SfM and NeRF

Shounak Naik
Robotics Engineering Department,
Worcester Polytechnic Institute,
Worcester, MA, USA.
ssnaik@wpi.edu

Venkatesh Mullur
Robotics Engineering Department,
Worcester Polytechnic Institute,
Worcester, MA, USA.
vmullur@wpi.edu

I. Introduction

In this project, we estimate 3D reconstruction and simultaneously obtain the camera poses of a monocular camera when given 6 image sequences. Basically, we are given images captured after the effect of motion and then the goal of this project is to find the depth of each pair of images and reconstruct of a full 3D scene. This is usually called Structure from Motion, and mainly referred to as SLAM in the robotics domain; we also implement the state-of-art paper "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis". This is a deep learning part where we estimate a full 3D view given a dataset of images of the scene from different perspectives. A novel view is reconstructed with an MLP neural network with the help of classical volume rendering techniques and positional encoding.

II. SfM

The main parts that are done in SfM are given below:

1) Reading the matching.txt files and accessing the feature matches that are previously given. We can also use SIFT features instead of these matches by using harris corners.
2) Outlier rejection using RANSAC
3) Estimating Fundamental Matrix
4) Estimating Essential matrix
5) Estimating camera poses from essential matrix
6) Checking chirality condition from triangulation
7) PnP
8) Bundle Adjustment

A. Dataset

The data given to us are a set of 5 images of Unity Hall at WPI, using a Samsung S22 Ultra's primary camera at f/1.8 aperture, ISO 50, and 1/500 sec shutter speed. The camera is calibrated after resizing using a Radial-Tangential model with 2 radial parameters and 1 tangential parameter using MATLAB R2022a's Camera Calibrator Application. The images provided are already distortion-corrected and resized to 800×600 px. Since Structure from Motion relies heavily on good features and their matching, keypoint matching (SIFT keypoints and descriptors are used for high robustness) data is also provided in the same folder for pairs of images. The folder contained 5 text files named matchingX.txt. In each file, the matching feature from the Xth image and every other image was mentioned in the following format. Each row was a new feature and every row contains for example:
Matching1.txt
2 255 255 255 5.08304 116.978 3 49.0748 166.783
3 79 71 51 7.15528 197.921 2 11.255 225.237 5 259.685 103.719
Here, the first element is the number of matches. The next three are the RGB values of the match, the next two are the x and y coordinates of the match in the Xth image. The next element shows that the match in the Xth image is corresponding to a match in this (Yth) image. And the next two elements shows the x and y coordinates of the Yth image. For example, the second row tells us that there are 3 matches overall. The next 3 elements (79, 71, 51) are the RGB values of the same. The next 2 elements (7.15528, 197.921) are the pixel coordinates of that match in the 1st image. The next element (2) tells us that this match is present in the second image and its pixel coordinates are (11.255, 225.237). It also tells us that this match is present in the 5th image and its pixel coordinates are (259.685, 103.719). In the code, FindMatches() function does this job.

B. Estimating Fundamental Matrix and Outlier Rejection using RANSAC

Now that we have accessed the SIFT features in images, we basically know the image coordinates. Now using these image coordinates we can find the Fundamental Matrix using outlier rejection using RANSAC. Assuming A is the coordinates of the matches in the first image and B is the coordinates of the matches of the second image. The fundamental matrix can be calculated using the following.

$$[A.transpose][Fundamental - Matrix][B] = 0$$

Basically, the calculation of the fundamental matrix can be written as:

$$\begin{bmatrix} xA' & yA' & 1 \end{bmatrix} * \begin{bmatrix} f11 & f12 & f13 \\ f21 & f22 & f23 \\ f31 & f32 & f33 \end{bmatrix} * \begin{bmatrix} xB \\ yB \\ 1 \end{bmatrix} = 0$$

Before using the coordinates of SIFT Feature matches of the two images for calculating the fundamental matrix, we first need to normalize them as follows.

$$\begin{bmatrix} x_1x'_1 & x_1y'_1 & x_1 & y_1x'_1 & y_1y'_1 & y_1 & x'_1 & y'_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_mx'_m & x_my'_m & x_m & y_mx'_m & y_my'_m & y_m & x'_m & y'_m & 1 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{21} \\ f_{31} \\ f_{12} \\ f_{22} \\ f_{32} \\ f_{13} \\ f_{23} \\ f_{33} \end{bmatrix} = 0$$

Fig. 1. 8-point algorithm of finding fundamental matrix

1) Compute the centroid of all corresponding points in a single image.

$$u_{dash} = \frac{1}{n}\sum_{i=1}^{n} u_i$$

$$v_{dash} = \frac{1}{n}\sum_{i=1}^{n} v_i$$

2) Recenter by subtracting the mean u and v coordinates from the original point correspondences to obtain.
3) Define the scale term s and s' to be the average distances of the centered points from the origin in both the left and right images.
4) Construct the transformation matrices Ta and Tb.
5) Compute the normalized correspondences.
6) Solve for the fundamental matrix F by applying the eight-point algorithm on the normalized set of point correspondences computed in the previous step.
7) After obtaining the normalized fundamental matrix Fnorm, retrieve the fundamental matrix in the original coordinate frame using the following formula.

$$F_{original} = T_b^T * F_{normalized} * T_a$$

Basically, to calculate the Fundamental matrix, we need 8 points, using RANSAC we will find the best fundamental matrix and reject the outliers.

- We first select any 8 random matches and calculate the fundamental matrix.
- Using this fundamental matrix, we calculate the error.
- If the error is less than the threshold value, the points are considered inliers.
- The fundamental matrix with a maximum number of inliers is selected as the final Fundamental Matrix as shown in fig 2.
- This is done for all the matches until the number of iterations is completed.

To calculate the F matrix from fig.1, we need to find the SVD of the first matrix and find the column corresponding to the least singular value from the right singular matrix.
After this is done, SVD clean-up is done on the F matrix caused due to the noise in correspondences and so, to enforce the rank 2 constraint of the S matrix, the last singular value of the estimated F must be set to zero. If F has a full rank then it will have an empty null space i.e. it won't have any point that is on the entire set of lines. Thus, there wouldn't be any epipoles.

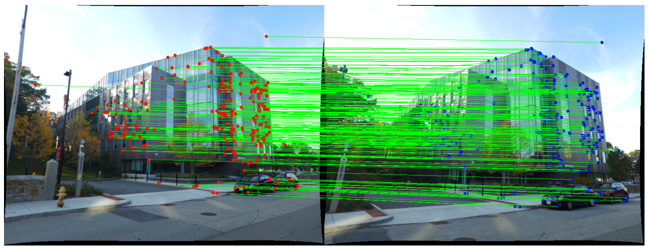

Fig. 2. Inliers after doing RANSAC to find Fundamental Matrix.

The best Fundamental Matrix found is given below with 237 inliers.

$$F = \begin{bmatrix} -2.363945e-08 & -2.983864e-05 & 1.259266e-02 \\ 3.255821e-05 & 2.512670e-06 & -3.338558e-02 \\ -1.446742e-02 & 3.172725e-02 & 1.000000e+00 \end{bmatrix}$$

C. Finding Essential Matrix from Fundamental Matrix

The Fundamental Matrix is the algebraic representation of the epipolar geometry that is defined in the original image space. But, when we want to estimate the depth in the real world, the intrinsic and extrinsic of the camera should be integrated. So, we take into consideration the essential matrix. This essential matrix obeys the Pinhole model unlike the fundamental matrix and can be found by the following equation.

$$E = K^T * F * K$$

where K is the camera calibration or the camera intrinsic matrix. The essential matrix is basically found by taking the above-given E matrix and then forcing the diagonal elements such that the last diagonal element is 0 and the other elements are 1.

$$E = \begin{bmatrix} 0.00159927 & -0.60735138 & 0.1215597 \\ 0.65869385 & 0.0471868 & -0.73597493 \\ -0.16547409 & 0.78143014 & 0.0238111 \end{bmatrix}$$

This essential matrix is decomposed into 4 poses which are 4 rotation matrices and 4 translation vectors. Further, we need to disambiguate these poses to find the correct pose.

D. Estimating Camera Poses

Since we get 4 camera poses from the essential matrix, we need to disambiguate the correct one using the depth positivity constraint. The following 4 camera poses were found from our essential matrix.

$$R = \begin{bmatrix} 0.99644881 & 0.02187593 & 0.08130937 \\ -0.02606067 & 0.99837044 & 0.0507672 \\ -0.08006629 & -0.05270589 & 0.99539514 \end{bmatrix}$$

and

$$C = \begin{bmatrix} 0.83552555 & 0.14311217 & 0.53048654 \end{bmatrix}^T$$

$$R = \begin{bmatrix} 0.99644881 & 0.02187593 & 0.08130937 \\ -0.02606067 & 0.99837044 & 0.0507672 \\ -0.08006629 & -0.05270589 & 0.99539514 \end{bmatrix}$$

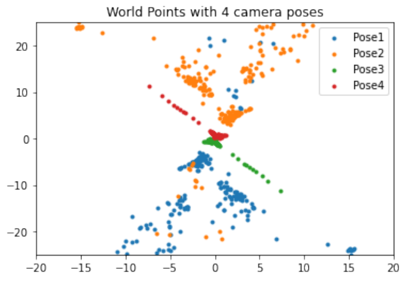

Fig. 3. All 4 camera Poses.

$$\begin{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ 1 \end{bmatrix}_\times P_1 \\ \begin{bmatrix} \mathbf{x}_2 \\ 1 \end{bmatrix}_\times P_2 \\ \vdots \\ \begin{bmatrix} \mathbf{x}_N \\ 1 \end{bmatrix}_\times P_N \end{bmatrix} \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix} = \boxed{0}$$

Fig. 4. Calculating Linear Triangulation

and

$$C = \begin{bmatrix} -0.83552555 & -0.14311217 & -0.53048654 \end{bmatrix}^T$$

$$R = \begin{bmatrix} 0.31759015 & 0.20070321 & 0.92674415 \\ 0.25113454 & -0.96024619 & 0.12189625 \\ 0.91436751 & 0.19402442 & -0.35536824 \end{bmatrix}$$

and

$$C = \begin{bmatrix} 0.83552555 & 0.14311217 & 0.53048654 \end{bmatrix}^T$$

$$R = \begin{bmatrix} 0.31759015 & 0.20070321 & 0.92674415 \\ 0.25113454 & -0.96024619 & 0.12189625 \\ 0.91436751 & 0.19402442 & -0.35536824 \end{bmatrix}$$

and

$$C = \begin{bmatrix} -0.83552555 & -0.14311217 & -0.53048654 \end{bmatrix}^T$$

To find the correct camera pose, we need to use the following depth positivity constraint or the chierality constraint.

$$r3(X - C) > 0$$

E. Linear and Non-Linear triangulation

With the given pairs of rotation matrices and translation vectors, now we have to generate 3D points. Basically, we have to triangulate them to construct 3D world points out of them. This can be done using the projection matrices of every pose and the image points.

Linear triangulation is calculated using the above-given in fig 4, where Px is the projection matrix and the x is the image points. We perform SVD on the $yellow$ highlighted matrix and find the right singular vector to get the $green$ matrix. It basically is in the form $Ax = B$

$$\epsilon_{\text{geom,CM}} = \left(\frac{P_1 \mathbf{X}}{P_3 \mathbf{X}} - x \right)^2 + \left(\frac{P_2 \mathbf{X}}{P_3 \mathbf{X}} - y \right)^2$$

Fig. 5. Calculating Non-Linear Triangulation

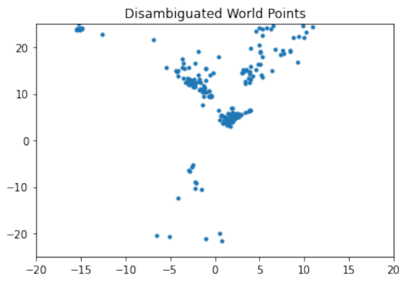

Fig. 6. Linear Triangulation

Linear triangulation takes the 2D points into 3D world points, but they are some sort of geometric error that is handled by non-linear triangulation. In non-linear triangulation, the image points are reprojected, and then the error is optimized, thus new image points are obtained.

We use the equations in the fig 5 to perform non linear triangulation for every every image where Px is the row in the projection matrix of that image. We use $optimize.least_squares$ from sklearn to reduce the geometric error.

F. Linear PnP and Non-linear PnP

With the camera intrinsic matrix, the 3D world points obtained from the non-linear triangulation and the image points of the next view, we can find the common features which are present and then perform Linear PnP. The 2D points are normalized by calibration matrix to get the camera poses.

$$K^{-1}x$$

We, now can solve the system of equations using 6 2D image points of the new image and corresponding 3D points to get a Linear PnP that gives a pose matrix of 3×4.

Since earlier we removed the normalized 2D points using the camera calibration matrix when we did $K^{-1}x$, or else we would have gotten a projection matrix as the output.

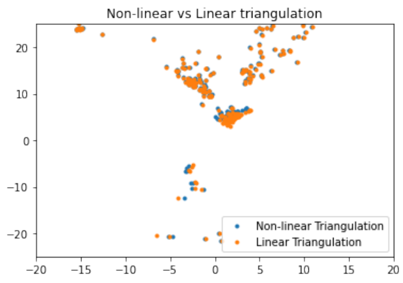

Fig. 7. Non-Linear Triangulation

```
$n = 0$
for $i$ = 1:M do
    // Choose 6 correspondences, $\hat{X}$ and $\hat{x}$, randomly
    $[C\ R] = \text{LinearPnP}(\hat{X}, \hat{x}, K)$;
    $\mathcal{S} = \emptyset$;
    for $j$ = 1:N do
        // Measure Reprojection error
        $e = \left(u - \frac{P_1^T \tilde{X}}{P_3^T \tilde{X}}\right)^2 + \left(v - \frac{P_2^T \tilde{X}}{P_3^T \tilde{X}}\right)^2$;
        if $e < \epsilon_r$ then
            $\mathcal{S} = \mathcal{S} \cup \{j\}$
        end
    end
    if $n < |\mathcal{S}|$ then
        $n = |\mathcal{S}|$;
        $\mathcal{S}_{in} = \mathcal{S}$
    end
end
```

Fig. 8. Non-Linear Triangulation

Whereas we got a pose matrix where the first three columns are the orthonormal rotation matrix that may contain errors. To counter this, we perform svd on it and then multiply the left and right singular matrix.
The translation vector is defined as the $t = -R^T C$ PnP is prone to error as there are outliers in the given set of point correspondences and to get rid of this error, we perform RANSAC which is given in the fig 8

The linear PnP reduces the algebraic error similar to the triangulation method, still these R and C found in the linrar PnP is erraneous and has to be optimized more using the reprojection error; essentially decreasing the geometric errors. For the same we use the following equation:

$$\min_{C,q} \sum_{i=1,J} \left(u^j - \frac{P_1^{jT} \widetilde{X_j}}{P_3^{jT} \widetilde{X_j}}\right)^2 + \left(v^j - \frac{P_2^{jT} \widetilde{X_j}}{P_3^{jT} X_j}\right)^2$$

here, P_j is the column of the projection matrix formed by the R and C in the previous step and the $\widetilde{X_j}$ is the homogeneous form of the world points. Here, instead of using the rotation matrix, it is efficient to use a quaternion to optimize the error. To optimize, we use *scipy.optimize.least_squares*
We observed that our non-linear error is more than the linear error in certain runtimes.

G. Bundle Adjustment and Visibility Matrix

It is basically refinement of all the camera poses that we got in the previous step.

$$\min_{\{C_i,q_i\}_{i=1}^{i},\{X\}_{j=1}^{J}} \sum_{i=1}^{I} \sum_{j=1}^{J} V_{ij} \left(\left(u^j - \frac{P_1^{jT} \tilde{X}}{P_3^{jT} \tilde{X}}\right)^2 + \left(v^j - \frac{P_2^{jT} \tilde{X}}{P_3^{jT} \tilde{X}}\right)^2 \right)$$

here V_{ij} is the Visibility matrix.
Basically, we want to optimize everything again with different perspectives using bundle adjustments. It is basically simultaneously refinement of the 3D coordinates describing the scene, the relative parameters and the optical characteristics of the camera.
Visibility matrix is a matrix of size of inliers to the number of images and has a value of 1 if the n^{th} inlier is visible from the image i or 0 if not visible.
Fig 9 shows an ideal bundle adjustment output which is taken a 3rd party resource online. For the sake of showing the output, we have borrowed the snippet for PnP and bundle adjustment.

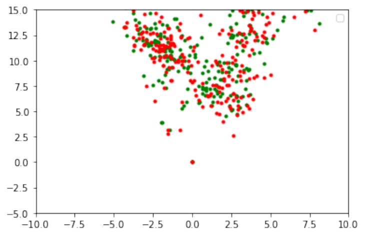

Fig. 9. Bundle Adjustment

III. Neural Radiance Fields: NeRF

NeRF is a neural rendering method that gives us novel scenes if we input the model with ground truth poses. The model optimizes on the input ground truth poses and then can be used as a function approximator for any input pose given in the scene. In simpler terms, NeRF is capable of giving a new view of an object if we give it some input views of the object.
The input given to the NeRF model is a 5D tensor (spatial location (x,y,z) and viewing direction (θ and ψ). The output is the RGB colors and the output density.

A. NeRF dataset

We have used the NumPy file of the Lego bulldozer as our dataset. This dataset contains 106 images and 106 respective poses. We have used 100 poses as our training data, 1 for validation and 5 for testing.

B. Ray Generation

NeRF relies on ray-based calculations. Thus it is imperative to understand how rays are generated for each pose. We can find out the camera position and the rotation matrix from the pose matrix. Based on the pinhole equation, the direction of the rays for each pixel in the image can be found out with respect to the camera frame. Using the rotation matrix, we can then get the ray representation in the world frame. These calculations are done in the *get rays* function in our code.

C. Ray Stratification

Ideally, we want to sample each point on the ray but since this is impossible, we rely on discrete points on the ray. We generate random points between the two ends of the ray. The 2 ends of the ray are taken arbitrarily. This stratification code can be found in the *get stratified* method.

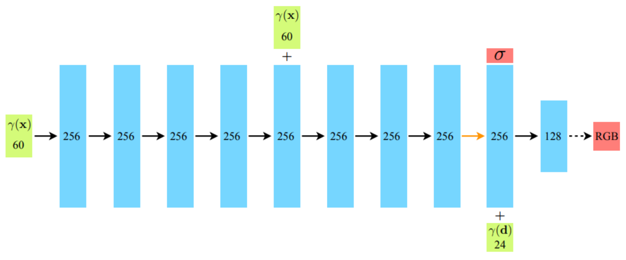

Fig. 10. NeRF Architecture implemented

$$\hat{C}(\mathbf{r}) = \sum_{i=1}^{N} T_i(1 - \exp(-\sigma_i\delta_i))\mathbf{c}_i\,, \text{ where } T_i = \exp\left(-\sum_{j=1}^{i-1}\sigma_j\delta_j\right), \tag{3}$$

Fig. 11. Volume Rendering Equation

D. Positional Encoding

This is a crucial part of the NeRF implementation. It is mentioned in the paper that Neural Networks aren't good with high-frequency inputs. To mitigate this, the paper recommends Positional Encoding in which we modulate our raw input with sine and cosine frequency bands. The paper recommends setting the number of frequency bands to be 10 for the spatial part and 4 for the viewing directions part. We have followed their recommendation.

E. NeRF model

We have built the exact architecture described in the NeRF paper. The architecture can be seen in the Figure 10. We take in the spatial part of the input and pass it through a 8 layered MLP. On the 4th layer of this MLP, we add the spatial input again as a form of skip connection. The 9th layer is again a Linear layer. This 9th layer spits out the 1 dimensional output density logit with the ReLU activation. For the RGB ouputs, the 9th layer is concatenated with the viewing directions input. The 11th layer is the output layer which gives us the RGB by using a sigmoid activate function. The orange arrow between the 8th and 9th arrow represents no activation. All the rest of the activations are ReLU.

F. Volume Rendering

This is the part that converts the output RGB values and the output density values back into a colour map. This is done by using the equation given in the figure 11. The σ is the output density. ∂ is the distance between consecutive points in a ray. c is the RGB values. This equation basically sums up the RGB values across views according to the output density.

G. Implementation Details

We trained our NeRF model for 7500 iterations with only 1 pose being inputted per iteration. We have batched the number of rays given as an input to our model. This is done since the GPU could not handle all the rays of the image all at once. Our model took 7 hours to train on the university cluster.

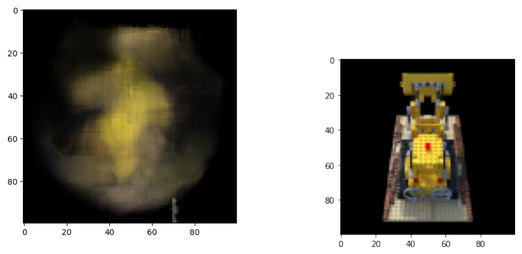

Fig. 12. Left image is the predicted image and the right one is the Ground truth

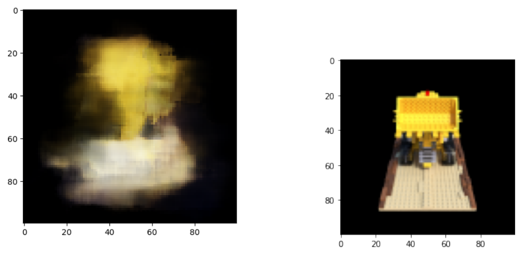

Fig. 13. Left image is the predicted image and the right one is the Ground truth

H. Results

To test the trained NeRF network, we passed 4 test poses as input. We then compared the ground truth image against the model's predicted colour map. The results can be seen in Figures 12 and 13.

I. Possible Improvements

The results from our model are blurry. We suggest two ways we could improve our results. 1) Training the NeRF for significantly more iterations. The original paper trained the network for 200,000 iterations. 2) Using Hierarchical Volume Sampling would help improve reconstruct texture in the image.

IV. Conclusions

The project helped us understand the structure of motion from epipolar geometry and the state-of-art deep learning paper NeRF and its research.

References

[1] https://arxiv.org/pdf/2003.08934.pdf

[2] $https://en.wikipedia.org/wiki/Structure_{f}rom_{m}otion$

[3] https://towardsdatascience.com/its-nerf-from-nothing-build-a-vanilla-nerf-with-pytorch-7846e4c45666

[4] $https://www.cc.gatech.edu/classes/AY2016/cs4476_{f}all/results/proj3/html/sdai30/index.html$

[5] https://github.com/sakshikakde

[6] https://github.com/akathpal