Project 4: Buildings Built in Minutes SfM and NeRF RBE549

Karter Krueger
Department of Robotics Engineering
Worcester Polytechnic Institute
Worcester, MA 01609
Email: kkrueger2@wpi.edu

Tript Sharma
Department of Robotics Engineering
Worcester Polytechnic Institute
Worcester, MA, 01609
Email: tsharma@wpi.edu

I. Phase 1: Structure from Motion

The first phase of the project was implementing the classic Structure from Motion (SfM) pipeline. SfM is the Computer Vision equivalent to SLAM in Robotics, which is widely used across nearly every robot. The purpose of SfM is typically more focused on photogrammetry applications rather than localization and mapping. The classic SfM pipeline takes in images from several unknown perspectives, detects points, matches them, and uses them to reconstruct 3D features of the scene, as seen in the following steps.

1) Data Parsing (of features and matches)
2) Fundamental Matrix Estimation
3) RANSAC for Fundamental Matrix
4) Essential Matrix Estimation
5) Camera Pose Estimation
6) Triangulation
    a) Linear Triangulation
    b) Disambiguate Camera Pose
    c) Non-linear Triangulation
7) PnP (Perspective-n-Point)
    a) Linear PnP
    b) PnP RANSAC
    c) Non-linear PnP
8) Bundle Adjustment
    a) Build Visibility Matrix
    b) Perform bundle Adjustment

A. Dataset and Parsing

In this project, we will reconstruct the scene of Unity Hall, a robotics building at WPI. To ensure consistent results, features points were previously extracted and matched using SIFT, and were provided in file-format for each image. The files were read using the defined format, which consists of the matches for each image, in list form, with the (u,v) coordinates of the matches from image 1 and then images 2-5 (for example).

B. Fundamental Matrix Estimation

Fundamental Matrix is used to define relationship between a pair of images from two different viewpoints. It incorporates epipolar geometry to describe this relationship. We can obtain point-to-point correspondences from the parsed dataset. Since Fundamental Matrix (F) is a $3x3$ matrix, we have 9 unknowns. With the last element being the normalizing factor, we are left with 8 unknown values. Hence, wew use the 8-point algorithm to compute the Fundamental matrix. using the constraint:

$$x2_i^T.F.x1_i^T = 0 \tag{1}$$

Using the equation, we computed A matrix such that $A.f = 0$, where f represents the elements of F. This is followed by SVD cleanup to contrain F matrix to a rank 2 matrix.

C. RANSAC for Fundamental Matrix

Since, there are noises incorporated because of wrong feature correspondences from SIFT feature matching. We remove outliers by using equation 1 a modification that the value should be less than a given threshold ϵ. After obtaining the best set of inliers, we compute F matrix again with this bbest set of inliers.

Using the Funamental Matrix RANSAC method, we cut the outliers out of the scene, as seen below in the original image and inlier matches image. You can see it cuts down the outliers significantly, plus we now have the F matrix we need for everything else. We also show results on another building dataset.

Fig. 1. Matches on Unity Hall from Image 1 to 4

Fig. 2. Inliers of matches on Unity Hall from Image 1 to 4

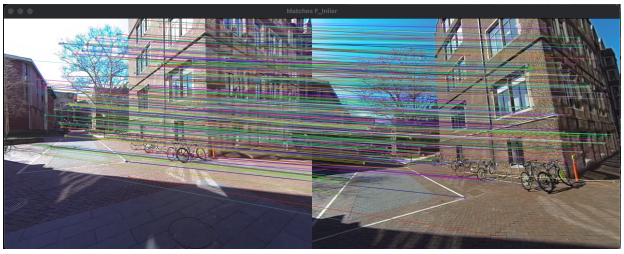

Fig. 3. Inliers of matches on another building image 1 to 4

D. Essential Matrix Estimation

Using the Fundamental Matrix (F) and camera intrinsic matrix (K), we can estimate the Essential matrix using the formula $E = K^T * F * K$. After calculating E, we perform cleanup using SVD to extract the components. We first use SVD to get U, D, and V^T, then set singular values to $D = diag[1, 1, 0]$. Finally, we reconstruct E for the final result, using $E = U * D * V^T$.

E. Camera Pose Estimation

Using the computed E matrix we can estimate camera poses as a combination of rotation and translation as given below:

$$P = KR[I - C] \tag{2}$$

which can be used to compute the four combinations of R and C.

You can see two examples below on camera pose estimation. First, on Unity Hall, and second on another sample building dataset.

F. Triangulation

Now that we have relationship between points in real world, we can use it to identify the 3D world point, X. Since we have the direction and we need the distance from camera to the world point, triangulation best suits the job.

1) Linear Triangulation: The world point projection should be the same as the corresponding point on the image. Using the camera poses and exploiting the mentioned fact we compute the world coordinate, X using the following equation: $xXP * X$. This is used to perform SVD and obtain X. Also, we considered the first camera to be at the origin to compute the triangulation

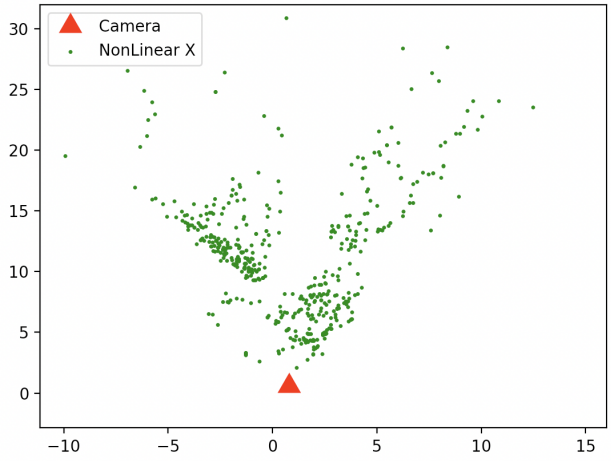

Fig. 4. Unity hall camera pose estimation

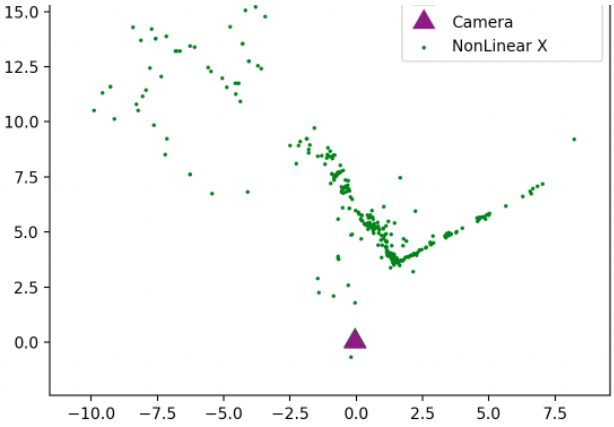

Fig. 5. Other building camera pose estimation

2) Camera Pose Disambiguation: We reject wrong camera poses by using the chirality condition i.e. the coordinate of the predicted world points must be in front of the camera. The chirality condition is given as $r3(X - C) > 0$ where r3 is the third row of the rotation matrix R, X is the world coordinate and C is the Camera origin or the translation vector.

3) Non Linear Triangulation: Once we have found the actual pose of the camera, the next task is to reduce the error between the reprojected world point and the detected image point. Since the linear error reduces algebraic error which is of less significance in the 3D world where a geometric error makes more sense. We compute the non linear error and try to converge the reprojected point and thereby find a more accurate world point. We used `scipy.optimize.least_squares` function for this.

We show results of both linear and non-linear triangulation on the unity and building datasets below. We also show their projection back onto the image plane for comparison as well.

G. Perspective-n-Point (PnP)

After we have computed the optimized world coordinates for two camera frames, we estimated the pose of other four

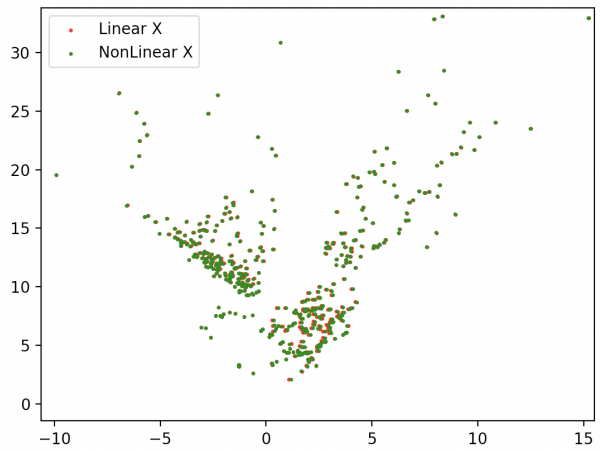

Fig. 6. Unity Hall triangulation

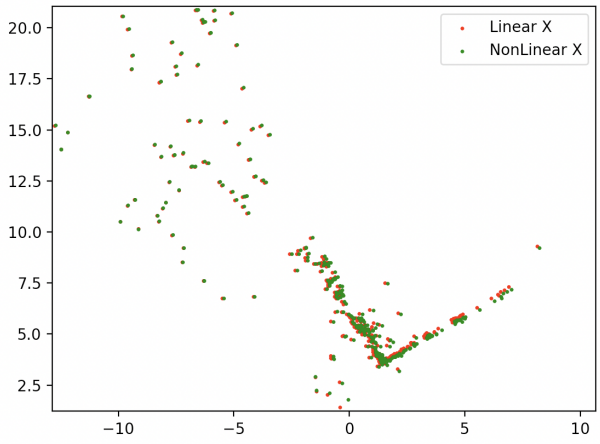

Fig. 7. Other building triangulation

Fig. 8. Unity Hall triangulation projection

cameras using Perspective-n-Points algorithm. We have x, X and K, the remaining unknowns are R and C in the pin-hole camera equation. Since the first image had correspondences

Fig. 9. Other building triangulation projection

with all the other images, it was used to compute the camera poses

1) Linear PnP: We solve a linear least squares equation involving 12 correspondences or 6 point correspondences to compute the new camera pose i.e. R and T. This was used to compute the SVD and extract R and T values from the last row of V^T.

2) PnP RANSAC: PnP is also prone to outliers which can produce erroneous camera poses. Hence, if the reprojection error for the new camera is less than a given threshold ϵ^* the points were added to the inlier set for a more robust camera pose estimation.

3) NonLinear PnP: Similar to triangulation, we compute the geometric loss using `scipy.optimize.least_squares` to further reduce the error and get a more accurate camera pose.

Using the nonlinear PnP method, we get the camera pose for each camera relative to the X points. We display these below on a plot with triangulated points, for Unity hall and the other building dataset.

H. Bundle Adjustment

Now that we have all the camera poses and the world coordinates we can refine the camera poses and the 3D coordinates together. This is called Bundle Adjustment, which is an optimizer to refine the positions of both cameras and points jointly. This is often performed using libraries such as "GTSAM" and others, but for this project, we will implement our own using the Scipy LeastSquare method, as described below.

1) Visibility Matrix: The visibility matrix is constructed when reading in the data files. When reading the "matching" files for each image, we build a matrix of all points (for example, around 10,000 in Unity Hall files) and mark a 1 for each camera column if the point is visible from that camera. This results in a n x m size matrix for n points and m cameras.

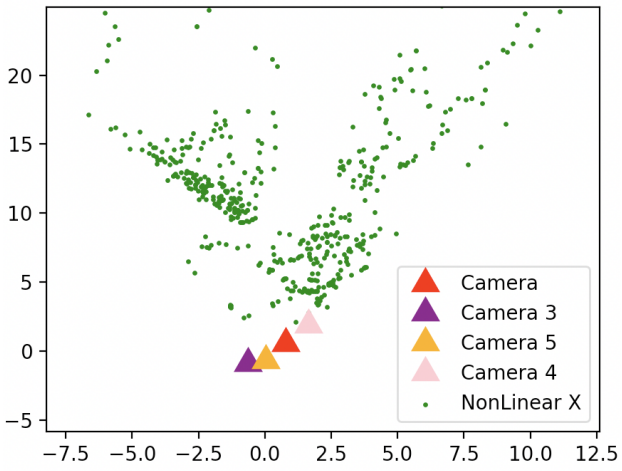

Fig. 10. Unity Hall PnP camera poses

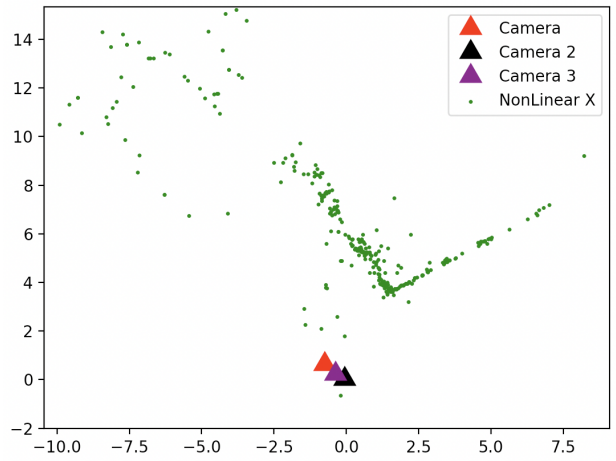

Fig. 11. Other building PnP camera poses

2) Bundle Adjustment: Bundle Adjustment uses the Scipy least_square optimizer like many of the former sections. The purpose of Bundle Adjustment is to take in all the world points X, pixel coordinates x for the coordinates in each image, the camera poses (as translation and quaternion representations of rotation matrix), and a visibility matrix from above. The visiblity matrix significantly reduces the computational requirements by specifying what jacobians must be computed behind the scenes and which can be skipped to run faster.

II. Phase 2: Neural Radiance Field (NeRF)

While SfM has long been around as a reliable approach to reconstruction, there has been recent work in deep learning that builds on the camera poses from SfM to allow for novel view synthesis of scenes. The key method that popularized this new version of novel view synthesis was unveiled in 2020, with a paper called Neural Radiance Fields, commonly known as NeRF. This method takes many images of a scene from known perspectives (translation and rotation of the camera). Using the known camera positions and intrinsic, we cast rays out into 3D space using a pin-hole camera model. We train a deep network to overfit to the scene with a mapping between 3D coordinates and viewing direction, to RGB color and opacity/density values. To aid the network's learning, fourier frequency embeddings of the XYZ coordinates and viewing direction vector are used.

In this project, we implemented NeRF from scratch by following along with the methods outlined the paper. We trained on the classic Lego model synthetic dataset generated using Blender. The steps for training a NeRF are below.

1) Data Loading
2) Network Creation
3) Frequency Embeddings
4) Ray Generation
5) Rendering
6) Training
7) Results and Improvements

A. Data Loading

We load data from the Blender synthetic dataset format from a folder of images and a json file of the corresponding transformation matrix of the camera, which consists of rotation and translation.

B. Network Creation

NeRF is implemented as an MLP (Multi-Layer Perceptron), which stands out compared to many recent image neural networks that use convolutional layers. The MLP in NeRF is comprised of 8 fully-connected layers of 256 nodes each, with ReLU activations in between. The input to the first and 5th layers is the positional encoding (described next) of 60 values. After the 8th layer, the network splits off into a sigma prediction for the density of the point, and then continues with two more fully-connected layers for prediction the RGB values. Before the RGB layers, the viewing direction encoding is concatenated as 24 values to the existing 256 (from the first 8 layers). This branch goes on to one more 256-node layer and finally a 128-node layer before Sigmoid is applied to the 3 RGB output values. A network diagram is shown below (as seen in the NeRF paper).

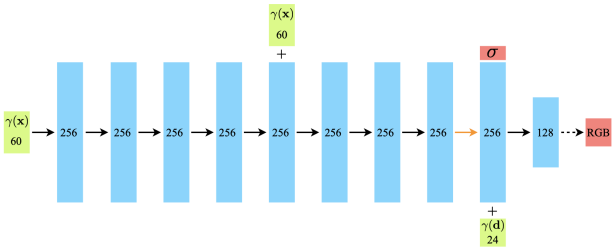

Fig. 12. Diagram of the NeRF MLP network (from the NeRF paper)

C. Frequency Embeddings

In the NeRF paper, they realized that adding frequency encodings, similar to fourier features, helped the performance of the network significantly with the fine-resolution features.

We have implemented this in a similar way, that uses sine and cosine functions, with multipliers that are powers of 2 that vary from 0 to L-1 features. As specified in the NeRF paper, it is suggested to use their tuned hyperparameters of $L = 10$ for XYZ coordinate embeddings and $L = 4$ for the viewing direction embeddings. This results in a total of 60 features for positions (10(L) * 2(functions) * 3(coordintes) = 60), and 24 for viewing directions (4(L) * 2(functions) * 3(values) = 24). These help the network learn higher frequencies due to the similarity to the fourier space where values are represented more in the frequency space. This is compared to a traditional neural network that typically only learns the lower frequencies to reduce loss.

D. Ray Generation

Rays for training are generated for every pixel in every image, where every ray is composed of an origin in XYZ global space, and a direction vector. Using the pin-hole camera model, we cast rays from the camera origin through the center of each pixel on the image plane, based on the focal distance between the image plane and camera origin. This gives us a direction vector for each ray as an XYZ vector. We then rotate the ray direction vectors based on the camera rotation matrix for the global space.

As part of datasets for NeRF, we also specify the near and far distances of the scene for each ray, which signal which section of the ray should be rendered for each ray. In the case of the Lego dataset, the near and far distances are 2 and 6, respectively. We re-parameterize the rays into NDC space by shifting the ray origins to the plane to be "near" units from the camera, and we re-scale the ray length so a ray magnitude of 1 is now equal to the "far" distane from the camera.

E. Rendering

Rendering of a scene is critical for training and testing of the NeRF. We first generate 3D points along each ray using an evenly-spaced distribution of 64 points (for example), along the ray from 0 to 1 (in NDC space, from near to far in the scene). Once we have 3D points, we pass them through the frequency embedder (explained in a previous section) to get 60 values, or 20 encodings per coordinate of the XYZ values. We also pass the normalized viewing direction vectors of the rays through a frequency embedder to get 24 values per point, or 8 per value of the XYZ viewing direction vectors.

With the above frequency embeddings of positions and viewing directions, we pass these through the nerual network, which outputs RGB and density values for each input point. Using the densities for each points, we generate "weights" using the rendering equation of: weights = alpha * cumulativeProduct(1 - alpha). This formula is used to give the highest values and importance to high weights that occur early on along the ray. This means solid objects closest to the camera will contribute the most color, and prevent occluded solid objects from being rendered. This formulation also allows solid objects behind translucent objects to still be rendered as they contribute to the weighted sum of RGB values.

In the NeRF paper, the rendering continues after the coarse points, by generating new "fine" points along ray that are distributed mostly around the objects of interest (with high weights values) rather than the open spaces. This new distribution of fine points is generated using an inverse CDF probabilistic formulation (which is also common to other domains such as particle-filters, as learned in office hours with professor Sanket). The new fine points are then encoded as before and passed through the fine network (rather than the coarse network used the first time). The fine outputs of the network are then rendered as the alpha-weighted sum of the RGB values as before. This results in the final rendered pixel values for each ray. We did not implement the fine network in our implementation in order to keep it simpler for time purposes as we learned, but we still spent significant time learning about how this would be implemented and gave it a try, but without success in the final rendering results.

F. Training

To train the NeRF, we sample a random photo from the dataset during each epoch and take a collection of 6500 random rays (the largest that fit in GPU memory) from the image. The list of rays are then rendered using the above weighted-sum rendering method. The resulting ray "pixel" values are then compared against the original pixel values of the training data. This difference is used in a Mean-Squared-Error loss formulation for the network to learn on. We used an Adam optimizer with a learning rate of $1e-5$ to train for up to 200,000 epochs. During the training process, we encountered a few issues such as the gradients being 0s due to a bug in the rendering, which we debugged through. In the end, we ended up also limiting the dataset to smaller resized versions of the original images and a crop of only the center of the image, to allow the network to train easier and faster.

G. Results and Improvements

As explained in the training section we trained on a resized version of the dataset and a crop on the center of the image to allow for faster and simpler training. The results are shown below with a comparison to the origin (un-resized, un-cropped) image from the training dataset. You will notice that only a yellow blob is shown, which is what was expected for the resized version of the dataset with only a coarse network in use (compared to both coarse and fine), and limited training time of 20,000 epochs for this example. Our rendering method pass the rays of a target image into the network and filled in the corresponding pixel values, very similar to the rendering method used during training.

While these results aren't pretty like the graphics in the paper, this still shows that our network was doing many things right and starting to render the correct color (in yellow blob form) of the Lego model with some structure starting to appear. We believe with more time to tweak and train the model, we would eventually get the same impressive visual results as the paper, as we continued to discover many of the code tricks to get there. Regardless of the quality of our result,

we had a phenomenal learning experience as we learned the math behind volume rendering and how the NeRF network is formulated with position encodings.

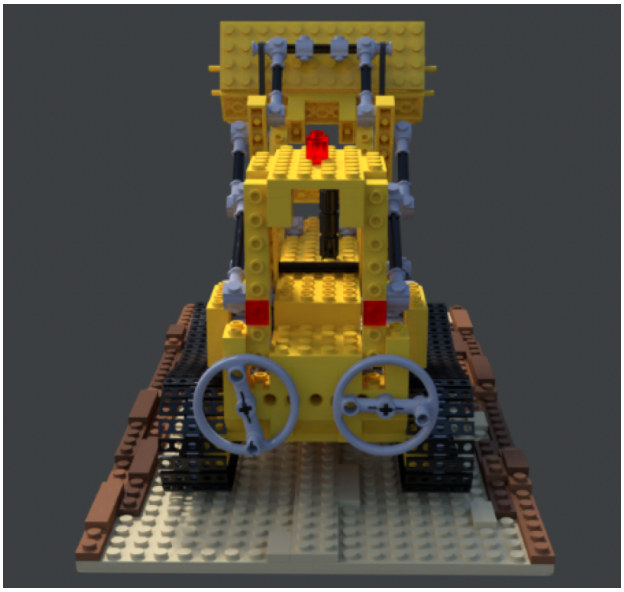

Fig. 13. Original training image for NeRF

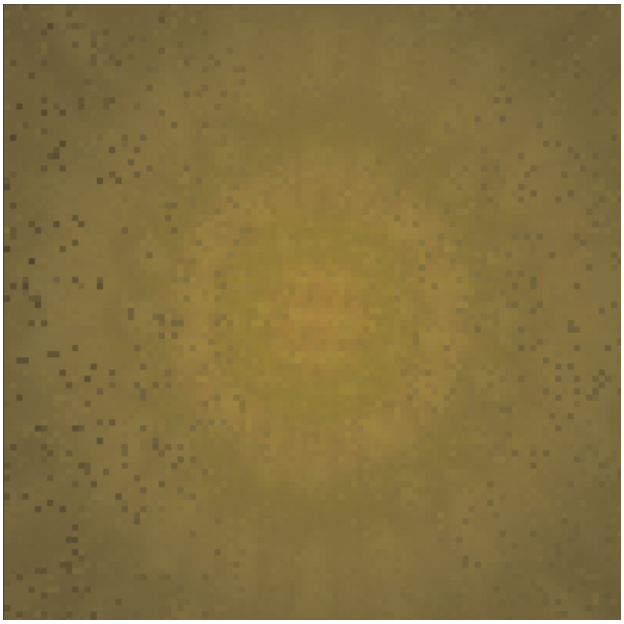

Fig. 14. Our simplified, cropped, down-sized, render