Task05: Homography

Name: Noreen Gao

Degree: BA

ID: 20786725

Problem 1: Homogeneous Coordinate (Lines and Points) (10 points)

The intersection of two lines l1 and l2, with l1 passing through the points (8,4) and (1,7), and l2 passing through the points (6,3) and (-3,8).

(a) Compute the intersection without using homogeneous coordinates

Problem 1

(a) l_1: (8,4), (1,7) $\quad$ l_2: (6,3), (-3,8)

l_1: $y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$

$y - 4 = \frac{7-4}{1-8}(x-8)$

$y = -\frac{3}{7}(x-8) + 4$

$= -\frac{3}{7}x + \frac{24+28}{7}$

$= -\frac{3}{7}x + \frac{52}{7}$

l_2: $y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$

$y - 3 = \frac{8-3}{-3-6}(x-6)$

$y = -\frac{5}{9}(x-6) + 3$

$= -\frac{5}{9}x + \frac{30+27}{9}$

$= -\frac{5}{9}x + \frac{19}{3}$

Intersection: $-\frac{3}{7}x + \frac{52}{7} = -\frac{5}{9}x + \frac{19}{3}$

$\frac{5}{9}x - \frac{3}{7}x = \frac{19}{3} - \frac{52}{7}$

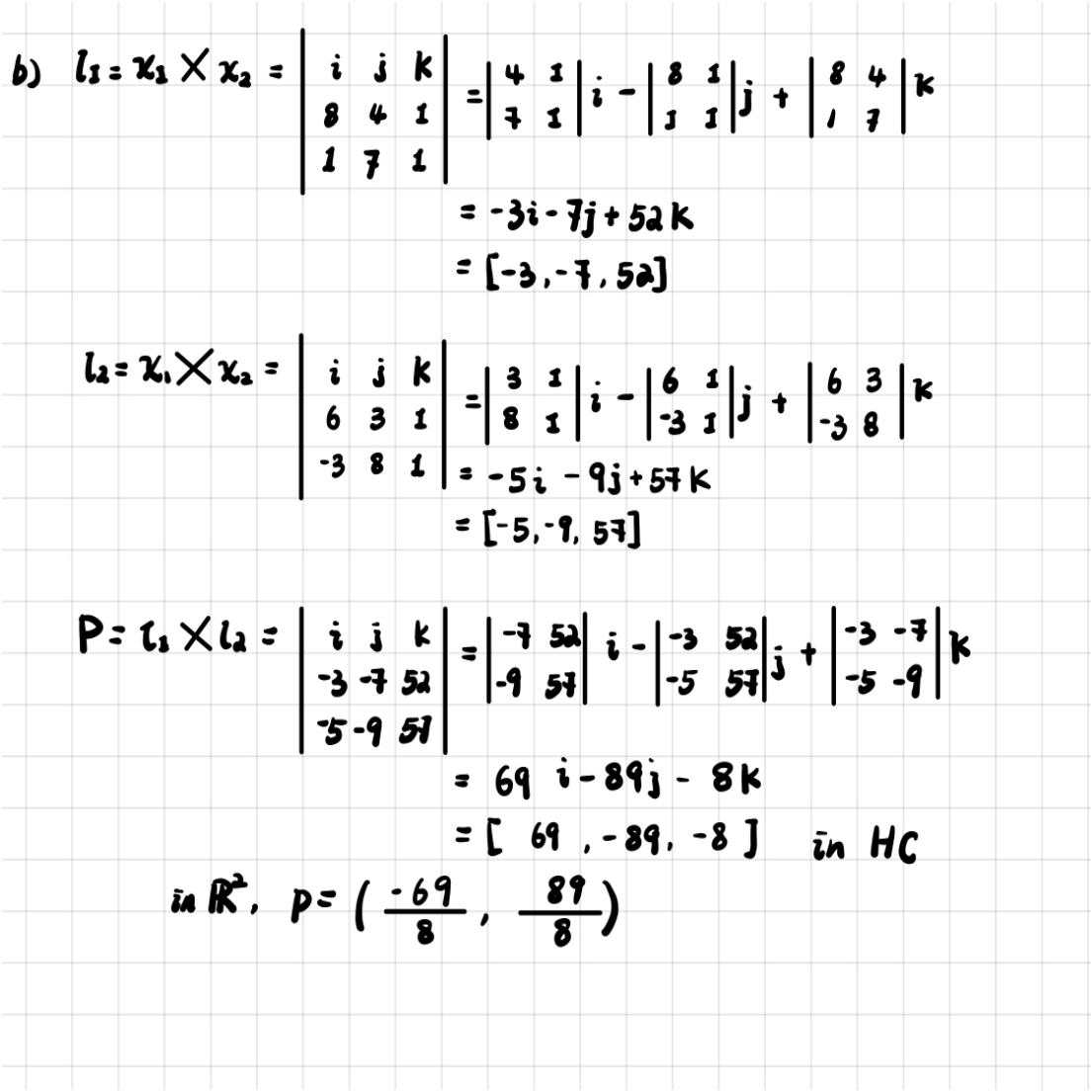
$\frac{8}{63}x = -\frac{23}{21}$

$x = -\frac{69}{8}$

$y = (-\frac{3}{7})(-\frac{69}{8}) + \frac{52}{7} = \frac{89}{8}$

$P(-\frac{69}{8}, \frac{89}{8})$

(b) Compute the intersection using homogeneous coordinates



Problem 2: Homogeneous Coordinate (Lines) (10 points)

The four lines (l1, l2, l3, and l4) are intersected at six points. Among the six points, please find four points that form a quadrangle with your code, not manually. Please explain your own algorithm to pick these four points.

$$
\begin{aligned}
l_1 &: y = 0.59756 \cdot x + 921.94 \\
l_2 &: y = -1.935 \cdot x + 6050.3 \\
l_3 &: y = 0.51575 \cdot x - 495.56 \\
l_4 &: y = -1.3858 \cdot x + 1685.5
\end{aligned}
$$

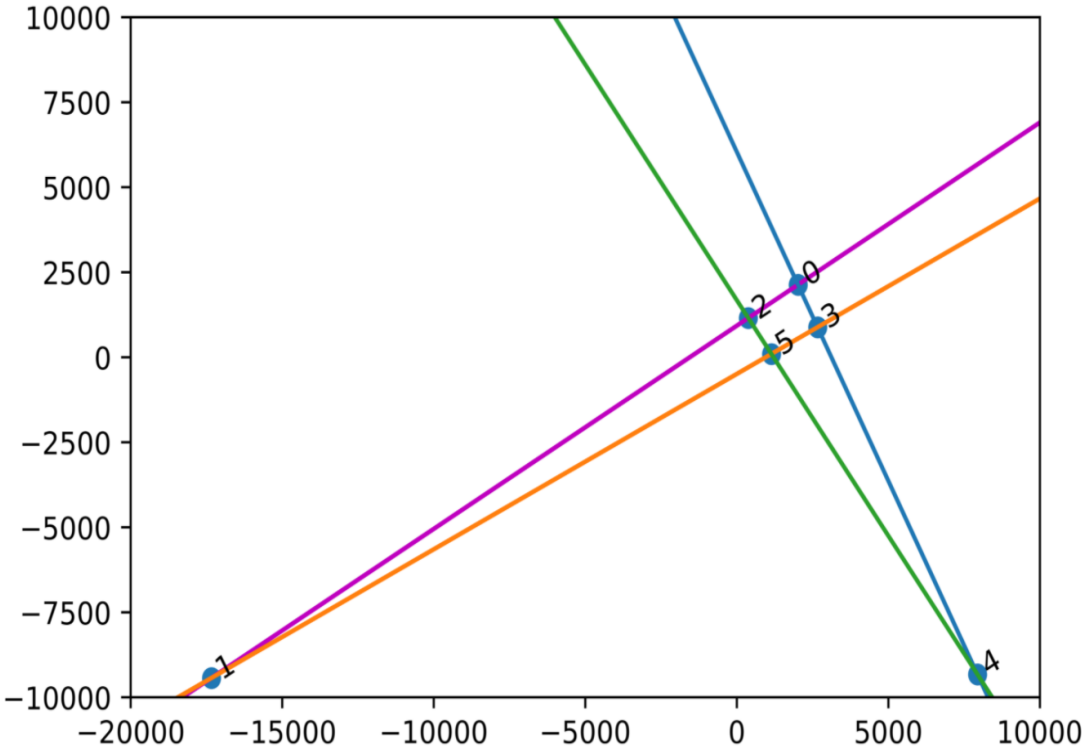
4 lines can form 6 intersection points, and there will be 15 combinations to select 4 points out of the 6. The fisrt step is to filter out the combinations containing collinear points, then only 3 combinations exist after this step. There are 3 different connection orders that will result in different shapes for each combination of 4 points. The second step is to filter out the connection orders that will make the edges of the quadrangle not on the lines. The thrid step is to filter out the complex and concave shapes of the quadrange.

[2024.970780554064, 2131.9815396278864]

[384.9830590513069, 1151.990476766699]

[1146.990612921038, 96.00040861402547]

[2670.9619504233397, 881.9886259308374]




```
plt.figure(dpi = 800)
# 4 lines in HC coordinate
l1 = np.array([0.59756, -1, 921.94])
l2 = np.array([-1.935, -1, 6050.3])
l3 = np.array([0.51575, -1, -495.56])
l4 = np.array([-1.3858, -1, 1685.5])
lines = [l1, l2, l3, l4]

intersection_pts = []

for ii in range(0,len(lines)):
    for jj in range(ii+1,len(lines)): # "ii+1": no repetition
        intersection_pts.append(np.cross(lines[ii], lines[jj]))

# select 4 points from 6 points: 15 index combinations
pts_combo_idx = list(it.combinations(range(len(intersection_pts)),4))

collinear_idx = []

for combo in pts_combo_idx:
    pt1 = intersection_pts[combo[0]]
    pt2 = intersection_pts[combo[1]]
    pt3 = intersection_pts[combo[2]]
    pt4 = intersection_pts[combo[3]]
    quad_points = [pt1, pt2, pt3, pt4]
```

```
    # select 3 points from 4 points: 4 combinations
    collin_combo = list(it.combinations(range(len(quad_points)),3))

    # check collinearity: any 2 lines formed by 3 points
    for collin_combo in collin_combo:
       line1 = np.cross(quad_points[collin_combo[0]],
quad_points[collin_combo[1]])
       line2 = np.cross(quad_points[collin_combo[0]],
quad_points[collin_combo[2]])

       # Collinear if the cross product of 2 lines is a zero vector
       if np.all(abs(np.cross(line1, line2)) < 1e-3):
          idx =  pts_combo_idx.index(combo) # record the idx of the 4-
points combination
          collinear_idx.append(idx)
          break

# combinations after filtering out the collinear points
pts_combo_idx = [pts_combo_idx[i] for i in range(len(pts_combo_idx)) if i
not in collinear_idx]


cn_orders = []
# 3 shape variations for 3 4-point combinations: 9 different variations
for connection orders
for combo in pts_combo_idx:
    order1 = [combo[0], combo[1], combo[2], combo[3], combo[0]]
    order2 = [combo[0], combo[1], combo[3], combo[2], combo[0]]
    order3 = [combo[0], combo[2], combo[1], combo[3], combo[0]]
    cn_orders.extend([order1,order2,order3])


# 1st Filtering: filter out all the connection orders not on the line
cn_order_on_line = [] # only keeps the connection orders on the line
for cn_order in cn_orders:

    four_edges_line = [] # a 1 x 4 logical vector for each cn_order's 4
edges

    for ii in range(len(cn_order)-1): # loop each edge
        edge =
np.cross(intersection_pts[cn_order[ii]],intersection_pts[cn_order[ii+1]])

        # check if each each edge lies on one of the 4 lines
        one_edge_four_lines= []
        for line in lines:
            if np.all(abs(np.cross(edge,line)) < 1e-3): # vector has to be
zero
               one_edge_four_lines.append(True)
            else:
                one_edge_four_lines.append(False)

        if np.any(one_edge_four_lines) == True: # one edge on any of the 4
```

```
lines is okay
            four_edges_line.append(True)
        else:
            four_edges_line.append(False)

    if np.all(four_edges_line) == True: # valid if all the edges are on
lines
        cn_order_on_line.append(cn_order)

# create a list of intersections points in Cartesian coordinate
intersection_pts_CT = []
for each in intersection_pts:
    intersection_pts_CT.append([each[0]/each[2],each[1]/each[2]])

# 2nd Filtering: filter out the complex and concave shapes
valid_cn_order = []
for each in cn_order_on_line:
    y_coord_cn_order = [intersection_pts_CT[i][1] for i in
range(len(intersection_pts)) if i in each]
    y_coord_OTHER = [intersection_pts_CT[i][1] for i in
range(len(intersection_pts)) if i not in each]
    # y-coord of the points other than the vertices should be either
larger or smaller than the max or min
    # of the y-coord of the 4 vertices
    if np.all(y_coord_OTHER > max(y_coord_cn_order)) or
np.all(y_coord_OTHER < min(y_coord_cn_order)) or\
        (np.any(y_coord_OTHER >max(y_coord_cn_order)) and
np.any(y_coord_OTHER < min(y_coord_cn_order))):
        valid_cn_order.append(each)

# printing the coordinates of valid 4 points in cartesian coordinates
valid_points_indx = valid_cn_order[0][0:4]
for each in valid_points_indx:
    print(intersection_pts_CT[each])

# plotting
x = np.arange(-20000,10000)
y1 = 0.59756*x + 921.94
y2 = -1.935*x + 6050.3
y3 = 0.51575*x - 495.56
y4 = -1.3858*x + 1685.5

plt.plot(x, y1,'m')
plt.plot(x, y2)
plt.plot(x, y3)
plt.plot(x, y4)

x_intersection_pts_CT = []
y_intersection_pts_CT = []
for inter_pt in intersection_pts_CT:
    x_intersection_pts_CT.append(inter_pt[0])
    y_intersection_pts_CT.append(inter_pt[1])

plt.scatter(x_intersection_pts_CT, y_intersection_pts_CT)
```

```
for i in range(6):
    plt.annotate(str(i), (x_intersection_pts_CT[i],
y_intersection_pts_CT[i]), color = 'k',rotation = 30)

plt.xlim([-20000, 10000]), plt.ylim([-10000, 10000])
plt.show()
```

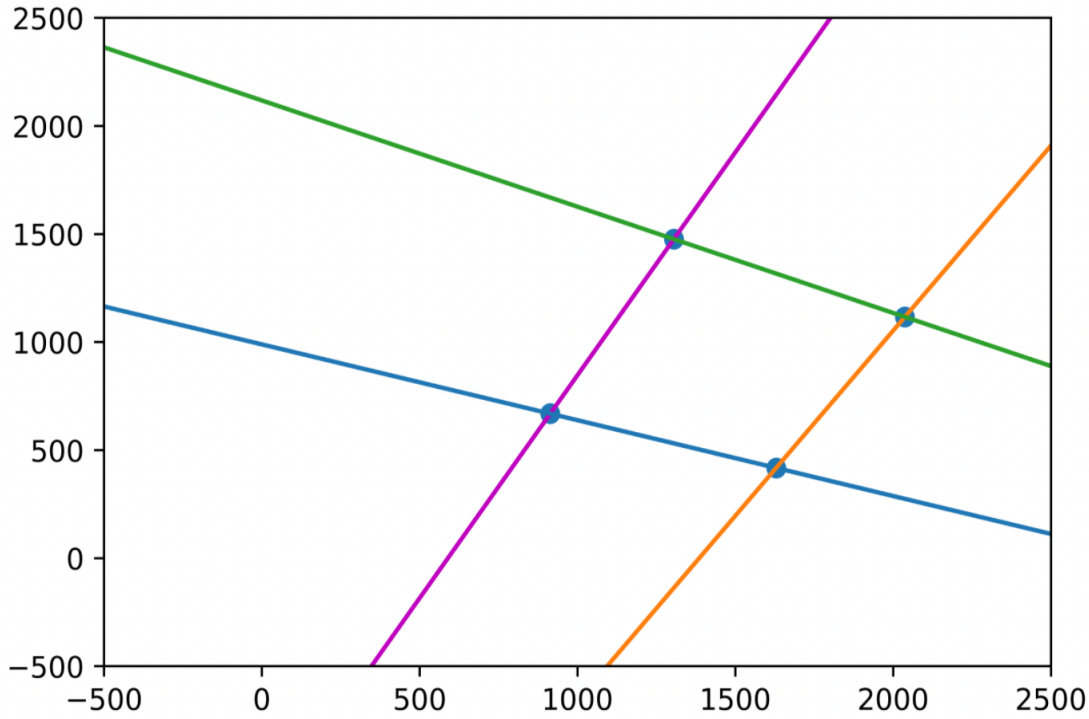
Problem 3: Homography (10 points)

Image 1 and Image 2 capture a flat rectangular table from different viewpoints. The pixel locations of the four corners of the table on Image 1 are the locations computed in Problem 2. When the perspective transformation from the points in image 1 to the ones in image 2 is

$$\mathbf{H} = \begin{bmatrix} 0.44973 & -0.3531 & 1147.5 \\ 0.2448 & 0.41441 & 96.5 \\ 0 & 0 & 1 \end{bmatrix}$$

Please find a line equation for each side of the table in Image 2 (a total of four sides). The line equation should be the form of y = a•x + b.

```
y1 =  2.062713582356457 X -1216.9130600164701
y2 =  -0.35087298842397036 X +  988.7927326123307
y3 =  1.7133780275236765 X -2374.7770245527804
y4 =  -0.4916980772362409 X +  2117.5838050649595

[<matplotlib.lines.Line2D at 0x147cdc520>]
```



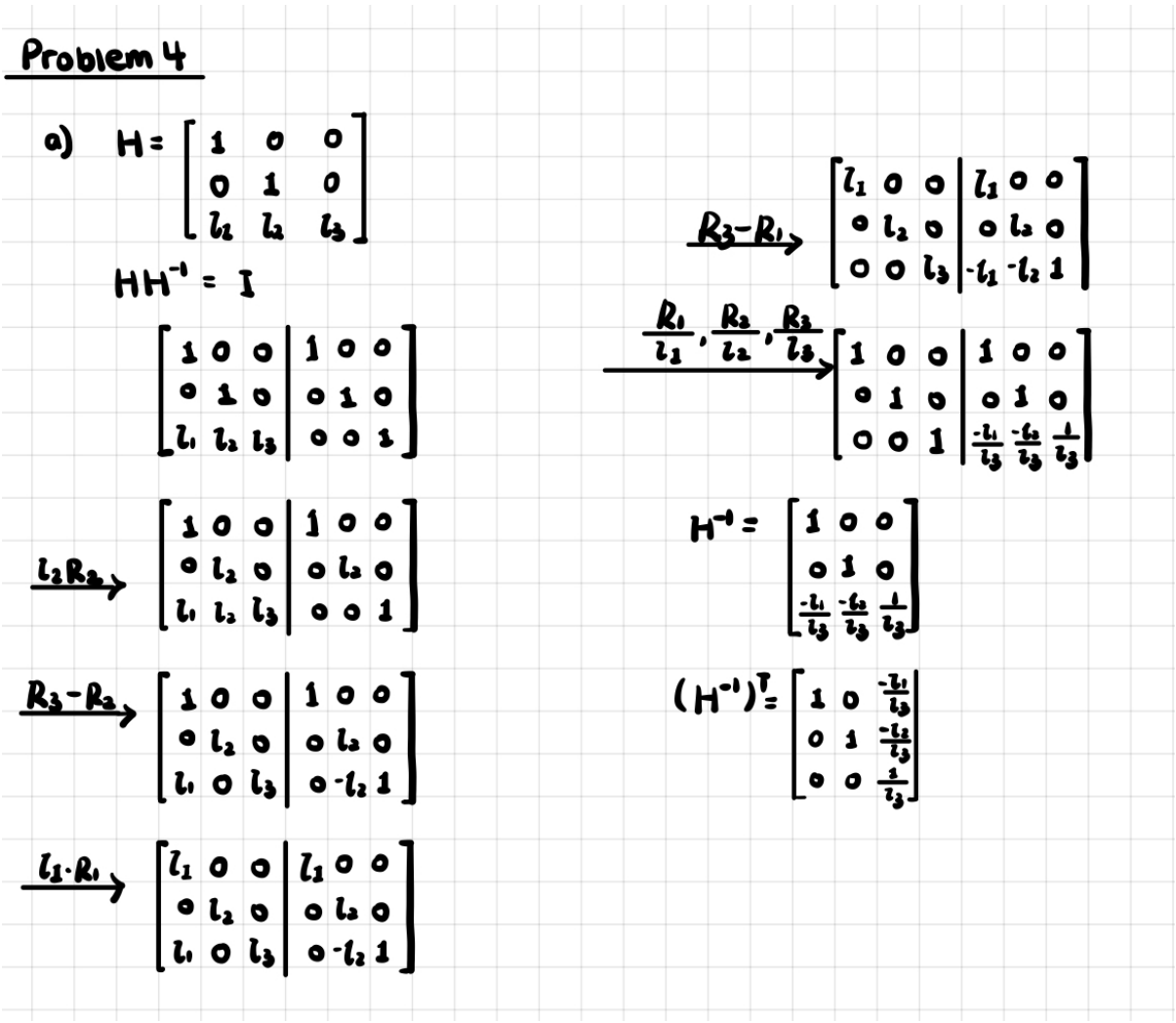
```
#### Q3
# The coordinates of valid 4 points in cartesian coordinates
valid_points_CT = []
for n in range(0,4):
    valid_points_CT.append(intersection_pts_CT[valid_cn_order[0][n]])

Img1_pts = valid_points_CT # four valid points from Problem 2
H = np.array([[0.44973, -0.3531, 1147.5], [0.2448, 0.41441, 96.5],[0, 0,
1]])

# Transformation of the 4 points on Image 2
Img2_pts = []
for n in range(0,4):
    Img2_pts.append(np.matmul(H,np.transpose(Img1_pts[n] + [1]))) #
concatenate

# Plot the transformed points on Image 2
xpts_img2 = []
ypts_img2 = []
for pt in Img2_pts:
    xpts_img2.append(pt[0])
    ypts_img2.append(pt[1])

plt.scatter(xpts_img2, ypts_img2)

# 4 lines in Img2
Img2_line1 = np.cross(Img2_pts[0], Img2_pts[1])
Img2_line2 = np.cross(Img2_pts[1], Img2_pts[2])
Img2_line3 = np.cross(Img2_pts[2], Img2_pts[3])
Img2_line4 = np.cross(Img2_pts[3], Img2_pts[0])

# print equations for 4 lines
print('y1 = ',Img2_line1[0]/(-Img2_line1[1]),'X', Img2_line1[2]/(-
Img2_line1[1]))
print('y2 = ',Img2_line2[0]/(-Img2_line2[1]),'X + ',  Img2_line2[2]/(-
Img2_line2[1]))
print('y3 = ',Img2_line3[0]/(-Img2_line3[1]),'X', Img2_line3[2]/(-
Img2_line3[1]))
print('y4 = ',Img2_line4[0]/(-Img2_line4[1]),'X + ', Img2_line4[2]/(-
Img2_line4[1]))

# Plot 4 lines in Img2
x = np.arange(-20000,10000)
y1 = (Img2_line1[0]*x + Img2_line1[2])/(-Img2_line1[1])
y2 = (Img2_line2[0]*x + Img2_line2[2])/(-Img2_line2[1])
y3 = (Img2_line3[0]*x + Img2_line3[2])/(-Img2_line3[1])
y4 = (Img2_line4[0]*x + Img2_line4[2])/(-Img2_line4[1])

plt.xlim([-500, 2500]), plt.ylim([-500, 2500])
plt.plot(x, y1,'m')
plt.plot(x, y2)
plt.plot(x, y3)
plt.plot(x, y4)
```

Problem 4: Linear Algebra (20 points)

Note that you should solve (a) and (b) by hand and (c) with a programming script.

(a) Find a transpose of inverse H (include your derivation)

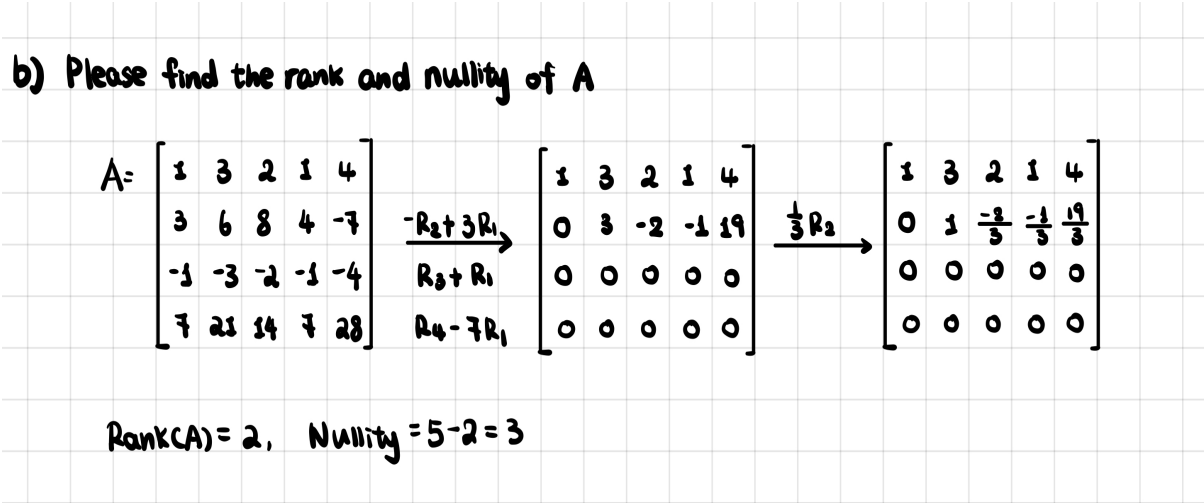
$$H = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_1 & l_2 & l_3 \end{bmatrix}$$



(b) What is the definition of a row and null space? Please find the rank and nullity of A.

$$\mathrm{A} = \begin{bmatrix} 1 & 3 & 2 & 1 & 4 \\ 3 & 6 & 8 & 4 & -7 \\ -1 & -3 & -2 & -1 & -4 \\ 7 & 21 & 14 & 7 & 28 \end{bmatrix}$$

The row space of A, denoted by row(A), is the subspace of Rn spanned by the row vectors of A. The null space of A, denoted by null(A), is the solution space of Ax = 0. This is a subspace of Rn.



(c) You need to find a non-singular (non-trivial) vector of x that satisfy Ax = 0. Does x exist? What is the null space of A and the nullity of A? Please explain your answer. Please answer these questions for each A below.

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 2 & 1 \\ 3 & 6 & 8 & 4 \\ 7 & 21 & 14 & 7 \\ 1 & 21 & 14 & -3 \\ -1 & 2 & 3 & 7 \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 2 & 1 \\ 3 & 6 & 8 & 4 \\ 1 & 21 & 14 & -3 \\ -1 & 2 & 3 & 7 \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 2 & 1 \\ 3 & 6 & 8 & 4 \\ 1 & 21 & 14 & -3 \end{bmatrix}$$

The number of the columns of A is sum of rank of A and the nullity of A. The nullspcae of A is the solution space of Ax = 0, and the dimention of the null space of A is called the nullity of A. X exists when the nullity is >= 1.

1)

```
A1:  [[ 1  3  2  1]
 [ 3  6  8  4]
 [ 7 21 14  7]
 [ 1 21 14 -3]
 [-1  2  3  7]]
nullspace of A1:  []
nullity of A1:  0
X does not exist
```

```
A1 = np.array([[1, 3, 2, 1], [3, 6, 8, 4],[7, 21, 14, 7], [1, 21, 14, -3],
[-1, 2, 3, 7]])
nullspace = null_space(A1)
rank = matrix_rank(A1)
nullity = A1.shape[1] - rank # number of cols - rank

print('A1: ', A1)
print('nullspace of A1: ' , nullspace)
print('nullity of A1: ', nullity)

if nullity <= 0:
    print('X does not exist')
else:
    print('X exists')
```

2)

```
A2:  [[ 1  3  2  1]
 [ 3  6  8  4]
 [ 1 21 14 -3]
 [-1  2  3  7]]
nullspace of A2:  []
nullity of A2:  0
X does not exist
```

```
A2 = np.array([[1, 3, 2, 1], [3, 6, 8, 4], [1, 21, 14, -3],[-1, 2, 3, 7]])
nullspace = null_space(A2)
rank = matrix_rank(A2)
nullity = A2.shape[1] - rank # number of cols - rank

print('A2: ', A2)
print('nullspace of A2: ' , nullspace)
print('nullity of A2: ', nullity)

if nullity <= 0:
    print('X does not exist')
else:
    print('X exists')
```

3)

```
A3:  [[ 1  3  2  1]
 [ 3  6  8  4]
 [ 1 21 14 -3]]
nullspace of A3:  [[-0.84810419]
 [ 0.1413507 ]
 [-0.04240521]
 [ 0.50886251]]
nullity of A3:  1
X exists
```

```
A3 = np.array([[1, 3, 2, 1], [3, 6, 8, 4], [1, 21, 14, -3]])
nullspace = null_space(A3)
rank = matrix_rank(A3)
nullity = A3.shape[1] - rank # number of cols - rank

print('A3: ', A3)
print('nullspace of A3: ' , nullspace)
print('nullity of A3: ', nullity)

if nullity <= 0:
    print('X does not exist')
else:
    print('X exists')
```

Problem 5: Image Overlay (10 points)

The code below shows an example of how to get 4 corners in the base image which bounds the area you want to project your picture onto. The process of getting the four corners is identical, so it's only shown once here. The order of how you select these 4 corner points does not matter because the 'sort_pts()' function will antomatically sort the selected 4 points from top-left to top-right to bottom-right to bottom-left. Once you load your base image, it allows you to select 4 points randomly on the image. The selected points are marked as red dots as shown below:

Then, the location of the selected 4 corners are printed out.

```
Selected Corners:  [[107. 362.]
 [653. 429.]
 [652. 801.]
 [ 99. 908.]]
```

```
# Obtained and Modified from https://pub.towardsai.net/what-is-
perspective-warping-opencv-and-python-750e7a13d386
# Display the base image and manually choose 4 corners using mouse clicks
def click_event(event, x, y, flags, params):
    if event == cv2.EVENT_LBUTTONDOWN:
        cv2.circle(base_image_copy, (x, y), 4, (0, 0, 255), -1)
        points.append([x, y])
        if len(points) <= 4:
            cv2.imshow('image', base_image_copy)

# initialize the list for selected points
points = []
# load the base image
base_image = cv2.imread('prob5_1.jpg')
```

```
base_image_copy = base_image.copy()

cv2.imshow('image', base_image_copy)
# Select 4 corner points using mouse click
cv2.setMouseCallback('image', click_event)
cv2.waitKey(0)
cv2.destroyAllWindows()

# sort the selected 4 points from top-left to top-right to bottom-right to
bottom-left.
def sort_pts(points):
    sorted_pts = np.zeros((4, 2), dtype="float32")
    s = np.sum(points, axis=1) # sum each row
    sorted_pts[0] = points[np.argmin(s)] # the index of the min value of
sum each row: top-left corner
    sorted_pts[2] = points[np.argmax(s)] ## the index of the min value of
sum each row: bottom-right corner

    diff = np.diff(points, axis=1) # y - x for each point
    sorted_pts[1] = points[np.argmin(diff)] # the top-right point has the
min (y-x) value
    sorted_pts[3] = points[np.argmax(diff)] # the bottom-left point has
the max (y-x) value

    return sorted_pts

sorted_pts = sort_pts(points)

print('Selected Corners: ' , sorted_pts)
```

Output Image:

The 4 corners selcted on the base image are used in the following script.

```
# Compute Homography
def ComputeH(img2_pts, size):
    img1_pts = np.array([(0,0), (size[0],0), (size[0], size[1]),(0,
size[1])])
    x = img1_pts[:,0]
    y = img1_pts[:,1]
    xprime = img2_pts[:,0]
    yprime = img2_pts[:,1]

    # matrix A
    A = np.zeros((img2_pts.shape[0]*2,9)) # 8 x 9 matrix filled with zeros
    A[::2,0:2] = img1_pts #row 1,3,4,5 and col 1,2 are filled with old
points
    A[1::2, 3:5] = img1_pts # row 2,4,6,8 and col 4,5 are filled with old
points
    A[::2, 2] = 1
    A[1::2, 5] = 1
    # col 7
    A[0::2,6] = -x*xprime
    A[1::2,6] = -x*yprime
    # col 8
    A[0::2,7] = -xprime*y
    A[1::2,7] = -yprime*y
```

```
    # col 9
    A[0::2,8] = -xprime
    A[1::2,8] = -yprime

    H = null_space(A)
    return H.reshape(3,3)

# Obtained and Modified from https://pub.towardsai.net/what-is-
perspective-warping-opencv-and-python-750e7a13d386

def combineImages(warp_img, base_img, corners):
    # create a mask for the blank image with the shape of the base image
    mask = np.zeros(base_img.shape, dtype=np.uint8)
    # fill the area bounded by 4 corners with white
    roi_corners = np.int32(corners)
    cv2.fillConvexPoly(mask, roi_corners, (255, 255, 255))
    # invert the mask colors
    mask = cv2.bitwise_not(mask)
    # insert your picture into the masked image
    masked_img = cv2.bitwise_and(base_img, mask)
    output = cv2.bitwise_or(warp, masked_img)
    return output

base_img = cv2.imread('prob5_1.jpg')
board_img = cv2.imread('prob5_board.jpg') # projected image on board
clock_img = cv2.imread('clock_wall.png') # projected clock image

# board image
# seleced 4 corners which bound the area for the meme picture projection
corner = np.array([[107, 362], [653, 429], [652, 801], [99, 908]])
H = ComputeH(corner, (board_img.shape[1], board_img.shape[0]))
warp = cv2.warpPerspective(board_img, H, (base_img.shape[1],
base_img.shape[0]))
output = combineImages(warp, base_img, corner)
# clock image
# selected 4 corners which bound the area for the clock picture projection
corner = np.array([[1222, 461], [1314, 454], [1315, 494], [1222, 499]])
H = ComputeH(corner, (clock_img.shape[1], clock_img.shape[0]))
warp = cv2.warpPerspective(clock_img, H, (output.shape[1],
output.shape[0]))
output = combineImages(warp, output, corner)
# plotting
plt.figure(figsize=(12,9))
plt.axis('off')
plt.imshow(output[:,:,::-1])
```

5(b)

Output Image:

The code is similar to 5(a), so only a section of code is shown.

```
## Prob5(b)
base_img = cv2.imread('prob5_2.jpg')
proj_img1 = cv2.imread('prob5(2)_Bill.png') # projected image on board
proj_img2 = cv2.imread('prob5(2)_383.png')

# Projected Image 1
# seleced 4 corners which bound the area for the picture projection
corner = np.array([[710, 37], [1089, 38], [1090, 199], [708, 199]])
H = ComputeH(corner, (proj_img1.shape[1], proj_img1.shape[0]))
warp = cv2.warpPerspective(proj_img1, H, (base_img.shape[1],
base_img.shape[0]))
output = combineImages(warp, base_img, corner)

# Projected Image 2
# selected 4 corners which bound the area for the clock picture projection
corner = np.array([[825, 261], [971, 260], [972, 301], [823, 302]])
H = ComputeH(corner, (proj_img2.shape[1], proj_img2.shape[0]))
warp = cv2.warpPerspective(proj_img2, H, (output.shape[1],
output.shape[0]))
output = combineImages(warp, output, corner)
# plotting
plt.figure(figsize=(12,9))
```

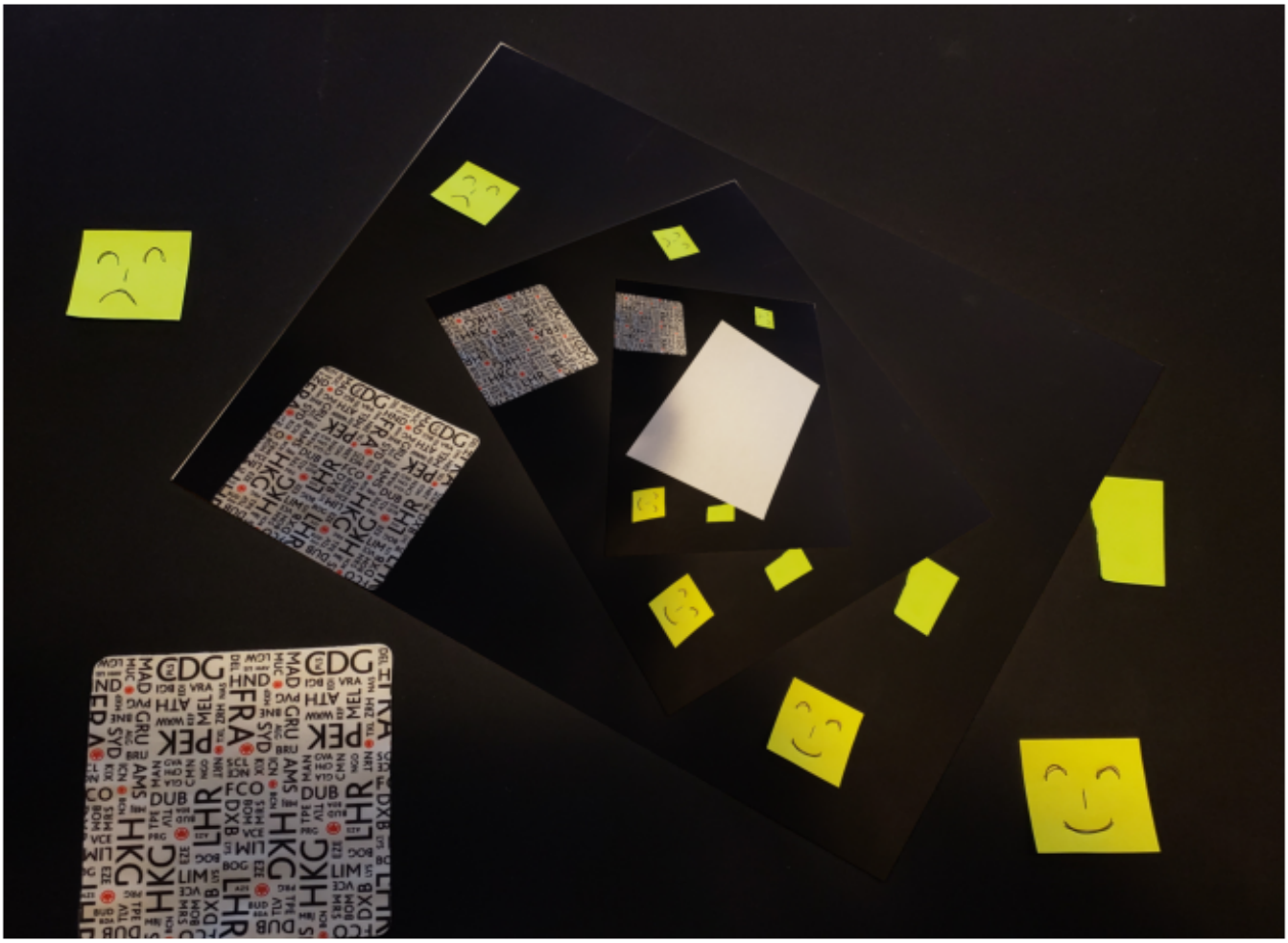
```
plt.axis('off')
plt.imshow(output[:,:,::-1])
```

5(c)

Output Image:

```
## Prob5(c)
base_img = cv2.imread('prob5_3.jpg')
proj_img1 = cv2.imread('prob5(3)_Biderman.jpg') # projected image on board

# selected 4 corners which bound the area for the picture projection
corner = np.array([[262, 292], [479, 206], [513, 332], [320, 413]])
H = ComputeH(corner, (proj_img1.shape[1], proj_img1.shape[0]))
warp = cv2.warpPerspective(proj_img1, H, (base_img.shape[1],
base_img.shape[0]))
output = combineImages(warp, base_img, corner)

# plotting
plt.figure(figsize=(12,9))
plt.axis('off')
plt.imshow(output[:,:,::-1])
```

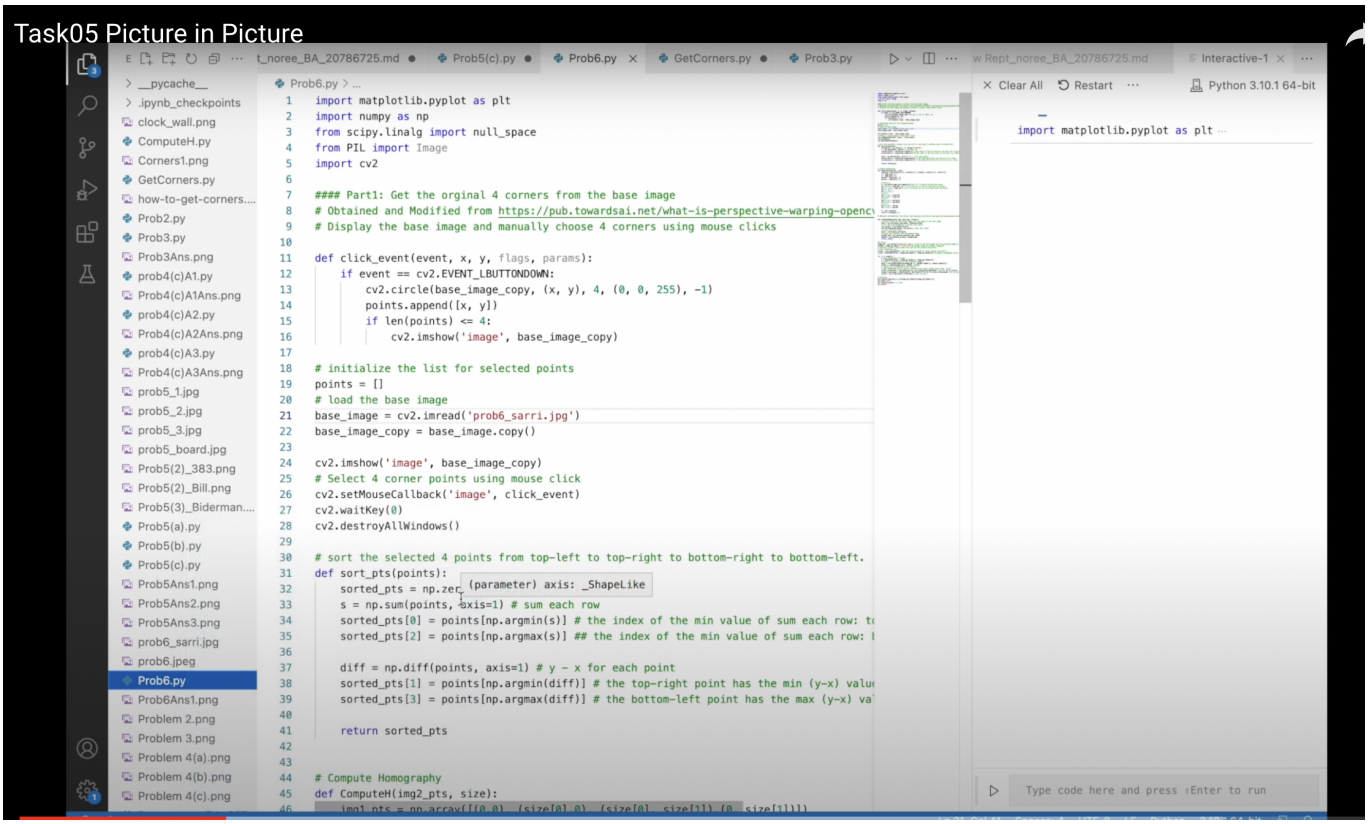
Problem 6: Picture in Picture in Picture in Picture (20 points)



```
## Prob 6
image_p6 = cv2.imread('prob6.jpeg') # used as the base image and the
projected image at the same time
output = image_p6.copy() # initialize the output image by cloning
'image_p6'
# seleced 4 corners which bound the area for the picture projection
# From 'GetCorners.py'
corner = np.array([[1141, 96], [2659, 878], [2019, 2131], [386, 1155]])
H_org = ComputeH(corner, (image_p6.shape[1], image_p6.shape[0])) #original
homography matrix

for i in range(3):
    # apply homography to image
    H = ComputeH(corner, (image_p6.shape[1], image_p6.shape[0]))
    # (base, H, (dimensions of the projected image))
    warp = cv2.warpPerspective(image_p6, H, (output.shape[1],
output.shape[0]))
    output = combineImages(warp, output, corner)
    # compute new transformed corner points
    # same homography martix used to transform the corner points every
time: 'H_org'
    corner_trasformed = [np.matmul(H_org, np.transpose(np.append(pt, 1)))
for pt in corner]
```

```
    corner_trasformed = [[pt[0]/pt[2], pt[1]/pt[2]] for pt in
corner_trasformed] # HC to Cartesian
    corner = np.array(corner_trasformed) # List to array

# plotting
plt.figure(figsize=(12,12*(image_p6.shape[0]/image_p6.shape[1])))
plt.axis('off')
plt.imshow(output[:,:,::-1])
plt.show()
```

Using My Own Image




```
#### Part1: Get the orginal 4 corners from the base image
# Obtained and Modified from https://pub.towardsai.net/what-is-
perspective-warping-opencv-and-python-750e7a13d386
# Display the base image and manually choose 4 corners using mouse clicks

def click_event(event, x, y, flags, params):
    if event == cv2.EVENT_LBUTTONDOWN:
        cv2.circle(base_image_copy, (x, y), 4, (0, 0, 255), -1)
        points.append([x, y])
        if len(points) <= 4:
            cv2.imshow('image', base_image_copy)

# initialize the list for selected points
points = []
# load the base image
base_image = cv2.imread('prob6_sarri.jpg')
base_image_copy = base_image.copy()

cv2.imshow('image', base_image_copy)
# Select 4 corner points using mouse click
cv2.setMouseCallback('image', click_event)
cv2.waitKey(0)
cv2.destroyAllWindows()

# sort the selected 4 points from top-left to top-right to bottom-right to
bottom-left.
def sort_pts(points):
    sorted_pts = np.zeros((4, 2), dtype="float32")
    s = np.sum(points, axis=1) # sum each row
    sorted_pts[0] = points[np.argmin(s)] # the index of the min value of
```

```
    sum each row: top-left corner
        sorted_pts[2] = points[np.argmax(s)] ## the index of the min value of
    sum each row: bottom-right corner

        diff = np.diff(points, axis=1) # y - x for each point
        sorted_pts[1] = points[np.argmin(diff)] # the top-right point has the
    min (y-x) value
        sorted_pts[3] = points[np.argmax(diff)] # the bottom-left point has
    the max (y-x) value

        return sorted_pts


    # Compute Homography
    def ComputeH(img2_pts, size):
        img1_pts = np.array([(0,0), (size[0],0), (size[0], size[1]),(0,
    size[1])])
        x = img1_pts[:,0]
        y = img1_pts[:,1]
        xprime = img2_pts[:,0]
        yprime = img2_pts[:,1]

        # matrix A
        A = np.zeros((img2_pts.shape[0]*2,9)) # 8 x 9 matrix filled with zeros
        A[::2,0:2] = img1_pts #row 1,3,4,5 and col 1,2 are filled with old
    points
        A[1::2, 3:5] = img1_pts # row 2,4,6,8 and col 4,5 are filled with old
    points
        A[::2, 2] = 1
        A[1::2, 5] = 1
        # col 7
        A[0::2,6] = -x*xprime
        A[1::2,6] = -x*yprime
        # col 8
        A[0::2,7] = -xprime*y
        A[1::2,7] = -yprime*y
        # col 9
        A[0::2,8] = -xprime
        A[1::2,8] = -yprime

        H = null_space(A)
        return H.reshape(3,3)

    # Obtained and Modified from https://pub.towardsai.net/what-is-
    perspective-warping-opencv-and-python-750e7a13d386

    def combineImages(warp_img, base_img, corners):
        # create a mask for the blank image with the shape of the base image
        mask = np.zeros(base_img.shape, dtype=np.uint8)
        # fill the area bounded by 4 corners with white
        roi_corners = np.int32(corners)
        cv2.fillConvexPoly(mask, roi_corners, (255, 255, 255))
        # invert the mask colors
        mask = cv2.bitwise_not(mask)
```

```
    # insert your picture into the masked image
    masked_img = cv2.bitwise_and(base_img, mask)
    output = cv2.bitwise_or(warp, masked_img)
    return output

## Prob 6
image_p6 = cv2.imread('prob6_sarri.jpg') # used as the base image and the
projected image at the same time
output = image_p6.copy() # initialize the output image by cloning
'image_p6'
# seleced 4 corners which bound the area for the picture projection
# From 'GetCorners.py'
corner = sort_pts(points) # Get the sorted original 4 corner points From
Part 1
H_org = ComputeH(corner, (image_p6.shape[1], image_p6.shape[0])) #original
homography matrix

for i in range(3):
    # apply homography to image
    H = ComputeH(corner, (image_p6.shape[1], image_p6.shape[0]))
    # (base, H, (dimensions of the projected image))
    warp = cv2.warpPerspective(image_p6, H, (output.shape[1],
output.shape[0]))
    output = combineImages(warp, output, corner)
    # compute new transformed corner points
    # same homography martix used to transform the corner points every
time: 'H_org'
    corner_trasformed = [np.matmul(H_org, np.transpose(np.append(pt, 1)))
for pt in corner]
    corner_trasformed = [[pt[0]/pt[2], pt[1]/pt[2]] for pt in
corner_trasformed] # HC to Cartesian
    corner = np.array(corner_trasformed) # List to array

# plotting
plt.figure(figsize=(12,12*(image_p6.shape[0]/image_p6.shape[1])))
plt.axis('off')
plt.imshow(output[:,:,::-1])
plt.show()
```

Problem 7: Build your 3D Planar Measurement Software (20 points)

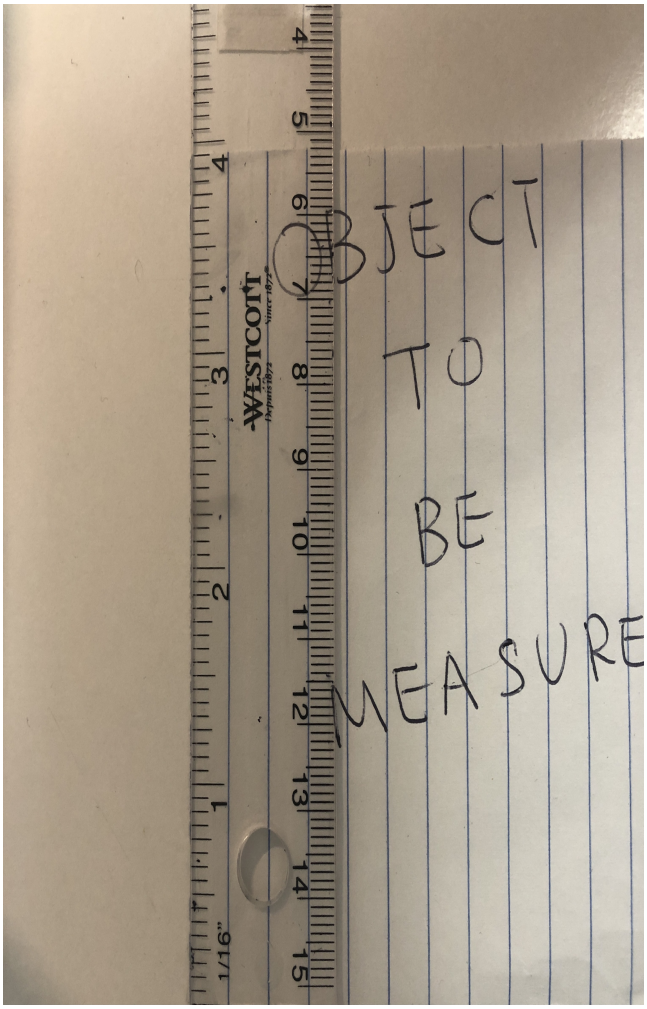
Measurement Results (in inches)

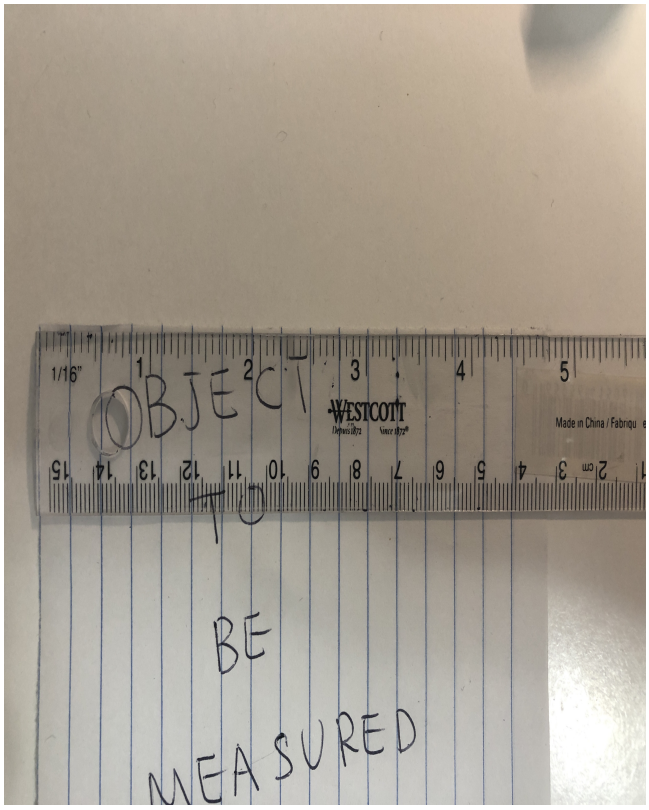
3.919573610297309

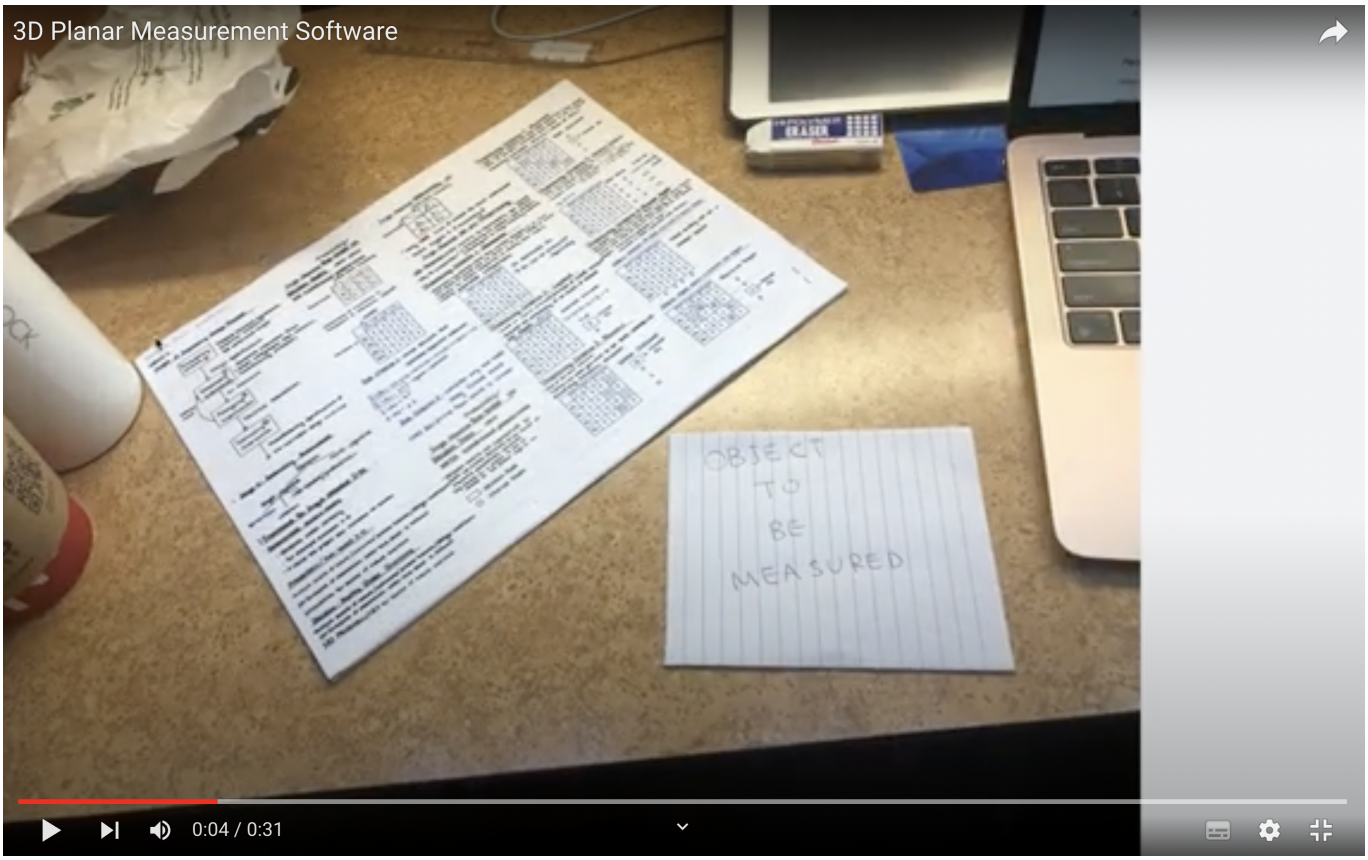
4.715850731095984

Physical Measurements

Short Side **Long Side**







```
# Obtained and Modified from https://pub.towardsai.net/what-is-
perspective-warping-opencv-and-python-750e7a13d386
# Display the base image and manually choose 4 corners using mouse clicks

def click_event(event, x, y, flags, params):
    if event == cv2.EVENT_LBUTTONDOWN:
        cv2.circle(base_image_copy, (x, y), 4, (0, 0, 255), -1)
        cornerpoints.append([x, y])
        if len(cornerpoints) <= 4:
            cv2.imshow('Select Corners of Paper', base_image_copy)


def click_event2(event, x, y, flags, params):
    if event == cv2.EVENT_LBUTTONDOWN:
        if len(points1) < 2:
            cv2.circle(base_image_copy, (x, y), 4, (0, 0, 255), -1)
            points1.append([x, y])
            poly_pts = np.array(points1).reshape((-1,1,2))
            cv2.polylines(base_image_copy, [poly_pts], False, (0, 0, 0),
2)
            cv2.imshow('Measure Distances', base_image_copy)

        elif len(points2) < 2:
            cv2.circle(base_image_copy, (x, y), 4, (0, 0, 255), -1)
            points2.append([x, y])
            poly_pts = np.array(points2).reshape((-1,1,2))
            cv2.polylines(base_image_copy, [poly_pts], False, (0, 0, 0),
2)
            cv2.imshow('Measure Distances', base_image_copy)



# Compute Homography
def ComputeH(img2_pts, size):
    img1_pts = np.array([(0,0), (size[0],0), (size[0], size[1]),(0,
size[1])])
    x = img1_pts[:,0]
    y = img1_pts[:,1]
    xprime = img2_pts[:,0]
    yprime = img2_pts[:,1]

    # matrix A
    A = np.zeros((img2_pts.shape[0]*2,9)) # 8 x 9 matrix filled with zeros
    A[::2,0:2] = img1_pts #row 1,3,4,5 and col 1,2 are filled with old
points
    A[1::2, 3:5] = img1_pts # row 2,4,6,8 and col 4,5 are filled with old
points
    A[::2, 2] = 1
    A[1::2, 5] = 1
    # col 7
    A[0::2,6] = -x*xprime
    A[1::2,6] = -x*yprime
    # col 8
```

```
    A[0::2,7] = -xprime*y
    A[1::2,7] = -yprime*y
    # col 9
    A[0::2,8] = -xprime
    A[1::2,8] = -yprime

    H = null_space(A)
    return H.reshape(3,3)

# load the base image
base_image = cv2.imread('prob7.jpg')
base_image_copy = base_image.copy()

# Select Corner Points
cornerpoints = []
cv2.imshow('Select Corners of Paper', base_image_copy)
# Select 4 corner points using mouse click
cv2.setMouseCallback('Select Corners of Paper', click_event)
cv2.waitKey(0)
cv2.destroyAllWindows()

# Compute Homography
paper_size = [11, 8.5] # select the longer side first, and then short side
H = ComputeH(np.array(cornerpoints), paper_size)
H_inv = np.linalg.inv(H)

# Select Start Points and End Points for measurement
points1 = []
points2 = []
base_image_copy = base_image.copy()
cv2.imshow('Measure Distances', base_image_copy)
cv2.setMouseCallback('Measure Distances', click_event2)
cv2.waitKey(0)
cv2.destroyAllWindows()

# Start Points and End Points: Cartisian --> HC
points1 = [[pt[0], pt[1], 1] for pt in points1]
points2 = [[pt[0], pt[1], 1] for pt in points2]

# Homography: Transform start points and end points to real-scale
real_points1 = [np.dot(H_inv, pt) for pt in points1]
real_points2 = [np.dot(H_inv, pt) for pt in points2]

# Start Points and End Points: HC --> Cartesian
real_points1 = [[pt[0]/pt[2], pt[1]/pt[2]] for pt in real_points1]
real_points2 = [[pt[0]/pt[2], pt[1]/pt[2]] for pt in real_points2]
real_points1 = np.array(real_points1)
real_points2 = np.array(real_points2)

# calculate distances
dist1 = np.linalg.norm(real_points1[1,:]-real_points1[0,:])
dist2 = np.linalg.norm(real_points2[1,:]-real_points2[0,:])
print(dist1)
print(dist2)
```