Computer Vision
Project 2 Report
Rakshit Sareen (rs5606)

1. Affine Image Transformation

1.1 Resampling

Nearest Neighbor:
In nearest neighbor, we take the intensity from the pixel which is closest to the non-grid position of the transformation. We find this by using the int() function of python, i.e by rounding to the nearest integer of a non-grid (x,y)-coordinate to the next integer coordinates. We define a transformation function which takes the transformation matrix and the gray image as input. In this we follow the backward mapping i.e. the transformation is generally applied from the target image backwards to the source image which means we step through each pixel of the new image, determine the position in the source image, and take/calculate the intensity at this pixel to be used for the target image.

1.2 Bilinear Interpolation

Bilinear Interpolation uses a weighted average of the four nearest cell centers. This means that the output value could be different than the nearest input but is always within the same range of values as the input. Since the values can change, Bilinear is not recommended for categorical data. Instead, it should be used for continuous data like elevation and raw slope values.

Bilinear interpolation requires a neighborhood extending one pixel to the right and below the central sample. If the subsample position is given by (u, v), the resampled pixel value will be:

$$(1-v) * [(1-u) * p00 + u * p01] + v * [(1-u) * p10 + u * p11]$$

We make a Bilinear transformation function which takes the transformation matrix and the gray image as the input.

We apply the following transformation using Nearest Neighbor and Bilinear Interpolation transformation method:

We use the following Gray Image to perform geometric translations:

Translation:

$$\text{Translation: } \begin{bmatrix} 1 & 0 & b_x \\ 0 & 1 & b_y \\ 0 & 0 & 1 \end{bmatrix}$$

We use the translation matrix

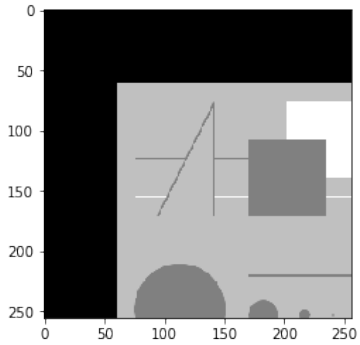
We give the value of bx and by depending on by how much we want to translate:
bx = 50
by = 55

Screenshot of the code:
For both nearest neighbors and Bilinear interpolation.

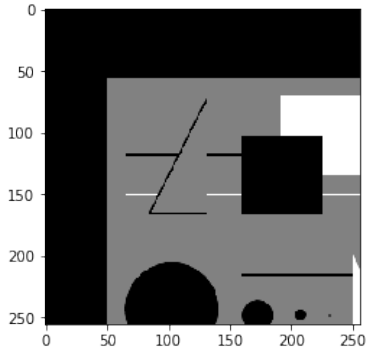
```
def translateImageNN(image,tx, ty):
    newImage = np.zeros_like(image)
    translate = np.array([[1,0,tx],[0,1,ty],[0,0,1]])
    inv_translate = np.linalg.inv(translate)
    for x in range(newImage.shape[0]):
        for y in range(newImage.shape[1]):
            point = np.array([[x],[y],[1]])
            new_point = np.matmul(inv_translate,point)
            a = int(new_point[0][0])
            b = int(new_point[1][0])
            #check if new image coordinates are present in the image
            if a > 0 and a < image.shape[0] and b > 0 and b < image.shape[1]:
                newImage[x][y] = image[a][b]
    return newImage
```

```
def translateBilinearTransformation(image, tx, ty):
    newImage = np.zeros_like(image)
    translate = np.array([[1,0,tx],[0,1,ty],[0,0,1]])
    inv_translate = np.linalg.inv(translate)
    intensity = 0
    row , col = image.shape[0] , image.shape[1]
    for x in range(newImage.shape[0]):
        for y in range(newImage.shape[1]):
            point = np.array([[x],[y],[1]])
            new_point = np.matmul(inv_translate,point)
            a,b = new_point[0][0] , new_point[1][0]
            if a < 0 or b < 0:
                continue
            xc,xf = int(math.ceil(a)),int(math.floor(a))
            yc,yf = int(math.ceil(b)),int(math.floor(b))
            if ((xc < row and xc > 0) and (yc < col and yc > 0) and (xf > 0 and xf < row) and (yf > 0 and yf < col)):
                if xc == xf and yc == yf:
                    intensity = image[xc,yc]
                elif xc == xf:
                    intensity = image[xc, yc] * (b - yf) + image[xc, yf] * (yc - b)
                elif yc == yf:
                    intensity = image[xc, yc] * (a - xf) + image[xf, yc] * (xc - a)
                else :
                    intensity = image[xf, yf] * (math.abs(xc - x) * math.abs(yc - y)) +\
                    image[xf, yc] * ((xc - x) * (y - yf)) +\
                    image[xc, yf] * ((x - xf) * (yc - y)) +\
                    image[xc, yc] * ((x - xf) * (y - yf))
            newImage[x][y] = 2*intensity
    return newImage
```

Result:



Nearest Neighborhood:



Bilinear Interpolation:

On zooming the image we see that the result obtained after applying nearest neighbourhood is that the image is patchy whereas after bilinear interpolation the result is better and not patchy. Thus the image quality obtained is better after applying bilinear interpolation. Also note that the image itself has straight edges, so the difference is very minute. In general as well, bilinear interpolation presents smooth images as compared to nearest neighbor interpolation.

Nearest Neigbourhood.

Bilinear Interpolation

Rotation:

Here we apply the following rotation matrix to rotate our image by theta =10 degrees.

$$\text{Rotation: } \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

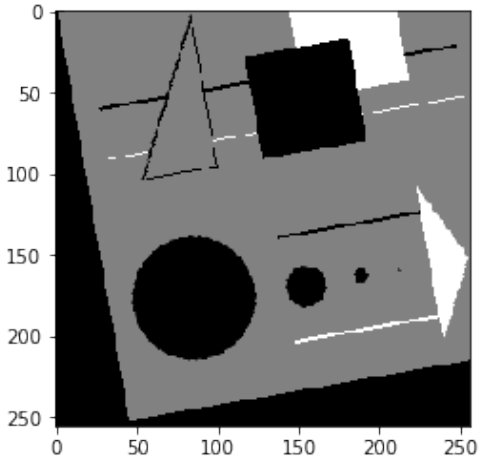
Screenshot of code

Both Nearest Neighbor and Bilinear Interpolation

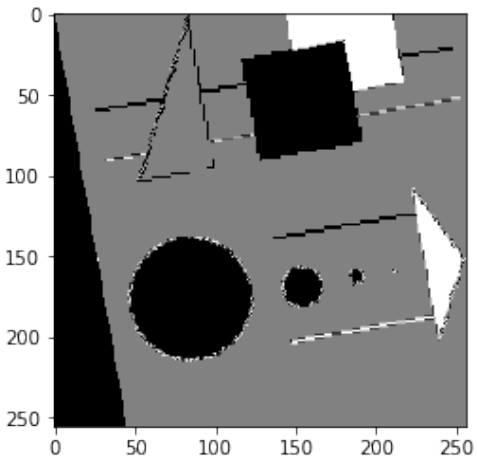
```
def rotateImageNN(image, degreeTheta):
    radians = math.radians(degreeTheta)
    sinTheta = math.sin(radians)
    cosTheta = math.cos(radians)
    newImage = np.zeros_like(image)
    rotate = np.array([[cosTheta,-sinTheta,0],[sinTheta,cosTheta,0],[0,0,1]])
    inv_rotate = np.linalg.inv(rotate)
    for x in range(newImage.shape[0]):
        for y in range(newImage.shape[1]):
            point = np.array([[x],[y],[1]])
            new_point = np.matmul(inv_rotate,point)
            a = int(new_point[0][0])
            b = int(new_point[1][0])
            #check if new image coordinates are present in the image
            if a > 0 and a < image.shape[0] and b > 0 and b < image.shape[1]:
            #if a < image.shape[0] and b < image.shape[1]:
                newImage[x][y] = 2*image[a][b]
    return newImage
```

```
def rotateBilinearTransformation(image, degreeTheta):
    newImage = np.zeros_like(image)
    radians = math.radians(degreeTheta)
    sinTheta = math.sin(radians)
    cosTheta = math.cos(radians)
    newImage = np.zeros_like(image)
    rotate = np.array([[cosTheta,-sinTheta,0],[sinTheta,cosTheta,0],[0,0,1]])
    inv_rotate = np.linalg.inv(rotate)
    intensity = 0
    row , col = image.shape[0] , image.shape[1]
    for x in range(newImage.shape[0]):
        for y in range(newImage.shape[1]):
            point = np.array([[x],[y],[1]])
            new_point = np.matmul(inv_rotate,point)
            a,b = new_point[0][0] , new_point[1][0]
            if a < 0 or b < 0:
                continue
            xc,xf = int(math.ceil(a)),int(math.floor(a))
            yc,yf = int(math.ceil(b)),int(math.floor(b))
            if ((xc < row and xc > 0) and (yc < col and yc > 0) and (xf > 0 and xf < row) and (yf > 0 and yf < col)):
                if xc == xf and yc == yf:
                    intensity = image[xc,yc]
                elif xc == xf:
                    intensity = image[xc, yc] * (b - yf) + image[xc, yf] * (yc - b)
                elif yc == yf:
                    intensity = image[xc, yc] * (a - xf) + image[xf, yc] * (xc - a)
                else :
                    intensity = image[xf, yf] * ((xc - x) * (yc - y)) +\
                    image[xf, yc] * ((xc - x) * (y - yf)) +\
                    image[xc, yf] * ((x - xf) * (yc - y)) +\
                    image[xc, yc] * ((x - xf) * (y - yf))
            newImage[x][y] = 2*intensity
    return newImage
```

Results:



Nearest Neighborhood



Bilinear Interpolation

Scaling:

Here we apply the following Scaling matrix to scale our image by the following scaling factor.

$$\text{Scaling: } \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

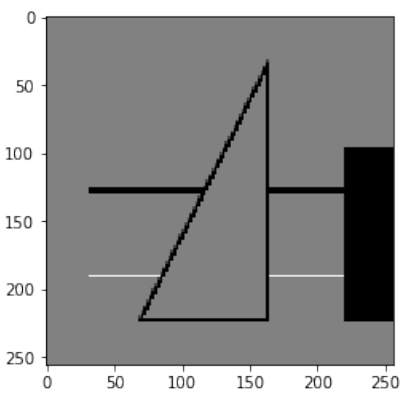
Taking scaling factor Sx = 2 and Sy = 2

Screenshot of the code:

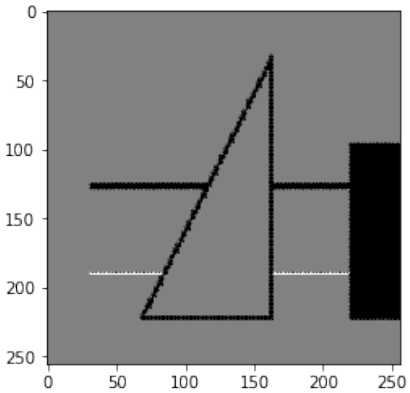
```python
def scaleImageNN(image, sx, sy):
    """
    Consider the image to be a numoy array
    sx, sy contains resizing information for x and y axis.
    intepolationType tells which type of interpolation to use - Nearest Neighbors / Bilinear

    """
    newImage = np.zeros_like(image)
    scale = np.array([[sx,0,0],[0,sy,0],[0,0,1]])
    inv_scale = np.linalg.inv(scale)
    for x in range(newImage.shape[0]):
        for y in range(newImage.shape[1]):
            point = np.array([[x],[y],[1]])
            new_point = np.matmul(inv_scale,point)
            a = int(new_point[0][0])
            b = int(new_point[1][0])
            #check if new image coordinates are present in the image
            if a < image.shape[0] and b < image.shape[1]:
            #if a > 0 and a < image.shape[0] and b > 0 and b < image.shape[1]:
                newImage[x][y] = image[a][b]
    return newImage
```

```python
def scaleBilinearTransformation(image, sx, sy):
    newImage = np.zeros_like(image)
    scale = np.array([[sx,0,0],[0,sy,0],[0,0,1]])
    inv_scale = np.linalg.inv(scale)
    for x in range(newImage.shape[0]):
        for y in range(newImage.shape[1]):
            point = np.array([[x],[y],[1]])
            new_point = np.matmul(inv_scale,point)
            a,b = new_point[0][0] , new_point[1][0]
            xc,xf = int(math.ceil(a)),int(math.floor(a))
            yc,yf = int(math.ceil(b)),int(math.floor(b))
            if xc == xf and yc == yf:
                intensity = image[xc,yc]
            elif xc == xf:
                intensity = image[xc, yc] * (b - yf) + image[xc, yf] * (yc - b)
            elif yc == yf:
                 intensity = image[xc, yc] * (a - xf) + image[xf, yc] * (xc - a)
            else :
                intensity = image[xf, yf] * ((xc - x) * (yc - y)) +\
                        image[xf, yc] * ((xc - x) * (y - yf)) +\
                        image[xc, yf] * ((x - xf) * (yc - y)) +\
                        image[xc, yc] * ((x - xf) * (y - yf))
            newImage[x][y] = 2*intensity
    return newImage
```

Result:



Nearest Neigborhood



Bilinear Interpolation

Shearing:

A transformation that slants the shape of an object is called the shear transformation. There are two shear transformations X-Shear and Y-Shear. One shifts X coordinates values and other shifts Y

coordinate values. However; in both the cases only one coordinate changes its coordinates and other preserves its values. Shearing is also termed as Skewing.

X-Shear

The X-Shear preserves the Y coordinate and changes are made to X coordinates, which causes the vertical lines to tilt right or left as shown in below figure.

The transformation matrix for X-Shear can be represented as −

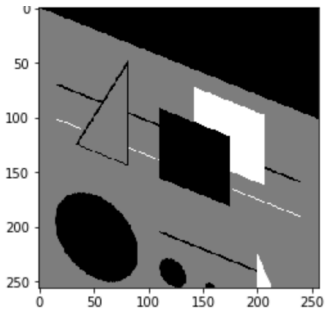
$$Xsh=\begin{bmatrix}1 & shx & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix} Xsh=\begin{bmatrix}1 & 0 & 0\\ shx & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

$$X' = X + Sh_x \cdot Y$$

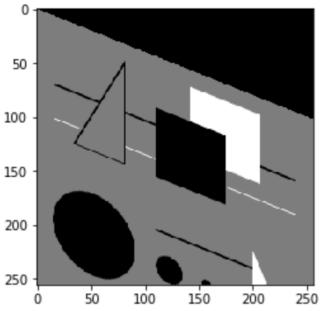
$$Y' = Y$$

Screenshot of code for x -shear

```
#shearing
sx = 0.4
shear_matrix = ([[1, sx, 0], [0, 1, 0], [0, 0, 1]])
shear_linear_image = transformation(shear_matrix, gray_image)
shear_bilinear_image = bilinear_trans(shear_matrix, gray_image)
plt.imshow(shear_linear_image, cmap = 'gray')
plt.show()
plt.imshow(shear_bilinear_image, cmap = 'gray')
```



Nearest Neighborhood.



Bilinear Interpolation

Y-Shear

The Y-Shear preserves the X coordinates and changes the Y coordinates which causes the horizontal lines to transform into lines which slopes up or down

The Y-Shear can be represented in matrix from as −

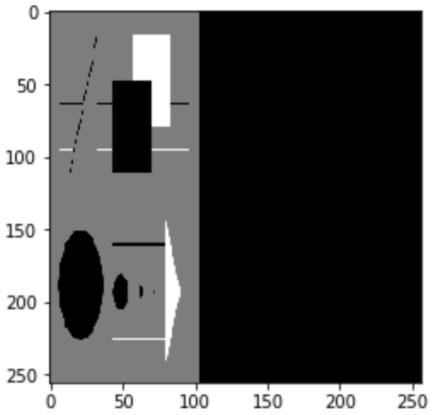
$$Ysh\begin{bmatrix}1 & 0 & 0\\ shy & 1 & 0\\ 0 & 0 & 1\end{bmatrix} Ysh\begin{bmatrix}1 & shy & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

$$Y' = Y + Sh_y \cdot X$$

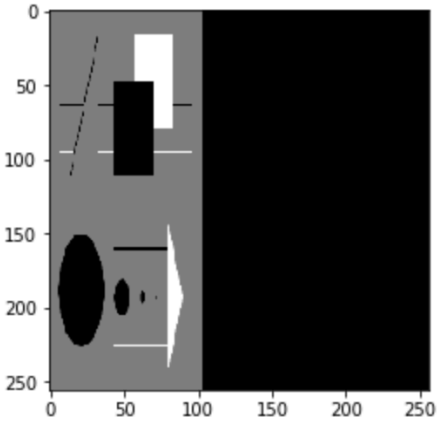
$$X' = X$$

```
sy = 0.4
shear_matrix = ([[1, 0, 0], [0, sy, 0], [0, 0, 1]])
shear_linear_image = transformation(shear_matrix, gray_image)
shear_bilinear_image = bilinear_trans(shear_matrix, gray_image)
plt.imshow(shear_linear_image, cmap = 'gray')
plt.show()
plt.imshow(shear_bilinear_image, cmap = 'gray')
```

Result



Nearest Neighborhood



Bilinear Interpolation

After applying various geometric transformation through nearest neighborhood and Bilinear Interpolation we conclude that the image obtained after Bilinear Interpolation has a better quality as compared to one obtained after applying nearest neighborhood.
The results of nearest neighbors is that it is patchy and bilinear interpolation are not. The line drawn by bilinear is smooth as compared to nearest neighbors.

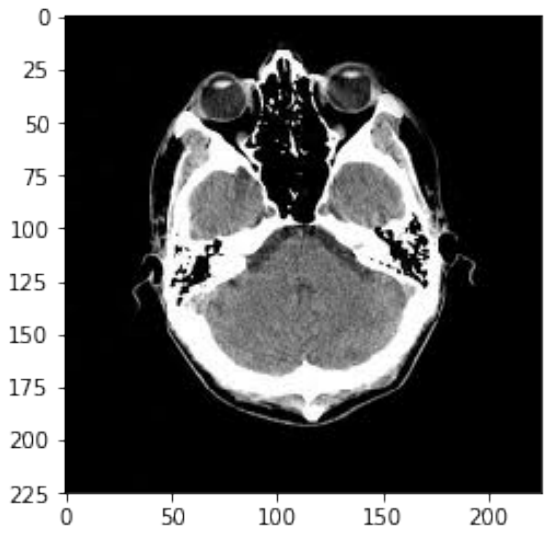
Nearest neighborhood.

Bilinear Interpolation

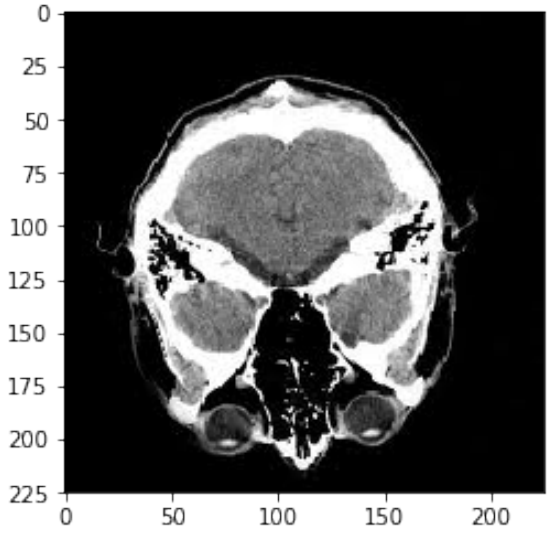
1.2 Calculation of affine transform from landmarks

As we know a transformation can be determined based on a set of corresponding landmarks in a source and a target image. Here we select three landmarks on an image with a mouse click on source image and again three point on the target image.

We take the following images as our source image and target image.



Source Image.



Target Image

We use the following code for selecting the landmarks and the finding the coordinates of the pixels on the landmarks

```
import cv2

src = []
tgt = []
def returnSourcePoints(event, x, y, flags, param):
    global src
    if event == cv2.EVENT_LBUTTONDOWN:
        print (x,y)
        #cv2.circle(src_img, (x, y), 200, (255, 0, 0), -1)
        src.append([x, y])
cv2.namedWindow('source-image', cv2.WINDOW_NORMAL)
cv2.startWindowThread()
cv2.setMouseCallback('source-image', returnSourcePoints)

cv2.imshow('source-image', src_img)
k = cv2.waitKey(0)
  # wait for ESC or space key to exit
cv2.waitKey(1)
cv2.destroyWindow('source-image')
cv2.waitKey(1)



def returnTargetPoints(event, x, y, flags, param):
    global tgt
    if event == cv2.EVENT_LBUTTONDOWN:
        print (x,y)
        #cv2.circle(tgt_img, (x, y), 100, (255, 0, 0), -1)
        tgt.append([x, y])
cv2.namedWindow('target-image', cv2.WINDOW_NORMAL)
cv2.startWindowThread()
cv2.setMouseCallback('target-image', returnTargetPoints)

cv2.imshow('target-image', tgt_img)
k = cv2.waitKey(0)
  # wait for ESC or space key to exit
cv2.waitKey(1)
cv2.destroyWindow('target-image')
cv2.waitKey(1)
```

$$\underbrace{\begin{bmatrix} x_1' \\ y_1' \\ x_2' \\ y_2' \\ x_3' \\ y_3' \end{bmatrix}}_{\mathbf{x}'} = \underbrace{\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{bmatrix}}_{\mathbf{X}} \underbrace{\begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \end{bmatrix}}_{\mathbf{a}}$$

48

This gives the following solution:

$$\mathbf{X}^{-1}\mathbf{x}' = \mathbf{a}$$

These are the images from the professors slides.

We calculate the six parameters from the input image and six from the target image. Then we set up a linear system of equation and the code to solve it as below.

```
def gauss(A):
    n = len(A)

    for i in range(0, n):
        # Search for maximum in this column
        maxEl = abs(A[i][i])
        maxRow = i
        for k in range(i+1, n):
            if abs(A[k][i]) > maxEl:
                maxEl = abs(A[k][i])
                maxRow = k

        # Swap maximum row with current row (column by column)
        for k in range(i, n+1):
            tmp = A[maxRow][k]
            A[maxRow][k] = A[i][k]
            A[i][k] = tmp

        # Make all rows below this one 0 in current column
        for k in range(i+1, n):
            c = -A[k][i]/A[i][i]
            for j in range(i, n+1):
                if i == j:
                    A[k][j] = 0
                else:
                    A[k][j] += c * A[i][j]

    # Solve equation Ax=b for an upper triangular matrix A
    x = [0 for i in range(n)]
    for i in range(n-1, -1, -1):
        x[i] = A[i][n]/A[i][i]
        for k in range(i-1, -1, -1):
            A[k][n] -= A[k][i] * x[i]
    return x
```

And the results have been verified using this

The affine parameters received are as follows for the above image.

```
[[ -1.03877206e+00]
 [ -1.59594442e-03]
 [  2.31072193e+02]
 [  3.02290650e-02]
 [ -1.10045062e+00]
 [  2.24367443e+02]]
```

And the affine matrix defined using the above parameters is as follows:

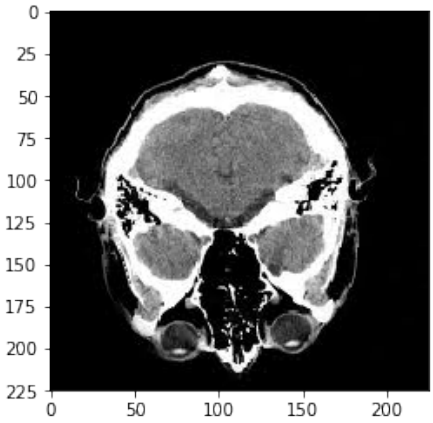
```
[[ -1.03877206e+00  -1.59594442e-03   2.31072193e+02]
 [  3.02290650e-02  -1.10045062e+00   2.24367443e+02]
 [  0.00000000e+00   0.00000000e+00   1.00000000e+00]]
```

Now, using the below code for general affine transformation given the affine matrix is as follows.

```
def affineTransformation(image,affine_matrix):
    newImage = np.zeros_like(image)
    #scale = np.array([[sx,0,0],[0,sy,0],[0,0,1]])
    inv_affine_matrix = np.linalg.inv(affine_matrix)
    for x in range(newImage.shape[0]):
        for y in range(newImage.shape[1]):
            point = np.array([[x],[y],[1]])
            new_point = np.matmul(inv_affine_matrix,point)
            a = int(new_point[0][0])
            b = int(new_point[1][0])
            #check if new image coordinates are present in the image
            #if a < image.shape[0] and b < image.shape[1]:
            if a > 0 and a < image.shape[0] and b > 0 and b < image.shape[1]:
                newImage[x][y] = image[a][b]
    return newImage
```

This function applies the transformation to the image and returns the new image.

The result of applying the above given affine matrix to the CTScan.jpg image is as follows:

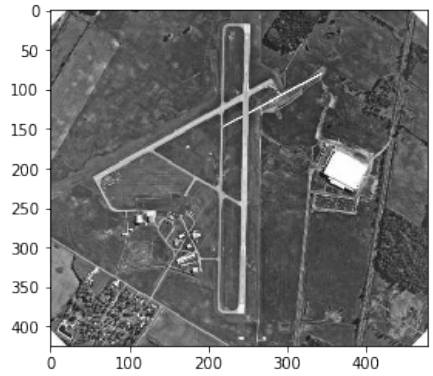


We can see that the image is rotated.

HOUGH TRANSFORM

Here I present the report for hough transform.
The intial steps carried out are the ones from the previous assignment and much of the code from the previous assignment has been used.

The images used for line detection are:

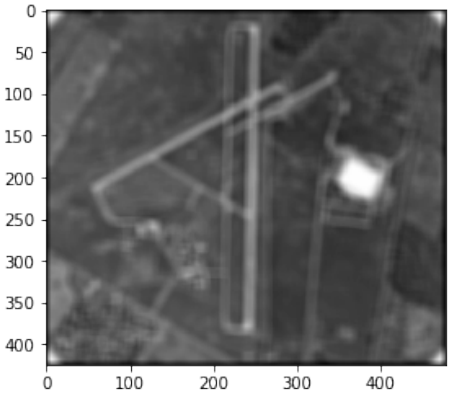
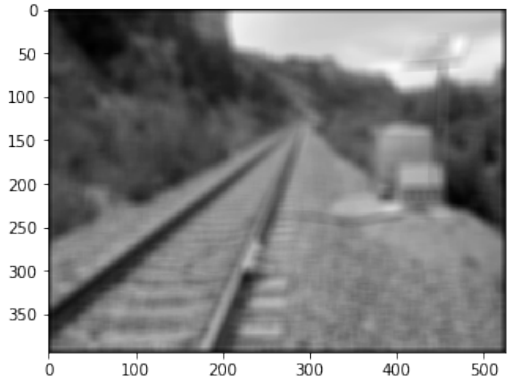



I detect lines in these two images.

The process is as follows:

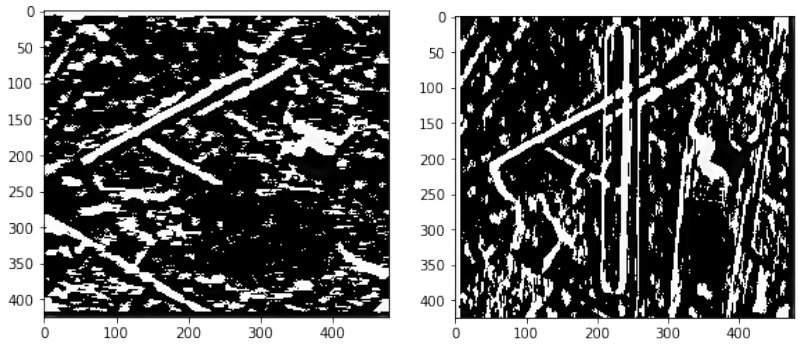
1. First I smooth the image and find the x and y derivative of the image
2. Then, using the x and y derivatives, I find the edge map

Results are presented below.

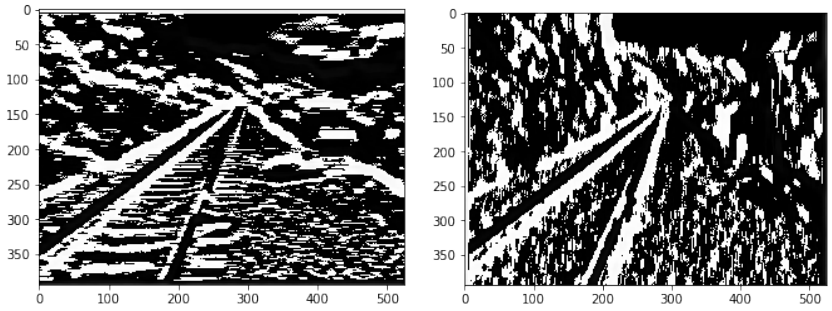



Both the images are blurrred using Box filter of 13X13.

The derivatives of the above are presented as follows:

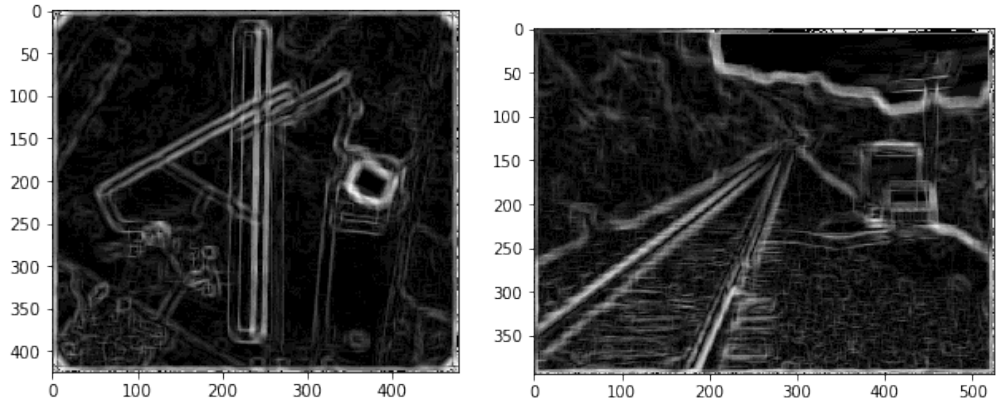


Runway Derivatives

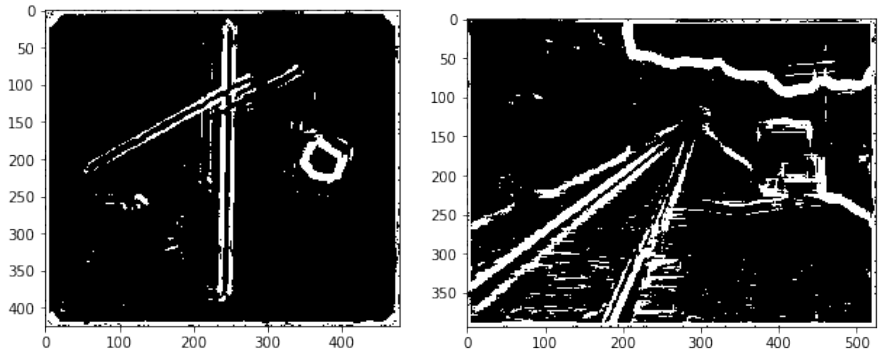


RailTrack Derivatives

EdgeMaps:



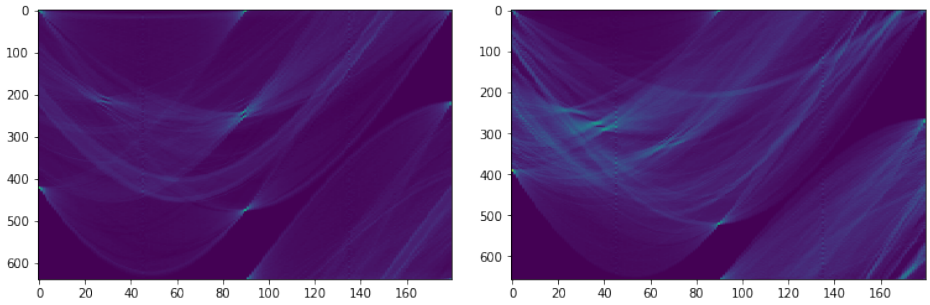
These are the edgemaps of the two images. We apply thresholding to the above images and the results are presented below:



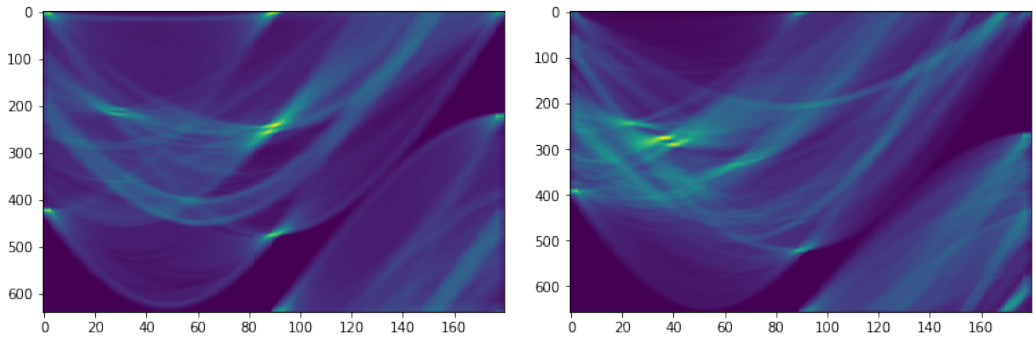
Images after thresholding of the edgemaps.

Now using these images, we calculate the hough transform of the images.
Code is presented below:

```
def houghTransform(image):
    diag = math.sqrt(math.pow(image.shape[0],2) + math.pow(image.shape[1],2))
    diag = int(diag)
    accumulator = np.zeros((diag,180))
    rho = 0
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            if image[i][j] != 0:
                for theta in range(0,180):
                    rho = int(i * math.cos(math.radians(theta)) + j* math.sin(math.radians(theta)))
                    accumulator[rho][theta] = accumulator[rho][theta] + 1
    return accumulator
```



Hough of the two images.
After this we smooth these hough transforms using 5X5 Box filter.
Results are presented below.

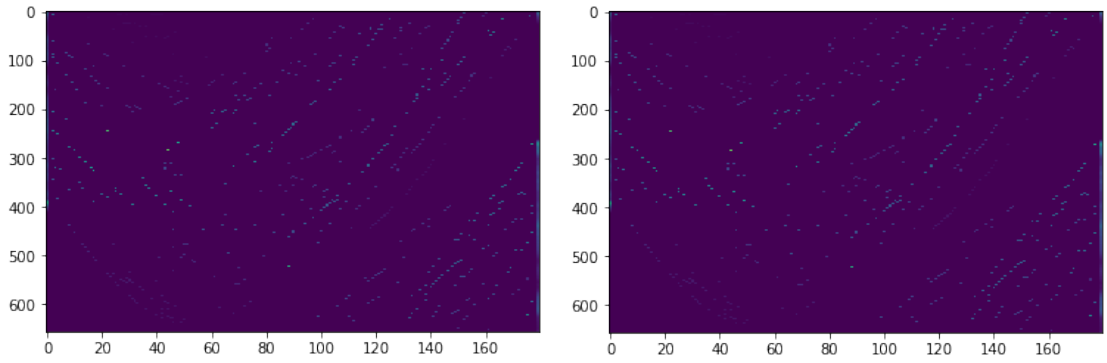


After this smoothing, we apply non maximum separation to get clear edges.

The code is presented below:

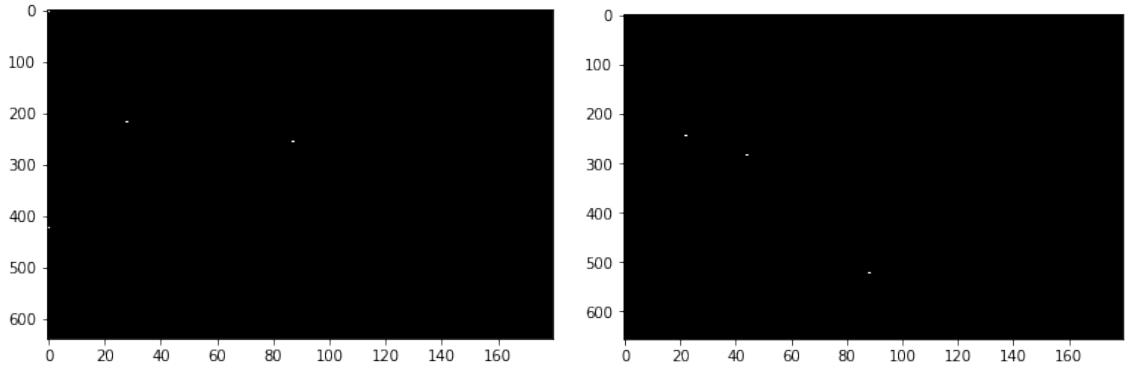
```
def nonMaximumSeparation(accumulator):
    indices = []
    output = np.zeros_like(accumulator)
    for i in range(1,accumulator.shape[0]-1):
        for j in range(1,accumulator.shape[1]-1):
            # if i am maximum in neighborhood, keep me, else make me 0
            me, up , down , left , right = accumulator[i][j] ,accumulator[i-1][j] \
            , accumulator[i+1][j] , accumulator[i][j-1], accumulator[i][j+1]
            upleft, upright, bottomleft, bottomright = accumulator[i-1][j-1] \
            , accumulator[i-1][j+1],accumulator[i+1][j-1],accumulator[i+1][j+1]
            #print up,down,left,right
            if me >= up and me >= down and me>=left and me>=right and me >=upleft and me>=upright \
            and me>=bottomleft and me>=bottomright:
                continue
                #output[i][j] = me
            else:
                indices.append((i,j))
                #output[i][j] = 0
    for point in indices:
        accumulator[point[0]][point[1]] = 0
    return accumulator
```

Using the above code, we get the following results:



These two are the results of applying non maximum separation in the two hough transforms that we get.

On these images, we apply thresholding to find the peaks.



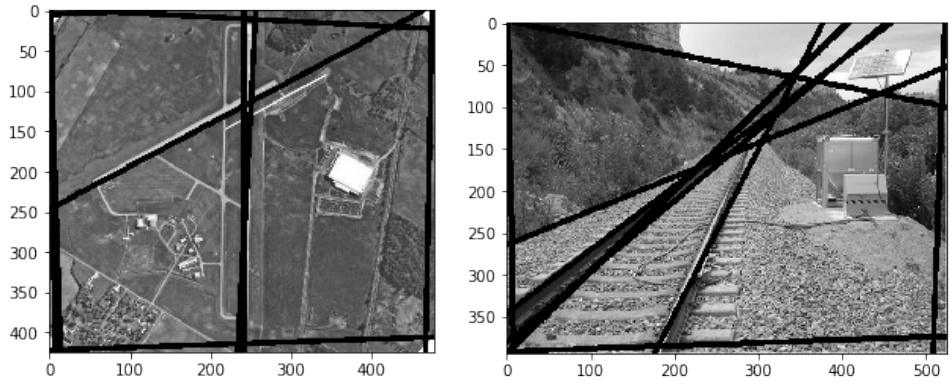
We can see the peaks being detected in the images.
Code to find the peaks in the maximally separated:

```
def findPeaksIndices(accumulator):
    output = []
    for i in range(accumulator.shape[0]):
        for j in range(accumulator.shape[1]):
            if accumulator[i][j] != 0 :
                output.append((i,j))
    return output
```

Now, after we get these peaks, we use them to plot the hough lines detected over to the original image. Code is presented below:

```
def houghLines(peaklist,image):
    for point in peaklist:
        rho = point[0]
        theta = point[1]
        sintheta = math.sin(math.radians(theta))
        costheta = math.cos(math.radians(theta))
        if theta != 0:
            y1 = int(rho/sintheta)
            y2 = int((rho - image.shape[0]*costheta)/sintheta)
            cv2.line(image,(y1,0),(y2,image.shape[0]),(0,255,0),5)
```

After plotting these lines, the results we get are as follows:



We can see that the runways and tracks in the two images have been detected.
Note that in the second image, mountain range has also been detected as edge. This is expected, because if we look at the edgemap of the railway tracks image, the mountain makes high threshold and thus we get a peak for that line in the hough space.

There were many challenges I faced in this project. The part to detect the besrt threshold has been challenging. We have to set the threshold for both edge map and hough image (after non maximum separation) and see what edges we get.
Also, figuring out the hough transform code has been challenging along with peak detection.
Overall, the project was very challenging and interesting.