Vision Aided Navigation 2024 - Final Project

Amitai ovadia - 312244254

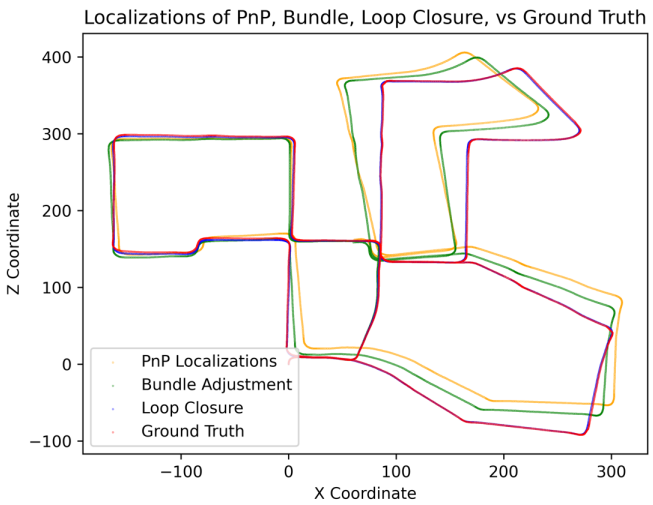
August 24, 2024

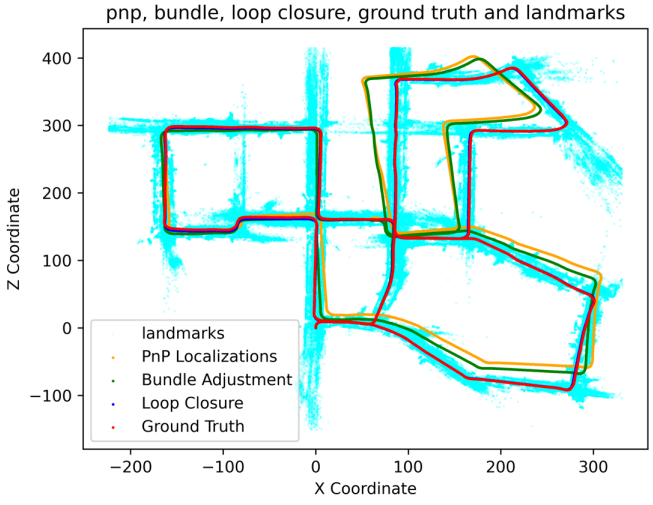
Amitai Ovadia 312244254
https://github.com/AmitaiOvadia/SLAMProject/tree/main/VAN_ex/code

Introduction

This project was done throughout the course "67604 Slam - Video Navigation" in the Hebrew University of Jerusalem, that was presided over by Mr David Arnon and Dr Refael Vivanti.
SLAM, which is short for Simultaneous Localization And Mapping, is the problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location within it. Published approaches are employed in self-driving cars, unmanned aerial vehicles, autonomous underwater vehicles, planetary rovers, newer domestic robots and even inside the human body.
This project is an implementation based mainly on the article "FrameSLAM: From bundle adjustment to real-time visual mapping" by Kurt Konolige and Motilal Agrawal, And tackles the SLAM problem as a pure computer vision problem, without utilizing other sources for it (such as LIDAR).
As the project data we are using the KITTI Benchmark Suite, for estimating the trajectory of a car with a grayscale stereo camera moving through the streets of an urban environment in Germany.

In these graphs: all the different localization results, with the landmarks detected along the process.





These are the main steps used to implement this project:

1. Finding an estimated trajectory using pnp - a deterministic approach
2. Building a features tracking database
3. Performing bundle adjustment optimizations over multiple windows
4. Creating a Pose Graph
5. Refining the Pose Graph using Loop Closures

Kitti Benchmark

The KITTI dataset is a project of Karlsruhe Institute of Technology and Toyota Technological Institute at Chicago.
KITTI uses a car, includes some sensors, traveling around several streets in Germany. The sensors are stereo camera (color and black-white), GPS and Lidar. KITTI's benchmark supplies us with ground truth which allows us to compare the results of our algorithms. In our project we use only the black and white stereo cameras.
It also supplies with:

- The intrinsic camera K of both the stereo pair
- The extrinsic camera of the right camera with respect to the left camera
- 3360 frames
- Ground truth extrinsic matrices during the drive

This is an example for a KITTI frame: the upper image is the left and the lower one is the right image of the stereo pair

Stereo Camera
A stereo camera is a type of camera with two or more lenses with a separate image sensor or film frame for each lens. This allows the camera to simulate human binocular vision.
In the KITTI benchmark we will use a stereo camera for capturing the 3D of the scene.

Camera matrices
Every camera, when assuming pinhole camera scenario, can be modeled by an extrinsic $[R|t]$ and intrinsic K camera matrices. They both define the projection matrix $P = K\,[R|t]$ which takes some 3D point X in homogeneous coordinates, and projects it to the senser coordinate system.

Extrinsic Camera Matrix
Holds all the information of how to translate from the global coordinate system, to the camera coordinate system.
For some 3D point in the world x it performs $Rx + t$, the rotation and translation needed to view it now in the camera coordinate system. That means that for example the camera center, which is of course the origin of the camera coordinate system, must be transferred to the origin.

In our case we have 2 extrinsic camera matrices, the first is the left camera matrix which is the origin:
$M_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$ and the right camera matrix which is the $M_2 = \begin{bmatrix} 1 & 0 & 0 & -0.54 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Not surprising, the difference in the right camera location is only 0.54 meters to the right on the x axis.

Intrinsic Camera Matrix

Projects a $3D$ point in the camera coordinate system, to a pixel in the image, or the camera image plane in the sensor's coordinate system. it includes intrinsic parameters of the camera such as the focal lengths in the x and y axis f_x, f_y, the principal point $c_x, x_y$and also skew and distortion if they exist.

In our case $K = \begin{bmatrix} 707 & 0 & 602 \\ 0 & 707 & 183 \\ 0 & 0 & 1 \end{bmatrix}$

1. Finding an estimated trajectory using pnp

In the following section we will be able to have a first estimation of the relative movement of the car between two consecutive frames, and subsequently composite all these movements on top of each other to get the first estimation of the entire vehicle's path.
So how can we do it?

- First step: Feature extraction and matching For every stereo pair that belongs to a certain frame, we extract key points and their corresponding descriptors. This is done using the AKAZE [1] algorithm, from this paper "Accelerated Embedded AKAZE Feature Detection Algorithm on FPGA" And implemented using the opencv library.

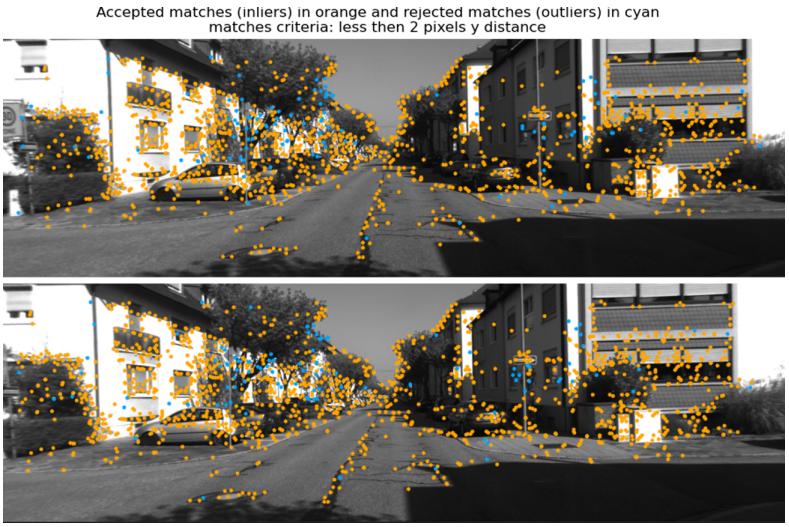
  This algorithm was chosen given that it results in more robust features, as well as very memory efficient descriptors, binary and in length 64.

  The next step is matching the left and right image descriptors and resulting in a list of corresponding points.

- Implementation details:

  - Before using the feature detection algorithm I **blurred** the image using a gaussian kernel of $\sigma = 1$ for eliminating noise in the image and allowing the feature extractor to concentrate on more robust features [2].
  - I used the AKAZE cv2 implementation, and **lowered the detection threshold** from $10^{-3}$to 10^{-4}. This was done in order to get a greater number of features, and compensate for the fact that I blurred the image.

- So how good are these matches? because of the epipolar constraints over a rectified stereo pair, we know that corresponding points must share the same y value, and so we have a way to filter out some of the matches by this metric: what is the y-value distance between them, filter all the matches that their distance is more than d pixels. I used $d = 1.5$ pixels [3].



[1] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/utils.py#L128
[2] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/utils.py#L66
[3] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/utils.py#L116

- The next step will be triangulating each of these points pairs, and find the 3D point that corresponds to each of these pairs.

  this is done using linear triangulation:[4]

  Let's define each 2D point as x, x' and their corresponding $3D$ point as X, and assume that all of them are in homogeneous coordinates.

  For each of the left and right cameras we have a corresponding projection matrices P and P'.

  We know that in projective geometry every line between 2 points can be defined as a cross product between them: $p_1 \times p_2$

  So, we would like to find X such that $x \times PX = 0$ and also $x' \times P'X = 0$.

  And now composing these constraints into a linear system we get:

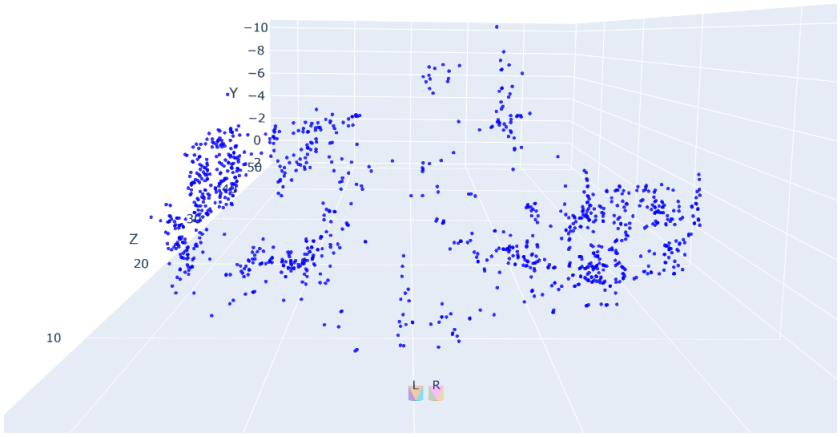
  $$A \in \mathbb{R}^{4\times4} = \begin{bmatrix} yp_3^T - p_2^T \\ p_1^T - xp_3^T \\ y'{p'_3}^T - {p'_2}^T \\ {p'_1}^T - x'{p'_3}^T \end{bmatrix} X = 0$$

  And this is a homogeneous linear system, of the kind:

  $min_x \|Ax\| = 0$ such that $\|x\| = 1$, because X is determined up to a constant.

  We will use the *svd* decomposition:

  $A = U\Sigma V^T$ and the solution will be to take the row in V that corresponds to the smallest eigenvalue of A.



- PnP trajectory calculation

  Here we are going to use the above tools to finally compute the relative movement between 2 consecutive frames.

  The PnP or P4P algorithm is a way to calculate the relative movement, that is to say, the rotation and translation between 2 views.

  For that we need at least 4 $3D$ points and their $2D$ correspondences in the 2 views, and the intrinsic camera matrix K.

  So what we did is as follows:

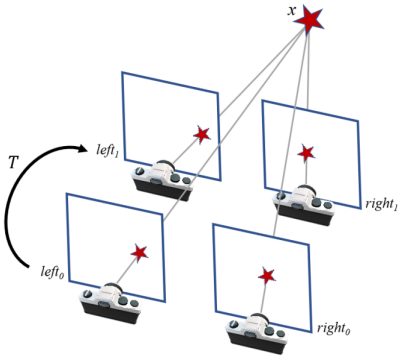
  - We found points that are corresponding across the 2 stereo pairs, by matching the points in the left cameras in both pairs and then for each left camera found the matching points in the right camera

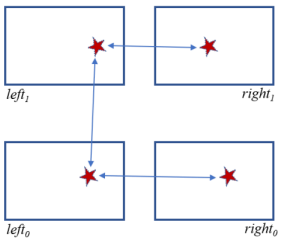
[4] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/utils.py#L176

- Triangulated the points in the first stereo pair's coordinate system (assumed to be the origin as we are interested in the relative movement)

- Now we can use the PnP algorithm to find the relative movement between the left cameras of the second pair vs the first one. [5]

- In the figures: an illustration of the PnP trajectory calculation, from finding points correspondences across the 2 stereo pairs, to the calculation of the relative movement using pnp





- Outliers removal using RANSAC

  A problem that arises here is that, if we want to make sure this relative movement is accurate, then we must be sure that there weren't any outliers who snack up into our PnP calculation, and the way we chose to do it is using the RANSAC algorithm.

  Random sample consensus (RANSAC) is an iterative method to estimate parameters of a mathematical model from a set of observed data that contains outliers, when outliers are to be accorded no influence on the values of the estimates. Therefore, it also can be interpreted as an outlier detection method (Wikipedia).

  The input to the RANSAC in our case are a set of 3D points, and their 4 lists of 2D points that are corresponding to the left and right cameras in the 2 pairs, also we have the intrinsic camera matrix K and the relative $[R|t]$ between the right and left cameras.

  We also input to the RANSAC the probability with which we want the final set of points chosen to be indeed inliers.

  RANSAC is used in our scenario by repeating the following steps: [6]

  - We select a random subset of size 4 from the data (enough for the $P4P$ algorithm) of hypothetical inliers.
  - The model is fitted to the set of hypothetical inliers using PnP, resulting in a relative movement matrix $M_{left_1 \to left_0}$ of the form $[R|t]$.
  - All the data is then tested against the fitted model: We assume that the extrinsic camera matrix of the left camera of the second pair is this estimated $M_{left_1 \to left_0}$, use it to find the extrinsic matrix of the right camera of the second pair, and then project back all the 3D points to the each of the cameras. We call 'inliers' all the points that their reprojection error was less than d pixels, I used $d = 1.5$.
  - We then update the number of iterations needed according to this formula [7]

    $iterations\ needed = \frac{log(1-p_{suc})}{log(1-w^4)}, w = inliers\ ratio$

    The formula is derived as follows:

[5] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/ex3/Ex3.py#L73
[6] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/ex3/Ex3.py#L141
[7] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/ex3/Ex3.py#L136

The chance that all are inliers from 4 picks is w^4, so the chance that not all are inliers is $1-w^4$ and the chance that not all inliers after k iteration is $1-p_{suc}=\left(1-w^4\right)^k \Longleftrightarrow k=\frac{log(1-p_{suc})}{log(1-w^4)}$

- We also update the ratio of inliers w during the iterations thus reducing significantly the number of iterations overwall
- After we are done iterating, we save the largest subset of inliers achieved, and use it to run PnP and get a refined estimation of the model.

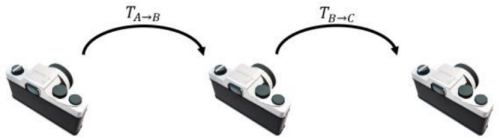
- Estimate the entire trajectory

  So now we have all the relative movements from one frame to the next, we can find the absolute extrinsic matrices for each frame.

  The way to compose 2 extrinsic matrix on each other is as follows:

  $$T=\begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \in \mathbb{R}^{4\times4}, T_{A\to B}=\begin{bmatrix} R_1 & t_1 \\ 0 & 1 \end{bmatrix}, T_{B\to C}=\begin{bmatrix} R_2 & t_2 \\ 0 & 1 \end{bmatrix}$$

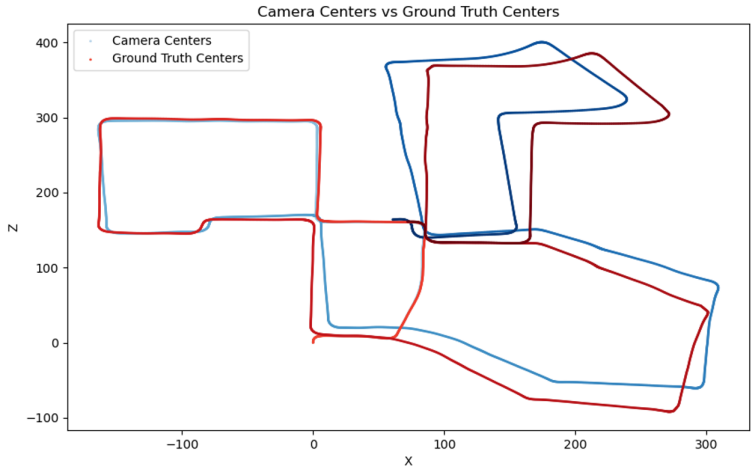
  $$\Rightarrow T_{A\to C}=T_{B\to C}T_{A\to B}=\begin{bmatrix} R_2 & t_2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} R_1 & t_1 \\ 0 & 1 \end{bmatrix}=\begin{bmatrix} R_2R_1 & R_2t_1+t_2 \\ 0 & 1 \end{bmatrix}$$

  And to get the camera center we want a point c such that $Rc+t=0$ (only the camera center moves to 0, in the camera coordinate system)

  And we get $c=-R^Tt$

This is how it looks like in the end of the PnP process:

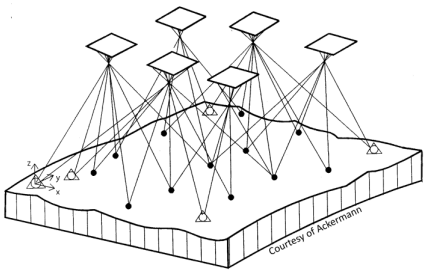


As is clear from this figure, there is a substantial drift that is an attribute of the accumulating mistakes in the pose estimation. We hope to better this estimation using bundle adjustment.

2. Building a features tracking database

Bundle adjustment, an introduction

For further refining the relative poses between consecutive stereo frames, we will use bundle adjustment.
Bundle adjustment is a crucial optimization technique in computer vision, particularly in 3D reconstruction and simultaneous localization and mapping (SLAM). It refines the 3D structure and camera parameters by minimizing the re-projection error between observed and predicted image points across multiple views. This process iteratively adjusts the parameters of both the 3D points and the camera poses to achieve a globally optimal solution.



For performing the bundle adjustment, we are going to have to create a database that tracks all the different landmarks, and provides us with an easy and efficient way to later find for each frame which landmarks correspond to it, and also for each of the landmarks, in which frames it's observed.
The code for the database implementation was given to us by David Arnon, and was modified to serve some more functionalities. [8]

Adding a frame to the data base [9]

The dataset is filled incrementally, one frame at a time in the following logic:

- For each frame:
  - Extract Features: Identify keypoints and descriptors from the left and right images.
  - Match Features: Establish links between the matched features in the left and right images.
- For the first frame:
  - Track Initialization: Create links between matched features, forming the initial tracks.
  - Add to Database: Store the initial links and features in the database.
- For each subsequent frame:
  - Match with Previous Frame: Match current frame features with those from the previous frame to maintain tracking continuity.
  - Outlier Detection: Apply RANSAC to remove unreliable matches.
  - Inlier Identification: Use valid matches to update tracks and create new links.
- Track Formation and Updating:

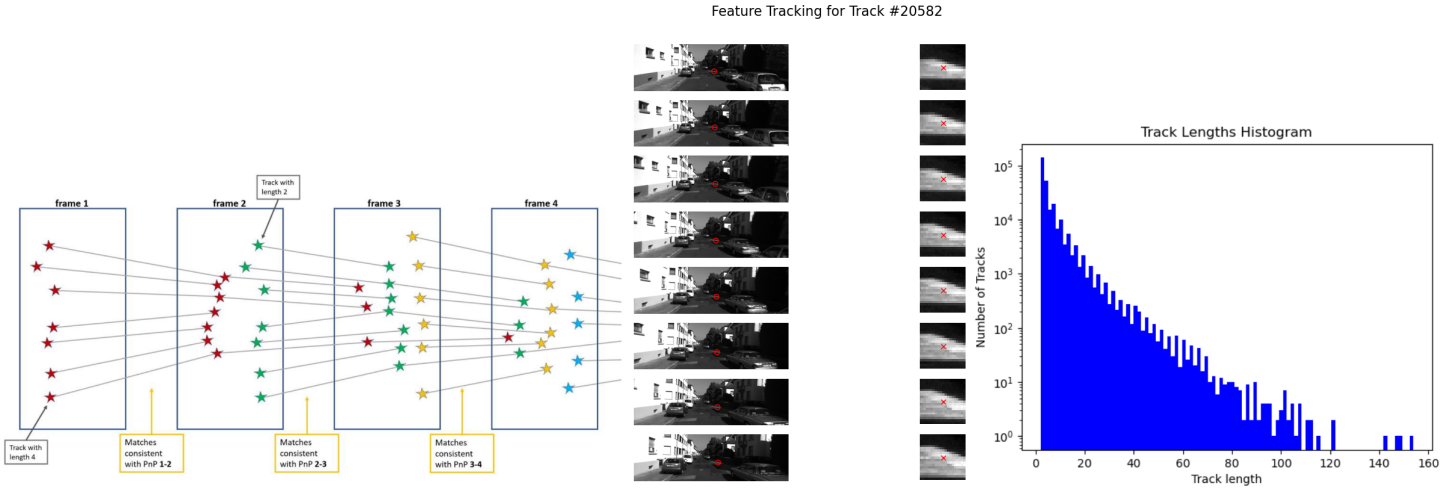
[8] https://github.com/AmitaiOvadia/SLAMProject/blob/main/VAN_ex/code/utils/tracking_database.py
[9] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/ex4/Ex4.py#L58

  - Track Creation and Continuation: New tracks are created for unmatched features, while existing tracks are continued with matched features.
  - Track Management: Update tracks with the new frame's features, refining links as needed.

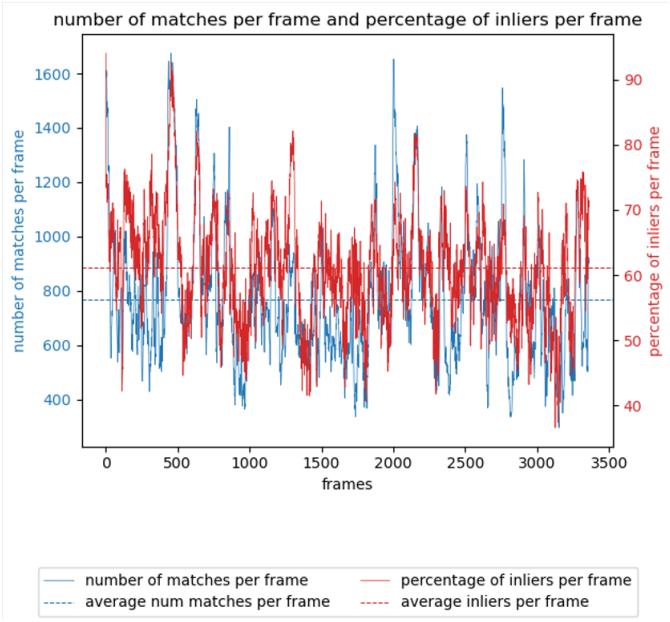
- Add Frame to Database:
  - Store Links and Features: Update the database with the new links and features.
  - Update Camera Pose: Record the camera's position and orientation for each frame.

- Finalization: Serialize (save) the tracking database after processing all frames, preserving the tracking information for future use.

Here is an illustration of the database objective, along with an example for tracking a certain landmark across multiple frames.
Also on the right is a tracking length histogram. I makes sense that it seems linear in log scale, which means it's really exponential, if we consider the probability of a certain landmark to appear in the next frame to be p, so the probability so in the frame after that to be $p^2$and so on and so forth, then the probability for a track to be of length n is p^n which is going down exponentially.



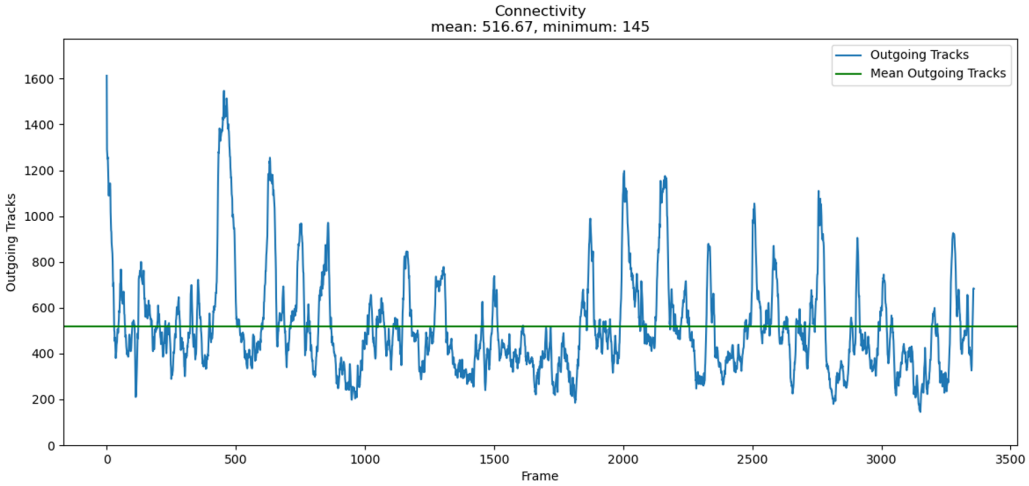
Here is a graph of the number of matches per frame, and also the the percentage of inliers per frame



We can see that these statistics are correlated to each other, usually plenty of matches are correlated with a large number of inliers.
It can be attributed to the fact that if there are plenty of matches, then the images probably represent very little change in scene between the 2 frames, and so no reason for points to be declared outliers.
Also Here are some more statistics:
Tracking Statistics:

- Total number of tracks: 402226
- Number of frames: 3360
- Mean track length: 5.26
- Maximum track length: 153
- Minimum track length: 2
- Mean number of frame links: 629.77

Another importante graph is connectivity:
It means how many tracks are shared between 2 consecutive frames. This score is important because if the connectivity suddenly dropped, it probably means that our pose estimation for this frame is bad, it will also compromise our future absolute localizations.



Here we can see that there was good connectivity throughout the process, as it never dropped below 145.

3. Performing Bundle Adjustment over Multiple Windows

When performing bundle adjustment, because as displayed before most of the tracks are short and do not extend to the entire movie, then if we try to solve a big bundle adjustment to all movie frames the problem will be very big and also very sparse. Our solution to this is to cut the movie into small parts, in some meaningful manner, and solve bundle adjustment for each part separately.
Every such part will be called 'bundle window' or in my code: 'bundelon' [10].

- division into key frames [11]

  We want to divide the movie into small parts such that each part is small enough to make the bundle solving efficient and not too sparse (many tracks are shared between the frames) but also not too small so that it will have plenty of constraints and get a better estimation of the camera poses.

  It was done in the following way:

  A keyframe is selected based on the lengths of feature tracks in the current frame. The process involves calculating the lengths of these tracks, which represent how long a feature remains visible across multiple frames. A percentile of these track lengths is then determined to represent a typical track length (in our case the 40_{th} percentile). The next keyframe is chosen by advancing in the sequence by this percentile length, ensuring that keyframes are spaced to capture significant feature continuity throughout the sequence.

  Also, each bundle window has an overlap of one frame with the window following it to provide continuity between the windows (the first frame of each bundle is the last one of the previous one).

- Single Bundle creation [12]

  All the frames between key frames are now bundle windows, and we solve the bundle adjustment problem for each of them.

  The way we do it is by creating a *gtsam* factor graph object, and adding to it all the subsequent factors.

  - The stereo camera pose associated with each frame: the first frame of such window is assumed to be the 'origin' and all the other poses are calculated relative to it.
  - All the tracks, and the cameres in which it is captured
  - for each such couple of track and camera we also add some uncertainty factor for the measurement of this factor in $2D$ by the $AKAZE$ detector. we chose it to be gaussian with $\sigma = 1$
  - A prior is also added, in the first frame being in the origin, with a 1 degree uncertainty for each of the pose euler angles, and 0.1 meter uncertainty in the x direction, 0.01 meter uncertainty in the y direction and 1 meter uncertainty in the z direction.

  Because the bundle adjustment problem is very non-convex we also add an initial guess for each of the factors. for the camera poses we add the PnP estimations, and also for the $3D$ landmarks we add their triangulated location as captured by the last frame they are tracked, mainly because the closest you are to a 3D point, the better stereo triangulation.

[10] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/BundleAdjusment.py#L14
[11] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/BundleAdjusment.py#L464
[12] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/BundleAdjusment.py#L157

- The last phase is optimizing the factor graph. [13]

  It is done using the Levenberg Marquardt algorithm, which is a way to to solve nonlinear least squares problems, balancing between a Gauss–Newton algorithm (GNA) and gradient descent.

  - in this figure we can see an example of the improvement the bundle achieved in terms of the reprojection error, over the initial estimation provided by the PnP estimated camera poses

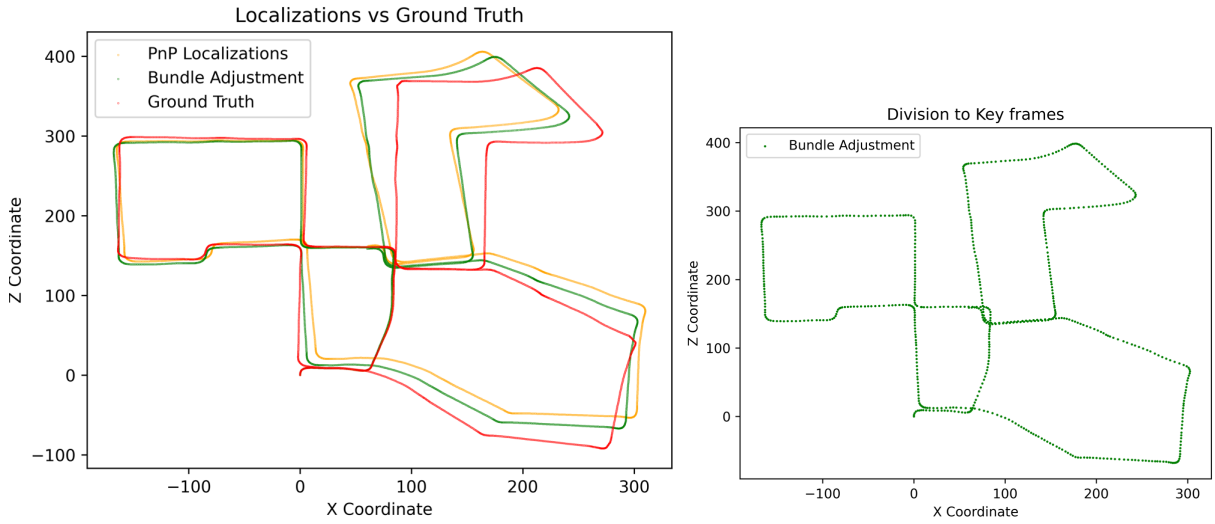
- The relative transformation (including covariance) extraction from the bundle result

  The bundle optimization will return the solution, that is the camera poses and the 3D landmarks positions with the maximum likelihood, based on the observed points and the injected prior.

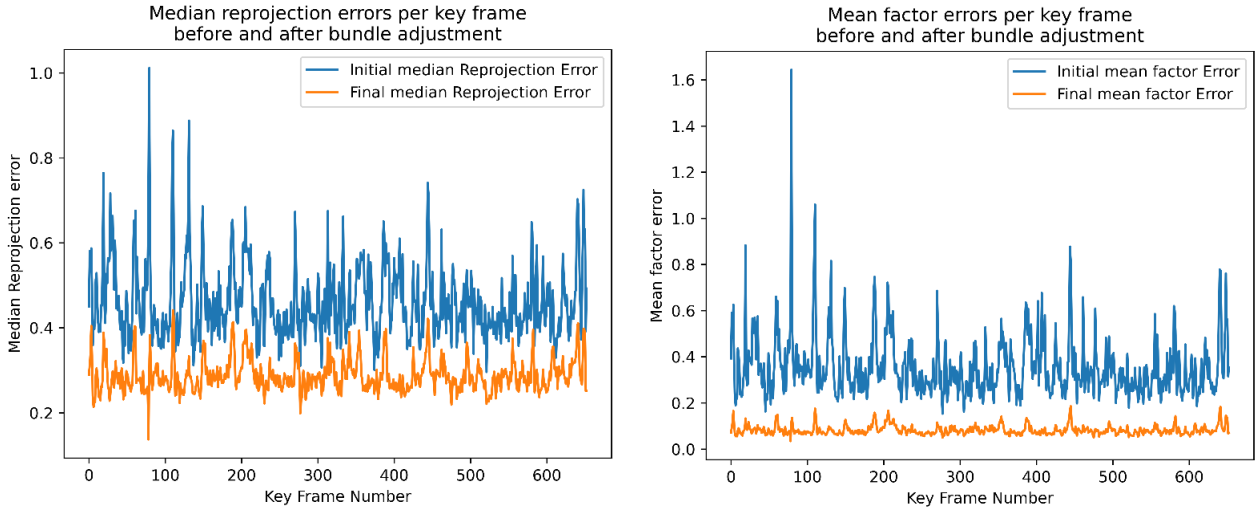
  It also returns the covariance matrix of the entire optimized factor graph solution, which we then use in order to extract the covariance, or in simple terms the uncertainty of the movement between the first and last frame of each window. It is done using marginalization and conditioning.

  Here are the bundle adjustment localization vs the pnp localizations. We can see it's a little bit better. but there is still a significant drift that is the result of accumulating errors. In the next graph we can see the division into keyframes, we can see that during turns the keyframes are more densely spread then when the trajectory is straight

[13] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/BundleAdjusment.py#L268



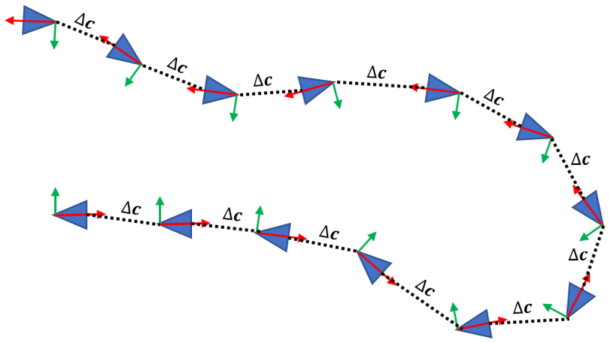
In the following figure we can see the improvement in the median reprojection error, and in the mean factor error, before and after the bundle adjustment.



We can see that there is an improvement, and also that the median reprojection error initially was not bad, this is because of a strict outlier removal policy of 1.5 pixels reprojection error threshold.

4. Pose Graph Creation

The next step that follows the bundle adjustment is the pose graph creation.



A pose graph is an object that holds only the camera poses obtained for each **key frame,** and its associated movement covariance or uncertainty with respect to the previous (or any other) key frame.
This is a much more compact representation of the trajectory then the previous bundle adjustment representation using all the cameras and landmarks, And the main use for it will actually be to find loop closures in the next step [14].
The pose graph is given only relative poses between different cameras and the assosiated covariance of this poses, and optimizes the poses using maximum likelihood [15].
When optimizing it using the only initial key frames poses, we get that the optimization actually did nothing as there was no new information added. but when we will inroduce more constraints in the form of loop closures, then the graph will change substantially to accomodate the new information.

[14] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/PoseGraph.py#L375
[15] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/PoseGraph.py#L408

5. Refining the Pose Graph using Loop Closures

[16]

In this step we are going to finally solve the drifting problem. This problem was due to the fact that we are composing the relative transformations on top of each other, and one small mistake in one of them along the way, especially in the azimuth angle, will result in a very large accumulative error.
We can exploit the fact that the vehicle actually comes back to some of its previous locations, and if we are able to recognize that, we will be able to impose further constraints in the form of loop closures: a calculated constraint between non-consecutive frames. These will act as 'staples', or anchors, that will fix the drift in this location and it's surroundings as they are factors with small covariances that will 'force' the pose graph optimizer to divide the errors differently across the graph to account for this new factor.
This is how it was done:

- Creating a loop closure graph object [17], such that every vertex in the graph is a camera pose of a certain key frame.

  The weights on the edges of this graph are some uncertainty measurement that is a function of the covariance between this 2 poses.

  we chose it to be, for the 2 poses p_1, p_2, $w(p_1, p_2) = \sqrt{|covariance\,[p_1, p_2]|}$, the square root of the determinant of the covariance matrix, as it is proportional to the 'volume' of the covariance matrix.

  Each edge also holds an attribute of the covariance matrix itself.

- Find candidtates for loop closure: [18]
  - We are performing a pose distance measurement as a way to find out if poses are approximately at the same location and orientation as one another, and we are likely to find an accurate rotation and translation between them using the PnP algorithm.
  - For each key frame, we look at all the previous key frames (from 60 frames back and up, no need to check close frames)
  - for every previous frame we find the shortest path in the graph (with the uncertainty weight defined earlier) [19]
  - We construct an approximated covariance matrix cov between the previous frame and the current frame by summing all the covariances of the intermiditate edges. [20]
  - We check the approximated Mahalanobis distance of the 2 poses as $\Delta_p^T \cdot cov \cdot \Delta_p$, $\Delta_p = p_{cur} - p_{prev}$ [21]
  - If the mahalanobis distance is under a certain threshold, in our case 750, then the couple $\{p_{cur}, p_{prev}\}$ are loop closure candidates.

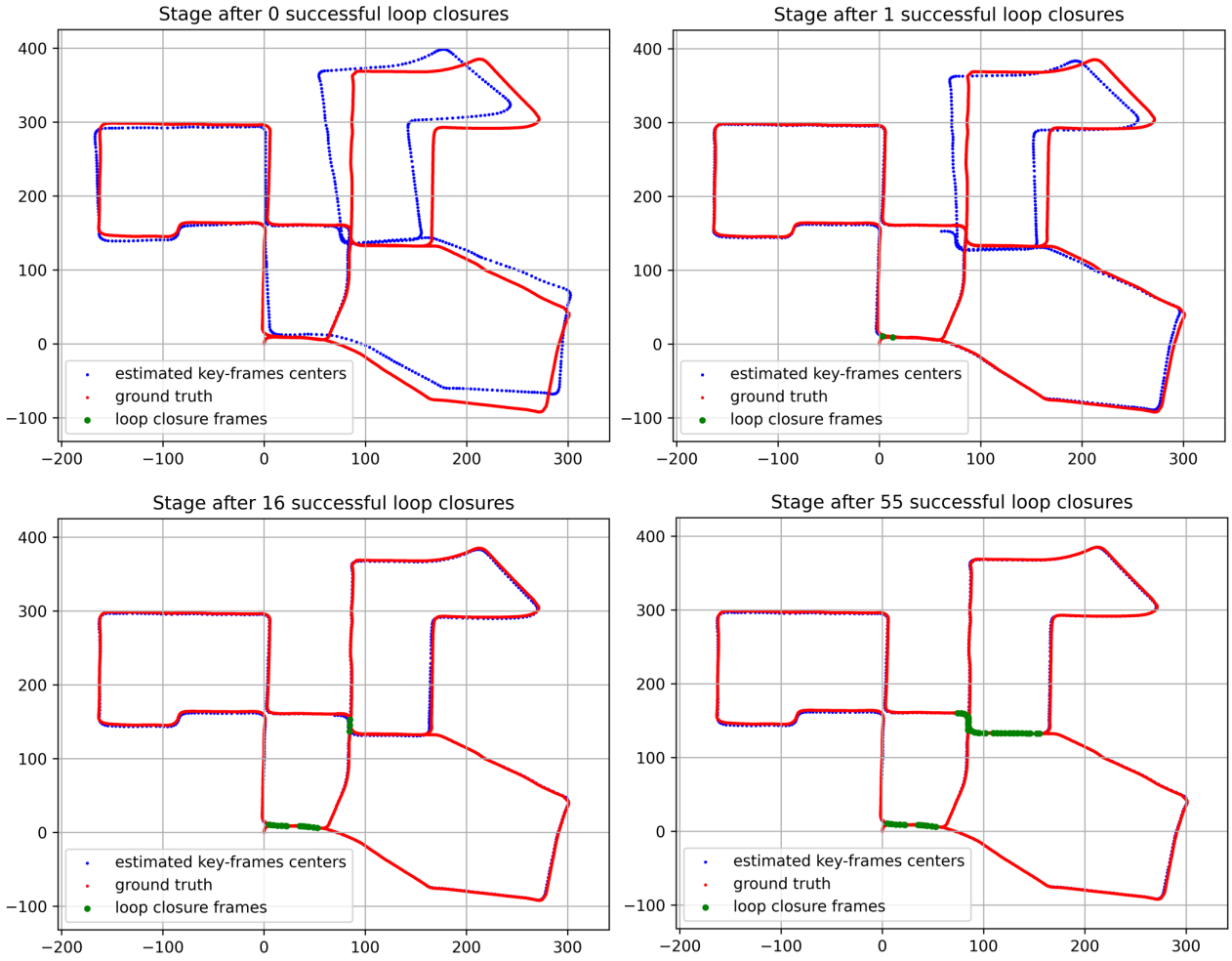
- performing the loop closure optimization.
  - After compiling all the loop closure candidate for p_{cur}:
  - For every loop closure candidate p_{cand} we want to find the relative pose between them, and the covariance associated with it, so we perform the PnP algorithem defined in the PnP estimation section, and discarding candidates with less than 50 inliers [22]

[16] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/PoseGraph.py#L21
[17] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/PoseGraph.py#L353
[18] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/PoseGraph.py#L153
[19] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/PoseGraph.py#L169
[20] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/PoseGraph.py#L336
[21] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/PoseGraph.py#L346
[22] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/PoseGraph.py#L272

– Then, using the result of the PnP as the initial estiamtation the matched landmarks as well, we optimize a small bundle adjusment that includes only this two cameras, and retrive the associated covariance. [23]

– We update the pose graph by adding this new found edge $\{p_{cur}, p_{cand}\}$, optimize the pose graph, and update the loop closure graph object.

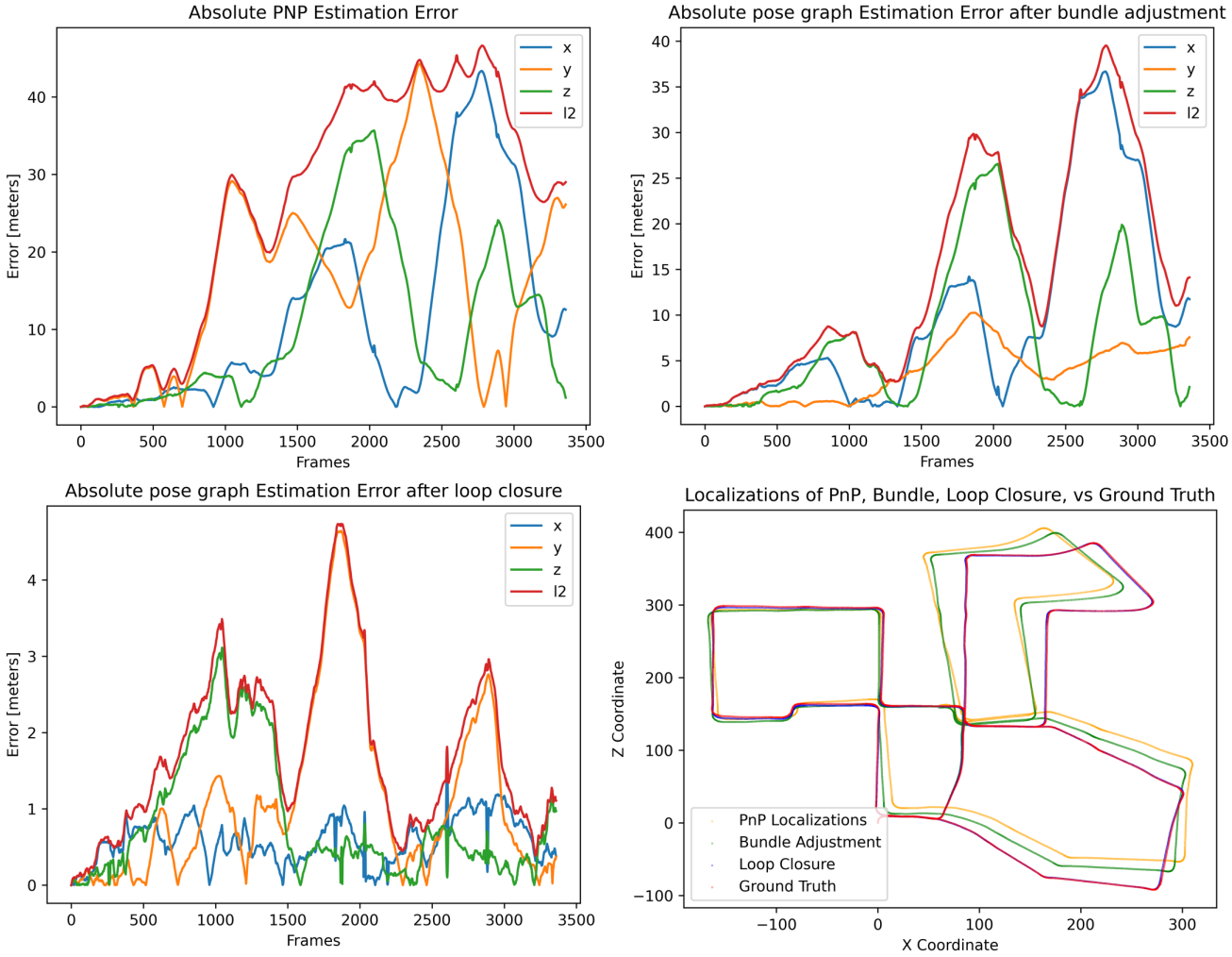
Here is how the pose graph looks like as more and more loop closures are added to it, with comparison to the ground truth locations:



[23] https://github.com/AmitaiOvadia/SLAMProject/blob/fcafb474671f3078c2a305bfc554414367019797/VAN_ex/code/utils/PoseGraph.py#L247

Performance Analysis

In this section I will provide some more quantitative measures of the SLAM system.
Here are graphs of the absolute localization errors in x, y, z and L_2 *norm*, after the initial pnp estimation, after the bundle adjustment (before the loop closure) and after loop closure:

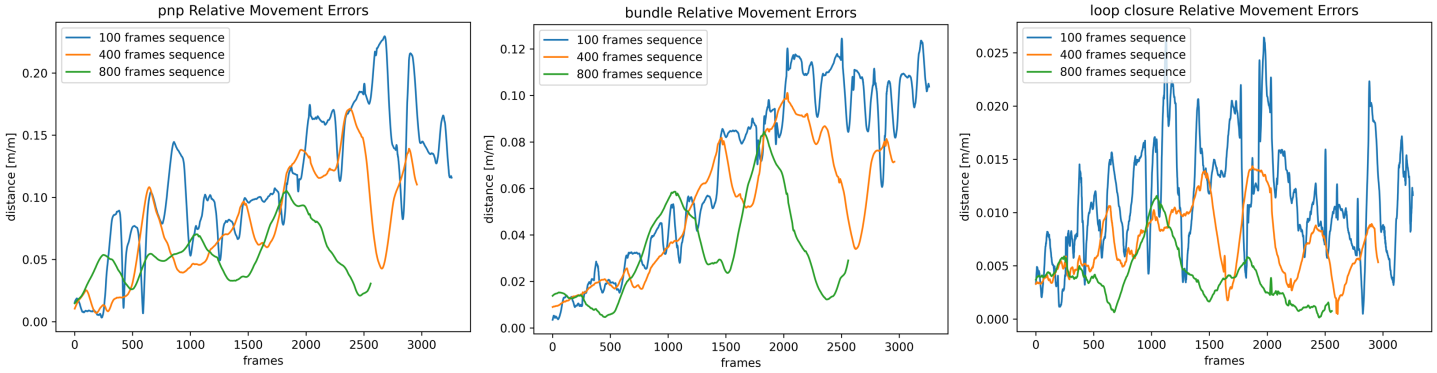


It's clear from the graphs the there is a real boost in the absolute estimation accuracy after the loop closure part, the maximum error comes down from around 40 meters to around 5 meters. The reason is of course the elimination of the drift, or the accumulating error.

Relative Error

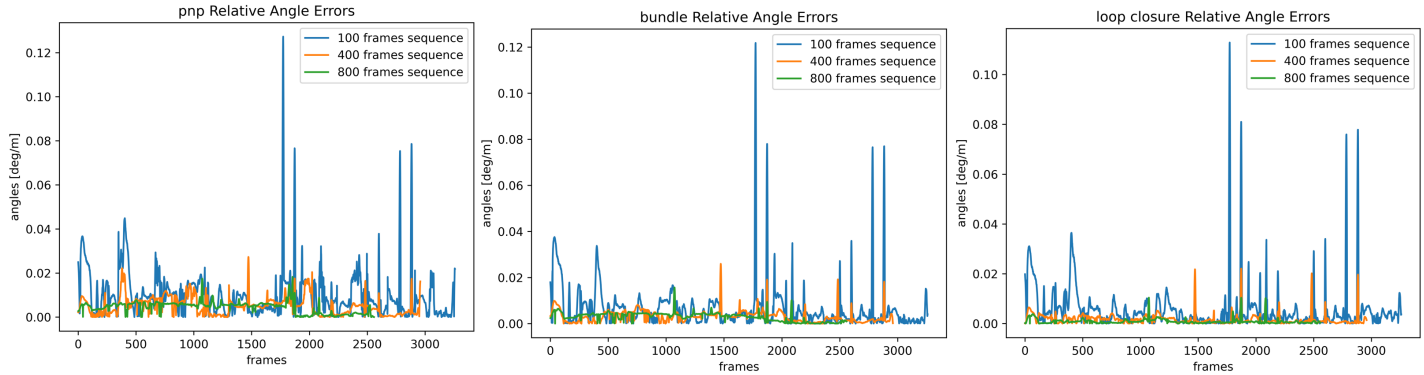
In the next graphs we will focus on the relative errors, both in localization and in pose angles.

A basic relative estimation error graph compares the relative pose estimation between two consecutive frames $\Delta C(c_i, c_i + 1)$ and the ground truth relative pose between them $\Delta C(gt_i, gt_i + 1)$.



It's clear from the graphs as well that there was a huge decrease in the relative error after the loop closures, due to the drift correction.

In these graphs I have done relative angles estimation in the same manner like in the relative localization, using the Rodrigues formula:



These results are quite baffling, because I was under the assumption that, like the localization errors, it will be much improved after the loop closure stage but you can still see the error spikes in all the 3 graphs. The only reason I can think about is that the ground truth rotations are a noisy this measurement is not a real testament for the accuracy of the SALM system, because if this angles errors spikes were real it would indicate that the angles I calculated were incorrect, and that will sure result in major localization errors as well, and we see that the errors are quite low after the loop closure stage.