RBE 549: Project 3 - Building built in Minutes: SfM and NeRF

Chinmay Kate
M.S. Robotics Engineering
Worcester Polytechnic Institute (WPI)
Worcester, MA 01609
Email: cskate@wpi.edu

Mandeep Singh
M.S. Robotics Engineering
Worcester Polytechnic Institute (WPI)
Worcester, MA 01609
Email: msingh2@wpi.edu

***Abstract*—In this project we implement computer vision methods to reconstruct 3D scenes and simultaneously obtain the camera poses using method called Structure from Motion. It uses technique which utilizes a series of 2D images and reconstruct the 3D structure of a scene. SfM can produce point cloud based 3-D Models similar to LiDAR. SfM uses Principle of Stereoscopic photogrammetry which uses triangulation method to calculate relative 3-D poses of object from stereo pairs.**

***Index Terms*—3D reconstruction, SfM, NeRF**

I. Phase 1 : Classical SfM pipeline

A. Overview

In the 3D reconstruction of scenes i.e In SfM pipeline most critical step is feature matching from the common points in the scene and eliminating the outliers using RANSAC algorithm. Then comes estimating Fundamental matrix which relates the corresponding points of two images from different views and later using this matrix to compute Essential Matrix. Camera poses are estimated and the right one is selected using cheirality constraints using Triangulation. We do this for n Perspectives and finally compute the re projection error and try to minimize this non-linear re-projection error using bundle adjustment.

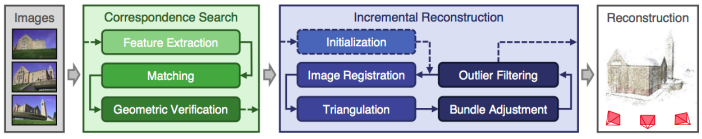


Fig. 1: Overview of SfM pipeline.

B. Dataset

Data been provided is of set of 5 images of Unity Hall. Fig 2 shows these 5 images taken from Samsung S22 Ultra's Main camera at f/1.8 aperture, ISO 50, 1/500 sec shutter speed. This camera is calibrated using Ran-tan Model with 2 radial parameters and 1 tangential parameters. So these 5 images are distortion corrected and resized to 800X600px.

C. Feature MAtching and RANSAC using Fundamental matrix

Good feature remains critical for Computer vision algorithm to work. Feature descriptor used is SIFT for it's high robustness in structure of motion problem. This data is provided in

Fig. 2: Images of Unity Hall

'matching.txt' file for all 5 images. We have 5 images and 4 matching '.txt' files. It has nFeatures: (the number of feature points of the ith image - each following row specifies matches across images given a feature location in the ith image and Each Row: (the number of matches for the jth feature) (Red Value) (Green Value) (Blue Value) (ucurrent image) (vcurrent image) (image id) (uimage id image) (vimage id image) (image id) (uimage id image) (vimage id image).

We need to extract these values from the '.txt' file and store in list of feature in x, feature in y, rgb-values, feature-flag map. These feature flag map contains 0's and 1's where if a point in ith image matches with other images then those image ids will have 1 rest zeros. These will help in PnP.

As data become noisy after SIFT feature descriptor, RANSAC is used with fundamental matrix with maximum no of Inliers. We use normalized 8-points Algorithm to calculate fundamental matrix. Fig 3 shows to calculate fundamental matrix. We normalize it as epipolar lines do not exactly pass through the center of point correspondences. We calculate the fundamental matrix using these normalized points and after that we retrieve the original fundamental matrix. Due to Noise in correspondances F can be full rank i.e 3, but we need to make it rank 2 by assigning zero to last diagonal element and thus we get the epipoles.

D. Essential Matrix Calculation

Relative camera Poses needs to be found between two images and using Fundamental matrix computed above and K matrix given which has the intrinsic values of camera in it Essential matrix is computed and is Decomposed using SVD. It's diagonal elements are again enforced to 1,1,0 due to this. This gives us the relative camera poses between two views. This is in image coordinates which has been normalised earlier unlike Fundamental matrix which is in pixel co-ordinates.

$$\begin{pmatrix} u'_1u_1 & u'_1v_1 & u'_1 & v'_1u_1 & v'_1v_1 & v'_1 & u_1 & v_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ u'_nu_n & u'_nv_n & u'_n & v'_nu_n & v'_nv_n & v'_n & u_n & v_n & 1 \end{pmatrix} \begin{pmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{pmatrix} = \mathbf{0}$$

$$\Leftrightarrow \mathbf{Af} = 0$$

Fig. 3: Fundamental Matrix calculation

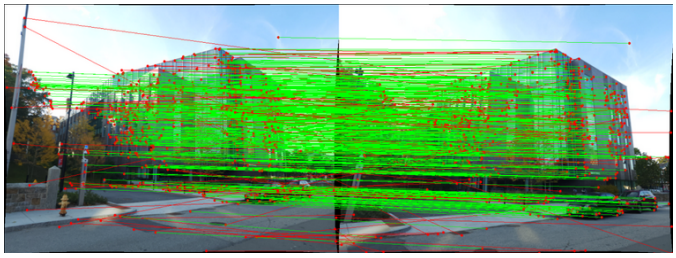


Fig. 4: Feature Matching for 1st 2 images

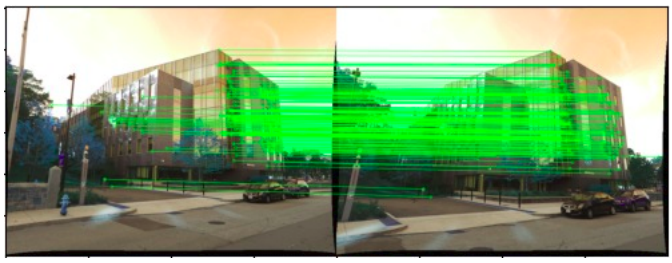


Fig. 5: Feature Matching of Inliers.

E. Camera Pose Estimation and Cheirality Condition using Triangulation

Here E matrix is decomposed using SVD and as camera pose has 6 DOF, 3 Rotational and 3 Translation. C's and R's are calculated using formula stated in Fig. 5. Here C is camera center and R is Camera Rotation.

We take two camera poses and point correspondence to find the X (3D-point) in the world using Linear Triangulation. We do this for all Camera poses to find X (3D point in front of camera having Z value positive. This is called depth positivity constraints.

Our task is to calculate unique camera pose out of 4 by removing the dis-ambiguity. This can be done using cheirality conditions i.e Reconstructed points should be in front of Cameras and r3(X-C) ¡ 0, Whereas r3 is third row of Rotational matrix.

After getting lineared triangulated 3D points, we try to minimize the re-projection error of the location of 3D points between actual points and re-projected points. In linear triangulation we minimize algebraic error and in Non-linear triangulation we try to minimize geometric error which is also called re-projection error which is more meaningful. So when we try to minimize the re-projection error we refine the location of 3D points. We get initial guess from the linear triangulation. We use function scipy.optimize and use trust region field as optimization. We write as re-projection error as given in Fig 6.

$$C_1 = U(:,3) \text{ and } R_1 = UWV^T$$
$$C_2 = -U(:,3) \text{ and } R_2 = UWV^T$$
$$C_3 = U(:,3) \text{ and } R_3 = UW^TV^T$$
$$C_4 = -U(:,3) \text{ and } R_4 = UW^TV^T$$

Fig. 6: Calculation of Different Camera Poses.

$$\min_x \sum_{j=1,2} \left(u^j - \frac{P_1^{jT}\widetilde{X}}{P_3^{jT}X}\right)^2 + \left(v^j - \frac{P_2^{jT}\widetilde{X}}{P_3^{jT}X}\right)^2$$

Fig. 7: Calculation of re-projection error.

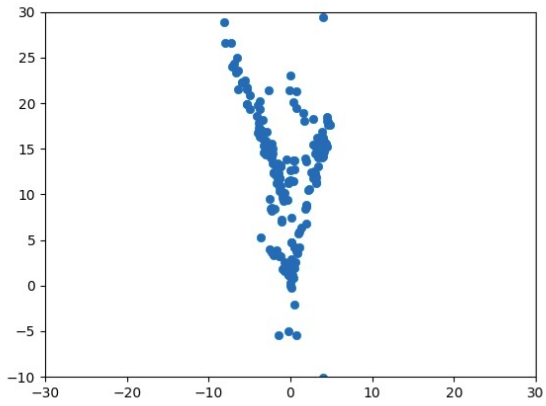


Fig. 8: Reconstruction of Scene with 2 Camera's after Cheirality condition.

F. Linear PnP, RANSAC, Non-Linear PnP.

We proceed with estimating camera poses with respect to reference alignment for the remaining images and again compute 3D points. We are given with 2D-3D correspondences and K matrix, we can again calculate R and C parameters. We normalize the 2D points with K matrix by performing $K^{-1} * x$ and removing interensic effect. We need at least 6 2D-3D correspondences to solve RT (3X4) matrix using SVD, which has translation and rotation elements. This has removed K effect from RT (3X4) matrix after normalization of 2D points. Here 3 columns of RT matrix has rotational elements which is orthonormal and due to errors we enforce it by decomposing

SVD and only multiply U and V. Also we find determinant of new R matrix and if it is -1, we multiply R matrix with -1. Translation vector is 3rd column of RT matrix.

As PnP produces too many errors we use RANSAC to eliminate the outliers using the re-projection error as discussed in above section and Fig. 6.Just like in triangulation, since we have the linearly estimated camera pose, we can refine the camera pose that minimizes the re-projection error. We again refine these location using Non-Linear PnP optimization using Scipy.optimize also we convert Rotation matrix into Quaternion as it is good choice to maintain orthogonality and translation vector when passing for optimization. This minimization is highly nonlinear because of the divisions and quaternion parameterization.

G. Bundle Adjustment

Till here we have computed all camera poses and 3D points. We need to refine these camera poses and 3D points together. We further want to refine the location points to get maximum accuracy and optimal values of the 3D points and cameras poses and so we perform Bundle adjustment. To start with the Bundle matrix we need to get Visibility matrix which finds the relationship between camera and point defined by V_{ij}, Here j is jth point visible in camera i.

For example, consider there are N image points, N3d world points, nC cameras (since number of image files provided were 6, maximum nC will be 6), where each camera has 6 extrinsic parameters, (Rotation: roll, pitch, yaw; Translation: cx,cy,cz). The sparsity matrix Mba has dimensions 2 N × (N3d 3 + nC 6). If image point at index 12 in N corresponds to world point at index 12 in N3d, then the elements of matrix Mba, that relate them, will be 1.

We can see high level of refining of the location using trust region reflective algorithm method of least square which is more robust to sparse problem. We get the refined 3D points and camera poses. And thus the pipeline is completed with the refinement. We can compare refinement before and after bundle adjustment.

H. Results Analysis

There are some observations regarding the refinement before and after Bundle adjustments and overall pipeline of the Structure of motion.

- Like it is mentioned in the Feature matching section, getting good features are always critical for any CV problem. And thus the bad data can cause lot of problem and needs to be refined and causes bad matching. It affects the F matrix calculation.
- Since this algorithm critically relies on optimization, We see this process of least square to be very slow and as discussed in class there are really better methods to implement the non-linear optimization to improve accuracy and speed.

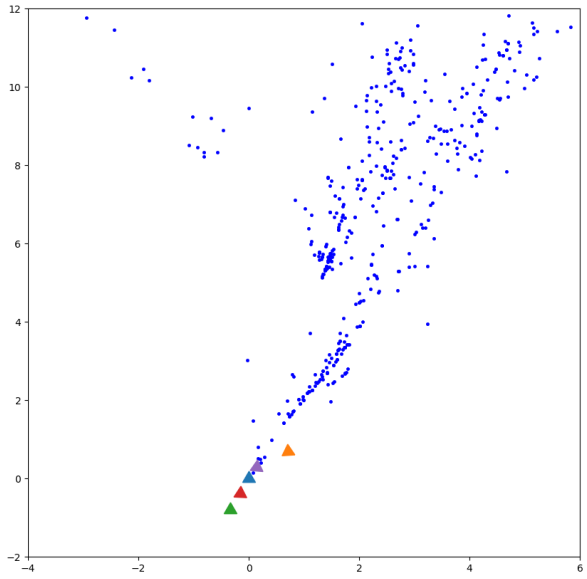


Fig. 9: The final reconstructed scene after Sparse Bundle Adjustment (SBA) for images 1 to 5.

II. Phase 2 : Deep Learning - NeRF

A. Introduction

In Deep Learning part, we will be implementing Neural Radiance Fields (NeRF) to synthesize novel views of complex scenes by optimizing a continuous volumetric scene function using a sparse set of input views. The input for NeRF is a 5D continuous array in which first 3 elements represent the 3d coordinates of the spatial location and the last two gives the direction of the ray formed by joining the particular image pixel to the camera center. The output of the NeRF is RGB color (radiance field) of the specific pixel and the volume density at that spatial location.

B. Model Input

First we will discuss what our dataset is for the model and how we preprocess it to feed to our neural network to get the desired output. We have a dataset of images of a lego structure as seen in Fig. . Along with the images we are also given camera poses of those images (i.e. camera to world transformation matrices).

Now, in NeRF's we are using classic volume rendering techniques. So, we will treat each image pixel in the image as a ray in real world and then we will sample points on that ray to get out input for the model. But first we have to convert everything to world co-ordinates, then only we can specify the ray direction and find 3D spatial co-ordinates.

1) Ray Generation: We have to generate rays from each pixel of the image. A typical formulation of ray looks like below equation:

Fig. 10: Sample picture of NeRF lego dataset.

$$\vec{r}(t) = \vec{o} + t\vec{d}$$

where 'o' is the origin, 't' is the sampling parameter and 'd' is the direction.
In our case our origin for the ray will be the pixel position of the image. Direction will be a unit vector along the vector joining the camera center and the particular pixel position. But currently all are values are in 2d image plane and in pixel coordinates. So, to get the rays follow the following steps:

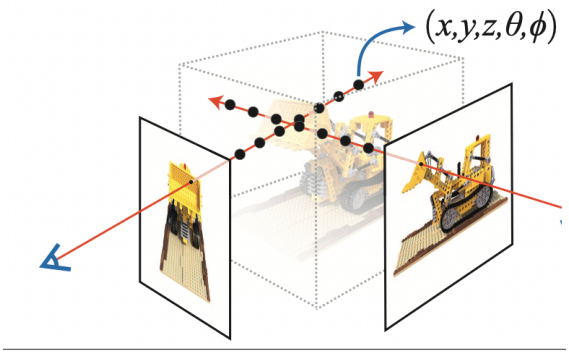
- Convert the image pixel coordinates to normalized coordinates with (i.e. wrt to camera center). The COLMAP frame is (X,-Y,-Z). Also, Z = -1 has been assumed.
- Now, we have ray direction wrt camera frame. If we multiply this vector by the camera to world transformation matrix (only rotation part), we will have this ray vector converted to world frame. and finally, we get the ray direction unit direction by dividing by the magnitude of the vector.
- The origin of the ray will be just the translation part of the camera to world transformation matrix

2) Sample Points: Now, we have both direction and origin of the ray and we are left to decide only sampling parameter to generate a ray. For sampling parameter, we are doing uniform sampling along the ray with some added noise so that the model is exposed to new data and thus better results could be obtained.

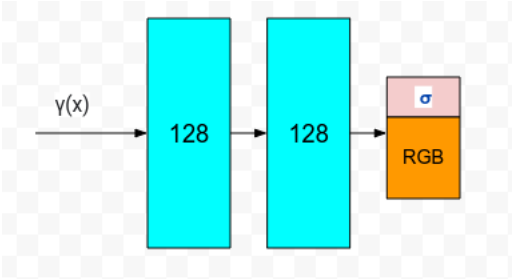
3) Positional Encoding: To get better results and render high frequency features we will be using positional encoding. the below given encoding function has been used in NeRF. In our function we have 6 terms for encoding. All the input values are encoded separately before we input it to the network for training.

C. Network - Multi Layer Perceptron

The input data is passed to our Network which is just a bunch of fully connected layers and giving us the output as volume density and RGB value at the particular sample point. The network architecture can be seen in the Fig. .



$$\gamma(p) = (\sin(2^0\pi p), \cos(2^0\pi p), \ldots, \sin(2^{L-1}\pi p), \cos(2^{L-1}\pi p))$$



Please note that we have used a very small network as compared to what has been implemented in the paper. Also, our input to the network are both spatial location and direction unlike the paper where direction input is given later while training only for the RGB values.

D. Volume Rendering

The output from the network is just RGB color value and volume density at a particular location. So, we use these predicted values to render the 3D scene. The predictions from the network are plugged into the classical volume rendering equation to derive the color of one particular point. the equation for which looks something like as shown below:

$$C \approx \sum_{i=1}^{N} T_i \alpha_i c_i$$

weights: T_i, α_i; colors: c_i

$$T_i = \prod_{j=1}^{i-1}(1 - \alpha_j)$$

Using volume density, we first calculate the transmittance till

$$\alpha_i = 1 - e^{-\sigma_i \delta t_i}$$

the particular sampling position and then multiply with the

predicted color at that position to get the final color in the image (radiance field). We repeat this process for all the pixels.

E. Loss Function

Once we get all the color (RGB) values after 3D volume rendering we can just compute photometric loss between these predicted color values and actual image values.

$$\mathcal{L} = \sum_{i}^{N} ||\mathcal{I}_i - \widehat{\mathcal{I}}_i||_2^2$$

$$\boldsymbol{\theta} = \arg\min_{\boldsymbol{\theta}} \mathcal{L}$$

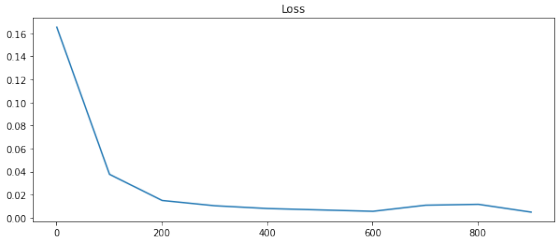
F. Network training parameters

Following are the parameters used to train Train NeRF deep learning model:

- Epochs = 1000
- Mini Batch size = 4096
- Near point = 2
- Far point = 2
- Number of samples on 1 ray = 32
- Learning rate = 5e-3
- Number of terms in encoding function = 6
- Image input size = 100 x 100

G. Results - Train and Test

The loss plot for the network for 1000 epochs has been plotted as shown below: Also, a sample rendered image can



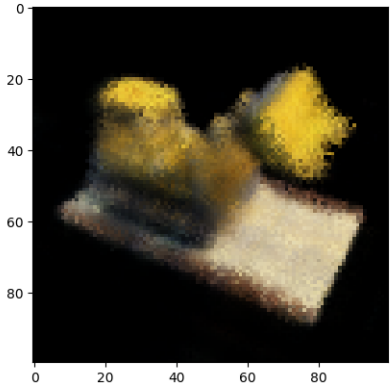
be seen after 1000 epochs. A sample gif has been attached with the results made with the predictions on the test set.

H. Conclusion and Problems Faced

In conclusion, the initial results are good. If the model is trained for more time and epochs the results can be far better. Also, if hierarchical sampling is used along with a bigger network, we can obtain better results.
The main problem was training time. If we used full sized images (800 x800), had used more number of samples along the ray, we will get sharp results. Also, proper vectorization of code is very critical to run the code fast and reduce the training time. This is where we faced difficulty to write vecotorized code and so took help from various online resources as mentioned in references.

(a)

Fig. 11: Original image



(a)

Fig. 12: Predicted image

Image ID	Linear pnp	Non - Linear pnp
1	-	-
2	-	-
3	2491.074	642.6049
4	136.06	96.2
5	1083.07	374.58

Fig. 13: PnP Linear and Non-Linear Error

Image ID	Linear Trinagulation					Non-Linear Triangulation				
	1	2	3	4	5	1	2	3	4	5
1	-	-	-	-	-	-	-	-	-	-
2	3.3185					3.27				
3	254.532	163.114	-	-	-	120.6052	16.2719	-	-	-
4	109.237	45.75	17.766	-	-	74.77	44.81	13.22	-	-
5	39491.52	136.13	45.723	8.06	-	355.96	134.07	11.234	7.44	-

Fig. 14: Triangulation Linear and Non-Linear Error.

III. References

- https://www.cc.gatech.edu/classes/AY2016/cs4476_fall/results/proj3/html/sdai30/index.html

Image ID	Bundle Adjustment				
	1	2	3	4	5
1	-	-	-	-	-
2	-	-	-	-	-
3	1.08592	40422.6	3622.316	-	-
4	87.8168	2669.78	4785.68	88.47314	-
5	530.72	27597.07	3871.86	3039.85	292.368

Fig. 15: Bundle Adjustment

- https://www.cis.upenn.edu/~cis580/Spring2015/Projects/proj2/proj2.pdf
- https://scipy-cookbook.readthedocs.io/items/bundle_adjustment.html
- https://pyimagesearch.com/2021/11/17/computer-graphics-and-deep-learning-with-nerf-using-tensorflow-and-keras-part-2/
- https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/vision/ipynb/nerf.ipynb#scrollTo=6HSYxrkAsW-S
- https://colab.research.google.com/drive/1rO8xo0TemN67d4mTpakrKrLp03b9bgCX#scrollTo=JovhcSy1NIhr