Structure from Motion

*Submitted in partial fulfillment of the requirements for the course of EECE5639.

Yash Mewada
dept. of Electrical and Computer Engineering
Northeastern University, Boston, MA
Email: mewada.y@northeastern.edu

***Abstract*—In recent years, RGB-D cameras have emerged as popular sensors in the field of computer vision. This project details a novel approach for Structure from Motion based on optical flow and factorization for the reconstruction of surfaces with few textures. An original image search and grouping strategy allows to the reconstruction of each 3D scene point using a large set of 2D homologous points extracted from a reference image and its superimposed images acquired from different viewpoints.**

***Index Terms*—Structure from Motion (SFM), 3D reconstruction, Optical Flow (OF), Triangulation, and Stereo Vision.**

I. Introduction

AS we watch the video output from a camera moving in a three-dimensional (3D) scene, our minds naturally obtain a feeling for or estimate of the motion of the camera as well as an idea of the geometry of the scene. Humans are well adapted to recovering geometry and motion from image sequences. It has been a computer vision goal to perform a similar task: from an image sequence taken by a camera undergoing unknown motion, extract the 3D shape of the scene as well as the camera motion. This is called the *structure from motion* problem.
There are many ways to solve this problem with camera intrinsic or with the fundamental and essential matrix decomposition if camera intrinsic is not available. This project focuses on one of the many ways of obtaining the structure and motion of the camera from the given sequences of the images called *Factorization*.

II. Methodology

A. Feature detection

The main pillar of this project depends on feature detection. The better and more useful features detected, the better a 3D output is generated. There are many feature detection (corner detection) algorithms to address this as below.

1) Harris Corner detection
2) SIFT (Scale Invariant Feature Transform) feature extraction.
3) Shi-Tomasi Corner Detector (Good features to track)

The latter one was used in this project. Basically, SIFT is a feature descriptor that is robust to various image transformations, while Harris and Shi-Tomasi are corner detectors that are fast and effective in detecting corners. The only difference between the Harris and Shi-Tomasi is the way they compute the corner matrix R.

$$R = \det(M) - k(trace(M))^2 - Harris \tag{1}$$

Where,

$$\det(M) = \lambda_1 \times \lambda_2$$

$$R = \min(\lambda_1, \lambda_2) - Shi - Tomasi \tag{2}$$

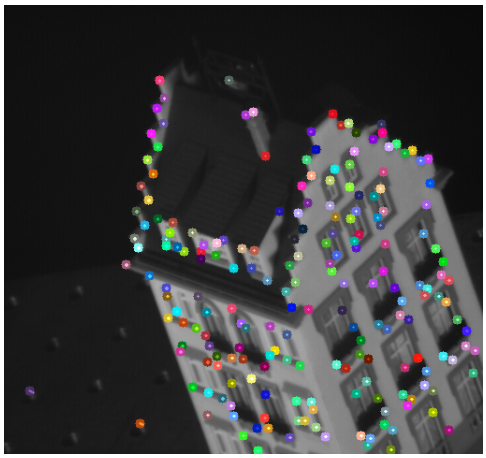

Fig. 1. Features from the first image

B. Feature Tracking using optical flow

After the feature from the first image was obtained, we have to track these features in the second frame using the optical flow of these features. *Optical Flow* is the task of per-pixel motion estimation between two consecutive frames. Basically, you calculate the shift of pixel as an object displacement.
Let's assume you have a pixel with intensity function as $I(x, y, t)$ where x,y is the pixel coordinate and t is the time frame. Now after a certain amount of time, the pixel intensity shifted to a new location, then the new location of this pixel intensity will be equal to...

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t) \tag{3}$$

Using Taylor series expansion we can rewrite the equation as.

$$I(x, y, t) - I(x + \Delta x, y + \Delta y, t + \Delta t) = 0$$

$$I_x^{'} u + I_y^{'} v = -I_t^{'}$$

Where $u = \frac{dx}{dt}$ and $v = \frac{dy}{dt}$ Hence the pixel motion between two consecutive frames can be written as.

$$I_1 - I_2 \approx I_x^{'} u + I_y^{'} v + I_t^{'} \tag{4}$$

Here the implementation of Sparse Optical flow with the Lucas Kanade method was used instead of the dense optical flow. The main difference between sparse and dense optical flow lies in the number of points at which motion vectors are estimated. Sparse optical flow only estimates the motion vectors at a few selected points in the image or video frame, while dense optical flow estimates the motion vectors at every pixel in the image or video frame. Let's say we same motion vector from (4).

$$\begin{cases} I'_x(p_1)u + I'_y(p_1)v = -I'_t(p_1) \\ I'_x(p_2)u + I'_y(p_2)v = -I'_t(p_2) \\ \dots \\ I'_x(p_n)u + I'_y(p_n)v = -I'_t(p_n) \end{cases} \tag{5}$$

In matrix form, we can rewrite the equation as...

$$A = \begin{bmatrix} I'_x(p_1) & I'_y(p_1) \\ I'_x(p_2) & I'_y(p_2) \\ \vdots & \vdots \\ I'_x(p_n) & I'_y(p_n) \end{bmatrix}, \ \gamma = \begin{bmatrix} dx \\ dy \end{bmatrix}, b = \begin{bmatrix} -I'_t(p_1)dt \\ -I'_t(p_2)dt \\ \vdots \\ -I'_t(p_n)dt \end{bmatrix} \tag{6}$$

As a result, we have a matrix equation: A γ = b. Using the Least Squares we can compute the answer vector γ :

$$\begin{aligned} A^T A\gamma = A^T b \quad &\Rightarrow \quad \underbrace{(A^T A)^{-1}(A^T A)}_{Identity\ matrix}\gamma = (A^T A)^{-1}A^T b \\ &\Rightarrow \quad \gamma = (A^T A)^{-1}A^T b \end{aligned} \tag{7}$$

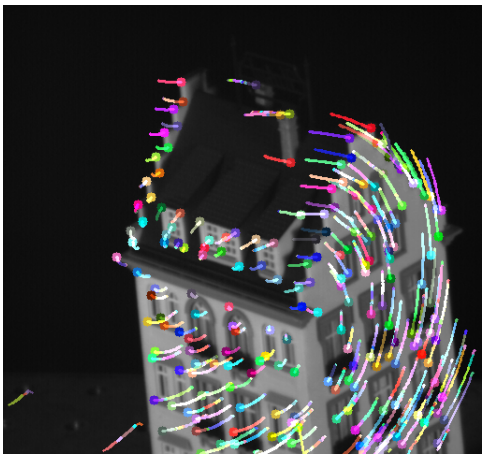

Fig. 2. Optical flow for 37^{th} frame

With the help of this optical flow, the same corner points in the first image were tracked across the sequence of 99 image frames whose size will be $2F \times N$; where F = number of frames and N = number of feature points. The W matrix was scaled by subtracting each value of this matrix from the mean of this matrix.

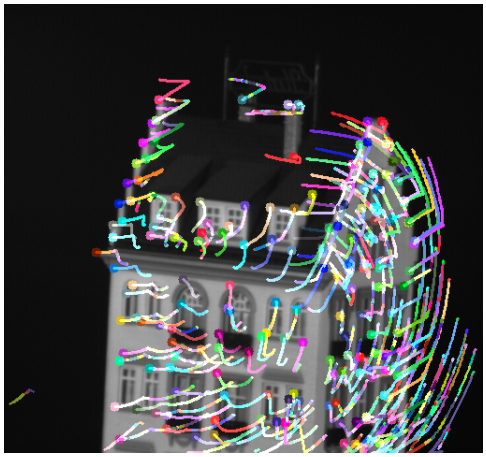

Fig. 3. Optical flow for 99^{th} frame

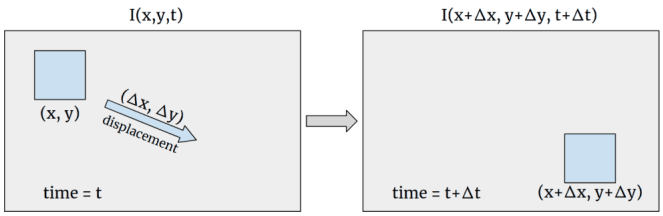

Fig. 4. Optical flow of the pixel

C. Factorization

The goal of the factorization part of the SFM is to combine point correspondence information from multiple points over multiple frames to solve the scene structure and camera motion (structure from motion). After the W matrix is obtained we can perform the singular value decomposition of this matrix to obtain the motion matrix (M) and structure matrix (S).

Algorithm 1 Factorization Algorithm

0: Form the centered measurement matrix W from N points tracked over F frames.
0: Compute SVD of $W = UDV^T$
- U is $2F \times 2F$
- D is $2F \times N$
- V^T is $N \times N$

0: Take the largest 3 eigenvalues, and form
- $D' = \text{diag}(\lambda_1, \lambda_2, \lambda_3)$, where $\lambda_1 \geq \lambda_2 \geq \lambda_3$ are the largest eigenvalues of W
- $U' = [u_1, u_2, u_3]$, where u_1, u_2, u_3 are the corresponding column vectors of U
- $V'^T = [v_1^T, v_2^T, v_3^T]$, where v_1^T, v_2^T, v_3^T are the corresponding row vectors of V^T

0: Define $M = U'D'^{1/2}$ and $S = D'^{1/2}V'^T$
0: Solve for Q that makes appropriate rows of M orthogonal
0: Final solution is $M^* = MQ$ and $S^* = Q^{-1}S$ =0

$$W_{2F\times N} = M_{2F\times 3}S_{3\times N} \tag{8}$$

The rank of the W matrix is 3 but in practice, it will never be equal to 3, hence we can brute-force the rank to 3. In practice, we can perform the singular value decomposition of the W matrix to form the M and S matrices.

$$W_{2F\times N} = U_{2F\times 2F} \quad D_{2F\times N} \quad V^T_{N\times N} \tag{9}$$

The rank of a matrix is equal to the number of nonzero eigenvalues. d11, d22, and d33 are only nonzero eigenvalues hence setting the rest of the values to 0. Reshaping the U, V^T, and D matrices to take only the first three columns of the entire matrix.

$$W_{2F\times N} = U_{2F\times 3} \quad D_{3\times 3} \quad V^T_{3\times N} \tag{10}$$

$$M = U \quad \sqrt{D} \tag{11}$$

$$S = \sqrt{D} \quad V^T \tag{12}$$

But still, the above decomposition has some annoying details because this is not a unique decomposition. i^T, j^T pairs (rows of M) are not necessarily orthogonal.

D. Solving the annoying details of Q matrix

From the above explanation, it is clear that we need a 3×3 matrix to have a unique decomposition of the W matrix. In structure from motion (SfM) using the factorization method, the Q matrix is a rotation matrix that is used to make the appropriate rows of the matrix M orthogonal.
The matrix Q is needed because the matrix M, which represents the motion of the camera, may not be orthogonal after the SVD decomposition of the measurement matrix W. This is because the SVD of W is not unique and there can be multiple ways to decompose W into M and S.
By finding the orthogonal matrix Q that makes the rows of M orthogonal, we ensure that M is a valid rotation matrix and that the camera motion is physically meaningful. Without Q, we may end up with a matrix M that is not a valid rotation matrix and does not correspond to a physically realizable camera motion.

$$\begin{aligned} \hat{i}_i^T Q Q^T \hat{i}_i &= 1 \\ \hat{j}_i^T Q Q^T \hat{j}_i &= 1 \\ \hat{i}_i^T Q Q^T \hat{j}_i &= 0 \end{aligned} \tag{13}$$

Furthermore, by transforming S using the inverse of Q, we ensure that the 3D structure of the scene is consistent with the camera motion. This is because the SVD of W only gives us a factorization of the measurements, and we need to use additional constraints, such as orthogonality, to recover the true structure and motion of the scene.
This is because an orthogonal matrix has the property that its transpose is its inverse, which means that multiplying M by Q does not change the norm of the rows of M. By making the rows of M orthogonal, we remove any ambiguity in the camera motion and ensure that it is physically meaningful.

$$Q = \begin{bmatrix} 0.06903336 & 0.0 & 0.0 \\ -0.00187322 & 0.05956381 & 0.0 \\ 0.02098447 & 0.01206917 & 0.0849465 \end{bmatrix} \tag{14}$$

Making M orthogonal does not necessarily make it unique, but it does ensure that it is a valid rotation matrix and that it corresponds to a physically meaningful camera motion .
Here the decomposition was performed using the **Cholesky** decomposition method and from the above Q matrix, we can see that the matrix is a symmetric matrix.

E. Output

For output python's 3D library, Open3D was used here.

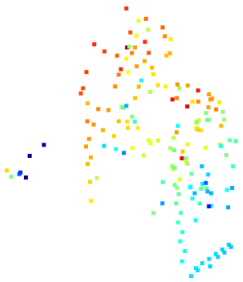

Fig. 5. PCD output

III. Conclusion

Based on the above discussion, performing structure from motion based on the factorization method was successfully performed. Although there is somewhat a resemblance between the images and the 3D point cloud generated a more detailed SFM algorithm can be performed when we have the camera intrinsics and we can obtain the Essential and Fundamental matrices based on those and obtain a dense and better SFM output.
But this project addressed the problem that even though we don't have the camera information we can perform the SFM and 3D reconstruction.

IV. Appendix

The code for our project can be found here : GitHub

```
import cv2
import glob
import os
import numpy as np
import open3d as o3d

"""
Author: Yash Mewada {mewada.y@northeastern.edu}
```

```
Created: April 21st 2023
"""
class SFM:
    def __init__(self,pathTofiles):
        """
        Initialise the class and parse the locations of all the images.
        """
        self.path = pathTofiles
        input = pathToFiles + str('/hotel/')
        os.chdir(input)
        self.images = []
        for file in list(glob.glob("*.png")):
            self.images.append(os.getcwd() + str('/') + file)
        self.images = np.sort(self.images)
        self.F_list = []
        self.E_list = []
        self.inliers1 = []
        self.inliers2 = []
        self.of_params = dict(winSize = (15,15),
                              maxLevel = 2,
                              criteria = (cv2.TermCriteria_EPS | cv2.TERM_CRITERIA_COUNT, 10, 0.0001))
        self.feature_params = dict( maxCorners = 170,
                           qualityLevel = 0.0001,
                           minDistance = 7,
                           blockSize = 7)
        # print(self.images)

    def factorise(self):
        """
        Factorize the correspondence matrix obtained from optical flow-based feature tracking.
        """

        os.chdir(self.path + str('/sift_output/'))
        index = 0

        p0 = cv2.imread(self.images[0])
        prev_gray = cv2.cvtColor(p0,cv2.COLOR_BGR2GRAY)
        prev_pts = cv2.goodFeaturesToTrack(prev_gray, mask = None, **self.feature_params)
        prev_pts = np.squeeze(prev_pts)
        color = np.random.randint(0,255,(500,3))

        mask = np.zeros_like(p0)
        W = np.zeros((len(self.images)*2,prev_pts.shape[0]))
        for i in range(len(self.images)-1):
            print(self.images[i+1])
            img = cv2.imread(self.images[i+1])
            curr_gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
            curr_pts,st,err = cv2.calcOpticalFlowPyrLK(prev_gray,curr_gray,prev_p
            st = st.reshape(curr_pts.shape[0], )

            if curr_pts is not None:
                curr_pts = curr_pts[st==1]
                prev_pts = prev_pts[st==1]

            for k in range(curr_pts.shape[0]):
                W[2*i:2*i+2, k] = curr_pts[k]

            for i, (new, old) in enumerate(zip(curr_pts, prev_pts)):
                a, b = new.ravel()
                c, d = old.ravel()
                mask = cv2.line(mask, (int(a), int(b)), (int(c), int(d)), color[i].tolist(), 2)
                img = cv2.circle(img, (int(a), int(b)), 5, color[i].tolist(), -1)
            img_new = cv2.add(img, mask)
            prev_gray = curr_gray
            prev_pts = curr_pts.copy()

            # cv2.imwrite(str(index) + ".png",img_new)
            index += 1
            cv2.imshow("gray",img_new)
            cv2.waitKey(0)
        # W = np.array(correspondences)
        W_mean = np.mean(W, axis=1, keepdims=True)
        W_reg = W - W_mean

        # Compute SVD of W
        U, D, VT = np.linalg.svd(W_reg,full_matrices=False)
        # D[3:] = 0  # set singular values 5 to 2m to 0
        # Keep the first 3 eigenvalues/,full_matrices=Falseeigenvectors
        U_3 = U[:, :3]
        D_3 = np.diag(np.sqrt(D[:3]))
        V_T_3 = VT[:3, :]

        # Compute M and S
        M = np.dot(U_3, D_3)
        S = np.dot(D_3, V_T_3)

        R_i = M[:100, :]
        R_j = M[100:200, :]
        G1, c1 = self.mat_G_maker(np.zeros((202, 6)), np.zeros((202, 1)), R_i, R_i, same=1)
        G2, c2 = self.mat_G_maker(np.zeros((202, 6)), np.zeros((202, 1)), R_j, R_j, same=1)
        G3, c3 = self.mat_G_maker(np.zeros((202, 6)), np.zeros((202, 1)), R_i, R_j, same=0)
        G = np.vstack((G1, G2, G3))
        c = np.vstack((c1, c2, c3))
        c = c.squeeze()
        GTG_inv = np.linalg.pinv(np.dot(G.T, G))
        l = np.dot(np.dot(GTG_inv, G.T), c)

        L = np.zeros((3, 3))

        L[0, 0] = l[0]
        L[1, 1] = l[3]
        L[2, 2] = l[5]

        L[0, 1] = l[1]
        L[1, 0] = L[0, 1]

        L[0, 2] = l[2]
        L[2, 0] = L[0, 2]

        L[1, 2] = l[4]
        L[2, 1] = L[1, 2]
        d_, V_ = np.linalg.eig(L)
        D = np.diag(d_)
        D[D < 0] = 0.00001

        L = np.dot(V_, np.dot(D, V_.T))

        Q = np.linalg.cholesky(L)
        R_true = np.dot(M, Q)
        S_true = np.dot(np.linalg.inv(Q), S)
        return S_true.T


    def g(self,v1, v2):
        [a, b, c] = v1
```

```
        [x, y, z] = v2

        res = [a*x, 2*a*y, 2*a*z, b*y, 2*b*z, c*z]
        return res


    def mat_G_maker(self,G, c, R1, R2, same=0):
        c.fill(same)
        n = R1.shape[0]
        for i in range(n):
            r1 = R1[i, :]
            r2 = R2[i, :]
            res = self.g(r1, r2)
            G[i, :] = res

        return G, c

    def write_ascii_Ply(self,points,filename):
        """
        Write an ASCII output PLY file with the 3D
    x, y, z coordinates of the points separated by
    commas.
        """
        # print(points)
        pcd = o3d.geometry.PointCloud()
        pcd.points =
    o3d.utility.Vector3dVector(points.astype('float64'))
        o3d.io.write_point_cloud(filename,
    pcd,write_ascii=True)

    def visualisePly(self,filename):
        pcd = o3d.io.read_point_cloud(filename)
        o3d.visualization.draw_geometries([pcd])




if __name__ == "__main__":
    pathToFiles =
    '/home/yash/Documents/Computer_VIsion/CV_OptionalProject/hotel'
    Plyfile =
    '/home/yash/Documents/Computer_VIsion/CV_OptionalProject/output.ply'

    p4 = SFM(pathToFiles)

    pts_3d = p4.factorise()
    p4.write_ascii_Ply(pts_3d,Plyfile)
    p4.visualisePly(Plyfile)
```

Listing 1. SFM

REFERENCES

[1] A Sequential Factorization Method for Recovering Shape and Motion from Image Streams

[2] Shape and Motion from Image Streams under Orthography: a Factorization Method

[3] Robert Collins, UPenn lecture slides Structure from Motion

[4] Structure From motion using Factorization