HW1

February 18, 2019

1 CSE 252A Computer Vision I Fall 2018 - Assignment 1

1.0.1 Instructor: David Kriegman

1.0.2 Assignment Published On: Tuesday, October 9, 2018

1.0.3 Due On: Tuesday, October 23, 2018 11:59 pm

1.1 Instructions

- Review the academic integrity and collaboration policies on the course website.
- This assignment must be completed individually.
- This assignment contains theoretical and programming exercises. If you plan to submit hand written answers for theoretical exercises, please be sure your writing is readable and merge those in order with the final pdf you create out of this notebook. You could fill the answers within the notebook iteself by creating a markdown cell.
- Programming aspects of this assignment must be completed using Python in this notebook.
- If you want to modify the skeleton code, you can do so. This has been provided just to provide you with a framework for the solution.
- You may use python packages for basic linear algebra (you can use numpy or scipy for basic operations), but you may not use packages that directly solve the problem.
- If you are unsure about using a specific package or function, then ask the instructor and teaching assistants for clarification.
- You must submit this notebook exported as a pdf. You must also submit this notebook as .ipynb file.
- You must submit both files (.pdf and .ipynb) on Gradescope. You must mark each problem on Gradescope in the pdf.
-

1.2 Late policy - 10% per day late penalty after due date up to 3 days.

1.3 Problem 1: Perspective Projection [5 pts]

Consider a perspective projection where a point

$$P = [\text{x y z}]^T$$

is projected onto an image plane Π' represented by $k = f' > 0$ as shown in the following figure. The first second and third coordinate axes are denoted by i, j, k respectively.

Consider the projection of two rays in the world coordinate system

$$Q1 = [7\ \text{-}3\ 1] + t[8\ 2\ 4]$$

$$Q2 = [2\ \text{-}5\ 9] + t[8\ 2\ 4]$$

where $-\infty \leq t \leq -1$.

Calculate the coordinates of the endpoints of the projection of the rays onto the image plane. Identify the vanishing point based on the coordinates.

1.4 Problem 1: Solution

Please find attached handwritten page

1.5 Problem 2: Thin Lens Equation [5 pts]

An illuminated arrow forms a real inverted image of itself at a distance of $w = 60$ cm, measured along the optical axis of a convex thin lens as shown above. The image is half the size of the object 1. How far from the object must the lens be placed? Whats is the focal length of the lens? 2. At what distance from the center of the lens should the arrow be placed so that the height of the image is the same? 3. What would be the type and location of image formed if the arrow is placed at a distance of 5 cm along the optical axis from the optical center?

1.6 Problem 2: Solution

We know the lens formula is given by:

$$1/f = 1/v - 1/u$$

where $\boldsymbol{f}$: focal length of the lens, $\boldsymbol{u}$: distance of object from lens, $\boldsymbol{v}$: distance of image from lens

Also, the magnification of lens is give by $\boldsymbol{m}$ such that:

$$m = Hi/Ho = v/u$$

where $\boldsymbol{Hi}$: Height of image, $\boldsymbol{Ho}$: Height of object

Sign Convention:

(a)We will take the measurement of variables in the direction of light as positive and opposite to the direction of light as negative. (b)Measurement in the direction above the optical axis is positive and below it, is negative.

1. Since, the formed image is inverted and half of the size of original object, so

$$m = -1/2 = v/u$$

$$u = -2v$$

and also it is given that:

$$|u| + |v| = 60cm$$
$$2v + v = 60cm$$
$$v = 20cm$$
$$u = 40cm$$

Hence, the object must be placed **40*cm*** away from the lens.

Now, in order to find the focal length ***f***, put the values of ***u*** and ***v*** in lens formula.

$$1/f = 1/20 - (-1/40)$$
$$1/f = 3/40$$
$$f = 40/3$$

Focal length is **40/3*cm***

2. Height of image same as height of object

$$m = -1 = v/u$$
$$u = -v$$

Putting the values in lens formula

$$1/f = 1/v + 1/v$$
$$1/f = 2/v$$
$$v = 2f$$
$$v = 2 * (40/3) = 80/3$$

Hence, the object must be placed **80/3*cm*** away from the lens.

3. Arrow is placed at 5cm from lens

$$u = -5cm$$
$$f = 40/3cm$$

Using the lens formula, calculate ***v***

$$1/v = 1/f + 1/u$$
$$1/v = 3/40 + (-1/5)$$
$$1/v = (15 - 40)/(40 * 5)$$
$$1/v = -(40 * 5)/25$$
$$v = -8$$

Calculate the magnification:

$$m = v/u$$
$$m = (-8)/(-5)$$
$$m = 1.6$$

Hence, image is formed **8*cm*** away from the lens at the same side as the object. Also, since the magnification is positive and greater than one, so image formed is **magnified and erect.**

1.7 Problem 3: Affine Projection [3 pts]

Show that the image of a pair of parallel lines in 3D space is a pair of parallel lines in an affine camera.

1.8 Problem 3: Solution

Please refer to the handwritten page

1.9 Problem 4: Image Formation and Rigid Body Transformations [10 points]

In this problem we will practice rigid body transformations and image formations through the projective and affine camera model. The goal will be to photograph the following four points

$$^{A}P_1 = [\text{-}1 \text{ -}0.5 \ 2]^T$$

,

$$^{A}P_2 = [1 \text{ -}0.5 \ 2]^T$$

,

$$^{A}P_3 = [1 \ 0.5 \ 2]^T$$

,

$$^{A}P_4 = [\text{-}1 \ 0.5 \ 2]^T$$

To do this we will need two matrices. Recall, first, the following formula for rigid body transformation

$$^{B}P = {}^{B}_{A}R \, {}^{A}P + {}^{B}O_A$$

Where ^{B}P is the point coordinate in the target (B) coordinate system. ^{A}P is the point coordinate in the source (A) coordinate system. $^{B}_{A}R$ is the rotation matrix from A to B, and $^{B}O_A$ is the origin of the coordinate system A expressed in B coordinates.

The rotation and translation can be combined into a single 4×4 extrinsic parameter matrix, P_e, so that $^{B}P = P_e \cdot {}^{A}P$.

Once transformed, the points can be photographed using the intrinsic camera matrix, P_i which is a 3×4 matrix.

Once these are found, the image of a point, ^{A}P, can be calculated as $P_i \cdot P_e \cdot {}^{A}P$.

We will consider four different settings of focal length, viewing angles and camera positions below. For each of these calculate:

a) Extrinsic transformation matrix,

b) Intrinsic camera matrix under the perspective camera assumption.

c) Intrinsic camera matrix under the affine camera assumption. In particular, around what point do you do the taylor series expansion?

d) Calculate the image of the four vertices and plot using the supplied functions

Your output should look something like the following image (Your output values might not match, this is just an example) 1. [No rigid body transformation]. Focal length = 1. The optical axis of the camera is aligned with the z-axis. 2. [Translation]. $^{B}O_A = [0 \ 0 \ 1]^T$. Focal length = 1. The optical axis of the camera is aligned with the z-axis. 3. [Translation and Rotation]. Focal length = 1.

${}^{B}_{A}R$ encodes a 30 degrees around the z-axis and then 60 degrees around the y-axis. ${}^{B}O_{A} = [0\ 0\ 1]^{T}$.
4. [Translation and Rotation, long distance]. Focal length = 5. ${}^{B}_{A}R$ encodes a 30 degrees around the z-axis and then 60 degrees around the y-axis. ${}^{B}O_{A} = [0\ 0\ 13]^{T}$.

> You can refer the Richard Szeliski starting page 36 for image formation and the extrinsic matrix.
>
> Intrinsic matrix calculation for perspective and affine camera models was covered in class and can be referred in slide 3 http://cseweb.ucsd.edu/classes/fa18/cse252A-a/lec3.pdf

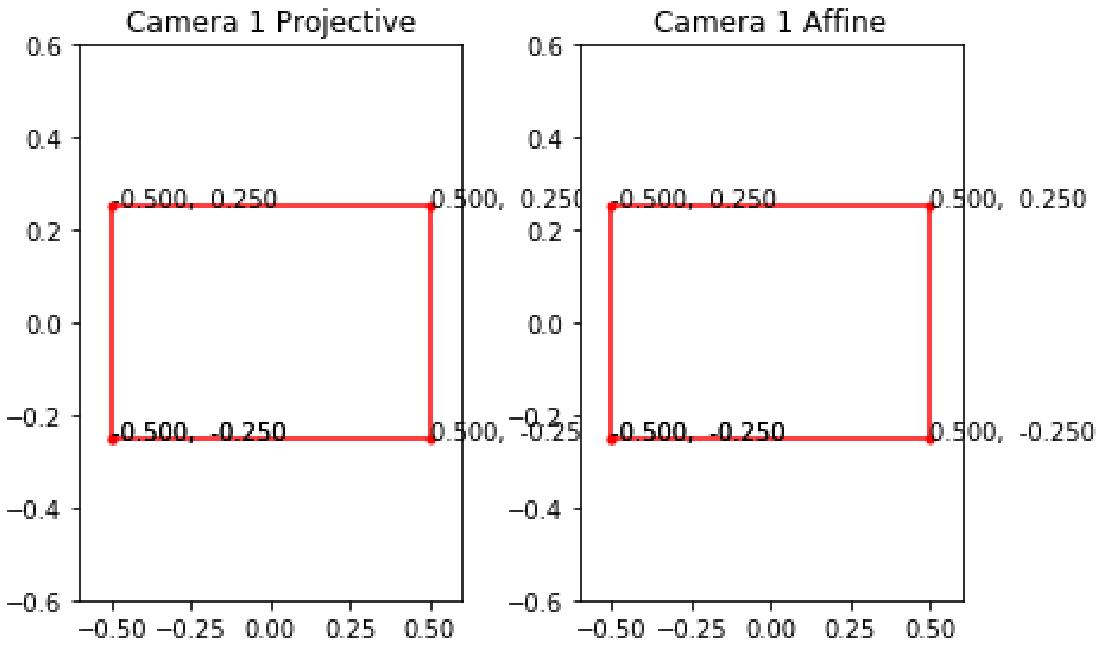
We will not use a full intrinsic camera matrix (e.g. that maps centimeters to pixels, and defines the coordinates of the center of the image), but only parameterize this with f, the focal length. In other words: the only parameter in the intrinsic camera matrix under the perspective assumption is f, and the only ones under the affine assumption are: f, x_0, y_0, z_0, where x_0, y_0, z_0 is the center of the taylor series expansion.

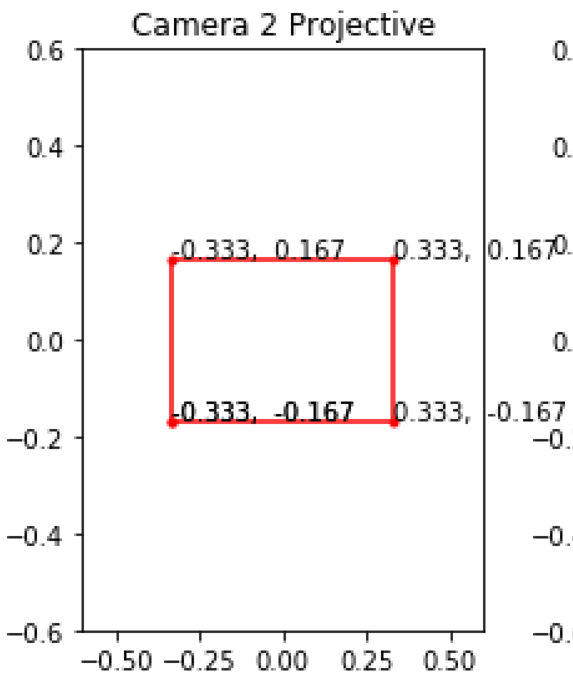
Note that the axis are the same for each row, to facilitate comparison between the two camera models.

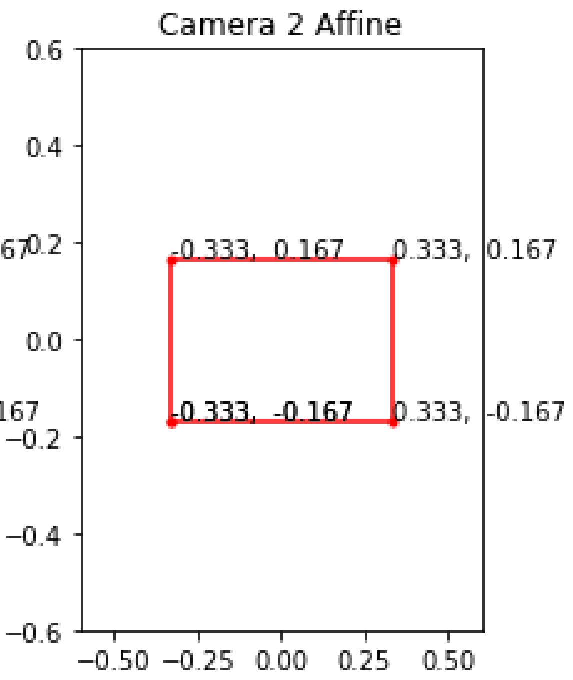
```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        import math


        # convert points from euclidian to homogeneous
        def to_homog(points):
            ones = [1. for _ in range(points.shape[1])]
            # append 1. for all points as last coordinate
            points = np.append(points, [ones], axis=0)

            return points


        # convert points from homogeneous to euclidian
        def from_homog(points_homog):
            # get all the last element for all homogeneous points
            w = points_homog[-1:][0]
            points_homog = np.delete(points_homog, -1, 0)

            # now divide each point with it's corresponding w
            for idx, wi in enumerate(w):
                points_homog[:, idx] /= wi

            return points_homog


        # project 3D euclidian points to 2D euclidian
        def project_points(P_int, P_ext, pts):
            # convert first points to homogeneous coordinates
```

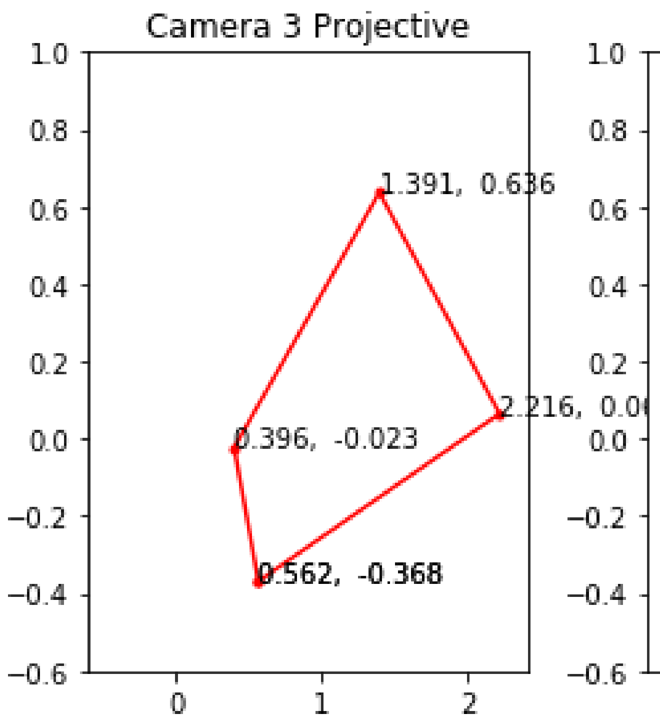
```
    pts_homo = to_homog(pts)
    # apply transformation using intrinsic and extrinsic matrix
    P_camera = np.matmul(P_int, P_ext)
    pts_proj = np.matmul(P_camera, pts_homo)
    # now convert them to eclidean coordinates
    pts_proj = from_homog(pts_proj)

    return pts_proj


# Change the three matrices for the four cases as described in the problem
# in the four camera functions geiven below. Make sure that we can see the
# formula (if one exists) being used to fill in the matrices. Feel free
# to document with comments any thing you feel the need to explain.

"""
Considering f = focal length of camera

Q1. Projective camera matrix 3x4 is given by:

[[1, 0, 0,   0],
 [0, 1, 0,   0],
 [0, 0, 1/f, 0]]

 also called P_intrinsic_projective

Q2. Affine camera matrix 3x4, by taylor series approx.
   around point (xo, yo, zo) is given by:

[[f/zo,  0,    -fxo/zo2,  fxo/zo ],
 [0,    f/zo,  -fyo/zo2,  fyo/zo ],
 [0,     0,      0,          1   ]]

 also called P_intrinsic_affine

Q3. Rigid transformation matrix 4x4 taking into account
   rotation and translation is given by
   [[R, O],
    [0, 1]]

  where R = Net Rotation matrix, O = Translation vector

Q4. Rotation matrix 3x3 about z-axis is given by
  [[cosA, -sinA, 0],
   [sinA, cosA,  0],
   [0,     0,    1]]

   where A = angle of rotation
```

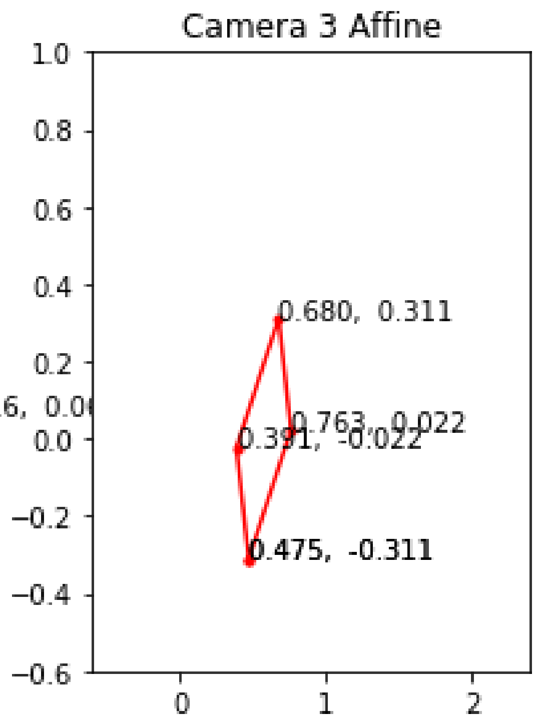
```
Q5. Rotation matrix 3x3 about y-axis is given by
  [[cosA,   0,  sinA],
   [ 0,     1,   0  ],
   [-sinA,  0,  cosA]]

   where A = angle of rotation
"""

def camera1():
    # given f=1, [No rigid body transformation]
    # using Q1 above putting f=1
    P_int_proj = np.eye(3,4)

    # taking taylor series of Q2 around point P=(0, 0, 2), since
    # this is the center of four points given to me
    P_int_affine = np.zeros((3,4))
    P_int_affine[0,0] = 1./2
    P_int_affine[1,1] = 1./2
    P_int_affine[2,3] = 1.

    # Since there is no any kind of Rigid body transformation
    P_ext = np.eye(4,4)

    return P_int_proj, P_int_affine, P_ext

def camera2():
    # given f=1, [Translation]
    # using Q1 above putting f=1
    P_int_proj = np.eye(3,4)

    # taking taylor series of Q2 around point P=(0, 0, 3),
    # since this is the center of four points given to me
    P_int_affine = np.zeros((3,4))
    P_int_affine[0,0] = 1./3
    P_int_affine[1,1] = 1./3
    P_int_affine[2,3] = 1.

    # Since there is no any kind of Rigid body transformation in terms of rotation
    # only translation exist
    P_ext = np.eye(4,4)
    P_ext[2,3] = 1.

    return P_int_proj, P_int_affine, P_ext

def camera3():
    # given f=1, [Translation and Rotation]
    # using Q1 above putting f=1
    P_int_proj = np.eye(3,4)
    P_int_affine = np.zeros((3,4))
    P_int_affine[0,0] = 1./3
    P_int_affine[1,1] = 1./3
    P_int_affine[2,3] = 1.

    # calculation of rotation matrix for rotation around z-axis
    # using Q4 by 30 degrees
    R_z = np.zeros((3,3))
    R_z[0,0], R_z[0,1] = math.sqrt(3)/2., -1/2.
    R_z[1,0], R_z[1,1] = 1/2., math.sqrt(3)/2.
    R_z[2,2] = 1
    # calculation of rotation matrix rotation around y-axis
    # using Q5 by 60 degrees
    R_y = np.zeros((3,3))
    R_y[0,0], R_y[0,2] = 1/2., math.sqrt(3)/2.
    R_y[1,1] = 1.
    R_y[2,0], R_y[2,2] = -math.sqrt(3)/2., 1/2.
    # Final Rotation matrix
    R = np.matmul(R_y, R_z)

    # Calculating extrinsic marix using rotation and translation
    P_ext = np.zeros((4,4))
    P_ext[:3,:3] = R[:,:]
    P_ext[2,3], P_ext[3,3] = 1., 1.

    return P_int_proj, P_int_affine, P_ext

def camera4():
    # f = 5
    P_int_proj = np.eye(3,4)
    P_int_proj[2,2] = 0.2

    P_int_affine = np.zeros((3,4))
    P_int_affine[0,0] = 5./15
    P_int_affine[1,1] = 5./15
    P_int_affine[2,3] = 1.

    # calculation of rotation matrix for rotation around z-axis
    # using Q4 by 30 degrees
    R_z = np.zeros((3,3))
    R_z[0,0], R_z[0,1] = math.sqrt(3)/2., -1/2.
    R_z[1,0], R_z[1,1] = 1/2., math.sqrt(3)/2.
    R_z[2,2] = 1
    # calculation of rotation matrix rotation around y-axis
    # using Q5 by 60 degrees
    R_y = np.zeros((3,3))
    R_y[0,0], R_y[0,2] = 1/2., math.sqrt(3)/2.
    R_y[1,1] = 1.
    R_y[2,0], R_y[2,2] = -math.sqrt(3)/2., 1/2.
    # Final Rotation matrix
    R = np.matmul(R_y, R_z)

    # Calculating extrinsic marix using rotation and translation
    P_ext = np.zeros((4,4))
    P_ext[:3,:3] = R[:,:]
    P_ext[2,3], P_ext[3,3] = 13., 1.

    return P_int_proj, P_int_affine, P_ext


# Use the following code to display your outputs
# You are free to change the axis parameters to better
# display your quadrilateral but do not remove any annotations

def plot_points(points, title='', style='.-r', axis=[]):
    inds = list(range(points.shape[1]))+[0]
    plt.plot(points[0,inds], points[1,inds],style)

    for i in range(len(points[0,inds])):
        plt.annotate(str("{0:.3f}".format(
            points[0,inds][i]))+",  "+str("{0:.3f}".format(
            points[1,inds][i])),(points[0,inds][i], points[1,inds][i]))

    if title:
        plt.title(title)
    if axis:
        plt.axis(axis)

    plt.tight_layout()

def main():
    point1 = np.array([[-1,-.5,2]]).T
    point2 = np.array([[1,-.5,2]]).T
    point3 = np.array([[1,.5,2]]).T
    point4 = np.array([[-1,.5,2]]).T
    points = np.hstack((point1,point2,point3,point4))

    for i, camera in enumerate([camera1, camera2]):
        P_int_proj, P_int_affine, P_ext = camera()
        plt.subplot(1, 2, 1)
        plot_points(project_points(P_int_proj, P_ext, points), title='Camera %d Projec
        plt.subplot(1, 2, 2)
        plot_points(project_points(P_int_affine, P_ext, points), title='Camera %d Affi
        plt.show()
    for i, camera in enumerate([camera3, camera4]):
        P_int_proj, P_int_affine, P_ext = camera()
        plt.subplot(1, 2, 1)
        plot_points(project_points(P_int_proj, P_ext, points), title='Camera %d Projec
        plt.subplot(1, 2, 2)
        plot_points(project_points(P_int_affine, P_ext, points), title='Camera %d Affi
        plt.show()

main()
```

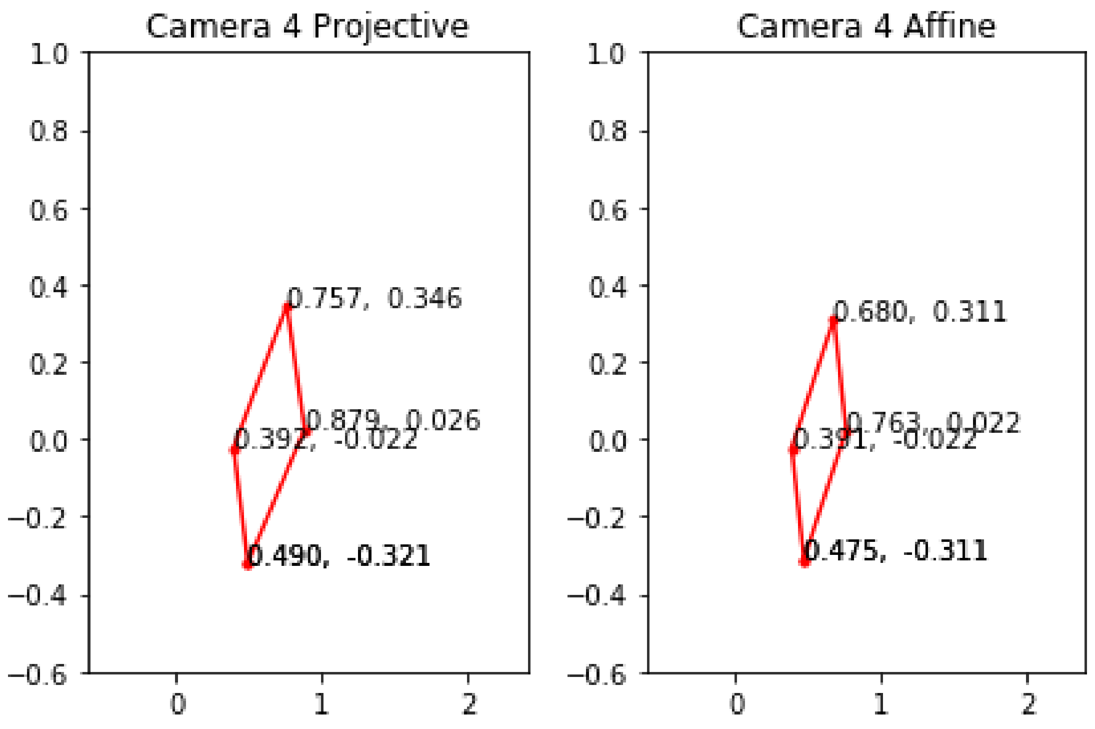



Camera 2 Projective
-0.333, 0.167
0.333, 0.167
-0.333, -0.167
0.333, -0.167


Camera 2 Affine
-0.333, 0.167
0.333, 0.167
-0.333, -0.167
0.333, -0.167


Camera 3 Projective
1.391, 0.636
2.216, 0.0
0.396, -0.023
0.562, -0.368


Camera 3 Affine
0.680, 0.311
0.763, 0.022
0.391, -0.022
0.475, -0.311



1. The actual points around which you did the taylor series expansion for the affine camera models.

Answer I choose the center of the four points for the Taylor series expansion. The point is (0,0,2) and then it is translated based on given translation vector while being used for each case.

2. How did you arrive at these points?

Answer This choice was made based on the reasoning that the center of the points after rotation and translation is the closest point to each of the four points. And since we choose the neighboring point while doing deriving affine camer model, so this will produce the best approximation result.

3. How do the projective and affine camera models differ? Why is this difference smaller for the last image compared to the second last?

Answer

The projective camera model involves no approximation while the affine camera model uses Taylor series expansion around a point to estimate image coordinates where we ignore higher order terms in taylor series expansion. The point that we choose is neighbor of points of image, so for small focal length the higher order terms that we are neglecting for affine camera approximation as still significant whereas for large focal length, these higher order terms has less significant, hence we have approximately same image for both cases.

1.10 Problem 5: Homography [12 pts]

You may use eig or svd routines in python for this part of the assignment.

Consider a vision application in which components of the scene are replaced by components from another image scene.

In this problem, we will implement partial functionality of a smartphone camera scanning application (Example: CamScanner) that, in case you've never used before, takes pictures of documents and transforms it by warping and aligning to give an image similar to one which would've been obtained through using a scanner.

The transformation can be visualized by imagining the use of two cameras forming an image of a scene with a document. The scene would be the document you're trying to scan placed on a table and one of the cameras would be your smart phone camera, forming the image that you'll be uploading and using in this assignment. There can also be an ideally placed camera, oriented in the world in such a way that the image it forms of the scene has the document perfectly algined. While it is unlikely you can hold your phone still enough to get such an image, we can use homography to transform the image you take into the image that the ideally placed camera would have taken.

This digital replacement is accomplished by a set of corresponding points for the document in both the source (your picture) and target (the ideal) images. The task then consists of mapping the points from the source to their respective points in the target image. In the most general case, there would be no constraints on the scene geometry, making the problem quite hard to solve. If, however, the scene can be approximated by a plane in 3D, a solution can be formulated much more easily even without the knowledge of camera calibration parameters.

To solve this section of the homework, you will begin by understanding the transformation that maps one image onto another in the planar scene case. Then you will write a program that implements this transformation and use it to warp some document into a well aligned document (See the given example to understand what we mean by well aligned).

To begin with, we consider the projection of planes in images. imagine two cameras C_1 and C_2 looking at a plane π in the world. Consider a point P on the plane π and its projection $p = [\text{u1, v1, 1}]^T$ in the image 1 and $q = [\text{u2, v2, 1}]^T$ in image 2.

There exists a unique, upto scale, 3×3 matrix H such that, for any point P:

$$q \approx Hp$$

Here $\approx$ denotes equality in homogeneous coordinates, meaning that the left and right hand sides are proportional. Note that H only depends on the plane and the projection matrices of the two cameras.

The interesting thing about this result is that by using H we can compute the image of P that would be seen in the camera with center C_2 from the image of the point in the camera with center at C_1, without knowing the three dimensional location. Such an H is a projective transformation of the plane, called a homography.

In this problem, complete the code for computeH and warp functions that can be used in the skeletal code that follows.

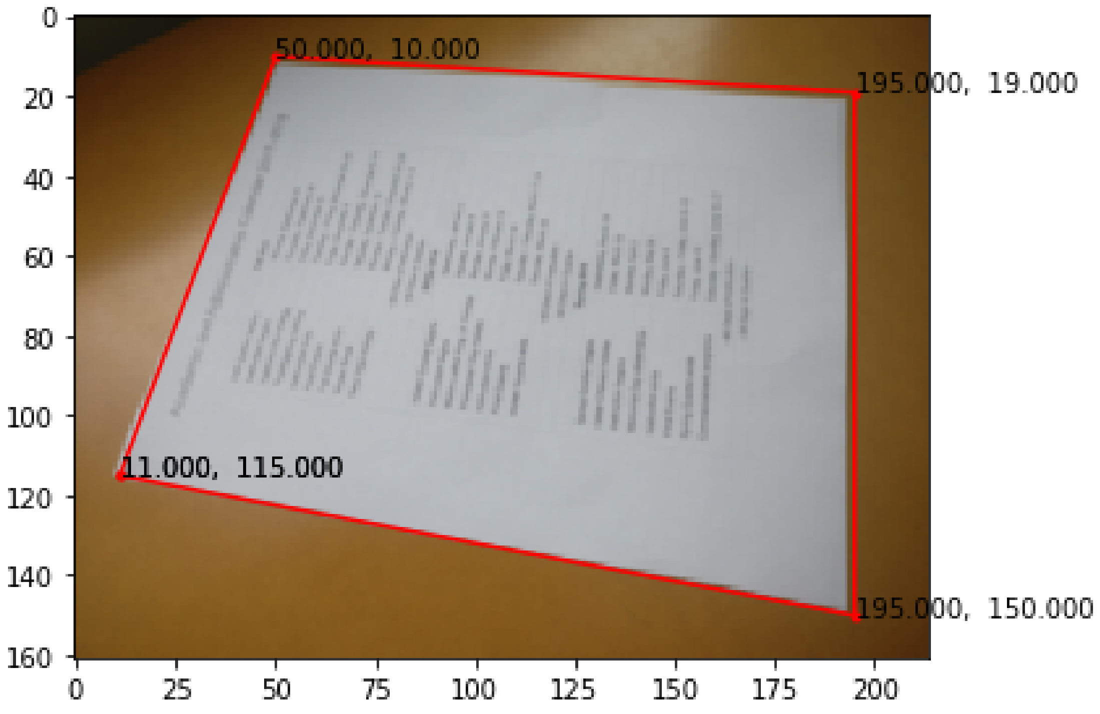
There are three warp functions to implement in this assignment, example ouputs of which are shown below. In warp1, you will create a homography from points in your image to the target image (Mapping source points to target points). In warp2, the inverse of this process will be done. In warp3, you will create a homography between a given image and your image, replacing your document with the given image.

1. 2. 3.
2. In the context of this problem, the source image refers to the image of a document you take that needs to be replaced into the target.
3. The target image can start out as an empty matrix that you fill out using your code.
4. You will have to implement the computeH function that computes a homography. It takes in the point correspondences between the source image and target image in homogeneous coordinates respectively and returns a 3×3 homography matrix.
5. You will also have to implement the three warp functions in the skeleton code given and plot the resultant image pairs. For plotting, make sure that the target image is not smaller than the source image.

Note: We have provided test code to check if your implementation for computeH is correct. All the code to plot the results needed is also provided along with the code to read in the images and other data required for this problem. Please try not to modify that code.

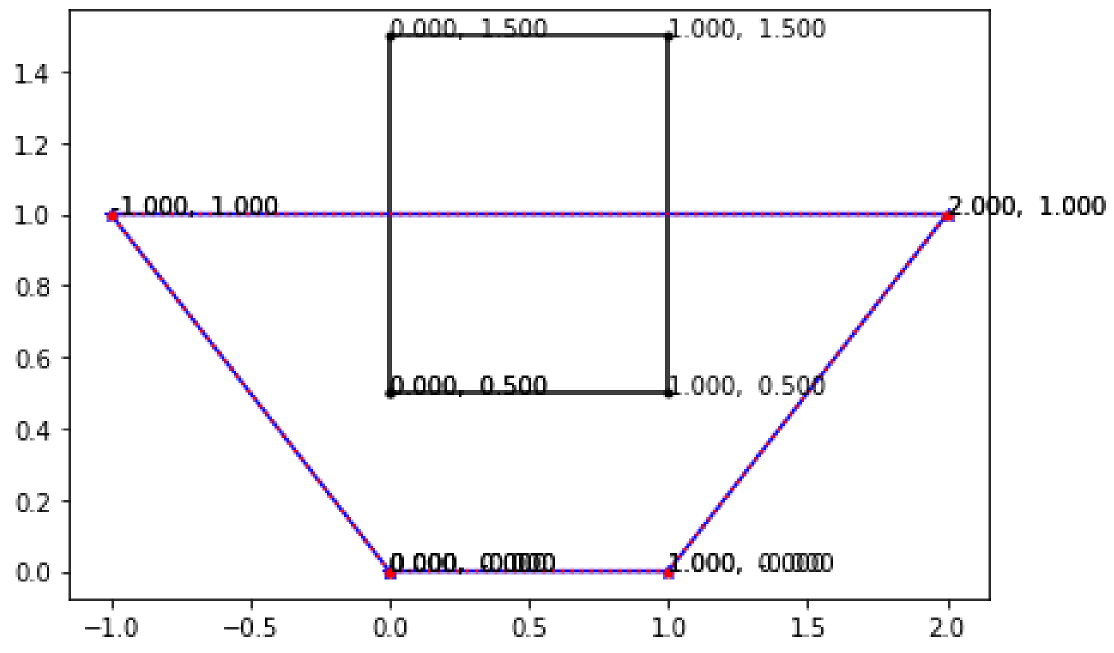
You may find following python built-ins helpful: numpy.linalg.svd, numpy.meshgrid

```
In [2]: import numpy as np
        from scipy.misc import imresize, imread
        from scipy.io import loadmat
        import matplotlib.pyplot as plt

        # to supress warning due to usage of old apis from scipy
        import warnings
        warnings.filterwarnings('ignore')

        img_path = "./photo.jpg"

        # Load image to be used - resize to make sure it's not too large
        # You can use the given image as well. A large image will make
        # testing your code take longer; once you're satisfied with
        # your result, you can, if you wish to, make the image larger
        # (or till your computer memory allows you to)

        source_image = imresize(imread(img_path),.1)[:,:,:3]/255.

        # display images
        plt.imshow(source_image)

        # Align the polygon such that the corners align with the document
        # in your picture. This polygon doesn't need to overlap with the
        # edges perftectly, an approximation is fine. The order of points
        # is clockwise, starting from bottom left.
        x_coords = [11,50,195,195]
        y_coords = [115,10,19,150]

        # Plot points from the previous problem is used to draw over your image
        # Note that your coordinates will change once you resize your image again
        source_points = np.vstack((x_coords, y_coords))
        plot_points(source_points)

        plt.show()
```



```
In [3]: def computeH(source_points, target_points):
            # returns the 3x3 homography matrix such that:
            # np.matmul(H, source_points) ~ target_points
            # where source_points and target_points are expected to be in homogeneous

            # Please refer the note on DLT algorithm given at:
            # https://cseweb.ucsd.edu/classes/wi07/cse252a/\
            # homography_estimation/homography_estimation.pdf

            # make sure points are 3D homogeneous
            assert source_points.shape[0]==3 and target_points.shape[0]==3

            # Calculate A matrix using given source and target points and then
            # solve equation Ah = 0, using SVD algorithm that will give V matrix.
            # H(Homography matrix) is 3x3 reshape of last column of matrix V
            A = []
            for i in range(0, source_points.shape[1]):
                x1, y1, _ = source_points[:, i]
```

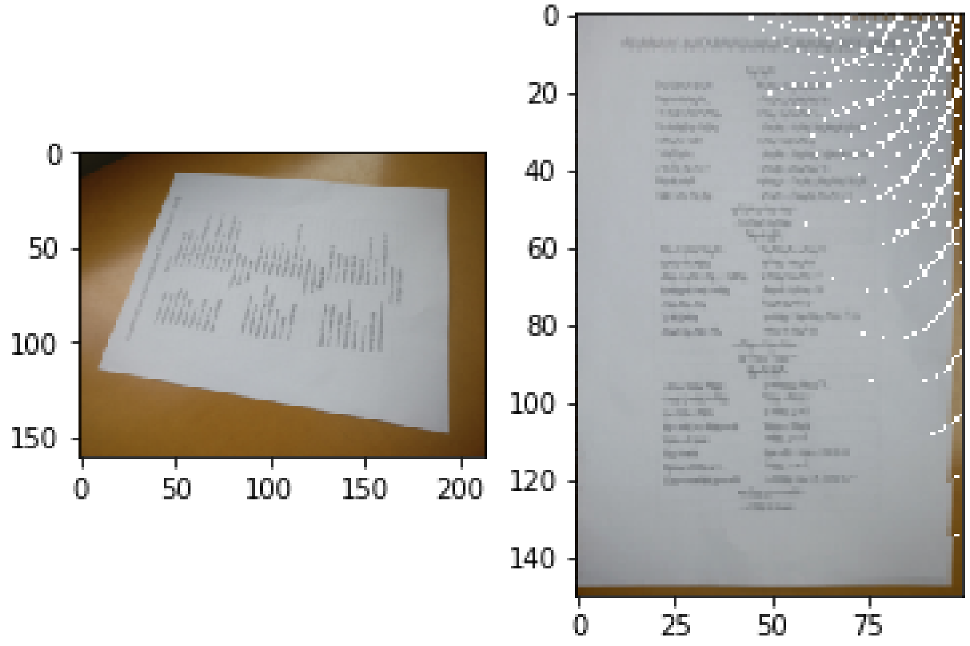
```
        x2, y2, _ = target_points[:, i]
        A.append([-x1, -y1, -1, 0, 0, 0, x2*x1, x2*y1, x2])
        A.append([0, 0, 0, -x1, -y1, -1, y2*x1, y2*y1, y2])

    A = np.asarray(A)

    U, S, V = np.linalg.svd(A)
    # Note SVD in numpy gives transpose of V, not actual V matrix
    # So we will do transpose to get V matrix
    H = V.T[:,-1].reshape(3, 3)

    return H


#######################################################
# test code. Do not modify
#######################################################
def test_computeH():
    source_points = np.array([[0,0.5],[1,0.5],[1,1.5],[0,1.5]]).T
    target_points = np.array([[0,0],[1,0],[2,1],[-1,1]]).T
    H = computeH(to_homog(source_points), to_homog(target_points))
    mapped_points = from_homog(np.matmul(H,to_homog(source_points)))

    plot_points(source_points,style='.-k')
    plot_points(target_points,style='*-b')
    plot_points(mapped_points,style='.:r')
    plt.show()
    print('The red and blue quadrilaterals should overlap if ComputeH is implemented c
test_computeH()
```

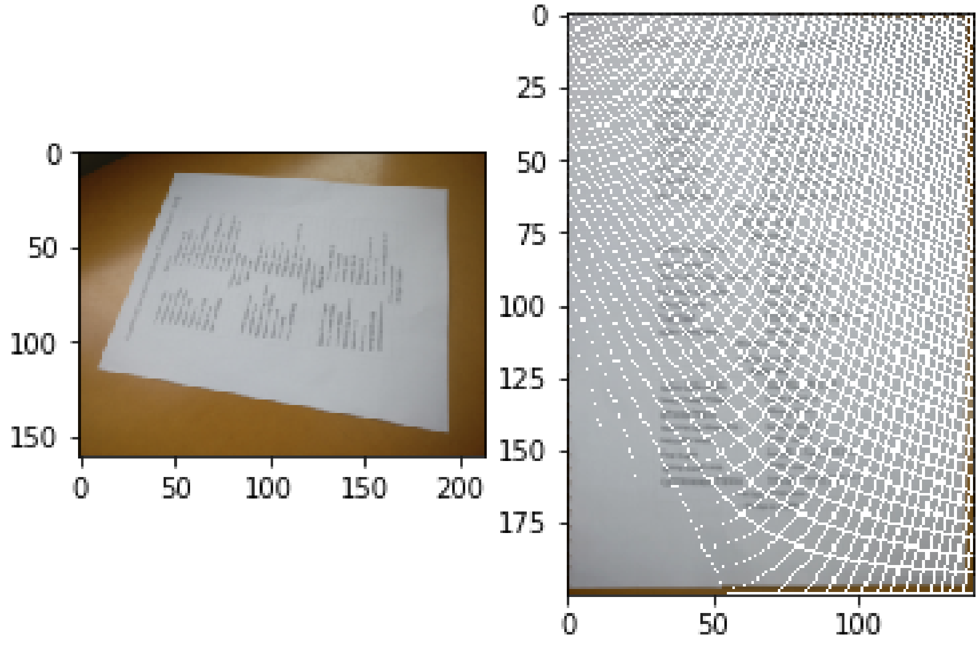


The red and blue quadrilaterals should overlap if ComputeH is implemented correctly.

```
In [4]: def warp(source_img, source_points, target_size):
            # Create a target image and select target points to create a homography
            # from source image to target image, in other words map all source points
            # to target points and then create,  a warped version of the image based
            # on the homography by filling in the target image.
            # Make sure the new image (of size target_size) has the same number of
            # color channels as source image

            assert target_size[2]==source_img.shape[2]
            # create an empty target image which we will starti filling
            target_image = np.ones(target_size) / 1.

            # selecting the corner points in the target image
            x_coords = [0, target_size[1]-1, target_size[1]-1, 0]
            y_coords = [0, 0, target_size[0]-1, target_size[0]-1]
            target_points = np.vstack((x_coords, y_coords))

            # compute H
            H = computeH(to_homog(source_points), to_homog(target_points))

            # Now map the points from source to target and
            # fill the pixel values in target image
            # x' = Hx
```

```
    for x in range(0, source_image.shape[1]):
        for y in range(0, source_image.shape[0]):
            # convert to homogeneous coordinate
            xy_homo = to_homog(np.array([[x, y]]).T)
            # apply homography
            mapped_x, mapped_y = from_homog(np.matmul(H,xy_homo)).T[0]
            if (0 <= mapped_x <= target_size[1]) and \
             (0 <= mapped_y <= target_size[0]):
                # Map only points in source image for which calculated
                # target points lie on targe image plane
                target_image[int(mapped_y), int(mapped_x), :] = source_img[y, x, :]

    # playing with interpolation, trying to remove noise
    target_image = imresize(target_image, target_size, interp='bilinear')

    return target_image


# Use the code below to plot your result
# Result with smaller noise and strains due to
# smaller size of target image
result = warp(source_image, source_points, (150,100,3))
plt.subplot(1, 2, 1)
plt.imshow(source_image)
plt.subplot(1, 2, 2)
plt.imshow(result)
plt.show()

# Use the code below to plot your result
# Result with larger noise and strains than before due to
# larger size of target image
result = warp(source_image, source_points, (200,140,3))
plt.subplot(1, 2, 1)
plt.imshow(source_image)
plt.subplot(1, 2, 2)
plt.imshow(result)
plt.show()
```



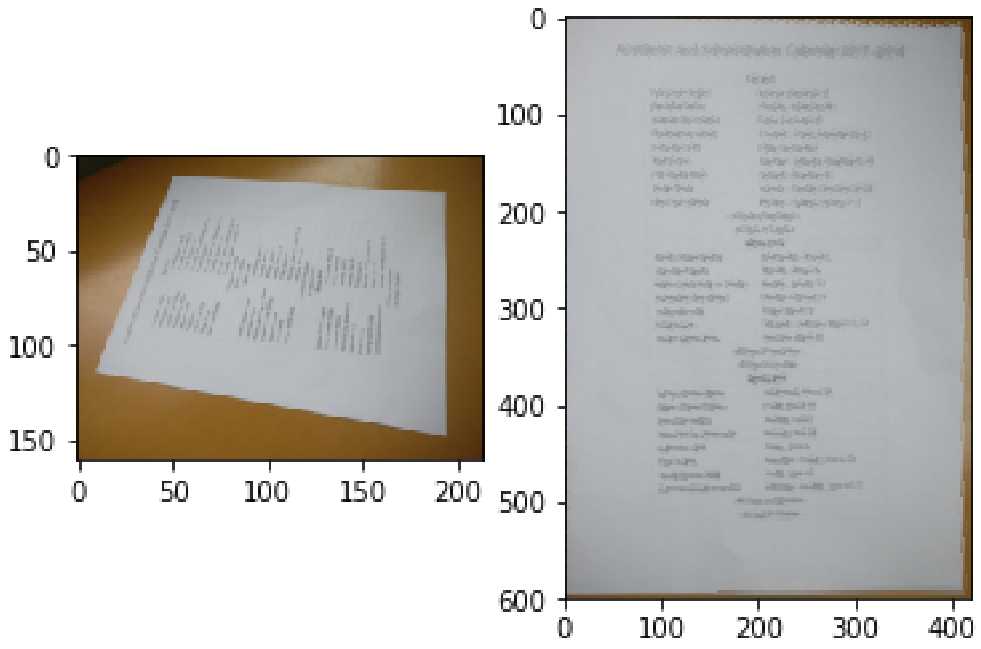


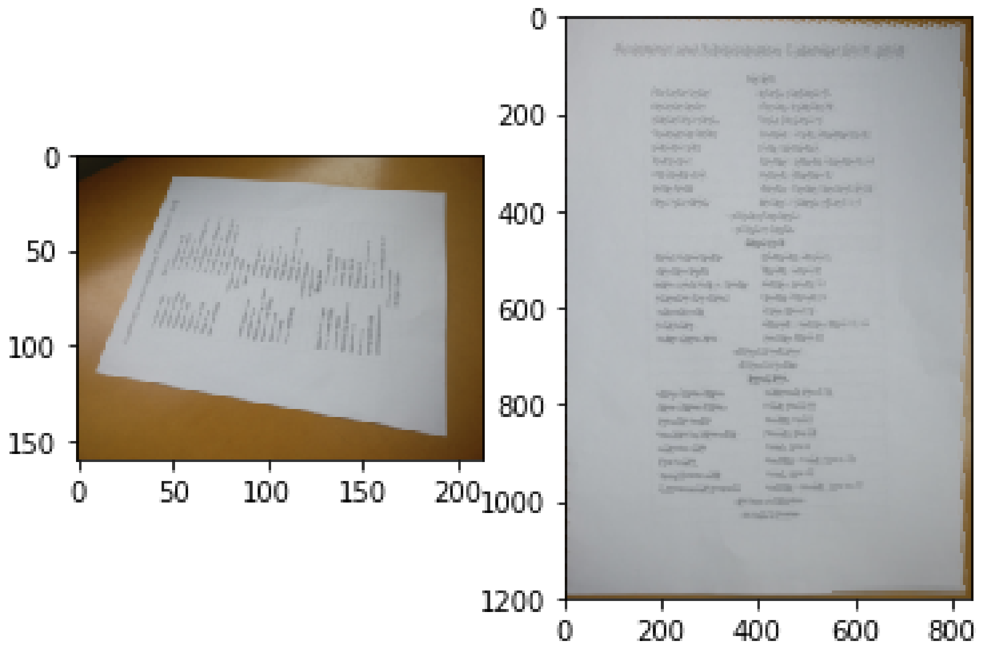
The output of warp1 of your code probably has some striations or noise. The larger you make your target image, the less it will resemble the document in the source image. Why is this happening?

Answer: This is due to the fact that there are some points in target image plane for which there is no mapping to source points on source image plane. Increasing the size of target image increases the number of such points.

To fix this, implement warp2, by creating an inverse homography matrix and fill in the target image.

```
In [5]: def warp2(source_img, source_points, target_size):
            # Create a target image and select target points to create a homography
            # from target image to source image, in other words map each
            # target point to a source point, and then create a warped version
            # of the image based on the homography by filling in the target image.
            # Make sure the new image (of size target_size) has the same number
            # of color channels as source image

            assert target_size[2]==source_img.shape[2]
            # create an empty target image which we will starti filling
            target_image = np.ones(target_size) / 1.

            # selecting the corner points in the target image
            x_coords = [0, target_size[1]-1, target_size[1]-1, 0]
            y_coords = [0, 0, target_size[0]-1, target_size[0]-1]
            target_points = np.vstack((x_coords, y_coords))

            # compute H
            H = computeH(to_homog(target_points), to_homog(source_points))

            # Now map the points from target to source and
            # fill the pixel values in target image
            # x' = Hx
            for x in range(0, target_image.shape[1]):
                for y in range(0, target_image.shape[0]):
                    # convert to homogeneous coordinate
                    xy_homo = to_homog(np.array([[x, y]]).T)
                    # apply homography
                    mapped_x, mapped_y = from_homog(np.matmul(H,xy_homo)).T[0]

                    target_image[y, x, :] = source_img[int(mapped_y), int(mapped_x), :]

            return target_image


        size_factor = 1
        # Use the code below to plot your result
        result = warp2(source_image, source_points, (600*size_factor,420*size_factor,3))
        plt.subplot(1, 2, 1)
        plt.imshow(source_image)
        plt.subplot(1, 2, 2)
        plt.imshow(result)
        plt.show()

        size_factor = 2
        # Use the code below to plot your result
        result = warp2(source_image, source_points, (600*size_factor,420*size_factor,3))
        plt.subplot(1, 2, 1)
        plt.imshow(source_image)
        plt.subplot(1, 2, 2)
        plt.imshow(result)
        plt.show()
```





Try playing around with the size of your target image in warp1 versus in warp2, additionally you can also implement nearest pixel interpolation or bi-linear interpolations and see if that makes a difference in your output.

In warp3, you'll be replacing the document in your image with a provided image. Read in "ucsd_logo.png" as the source image, keeping your document as the target.

```
In [6]: # Load the given UCSD logo image
        source_image2 = imread('ucsd_logo.png')[:,:,:3]/255.

        def warp3(target_image, target_points, source_image):
            # check for same channels
            assert target_image.shape[2]==source_image.shape[2]
            source_size = source_image.shape

            # selecting the corner points in source size
            x_coords = [0, source_size[1]-1, source_size[1]-1, 0]
            y_coords = [0, 0, source_size[0]-1, source_size[0]-1]
            source_points = np.vstack((x_coords, y_coords))

            # compute H
            H = computeH(to_homog(source_points), to_homog(target_points))

            # Now fill the elements in target image
            for x in range(0, source_image.shape[1]):
                for y in range(0, source_image.shape[0]):
                    # convert to homogeneous coordinate
            xy_homo = to_homog(np.array([[x, y]]).T)
            # apply homography
            mapped_x, mapped_y = from_homog(np.matmul(H,xy_homo)).T[0]

            target_image[int(mapped_y), int(mapped_x), :] = source_image[x, y, :]

    return target_image

# Use the code below to plot your result
result = warp3(source_image, source_points, source_image2)
plt.subplot(1, 2, 1)
plt.imshow(source_image2)
plt.subplot(1, 2, 2)
plt.imshow(result)
plt.show()
```