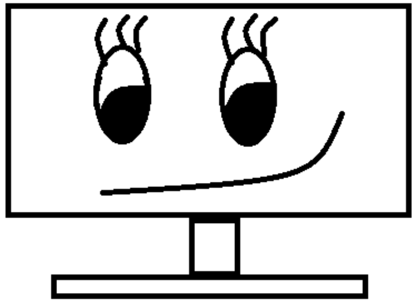
Seneca



CVI620/ DPS920
Introduction to Computer Vision

Geometric Transformations, Noise & Filtering

Seneca College

Vida Movahedi

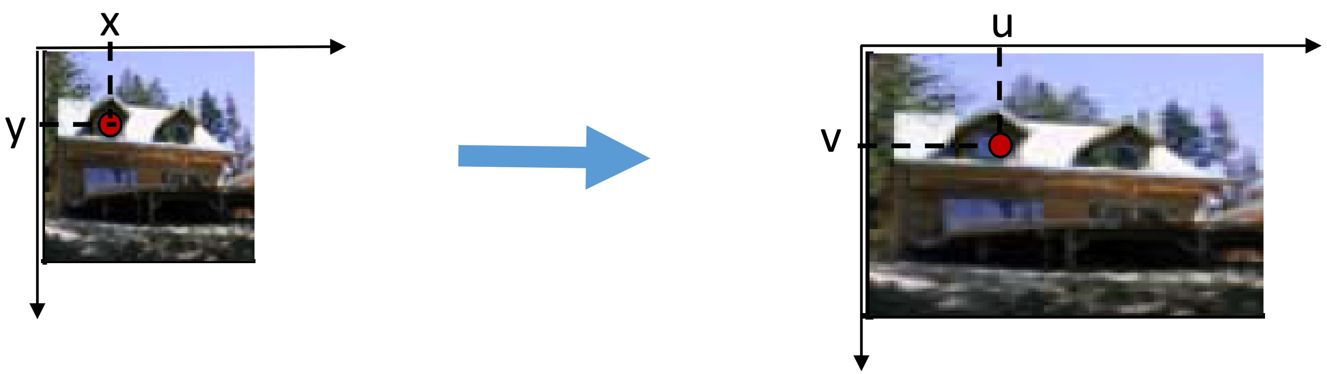
Overview

- Geometric Transformations
- Noise
  - Gaussian
  - Impulsive (Salt & Pepper)
- Filtering
  - Linear Filtering
  - Nonlinear Filtering

Geometric Transformation

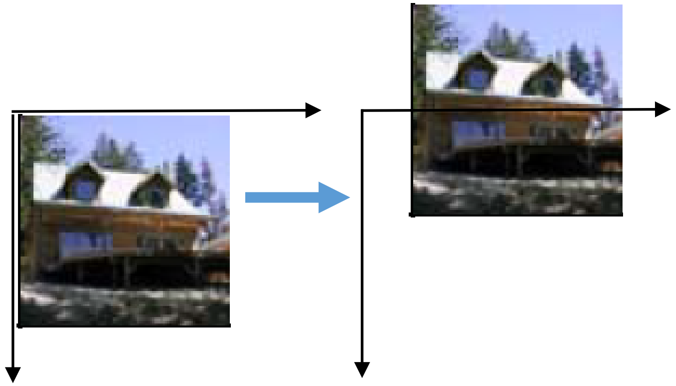
2D Transformations

- A pixel in the source image at location (x,y) is mapped to location (u,v) in the destination image

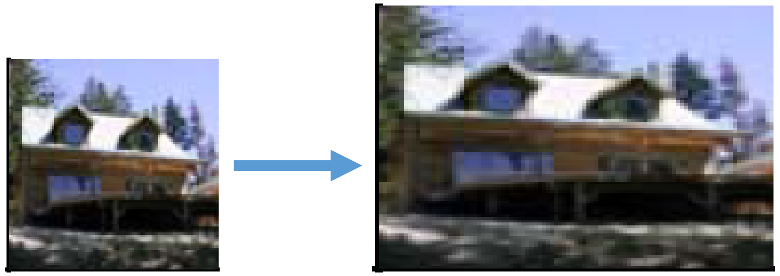


Types of 2D Transformation

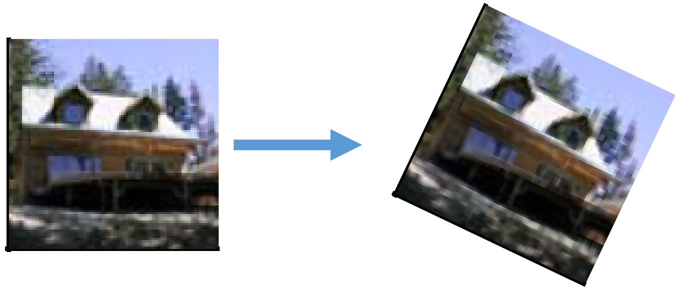
- Translation – pixels move in the same direction
  - $u = x + t_x$
  - $v = y + t_y$



- Scale or resize
  - $u = x * s_x$
  - $v = y * s_y$



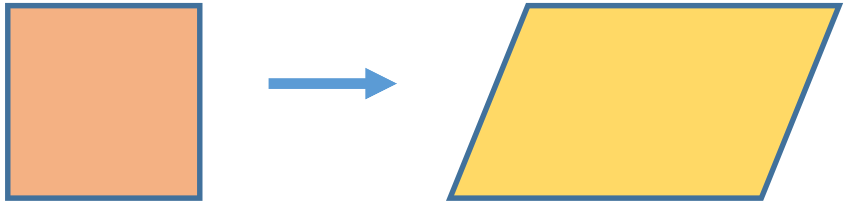
- Rotation
  - $u = x * \cos\theta - y * \sin\theta$
  - $v = y * \sin\theta + x * \cos\theta$



Types of 2D Transformation (cont.)

- Shear
  - $u = x + y * sh_x$
  - $v = y + x * sh_y$

  - If $sh_y = 0$



Matrix Notation for Affine Transformations

- ***Affine:***

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- Translation:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- Scale/ Resize:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

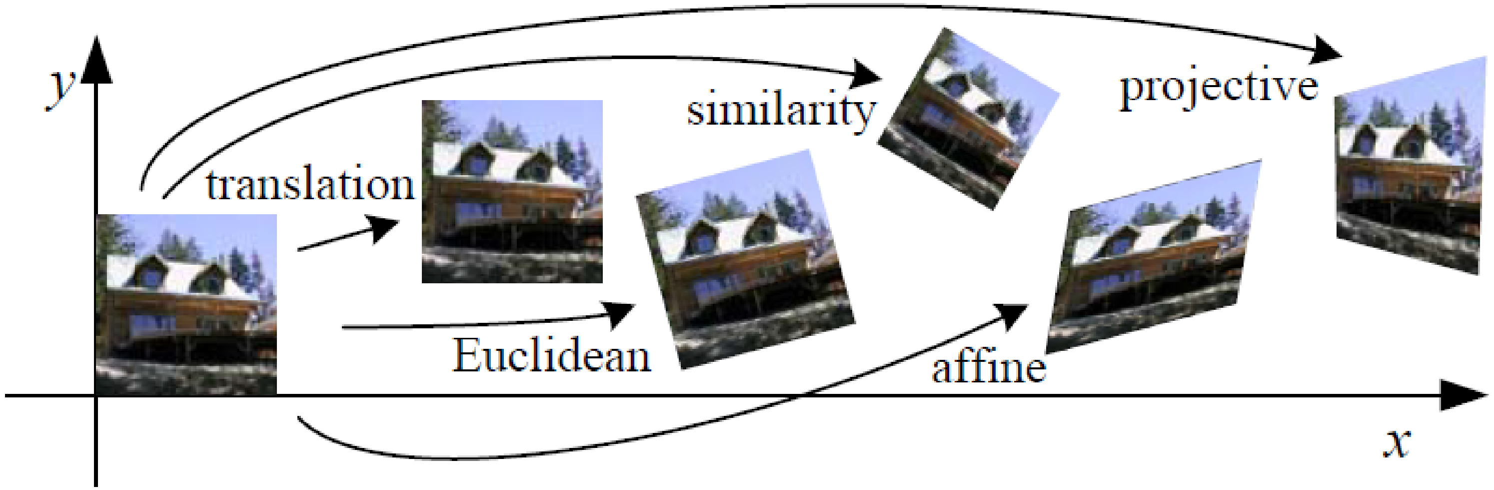
- Rotation:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- Shear:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2D Geometric Image Transformations



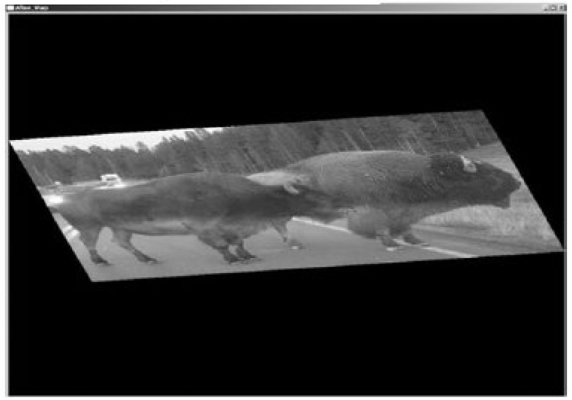
Transformation	Matrix	# DoF	Preserves	Icon
translation	$\begin{bmatrix} I \mid t \end{bmatrix}_{2\times 3}$	2	orientation	
rigid (Euclidean)	$\begin{bmatrix} R \mid t \end{bmatrix}_{2\times 3}$	3	lengths	
similarity	$\begin{bmatrix} sR \mid t \end{bmatrix}_{2\times 3}$	4	angles	
affine	$\begin{bmatrix} A \end{bmatrix}_{2\times 3}$	6	parallelism	
projective	$\begin{bmatrix} \tilde{H} \end{bmatrix}_{3\times 3}$	8	straight lines	

Original

Perspective

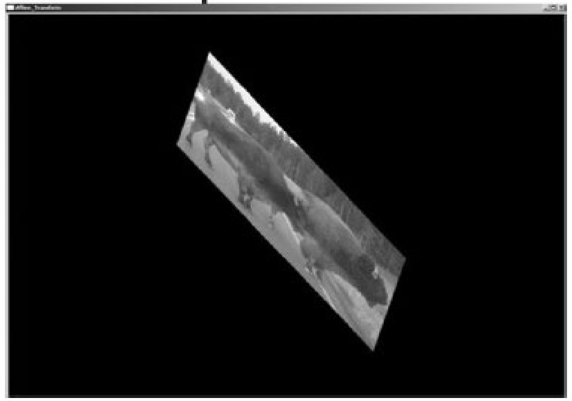
Rotation and scale

Affine warp



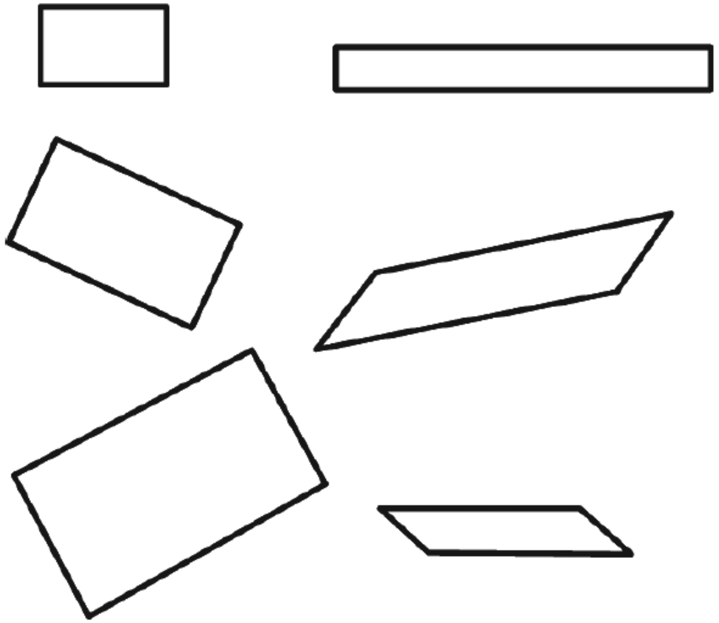
Affine scale

Rotation warp and scale



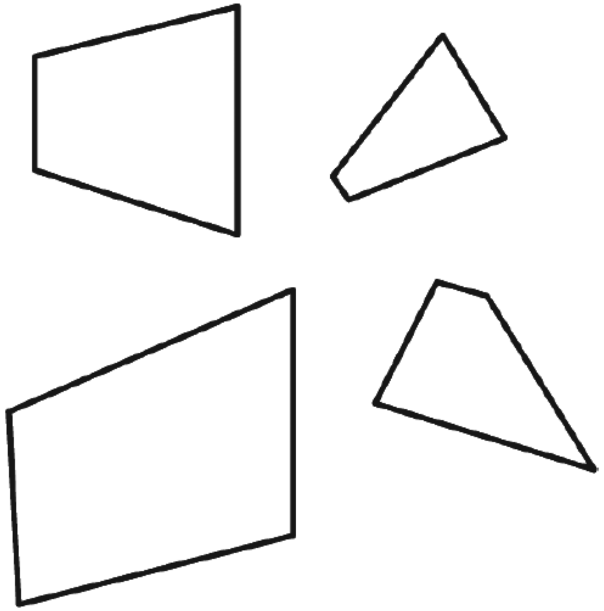
Affine (2x2)

Parallelograms



Perspective (3x3)
(or "Homography")

Trapezoids
(Includes all of Affine)



Affine Transform using OpenCV

- Given the 2x3 transform matrix M, find the result *dst*

```
void cv::warpAffine(
  cv::InputArray    src,                              // Input image
  cv::OutputArray   dst,                              // Result image
  cv::InputArray    M,                                // 2-by-3 transform mtx
  cv::Size          dsize,                            // Destination image size
  int               flags       = cv::INTER_LINEAR,   // Interpolation, inverse
  int               borderMode  = cv::BORDER_CONSTANT, // Pixel extrapolation
  const cv::Scalar& borderValue = cv::Scalar()        // For constant borders
);
```

Python:

cv.**warpAffine**(src, M, dsize[, dst[, flags[, borderMode[, borderValue]]]]) -> dst

Get the similarity transform matrix

```
cv::Mat cv::getRotationMatrix2D(                    // Return 2-by-3 matrix
  cv::Point2f  center                               // Center of rotation
  double       angle,                               // Angle of rotation
  double       scale                                // Rescale after rotation
);
```

Python:

cv.**getRotationMatrix2D**(center, angle, scale) -> retval

Rotate an image

```
height, width = img.shape[0:2]
angle = 30; scale = 1
rotationMatrix = cv.getRotationMatrix2D((width/2, height/2), angle, scale)
rotatedImage = cv.warpAffine(img, rotationMatrix, (width, height))
```

Find the transform

- Given the resulting image (or transformed coordinates of points), find the transformation matrix

```
Mat cv::getAffineTransform (InputArray src,
                            InputArray dst )
```

```
Python:
cv.getAffineTransform(src, dst) -> retval
```

Find the inverse transform

- Given the transform matrix, find the inverse

```
void cv::invertAffineTransform(
  cv::InputArray  M,                                  // Input 2-by-3 matrix
  cv::OutputArray iM                                  // Output also a 2-by-3 matrix
);
```

```
Python:
cv.invertAffineTransform(  M[, iM] ) -> iM
```

Perspective Transform

- Given the 3x3 transform matrix M, find the result *dst*

```
void cv::warpPerspective(
  cv::InputArray     src,                              // Input image
  cv::OutputArray    dst,                              // Result image
  cv::InputArray     M,                                // 3-by-3 transform mtx
  cv::Size           dsize,                            // Destination image size
  int                flags       = cv::INTER_LINEAR,   // Interpolation, inverse
  int                borderMode  = cv::BORDER_CONSTANT, // Extrapolation method
  const cv::Scalar& borderValue = cv::Scalar()        // For constant borders
);
```

```
cv.warpPerspective(src, M, dsize[, dst[, flags[,
   borderMode[, borderValue]]]]  ) -> dst
```

Find the perspective transform

- Given the resulting image (or transformed coordinates of points), find the transformation matrix

```
Mat cv::getPerspectiveTransform (InputArray   src,
                    InputArray   dst,
                    int  solveMethod= DECOMP_LU )
```

```
Python:
cv.getPerspectiveTransform(src, dst[, solveMethod]) ->retval
```

Noise in images

Noise [3]

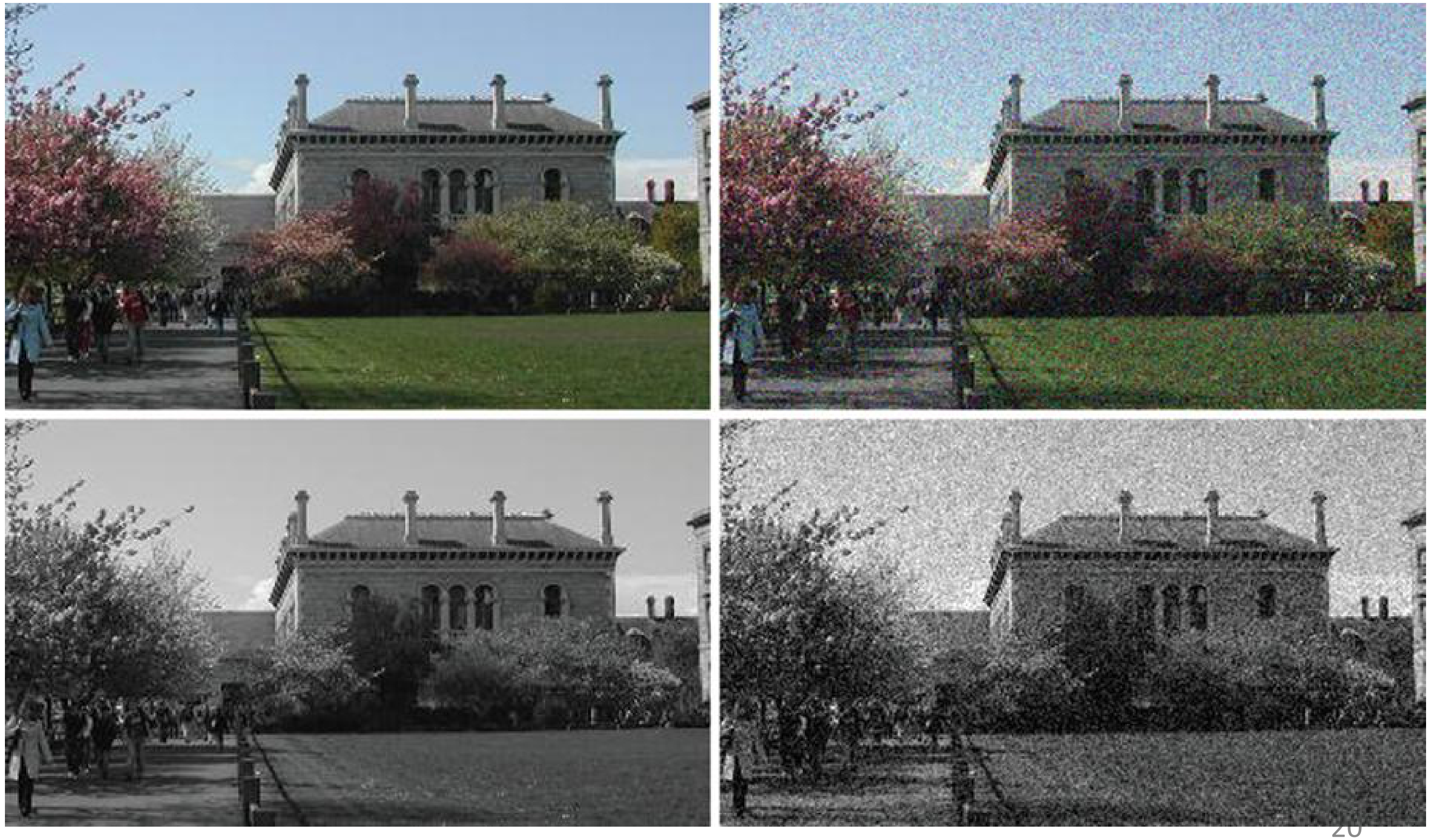
- Noise: anything that degrades the ideal image
- Sources of noise:
  - The environment,
  - The imaging device,
  - Electrical interference,
  - The digitization process, and so on.

- Noise is additive and random:

$$\hat{I}(i,j) = I(i,j) + n(i,j)$$

Gaussian Noise [3]

- A good approximation of real noise
- Modelled as a Gaussian (normal distribution with mean of 0)
- $n \sim N(\mu = 0, \sigma)$



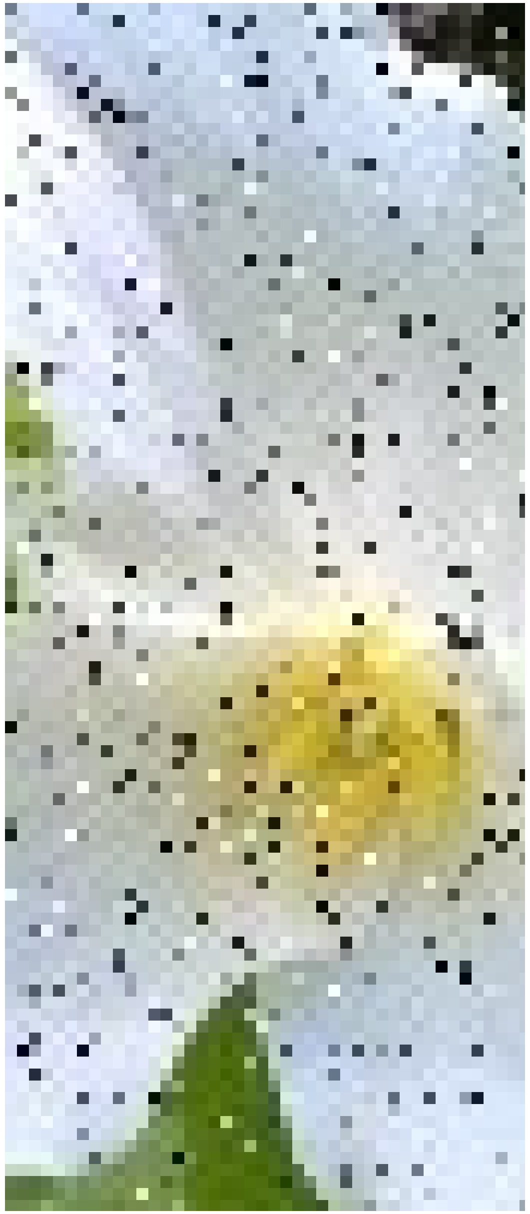
Impulsive noise- Salt and Pepper Noise

- Impulsive: noise peaks or spikes
- Salt & Pepper is a model often used for impulsive noise
- Random values of brightness (darker or lighter)
- At random pixels the image

Examples

p = 0.1

p = 0.5



Correcting noise

- Observation: The image does not change sharply most of the time (low frequency), while noise is a sharp peak (high frequency)
- Therefore using the values of the neighbors, we can often lower the noise
- Take the average of the neighboring pixels
  - This is equivalent to low-pass filtering
- Disadvantage: This will reduce the sharpness of edges in the image

Averaging

- The value at pixel (i, j) is calculated as the average of the pixels in its neighborhood
- Suitable for removing random noise, or smoothing

j →

i ↓

62	79	23	119	120	105	4	0
10	10	9	62	12	78	34	0
10	58	197	46	46	0	0	48
176	135	5	188	191	68	0	49
2	1	1	29	26	37	0	77
0	89	144	147	187	102	62	208
255	252	0	166	123	62	0	31
166	63	127	17	1	0	99	30

I_{in}

				?			

I_{out}

5x5 neighborhood

$$\text{new value}=\frac{9+62+\cdots+102+62}{25}$$

Linear Filtering

Linear filtering

- **Filtering**: an algorithm that starts with some image $I_{in}(i, j)$ and computes a new image $I_{out}(i, j)$ using a neighborhood operator
- **Kernel**: A template defining the neighborhood and the operator
- **Linear filter / linear kernel**: Values are calculated as a weighted sum of values in the neighborhood

$$I_{\text{out}}(i,j) = \sum_{x,y \in \text{Kernel}} k(x,y) \cdot I_{\text{in}}(i+x, j+y)$$

Averaging- box kernel

- Averaging is equivalent to convolution with a box kernel

j →, i ↓

62	79	23	119	120	105	4	0
10	10	9	62	12	78	34	0
10	58	197	46	46	0	0	48
176	135	5	188	191	68	0	49
2	1	1	29	26	37	0	77
0	89	144	147	187	102	62	208
255	252	0	166	123	62	0	31
166	63	127	17	1	0	99	30

$$k = 1/25 \times \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & \underline{\mathbf{1}} & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

5x5 (normalized) box kernel

- $I_{out} = I_{in} * k$

Convolution (*) is a mathematical operation

$$k = 1/9 \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & \underline{\mathbf{1}} & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

3x3 (normalized) box kernel

5x5 (normalized) box kernel

```
// Using this function
blurred = cv.blur(noisy, (5, 5))

// Or use this function
boxed= cv.boxFilter(noisy, -1, (5,5)); #-1: use src depth

// Or build a box kernel yourself and then filter
myKernel = np.ones([5, 5]) / 25.0;
filtered = cv.filter2D(noisy, -1, myKernel)
```

Examples

Salt & pepper noise with p = 0.1

After 5x5 box filter

Separable filtering

Some filters are separable into smaller filters. Applying smaller filters is faster (faster implementation).

For example:
Convolving with

$\frac{1}{25}$ x

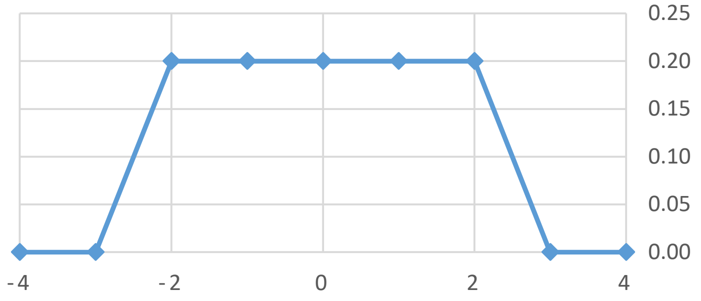
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

5x5 (normalized) box kernel

Is equivalent to
convolving with

1/5

1	1	1	1	1



A low-pass filter

And then
convolving with

1/5

1
1
1
1
1

Separable filtering

- **2D filter**

```
myKernel = np.ones([5, 5]) / 25.0;
filtered = cv.filter2D(noisy, -1, myKernel)
```

- **1D filter**

```
myKernel = np.ones(5)/ 5;
filtered = cv.sepfilter2D(noisy,-1, myKernel, myKernel)
```

Gaussian Filter (smoothing)

- The Gaussian Filter (2-D bell curve) is separable
- It can be applied by first convolving with a 1D Gaussian Filter horizontally and then vertically
- The 1-D kernel array can be obtained by:

```
cv.getGaussianKernel(
        ksize,              # kernel size
        sigma               # Gaussian half-width) → retval
```

- It can be applied using sepfilter2D (instead of filter2D)

Examples of Gaussian Filters

- `sigma = 2.0`
- `myKernel = cv.getGaussianKernel(5,sigma)`
- `filtered = cv.sepFilter2D(noisy, -1, myKernel, myKernel)`

- Values of above filter are: `[0.152, 0.222, 0.251, 0.222, 0.152]`
- If sigma = 1.0, kernel values: `[0.054, 0.244, 0.403, 0.244, 0.054]`
- Recall 1D averaging filter: `[0.200, 0.200, 0.200, 0.200, 0.200]`

Salt & pepper noise with p = 0.1

After 5x5 box filter

Gaussian filter with sigma = 2.0

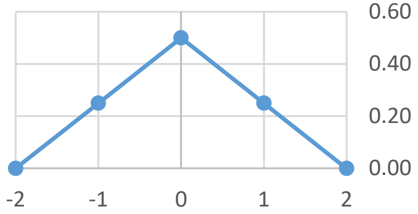
Gaussian filter with sigma = 1.0

Bilinear kernel

- Also smoothing (removing noise)
- Equivalent to convolving with two separable ‘tent’ functions
- Example: 3x3 bilinear kernel:

1-D Tent kernel:

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



2-D Bilinear kernel:

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

Examples

p = 0.1

After 3x3 bilinear filter

Nonlinear Filtering

Nonlinear filter

- The output pixel value is not a linear function of pixel values in the input
- Example: Median Filter
- The output value is the median of the pixels in the neighborhood

```
cv.medianBlur (
                src,       # Input image
                ksize )    # kernel size
)
              → dst        # Output image
```

Examples

p = 0.1

```
medBlur = cv.medianBlur(noisy, 5)
```

Overview

- Geometric Transformation transforms the location of pixels (not their intensity/ color values). In affine transformations, parallelism is preserved. Although orientations, lengths, angles and parallelism may all change by projective transformations, straight lines will still be straight lines.
- Noise refers to anything that degrades the ideal image. Two mathematical models for noise are the Gaussian noise model and the Impulsive (or Salt & Pepper) noise model.
- Filtering is used for removing noise. With a linear filter, the output pixel value is a linear function of pixel values in the input(noisy) image. Common kernels are: box, Gaussian, and bilinear kernels. The median filter is a nonlinear filter that can remove noise, without blurring the image.

References

[1] Computer Vision: Algorithms and Applications, R. Szeliski (http://szeliski.org/Book)

[2] Learning OpenCV 3, A. Kaehler & G. Bradski

- Available online through Safari Books, Seneca libraries
- https://senecacollege-primo.hosted.exlibrisgroup.com/primo-explore/fulldisplay?docid=01SENC_ALMA5153244920003226&context=L&vid=01SENC&search_scope=default_scope&tab=default_tab&lang=en_US

[3] Practical introduction to Computer Vision with OpenCV, Kenneth Dawson-Howe

- Available through Seneca libraries
- https://senecacollege-primo.hosted.exlibrisgroup.com/primo-explore/fulldisplay?docid=01SENC_ALMA5142810950003226&context=L&vid=01SENC&search_scope=default_scope&tab=default_tab&lang=en_US