prob3

September 20, 2019

1 EECS16A: Homework 3

1.1 Problem 4: Image Stitching

This section of the notebook continues the image stiching problem. Be sure to have a `figures` folder in the same directory as the notebook. The `figures` folder should contain the files:

```
Berkeley_banner_1.jpg
Berkeley_banner_2.jpg
stacked_pieces.jpg
lefthalfpic.jpg
righthalfpic.jpg
```

Note: This structure is present in the provided HW3 zip file.
Run the next block of code before proceeding

[1]:
```
import numpy as np
import numpy.matlib
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from numpy import pi, cos, exp, sin
import matplotlib.image as mpimg
import matplotlib.transforms as mtransforms


%matplotlib inline

#loading images
image1=mpimg.imread('figures/Berkeley_banner_1.jpg')
image1=image1/255.0
image2=mpimg.imread('figures/Berkeley_banner_2.jpg')
image2=image2/255.0
image_stack=mpimg.imread('figures/stacked_pieces.jpg')
image_stack=image_stack/255.0


image1_marked=mpimg.imread('figures/lefthalfpic.jpg')
image1_marked=image1_marked/255.0
```

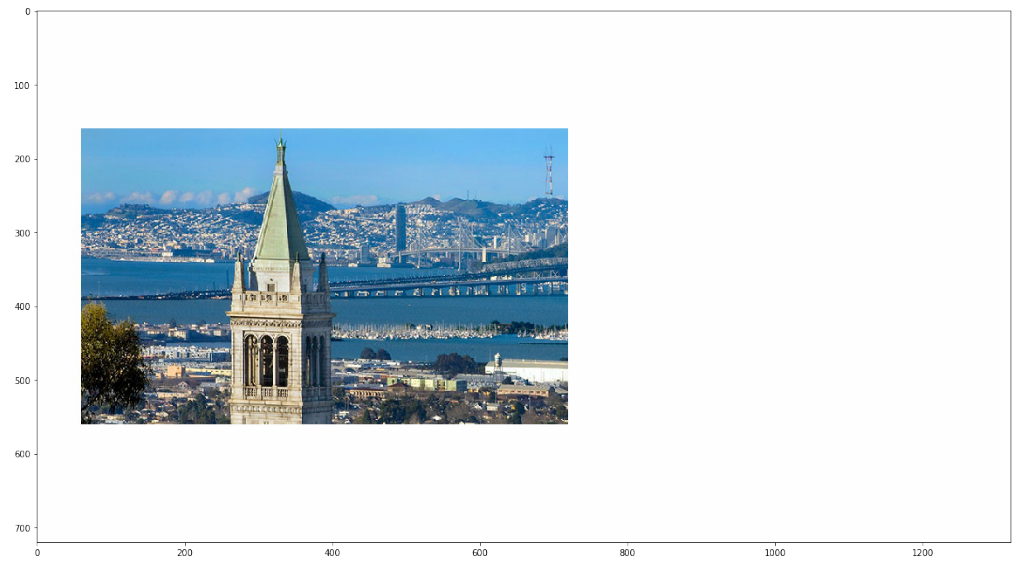
```
image2_marked=mpimg.imread('figures/righthalfpic.jpg')
image2_marked=image2_marked/255.0

def euclidean_transform_2to1(transform_mat,translation,image,position,LL,UL):
    new_position=np.round(transform_mat.dot(position)+translation)
    new_position=new_position.astype(int)


    if (new_position>=LL).all() and (new_position<UL).all():
        values=image[new_position[0][0],new_position[1][0],:]
    else:
        values=np.array([2.0,2.0,2.0])

    return values

def euclidean_transform_1to2(transform_mat,translation,image,position,LL,UL):
    transform_mat_inv=np.linalg.inv(transform_mat)
    new_position=np.round(transform_mat_inv.dot(position-translation))
    new_position=new_position.astype(int)

    if (new_position>=LL).all() and (new_position<UL).all():
        values=image[new_position[0][0],new_position[1][0],:]
    else:
        values=np.array([2.0,2.0,2.0])

    return values

def solve(A,b):
    try:
        z = np.linalg.solve(A,b)
    except:
        raise ValueError('Rows are not linearly independent. Cannot solve␣
 ↪system of linear equations uniquely. :)')
    return z
```

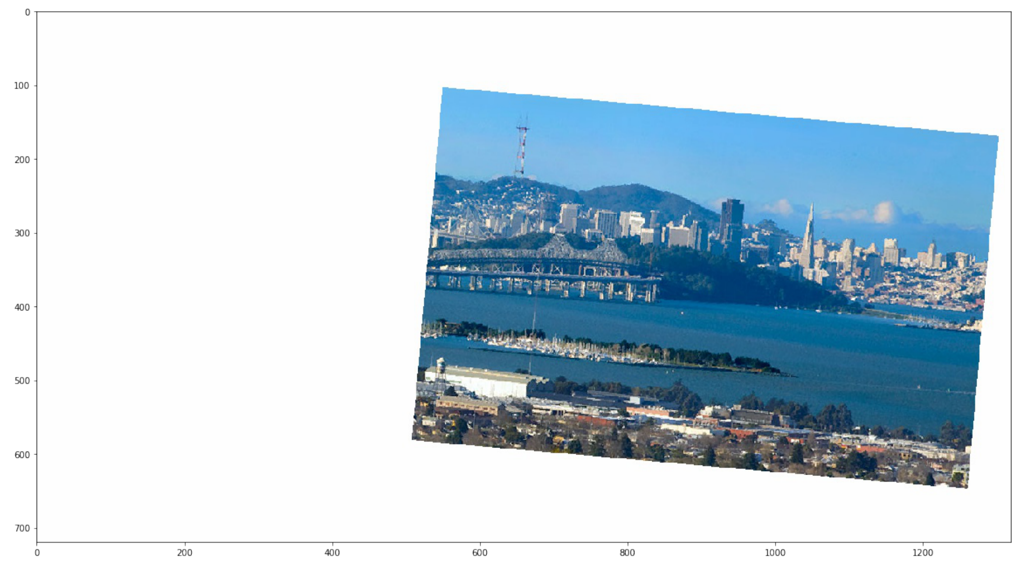
We will stick to a simple example and just consider stitching two images (if you can stitch two pictures, then you could conceivably stitch more by applying the same technique over and over again).

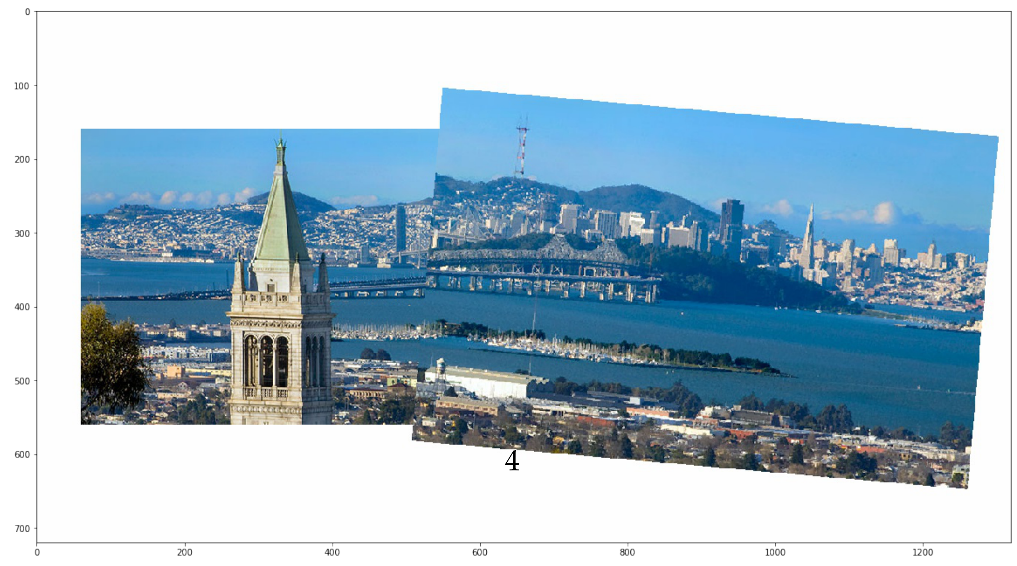
Daniel decided to take an amazing picture of the Campanile overlooking the bay. Unfortunately, the field of view of his camera was not large enough to capture the entire scene, so he decided to take two pictures and stitch them together.

The next block will display the two images.

[2]:
```
plt.figure(figsize=(20,40))

plt.subplot(311)
plt.imshow(image1)
plt.subplot(312)
plt.imshow(image2)

plt.subplot(313)
plt.imshow(image_stack)

plt.show()
```



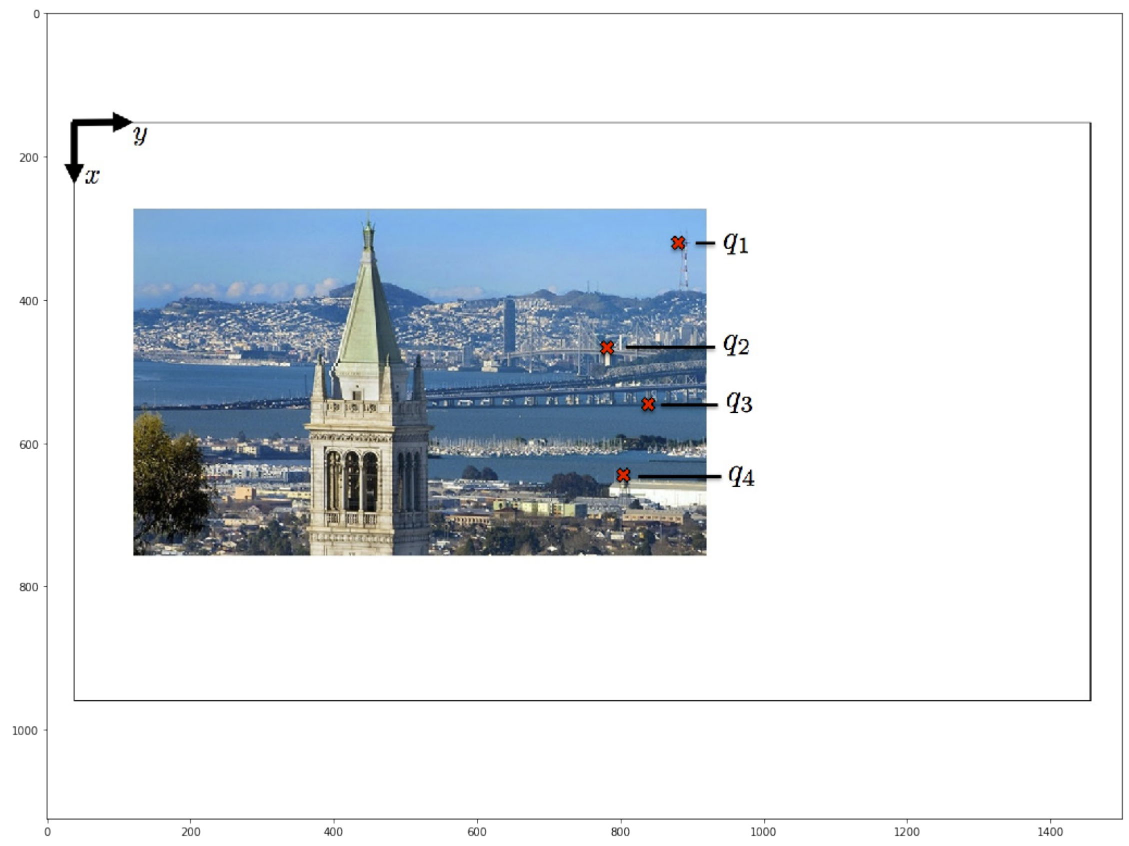




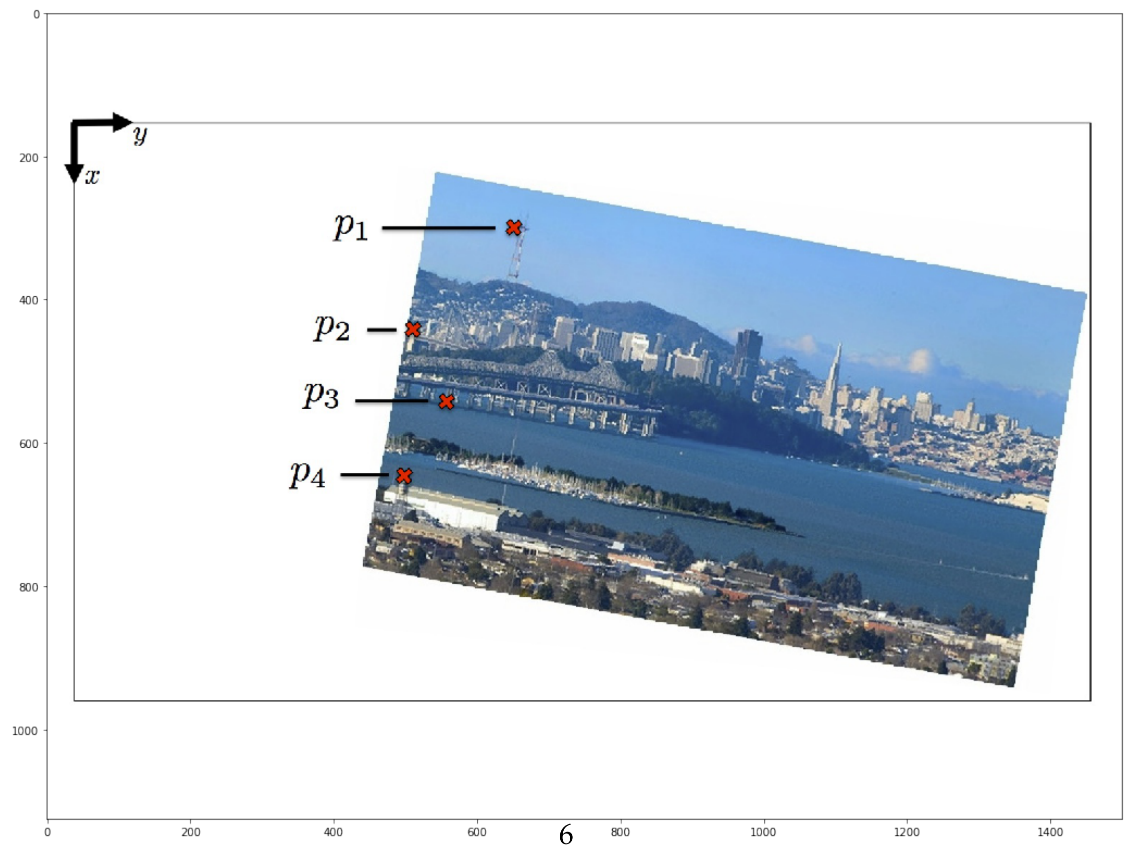
Once you apply Marcela's algorithm on the two images you get the following result (run the next block):

[3]:
```
plt.figure(figsize=(20,30))

plt.subplot(211)
plt.imshow(image1_marked)

plt.subplot(212)
plt.imshow(image2_marked)
```

[3]:
```
<matplotlib.image.AxesImage at 0x7f5abf2d3cc0>
```



0
200
400
600
800
1000
0
200
400
600
800
1000
1200
1400
y
x
q_1
q_2
q_3
q_4



0
200
400
600
800
1000
0
200
400
600
800
1000
1200
1400
y
x
p_1
p_2
p_3
p_4

As you can see Marcela's algorithm was able to find four common points between the two images. These points expressed in the coordinates of the first image and second image are

$$\vec{p}_1 = \begin{bmatrix} 200 \\ 700 \end{bmatrix} \qquad \vec{p}_2 = \begin{bmatrix} 310 \\ 620 \end{bmatrix} \qquad \vec{p}_3 = \begin{bmatrix} 390 \\ 660 \end{bmatrix} \qquad \vec{p}_4 = \begin{bmatrix} 460 \\ 630 \end{bmatrix} \tag{1}$$

$$\vec{q}_1 = \begin{bmatrix} 162.2976 \\ 565.8862 \end{bmatrix} \qquad \vec{q}_2 = \begin{bmatrix} 285.4283 \\ 458.7469 \end{bmatrix} \qquad \vec{q}_3 = \begin{bmatrix} 385.2465 \\ 498.1973 \end{bmatrix} \qquad \vec{q}_4 = \begin{bmatrix} 465.7892 \\ 455.0132 \end{bmatrix} \tag{2}$$

It should be noted that in relation to the image the positive x-axis is down and the positive y-axis is right. This will have no bearing as to how you solve the problem, however it helps in interpreting what the numbers mean relative to the image you are seeing.

Using the points determine the parameters $R_{11}, R_{12}, R_{21}, R_{22}, T_x, T_y$ that map the points from the first image to the points in the second image by solving an appropriate system of equations. Hint: you do not need all the points to recover the parameters.

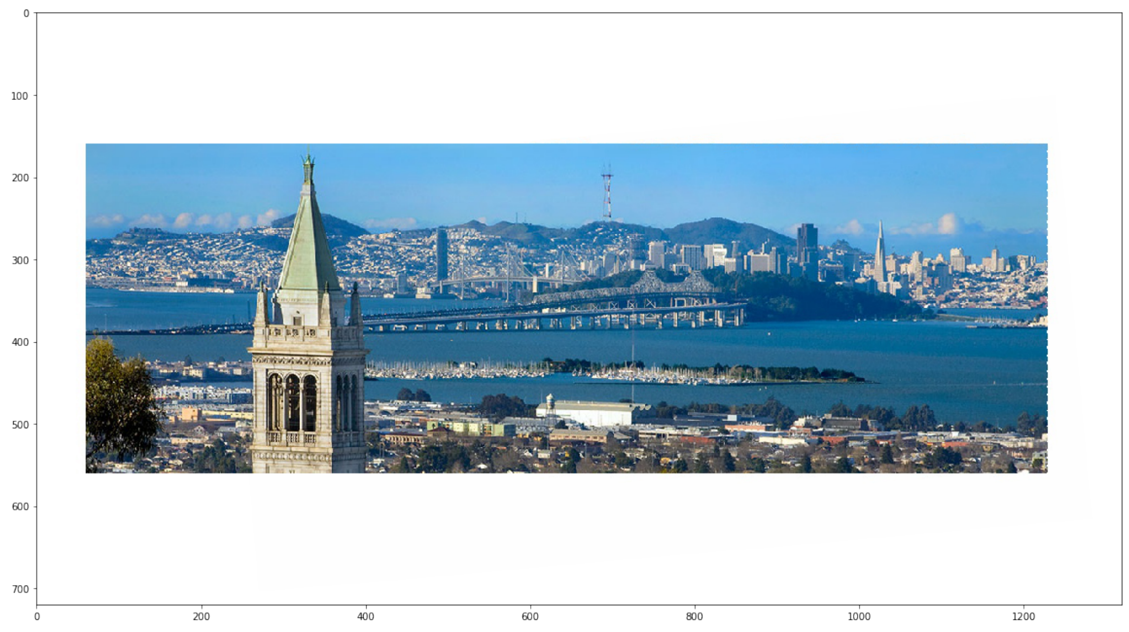
[4]:
```
# Note that the following is a general template for solving for 6 unknowns from␣
↪6 equations represented as Az = b.
# You do not have to use the following code exactly.
# All you need to do is to find parameters R_11, R_12, R_21, R_22, T_x, T_y.
# If you prefer finding them another way it is fine.

# fill in the entries
A = np.array([[200,700,0,0,1,0],
              [0,0,200,700,0,1],
              [310,620,0,0,1,0],
              [0,0,310,620,0,1],
              [390,660,0,0,1,0],
              [0,0,390,660,0,1]])

# fill in the entries
b = np.array([[162.2976],[565.8862],[285.4283],[458.7469],[385.2465],[498.
↪1973]])

A = A.astype(float)
b = b.astype(float)

# solve the linear system for the coefficiens
z = solve(A,b)

#Parameters for our transformation
R_11 = z[0,0]
R_12 = z[1,0]
R_21 = z[2,0]
R_22 = z[3,0]
T_x  = z[4,0]
T_y  = z[5,0]
```

Stitch the images using the transformation you found by running the code below.

1.1.1 Note that it takes about 40 seconds for the block to finish running on a modern laptop.

[5]:
```
matrix_transform=np.array([[R_11,R_12],[R_21,R_22]])
translation=np.array([T_x,T_y])

#Creating image canvas (the image will be constructed on this)
num_row,num_col,blah=image1.shape
image_rec=1.0*np.ones((int(num_row),int(num_col),3))

#Reconstructing the original image

LL=np.array([[0],[0]]) #lower limit on image domain
UL=np.array([[num_row],[num_col]]) #upper limit on image domain

for row in range(0,int(num_row)):
    for col in range(0,int(num_col)):
        #notice that the position is in terms of x and y, so the c
        position=np.array([[row],[col]])
        if image1[row,col,0] > 0.995 and image1[row,col,1] > 0.995 and␣
 ↪image1[row,col,2] > 0.995:
            temp =␣
 ↪euclidean_transform_2to1(matrix_transform,translation,image2,position,LL,UL)
            image_rec[row,col,:] = temp
        else:
            image_rec[row,col,:] = image1[row,col,:]


plt.figure(figsize=(20,20))
plt.imshow(image_rec)
plt.axis('on')
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



1.1.2 Part E: Failure Mode Points

$$\vec{p}_1 = \begin{bmatrix} 390 \\ 660 \end{bmatrix} \qquad \vec{p}_2 = \begin{bmatrix} 425 \\ 645 \end{bmatrix} \qquad \vec{p}_3 = \begin{bmatrix} 460 \\ 630 \end{bmatrix} \tag{3}$$

$$\vec{q}_1 = \begin{bmatrix} 385 \\ 450 \end{bmatrix} \qquad \vec{q}_2 = \begin{bmatrix} 425 \\ 480 \end{bmatrix} \qquad \vec{q}_3 = \begin{bmatrix} 465 \\ 510 \end{bmatrix} \tag{4}$$

[5]:
```
# Note that the following is a general template for solving for 6 unknowns from
 ↪6 equations represented as Az = b.
# You do not have to use the following code exactly.
# All you need to do is to find parameters R_11, R_12, R_21, R_22, T_x, T_y.
# If you prefer finding them another way it is fine.

# fill in the entries
A = np.array([[390,660,0,0,1,0],
              [0,0,390,660,0,1],
              [425,645,0,0,1,0],
              [0,0,425,645,0,1],
              [460,630,0,0,1,0],
              [0,0,460,630,0,1]])

# fill in the entries
b = np.array([[385],[450],[425],[480],[465],[510]])

A = A.astype(float)
b = b.astype(float)
```

```
# solve the linear system for the coefficiens
z = solve(A,b)

#Parameters for our transformation
R_11 = z[0,0]
R_12 = z[1,0]
R_21 = z[2,0]
R_22 = z[3,0]
T_x  = z[4,0]
T_y  = z[5,0]
```

```
      ␣
↪---------------------------------------------------------------------------

       LinAlgError                               Traceback (most recent call␣
↪last)

       <ipython-input-1-4ee2c81e8dee> in solve(A, b)
        51     try:
   ---> 52         z = np.linalg.solve(A,b)
        53     except:


       ~/anaconda3/lib/python3.7/site-packages/numpy/linalg/linalg.py in␣
↪solve(a, b)
       402     extobj = get_linalg_error_extobj(_raise_linalgerror_singular)
   --> 403     r = gufunc(a, b, signature=signature, extobj=extobj)
       404


       ~/anaconda3/lib/python3.7/site-packages/numpy/linalg/linalg.py in␣
↪_raise_linalgerror_singular(err, flag)
        96 def _raise_linalgerror_singular(err, flag):
   ---> 97     raise LinAlgError("Singular matrix")
        98


       LinAlgError: Singular matrix


   During handling of the above exception, another exception occurred:


       ValueError                                Traceback (most recent call␣
↪last)
```

```
    <ipython-input-5-48bdfaeadea9> in <module>
     19
     20 # solve the linear system for the coefficiens
---> 21 z = solve(A,b)
     22
     23 #Parameters for our transformation


    <ipython-input-1-4ee2c81e8dee> in solve(A, b)
     52         z = np.linalg.solve(A,b)
     53     except:
---> 54         raise ValueError('Rows are not linearly independent. Cannot solve system of linear equations uniquely. :)')
     55     return z


    ValueError: Rows are not linearly independent. Cannot solve system of linear equations uniquely. :)
```