

UNIT 3 – Edges and Geometric Transformation

- Problem Specification and Algorithm
- Affine Transformations
- Perspective Transformations
- Specification of More Complex Transformations
- Interpolation
- Modelling and Removing Distortion from Cameras
- Edge Detection
- Contour Segmentation
- Hough Transform

Geometric transformations (or operations) are used in image processing for a variety of reasons.

- They allow us to bring multiple images into the same frame of reference so that they can be combined (e.g. when forming a mosaic of two or more images) or compared (e.g. when comparing images taken at different times to see what changes have occurred).
- They can be used to eliminate distortion (e.g. barrel distortion from a wide angle lenses) in order to create images with evenly spaced pixels.
- Geometric transformations can also simplify further processing (e.g. by bringing the image of a planar object into alignment with the image axes;

Problem Specification and Algorithm

Given a distorted image $f(i, j)$ and a corrected image $f'(i', j')$ we can model the geometric transformation between their coordinates as $i = T_i(i', j')$ and $j = T_j(i', j')$ that is given the coordinates (i', j') in the corrected image, the functions $T_i()$ and $T_j()$ compute the corresponding coordinates (i, j) in the distorted image.

There are two main scenarios for determining the correspondences:

1. Obtaining the distorted image by imaging a known pattern (such as in Figure 5.2) so that the corrected image can be produced directly from the known pattern.
2. Obtaining two images of the same object, where one image (referred to as the distorted image) is to be mapped into the frame of reference of the other image (referred to as the corrected image).

Once sufficient correspondences are determined between the sample distorted and corrected images, it is relatively straightforward to compute the geometric transformation function. Once the transformation is determined it can be applied to the distorted image and to any other images that require the same ‘correction’.

A geometric transformation is applied as follows: For every point in the output/corrected image (i', j'):

- Determine where it came from (i, j) using $T_i()$ and $T_j()$.
- Interpolate a value for the output point from close neighbouring points in the input image, bearing in mind that $T_i()$ and $T_j()$ will compute real values and hence the point (i, j) is likely to be between pixels.

AFFINE TRANSFORMATIONS

In Affine Transformation, all parallel lines in the original image will still be parallel in the output image.

To apply affine transformation on an image, we need three points on the input image and corresponding point on the output image.

So first, we define these points and pass to the function **cv2.getAffineTransform()**. It will create a 2×3 matrix, we term it a transformation matrix **M**.

We can find the transformation matrix **M** using the following **syntax** –

M = cv2.getAffineTransform(pts1,pts2)

Where **pts1** is an array of three points on the input image and **pts2** is an array of the corresponding three points on the output image. The translation matrix **M** is a numpy array of type **np.float64**.

We pass **M** to the **cv2.warpAffine()** function as an argument. See the **syntax** given below –

cv2.warpAffine(img,M,(cols,rows))

where,

- **img** – The image to be affine transformed.

- **M** – Affine transformation matrix defined above.
- **(cols, rows)** – Width and height of the image after affine transformation.

Steps

To perform an image affine transformation, you can follow the steps given below –

1. Import the required library. In all the following Python examples, the required Python library is **OpenCV**. Make sure you have already installed it.
 - `import cv2`
2. Read the input image using **cv2.imread()** function. Pass the full path of the input image.
 - `img = cv2.imread('lines.jpg')`
3. Define **pts1** and **pts2**. We need three points from the input image and their corresponding locations in the output image.
 - `pts1 = np.float32([[50,50],[200,50],[50,200]])`
 - `pts2 = np.float32([[10,100],[200,50],[100,250]])`
4. Compute the affine transform matrix **M** using **cv2.getAffineTransform(pts1, pts2)** function.
 - `M = cv2.getAffineTransform(pts1, pts2)`
5. Transform the image using **cv2.warpAffine()** method. `cols` and `rows` are the desired width and height of the image after transformation.
 - `dst = cv2.warpAffine(img,M,(cols,rows))`
6. Display the affine transformed image.
 - `cv2.imshow("Affine Transform", dst)`

Many common transformations can be described using the affine transform, which is defined as follows:

$$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix} \begin{bmatrix} i' \\ j' \\ 1 \end{bmatrix} \quad (5.1)$$

Different Affine Transformation and Implementation


1. Translation

A translation is a function that moves every point with a constant distance in a specified direction. It is specified as t_x and t_y which will provide the orientation and the distance.

t_x : Width shift.


t_y : Height shift.

The mathematical transformation is the following...

	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$	t_x specifies the displacement along the x axis t_y specifies the displacement along the y axis.
---	---	---

2. Rotation


Rotation is a circular transformation around a point or an axis. We can specify the angle of rotation to rotate our image around a point or an axis.

	$\begin{bmatrix} \cos(q) & \sin(q) & 0 \\ -\sin(q) & \cos(q) & 0 \\ 0 & 0 & 1 \end{bmatrix}$	q specifies the angle of rotation.
---	--	--------------------------------------

theta: Rotation angle in degrees.

3. Scaling

Scaling is a linear transformation that enlarges or shrinks objects by a scale factor that is the same in all directions. We can specify the values of the s_x and s_y to enlarge or shrink our images. It is basically zooming in the image or zooming out the image.

	$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	s_x specifies the scale factor along the x axis s_y specifies the scale factor along the y axis.
---	---	---

s_x : Zoom in x direction.

s_y : Zoom in y direction

4. Shear

Shear is sometimes also referred to as transvection. A transvection is a function that shifts every point with constant distance in a basis direction(x or y).



$$\begin{bmatrix} 1 & sh_y & 0 \\ sh_x & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

sh_x specifies the shear factor along the x axis

sh_y specifies the shear factor along the y axis.

shear: Shear angle in degrees.

PYTHON CODE - ROTATION

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('food.jpeg')
rows, cols, ch = img.shape

pts1 = np.float32([[50, 50],
                   [200, 50],
                   [50, 200]])

pts2 = np.float32([[10, 100],
                   [200, 50],
                   [100, 250]])

M = cv2.getAffineTransform(pts1, pts2)
dst = cv2.warpAffine(img, M, (cols, rows))

plt.subplot(121)
plt.imshow(img)
plt.title('Input')

plt.subplot(122)
plt.imshow(dst)
plt.title('Output')
```

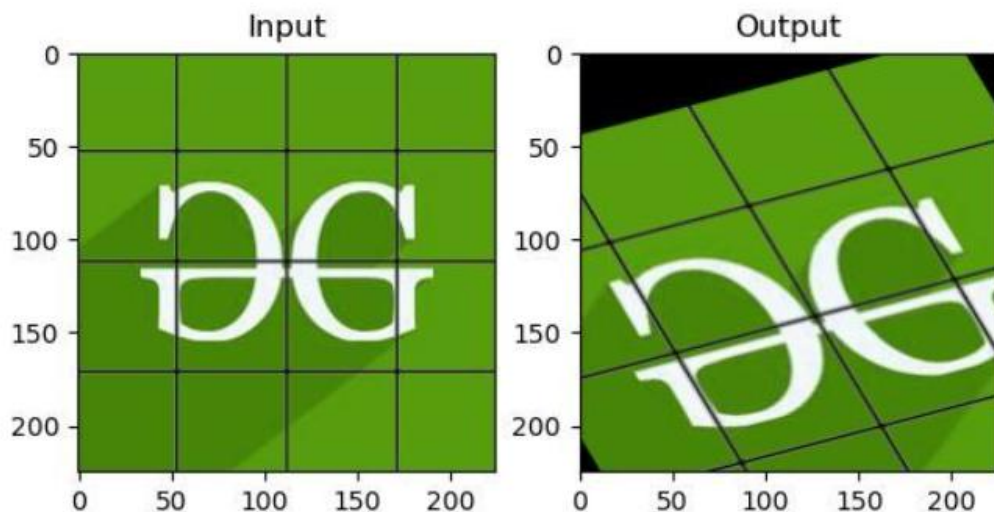
```
plt.show()

# Displaying the image
while(1):

    cv2.imshow('image', img)
    if cv2.waitKey(20) & 0xFF == 27:
        break

cv2.destroyAllWindows()
```

Output:



PERSPECTIVE TRANSFORMATION

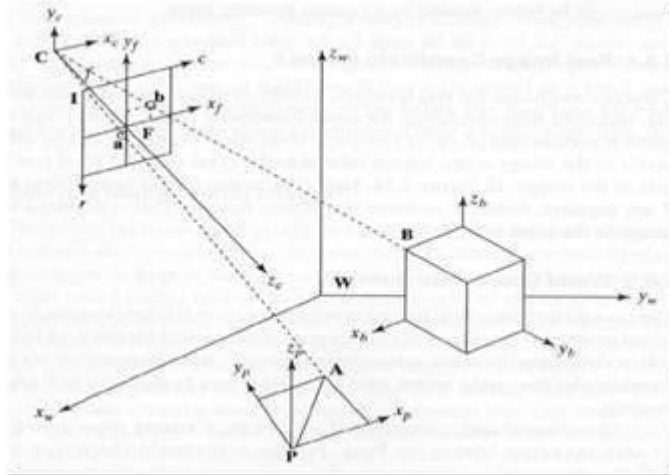
When human eyes see near things they look bigger as compare to those who are far away. This is called perspective in a general way. Whereas transformation is the transfer of an object e.t.c from one state to another.

So overall, the perspective transformation deals with the conversion of 3d world into 2d image. The same principle on which human vision works and the same principle on which the camera works.

We will start this by the concept of frame of reference:

Frame of reference:

Frame of reference is basically a set of values in relation to which we measure something.



5 frames of reference

In order to analyze a 3d world/image/scene, 5 different frame of references are required.

- Object
- World
- Camera
- Image
- Pixel

Object coordinate frame

Object coordinate frame is used for modeling objects. For example, checking if a particular object is in a proper place with respect to the other object. It is a 3d coordinate system.

World coordinate frame

World coordinate frame is used for co-relating objects in a 3 dimensional world. It is a 3d coordinate system.

Camera coordinate frame

Camera co-ordinate frame is used to relate objects with respect of the camera. It is a 3d coordinate system.

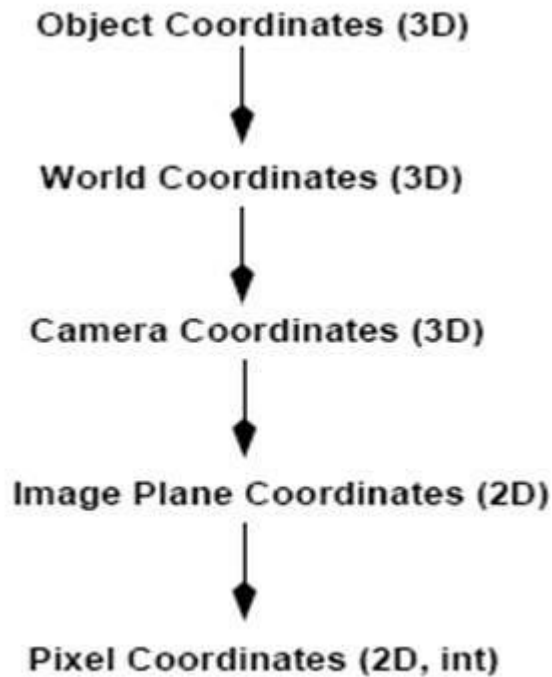
Image coordinate frame

It is not a 3d coordinate system, rather it is a 2d system. It is used to describe how 3d points are mapped in a 2d image plane.

Pixel coordinate frame

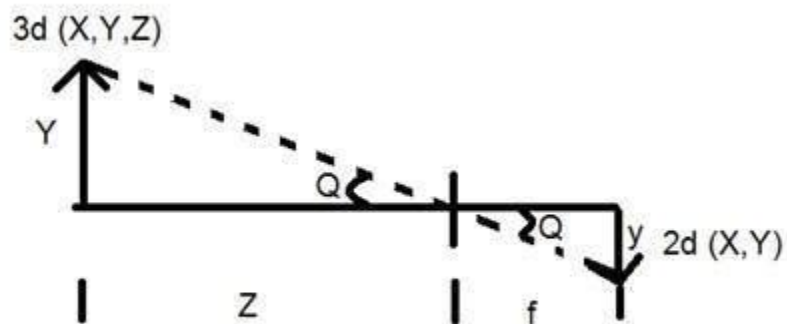
It is also a 2d coordinate system. Each pixel has a value of pixel co ordinates.

Transformation between these 5 frames



Thats how a 3d scene is transformed into 2d, with image of pixels.

Now we will explain this concept mathematically.



Where

$Y = 3d$ object

$y = 2d$ Image

f = focal length of the camera

Z = distance between object and the camera

Now there are two different angles formed in this transform which are represented by Q .

The first angle is

$$\tan \theta = -\frac{y}{f}$$

Where minus denotes that image is inverted. The second angle that is formed is:

$$\tan \theta = \frac{Y}{Z}$$

Comparing these two equations we get

$$Y = -f \frac{Y}{Z}$$

From this equation, we can see that when the rays of light reflect back after striking from the object, passed from the camera, an invert image is formed.

We can better understand this, with this example.

For example

Calculating the size of image formed

Suppose an image has been taken of a person 5m tall, and standing at a distance of 50m from the camera, and we have to tell that what is the size of the image of the person, with a camera of focal length is 50mm.

Solution:

Since the focal length is in millimeter, so we have to convert every thing in millimeter in order to calculate it.

So,

$$Y = 5000 \text{ mm.}$$

$$f = 50 \text{ mm.}$$

$$Z = 50000 \text{ mm.}$$

Putting the values in the formula, we get

$$Y = -f \frac{Y}{Z} = -50 \times 5000 / 50000$$

$$= -5 \text{ mm.}$$

Again, the minus sign indicates that the image is inverted.

INTERPOLATION

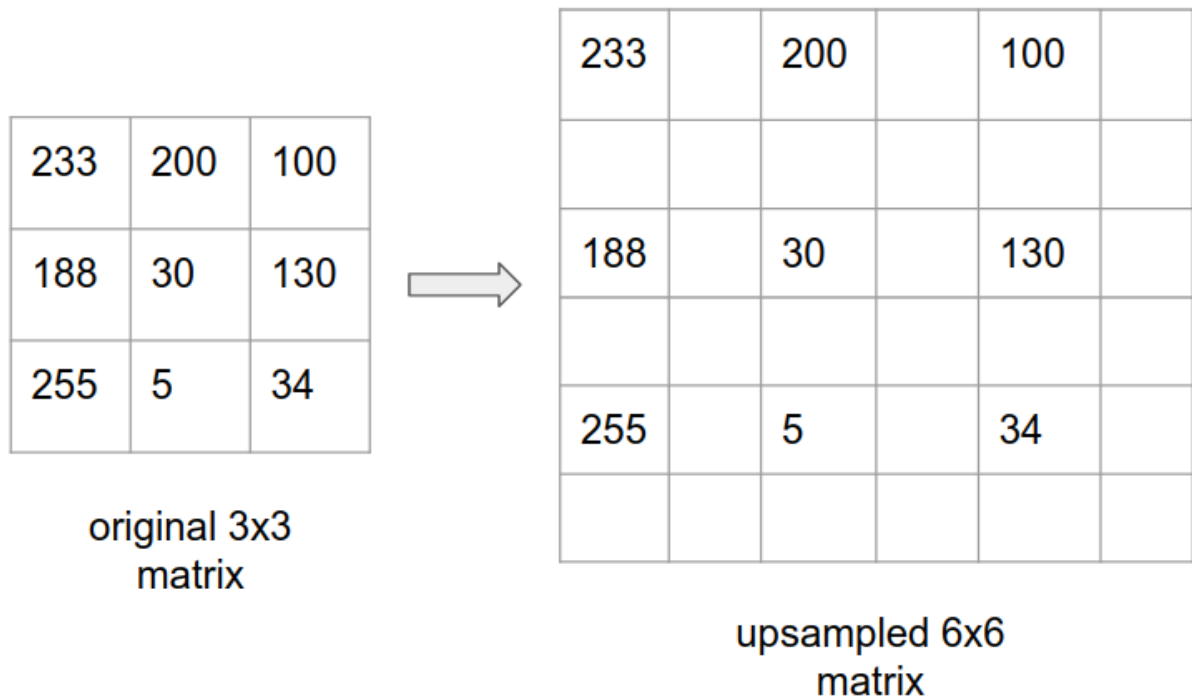
As each point in the output image will map to real coordinates in the input (distorted) image, it is unlikely that the coordinates selected in the input image will correspond precisely to any one pixel. Hence we need to interpolate the value for the output point from the nearby surrounding pixels in the input image. There are a variety of possible interpolation schemes and three of these will be detailed here.

Interpolation is generally used to estimate new data points from the known data points by statisticians for better understanding of the underlying data, can be used to approximate complex functions for efficient experimentations or even used for scaling images!

Image Resizing — Basic Idea:

A 2-d image is basically represented as a 2-dimensional matrix with each cell in the matrix containing a pixel value. So when we say scaling up this matrix, it means creating a bigger matrix than the original one and fill up the missing pixel values in this bigger matrix.

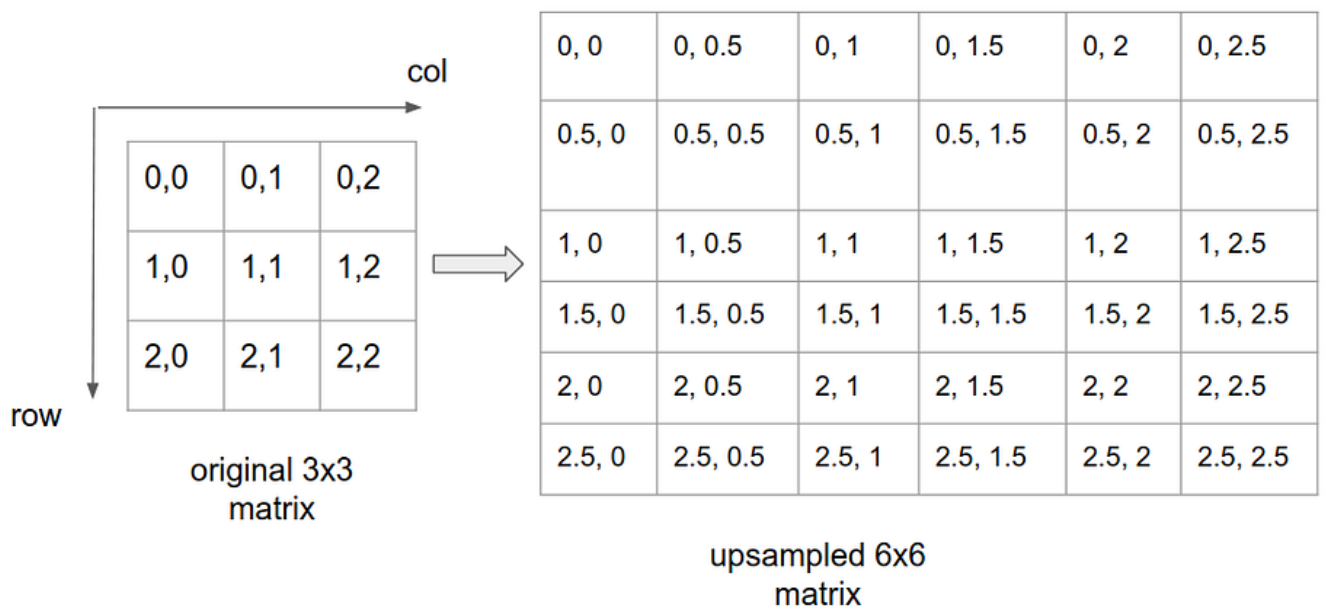
Let us look at an image to make the idea more concrete:



1. Consider an *input matrix* of size 3×3 with each cell containing some pixel values in the range $0-255$. row, col indices starts from 0 to $n-1$ where n is the row/col length.
2. Now let's create a new 6×6 *output(upscaled) empty matrix*.
3. Inorder to start filling the pixel values for the new matrix, we first have to represent the *output coordinate space* *interms of the input coordinate space* i.e. for every (row, col) in the output matrix, what is the corresponding (row, col) in the input matrix? This is just the scaling factor which is $1/2$ in our case.

4. To make it more clearer, the row scaling factor is $1/2$ and column scaling factor is $1/2$ (row and col will have separate scaling factors but since our eg considers a square matrix, both are same here).
5. *row 0, col 0* in output is mapped to *row 0, col 0* in input, whereas *row 1, col 1* in output is mapped to *row 0.5, col 0.5* in input and so on.

Now let's look at the mapped coordinate space:



The above image shows how the transformed (*row, col*) coordinates look like for the output matrix.

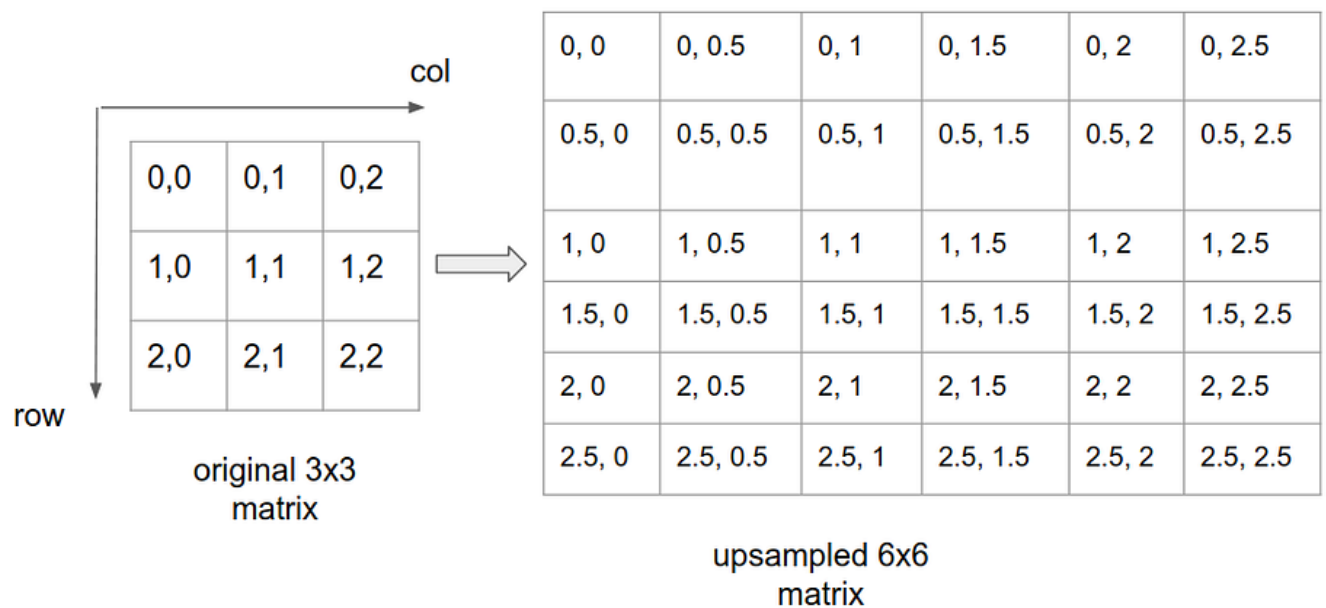
One can see that there are known cells such as 0, 0 or 2, 2 (in integers) and there are also unknown cells such as 1.5, 2 or 0, 2.5 (in floating point).

It is easier to fill in the pixel values for the known cells by simply picking the corresponding pixel values of those (row, col) cells from the original matrix but how do we find the pixel values for the unknown cells?

This is same as estimating new data points given some set of input data points where the given data points are pixel intensities from our input 3x3 matrix and the unknown pixel intensities(or new data points) are computed using them. Hence interpolation is used to find those unknown pixel values.

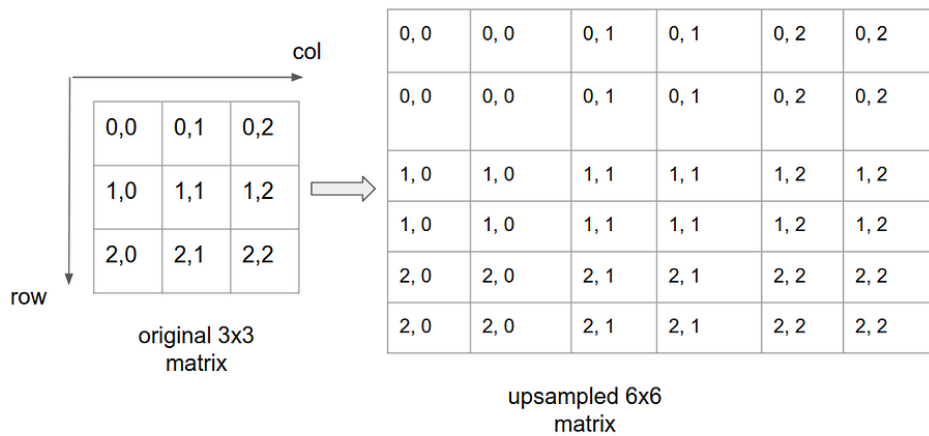
1. Nearest Neighbour Interpolation:

Nearest neighbour interpolation is quite a straightforward way of computing the pixel values for a new image.



Here in the above image, a simple way to find the values for unknown cells such as (0.5, 0.5) or (2, 2.5) is to simply round them off to nearest integer. for eg: (2, 2.5) to (2, 2) or (1.5, 2.5) to (1, 2). *This is called nearest neighbour interpolation.*

The resultant coordinate space looks like something like:



Now that all are known cells, the pixel values can simply be taken from the original matrix.

$$f'(i', j') = f(\text{round}(i), \text{round}(j))$$

In nearest neighbour interpolation we simply round the real coordinates so that we use the nearest pixel value. This scheme results in very distinct blocky effects that are frequently visible.

The algorithm is as follows:

1. Consider an input *image I* of dimensions (hi, wi, c) where c=3. Let's assume the *image I* has to be resized to twice its size. i.e. output *image R* of dimensions (hr, wr, c).
2. For every x, y coordinates in the output image, find its corresponding x, y coordinates in the input image.
3. Now each of the remapped x and y output coordinates can either correspond to the actual input pixel coordinates or coordinates in-between them (i.e. floating point coordinates).

4. For floating-point coordinates, we could simply round them off and pick the corresponding integer coordinate — hence called nearest neighbour interpolation.

PYTHON CODE:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow

image = cv2.imread("insta.png")
# Loading the image

half = cv2.resize(image, (0, 0), fx = 0.1, fy = 0.1)
bigger = cv2.resize(image, (960, 960))

stretch_near = cv2.resize(image, (960, 960), interpolation = cv2.INTER_NEAREST)
cv2_imshow()
```

2. Bilinear Interpolation:

Bilinear interpolation makes use of linear interpolation method to compute the pixel values for a new image.

The method works as follows:

1. For any unknown (row, col) cell in the upsampled matrix, pick the 4 nearest pixels. These nearest pixels can be obtained by doing $int(row)$, $int(col)$, $int(row) + 1$ and $int(col) + 1$. Let's call it $row1$, $col1$, $row2$ and $col2$
2. Perform linear interpolation at $(row1, col)$ using $(row1, col1)$ and $(row1, col2)$ and similarly linear interpolation at $(row2, col)$ using $(row2, col1)$ and $(row2, col2)$. Both are along x-directions.
3. Do one final linear interpolation at (row, col) using $(row1, col)$ and $(row2, col)$.
4. The above steps are repeated for every unknown (row, col) cell in the new matrix.

- Note that the same pixel value can be obtained by doing two linear interpolation along y-directions first and then along x-direction.

Let us take an example and see how this method works in practice:

- For any unknown (row, col) cell in the upsampled matrix, pick the 4 nearest pixels. For eg: for cell $(0.5, 0.5)$, the 4 nearest pixels are $(0, 0)$, $(0, 1)$, $(1, 0)$, $(1, 1)$.

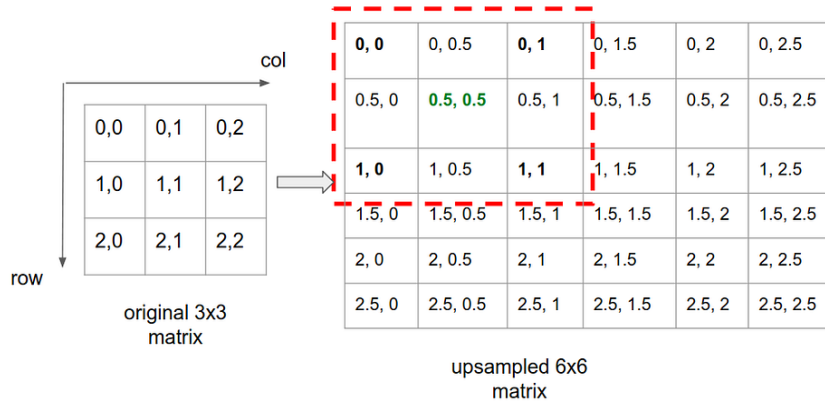


Fig. Finding the pixel value at $(0.5, 0.5)$

- Finding the pixel value at $(0.5, 0.5)$ means first finding the pixels values at $(0, 0.5)$ and $(1, 0.5)$ and then using them to find the value at $(0.5, 0.5)$
- Do linear interpolation twice along x-direction — one at $(0, 0.5)$ using $\langle (0, 0), (0, 1) \rangle$ and another at $(1, 0.5)$ using $\langle (1, 0), (1, 1) \rangle$
- Then another interpolation along y-direction — at $(0.5, 0.5)$ using $(0, 0.5)$ and $(1, 0.5)$

Bilinear Interpolation Equation

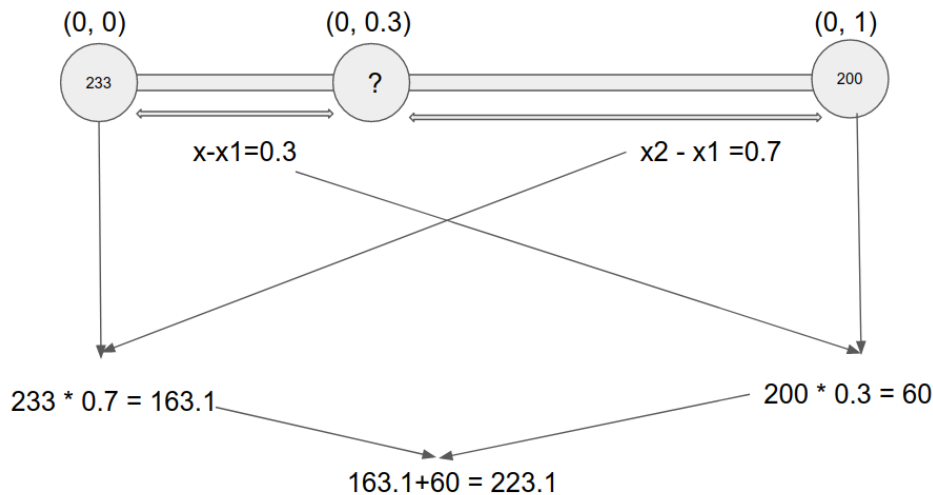
Let $(x1, y)$ and $(x2, y)$ be the pixel coordinates in the new matrix and their intensities be $I1$ and $I2$ where $x2 > x1$.

Pixel intensity at an unknown point $(x, y1)$ can be computed as:

$$I_{new} = \frac{x_2 - x}{x_2 - x_1} * I_1 + \frac{x - x_1}{x_2 - x_1} * I_2$$

where $x_1 \geq x \geq x_2$

Pictorially, this can be depicted as:



$I_1=233$, $I_2=200$ and $I_{new}=223.1$

New pixel intensity is nothing but the weighted sum of pixel intensities of the nearest two pixels where the weight is determined by the distance from the nearest pixels.

The algorithm is as follows:

1. Consider an input *image I* of dimensions (h_i, w_i, c) where $c=3$. Let's assume the *image I* has to be resized to twice its size. i.e. output *image R* of dimensions (h_r, w_r, c).
2. For every x, y coordinates in the output image, find its corresponding x, y coordinates in the input image.
3. Now each of the remapped x and y output coordinates can either correspond to the actual input pixel coordinates or coordinates in-between them (i.e. floating point coordinates).

4. For floating-point coordinates, get the nearest four pixels by by doing $\text{int}(\text{row})$, $\text{int}(\text{col})$, $\text{int}(\text{row}) + 1$ and $\text{int}(\text{col}) + 1$.
5. For every unknown cell, perform linear interpolation twice along x-direction and one more along y-direction to compute the pixel intensity. Note that there can also be only one coordinate which is float while the other is integer — for eg: (0, 0.5). *In such cases, it is enough to do linear interpolation once along either x or y direction to compute the unknown cell.*

PYTHON CODE

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow

image = cv2.imread("insta.png")
# Loading the image

stretch_near = cv2.resize(image, (960, 960), interpolation = cv2.INTER_LINEAR)
cv2_imshow(stretch_near)
```

3. Bi-Cubic Interpolation

Bi-cubic interpolation overcomes the step-like boundary problems seen in nearest neighbour interpolation and addresses the bilinear interpolation blurring problem. It improves the brightness interpolation by approximating locally using 16 neighbouring points on a bi-cubic polynomial surface. This function is very similar to the Laplacian, which is sometimes used to sharpen images. Bi-cubic interpolation is frequently used in raster displays, which is why images will rarely appear blocky when blown up on your screen.

PYTHON CODE:

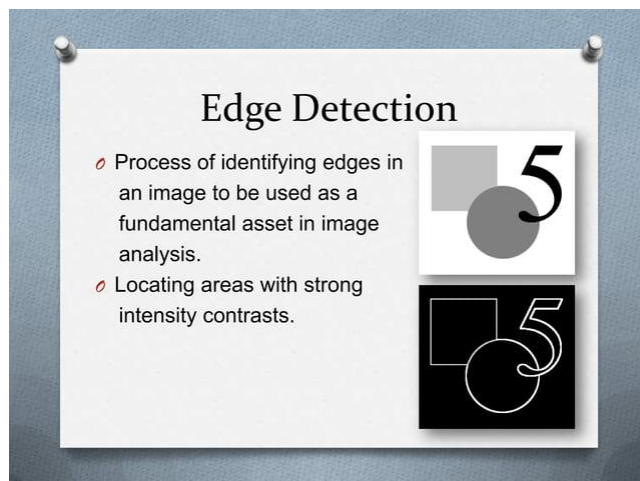
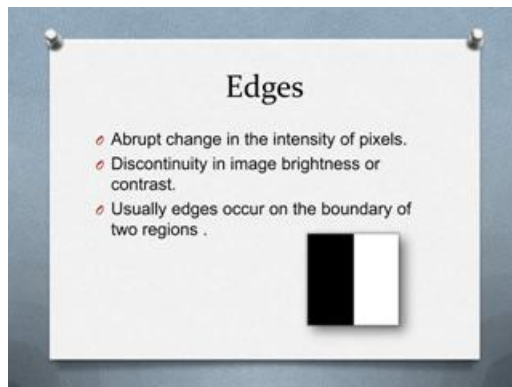
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

image = cv2.imread("insta.png")
# Loading the image
```

```
stretch_near = cv2.resize(image, (1920, 1920), interpolation = cv2.INTER_CUBIC)
```

```
cv2.imshow(stretch_near)
```

EDGE DETECTION



Edge Detection Usage

- ◊ Reduce unnecessary information in the image while preserving the structure of the image.
- ◊ Extract important features of an image
 - ◊ Corners
 - ◊ Lines
 - ◊ Curves
- ◊ Recognize objects, boundaries, segmentation.
- ◊ Part of computer vision and recognition.

Edge Types

◊ Step Edge



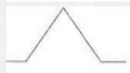
◊ Ramp Edge



◊ Ridge

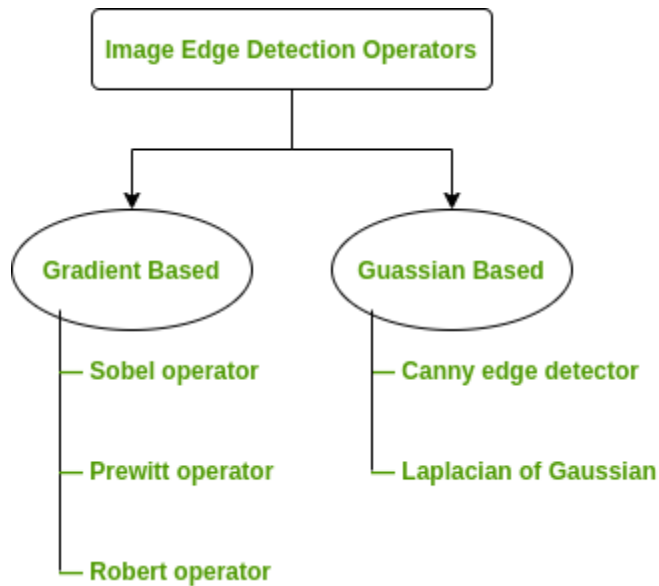


◊ Roof



Two types:

- **Gradient** – based operator which computes first-order derivations in a digital image like, Sobel operator, Prewitt operator, Robert operator
- **Gaussian** – based operator which computes second-order derivations in a digital image like, Canny edge detector, Laplacian of Gaussian



Sobel Operator: It is a discrete differentiation operator. It computes the gradient approximation of image intensity function for image edge detection. At the pixels of an image, the Sobel operator produces either the normal to a vector or the corresponding gradient vector. It uses two 3 x 3 kernels or masks which are convolved with the input image to calculate the vertical and horizontal derivative approximations respectively –

$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad M_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

% MATLAB code for Sobel operator

% edge detection

k=imread('logo.png');

k=rgb2gray(k);

k1=double(k);

s_msk=[-1 0 1; -2 0 2; -1 0 1];

kx=conv2(k1, s_msk, 'same');

ky=conv2(k1, s_msk, 'same');

ked=sqrt(kx.^2 + ky.^2);

%display the images.

imtool(k,[]);

%display the edge detection along x-axis.

imtool(abs(kx), []);

%display the edge detection along y-axis.

imtool(abs(ky),[]);

%display the full edge detection.

imtool(abs(ked),[]);

Advantages:

1. Simple and time efficient computation
2. Very easy at searching for smooth edges

Limitations:

1. Diagonal direction points are not preserved always
2. Highly sensitive to noise
3. Not very accurate in edge detection
4. Detect with thick and rough edges does not give appropriate results

Prewitt Operator: This operator is almost similar to the sobel operator. It also detects vertical and horizontal edges of an image. It is one of the best ways to detect the orientation and magnitude of an image. It uses the kernels or masks –

$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad M_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

```
% MATLAB code for prewitt
% operator edge detection
k=imread("logo.png");
k=rgb2gray(k);
k1=double(k);
p_msk=[-1 0 1; -1 0 1; -1 0 1];
kx=conv2(k1, p_msk, 'same');
ky=conv2(k1, p_msk, 'same');
ked=sqrt(kx.^2 + ky.^2);
```

```
% display the images.
```

```
imtool(k,[]);
```

```
% display the edge detection along x-axis.
```

```
imtool(abs(kx), []);
```

```
% display the edge detection along y-axis.
```

```
imtool(abs(ky),[]);
```

```
% display the full edge detection.
```

```
imtool(abs(ked),[]);
```

Advantages:

1. Good performance on detecting vertical and horizontal edges
2. Best operator to detect the orientation of an image

Limitations:

1. The magnitude of coefficient is fixed and cannot be changed
2. Diagonal direction points are not preserved always
3. **Robert Operator:** This gradient-based operator computes the sum of squares of the differences between diagonally adjacent pixels in an image through discrete differentiation. Then the gradient approximation is made. It uses the following 2 x 2 kernels or masks –

$$M_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad M_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

```
import cv2
```

```
import numpy as np
```

```
from scipy import ndimage
```

```
roberts_cross_v = np.array( [[1, 0 ],
```

```
[0,-1 ]] )
```

```
roberts_cross_h = np.array( [[ 0, 1 ],
```

```
[-1, 0 ]] )
```

```

img = cv2.imread('input.webp',0).astype('float64')
img/=255.0
vertical = ndimage.convolve( img, roberts_cross_v )
horizontal = ndimage.convolve( img, roberts_cross_h )

edged_img = np.sqrt( np.square(horizontal) + np.square(vertical))
edged_img*=255
cv2.imwrite('output.jpg',edged_img)

```

Advantages:

1. Detection of edges and orientation are very easy
2. Diagonal direction points are preserved

Limitations:

1. Very sensitive to noise
2. Not very accurate in edge detection

SECOND ORDER OPERATORS

1. **Marr-Hildreth Operator or Laplacian of Gaussian (LoG):** It is a gaussian-based operator which uses the Laplacian to take the second derivative of an image. This really works well when the transition of the grey level seems to be abrupt. It works on the zero-crossing method i.e when the second-order derivative crosses zero, then that particular location corresponds to a maximum level. It is called an edge location. Here the Gaussian operator reduces the noise and the Laplacian operator detects the sharp edges.

The Gaussian function is defined by the formula:

$$G(x, y) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp - \left(\frac{x^2 + y^2}{2\sigma^2} \right)$$

Where Sigma is the standard deviation. And the LoG operator is computed from

$$\text{LoG} = \frac{\partial^2}{\partial x^2} G(x, y) + \frac{\partial^2}{\partial y^2} G(x, y) = \frac{x^2 + y^2 - 2\sigma^2}{\sigma^4} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

#OPENCV implementation

```
import cv2
import matplotlib.pyplot as plt
image = cv2.imread(r"E:\eye.png", cv2.IMREAD_COLOR)
image = cv2.GaussianBlur(image, (3, 3), 0)
image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
filtered_image = cv2.Laplacian(image_gray, cv2.CV_16S, ksize=3)
# Plot the original and filtered images
plt.figure(figsize=(10, 5))
plt.subplot(121)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.subplot(122)
plt.imshow(filtered_image, cmap='gray')
plt.title('LoG Filtered Image')
plt.show()
```

Advantages:

1. Easy to detect edges and their various orientations
2. There is fixed characteristics in all directions

Limitations:

1. Very sensitive to noise
2. The localization error may be severe at curved edges
3. It generates noisy responses that do not correspond to edges, so-called “false edges”

Canny Operator: It is a gaussian-based operator in detecting edges. This operator is not susceptible to noise. It extracts image features without affecting or altering the feature. Canny edge detector have advanced algorithm derived from the previous work of Laplacian of Gaussian operator. It is widely used an optimal edge detection technique. It detects edges based on three criteria:

1. Low error rate
2. Edge points must be accurately localized
3. There should be just one single edge response

```
import cv2  
img = cv2.imread('test.jpeg') # Read image  
# Setting parameter values  
t_lower = 50 # Lower Threshold  
t_upper = 150 # Upper threshold  
# Applying the Canny Edge filter  
edge = cv2.Canny(img, t_lower, t_upper)  
cv2.imshow('original', img)  
cv2.imshow('edge', edge)  
cv2.waitKey(0)  
cv2.destroyAllWindows()
```

Advantages:

1. It has good localization
2. It extract image features without altering the features
3. Less Sensitive to noise

Limitations:

1. There is false zero crossing
2. Complex computation and time consuming

Some Real-world Applications of Image Edge Detection:

- medical imaging, study of anatomical structure
- locate an object in satellite images
- automatic traffic controlling systems
- face recognition, and fingerprint recognition

CONTOUR SEGMENTATION

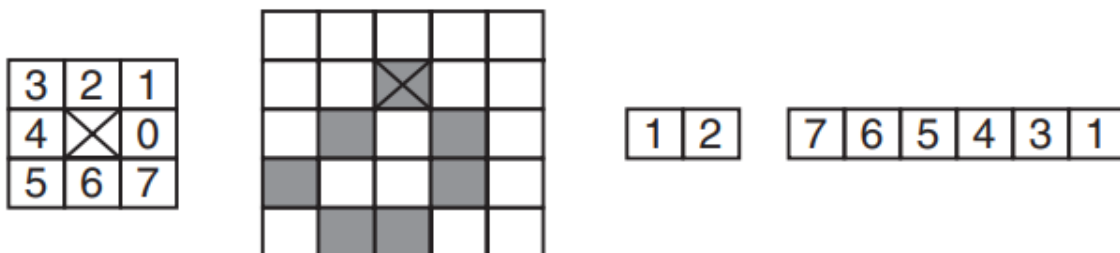
Unfortunately, extracting an edge image is not sufficient for most applications. We need to extract the information contained in the edge image and represent it more explicitly so that it can be reasoned with more easily. The extraction of edge data involves firstly deciding on which points are edges (as most have a non-zero gradient). This is typically achieved by edge image thresholding and non-maxima suppression (or through the use of Canny). The edge data then needs to be extracted from the image domain (e.g. using graph searching, border refining and so on) and represented in some fashion (e.g. BCCs, graphs, sequences of straight line segments and other geometric representations).

Basic Representations of Edge Data

There are many ways in which edge image data can be represented outside of the image domain. Here we will present two of the simplest: boundary chain codes and directed graphs.

1. Boundary Chain Codes

A boundary chain code (BCC) consists of a start point and a list of orientations to other connected edge points. The start point is specified by the (row,column) pair, and then the direction to the next point is repeatedly specified simply as a value from 0 to 7 (i.e. the eight possible directions to a neighbouring pixel). If considering this as a shape representation then we have to consider how useful it will be for further processing (e.g. shape/object recognition). BCCs are orientation dependant in that if the orientation of the object/region changes then the representation will change significantly. The representation will also change somewhat with object scale. It is only position dependent to the extent of the start point, although it should be noted that the start point is somewhat arbitrary for a closed contour



2. Directed Graphs

A directed graph is a general structure consisting of nodes (which correspond to edge pixels in this case) and oriented arcs (connections between bordering pixels/nodes). To create this type of graph, we add all edge pixels to the graph as nodes if their gradient values $s(x_i)$ are greater than some threshold (T). To decide on which nodes are then connected by arcs, we look at the orientation $s(x_i)$ associated with each node n_i to determine which neighbouring pixels x_j could be connected. The orientation vectors which have been quantised to eight possible directions, are orthogonal to the edge contours and hence the most likely next pixel will be to the side of the point (and hence is a possible arc). We also allow pixels on either side of this pixel (i.e. $\pm 45^\circ$) to

be considered to be connected via an arc. If any of these pixels have an associated node n_j then we add in a directed arc from n_i to n_j if the difference in orientation between the two corresponding pixels is less than $\pi/2$.

