

# 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  $R$ .

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

Where,

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

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

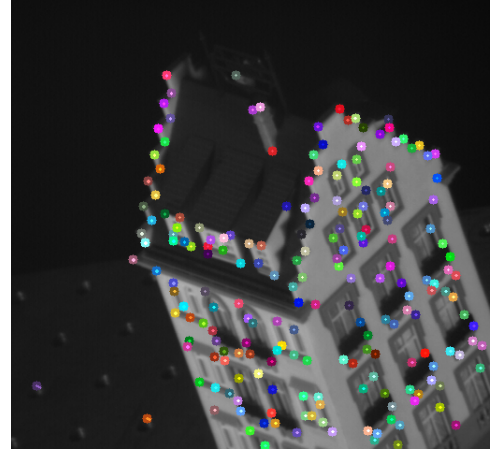


Fig. 1. Features from the first image

### 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...

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t) \quad (3)$$

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.

$$I_1 - I_2 \approx I'_x u + I'_y v + I'_t \quad (4)$$

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).

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

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

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

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

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

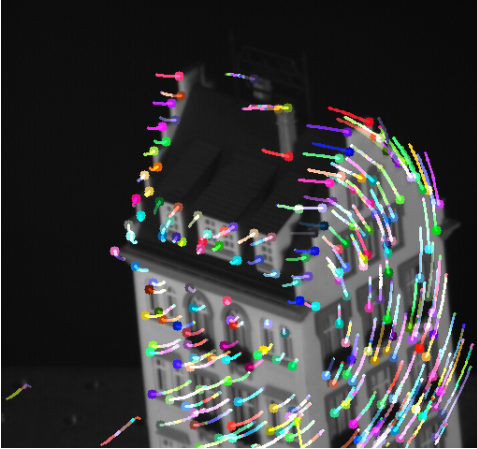


Fig. 2. Optical flow for 37<sup>th</sup> frame

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.

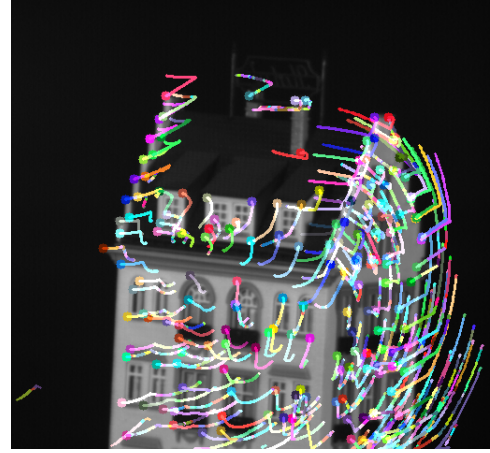


Fig. 3. Optical flow for 99<sup>th</sup> frame

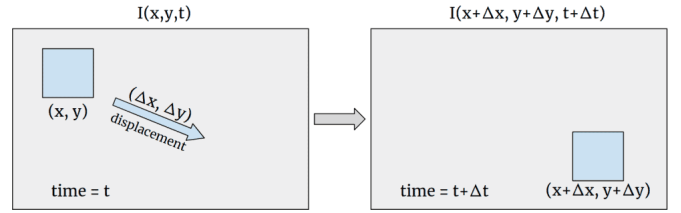


Fig. 4. Optical flow of the pixel

### 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  $S^* = Q^{-1} S = 0$

$$W_{2F \times N} = M_{2F \times 3} S_{3 \times N} \quad (8)$$

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.

$$W_{2F \times N} = U_{2F \times 2F} D_{2F \times N} V_{N \times N}^T \quad (9)$$

The rank of a matrix is equal to the number of nonzero eigenvalues.  $d_{11}$ ,  $d_{22}$ , and  $d_{33}$  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.

$$W_{2F \times N} = U_{2F \times 3} D_{3 \times 3} V_{3 \times N}^T \quad (10)$$

$$M = U \sqrt{D} \quad (11)$$

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

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  $3 \times 3$  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.

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

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.

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

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.

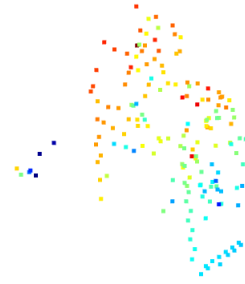


Fig. 5. PCD output

### 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](#)

```
1 import cv2
2 import glob
3 import os
4 import numpy as np
5 import open3d as o3d
6
7 """
8 Author: Yash Mewada {mewada.y@northeastern.edu}
```

```

9 Created: April 21st 2023
10 """
11 class SFM:
12     def __init__(self, pathToFiles):
13         """
14         Initialise the class and parse the
15         locations of all the images.
16         """
17         self.path = pathToFiles
18         input = pathToFiles + str('/hotel/')
19         os.chdir(input)
20         self.images = []
21         for file in list(glob.glob("*.png")):
22             self.images.append(os.getcwd() +
23                               str('/') + file)
24         self.images = np.sort(self.images)
25         self.F_list = []
26         self.E_list = []
27         self.inliers1 = []
28         self.inliers2 = []
29         self.of_params = dict(winSize = (15,15),
30                               maxLevel = 2,
31                               criteria =
32                               (cv2.TermCriteria_EPS |
33                               cv2.TERM_CRITERIA_COUNT, 10, 0.0001))
34         self.feature_params = dict( maxCorners =
35                                     170,
36                                     qualityLevel = 0.0001,
37                                     minDistance = 7,
38                                     blockSize = 7)
39         # print(self.images)
40
41     def factorise(self):
42         """
43         Factorize the correspondence matrix
44         obtained from optical flow-based feature
45         tracking.
46         """
47         os.chdir(self.path + str('/sift_output/'))
48         index = 0
49
50         p0 = cv2.imread(self.images[0])
51         prev_gray =
52         cv2.cvtColor(p0, cv2.COLOR_BGR2GRAY)
53         prev_pts =
54         cv2.goodFeaturesToTrack(prev_gray, mask = None,
55                                **self.feature_params)
56         prev_pts = np.squeeze(prev_pts)
57         color = np.random.randint(0, 255, (500, 3))
58
59         mask = np.zeros_like(p0)
60         W =
61         np.zeros((len(self.images)*2, prev_pts.shape[0]))
62         for i in range(len(self.images)-1):
63             print(self.images[i+1])
64             img = cv2.imread(self.images[i+1])
65             curr_gray =
66             cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
67             curr_pts, st, err =
68             cv2.calcOpticalFlowPyrLK(prev_gray, curr_gray, prev_pts,
69                                     st = st.reshape(curr_pts.shape[0], ))
70
71             if curr_pts is not None:
72                 curr_pts = curr_pts[st==1]
73                 prev_pts = prev_pts[st==1]
74
75             for k in range(curr_pts.shape[0]):
76                 W[2*i:2*i+2, k] = curr_pts[k]
77
78             for i, (new, old) in
79             enumerate(zip(curr_pts, prev_pts)):
80                 a, b = new.ravel()
81                 c, d = old.ravel()

```

```

69         mask = cv2.line(mask, (int(a),
70                                int(b)), (int(c), int(d)), color[i].tolist(), 2)
71         img = cv2.circle(img, (int(a),
72                                int(b)), 5, color[i].tolist(), -1)
73         img_new = cv2.add(img, mask)
74         prev_gray = curr_gray
75         prev_pts = curr_pts.copy()
76
77         # cv2.imwrite(str(index) +
78         ".png", img_new)
79         index += 1
80         cv2.imshow("gray", img_new)
81         cv2.waitKey(0)
82         # W = np.array(correspondences)
83         W_mean = np.mean(W, axis=1, keepdims=True)
84         W_reg = W - W_mean
85
86         # Compute SVD of W
87         U, D, VT =
88         np.linalg.svd(W_reg, full_matrices=False)
89         # D[3:] = 0 # set singular values 5 to 2m
90         to 0
91         # Keep the first 3
92         eigenvalues/, full_matrices=False eigenvectors
93         U_3 = U[:, :3]
94         D_3 = np.diag(np.sqrt(D[:3]))
95         V_T_3 = VT[:3, :]
96
97         # Compute M and S
98         M = np.dot(U_3, D_3)
99         S = np.dot(D_3, V_T_3)
100
101         R_i = M[:100, :]
102         R_j = M[100:200, :]
103         G1, c1 = self.mat_G_maker(np.zeros((202,
104         6)), np.zeros((202, 1)), R_i, R_i, same=1)
105         G2, c2 = self.mat_G_maker(np.zeros((202,
106         6)), np.zeros((202, 1)), R_j, R_j, same=1)
107         G3, c3 = self.mat_G_maker(np.zeros((202,
108         6)), np.zeros((202, 1)), R_i, R_j, same=0)
109         G = np.vstack((G1, G2, G3))
110         c = np.vstack((c1, c2, c3))
111         c = c.squeeze()
112         GTG_inv = np.linalg.pinv(np.dot(G.T, G))
113         l = np.dot(np.dot(GTG_inv, G.T), c)
114
115         L = np.zeros((3, 3))
116
117         L[0, 0] = l[0]
118         L[1, 1] = l[3]
119         L[2, 2] = l[5]
120
121         L[0, 1] = l[1]
122         L[1, 0] = L[0, 1]
123
124         L[0, 2] = l[2]
125         L[2, 0] = L[0, 2]
126
127         L[1, 2] = l[4]
128         L[2, 1] = L[1, 2]
129         d_, V_ = np.linalg.eig(L)
130         D = np.diag(d_)
131         D[D < 0] = 0.00001
132
133         L = np.dot(V_, np.dot(D, V_.T))
134
135         Q = np.linalg.cholesky(L)
136         R_true = np.dot(M, Q)
137         S_true = np.dot(np.linalg.inv(Q), S)
138         return S_true.T
139
140     def g(self, v1, v2):
141         [a, b, c] = v1

```

```

134     [x, y, z] = v2
135
136     res = [a*x, 2*a*y, 2*a*z, b*y, 2*b*z, c*z]
137     return res
138
139
140     def mat_G_maker(self,G, c, R1, R2, same=0):
141         c.fill(same)
142         n = R1.shape[0]
143         for i in range(n):
144             r1 = R1[i, :]
145             r2 = R2[i, :]
146             res = self.g(r1, r2)
147             G[i, :] = res
148
149         return G, c
150
151     def write_ascii_Ply(self,points,filename):
152         """
153         Write an ASCII output PLY file with the 3D
154         x, y, z coordinates of the points separated by
155         commas.
156         """
157         # print(points)
158         pcd = o3d.geometry.PointCloud()
159         pcd.points =
160         o3d.utility.Vector3dVector(points.astype('float64'))
161         o3d.io.write_point_cloud(filename,
162         pcd,write_ascii=True)
163
164
165     def visualisePly(self,filename):
166         pcd = o3d.io.read_point_cloud(filename)
167         o3d.visualization.draw_geometries([pcd])
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

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](#)