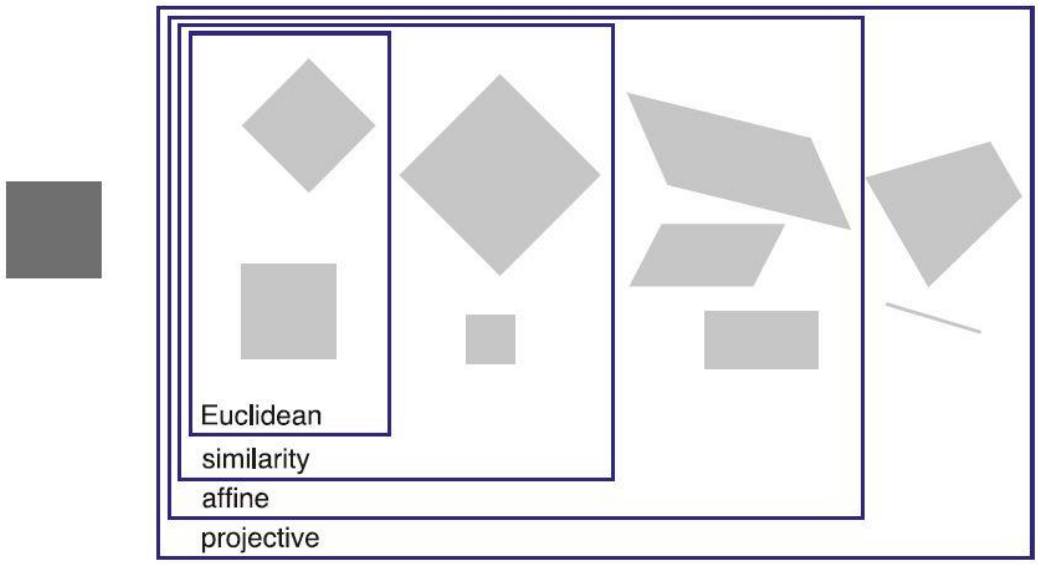
Geometric Transformations

By:
Ahmad Nadeem Saigol

Contents

Geometric Transformations

The geometric transformations are kind of transformations which transforms the geometry by changing its certain aspects while preserving others. In technical terms, it is a kind of function whose domain are set of points such that a function is injective so that its inverse exists. Overall, these transformations have three main groups: Euclidean, Affine and Projective transformations. These are interconnected with each other as shown below: Euclidean being most limited to Projective being most general.



Few points to note:

- The transformation is described for 2D. However, they can easily be extended to 3D by adding third axes (that is changing the size of matrix to compensate for the third axis)
- Transformations on the images have been implemented using OpenCV (4.1.2) in Python (3.7.10) in Google Colab and using Image Processing Toolbox (10.3) in MATLAB (R2018b).
- The image (in jpeg format) on which transformations is applied is:
- 
- The image is read in the environment and stored in a variable through '`img=cv2.read('path-to-image')`' (in Python) and '`img=imread('path-to-image')`' (in MATLAB)

- The dimension of the image is (600,900,3) which can be verified through '`img.shape`' (in Python) and '`size(img)`'(in MATLAB)
- To display the image, in Python, normally '`cv2.imshow(img)`' is used. However, In Google Colab, this causes the Jupyter Notebook to crash. As an alterative, '`cv2_imshow(img)`' can be used which can be imported from '`google.colab.patches`'. In MATLAB, the image is displayed using '`imshow(img)`'
- It may be noted that by default in Python and in MATLAB, the origin (coordinate system) is on top left corner (viewed by person looking at the screen).
- The images formed by Python and MATLAB were almost same. However, the graphics seemed a little bit different which can be due to the way it saves the image and also that image from Google Colab was in format 'jpg' and image from MATLAB was in format 'png'

Euclidean Transformations

When these transformations are applied, lengths and angle measures of the geometry do not change. This means that lines transform to lines, planes transform to planes, circles transform to circle and ellipsoids transform to ellipsoids. Only position and orientation of the object will change. It can either be translation, rotation, reflection, or their combination.

The general equation is given by:

$$\boldsymbol{x}' = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \end{bmatrix} \bar{\boldsymbol{x}}$$

$$\boldsymbol{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Translation:

It translates the geometry to a new position with respect to initial position. Let us say we have an augmented vector x, then

$$\boldsymbol{x}' = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{t} \end{bmatrix} \bar{\boldsymbol{x}}$$

Where I = identity matrix of size 2x2 and t = 2x1

It preserves the orientation of the object. This is useful when we want to hide a part of the image, crop an image, shift an image, or animate an image using image translation in loops.

Code:

Let us say we want to translate the image to a point (100,200,1)

Python:

We need to first define a transformation matrix (R=I, t=[100,200]) as follows:

```
T=np.float32([[1, 0, 100,], [0, 1, 200]])
```

After that we perform the transformation with:

```
trans_img=cv2.warpAffine(img, T, (width, height))
```

This function takes image, transformation matrix and tuple of width and height as argument.

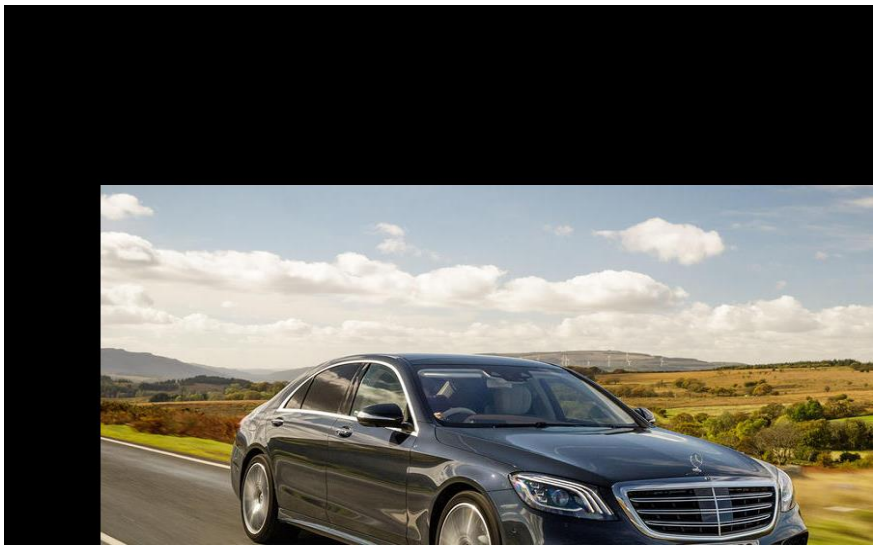
MATLAB:

The code for the transformation is:

```
trans_img = imtranslate(img, [100,200])
```

It takes the image and the point as arguments.

Result:



Rotation:

It rotates the geometry to a new position around a point. Mathematically it is given by,

$$x' = \begin{bmatrix} R & t \end{bmatrix} \bar{x}$$

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Where t =0.

It preserves the position of the object. It is commonly used to improve the visual appearance of an image, although it can be useful as a preprocessor in applications where directional operators are involved.

Code:

Let us say we want to rotate an image by 30 degree about the center of the image.

Python:

Just like translation we first need to define transformation matrix which is achieved by:

```
R=cv2.getRotationMatrix2D((width/2, height/2), 30, 1 )
```

This function takes center of rotation, angle in degree and scaling factor as argument and calculates the matrix using the formula:

$$\begin{bmatrix} \alpha & \beta & (1-\alpha)\cdot center.x - \beta \cdot center.y \\ -\beta & \alpha & \beta \cdot center.x + (1-\alpha)\cdot center.y \end{bmatrix}$$

where:

$$\alpha = scale \cdot \cos\theta,$$
$$\beta = scale \cdot \sin\theta$$

And then transformation is applied to the image:

```
rotate_img=cv2.warpAffine(img, R, (width, height))
```

It may be noted that the image gets cropped.

MATLAB:

The code for the transformation is:

```
rotated_img=imrotate(img,30)
```

It rotates the image by 30 degree around its center point. Moreover, it makes the output image large enough to contain the entire rotated image.

Result:

Python:

MATLAB:

Reflection:

It represents a flip of a figure. It maps every point of a figure to an image across a line of symmetry using a reflection matrix. Following table describes which matrix needs to be multiplied by the points of the geometry:

TYPE OF REFLECTION	Matrix to be multiplied
Reflection about the x- axis	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Reflection about the y-axis	$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$
Reflection about the line y = x	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Reflection about the line y = -x	$\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$
Reflection about the origin	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$

Code:

Let us say we want to flip an image horizontally:

Python:

Flipping can be achieved using the following function:

```
flip_image=cv2.flip(img, 1)
```

This function takes image and a scaler value as argument. The latter parameter dedicates how reflection will be carried out: 0 for vertical, 1 for horizontal and -1 for both.

Result:

Python:

Affine Transformations

This transformation preserves lines and parallelism but does not angels and distances, associated with geometry of the object. However, it does preserve ratios of distance between points lying on a straight line. It does not alter degree of the polynomial, parallel lines/lanes are transformed to parallel line/planes and intersecting lines /plane are transformed to intersecting lines/plane. It is typically used to correct for geometric distortions or deformations that occur with non-ideal camera angles.

The general equation in given by:

$$\boldsymbol{x}' = \boldsymbol{A}\bar{\boldsymbol{x}}$$

$$\boldsymbol{x}' = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix} \bar{\boldsymbol{x}}.$$

Similarity Transform:

It is equivalent to scaling the object size by factor's'. It is also known as Scaled Rotation. Mathematically it is given by,

$$\boldsymbol{x}' = \begin{bmatrix} s\boldsymbol{R} & \boldsymbol{t} \end{bmatrix} \bar{\boldsymbol{x}} = \begin{bmatrix} a & -b & t_x \\ b & a & t_y \end{bmatrix} \bar{\boldsymbol{x}},$$

Where s = arbitrary factor. If R=Identity matrix and t is zero vector, then it will only perform the scaling of the object.

It preserves angles of the geometry.

Code:

Let us consider two cases: scale down an image by half; and rotate an image by 30 degree around center point scaled down by half.

Python:

Case 1: It can be achieved using the command:

```
res = cv2.resize(img, (int(width / 2), int(height / 2)), interpolation
= cv2.INTER_CUBIC)
```

'`cv2.INTER_CUBIC`' is used for shrinking while '`cv2.INTER_CUBIC`' is used for zooming.

Case 2: This can be done using the same method described under "Rotation" by changing the scaling argument to 0.5.

```
S=cv2.getRotationMatrix2D((width/2, height/2), 30, 0.5 )
scale_img=cv2.warpAffine(img, S, (width, height))
```

It may be noted that in this case it does not crop the image.

MATLAB:

Case 1: The code for the transformation is:

```
rotated_img=imresize(img,0.5)
```

It takes image and scaling factor as arguments. This can be verified by checking the dimension of the image (300,450,3).

Result:

Case 1 (Python & MATLAB):

Case 2 (Python):

Shear:

Its effects are such that it "'pushes" the geometric object in a direction parallel to a coordinate axis (2D). The amount by which it pushes is determined by shear factor. Mathematically it is given by,

Horizontal Shear	$\begin{bmatrix} 1 & s_h & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x + s_v * y$ $y' = y$
Vertical Shear	$\begin{bmatrix} 1 & 0 & 0 \\ s_v & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x$ $y' = x * s_h + y$

Code:

Let us scale down the image by 25% and provide equal shear of 0.5 in both axes:

Python:

It can be achieved by defining the transformation matrix and then passing it to warpAffine() function as follows:

```
Sh=np.float32([[0.75, 0.5, 0,], [0.5, 0.75, 0]])
Shear_img=cv2.warpAffine(img, Sh, (width, height))
```

It may be noted that in this case it does crop the image.

MATLAB:

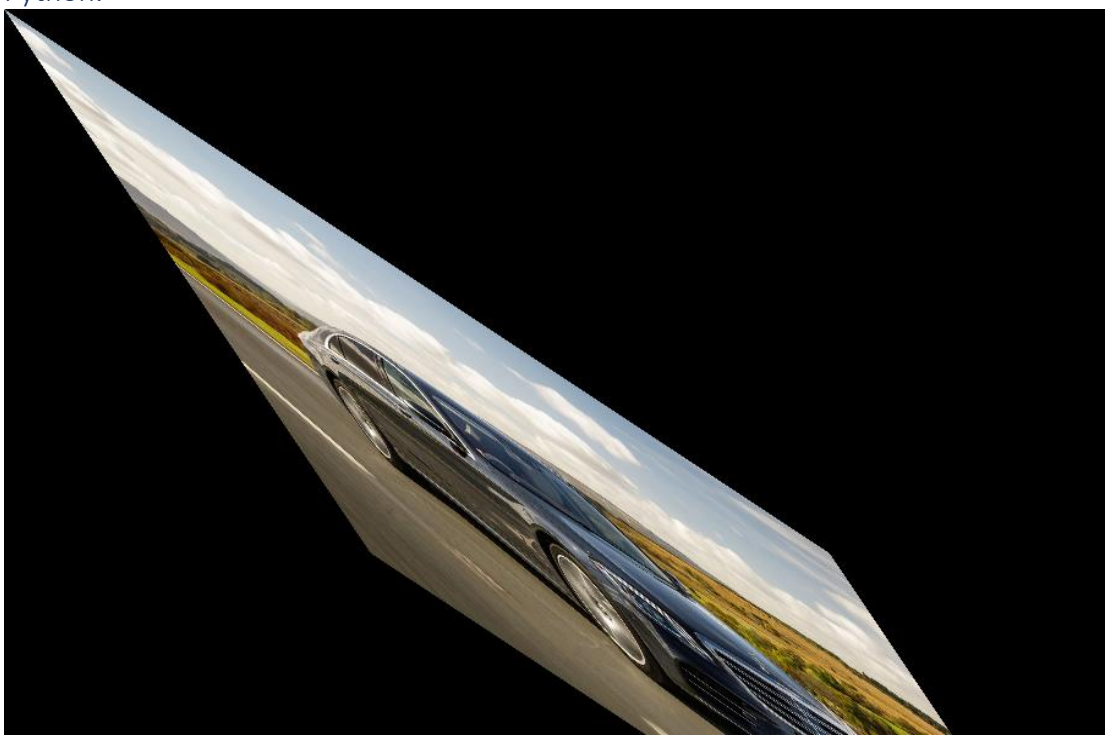
The same transformations in MATLAB can be obtained using the following lines:

```
Sh=maketform('affine', [0.75 0.5 0; 0.5 0.75 0; 0 0 1])
Shear_img=imtransform(img,Sh, 'cubic')
```

It may be noted that it does not crop the image.

Result:

Python:



MATLAB:

Projective Transformations

These are the most general "linear" transformations and require the use of the homogenous coordinates. In case of 2D, e.g., the last row of the homogenous transformations has been kept as [0 0 0 1] but in case of projective transformation, it does not have to be. These transformation can bring finite points to infinity and points at infinity to finite range. It preserves the straight lines. Or in other words, it preserves the cross ratio. One of the applications of this transformation is to capture information in the image taken from different angles. It aligns the image properly. For 3D, it is given by:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \\ p_{41} & p_{42} & p_{43} & p_{44} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Code:

Let us try to generate projective transformation for taking car out of the picture.

Python:

We need to specify four points both on reference image and transformed image to find the unknowns of the projective transformations. Pts1 represent points on reference image (obtained using '`imtool(img)`' command in MATLAB') while Pts2 represent points on transformed image (same size as that of reference image). The code is as follows:

```
pts1=np.float32([[130, 194],[130, 461], [735,195], [735,460]])
pts2=np.float32([[0, 0], [0,600], [900,0], [900, 600]])

P = cv2.getPerspectiveTransform(pts1,pts2)
proj_img=cv2.warpPerspective(img, P, (width, height) )
```

MATLAB:

The same thing can be achieved using the following functions in the MATLAB:

```
P=fitgeotrans(pts1, pts2,'projective')
Proj_img=imwrap(img, P)
```

.

Result:

Python: