ECSE 6650 Final Project: Structure from Motion

Devavrat Jivani and Yogish Didgi

1. Introduction

Object tracking and 3D reconstruction are one of the fundamental problems in computer vision. A good object tracking algorithm finds application in surveillance systems, video compression etc. and a 3D reconstruction algorithm is useful in applications like visualisation, information extraction from the images. The tracker keeps track of an object as it moves across the video frames. The reconstruction algorithm tries to capture as much 3D information (e.g. 3D coordinates) of the object as possible. There are several algorithms which tackle these problems. In this project, we implement optical flow and Kalman filter techniques to perform the tracking followed by rigid and non-rigid factorization methods for 3D reconstruction. These methods will be explained in detail in the following sections.

2. Background

2.1 Optical Flow

The objects in a scene can move in three-dimensional world. The motion can be characterized by movement of the objects in the scene and a static camera or the scene being static and the camera moving or both the camera and the scene being dynamic. The idea is there is relative motion between the scene and the camera. When we capture an image of the scene into a two-dimensional plane, some information is lost. Subsequently, we can measure only a component of the actual motion flow vector of the object. Turns out, the component that can be extracted using the images is the projection of the actual motion flow vector along the gradient in the image. This component is called the optical flow vector. This is depicted in figure 1. We also assume that a rigid object is causing the relative motion.

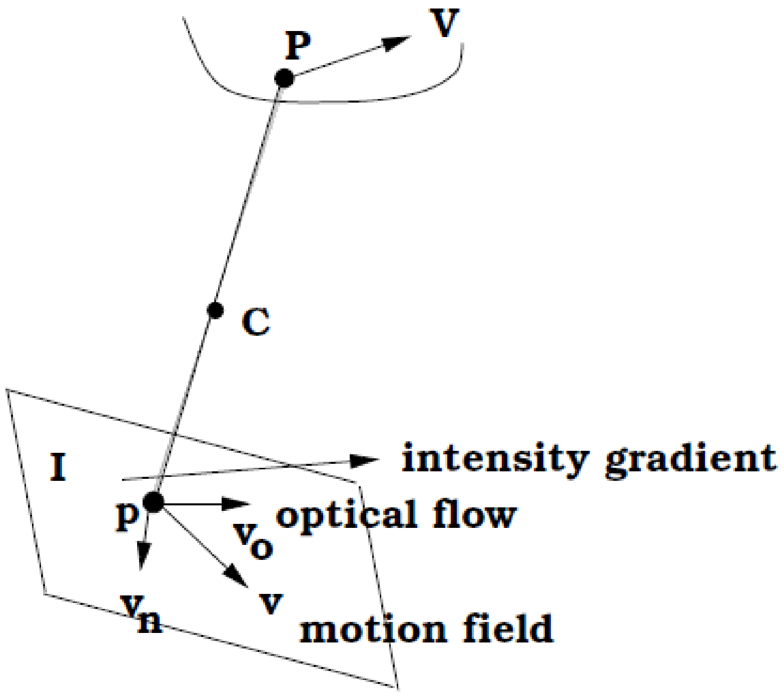


Figure 1: Motion field vs optical flow [1]

2.1.1 Image brightness constancy

The fundamental assumption for estimating the optical flow is the image brightness change constancy. Let $I^t(x,y)$ be the intensity of a pixel (x,y) at time t. We assume that the intensity of that pixel at time t+1, $I^{t+1}(x,y)$ remains the same. This can be mathematically represented as,

$$\frac{dI}{dt} = 0 \tag{1}$$

The above assumption is valid under two conditions. One, the motion is translational motion. Or two, illumination direction is parallel to the angular velocity for Lambertian surface [1]. Equation (1) can be written as

$$\frac{dI}{dt} = \frac{\partial I}{\partial x}\frac{dx}{dt} + \frac{\partial I}{\partial y}\frac{dy}{dt} + \frac{\partial I}{\partial t} = 0 \tag{2}$$

$\frac{\partial I}{\partial x}$ and $\frac{\partial I}{\partial y}$ represent spatial intensity gradient and $\frac{\partial I}{\partial t}$ represents temporal intensity gradient. The coefficients of the spatial intensity gradients in equation (2) are the components of the motion field.

$$(\nabla I)^t v + I_t = 0 \tag{3}$$

Equation (3) has no constraint on v when it is orthogonal to ∇I. v represents the optical flow vector.

2.1.2 Lucas-Kanade method

For a pixel in an image, equation (3) has 2 unknowns of the optical flow vector. In this method, it is assumed that in a neighborhood, all the pixels have the same optical flow. For each pixel p and a NxN neighborhood, we have,

$$Av = b \tag{4}$$

where,

$$A = \begin{bmatrix} \nabla^t I(x_1, y_1) \\ \nabla^t I(x_1, y_2) \\ \vdots \\ \nabla^t I(x_1, y_N) \end{bmatrix} \tag{5}$$

$$b = \begin{bmatrix} -I_t(x_1, y_1) \\ -I_t(x_1, y_2) \\ \vdots \\ -I_t(x_1, y_N) \end{bmatrix} \tag{6}$$

The solution is given by,

$$v(x,y) = (A^t A)^{-1} A^t B \tag{7}$$

Using equation (7), optical flow can be computed for each pixel. This is known as dense-flow estimation. This method requires only the first order derivative. The image is smoothed using a Gaussian filter before computing the derivatives. The derivatives can be computed using a simple difference operator like Sobel.

2.1.3 Image brightness change constancy method

The method described in section 2.2 says that the image brightness has to be a constant across frames at each pixel. In this method, it is assumed that the change of the brightness has to be a constant. This means that mathematically, the second derivative has to be zero.

$$\frac{d^2I}{dt\,dx} = 0, \frac{d^2I}{dt\,dy} = 0, \frac{d^2I}{dt\,dt} = 0 \tag{8}$$

$$v_xI_{xx} + v_yI_{yx} + I_{tx} = 0 \tag{9}$$
$$v_xI_{xy} + v_yI_{yy} + I_{ty} = 0 \tag{10}$$
$$v_xI_{xt} + v_yI_{yt} + I_{tt} = 0 \tag{11}$$
$$v_xI_x + v_yI_y + I_t = 0 \tag{12}$$

Equation (3) has been re-written as (12). We have four equations in 2 unknowns. They can be written as

$$\text{Av = b} \tag{13}$$

where,

$$A = \begin{bmatrix} I_x & I_y \\ I_{xx} & I_{yx} \\ I_{xy} & I_{yy} \\ I_{xt} & I_{yt} \end{bmatrix} \tag{14}$$

$$b = -\begin{bmatrix} I_t \\ I_{tx} \\ I_{ty} \\ I_{tt} \end{bmatrix} \tag{15}$$

The solution is given by,

$$v = (A^tA)^{-1}A^tB \tag{16}$$

If the condition of matrix A is large, then we will not be able to compute its pseudo-inverse with numerical accuracy. When this happens, in the implementation, we just set the flow to zero. Also, if the computed flow is less than a certain value which is numerically significant, even then we set the flow to zero.

In equation (12), we assumed that the constant of brightness change was zero. This can be assumed to be a constant that varies across each of the neighborhoods. This adds another unknown to be computed but we are still left with one more equation than the number of unknowns. So, the system of linear equations can be solved. The equation (9) now changes to,

$$v_x I_x + v_y I_y + I_t = c \tag{17}$$

where, c is a constant to be determined.

The linear system of equations can still be modeled using equation (13). However, they represent different quantities as shown below.

$$A = \begin{bmatrix} I_x & I_y & -1 \\ I_{xx} & I_{yx} & 0 \\ I_{xy} & I_{yy} & 0 \\ I_{xt} & I_{yt} & 0 \end{bmatrix} \tag{18}$$

$$v = \begin{bmatrix} v_x \\ v_y \\ c \end{bmatrix} \tag{19}$$

$$b = -\begin{bmatrix} I_t \\ I_{tx} \\ I_{ty} \\ I_{tt} \end{bmatrix} \tag{20}$$

Equation (19) shows that, the optical flow vector is just the first two components and the third component is the brightness change constant.

2.1.4 Cubic facet model

Image derivatives are computed using numerical approximation of continuous derivatives. In this method, image derivatives are computed analytically. A cubic facet model is fit for a block of pixels (spatially and temporally) in a neighborhood R. This model then represents the continuous function for image intensity which approximates image surface at (x,y,t). Now, the image derivatives can be computed analytically using the cubic function. This method has the advantage that it is more robust to noise and the function approximation provides an inherent smoothing operation.

For each k x k x k block of pixels in the given image sequence, we fit the following cubic model,

$$\begin{aligned} I(x,y,t) &= a_1 + a_2 x + a_3 y + +a_4 t + +a_5 x^2 + a_6 xy + \\ & a_7 y^2 + a_8 yt + a_9 t^2 + a_{10} xt + a_{11} x^3 + a_{12} x^2 y + \\ & a_{13} xy^2 + a_{14} y^3 + a_{15} y^2 t + a_{16} yt^2 + a_{17} t^3 + \\ & a_{18} x^2 t + a_{19} xt^2 + a_{20} xyt \end{aligned} \tag{21}$$

The solution for the coefficients a ($[a_1, a_2, \dots, a_{20}]^t$) is given by,

$$a = (D^t D)^{-1} D^t J \tag{22}$$

where,

$$D = \begin{bmatrix} 1 & x_1 & y_1 & t_1 & \dots & x_1y_1t_1 \\ 1 & x_2 & y_1 & t_1 & \dots & x_2y_1t_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_k & y_1 & t_1 & \dots & x_ky_1t_1 \\ 1 & x_1 & y_2 & t_1 & \dots & x_1y_2t_1 \\ 1 & x_2 & y_2 & t_1 & \dots & x_2y_2t_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_k & y_2 & t_1 & \dots & x_ky_2t_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_1 & y_k & t_k & \dots & x_1y_kt_k \\ 1 & x_2 & y_k & t_k & \dots & x_2y_kt_k \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_k & y_k & t_k & \dots & x_ky_kt_k \end{bmatrix} \tag{23}$$

$$J = \begin{bmatrix} I(x_1, y_1, t_1) \\ I(x_2, y_1, t_1) \\ \vdots \\ I(x_k, y_k, t_k) \end{bmatrix} \tag{24}$$

Note that, for this model, we need to use local coordinates for x, y and t. Hence, these values span the range -1 to 1. This also leads to the advantage that, D can be computed only once for the entire image, since it's the same at every point. Typically, the neighborhood is chosen to be 5, both spatially and temporally.

Once the pixel neighborhood is fit with a cubic model, we can compute its derivates analytically. Hence, equations (14) and (15) can now be represented as,

$$A = \begin{bmatrix} a_2 & a_3 \\ 2a_5 & a_6 \\ a_6 & 2a_7 \\ a_{10} & a_8 \end{bmatrix} \tag{25}$$

$$b = -\begin{bmatrix} a_4 \\ a_{10} \\ a_8 \\ 2a_9 \end{bmatrix} \tag{26}$$

2.1.5 Iterative optical flow

The output of an optical flow algorithm is the average flow of the set of pixels in the neighborhood. The formulation of optical flow works best only if the net flow between the images is very small. Hence, the basic optical flow algorithm isn't very useful for tracking an object across several frames. To overcome this problem, we implement the iterative optical flow algorithm [3].

In iterative optical flow algorithm, we compute the optical flow as in previous sections. Then, the estimated flow is used to shift the search window in the next frame. The gradient is recomputed for the shifted window and the substituted in the flow constraint equations. This gives rise to a refined flow. The sum of the entries in the vector ‘b’ as in equations (6) and (15) indicates the error in the match of the patches between the frames. This process is carried out until the error becomes small or starts increasing with respect to previous iteration. The flow is accumulated over all the iterations and the final flow is assigned as the final location of the tracked point.

2.2 Kalman filter

Kalman filter comes from the theory of optimal estimation theory. The problem of feature tracking is considered as a dynamic system with state changes occurring between frames contributing to the movement of the feature point. The tracking of feature points across frames also fits in nicely with the framework of recursive systems where the feature movement across frames can be considered independently across successive frames. The Kalman filtering technique is a recursive procedure that estimates the next state of a dynamic system, based on the system’s current state and its measurement [1]. For the purposes of object tracking, the Kalman filter estimates the position and uncertainty of a moving feature point in the next frame [2] given the position of the feature in the current frame. The technique also outputs a confidence region in the next frame, around the predicted position, to search for the feature.

The Kalman filter makes the assumption that the underlying dynamical system is linear in nature. And the noise in the system is assumed to be a Gaussian random variable.

Consider a feature point at a location (x_t, y_t) in the image frame at time t. Let the motion of the feature point be moving with velocity given by $v_t = (v_{x,t}, v_{y,t})^t$. The state vector at time t is composed of both the position and the velocity of the feature point. Hence, the state vector is given by,

$$s_t = (x_t, y_t, v_{x,t}, v_{y,t})^t \tag{27}$$

The objective is to compute the state vector in the next frame i.e. at time t+1. In other words, given s_t, estimate s_{t+1}.

Based on the assumption made earlier that, the state dynamics is modeled on a linear system, the state at time t+1 is related to state at time t as follows,

$$s_{t+1} = \phi s_t + w_t \tag{28}$$

where, ϕ is the state transition matrix and $w_t$is the state perturbation or noise modeled as a normal distribution, N(0, Q), meaning it has a mean of zero and covariance given by Q.

The feature movement between the consecutive frames is assumed to be small, so that the motion of feature points’ positions is uniform from frame to frame. The state transition matrix is given by,

$$\phi = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{29}$$

The measurement model is modeled as,

$$z_t = Hs_t + \varepsilon_t \tag{30}$$

where, z_t = (z_{x_t}, z_{y_t}) is the measured feature point position. Matrix H, relating the current state to current measurement is called the measurement matrix. ε_t represents the measurement noise and it is modeled as a normal random variable, N(0, R), where R is the covariance matrix. Measurement is obtained using the feature detection methods. In our case, we will be using a simple sum of squared difference based template matching technique, which will be explained in a later section. Since z_t involves only position, we assume that z_t is same as s_t plus some measurement noise. Hence, H is given by,

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{31}$$

Kalman filtering consists of two steps; state prediction and state updating. State prediction is preformed using the state model and the state updating is done using the measurement model.

2.2.1 State prediction

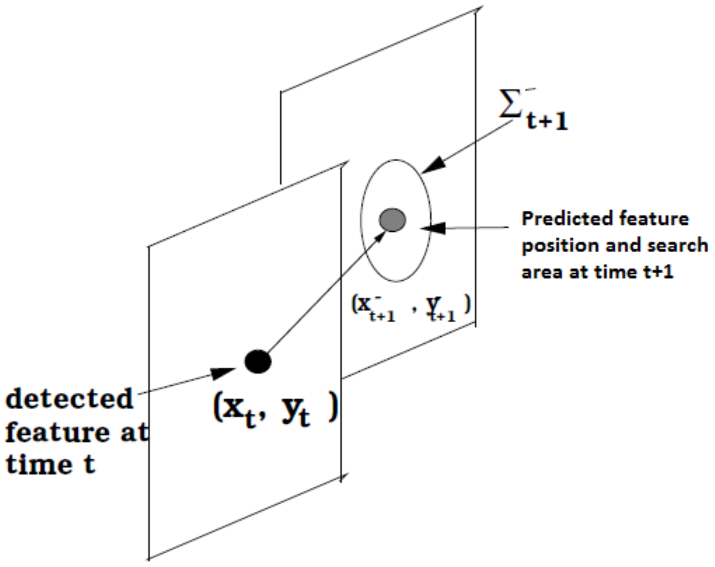

$\sum_{t+1}^{-}$
Predicted feature position and search area at time t+1
$(x_{t+1}^{-}, y_{t+1}^{-})$
detected feature at time t
(x_t, y_t)

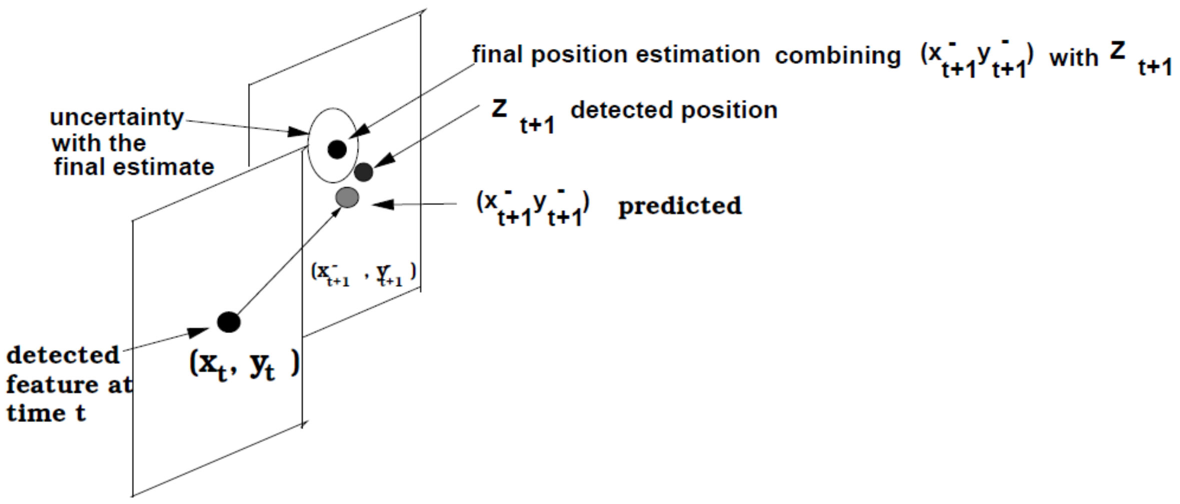
Figure 2: State prediction

Given the current state, s_t, and its covariance matrix, $\sum_t$, state prediction involves two steps, state projection, s_{t+1}^{-}, and error covariance estimation, $\sum_{t+1}^{-}$. $\sum_{t+1}^{-}$ represents the confidence of prediction. This is shown in figure 1. For the first frame, the initial position of the face and the velocity of the feature point are already given to us. So, state s_t is a known quantity. $\sum_t$ is assumed to be certain value as will be discussed later. Given these information, the following equations result.

$$s_{t+1}^{-} = \phi s_t \tag{32}$$

$$\Sigma_{t+1}^{-} = \phi \Sigma_t \phi^t + Q \tag{33}$$

2.2.2 State updating



final position estimation combining $(x_{t+1}^{-}, y_{t+1}^{-})$ with z_{t+1}
uncertainty with the final estimate
z_{t+1} detected position
$(x_{t+1}^{-}, y_{t+1}^{-})$ predicted
$(x_{t+1}^{-}, y_{t+1}^{-})$
detected feature at time t
(x_t, y_t)

Figure 3: State updating

The state updating consists of three steps.

a. Measuring process to get z_{t+1}
b. Combine z_{t+1} and s_{t+1}^{-} to get a final state estimate, s_{t+1}
c. Obtain the posteriori error covariance estimate, $\sum_{t+1}$

For getting the measurement, a feature detector is used. The search is initialized at the predicted position and the search window size is decided by the covariance matrix, $\sum_{t+1}^{-}$ as shown in figure 2. The search region is centered at the predicted location and the window size is chosen to be $k\sigma_x \times k\sigma_y$. K is a parameter chosen to be either 2 or 3. σ_x and σ_y are the square root of the eigen values of the first 2x2 submatrix of $\sum_{t+1}^{-}$. In this project, instead of a feature detector, we are using a template matching technique based on sum of squared differences (SSD). The face region is assumed to be a fixed width and height. The correlation window is slid over the search space and the SSD value is computed at each location. After all the positions have been visited, the position with the least SSD value is chosen as the best match candidate. This gives rise to the measurement, z_{t+1}.

Next step involves making use of the prediction and measurement to get the estimate of the next state. The final state estimate, s_{t+1}, is given by,

$$s_{t+1} = s_{t+1}^{-} + K_{t+1}(z_{t+1} - Hs_{t+1}^{-}) \tag{34}$$

where, K_{t+1} is called the Kalman gain given by,

$$K_{t+1} = \Sigma_{t+1}^{-} H^T (H \Sigma_{t+1}^{-} H^T + R)^{-1} \tag{35}$$

Finally, the covariance matrix is updated as,

$$\Sigma_{t+1} = (I - K_{t+1} H) \Sigma_{t+1}^{-} \tag{36}$$

where, I is a identity matrix.

After each time and measurement update pair, the Kalman filter recursively conditions current estimate on all of the past measurements and the process is repeated with the previous posterior estimates used to project or predict a new a priori estimate [1]. The trace of the state covariance matrix is often used to indicate the uncertainty of the estimated position [1].

2.2.3 Initialization

The Kalman filter is performed from the third frame of a sequence. The first two frames are assumed to have the feature point located well. The state variables are initialized accordingly. The position variables x_t, y_t are initialized to the position of the feature in the second frame. The velocity variables $v_{x,t}, v_{y,t}$ are initialized to the difference of the positions in the first and second frame.

$$x_t = x_2 \tag{37}$$

$$y_t = y_2 \tag{38}$$

$$v_{x,t} = x_2 - x_1 \tag{39}$$

$$v_{y,t} = y_2 - y_1 \tag{40}$$

The covariance matrix is initialized as,

$$\Sigma_t = \begin{bmatrix} 100 & 0 & 0 & 0 \\ 0 & 100 & 0 & 0 \\ 0 & 0 & 25 & 0 \\ 0 & 0 & 0 & 25 \end{bmatrix} \tag{41}$$

The matrices Q and R are assumed to be constant across all the frames and they are initialized as,

$$Q = \begin{bmatrix} 16 & 0 & 0 & 0 \\ 0 & 16 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix} \tag{42}$$

$$R = \begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix} \tag{43}$$

The values 16 and 4 in Q indicate that the positional and velocity error has a variance of 16 and 4, leading to a standard deviation of 4 and 2 pixels per frame. With lower values in the initial guess, the tracking seemed to lose track and drift from the face after a few frames.

Similarly, the value of 4 in R indicates that error in the measurement model is 2 pixels in x and y direction. The value of K, search window size scale factor was chosen to be 5.

2.2.4 Multiple Kalman tracking

So far we have discussed theory pertaining to tracking only one point in the image from frame to frame. To track an object effectively, the tracker needs to track its entire area or at the very least should track most of the interest points on the object. This is accomplished by using multiple Kalman tracking. In this technique, a Kalman filter is initialized on each of the feature points we are interested in tracking on the object. There is no sharing of parameters or state information between the different Kalman filters and they operate independent of each other on each frame. Once we have the tracking results for each frame in this manner, we can use it to perform the 3D reconstruction. It is observed that having multiple Kalman filters operating on an image at the same time significantly slows down the tracking process.

2.3 3D Reconstruction

The objective of 3D reconstruction is to recover the 3D shape of the object given its motion field estimated from a sequence of images. From the optical flow methods and the Kalman filter technique discussed in the previous sections, we were able to track a set of feature points of the object in a sequence of images. This information will be used to extract the shape of the object. We will now discuss the subspace factorization methods for rigid [1] and non-rigid body reconstruction [4].

2.3.1 Rigid reconstruction

Consider we have tracked M non-coplanar 3D points on the object in motion across N (N >= 3) frames. We will make the assumption that the object being tracked is rigid and the camera has an orthographic projection model.

Let $p_{ij} = (c_{ij}, r_{ij})$ denote the ‘j’th image point on the ‘i’th image frame. Let $\overline{c_i}$ and $\overline{r_i}$ be the centroid of the image points on the ‘i’th image frame. Let $P_j = (x_j, y_j, z_j)$ be the ‘j’th 3D point relative to the object frame and let $\overline{P}$ be the centroid of the 3D points. We convert the image coordinates to relative coordinates by subtracting the centroid.

$$c'_{ij} = c_{ij} - \overline{c_i} \tag{44}$$
$$r'_{ij} = r_{ij} - \overline{r_i} \tag{45}$$
$$P'_j = P_j - \overline{P} \tag{46}$$

They are related using the orthographic projection equation as,

$$\begin{bmatrix} c'_{ij} \\ r'_{ij} \end{bmatrix} = \begin{bmatrix} r_{i,1} \\ r_{i,2} \end{bmatrix} P'_j \tag{47}$$

where, $r_{i,1}$ and $r_{i,2}$ are the first two rows of the rotation matrix between camera frame ‘i’ and the object frame. Since, we have N images, each with M points, we can write,

$$W = RS \tag{48}$$

where,

W is a 2NxM matrix, R is a 2Nx3 matrix and S is a 3xM matrix given by,

$$W^{2N\times M} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1M} \\ r_{11} & r_{12} & \cdots & r_{1M} \\ c_{21} & c_{22} & \cdots & c_{2M} \\ r_{21} & r_{22} & \cdots & r_{2M} \\ \vdots & \vdots & \vdots & \vdots \\ c_{N1} & c_{N2} & \cdots & c_{NM} \\ r_{N1} & r_{N2} & \cdots & r_{NM} \end{pmatrix} \tag{49}$$

$$R = \begin{pmatrix} \mathbf{r}_{1,1} \\ \mathbf{r}_{1,2} \\ \vdots \\ \mathbf{r}_{N,1} \\ \mathbf{r}_{N,2} \end{pmatrix} \tag{50}$$

$$S = [P'_1 \; P'_2 \ldots, P'_M] \tag{51}$$

Note that the entries in W contain relative coordinates. The symbol ‘ from their notation has been dropped for convenience. R gives the relative orientation of each frame to the object frame. S contains the 3D coordinates of the feature points on the object. We can notice that, matrix W has a maximum rank of 3 since its component matrices R and S have a rank of 3. But due to image noise, rank of W may be more than 3. We will use singular value decomposition (SVD) to enforce the rank constraint. SVD of W is given by,

$$W = UDV^t \tag{52}$$

We retain only the 3 significant eigen values in D and replace the others with 0. Similarly, we keep the first 3 columns of U and V which correspond to the three largest eigen values in D. This results in a new rank-3 approximation of W given by,

$$W' = U'D'V'^t \tag{53}$$

where,
U’ is a 2Nx3 matrix, D’ is a 3x3 matrix and V’ is a Mx3 matrix.

We can now get an estimate of R and S by decomposing W’. We have,

$$R' = U'D'^{1/2} \tag{54}$$
$$S' = D'^{1/2}V'^t \tag{55}$$

The solution in equations (54) and (55) is only upto an affine transformation, since for any invertible 3x3 matrix Q, R = R'Q and S = inv(Q)S' also satisfy the equations. We make use of the knowledge of orthonormality constraints for each of the images' rotation matrix to determine Q. We have,

$$r_{i,1}{r_{i,1}}^t = 1 \tag{56}$$
$$r_{i,2}{r_{i,2}}^t = 1 \tag{57}$$
$$r_{i,1}{r_{i,2}}^t = 0 \tag{58}$$

Since, R = R'Q, we have,

$$r'_{i,1}QQ^t{r'_{i,1}}^t = 1 \tag{59}$$
$$r'_{i,2}QQ^t{r'_{i,2}}^t = 1 \tag{60}$$
$$r'_{i,1}QQ^t{r'_{i,2}}^t = 0 \tag{61}$$

where, $r'_{i,1}$ and $r'_{i,2}$ are the first and second rows of the rotation matrix R' for image i.

Given N images, we can solve for A = QQ^t subject to the constraint that A is a symmetric matrix and has only 6 unknowns. Given A, Q can be obtained using Choleski factorization. Once Q is obtained, the final motion estimate is

$$R = R'Q \tag{62}$$

and the final structure estimate is

$$S = Q^{-1}S' \tag{63}$$

2.3.2 Non-rigid reconstruction

A lot of structure from motion techniques assume that the 3D object being reconstructed is rigid. This assumption is not always true. For instance, we might want to reconstruct the 3D structure of an animal in motion or a person's face while talking. The popular Tomasi-Kanade Factorization method discussed previously is based assumes that the rank of the tracking matrix W is 3. Further the reconstruction occurs under orthographic projection. Here we have also implemented and evaluated the non-rigid 3D reconstruction method developed in [4]. The theoretical discussion which is next closely follows the paper itself.

This method proposes that the 3D shape in each frame is a linear combination of a set of basis shapes. Using this, the tracking matrix is found to be of higher rank and can be factorized into 3D shape and motion in a manner similar to the first method. This method demonstrates how 3D non-rigid shapes can be recovered for a scaled orthographic projection.

The core idea in this method is to describe the shape of a non-rigid object as a key-frame basis set given as $S_1, S_2, \ldots S_K$. Further, each key-frame S_i is a $3 \times M$ matrix describing M points. The shape of any specific configuration can be expressed as a linear combination of this basis set. This is given as

$$S = \sum_{i=1}^{K} l_i S_i \tag{64}$$

Here l_i is a scalar. Now if we consider the orthographic projection, the 3D coordinates of the M points in this configuration will relate to their 2D image coordinates according to the equation

$$\begin{bmatrix} u_1 & u_2 & \cdots & u_M \\ v_1 & v_1 & \cdots & v_M \end{bmatrix} = R\left(\mathrm{S} = \textstyle\sum_{i=1}^{K} l_i S_i\right) + T \tag{65}$$

Here (u_i, v_i) are image coordinates, R is the 3D rotation matrix and T is the 3D translation. We know that for orthographic projection, R contains only the first two rows of the actual camera rotation matrix. This is given as

$$R = \begin{bmatrix} r_1 & r_2 & r_3 \\ r_4 & r_5 & r_6 \end{bmatrix} \tag{66}$$

Just like in the Tomasi-Kanade Factorization approach, we can eliminate the T matrix by using relative coordinates. To do this we subtract the coordinate of each image point from the centroid of the object. Once we do this, we can express equation (65) as a matrix

$$\begin{bmatrix} u_1 & u_2 & \cdots & u_M \\ v_1 & v_1 & \cdots & v_M \end{bmatrix} = \begin{bmatrix} l_1 R & l_2 R & \cdots & l_K R \end{bmatrix} . \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_K \end{bmatrix} \tag{67}$$

Using the 2D tracking data from frame to frame available using any of the tracking methods discussed previously, we can express the relation in (67) for all such points in the form

$$W = \begin{bmatrix} u_1^{(1)} & \cdots & u_M^{(1)} \\ v_1^{(1)} & \cdots & u_M^{(1)} \\ u_1^{(2)} & \cdots & u_M^{(2)} \\ v_1^{(2)} & \cdots & v_M^{(2)} \\ & \vdots & \\ u_1^{(N)} & \cdots & u_M^{(N)} \\ v_1^{(N)} & \cdots & v_M^{(N)} \end{bmatrix} = \begin{bmatrix} l_1^{(1)} R^{(1)} & \cdots & l_K^{(1)} R^{(1)} \\ l_1^{(2)} R^{(2)} & \cdots & l_K^{(2)} R^{(2)} \\ & \vdots & \\ l_1^{(N)} R^{(N)} & \cdots & l_K^{(N)} R^{(N)} \end{bmatrix} . \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_K \end{bmatrix} \tag{68}$$

Here $(u_i^{(t)}, v_i^{(t)})$ denoted the i^{th} point in the t^{th} frame. W is the tracking matrix. It is easy to see that W has rank $3K$. It can be factored into two matrices Q and B using singular value decomposition and only considering the first $3K$ singular values and vectors as in the previous factorization method. Q contains the pose and B contains the key-frame basis set.

$$\mathrm{W}^{2N\times M} = \hat{U}.\hat{D}.\hat{V}^T = \hat{Q}^{2N\times 3K} . \hat{B}^{3K\times M} \tag{69}$$

We would like to now extract the camera rotation matrix $R^{(t)}$ and weights $l_i^{(t)}$ from the $\hat{Q}$ thus obtained. To represent this we use the two consecutive rows of $\hat{Q}$ that correspond to a particular time frame t as follows

$$q^{(t)} = \left[l_1^{(t)} R^{(t)} \quad \dots \quad l_K^{(t)} R^{(t)} \right] = \begin{bmatrix} l_1 r_1 & l_1 r_2 & l_1 r_3 & \dots & l_K r_1 & l_K r_2 & l_K r_3 \\ l_1 r_4 & l_1 r_5 & l_1 r_6 & \dots & l_K r_4 & l_K r_5 & l_K r_6 \end{bmatrix} \tag{70}$$

We can reorder this representation in such a way so as to get

$$\bar{q}^{(t)} = \begin{bmatrix} l_1 \\ l_2 \\ \vdots \\ l_K \end{bmatrix} . [r_1 \quad r_2 \quad r_3 \quad r_4 \quad r_5 \quad r_6] \tag{71}$$

This tells us that $\bar{q}^{(t)}$ has rank 1 and can be further factored into pose $\hat{R}^{(t)}$ and configuration weights $l_i^{(t)}$ by singular value decomposition. We can thus obtain this information successively for each frame by performing the above operations to all time blocks of $\hat{Q}$. The last step we need to take is to enforce the orthonormality of the rotation matrices. As discussed for the rigid factorization method, a linear transformation G is found by solving the least squares problem given below

$$[r_1 \quad r_2 \quad r_3] G G^T [r_1 \quad r_2 \quad r_3]^T = 1 \tag{72}$$

$$[r_4 \quad r_5 \quad r_6] G G^T [r_4 \quad r_5 \quad r_6]^T = 1 \tag{73}$$

$$[r_1 \quad r_2 \quad r_3] G G^T [r_4 \quad r_5 \quad r_6]^T = 0 \tag{74}$$

The G thus obtained can map all $\hat{R}^{(t)}$ into orthonormal matrices $R^{(t)}$. This inverse transformation must also be applied to the key-frame basis to keep the factorization consistent. In this manner, with the extracted 2D tracking data, we can form a non-rigid 3D shape matrix with K degrees of freedom.

3 Data

We worked on 5 different image sequences for this project. 3 of them were rigid bodies in motion and 2 of them were non-rigid. We took publicly available datasets [5] and [6] for testing the implementation initially. We shot 3 videos using a phone camera with a 640x480 resolution and 30 fps.

4 Procedure

We start off with manually marking the interest points in the object for the first three images in all the sequences. While it is understood that Optical Flow techniques are generally used for dense motion estimation, we use them here to track certain interest points only. The objective behind doing this is to ensure that the reconstructed 3D shape obtained is better suited for visualization. We have generally limited the number of frames to around 10 - 15 for the sequence, except for the face sequences for which we have used up to 50 frames so that enough non-rigid movement is obtained. The face dataset in [5] already had 68 annotated feature points for all the frames. We use the data from the first two frames' data for Kalman filter technique for initializing the state vector, whereas the third frame's data is used as initial position for optical flow estimation. All the data is stored in a 'data.txt' file inside each of the sequence folder.

Next we read the frames from each of the video sequences. For each of the frame read, we compute the estimated position in the next frame using either the iterative optical flow techniques or Kalman filter technique. Note that a Kalman filter needs to be initialized for each of the feature point tracked in the images. Also, for iterative optical flow estimation we used numerical approximation for gradients instead of cubic facet model fitting to reduce the implementation complexity. The estimate of the positions results in tracking of the feature points across the frames. This position data is stored in a matrix.

The position matrix stored is then passed to the reconstruction function, which computes the 3D structure estimate using either the rigid or non-rigid factorization. Finally, we visualize the extracted 3D points using a scatter plot. To help in visualizing the structure better, we connect the 3D points using lines.

5 Results

The results of the project are made up of two components; tracking results and the reconstruction results. We will be showing the tracking results for the three methods; optical flow methods 1 and 2, and Kalman filtering. The reconstruction results will be for rigid and non-rigid factorization methods.

The tracking result for sequence 'Hotel' is shown in figure 4, 5 and 6. The reconstruction result is shown in figure 7, 8 and 9. Note that the tracking output is shown only for the first and last frame of the sequence. The tracking result for all the frames can be seen in the attached video. For the reconstruction output, note that the reconstruction using rigid factorization gives only one output. But when we use the non-rigid factorization, we can recover 3D shape from all the frames in the sequence. In the figures, for the non-rigid factorization output, we are showing the reconstruction result for only the first and last frame of the sequence.

Qualitatively, we can observe that, optical flow method 1 and Kalman filtering performs better tracking than optical flow method 2. But this doesn't seem to have any impact on the reconstruction results. The reconstruction output seemed to be similar for all the three tracking techniques. It is hard to differentiate between the two reconstruction techniques (rigid and non-rigid) since the object being tracked is rigid.

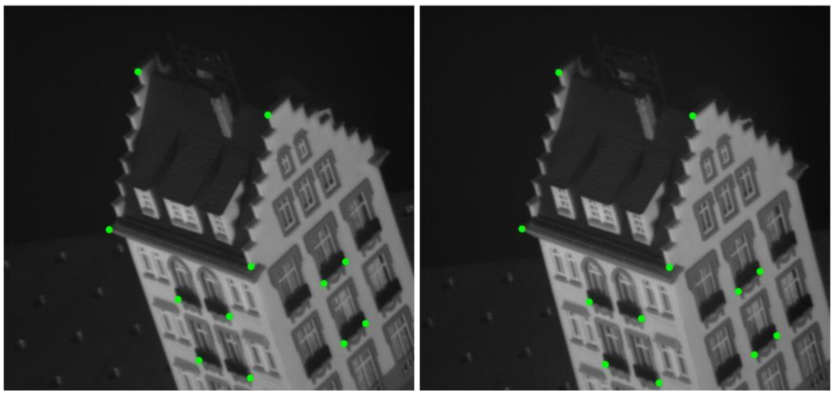


Figure 4: Tracking using Optical Flow method 1

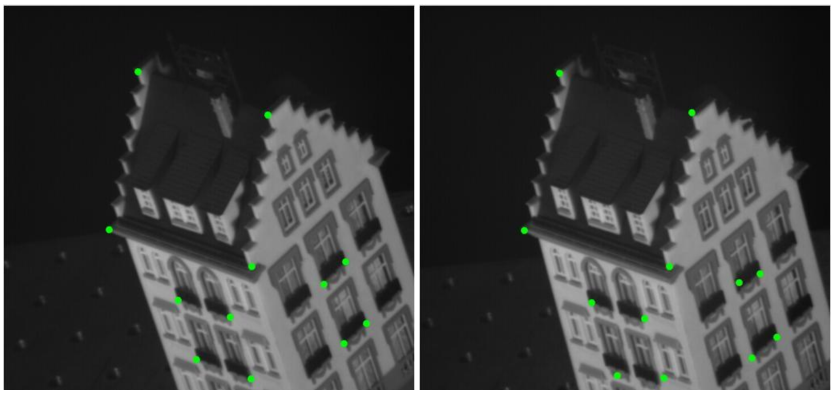


Figure 5: Tracking using Optical Flow method 2

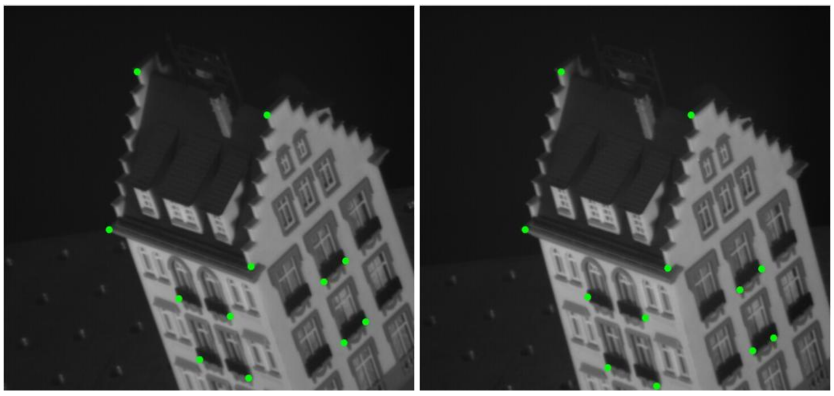


Figure 6: Tracking using Kalman filter

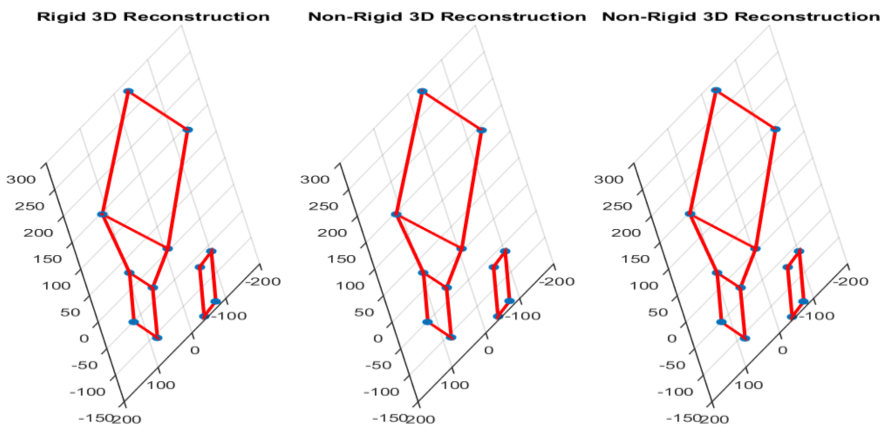


Figure 7: Reconstruction using optical flow method 1 output

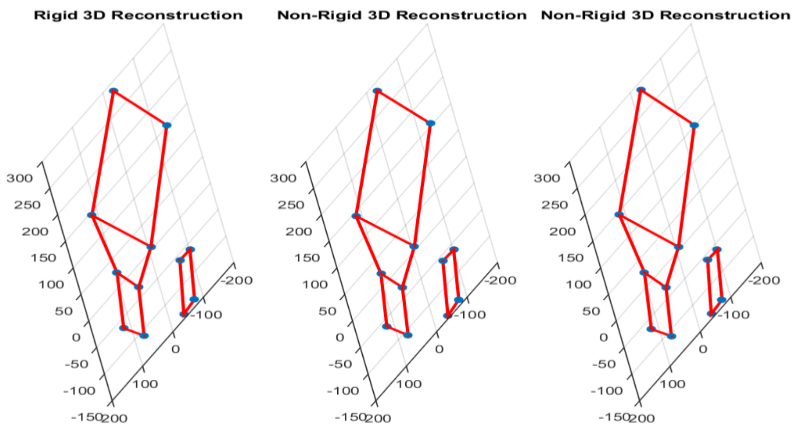


Figure 8: Reconstruction using optical flow method 2 output

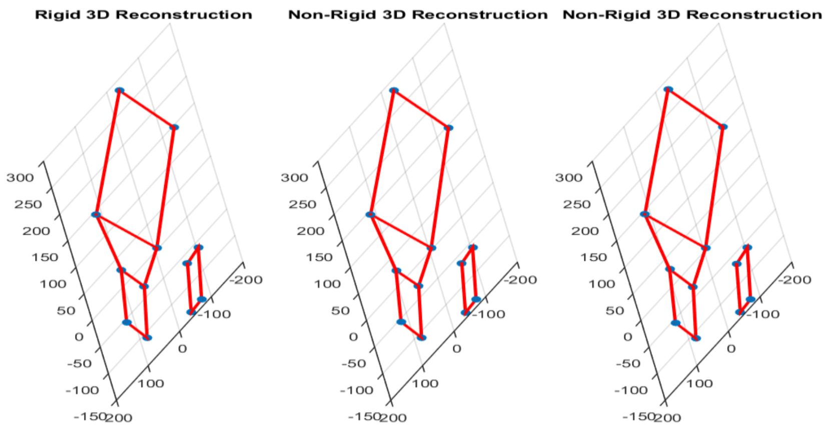


Figure 9: Reconstruction using kalman output

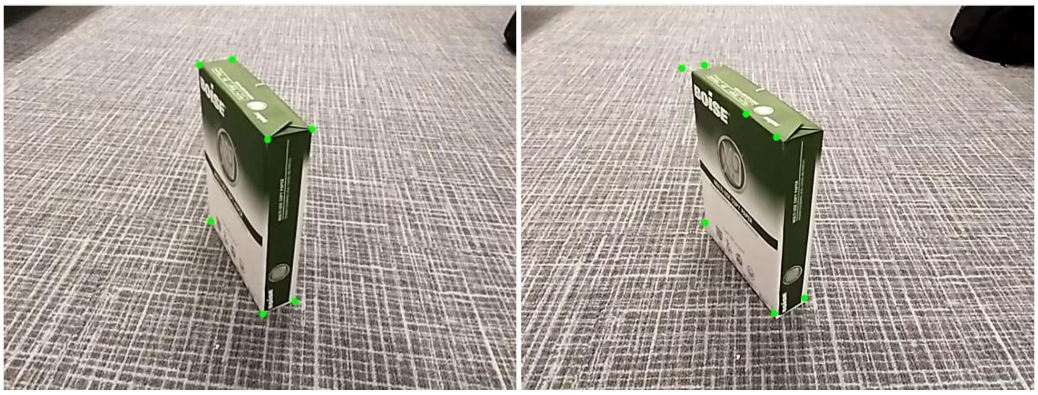


Figure 10: Tracking using optical flow method 1

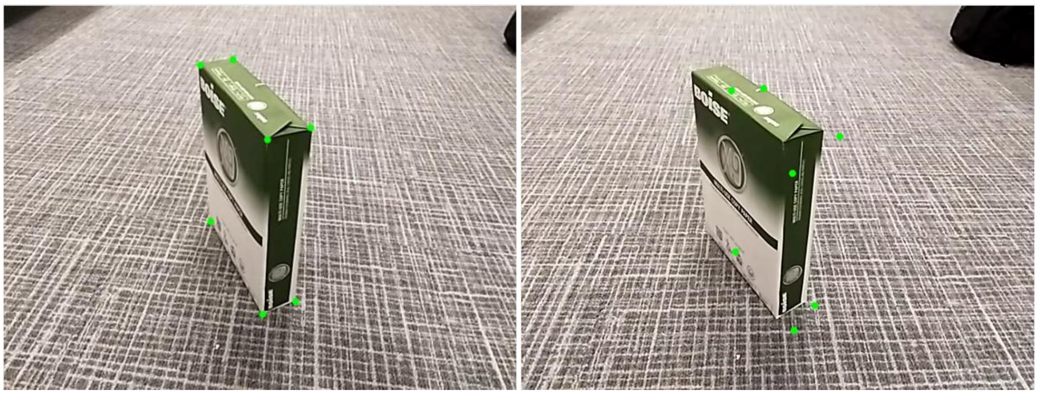


Figure 11: Tracking using optical flow method 2

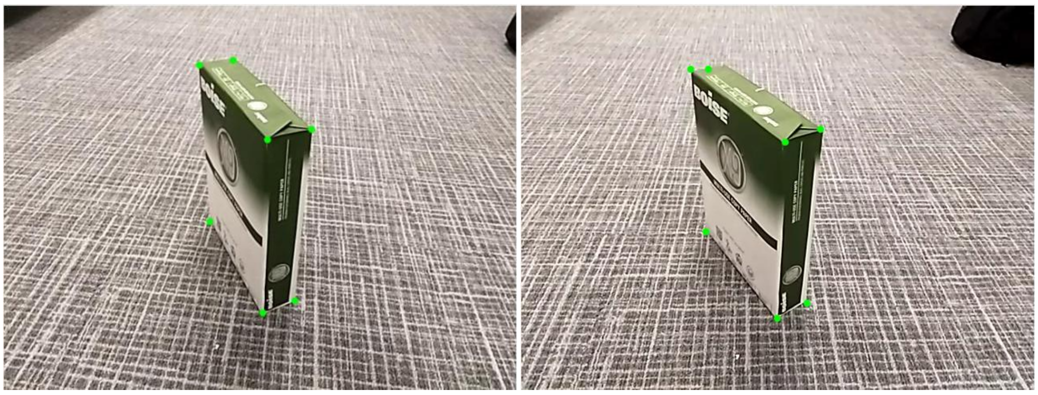


Figure 12: Tracking using kalman

The result for ‘Paper’ sequence is shown in figures 10 to 15. The Kalman filter technique produced the best tracking of the three methods. Consequently, the reconstruction done using the Kalman filter seems to produce the best output. All the edges seem to be perfectly orthogonal for the final reconstruction. Using the optical flow tracking data, the reconstruction resulted in skewed edges with both the reconstruction techniques.

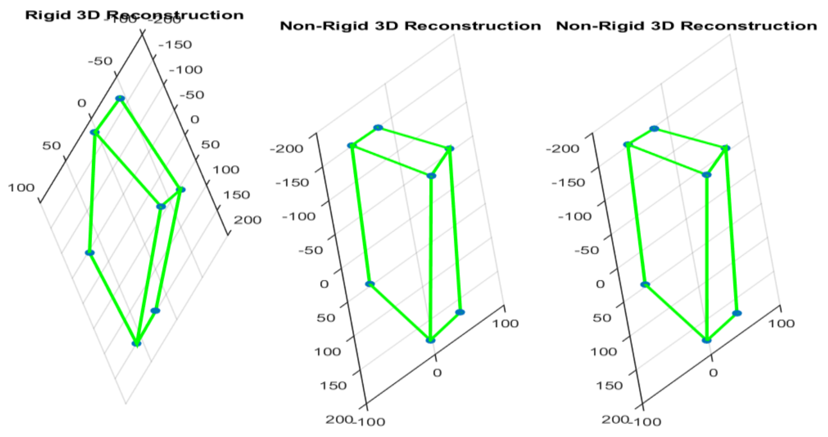


Figure 13: Reconstruction using optical flow method 1 output

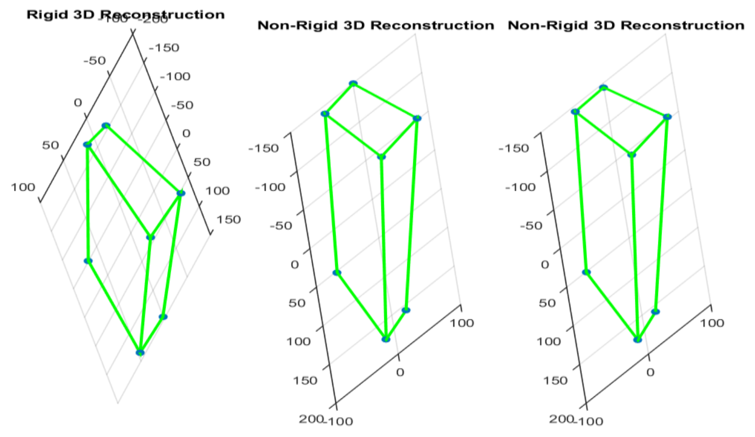


Figure 14: Reconstruction using optical flow method 2 output

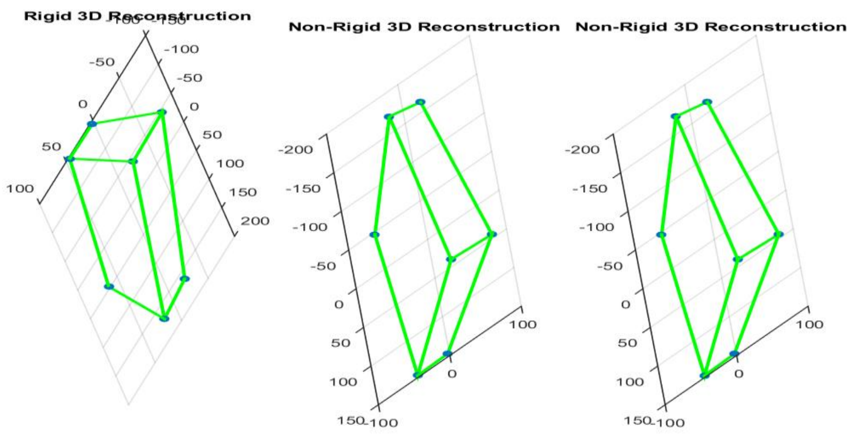


Figure 15: Reconstruction using kalman output

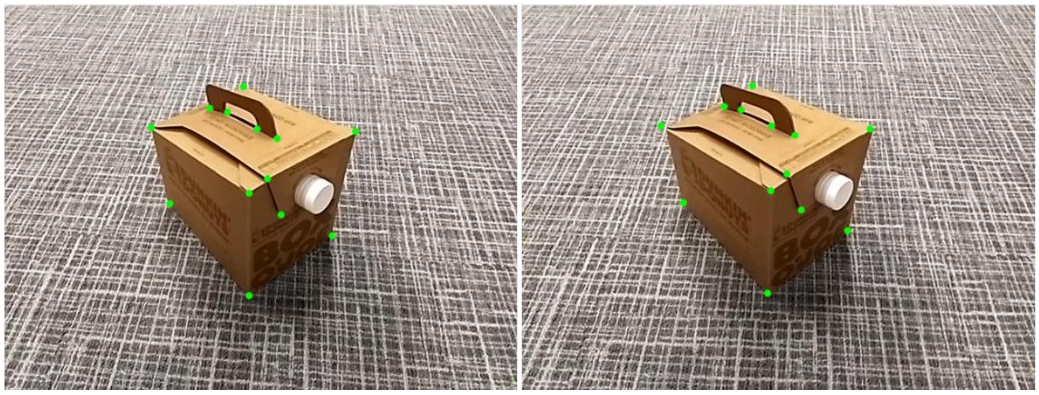


Figure 16: Tracking using optical flow method 1

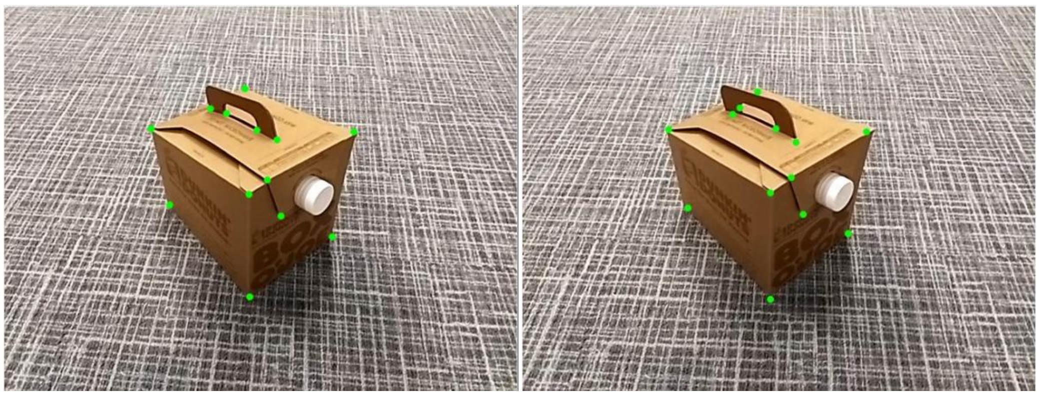


Figure 17: Tracking using optical flow method 2

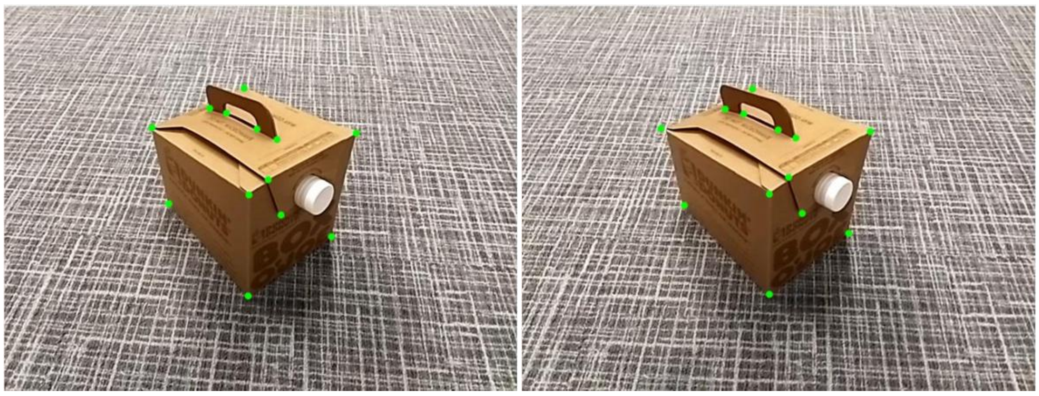


Figure 18: Tracking using Kalman

The result for the 'boxofjoe' sequence is shown in figures 16 to 21. The tracking results show that optical flow method 1 and Kalman filter technique are superior to optical flow method 2. However, the tracking of the major corners of the box were tracked correctly by all the methods and hence the reconstruction using any of the three techniques gives a good output. Also, there isn't much difference between rigid and non-rigid reconstruction since the object is rigid.

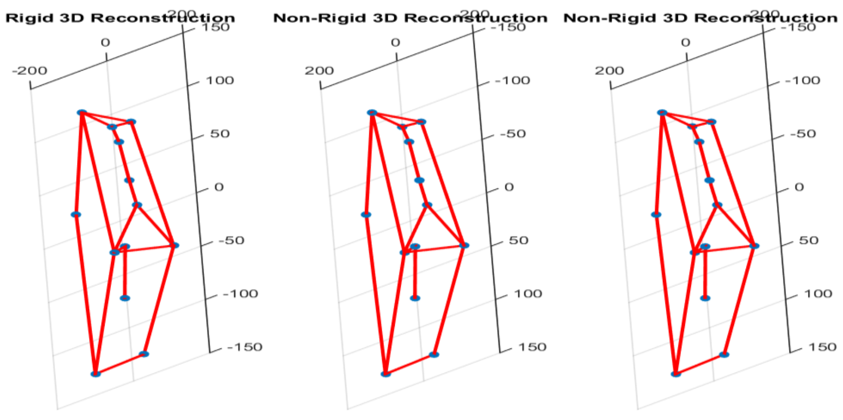


Figure 19: Reconstruction using optical flow method 1 output

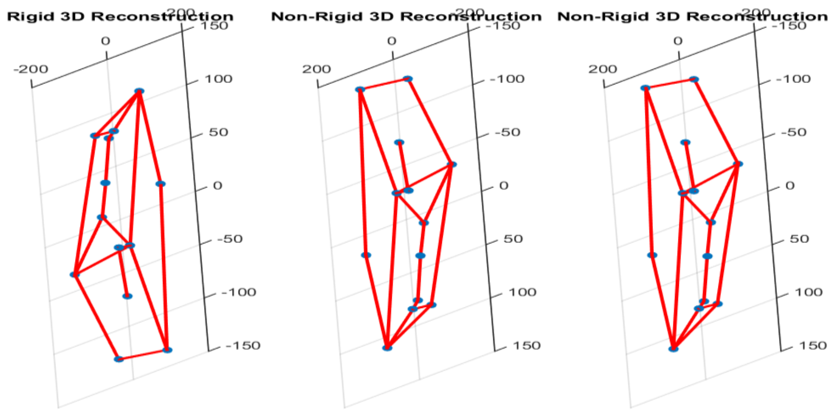


Figure 20: Reconstruction using optical flow method 2 output

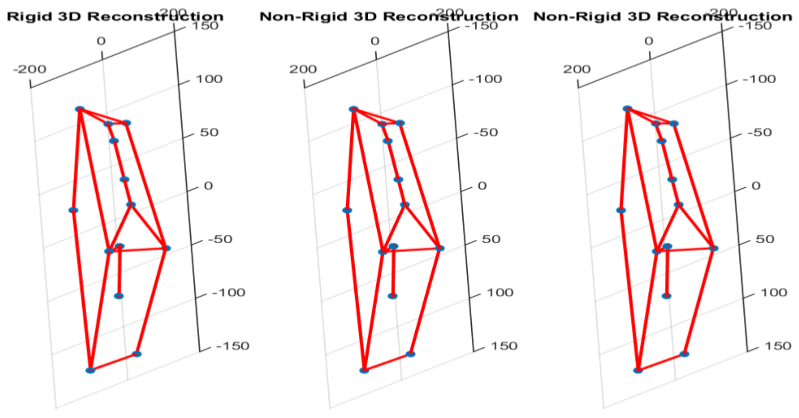


Figure 21: Reconstruction using kalman output

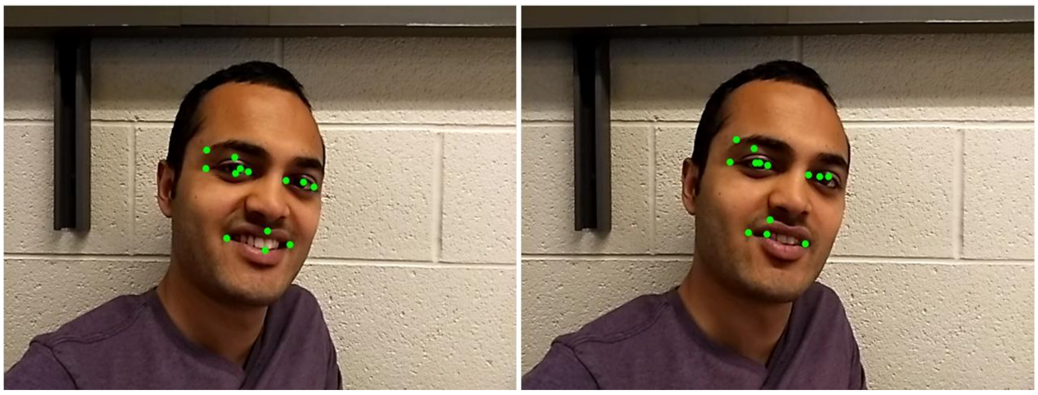


Figure 22: Tracking using optical flow method 1

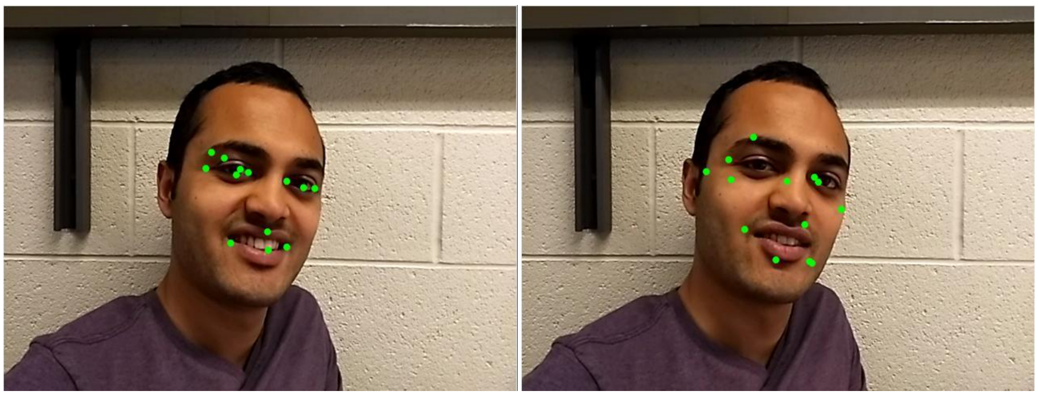


Figure 23: Tracking using optical flow method 2

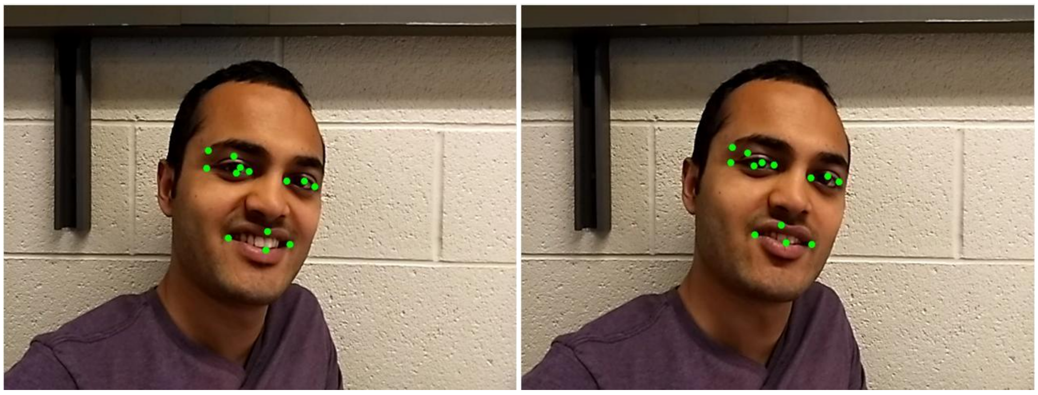


Figure 24: Tracking using kalman

The result for ‘gyan’ sequence is shown in figures 22 to 27. The tracking of any of the methods is not very satisfactory. However, Kalman does the best among the three. This may be due to the feature points near the eyes being in shadows. The reconstruction done using both the rigid and non-rigid formulization surprisingly produced similar results. Also, note that there is movement of the mouth from the first to the last frame and the non-rigid reconstruction helps in this regard by producing different reconstruction for each of the frames in the sequence.

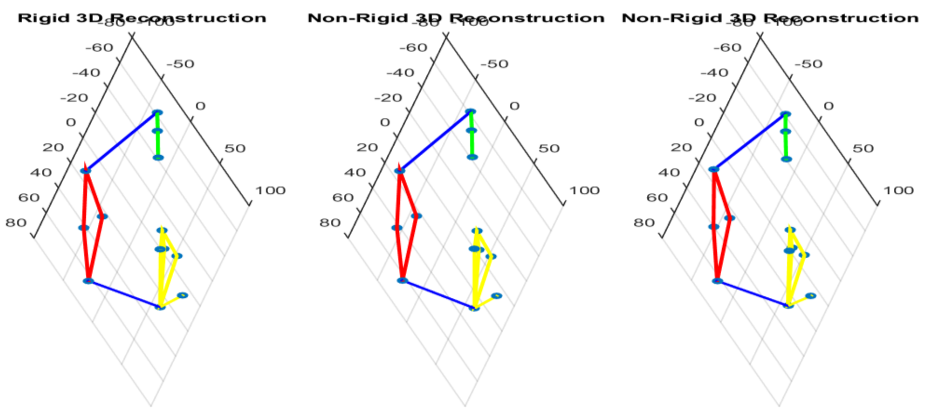


Figure 25: Reconstruction using optical flow method 1 output

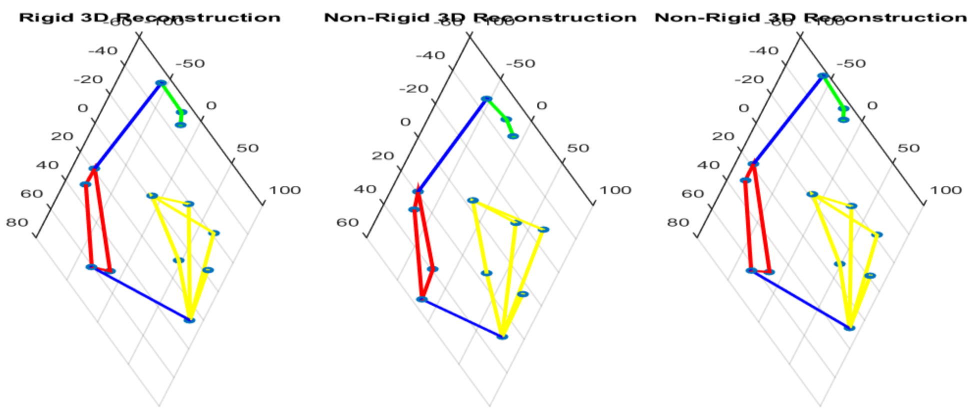


Figure 26: Reconstruction using optical flow method 2 output

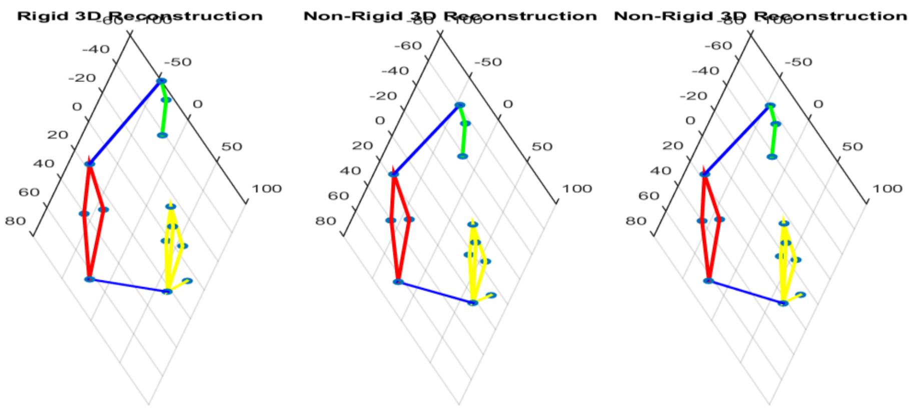


Figure 27: Reconstruction using kalman output

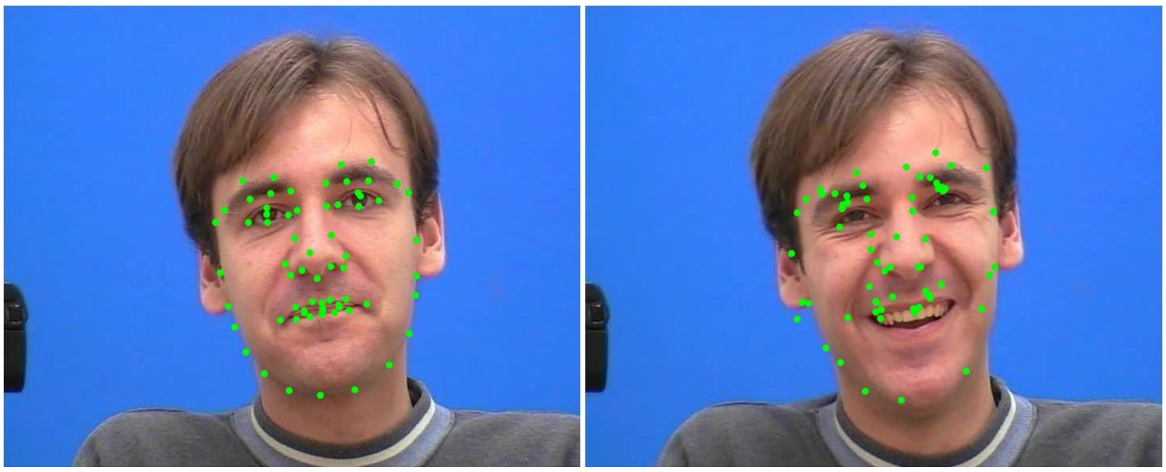


Figure 28: Tracking using optical flow method 1

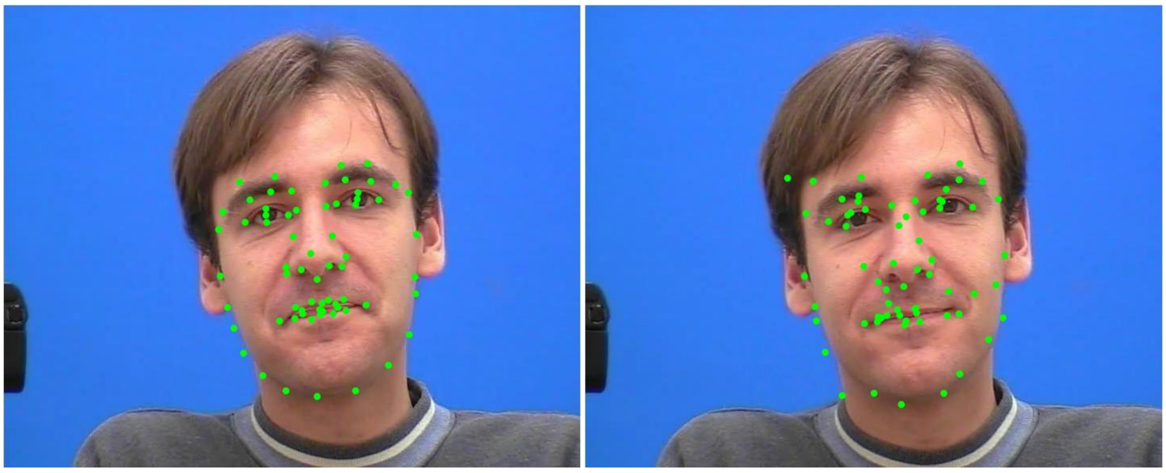


Figure 29: Tracking using optical flow method 2

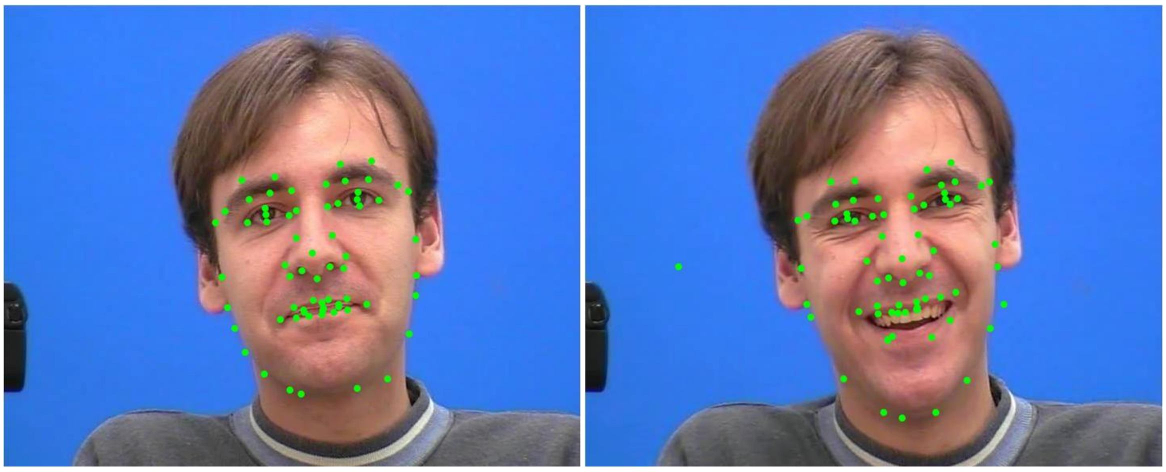


Figure 30: Tracking using kalman

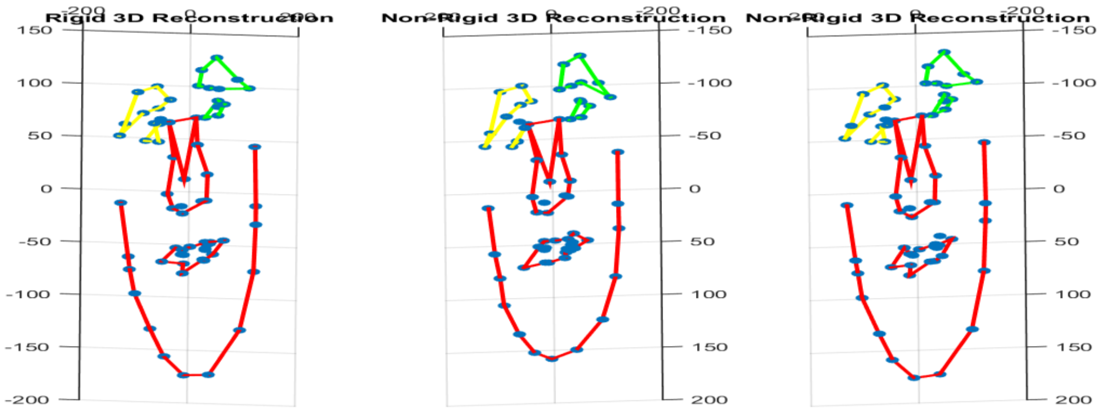


Figure 31: Reconstruction using optical flow method 1 output

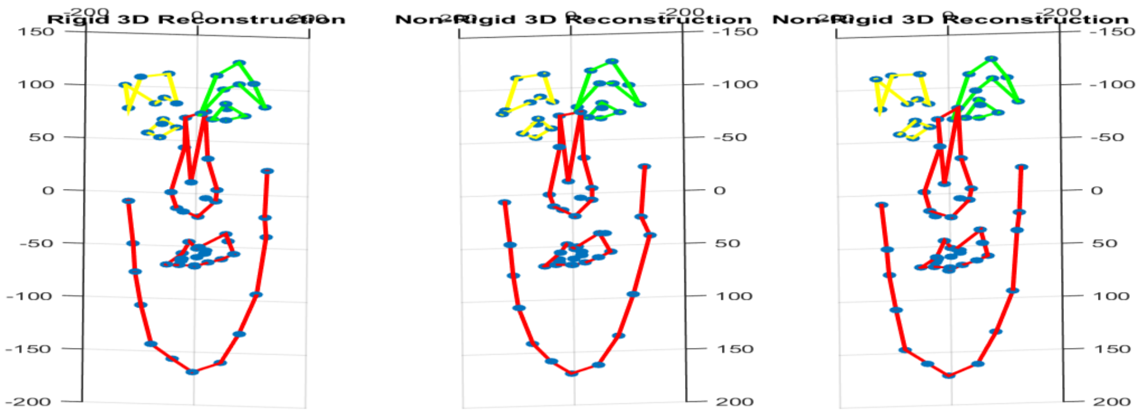


Figure 32: Reconstruction using optical flow method 2 output

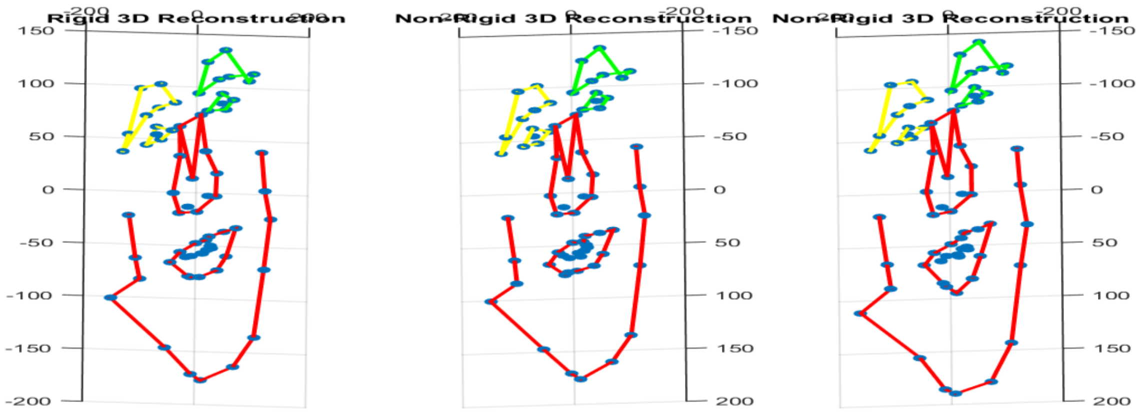


Figure 33: Reconstruction using kalman output

The result for ‘franck’ sequence is shown in figures 28 to 33. The Kalman filter again produces the best tracking output. Note that, the result for optical flow method 2 was not suitable for reconstruction due to tracking errors. Hence, the results shown here for optical flow method 2 are for fewer frames than for optical flow method 1 and Kalman. As can be seen by comparing figures 30 and 33, non rigid reconstruction is able to reconstruct the subtle face movements (opening/closing of mouth), which the rigid reconstruction cannot.

6 Conclusion

In this project, we implemented a feature tracking system based on iterative optical flow techniques and Kalman filter technique. The tracked points were then used to get an estimate of the shape of the object using the factorization methods. We implemented both rigid and non-rigid body factorization methods for reconstruction. We found that Kalman filter technique, although slow, performed very well for the tracking problem over the optical flow methods. Non-rigid factorization methods generally performed better than the rigid body factorization for structure estimation of the object. It is easy to see that the 3D structure from motion approach would benefit greatly by the use of feature extraction techniques like SIFT or SURF. This would enable accurate, more reliable tracking between frames and thus yield better 3D structure estimation. Though the methods, applied in this project yield promising results, there is much that can be done to improve them. As an extension to the project, it would be interesting to see a mobile application which can reconstruct 3D object using the live video stream from the phone’s camera.

7 References

[1] Lecture notes, Prof. Qiang Ji, “*https://www.ecse.rpi.edu/Homepages/qji/CV/ecse6650_lecture_notes.html*”
[2] Emanuele Trucco, Alessandro Verri, “*Introductory Techniques for 3D Computer Vision Approach*”
[3] Jae Kyu Suhr, “*http://web.yonsei.ac.kr/jksuhr/articles/Kanade-Lucas-Tomasi%20Tracker.pdf*”
[4] C. Bregler, A. Hertzmann and H. Biermann, “*Recovering Non-Rigid 3D Shape from Image Streams*”, CVPR, 2000
[5] “http://www-prima.inrialpes.fr/FGnet/data/01-TalkingFace/talking_face.html”
[6] “http://www.cs.cmu.edu/~./abhishek/softwares.html”

Annexure A: Code

```
%Main file
function FinalProject
close all
clear all
flowMethods = {'OF1','OF2','Kalman'};
reconstructionMethods = {'Rigid','Non-rigid'};
flowMethodUsed = 1;
reconstructionMethodUsed = 1;
%Get data
videoName =
{'./Videos/Cube_Short','./Videos/Hotel','./Videos/franck_images-
0999/images','./Videos/boxofjoe','./Videos/paper','./Videos/gyan'};
imageName = {'','hotel.seq','franck_','','',''};
imageEnd = {'.png','.png','.jpg'};
numFrames = [7,15,20,20,20,20];
numFeatures = [7,12,68,13,7,13];
ID = 6;%2-6

outVideoName = sprintf('%s/output_%d.avi',videoName{ID},flowMethodUsed);
outputVideoObj = VideoWriter(outVideoName);
outputVideoObj.FrameRate = 5;%5
open(outputVideoObj);
%%
matPosition = zeros(2*(numFrames(ID) - 4),numFeatures(ID));%matrix
containing all feature positions for all frames
numFramesToRead = 5;
%Get the initial state of the feature points
if(ID == 1)
    stateCurrent = getInitialState(videoName{ID}, numFeatures(ID),
flowMethods{flowMethodUsed});
elseif(ID == 3)
    stateCurrent = zeros(numFeatures(ID),4);
    ptsFolder = sprintf('%s/../../franck_points/points',videoName{ID});
    for i = 1:3
        fileID = fopen(sprintf('%s/franck_%05d.pts',ptsFolder,i-1));
        %3 useless info
        dummy = fgetl(fileID);
        dummy = fgetl(fileID);
        dummy = fgetl(fileID);

        for j = 1:numFeatures(ID)
            temp = fscanf(fileID,'%f%f',2);
            stateCurrent(j,3:4) = temp' - stateCurrent(j,1:2);
            stateCurrent(j,1:2) = temp';
        end
        fclose(fileID);
    end
else
    stateCurrent = getInitialState(videoName{ID}, numFeatures(ID),
flowMethods{flowMethodUsed});
end
%%
kalmanParams = 0;
%Flow estimation and Tracking
for iter1 = 3:numFrames(ID)-2
    fprintf('Processing image: %d\n',iter1);
    for iter2 = 1:numFramesToRead
        if(ID == 3)
```

```
            imageSet(:,:,:,iter2) = 
imread(sprintf('%s/franck_%05d.jpg',videoName{ID},iter1-3+iter2 - 1));
        elseif(ID == 2)
            %Check this
            imageSet(:,:,:,iter2) = 
imread(sprintf('%s/hotel.seq%d.png',videoName{ID},iter1-3+iter2 - 1));
        else
            imageSet(:,:,:,iter2) = 
imread(sprintf('%s/%d.png',videoName{ID},iter1-3+iter2));
        end
    end

    [stateNext, kalmanParams] = getNextState(stateCurrent, imageSet, 
flowMethods{flowMethodUsed}, kalmanParams);

    %Show images
    if(flowMethodUsed == 3)
        imshow(imageSet(:,:,:,3));
    else
        imshow(imageSet(:,:,:,4));
    end
    hold on;
    for iterPlot = 1:numFeatures(ID)
        plot(stateNext(iterPlot,1), 
stateNext(iterPlot,2),'Marker','o','MarkerFaceColor','g','MarkerEdgeColor',
'g');
%          rect = rectangle('Position',[stateNext(iterPlot,1) - 25, 
stateNext(iterPlot,2) - 25, 50, 50]);
%          rect.LineWidth = 2;
%          rect.EdgeColor = 'r';
    end
    hold off;
    pause(0.001);

    figure_1 = getframe;
    [outputImage, map] = frame2im(figure_1);
    %Make a movie
    writeVideo(outputVideoObj,outputImage);

    %store all positions in a matrix
    matPosition(2*(iter1-2) - 1:2*(iter1-2),:) = stateNext(:,1:2)';
    stateCurrent = stateNext;
end
close(outputVideoObj);
saveName = sprintf('%s/matPosition_%d.mat',videoName{ID},flowMethodUsed);
save(saveName,'matPosition');
%%
%Perform reconstruction
pts3d_Rigid = perform3DReconstructionRigid(matPosition);
pts3d_NonRigid = perform3DReconstructionNonRigid(matPosition);
%Visualize the recovered 3D data
visualizeData(pts3d_Rigid, pts3d_NonRigid, ID);
figure_1 = getframe;
[outputImage, map] = frame2im(figure_1);
outputImageName = sprintf('%s/Reconstruction_%d.png', videoName{ID}, 
flowMethodUsed);
saveas(gcf,outputImageName);
% imwrite(outputImage, outputImageName);
end
```

```
%Function to get the initial state of the feature points
%Inputs
    %videoName - name of the video being used
    %numFeatures - number of features tracked
    %flowMethod - method used for tracking
%Ouputs
    %stateCurrent - initial state vector
function stateCurrent = getInitialState(videoName, numFeatures, flowMethod)
data = importdata(sprintf('%s/data.txt',videoName));

if(strcmp(flowMethod,'Kalman'))
    stateCurrent = zeros(numFeatures, 4);
    %Columns 1,2 contain x,y position data for 2nd frame-needed for kalman
    stateCurrent(:,1:2) = data(numFeatures+1:2*numFeatures,:);
    %Columns 3,4 contain vx,vy velocity data for 2nd frame-needed for
kalman
    stateCurrent(:,3:4) = data(numFeatures+1:2*numFeatures,:) -
data(1:numFeatures,:);
else
    stateCurrent = zeros(numFeatures, 2);
    %Columns 1,2 contain x,y position data for 3rd frame-needed for optical
flow
    stateCurrent(:,1:2) = data(2*numFeatures+1:end,:);
end
end

%Function to get the next state of the feature points
%Inputs
    %stateCurrent - state vector in the current frame
    %imageSet - set of input images
    %flowMethod - method used for tracking
    %kalmanParams - covariance matrix for each feature point needed for
recursion
%Ouputs
    %stateNext - state vector in the next frame
    %kalmanParams - covariance matrix for each feature point needed for
recursion
function [stateNext, kalmanParams] = getNextState(stateCurrent, imageSet,
flowMethod, kalmanParams)
numImages = size(imageSet,4);
midImageIndex = uint32(ceil(numImages/2));
numFeatures = size(stateCurrent, 1);
stateNext = zeros(size(stateCurrent));

if(strcmp(flowMethod,'OF1'))
    %Initialization parameters
    parameters = getConfigParams(1);
    %Call OF1 method and get position from optical flow vectors (just add?
vector addition;)
%     stateNext = calcOpticalFlow_1(imageSet, numImages, stateCurrent,
parameters.OFWinSizeSpatial, parameters.thresCondition,
parameters.thresMin);
    stateNext = iterativeOpticalFlow_1(imageSet, numImages, stateCurrent,
parameters.OFWinSizeSpatial, ...
                                    parameters.thresCondition,
parameters.thresMin, parameters.thresStopSSD, parameters.thresStopIters);
elseif(strcmp(flowMethod,'OF2'))
    %Initialization parameters
    parameters = getConfigParams(1);
    %Call OF2 method and get position from optical flow vectors (just add?
vector addition;)
%     stateNext = calcOpticalFlow_2(imageSet, numImages, stateCurrent,
parameters.OFWinSizeSpatial, ...
%                                     parameters.OFWinSizeTemporal,
parameters.thresCondition, parameters.thresMin);
    stateNext = iterativeOpticalFlow_2(imageSet, numImages, stateCurrent,
parameters.OFWinSizeSpatial, ...
                                    parameters.OFWinSizeTemporal,
parameters.thresCondition, parameters.thresMin, ...
                                    parameters.thresStopSSD,
parameters.thresStopIters);
else
    %Initialization parameters
    parameters = getConfigParams(2);

    %Call Kalman method
    stateNextDummy = stateCurrent;
    if(sum(sum(kalmanParams)) ~= 0)
        sigmaNextDummy = kalmanParams;
    else
        sigmaNextDummy = repmat(parameters.sigma_init,[1 1 numFeatures]);
    end
    kalmanParamsDummy =
zeros(size(parameters.sigma_init,1),size(parameters.sigma_init,2),numFeatur
es);
    %Multiple kalman; One for each feature point
    for iter3 = 1:numFeatures
        [stateNextDummy2, sigmaNextDummy2] =
KalmanFiltering(imageSet(:,:,:,midImageIndex - 1),
imageSet(:,:,:,midImageIndex), stateNextDummy(iter3,:)', parameters,
sigmaNextDummy(:,:,iter3));
        stateNext(iter3,:) = stateNextDummy2';
        kalmanParamsDummy(:,:,iter3) = sigmaNextDummy2;
    end
    kalmanParams = kalmanParamsDummy;
end

end

%Function for returning the configuration parameters
%Inputs
    %methodType - flag determining which approach is used for feature
tracking
%Outputs
    %parameters - configuration parameters
function parameters = getConfigParams(methodType)
if(methodType == 1)
    parameters = getOpticalFlowConfigParams();
else
    parameters = getKalmanConfigParams();
end
end

%Function for returning the optical flow configuration parameters
%Inputs
    %
%Outputs
    %parameters - optical flow configuration parameters
function parameters = getOpticalFlowConfigParams()
```

```
%Window size
f1 = 'OFWinSizeSpatial';v1 = 9;
f2 = 'OFWinSizeTemporal';v2 = 5;
%Thresholds
f3 = 'thresCondition';v3 = 1e3;
f4 = 'thresMin';v4 = 1e-4;
f5 = 'thresStopSSD';v5 = 0.1;
f6 = 'thresStopIters';v6 = 40;
parameters = struct(f1,v1,f2,v2,f3,v3,f4,v4,f5,v5,f6,v6);
end

%Function for returning the kalman filter configuration parameters
%Inputs
    %
%Outputs
    %parameters - kalman filter configuration parameters
function parameters = getKalmanConfigParams()
%For SSD
f1 = 'searchWidth';v1 = 10;
f2 = 'searchHeight';v2 = 10;

f3 = 'windowSizeScale';v3 = 5;
f4 = 'colorProcessing';v4 = 0;

f5 = 'STM';v5 = eye(4);v5(1,3) = 1;v5(2,4) = 1;
f6 = 'H';v6 = zeros(2,4);v6(1,1) = 1;v6(2,2) = 1;
f7 = 'Q';v7 = 4*eye(4);v7(1,1) = 16;v7(2,2) = 16;
f8 = 'R';v8 = 4*eye(2);
f9 = 'sigma_init';v9 = 25*eye(4);v9(1,1) = 100;v9(2,2) = 100;

parameters = struct(f1,v1,f2,v2,f3,v3,f4,v4,f5,v5,f6,v6,f7,v7,f8,v8,f9,v9);
end

%Estimate optical flow using Lucas-Kanade method (image brightness
constancy)
%Inputs
    %imageSet - set of input images
    %numImages - number of input images
    %stateCurrent - feature positions where optical flow needs to be
estimated
    %OFWinSizeSpatial - window size for constancy assumption
    %thresCondition - threshold for max value of the condition of the
matrix
    %thresMin - threshold for min value of the flow vector magnitude
%Ouputs
    %stateNext - feature positions in the next image
function stateNext = calcOpticalFlow_1(imageSetRGB, numImages,
stateCurrent, OFWinSizeSpatial, thresCondition, thresMin)
vctX = stateCurrent(:,1);
vctY = stateCurrent(:,2);
flowVct = zeros(size(vctX,1),2);

for i = 1:numImages
    if(size(imageSetRGB,3) ~= 1)
        imageSet(:,:,i) = rgb2gray(imageSetRGB(:,:,:,i));
    else
        imageSet(:,:,i) = (imageSetRGB(:,:,:,i));
    end
end
```

```
midImageIndex = int32((numImages + 1)/2);%equivalent to ceil
OFWinSizeSpatial_1 = int32(OFWinSizeSpatial - 1);
OFWinSizeSpatial_2 = int32(OFWinSizeSpatial_1/2);
[m,n] = size(imageSet(:,:,1));
flowImage = zeros(m, n, 2);

%Smoothen the image first
imageSetSmoothed = zeros(size(imageSet));
for i = 1:numImages
    imageSetSmoothed(:,:,i) = imgaussfilt(imageSet(:,:,i));
end

%Compute gradients using differences
% [gx,gy] = gradient(imageSetSmoothed(:,:,midImageIndex));
imageTranslateX = imtranslate(imageSetSmoothed(:,:,midImageIndex),[-1 0]);
imageTranslateY = imtranslate(imageSetSmoothed(:,:,midImageIndex),[0 -1]);
imageTranslateX(:,end) = imageTranslateX(:,end - 1);
imageTranslateY(end,:) = imageTranslateY(end - 1,:);

gx = imageTranslateX - imageSetSmoothed(:,:,midImageIndex);
gy = imageTranslateY - imageSetSmoothed(:,:,midImageIndex);
gt = imageSetSmoothed(:,:,midImageIndex + 1) -
imageSetSmoothed(:,:,midImageIndex);

for i = 1:length(vctX)
    x = vctX(i);
    y = vctY(i);

    matA = [reshape(gx(y-OFWinSizeSpatial_2:y+OFWinSizeSpatial_2, x-
OFWinSizeSpatial_2:x+OFWinSizeSpatial_2),[],1) ...
            reshape(gy(y-OFWinSizeSpatial_2:y+OFWinSizeSpatial_2, x-
OFWinSizeSpatial_2:x+OFWinSizeSpatial_2),[],1)];
    matB = -reshape(gt(y-OFWinSizeSpatial_2:y+OFWinSizeSpatial_2, x-
OFWinSizeSpatial_2:x+OFWinSizeSpatial_2),[],1);
    %Check condition number of A
    cond_matA = cond(matA);
    if(cond_matA > thresCondition)
        flow = [0 0]';
    else
        flow = inv(matA'*matA)*matA'*matB;
        if(norm(flow) < thresMin)
            flow = [0 0]';
        end
    end
    flowVct(i,:) = flow;
end

%get position from optical flow vectors
stateNext = stateCurrent + flowVct;
end

%Estimate optical flow using Lucas-Kanade method (image brightness
constancy)
%Inputs
    %imageSet - set of input images
    %numImages - number of input images
    %stateCurrent - feature positions where optical flow needs to be
estimated
    %OFWinSizeSpatial - window size for constancy assumption
    %OFWinSizeTemporal - temporal window size for constancy assumption
    %thresCondition - threshold for max value of the condition of the
matrix
    %thresMin - threshold for min value of the flow vector magnitude
%Ouputs
    %stateNext - feature positions in the next image
function stateNext = calcOpticalFlow_2(imageSetRGB, numImages,
stateCurrent, OFWinSizeSpatial, OFWinSizeTemporal, thresCondition,
thresMin)
vctX = stateCurrent(:,1);
vctY = stateCurrent(:,2);
flowVct = zeros(size(vctX,1),2);

for i = 1:numImages
    if(size(imageSetRGB,3) ~= 1)
        imageSet(:,:,i) = rgb2gray(imageSetRGB(:,:,:,i));
    else
        imageSet(:,:,i) = (imageSetRGB(:,:,:,i));
    end
end

midImageIndex = int32((numImages + 1)/2);%equivalent to ceil
OFWinSizeSpatial_1 = int32(OFWinSizeSpatial - 1);
OFWinSizeSpatial_2 = int32(OFWinSizeSpatial_1/2);
OFWinSizeTemporal_1 = int32(OFWinSizeTemporal - 1);
OFWinSizeTemporal_2 = int32(OFWinSizeTemporal_1/2);

%Construct D matrix
matD = zeros(OFWinSizeSpatial*OFWinSizeSpatial*OFWinSizeTemporal, 20);
index = 1;
for t = -OFWinSizeTemporal_2:1:OFWinSizeTemporal_2
    for x = -OFWinSizeSpatial_2:1:OFWinSizeSpatial_2
        for y = -OFWinSizeSpatial_2:1:OFWinSizeSpatial_2
            matD(index,:) =
[1,x,y,t,x*x,x*y,y*y,y*t,t*t,x*t,x^3,y*x^2,x*y^2,y^3,t*y^2,y*t^2,t^3,t*x^2,
x*t^2,x*y*t];
            index = index + 1;
        end
    end
end
matD_pinv = inv(matD'*matD)*matD';%pseudo-inverse of D

for i = 1:length(vctX)
    x = vctX(i);
    y = vctY(i);

    matJ = reshape(imageSet(y-OFWinSizeSpatial_2:y+OFWinSizeSpatial_2, x-
OFWinSizeSpatial_2:x+OFWinSizeSpatial_2, ...
                    midImageIndex-
OFWinSizeTemporal_2:midImageIndex+OFWinSizeTemporal_2),[],1);
    a = matD_pinv*double(matJ);%Get the model coefficients

    %Construct matrix A, B for linear system
    matA = [a(2) a(3); 2*a(5) a(6); a(6) 2*a(7); a(10) a(8)];
    matB = -[a(4) a(10) a(8) 2*a(9)]';
    matA = [a(2) a(3) -1; 2*a(5) a(6) 0; a(6) 2*a(7) 0; a(10) a(8) 0];
    matB = -[a(4) a(10) a(8) 2*a(9)]';

    %Check condition number of A
    cond_matA = cond(matA);
    if(cond_matA > thresCondition)
```

```
        flow = [0 0 0]';
    else
        flow = inv(matA'*matA)*matA'*matB;
        if(norm(flow(1:2)) < thresMin)
            flow = [0 0 flow(3)]';
        end
    end
    flowVct(i,:) = flow(1:2);
end

%get position from optical flow vectors
stateNext = stateCurrent + flowVct;
end

%Estimate optical flow using iterative Lucas-Kanade method (image
brightness constancy)
%Inputs
    %imageSetRGB - set of input images
    %numImages - number of input images
    %stateCurrent - feature positions where optical flow needs to be
estimated
    %OFWinSizeSpatial - window size for constancy assumption
    %thresCondition - threshold for max value of the condition of the
matrix
    %thresMin - threshold for min value of the flow vector magnitude
    %thresStopSSD - threshold for difference of SSD scores
    %thresStopIters - threshold for number of iterations of flow estimation
%Ouputs
    %stateNext - feature positions in the next image
function stateNext = iterativeOpticalFlow_1(imageSetRGB, numImages,
stateCurrent, OFWinSizeSpatial, thresCondition, thresMin, thresStopSSD,
thresStopIters)
vctX = stateCurrent(:,1);
vctY = stateCurrent(:,2);
flowVct = zeros(size(vctX,1),2);

for i = 1:numImages
    if(size(imageSetRGB,3) ~= 1)
        imageSet(:,:,i) = rgb2gray(imageSetRGB(:,:,:,i));
    else
        imageSet(:,:,i) = (imageSetRGB(:,:,:,i));
    end
end

midImageIndex = int32((numImages + 1)/2);%equivalent to ceil
OFWinSizeSpatial_1 = int32(OFWinSizeSpatial - 1);
OFWinSizeSpatial_2 = int32(OFWinSizeSpatial_1/2);
[m,n] = size(imageSet(:,:,1));
flowImage = zeros(m, n, 2);

%Smoothen the image first
imageSetSmoothed = zeros(size(imageSet));
for i = 1:numImages
    imageSetSmoothed(:,:,i) = imgaussfilt(imageSet(:,:,i));
end

%Compute gradients using differences
% [gx,gy] = gradient(imageSetSmoothed(:,:,midImageIndex));
imageTranslateX = imtranslate(imageSetSmoothed(:,:,midImageIndex),[-1 0]);
imageTranslateY = imtranslate(imageSetSmoothed(:,:,midImageIndex),[0 -1]);
```

```
imageTranslateX(:,end) = imageTranslateX(:,end - 1);
imageTranslateY(end,:) = imageTranslateY(end - 1,:);

gx = imageTranslateX - imageSetSmoothed(:,:,midImageIndex);
gy = imageTranslateY - imageSetSmoothed(:,:,midImageIndex);
gt = imageSetSmoothed(:,:,midImageIndex + 1) -
imageSetSmoothed(:,:,midImageIndex);

for i = 1:length(vctX)
    x = vctX(i);
    y = vctY(i);
    [gridX, gridY] = meshgrid(x-OFWinSizeSpatial_2:x+OFWinSizeSpatial_2, y-
OFWinSizeSpatial_2:y+OFWinSizeSpatial_2);
    matA1 = interp2(gx, double(gridX), double(gridY));
    matA2 = interp2(gy, double(gridX), double(gridY));
    matA = [reshape(matA1,[],1) reshape(matA2,[],1)];
    matBTemp = interp2(gt, double(gridX), double(gridY));
    matB = -reshape(matBTemp,[],1);
%     matA = [reshape(gx(y-OFWinSizeSpatial_2:y+OFWinSizeSpatial_2, x-
OFWinSizeSpatial_2:x+OFWinSizeSpatial_2),[],1) ...
%             reshape(gy(y-OFWinSizeSpatial_2:y+OFWinSizeSpatial_2, x-
OFWinSizeSpatial_2:x+OFWinSizeSpatial_2),[],1)];
%   matB = -reshape(gt(y-OFWinSizeSpatial_2:y+OFWinSizeSpatial_2, x-
OFWinSizeSpatial_2:x+OFWinSizeSpatial_2),[],1);

    scoreSSD = sum(matB.^2);
    %Check condition number of A
    cond_matA = cond(matA);
    zeroFlag = 0;
    if(cond_matA > thresCondition)
        flow = [0 0]';
        zeroFlag = 1;
    else
        flow = inv(matA'*matA)*matA'*matB;
        if(norm(flow) < thresMin)
            flow = [0 0]';
            zeroFlag = 1;
        end
    end

    iter = 0;
    u = flow(1); v = flow(2);
    netFlow = [u v];
    diffScoreSSD = realmax;

    while((iter < thresStopIters) && (diffScoreSSD > thresStopSSD) &&
(zeroFlag == 0))
        imageShifted = interp2(imageSetSmoothed(:,:,midImageIndex + 1),
double(gridX) + u, double(gridY) + v);
        gtNew = imageShifted - imageSetSmoothed(y-
OFWinSizeSpatial_2:y+OFWinSizeSpatial_2, x-
OFWinSizeSpatial_2:x+OFWinSizeSpatial_2, midImageIndex);
        matB = -reshape(gtNew,[],1);
        newScoreSSD = sum(matB.^2);

        flow = inv(matA'*matA)*matA'*matB;
        if(norm(flow) < thresMin)
            flow = [0 0]';
            zeroFlag = 1;
        end
        u = flow(1);
        v = flow(2);
        diffScoreSSD = abs(scoreSSD - newScoreSSD);
        if(newScoreSSD > scoreSSD)
            break;
        end
        scoreSSD = newScoreSSD;
        netFlow = netFlow + [u v];
        iter = iter + 1;
    end
%     if(zeroFlag)
%         break;
%     end
    flowVct(i,:) = netFlow;
end

%get position from optical flow vectors
stateNext = stateCurrent(:,1:2) + flowVct;
end

%Estimate optical flow using iterative method (image brightness change
constancy)
%Inputs
    %imageSetRGB - set of input images
    %numImages - number of input images
    %stateCurrent - feature positions where optical flow needs to be
estimated
    %OFWinSizeSpatial - window size for constancy assumption
    %OFWinSizeTemporal - temporal window size for constancy assumption
    %thresCondition - threshold for max value of the condition of the
matrix
    %thresMin - threshold for min value of the flow vector magnitude
    %thresStopSSD - threshold for difference of SSD scores
    %thresStopIters - threshold for number of iterations of flow estimation
%Ouputs
    %stateNext - feature positions in the next image
function stateNext = iterativeOpticalFlow_2(imageSetRGB, numImages,
stateCurrent, OFWinSizeSpatial, OFWinSizeTemporal, thresCondition,
thresMin, thresStopSSD, thresStopIters)
vctX = stateCurrent(:,1);
vctY = stateCurrent(:,2);
flowVct = zeros(size(vctX,1),2);

for i = 1:numImages
    if(size(imageSetRGB,3) ~= 1)
        imageSet(:,:,i) = rgb2gray(imageSetRGB(:,:,:,i));
    else
        imageSet(:,:,i) = (imageSetRGB(:,:,:,i));
    end
end
[m,n,ch] = size(imageSet(:,:,1));
midImageIndex = int32((numImages + 1)/2);%equivalent to ceil
OFWinSizeSpatial_1 = int32(OFWinSizeSpatial - 1);
OFWinSizeSpatial_2 = int32(OFWinSizeSpatial_1/2);
OFWinSizeTemporal_1 = int32(OFWinSizeTemporal - 1);
OFWinSizeTemporal_2 = int32(OFWinSizeTemporal_1/2);

%Smoothen the image first
imageSetSmoothed = zeros(size(imageSet));
for i = 1:numImages
```

```
    imageSetSmoothed(:,:,i) = imgaussfilt(imageSet(:,:,i));
end
%Compute gradients using differences
%Gradients at t
imageTranslateXplus_t = imtranslate(imageSetSmoothed(:,:,midImageIndex),[-1
0]);%It(x+1)
imageTranslateYplus_t = imtranslate(imageSetSmoothed(:,:,midImageIndex),[0
-1]);%It(y+1)
imageTranslateXminus_t = imtranslate(imageSetSmoothed(:,:,midImageIndex),[1
0]);%It(x - 1)
imageTranslateYminus_t = imtranslate(imageSetSmoothed(:,:,midImageIndex),[0
1]);%It(y - 1)

imageTranslateXplus_t(:,end) = imageTranslateXplus_t(:,end - 1);
imageTranslateYplus_t(end,:) = imageTranslateYplus_t(end - 1,:);
imageTranslateXminus_t(:,1) = imageTranslateXplus_t(:,2);
imageTranslateYminus_t(1,:) = imageTranslateYplus_t(2,:);

gx_t = imageTranslateXplus_t - imageSetSmoothed(:,:,midImageIndex);
gy_t = imageTranslateYplus_t - imageSetSmoothed(:,:,midImageIndex);

%Gradients at t+1
imageTranslateXplus_t1 = imtranslate(imageSetSmoothed(:,:,midImageIndex +
1),[-1 0]);%Itp1(x+1)
imageTranslateYplus_t1 = imtranslate(imageSetSmoothed(:,:,midImageIndex +
1),[0 -1]);
imageTranslateXplus_t1(:,end) = imageTranslateXplus_t1(:,end - 1);
imageTranslateYplus_t1(end,:) = imageTranslateYplus_t1(end - 1,:);

gx_t1 = imageTranslateXplus_t1 - imageSetSmoothed(:,:,midImageIndex + 1);
gy_t1 = imageTranslateYplus_t1 - imageSetSmoothed(:,:,midImageIndex + 1);

gt = imageSetSmoothed(:,:,midImageIndex + 1) -
imageSetSmoothed(:,:,midImageIndex);

gxTranslateY = imtranslate(gx_t,[0 -1]);

gxx = imageTranslateXplus_t + imageTranslateXminus_t -
2*imageSetSmoothed(:,:,midImageIndex);
gxy = gxTranslateY - gx_t;
gyy = imageTranslateYplus_t + imageTranslateYminus_t -
2*imageSetSmoothed(:,:,midImageIndex);
gxt = gx_t1 - gx_t;
gyt = gy_t1 - gy_t;
gtt = imageSetSmoothed(:,:,midImageIndex + 1) +
imageSetSmoothed(:,:,midImageIndex - 1) -
2*imageSetSmoothed(:,:,midImageIndex);
zeroFlag = 0;
for i = 1:length(vctX)
    x = vctX(i);
    y = vctY(i);
    [gridX, gridY] = meshgrid(x-OFWinSizeSpatial_2:x+OFWinSizeSpatial_2, y-
OFWinSizeSpatial_2:y+OFWinSizeSpatial_2);

    matA = [interp2(gx_t,x,y) interp2(gy_t,x,y) -1; ...
            interp2(gxx,x,y) interp2(gxy,x,y) 0; ...
            interp2(gxy,x,y) interp2(gyy,x,y) 0; ...
            interp2(gxt,x,y) interp2(gyt,x,y) 0];
    matB = -[interp2(gt,x,y) interp2(gxt,x,y) interp2(gyt,x,y)
interp2(gtt,x,y)]';
```

```
    %Construct matrix A, B for linear system
%{
    matA = [gx_t(y,x) gy_t(y,x) -1; ...
            gxx(y,x) gxy(y,x) 0; ...
            gxy(y,x) gyy(y,x) 0; ...
            gxt(y,x) gyt(y,x) 0];
    matB = -[gt(y,x) gxt(y,x) gyt(y,x) gtt(y,x)]';
%}
    scoreSSD = sum(matB.^2);
    %Check condition number of A
    cond_matA = cond(matA);
    if(cond_matA > thresCondition)
        flow = [0 0 0]';
        zeroFlag = 1;
    else
        flow = inv(matA'*matA)*matA'*matB;
        if(norm(flow(1:2)) < thresMin)
            flow = [0 0 flow(3)]';
            zeroFlag = 1;
        end
    end

    iter = 0;
    u = flow(1); v = flow(2);
    netFlow = [u v];
    diffScoreSSD = realmax;
    while((iter < thresStopIters) && (diffScoreSSD > thresStopSSD) &&
(zeroFlag == 0) ...
            && (x-OFWinSizeSpatial_2+u >= 1) && (x+OFWinSizeSpatial_2+u <=
n) ...
            && (y-OFWinSizeSpatial_2+v >= 1) && (y+OFWinSizeSpatial_2+v <=
m))
        imageShifted = interp2(imageSetSmoothed(:,:,midImageIndex + 1),
double(gridX) + u, double(gridY) + v);

        gx_imageShifted = imtranslate(imageShifted,[-1 0]) - imageShifted;
        gxt_temp = gx_imageShifted - interp2(gx_t,double(gridX),
double(gridY));
        gy_imageShifted = imtranslate(imageShifted,[0 -1]) - imageShifted;
        gyt_temp = gy_imageShifted - interp2(gy_t,double(gridX),
double(gridY));

        gt_temp = imageShifted -
interp2(imageSetSmoothed(:,:,midImageIndex),double(gridX), double(gridY));
%          imageSetSmoothed(y-OFWinSizeSpatial_2:y+OFWinSizeSpatial_2,x-
OFWinSizeSpatial_2:x+OFWinSizeSpatial_2,midImageIndex);
        gtt_temp = gt_temp + interp2(imageSetSmoothed(:,:,midImageIndex -
1),double(gridX), double(gridY)) ...
                                        -
interp2(imageSetSmoothed(:,:,midImageIndex),double(gridX), double(gridY));
%          imageSetSmoothed(y-OFWinSizeSpatial_2:y+OFWinSizeSpatial_2,x-
OFWinSizeSpatial_2:x+OFWinSizeSpatial_2,midImageIndex - 1)...
%                      - imageSetSmoothed(y-
OFWinSizeSpatial_2:y+OFWinSizeSpatial_2,x-
OFWinSizeSpatial_2:x+OFWinSizeSpatial_2,midImageIndex);

        matA = [interp2(gx_t,x,y) interp2(gy_t,x,y) -1; ...
                interp2(gxx,x,y) interp2(gxy,x,y) 0; ...
                interp2(gxy,x,y) interp2(gyy,x,y) 0; ...
                gxt_temp(3,3) gyt_temp(3,3) 0];
        matB = -[gt_temp(3,3) gxt_temp(3,3) gyt_temp(3,3) gtt_temp(3,3)]';

        newScoreSSD = sum(matB.^2);

        flow = inv(matA'*matA)*matA'*matB;
        if(norm(flow) < thresMin)
            flow = [0 0]';
            zeroFlag = 1;
        end
        u = flow(1);
        v = flow(2);
        diffScoreSSD = abs(scoreSSD - newScoreSSD);
        if(newScoreSSD > scoreSSD)
            break;
        end
        scoreSSD = newScoreSSD;
        netFlow = netFlow + [u v];
        iter = iter + 1;
    end
%     if(zeroFlag)
%         break;
%     end
    flowVct(i,:) = netFlow;
end

%get position from optical flow vectors
stateNext = stateCurrent(:,1:2) + flowVct;
end

%Function for tracking the feature across images using kalman filter
%Inputs
    %imageCurrent - current image
    %imageNext - next image
    %stateCurrent - state in current image
    %parameters - kalman filter configuration parameters
    %sigmaCurrent - covariance estimate of current state
%Outputs
    %stateNext - state in next image
    %sigmaNext - covariance estimate of next state
function [stateNext, sigmaNext] = KalmanFiltering(imageCurrent, imageNext,
stateCurrent, parameters, sigmaCurrent)
STM = parameters.STM;
H = parameters.H;
Q = parameters.Q;
R = parameters.R;

searchWidth = parameters.searchWidth;
searchHeight = parameters.searchHeight;
windowSizeScale = parameters.windowSizeScale;
colorProcessing = parameters.colorProcessing;
searchSize = [searchWidth, searchHeight]';

stateNextEstimate = computeStateEstimate(stateCurrent, STM);
sigmaNextEstimate = computeSigmaEstimate(sigmaCurrent, STM, Q);
kalmanGain = computeKalmanGain(sigmaNextEstimate, H, R);
stateMeasurement = getMeasurement(imageCurrent, imageNext, stateCurrent,
stateNextEstimate, sigmaNextEstimate, searchSize, windowSizeScale,
colorProcessing);
```

```
stateNext = updateState(stateNextEstimate, stateMeasurement, kalmanGain,
H);
sigmaNext = updateSigma(sigmaNextEstimate, kalmanGain, H);
end

%estimate next state of the system
%Inputs
    %stateCurrent - state in current image
    %STM - state transition matrix
%Ouputs
    %stateNextEstimate - estimate of the next state
function stateNextEstimate = computeStateEstimate(stateCurrent, STM)
stateNextEstimate = STM*stateCurrent;
end

%compute estimate of the covariance
%Inputs
    %sigmaCurrent - covariance estimate of current state
    %STM - state transition matrix
    %Q - covariance of noise in covariance estimate
%Ouputs
    %sigmaNextEstimate - covariance estimate of next state
function sigmaNextEstimate = computeSigmaEstimate(sigmaCurrent, STM, Q)
sigmaNextEstimate = STM*sigmaCurrent*STM' + Q;
end

%compute kalman gain of the system
%Inputs
    %sigmaNextEstimate - covariance estimate of next state
    %H - measurement matrix
    %R - covariance of noise in measurement
%Ouputs
    %kalmanGain - computed kalman gain
function kalmanGain = computeKalmanGain(sigmaNextEstimate, H, R)
tempMat_1 = sigmaNextEstimate*H';
tempMat_2 = H*tempMat_1 + R;
kalmanGain = tempMat_1*inv(tempMat_2);
end

%measure the next state of the system
%Inputs
    %inputImageCurrent - current image
    %inputImageNext - next image
    %stateCurrent - state in current image
    %stateNextEstimate - estimate of the next state
    %sigmaNextEstimate - covariance estimate of next state
    %faceSize - face bounding box
    %windowSizeThreshold - scale factor for search window size
    %colorProcessing - flag to decide which image to work on HSV/grayscale
%Ouputs
    %stateMeasurement - measurement of next state
function stateMeasurement = getMeasurement(inputImageCurrent,
inputImageNext, stateCurrent, stateNextEstimate, sigmaNextEstimate,
faceSize, windowSizeThreshold, colorProcessing)
[imageHeight, imageWidth, imageChannels] = size(inputImageCurrent);
%Determine window size
eigenValues = eig(sigmaNextEstimate(1:2,1:2));
windowSizeX = int32(windowSizeThreshold*sqrt(eigenValues(1)));
windowSizeY = int32(windowSizeThreshold*sqrt(eigenValues(2)));
```

```
%Get face region in current frame
faceSizeWidth_1 = faceSize(1)/2;
faceSizeHeight_1 = faceSize(2)/2;
xMin = int32(stateCurrent(1) - faceSizeWidth_1);
xMax = int32(xMin + faceSize(1) - 1);
yMin = int32(stateCurrent(2) - faceSizeHeight_1);
yMax = int32(yMin + faceSize(2) - 1);

if(colorProcessing)
    inputImageCurrent = rgb2hsv(inputImageCurrent);
    inputImageNext = rgb2hsv(inputImageNext);
    tempMatCurrent = double(inputImageCurrent(yMin:yMax, xMin:xMax,:));
    tempMatMask = ones(size(tempMatCurrent));
%     tempMatMask(faceSizeHeight_1-9:faceSizeHeight_1+9,faceSizeWidth_1-
9:faceSizeWidth_1+9,:) = 0;
else
    %Grayscale images
    if(size(inputImageCurrent,3) ~= 1)
        inputImageCurrentGray = double(rgb2gray(inputImageCurrent));
        inputImageNextGray = double(rgb2gray(inputImageNext));
    else
        inputImageCurrentGray = double(inputImageCurrent);
        inputImageNextGray = double(inputImageNext);
    end
    tempMatCurrent = inputImageCurrentGray(yMin:yMax, xMin:xMax);
    tempMatMask = ones(size(tempMatCurrent));
%     tempMatMask(faceSizeHeight_1-9:faceSizeHeight_1+9,faceSizeWidth_1-
9:faceSizeWidth_1+9) = 0;
end
tempMatCurrent = tempMatCurrent.*tempMatMask;

vectorSSD = zeros(windowSizeX*windowSizeY,1);

%Get the search window dimensions
windowSizeX_1 = windowSizeX/2;
windowSizeY_1 = windowSizeY/2;
xMin = stateNextEstimate(1) - windowSizeX_1;
xMax = xMin + windowSizeX - 1;
yMin = stateNextEstimate(2) - windowSizeY_1;
yMax = yMin + windowSizeY - 1;
index = 1;

minValLoop = realmax;
minIndexLoopX = 0;
minIndexLoopY = 0;

%Iterate through all pixels in search window
for y = yMin:yMax
    for x = xMin:xMax
        yMinTemp = y - faceSizeHeight_1;
        yMaxTemp = yMinTemp + faceSize(2) - 1;
        xMinTemp = x - faceSizeWidth_1;
        xMaxTemp = xMinTemp + faceSize(1) - 1;

        %Handle border cases
        xMinChange = 1; xMaxChange = 0; yMinChange = 1; yMaxChange = 0;
        if (xMinTemp < 1)
            xMaxChange = 1 - xMinTemp;
            xMinTemp = 1;
        end
        if (xMaxTemp > imageWidth)
            xMinChange = xMaxTemp - imageWidth + 1;
            xMaxTemp = imageWidth;
        end
        if (yMinTemp < 1)
            yMaxChange = 1 - yMinTemp;
            yMinTemp = 1;
        end
        if (yMaxTemp > imageHeight)
            yMinChange = yMaxTemp - imageHeight + 1;
            yMaxTemp = imageHeight;
        end
        if(colorProcessing)
            tempMatNext = zeros(faceSize(2),faceSize(1),imageChannels);
            tempMatNext(yMinChange:(faceSize(2) - yMaxChange),
xMinChange:(faceSize(1) - xMaxChange),:) ...
                        = double(inputImageNext(yMinTemp:yMaxTemp,
xMinTemp:xMaxTemp,:));
            tempMatNext = tempMatNext.*tempMatMask;
            vectorSSD(index) = (sum(sum(sum((tempMatCurrent -
tempMatNext).*(tempMatCurrent - tempMatNext)))));%perform SSD
        else
            tempMatNext = zeros(faceSize(2),faceSize(1));
            tempMatNext(yMinChange:(faceSize(2) - yMaxChange),
xMinChange:(faceSize(1) - xMaxChange)) ...
                        = inputImageNextGray(yMinTemp:yMaxTemp,
xMinTemp:xMaxTemp);
            tempMatNext = tempMatNext.*tempMatMask;
            vectorSSD(index) = (sum(sum((tempMatCurrent -
tempMatNext).*(tempMatCurrent - tempMatNext))));%perform SSD
        end
        %Store the best candidate
        if (vectorSSD(index) < minValLoop)
            minValLoop = vectorSSD(index);
            minIndexLoopX = x;
            minIndexLoopY = y;
        end
        index = index + 1;
    end
end
stateMeasurement = double([minIndexLoopX minIndexLoopY]');
end

%update the state of the system
%Inputs
    %stateNextEstimate - estimate of the next state
    %stateMeasurement - measurement of next state
    %kalmanGain - kalman gain factor
    %H - measurement matrix
%Ouputs
    %stateNext - updated next state
function stateNext = updateState(stateNextEstimate, stateMeasurement,
kalmanGain, H)
stateNext = stateNextEstimate + kalmanGain*(stateMeasurement -
H*stateNextEstimate);
end

%update the covariance matrix
%Inputs
    %sigmaNextEstimate - estimate of the covariance matrix
    %kalmanGain - kalman gain factor
    %H - measurement matrix
%Ouputs
    %sigmaNext - updated covariance matrix
function sigmaNext = updateSigma(sigmaNextEstimate, kalmanGain, H)
tempMat_1 = kalmanGain*H;
sigmaNext = (eye(size(tempMat_1,1)) - tempMat_1)*sigmaNextEstimate;
end

%Function for performing rigid factorization for 3D reconstruction
%Inputs
    %matPosition - matrix containing the 2D points of all feature points in
all the frames
%Outputs
    %points3D - matrix containing the estimated 3D points of all feature
points
function points3D = perform3DReconstructionRigid(matPosition)

%centroid subtraction
[numImagesX2,numPoints] = size(matPosition);
numImages = uint32(numImagesX2/2);
matCentroid = sum(matPosition, 2)/numPoints;
matPositionRelative = matPosition - repmat(matCentroid,1,numPoints);

%factorization
[U1,D1,V1] = svd(matPositionRelative);

%Impose rank=3 constraint
D1(4:end,4:end) = 0;
U2 = U1(:,1:3);
D2 = D1(1:3,1:3);
V2 = V1(:,1:3);

R1 = U2*sqrt(D2);
S1 = sqrt(D2)*V2';

%Construct AX = b
A = zeros(3*numImages, 6);%Number of unknowns = 6;because of symmetry
b = zeros(3*numImages, 1);
for i = 1:numImages
    iter_A = 3*(i - 1);
    iter_R1 = 2*(i - 1);
    row1 = R1(iter_R1 + 1,:);%1x3
    row2 = R1(iter_R1 + 2,:);%1x3
    row11 = row1'*row1;%3x3
    row22 = row2'*row2;%3x3
    row12 = row1'*row2;%3x3
    row11temp = [row11(1) row11(2)+row11(4) row11(3)+row11(7) row11(5)
row11(6)+row11(8) row11(9)]';
    row22temp = [row22(1) row22(2)+row22(4) row22(3)+row22(7) row22(5)
row22(6)+row22(8) row22(9)]';
    row12temp = [row12(1) row12(2)+row12(4) row12(3)+row12(7) row12(5)
row12(6)+row12(8) row12(9)]';
    %A(iter_A + 1:iter_A + 3,:) = [row11(:) row22(:) row12(:)]';
    A(iter_A + 1:iter_A + 3,:) = [row11temp row22temp row12temp]';
    b(iter_A + 1:iter_A + 3,:) = [1; 1; 0];
end
% X = A\b;
X = (inv(A'*A))*A'*b;
matX = [X(1) X(2) X(3);...
        X(2) X(4) X(5);...
        X(3) X(5) X(6);];
% Q1 = reshape(X,[3 3]);
[Q2,PD_flag] = chol(matX,'lower');
if(PD_flag)%Check for positive definiteness
    [vec,val] = eig(matX);
    val(val<0) = eps;
    Q3 = vec*val*vec';
    points3D = inv(chol(Q3))*S1;
else
    points3D = inv(Q2)*S1;
end
end

%Function for performing non-rigid factorization for 3D reconstruction
%Inputs
    %matPosition - matrix containing the 2D points of all feature points in
all the frames
%Outputs
    %points3D - matrix containing the estimated 3D points of all feature
points in all the frames
function points3D = perform3DReconstructionNonRigid(matPosition)
[numImagesX2,numPoints] = size(matPosition);
numImages = uint32(numImagesX2/2);

%Parameter
numBasis = 16;%16
if((3*numBasis) > min(numImagesX2,numPoints))
    numBasis = floor(min(numImagesX2,numPoints)/3);
end

%centroid subtraction
matCentroid = sum(matPosition, 2)/numPoints;
matPositionRelative = matPosition - repmat(matCentroid,1,numPoints);
points3D = zeros(numImages*3,numPoints);

%factorization
[U1,D1,V1] = svd(matPositionRelative);

%Impose rank=3*k constraint
% D1(4:end,4:end) = 0;
U2 = U1(:,1:3*numBasis);
D2 = D1(1:3*numBasis,1:3*numBasis);
V2 = V1(:,1:3*numBasis);

Q = U2*sqrt(D2);
B = sqrt(D2)*V2';

L = zeros(numImages, numBasis);
R1 = zeros(numImagesX2, 3);

for i = 1:numImages%(size(Q,1)/2)
    q = Q(2*i - 1:2*i,:);
    q_bar = reorder(q, numBasis);
    [U,D,V] = svd(q_bar);
    %Impose rank=1 constraint
    L(i,:) = U(:,1)*sqrt(D(1));
    R1(2*i - 1:2*i,:) = reshape(sqrt(D(1))*V(:,1)',3,2)';
end
```

```
%Construct AX = b
A = zeros(3*numImages, 6);%Number of unknowns = 6;because of symmetry
b = zeros(3*numImages, 1);
for i = 1:numImages
    iter_A = 3*(i - 1);
    iter_R1 = 2*(i - 1);
    row1 = R1(iter_R1 + 1,:);%1x3
    row2 = R1(iter_R1 + 2,:);%1x3
    row11 = row1'*row1;%3x3
    row22 = row2'*row2;%3x3
    row12 = row1'*row2;%3x3
    row11temp = [row11(1) row11(2)+row11(4) row11(3)+row11(7) row11(5)
row11(6)+row11(8) row11(9)]';
    row22temp = [row22(1) row22(2)+row22(4) row22(3)+row22(7) row22(5)
row22(6)+row22(8) row22(9)]';
    row12temp = [row12(1) row12(2)+row12(4) row12(3)+row12(7) row12(5)
row12(6)+row12(8) row12(9)]';
    %A(iter_A + 1:iter_A + 3,:) = [row11(:) row22(:) row12(:)]';
    A(iter_A + 1:iter_A + 3,:) = [row11temp row22temp row12temp]';
    b(iter_A + 1:iter_A + 3,:) = [1; 1; 0];
end

X = (inv(A'*A))*A'*b;
matX = [X(1) X(2) X(3);...
        X(2) X(4) X(5);...
        X(3) X(5) X(6);];
[Q2,PD_flag] = chol(matX,'lower');
if(PD_flag)%Check for positive definiteness
    [vec,val] = eig(matX);
    val(val<0) = eps;
    Q3 = vec*val*vec';
    for i = 1:numBasis
        basis(3*i-2:3*i,:) = inv(chol(Q3))*B(3*i-2:3*i,:);
    end
else
    for i = 1:numBasis
        basis(3*i-2:3*i,:) = inv(Q2)*B(3*i-2:3*i,:);
    end
end
%Get 3D points estimation
for i = 1:numImages
    for j = 1:numBasis
        points3D(3*i-2:3*i,:) = points3D(3*i-2:3*i,:) + L(i,j)*basis(3*j-
2:3*j,:);
    end
end
end

function b = reorder(a, k)
b = zeros(k,6);
for i = 1:k
    b(i,:) = [a(1,3*i-2:3*i) a(2,3*i-2:3*i)];
end
end

%Function for visualizing the 3D point set
%Inputs
    %pts3d_Rigid - 3D points estimated from rigid factorization
    %pts3d_NonRigid - 3D points estimated from non-rigid factorization
    %id - id for video data being used
```

```
%Outputs
    %
function visualizeData(pts3d_Rigid, pts3d_NonRigid, id)
figure,
if(id == 2)
    subplot(1,2,1),visualize_hotel(pts3d_Rigid);title('Rigid 3D
Reconstruction');view(-120,90);
    subplot(1,2,2),visualize_hotel(pts3d_NonRigid(1:3,:));title('Non-Rigid
3D Reconstruction');view(-120,90);
elseif(id == 3)
    subplot(1,2,1),visualize_franck(pts3d_Rigid);title('Rigid 3D
Reconstruction');view(-91,-90);
    subplot(1,2,2),visualize_franck(pts3d_NonRigid(1:3,:));title('Non-Rigid
3D Reconstruction');view(91,-90);
elseif(id == 4)
    subplot(1,2,1),visualize_boxofjoe(pts3d_Rigid);title('Rigid 3D
Reconstruction');view(-80,-90);
    subplot(1,2,2),visualize_boxofjoe(pts3d_NonRigid(1:3,:));title('Non-
Rigid 3D Reconstruction');view(100,-90);
elseif(id == 5)
    subplot(1,2,1),visualize_paper(pts3d_Rigid);title('Rigid 3D
Reconstruction');view(130,-90);
    subplot(1,2,2),visualize_paper(pts3d_NonRigid(1:3,:));title('Non-Rigid
3D Reconstruction');view(70,90);
elseif(id == 6)
    subplot(1,2,1),visualize_gyan(pts3d_Rigid);title('Rigid 3D
Reconstruction');view(140,-90);
    subplot(1,2,2),visualize_gyan(pts3d_NonRigid(1:3,:));title('Non-Rigid
3D Reconstruction');view(140,-90);
end

end

function visualize_hotel(pts)
scatter3(pts(1,:)',pts(2,:)',pts(3,:)','filled');
hold on;
ptsTemp = pts(:,1:2);
ptsTemp(:,3) = pts(:,4);
ptsTemp(:,4) = pts(:,3);
ptsTemp(:,5) = pts(:,1);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','r','LineWidth',2);
ptsTemp = pts(:,5:6);
ptsTemp(:,3) = pts(:,8);
ptsTemp(:,4) = pts(:,7);
ptsTemp(:,5) = pts(:,5);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','r','LineWidth',2);
ptsTemp = pts(:,9:10);
ptsTemp(:,3) = pts(:,12);
ptsTemp(:,4) = pts(:,11);
ptsTemp(:,5) = pts(:,9);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','r','LineWidth',2);
ptsTemp = pts(:,1);
ptsTemp(:,2) = pts(:,5);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','r','LineWidth',2);
ptsTemp = pts(:,2);
ptsTemp(:,2) = pts(:,6);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','r','LineWidth',2);
hold off;
end

function visualize_boxofjoe(pts)
```

```
scatter3(pts(1,:)',pts(2,:)',pts(3,:)','filled');
hold on;
ptsTemp = pts(:,1:4);
ptsTemp(:,5) = pts(:,1);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','r','LineWidth',2);
ptsTemp = pts(:,1);
ptsTemp(:,2:4) = pts(:,9:11);
ptsTemp(:,5) = pts(:,3);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','r','LineWidth',2);
ptsTemp = pts(:,10);
ptsTemp(:,2) = pts(:,2);
ptsTemp(:,3:4) = pts(:,12:13);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','r','LineWidth',2);
ptsTemp = pts(:,7);
ptsTemp(:,2:3) = pts(:,5:6);
ptsTemp(:,4) = pts(:,8);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','r','LineWidth',2);
ptsTemp = pts(:,1);
ptsTemp(:,2) = pts(:,7);
ptsTemp(:,3) = pts(:,4);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','r','LineWidth',2);
ptsTemp = pts(:,2);
ptsTemp(:,2) = pts(:,8);
ptsTemp(:,3) = pts(:,3);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','r','LineWidth',2);
hold off;
end

function visualize_franck(pts)
scatter3(pts(1,:)',pts(2,:)',pts(3,:)','filled');
hold on;
ptsTemp = pts(:,2:14);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','r','LineWidth',2);%
Face outline
ptsTemp = pts(:,15:21);
ptsTemp(:,8) = pts(:,15);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','g','LineWidth',2);%
person's left eyebrow
ptsTemp = pts(:,1);
ptsTemp(:,2:7) = pts(:,22:27);
ptsTemp(:,8) = pts(:,1);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','y','LineWidth',2);%
person's right eyebrow
ptsTemp = pts(:,33:36);
ptsTemp(:,5) = pts(:,33);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','g','LineWidth',2);%
person's left eye
ptsTemp = pts(:,28:31);
ptsTemp(:,5) = pts(:,28);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','y','LineWidth',2);%
person's right eye
ptsTemp = pts(:,38:46);
ptsTemp(:,10) = pts(:,38);
ptsTemp(:,11) = pts(:,68);
ptsTemp(:,12) = pts(:,46);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','r','LineWidth',2);%
person's nose
ptsTemp = pts(:,49:60);
ptsTemp(:,13) = pts(:,49);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','r','LineWidth',2);%
person's mouth
```

```
hold off;
end

function visualize_gyan(pts)
scatter3(pts(1,:)',pts(2,:)',pts(3,:)','filled');
hold on;
ptsTemp = pts(:,1:3);
ptsTemp(:,4) = pts(:,6);
ptsTemp(:,5) = pts(:,5);
ptsTemp(:,6) = pts(:,2);
ptsTemp(:,7) = pts(:,4);
ptsTemp(:,8) = pts(:,6);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','y','LineWidth',2);
ptsTemp = pts(:,7:9);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','g','LineWidth',2);
ptsTemp = pts(:,10:11);
ptsTemp(:,3) = pts(:,13);
ptsTemp(:,4) = pts(:,12);
ptsTemp(:,5) = pts(:,10);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','r','LineWidth',2);
ptsTemp = pts(:,2);
ptsTemp(:,2) = pts(:,10);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','b','LineWidth',2);
ptsTemp = pts(:,9);
ptsTemp(:,2) = pts(:,13);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','b','LineWidth',2);
hold off;
end

function visualize_paper(pts)
scatter3(pts(1,:)',pts(2,:)',pts(3,:)','filled');
hold on;
ptsTemp = pts(:,1:2);
ptsTemp(:,3) = pts(:,4);
ptsTemp(:,4) = pts(:,3);
ptsTemp(:,5) = pts(:,1);
ptsTemp(:,6:8) = pts(:,5:7);
ptsTemp(:,9) = pts(:,4);
ptsTemp(:,10) = pts(:,3);
ptsTemp(:,11) = pts(:,6);
line(ptsTemp(1,:)',ptsTemp(2,:)',ptsTemp(3,:)','color','g','LineWidth',2);
hold off;
end
```