

ASSIGNMENT 2

COMPUTER VISION _ SPRING 2019

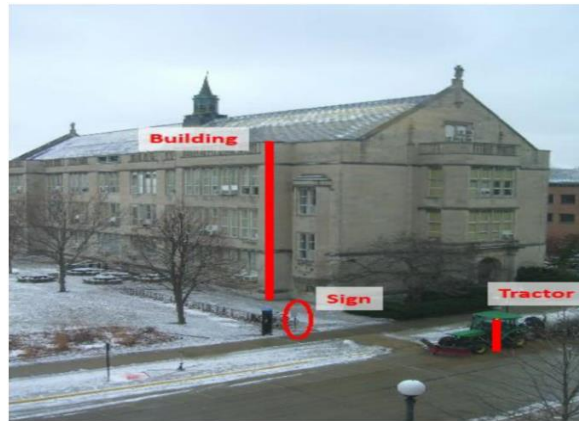
B S N V Chaitanya(S20160020115) D Sumanth(S20160020125)

K Anjali Poornima(S20160020132)

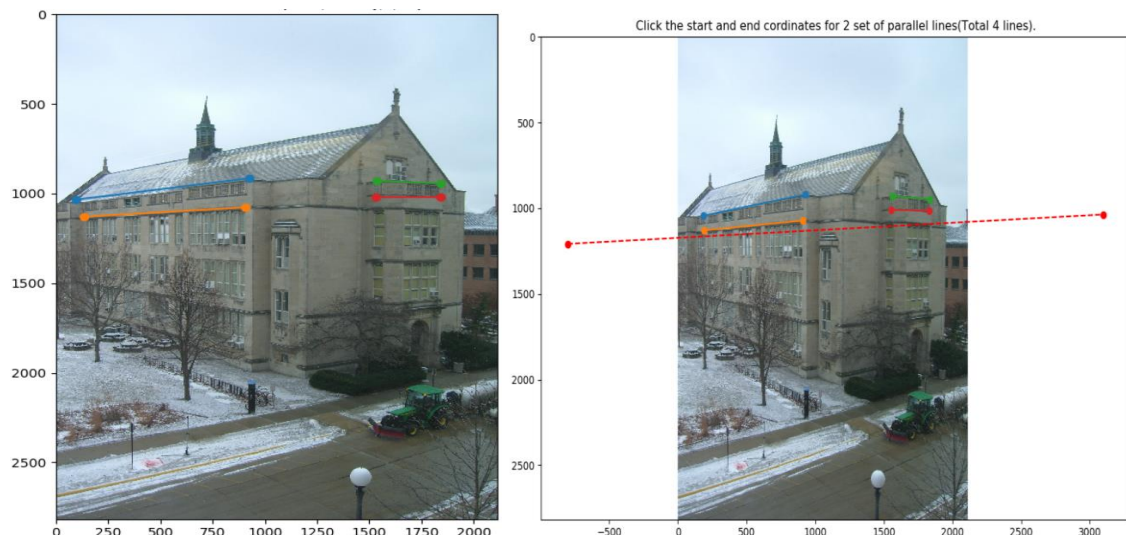
Question 1 : Single-View Metrology

Analysing the steps to calculate the height of an object using reference object:

1. Consider the given image. Given that the sign is 1.65 meter. The objective is to estimate the height of building, tractor and the camera.



2. Initially, the user is asked to enter the number of objects he wanted to estimate height. The 2 sets of parallel lines are to be drawn, to calculate the vanishing points and corresponding vanishing line.

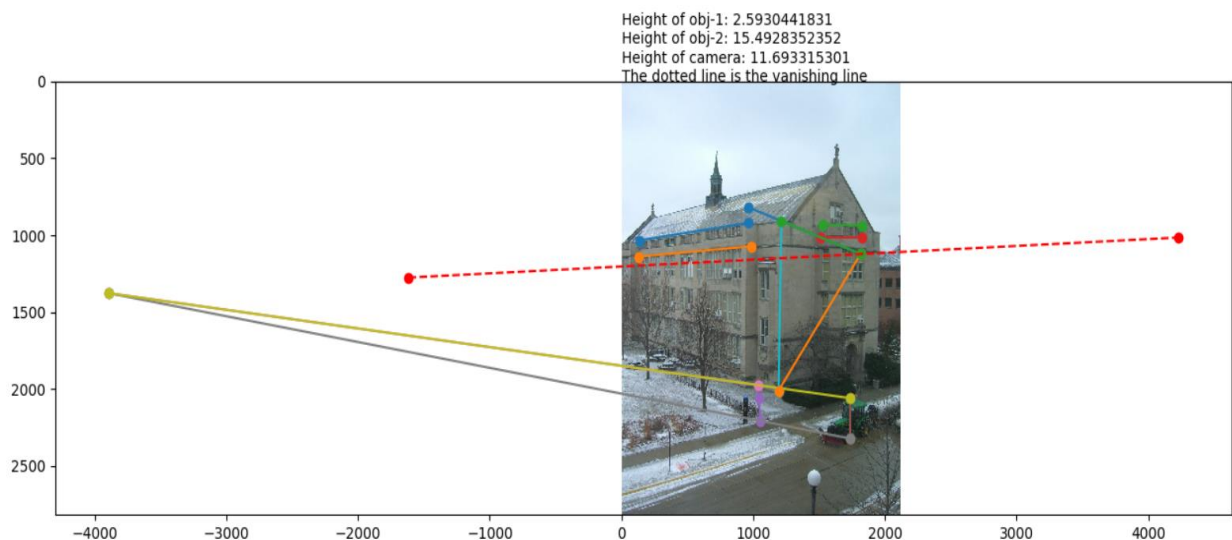


- Next, the points are taken for reference object. In this case, it is a sign board of length 1.65 meters.
- Then, the user is asked to draw the end points of the objects and the height is calculated using cross-ratio, by equating the cross ratio in image and scene cross ratios ($(|b - t| |V_z - r|) / (|b - r| |V_z - t|) = H/R$). This is done by getting the line equations and finding the intersection points.
- Below are the illustrations that shows the horizon line, and the lines and measurements used to estimate the heights of the building, tractor, and camera(all measurements are in meters).

Object 1 - Tractor

Object 2 - Building

```
Enter the number of objects for which you want to get the height: 2
The first line equation is y = -0.137931034483*x + 1052.514059
The second line equation is y = -0.0782122905028*x + 1149.01936682
-----
The first line equation is y = 0.031746031746*x + 880.628196259
The second line equation is y = -0.0*x + 1014.91739616
-----
Height of the obj1 : 2.5930441831
Height of the obj2 : 15.4928352352
-----
Height of camera : 11.693315301
```



Code:

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

def getIntersection(l1, l2, toPrint = False):
    def lineParams(p1, p2):
        if(p1[0] == p2[0]):
            raise("You can't take lines parallel to the image plane")
        m = (p1[1] - p2[1]) / (p1[0] - p2[0])
        c = p1[1] - m * p1[0]
        return [m, c]
    def lines(params):
        return "y = " + str(params[0]) + "*x + " + str(params[1])
    def intersectionPoints(par1, par2):
        if(par1[0] == par2[0]):
            return None
        x = (par2[1] - par1[1]) / (par1[0] - par2[0])
        y = par1[0] * x + par1[1]
        return [x, y]
```

```

        raise("the lines shouldn't be parallel!!!")
        x = (par1[1] - par2[1])/(par2[0] - par1[0])
        y = par1[0]*x + par1[1]
        return (x, y)
    params1 = lineParams(l1[0], l1[1])
    params2 = lineParams(l2[0], l2[1])
    if(toPrint):
        print("The first line equation is " + lines(params1))
        print("The second line equation is " + lines(params2))
        print("-----")
    return intersectionPoints(params1, params2)

def getVanishingPoints(nLines = 4):
    plt.title("Click the start and end coordinates for "+str(nLines//2)+" set of parallel lines(Total " +str(nLines)+" lines).")
    points = plt.ginput(nLines*2)
    vanishingPoints = []
    for i in range(0, len(points), nLines):
        A1 = points[i]
        A2 = points[i+1]
        B1 = points[i+2]
        B2 = points[i+3]
        plt.plot([A1[0], A2[0]], [A1[1], A2[1]], marker = 'o')
        plt.plot([B1[0], B2[0]], [B1[1], B2[1]], marker = 'o')
        vanishingPoints.append(getInterSetion([A1, A2], [B1, B2], True))
    return vanishingPoints

def getPointsForObj():
    points = plt.ginput(n = 2, timeout = 0)
    top = points[0]
    bottom = points[1]
    plt.plot([top[0], bottom[0]], [top[1], bottom[1]], marker = 'o')
    return top, bottom

def getHeight(height, refTop, refBottom, heightPlt):
    return height * (np.linalg.norm(np.array(heightPlt)-np.array(refBottom))*1.0/np.linalg.norm(np.array(refTop)-
np.array(refBottom)))

image = scipy.misc.imread("img.jpg")
N = int(input("Enter the number of objects for which you want to get the height: "))
plt.imshow(image)
vanishingPoints = getVanishingPoints()
plt.imshow(image)
plt.title("Click the coordinates for the pole(ref obj) (Top - Bottom)")
pole_top, pole_bottom = getPointsForObj()
poleHeight = 1.65
heights = ""
for i in range(1, N+1):
    plt.title("Click the coordinates of the obj-"+str(i)+" for which height is to be calculated (Top - Bottom)")
    obj_top, obj_bottom = getPointsForObj()

    vIPoint = getInterSetion([pole_bottom, obj_bottom], [vanishingPoints[0], vanishingPoints[1]])
    toGetHeightPlt = getInterSetion([pole_top, pole_bottom], [obj_top, vIPoint])

    plt.plot([vIPoint[0], toGetHeightPlt[0]], [vIPoint[1], toGetHeightPlt[1]], marker = 'o')
    plt.plot([obj_bottom[0], vIPoint[0]], [obj_bottom[1], vIPoint[1]], marker = 'o')
    plt.plot([obj_top[0], vIPoint[0]], [obj_top[1], vIPoint[1]], marker = 'o')

    objHeight = getHeight(poleHeight, pole_top, pole_bottom, toGetHeightPlt)
    print("Height of the obj"+str(i)+" : " + str(objHeight))
    heights += "Height of obj-" + str(i) + ": " + str(objHeight) + "\n"

print("\n-----\n")
forCameraHeight = getInterSetion([pole_bottom, pole_top], [vanishingPoints[0], vanishingPoints[1]])
cameraHeight = getHeight(poleHeight, pole_top, pole_bottom, forCameraHeight)
print("Height of camera : " + str(cameraHeight))
heights += "Height of camera" + ": " + str(cameraHeight) + "\nThe dotted line is the vanishing line"
plt.title("")

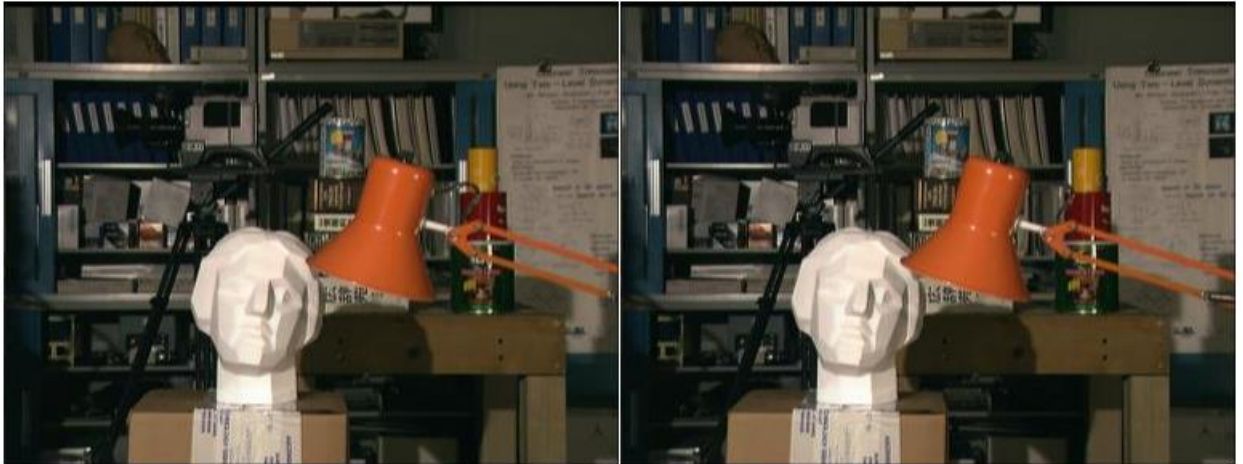
```

```
plt.text(0, 0, heights)
plt.plot([vanishingPoints[0][0], vanishingPoints[1][0]], [vanishingPoints[0][1], vanishingPoints[1][1]], color='red', marker
= 'o', linestyle='dashed')
plt.show()
```

Question 2 : Stereo Matching

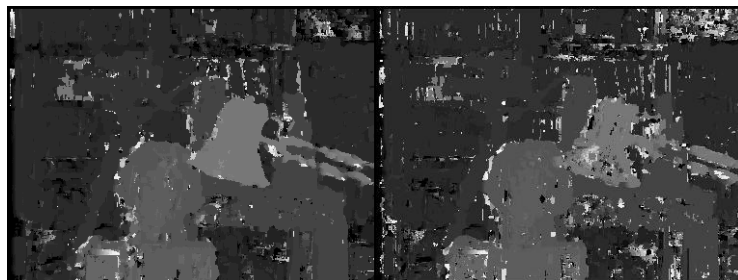
Analysing the steps to obtain Disparity Map:

1. The goal is to implement a simple window-based stereo matching algorithm for rectified stereo pairs. The given stereo pair:



2. Disparity range and window size are initialized. The disparity map is obtained using two matching techniques ; SSD and correlation.
3. For every pixel in the right image, we extract a block(according to window size) around it and search along the same row in the left image for the block that best matches it.
4. Here we search in a range (Disparity Range) of pixels around the pixel's location in the first image, and we use some matching technique(here, both SSD and correlation are considered) to compare the image regions.
5. We need only search over columns and not over rows because the images are rectified.
6. In case of SSD(Sum of Squared Distances) the least value is taken as best match. In case of Correlation, the one with highest value is taken as the best match.

(Disparity range = 30, window size = 7)



Output using SSD Function

Output using Correlation function

Code:

```
import numpy as np
import cv2
import time
from scipy import signal

def SSD(left, right, h, w, halfBlockSize, disparityRange):
    offset_adjust = 255 / disparityRange
    for y in range(halfBlockSize, h - halfBlockSize):
        if (y%10 == 0):
            print("Image row " + str(y) + "/" + str(h) + " ( " + str(int(y * 100/h)) + "% )\n")
        for x in range(halfBlockSize, w - halfBlockSize):
            best_offset = 0
            prev_ssd = 2147483647
            ssds = []
            numBlocks = 0
            for offset in range(disparityRange):
                block = left[y - halfBlockSize : y + halfBlockSize, x - halfBlockSize : x + halfBlockSize]
                template = right[y - halfBlockSize : y + halfBlockSize, x - halfBlockSize - offset : x + halfBlockSize - offset]
                if(template.shape != block.shape):
                    template = np.zeros(block.shape)
                numBlocks += 1
                ssdTemp = block - template
                ssdTemp = ssdTemp ** 2
                ssd = np.sum(ssdTemp)
                ssds.append(ssd)
                if ssd < prev_ssd:
                    prev_ssd = ssd
                    best_offset = offset
            depth[y, x] = best_offset * offset_adjust
    return depth

def Correlation(left, right, h, w, halfBlockSize, disparityRange):
    offset_adjust = 255 / disparityRange
    for y in range(halfBlockSize, h - halfBlockSize):
        if (y%10 == 0):
            print("Image row " + str(y) + "/" + str(h) + " ( " + str(int(y * 100/h)) + "% )\n")
        for x in range(halfBlockSize, w - halfBlockSize):
            best_offset = 0
            prev_corr = 0
            corrs = []
            numBlocks = 0
            for offset in range(disparityRange):
                temp = 0
                corr = 0
                block = left[y - halfBlockSize : y + halfBlockSize, x - halfBlockSize : x + halfBlockSize]
                template = right[y - halfBlockSize : y + halfBlockSize, x - halfBlockSize - offset : x + halfBlockSize - offset]
                for i in range(-halfBlockSize, halfBlockSize):
                    for j in range(-halfBlockSize, halfBlockSize):
                        temp = (int(left[y+i, x+j]) - np.mean(block)) * (int(right[y+i, x+j - offset]) - np.mean(template))
                        temp /= (np.std(block) * np.std(template))
                        corr += temp
                corr /= (2*halfBlockSize + 1)
                corrs.append(corr)
                if corr > prev_corr:
                    prev_corr = corr
                    best_offset = offset
            depth[y, x] = best_offset * offset_adjust
    return depth

left = cv2.imread("tsukuba1.ppm", 0)
right = cv2.imread("tsukuba2.ppm", 0)
h, w = left.shape
depth = np.zeros((h, w), np.uint8)
depth_corr = np.zeros((h, w), np.uint8)
disparityRange = 30
halfBlockSize = 3
```

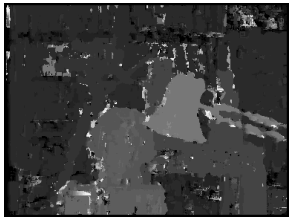
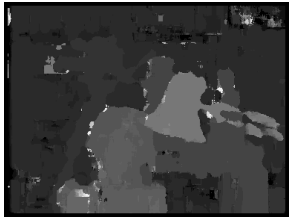
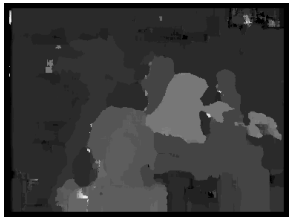
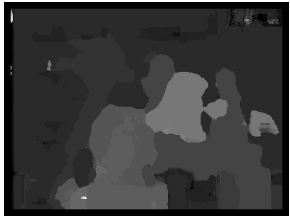
```

blockSize = 2 * halfBlockSize + 1
start_time_SSD = time.time()
depth = SSD(left, right, h, w, halfBlockSize, disparityRange)
cv2.imwrite("DepthMap_"+str(blockSize)+"_"+str(disparityRange)+"_ssd.png",depth)
time_SSD = time.time() - start_time_SSD
start_time_Corr = time.time()
depth_corr = Correlation(left, right, h, w, halfBlockSize, disparityRange)
cv2.imwrite("DepthMap_"+str(blockSize)+"_"+str(disparityRange)+"_corr.png",depth)
time_Corr = time.time() - start_time_Corr
print("Time taken for SSD is " + str(time_SSD))
print("Time taken for Correlation is " + str(time_Corr))

```

Observations:

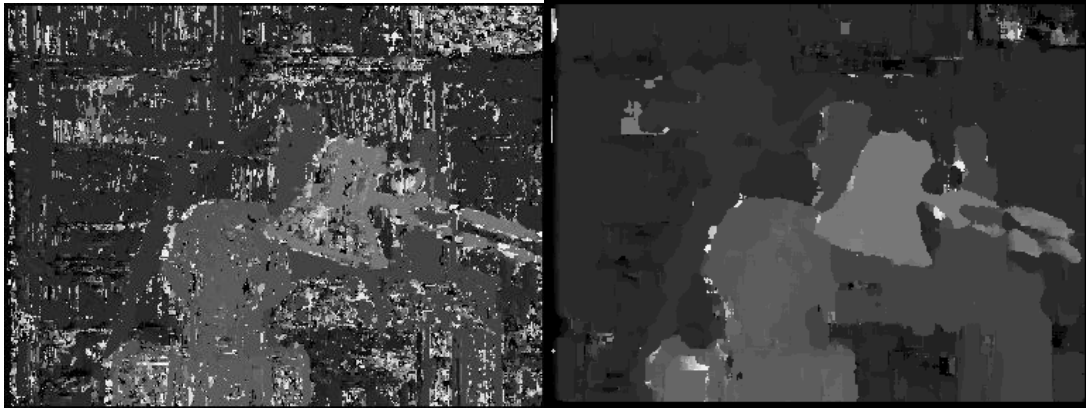
Search window size : As the search window increases the smoothness increased. But as the Search window increased, the edges were lost. The running time slightly increased when the search window increased while using SSD. But when using correlation, there was significant increase in the run time. Below is a table that illustrates different search window sizes for same disparity range:

Window Size	Time for SSD(sec)	Time - Correlation(sec)	Output for SSD
7	18.9359	246.6527	
11	19.1474	318.5333	
15	18.64	476.05	
19	28.599	938.035	

Disparity range: Disparity range is the range of the scanline in the second image that should be traversed in order to find a match for a given location in the first image. The observation is that, as the disparity range increased, the runtime increased. Below is the table that shows the runtime for SSD and Correlation for window size 3.

Disparity Range	SSD Time	Correlation Time
15	10.69126	124.266
30	19.7047	236.184
45	29.09067	361.8878

Matching Function: There are two matching functions used ; SSD (Sum of Squared Distances) and Correlation. SSD proved to perform better when compared with Correlation function both in terms of quality and Time. This time difference can be seen in all the above tables. The quality can be observed in the below images for disparity range of 30 and window size of 11



Output using normalized correlation

Output using SSD