Robust Line feature based Visual Odometry

Anonymous CVPR 2021 submission

Paper ID 2021

Abstract

Point feature based camera motion estimation fails under certain cases of varying illumination and featureless environments. To overcome this, we implement line feature based odometry estimation as proposed by Yan Lu and Dezhen Song in their paper titled, "Robustness to Lighting Variations: An RGB-D Indoor Visual Odometry Using Line Segments", from IROS 2015. The key idea is to robustly estimate 3D line segments in the scene considering covariance of line detection, sampling and reprojection. Ransac with Mahalanobis error metric along with a Levenberg-Marquardt optimization for motion refinement gives good camera pose estimates. We experiment with the TUM fr1 xyz dataset and report average relative pose errors of around 6 cms in translation and 3.5 degrees in rotation.

1. Introduction

The standard approach to visual odometry(VO) uses point correspondences between consecutive frames. However, this approach fails under a few circumstances. Texture less environments such as long hallways tend to be challenging for feature detectors. In inconsistent lighting conditions, feature matching tends to break down due to difference in illumination of the feature patches. The paper that we referred to suggested an example scenario where the number of point features are significantly low as compared to line matches as shown in Fig. 1. We include the same for context. However, it has been observed that these conditions are rich in geometric cues such as lines. One solution to the previously mentioned problems related to different conditions is to use line features in the scene. Line features are detected using edge detection which is more robust to illumination changes. But the 3D lines obtained from image pairs are noisy due to sampling and reprojection errors. If handled correctly, line segments can be used to analytically solve for camera motion between consecutive image frames.

In this project, we aimed at recovering relative pose between RGBD frames for such scenes utilizing 3D lines estimated in the environment. We also modelled noise in the depth map as a gaussian distribution per point and considered it in our estimation process. We tested our system on the TUM-FR1 XYZ dataset.

Figure 1. SIFT keypoints 97 in left image, 58 in right image while only 1 match. Hough lines 112 in left, 94 in right and 12 matches from Line Matching

2. Problem Formulation

Given a pair of images I_1 and I_2 and their corresponding depth images D_1 and D_2 , estimate the rotation R and translation t of the camera between the pair of images.

3. Methodology

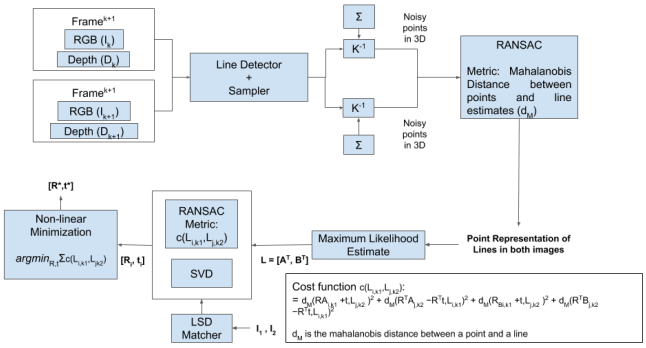

Figure 2. roblineVO pipeline

Our implementation follows many parts of the conventional visual odometry pipeline. An outline of our pipeline

is shown in Fig. 2. We perform feature extraction and matching but in the line space. Instead of estimating F matrix for motion estimation, we perform outlier rejection at multiple stages. Firstly to remove outlier points from line segments and secondly to remove false line matches between images. Given 3D line correspondences, we use an SVD based algorithm as suggested by the original paper to estimate the rotation and translation.

Our implementation can be classified into the following 9 modules:

1. Line detection and matching
2. 2D sampling
3. Reprojection using K
4. Covariance propagation
5. Outlier 3D point rejection using RANSAC
6. MLE for 3D line estimation
7. Motion estimation using RANSAC
8. Ransac for removing false line matches
9. Optimization for R and t.

We make a few assumptions about the system at this stage. We assume the input data frame is a calibrated RGB-D image, i.e, we know the intrinsics of the camera. Moreover, we also assume that we know the noise/uncertainty model of the depth measurement. In our experiments using the TUM dataset, the data is from a Kinect sensor which has a quadratic depth noise model. We also take for granted that the RGB and depth image are registered onto one another. This is done using a script called associate.py in our codebase [provided by TUM dataset]. In the upcoming sections, we will dive deeper into implementation details of each module.

3.1. Line Detection and matching

The first step in the pipeline is the line detection and matching step. We use the line segment detector[2] to detect lines in the image. The length of the detected lines is constrained to be above a minimum value of 25 pixels. The line band descriptor is used to match the lines between two frames. The lines which do not have a match in the conjugate image are rejected at this stage.

3.2. 2D sampling

To keep the computation within bounds, lines were sampled such that there were a maximum of 100 points in a line. The length of a line in pixels was considered while sampling points. For lengths less than 100, we sample using the Bresenham algorithm[1]. Bresenham samples uniformly and finds the integer pixels that connect endpoints. This was preferred over a simple slope-intercept interpolation as there's no floating point division in this method, resulting in faster sampling. However, for line lengths greater than 100 pixels, we sample 100 points using slope-intercept form of a 2D line and sample at uniform distances. Example of the sampling algorithm is shown in Fig. 3 and 4

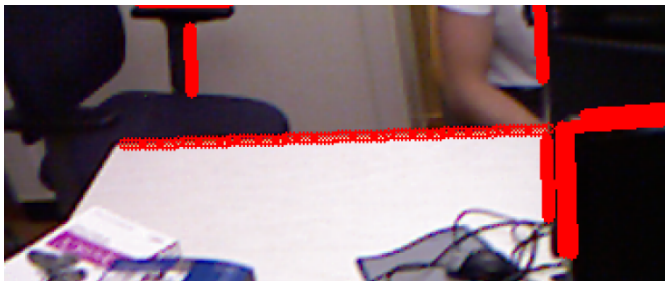

Figure 3. Long lines sampled along edge of table

Figure 4. Long and short lines sampled on the keyboard

3.3. Reprojection using K

The sampled points in 2D are reprojected in 3D using the camera matrix K. The obtained 3D position of the point can be represented as,

$$G_j = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} (u_j - c_u)/f_x \\ (v_j - c_v)/f_y \\ d_j \end{bmatrix} \quad (1)$$

where u_j and v_j represent the pixel coordinates of the sampled 2D points, f_x and f_y are focal lengths along the respective axes and $d_j$is the depth of the point. The points whose depth value is not defined in the corresponding positions in the depthmaps are not reprojected in 3D. Hence, even after sampling a fixed number of points for each line, the number of reprojected points per line may not be the same.

3.4. Covariance propagation

Errors could've been introduced in the line detection phase or the sampling phase. Due to which we can assign a

covariance to the sampled 2D point on a line. The covariance can be represented by Σ_{2D} given by,

$$\Sigma_{2D} = \begin{bmatrix} \sigma_x^2 & 0 & 0 \\ 0 & \sigma_y^2 & 0 \\ 0 & 0 & \sigma_d^2 \end{bmatrix} \tag{2}$$

where σ_x and σ_y represents the standard deviation of the sampled pixel in the x and y directions in the image, σ_d represents the deviation of the depth value measured by the depth sensor at the sampled pixel. The TUM dataset was obtained using a Kinect. According to [5], the depth accuracy is 1.5 mm at 50 cm and about 5 cm at 5 m. This can be modelled as a quadratic dependence on the measured depth value.

$$\sigma_d = c_1 d^2 + c_2 d^3 + c_3 \tag{3}$$

where d is the value of the measured depth in m. As suggested by [4], we use c_1 = 0.00273, c_2 = 0.00074, c_3 = 0.00058.

In short, covariance propagation can be explained by the following equations in which **X** [n × 1] is a multi-variate random variable from the Gaussian distribution $N(\mu_x, \Sigma_x)$, **F** is a matrix [m × n], **t** [n × 1] is a deterministic vector (not a random variable i.e, no covariance/uncertainty).

$$\begin{aligned} X + t &\sim N(\mu_x + t, \Sigma_x) \\ FX &\sim N(F\mu_x, F\Sigma_x F^T) \\ FX + t &\sim N(F\mu_x + t, F\Sigma_x F^T) \end{aligned} \tag{4}$$

The reprojection function maps u[x-coordinate of 2D pixel],v[y-coordinate of 2D pixel],d[depth] to the 3D coordinate X, Y, Z. In order to propagate the covariance of u,v,d to 3D, we can use a first order approximation of the non-linear reprojection function.

$$\begin{aligned} G(u,v,d) &= \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \\ &= \begin{bmatrix} (u - c_x)d/f_x \\ (v - c_y)d/f_y \\ d \end{bmatrix} \\ &= G(u_j, v_j, d_j) + J_j \begin{bmatrix} u - u_j \\ v - v_j \\ d - d_j \end{bmatrix} \end{aligned} \tag{5}$$

where u_j, v_j, d_j represent the 2D coordinate and depth at the j-th pixel abount which we are linearizing. J_j is the jacobian from linearization at the j-th pixel. Using the covariance propagation rule III, we get

$$\begin{aligned} \Sigma_{3D} &= covariance(\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}) \\ &= J_j \Sigma_{2D} J_j^T \end{aligned}$$

$$J_j = \frac{\partial G}{\partial (u,v,d)}|_{u_j,v_j,d_j} = \begin{bmatrix} d_j/f_x & 0 & (u_j - c_x)/f_x \\ 0 & d_j/f_y & (v_j - c_y)/f_y \\ 0 & 0 & 1 \end{bmatrix} \tag{6}$$

In our implementation, we experimented with $\sigma_x = \sigma_y$ = 1px. We're not quite sure if the depth noise modelling is perfectly transferable to TUM dataset and is open for future investigation.

3.5. Outlier 3D points rejection using Using RANSAC

The reprojected points in 3D may or may not belong to an actual line in 3D. Even if they represent an actual line, there may be outlying points due to errors in reprojection. To eliminate these outliers, RANSAC is applied. The mahalanobis distance is used as the metric to qualify a point as an inlier. More specifically, if P is a 3D point and Σ be the covariance of the point and let a line be defined by its end coordinates (A, B) , then the mahalanobis distance between the point and the line can be defined as,

$$d_M(P, L) = \frac{\|A' \times B'\|}{\|A' - B'\|} \tag{7}$$

where, A' and B'can be obtained after applying SVD to the covariance of point P .

$$\Sigma = UDU^T$$

$$A' = D^{\frac{-1}{2}} U^T (A - X)$$

$$B' = D^{\frac{-1}{2}} U^T (B - X)$$

3.6. Maximum likelihood estimate for 3D lines

There's no guarantee that the inlier points coming from RANSAC for each line actually lies on the line defined by the endpoints of the line segments. This is because the inlier metric used in RANSAC is the mahalanobis distance between a point and a line and there is a threshold up to which a point passes through as a support for a particular line hypothesis. Because of this threshold, there's more room to refine the line points to ensure we get the best line estimate.

To do this for one line, the line is parameterized by its end points and scalar λ_is to denote intermediate points as shown in Eq. 8. **p** is the vector of parameters that are to

be optimized which includes the 6 coordinates of endpoints, and the n-2 intermediate points parameterized by scalar λ_is.

$$h(p) = \begin{bmatrix} \hat{G}_{1x} \\ \hat{G}_{1y} \\ \hat{G}_{1z} \\ (\lambda_2)\hat{\mathbf{G}}_\mathbf{1} + (1-\lambda_2)\hat{\mathbf{G}}_\mathbf{n} \\ (\lambda_3)\hat{\mathbf{G}}_\mathbf{1} + (1-\lambda_3)\hat{\mathbf{G}}_\mathbf{n} \\ \vdots \\ (\lambda_{n-1})\hat{\mathbf{G}}_\mathbf{1} + (1-\lambda_{n-1})\hat{\mathbf{G}}_\mathbf{n} \\ \hat{G}_{nx} \\ \hat{G}_{ny} \\ \hat{G}_{nz} \end{bmatrix} \tag{8}$$

Theoretically, the parameter vector can have up to 104 values in it which would correspond to all 100 sampled points passing through RANSAC as inliers and endpoints (6) + intermediate parameterized points (98) would make up the $\mathbf{p}$ vector.

We can minimize the error between this estimated line and actual reprojected 3D points taking into consideration how confident we are of the reprojected 3D point. This would form a weighted least squares problem where the weights of the errors would be from the inverse of the propagated covariance for each 3D point. Note that the parameterization($\mathbf{p_i}$) is different for endpoints as compared to intermediate points. So for every 3D sampled point ($\mathbf{X_i}$) in the line that came out as an inlier from RANSAC, we'd like to minimize Eq. 9

$$\min_{\mathbf{p_i}} \sum_i (\mathbf{X_i} - \mathbf{h}(\mathbf{p_i}))^T \Sigma^{-1}_{3D_{X_i}} (\mathbf{X_i} - \mathbf{h}(\mathbf{p_i})) \tag{9}$$

This can be minimized using the Levenberg-Marquardt algorithm which is used to minimize functions of the form shown in Eq. 10

$$\begin{aligned} cost &= \frac{1}{c_1^2}|f_1(\mathbf{x})|^2 + \frac{1}{c_2^2}|f_2(\mathbf{x})|^2 + \ldots + \frac{1}{c_n^2}|f_n(\mathbf{x})|^2 \\ &= \mathbf{f^T} diag(\mathbf{W})\mathbf{f} \\ &= (\mathbf{W}^{\frac{-1}{2}}\mathbf{f})^T(\mathbf{W}^{\frac{-1}{2}}\mathbf{f}) \end{aligned} \tag{10}$$

Our problem with Eq. 9 is the inverse covariance is not a diagonal matrix, so we can't plug it in directly into an LM solver. To overcome this, we can diagonalize it using eigen decomposition as shown in Eq. 11. Note that Σ_{3D} is a symmetric matrix. So is Σ^{-1}_{3D}. For symmetric matrices, the eigen decomposition is same as SVD. More specifically, the eigen vectors are now orthonormal.

$$\begin{aligned} \Sigma &= UDU^{-1} \\ \Sigma^{-1} &= UD^{-1}U^{-}1 \\ &= UD^{-1}U^T \end{aligned} \tag{11}$$

$$\begin{aligned} &(\mathbf{X} - \mathbf{h}(\mathbf{p}))^T \Sigma^{-1}_{3D_X} (\mathbf{X} - \mathbf{h}(\mathbf{p})) \\ &= (\mathbf{X} - \mathbf{h}(\mathbf{p}))^T UD^{-1}U^T (\mathbf{X} - \mathbf{h}(\mathbf{p})) \\ &= (\mathbf{X} - \mathbf{h}(\mathbf{p}))^T UD^{\frac{-1}{2}} D^{\frac{-1}{2}} U^T (\mathbf{X} - \mathbf{h}(\mathbf{p})) \\ &= (D^{\frac{-1}{2}}U^T(\mathbf{X} - \mathbf{h}(\mathbf{p})))^T (D^{\frac{-1}{2}}U^T(\mathbf{X} - \mathbf{h}(\mathbf{p}))) \end{aligned} \tag{12}$$

Thus we multiply a factor of $D^{\frac{-1}{2}}U^T$ to the error to get the residuals for Levenberg-marquardt optimization. We use the levmar C/C++ library provided by [3]. The initial estimates for the optimization parameters λ_i are taken as ratio of distance between start point and intermediate point to the distance between endpoints of the line. We run LM for 10 iterations which may be too low for proper convergence. This is also a point to explore in the future. Fig. 5 shows lines before and after optimization.

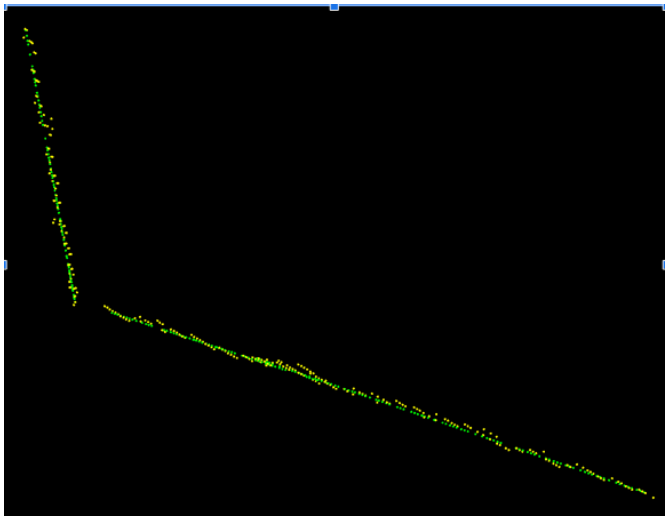

Figure 5. Yellow points represent 3D sampled points in the line before optimization. Green points represent reprojected line segment post-optimization

3.7. Motion Estimation Using SVD

After obtaining the 3D lines post non-linear optimization, we compute an estimate of the camera rotation and translation using pairs of corresponding lines. To compute the estimate of the camera pose, we follow the method outlined in [6]. Atleast two pairs of corresponding lines are required to compute the camera pose. Given (A, B) are the endpoints of the line segment, the line is represented by two

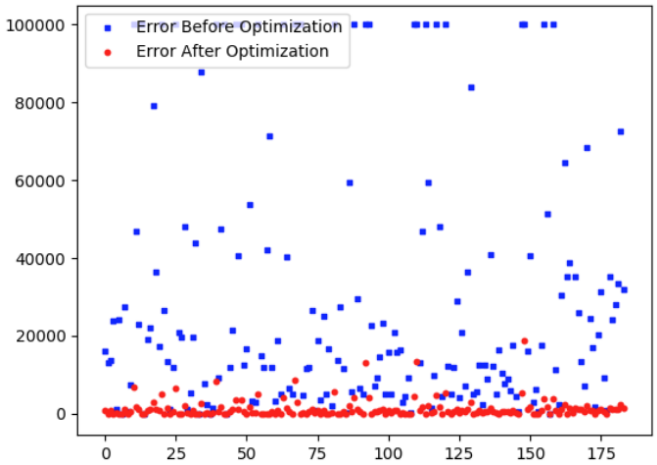

Figure 6. X-axis represent point number and Y-axis represents the error cost for a point

different vectors (u, d) computed as follows, compute the direction vector of the line and the midpoint as,

$$l = B - A \quad \text{and} \quad m = \frac{A + B}{2}$$

$$u = \frac{l}{\|l\|} \tag{13}$$

$$d = \frac{l \times m}{\|l\|} \tag{14}$$

For a line pair (u_i, d_i) and (u'_i, d'_i) , A matrix A_i is computed as follows,

$$A_i = \begin{bmatrix} 0 & (u_i - u'_i)^T \\ -(u_i - u'_i) & (u_i + u'_i) \end{bmatrix}$$

For n such line pairs, a matrix A is computed incrementally as follows,

$$A = \sum_{i=1}^{n} A_i^T A_i$$

We use n=2 in our implementation. The rotation quaternion between that represents the camera rotation between the pair of frames is obtained by minimizing the following objective,

$$\min_q \sum_{i=1}^{n} q^T A_i A q \tag{15}$$

Since A is a symmetric matrix, the solution to this problem is obtained as the vector q_{min} corresponding to the smallest eigenvalue of A .

3.8. RANSAC For Finding Best Camera Pose

The above procedure of computing the initial estimate of the camera pose requires corresponding pairs of non-parallel lines to yield a good estimate. However, the lines are initially matched in 2D and the same correspondences are applied in 3D. This may result in accuracies because there are no geometric constraints applied on these line matches. Hence to obtain the best estimate of camera pose, RANSAC is applied to obtain different pairs of corresponding lines. In this RANSAC process, an initial camera pose is computed from two non-parallel correspondences between the images. Given that a rotation R and a translation t is computed using the previously outlined method, the error metric used to determine whether the line match $L_{i,k_1} \leftrightarrow Lj, k_2$ is an inlier is,

$$\begin{aligned} e_{R,t}(L_{i,k_1}, L_{j,k_2}) = {} & d_M(RA_{i,k_1} + t, L_{j,k_2})^2 \\ & + d_M(R^T A_{j,k_2} - R^T t, L_{i,k_1})^2 \\ & + d_M(RB_{i,k_1} + t, L_{j,k_2})^2 \\ & + d_M(R^T B_{j,k_2} - R^T t, L_{i,k_1})^2 \end{aligned} \tag{16}$$

3.9. Optimizing The Camera Pose

Once an initial estimate of the camera pose is obtained, the camera rotation and translation can be further refined using the inlier set from RANSAC using non-linear optimization. We use the same error metric as used in ransac as our optimization objective. Mathematically this can be represented as,

$$(R', t') = \arg\min_{R,t} \sum_{L_{i,k_1} \leftrightarrow L_{j,k_2}} e_{R,t}(L_{i,k_1}, L_{j,k_2}) \tag{17}$$

The rotation is represented in rodriguez rotation form for this optimization process and again converted into matrix form post optimization. We reuse the levmar library here to perform this optimization. The residuals in this case are the 4 individual mahalanobis distances in eq. 16. So for every line, we get 4 residuals. All line's residuals are stacked together to make up the final residual for optimization.

4. Results

We evaluate on the 30 second TUM fr1 xyz sequence. This dataset was chosen as it provided depth registered RGB image. Also, it has a high accuracy ground truth camera trajectory also provided. To time synchronize the RGB and depth image, they provide a synchronization python script, a copy of which is available in our codebase. They also provide the calibrated intrinsics of the camera.

We evaluate our results quantitatively using the relative pose error metric. We chose this metric as compared to Absolute trajectory error metric as our implementation is a visual odometry system that operates only on a frame pair at a time without considering previous camera poses. Thus RPE was an apt metric. We evaluate with a sliding window of 0.1 seconds. This means that transformation between a 0.1 second delay is calculated in the pose estimate; the nearest corresponding ground truth interval's pose difference is also calculated; The difference between these two transformations (x, y, quaternion) is recorded. Over the entire sequence, the absolute translational and rotational error is calculated and reported.

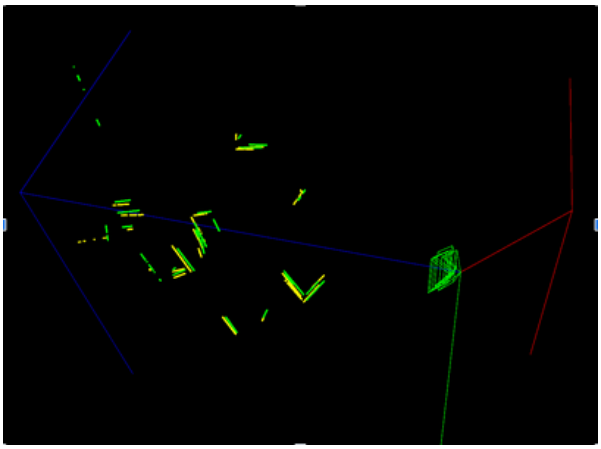

Figure 7. Some camera poses estimated at the start of the sequence. RGB arrows represent world coordinate system (XYZ). Green rectangular pyramids denote old camera poses. Red rectangular pyramid represents current camera pose. Yellow points are line segment points post refinement in F_k, green points are line segment points post refinement in F_{k+1}

Fig. 7 shows some of the estimated camera poses and the refined 3D line segments in scene. We realize that our implentation performs poorly on recovering/estimating the global trajetory of the camera. This is due to sudden jumps in translation and rotation at some instances in time. As VO is incremental, one small mistake accumulates to large errors in the future.

However, the system works well locally and the errors are mostly less than 10cms in translation and 0.1 rad in rotation. This can be seen from the individual RPE errors plotted in Fig. 8 and 9.

Fig. 10 show all a picture of all the camera poses (down sampled for clarity) plotted for the xyz dataset. As can be seen, the global trajectory of cameras has abrupt jumps. In the context of relative pose errors with 0.1 second window, the average translational error was 6.225 cms and rotational error was 0.0685 rad. The error reported in [4] for the same dataset is 4.5 mm and 0.29 degrees for a 1 second window.

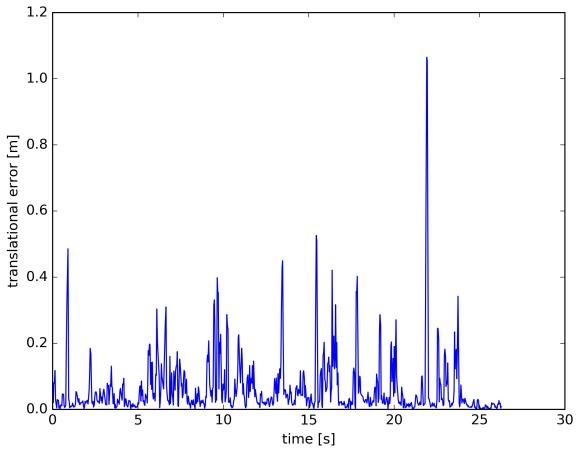

Figure 8. There are sudden spikes in translational error at certain timestamps

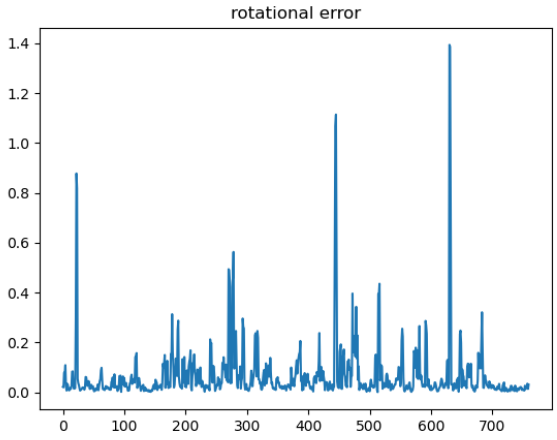

Figure 9. Spikes in error are also visible in rotation component

5. Issues

We used C++ to develop software for our project. There were a number of implementation issues that we faced along the way. Firstly, we started off on the wrong foot by using hough transform for line detection. Midway through our project we had to shift to a different line extraction procedure which also allowed to easily match lines in the images. We ended up using the line segment detector and line band descriptor in our final pipeline. Next we found it hard to debug our code while trying to implement the non-linear optimization to refine the obtained 3D lines. We had to step back and rectify a bunch of bugs in our covariance propagation code.

As we were using C++ for the first time at such a scale, we did not necessarily write optimized software which caused our program to run very slowly. We tried to mitigate

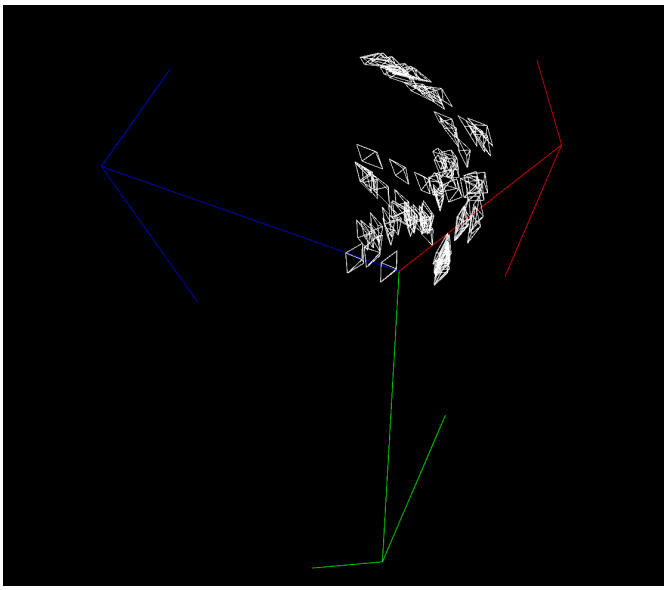

Figure 10. Camera poses of xyz trajectory down sampled by a factor of 10

this issue by reducing the number of line segments detected in the images by constraining the line length to be above a minimum value. We also significantly reduced the number of iterations of our non-linear optimization code. This caused the performance of our system to drop in terms of final accuracy.

6. Conclusion and Future Work

Our implementation is a minimal reproduction of the idea presented in [4]. However, there are many places for improvement. One glaring issue we haven't brought up throughout the report is the computation efficiency of the implementation. Our current version is extremely slow and ran for 1.5 hours to extract the whole trajectory. This is impractical for any online application. The issue is in our C++ methods where we are recreating line objects and covariances redundantly as they are passed between RANSAC calls. The solution would be to convert the implementation to depend on index/pointers to the line objects.

Slow computation should not create a hit on accuracy. However, we see that the accuracy also takes a hit where there are sudden jumps in poses. This could be due to three reasons: 1. Running LM for very low number of iterations to help speed up processing 2. Wrong coefficients of the Kinect depth noise model 3. Wrong initial 2D covariance. We look to investigate these three in the future.

Assuming errors from our end are corrected, we could make improvements on the algorithm all together. We could optimize for camera R and t by jointly optimizing over point and line correspondences. We could go one step further and try to associate objects/planes in scene to add to the ransac support estimation stage. We could keep track of previous camera poses and line descriptions in memory and enforce loop closure constraints to optimize for all previous camera poses to refine estimates at loops. Our implementation can be found at: https://github.com/SubramanianKrish/roblineVO.

References

[1] Sampling algorithm. https://stackoverflow.com/questions/2801814. *Stack overflow*. 2

[2] Rafael Grompone von Gioi, Jérémie Jakubowicz, Jean-Michel Morel, and Gregory Randall. LSD: a Line Segment Detector. *Image Processing On Line*, 2:35–55, 2012. 2

[3] Manolis Lourakis. Levenberg-Marquardt in C/C++, Nov 2011. [Online; accessed 11. Dec. 2020]. 4

[4] Yan Lu and Dezhen Song. Robustness to lighting variations: An RGB-D indoor visual odometry using line segments. *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 688–694, Sep 2015. 3, 6, 7

[5] Kinect precision. https://stackoverflow.com/questions/7696436. *Stack overflow*. 3

[6] Zhengyou Zhang and Olivier D Faugeras. Determining motion from 3d line segment matches: a comparative study. *Image and Vision Computing*, 9(1):10 – 19, 1991. The first BMVC 1990. 4