Machine Vision Coursework 2

gonzalo.ayquipa.16

January 2018

1 Homographies Part I

1.1 A. Practical 1A

First we define a set of two dimensional Cartesian points (5 examples), we then converted those points into their homogeneous representation (which means padding with 1s the last row), so now our points array have the shape of 3x5.

For the toy example, the homography matrix is given (shape of 3x3), so we just need to apply this matrix to the points (in the homogeneous representation) to get the new points (pts2Hom in code). Then, we proceeded to convert back to Cartesian coordinates and add a small noise. We plot the first and the second set of points

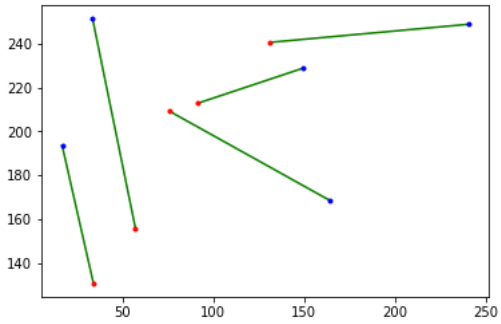

Figure 1: Homography example

In the plot above, the red points are the original points and the magenta points are the set after the homography transformation and the noise addition.

a. First TODO: We were tasked to complete the **calBestHomography**

```
def calcBestHomography(pts1Cart, pts2Cart):

    # TO DO:
    # First convert points into homogeneous representation
```

```
    pts1Hom = np.concatenate((pts1Cart, np.ones((1,pts1Cart.shape[1]))), axis
    =0)
    pts2Hom = np.concatenate((pts2Cart, np.ones((1,pts2Cart.shape[1]))), axis
    =0)

    # Then construct the matrix A, size (n_points,9)
    num_examples = 10
    A = np.zeros((num_examples,9))

    #Populate the matrix
    for i in range(5):
        uv = pts1Hom[:, i]
        xy = pts2Hom[:, i]
        A[2 * i,     :] = np.hstack([[0, 0, 0], -uv.T, uv.T * xy[1]])
        A[2 * i + 1, :] = np.hstack([uv.T, [0, 0, 0], -uv.T * xy[0]])

    # Solve Ah = 0
    h = solveAXEqualsZero(A)

    # Reshape h into the matrix H, values of h go first into rows of H
    H = np.reshape(h, (3, 3))

    return H

```

Listing 1: calcBestHomography

As we can see in the code above, first we need to convert the points in their homogeneous representation. Then we create a matrix of size 10x9, in this case the number of examples per set of points is 5, that is why we need 10 rows. After that, we run the script to populate the matrix according to the form below:

$$\begin{bmatrix} 0 & 0 & 0 & -u_1 & -v_1 & -1 & y_1u_1 & y_1v_1 & y_1 \\ u_1 & v_1 & 1 & 0 & 0 & 0 & -x_1u_1 & -x_1v_1 & -x_1 \\ 0 & 0 & 0 & -u_2 & -v_2 & -1 & y_2u_2 & y_2v_2 & y_2 \\ u_2 & v_2 & 1 & 0 & 0 & 0 & -x_2u_2 & -x_2v_2 & -x_2 \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ 0 & 0 & 0 & -u_I & -v_I & -1 & y_Iu_I & y_Iv_I & y_I \\ u_I & v_I & 1 & 0 & 0 & 0 & -x_Iu_I & -x_Iv_I & -x_I \end{bmatrix}$$

We then run the script **solveAXEqualsZero**, which is described below. This function return a column of 9 elements (shape of 9x1). Then, we reshape the column in order to have a matrix similar to the one below:

$$\begin{bmatrix} \phi_{11} & \phi_{12} & \phi_{13} \\ \phi_{21} & \phi_{22} & \phi_{23} \\ \phi_{31} & \phi_{32} & \phi_{33} \end{bmatrix}$$

```
def solveAXEqualsZero(A):
    # TO DO: Write this routine - it should solve Ah = 0
    u,s,v = np.linalg.svd(A)
    v_last = v[-1,:]
    return v_last

```

Listing 2: solveAXEqualsZero

b. Second TODO: As we can see in the code above, we pass the matrix of 10X9 to the function as an argument, and then perform SVD, returning only the last column of V. This happens since we have the following conditions:

$$\begin{bmatrix} 0 & 0 & 0 & -u_1 & -v_1 & -1 & y_1u_1 & y_1v_1 & y_1 \\ u_1 & v_1 & 1 & 0 & 0 & 0 & -x_1u_1 & -x_1v_1 & -x_1 \\ 0 & 0 & 0 & -u_2 & -v_2 & -1 & y_2u_2 & y_2v_2 & y_2 \\ u_2 & v_2 & 1 & 0 & 0 & 0 & -x_2u_2 & -x_2v_2 & -x_2 \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ 0 & 0 & 0 & -u_I & -v_I & -1 & y_Iu_I & y_Iv_I & y_I \\ u_I & v_I & 1 & 0 & 0 & 0 & -x_Iu_I & -x_Iv_I & -x_I \end{bmatrix} \begin{bmatrix} \phi_{11} \\ \phi_{12} \\ \phi_{13} \\ \phi_{21} \\ \phi_{22} \\ \phi_{23} \\ \phi_{31} \\ \phi_{32} \\ \phi_{33} \end{bmatrix} = 0$$

So we have the form:

$$\mathbf{A}\phi = 0$$

Now in the code below, first we call the function **calBestHomography** giving both set of points in cartesian coordinates. This function will return the homography matrix (shape 3x3). After doing that, we will apply the estimated homography matrix to set of points 1 (**pts1Hom** in homogeneous representation) and will estimate the second set of points in the homogeneous representation. In order to plot both set of points we convert the second estimated set of points (**pts2EstHom**) in cartesian coordinates.

```
# TO DO: Fill in the details of this function from above
HEst = calcBestHomography(pts1Cart, pts2Cart)
#print("size of 1hom {}".format(pts1Hom.shape))

# Apply estimated homography to points
pts2EstHom = np.matmul(HEst,pts1Hom)

# Convert back to Cartesian coordinates
pts2EstCart = pts2EstHom[0:2,:] / np.tile([pts2EstHom[2,:]],(2,1))

# Calculate the mean squared distance from actual points
sqDiff = np.mean(sum((pts2Cart - pts2EstCart)**2))

```

```
# Draw figure with points before and after applying the estimated homography
nPoint = pts1Cart.shape[1]

# Plot a green line between pairs of actual points (red) and estimated points (magenta)
for cPoint in range(0, nPoint):
    plt.plot([pts2Cart[0,cPoint], pts2EstCart[0,cPoint]], [pts2Cart[1,cPoint], pts2EstCart[1,cPoint]], 'g-')
    plt.plot(pts2Cart[0,cPoint],pts2Cart[1,cPoint],'r.',pts2EstCart[0,cPoint],pts2EstCart[1,cPoint],'m.')

plt.show()

```

Listing 3: Routine

In the plot below we see the second set of points (**pts2Cart**) and the second set of estimated points (**pts2EstCart**).

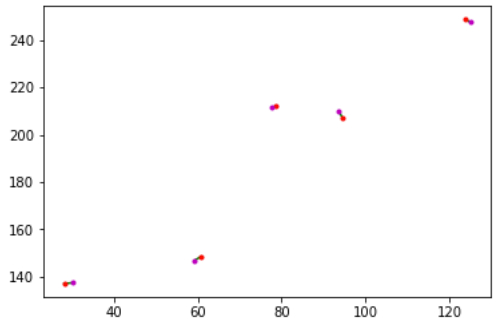

Figure 2: Estimated set of points

c. Third TODO: Convince yourself that the homography is ambiguous up to scale (by multiplying it by a constant factor and showing it does the same thing). Can you see why this is the case mathematically?

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \phi_{11} & \phi_{12} & \phi_{13} \\ \phi_{21} & \phi_{22} & \phi_{23} \\ \phi_{31} & \phi_{32} & \phi_{33} \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \phi_{11}u + \phi_{12}v + \phi_{13} \\ \phi_{21}u + \phi_{22}v + \phi_{23} \\ \phi_{31}u + \phi_{32}v + \phi_{33} \end{bmatrix}$$

$$\begin{bmatrix} \lambda x \\ \lambda \end{bmatrix} = \begin{bmatrix} \phi_{11}u + \phi_{12}v + \phi_{13} \\ \phi_{31}u + \phi_{32}v + \phi_{33} \end{bmatrix}$$

$$x = \frac{\phi_{11}u + \phi_{12}v + \phi_{13}}{\phi_{31}u + \phi_{32}v + \phi_{33}}$$

$$y = \frac{\phi_{21}u + \phi_{22}v + \phi_{23}}{\phi_{31}u + \phi_{32}v + \phi_{33}}$$

d. Show empirically that your homography routine can map any four points exactly to any other four points.

In order to prove this, we are going to comment out the lines related to the noise addition and we are going to run again all the routine.

```
# Add a small amount of noise
#noiseLevel = 4.0
#pts2Cart = pts2Cart + np.random.normal(0, noiseLevel, pts2Cart.shape)

```

Listing 4: Routine

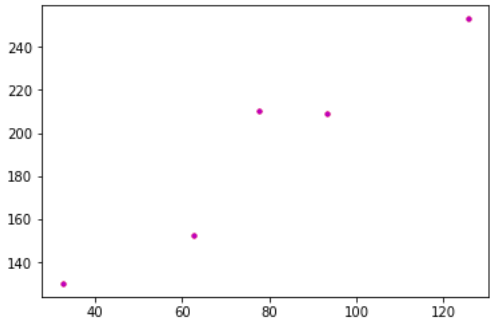

Figure 3: Estimated set of points without noise

As we can see in the figure above, there is no more lines, we mapped to the exactly the same point.

```
pts2EstCart

#Output:
array([[ 125.70884347,   32.48784502,   62.56729828,   77.593613  ,
          93.14243158],
       [ 253.02757825,  130.33247339,  152.58250426,  210.06357774,
         209.40250472]])

pts2Cart

```

```
#Output:
array([[ 125.70884347,   32.48784502,   62.56729828,   77.593613  ,
          93.14243158],
       [ 253.02757825,  130.33247339,  152.58250426,  210.06357774,
         209.40250472]])

```

Listing 5: Routine

1.2 B. Practical 1B

In this part of the lab, we were tasked to apply the homography to make a panorama of images that are related by a homography.

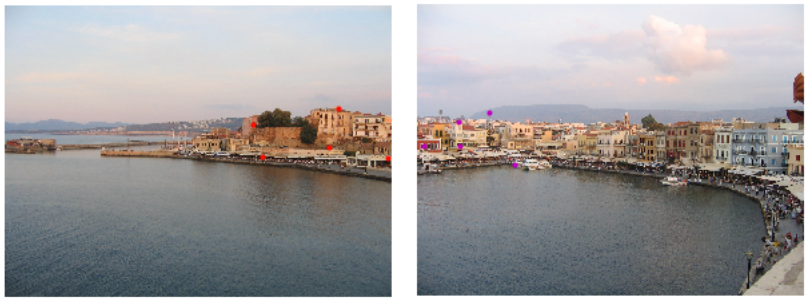

Figure 4: Photos, Image2(Left) Image3 (Right)

Figure 5: Photo to be used as a reference, Image1

We follow the instructions, first we calculate the homography matrix using the points of reference in image 1 and the points from image 2. Then we start a for loop over all the pixels in the image 1. We convert the cartesian coordinates of each pixel in their homogeneous representation and then apply the homography matrix. Once we did that, we need to convert back to cartesian and then check if the coordinates exist within the dimensions of image 1, if so, we copy the color values in the image 1.

```
# TO DO: Calculate homography from pts1 to pts2

HEst = calcBestHomography(pts1, pts2)

    #      Draw new image1

row_im1 = im1.shape[0]
col_im1 = im1.shape[1]
```

```
9
10 row_im2 = im2.shape[0]
11 col_im2 = im2.shape[1]
12
13 # TO DO:
14 # For every pixel in image 1:
15 for x in range(1,row_im1+1):
16
17     for y in range(1,col_im1+1):
18
19         #Every pixel in image 1
20         pts1Cart = np.array([[y],
21                              [x]])
22
23         # First convert points into homogeneous representation
24         pts1Hom = np.concatenate((pts1Cart, np.ones((1,pts1Cart.shape[1]))),
    axis=0)
25
26         # Apply estimated homography to each pixel
27         pts2EstHom = np.matmul(HEst,pts1Hom)
28
29         # Convert back to Cartesian coordinates
30         pts2EstCart = pts2EstHom[0:2,:] / np.tile([pts2EstHom[2,:]],(2,1))
31
32         # If it the transformed position is within the boundary of image 2:
33         if (int(np.round(pts2EstCart[1,0])) in range(1,row_im2+1)) and
34         (int(np.round(pts2EstCart[0,0])) in range(1,col_im2+1)):
35             #Copy pixel from image 2 to position in img1
36             i = int(np.round(pts2EstCart[1,0]))
37             j = int(np.round(pts2EstCart[0,0]))
38             im1[x-1,y-1,:] = im2[i-1,j-1,:]
39
```

Listing 6: Routine

We do the same for image 1 and image 3. The final results is below:

Figure 6: Estimated set of points

2 Homographies Part II

2.1 C. Practical 2A

This project explores the geometry of a single camera. The aim is to take several points on a plane, and predict where they will appear in the camera image. Based on these observed points, we will then try to re-estimate the Euclidean transformation relating the plane and the camera.
The goal of the code below is to project points in XCart through projective camera.

- 1. TODO: Converts the 3D XCart points to homogeneous coordinates (4-D having 1s in the last row)

$$\begin{bmatrix} u \\ v \\ w \\ 1 \end{bmatrix}$$

- 2. TODO: Apply the extrinsic matrix to Xhom to move the frame of reference of the camera (this is only is a matrix multiplication). The extrinsic matrix has the following form:

$$T = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \tau_x \\ w_{21} & w_{22} & w_{23} & \tau_y \\ w_{31} & w_{32} & w_{33} & \tau_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  After the multiplication, we have a matrix as follows:

$$\begin{bmatrix} w_{11}u + w_{12}v + w_{13}w + \tau_x \\ w_{21}u + w_{22}v + w_{23}w + \tau_y \\ w_{31}u + w_{32}v + w_{33}w + \tau_z \\ 0 + 0 + 0 + 1 \end{bmatrix}$$

- 3. TODO: Remove 4th row

$$\begin{bmatrix} w_{11}u + w_{12}v + w_{13}w + \tau_x \\ w_{21}u + w_{22}v + w_{23}w + \tau_y \\ w_{31}u + w_{32}v + w_{33}w + \tau_z \end{bmatrix}$$

- 4. TODO: Move points to image coordinates XImHom by applying intrinsic matrix. Below the intrinsic matrix is on the left.

$$\begin{bmatrix} \phi_x & \gamma & \delta_x \\ 0 & \phi_y & \delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w_{11}u + w_{12}v + w_{13}w + \tau_x \\ w_{21}u + w_{22}v + w_{23}w + \tau_y \\ w_{31}u + w_{32}v + w_{33}w + \tau_z \end{bmatrix}$$

- 5. TODO: Convert points back to Cartesian coordinates XImCart. According to the model we will have on the left the coordinates in homogeneous representation, so we have to convert them back to Cartesian.

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \phi_x & \gamma & \delta_x \\ 0 & \phi_y & \delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w_{11}u + w_{12}v + w_{13}w + \tau_x \\ w_{21}u + w_{22}v + w_{23}w + \tau_y \\ w_{31}u + w_{32}v + w_{33}w + \tau_z \end{bmatrix}$$

```
#The goal of this function is to project points in XCart through projective
    camera
#defined by intrinsic matrix K and extrinsic matrix T.
def projectiveCamera(K,T,XCart):


    # TO DO: Convert Cartesian 3d points XCart to homogeneous coordinates XHom
    # First convert points into homogeneous representation
    XHom = np.concatenate((XCart, np.ones((1,XCart.shape[1]))), axis=0)

    # TO DO: Apply extrinsic matrix to XHom, to move to frame of reference of
    camera
    XCamHom = np.matmul(T,XHom)

    # TO DO: Project points into normalized camera coordinates xCamHom (remove
     4th row)
    XCamHom_norm = XCamHom[:-1,:]

    # TO DO: Move points to image coordinates xImHom by applying intrinsic
    matrix
    xImHom = np.matmul(K,XCamHom_norm)

    # TO DO: Convert points back to Cartesian coordinates xImCart
    XImCart = xImHom[0:2,:] / np.tile([xImHom[2,:]],(2,1))

    return XImCart

```

Listing 7: Routine

Now we have to write code to compute the extrinsic matrix. In the code above we assume the extrinsic matrix (T) was given.

- 1. TODO: Converts the 3D XCart points to homogeneous coordinates (4-D having 1s in the last row)

$$\begin{bmatrix} u \\ v \\ w \\ 1 \end{bmatrix}$$

- 2. TODO: Convert image co-ordinates XImHom to normalized camera coordinates XCamHom. We need to multiply for the inverse of the intrinsic matrix:

$$x' = \Lambda^{-1} x$$

$$\lambda_i \begin{bmatrix} x_i' \\ y_i' \\ 1 \end{bmatrix} = \begin{bmatrix} \phi_{11} & \phi_{12} & \phi_{13} \\ \phi_{21} & \phi_{22} & \phi_{23} \\ \phi_{31} & \phi_{32} & \phi_{33} \end{bmatrix} \begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix}$$

Notice that we are using a **w** of elements u and v, instead of u,v,w. This is because this problem cannot be solved in closed form and requires optimization. So, we are going to consider the **w** as a vector of u and v, in order to perform the homography transformation seen in previous exercises.

- 3. TODO: Apply the extrinsic matrix to Xhom to move the frame of reference of the camera (this is only is a matrix multiplication). We use **caclBestHomography** in order to handle this, the output matrix has a shape of 3.3

- 4. TODO: Estimate the first two columns of rotation matrix R from the first two. We created a function (shown below) called **solve_rotation**, which will perform SVD of the first two columns we got in item 3.

$$\begin{bmatrix} \phi_{11}' & \phi_{12}' \\ \phi_{21}' & \phi_{22}' \\ \phi_{31}' & \phi_{32}' \end{bmatrix} = ULV^T$$

The matrix of ϕ is the homography matrix that returns in the item 3, a matrix of shape 3x3.

$$\begin{bmatrix} \omega_{11} & \omega_{12} \\ \omega_{21} & \omega_{22} \\ \omega_{31} & \omega_{32} \end{bmatrix} = U \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} V^T$$

After this, we will get the two columns of omegas at the left.

- 5. TODO: Estimate the third column of the rotation matrix by taking the cross product. We use **np.cross** to do this. No we have a matrix of shape 3x3. (rotation matrix)

$$\begin{bmatrix} \omega_{11} & \omega_{12} & \omega_{13} \\ \omega_{21} & \omega_{22} & \omega_{23} \\ \omega_{31} & \omega_{32} & \omega_{33} \end{bmatrix}$$

- 6. TODO: Check that the determinant of the rotation matrix is positive, if not then multiply last column by -1. We concatenate the first two columns with the third one, and the compute the determinant and state the conditions in the instructions.

- 7. TODO: Estimate the translation vector by finding the scaling factor. We compute the formula below:

$$\lambda' = \frac{\sum_{m=1}^{3}\sum_{n=1}^{2}\phi'_{mn}/\omega_{mn}}{6}$$

$$\tau = [\phi'_{13}, \phi'_{23}, \phi'_{33}]/\lambda'$$

- 8. TODO: Check whether the τ_z is negative, if so, multiply the vector t by -1 and the first two columns of R by -1.

- 9. TODO: Assemble, we concatenated the columns to return the extrinsic matrix. The output is a matrix as follows:

$$T = \begin{bmatrix} \omega_{11} & \omega_{12} & \omega_{13} & \tau_x \\ \omega_{21} & \omega_{22} & \omega_{23} & \tau_y \\ \omega_{31} & \omega_{32} & \omega_{33} & \tau_z \end{bmatrix}$$

```
# Goal of function is to estimate pose of plane relative to camera (extrinsic matrix)
# given points in image xImCart, points in world XCart and intrinsic matrix K

def estimatePlanePose(XImCart,XCart,K):

    # TO DO: Convert Cartesian image points XImCart to homogeneous representation XImHom
    XImHom = np.concatenate((XImCart, np.ones((1,XImCart.shape[1]))), axis=0)

    # TO DO: Convert image co-ordinates XImHom to normalized camera coordinates XCamHom
    XCamHom = np.matmul(inv(K),XImHom)

    # TO DO: Estimate homography H mapping homogeneous (x,y) coordinates of positions
    # in real world to XCamHom (convert XCamHom to Cartesian, calculate the homography) -
    # use the routine you wrote for Practical 1B

    #Convert to cartesian
    XCamCart = XCamHom[0:2,:] / np.tile([XCamHom[2,:]],(2,1))

    #Remove last row
    XCart = XCart[:-1,:]
    #Compute homography
    H = calcBestHomography(XCart,XCamCart)

    # TO DO: Estimate first two columns of rotation matrix R from the first two
    # columns of H using the SVD
    Rest = solve_rotation(H[:,:2])
```

```
27
28      # TO DO: Estimate the third column of the rotation matrix by taking the
        cross
29      # product of the first two columns
30      Rest_2 = np.cross(Rest[:,0],Rest[:,1])
31      Rest_2 = Rest_2.reshape((3,1))
32
33      # TO DO: Check that the determinant of the rotation matrix is positive #-
        if not then multiply last column by -1.
34      Rest_final = np.hstack((Rest,Rest_2))
35      print(Rest_final.shape)
36
37      if det(Rest_final)<0:
38          Rest_final[:,-1] = Rest_final[:,-1] * -1
39
40      # TO DO: Estimate the translation t by finding the appropriate scaling #
        factor k
41      # and applying it to the third colulmn of H
42      scaling_factor = np.sum(H[:,:2]/Rest_final[:,:2])/6
43      t = H[:,-1]/scaling_factor
44      t = t.reshape((3,1))
45
46      # TO DO: Check whether t_z is negative - if it is then multiply t by -1 #
        and the first two columns of R by -1.
47      if t[-1,:] < 0:
48          t[:,:] = t[:,:] * -1
49          Rest_final[:,:2] = Rest_final[:,:2] * -1
50
51      # TO DO: Assemble transformation into matrix form
52      T = np.hstack((Rest_final,t))
53
54      return T
55
```

Listing 8: Routine

As we mentioned above, we used the the same **calcBestHomography** function as in exercise 1A.
The function below is used in the function **estimatePlanePose** to estimate the first two columns of the rotation matrix.

```
1
2  def solve_rotation(A):
3      u,s,v = np.linalg.svd(A)
4      matrix = np.array([[1,0],[0,1],[0,0]])
5      result = np.matmul(np.matmul(u,matrix),v)
6      return result
7
```

Listing 9: Routine

The final routine is defined as follows:

```
1
2  # TO DO: Use the general pin-hole projective camera model discussed in the
       lectures to estimate where the four points on the plane will appear in the
```

```
    image.  Fill in the details of the function "projectiveCamera" - body of
    function appears below:

XImCart = projectiveCamera(K,T,XCart)

# TO DO: Add noise (standard deviation of one pixel in each direction) to the
    pixel positions to simulate having to find these points in a noisy image.
    Store the results back in xImCart
XImCart = XImCart + np.random.normal(1,size=(2,5))

# TO DO: Now we will take the image points and the known positions on the card
    and estimate
# the extrinsic matrix using the algorithm discussed in the lecture.  Fill in
    the details of
# the function "estimate plane pose"
TEst = estimatePlanePose(XImCart,XCart,K)


```

Listing 10: Routine

- 1. TODO: Use the pinhole projective camera model. For this TODO we used the intrinsic matrix K, the extrinsic matrix T and the points in the plane XCart (All given)
- 2. TODO: We are going to add noise of 1 std to each value in XImCart.
- 3. TODO: After doing that, we are going to estimate the extrinsic matrix using the same XCart and K, but using the new coordinates of the image.

2.2 D. Practical 2B

In this part we are going to use all the functions created in the part 2A and we are going to take a real image containing a planar black square and figure out the transformation between the square and the camera.

We are given the points needed to estimate the extrinsic matrix, the intrinsic matrix given too.

```
# Load image:
im = plt.imread('test104.jpg')

# Define points on image
XImCart = np.array([[140.3464,  212.1129,  346.3065,  298.1344,   247.9962],
                    [308.9825,  236.7646,  255.4416,  340.7335,   281.5895]])

# Define 3D points of plane
XCart = np.array([[-50, -50,  50,  50, 0],
                  [50, -50, -50,  50, 0],
                  [0,    0,   0,   0, 0]])

# We assume that the intrinsic camera matrix K is known and has values:
```

```
K = np.array([[640,   0,   320],
              [0,  640,  240],
              [0,   0,   1]])

# Draw image and 2d points
plt.imshow(im)
plt.plot(XImCart[0,],XImCart[1,],'r.')
plt.axis('off')
plt.show()

```

Listing 11: Routine

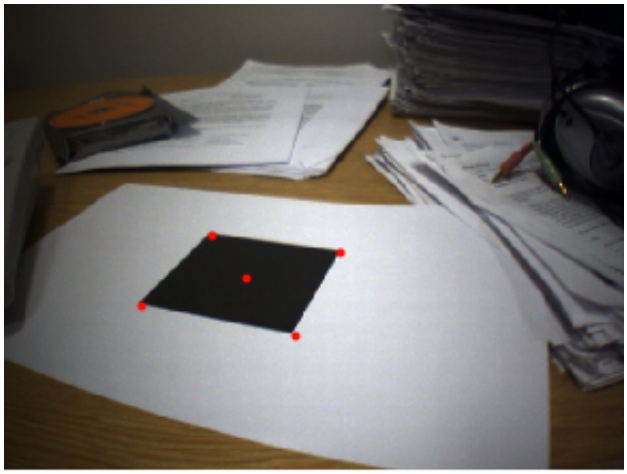

Figure 7: Estimated set of points

We create a helper function to create the lines between the new points in a form of a cube.

```
def connectpoints(x,y):
    x0, x1 = x[0,0], x[0,1]
    y0, y1 = y[1,0], y[1,1]
    x1, x2 = x[0,1], x[0,2]
    y1, y2 = y[1,1], y[1,2]
    x2, x3 = x[0,2], x[0,3]
    y2, y3 = y[1,2], y[1,3]
    x4, x5 = x[0,4], x[0,5]
    y4, y5 = y[1,4], y[1,5]
    x5, x6 = x[0,5], x[0,6]
    y5, y6 = y[1,5], y[1,6]
    x6, x7 = x[0,6], x[0,7]
    y6, y7 = y[1,6], y[1,7]

    plt.plot([x0,x1,x2,x3,x0,x4,x5,
    x6,x7,x4,x5,x1,x2,x6,x7,x3],
    [y0,y1,y2,y3,y0,y4,y5,
    y6,y7,y4,y5,y1,y2,y6,y7,y3],marker = 'o')


```

Listing 12: Routine

In the code below, we estimate the extrinsic matrix and then we project the 3D Points (XWifreFrameCart) using K and the extrinsic matrix estimated.

```
# TO DO: Use your routine to calculate TEst, the extrinsic matrix relating the
# plane position to the camera position.
TEst = estimatePlanePose(XImCart,XCart,K)
#print(TEst.shape)

# Define 3D points of plane
XWireFrameCart = np.array([[-50, -50,  50,  50, -50, -50,  50,  50],
                   [50, -50, -50,  50,  50, -50, -50,  50],
                    [0,   0,   0,   0, -100, -100, -100, -100]]);


# TO DO: Draw a wire frame cube, by projecting the vertices of a 3D cube through
# the projective camera and drawing lines betweeen the resulting 2d image points
XWireFrameCartProjected = projectiveCamera(K,TEst,XWireFrameCart)

plt.imshow(im)
plt.plot(XImCart[0,],XImCart[1,],'r.')
plt.plot(XWireFrameCartProjected[0,],XWireFrameCartProjected[1,],'g.')
connectpoints(XWireFrameCartProjected,XWireFrameCartProjected)
plt.axis('off')
plt.show()

# Draw image, 2d points and projected 3D cube points
plt.imshow(im)
plt.plot(XImCart[0,],XImCart[1,],'r.')
plt.plot(XWireFrameCartProjected[0,],XWireFrameCartProjected[1,],'g.')
plt.axis('off')
plt.show()

```

Listing 13: Routine

In the image below we can see the points projected in green on the left and on the right the wireframe of a cube using those points.
We can see that the cube is slightly askew, but it has a good shape. This could happen when we extract the first two columns from the homography matrix to then apply SVD since this last operation find the closest valid first two columns of a rotation matrix to the first two columns of the homography matrix. Also, this problem requires nonlinear optimization, however we are getting an approximation by using the homogeneous representation.

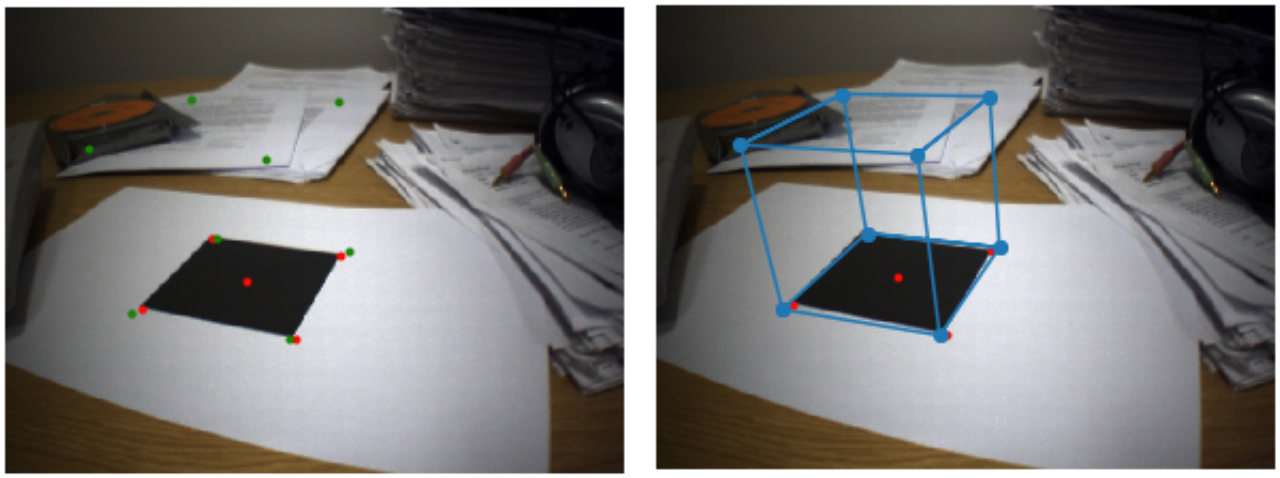

Figure 8: Wireframe of a cube

3 Condensation

3.1 E. Practical 9A

For this exercise we are going to perform Condensation which is a method to perform Particle Filtering, which is a technique to represent the probability density as a set of particles in the state space. Particle Filtering deals with the issues in Extended Kalman Filters, which cannot deal with multimodal distributions. Below the code that shows the TODOs for this section.

```
1
2 # TO DO: normalize the weights (may be trivial this time)
3 weight_of_samples = weight_of_samples/np.sum(weight_of_samples)
4
5 # We loop here to see what factored resampling would look like.
6 for iTime in range(10):
7     print('Iteration ', iTime, ':')
8     # TO DO: compute the cumulative sume of the weights
9     cum_hist_of_weights = np.cumsum(weight_of_samples)
10
11    # Predict where the particles we plucked from the old distribution of
12    # state-space will go in the next time-step. This means we have to apply
13    # the motion model to each old sample.
14    particles_new = np.zeros_like(particles_old)
15    for particleNum in range(numParticles):
16        # TO DO: Incorporate some noise, e.g. Gaussian noise with std 10,
17        # into the current location (particles_old), to give a Brownian
18        # motion model.
19        particles_new[particleNum, :] = particles_old[samples_to_propagate[
    particleNum], :] + 10*np.random.normal(size=2)
20
21    # TO DO: normalize the weights
```

```
22      weight_of_samples = weight_of_samples/np.sum(weight_of_samples)
23
24
```

Listing 14: Routine

- 1. TODO: Normalise the weights. We do that in order to have a distribution of weights.
- 2. TODO: Compute the cummulative distribution of the weights. We do that by using **np.cumsum**
- 3. TODO: add some noise to the Gaussian to give a Brownian motion model.
- 3. TODO: Normalise the weights again.

Below we see that the particles start at random.

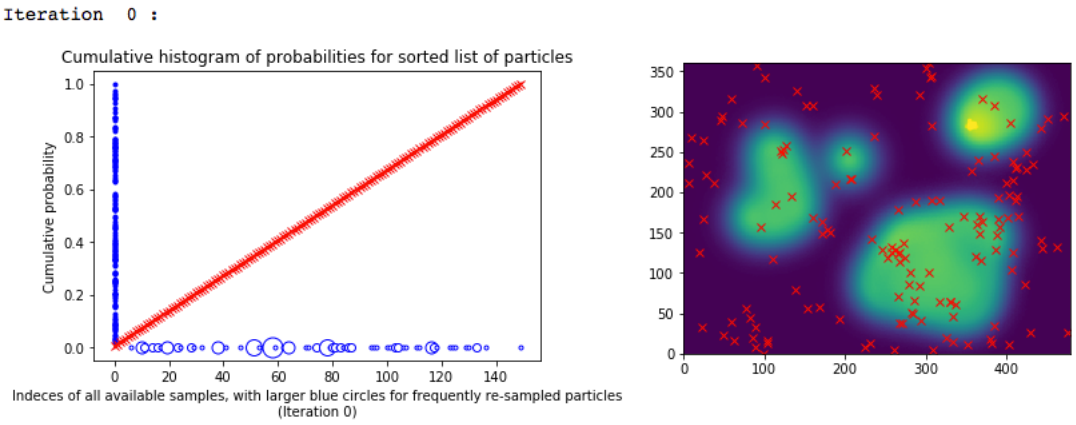

Figure 9: Iteration 0

After two iteration, we start to see how the particles cluster related to the green areas.

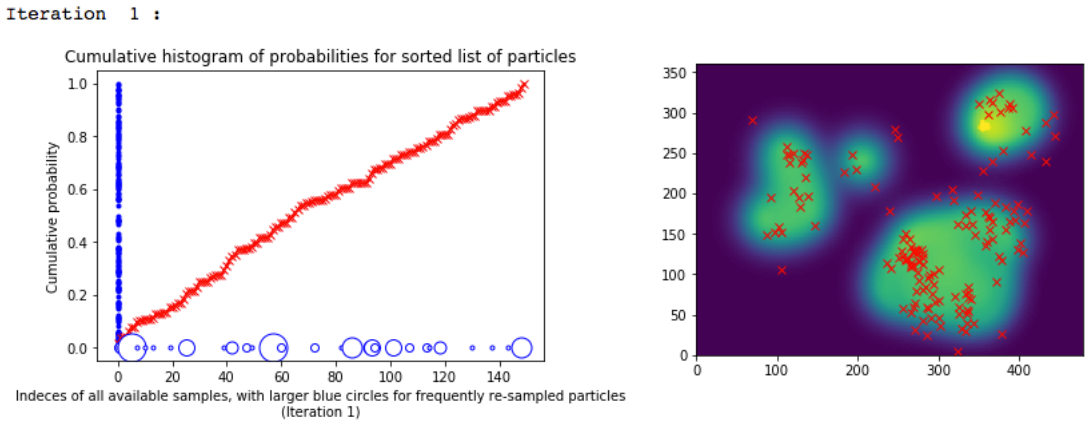

Figure 10: Iteration 1

In the third iteration we start to see more concentrated clusters.

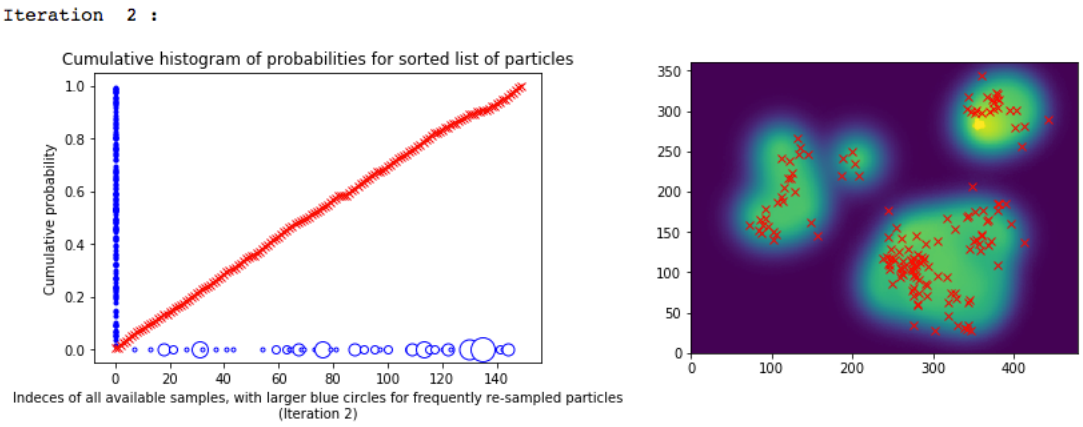

Figure 11: Iteration 2

For the rest of the iterations we see a similar pattern. In this case we show the forth and fifth iterations.

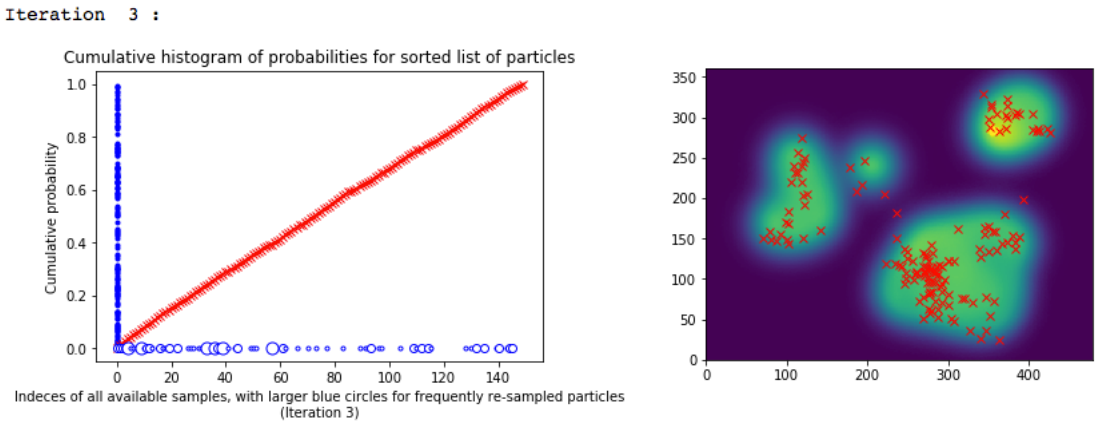

Figure 12: Iteration 3

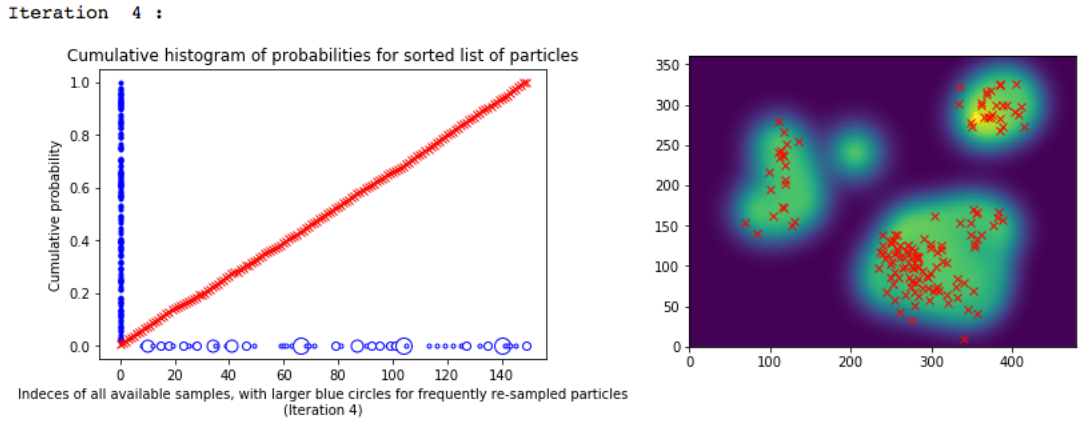

Figure 13: Iteration 4

In the last iteration we can see a more defined clusters compared to the first iterations.

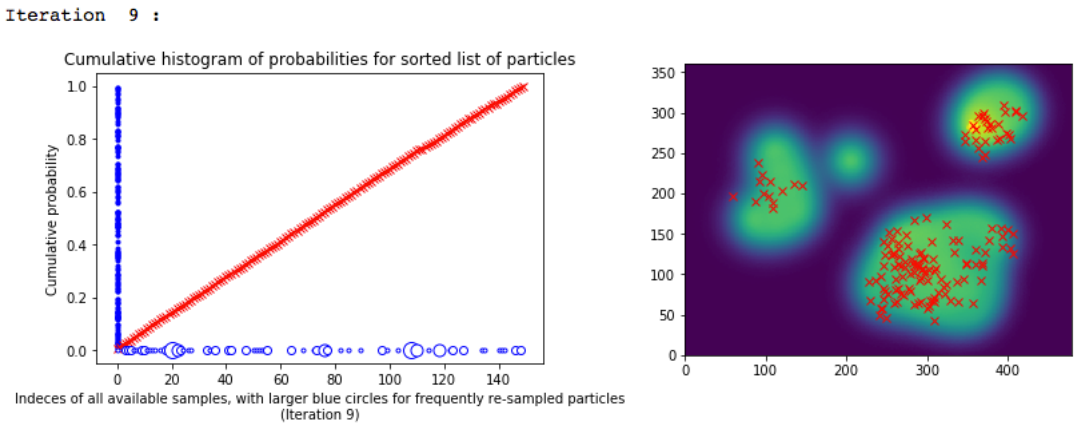

Figure 14: Iteration 9

3.2 F. Practical 9B

Now, we will track a given shape (template) as it moves in a sequence of frames. In this case we are going to use 1000 particles. The TODO are quite similar to the ones in 9A.

```
1
2 # TO DO: normalize the weights (may be trivial this time)
3 weight_of_samples = weight_of_samples/np.sum(weight_of_samples)
4
5 for iTime in range(22):
6     print('Processing Frame', iTime)
7     # TO DO: compute the cumulative sum of the weights.
8     cum_hist_of_weights = np.cumsum(weight_of_samples)
9
10    for particleNum in range(numParticles):
11        # TO DO: Incorporate some noise, e.g. Gaussian noise with std 10,
12        # into the current location (particles_old), to give a Brownian
13        # motion model.
14
15        particles_new[particleNum, 2:] = particles_old[samples_to_propagate[
    particleNum], 2:] + 10*np.random.normal(size=2)
16
17        particles_new[particleNum, :2] = particles_old[samples_to_propagate[
    particleNum], :2] + 10*np.random.normal(size=2)
18
19        particles_new[particleNum, 2:] = np.round(particles_new[particleNum,
    2:])
20        particles_new[particleNum, :2] = np.round(particles_new[particleNum,
    :2]) # Round the particles_new to simplify Likelihood evaluation.
21
22    # TO DO: normalize the weights [done]
23    weight_of_samples = weight_of_samples/np.sum(weight_of_samples)
24
```

Listing 15: Routine

Below we show different frames of cars, where we are tracking a wheel. Notice how all the particles are at random positions at Frame 0, and then start to cluster next to the wheels.

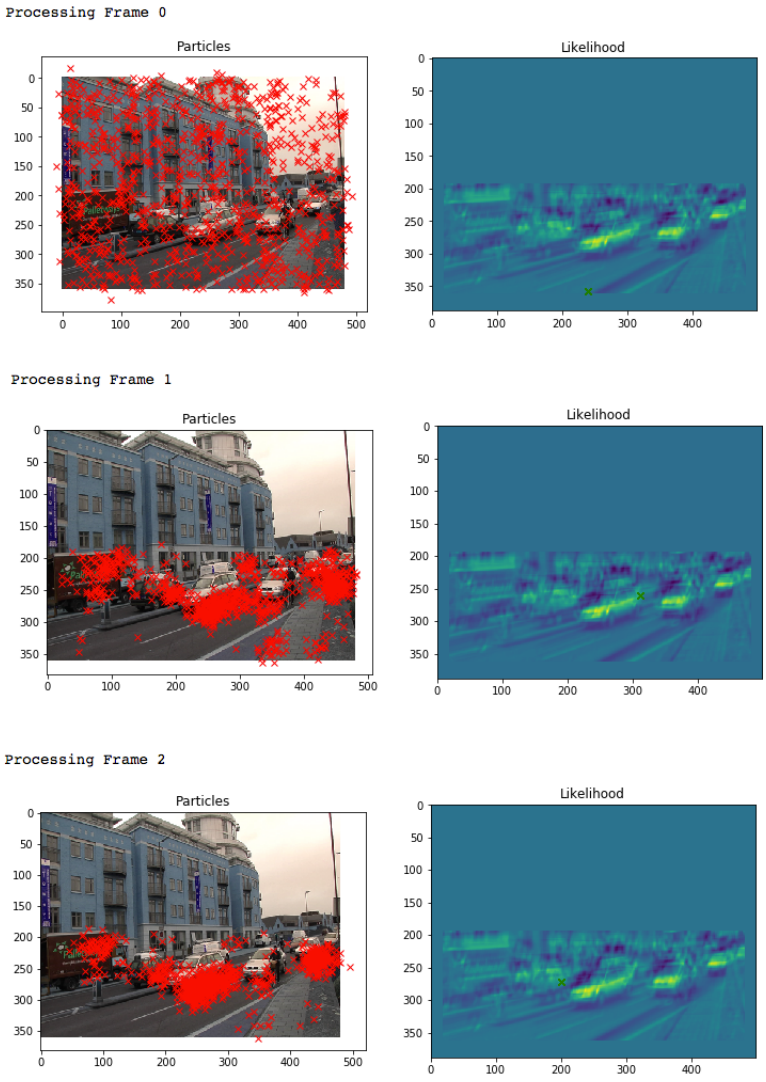

Figure 15: Frames 0-2

As we can see in the fig 16, the particles group better than in previous frames. In the next images, we will see how the particles track the wheels along the frames.

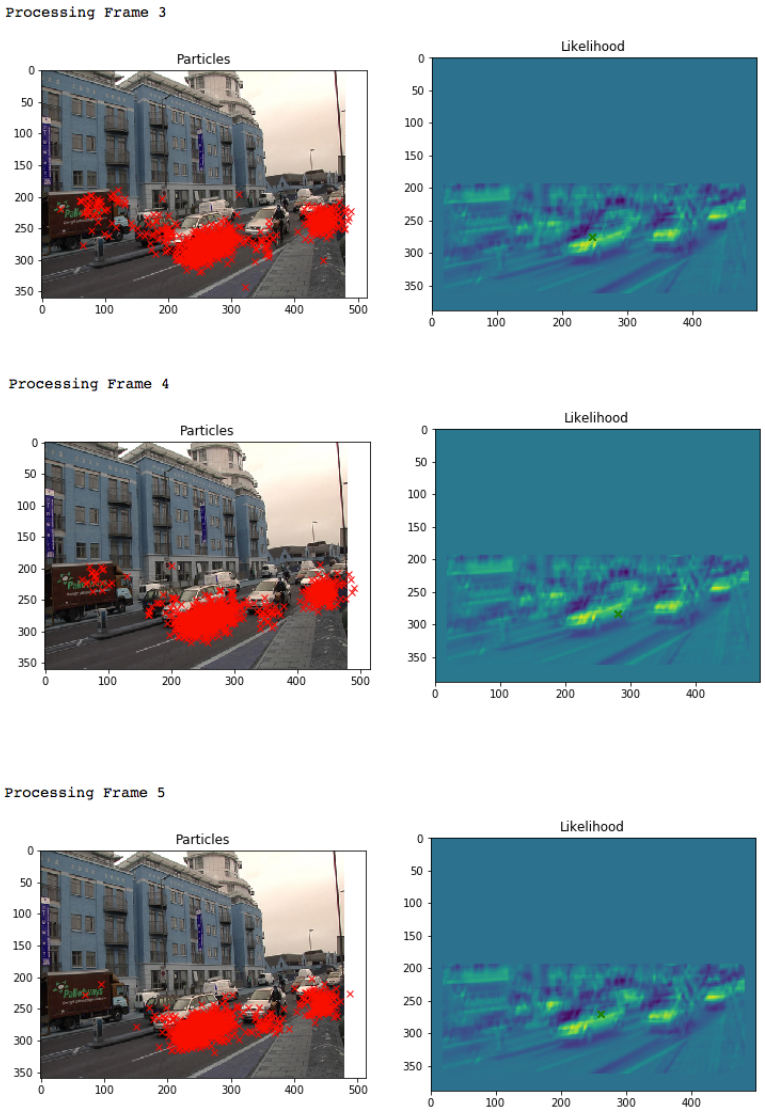

Figure 16: Frames 3-5

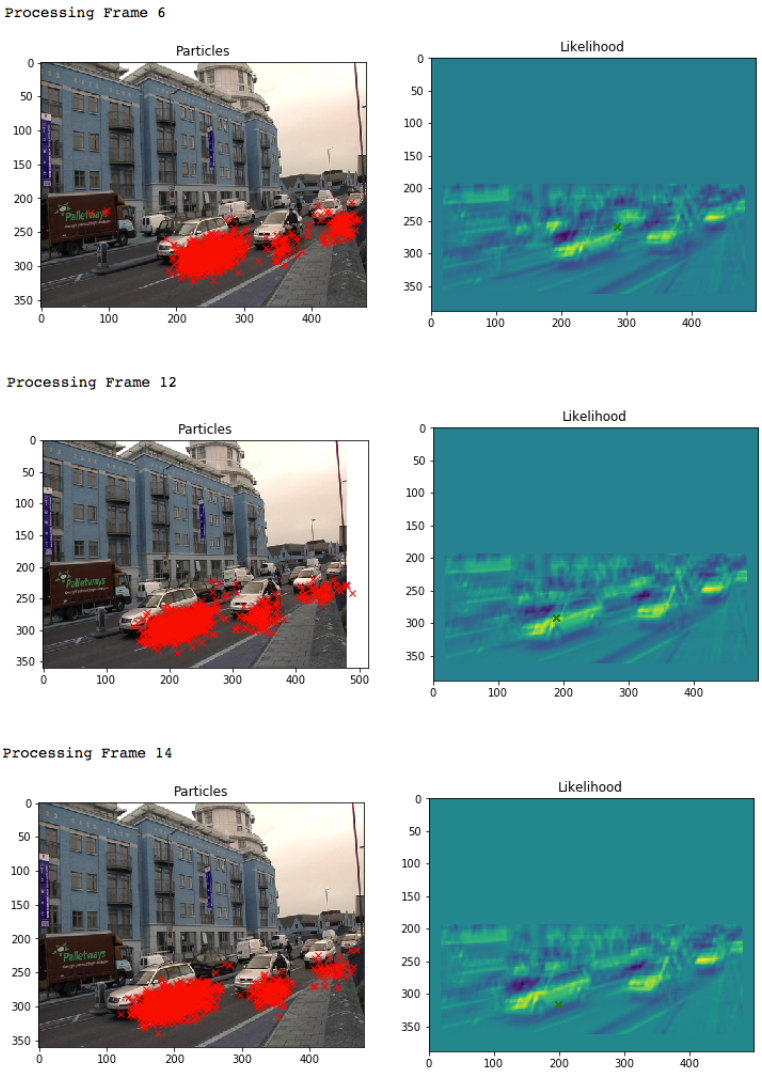

Figure 17: Frames 6,12,14

In the image below, we see that the particles follow the wheels in the correct way, even the wheels of the last car at the back.

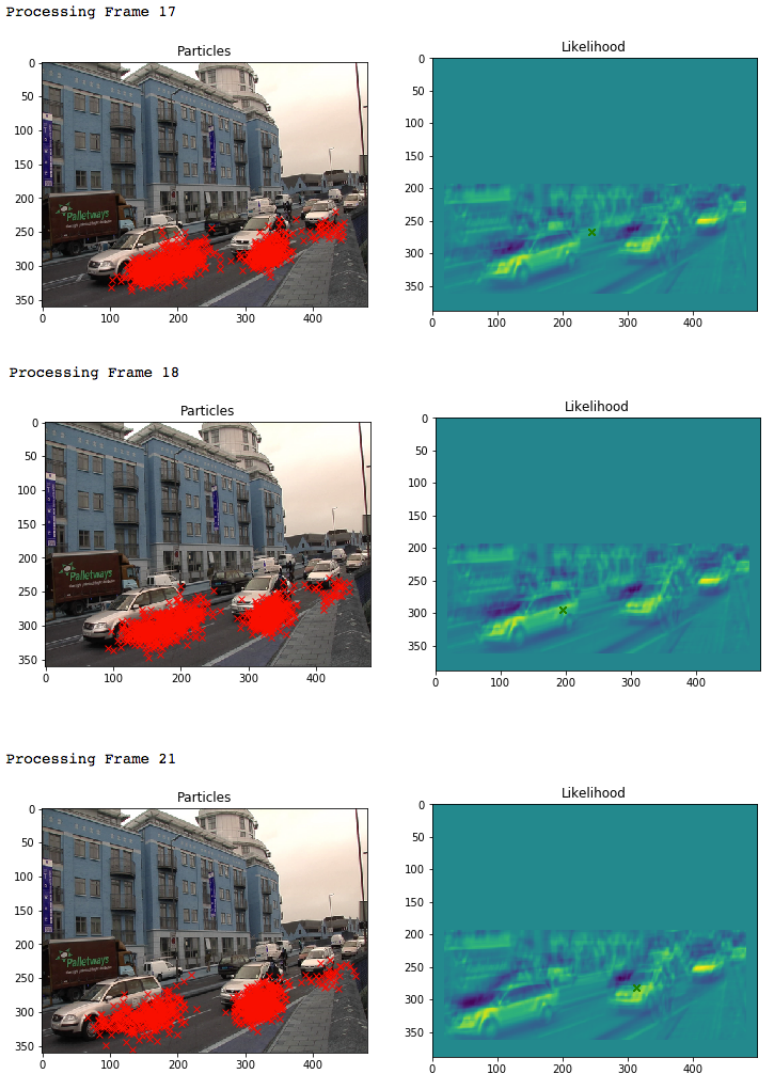

Figure 18: Frames 17,18,21

4 References

Prince, S.J.D. Computer Vision: Models Learning and Inference. Cambridge University Press. 2012