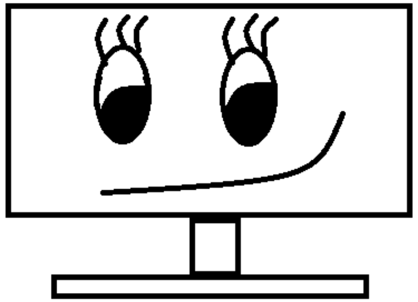
Seneca



Computer Vision

Geometric Transformations, Noise & Filtering

Seneca Polytechnic

Credit: Vida Movahedi

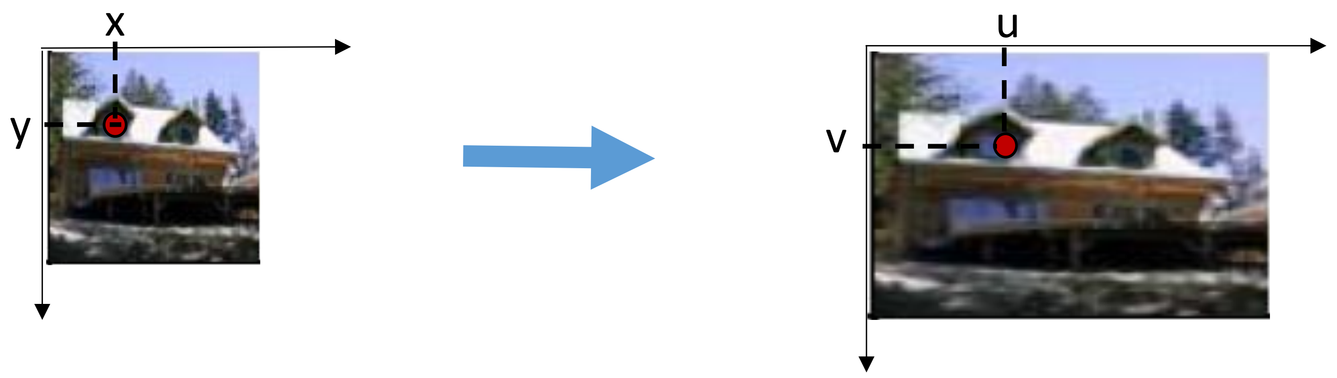
Overview

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

Geometric Transformation

2D Transformations

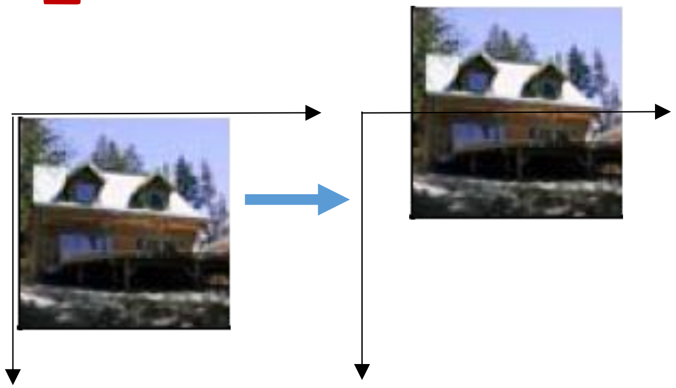
A pixel in the source image at **location (x,y)** is mapped to **location (u,v)** in the destination image



- Application: image matching and stitching, image registration & alignment, object detection, …

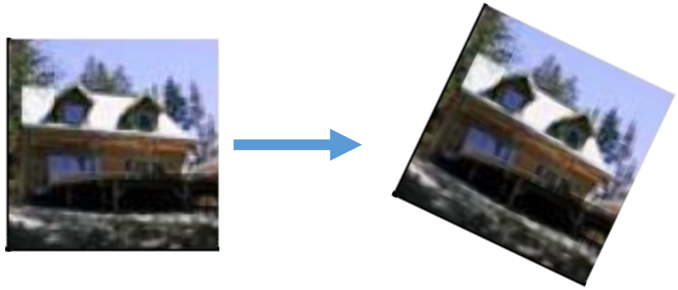
Types of 2D Transformation [1]

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



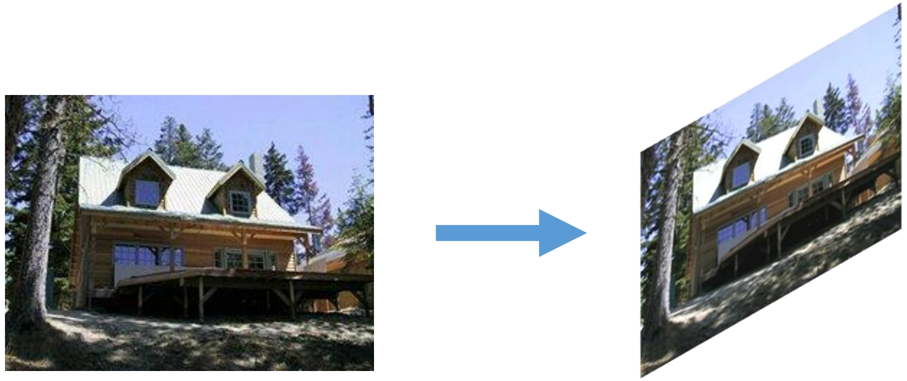
- **Scale or Resize**
  - $u = x * s_x$
  - $v = y * s_y$

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

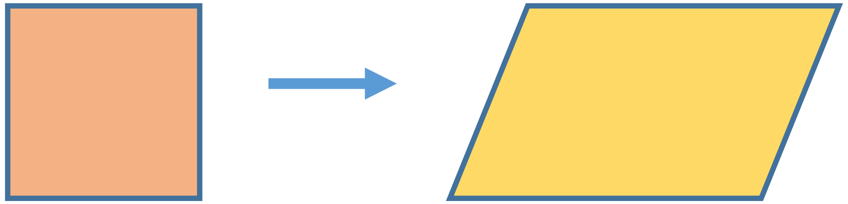


Types of 2D Transformation [1] (cont.)

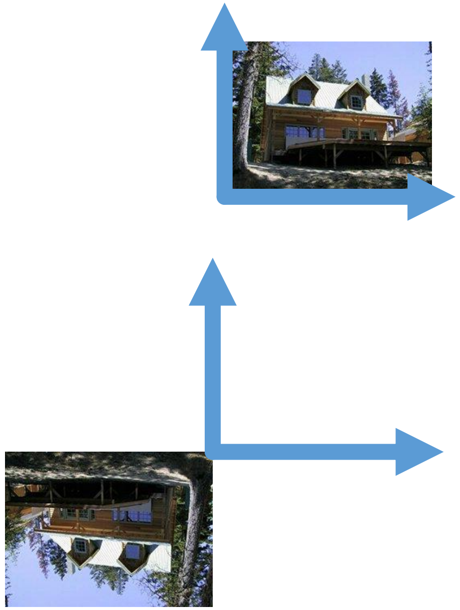
- **Shear**
  - $u = x + y * sh_x$
  - $v = y + x * sh_y$



  - If $sh_y = 0$



- **Mirror**
  - Mirror about x-axis and y-axis
  - $u = - x$
  - $v = - y$



Common Parametric Transformation [1]

Translation

Rotation

Aspect

Affine

Perspective

Parametric Transformations [1]

- **Parametric** or **global** transformation T applies a global deformation to an image, where the behavior of the transformation is controlled by a small number of **parameters**.

- Transformation T changes image coordinates:
  - T is *global* as it is the same for any point p
  - T is *parametric* as it can be described by a few parameters (numbers)
  - T does **not** depend on image content
  - Changing the *domain* of the image

Properties of Affine Transformations

- An affine transformation, is a **geometric transformation** that preserves **points**, **straight lines** and **plains**, as well as **parallelism** (but **not** necessarily **distances** and **angles**).
  - Origin does not necessarily map to origin
  - Lines map to lines
  - Parallel lines remain parallel
  - Ratios are preserved (size is not)
  - Closed under composition
- We can combine transformations via matrix multiplication

Matrix Notation for Affine Transformations

- A transformation that can be expressed in the form of a matrix multiplication (**linear transformation**) followed by a vector addition (**translation**) U = I. X + T

- The usual way to represent an **Affine Transformation** is by using a 2X3 matrix

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

Matrix Notation for Affine Transformations

- Translation (vector addition)

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

- Scale/Resize (linear transformation)

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

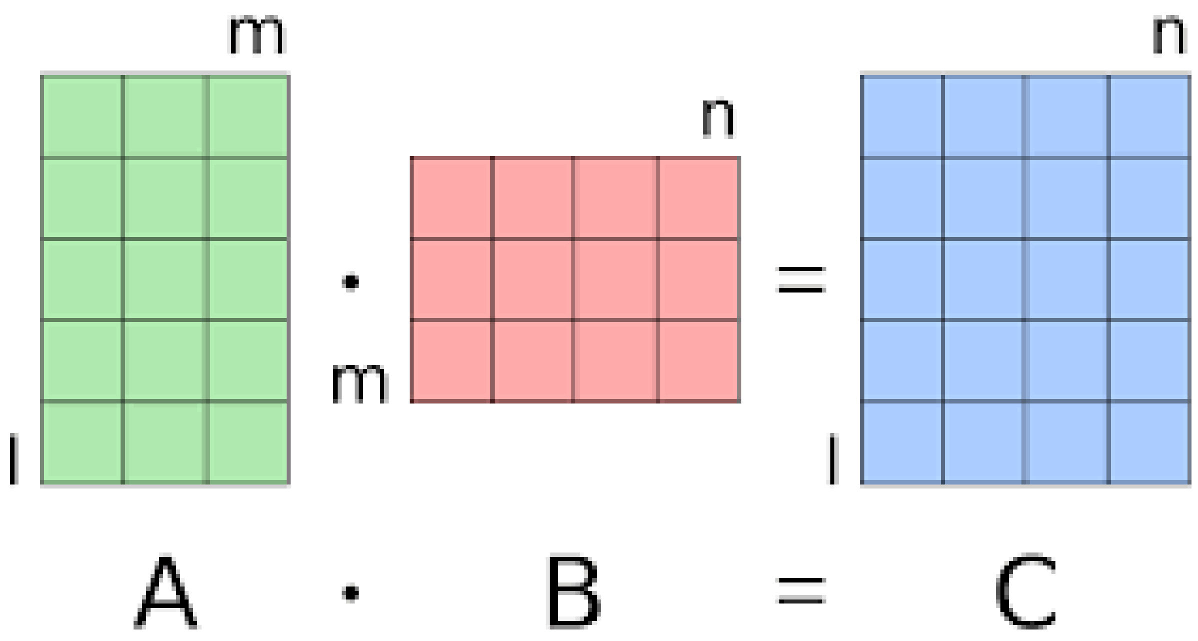
- Rotation (linear transformation)

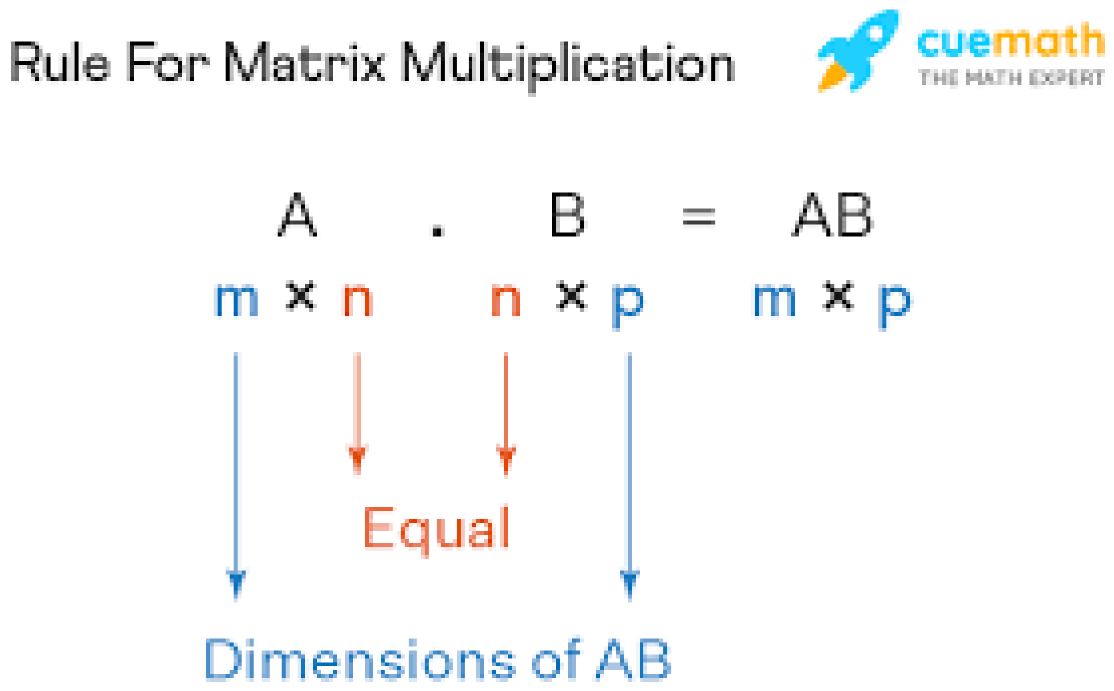
$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- Shear (linear transformation)

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

- Identity and Reflection (linear transformation)


m
n
n
m
l
l
A
•
B
=
C


Rule For Matrix Multiplication
cuemath
THE MATH EXPERT
A . B = AB
m × n n × p m × p
Equal
Dimensions of AB

[A Complete Beginners Guide to Matrix Multiplication for Data Science with Python Numpy | by GreekDataGuy | Towards Data Science]

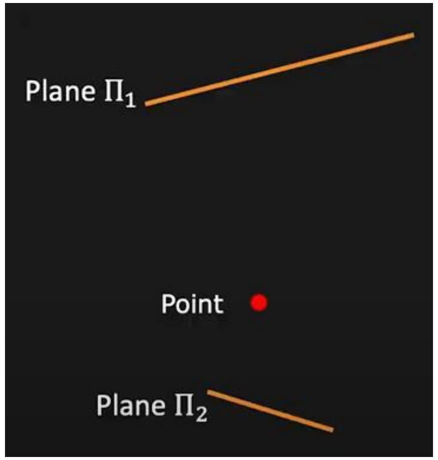
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \text{x} \begin{bmatrix} 10 & 11 \\ 20 & 21 \\ 30 & 31 \end{bmatrix}$$

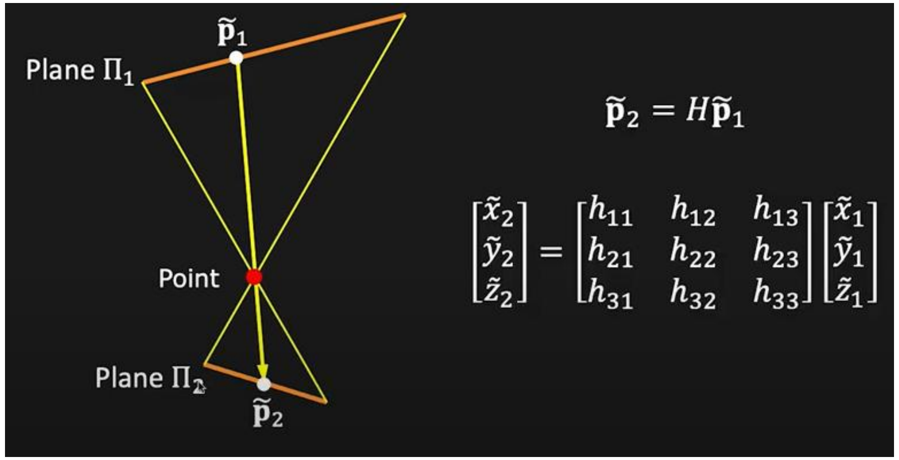
$$= \begin{bmatrix} 1\text{x}10 + 2\text{x}20 + 3\text{x}30 & 1\text{x}11 + 2\text{x}21 + 3\text{x}31 \\ 4\text{x}10 + 5\text{x}20 + 6\text{x}30 & 4\text{x}11 + 5\text{x}21 + 6\text{x}31 \end{bmatrix}$$

$$= \begin{bmatrix} 10+40+90 & 11+42+93 \\ 40+100+180 & 44+105+186 \end{bmatrix} = \begin{bmatrix} 140 & 146 \\ 320 & 335 \end{bmatrix}$$

Projective/Homography Transformation

- Properties of projective transformations:
  - Origin does not necessarily map to origin
  - Lines map to lines
  - Parallel lines do not necessarily remain parallel
  - Ratios are not preserved
  - Closed under composition
- A projective matrix maps one plane to another plane through a point

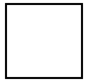
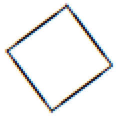
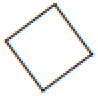
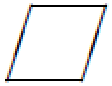
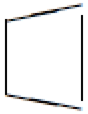




[3x3 Image Transformations | Image Stitching]

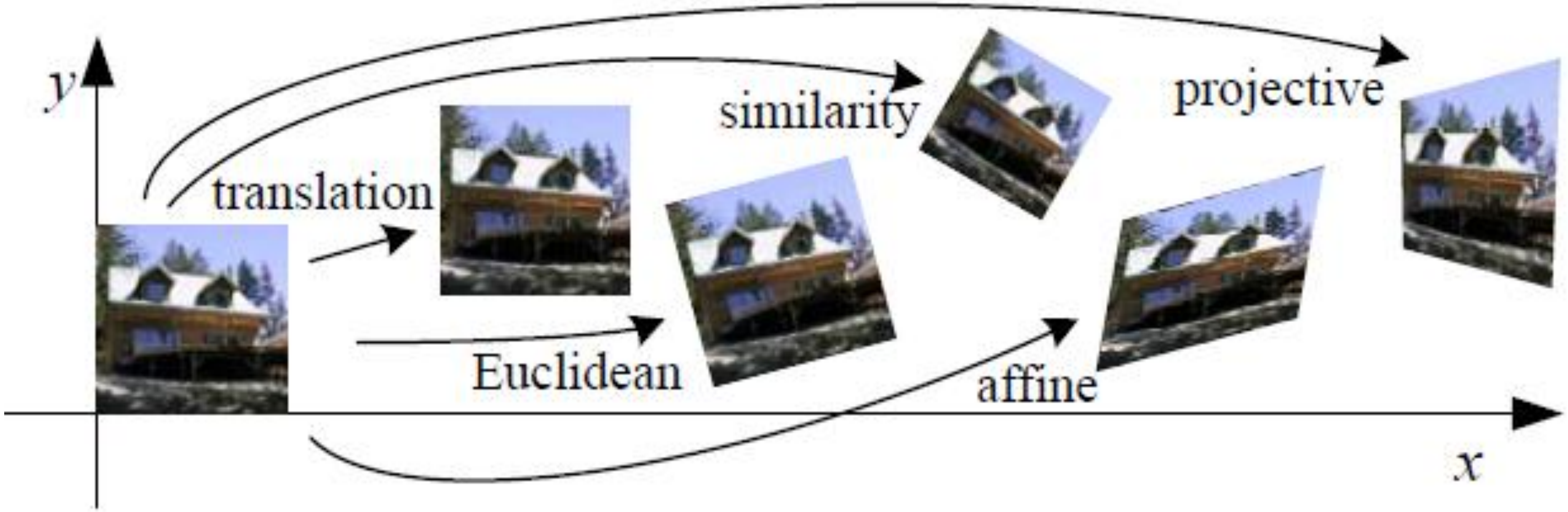
Parametric Transformations [1]

- Rotation + Translation (2D rigid body motion or 2D Euclidean transformation): $x' = Rx + t$
- Scaled Rotation or Similarity Transform: $x' = sRx + t$, where s is an arbitrary scale factor

Transformation	Matrix	# DoF	Preserves	Icon
translation	$\begin{bmatrix} I & t \end{bmatrix}_{2\times 3}$	2	orientation	
rigid (Euclidean)	$\begin{bmatrix} R & t \end{bmatrix}_{2\times 3}$	3	lengths	
similarity	$\begin{bmatrix} sR & t \end{bmatrix}_{2\times 3}$	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	

Parametric Transformations [1]

- Basic set of 2D geometric image transformations



Original

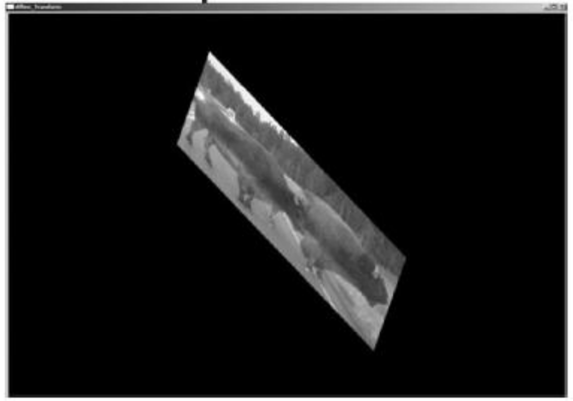
Perspective

Rotation and scale

Affine warp

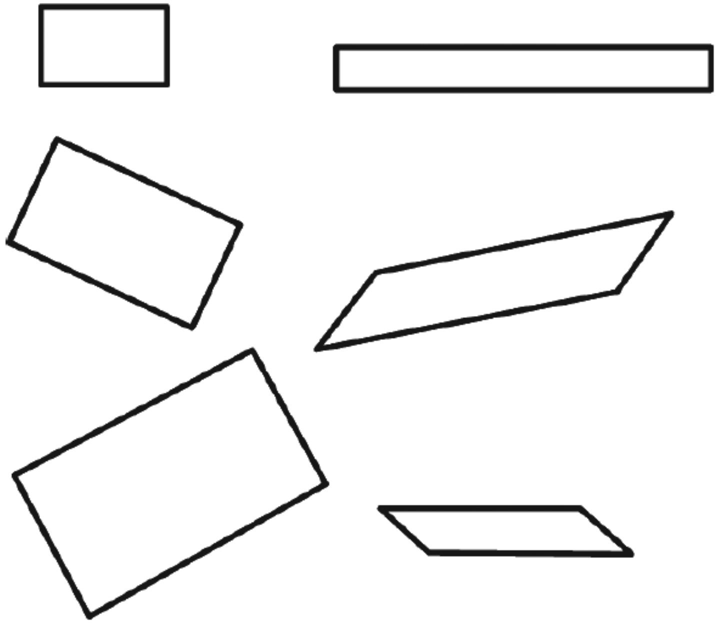
Affine scale

Rotation warp and scale



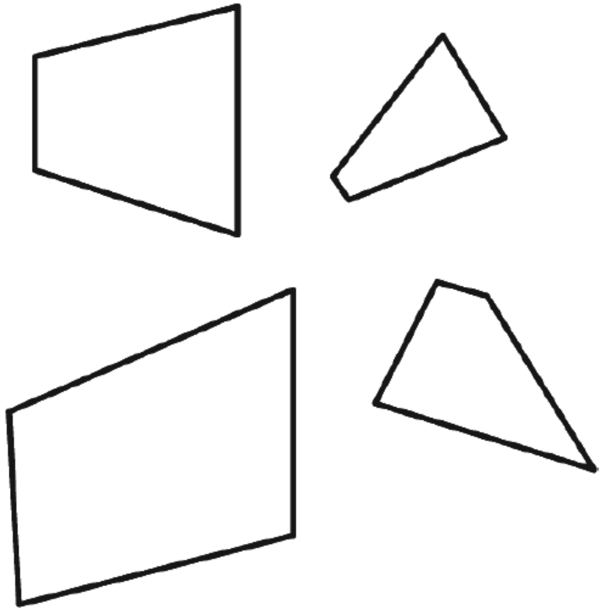
Affine (2x2)

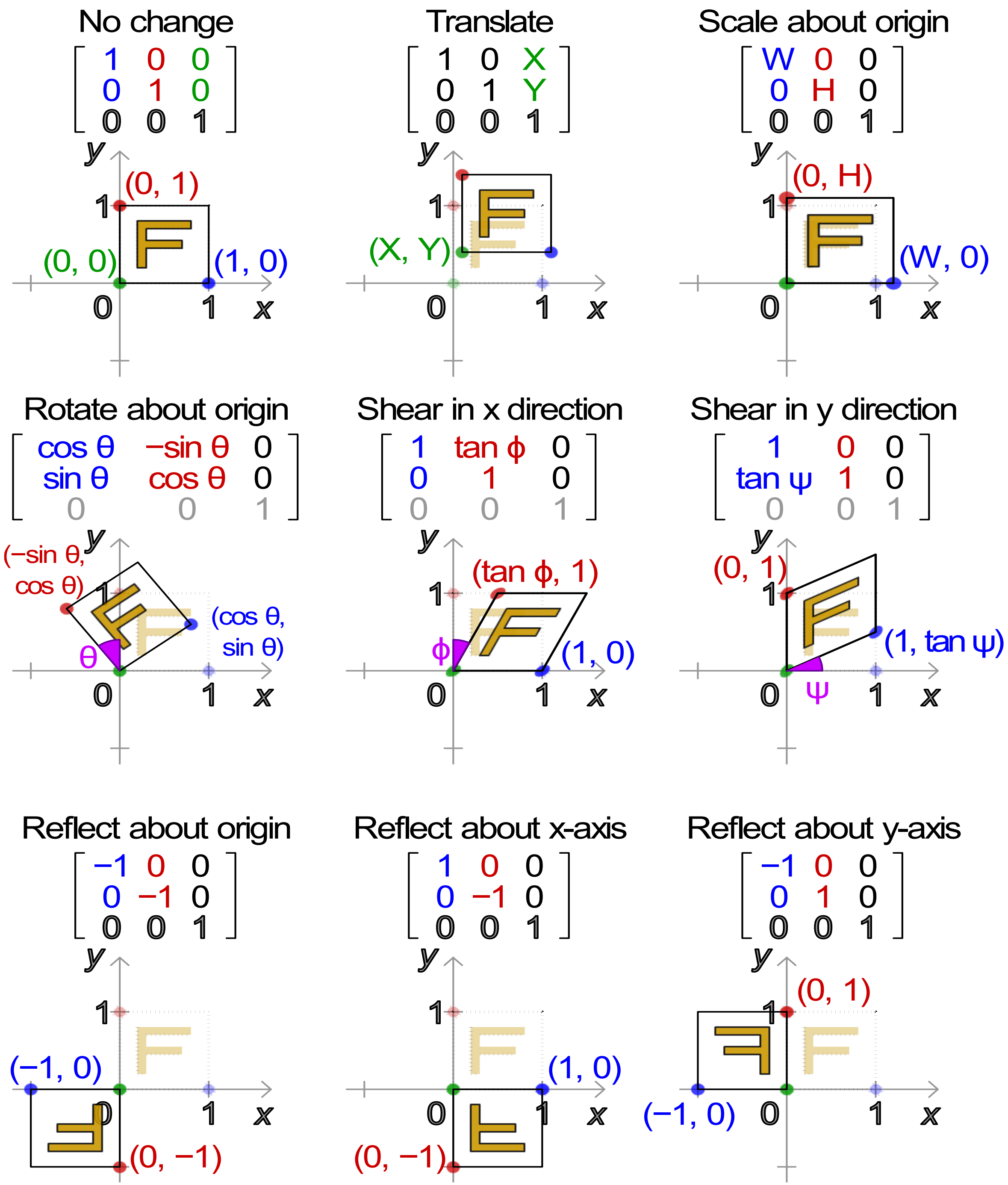
Parallelograms



Perspective (3x3)
(or "Homography")

Trapezoids
(Includes all of Affine)




No change
1 0 0
0 1 0
0 0 1
(0, 1)
(0, 0)
(1, 0)
Translate
1 0 X
0 1 Y
0 0 1
(X, Y)
Scale about origin
W 0 0
0 H 0
0 0 1
(0, H)
(W, 0)
Rotate about origin
cos θ −sin θ 0
sin θ cos θ 0
0 0 1
(−sin θ, cos θ)
(cos θ, sin θ)
θ
Shear in x direction
1 tan ϕ 0
0 1 0
0 0 1
(tan ϕ, 1)
(1, 0)
ϕ
Shear in y direction
1 0 0
tan ψ 1 0
0 0 1
(0, 1)
(1, tan ψ)
ψ
Reflect about origin
−1 0 0
0 −1 0
0 0 1
(−1, 0)
(0, −1)
Reflect about x-axis
1 0 0
0 −1 0
0 0 1
(1, 0)
(0, −1)
Reflect about y-axis
−1 0 0
0 1 0
0 0 1
(0, 1)
(−1, 0)
x
y

Get an Affine Transformation

Affine transformation represents a **relation between two images** such that T = MX. The information about this relation can come, roughly, in **two** ways:

- If M (the relation) is known (i.e. we have the 2-by-3 matrix), then we can easily find T.
- If we know both X and T and we also know that they are related, we can find M

OpenCV: Affine Transformations

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]

- Images are normally affected by noise.
- Noise: anything that **degrades** the ideal image to some degrees
- Sources of noise:
  - The environment,
  - The imaging device,
  - Electrical interference,
  - The digitization process, and so on.

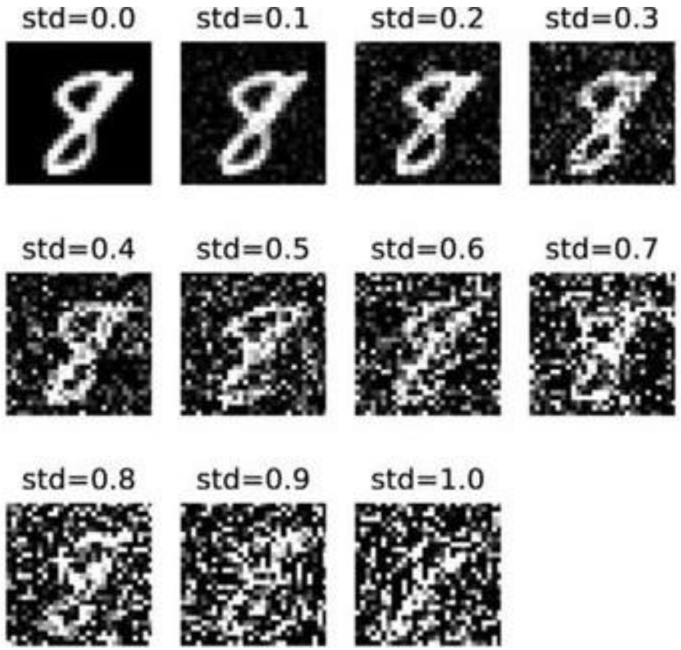
- Noise can be **additive** and **random**:

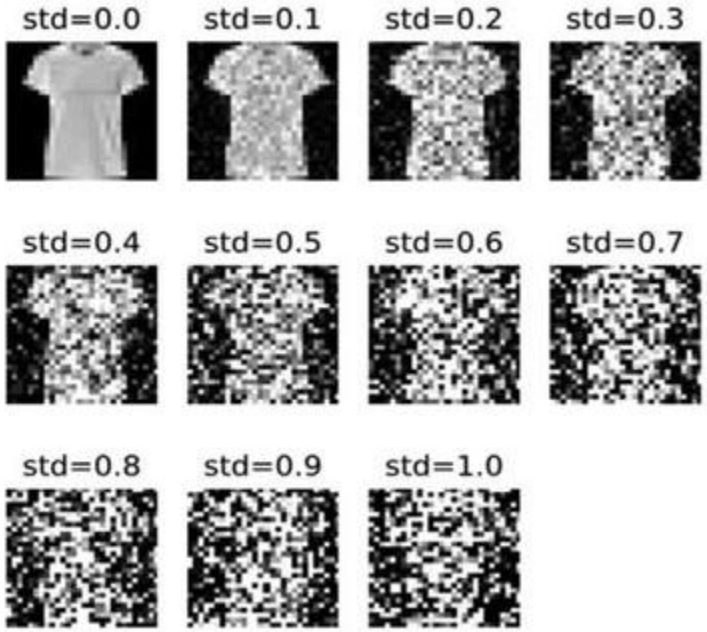
$$\hat{I}(i,j) = I(i,j) + n(i,j)$$

Gaussian Noise

- Gaussian Noise is a good approximation of real noise
- Modelled as a Gaussian (normal distribution with mean of 0)

$$n \sim N(\mu = 0, \sigma)$$



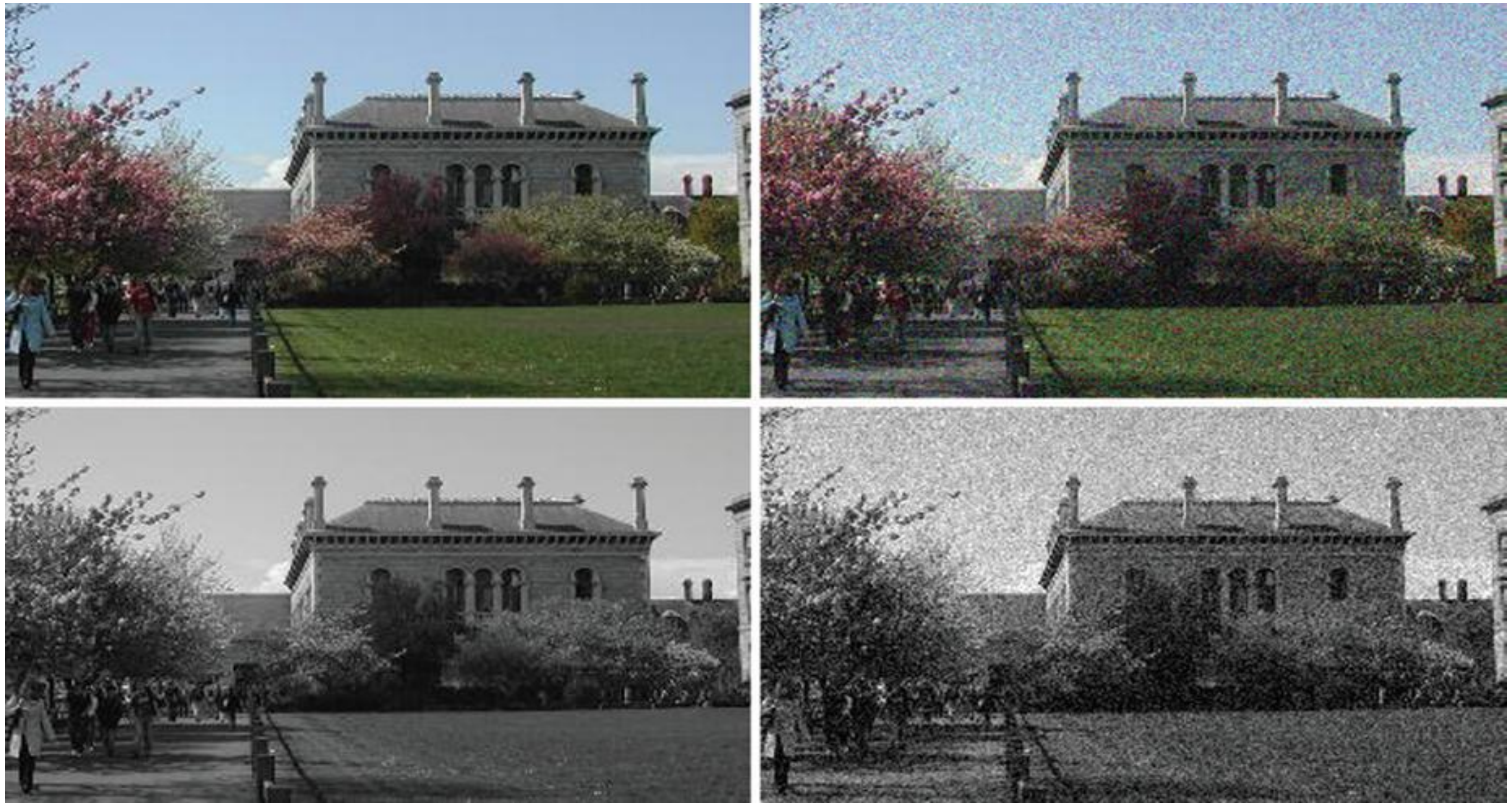


[Gaussian noise image classification (A) Additive white Gaussian noise]

Gaussian Noise [3]

- Gaussian Noise is a good approximation of real noise
- Modelled as a Gaussian (normal distribution with mean of 0)

$$n \sim N(\mu = 0, \sigma)$$



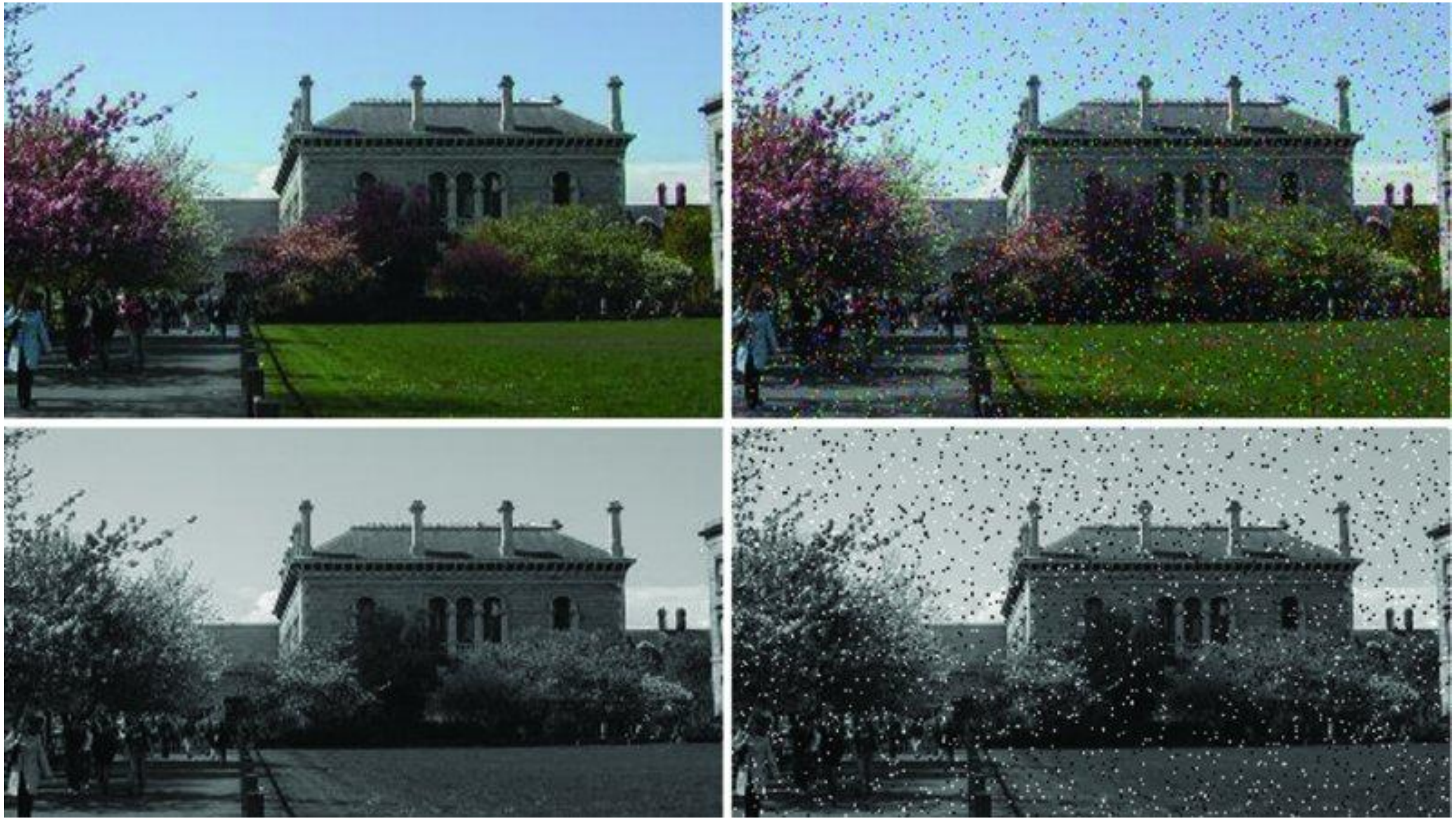
Color and grey-scale images (left) with Gaussian noise added with a mean of 0 and a standard deviation of 20 (right).

Impulsive Noise - Salt and Pepper Noise

- Impulse noise is corruption with individual noisy pixels whose **brightness** differs significantly from that of the **neighborhood**.

- Random values of brightness (darker or lighter) at random pixels the of the image

- Salt & Pepper noise is **a type of impulse noise** where saturated impulse noise affects the image (i.e. it is corrupted with **pure white and black** pixels).



Colour and grey-scale images (left) with 10% Salt and pepper noise (right).

Examples

p = 0.1

p = 0.5

Linear & Non-Linear Filtering

Noise Removal

- Given a camera and a still scene, how can you reduce noise? → Take lots of images and average them!

Noise Removal

- 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 (blurring of sharp edges)

Point vs Neighborhood Operators (recap)

- **Point Operators:**

  The value of each pixel in the output depends only on the value of the **same pixel** in the input (and possibly **some global information** or some parameters)

  Example: brightness adjustment

- **Neighborhood Operators:**

  The value of each pixel in the output depends on the value of the **pixel** and the value of **its neighbors** in the input

  Example: Smoothing or blurring

Averaging

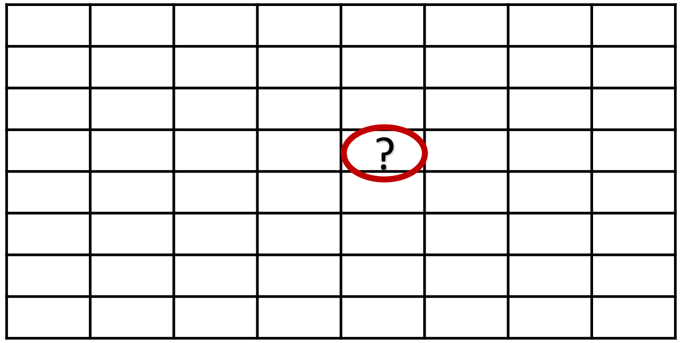
- The value at pixel (i, j) is calculated as the average of the pixels in its neighborhood
- Suitable for removing **random noise**, or **smoothing**

$j \rightarrow$, $i \downarrow$

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}

5x5 neighborhood



I_{out}

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

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**

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

Linear Filtering

- Neighborhood filtering (convolution): The image on the left is convolved with the filter in the middle to yield the image on the right.

45	60	98	127	132	133	137	133
46	65	98	123	126	128	131	133
47	65	96	115	119	123	135	137
47	63	91	107	113	122	138	134
50	59	80	97	110	123	133	134
49	53	68	83	97	113	128	133
50	50	58	70	84	102	116	126
50	50	52	58	69	86	101	120

$f(x,y)$

*

0.1	0.1	0.1
0.1	0.2	0.1
0.1	0.1	0.1

$h(x,y)$

=

69	95	116	125	129	132
68	92	110	120	126	132
66	86	104	114	124	132
62	78	94	108	120	129
57	69	83	98	112	124
53	60	71	85	100	114

$g(x,y)$

Averaging - Box Kernel

- Averaging is equivalent to **convolution** with a box kernel and each point is equally weighted

$j \rightarrow$, $i \downarrow$

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

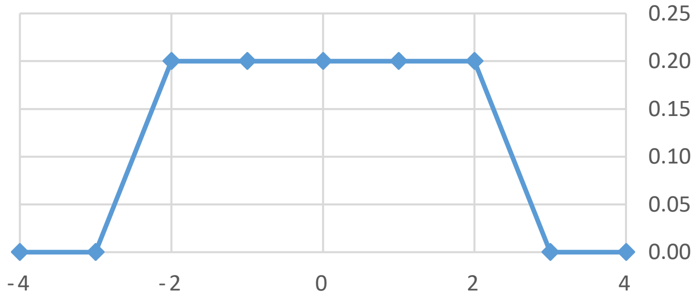
1	1	1	1	1
1	1	1	1	1
1	1	<u>**1**</u>	1	1
1	1	1	1	1
1	1	1	1	1

5x5 (normalized) box kernel

Is equivalent to
convolving with

1/5

1	1	<u>1</u>	1	1



A low-pass filter

And then
convolving with

1/5

1
1
<u>1</u>
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

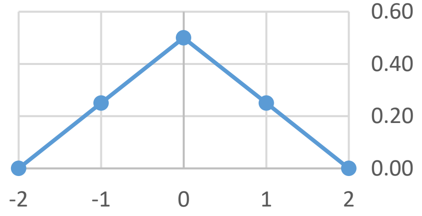
Gaussian filter with sigma = 1.0

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 (Good at dealing with **noise**, damages thin lines and corners)
- 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** by R. Szeliski

Computer Vision: Algorithms and Applications, 2nd ed. (szeliski.org)

[2] **Learning OpenCV 3** by A. Kaehler & G. Bradski
Available online via Seneca Libraries: Learning OpenCV 3 : computer Vision in C++ with the OpenCV Library - Seneca (exlibrisgroup.com)

[3] **A Practical Introduction to Computer Vision with OpenCV** by Kenneth Dawson-Howe

Available online via Seneca Libraries: A Practical Introduction to Computer Vision with OpenCV. - Seneca (exlibrisgroup.com)

Readings

Chapter 2.4, 2.5, 5.2 [1]
Chapter 10 – 11 [3]

- [1] **A Practical Introduction to Computer Vision with OpenCV**
  by Kenneth Dawson-Howe
  Available online via Seneca Libraries: A Practical Introduction to Computer Vision with OpenCV. - Seneca (exlibrisgroup.com)

- [2] **Learning OpenCV 4 Computer Vision with Python 3**
  by J. Howse & J. Minichino
  Available online via Seneca Libraries: Learning OpenCV 4 Computer Vision with Python 3 : get to grips with tools, techniques, and algorithms for computer vision and machine learning - Seneca (exlibrisgroup.com)

- [3] **Learning OpenCV 3**
  by A. Kaehler & G. Bradski
  Available online via Seneca Libraries: Learning OpenCV 3 : computer Vision in C++ with the OpenCV Library - Seneca (exlibrisgroup.com)