EE5175 - Lab 3

Image mosaicing

Nikilesh B
EE17B112

Image mosaicing is the alignment and stitching of a collection of images having overlapping regions into a single image.

Here, we have been given three images which were captured by panning the scene left to right. These images (img1.png, img2.png and img3.png) capture overlapping regions of the same scene from different viewpoints. The images are attached below :

Img1

Img2

Img3

Our task is to determine the geometric transformations (homographies) between these images and stitch them into a single image.

The **steps** followed are as shown below :

1. Taking img2.png as the reference image.

2. Determining the homography H_{21} between I_2 = img2.png and I_1 = img1.png such that $I_1 = H_{21}\, I_2$.

3. Determine homography H_{23} between I_2 = img2.png and I_3 = img3.png such that $I_3 = H_{23}\, I_2$.

4. Create an empty canvas. For every pixel in the canvas, and corresponding points in I_1, I_2 and I_3 using H_{21}, identity matrix and H_{23} respectively (target-to-source mapping). Blending The three values by averaging them. Employing the values in blending only if it falls within the corresponding image bounds.

Determining homography between two images:

- Determining SIFT features of the two images and determining correspondences between them. File sift_corresp.m returns the SIFT correspondences between two images.
- Running the RANSAC on matched points (correspondences) to remove outliers (wrong matches), and find the homography between the two images.

  The code for ransac is attached below.

```
def ransac(corresp1,corresp2) :
      frac = 0  ; niter = 0
      while(frac <= 0.99) :
            Lmp = len(corresp1)    #Generate 4 random numbers from the set
            r = rn.sample(range(0,Lmp),4)
            a = [corresp1[r[i]] for i in range(0,len(r))]
            b = [corresp2[r[i]] for i in range(0,len(r))]
            #Take these 4 points and find homography#Fill in the matrix
            vsize = 9 ; eqns = 4
            A = np.zeros((int(2*eqns),vsize))
            for i in range(0,eqns) :        #Loop to fill in the values
            A[int(2*i)][0] = b[i][0]
            A[int(2*i)][1] = b[i][1]
            A[int(2*i)][2] = 1
            A[int(2*i)][3] = 0
```

```
            A[int(2*i)][4] = 0
            A[int(2*i)][5] = 0
            A[int(2*i)][6] = -b[i][0]*a[i][0]
            A[int(2*i)][7] = -b[i][1]*a[i][0]
            A[int(2*i)][8] = -a[i][0]

            A[int(2*i)+1][0] = 0
            A[int(2*i)+1][1] = 0
            A[int(2*i)+1][2] = 0
            A[int(2*i)+1][3] = b[i][0]
            A[int(2*i)+1][4] = b[i][1]
            A[int(2*i)+1][5] = 1
            A[int(2*i)+1][6] = -b[i][0]*a[i][1]
            A[int(2*i)+1][7] = -b[i][1]*a[i][1]
            A[int(2*i)+1][8] = -a[i][1]

            h = null_space(A)     #Find nullspace of the matrix
            H = h.reshape((3,3))      #Put h in order

            C = []     #Check with remaining points and see fraction
            iterset = list(set(np.arange(0,Lmp)).difference(r))
            bvec = np.zeros((3,1))
            avec = np.zeros((2,1))
            eps = 10
            #12 0.75 good  #16 0.75 works
            bvec[2] = 1
            for item in iterset :
            bvec[0] = corresp2[item][0]
            bvec[1] = corresp2[item][1]

            atemp = H @ bvec
            avec[0] = atemp[0]/atemp[2]
            avec[1] = atemp[1]/atemp[2]

            dist = np.sqrt(pow(corresp1[item][0]-avec[0],2) +
pow(corresp1[item][1]-avec[1],2))
            if dist < eps :
                  C.append(item)
             #Check how good in the consensus set
            frac = len(C)/len(iterset)
            niter = niter+1
      return H,frac,niter,C
```

- Running SIFT and obtaining matching key points.

```
correspa1 = []
correspa2 = []
with open("corresp_mosaic2_1.csv") as csvfile:
      reader = csv.reader(csvfile, quoting=csv.QUOTE_NONNUMERIC) # change contents to floats
      for row in reader: # each row is a list
      correspa1.append(row[:2])
      correspa2.append(row[2:])
correspa1 = np.array(correspa1)
correspa2 = np.array(correspa2)
```

- Creating the mosaic

```
nrows = src2.shape[0]
ncolumns = src1.shape[1] + src2.shape[1] + src3.shape[1]
print(src1.shape[1],src2.shape[1])

canvas = np.zeros((nrows,ncolumns))
countcnvs = np.zeros((nrows,ncolumns))

canvas1 = bilnr(src1, H1, nrows, ncolumns)
canvas2 = bilnr(src2, np.identity(3), nrows, ncolumns)
canvas3 = bilnr(src3, H3, nrows, ncolumns)
```

- We use **bilinear interpolation** ;

```
def bilnr(src, H, rows, cols) :
      #Creating vector to multiply Hinv
      x = list(np.arange(0,rows))
      x = x*cols
      x = np.array(x)
      x = x.reshape(cols,rows)
      x = x.T
      x = x.reshape(int(rows*cols),1)
      y = list(np.arange(0,cols))
      y = y*rows
      y = np.array(y)
      y = y.reshape(int(rows*cols),1)
```

```
    o = np.ones((int(rows*cols),1))
    xy = np.array([x,y,o])
    xy = xy.T
    xy = xy[0]
    xy = xy.T
    xy[1] = xy[1] - cenx

    #Target to source mapping
    xy_temp = np.linalg.inv(H)@ xy
    x = xy_temp[0]/xy_temp[2]
    y = xy_temp[1]/xy_temp[2]

    xf = x.astype(int)
    yf = y.astype(int)

  #distance from pixel
    a = x-xf
    b = y-yf

    Ival = np.zeros(xf.shape)
    #Find intensity
    for i in range(0,len(xf)) :
          if xf[i] < src.shape[0]-1 and yf[i] < src.shape[1]-1 and
xf[i]>0 and yf[i]>0 :
          Ival[i] = (1-a[i])*(1-b[i])*src[xf[i]][yf[i]] +
(1-a[i])*(b[i])*src[xf[i]][yf[i]+1] + (a[i])*(1-b[i])*src[xf[i]+1][yf[i]] +
(a[i])*(b[i])*src[xf[i]+1][yf[i]+1]

    Ival = Ival.reshape(rows,cols)
    return Ival
```

Atlast the output image obtained is attached below:

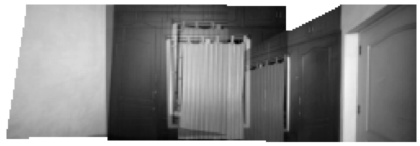
Capturing our own data:

Using our mobile phone camera to capture images (three or more), and running our code to generate the mosaic. We also ensure that there is adequate overlap between successive images, and the camera is imaging a far-away scene .

We bring down the resolution of the input images such that the width < 1000 pixels, and convert them to grayscale before using them for mosaicing.

The three images are attached below:

After running through the code the image obtained is attached below:



Observation:

The output image is distorted because the image was captured in a relatively closer distance and relatively had less overlapping regions.