CSE428: Image Processing

Lecture 5

Neighbourhood processing: Part 1

Contents

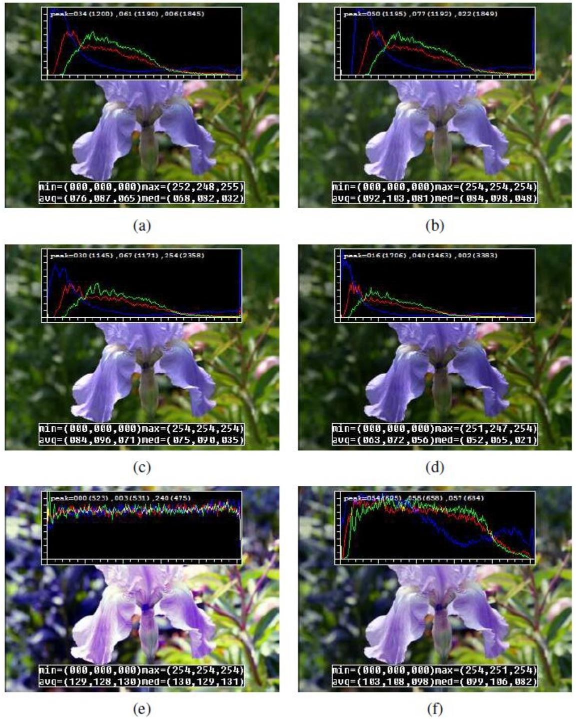
Spatial Operations

Can be subdivided into 3 broad categories:

1. Single-pixel operations or **point processing** (covered in the last lecture)
2. Neighborhood operations or **neighborhood processing**
3. Geometric spatial **transformations** (won’t go into too much detail)

Single-pixel operations

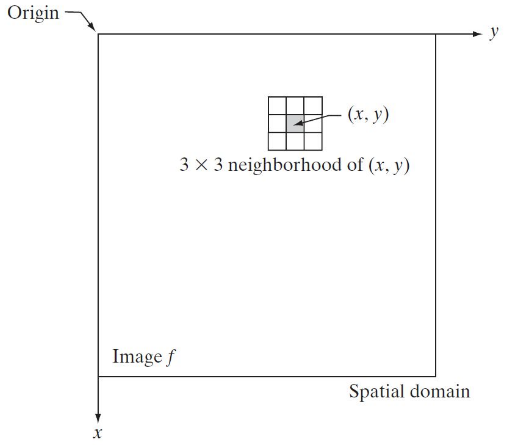
- Simplest possible operation on a digital image: alter the pixel intensities
- If *T* is a transfer function, *s* = *T(z)* or
    - z -> original image pixel intensity s -> mapped intensity
- Some local image processing operations:
    - (a) original image along with its three color histograms;
    - (b) brightness increased (additive offset, b = 16);
    - (c) contrast increased (multiplicative gain, a = 1:1);
    - (d) gamma (partially) linearized (γ = 1:2);
    - (e) full histogram equalization;
    - (f) partial histogram equalization


min=(000,000,000)max=(252,248,255)
avg=(076,087,065)med=(068,082,032)
(a)
min=(000,000,000)max=(254,254,254)
avg=(092,103,081)med=(084,098,048)
(b)
min=(000,000,000)max=(254,254,254)
avg=(084,096,071)med=(075,090,035)
(c)
min=(000,000,000)max=(251,247,254)
avg=(063,072,056)med=(052,065,021)
(d)
min=(000,000,000)max=(254,254,254)
avg=(129,128,130)med=(130,129,131)
(e)
min=(000,000,000)max=(254,251,254)
avg=(103,108,098)med=(099,106,082)
(f)

Computer Vision: Algorithms and Applications, 2nd Edition, Richard Szeliski

Neighborhood

- A pixel p at coordinates has four horizontal and vertical neighbors whose coordinates are given by
  - (x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1)
  - This set of pixels, called the 4-neighbors of p, $N_4(p)$
- The four diagonal neighbors of p
  - (x + 1, y + 1), (x + 1, y - 1), (x - 1, y + 1), (x - 1, y - 1)
  - Diagonal neighbors of p, $N_D(p)$
- 4-neighbors together with the diagonal neighbors are called the 8-neighbors of p: $N_8(p)$



Digital Image Processing, Third Edition,
Rafael C. Gonzalez & Richard E. Woods

Distance measures

- For pixels p, q, and z, with coordinates (x, y), (s, t), and (v, w)
- Euclidean distance between p and q
  - $D_e(p, q) = [(x - s)^2 + (y - t)^2]^{½}$
- D_4 distance (city-block distance) between p and q
  - $D_4(p, q) = |x - s| + |y - t|$
  - The pixels with $D_4 = 1$ are the 4-neighbors of (x, y)
- D_8 distance (chessboard distance) between p and q
  - $D_8(p, q) = \max(|x - s|, |y - t|)$
  - The pixels with $D_8 = 1$ are the 8-neighbors of (x, y)

$$\begin{array}{ccccc} & & 2 & & \\ & 2 & 1 & 2 & \\ 2 & 1 & 0 & 1 & 2 \\ & 2 & 1 & 2 & \\ & & 2 & & \end{array}$$

$$\begin{array}{ccccc} 2 & 2 & 2 & 2 & 2 \\ 2 & 1 & 1 & 1 & 2 \\ 2 & 1 & 0 & 1 & 2 \\ 2 & 1 & 1 & 1 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{array}$$

Digital Image Processing, Third Edition,
Rafael C. Gonzalez & Richard E. Woods

Linear vs Nonlinear Operations

Consider a general operator, *H* such that : $\boldsymbol{H}\,[\boldsymbol{f}(x, y)] = \boldsymbol{g}(x, y)$

H is said to be a linear operator if the following is true:

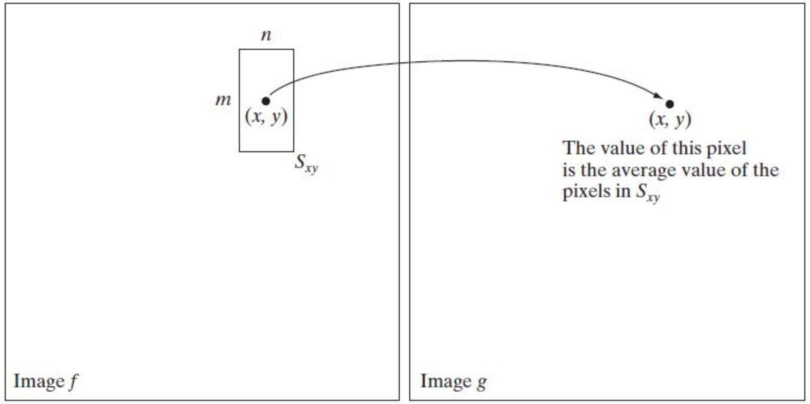
$$H\,[a_1 f_1(x, y) + a_2 f_2(x, y)] = a_1 H\,[f_1(x, y)] + a_2 H\,[f_2(x, y)]$$

1. *Additivity* property: output of a linear operation due to the sum of two inputs is the same as performing the operation on the inputs individually and then summing the results
2. *Homogeneity* property: the output of a linear operation to a constant times an input is the same as the output of the operation due to the original input multiplied by that constant

Neighborhood operations

S_{xy} is the set of coordinates of a neighborhood centered on an arbitrary point (x, y) in an image f

- *Neighborhood processing* generates a corresponding pixel at the same coordinates in an output (processed) image, g, such that the value of that pixel is determined by a specified operation involving the pixels in the input image with coordinates in S_{xy}


n
m
(x, y)
S_{xy}
Image f
(x, y)
The value of this pixel is the average value of the pixels in S_{xy}
Image g

Digital Image Processing, Third Edition, Rafael C. Gonzalez & Richard E. Woods

Neighborhood operations: Mechanism

Spatial filtering consists of

1. A **neighborhood** (defined by the filter kernel, or mask)
2. A **predefined operation** (which is performed on the neighborhood)
    - Linear operation
    - Nonlinear operation

Filtering creates a new pixel at the same position of the center of the neighborhood, and whose value is the result of the filtering operation

A processed (filtered) image is generated as the center of the filter visits each pixel in the input image

Kernels (Also known as Filters/Masks)

- In image processing, a kernel (also known as a filter or convolution mask/matrix) is **a small matrix** used to apply effects like blurring, sharpening, edge detection, and more to an image.
- The kernel is applied to the image through a process called **convolution**, where the kernel is **systematically moved across the image, performing calculations to produce a modified image.**
- The process of filtering is also known as convolving a mask with an image. As this process is same of convolution so filter masks are also known as convolution masks.

Kernels (Also known as Filters/Masks)

- The general process of filtering and applying masks consists of moving the filter/mask from point to point in an image. At each point (x,y) of the original image, the response of a filter is calculated by a pre defined relationship. All the filters values are pre defined.
- Here are some common types of filters:
  1. Smoothing Filters (Low-pass filters)
  2. Sharpening Filters (High-pass filters)
  3. Edge Detection Filters
  4. Non-Linear Filters

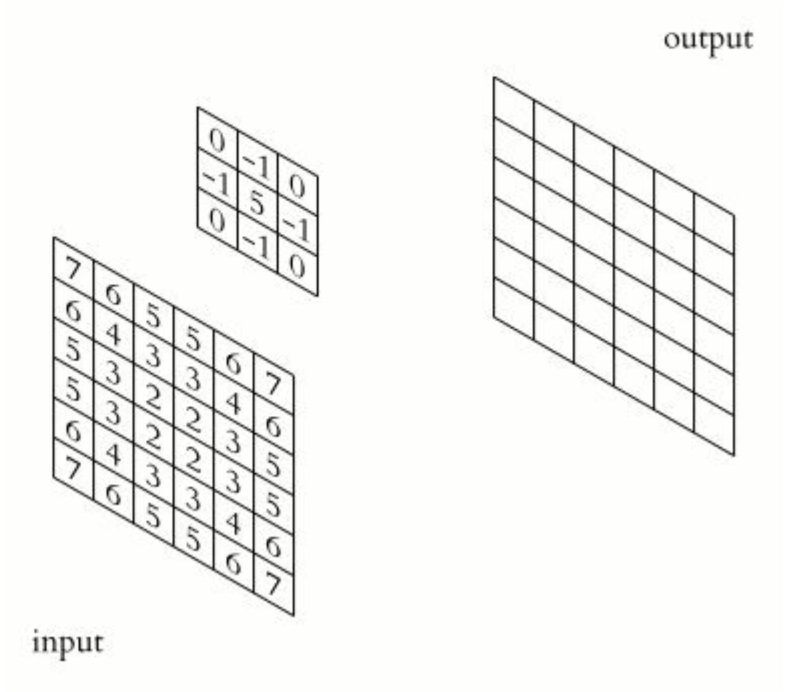
Key Concepts of Kernels

- Kernels are typically small matrices, such as 3x3, 5x5, or 7x7. The size of the kernel determines the area of the image that each calculation affects.
- The values within the kernel matrix determine the specific effect the kernel will have on the image. These values can be positive or negative and are used in weighted sums with the pixel values of the image.
- The kernel is placed on top of the image, covering a portion of it. Each element of the kernel is multiplied by the corresponding image pixel value. The results are summed to get the value of the new pixel in the output image. The kernel is then moved to the next position (typically one pixel to the right or down) and the process is repeated. This whole process is known as **Convolution**.

Key Concepts of Kernels

- The animation here demonstrates the process of filtering. Here, edge Padding was applied to preserve the dimensions of the input image.

- Kernel, $w(x, y) = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

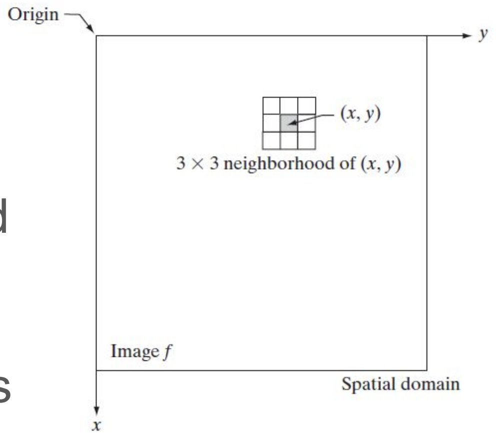


Spatial Filtering

Neighborhood processing is also called ***spatial filtering***

- *g*(x, y) = *T* [*f*(x, y)]
- Input image: *f*(x, y)
- Output image: *g*(x, y)
- *T* [.] is an operator in f, defined over a neighborhood
  - Linear operation -> linear spatial filtering
  - Nonlinear operation -> nonlinear spatial filtering
- *T* can be applied to a single image or multiple images

Example: a 3x3 neighborhood of (x, y)



Digital Image Processing, Third Edition, Rafael C. Gonzalez & Richard E. Woods

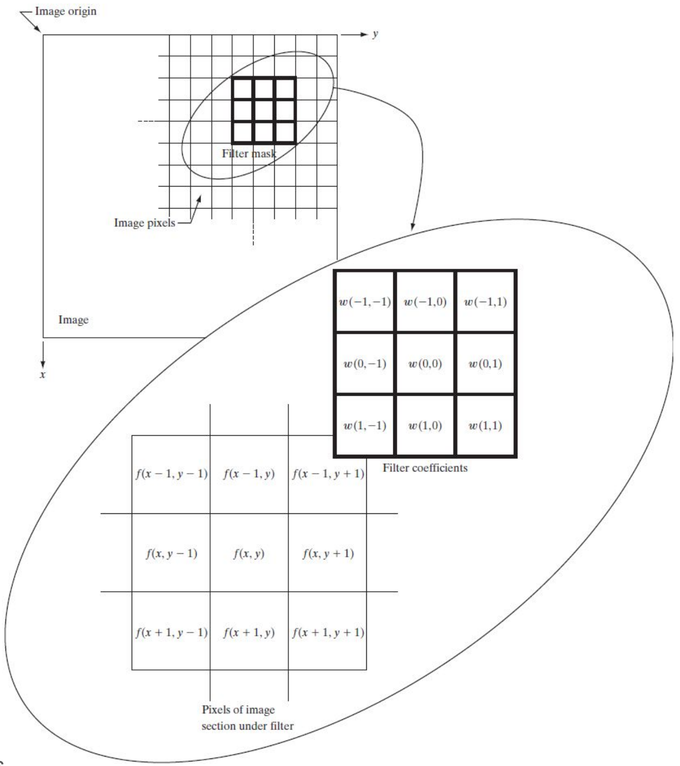
Linear Spatial Filtering

Linear spatial filtering in a m x n neighborhood

- Mask/filter/kernel size: m x n
  - Assume m = 2a + 1, n = 2b + 1 (both odd numbers)
- Image size: M x N

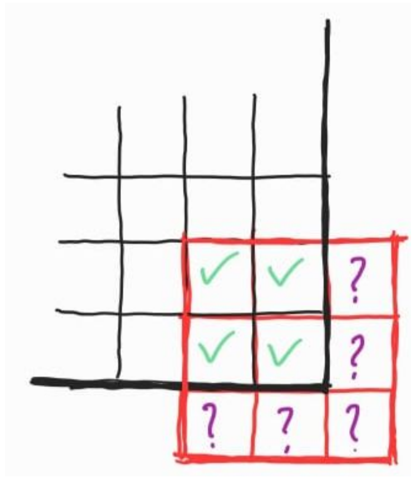
$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t)$$

Example: *g*(x, y) = *w*(-1, -1) *f*(x-1, y-1) + w(-1, 0) *f*(x -1, y) + … + *w*(0, 0) *f*(x, y) + ... + *w*(1, 1) *f*(x+1, y+1)



Digital Image Processing, Third Edition, Rafael C. Gonzalez & Richard E. Woods

But.. what about the border pixels?



Solution 1: Don’t visit the border pixels!

Result: Image is ***shrinked***

Solution 2: *Padding*

Result: Image shape *stays the* ***same***

Image Padding

1. **Zero** padding: extend the borders of the original image with zeros
2. **Constant** padding: extend the borders of the original image with a constant pixel value
3. **Mirror** padding: extend the borders of the original image by reflecting the edge pixels
4. **Clamp/Edge/Nearest** padding: extend the borders of the original image by repeating edge pixels

More ways: https://scikit-image.org/docs/dev/api/skimage.util.html#skimage.util.pad

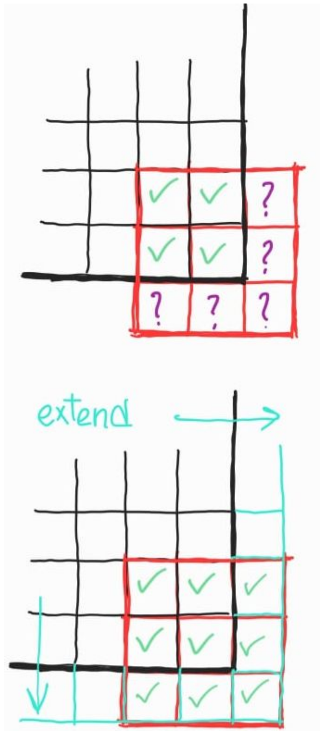
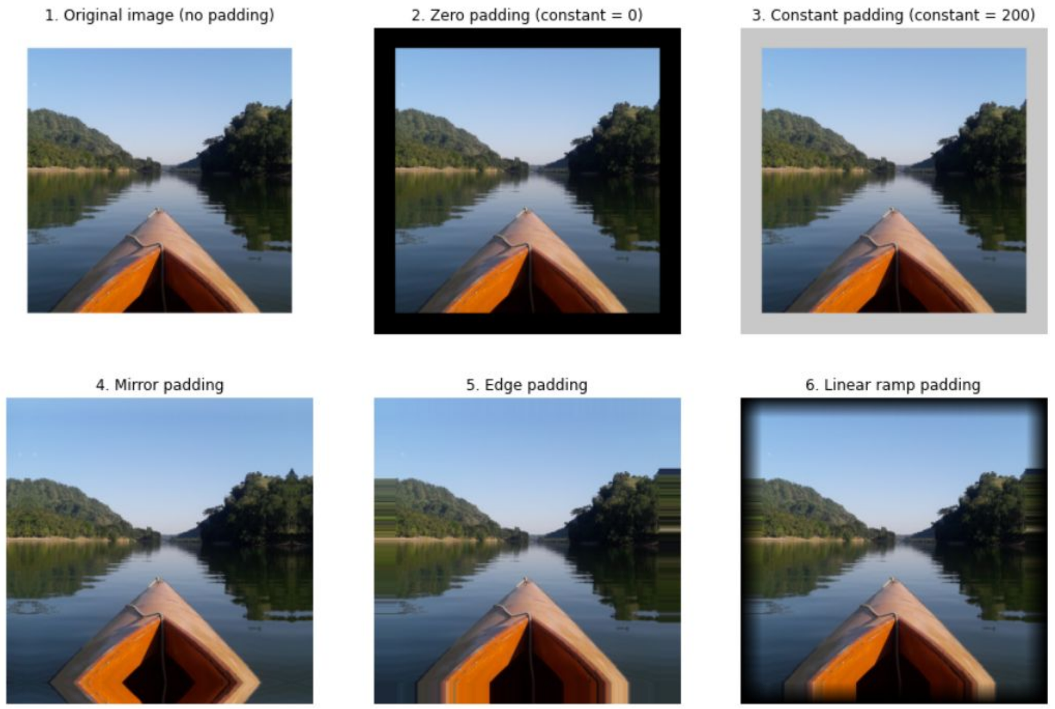


Image Padding

A 1448x1448x3 image padded with 200 pixels on each side:

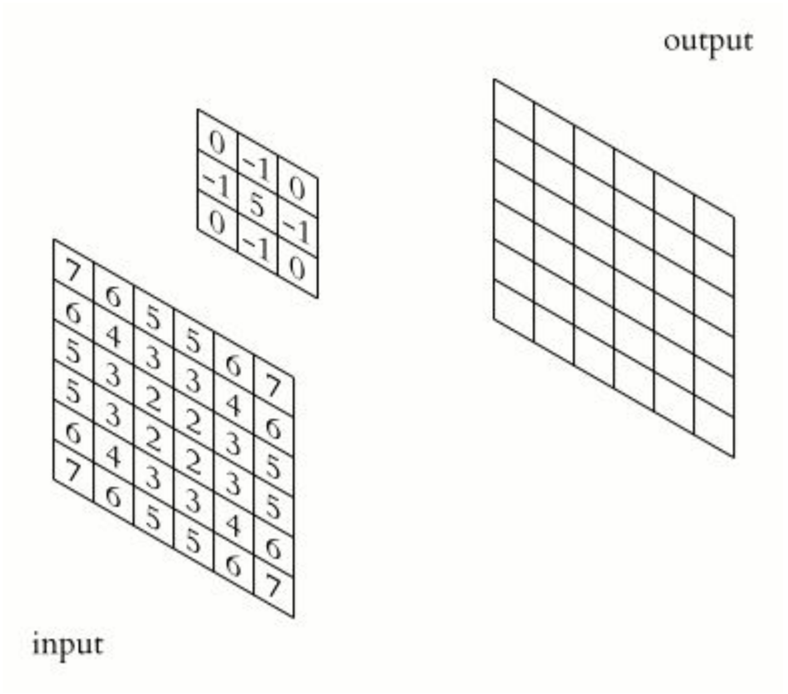
1. Original image
2. Zero padding
3. Constant padding
4. Mirror padding
5. Edge padding
6. Linear(ramp) padding


1. Original image (no padding)
2. Zero padding (constant = 0)
3. Constant padding (constant = 200)
4. Mirror padding
5. Edge padding
6. Linear ramp padding

Linear Spatial Filtering: Demo

- Kernel shape: 3x3
- Input image shape: 6x6
- Output image shape: 6x6
- Padding: Nearest
- Step (stride): 1

- Kernel, $w(x, y) = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$



7	6	5	5	6	7
6	4	3	3	4	6
5	3	2	2	3	5
5	3	2	2	3	5
6	4	3	3	4	6
7	6	5	5	6	7

0	-1	0
-1	5	-1
0	-1	0

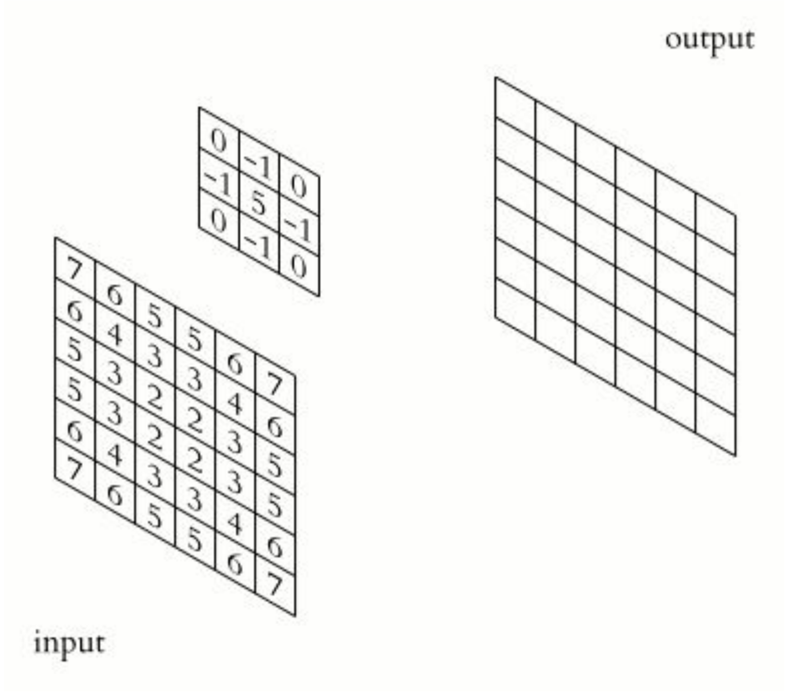
https://commons.wikimedia.org/wiki/File:2D_Convolution_Animation.gif

Linear Spatial Filtering: Output Shape

- Kernel shape: **m** x **n**
- Input image shape: **M** x **N**
- Number of pixels padded: **p**
- Stride: **s**
- Output image shape: **H** x **W**

$$H = floor\left(\frac{M + 2p - m}{s} + 1\right)$$

$$W = floor\left(\frac{N + 2p - n}{s} + 1\right)$$



https://commons.wikimedia.org/wiki/File:2D_Convolution_Animation.gif

Correlation

Correlation between two 2D signals *w*(x, y) & *f*(x, y) is defined as:

$$w(x, y) \star f(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t) f(x+s, y+t)$$

- Outputs a ***measure of similarity*** of the two signals as a function of the displacement of one relative to the other
- Also called the cross-correlation
- Basically a sliding dot product (vector element wise multiplication)
- Used for searching a long signal [*f*(x, y)] for a shorter signal [*w*(x, y)] (a known feature)

Linear spatial filtering ⇔ 2D signal correlation

Correlation to Convolution

- Linear spatial filtering is basically a 2D signal ***correlation*** operation
- If we rotate the original filter *w*(x, y) 180^0 and then proceed, it's called the ***convolution*** operation
- Mathematically,

$$Correlation:\ w(x, y) \star f(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t) f(x+s, y+t)$$

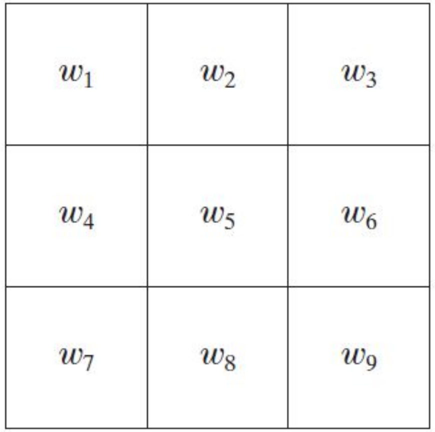
$$Convolution:\ w(x, y) \bigstar f(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t) f(x-s, y-t)$$

For symmetric filters,
convolution and correlation both yield the same result!

Smoothing Spatial Filters

- Linear
  - Box/average filtering
  - Gaussian filtering
- Non-linear
  - Median filtering

Example: A 3 x 3 filter with coefficients w_1 through w_9



w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

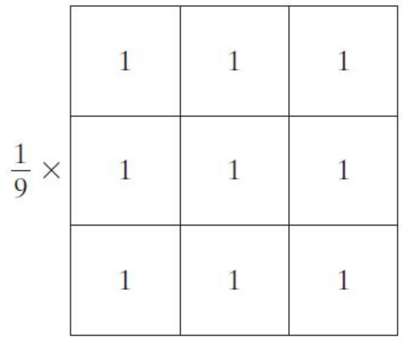
Digital Image Processing, Third Edition,
Rafael C. Gonzalez & Richard E. Woods

Linear Smoothing Filters

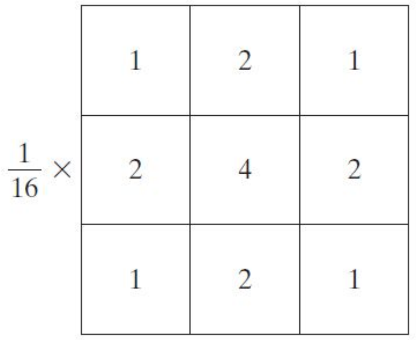
1. Averaging filter
   a. Average of all the pixels encompassed by the filter kernel
   b. An m x n averaging *box kernel*: *w*(x, y) = 1 / m*n
2. Gaussian filter
   a. *Weighted* average of all the pixels encompassed by the gaussian filter kernel
   b. An m x n *gaussian kernel*: *w*(x, y) = exp(-(x^2+y^2)/2σ^2)

Example: a 3x3 Box filter and a 3x3 Gaussian filter

- (note that the filters are <u>normalized</u> so that the intensity of the output pixel remains unchanged)



$$\frac{1}{9}\times\begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline\end{array}$$



$$\frac{1}{16}\times\begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline\end{array}$$

Digital Image Processing, Third Edition,
Rafael C. Gonzalez & Richard E. Woods

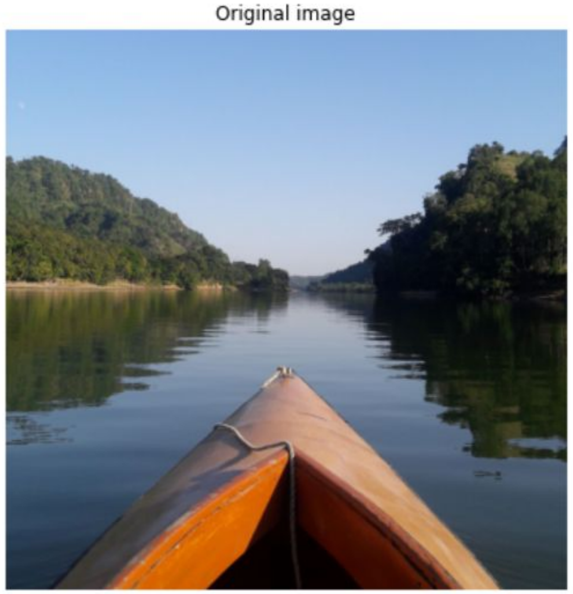
Linear Smoothing Filters

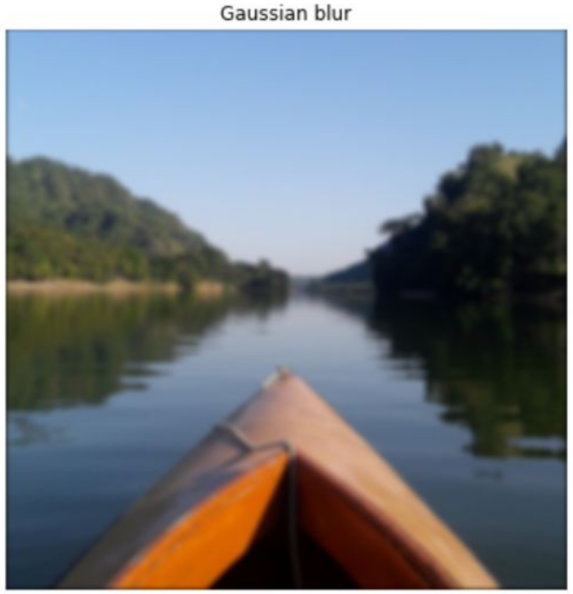
Applications

- Image **Blurring** (As the name suggests)
- Image **Sharpening** (wait, sharpening from blurring? what?)
- Image **Denoising**
- **Low pass filtering** (getting rid of high variations)

Gaussian Filtering:The Gaussian Blur

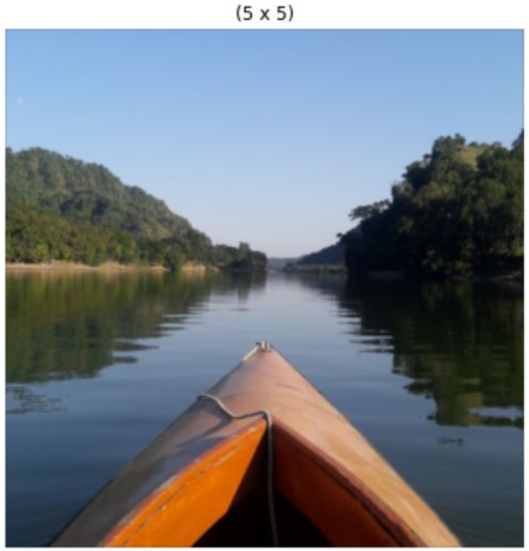
Result of filtering an image with with (25 x 25), $\sigma^2 = 8$ Gaussian kernel

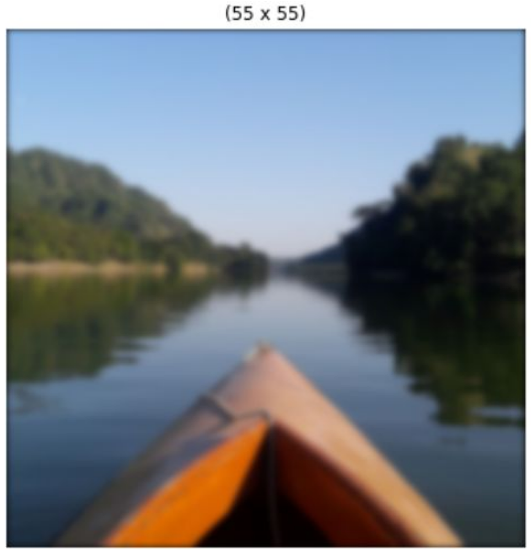


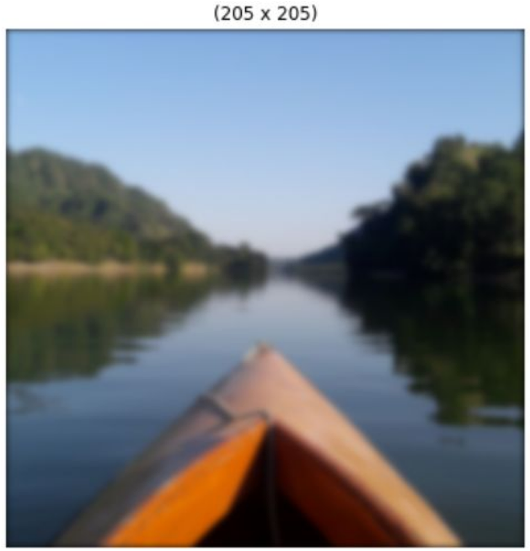


Gaussian Filtering:The Gaussian Blur

Increasing the kernel shape increases blurriness, borders become more prominent



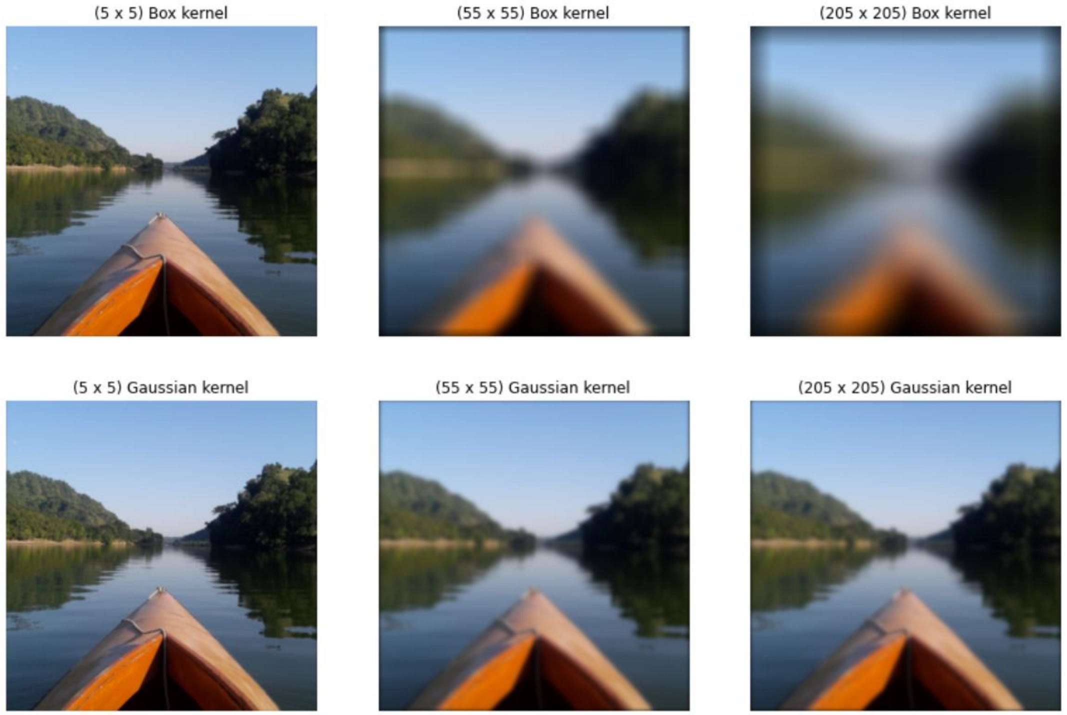




Gaussian Filtering vs Average Filtering

For the same kernel size the gaussian blur *preserves more detail* of the image

Even though the image is blurred the edges are somewhat better preserved for the gaussian blurring (edge preserving blurring)



Order Statistic Filtering

Response is based on *ordering* (ranking) the pixels contained in the pixel neighborhood (S_{xy})

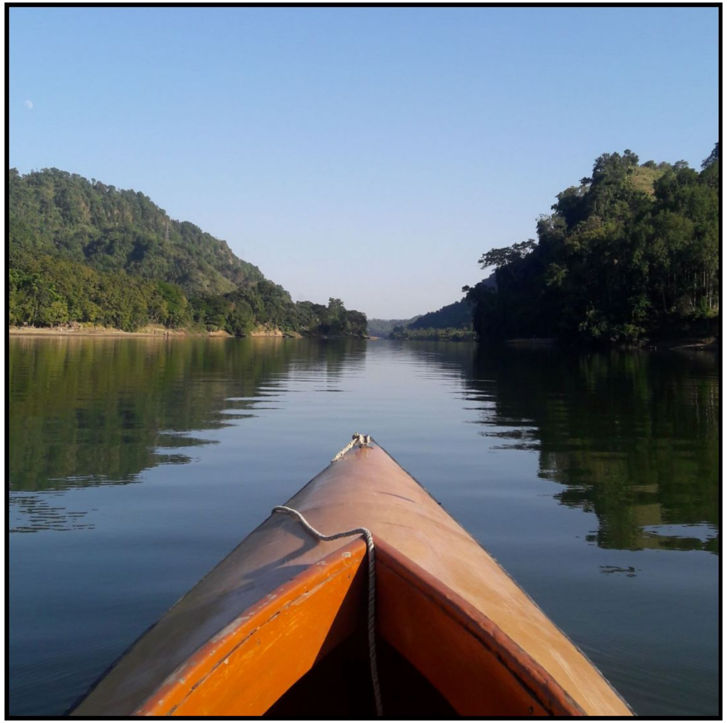
- **Median** filtering: $\hat{f}(x,y) = \underset{(s,t)\in S_{xy}}{\text{median}}\{g(s,t)\}$
- **Max** and **min** filtering: $\hat{f}(x,y) = \max_{(s,t)\in S_{xy}}\{g(s,t)\}$ $\hat{f}(x,y) = \min_{(s,t)\in S_{xy}}\{g(s,t)\}$
- **Midpoint** filtering: $\hat{f}(x,y) = \frac{1}{2}\left[\max_{(s,t)\in S_{xy}}\{g(s,t)\} + \min_{(s,t)\in S_{xy}}\{g(s,t)\}\right]$
- **Alpha-trimmed** mean filtering: $\hat{f}(x,y) = \frac{1}{mn-d}\sum_{(s,t)\in S_{xy}} g_r(s,t)$

Median Filtering

Median Filter (Order-statistic filter)

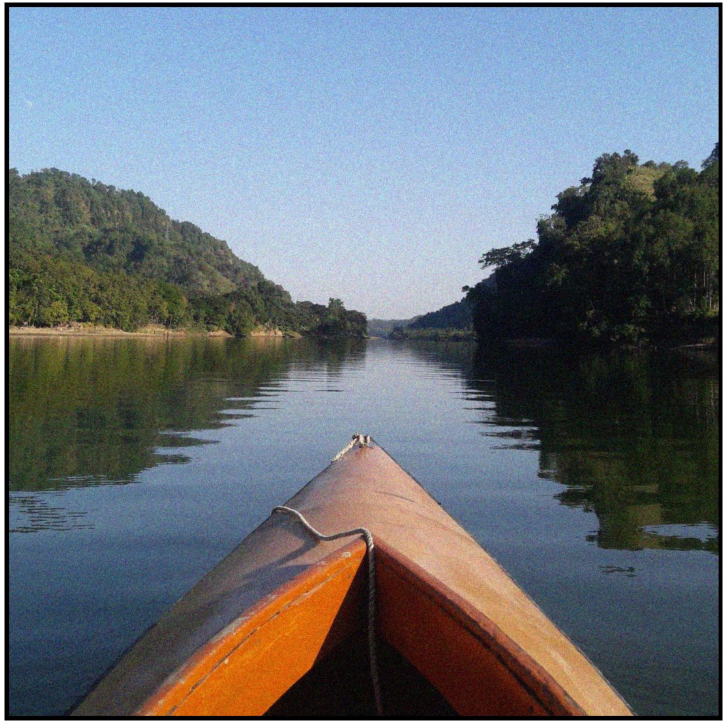
- nonlinear spatial filter
- each pixel value is replaced by the median of the intensity values in the neighborhood
- excellent noise-reduction capabilities
- less blurring than linear filters
- particularly effective with impulse noise (salt and pepper noise)

Noise-corrupted Image

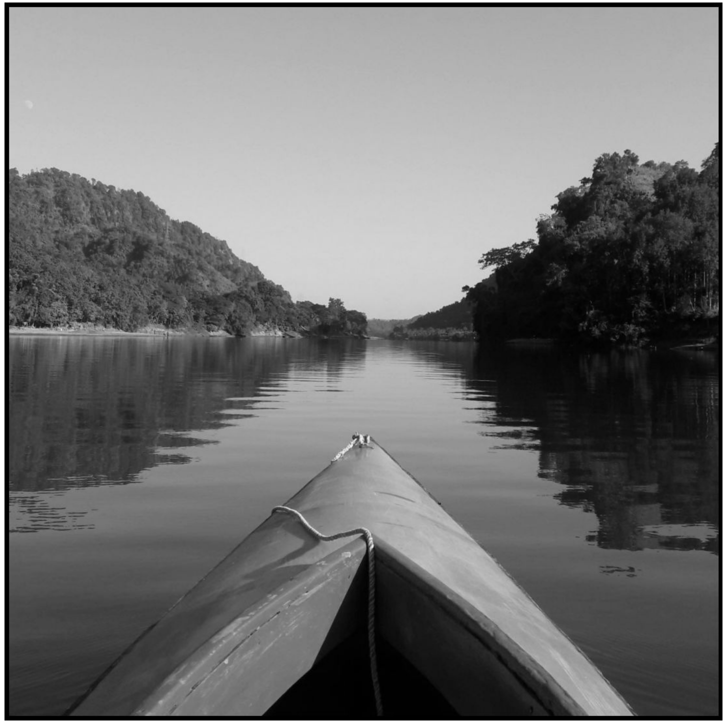


< Original image (left)

Noisy image > (right) (Gaussian noise)

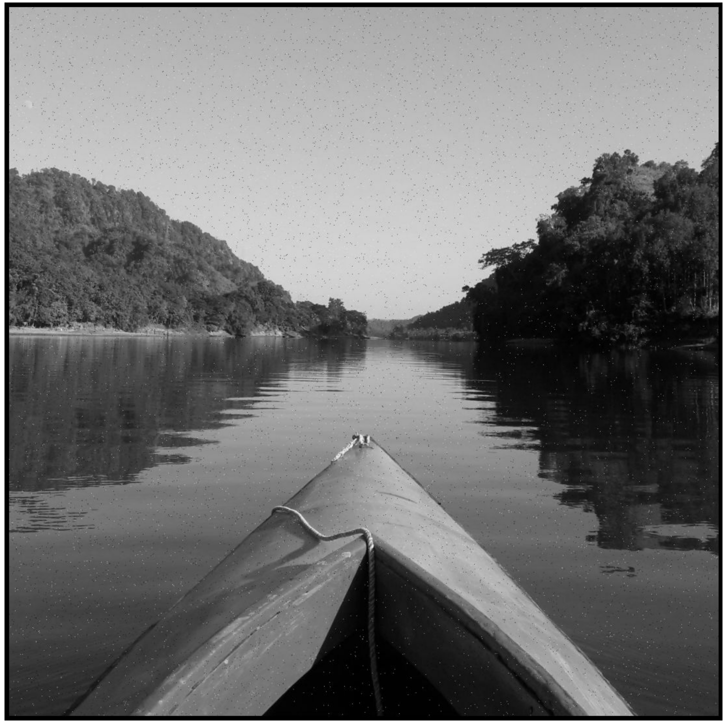


Noise Reduction Using Linear Filtering

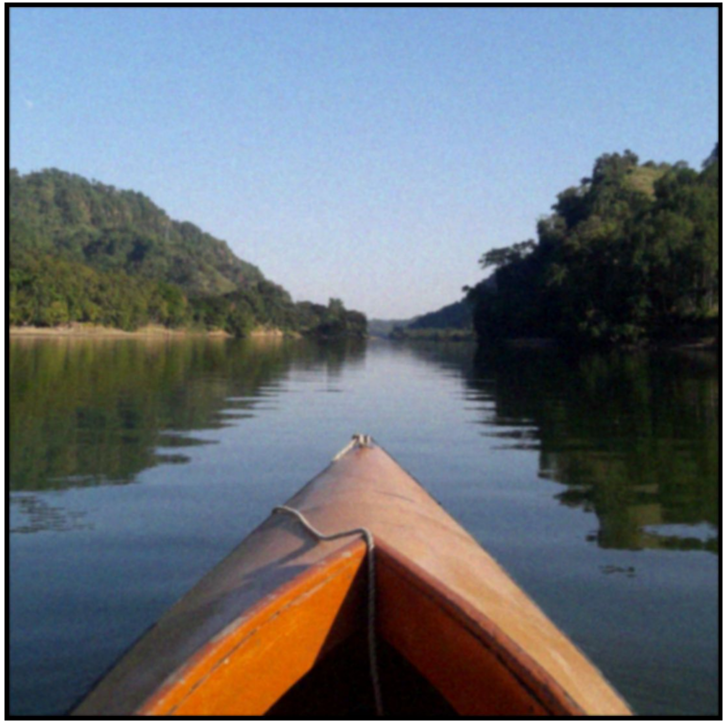


< Original grayscale image (left)

Noisy image **>** (right) (Salt and pepper noise, pretty common for old b&w cameras)

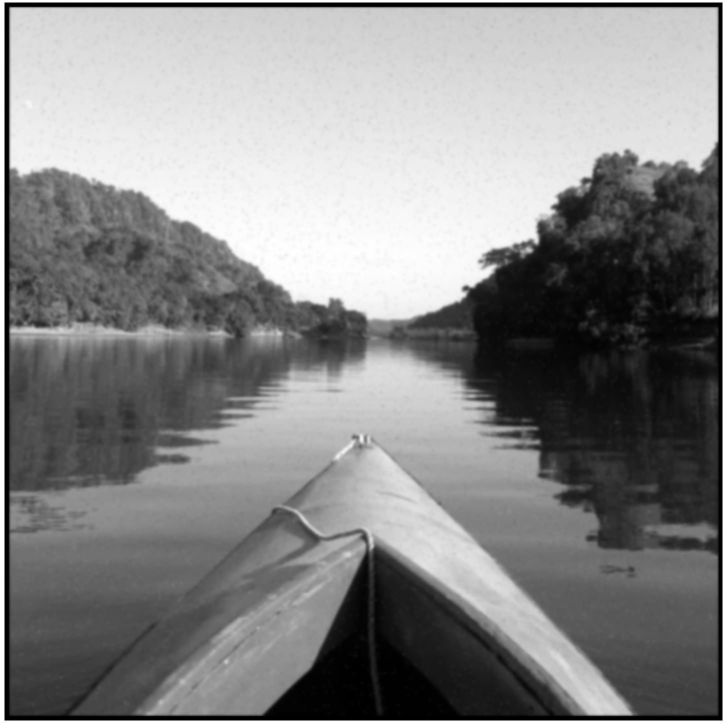


Noise Reduction Using Linear Filtering

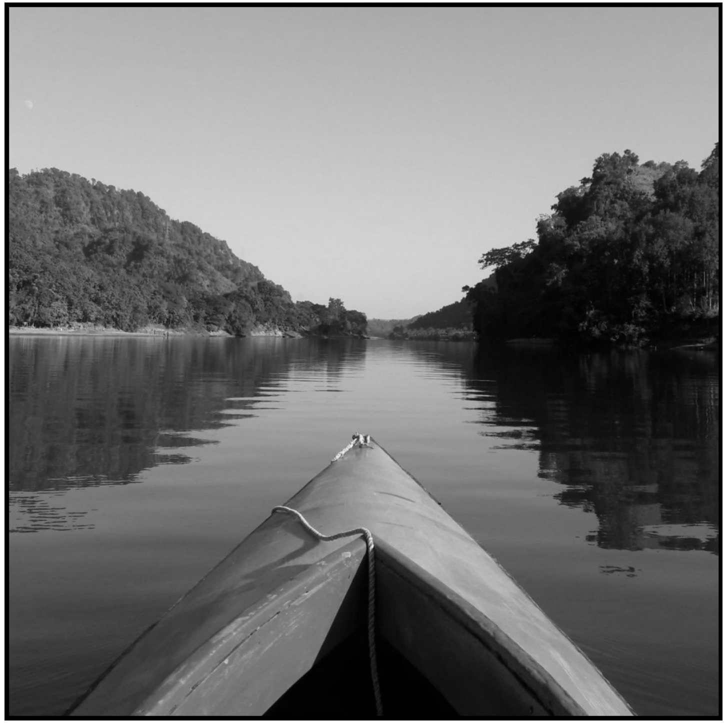


Gaussian filter used to denoise both noisy images (25 x 25) $\sigma^2=3$

Output is blurred, s&p not properly removed

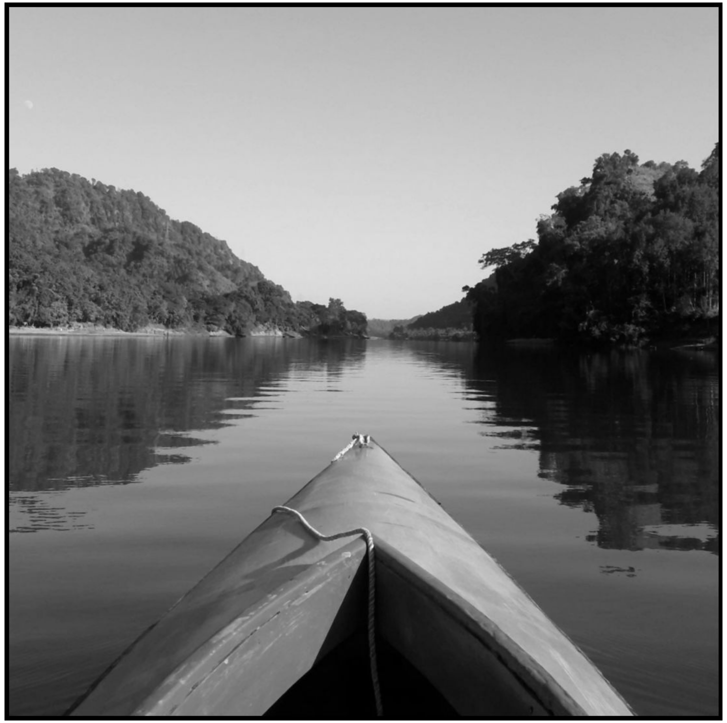


Noise Reduction Using Nonlinear Filtering



Median filter is used to denoise the salt and pepper noise

(can you tell the original from the filtered one?)

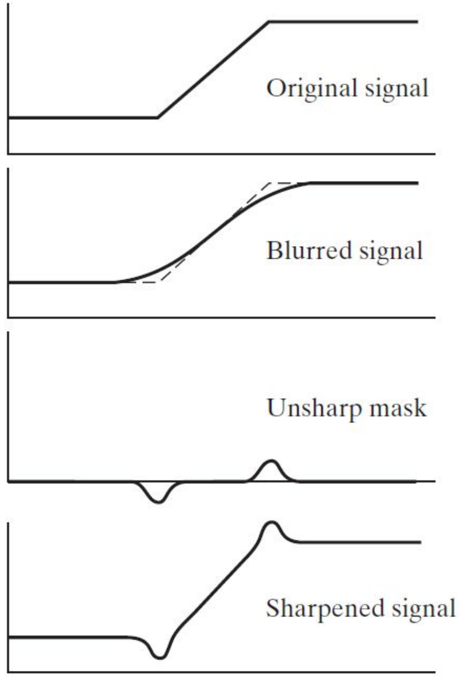


Unsharp Masking & High Boost Filtering

This unsharp masking process consists of the following steps:

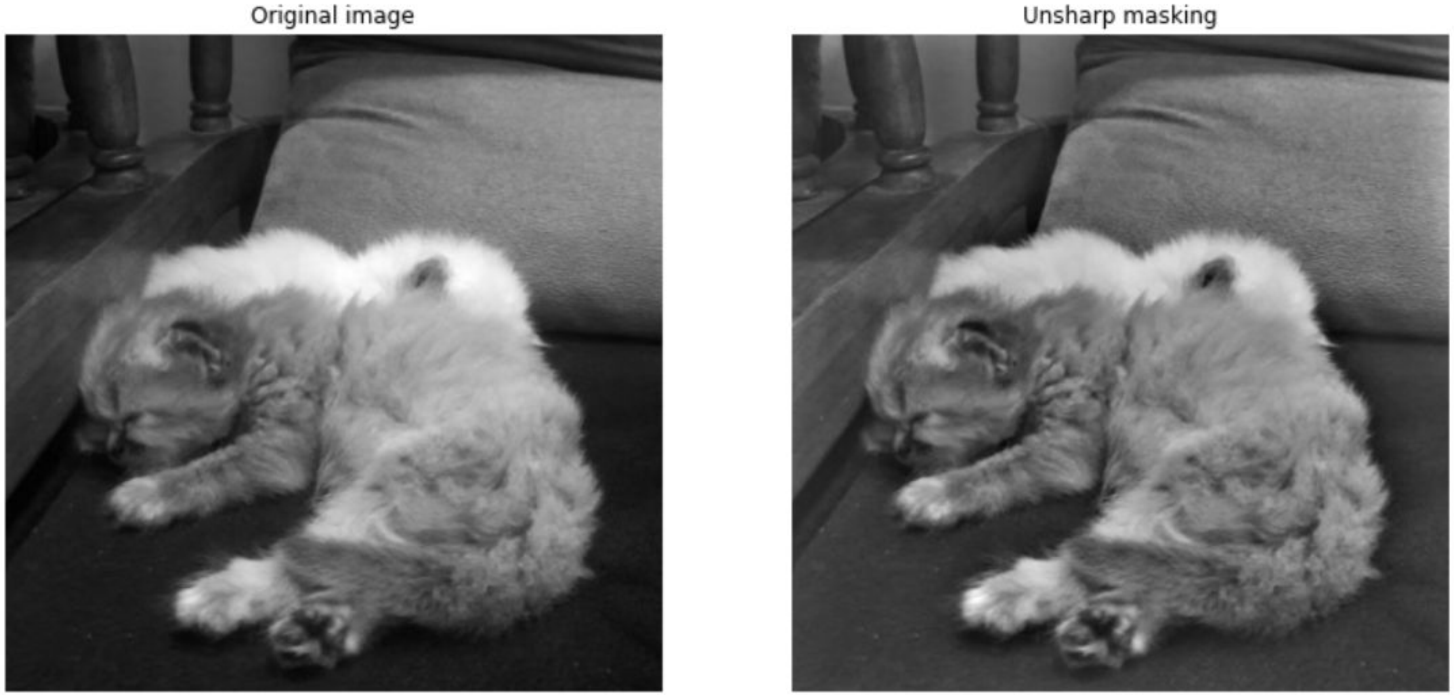
1. Blur the original image
   a. $f'(x, y) = \text{Blur}[f(x, y)]$
2. Subtract the blurred image from the original (the resulting difference is called the mask)
   a. $g_{mask}(x, y) = f(x, y) - f'(x, y)$
3. Add the mask to the original
   a. $g(x, y) = f(x, y) + \boldsymbol{k} * g_{mask}(x, y)$

***k* = 1** is *unsharp masking*, ***k* > 1** is *high boost filtering*

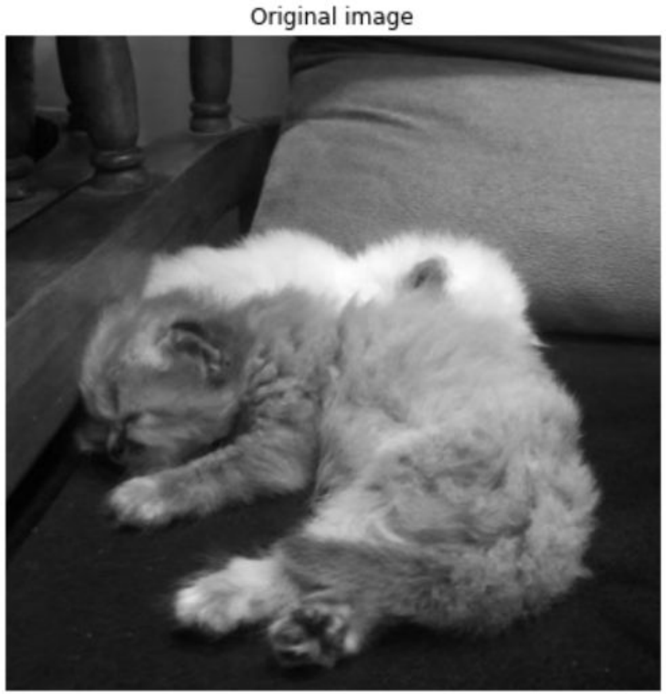


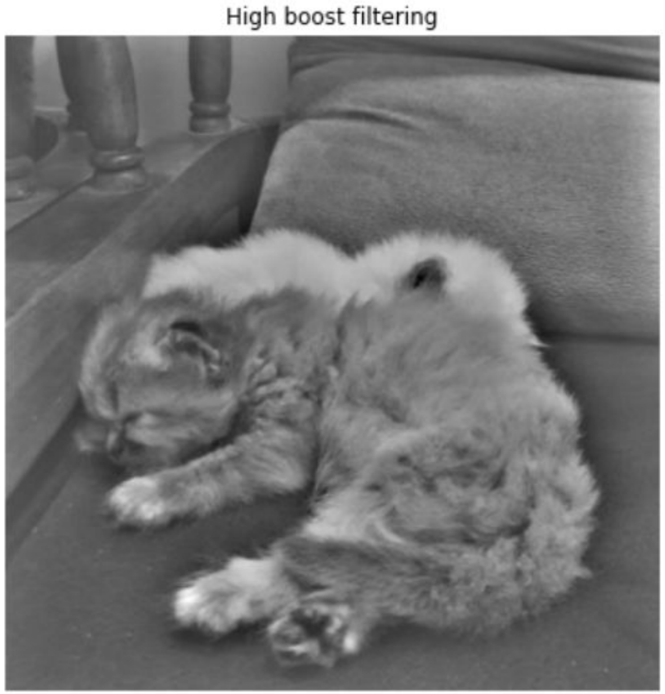
Digital Image Processing, Third Edition, Rafael C. Gonzalez & Richard E. Woods

Unsharp Masking



High Boost Filtering





Edge Detection

- Sudden changes of discontinuities in an image are known as edges. Significant transitions in an image are called as edges.
- Most of the shape information of an image is enclosed in edges.
- Often we detect these edges in an image by using filters and then by enhancing those areas of image which contain edges, we increase the sharpness of the image and the image becomes clearer.
- Some of the masks used for edge detection are: Prewitt Operator, Sobel Operator, Robinson Compass Masks, Krisch Compass Masks, Laplacian Operator.

Derivative Masks

- Edges are calculated by using difference between corresponding pixel intensities of an image. All the masks that are used for edge detection are also known as derivative masks.
- An image is a signal so changes in a signal can only be calculated using differentiation. That's why these operators are also called derivative operators or derivative masks.
- All the derivative masks should have the following properties:
  1. Opposite sign should be present in the mask.
  2. Sum of mask should be equal to zero.
  3. More weight means more edge detection.

Prewitt Operator

- Prewitt operator provides us with two separate masks. One for detecting edges in **horizontal** direction and another for detecting edges in an **vertical** direction.
- Mask for detecting edges in Vertical direction:

  $$\begin{matrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{matrix}$$

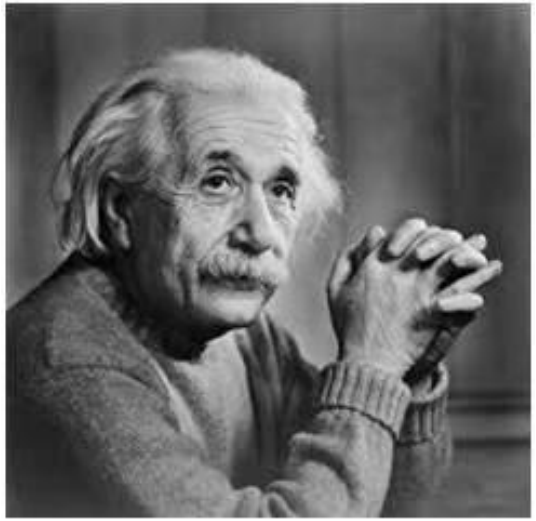
- Mask for detecting edges in Horizontal direction:

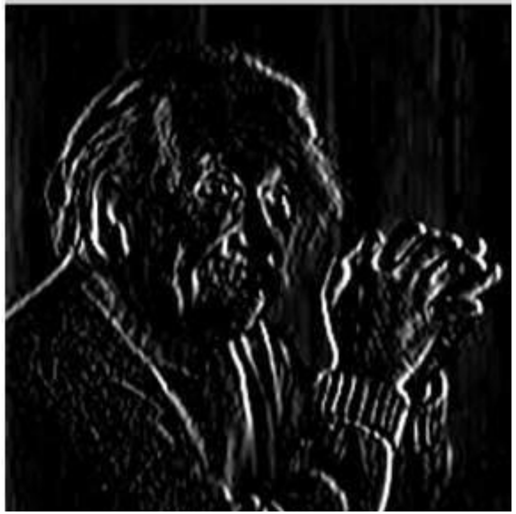
  $$\begin{matrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{matrix}$$

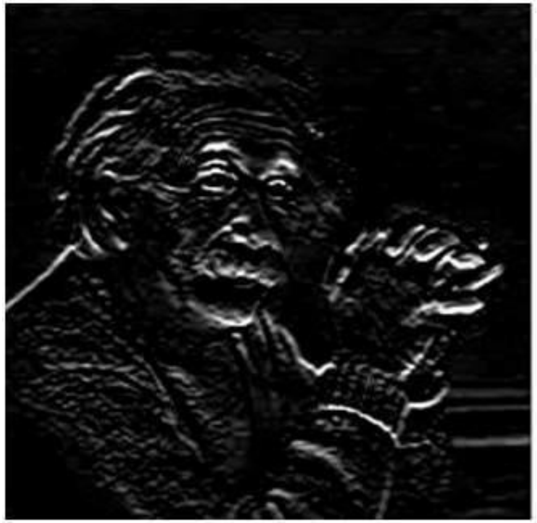
Prewitt Operator

- When we apply the vertical one it prominents the vertical edges. It simply works like as first order derivative and calculates the difference of pixel intensities in a edge region. As the center column is of zero, it does not include the original values of a pixel but rather it calculates the difference of right and left pixel values around that edge. This increases the edge intensity and it become enhanced comparatively to the original image.
- Similar thing happens for the horizontal one too. As the center row of the mask consists of zeros so it does not include the original pixel values of edges in the image, rather it calculates the difference of above and below pixel intensities of the particular edge. Thus increasing the sudden change of intensities and making the edge more visible. Both the above masks follow the principle of derivative mask. Both masks have opposite sign in them and both masks sum equals to zero.

Prewitt Operator


Original Image


After applying Vertical Mask


After applying Horizontal Mask

Sobel Operator

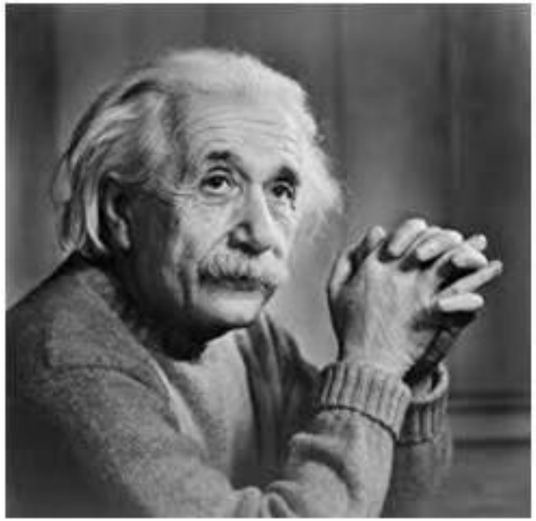
- Sobel operator also provides us with two separate masks. One for detecting edges in **horizontal** direction and another for detecting edges in an **vertical** direction.
- The Sobel operator usually incorporates a weighting of 2 in the middle row/column of the kernels. This weighted approach gives more emphasis to the central pixels, making the Sobel operator more sensitive to edges and less sensitive to noise compared to the Prewitt operator.
- Mask for detecting edges in Vertical direction:

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

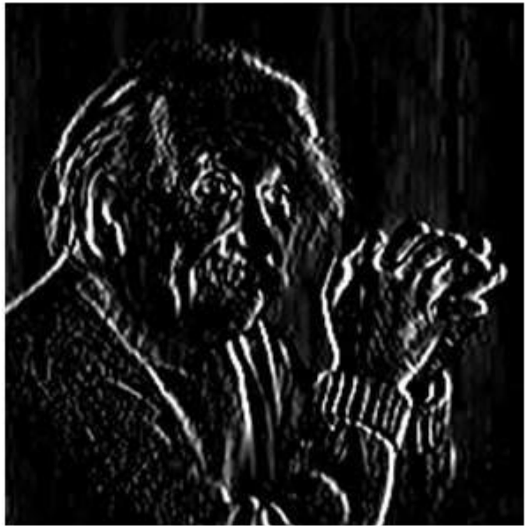
- Mask for detecting edges in Horizontal direction:

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

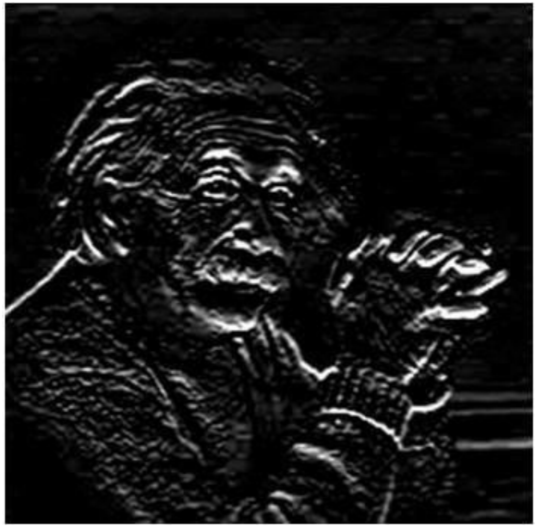
Sobel Operator



Original Image



After applying Vertical Mask



After applying Horizontal Mask

Robinson Compass Masks

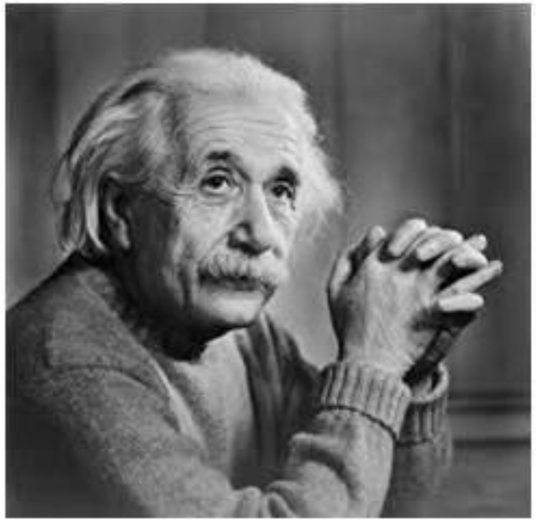
- Robinson compass mask is another type of derivative mask which is used for edge detection. This operator is also known as direction mask. In this operator we take one mask and rotate it in all the 8 compass major directions that are following: *North, North West, West, South West, South, South East, East, North East*. Each mask gives us the edges on its direction.
- Ex.: Mask for detecting edges in North West direction:

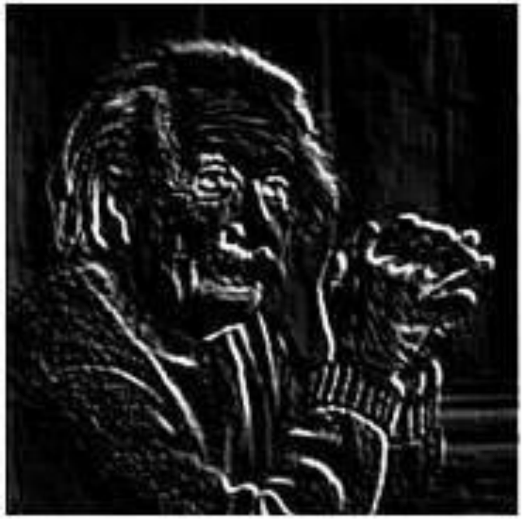
$$\begin{matrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{matrix}$$

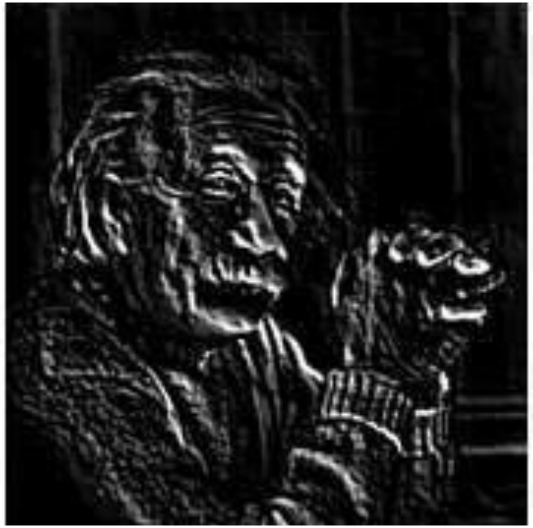
- Ex.: Mask for detecting edges in East direction:

$$\begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix}$$

Robinson Compass Masks


Original Image


After applying Mask for North West Direction


After applying Mask for South East Direction

Laplacian Operator

- Laplacian Operator is also a derivative operator which is used to find edges in an image. The major difference between Laplacian and other operators like Prewitt, Sobel, Robinson and Kirsch is that these all are first order derivative masks but Laplacian is a second order derivative mask. In this mask we have two further classifications one is Positive Laplacian Operator and other is Negative Laplacian Operator.
- Another difference between Laplacian and other operators is that unlike other operators Laplacian didn’t take out edges in any particular direction but it take out edges in following classification.

  1. Inward Edges
  2. Outward Edges

Laplacian Operator

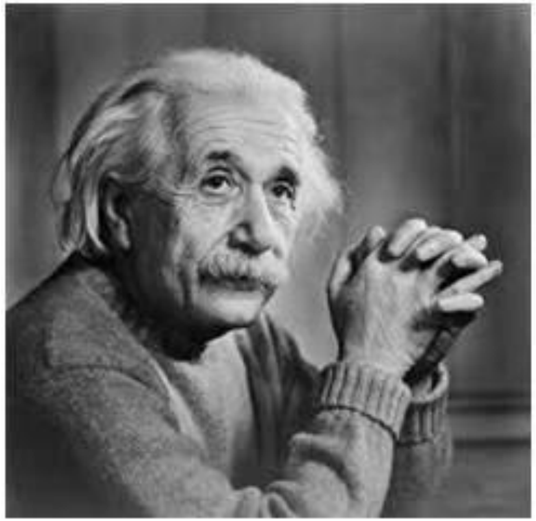
- Positive Laplacian Operator:

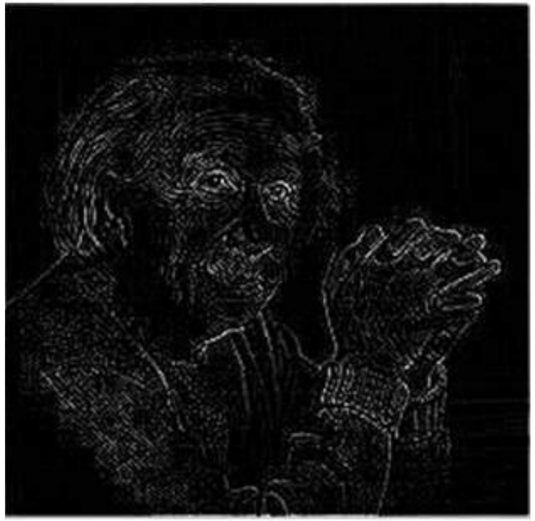
$$\begin{matrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{matrix}$$

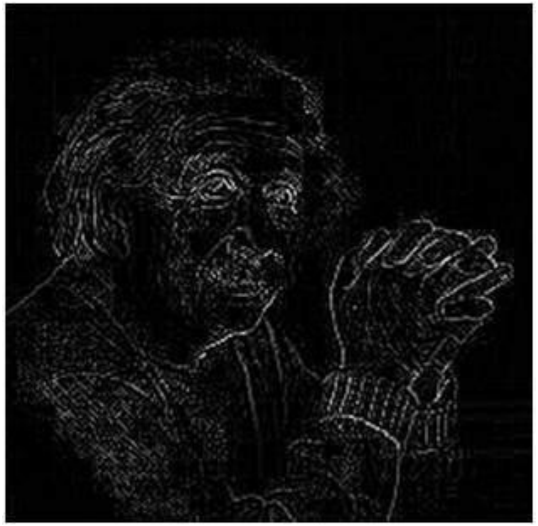
- Negative Laplacian Operator:

$$\begin{matrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{matrix}$$

Laplacian Operator


Original Image


After applying Positive Laplacian Operator


After applying Negative Laplacian Operator

Geometric Transformations

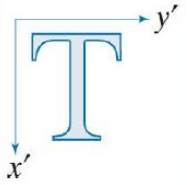
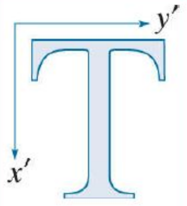
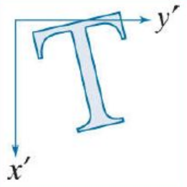
- We use geometric transformations modify the spatial arrangement of pixels in an image
- These transformations are called rubber-sheet transformations because they may be viewed as analogous to “printing” an image on a rubber sheet, then stretching or shrinking the sheet according to a predefined set of rules

Geometric Transformations

- (x, y) are pixel coordinates in the original image
- (x’, y’) are the corresponding pixel coordinates of the transformed image
- Our interest is in so-called affine transformations, which include scaling, translation, rotation, and shearing
- The key characteristic of an affine transformation in 2-D is that it preserves points, straight lines, and planes
-

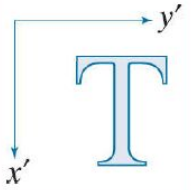
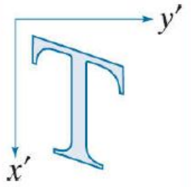
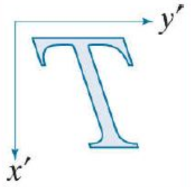
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \mathbf{A} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Geometric Transformations

Transformation Name	Affine Matrix, A	Coordinate Equations	Example
Identity	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x$ $y' = y$	
Scaling/Reflection (For reflection, set one scaling factor to -1 and the other to 0)	$\begin{bmatrix} c_x & 0 & 0 \\ 0 & c_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = c_x x$ $y' = c_y y$	
Rotation (about the origin)	$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x\cos\theta - y\sin\theta$ $y' = x\sin\theta + y\cos\theta$	

Digital Image Processing, International Edition, Rafael C. Gonzalez & Richard E. Woods

Geometric Transformations

Transformation Name	Affine Matrix, A	Coordinate Equations	Example
Translation	$\begin{bmatrix}1 & 0 & t_x\\0 & 1 & t_y\\0 & 0 & 1\end{bmatrix}$	$x' = x + t_x$ $y' = y + t_y$	
Shear (vertical)	$\begin{bmatrix}1 & s_v & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}$	$x' = x + s_v y$ $y' = y$	
Shear (horizontal)	$\begin{bmatrix}1 & 0 & 0\\s_h & 1 & 0\\0 & 0 & 1\end{bmatrix}$	$x' = x$ $y' = s_h x + y$	

Digital Image Processing, International Edition, Rafael C. Gonzalez & Richard E. Woods

Tutorial

Basic python implementation of some of the algorithms taught in class can be found in the following google colab notebook: https://colab.research.google.com/drive/1JF3ww3l-bBzoC0iy9Ado7eQLcxC1hVFU?usp=sharing

Thank you!