LfD code study

Wednesday, December 30, 2020 8:06 AM

Main.py

Notations:

- Ms_t : many matrices transposed
- prefixe specify input or estimate variable, eg inputCs, estCs.
- C: ellipse in dual form, 3x3
- Q: Quadric/ellipsoid in dual form 4x4, in the world frame.
- K: camera intrinsics 3x3
- M: pose matrix, 3x4, transforms points from world frame to camera frame.
- P: projection matrix: $P = KM \in R^{3\times4}$.
- visibility:
  - false: object not in image, or detector fails.
- object not detected in at least 3 frames is ignored.

Set the parameters for the algorithm and load the input data.

- Data association is implicitly defined in the data structures, each column of visibility corresponds to one object.
- In bbs, each object has four columns,
  - bounding boxes [X0, Y0, X1, Y1]
  - [X0, Y0] = top-left corner
  - [X1, Y1] = bottom-right corner
  - [n_frames x n_objects x 4] size
- Ms_t is camera pose matrices, transposed and stacked.
  - [n_frames x 4 x 3]
  - each 3x4 matrix transforms points from world frame to camera frame.
- visibility:
  - indicate whether a detection is available for a given object, on a frame.
  - [n_frames x n_objects].
  - n_frames = visibility.shape[0]
    n_objects = visibility.shape[1].

run the algorithm, estimate the ellipsoids.

[inputCs, estCs, estQs] = compute_estimates(bbs, K, Ms_t, visibility)

- estimate one ellipsoid per object, given detection bounding boxes and camera parameters.
- returns
  - inputCs: ellipses fitted to the input bounding boxes, for each image and each object, in dual form.
    size: n_frames x 3 X n_objects x 3, each ellipse is described by a 3x3 matrix.
  - estCs: ellipses resulting from the projection of the estimated ellipsoids, in dual form,
    size: n_frames x 3 X n_objects x 3, each ellipse is 3x3.
  - estQs_second_step: estimated ellipsoids, one per detected object, in dual form,
    size: n_objects x 4 x 4.

- Compute the stacked and transposed projection matrices.
  - $Ps\_t = (K\, Ms\_t^T)^T$, size: n_frames x 4 x 3.

- Compute ellipses inscribed in the detection bounding boxes.
  - inputCs = fit_ellipses_in_bbs(bbs, visibility)
  - it calls fit_one_ellipse_in_bb(bb).

```
#Encode the ellipse size (axes)
width = abs(bb[2] - bb[0]) / 2
height = abs(bb[3] - bb[1])/2
Ccn = np.vstack((np.hstack((np.diag((1/width**2, 1/height**2)), np.zeros((2,1)))), np.array([0,0,-1])))
```

} bounding box size.

$$C_{cn} = \begin{bmatrix} \frac{1}{a^2} & 0 & 0 \\ 0 & \frac{1}{b^2} & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Bounding box to ellipse:

- bounding box is $\beta = [x_{min}, y_{min}, x_{max}, y_{max}]$
- $a = \left|\frac{x_{max} - x_{min}}{2}\right|$, $b = \left|\frac{y_{max} - y_{min}}{2}\right|$
- $X_0 = \begin{bmatrix} \frac{x_{min} + x_{max}}{2} \\ \frac{y_{min} + y_{max}}{2} \end{bmatrix}$
- $C_c = \begin{bmatrix} \frac{1}{a^2} & 0 & 0 \\ 0 & \frac{1}{b^2} & 0 \\ 0 & 0 & -1 \end{bmatrix}$

```
#Encode ellipse location
center = np.array(((bb[0] + bb[2])/2, (bb[1] + bb[3]) / 2)),
P = np.vstack((np.hstack((np.eye(2,2), center.reshape(2,1))), np.array([0,0,1]))).
Cinv = P.dot(np.linalg.inv(Ccn)).dot(P.transpose()).
```

- $P = \begin{bmatrix} 1 & 0 & X_0(1) \\ 0 & 1 & X_0(2) \\ 0 & 0 & 1 \end{bmatrix}$
- $C_{inv} = P C_c^{-1} P^T$

```
# Force matrix to be symmetric:
Cinv = 0.5 *(Cinv + Cinv.transpose())
```

$$C_i = \frac{C_{inv} + C_{inv}^T}{2}$$

```
#Scale ellipse so that element[2,2] =1.
C = Cinv / Cinv[2, 2]
```

- $C_\xi = \frac{C_i}{C_i(3,3)}$

```
C = C* np.sign(C[0,0] + C[1,1])
```

- $C = C_\xi \cdot \text{sign}(C_\xi(1,1) + C_\xi(2,2))$
  sign is signum function.

```
return C.
```

- Set the initial ellipsoids centers to the origin.

```
input_ellipsoids_centers = np.dot(np.array(([0], [0], [0], [1])), (np.ones((1, n_objects))))
```

- **perform the first round of estimation. ↓**

```
estQS_first_step = estimate_ellipsoids(Ps_t, input_ellipsoids_centers, inputCs, visibility)
```

- Ps_t: stacked transposed projection matrix, n_frames * 4 x 3.
- input_ellipsoid_centers: represented in homogeneous coordinates, 4X n_objects.
- inputCs: ellipses fitted to the bounding box in dual form, n_frames*3 X n_objects*3.
- visibility: object visibility, n_frames x n_objects.
- returns: estimated ellipsoids in dual form, n_objects x4x4.

```
#Initialize output structure
estQs = np.zeros((n_objects, 4, 4))

for obj in range(n_objects):
    #if there are at least 3 detections
    if sum(visibility[:, obj]) >= 3:
        #select only the Cs [3x3] for the current object, for the frames in which
        # it was detected
        # Create a mask with true in the desired location.
        row_selector = np.kron(visibility[:, obj], np.ones(3).reshape(1,3))
        row_selector = np.array(row_selector, dtype = bool) [0].
        # Apply the mask
        selectedCs = inputCs [row_selector, obj * 3 : obj*3 + 3].

        #select the corresponding projection matrices,
        # Create the mask
        row_selector = np.kron(visibility[:,obj], np.ones(4), reshape(1,4))
        row_selector = np.array(row_selector, dtype= bool) [0].
        # Apply the mask.
        selectedPs_t = Ps_t[row_selector, :]

        #Compute the translation matrix due to the center of the current ellipsoid.
        translM = np.eye(4)
        translM[0:3,3] = input_ellipsoids_centers[0:3, obj].

        #loop over the frames in which the current object is present
        # apply the translation matrix to each projection matrix, to apply the initial estimate.
        for instance_id in range(math.floor(selectedPs_t.shape[0]/4)):
            first = np.hstack((np.eye(3), np.zeros((3,1))))
            second = np.vstack((selectedPs_t[instance_id * 4: instance_id * 4 + 4,:].transpose(), np.array([0,0,0,1])))
            selectedPs_t[instance_id * 4: instance_id * 4 + 4,:] = np.dot(np.dot(first, second), translM).transpose().

        # Estimate the parameters of the current ellipsoid.
        estQ = estimate_one_ellipsoid(selectedPs_t, selectedCs)
```

```
# num of frames the object is detected.
n_views = math.floor(Cs.shape[0]/3)

# Matrix of the linear system
M = np.zeros((6* n_views, 10 + n_views))

# Compute B matrix and stack them into M.
for index in range(n_views):
    # get center and axes of current ellipse.
    [center, axes, _] = dual_ellipse_to_parameters(Cs[3*index : 3* index +3, :])
```

```
if C[2,2] > 0:
    C = C / -C[2,2]
center = (-C[0:2, 2]).reshape(2,1)
T = np.vstack((np.hstack((np.eye(2), -center)), np.array([0,0,1])))
C_center = T.dot(C).dot(T.transpose())
C_center = 0.5 * (C_center + C_enter.transpose())
D, V = np.linalg.eig(C_center[0:2,0:2])
axes = np.sqrt(abs(D))
R = V

reture center, axes, R.
```

center: ellipse center
axes: ellipses axes length
R : ellipse orientation.

This corresponds to

$$C_{if}^* = H_{if} \check{C}_{if}^* H_{if}^T$$

```
# Compute T, a transformation used to precondition the ellipse.
# T centers the ellipse and scales the axes.
div_f = np.linalg.norm(axes)
T = np.linalg.inv((np.vstack((np.hstack((np.eye(2) * div_f, center)), np.array([0,0,1])))))
T_t = T.transpose()

# Compute P_fr, applying T to the projection matrix.
P_fr = np.dot(Ps_t [4* index : 4*index +4,:], T_t)
```

Code block above corresponds to

$$\beta_{if} \check{C}_{if}^* = H_{if}^{-1} P_f Q_i^* P_f^T H_{if}^T \quad (15)$$

$$\beta_{if} \check{C}_{if}^* = H_{if}^{-1} P_f T_i T_i^{-1} Q_i^* T_i^{-1} T_i^T P_f^T H_{if}^{-T} \quad (16).$$

- T_i is the translation matrix whose last column contains the translation parameters of $\tilde{Q}_i^*$.
- $H_{if}^{-1} P_f T_i$ is the new projection matrix.

```
# compute the coefficients for the linear system
B = compute_B(P_fr)
```

```
B = np.zeros([6, 10])
# vectorise P, one row after the other
# C order = row_major
vec_p = np.reshape(P_fr, (12,1), order = 'C')
r = vec_p[0:9]
t = vec_p[9:12]
# Fill B
B[0,:] = (r[0] ** 2,
    ...
return B
```

The B matrix is G in (4) in paper

$$G_f = D(P \otimes P) E$$

The B[0,:] = ... terms came from (5) in paper.

$$\beta_{if} \check{C}_{if}^* = H_{if}^{-1} P_f Q_i^* P_f^T H_{if}^{-T} \text{ in } (15)$$

```
# apply T to the ellipse ↑
C_t = np.dot(np.dot(T, Cs[3*index : 3* index +3, :]), T_t)
```

↗ convert C_{if}^* to $\check{C}_{if}^*$ as in (12).

```
# transform the ellipse to vector form.
C_tv = symmetric_mat_3_to_vector(C_t)
```

serializes a symmetric 3x3 matrix

```
C_tv /= - C_tv[5]
```

- C_tv is the c_{if}^* in (6). $c_{if}^* = \text{vech}(C_{if}^*)$.
- C_t is C_{if}^* in (3).

```
# write the obtained coefficients to the correct slice of M.
M[6*index : 6*index + 6, 0: 10] = B
M[6* index : 6* index +6, 10 + index ] = - C_tv.
```

```
_, _, V = np.linalg.svd(M)
w = V[-1, :]
```

Above code block is from (9)

$$\tilde{w}_i = \underset{w}{\text{argmin}} \|\tilde{M}_i w\|_2^2 \ \text{s.t.} \ \|w\|_2^2 = 1$$

(9) is solved by applying SVD to $\tilde{M}_i$, taking the right singular vector associated to the minimum singular value.

```
Qadjv = w[0:10]
adj_Q = vector_to_symmetric_mat_4(Qadjv)
return adj_Q.
```

The code block above: after (9) in paper

The first 10 entries of $\tilde{w}_i$ are the vectorized elements of the estimated dual quadric, denoted by $\tilde{v}_i^*$.

To get back the estimated matrix of the quadric in the primal space, we obtain first the dual estimated quadric by:

$$\tilde{Q}_i^* = \text{vech}^{-1}(\tilde{v}_i^*) \quad (10)$$

```
        # reapply the translation which had been removed.
        estQ = np.dot(translM, np.dot(estQ, translM.transpose())).

        # force the estQ matrix to be symmetric
        # estQ = (estQ + estQ^T)/2
        estQ = 0.5*(estQ + estQ.transpose())

        # Scale the ellipsoid to put it in standard form
        # element[3,3] set to -1
        estQ /= - estQ[3,3]

        # Store the results for the current ellipsoid into the output data structure.
        estQs[obj,:,:] = estQ.

    else:
        # In case there are fewer than 3 detections
        estQs[obj,:,:] = np.nan

return estQs.
```

first round of estimation ↑.

Note that the code only performs the linear optimization, and does not do the non-linear refinement.

- extract the centers of the current estimate for the ellipsoids

```
first_step_ellipsoids_centers = input_ellipsoids_centers
for object_id in range(n_objects):
    first_step_ellipsoids_centers[0:3, object_id] = estQs_first_step[object_id, 0:3, 3]
```

- perform the second round of estimation, exploiting the centers estimated at the previous step

```
estQs_second_step = estimate_ellipsoids(Ps_t, first_step_ellipsoids_centers, inputCs, visibility)
```

- Project the estimated ellipsoids onto the images

```
estCs = project_ellipsoids(Ps_t, estQs_second_step, visibility).
```