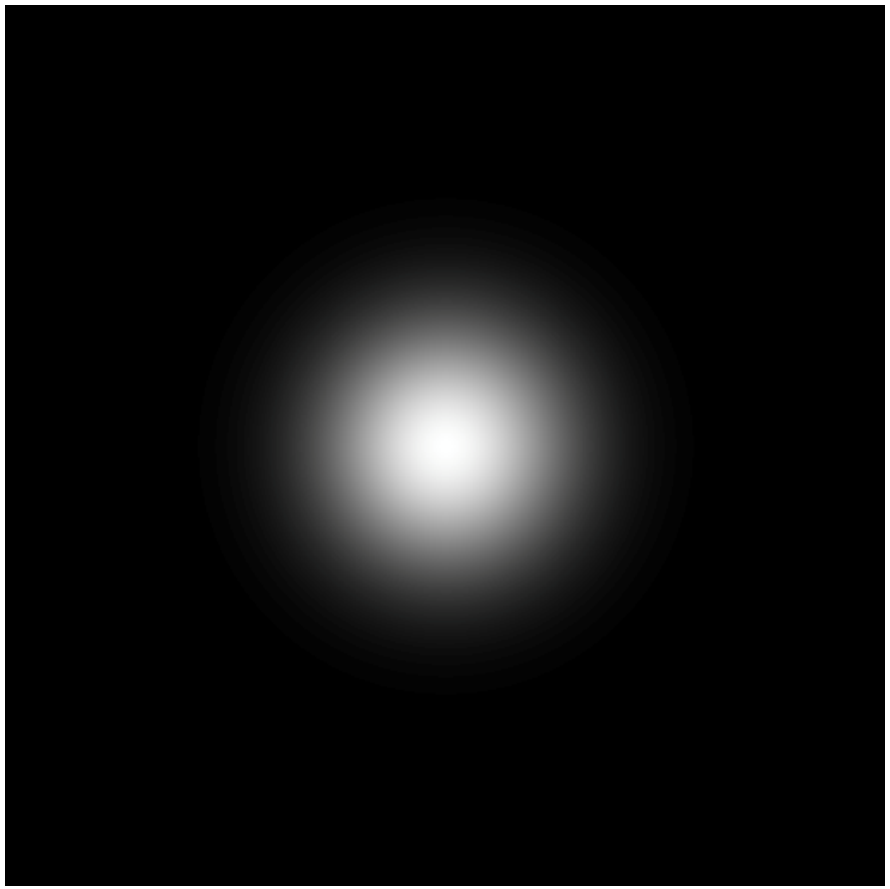
- This one is a 5x5 Gaussian low-pass (or smoothing) filter kernel, with σ=1



	1	2	3	4	5	6	7	8	9	10	11	12	13
1	1	3	6	10	15	19	20	19	15	10	6	3	1
2	3	9	18	30	45	57	60	57	45	30	18	9	3
3	6	18	36	60	90	114	120	114	90	60	36	18	6
4	10	30	60	100	150	190	200	190	150	100	60	30	10
5	15	45	90	150	225	285	300	285	225	150	90	45	15
6	19	57	114	190	285	361	380	361	285	190	114	57	19
7	20	60	120	200	300	380	400	380	300	200	120	60	20
8	19	57	114	190	285	361	380	361	285	190	114	57	19
9	15	45	90	150	225	285	300	285	225	150	90	45	15
10	10	30	60	100	150	190	200	190	150	100	60	30	10
11	6	18	36	60	90	114	120	114	90	60	36	18	6
12	3	9	18	30	45	57	60	57	45	30	18	9	3
13	1	3	6	10	15	19	20	19	15	10	6	3	1

To get a picture, consider the coefficients in the Gaussian filter as an image – intensity reflects how much the location will affect the center pixel in the result image

- Near the center, the image intensity will have a major effect in the output
- Farther away, the effect is less (but not zero)
- The circularity means that degree of influence is only dependent on radius, not direction



Successively greater values of sigma – width of the Gaussian curve – produce more smoothing (note that the size of the kernel needs to increase too)

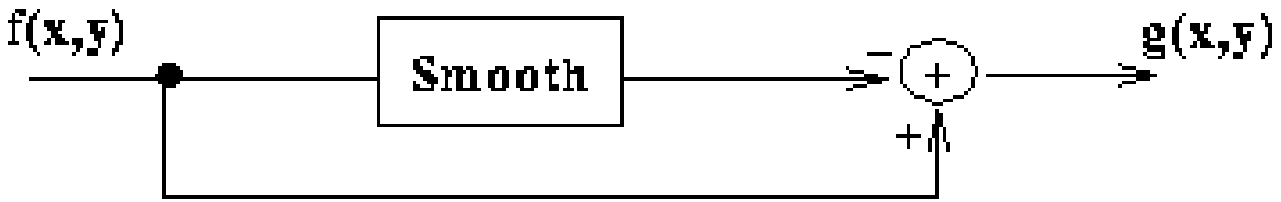
- Remember that the Gaussian kernel has two interrelated parameters: size N and Gaussian width σ
- One rule of thumb is that the filter should be at least six times sigma wide – if $\sigma = 2$ pixels, then the filter should be 13 by 13 or larger

Sharpening an Image

- "Image sharpening" refers to increasing the apparent crispness of the image
  - This usually means that *edges* (changes in intensity) in the image get more dramatic (higher contrast)
- This is usually done using an "unsharp mask"
  - Huh? Unsharp? But don't we want it to be *more* sharp?

Sharpening is making less smooth

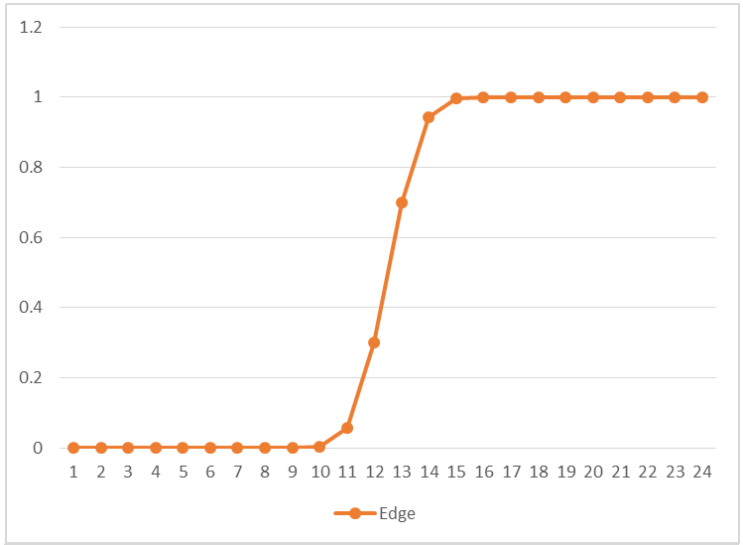
- We have kernels that make the image smoother
- So, what if we take the original image and subtract some of the smoothness?
  - Take the original image and subtract the smoothed image, pixel by pixel, with a scale factor so we don't subtract all the smoothness

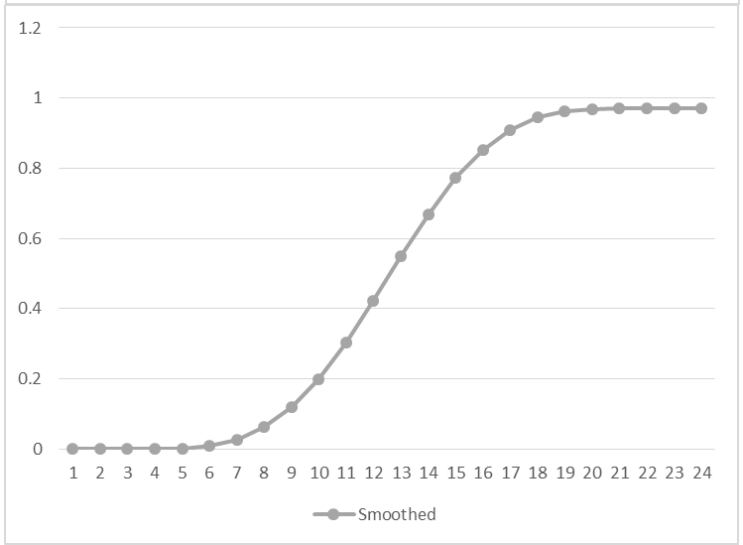


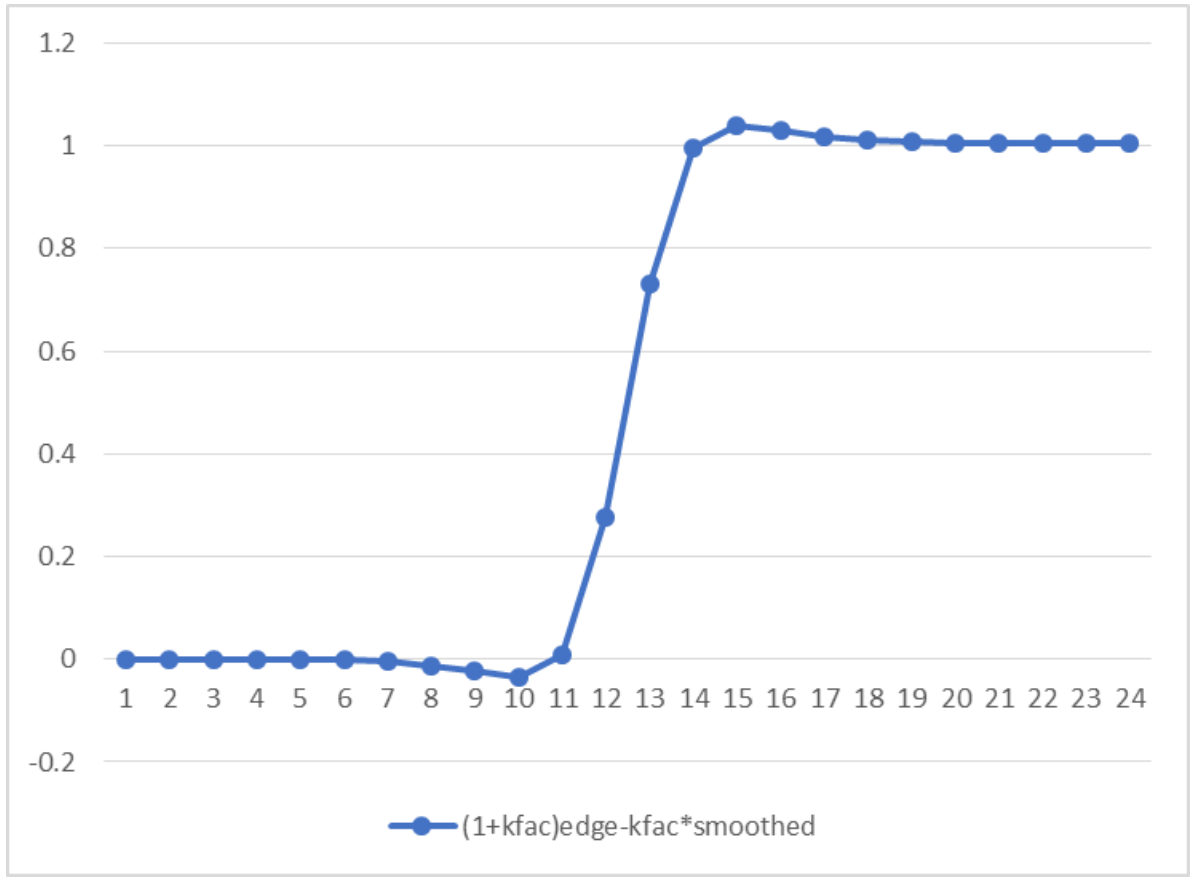
Original – k*smoothed = ?

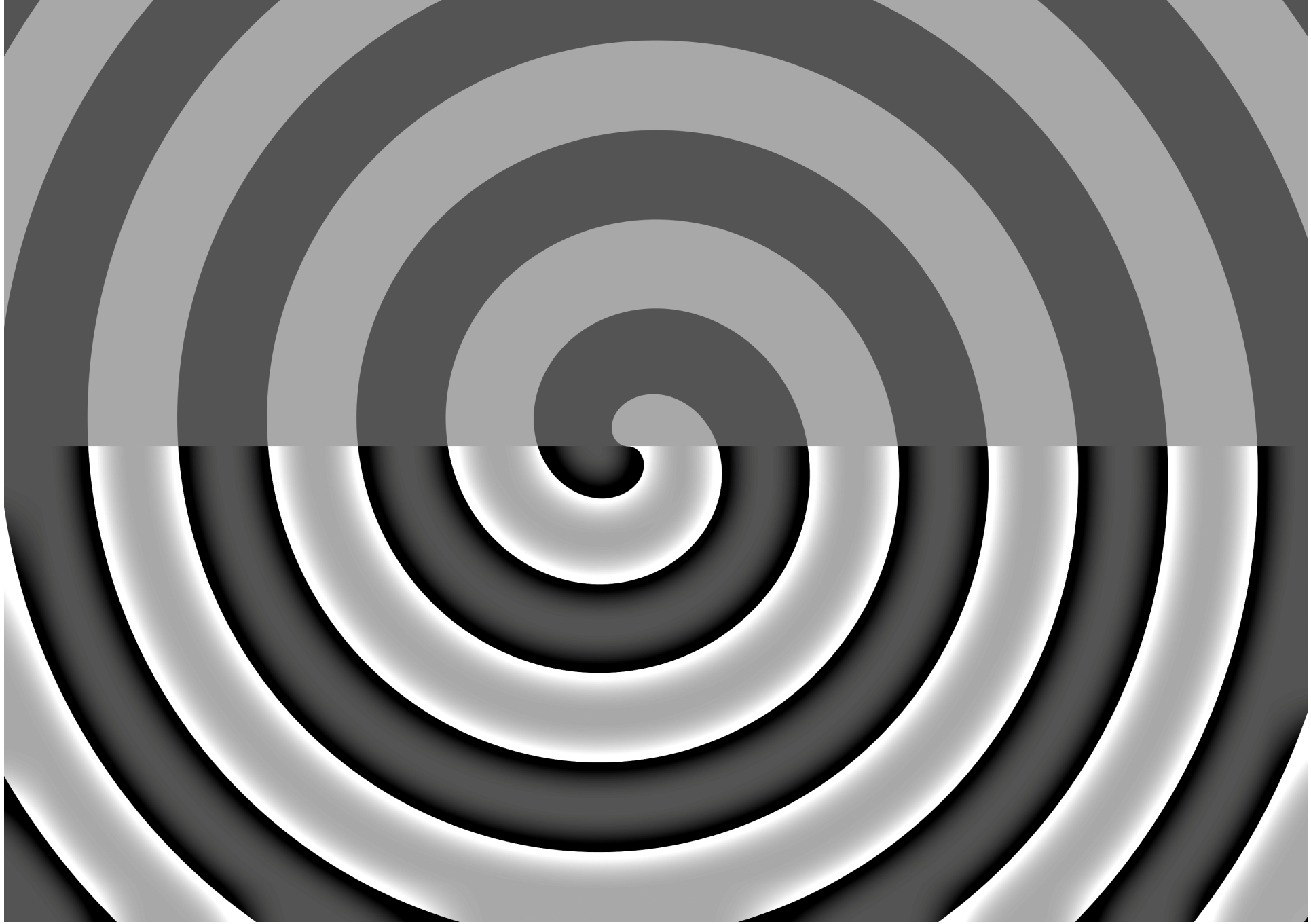
- Subtracting the smoothed image has an apparent sharpening effect
  - We are deemphasizing the lower spatial frequencies

Consider a single edge (in 1D) being sharpened by subtraction of the smoothed version









If we subtract ALL of the smoothed image, all that's left are the sharp transitions in the image

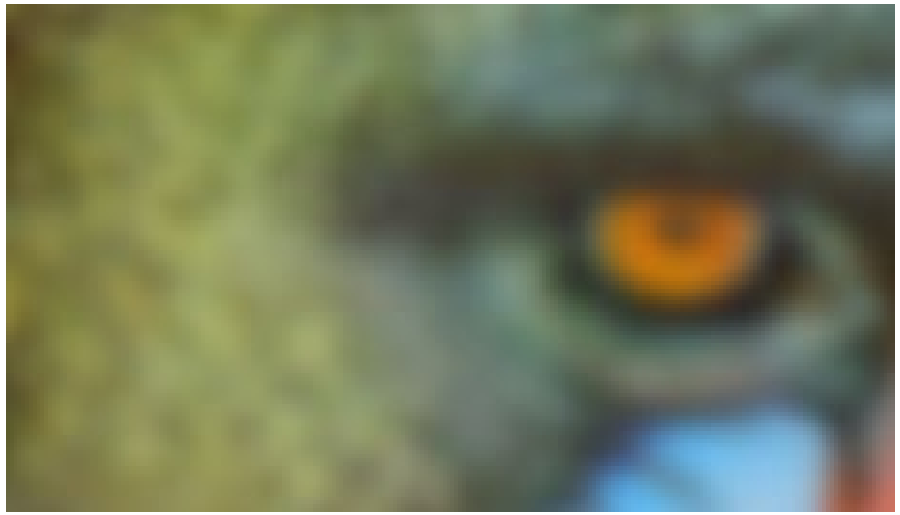
- 1 *

- This is called an *edge image*
- Bright areas are transitions in the original image

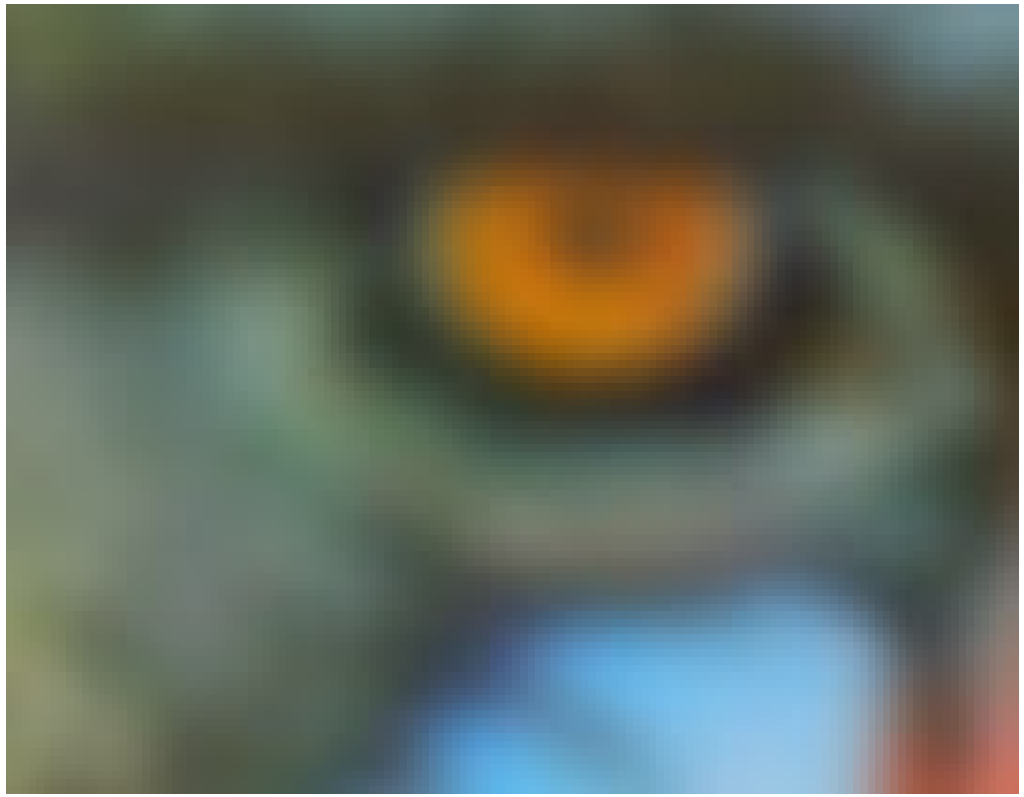
COLOR FILTERING

A color image contains three component planes; can we apply a kernel (for example, a Gaussian filter) to the color planes separately and use the result?

A color image contains three component planes; can we apply a kernel (for example, a Gaussian filter) to the color planes separately and use the result?



A color image contains three component planes; can we apply a kernel (for example, a Gaussian filter) to the color planes separately and use the result?



Consider an image with many colors and transitions between them

Smoothing with a Gaussian kernel seems to produce an appropriately blurred image ($\sigma = 2$)

Smoothing with a Gaussian kernel seems to produce an appropriately blurred image ($\sigma = 2$)

Looking at the pixel level does reveal some oddities – is this still acceptable?

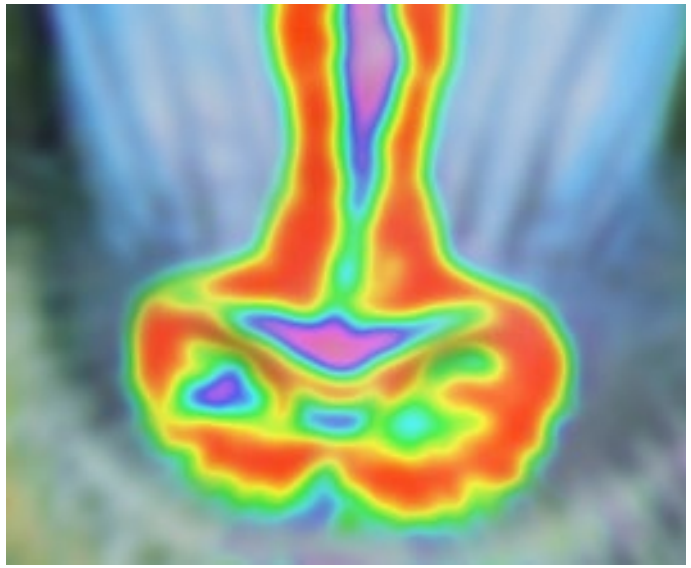
What happens if we apply smoothing and other linear kernels to the HSI planes? The intensity will respond well, but what about hue and saturation?

- What happens when we average adjacent color pixels with different hue values?
  - The average of red and blue might be green; does this make sense?

- Averaging saturation can be just as bad – changing pale colors into more vivid ones

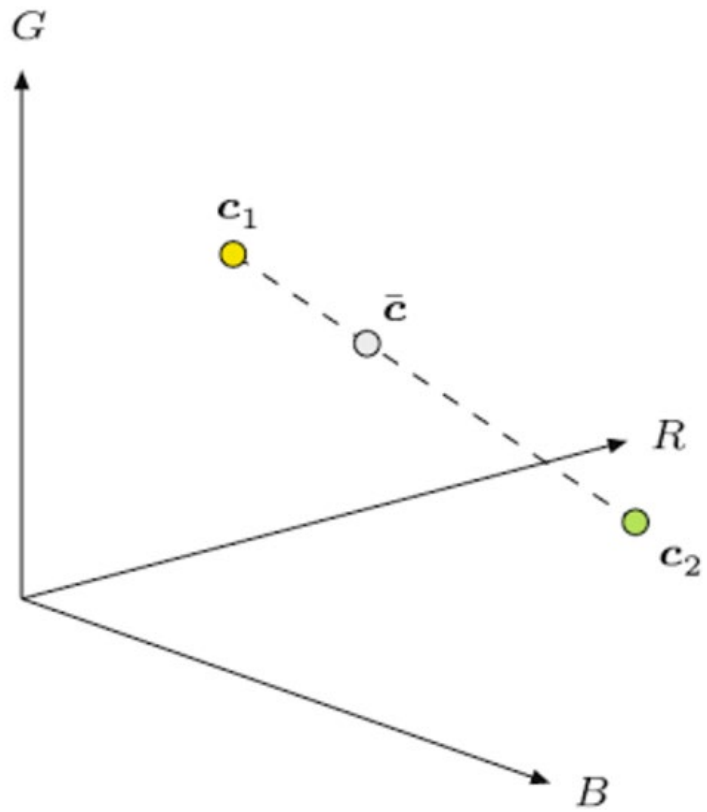
What happens if we apply smoothing and other linear kernels to the HSI planes? The intensity will respond well, but what about hue and saturation?

- What happens when we average adjacent color pixels with different hue values?
  - The average of red and blue might be green; does this make sense?

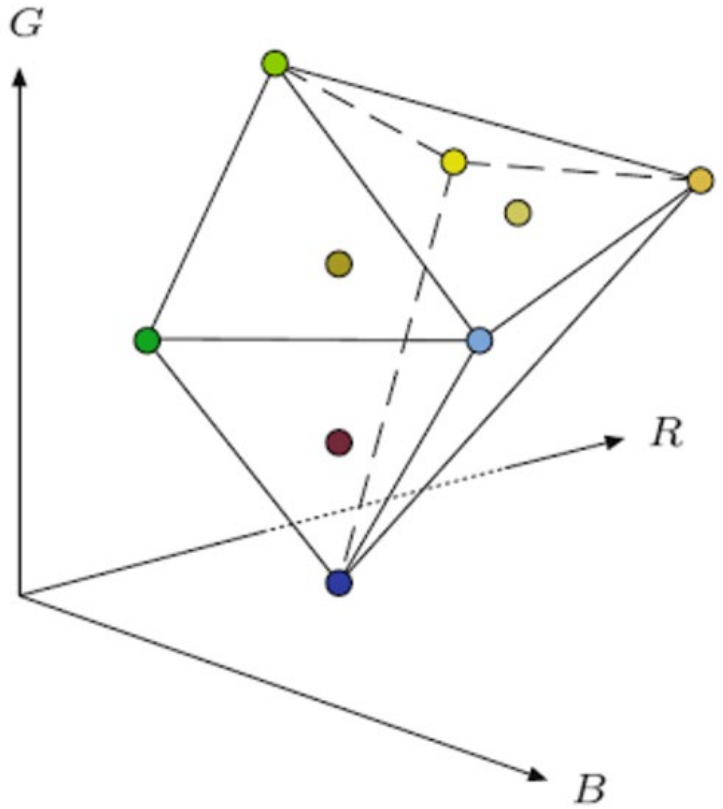


- Averaging saturation can be just as bad – changing pale colors into more vivid ones

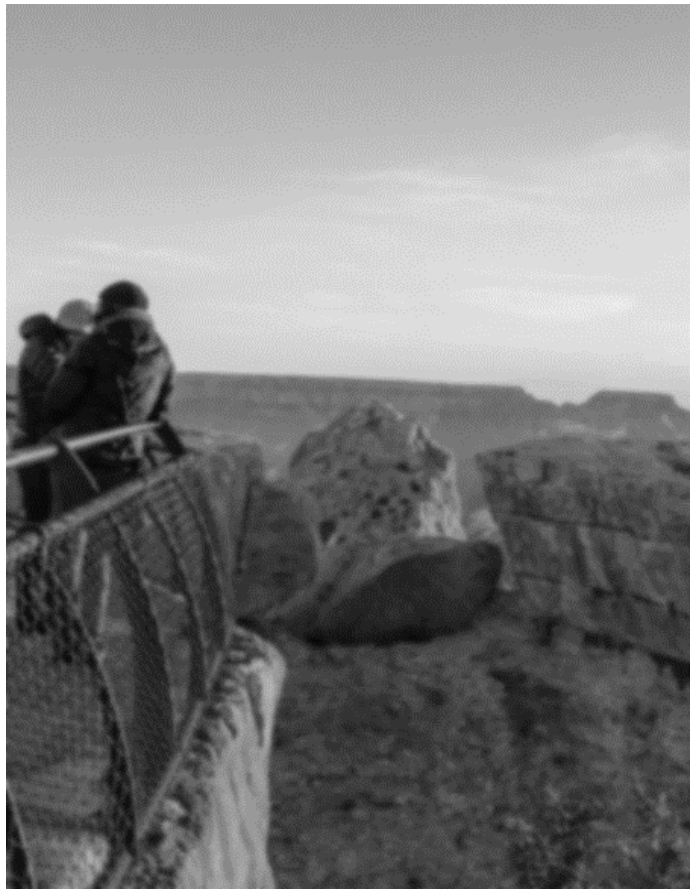
A weighted average of two RGB color pixel values will always result in a value somewhere on the line connecting the two points in RGB space



The result of the linear combination of several colors (as done by a kernel operation) will be within the polygon defined by the individual color points



Sharpening a color image using the unsharp mask is generally done by converting to HSI or HSV and applying the unsharp process to the intensity (or value) plane



Conclusions on filtering: smoothing RGB planes can be acceptable but not HSI planes

- Filtering methods that work with the vector quantities of the color values are also in existence
- For smoothing, it’s generally not worth the effort

Today's Objectives

- Gamma Correction
- Image Arithmetic
- Filtering
- Kernels
- Smoothing
- Gaussian Smoothing
- Filtering in Color Images