Re-Implementation of Camera Preconditioning for Neural Radiance Fields in Nerfstudio

Roshan Roy, Sahil Jain, Bhuvan Jhamb

March 30, 2025

1 Introduction

Radiance field methods like NeRFs and 3D Gaussian splatting typically rely on precise camera parameters and optimize over 3D points, with camera poses often precomputed using Structure-from-Motion (SfM) tools like COLMAP. However, accurate camera poses are not always available—particularly in casually captured scenes or wide-baseline views—where COLMAP may struggle to converge. Alternatively, if rough camera poses are available, one might consider bypassing the COLMAP pipeline to directly refine the poses, as COLMAP tends to be relatively slow. Thus, there is some interest in joint optimization of scene representation and camera parameters.

This joint optimization in NeRFs or Gaussian splatting is challenging due to its ill-conditioned nature and susceptibility to local minima. Furthermore, the choice of camera parameterization also affects the optimization process. To improve optimization, CamP [1] studied the effect of different camera parameterizations and introduced a promising approach using preconditioning. As the original implementation is in JAX, our objective is to port this technique to PyTorch and integrate it with Nerfstudio, making it more accessible to the broader community.

More concretely, our goal is to:

- Implement additional camera parameterizations (as introduced in CamP [1]) into Nerfstudio, making it accessible to all NeRF models.
- Study the effect of different camera parameterizations on train and test-time camera optimization.
- Implement camera preconditioning (inspired by the JAX implementation for CamP [1]) in PyTorch and integrate it with Nerfstudio.
- Reasonably reproduce the experiments section outlined in the original paper to test the validity of our implementation.
- Submit a clean, organized and documented pull request to Nerfstudio with these changes.

This report is organized as follows: Section 2 provides an overview of camera optimization in neural radiance fields and preconditioning. Section 3 outlines the enhancements made to Nerfstudio for a successful implementation, followed by the experimental results in Section 4. Finally, Section 5 concludes the report and discusses potential future work.

2 Background

2.1 Optimization problem

Given a set of n images $\{I_i\}$ and initial camera parameters $\boldsymbol{\phi}_0^i$ for $i \in \{1, \ldots, n\}$, we aim to derive a scene model M_θ with optimized NeRF parameters θ^* and camera parameters $\boldsymbol{\phi}_i^*$ that accurately reproduce the observed images.

The training objective is to minimize a loss function $L(D; \theta, \boldsymbol{\phi})$ over a dataset D containing all observed pixels, where each pixel j is associated with a camera index $d_j \in \{1, \ldots, n\}$, a location $\mathbf{p}_j$, and an RGB color $\mathbf{c}_j$.

2.2 Camera Parameterization

A camera parameterization defines a projection function $\Pi(\mathbf{x}; \boldsymbol{\phi}) : \mathbb{R}^3 \rightarrow \mathbb{R}^2$ that maps a 3D point $\mathbf{x}$ to a 2D pixel location $\mathbf{p}$. Various parameterizations have different sensitivities to the objective function, impacting the optimization landscape and reconstruction quality.

CamP analyzes the effects of several parameterizations, summarized below:

- **Intrinsics**: Used to represent camera intrinsics parameters. Focal length has a multiplicative residual and is updated as $f' = f \exp(\Delta f)$. Principal points have additive residuals are updated as $u^{'} = u_0 + \Delta u^{'}$ and $v^{'} = v_0 + \Delta v^{'}$. Two second-order radial distortion coefficients also have additive residuals are updated as $k_1^{'} = k_1 + \Delta k_1^{'}$ and $k_2^{'} = k_2 + \Delta k_2^{'}$.
- **SE3**: A $\mathfrak{se}(3)$ screw-axis parameterization $S = (r; v) \in R^6$ for rigid pose of the camera; similar to the parametrization used in BARF. The additive residual ΔS updates the current estimate S_0 as $S^{'} = S_0 + \Delta S^{'}$ [2].
- **SO3xR3**: A 6D vector where the first three parameters represent a standard $\mathfrak{so}(3)$ rotation, and last three numbers represent camera translation.
- **FocalPose+Intrinsics**: This camera parametrization uses SO3 updates for rotation, and joint translation and focal length updates as introduced in FocalPose[3]. We use residual updates in place of sequential updates from a feed-forward network. Focal length updates as $f' = f \exp(\Delta f)$, camera translation parameters update as $x^{'} = ((\Delta x/f) + (x/z)) \cdot z^{'}, y^{'} = ((\Delta y/f) + (y/z)) \cdot z^{'}$ and $z' = z \exp(\Delta z)$.
- **SCNeRF [4]**: A 6D continuous vector for rotation[5] where the first and last three elements represent the two orthogonal axes of rotation. When transformed to a valid 3×3 rotation matrix, gimbal lock is avoided. 3D translation parameters (t_x, t_y, t_z), principal points (u_0, v_0) and radial distortion coefficients (k_1, k_2)have additive residuals. In this parametrization, focal lengths (f_u, f_v) also have additive residuals. Additive updates are identical to previous parametrizations.

We aim to reproduce the results of CamP [1] on these parametrizations.

2.3 Camera Preconditioning

When optimizing camera parameters, discrepancies in parameter scales can hinder the process. For instance, focal lengths are typically expressed in pixel units, while translation is measured in world units, which can introduce scale imbalances. Additionally, depending on how the parameters are defined, translation and rotation can be highly correlated, complicating optimization and degrading results. To address these issues, a preconditioning matrix can be applied, which rescales and decouples the parameters, transforming the optimization problem into a more well-conditioned one for improved convergence.

To develop a suitable preconditoner, [1] studies the sensitivity of the camera-projection matrix $\Pi(x; \phi)$ to the camera parameters ϕ. Instead of looking at $\Pi(\phi)$, a modified version of the projection function $\Pi^m(\phi) : \mathbb{R}^k \rightarrow \mathbb{R}^{2m}$ is studied which is the concatenation of the projection of m scene points. The sensitivity of the projected points on the camera parameters can be calculated as $\mathbf{J}_\Pi = \frac{d\Pi^m}{d\phi}$ where $\mathbf{J}_\Pi \in \mathbb{R}^{2m \times k}$. The matrix $\Sigma_\Pi = \mathbf{J}_\Pi^T \mathbf{J}_\Pi \in \mathbb{R}^{k \times k}$ gives us the correlation between the different camera parameters.

The goal is to find a preconditioning matrix $\mathbf{P}$ such that substituting ϕ with $\mathbf{P}^{-1}\tilde{\phi}$ in $\Pi^m(\mathbf{P}^{-1}\tilde{\phi}) = \tilde{\Pi}^m(\tilde{\phi})$ gives us a $\Sigma_{\tilde{\Pi}}$ which is equal to the identity matrix. The authors choose $\mathbf{P}^{-1} = \Sigma_\Pi^{-1/2}$ to satisfy this condition.

This preconditioner is optimal only with respect to the initial estimate of the camera parameters ϕ^0 as the camera parameter estimates deviate from this value. However in practice, this still works well.

3 Methodology

In this section, we go over the requirements, functional design and implementation details of the features required for this project. We specify which features are already available in Nerfstudio and which we had to add.

3.1 Requirements

To enable the joint training of NeRFs and camera parameters, the following features are necessary:

1. Camera optimization: A mechanism to jointly update the camera parameters, alongside scene parameters during training and testing based on a configurable parameterization scheme (outlined in the previous section). Currently, Nerfstudio only supports optimization of camera extrinsics for two parameterizations (`SE3` and `SO3xR3`). It does not have any support for intrinsics optimization.

2. Preconditioning: A class to compute the preconditioning matrix based on the chosen parametrization scheme, at the start of training. There must be a way to apply the preconditioning matrix to the camera deltas during optimization. Care must be taken to either optimize over the new preconditioned parameter space or recover the original parameters, prior to gradient backpropagation during training. The class must be able to configure the cadence at which the preconditioning matrix is recalculated during training. Currently, Nerfstudio does not have any support for preconditioning.

3. Controlling noise addition: Camera optimization is known to be a challenging optimization problem when camera parameters are already near-perfect. The effect of optimization is more pronounced when the initial camera parameters have an appreciable degree of noise. In order to effectively validate the camera optimization implementation, we need a method to inject noise (with a configurable noise distribution) into ground-truth camera parameters. This noise injection is a well-documented feature in many open-source camera optimization pipelines but is noticeably absent in Nerfstudio.

4. Test-time optimization: A framework for test-time optimization of camera parameters to evaluate image quality metrics effectively. Crucially, this is also not currently implemented in Nerfstudio. This limitation can negatively impact quantitative metrics such as PSNR, even if the qualitative results appear improved.

The discrepancy arises because both training and test camera poses are derived from COLMAP and are not ground truth. While training–time optimization adjusts the camera poses to a distribution better suited for the model, test–time camera poses remain unoptimized, leading to inconsistencies.

To address this, we introduce a test-time camera optimizer. This optimizer refines camera poses during test time, running for 500 iterations to align them with the optimized distribution from training. This ensures improved consistency and performance across both training and test stages.

3.2 Functional Design

All our changes towards camera optimization must conform to Nerfstudio API in order to make it usable across all NeRF implementations. Additionally, it is important for all the parameters to be easily configurable using command line arguments. Keeping this in mind, we implement the following interfaces for each feature.

1. Camera optimization: Camera optimization can be easily controlled by using the command line argument to either disable camera optimization or select a desired parameterization. We add the other options.

```
--pipeline.camera-optimizer.mode::  str, ["off", "SO3xR3", "SE3", "SCNeRF", "FocalPose"]
```

2. Camera preconditioning: Camera preconditioning parameters control the sampling scheme of 3D points used to compute jacobian $\mathbf{J}_\Pi$. They also set the padding values used in computation of preconditioning matrix from $\mathbf{J}^T\mathbf{J}$.

```
--pipeline.camera-optimizer.use_preconditioning::  bool,
--pipeline.camera-optimizer.preconditioning_num_points::  bool,
--pipeline.camera-optimizer.preconditioning_depth_near::  float,
--pipeline.camera-optimizer.preconditioning_depth_far::  float,
--pipeline.camera-optimizer.preconditioning_diag_abs_padding::  float,
--pipeline.camera-optimizer.preconditioning_diag_rel_padding::  float,
```

3. Controlling noise addition: To add noise to the camera parameters including rotation, translation and focal length, we opt for the command line arguments

```
--pipeline.camera-optimizer.focal_noise_std::  float,
--pipeline.camera-optimizer.rotation_noise_std::  float,
--pipeline.camera-optimizer.translation_noise_std::  float
```

4. Test-time optimization: To enable test-time optimization of camera parameters, we add two command arguments to Nerfstudio letting users control whether to enable test-time optimization, and how many iterations of test-time optimization to perform:

```
--pipeline.eval-optimize-cameras::  bool
--pipeline.eval-num-iters::  int
```

3.3 Implementation Details

For implementing the required features, we reuse the existing Nerfstudio pipeline wherever possible to allow for a smooth merging process at the end. At a high level, camera optimization is implemented as a PyTorch layer. This layer accepts a ray bundle generated using the original camera parameters and outputs an updated ray bundle after applying pose and intrinsics corrections.

1. Camera optimization: For pose optimization, we adopt the approach implemented in Nerfstudio. Specifically, we learn delta adjustments for each camera parameter, which are initialized to zero. For poses, these deltas are directly applied to the original parameters, with the rotation updated as $R_{\text{new}} = \Delta R \cdot R_{\text{orig}}$ and the translation updated as $T_{\text{new}} = \Delta T + T_{\text{orig}}$. This formulation ensures that the optimization process refines the camera poses incrementally while preserving consistency with the original configuration. Optimizing intrinsics is a bit more covoluted.

Nerfstudio does not currently support the optimization of camera intrinsics. To address this limitation, we have implemented functionality to optimize camera intrinsics as part of the framework.

Nerfstudio maintains a ray bundle for each camera, and all optimization operations are performed on this ray bundle. Consequently, updating the camera intrinsics after each iteration requires updating the corresponding ray bundle.

To update the ray bundle using the optimized intrinsics, the following steps are performed after each iteration:

- Compute the pixel coordinates for each ray.
- Generate new rays using the updated intrinsics, translation and rotation

2. Preconditioning: For camera preconditioning, we adopt a similar approach as outlined in CamP [1]. First, we calculate the preconditioning matrix P for each camera in the scene, as described in 2.3. In practice, a regularization term $\lambda\,\text{diag}(\Sigma_\Pi) + \mu I$ is added to Σ_Π before finding its inverse square root $\Sigma_\Pi^{-1/2}$. Here, λ and μ are hyperparameters. The choice of camera parametrization affects the shape of P. During each training iteration, we first use P to transform the `CameraDelta` u as $u^{'} = P \cdot u$ in the preconditioned space. We ensure that all gradient updates to $u^{'}$ occur in the preconditioned space, precluding the need to revert to the original space with P^{-1}. As in intrinsics optimization, focal lengths have multiplicative updates $f_x^{'} = f_x \cdot \exp\left(u_{f_x}^{'}\right)$ and principal points have additive updates $c_x^{'} = c_x + u_{c_x}^{'}$. The authors find no major improvement in re-calculating the P matrix at a pre-scheduled cadence, and so we avoid implementing EMA updates to P.

The major differences in our PyTorch re-implementation of the JAX code are:

- **Vectorization:** JAX `vmap` natively supports function vectorization. Since this is unavailable in Pytorch, we create a vectorized implementation for calculation of P over a batch of cameras.
- **3D Operations:** PyTorch, unlike PyTorch3D, does not have standard library functions for perspective projection and unprojection, so we implement them from scratch.
- **Assertion testing:** We enforce several geometry-based assertions within the preconditioner class to help future contributors make bug-free updates to the preconditioning logic.

3. Controlling noise addition: Pose noise is applied when `position_noise_std` or `orientation_noise_std` is non-zero, generating 6D Gaussian noise (for position and orientation) for each camera and mapping it to SE(3) space. Similarly, focal length noise is introduced when `focal_length_noise_std` is non-zero, generating 2D Gaussian noise in logarithmic space to perturb the focal lengths. When the noise standard deviations are zero, no noise is added.

4. Test-time optimization: We create a new instance of the PyTorch camera optimizer layer for test-time optimization. Before evaluating the metrics, this layer is trained for as many iterations as specified by the user by keeping the NeRF model fixed.

4 Results

Datasets To evaluate our implementation, we follow the approach outlined in the CamP paper [1]. We skip evaluation on synthetic datasets and instead evaluate on the real-world MipNerf360 [6] dataset. Since the camera parameters provided by COLMAP for this dataset are already very accurate, we use a perturbed version of the cameras. We perturb the rotation and translation with $\mathcal{N}(0, 0.005)$. We also perturb the focal length with a random scale $\exp\mathcal{N}(0, log1.02)$. We set the initial distortion parameters to zero. We use seven scenes from the dataset.

Evaluation For the scene representation, we use Nerfacto [7], the default NeRF model provided by Nerfstudio. This representation incorporates several aspects from other works such as hash grids from InstantNGP [8] and anti-aliasing from MipNerf360 [6] and is quick to train. For quantitative evaluation, we use the standard image quality metrics - PSNR, SSIM and LPIPS. We optimize the test cameras for 500 iterations while keeping the scene representation fixed. The parameterization at test-time is chosen to be the same as the one used during training.

From Table 1, we can see that enabling preconditioning helps improve the results. We can also observe that the SCNeRF parameterization of a 6D rotation and 3D pose performs worse than the others. This is in line with the results in the original CamP paper and can also be observed in Figure 1, where the bush on the left is blurrier and lacks detail for SCNeRF as compared to the other parameterizations.

Camera Param.	**CamP**	**PSNR↑**	**SSIM↑**	**LPIPS↓**
No camera opt.		17.792	0.456	0.663
SCNeRF		25.645	0.724	0.270
SCNeRF	✓	26.301	0.773	0.256
SE3+Focal+Intrinsics		26.060	0.740	0.255
SE3+Focal+Intrinsics	✓	26.790	0.852	0.243
SO3xR3+Focal+Intrinsics		25.997	0.736	0.259
SO3xR3+Focal+Intrinsics	✓	26.787	0.793	0.247
FocalPose+Intrinsics		25.867	0.731	0.269
FocalPose+Intrinsics	✓	26.425	0.798	0.251

Table 1: Performance on MipNerf360 perturbed dataset

5 Conclusion and Future Work

We have introduced several improvements to enhance the optimization characteristics of joint camera pose and scene estimation. Specifically, we have added:

- Intrinsics and distortion optimization
- Test-time camera optimization
- A preconditioner as proposed in CamP
- Two new camera parameterizations

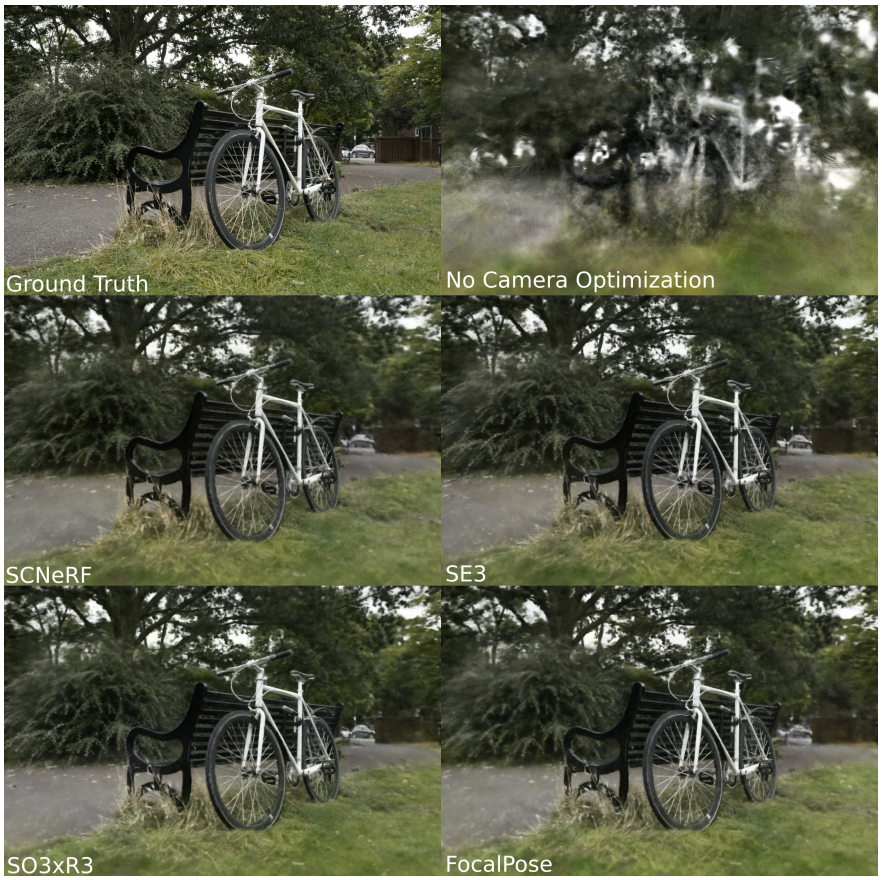

Figure 1: Performance on "bicycle" scene of MipNerf360 across various parameterizations

Currently, we are cleaning up the codebase to submit a pull request to Nerfstudio. Other than preconditioning, we believe a few key features that we have implemented like intrinsics optimization and test-time optimization should be natively supported in a widely used package like Nerfstudio. An exciting direction for future work would be to extend preconditioning techniques and the parameterization experiments to 3D Gaussian splatting, as these methods are independent of the choice of 3D scene representation.

References

[1] Philipp; Mildenhall Ben; Barron Jonathan T.; Martin-Brualla Ricardo Park, Keunhong; Henzler. Camp: Camera preconditioning for neural radiance fields. *ACM Trans. Graph.*, 2023.

[2] Chen-Hsuan Lin, Wei-Chiu Ma, Antonio Torralba, and Simon Lucey. Barf: Bundle-adjusting neural radiance fields, 2021.

[3] Georgy Ponimatkin, Yann Labbé, Bryan Russell, Mathieu Aubry, and Josef Sivic. Focal length and object pose estimation via render and compare, 2022.

[4] Yoonwoo Jeong, Seokjun Ahn, Christopher Choy, Animashree Anandkumar, Minsu Cho, and Jaesik Park. Self-calibrating neural radiance fields, 2021.

[5] Yi Zhou, Connelly Barnes, Jingwan Lu, Jimei Yang, and Hao Li. On the continuity of rotation representations in neural networks, 2020.

[6] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. *CVPR*, 2022.

[7] Matthew Tancik, Ethan Weber, Evonne Ng, Ruilong Li, Brent Yi, Terrance Wang, Alexander Kristoffersen, Jake Austin, Kamyar Salahi, Abhik Ahuja, David Mcallister, Justin Kerr, and Angjoo Kanazawa. Nerfstudio: A modular framework for neural radiance field development. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Proceedings*, SIGGRAPH '23, page 1–12. ACM, July 2023.

[8] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.*, 41(4):102:1–102:15, July 2022.