P2 - Buildings built in minutes - SfM and NeRF

Uday Sankar
usankar@wpi.edu

Gowri Shankar Sai Manikandan
gmanikandan@wpi.edu

Shaurya Parashar
sparashar@wpi.edu

***Abstract*—In this project, we intend to reconstruct a 3D scene and simultaneously obtain the camera poses of a monocular camera with respect to given scene. This is to be achieved by two approaches - Structure from Motion (SfM) and Neural Radiance Fields(NeRF). Based on the approach taken, the project is split into two phases. In phase 1, features will be read from the given text files and which will be used to reconstruct the 3D-scene. In phase 2, a NeRF based network is implemented for the 3D reconstruction.**

I. PHASE 1: CLASSICAL APPROACH TO SFM

A. Introduction

The objective of this section is to reconstruct a 3D scene from a set of images using classical computer vision techniques. This can be achieved by using the following steps:

1) Feature Matching and Outlier rejection using RANSAC.
2) Estimating Fundamental Matrix.
3) Estimating Essential Matrix from Fundamental Matrix.
4) Check for Cheirality Condition using Triangulation.
5) Perspective-n-Point.
6) Bundle Adjustment.

The pipeline in the form of algorithm is shown in figure 1.

Data: Image Matches, K
Result: X, C, R
for *all possible pair of images* **do**
  // Reject outlier correspondences
  [x1, x2] = GetInliersRANSAC(x1, x2);
end
// For first two images
F = EstimateFundamentalMatrix(x1, x2);
E = EssentialMatrixFromFundamentalMatrix(F, K);
[Cset, Rset] = ExtractCameraPose(E);
// Perform linear triangulation
for *i = 1:4* **do**
  Xseti = LinearTriangulation(K, zeros(3,1), eye(3), Cseti, Rseti, x1, x2);
end
// Check cheirality condition
[C R] = DisambiguateCameraPose(Cset, Rset, Xset);
// Perform Non-linear triangulation
X = NonlinearTriangulation(K, zeros(3,1), eye(3), C, R, x1, x2, X0));
Cset = C, Rset = R;
// Register camera and add 3D points for the rest of images
for *i=3:I* **do**
  // Register the i^{th} image using PnP.
  [Cnew Rnew] = PnPRANSAC(X, x, K);
  [Cnew Rnew] = NonlinearPnP(X, x, K, Cnew, Rnew);
  Cset = Cset∪Cnew, Rset = Rset∪Rnew;
  // Add new 3D points.
  Xnew = LinearTriangulation(K, C0, R0, Cnew, Rnew, x1, x2);
  Xnew = NonlinearTriangulation(K, C0, R0, Cnew, Rnew, x1, x2, X0);
  X = X∪Xnew;
  // Build Visibility Matrix.
  V = BuildVisibilityMatrix(traj);
  // Perform Bundle Adjustment.
  [Cset Rset X] = BundleAdjustment(Cset, Rset, X, K, traj, V);
end

Fig. 1: The pipeline of SfM in the form of an algorithm.

B. Feature Matching and Outlier rejection using RANSAC

Firstly, the feature matches are obtained from the text files provided with the data. Then, RANSAC is performed on it to obtain the inlier correspondences. These correspondences can now be used to compute the fundamental matrix. This can be done using the 8-point algorithm.
We need a minimum of seven-point correspondences to compute the Fundamental Matrix. In the 8-point algorithm, we randomly choose 8-point correspondences and use them to compute the Fundamental matrix for the two views. We use the property of the Fundamental Matrix $x'^T F x = 0$ to define the error for thresholding in RANSAC. We run 1000 iterations with an error threshold of 0.05

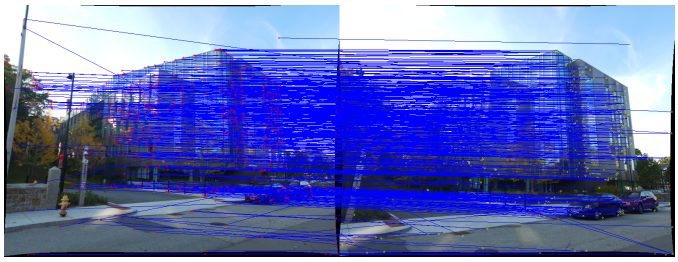

Fig. 2: Feature correspondences between image 1 and 2 before RANSAC

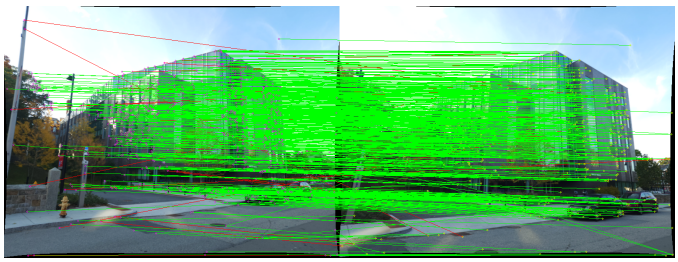

Fig. 3: Feature correspondences between image 1 and 2 after RANSAC

C. Estimating Fundamental Matrix

The Fundamental matrix is a mathematical description of the geometric relationship between two views of a scene. Specifically, the fundamental matrix relates a point in one view to the corresponding epipolar line in the other view. In our implementation, we estimate the fundamental matrix between the views obtained from the first two images. Using this, we can find the epipoles and epipolar lines in both images.

The fundamental matrix has 7 degrees of freedom and is ideally a matrix of rank 2. However, there is noise and distortion in the data, so the rank is not exactly zero. This means that the epipolar lines don't intersect at a single point. To deal with this, we enforce the rank-2 constraint by making the least singular value of the $m \times 9$ A matrix zero and recomputing the Fundamental Matrix. Equation 1 shows the value for our Fundamental Matrix

$$\mathrm{F} = \begin{matrix} -7.82311237e^{-07} & -5.09308444e^{-06} & 6.56641387e^{-03} \\ 4.70022641e^{-06} & 4.20246751e^{-07} & -4.61649491e^{-03} \\ -6.14401877e^{-03} & 4.70015352e^{-03} & -2.47418851e^{-02} \end{matrix} \quad (1)$$

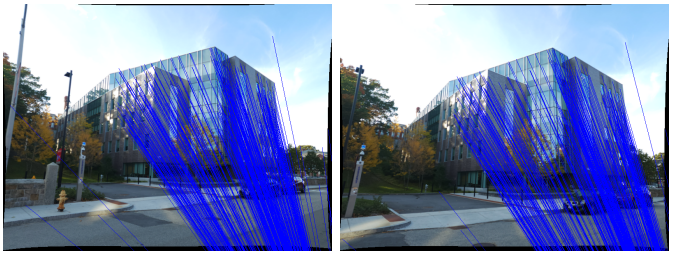

Fig. 4: The points connected with the epipoles in the images 1 and 2.

D. Essential Matrix from Fundamental Matrix

Mathematically, the Essential Matrix is a specialized case of a fundamental matrix that uses normalized image coordinates. The essential matrix can be thought of as the fundamental matrix with the inherent assumption of a calibrated camera. The Essential Matrix has 5 degrees of freedom. It can be decoupled into a rotation and a translation between the two views. We use this matrix to estimate the camera centers and rotations, thus giving us the camera pose.

Equation 1 shows the value for our Essential Matrix

$$\mathrm{E} = \begin{pmatrix} -0.05662862 & -0.45446132 & 0.76931046 \\ 0.39898129 & 0.0405199 & -0.41851787 \\ -0.80855968 & 0.45677777 & -0.00443761 \end{pmatrix} \quad (2)$$

E. Estimating Camera Pose

We can get the Camera Matrices from the Essential Matrix as described on the project webpage. We assume that the first camera matrix $P = [I|0]$, which means, we define a starting point as there is no concept of a reference point like the origin for this problem.

It is important to note that the factorization of the Essential Matrix into a skew-symmetric and a rotation matrix $E = SR$ leads to two possible solutions for the rotation matrix $R = UWV^T$ or UW^TV^T, and two values for the translation vector t, one positive, and one negative. Therefore, for our assumption $P = [I|O]$, there are four possible solutions for the second camera pose. We deal with this as mentioned on the project webpage in *DisambiguateCameraPose.py*

F. Triangulation Check for Cheirality Condition

1) Linear Triangulation: This is the first step in the creation of our "Structure". We use the obtained camera poses to triangulate a 3D world point. This is done by minimizing the algebraic error using linear least squares method.

To check the Cheirality condition, triangulate the 3D points (given two camera poses) using linear least squares to check the sign of the depth Z in the camera coordinate system w.r.t. camera center. A 3D point X is in front of the camera if the value of Z is positive.

2) Non-linear Triangulation: Linear triangulation provides us with the 3D world coordinates, but these points aren't accurate. Other than the limited capabilities of linear optimization methods, intuitively, the algebraic error doesn't model any geometric properties of the two-view problem. We use non-linear optimization to minimize the geometric error obtained by reprojecting these points, which is a more intuitive way to model this error.

G. Perspective-n-Points

We found the camera pose and 3D world points for two views. Now, we need to add more views to get more 3D points. Given 2D-3D point correspondences and camera intrinsic, we use linear least squares optimization to get the 6-DOF camera pose. This optimization doesn't ensure the orthogonality of the rotation matrix obtained, we enforce this condition ourselves in our code.

Because we are using point correspondences in our optimization, there is a possibility of outliers which will make the computed poses prone to error. In order to make our algorithm more robust to outliers, we perform RANSAC to refine our selection of feature correspondences.

Similar to the case of triangulation, the limited capability of linear optimization, along with the non-intuitive nature of the minimized algebraic error leads to inaccurate results. Therefore, to overcome this inaccuracy, similarly to the case of triangulation, we use non-linear least squares optimization on the inliers obtained.

H. Bundle Adjustment

After successfully adding all 5 views, we have decent estimates of camera poses and 3D world points. However, it is important to note that poses and 3D points have been estimated independently, using each other. The obvious issue with this approach is the inevitable accumulation of error for both estimates. Using erroneous poses to 3D estimate points, or conversely, using erroneous point correspondences to estimate poses for new views only increases the error.

In order to deal with this, we use Bundle Adjustment to optimize for poses as well as 3D points simultaneously. At first thought, this problem seems very complex, and even more than that, computationally expensive. However, we exploit the fact that the visibility matrix is actually really sparse. We can break the Jacobian down into components, where the major chunk is a block diagonal matrix. This sparsity of the visibility matrix allows us to use Sparse Bundle Adjustment(SBA) to optimize for poses and 3D points, all at once.

I. Reprojection Errors

Table **??** shows the values for reprojection errors after each stage. These error values are extremely high, which indicates

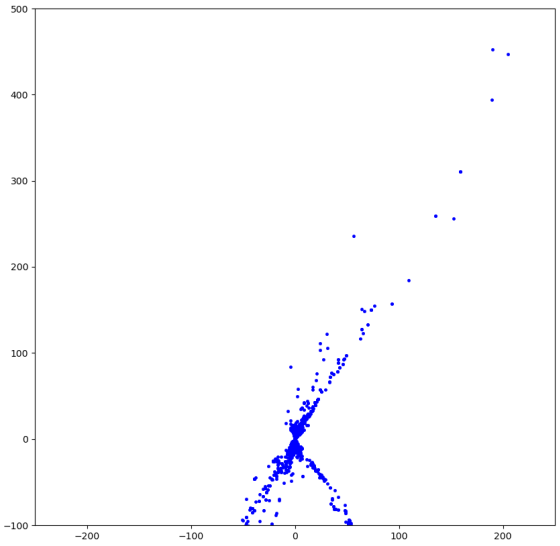

Fig. 5: Our output for bundle adjustment.

issues with our method that need to be resolved. Inliers computed after RANSAC seem decent, the implementation of the 8-point algorithm is correct, however, it can only be confirmed after attempting to visualize additional properties of the Fundamental Matrix. There could also be an issue with the implementation of linear triangulation. Non-linear triangulation reduces the error significantly, which makes it less likely to be buggy.

Stage (after)	Linear Triangulation	Non-linear Triangulation	Linear PnP	Non-linear PnP
1 and 2	801686.056	1058.804	N/A	N/A
1 and 3	521736.365	1487.306	12606501.605	488.239
1 and 4	3210.958	1220.739	379774.094	3743.646
1 and 5	412694.327	9780.350	6729451.654	28352.638
2 and 3	1070.840	1012.410	N/A	N/A
2 and 4	3499.124	849.977	N/A	N/A
2 and 5	28441.251	8059.421	N/A	N/A
3 and 4	11860.719	171.172	N/A	N/A
3 and 5	252718.230	10413.955	N/A	N/A
4 and 5	235330.370	5940.815	N/A	N/A

Table 1: Reprojection Errors

II. PHASE 2: DEEP LEARNING APPROACH: NERF

A. Iteration 1

1) Neural Network: We created a neural network of the same number of layers, channels, and positional and direction encoding as given in the NeRF research paper(Figure 29).

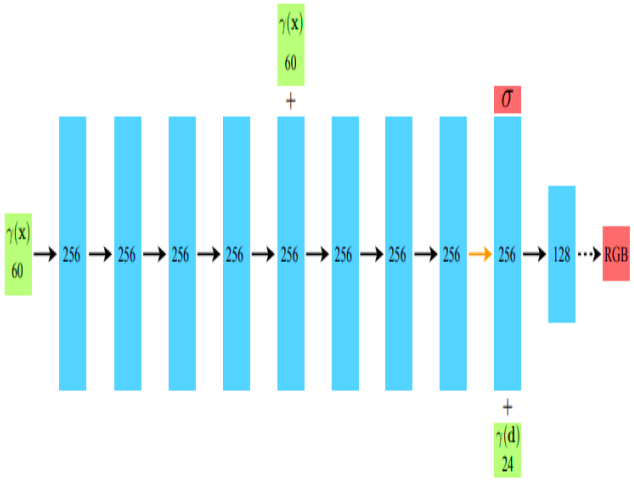

Fig. 6: NeRF Neural Network Architecture

2) Dataset and Dataset Loader: For the dataset, we considered the original LEGO dataset, the link of which was given in the project details as well. The images were **downsampled from 800*800 to 100*100 pixels**, for easier computation. We built a custom dataset loader for the same.

Focal length was calculated using the given camera angle x, and width of the image. Then, the ray directions and ray origins were computed using a mesh grid. For the ground truth, we considered each pixel from each image as ground truth, for the computed rendered ray. The total number of pixels thus for 100 images was 1,000,000. It's important to note that all pixels from all images were opened up to form a tensor of (total pixels,3). The rendered rays were also computed in this form, as it's faster to compute loss pixel-wise compared to the whole image. In addition, positional and directional encoding was also done.

3) Rendering: The render function takes as input a nerf-model object, which represents a Neural Radiance Field (NeRF) model, and a set of ray origins and ray directions.

We create a one-dimensional tensor t, which has a number of evenly sampled points along each ray, with a proper range of distance defined between them. Uniform Perturbing to the sampling points is done in order to add randomness in the sampling of each ray.

For each sampled point, we compute the color and density by calling the nerf model with the 3D point and direction of the ray passing through that point. The colors and densities are reshaped into a tensor that has the same shape as the sampled points along the rays.

Next, the accumulated transmittance and weights for each sampled point along the ray are computed. Transmittance is the fraction of light that is transmitted through a medium, and in this case, it represents the amount of light that is not absorbed or scattered by the medium along the ray. The accumulated transmittance is computed by taking the cumulative product of (1 - the absorption coefficient) at each sampled point.

The weights are computed as the product of the accumulated transmittance and the absorption coefficient at each point.

Finally, we find out the final color by summing the product of the weights and colors across all sampled points along the ray. The weighted sum is also computed, and a correction factor is added to ensure that the final color is normalized.

In summary, we are performing rendering by getting the colors of a set of rays passing through an image plane using a NeRF model by sampling points along each ray, perturbing the sampling positions, computing the colors and densities at each sampled point using the NeRF model, and accumulating the transmittance and weights for each sampled point along the ray to compute the final color.

4) Result: Adam Optimizer was used, along with trying out several learning rates, 5e-4, being one of the training losses in Figure 30. The model was trained for 500 epochs, with the batch size being 100.

We were getting an erroneous training plot for the NeRF model. The loss didn't seem to converge, the model seems to be getting stuck, rather than learning anything. We tried rectifying the code at each step, starting from the data loader to the final image. We changed the data loader, We also tried changing the learning rate, and other hyperparameters. However, we were unable to figure out the issue, and as a result, mostly got black color as output rays (Figure 31).

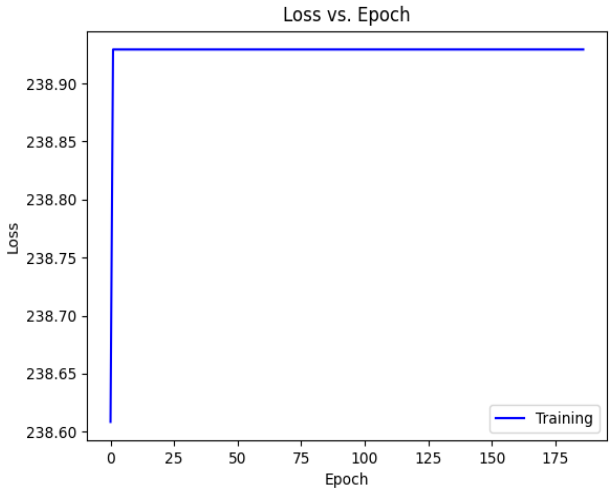

Fig. 7: NeRF Loss Curve

Fig. 8: NeRF Output

B. Tiny Nerf

After several failed attempts to find out the error in our implementation of NERF, we studied and did the implementation done in the site [1], named Tiny-NeRF (https://morioh.com/p/c20de9c3e270). This was not coded by us, but we implemented this in order to draw comparisons on where our implementation might be failing. Tiny-NeRF has a compacted neural network architecture of 3 fully connected layers, instead of the original paper's eight layers. Otherwise, the pipeline is almost the same as that of the original NeRF. The ray direction and ray origins were obtained using a mesh grid (100*100), then a sampling of points are done on the rays, in order to find color and density for each point using tiny-NeRF model, which is then cumulated to a single color for the pixel. The result that we got after implementing tiny-NeRF is also given below, for 700 iterations(Figure 32).

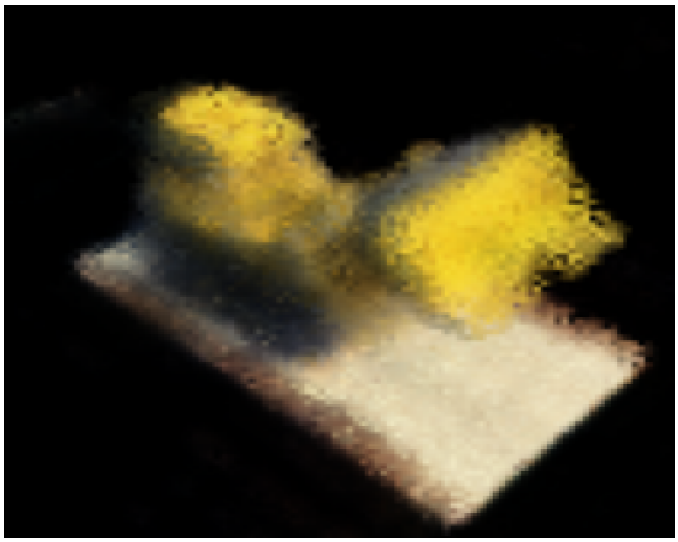

Fig. 9: Tiny NeRF Output

C. Probable reasons for our model failure

- The first reason we think may be due to the length of the neural network. The number of parameters is way less in tiny-NeRF. So, it does not need a large number of epochs to produce good results. However, that is not the case with NeRF. It may need way more than the number of epochs we were training it for.
- However, we are getting stagnant training loss, which effectively means that the model is not learning anything. We think the reason maybe not be enough data, as the original data is images of 800*800. Given that the original NeRF neural network is very big, resizing the images to 100*100 might have been insufficient.
- A major difference between our model and tiny-NeRF is that they usually consider **one viewpoint as the ground truth image**, for pixels from all images. We considered **individual pixels from each image as the ground truth for each individual ray**. We chose this method over one image as ground truth, in order to avoid any possible overfitting on a single viewpoint, and also, it can lead to poor dimensionalities in areas unexplored by the ground truth. So, during the reshaping of all pixels, from their respective images into a large tensor of (pixels,3), there may have been a mismatch in the correspondence of pixels, which could have led to errors.

D. Iteration 2

From the implementation of Tiny NeRF, we inferred that insufficient data to such a big network should mostly be the reason for stagnant loss over epochs. So, we loaded the original images **(800*800)** again, and have put the model on training on A100 GPU in turing. However, given the large dataset, and the fact that every pixel from every image has to processed for ground truth, it is taking more than 2-3 hours to pass even one epoch, due to which, we would not be able to submit its output.

III. References

References

[1] Michio JP. Nerf-Pytorch — A PyTorch re-implementation of Neural Radiance Fields nerf, 2021.