Really Adverse Weather Radar Odometry and Mapping (RAW-ROAM): An open-source implementation of RadarSLAM Odometry

Kevin Liu (kevinliu), Samuel Leong (scleong), Brian Zhang (bez), Alex Chen (anc2)

May 4, 2022

Contents

Abstract

In the field of SLAM, vision and LIDAR based SLAM systems have been increasingly popular over the recent years. While these systems achieve impressive results in outdoor environments in typical weather, these sensors often fail in more adverse environmental conditions such rain, snow and fog. Really Adverse Weather Radar Odometry and Mapping (RAW-ROAM) is a Radar-based SLAM which takes advantage of Radar's greater robustness in extreme weather conditions to address these shortcomings of previous SLAM systems. We aim to create the first open-source implementation of current state-of-the-art radar-based SLAM by Hong, Petillot and Wang [1], focusing specifically on the odometry component of RAW-ROAM with additional improvements to the original algorithm. To implement the odometry and make improvements, we will refactor various aspects of their paper, and use ideas drawn from other radar odometry papers such as [2] and classical computer vision methods and data structures [3, 4] in order to perform accurate and efficient for real-time usage radar odometry reduce drift. We will evaluate our results on the Oxford Robot RadarCar dataset [5, 6], similar to that of the original paper.

1 Introduction

Simultaneous localization and mapping (SLAM) is a contemporary topic in robotics which aims to perform pose tracking while accurately mapping the space around the robot. There are many methods of SLAM which are reliant upon different sources of information, typically estimating movement information from odometry performed by encoders or other methods while also evaluating and building a map and pose correcting using information from sensors such as cameras or LiDAR. However, many challenges are presented in the problem, and one of the biggest challenges is that of sensor noise and sensor inaccuracies.

Most cameras and LiDAR sensors are not robust to changes in illumination, which can occur in multitudes of situations. In the case where there is low light illumination, cameras and LIDARs will have more motion blur, noises, and losses of texture [7]. Additionally, LIDARs and cameras are not invariant to situations with high concentration of dense particles, such as the fog, snow, rain, or storm conditions. In misty, rainy, or snowy weather, the density of the particles will significantly affect the point clouds generated by LiDAR and the images generated by cameras. Creating a weather-invariant SLAM system is still an active area of research, and the RAW-ROAM method is one that aims to solve such a problem.

Radar is a sensor that has the potential to combat weather adversity. Radar sensors utilize a significantly lower frequency range (GHz) as compared to LiDARs (THz to PHz). Due to this, it can penetrate more easily through particles and therefore can operate more reliably in weather conditions. It also offers a further sensor range, relative velocity estimations (from Doppler effect), etc. In recent research, radar sensing has also shown the application of creating dense maps rather than range restricted maps, which further increases its viability. Radar is therefore a prime choice in combating weather adverse conditions where other sensors fail.

However, radar brings many challenges, including significantly lower resolution and higher signal-to-noise ratio compared to traditional sensors. Moreover, its sensing patterns, geometry, and data formation is all very different from vision and LiDAR. This means that robust feature detection algorithms used in other sensors cannot be directly applied to radar data, and intermediate steps have to be taken to clean up the data from radar-specific sensing problems, such as radar motion distortion. RAW-ROAM describes a full pipeline that aims to use radar data to generate mapping and odometry information. We aim to reimplement and possibly improve upon the RadarSLAM algorithm [1, 7], focusing specifically on the odometry and mapping components (i.e. RAW-ROAM).

2 Contributions

As of the paper's creation, no other open-source implementation of RadarSLAM has been made available. While parameters from the paper have been released to allow for reimplementation, the lack of an implementation of this contemporary method will be troublesome to future researchers in the field. As such, we aim to make available **the first open-source implementation** of RAW-ROAM odometry to give insight into RadarSLAM and its viabilities. Additionally, the 2020 and 2021 papers describe similar but different methodologies to create the radar odometry pipeline. We will combine the 2020 and 2021 Radar SLAM papers[1, 7] to try to reimplement and improve upon the RAW-ROAM odometry algorithm.

3 Algorithm

3.1 Overview

An visual overview of our pipeline is shown in Figure 1 below. Our pipeline consists of 3 main parts: feature extraction, odometry and local mapping, as well as a partially complete loop closure thread. The specifics of the algorithm are found in the various sections below:

Thread	**Relevant Sections**
Feature Extraction	Feature Selection, Feature Tracking
Odometry	Relative Transformation Calculation, Graph-Based Outlier Rejection, Rotation Correction with Fourier-Mellin Transform (FMT)
Local Mapping	Keyframe Selection, Motion-Compensated Bundle Adjustment
Loop Closure	Future Work

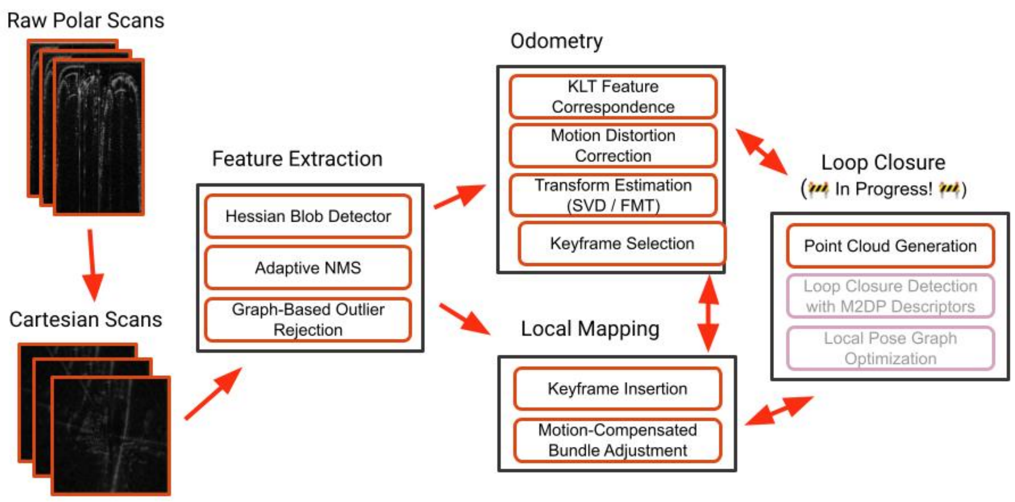
Raw Polar Scans
Cartesian Scans
Feature Extraction
Hessian Blob Detector
Adaptive NMS
Graph-Based Outlier Rejection
Odometry
KLT Feature Correspondence
Motion Distortion Correction
Transform Estimation (SVD / FMT)
Keyframe Selection
Local Mapping
Keyframe Insertion
Motion-Compensated Bundle Adjustment
Loop Closure
(🚧 In Progress! 🚧)
Point Cloud Generation
Loop Closure Detection with M2DP Descriptors
Local Pose Graph Optimization

Figure 1: Visual Overview of Pipeline

3.2 Code Base

For the sake of easy prototyping and to reduce issues with difficult C++ dependency setup, we decided to implement the algorithm from scratch in Python, with vectorized numpy and the C++-based OpenCV libraries for efficiency. While this means that the algorithm will not be able to run in real-time, we are still contributing significantly to the broader scientific community because it is the first open-source implementation of RadarSLAM available publicly. The code is well documented, and because of the extensive use of OpenCV, it should be relatively easy for others to port over to C++. It is hosted publicly on GitHub at `https://github.com/Samleo8/RadarSLAMPy/`https://github.com/Samleo8/RadarSLAMPy/.

3.3 Input Data and Coordinate Conversion

In a 360-degree radar scan, the sensor sends out numerous azimuth scans and waits for echos as the beams bounces off of objects in the environment, as shown in Figure 2 below. A raw polar image in azimuth and range units can be converted into a Cartesian image using bilinear interpolation. This is done using OpenCV's `warpPolar()` function. Note that elevation information such as object height is thrown out and the data is collapsed into a single 2D image along a flat plane. Due to the decreasing accuracy of radar with increasing range, we cropped out information from radar scans at a distance greater than 87.5 meters, as per the paper's parameters.

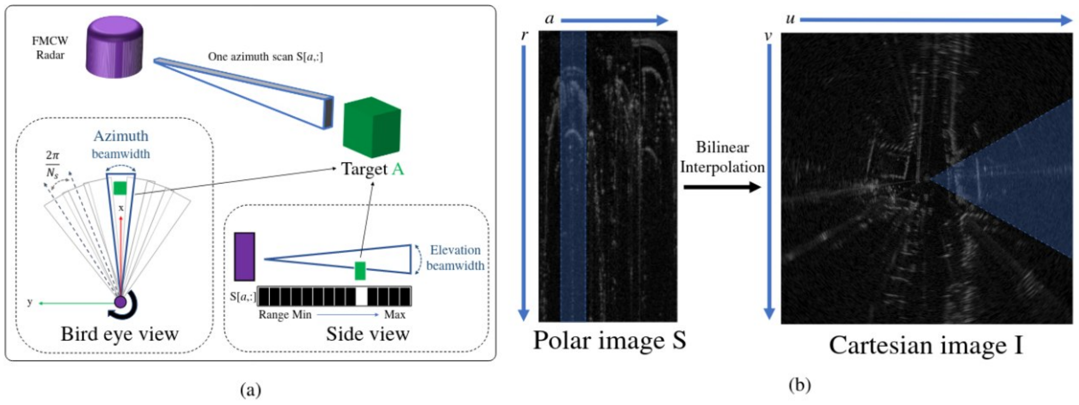

Figure 2: Radar scanning operation, and log-polar and Cartesian representations of the radar image. Taken from [7]

3.4 Feature Selection

In these Cartesian radar scans, a small patch of higher response corresponds to a detected object. Feature points can thus be extracted efficiently via a basic Hessian Blob Detector [7]. The Hessian blob detector will be able to extract homogenous areas, or in our case, large significant areas of static and dynamic obstacles. This approach locates maximas in the Determinant of Hessian of the radar image and is computed using box filters instead of convolutions as is the method when using the alternate Laplacian of Gaussian (LoG) or Difference of Gaussian (DoG) approaches to blob detection. We found the DoH to be the fastest and most robust of the three approaches as it suppresses the responses of small specks in the radar image that could be the result of radar noise. We further thresholded the candidate points by their Hessian response. The parameters we used can be seen in Table 1 below:

Table 1: Determinant of Hessian (DoH) Blob Detector Parameters

Parameter	`min_sigma`	`max_sigma`	`num_sigma`	`threshold`
Value	1	30	10	0.1

These points are further pruned using an adaptive non-maximal suppression (ANMS) approach to create a set of points which are spatially distributed evenly across the entire radar image while retaining the most relevant ones [3]. This allows us to discard many redundant feature points clustered together from our blob detector. In our implementation, we use a suppression via Square Covering (SSC) implementation of ANMS. We find that ANMS filtering drastically reduces the number of feature points by an adjustable parameter $(K < N)$ in an outdoor sequence.

3.5 Feature Tracking

We utilize a Kanade–Lucas–Tomasi (KLT) tracker, a fast method commonly used in visual odometry, to find correspondences of sparse feature points. For two subsequent Cartesian radar scans and a set of blob features on the first scan, KLT attempts to iteratively guess the optical flow of the blob features between the two frames. It works by using image gradients on the radar scans to identify potential matching blobs between the scans and minimize the residual difference of the returned point correspondences and their 2D neighborhoods for local tracking. We use the OpenCV implementation of KLT, which exploits image pyramids [8] and RANSAC to improve the performance of the operation in the event of large rotations or translations.

3.6 Relative Transformation Calculation

From the set of feature correspondences $\Omega_t = \{(p_0, q_0), (p_1, q_1), \ldots, (p_n, q_n)\}$ between Cartesian scans $\mathcal{S}_{t-1}$ and $\mathcal{S}_t$, we are able to estimate the movement of the robot during this period by using the following SVD-based approach proposed by Arun et al. [9]:

Consider the unknown transformation matrices R, t to find, and their relation $\mathbf{q} = R \cdot \mathbf{p} + t$. We then construct the matrix H and find its singular value decomposition:

$$H = (\mathbf{p} - \text{centroid}_{\mathbf{p}})(\mathbf{q} - \text{centroid}_{\mathbf{q}})^\top \qquad\qquad U, S, V = \text{SVD}(H)$$

Then, we obtain the rotation and translation as follows:

$$R = VU^\top \qquad\qquad t = \text{centroid}_{\mathbf{q}} - R \cdot \text{centroid}_{\mathbf{p}}$$

This result is a rotation matrix R and translation vector t of blobs across corresponding radar frames in pixels. Using the intrinsic parameters of our radar sensor such as the range resolution, we are able convert this transform into the ego movement of the robot in the world frame. By computing this transformation between all consecutive frames, we are able to integrate the trajectory of the robot over the course of the entire sequence. Note that in this setup, KLT may fail to find good point correspondences for certain feature points in the previous frame p_i. This may be due to noise, occlusion, or typically an object moving out of sensor range and are subsequently discarded and not included in the set of correspondences Ω. However, in order to maintain a healthy set of feature points for tracking, we replenish them if the number of features falls below a threshold `min_num_features` = 80.

3.7 Graph-Based Outlier Rejection

Ideally, we want to only track static features like roads, lamp-posts, trees and signages, instead of dynamic features such as moving cars, pedestrians and cyclists. Unlike in the case of camera images or 3D LiDAR point clouds, the unique birds-eye-view geometry that the radar sensor affords us enables the pruning away of dynamic outliers. This is because static features in a 2D birds-eye-view must satisfy the rigid-body constraint across frames: for any 2 arbitrary (static) points, the distance between them must be the same between frames. Therefore, to find out whether a point satisfies this constraint, we use a graph-based method as follows.

Consider two frames A and B, and the set of n correspondences between them

$$\mathbf{p}_A = \left\{\mathbf{p}_A^i | i \in [n]\right\} \qquad\qquad \mathbf{p}_B = \left\{\mathbf{p}_B^i | i \in [n]\right\}$$

We then form an unweighted graph $\mathcal{G}$ whose nodes are the corresponding feature points from the above set. We then form an adjacency matrix, connecting 2 arbitrary nodes i and j together if only they satisfy the following constraint:

$$\left\|\mathbf{p}_A^i - \mathbf{p}_B^i\right\|_2^2 - \left\|\mathbf{p}_A^i - \mathbf{p}_B^i\right\|_2 < \epsilon$$

where the value of ϵ used in our implementation is 0.5 meters.

In other words, we only connect 2 nodes (points) if the Euclidean distance between them in frame A is "close enough" to the distance in frame B. The inlier set is therefore the set of nodes (points) which are still connected to the other nodes in $\mathcal{G}$, and finding this inlier set is equivalent to finding the maximum clique of $\mathcal{G}$ [1, 7]. In code, this is solved efficiently using the Python `networkx` library.

Figure 3 below visualizes how the graph-based outlier rejection works. The object in dark red is a moving outlier and fails to satisfy the rigid-body constraint, unlike the feature points shown as the green boxes. As visualized in the figure, these green boxes will form a clique with each other in graph $\mathcal{G}$.

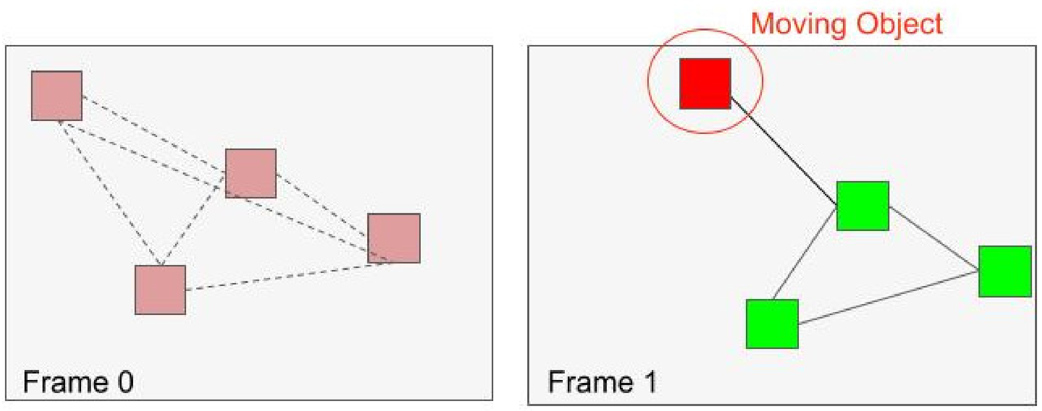

Figure 3: Visualization of graph formed in outlier rejection

Figure 4 below shows a real example of outlier rejection working to remove dynamic outliers, including moving features (eg. moving cyclist or pedestrian) and radar sweep errors (eg. multipath, noise).

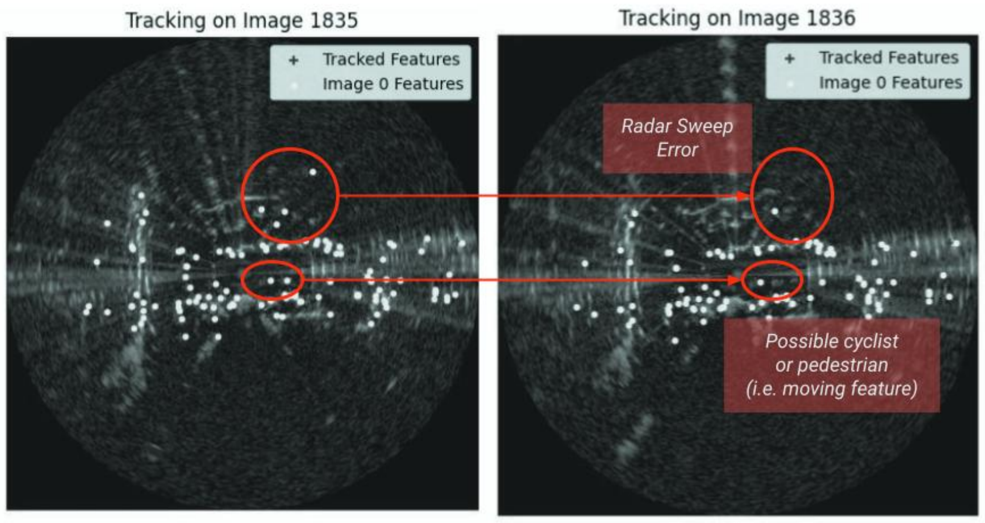

Figure 4: Example of outlier rejection working on real data

This graph-based outlier rejection is extremely effective and key to ensuring that only static features are tracked for a robust odometry estimate. Tests show that an average 28% of feature points are removed using this algorithm.

3.8 Rotation Correction with Fourier-Mellin Transform (FMT)

As noted in the paper [7], KLT sometimes fails to track properly when rotations are large. To resolve this problem, we present a novel contribution to correct for rotations before applying KLT, using the Fourier Mellin Transform (FMT) as used in [2] to get an initial estimate for rotation. In summary, we first convert the radar image into log-polar space, and perform phase correlation to obtain the relative translation in the log-polar domain, which corresponds to a change in angle (rotation) and scale. We followed the suggestions in the paper to downsample only in the radial direction to save on memory and computation while keeping precise angular resolution. However, we also additionally cropped the image using the maximum range value specified in the RadarSLAM paper [7] because we were finding that the long multipaths in the uncropped image were significantly throwing off the rotation estimates in the FMT algorithm. Figure 5 below shows an example of rotation correction using the FMT algorithm. Currently, this feature is turned off for the sequence we are testing on because the algorithm works fine without it, but might come in useful in situations where relative rotations are more severe (e.g. with sharper turns).

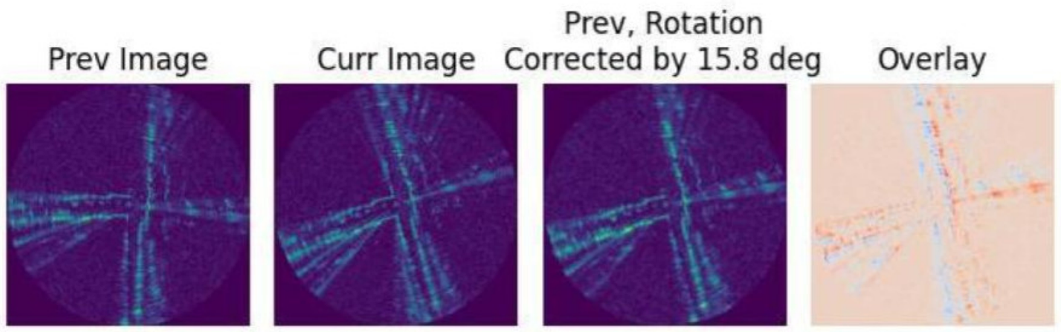

Figure 5: Example of Fourier Mellin transform correctly accounting for large rotations

3.9 Keyframe Selection

In order to reduce memory requirements, especially for real-world large-scale systems, we do not log the pose and map point information for every frame to be used for local mapping and bundle adjustment. Instead, we choose *keyframes*, which contain pose information and feature points (from Feature Selection) to be used as map points. A frame $\mathcal{F}$ is selected to be a keyframe if it matches any of the following criteria from [1]:

- First frame of sequence

- Euclidean distance (relative position) between the previous keyframe, $\mathcal{K}$, is less than threshold δ_d:

$$\left\|\mathcal{F}_{(x,y)} - \mathcal{K}_{(x,y)}\right\|_2 < \delta_d$$

- Rotation (relative orientation) between the previous keyframe, $\mathcal{K}$, is less than threshold δ_θ:

$$|\mathcal{F}_\theta - \mathcal{K}_\theta| < \delta_\theta$$

- Number of map points in $\mathcal{F}$ is below the `min_num_features` threshold used in Section 3.4 above.

The parameters used in the code are shown in Figure 3.9 below.

Table 2: Keyframe Selection Parameters

Parameter	δ_d	δ_θ	`min_num_features`
Value	2 m	0.2 rad	80

3.10 Motion-Compensated Bundle Adjustment

3.10.1 Motion Distortion

When using the radar sensor, which operates at relatively low frequencies (4Hz for our dataset) compared to LiDAR and cameras, we must compensate for the movement of the robot during the course of each scan. As aforementioned, the mechanical scanning radar takes numerous azimuth scans as it rotates a full 360-degrees. During the course of a full 360-degree scan, a high speed vehicle could have traveled multiple meters and degrees meaning that each azimuth scan could have been taken from a different position and orientation in space. In reconstructing these 2D radar scans, we assumed that the robot was stationary, however this assumption results in severe motion distortion as visualized in Figure 6. This phenomenon is especially evident during sequences in which the robot is turning as this adds or subtracts from the absolute rotation of the rotating radar scanner. As a result, motion distortion is responsible for a great deal of drift when doing pure dead reckoning as seen in Figure 7 below.

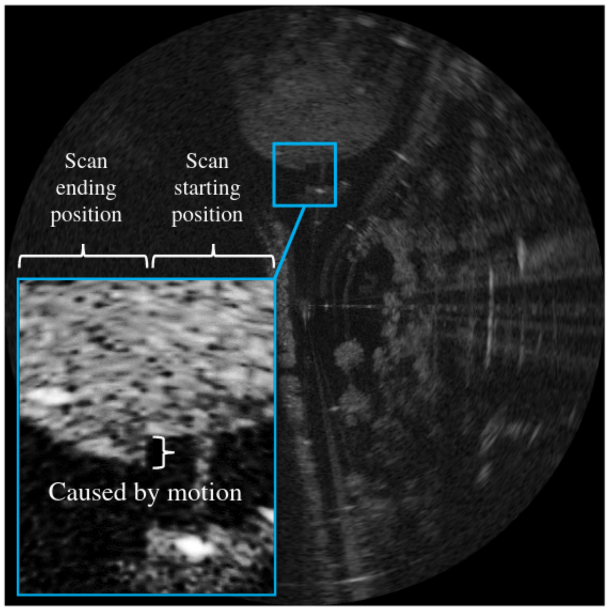

Figure 6: Example of motion distortion, taken from [7]

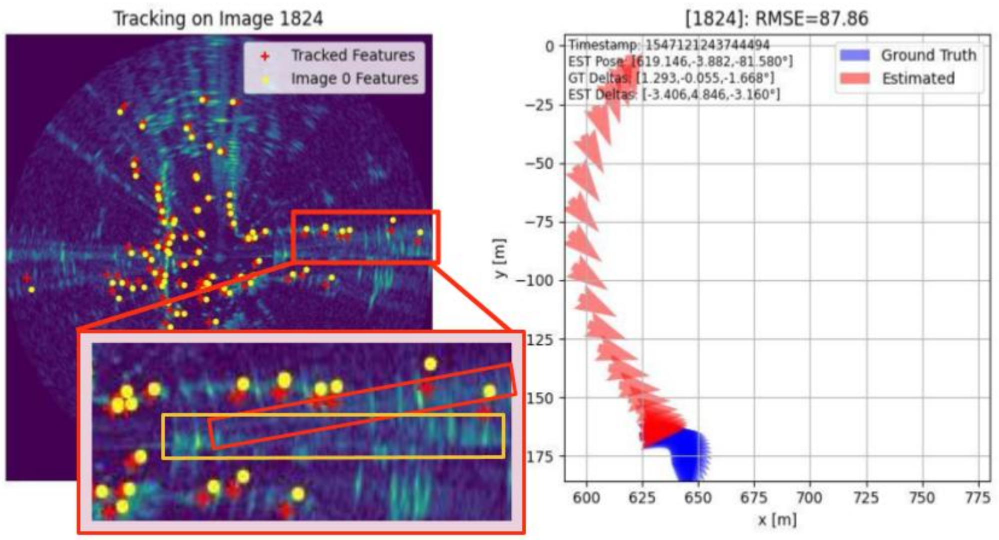

Figure 7: Motion distortion on real data affecting estimates during turns.

To account for motion distortion, the velocity of the vehicle is explicitly modeled to compensate for the distortion in odometry [7]. Suppose that a full radar scan $\mathcal{S}_j$ takes time Δt to complete. Suppose also that the vehicle has a velocity vector $\vec{\mathbf{v}}_j = \begin{bmatrix} v_x & v_y & v_\theta \end{bmatrix}^\top$ for the period of this scan. Then, for the time range of azimuth scans $t \in \left[-\frac{\Delta t}{2}, \frac{\Delta t}{2}\right]$, the pose of the robot in the scan frame can be modelled as:

$$\mathbf{p}_{j_t} = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} v_x t \\ v_y t \\ v_\theta t \end{bmatrix} \implies \mathbf{T}^j_{j_t} \begin{bmatrix} \cos\theta_t & -\sin\theta_t & x_t \\ \sin\theta_t & \cos\theta_t & y_t \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos(v_\theta t) & -\sin(v_\theta t) & v_x t \\ \sin(v_\theta t) & \cos(v_\theta t) & v_y t \\ 0 & 0 & 1 \end{bmatrix}$$

where $\mathbf{T}^j_{j_t}$ is the pose of the robot in the scan frame (j) in "homogeneous matrix form", and can also be thought of as the operation to "undistort" a motion-distorted point.

If we denote $\mathbf{T}^w_j$ as the transformation from the scan frame to the world frame (w), obtained directly from odometry estimates as per Section 3.6, then we can effectively obtain our motion-compensated robot pose at scan time t as:

$$\mathbf{T}^w_{j_t} = \mathbf{T}^w_j \, \mathbf{T}^j_{j_t}$$

In the same vein, if the i^{th} keypoint $\mathbf{q}^i_w$ in the world frame is observed as $\mathbf{q}^i_{j_t}$ in the azimuth scan at time t, then its motion compensated location (j_0) in the scan frame is [7]:

$$\mathbf{q}^i_{j_0} = \mathbf{T}^j_{j_t} \mathbf{q}^i_{j_t}$$

3.11 Bundle Adjustment Optimization

We then formulate an optimization problem as in [7], which aims to minimise both the residual of the keypoint positions, e_p, and the residuals of the estimated velocity e_v simultaneously":

$$e_p = \rho_c \left(\left(\mathbf{T}^w_j\right)^{-1} \mathbf{q}^i_w - \mathbf{q}^i_{j_0} \right)$$
$$e_v = \vec{\mathbf{v}}_j - \vec{\mathbf{v}}_{prior}$$

where ρ_c is the Cauchy robust cost function, and $\vec{\mathbf{v}}_{prior}$ is the estimated velocity between frames (relative pose divided by Δt), obtained from the relative pose of the keypoints. The initial conditions for both $\mathbf{q}^i$ and $\vec{\mathbf{v}}_{prior}$ are obtained from naive KLT tracking, as explained in Section 3.6 above.

This "velocity" residual term is important because our initial estimate of velocity (obtained from naive

KLT tracking) is extremely off. Moreover, the velocity estimate is dependent on the incorrect estimates of keypoint positions which we are trying to optimize for, and so both the keypoint positions and velocity need to be optimized together.

It is noted that here we are effectively performing motion-distortion-compensated bundle adjustment by obtaining better estimates of (local) map points and robot poses using a motion model. The optimization is solved by the Levenberg–Marquardt algorithm, and our implementation uses the `scipy` solver library to do so.

4 Experiments and Results

5 Future Work

Because of time constraints and issues with the `g2o` library, we were unable to get pose graph optimization and loop closures working. However, the code base for generating the point clouds and M2DP descriptors [10] required for the loop closure portion of the pipeline is in our repository, and other people would be able to build from it if they want to.

6 References

[1] Z. Hong, Y. Petillot, and S. Wang, "Radarslam: Radar based large-scale slam in all weathers," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5164–5170, IEEE, 2020.

[2] Y.-S. Park, Y.-S. Shin, and A. Kim, "Pharao: Direct radar odometry using phase correlation," *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2617–2623, 2020.

[3] O. Bailo, F. Rameau, K. Joo, J. Park, O. Bogdan, and I. Kweon, "Efficient adaptive non-maximal suppression algorithms for homogeneous spatial keypoint distribution," *Pattern Recognition Letters*, vol. 106, 02 2018.

[4] J. Konc and D. Janežič, "An improved branch and bound algorithm for the maximum clique problem," *MATCH - Communications in Mathematical and in Computer Chemistry*, vol. 58, 01 2007.

[5] D. Barnes, M. Gadd, P. Murcutt, P. Newman, and I. Posner, "The oxford radar robotcar dataset: A radar extension to the oxford robotcar dataset," *arXiv preprint arXiv: 1909.01300*, 2019.

[6] W. Maddern, G. Pascoe, C. Linegar, and P. Newman, "1 Year, 1000km: The Oxford RobotCar Dataset," *The International Journal of Robotics Research (IJRR)*, vol. 36, no. 1, pp. 3–15, 2017.

[7] Z. Hong, Y. Petillot, A. Wallace, and S. Wang, "Radar slam: A robust slam system for all weather conditions," *arXiv preprint arXiv:2104.05347*, 2021.

[8] J. yves Bouguet, "Pyramidal implementation of the lucas kanade feature tracker," *Intel Corporation, Microprocessor Research Labs*, 2000.

[9] J. H. Challis, "A procedure for determining rigid body transformation parameters," *Journal of Biomechanics*, vol. 28, no. 6, pp. 733–737, 1995.

[10] L. He, X. Wang, and H. Zhang, "M2dp: A novel 3d point cloud descriptor and its application in loop closure detection," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 231–237, IEEE, 2016.