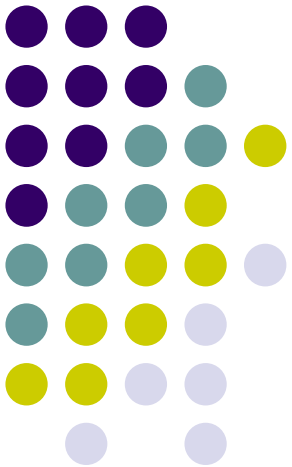
Digital Image Processing (CS/ECE 545)
Lecture 3: Point Operations (Part 2) & Filters (Part 1) (Neighborhood and Spatial Processing)

Prof Emmanuel Agu

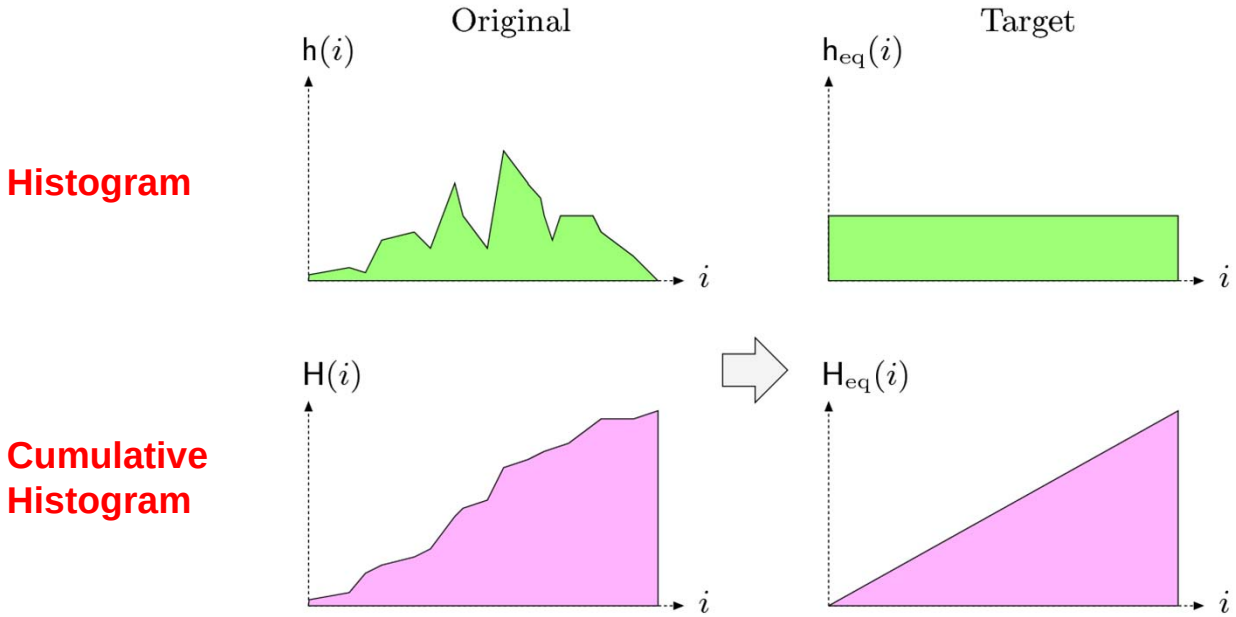
Computer Science Dept.

Worcester Polytechnic Institute (WPI)

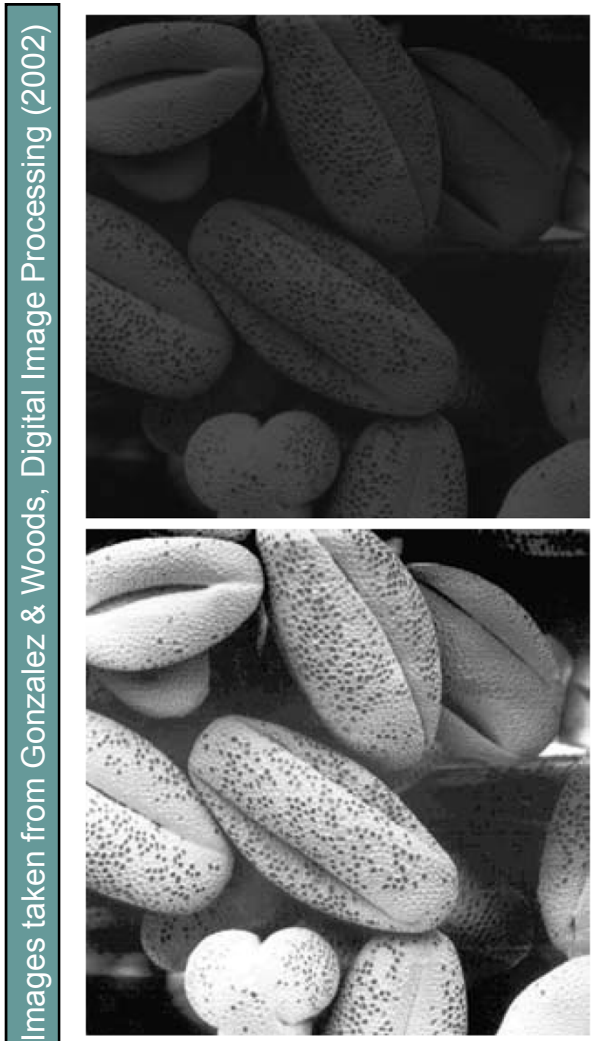


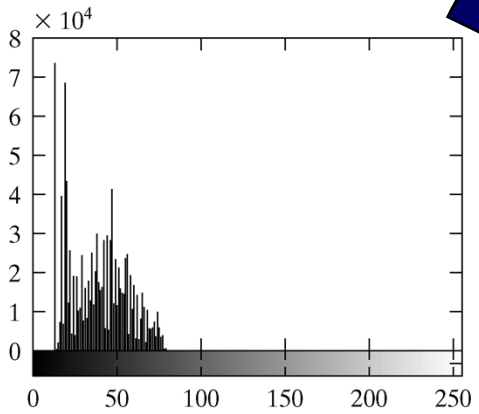
Recall: Histogram Equalization

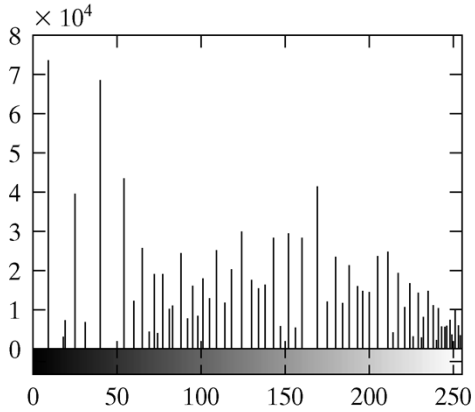
- Adjust 2 different images to make their histograms (intensity distributions) similar
- Apply a point operation that changes histogram of modified image into **uniform distribution**

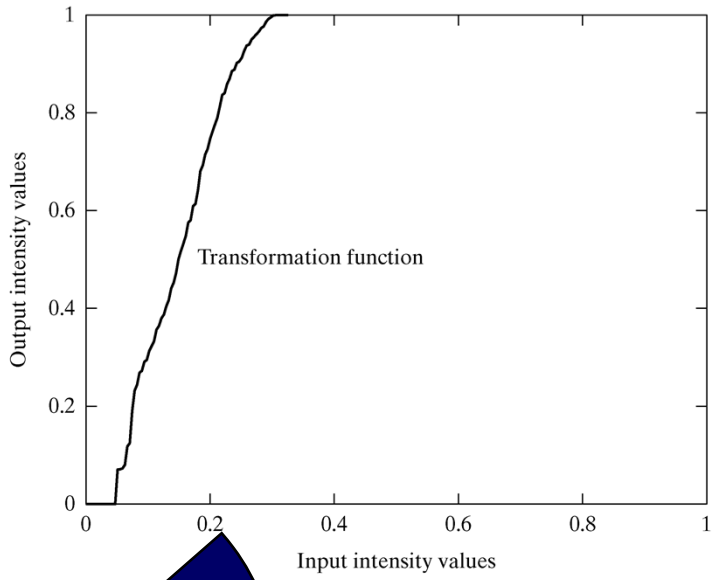


Recall: Equalization Transformation Function



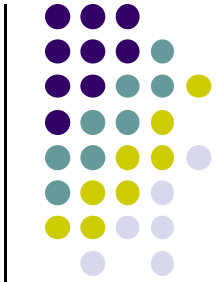






Images taken from Gonzalez & Woods, Digital Image Processing (2002)

Linear Histogram Equalization



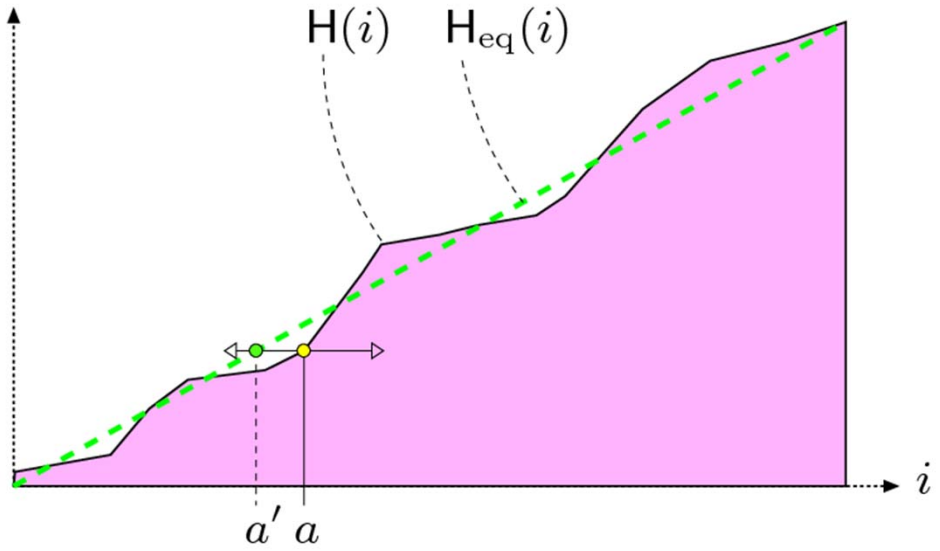
- Histogram cannot be made exactly flat – peaks cannot be increased or decreased by point operations.
- Following point operation makes histogram as flat as possible:

(assuming *M x N* image and pixels in range [0, *K* - 1])

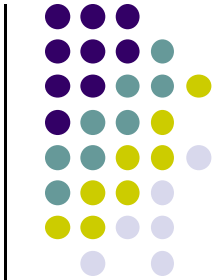
Point operation that returns Linear equalized value of *a*

$$f_{\text{eq}}(a) = \left\lfloor \mathsf{H}(a) \cdot \frac{K-1}{MN} \right\rfloor$$

Cumulative Histogram: Σ how many times intensity a occurs



Effects of Linear Histogram Equalization

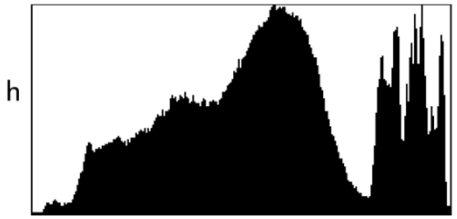


Original Image *I*

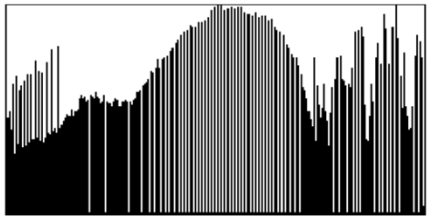
(a) (b)

Image *I'* after Linear Equalization

Original histogram



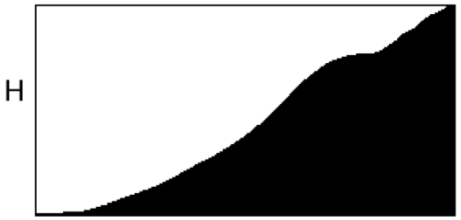
(c)



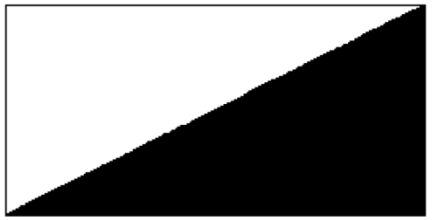
(d)

Histogram after Linear Equalization

Cumulative Histogram



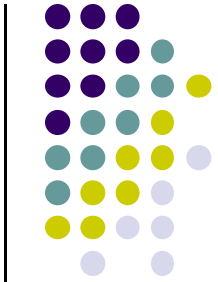
(e)



(f)

Cumulative Histogram After Linear Equalization

Sample Linear Equalization Code



```
1   public void run(ImageProcessor ip) {
2       int w = ip.getWidth();
3       int h = ip.getHeight();
4       int M = w * h;   // total number of image pixels
5       int K = 256;     // number of intensity values
6
7       // compute the cumulative histogram:
8       int[] H = ip.getHistogram();
9       for (int j = 1; j < H.length; j++) {
10          H[j] = H[j-1] + H[j];
11      }
12
13      // equalize the image:
14      for (int v = 0; v < h; v++) {
15          for (int u = 0; u < w; u++) {
16              int a = ip.get(u, v);
17              int b = H[a] * (K-1) / M;
18              ip.set(u, v, b);
19          }
20      }
21  }
```

Obtain histogram of image *ip*

Compute cumulative histogram in place

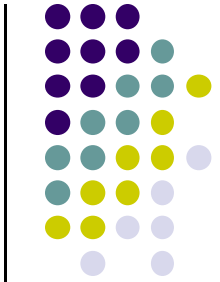
Get intensity value at (u,v)

Equalize pixel intensity

Cumulative Histogram: Σ how many times intensity a occurs

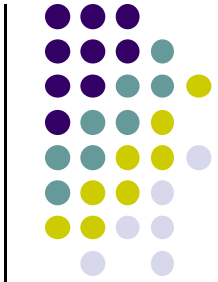
$$f_{\text{eq}}(a) = \left\lfloor \mathsf{H}(a) \cdot \frac{K-1}{MN} \right\rfloor$$

Histogram Specification



- Real images never show uniform distribution (unnatural)
- Most real images, distribution of pixel intensities is gaussian
- **Histogram specification**
  - modifies an image's histogram into an arbitrary intensity distribution (may not be uniform)
- Image 1's histogram can also be used as target for image 2
  - Why? Makes images taken by 2 different cameras to appear as if taken by same camera

Images and Probability



Histograms can be interpreted as probabilities.

Question: If I pick a pixel from an image at random, what is the probability that the pixel has intensity i?

Answer: $P(I(u,v) = i) = \frac{\text{\# pixels with value } i}{\text{total \# pixels}}$

Or, in terms of the histogram h:

$$P(I(u,v) = i) = h(i)/wh$$

Histogram Specification

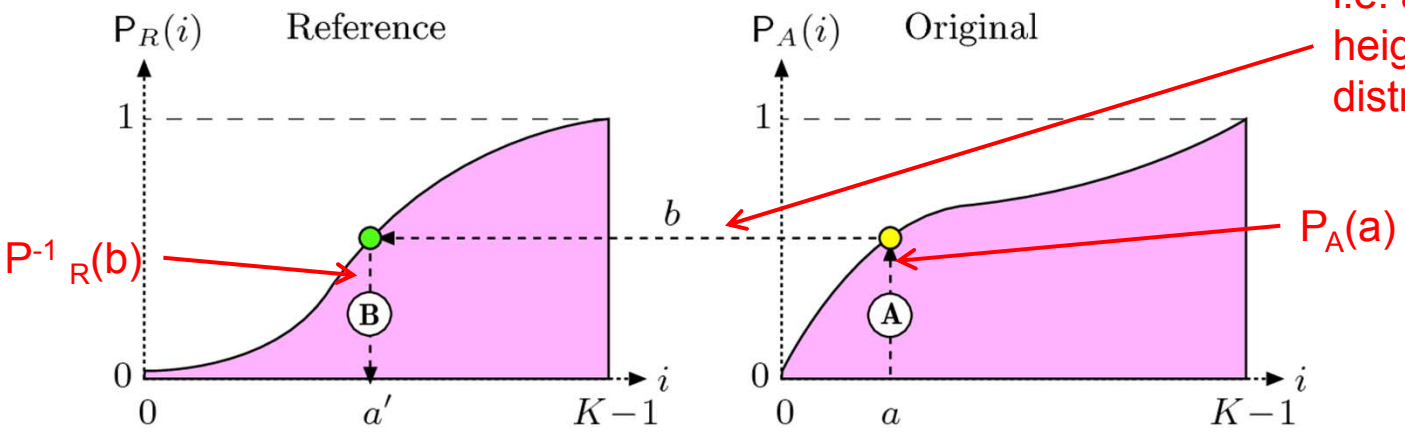
- Find a mapping such that distribution of *a* matches some reference distribution.i.e

$$a' = f_{\text{hs}}(a)$$

Mapping function: maps distribution on right to equivalent point (same height) On distribution on left

to convert original image I_A *into* $I_{A'}$ such that

$$\mathrm{P}_{A'}(i) \approx \mathrm{P}_R(i) \qquad \text{for } 0 \le i < K$$



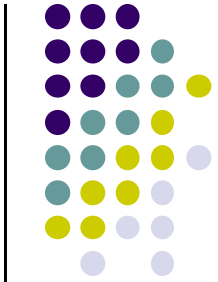
i.e. *a* and *a'* have same height (b) on different CDF distributions

$P^{-1}{}_R(b)$

$P_A(a)$

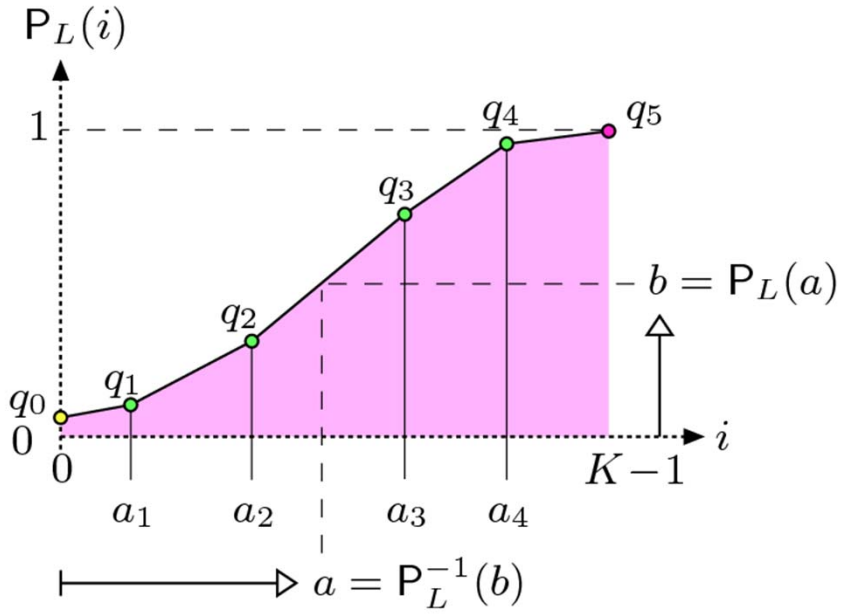
$$f_{\text{hs}}(a) = a' = \mathrm{P}_R^{-1}\big(\mathrm{P}_A(a)\big)$$

Adjusting Linear Distribution Piecewise



- In practice, reference distribution may be specified as a *piecewise linear* function

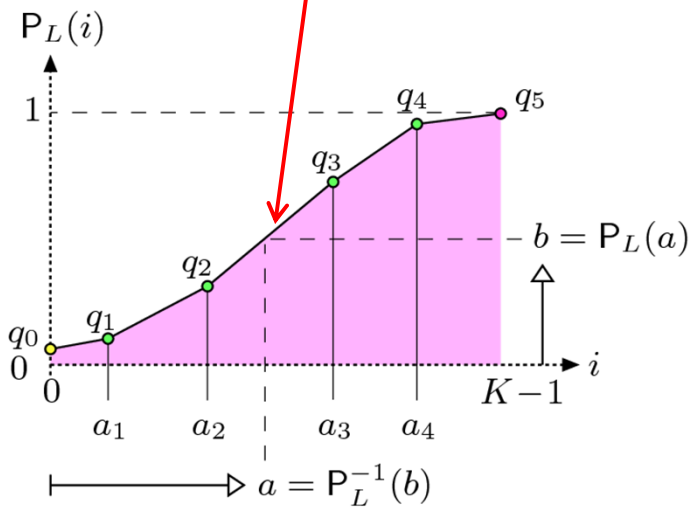
$$\mathcal{L} = [\langle a_0, q_0\rangle, \langle a_1, q_1\rangle, \ldots \langle a_k, q_k\rangle, \ldots \langle a_N, q_N\rangle]$$



- 2 endpoints are fixed $\langle 0, q_0\rangle$ and $\langle K-1, 1\rangle$

Adjusting Linear Distribution Piecewise

For each segment, linearly
Interpolate to find any value



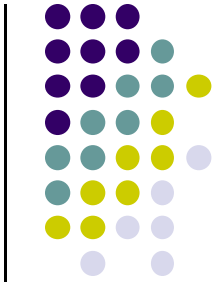
$$\mathsf{P}_L(i) = \begin{cases} q_m + (i - a_m) \cdot \dfrac{(q_{m+1} - q_m)}{(a_{m+1} - a_m)} & \text{for } 0 \le i < K-1 \\ 1 & \text{for } i = K-1 \end{cases}$$

$$\mathsf{P}_L^{-1}(b) = \begin{cases} 0 & \text{for } 0 \le b < \mathsf{P}_L(0) \\ a_n + (b - q_n) \cdot \dfrac{(a_{n+1} - a_n)}{(q_{n+1} - q_n)} & \text{for } \mathsf{P}_L(0) \le b < 1 \\ K-1 & \text{for } b \ge 1 \end{cases}$$

We also need the inverse mapping

$$n = \max\{j \in \{0, \ldots N-1\} \mid q_j \le b\}$$

Adjusting Linear Distribution Piecewise

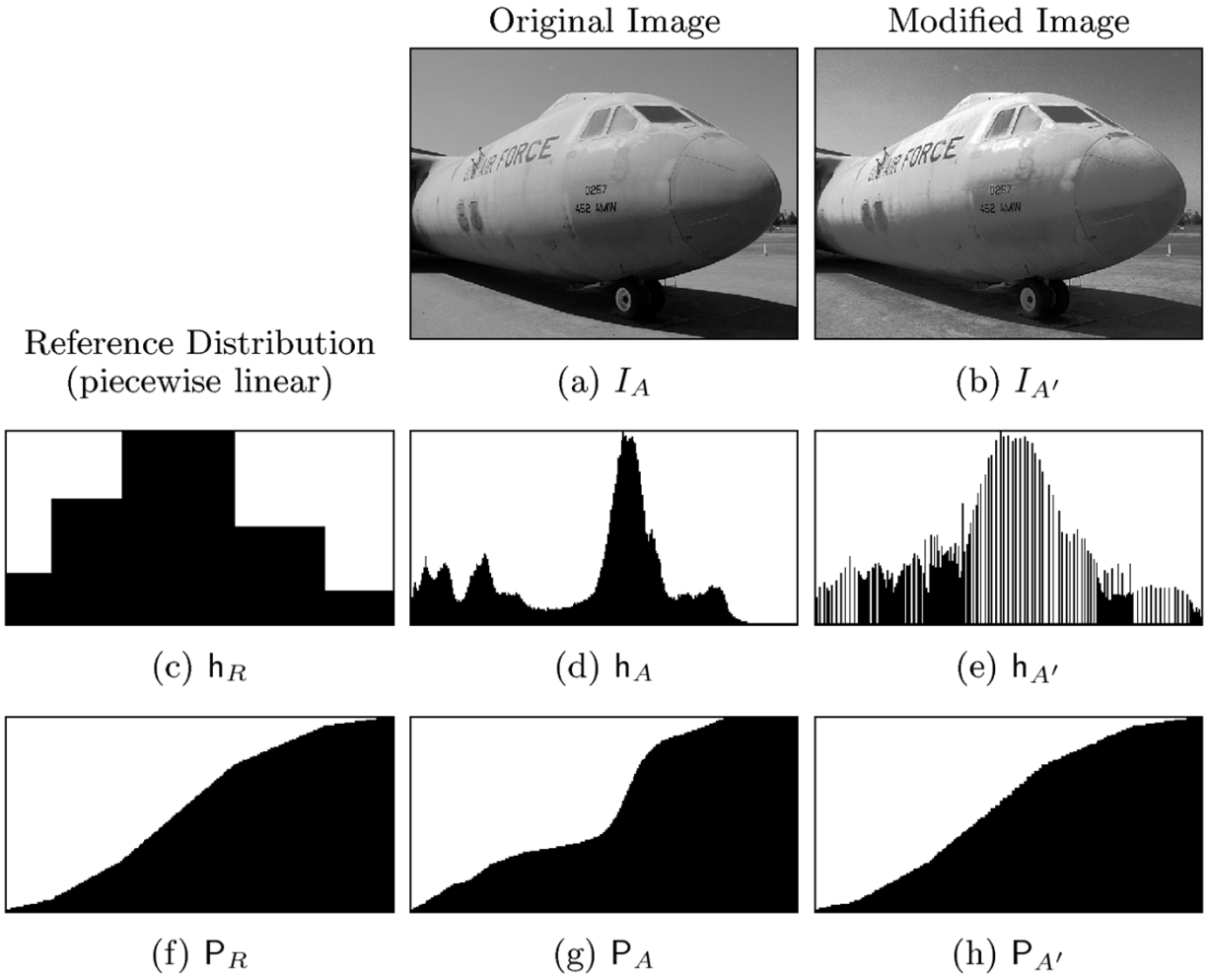


PiecewiseLinearHistogram(h_A, $\mathcal{L}_R$)

h_A: histogram of the original image.
$\mathcal{L}_R$: reference distribution function, given as a sequence of $N+1$ control points $\mathcal{L}_R = [\langle a_0, q_0\rangle, \langle a_1, q_1\rangle, \ldots \langle a_N, q_N\rangle]$, with $0 \le a_k < K$ and $0 \le q_k \le 1$.

Let $K \leftarrow \text{Size}(\mathsf{h}_A)$
Let $\mathsf{P}_A \leftarrow \text{Cdf}(\mathsf{h}_A)$ ▷ cdf for h_A (Alg. 5.1)
Create a table $f_{\text{hs}}[\,]$ of size K ▷ mapping function f_{hs}
for $a \leftarrow 0 \ldots (K-1)$ **do**
  $b \leftarrow \mathsf{P}_A(a)$
  if $(b \le q_0)$ **then**
    $a' \leftarrow 0$
  else if $(b \ge 1)$ **then**
    $a' \leftarrow K-1$
  else
    $n \leftarrow N-1$
    while $(n \ge 0) \wedge (q_n > b)$ **do** ▷ find line segment in $\mathcal{L}_R$
      $n \leftarrow n-1$
    $a' \leftarrow a_n + (b-q_n) \cdot \dfrac{(a_{n+1}-a_n)}{(q_{n+1}-q_n)}$ ▷ see Eqn. (5.23)
  $f_{\text{hs}}[a] \leftarrow a'$
return f_{hs}.

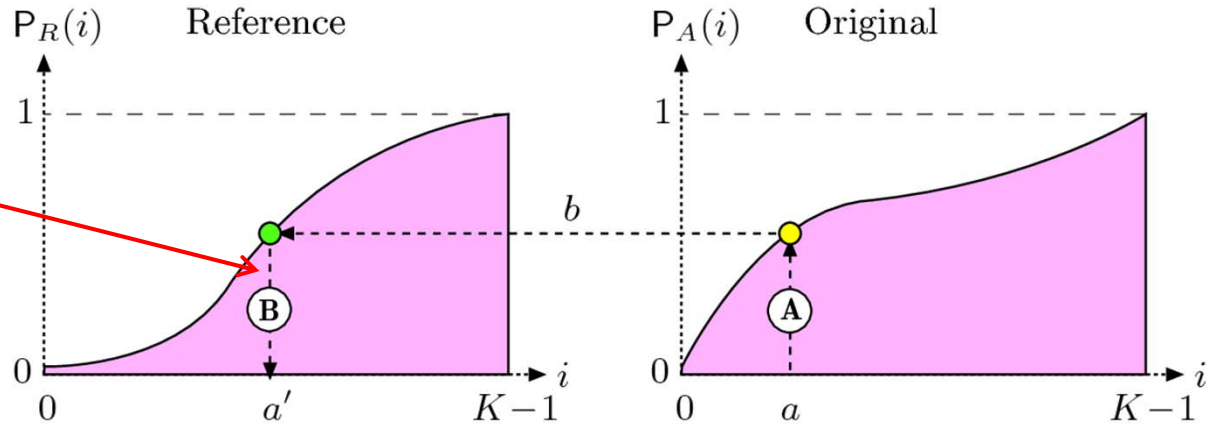
Adjusting Linear Histogram Piecewise



Histogram Matching

- Prior method needed reference distribution to be invertible

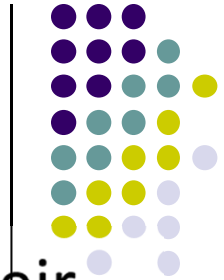
Has to be invertible



$$f_{\text{hs}}(a) = a' = \mathsf{P}_R^{-1}\big(\mathsf{P}_A(a)\big)$$

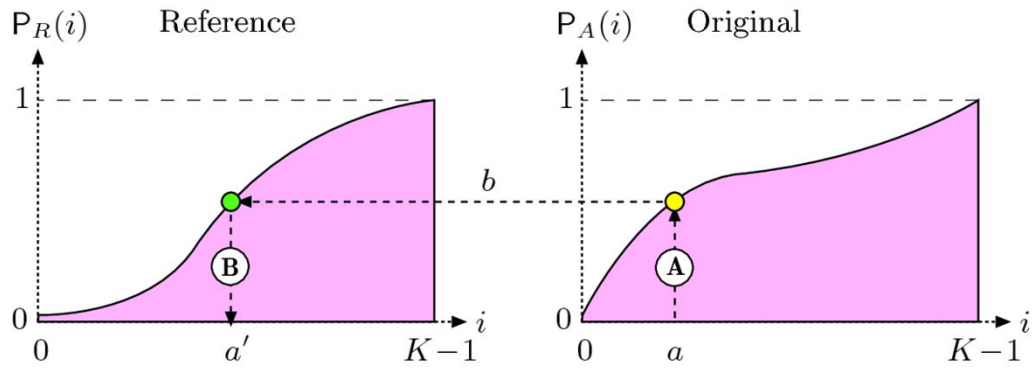
- What if reference histogram is not invertible?
- For example not invertible if histogram has some intensities that occur with probability 0? i.e. p(k) = 0
- Use different method called **histogram matching**

Histogram Matching

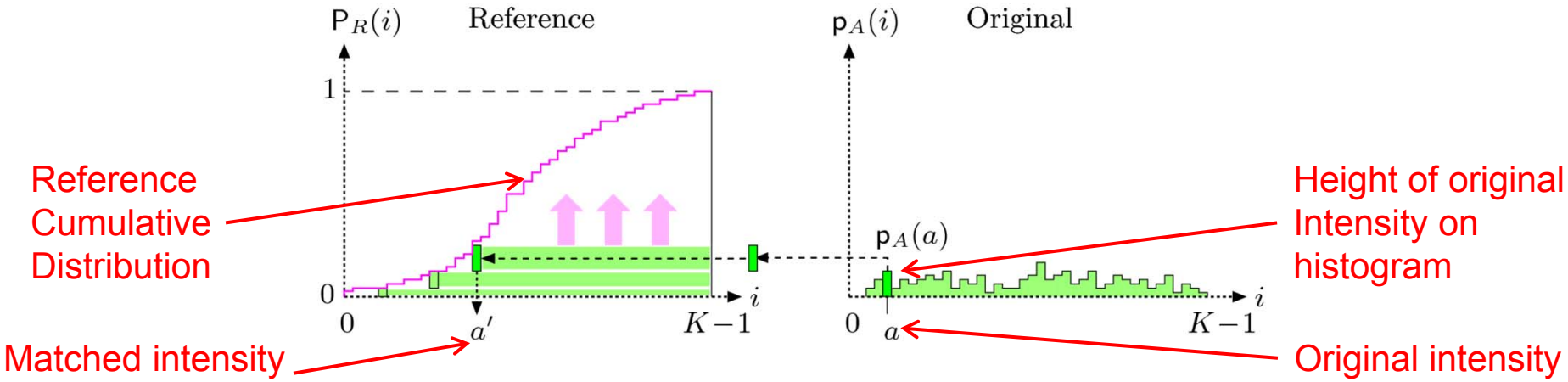


- Given two images I_A and I_B, we want to make their intensity profiles look as similar as possible.
- How?
  - "Match" their cumulative histograms H_A and H_B
- Works well for images with similar content.
- Looks bad for images with different content.

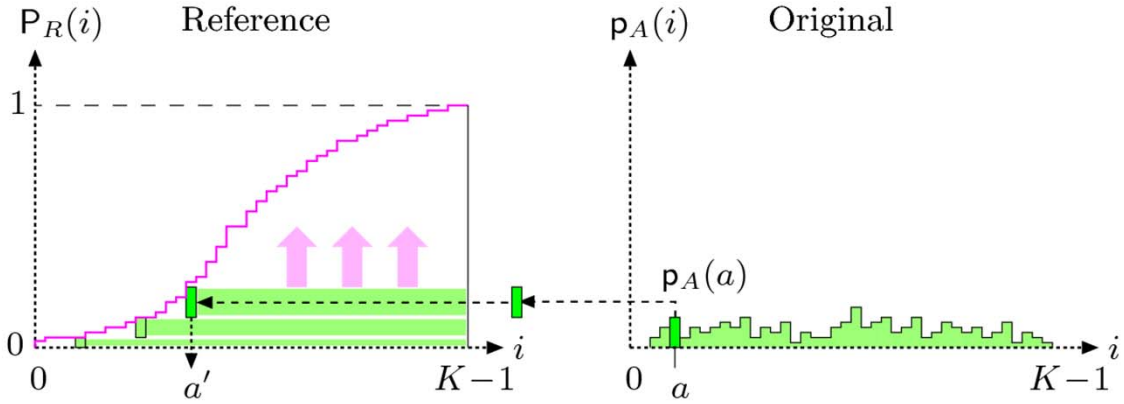
Adjusting to a Given Histogram



$$f_{\text{hs}}(a) = a' = \min\{\, j \mid (0 \le j < K) \wedge (P_A(a) \le P_R(j)) \,\}$$

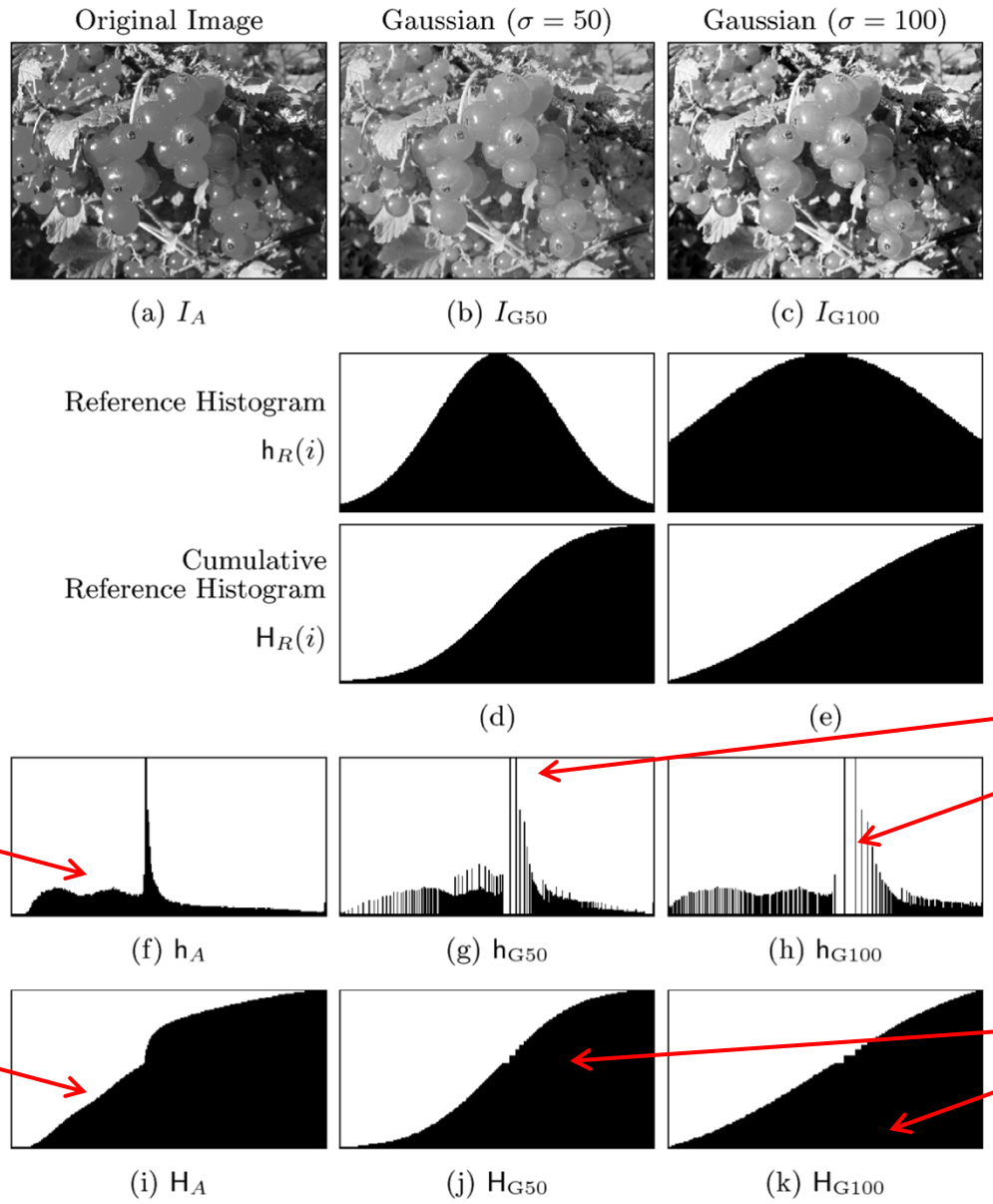


Adjusting to a Given Histogram




```
1:  MatchHistograms(h_A, h_R)
        h_A: histogram of the target image
        h_R: reference histogram (of same size as h_A)
2:      Let K ← Size(h_A)
3:      Let P_A ← Cdf(h_A)                          ▷ cdf for h_A (Alg. 5.1)
4:      Let P_R ← Cdf(h_R)                          ▷ cdf for h_R (Alg. 5.1)
5:      Create a table f_hs[ ] of size K            ▷ pixel mapping function f_hs
6:      for a ← 0...(K−1) do
7:          j ← K−1
8:          repeat
9:              f_hs[a] ← j
10:             j ← j − 1
11:         while (j ≥ 0) ∧ (P_A(a) ≤ P_R(j))
12:     return f_hs.
```

Adjusting to a Given Histogram



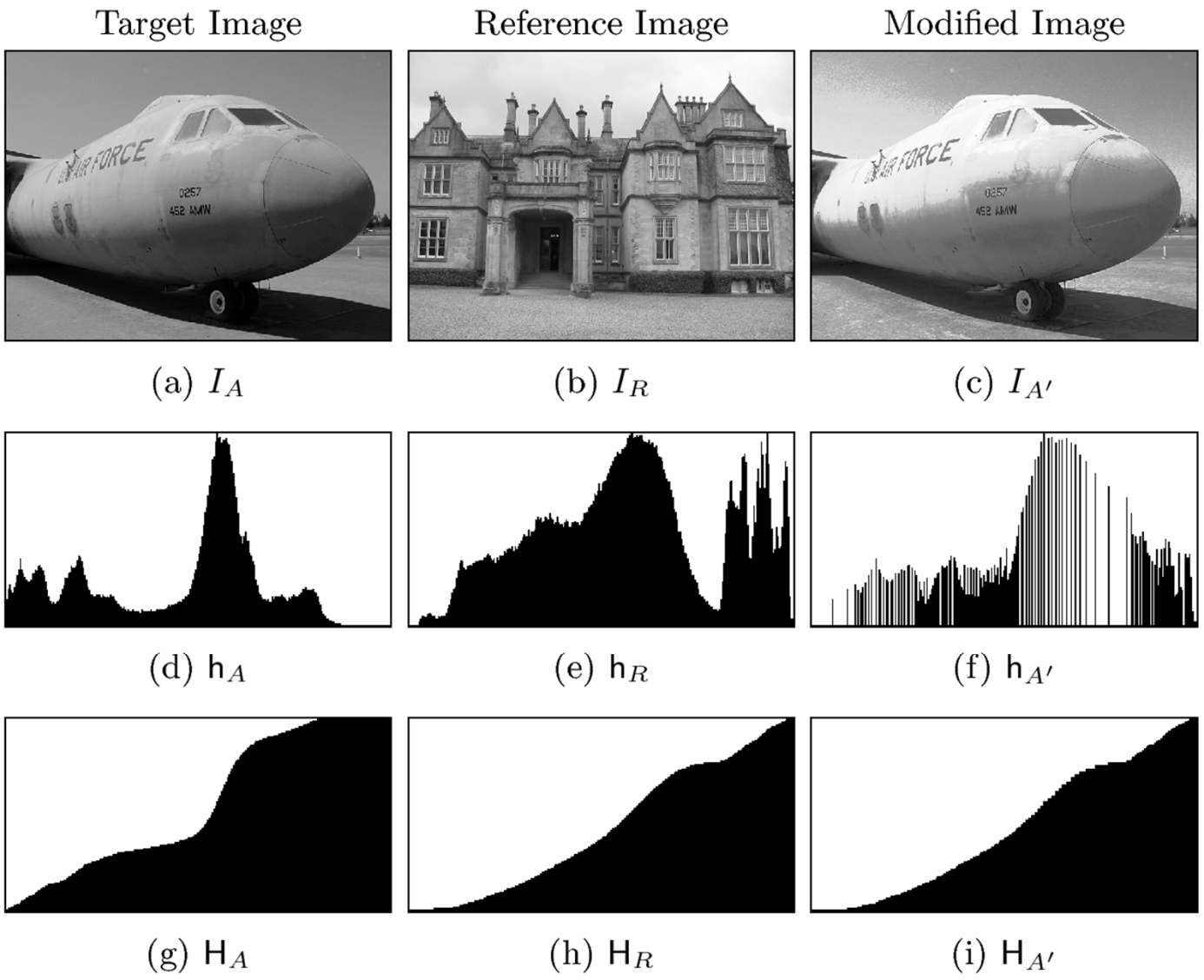
original histogram

Original histogram after matching

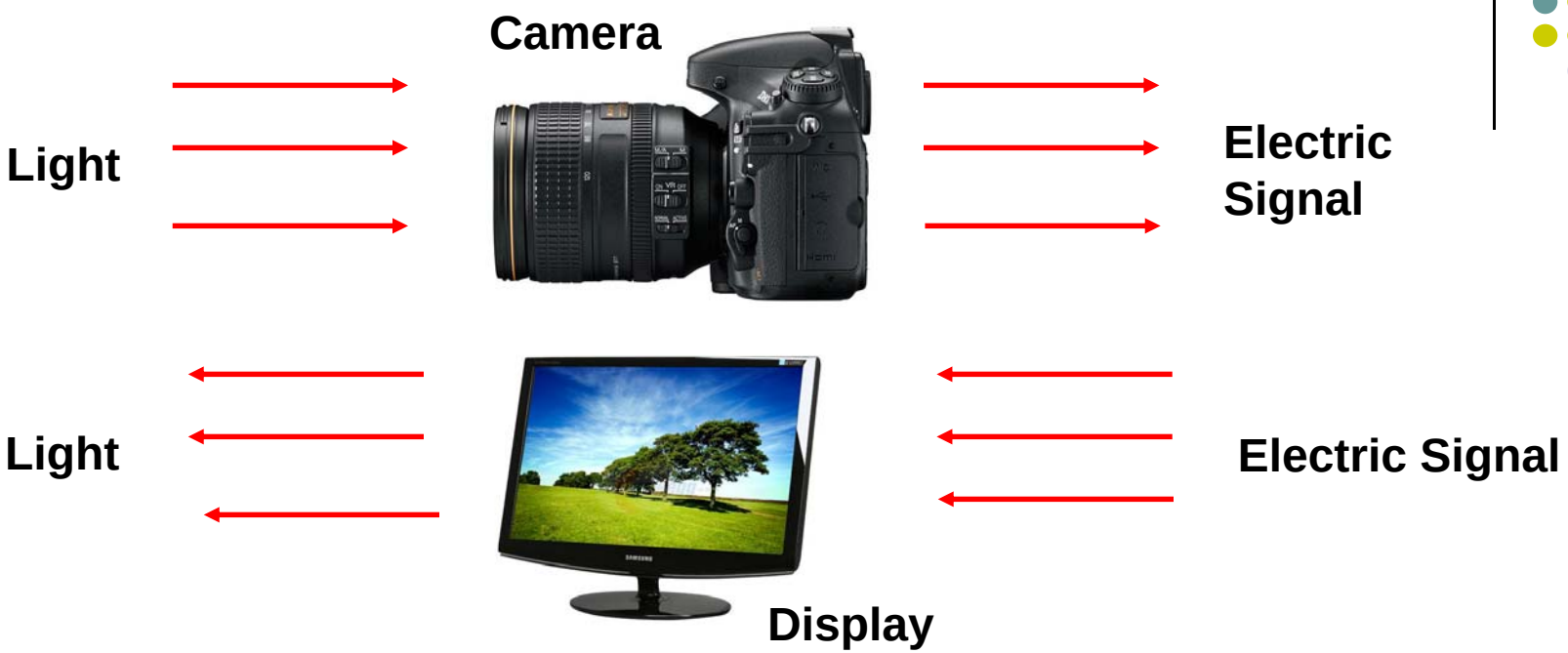
CDF of original histogram

CDF of original histogram after matching

Adjusting to a Given Histogram

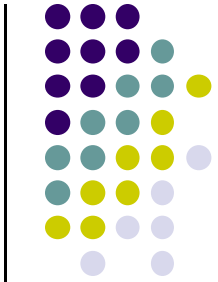


Gamma Correction



- Different camera sensors
  - Have different responses to light intensity
  - Produce different electrical signals for same input
- How do we ensure there is consistency in:
  a) Images recorded by different cameras for given light input
  b) Light emitted by different display devices for same image?

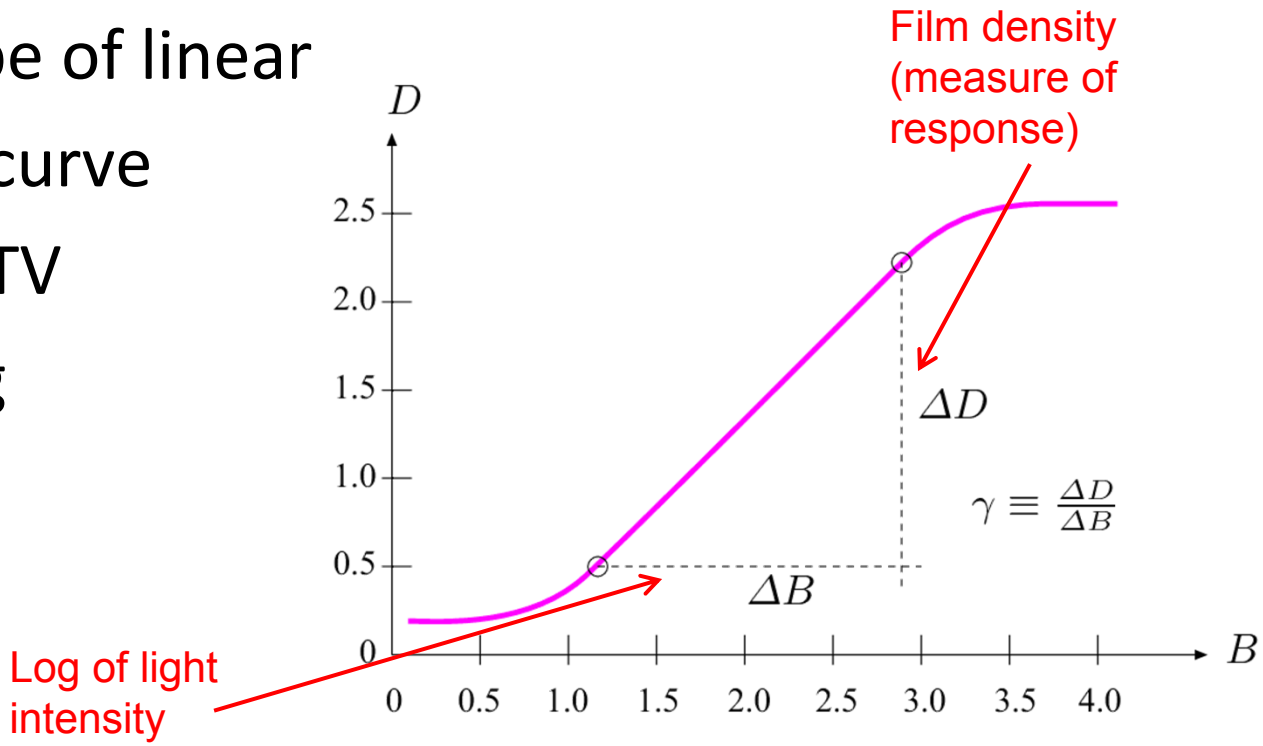
Gamma Correction



- What is the relation between:
  - **Camera:** Light on sensor vs. **"intensity"** of corresponding pixel
  - **Display:** Pixel intensity vs. light from that pixel
- Relation between pixel value and corresponding physical quantity is usually complex, nonlinear
- An approximation ?

What is Gamma?

- Originates from analog photography
- **Exposure function:** relationship between:
  - logarithmic light intensity vs. resulting film density.
- **Gamma:** slope of linear range of the curve
- The same in TV broadcasting



What is Gamma?

- **Gamma function:** a good approximation of exposure curve
- Inverse of a Gamma function is another gamma function with

$$\bar{\gamma} = 1/\gamma$$

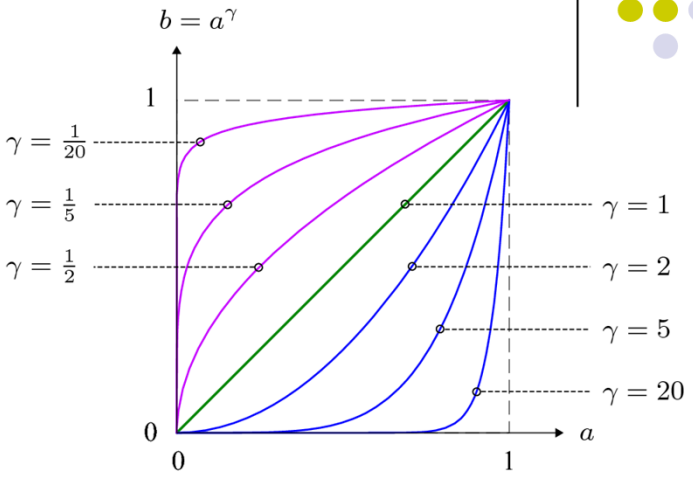
- Gamma of CRT and LCD monitors:
- 1.8-2.8 (typically 2.4)

$$s = B^{\gamma_c}$$

Output signal
Raised by gamma

$$b = s^{1/\gamma_c} = (B^{\gamma_c})^{1/\gamma_c} = B^{(\gamma_c \frac{1}{\gamma_c})} = B^1$$

Correct output signal
By dividing by 1/ gamma
(called Gamma correction)



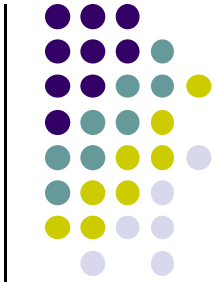
$$b = f_\gamma(a) = a^\gamma \quad \text{for } a \in \mathbb{R}\,, \gamma > 0$$

$$a = f_\gamma^{-1}(b) = b^{1/\gamma}$$

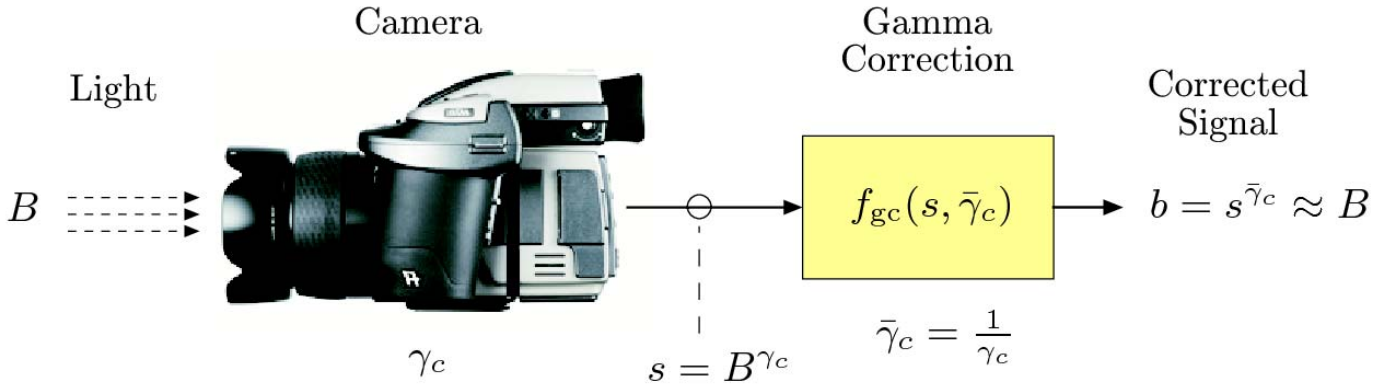
$$f_\gamma^{-1}(b) = f_{\bar{\gamma}}(b)$$

$$\bar{\gamma} = 1/\gamma$$

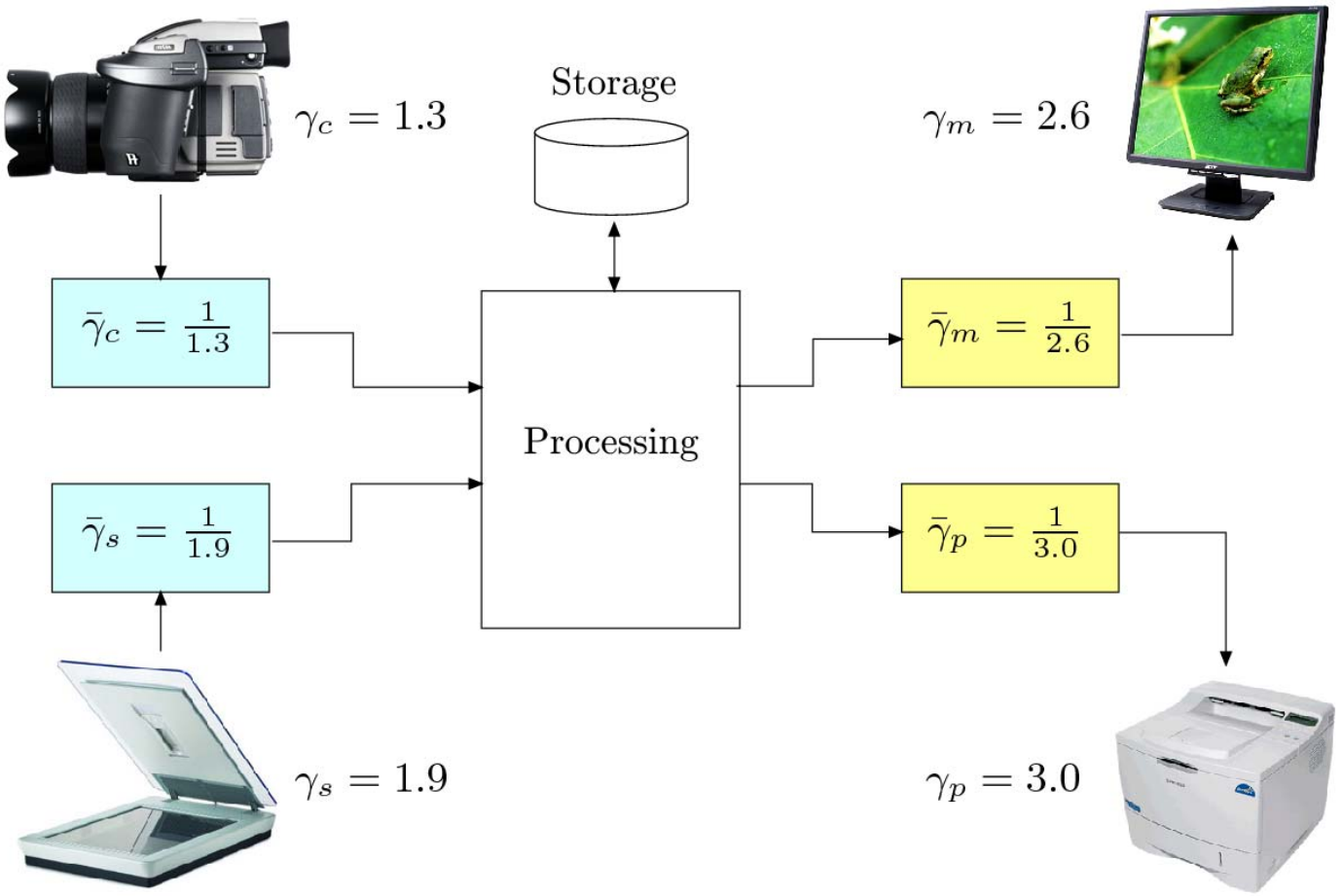
Gamma Correction



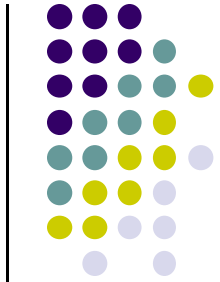
- Obtain a measurement **b** proportional to original light intensity **B** by applying inverse gamma function
- Gamma correction is important to achieve a device independent representation



Gamma Correction



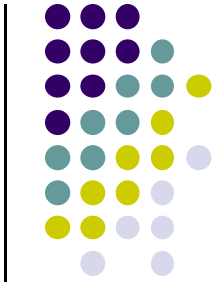
Gamma Correction Code



```
1   public void run(ImageProcessor ip) {
2       // works for 8-bit images only
3       int K = 256;
4       int aMax = K - 1;
5       double GAMMA = 2.8;
6
7       // create a lookup table for the mapping function
8       int[] Fgc = new int[K];
9
10      for (int a = 0; a < K; a++) {
11          double aa = (double) a / aMax;   // scale to [0, 1]
12          double bb = Math.pow(aa,GAMMA); // gamma function
13          // scale back to [0, 255]:
14          int b = (int) Math.round(bb * aMax);
15          Fgc[a] = b;
16      }
17
18      ip.applyTable(Fgc);   // modify the image ip
19  }
```

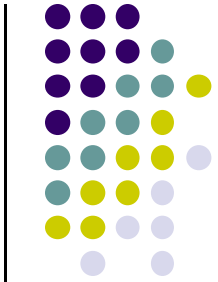
Compute corrected intensity and store in lookup table **fgc**

Point Operations in ImageJ



`void abs()`	$I'(u,v) \leftarrow \lvert I(u,v)\rvert$
`void add(int p)`	$I'(u,v) \leftarrow I(u,v) + p$
`void gamma(double g)`	$I'(u,v) \leftarrow \big(I(u,v)/255\big)^{g} \cdot 255$
`void invert(int p)`	$I'(u,v) \leftarrow 255 - I(u,v)$
`void log()`	$I'(u,v) \leftarrow \log_{10}\big(I(u,v)\big)$
`void max(double s)`	$I'(u,v) \leftarrow \max\big(I(u,v), s\big)$
`void min(double s)`	$I'(u,v) \leftarrow \min\big(I(u,v), s\big)$
`void multiply(double s)`	$I'(u,v) \leftarrow \mathrm{round}\big(I(u,v) \cdot s\big)$
`void sqr()`	$I'(u,v) \leftarrow I(u,v)^2$
`void sqrt()`	$I'(u,v) \leftarrow \sqrt{I(u,v)}$

ImageJ Operations involving 2 images



ADD	$ip1 \leftarrow ip1 + ip2$
AVERAGE	$ip1 \leftarrow (ip1 + ip2) \,/\, 2$
DIFFERENCE	$ip1 \leftarrow \lvert ip1 - ip2 \rvert$
DIVIDE	$ip1 \leftarrow ip1 \,/\, ip2$
MAX	$ip1 \leftarrow \max(ip1, ip2)$
MIN	$ip1 \leftarrow \min(ip1, ip2)$
MULTIPLY	$ip1 \leftarrow ip1 \cdot ip2$
SUBTRACT	$ip1 \leftarrow ip1 - ip2$

Example: Alpha Blending

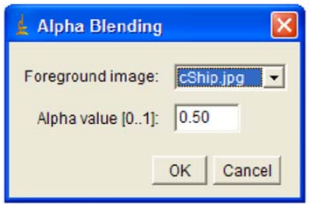
$$I'(u,v) \leftarrow \alpha \cdot I_{\mathrm{BG}}(u,v) + (1-\alpha) \cdot I_{\mathrm{FG}}(u,v)$$

I_{BG}

$\alpha = 0.25$

I_{FG}

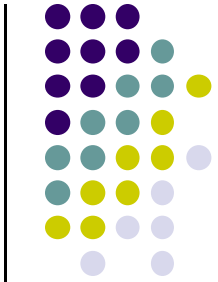
$\alpha = 0.50$



`GenericDialog`

$\alpha = 0.75$

Alpha Blending PlugIn



```
1  import ij.IJ;
2  import ij.ImagePlus;
3  import ij.WindowManager;
4  import ij.gui.GenericDialog;
5  import ij.plugin.filter.PlugInFilter;
6  import ij.process.*;
7
8  public class Alpha_Blending implements PlugInFilter {
9
10   static double alpha = 0.5; // transparency of foreground image
11   ImagePlus fgIm = null;     // foreground image
12
13   public int setup(String arg, ImagePlus imp) {
14     return DOES_8G;
15   }
16
17   public void run(ImageProcessor bgIp) { // background image
18     if(runDialog()) {
19       ImageProcessor fgIp
20          = fgIm.getProcessor().convertToByte(false);
21       fgIp = fgIp.duplicate();
22       fgIp.multiply(1-alpha);
23       bgIp.multiply(alpha);
24       bgIp.copyBits(fgIp, 0, 0, Blitter.ADD);
25     }
26   }
27
28   // continued ...
```

What Is Image Enhancement?

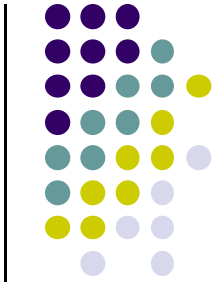
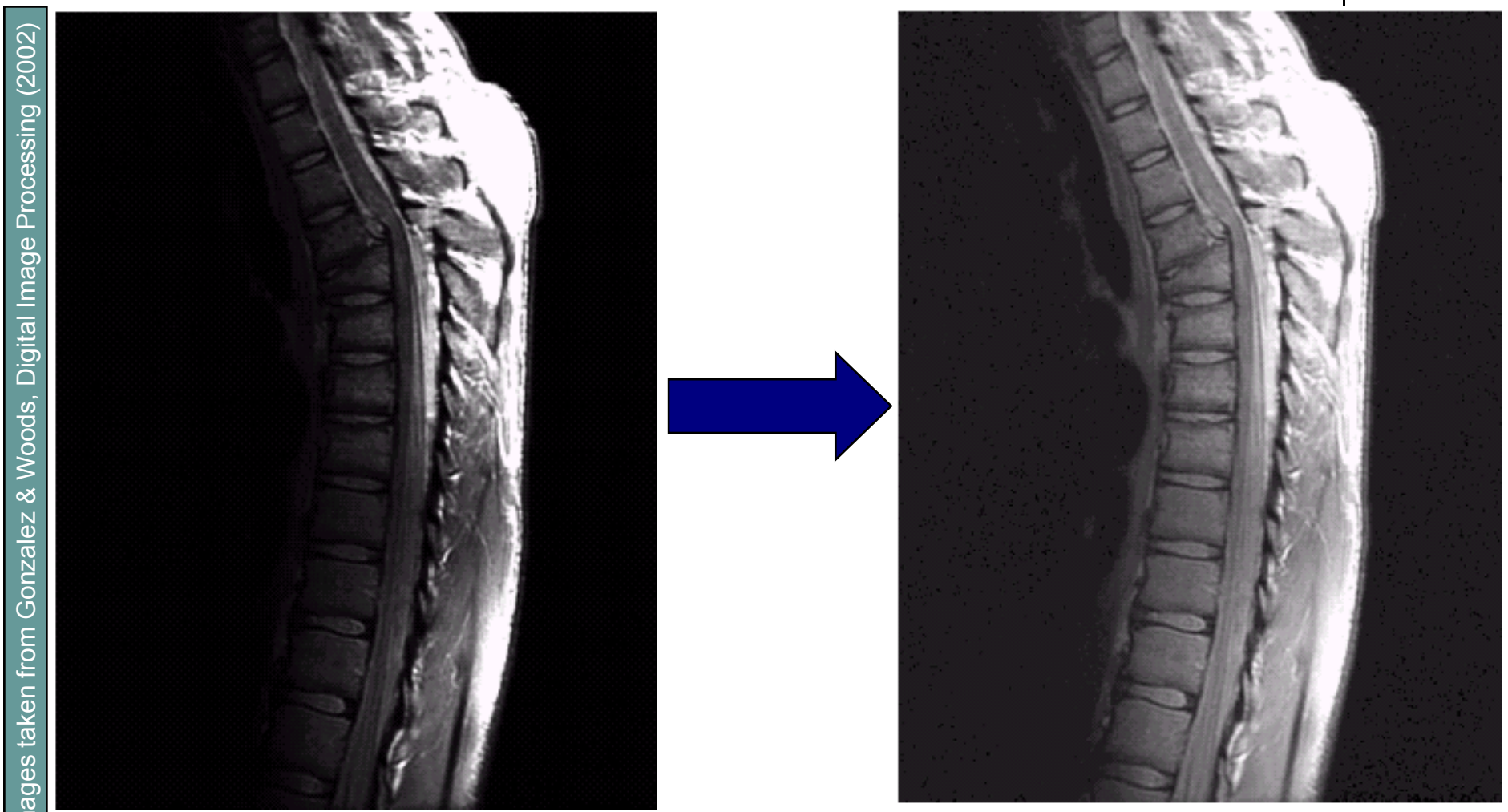


Image enhancement makes images more useful by:

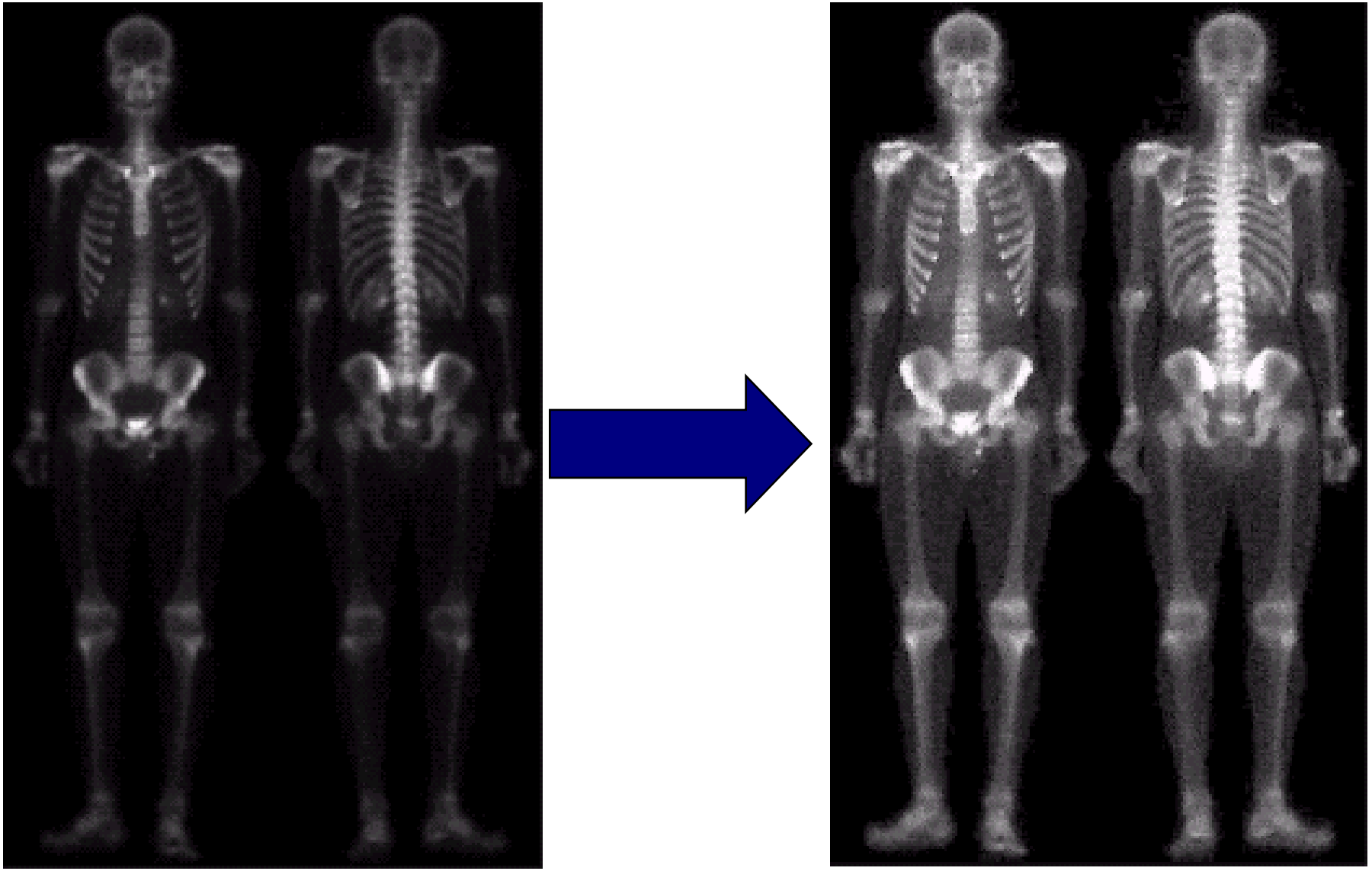
- Highlighting interesting detail in images
- Removing noise from images
- Making images more visually appealing

Image Enhancement Examples



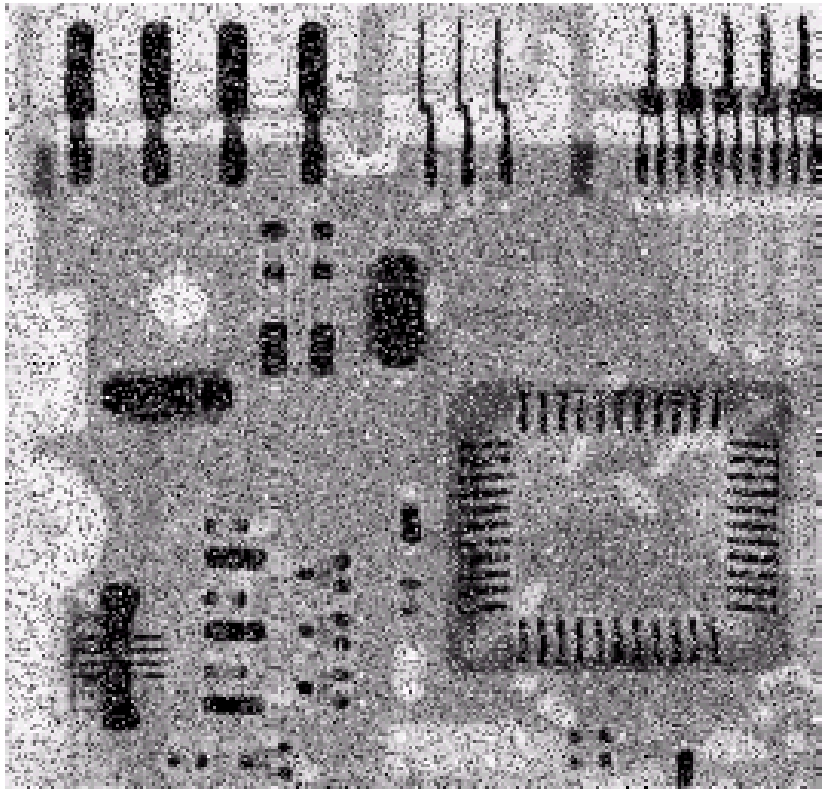
Images taken from Gonzalez & Woods, Digital Image Processing (2002)

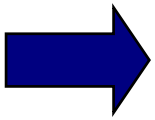
Image Enhancement Examples (cont…)

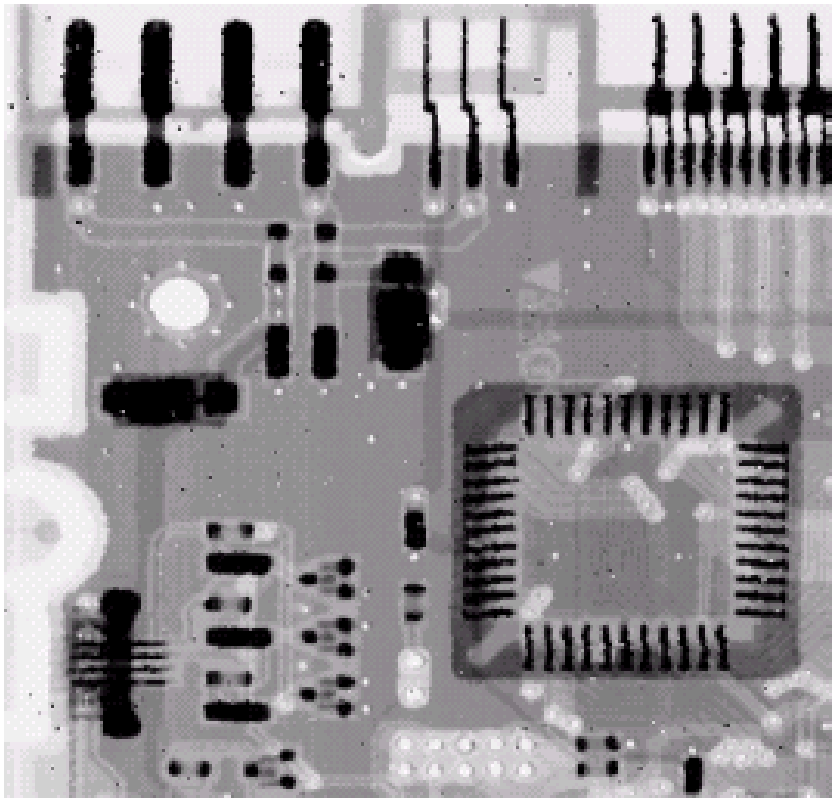


Images taken from Gonzalez & Woods, Digital Image Processing (2002)

Image Enhancement Examples (cont…)





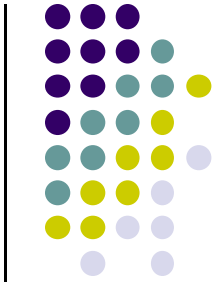


Images taken from Gonzalez & Woods, Digital Image Processing (2002)

Image Enhancement Examples (cont…)

Images taken from Gonzalez & Woods, Digital Image Processing (2002)

Spatial & Frequency Domains



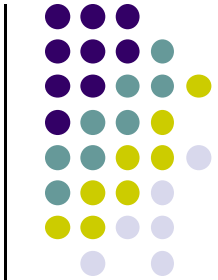
There are two broad categories of image enhancement techniques

- Spatial domain techniques
  - Direct manipulation of image pixels (intensity values)
- Frequency domain techniques
  - Manipulation of Fourier transform or wavelet transform of an image

First spatial domain techniques

Later: frequency domain techniques

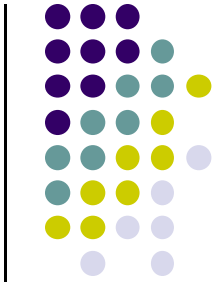
What is a Filter?



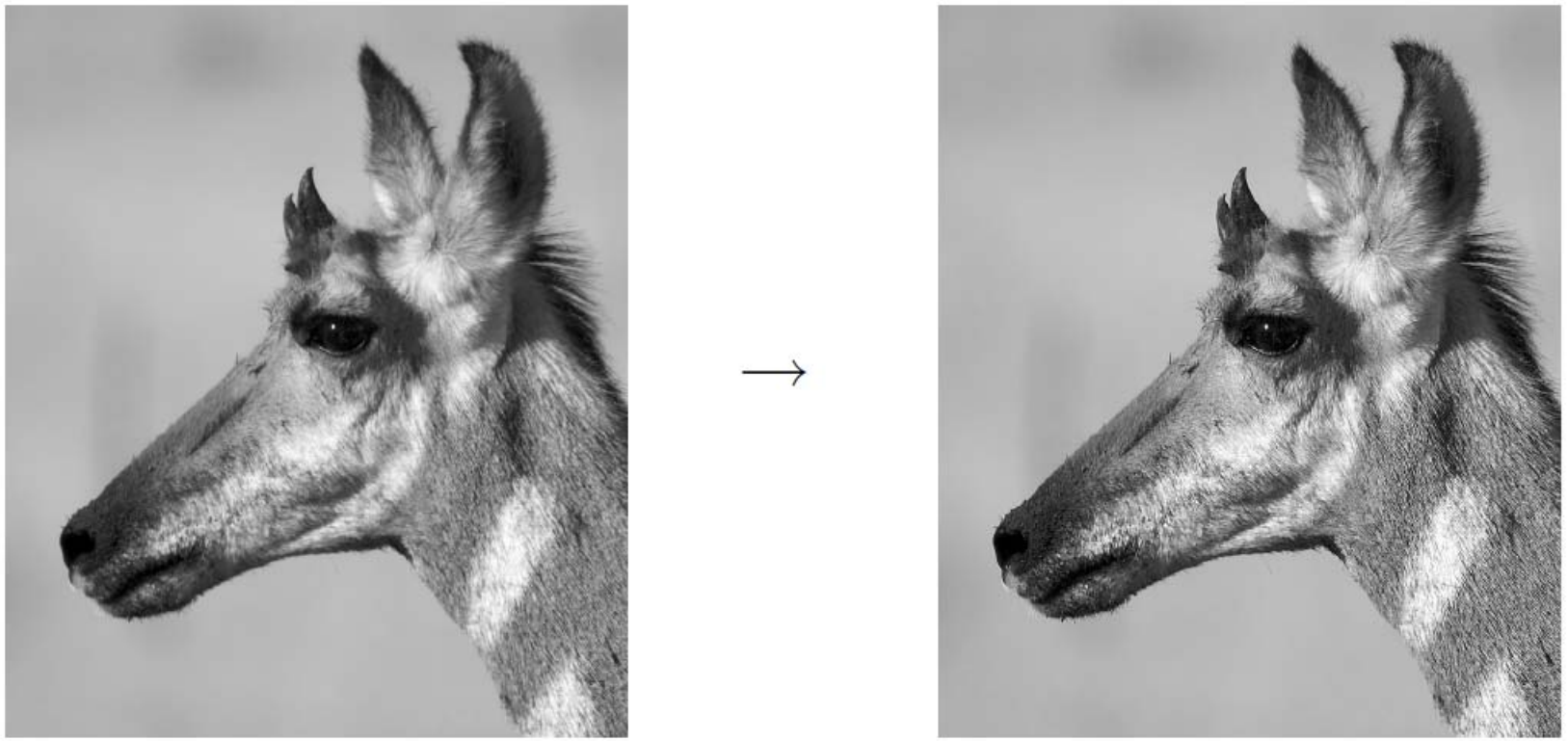
- Capabilities of point operations are limited
- **Filters:** combine **pixel's value** + **values of neighbors**
- **E.g blurring:** Compute average intensity of block of pixels

- Combining multiple pixels needed for certain operations:
  - Blurring, Smoothing
  - Sharpening

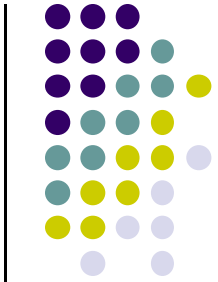
What Point Operations Can’t Do



- Example: sharpening

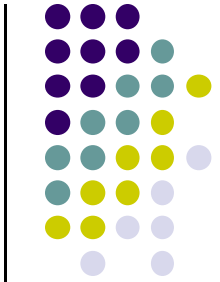


What Point Operations Can't Do

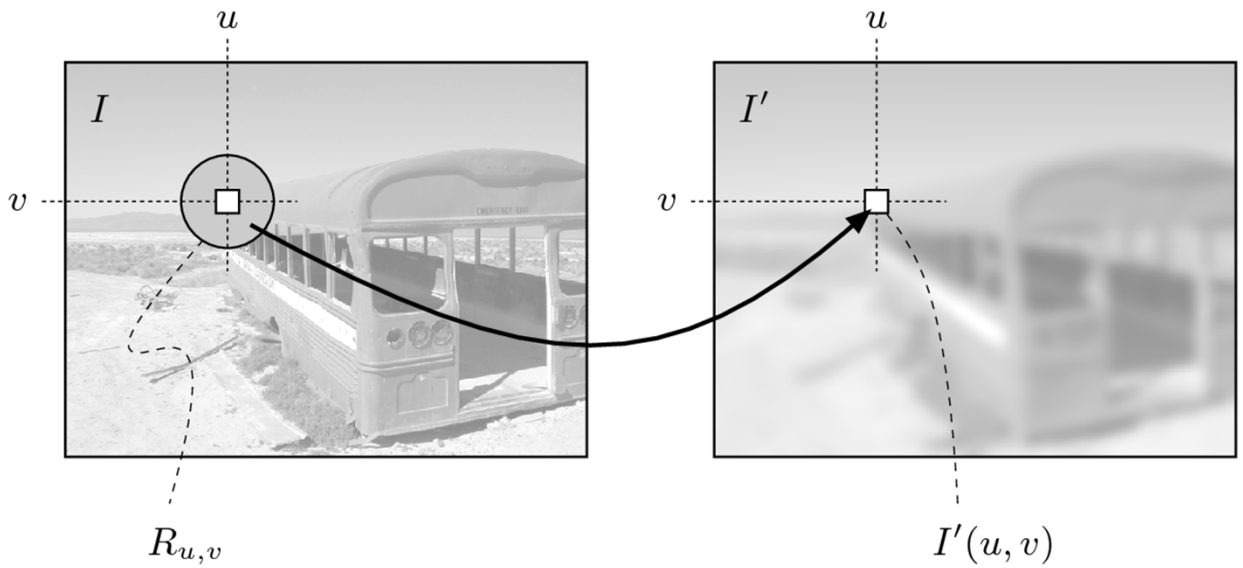


- Other cool artistic patterns by combining pixels

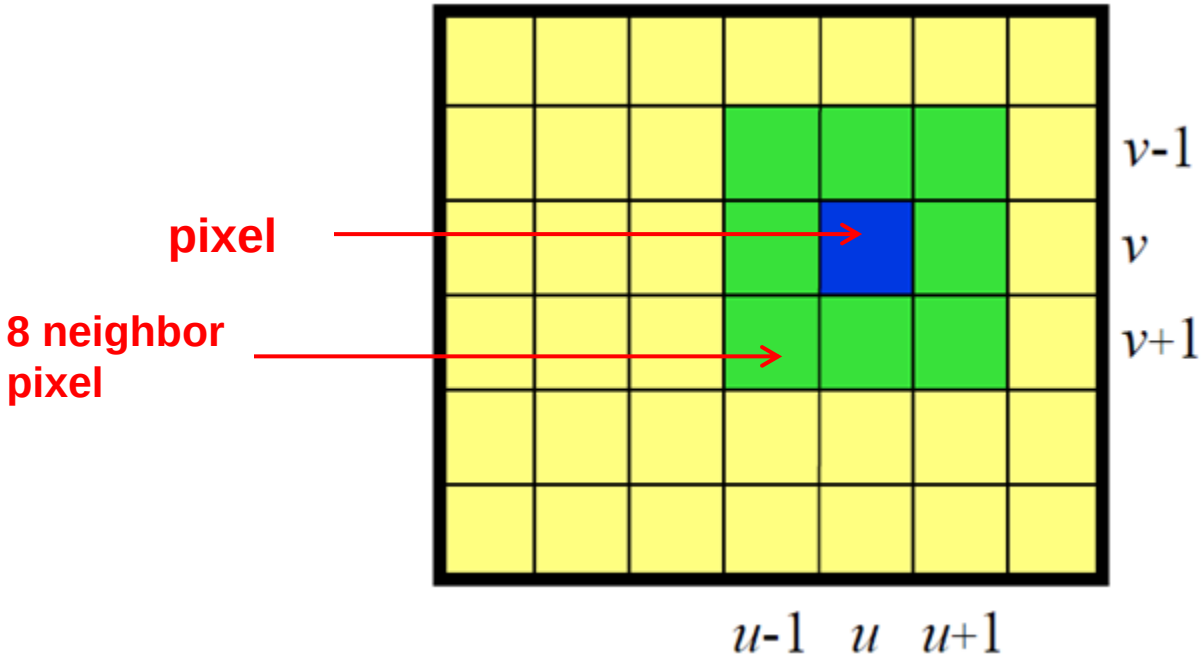
Definition: Spatial Filter



- An image operation that combines each pixel's intensity $I(u, v)$ with that of neighboring pixels
- **E.g:** average/weighted average of group of pixels

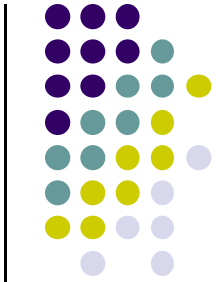


Example: Average (Mean) of 3x3 Neighborhood



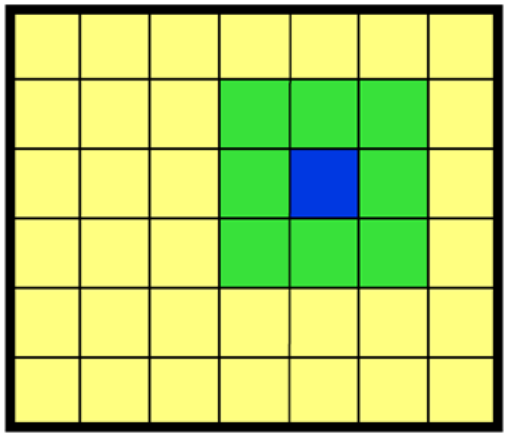
Blurring: Replace each pixel with AVERAGE Intensity of pixel + neighbors

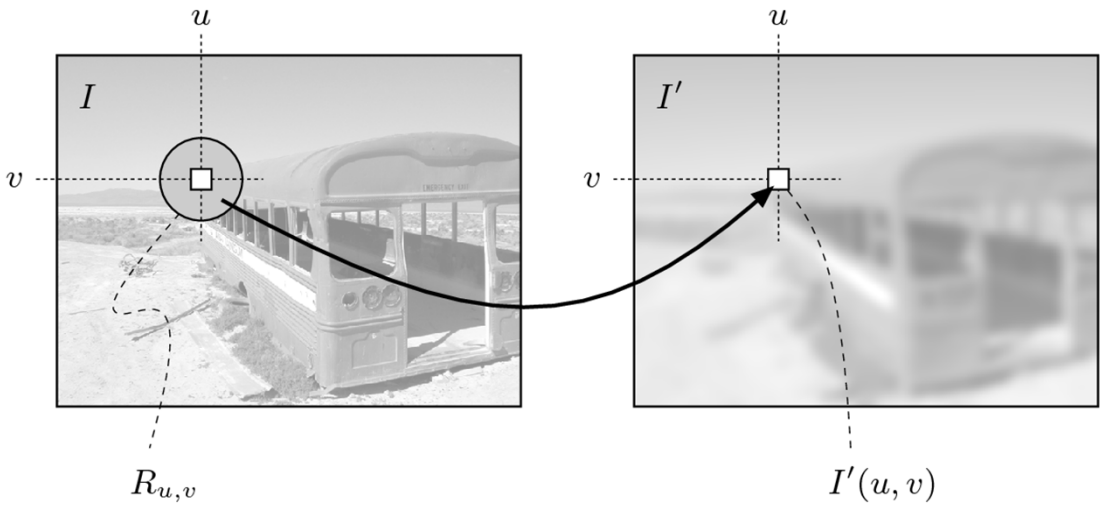
Smoothing an Image by Averaging



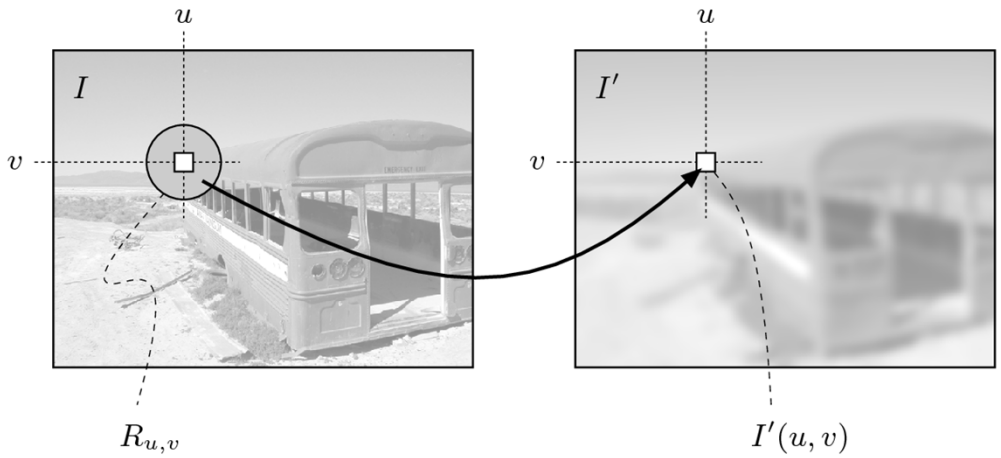
- Replace each pixel by average of pixel + neighbors
- For 3x3 neighborhood:

$$I'(u,v) \leftarrow \frac{p_0 + p_1 + p_2 + p_3 + p_4 + p_5 + p_6 + p_7 + p_8}{9}$$



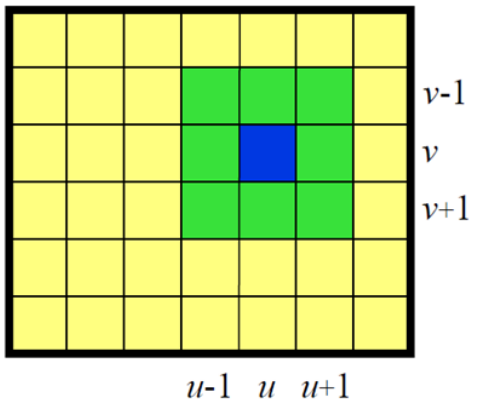


Smoothing an Image by Averaging



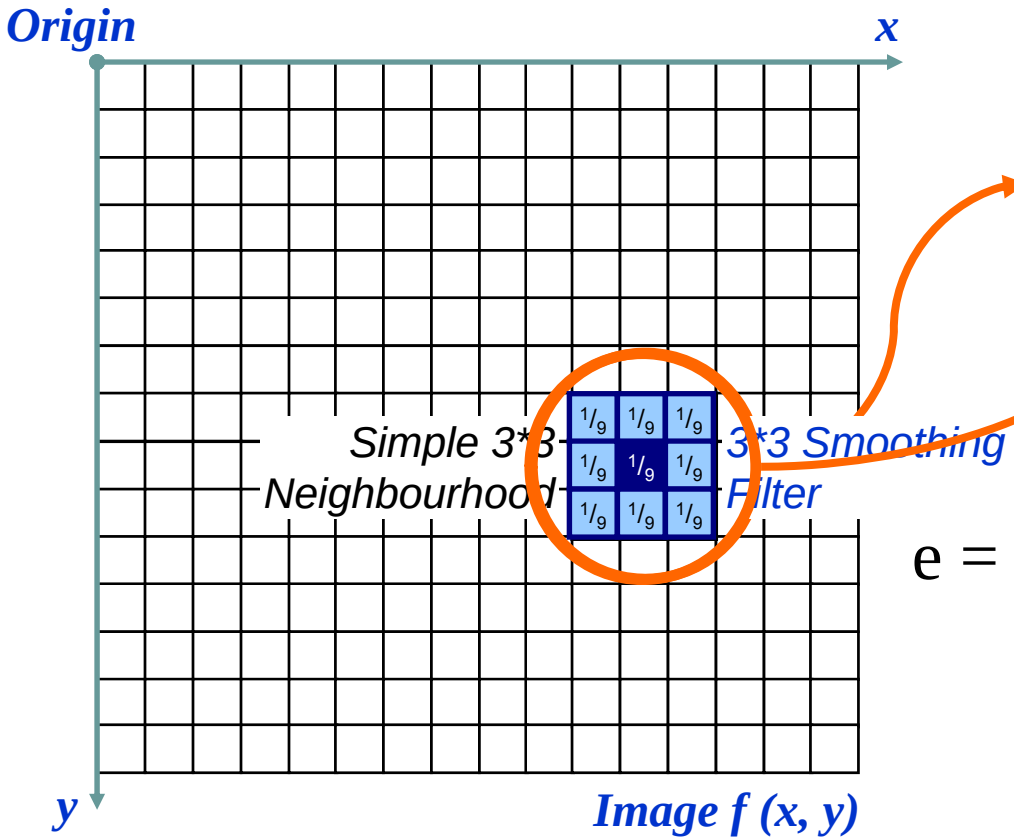
$$I'(u,v) \leftarrow \frac{p_0 + p_1 + p_2 + p_3 + p_4 + p_5 + p_6 + p_7 + p_8}{9}$$

$$I'(u,v) \leftarrow \tfrac{1}{9} \cdot [\, I(u-1,v-1) + I(u,v-1) + I(u+1,v-1) + I(u-1,v) + I(u,v) + I(u+1,v) + I(u-1,v+1) + I(u,v+1) + I(u+1,v+1) \,]$$



$$I'(u,v) \leftarrow \frac{1}{9} \cdot \sum_{j=-1}^{1} \sum_{i=-1}^{1} I(u+i, v+j)$$

Example: Smoothing Spatial Filtering



104	100	108
99	106	98
95	90	85

Original Image Pixels

*

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

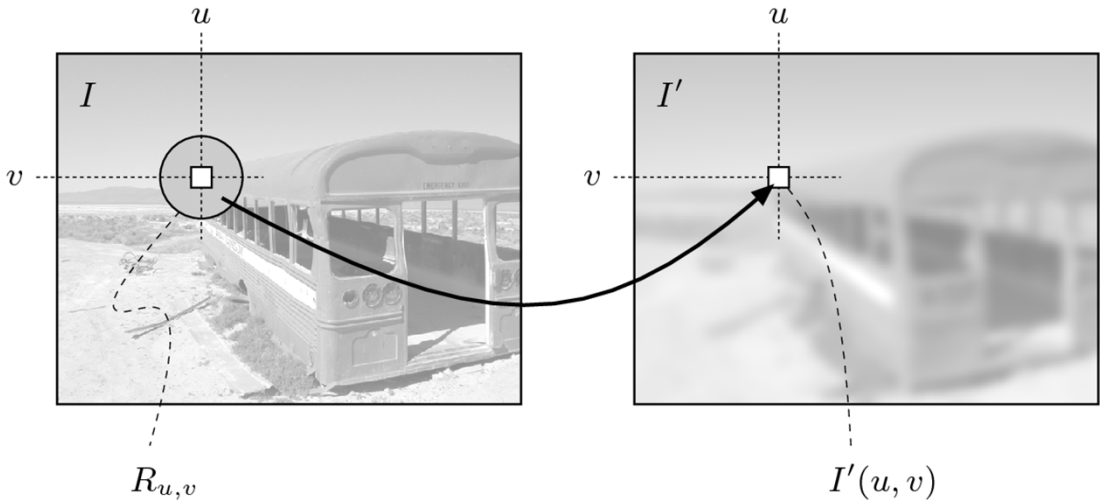
Filter

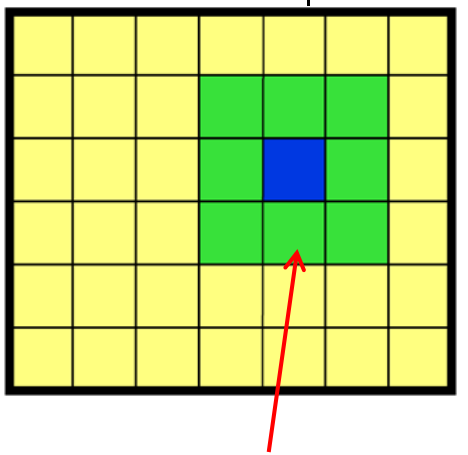
$$e = \tfrac{1}{9}*106 + \tfrac{1}{9}*104 + \tfrac{1}{9}*100 + \tfrac{1}{9}*108 + \tfrac{1}{9}*99 + \tfrac{1}{9}*98 + \tfrac{1}{9}*95 + \tfrac{1}{9}*90 + \tfrac{1}{9}*85$$

$$= 98.3333$$

The above is repeated for every pixel in the original image to generate the smoothed image

Smoothing an Image by Averaging





Previous example:
Filter size: 3x3

- Many possible filter parameters (size, weights, function, etc)
- **Filter size (size of neighborhood):** 3x3, 5x5, 7x7, …,21x21,..
- **Filter shape:** not necessarily square. Can be rectangle, circle, etc
- **Filter weights:** May apply unequal weighting to different pixels
- **Filters function:** can be linear (a weighted summation) or nonlinear

The Filter Matrix

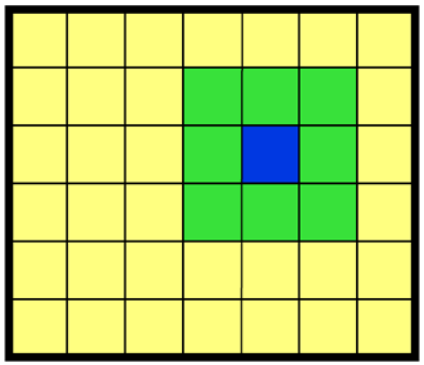
$$I'(u,v) \leftarrow \frac{p_0 + p_1 + p_2 + p_3 + p_4 + p_5 + p_6 + p_7 + p_8}{9}$$

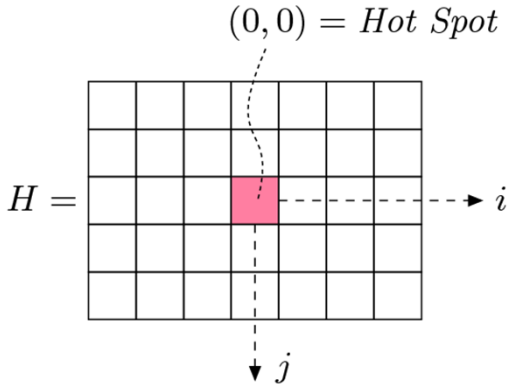
$$\begin{aligned} I'(u,v) \leftarrow \tfrac{1}{9} \cdot [\, & I(u-1,v-1) + I(u,v-1) + I(u+1,v-1) + \\ & I(u-1,v) + I(u,v) + I(u+1,v) + \\ & I(u-1,v+1) + I(u,v+1) + I(u+1,v+1) \,] \end{aligned}$$

$$H(i,j) = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

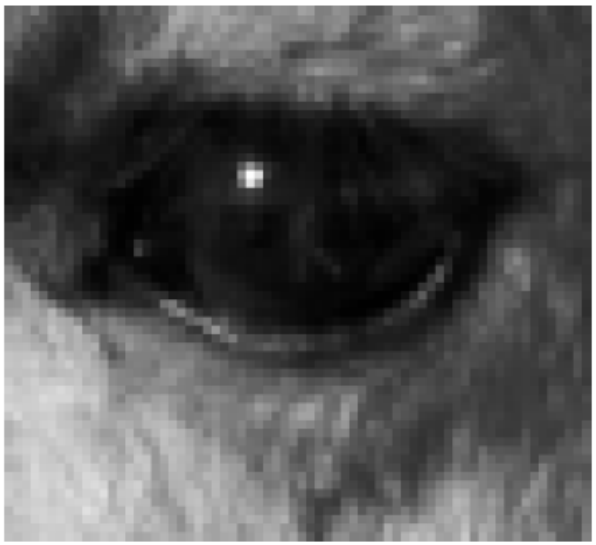
Filter operation can be expressed as a matrix
Example: averaging filter

Filter matrix also called filter mask ***H(i,j)***





Example: What does this Filter Do?

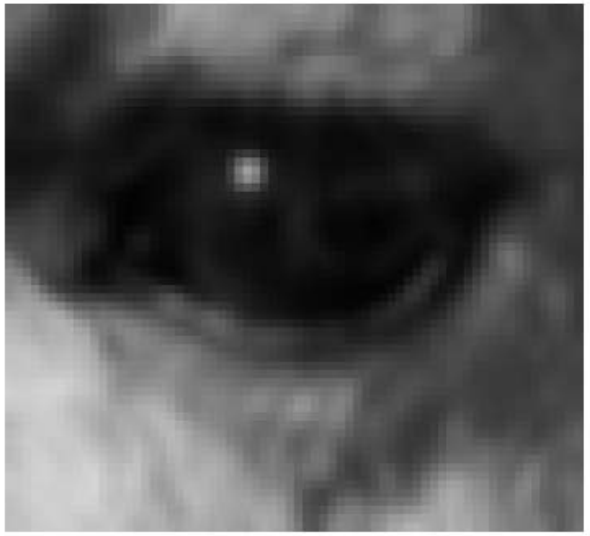


0	0	0
0	1	0
0	0	0

Identity function (leaves image alone)

What Does this Filter Do?

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$



Mean (averages neighborhood)

Mean Filters: Effect of Filter Size

Original

7×7

15×15

41×41

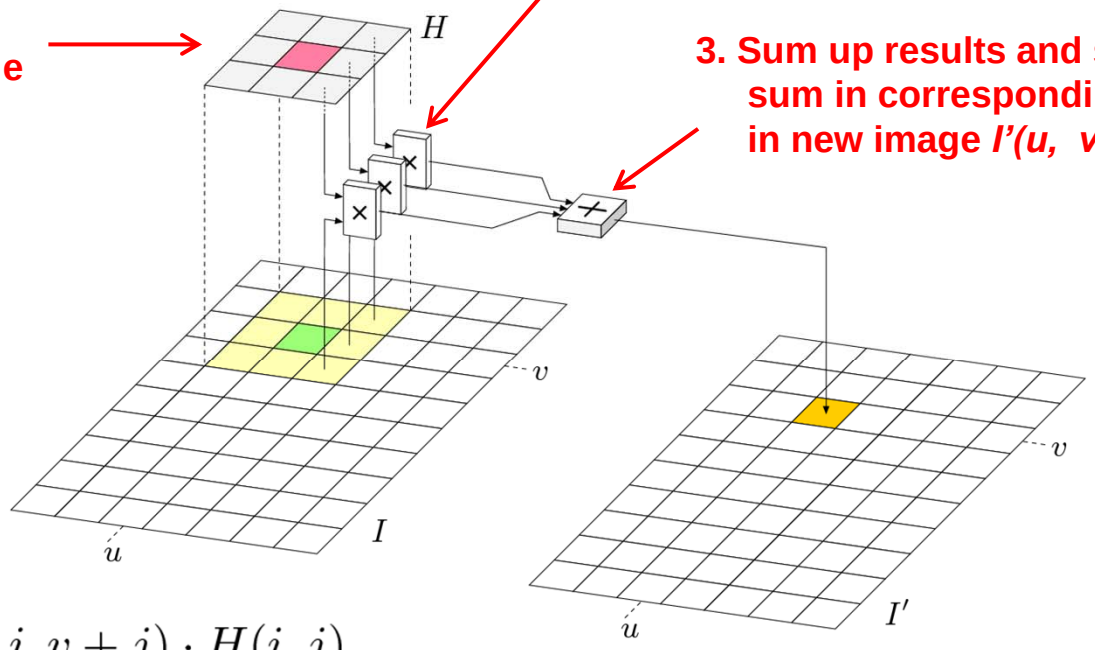
Applying Linear Filters: Convolution

For each image position *I(u,v):*

1. Move filter matrix *H* over image such that *H(0,0)* coincides with current image position *(u,v)*

2. Multiply all filter coefficients *H(i,j)* with corresponding pixel *I(u + i, v + j)*

3. Sum up results and store sum in corresponding position in new image *I'(u, v)*



Stated formally:

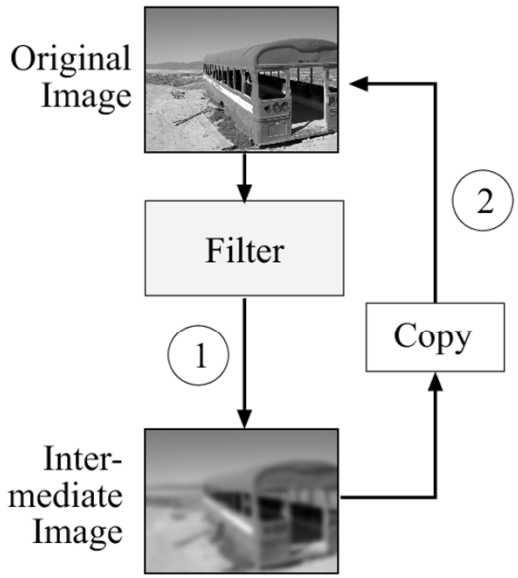
$$I'(u,v) \leftarrow \sum_{(i,j)\in R_H} I(u+i, v+j) \cdot H(i,j)$$

***R_H* is set of all pixels Covered by filter.**
For 3x3 filter, this is:

$$I'(u,v) \leftarrow \sum_{i=-1}^{i=1} \sum_{j=-1}^{j=1} I(u+i, v+j) \cdot H(i,j)$$

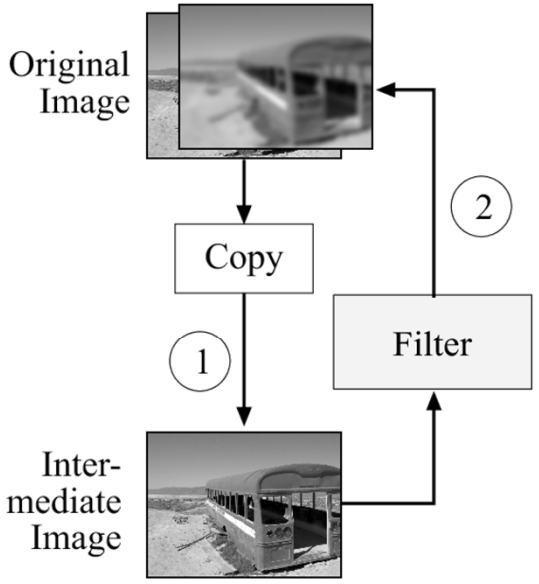
Computing Filter Operation

- Filter matrix ***H*** moves over each pixel in original image ***I*** to compute corresponding pixel in new image ***I'***
- Cannot overwrite new pixel value in original image ***I*** Why?



Version A

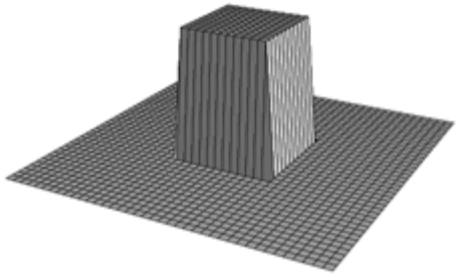
Store results ***I'*** in intermediate image, then copy back to replace ***I***



Version B

Copy original image ***I*** to intermediate image, use it as source, then store results ***I'*** to replace original image

Simple 3x3 Averaging Filter ("Box" Filter)



```
1  import ij.*;
2  import ij.plugin.filter.PlugInFilter;
3  import ij.process.*;
4
5  public class Filter_Average3x3 implements PlugInFilter {
6      ...
7      public void run(ImageProcessor orig) {
8          int w = orig.getWidth();
9          int h = orig.getHeight();
10         ImageProcessor copy = orig.duplicate();
11
12         for (int v = 1; v <= h-2; v++) {
13             for (int u = 1; u <= w-2; u++) {
14                 //compute filter result for position (u, v)
15                 int sum = 0;
16                 for (int j = -1; j <= 1; j++) {
17                     for (int i = -1; i <= 1; i++) {
18                         int p = copy.getPixel(u+i, v+j);
19                         sum = sum + p;
20                     }
21                 }
22                 int q = (int) Math.round(sum/9.0);
23                 orig.putPixel(u, v, q);
24             }
25         }
26     }
27 } // end of class Filter_Average3x3
```

No explicit filter matrix since all coefficients are the same (1/9)

No clamping required

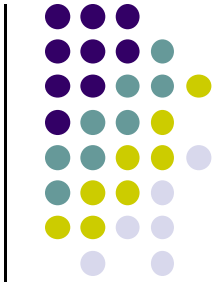
Make copy of original image to use as source

Loop over all pixels in image

Filter computation by adding current pixel's neighbors

Store result back in original image

Weighted Smoothing Filters

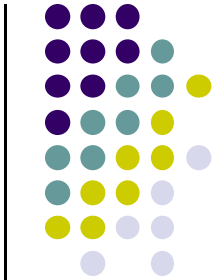


- More effective smoothing filters can be generated by allowing different pixels in the neighbourhood different weights in the averaging function
  - Pixels closer to central pixel more important
  - Often referred to as a *weighted averaging*

$^1/_{16}$	$^2/_{16}$	$^1/_{16}$
$^2/_{16}$	$^4/_{16}$	$^2/_{16}$
$^1/_{16}$	$^2/_{16}$	$^1/_{16}$

Weighted averaging filter

Another Smoothing Filter

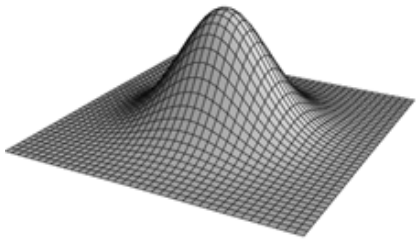


```
1   public void run(ImageProcessor orig) {
2       int w = orig.getWidth();
3       int h = orig.getHeight();
4       // 3 × 3 filter matrix
5       double[][] filter = {
6           {0.075, 0.125, 0.075},
7           {0.125, 0.200, 0.125},
8           {0.075, 0.125, 0.075}
9       };
10      ImageProcessor copy = orig.duplicate();
11
12      for (int v = 1; v <= h-2; v++) {
13          for (int u = 1; u <= w-2; u++) {
14              // compute filter result for position (u, v)
15              double sum = 0;
16              for (int j = -1; j <= 1; j++) {
17                  for (int i = -1; i <= 1; i++) {
18                      int p = copy.getPixel(u+i, v+j);
19                      // get the corresponding filter coefficient:
20                      double c = filter[j+1][i+1];
21                      sum = sum + c * p;
22                  }
23              }
24              int q = (int) Math.round(sum);
25              orig.putPixel(u, v, q);
26          }
27      }
28  }
```

Use real filter matrix with coefficents
Apply bell-shaped function *H(i,j)*

$$H(i,j) = \begin{bmatrix} 0.075 & 0.125 & 0.075 \\ 0.125 & \mathbf{0.2} & 0.125 \\ 0.075 & 0.125 & 0.075 \end{bmatrix}$$

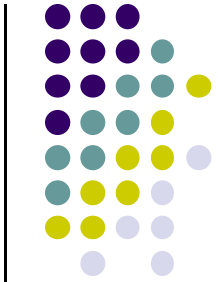
Bell-shaped function *H(i,j)*?
- More weight applied to center



Apply filter

Store result back in original

Integer Coefficients



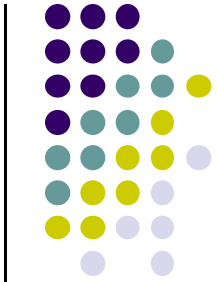
- Instead of floating point coeffients, more efficient, simpler to use:

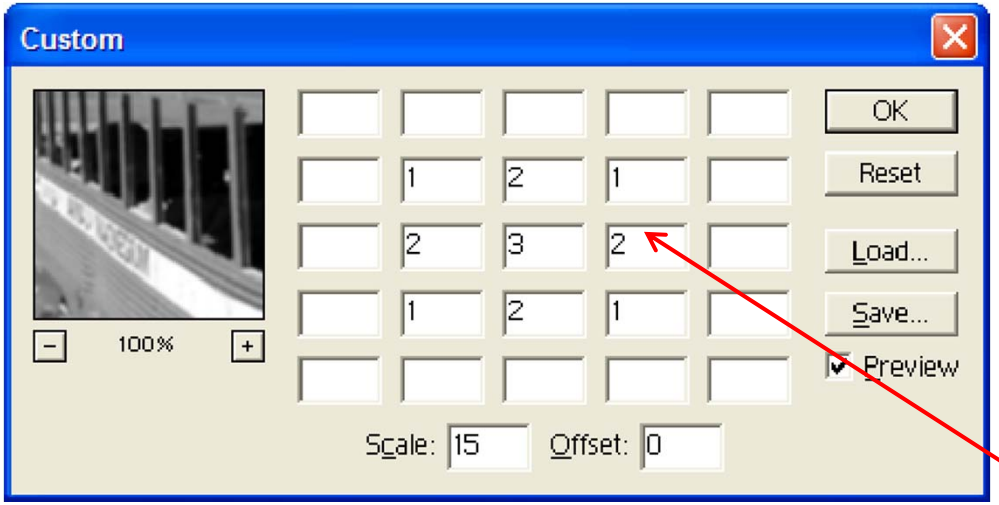
scaling factor + integer coefficients

$$H(i,j) = \begin{bmatrix} 0.075 & 0.125 & 0.075 \\ 0.125 & \underline{0.200} & 0.125 \\ 0.075 & 0.125 & 0.075 \end{bmatrix} = \frac{1}{40} \begin{bmatrix} 3 & 5 & 3 \\ 5 & \underline{8} & 5 \\ 3 & 5 & 3 \end{bmatrix}$$

$$H(i,j) = s \cdot H'(i,j)$$

Example: 5x5 Filter in Adobe Photoshop





Integer filter coefficients

$$I'(u,v) \leftarrow \text{Offset} + \frac{1}{\text{Scale}} \sum_{j=-2}^{j=2} \sum_{i=-2}^{i=2} I(u+i, v+j) \cdot H(i,j)$$

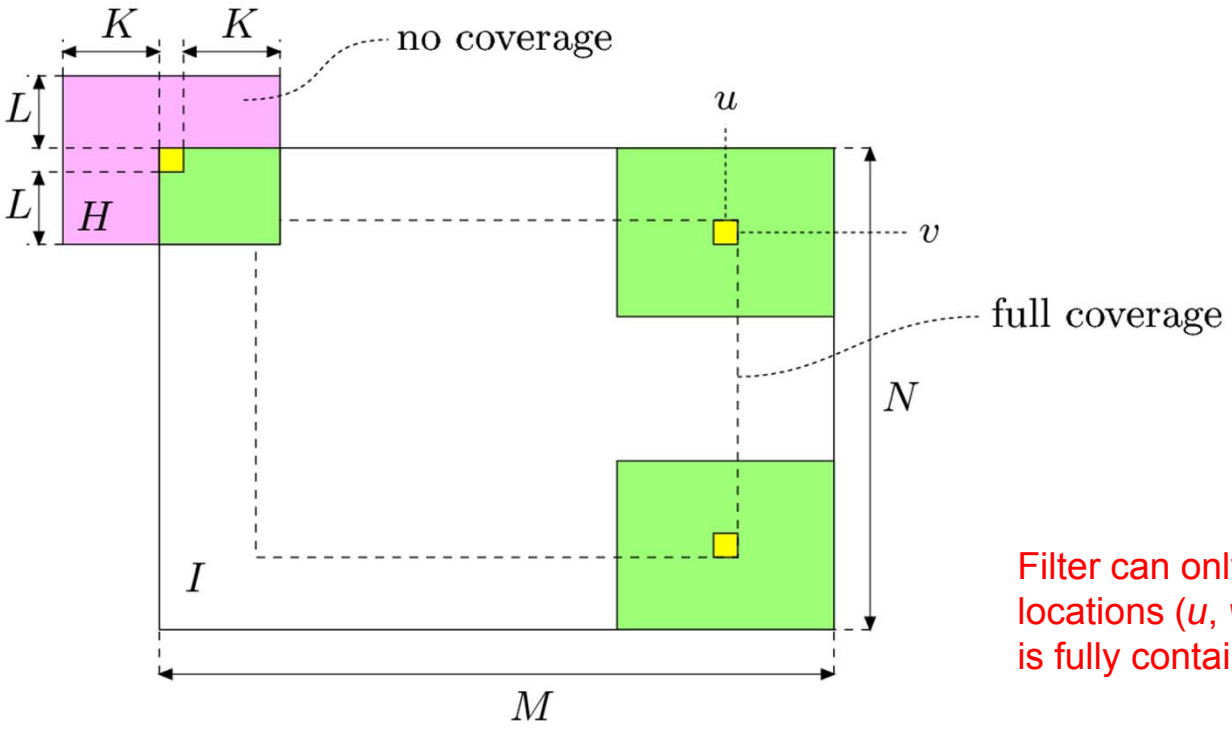
If resulting pixel value is negative, offset shifts it into visible range

Scaling factor for coeffients

Computation Range

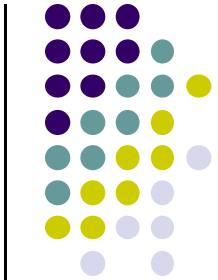
- For a filter of size (2K+1) x (2L+1), if image size is MxN, filter is computed over the range:

$$K \leq u' \leq (M-K-1) \qquad \text{and} \qquad L \leq v' \leq (N-L-1)$$



Filter can only be applied at image locations (*u*, *v*) where filter matrix *H* is fully contained in the image

Filter of Arbitrary Size

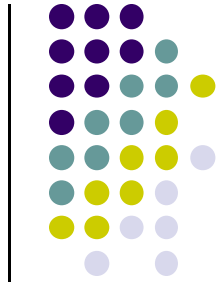


```
1   public void run(ImageProcessor orig) {
2       int M = orig.getWidth();
3       int N = orig.getHeight();
4
5       // filter matrix of size (2K + 1) × (2L + 1)
6       int[][] filter = {
7           {0,0,1,1,1,0,0},
8           {0,1,1,1,1,1,0},
9           {1,1,1,1,1,1,1},
10          {0,1,1,1,1,1,0},
11          {0,0,1,1,1,0,0}
12      };
13      double s = 1.0/23;   // sum of filter coefficients is 23
14
15      int K = filter[0].length/2;
16      int L = filter.length/2;
17
18      ImageProcessor copy = orig.duplicate();
19
20      for (int v = L; v <= N-L-1; v++) {
21          for (int u = K; u <= M-K-1; u++) {
22              // compute filter result for position (u, v)
23              int sum = 0;
24              for (int j = -L; j <= L; j++) {
25                  for (int i = -K; i <= K; i++) {
26                      int p = copy.getPixel(u+i, v+j);
27                      int c = filter[j+L][i+K];
28                      sum = sum + c * p;
29                  }
30              }
31              int q = (int) Math.round(s * sum);
32              if (q < 0)  q = 0;
33              if (q > 255) q = 255;
34              orig.putPixel(u, v, q);
35          }
36      }
37  }
```

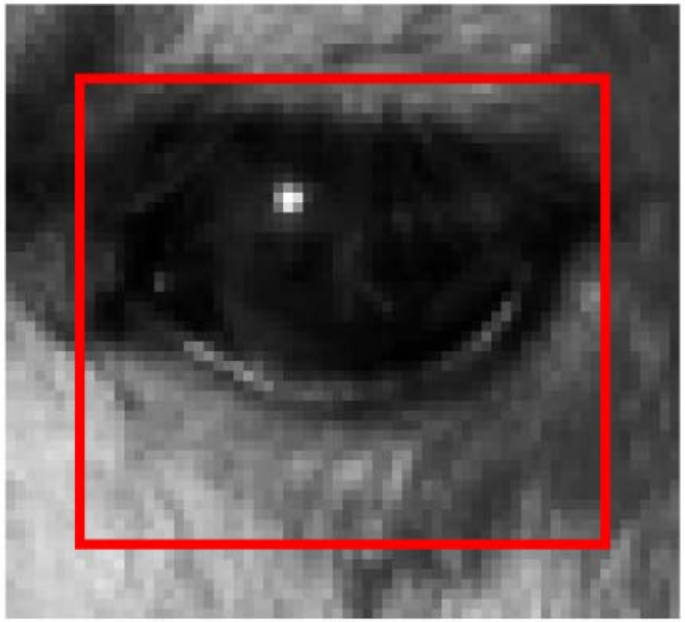
Declare filter

Clamp result as a precaution

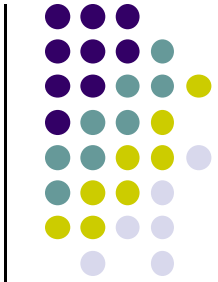
What to do at image boundaries?



a) Crop

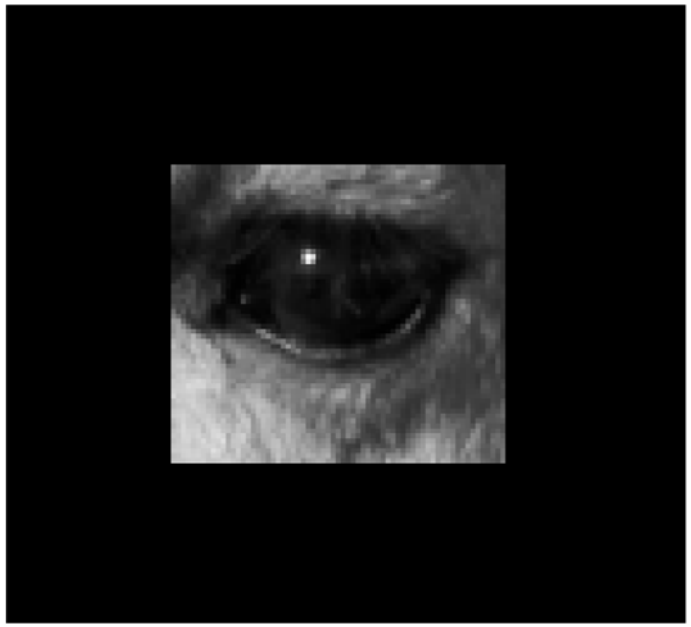


What to do at image boundaries?



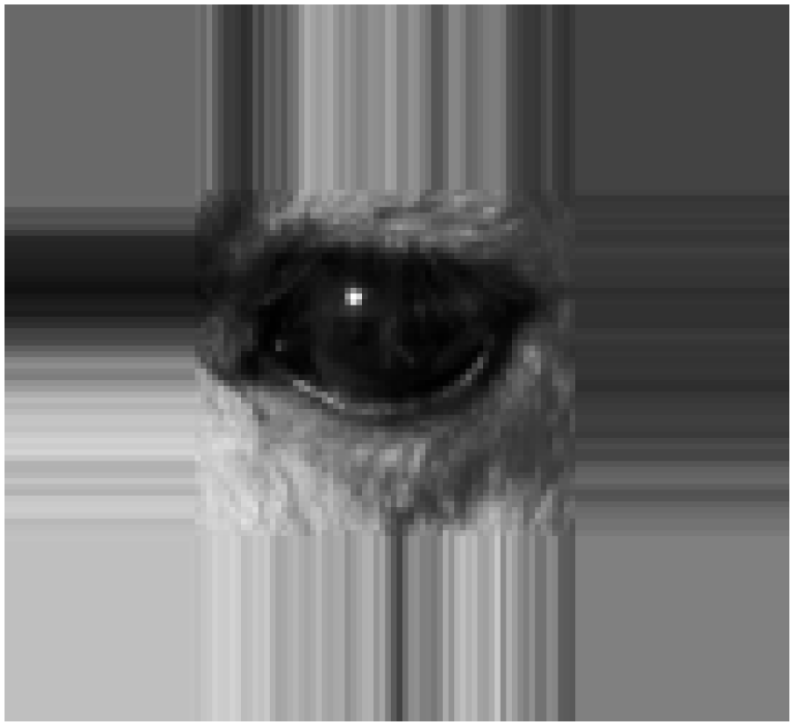
a) Crop

b) Pad

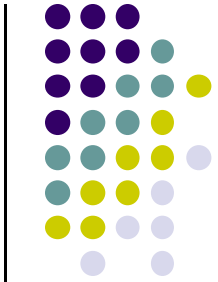


What to do at image boundaries?

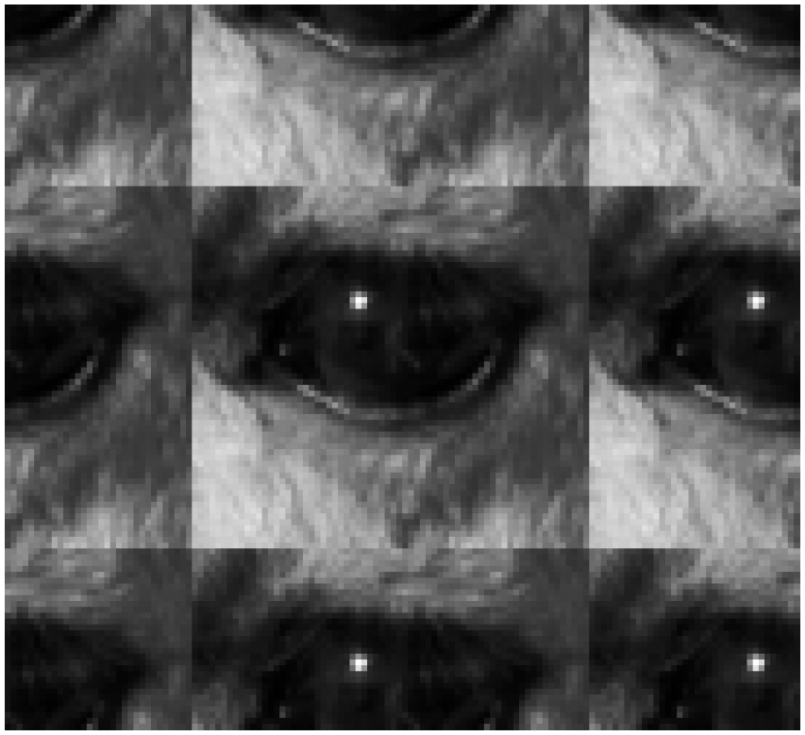
a) Crop
b) Pad
c) Extend



What to do at image boundaries?

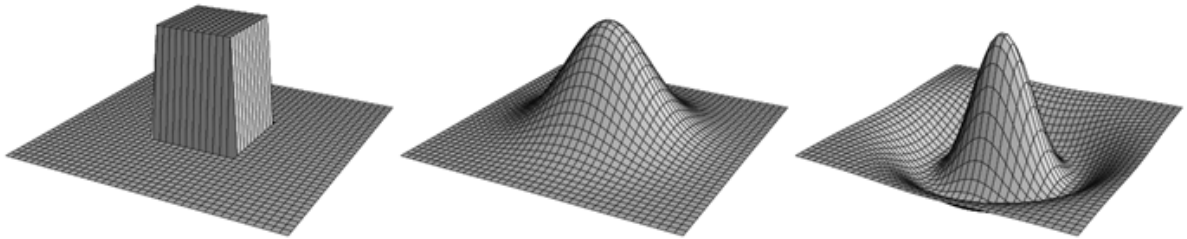


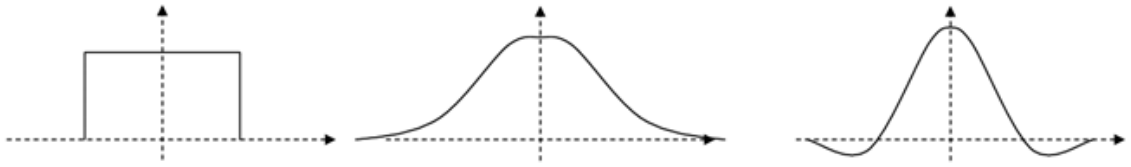
a) Crop
b) Pad
c) Extend
d) Wrap



Linear Filters: Smoothing Filters

- 2 main classes of linear filters:
  - **Smoothing:** +ve coeffients (weighted average). E.g box, gaussian
  - **Difference** filters: +ve and –ve weights. E.g. Laplacian





0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

(a)

Box

0	1	2	1	0
1	3	5	3	1
2	5	9	5	2
1	3	5	3	1
0	1	2	1	0

(b)

Gaussian

0	0	-1	0	0
0	-1	-2	-1	0
-1	-2	16	-2	-1
0	-1	-2	-1	0
0	0	-1	0	0

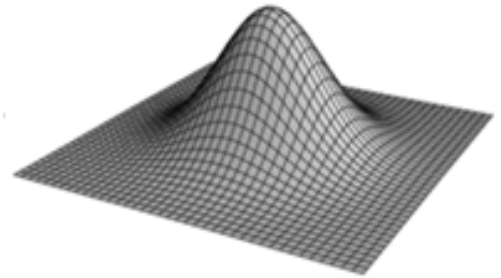
(c)

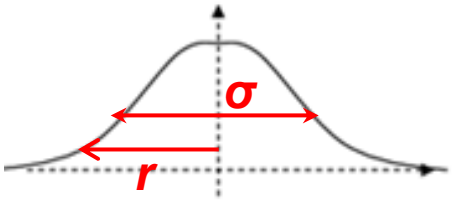
Laplacian

Gaussian Filter

$$G_\sigma(r) = e^{-\frac{r^2}{2\sigma^2}} \qquad \text{or} \qquad G_\sigma(x,y) = e^{-\frac{x^2+y^2}{2\sigma^2}}$$

- where
  - ***σ*** is width (standard deviation)
  - ***r*** is distance from center



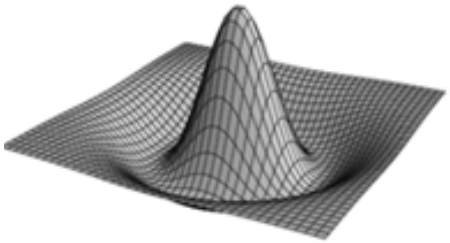


0	1	2	1	0
1	3	5	3	1
2	5	9	5	2
1	3	5	3	1
0	1	2	1	0

Gaussian filter

Difference Filters

- **Coefficients:** some +ve, some negative
- Example: Laplacian filter
- Computation is difference



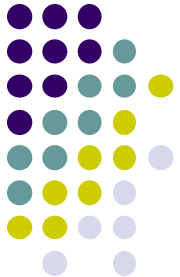
$$\sum(+ve\ coeffients) - \sum(-ve\ coeffients)$$

$$I'(u,v) = \sum_{(i,j)\in R_H^+} I(u+i,v+j)\cdot|H(i,j)| - \sum_{(i,j)\in R_H^-} I(u+i,v+j)\cdot|H(i,j)|$$

0	0	-1	0	0
0	-1	-2	-1	0
-1	-2	16	-2	-1
0	-1	-2	-1	0
0	0	-1	0	0

Laplacian filter

Mathematical Properties of Convolution

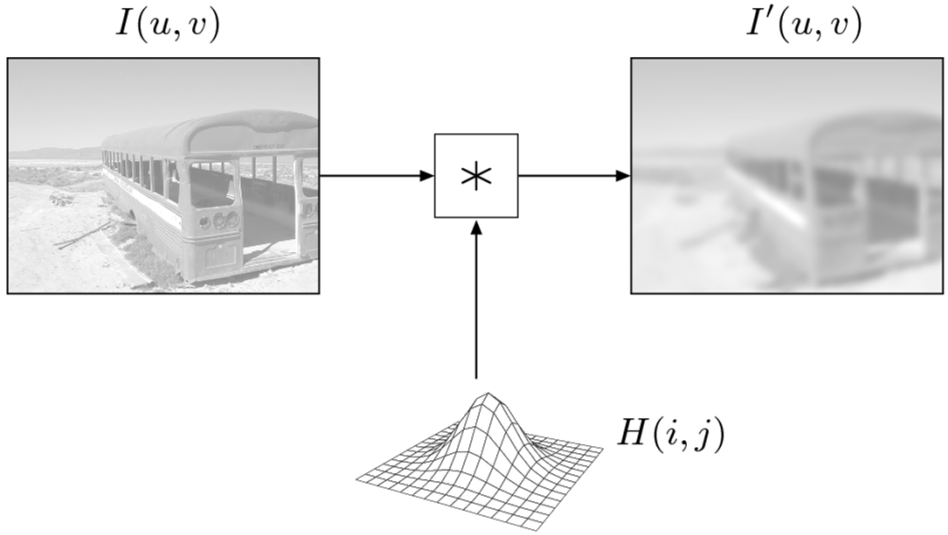


- Applying a filter as described called ***linear convolution***
- For discrete 2D signal, convolution defined as:

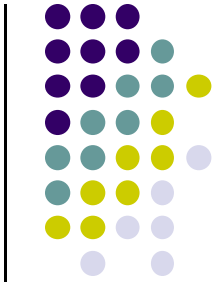
$$I'(u,v) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(u-i, v-j) \cdot H(i,j)$$

Formal definition:
Sum to ± ∞

$$I' = I * H$$



Properties of Convolution



- Commutativity

Same result if we convolve image with filter or vice versa

$$I * H = H * I$$

- Linearity

If image multiplied by scalar
Result multiplied by same scalar

$$(s \cdot I) * H = I * (s \cdot H) = s \cdot (I * H)$$

$$(I_1 + I_2) * H = (I_1 * H) + (I_2 * H)$$

If 2 images added and convolve result with a kernel *H*,
Same result if each image is convolved individually + added

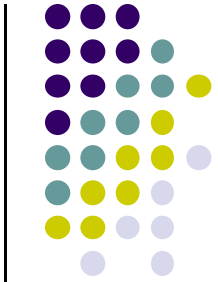
(notice)

$$(b + I) * H \neq b + (I * H)$$

- Associativity

$$A * (B * C) = (A * B) * C$$

Order of filter application irrelevant
Any order, same result



References

- Wilhelm Burger and Mark J. Burge, Digital Image Processing, Springer, 2008
  - Histograms (Ch 4)
  - Point operations (Ch 5)
  - Filters (Ch 6)
- University of Utah, CS 4640: Image Processing Basics, Spring 2012
- Rutgers University, CS 334, Introduction to Imaging and Multimedia, Fall 2012