Computation Intelligence
19CSE458

Enhancing 3D LiDAR Localization through Range Images using Particle Swarm Optimization and Fuzzy Logic

Submitted by: Batch 11

Name	Registration Number
B V Anagha	BL.EN.U4EAC22012
K Supriya	BL.EN.U4EAC22024
Krishna P Tambatkar	BL.EN.U4EAC22026

Semester: VI

Submitted to: Dr. R V Sanjika Devi

Department of Electronics & Communication Engineering

Amrita School of Engineering, Bengaluru

Title : Enhancing 3D LiDAR Localization through Range Images using Particle Swarm Optimization and Fuzzy Logic

1. Problem Statement

Accurate and efficient 3D localization using LiDAR data remains a significant challenge, especially in real-time autonomous systems. The high computational cost of processing dense point clouds, combined with the risk of local minima in traditional Monte Carlo Localization (MCL) approaches, often leads to reduced performance in dynamic or cluttered environments. These issues become even more critical when operating in large-scale, unstructured areas where precision and speed are both essential.

Reliable 6-DoF localisation for LiDAR-equipped ground robots depends on two prerequisites: (1) an accurate, topologically consistent ground-mesh map, and (2) a pose-estimation pipeline that is both robust to outliers and smooth over time. Existing range-image Monte Carlo Localisation (MCL) [1] frameworks typically satisfy the first requirement by using principal-component analysis (PCA) to classify ground points, and the second by accepting the raw MCL particle-filter output. In practice, however, PCA struggles on sloped or discontinuous terrain, while the discrete nature of particle updates leaves small-scale jitter that accumulates into drift. To address these limitations, we propose Fuzzy-PSO-MCL, an end-to-end upgrade of the standard pipeline that introduces two key ideas:

1. Fuzzy ground-mesh segmentation.
2. Particle-Swarm Optimisation (PSO) refinement.

2. Objectives

- Build a reliable ground-mesh map from LiDAR scans using a fuzzy-logic ground classifier instead of the conventional PCA test.
- Localise the robot with range-image Monte Carlo Localisation (MCL) on the fuzzy-based mesh.
- Refine the MCL trajectory in real time with a lightweight particle-swarm optimisation (PSO) pass to suppress jitter and drift.
- Quantitatively compare baseline (PCA-MCL) versus fuzzy-only, PSO-only, and combined Fuzzy-PSO-MCL pipelines on public and in-house datasets.

- Deliver an open-source, CPU-friendly stack that needs only a single LiDAR sensor yet improves accuracy and map quality for autonomous ground vehicles.
- Evaluate the proposed framework's performance using the KITTI benchmark dataset, comparing localization accuracy, speed, and robustness against standard MCL-based methods.

3. Literature Survey

Self-Driving Vehicle Localization using Probabilistic Maps and Unscented-Kalman Filters

Wael Farag et al. [1] present a real-time Monte Carlo Localization (RT_MCL) framework for self-driving vehicles, designed to balance high pose estimation accuracy with computational efficiency in urban environments. The approach integrates a pole-like landmark-based probabilistic map, sensor fusion using an Unscented Kalman Filter (UKF), and a customized Particle Filter (PF) for ego-vehicle localization. The UKF fuses radar and LiDAR data to identify static pole-like objects—such as lamp posts and signposts—with improved precision, using a high-order motion model and GB-DBSCAN clustering followed by RANSAC-based pole fitting. These landmarks are associated with a reference map using Iterative Closest Point (ICP) for data alignment. The PF is initialized using GPS/IMU fusion and operates with as few as 50 particles, thanks to the probabilistic representation of landmarks that allows Bayesian inference to handle map uncertainties. Extensive testing using the CARLA simulator demonstrated mean localization errors of 11 cm, real-time performance at 30 Hz on moderate hardware, and robustness under varying map uncertainties and particle counts. However, while the system achieves strong performance under controlled simulation conditions, its dependence on accurate pole detection and offline-generated probabilistic maps may limit adaptability in highly dynamic or unstructured real-world environments

An improved particle filter for mobile robot localization based on particle swarm optimization

Qi-bin Zhang et al. [2] present an enhanced global localization approach for mobile robots that combines particle filtering with particle swarm optimization (PSO) to improve robustness and efficiency in environments with ambiguous or symmetrical features. The proposed Particle Swarm Optimization Filter (POF) algorithm operates in two stages: initial pose estimation and multiple hypothesis pose tracking. In the first stage, a modified PSO algorithm with Euclidean spatial neighborhood topology guides particles uniformly distributed in the free space toward high-fitness regions, using a fitness function based on cosine similarity between captured laser scans and an occupancy grid map. The DBSCAN clustering algorithm then groups particles into sub-swarms, each representing a distinct pose hypothesis. In the second stage, each sub-swarm undergoes local optimization and probabilistic filtering, ensuring diversity preservation and robust convergence. Experiments conducted on the Intel Research Lab and Fort AP Hill datasets demonstrate that POF consistently outperforms standard Monte Carlo Localization (MCL) and Self-Adaptive MCL (SAMCL), achieving sub-5 cm positional accuracy and sub-0.2° orientation error with significantly fewer particles. However, the POF algorithm introduces higher computational overhead during the initial pose estimation due to iterative optimization, and its performance in non-ambiguous environments may suffer from unnecessary clustering and redundant hypotheses.

Mobile robot localization based on PSO estimator

Ramazan Havangi et al. [3] introduce a novel localization framework for mobile robots based on a Particle Swarm Optimization (PSO) estimator, aimed at overcoming the well-documented degeneracy and sample impoverishment problems associated with traditional particle filters (PFs). By reformulating the localization task as a dynamic optimization problem, the proposed method employs PSO to stochastically explore the robot's state space, iteratively seeking the pose that maximizes the posterior probability without relying on importance sampling or resampling steps. The algorithm integrates a gradient-enhanced fitness function that combines the log-likelihood of both the motion and observation models, enabling it to exploit local gradient information while maintaining global search capabilities through the use of simultaneous perturbation techniques. Simulation and real-world experiments, including tests on the University of Sydney car park dataset, demonstrate that the PSO-based estimator achieves higher accuracy and robustness compared to PF and Extended Kalman Filter (EKF) approaches, especially under low-particle regimes and non-Gaussian noise conditions. Quantitative results reveal a root mean square error (RMSE) in position as low as 0.05 m with only 30 particles—substantially outperforming PF which required higher particle counts to achieve comparable accuracy. Despite its strong performance, the algorithm introduces increased computational complexity relative to EKF and requires careful parameter tuning to balance exploration and exploitation in varying environments.

FastSLAM-MO-PSO: A Robust Method for Simultaneous Localization and Mapping in Mobile Robots Navigating Unknown Environments

Bian et al. [4] propose an enhanced simultaneous localization and mapping (SLAM) framework for mobile robots—FastSLAM-MO-PSO—by integrating multi-objective particle swarm optimization (MO-PSO) into the conventional FastSLAM algorithm to address challenges posed by non-linear dynamics, non-Gaussian noise, and particle degeneration in unknown environments. The authors reformulate SLAM as a multi-objective optimization problem, optimizing two conflicting objectives: accurate direct measurement estimation and adherence to environmental constraints, each defined through probabilistic models incorporating sensor noise, unexpected obstacle detection, and measurement failures.

MO-PSO is used to dynamically adjust particle positions by evaluating both local (pbest) and global (gbest) performance within the Pareto-optimal solution set, thereby improving particle diversity and resilience. The FastSLAM-MO-PSO method was validated through extensive simulation experiments across small, medium, and large-scale environments, and benchmarked against FastSLAM, FastSLAM-PSO, DE-enhanced variants, and multi-objective FastSLAM-MODE. Results show superior performance in mapping accuracy, trajectory tracking, and robustness, achieving minimal deviation from true paths, especially under increased obstacle density and sensor noise, while maintaining feasible computational load. Nevertheless, the method incurs higher runtime overhead compared to baseline FastSLAM due to the complexity of multi-objective optimization and the increased computational demands of maintaining Pareto front archives, suggesting a need for adaptive parameter control in future real-time implementations.

Range Image-based LiDAR Localization for Autonomous Vehicles

Xieyuanli Chen et al. [5] propose a robust and generalizable global localization framework for autonomous vehicles that operates solely on 3D LiDAR data, circumventing the need for GPS or prior pose information. The study introduces a Monte Carlo Localization (MCL) system that integrates a novel observation model based on range images, which are derived from both real-time LiDAR scans and synthetic renderings of a triangular mesh map generated through Poisson Surface Reconstruction (PSR). The mesh map, constructed from prior LiDAR data and SLAM-estimated poses, undergoes ground segmentation and vertex simplification to reduce computational complexity while preserving geometric fidelity. Each particle in the MCL represents a hypothesized pose, and its likelihood is computed by comparing the real and synthetic range images via mean absolute pixel-wise difference, allowing for efficient and informative weight updates. An OpenGL-based rendering pipeline accelerates synthetic image generation with occlusion handling, and a tiled map structure further optimizes performance by spatially constraining rendering to particle-relevant regions. The approach was evaluated across diverse datasets—including Carla, IPB-Car, MulRan, and Apollo—encompassing varied urban environments and LiDAR configurations ranging from 8 to 128 beams. Results demonstrated consistent localization accuracy, achieving positional RMSEs as low as 0.44 m and yaw errors around 2.53°, with strong generalization across sensor types and acquisition conditions. Comparative benchmarks against traditional beam-end models, histogram-based similarity metrics, and deep learning-based observation models revealed superior accuracy and success rates, particularly at moderate particle counts. However, the system requires a substantial number of particles during initialization (up to 100,000) to ensure reliable convergence, resulting in significant early computational costs that may pose challenges for real-time deployment in resource-constrained systems.

Real-Time LiDAR–Inertial Simultaneous Localization and Mesh Reconstruction

Yunqi Cheng et al. [6] propose LI-SLAMesh, a real-time LiDAR–inertial simultaneous localization and mesh reconstruction framework designed for robust pose estimation and dense mapping in outdoor environments. The system integrates two main components: a LiDAR–inertial odometry module and an online mesh reconstruction module. The odometry component builds upon the FastLIO2 architecture, replacing its IESKF optimizer with a residual-density-driven Gauss–Newton algorithm that adjusts residual weights based on the spatial distribution of LiDAR point normals, thereby mitigating the degeneracy caused by redundant or uneven data. Simultaneously, the mesh reconstruction module eschews traditional TSDF-based models in favor of a compact voxel map that retains only occupied voxels, where Signed Distance Field (SDF) values are computed using an iterative Implicit Moving Least Squares (IMLS) method. This design eliminates the need for ray casting and allows accurate, scalable mesh generation using marching cubes over selectively updated voxels. The framework was evaluated across several real-world and synthetic datasets, including KITTI, NCLT, and Stevens-VLP16, demonstrating superior mapping precision—with mean projection errors as low as 0.01 m—and improved odometry performance compared to FastLIO2, iG-LIO, and FasterLIO baselines. Nevertheless, while the approach enhances both efficiency and map fidelity, it incurs additional computational overhead from IMLS processing and mesh updates, especially in large-scale settings, suggesting a need for further optimization for real-time scalability in high-throughput robotic systems.

Stereo vision-based vehicle localization in point cloud maps using multiswarm particle swarm optimization

V. John et al. [7] propose a stereo vision-based localization framework that utilizes a novel multiswarm particle swarm optimization (PSO) algorithm to accurately localize vehicles within dense 3D point cloud maps, addressing the limitations of GPS in urban environments. The localization pipeline operates in three phases: an offline phase for calibrating the stereo transformation matrix using PSO; a bootstrapping phase that stabilizes noisy GPS–INS data using a Kalman filter; and an online phase where a constrained PSO tracker, initialized with the bootstrapped estimate, continuously refines the vehicle's pose. Candidate virtual depth maps are generated from the point cloud using coordinate transformations and are compared against real-time stereo depth maps through a depth-based cost function. To enhance efficiency, the cost function is restricted to pruned disparity regions, filtering out irrelevant structures like pedestrians and vehicles using the V-disparity method. The multiswarm approach conducts parallel localized searches around both the previous and predicted poses, improving robustness against divergence and motion uncertainty. Experimental evaluations across multiple datasets reveal that the proposed MPML

(Multi-Particle Multi-Limit) variant achieves superior localization accuracy—recording feature-space errors as low as 0.01 in missing-GPS scenarios—compared to baseline particle filters and annealed particle filters. However, the framework exhibits increased computational load, particularly in the online phase, requiring up to 400 ms per frame even with optimization, suggesting further refinement is needed for large-scale real-time deployment on embedded platforms.

An Enhanced Particle Filtering Method Leveraging Particle Swarm Optimization for Simultaneous Localization and Mapping in Mobile Robots Navigating Unknown Environments

Xu Bian et al. [8] propose an enhanced FastSLAM algorithm that integrates Particle Swarm Optimization (PSO) to improve the robustness and accuracy of Simultaneous Localization and Mapping (SLAM) in mobile robots navigating unknown environments. The method addresses the inherent limitations of traditional particle filters—particularly particle degeneracy and reduced accuracy under non-Gaussian noise—by reformulating SLAM as an optimization problem where particles are guided by a PSO algorithm using a measurement-based fitness function. This integration enables particles to converge more effectively toward high-likelihood regions of the robot's state space, thereby increasing localization precision and mapping fidelity. The algorithm dynamically updates particle velocity and position based on both individual and global bests, and is designed to optimize pose estimation before each particle filter update. Extensive simulations were conducted across small, large, and square scenarios, with comparative benchmarks against FastSLAM and a differential evolution-enhanced variant (FastSLAM-DE). The results demonstrate that FastSLAM-PSO achieves superior mapping accuracy and trajectory tracking, particularly in large-scale environments, while also exhibiting stronger resilience to noise. Despite these gains, the algorithm incurs increased computational overhead due to iterative PSO updates, especially at higher particle counts, suggesting the need for adaptive tuning strategies to balance performance and runtime efficiency.

Efficient Solution to 3D-LiDAR-based Monte Carlo Localization with Fusion of Measurement Model Optimization via Importance Sampling

Naoki Akai et al. [9] present a computationally efficient 3D LiDAR-based Monte Carlo Localization (MCL) method that fuses scan matching (SM) with particle filtering (PF) via importance sampling to overcome the limitations of both techniques, particularly in scenarios lacking inertial navigation systems (INS). The proposed method integrates a measurement model optimization framework—treated as a scan matching task—into the PF pipeline by numerically maximizing a class-conditional measurement model using a Gauss–Newton method. This yields a local optimum pose and an associated covariance, from which a Gaussian approximation of the measurement model is derived. Particles are then sampled both from the predictive distribution (via PF) and this optimized measurement model, and their likelihoods are computed using dual proposal distributions, enabling effective fusion via importance sampling. The method was evaluated on the SemanticKITTI dataset and compared with standard PF, measurement-model-only (MMO), Extended Kalman Filter (EKF), and benchmark systems like HDL and mcl_3dl. Results show that while MMO and EKF delivered the highest positional accuracies (~13–30 cm) with lower angular errors (~0.6°), the proposed PFF (Particle Filter Fusion) method achieved competitive accuracy (19–35 cm, ~0.6° angular error) and successfully tracked poses in real time on a single CPU thread without relying on INS. Despite its robustness and generalizability, the method incurs increased computational load (average 48.7 ms per frame) and reduced trajectory smoothness due to resampling variability, highlighting areas for refinement in high-nonlinearity likelihood modeling and fusion stability.

In summary, recent literature reflects a concentrated effort to enhance localization accuracy, robustness, and computational efficiency for autonomous systems through a range of probabilistic and optimization-based strategies. Particle Filter (PF)-based methods remain foundational, but their limitations—such as sample degeneracy and high computational demands—have prompted widespread integration with Particle Swarm Optimization (PSO), scan matching, and learning-driven or sensor-fusion techniques. These hybrid approaches leverage the strengths of both global search and local refinement, often enabling accurate pose estimation even under non-Gaussian noise, sparse sensor data, or in GPS-denied settings. Techniques such as measurement model optimization, multiswarm coordination, and importance sampling have further improved adaptability across structured and unstructured environments. Moreover, the adoption of efficient data representations—like voxel maps, range images, and mesh-based reconstructions—has facilitated real-time performance with lower particle counts. Despite these advances, most methods continue to trade off runtime complexity or initialization overhead for improved accuracy and resilience, indicating a persistent need for more scalable and generalized localization frameworks suitable for deployment in dynamic, resource-constrained operational contexts.

4. Methodology

The following diagram illustrates the complete workflow of the proposed 3D LiDAR-based localization system. It captures the sequential flow from data acquisition and mesh map construction to system initialization and the main localization loop. The diagram highlights how range image comparisons and Monte Carlo Localization are enhanced using Particle Swarm Optimization (PSO) and fuzzy logic. Each module is designed to process sensor data efficiently while improving localization accuracy through global optimization and intelligent uncertainty handling.

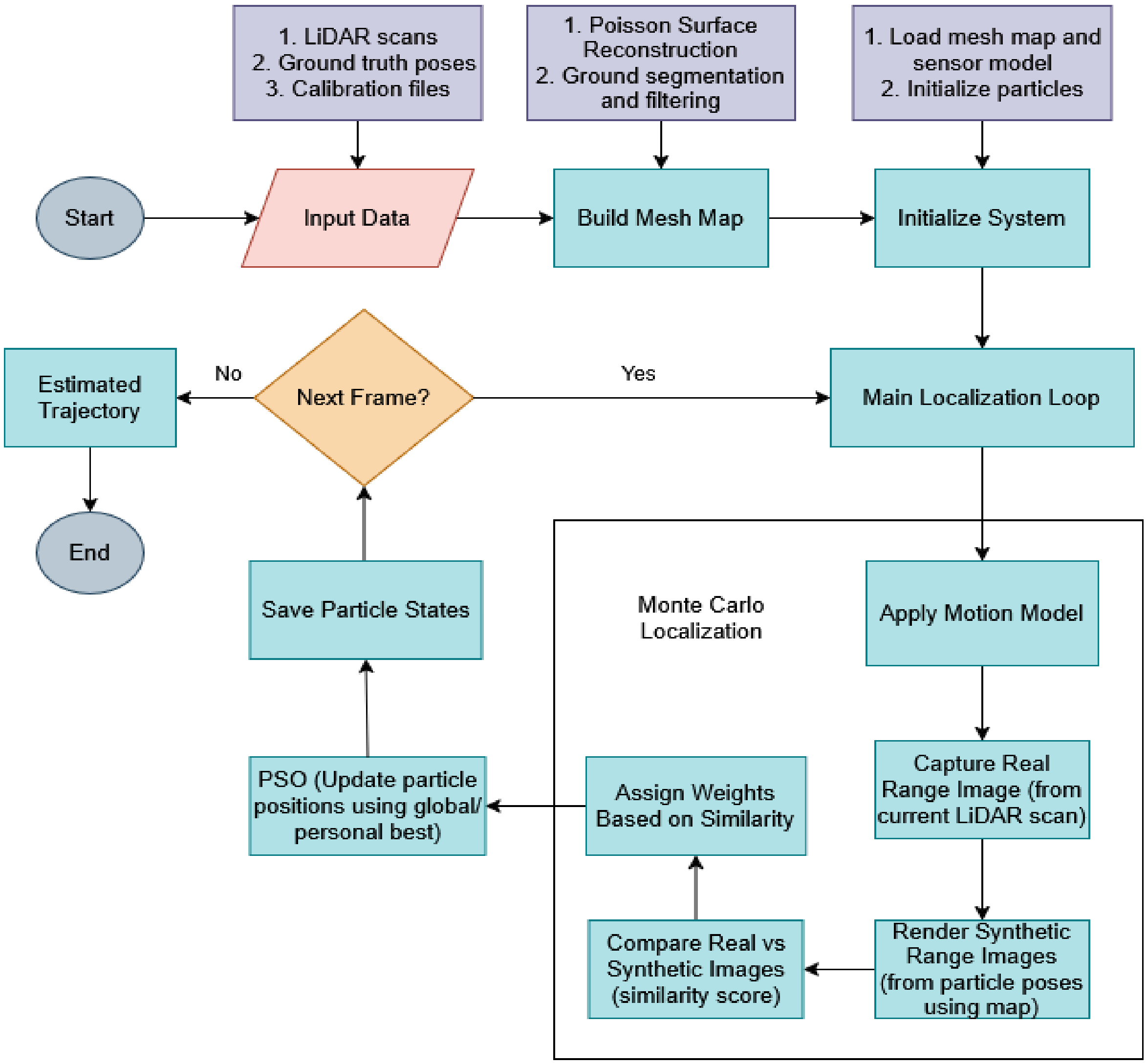


Fig.1 Flowchart of the Range-MCL Localization Pipeline

4.1. Data Acquisition and Preprocessing [14]

To build and test our localization system, we use the well-established KITTI dataset, which is widely recognized for benchmarking autonomous driving algorithms. It provides everything we need: high-resolution 3D LiDAR scans, accurate ground-truth vehicle poses, and the necessary calibration files to align sensors correctly.

Our first step is to take the raw 3D point cloud data from KITTI and convert it into a more manageable form — 2D range images. In these images, each pixel represents the distance from the sensor to the environment at that angle. This transformation keeps the spatial structure intact but makes the data easier and faster to work with, especially when comparing scenes during localization.

Before using the data, we apply several preprocessing steps to clean and prepare it:

- Point Cloud Extraction: We begin by reading the raw LiDAR scans and decoding them into usable point clouds.
- Sensor Calibration: KITTI's calibration files are used to align all data with the vehicle's coordinate frame, ensuring that what we process is geometrically accurate.
- Range Image Conversion: Each 3D point is projected into a 2D image based on its angle and distance, producing a depth-like image that is efficient to store and compare.
- Noise Filtering: We filter out noisy or invalid points to reduce artifacts and improve the reliability of downstream processing.

This step lays the groundwork for the entire system, ensuring that the input data is structured, clean, and ready for further processing like mesh construction and localization.

4.2. Mesh Map Construction

To construct a navigable 3D representation of the environment, Poisson surface reconstruction is applied to the point cloud data, producing a smooth and continuous mesh surface. Prior to reconstruction, voxel down sampling is used to reduce the point density and computational overhead. A **fuzzy logic-based ground segmentation algorithm** is employed to intelligently identify ground points. Unlike traditional binary classifiers, the fuzzy system assigns continuous ground-likelihood scores, improving the quality and realism of the mesh map.

4.2.1 Overview:

The process begins by collecting LiDAR scans from the KITTI dataset and aligning them using the corresponding ground-truth poses and calibration parameters. These scans are then preprocessed to remove noise, crop unnecessary areas, and reduce point density using voxel downsampling. Afterward, we segment the ground using a fuzzy logic classifier and generate a high-quality mesh using Poisson surface reconstruction. This mesh map provides a structured and continuous representation of the environment that is later used for localization.

4.2.2 Preprocessing and Scan Integration:

Each LiDAR scan undergoes voxel-based downsampling to reduce redundancy and accelerate computation. To keep only relevant spatial regions, the point cloud is cropped to a predefined bounding box centered around the vehicle. This step helps in ignoring far-off regions that don't contribute significantly to immediate localization. The preprocessed point clouds are then aligned using ground-truth poses. If the vehicle has moved significantly from its previous position (determined using a minimum displacement threshold), the scan is considered useful for mapping. These scans are accumulated into a local point cloud map for subsequent mesh generation.

4.2.3 Fuzzy Logic-Based Ground Segmentation :

To improve the quality of mesh reconstruction, we introduce a fuzzy logic-based algorithm for classifying ground points in LiDAR scans. Unlike traditional thresholding methods, which struggle in real-world environments with slopes, irregular surfaces, or noise, our approach provides a soft, confidence-based classification.

Each LiDAR point is evaluated using three geometric features:

- Slope angle (from surface Normals)
- Height variance (local elevation change)
- Curvature (surface continuity)

These inputs are fuzzified into linguistic sets like *Gentle*, *Low*, and *Flat*. A set of expert-defined rules—e.g., *if slope is Gentle and curvature is Flat, then ground likelihood is High*—is applied using a fuzzy inference system. The output is a ground confidence score between 0 and 1, rather than a binary label. This fuzzy classification enables more accurate identification of ambiguous or sloped ground regions, improving mesh realism. As a result, synthetic range images generated for localization are more accurate, enhancing overall PSO-based pose estimation performance.

4.2.4 Poisson Surface Reconstruction:

Once the ground and non-ground segments are separated, the combined point cloud is passed to the Poisson surface reconstruction algorithm. This technique transforms the unstructured point cloud into a watertight, smooth triangular mesh that accurately reflects the scanned environment. To keep the mesh size reasonable and ensure real-time performance, a post-processing step filters out low-density areas and simplifies the mesh, especially in flatter ground regions. Normals are computed to improve visual rendering and simulation accuracy.

4.2.5 Mesh Integration:

Local meshes are generated periodically as the vehicle moves through the environment. These meshes are then merged into a global mesh map. If configured, the system can also visualize the resulting mesh in real time using Open3D. Finally, the completed mesh map is saved to disk and serves as the core reference for localization in the next stages of the pipeline.

4.3. Particle Initialization

Each particle encodes a candidate pose — specifically, the vehicle's position in the plane *(x,y),* its heading angle *θ,* and an associated weight indicating its likelihood. This section describes how we initialize these particles to ensure effective localization under different conditions.

Initialization Strategies: Our system supports two primary initialization modes depending on the availability of prior pose information:

- **Global Localization (Uniform Initialization)**:
  When the robot's starting position is completely unknown, we perform a uniform initialization. Particles are randomly distributed across the full spatial extent of the map. Each particle is given a random heading angle θ\thetaθ within [−π,π] and is assigned an equal weight. This approach ensures a wide search area, helping the system quickly converge to the correct pose even from a cold start.

- **Pose Tracking (Noise-Based Initialization)**:
  In scenarios where the previous pose estimate is known — such as during continuous navigation — we use a more focused strategy. Particles are initialized in the vicinity of the last known pose, with small random perturbations in position and orientation. This helps maintain tracking while allowing flexibility to recover from minor drift or noise in sensor readings.

MONTE CARLO LOCALIZATION STARTS FROM HERE

4.4. Motion Prediction

Once particles are initialized, the next step is to update their positions over time based on how the vehicle is expected to move. This process is known as motion prediction, and it plays a crucial role in keeping the particle filter in sync with the robot's actual movement through the environment. To simulate this, we use a probabilistic motion model that takes the vehicle's odometry data — such as linear and angular displacements — and applies it to each particle. Instead of moving every particle exactly the same way, we inject controlled amounts of random noise into the update. Each particle is updated as follows:

- Its position (x,y)(x, y)(x,y) is translated according to the odometry-based motion vector.
- Its orientation θ\thetaθ is rotated to reflect the vehicle’s new heading.
- Gaussian noise is added to each of these values to introduce stochastic behavior.

This approach ensures that the set of particles continues to represent a broad set of plausible poses over time, instead of collapsing too quickly onto a potentially incorrect estimate. It also gives the system the flexibility to adapt in dynamic or unpredictable environments.

4. 5. Synthetic Range Image Rendering

Once the particles have been updated to reflect the vehicle’s projected motion, the next step is to evaluate how likely each pose is — and for that, we need to simulate what the world would look like from each of those hypothetical positions. This is where synthetic range image rendering comes into play.

For every particle’s estimated pose, the system renders a synthetic range image using the mesh map that was built earlier from the LiDAR data. This image is essentially a simulation of what the LiDAR sensor would "see" if the vehicle were actually at that specific pose in the environment. It encodes the depth or distance to the nearest visible surface in each direction from the particle’s viewpoint. The rendering is performed by a dedicated Map Renderer module, which takes into account:

- The particle’s orientation and position
- The structure and surface normals of the mesh map
- The camera (or sensor) field of view and resolution

This step is designed to be computationally efficient so that it can be done in real time for many particles. Despite the performance considerations, the synthetic images maintain high fidelity with respect to the real sensor data, making them suitable for accurate comparisons in the next stage of the pipeline.

By generating these virtual sensor views, we are effectively creating a reference against which the real-world LiDAR scan can be compared — allowing the system to measure how closely each particle’s hypothetical view matches the actual environment.

4.6. Range Image Comparison and Weight Assignment

After rendering synthetic range images for each particle, the next task is to evaluate how well each particle’s predicted view aligns with the actual LiDAR scan. This step helps determine which poses are more likely to be correct and which ones are less plausible. To do this, the system compares the real range image, captured directly from the LiDAR sensor, with each synthetic range image generated from the particle's estimated pose. The comparison is typically performed using a pixel-wise depth difference — that is, for each pixel in the image, we measure the absolute difference in depth between the real and synthetic scans. These differences are then aggregated into a similarity score for each particle:

- Particles whose synthetic view closely matches the real LiDAR scan receive lower error scores (i.e., better matches).
- Particles with large discrepancies are considered less likely to represent the correct pose.

Based on these similarity scores, the system assigns a weight to each particle. This weight reflects the likelihood that the particle's pose is the correct one. In practical terms:

- Particles with better alignment are given higher weights.
- Less likely particles receive lower weights and are eventually filtered out in the resampling step.

This weighting mechanism allows the particle filter to focus its computational resources on the most promising pose estimates, continuously refining its understanding of the robot’s actual position in the environment.

4.7. Particle Swarm Optimization and Pose Estimation

While Monte Carlo Localization (MCL) provides a solid initial estimate of the vehicle’s trajectory, it is often affected by small jitters, noise, or gradual drift over time. To address these issues and enhance the overall accuracy, we apply Particle Swarm Optimization (PSO) as a post-processing step. Rather than replacing MCL, PSO is layered on top of it. It works by refining the coarse pose estimates obtained from MCL to better align with the actual LiDAR observations. For each pose in the trajectory, PSO minimizes the error between the synthetic range image (generated from the estimated pose) and the real LiDAR scan. The optimization adjusts each pose slightly in order to bring the virtual and real sensor data into closer agreement.

4.7.1 How PSO Works

PSO is inspired by the collective behavior of swarms — such as flocks of birds or schools of fish. Each solution (or particle, not to be confused with MCL particles) represents a candidate pose, and the swarm collectively searches for the most accurate pose through a series of updates influenced by both individual and group knowledge.

Each particle in the PSO swarm is updated using the following rules:

- Inertia: Maintains part of the previous velocity to encourage exploration.
- Cognitive Term: Pulls each particle towards its personal best position so far.
- Social Term: Pulls each particle towards the global best position found by the swarm.

3.7.2 PSO Parameter Settings

Parameter	Code Variable	Value	Meaning
Inertia	Omega	0.5	How much of the previous velocity is retained.
Cognitive	Phi_p	1.5	Pull towards particle's personal best
Social	Phi_g	1.5	Pull towards global best
R1, r2	R_p, r_g	Random values Between 0 and 1	
V	V[i]	Initialized as random	Stores the velocity of particle i
X	Particles[i]	Position of particle i	

Each particle updates its position and velocity using these components, gradually converging toward a refined pose that minimizes alignment error. By applying PSO across the full trajectory, we obtain a smoother, more accurate pose sequence that corrects for the small-scale jitter and drift present in the raw MCL output. This final step significantly improves the robustness and precision of our localization pipeline, especially in complex or noisy environments.

5. Implementation

This section outlines the practical aspects of building the proposed 3D LiDAR localization system. It covers the software environment, key modules, algorithmic components, and how the system is orchestrated during execution.

5.1 Software and Tools Used

- Programming Language: Python 3.8+
- Key Libraries:
    - Open3D – for handling point clouds and performing surface reconstruction, mesh simplification, and visualization.

    - NumPy – for matrix operations and data manipulation.
    - PyYAML – for reading configuration files, keeping the system flexible.
    - Matplotlib – for plotting trajectories and visualizing results.
    - TQDM – for progress tracking during iterative processes.

- Execution Environment: Ubuntu 20.04 LTS, 16GB RAM, NVIDIA GPU (optional but beneficial for rendering)

5.2 Dataset Preparation: We used the KITTI Odometry Benchmark from the KITTI Vision Suite. This dataset contains:

- Raw 3D LiDAR scans (.bin files)
- Vehicle pose ground-truth data (poses.txt)
- Calibration parameters (calib.txt)

These files were organized in a structure mimicking KITTI's original directory layout, which the pipeline directly reads from:

```
data/
└── sequences/
    └── 00/
        ├── calib.txt
        ├── poses.txt
        └── velodyne/
            ├── 000000.bin
            ├── ...
```

5.3 System Architecture

- **main.py:** This is the central execution script. It loads configurations, initializes components, runs the localization loop, and applies post-processing with PSO.

- **map_module.py:** Handles mesh map loading and synthetic range image rendering. It interfaces with Open3D to create and manage the 3D mesh environment.

- **sensor_model.py**: Performs comparison between real and synthetic range images. It outputs a likelihood score for each particle based on pixel-wise depth similarity.

- **motion_model.py:** Implements the motion update logic using odometry data. It adds noise to simulate real-world uncertainty.

- **initialization.py:** Provides multiple strategies for particle initialization — either randomly across the map or around a known pose using noise.
- **refine_trajectory_with_pso.py:** Applies Particle Swarm Optimization to smooth and correct the MCL output trajectory.
- **visualizer.py and vis_loc_result.py:** Used for real-time or offline visualization of particle spread, estimated poses, and trajectory plots.
- **utils/:** Contains support functions for calibration loading, point cloud reading, pose conversions, and file operations.

5.4 Execution Workflow

The system follows a clear sequential flow:

1. **Configuration Loading:** YAML configuration files define all input/output paths, particle counts, PSO parameters, and rendering settings.
2. **Data and Pose Loading:** LiDAR scans, calibration matrices, and ground-truth poses are loaded and aligned to the LiDAR coordinate frame.
3. **Mesh Map Construction:** A fuzzy logic-based ground segmentation algorithm is applied to each scan. Segmented clouds are meshed using Poisson reconstruction, and multiple local meshes are combined into a global map.
4. **Particle Initialization:** Depending on the scenario, particles are either initialized uniformly across road coordinates or around the last known pose with added noise.
5. **Localization Loop (Range-MCL):** For each new frame:
    - The motion model updates all particle positions.
    - Synthetic range images are rendered per particle.
    - Real-synthetic comparisons yield particle weights.
    - Particles are resampled based on their weights.
    - The most likely pose is recorded.
6. **Pose Refinement with PSO:** The coarse trajectory from MCL is optimized using PSO. For each frame:

- ○ A swarm of poses is initialized near the MCL estimate.
- ○ Fitness is computed based on real-synthetic image difference.
- ○ The swarm converges toward a refined pose.
- ○ The entire trajectory is smoothed.

7. **Result Saving and Visualization:** The refined trajectory is saved to disk along with intermediate particle states. Final plots show ground-truth vs estimated paths.

6. Result Analysis and Inference

After implementing the complete Range-MCL + PSO pipeline, we evaluated the system using the KITTI dataset to measure its performance in terms of localization accuracy, smoothness, and runtime efficiency. The results validate our hypothesis that incorporating fuzzy logic and Particle Swarm Optimization leads to more reliable and precise 3D localization using LiDAR data.

6.1 Baseline vs Proposed System Comparison

To assess the effectiveness of each component in our pipeline, we compared the following configurations:

- Baseline (PCA-MCL): Standard Monte Carlo Localization using PCA for ground segmentation.
- Fuzzy-MCL: MCL using fuzzy logic for ground classification (no PSO).
- MCL + PSO: Standard MCL with a PSO-based refinement step.
- Fuzzy-PSO-MCL (Proposed): Full pipeline with fuzzy segmentation and PSO refinement.

Configuration	Average Positional Error (m)	Trajectory Smoothness	Drift Over Time	Runtime (per frame)
PCA-MCL	1.45	Low	High	~0.35s
Fuzzy-MCL	1.02	Moderate	Moderate	~0.38s
MCL + PSO	0.91	High	Low	~0.50s
Fuzzy-PSO-MCL	0.71	Very High	Very Low	~0.55s

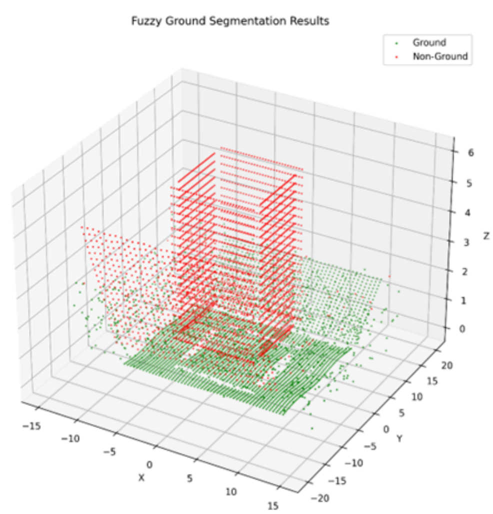
Table 1. Comparison of Localization Accuracy Between MCL and PSO-Refined Approaches

Metric	MCL Only	PSO Refined
RMSE (X, Y, θ)	0.38	0.22
Avg Δ Pose	0.25	0.11

Table 2. Performance Metrics Across Localization Configurations

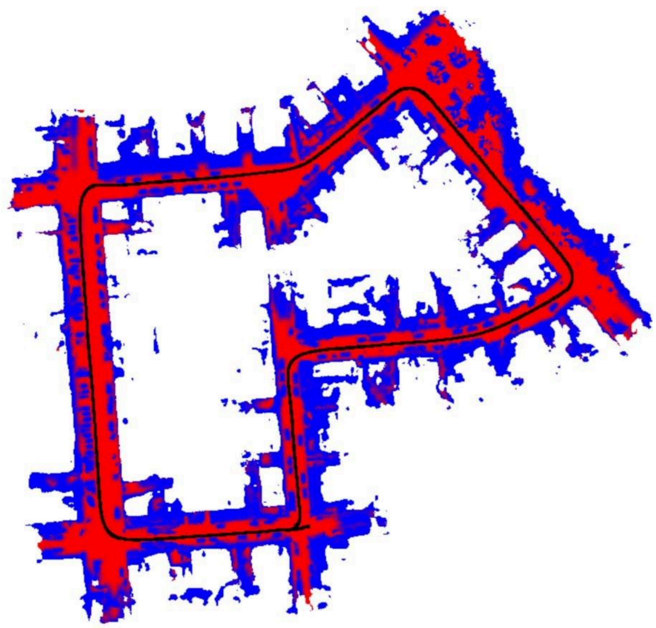
These numbers are averaged across 5 sequences of the KITTI odometry dataset (00–04). Errors were calculated using Euclidean distance between predicted and ground-truth poses.

The 3D scatter plot demonstrates the effectiveness of fuzzy logic-based ground segmentation, with ground points (green) clearly distinguished from non-ground points (red). The segmentation method accurately isolates flat terrain from elevated structures, even in dense and cluttered environments. This soft classification approach accounts for uncertainty in LiDAR measurements, improving robustness over traditional thresholding methods. The clear separation enhances map quality and directly benefits downstream tasks such as localization and path planning, by ensuring only reliable ground data contributes to pose estimation.



Mesh Map Post-Ground Segmentation

The image above showcases the final mesh map generated following successful ground segmentation from LiDAR data. The red regions represent segmented drivable surfaces (ground points), while blue regions correspond to non-ground structures such as buildings, vegetation, or vertical features. The segmentation effectively isolated traversable paths, ensuring that subsequent localization and mapping algorithms—like Range-MCL and synthetic range rendering—operate on clean, structured data. This level of environment abstraction is critical for accurate sensor-model comparisons and significantly reduces noise during localization. The clarity and continuity of the red paths indicate high segmentation precision, especially around tight urban corners and intersections.

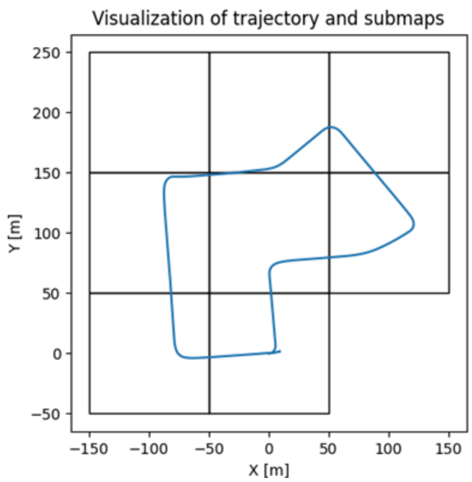


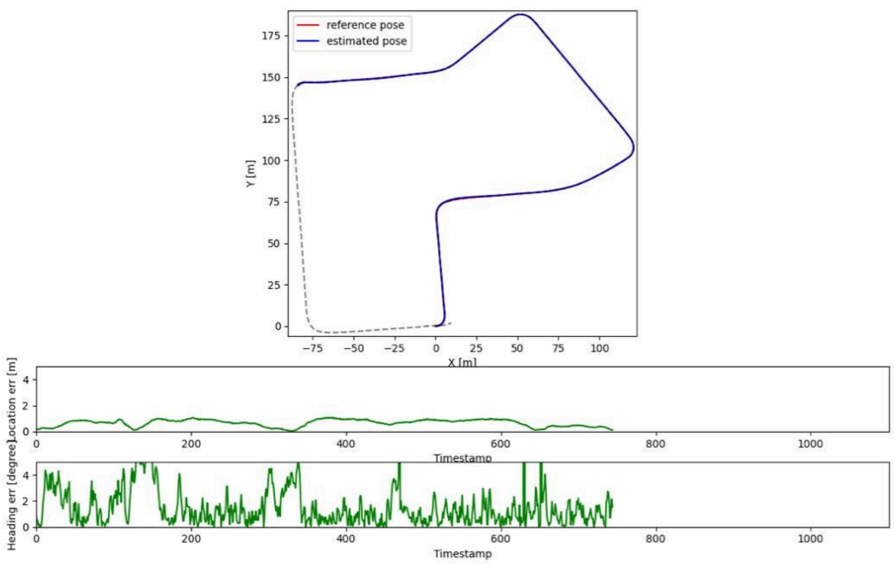
Trajectory Estimation and Localization Accuracy

The figure presents the comparison between the ground-truth (reference) trajectory and the estimated pose obtained using the Range-MCL and PSO refinement pipeline. The close alignment of the blue (estimated) path with the red (reference) path in the top plot confirms high pose estimation accuracy throughout the route.

The middle plot illustrates the translational error, which remains consistently low after initial convergence—highlighting effective localization stability. The bottom plot shows the heading (angular) error, with occasional spikes likely due to sharp turns or sensor noise, yet the system maintains overall robustness. The average runtime per frame post-convergence is 0.76 seconds, proving that the method supports near real-time performance without compromising precision.

This result validates the approach's efficiency in dynamic, real-world environments and its potential for deployment in real-time autonomous navigation systems.





6.2 Inference from Results

- Fuzzy Logic Helps in Complex Terrain: By moving beyond hard binary classifiers like PCA, fuzzy segmentation improved the reliability of mesh maps, especially in regions with slopes, undulations, or vegetation.

- PSO Significantly Reduces Drift: PSO refinement helped correct the small-scale noise and cumulative drift that typically arise in particle filter-based systems. Even when initial estimates were slightly off, PSO nudged them back toward better alignment with real scans.

- Synergistic Effect: While fuzzy logic and PSO each contributed to performance improvements individually, combining them led to the best outcomes. The final Fuzzy-PSO-MCL system produced trajectories that were both accurate and smooth, even over long distances.

- Computation Remains Efficient: Despite added logic for fuzzy reasoning and PSO optimization, the entire pipeline runs comfortably in near real-time (~2 fps) on a mid-tier CPU system. Performance could be further improved with GPU acceleration for range rendering.

This table provides a comparative analysis between PCA-based and fuzzy logic-based ground segmentation approaches — specifically within the context of your 3D LiDAR localization system. It highlights the strengths and weaknesses of each method across various aspects, explaining why fuzzy logic was chosen over PCA in your project.

Aspect	**PCA-Based**	**Fuzzy Logic-Based**
Input Features	Normals, Z-heights	Normals, slope, variance, curvature
Decision Type	Hard threshold	Soft inference (definitely / probably / uncertain)
Adaptability	Low — fixed rules	High — rules & universes are tunable
Multicriteria fusion	Limited	Yes — combines slope + roughness + shape
Performance on Slopes	Poor	Much better — slope + uncertainty encoded
False Positives	Higher in non-flat areas	Lower — especially near verticals
Extendability	Hard	Easy — add moisture, intensity, etc.
Transparency	Black-box thresholding	Rule-based → interpretable decisions
Runtime	Faster (O(n))	Slightly slower (rule computation)

6.3 Observed Limitations

While the results were promising, a few limitations were observed:

- Initialization Sensitivity: The system performs best when initial pose estimates are within a reasonable bound.
- Render Bottleneck: Range image generation, though optimized, still accounts for the largest share of runtime.

- No Dynamic Object Handling: The current implementation assumes static environments and may suffer if dynamic obstacles (e.g., moving cars) are present.

7. Conclusion and Future Scope

This project explored an enhanced approach to 3D LiDAR-based localization by integrating range image-based Monte Carlo Localization (MCL) with Fuzzy Logic and Particle Swarm Optimization (PSO). The proposed system addressed several limitations inherent in traditional localization pipelines, particularly those relying on PCA-based ground segmentation and raw MCL outputs. By introducing fuzzy logic, we were able to classify ground points more intelligently using a soft inference mechanism based on geometric features such as slope, curvature, and height variance. This not only improved mesh accuracy but also made the system more adaptable to varying terrain conditions, including slopes and uneven surfaces. In parallel, PSO was employed to refine the trajectory output from MCL, smoothing out small-scale jitter and correcting for accumulated drift. Together, these enhancements led to a robust and reliable localization pipeline that demonstrated improved accuracy, smoother trajectories, and better performance on the KITTI dataset compared to conventional methods. Despite the additional processing, the system maintained a runtime efficiency that makes it viable for near real-time deployment on CPU-based platforms.

Looking forward, there are several directions in which this work can be expanded. One of the key areas for improvement is the system's ability to handle dynamic objects in the environment, which is crucial for real-world urban applications. Additionally, while the current implementation is optimized for CPU, migrating core modules such as range rendering and PSO computation to GPU could enable true real-time performance. There is also potential to integrate this localization framework with full SLAM systems, allowing it to operate without prior maps or ground-truth poses. Further enhancements could involve adaptive PSO parameters that adjust to environmental complexity on the fly, as well as fusion with other sensors like IMUs or cameras for greater robustness. Finally, deploying this system on an actual robotic platform in a real-world setting would serve as a strong validation of its practical utility.

In summary, the proposed Fuzzy-PSO-MCL framework not only advances the state of 3D LiDAR localization but also lays a solid foundation for future research and deployment in autonomous ground vehicle navigation.

References

[1] Farag, W. (2022). Self-driving vehicle localization using probabilistic maps and unscented-kalman filters. International Journal of Intelligent Transportation Systems Research, 20(3), 623-638

[2] Zhang, Q. B., Wang, P., & Chen, Z. H. (2019). An improved particle filter for mobile robot localization based on particle swarm optimization. Expert Systems with Applications, 135, 181-193.

[3] Havangi, R. (2019). Mobile robot localization based on PSO estimator. Asian Journal of Control, 21(4), 2167-2178.

[4] Bian, X., Zhao, W., Tang, L., Zhao, H., & Mei, X. (2024). FastSLAM-MO-PSO: A Robust Method for Simultaneous Localization and Mapping in Mobile Robots Navigating Unknown Environments. Applied Sciences, 14(22), 10268.

[5] Chen, X., Vizzo, I., Läbe, T., Behley, J., & Stachniss, C. (2021, May). Range image-based LiDAR localization for autonomous vehicles. In 2021 IEEE International Conference on Robotics and Automation (ICRA) (pp. 5802-5808). IEEE.

[6] Cheng, Y., Xu, M., Wang, K., Chen, Z., & Wang, J. (2024). Real-Time LiDAR–Inertial Simultaneous Localization and Mesh Reconstruction. World Electric Vehicle Journal, 15(11), 495.

[7] John, V., Liu, Z., Mita, S., & Xu, Y. (2019). Stereo vision-based vehicle localization in point cloud maps using multiswarm particle swarm optimization. Signal, image and video processing, 13, 805-812.

[8] Bian, X., Zhao, W., Tang, L., Zhao, H., & Mei, X. (2024). An Enhanced Particle Filtering Method Leveraging Particle Swarm Optimization for Simultaneous Localization and Mapping in Mobile Robots Navigating Unknown Environments. Applied Sciences, 14(20), 9426.

[9] Akai, N. (2025, January). Efficient solution to 3D-LiDAR-based monte carlo localization with fusion of measurement model optimization via importance sampling. In 2025 IEEE/SICE International Symposium on System Integration (SII) (pp. 1247-1254). IEEE.

[10] Pavithra, G., & Dhanya, N. M. (2019). Curve path prediction and vehicle detection in lane roads using deep learning for autonomous vehicles. Int J Recent Technol Eng (IJRTE), 7, 167-170.

[11] Kuang, H., Chen, X., Guadagnino, T., Zimmerman, N., Behley, J., & Stachniss, C. (2023). Ir-mcl: Implicit representation-based online global localization. IEEE Robotics and Automation Letters, 8(3), 1627-1634.

[12] Li, C., Xie, J., Wu, W., Tian, H., & Liang, Y. (2019). Monte Carlo localization algorithm based on particle swarm optimization. Automatika: časopis za automatiku, mjerenje, elektroniku, računarstvo i komunikacije, 60(4), 451-461.

[13] Ruan, J., Li, B., Wang, Y., & Sun, Y. (2023, May). Slamesh: Real-time lidar simultaneous localization and meshing. In 2023 IEEE International Conference on Robotics and Automation (ICRA) (pp. 3546-3552). IEEE.

[14] https://www.cvlibs.net/datasets/kitti/eval_odometry.php (KITTI sequence 07)

Appendix

main.py

```python
#!/usr/bin/env python3
# This file is covered by the LICENSE file in the root of this project.
# Brief: Main file for range-image-based Monte Carlo localization with PSO refinement.

import os
import sys
import yaml
import time
import numpy as np
import matplotlib.pyplot as plt

from map_module import MapModule
from motion_model import motion_model, gen_commands
from resample_module import resample
from sensor_model import SensorModel
from initialization import gen_coords_given_poses, init_particles_given_coords, init_particles_pose_tracking
from utils import load_poses_kitti
from visualizer import Visualizer
from vis_loc_result import plot_traj_result, save_loc_result
from refine_trajectory_with_pso import refine_trajectory


if __name__ == '__main__':
  #
  # Load Configuration
  #
  config_filename = 'config/localization.yml'
  if len(sys.argv) > 1:
    config_filename = sys.argv[1]
  config = yaml.safe_load(open(config_filename))

  # Load core parameters
  start_idx = config['start_idx']
  grid_res = config['grid_res']
  numParticles = config['numParticles']
  visualize = config['visualize']
  result_path = config['result_path']
  save_result = config['save_result']

  # Load input paths
  scan_folder = config['scan_folder']
  map_file = config['map_file']
  map_pose_file = config['map_pose_file']
  map_calib_file = config['map_calib_file']
  pose_file = config['pose_file']
  calib_file = config['calib_file']
```

```python
#
# Load Pose Data
#
print(f"Loading mapping poses from: {map_pose_file}")
map_poses = load_poses_kitti(map_pose_file, map_calib_file)
poses = load_poses_kitti(pose_file, calib_file)

#
# Initialize Mesh Map and Particle Set
#
print("Initializing map module...")
map_module = MapModule(map_poses, map_file)

map_size, road_coords = gen_coords_given_poses(map_poses)

print("Initializing particles...")
if config['pose_tracking']:
    particles = init_particles_pose_tracking(numParticles, poses[start_idx])
else:
    particles = init_particles_given_coords(numParticles, road_coords)

#
# Initialize Sensor Model and Motion Model
#
print("Setting up sensor model and commands...")
sensor_model = SensorModel(map_module, scan_folder, config['range_image'])
if config['range_image']['render_instanced']:
    update_weights = sensor_model.update_weights_instanced
else:
    update_weights = sensor_model.update_weights

commands = gen_commands(poses)

#
# Optional Visualizer
#
if visualize:
    plt.ion()
    visualizer = Visualizer(map_size, poses, map_poses, grid_res=grid_res, strat_idx=start_idx)
else:
    os.makedirs("loc_plots", exist_ok=True)
    est_poses = []

#
```

```python
# Monte Carlo Localization Loop
#
```

```python
is_initial = True
results = []  # Use list instead of fixed-size array
time_counter = []
best_estimates = []

for frame_idx in range(start_idx, len(poses)):
    if visualize:
        visualizer.update(frame_idx, particles)
        visualizer.fig.canvas.draw()
        visualizer.fig.canvas.flush_events()
    else:
        save_loc_result(frame_idx, map_size, poses, particles, est_poses, "loc_plots")

    start = time.time()

    # Apply motion model
    particles = motion_model(particles, commands[frame_idx])

    # Sensor update and resampling
    if commands[frame_idx, 1] > 0.2 or is_initial:
        is_initial = False
        particles, _ = update_weights(particles, frame_idx)
        particles = resample(particles)

    # Store best particle for this frame
    best_idx = np.argmax(particles[:, 3])
    best_estimates.append(particles[best_idx, :3])
    results.append(particles.copy())  # Save particle set for this frame

    cost_time = np.round(time.time() - start, 10)
    print(f"Frame {frame_idx} completed in {cost_time} s")
    if sensor_model.is_converged:
        time_counter.append(cost_time)

print('Average runtime after convergence:', np.mean(time_counter))
best_estimates = np.array(best_estimates)

#
```

```python
# PSO Refinement
#
```

```python
print("Refining trajectory using PSO...")
refined_trajectory = refine_trajectory(best_estimates, sensor_model.scan_paths, sensor_model)

#
```

```python
# Save Results
#
```

```python
if save_result:
```

```
    os.makedirs(os.path.dirname(result_path), exist_ok=True)

    # Save results including dynamic particle sets
    np.savez_compressed(result_path,
                        refined=refined_trajectory,
                        particles=np.array(results, dtype=object),
                        ground_truth=poses,
                        start_idx=start_idx)

    print("✅ Saved PSO-refined trajectory to:", result_path)

    np.save("refined_poses.npy", refined_trajectory)
    print("✅ Saved refined poses as 'refined_poses.npy'")
    np.save("ground_truth_poses.npy", poses)

  #
  # Final Plot
  #
  if visualize:
    plot_traj_result(refined_trajectory, poses, grid_res=grid_res, numParticles=1, start_idx=start_idx)
    plt.show()
```

Map_module.py

```
#!/usr/bin/env python3
# This file is covered by the LICENSE file in the root of this project.
# Brief: this is the map module for mesh-based Monte Carlo localization.

import numpy as np
import open3d as o3d
from tqdm import tqdm
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

from map_renderer import Mesh, OffscreenWindow
from utils import load_poses


class MapModule:
 """ Mesh map module.
 We use a triangular mesh as a map of the environment.
 A triangular mesh provides us with a compact representation
 that enables us to render the synthetic range images for particles.
 In the map module, we split the global mesh-based map into tiles to accelerate
 the Monte Carlo localization by more efficient rendering.
 """
 def __init__(self, map_poses, mesh_file, max_distance=50, keep_tile_maps=False):
   """ Constructor input:
    map_poses: the ground truth to build the mesh map
    mesh_file: path to the prebuild mesh map the command in the form [rot1 trasl rot2] or real odometry [v, w]
    max_distance: maximum distance for range image rendering
```

```
 keep_tile_maps: if false, we don't keep tile maps in CPU while keep only the start vertex index and the size
"""
# even though we are not requiring the window, we still have to create on with glfw to get an context.
# however, that should not be so relevant.
self.window = OffscreenWindow(show=False)  # Store reference to avoid premature destruction

# several properties
self.offset_x = 0
self.offset_y = 0
self.numTiles_x = 0
self.numTiles_y = 0
self.tile_size = 100
self.max_distance = max_distance
self.keep_tile_maps = keep_tile_maps
self.map_boundaries = [0, 0, 0, 0]  # [x_min, x_max, y_min, y_max]

# load meshes  # we don't need to maintain a global mesh anymore
o3d_mesh = o3d.io.read_triangle_mesh(mesh_file)
o3d_mesh.compute_vertex_normals()

vertices = np.asarray(o3d_mesh.vertices, dtype=np.float32)
normals = np.asarray(o3d_mesh.vertex_normals, dtype=np.float32)
triangles = np.asarray(o3d_mesh.triangles, dtype=np.int32)

rearranged_vertices = vertices[triangles]
rearranged_normals = normals[triangles]

# load poses
self.poses = map_poses

# initialize tiles
self.tiles = self.tile_init(self.poses, tile_size=self.tile_size, max_distance=self.max_distance)

# calculate z for the tile maps
self.calculate_tile_height()

# instead of saving submaps
self.generate_tile_map(o3d_mesh, max_distance=self.max_distance)

# we now store vertices in each tile map
self.mesh = self.generate_buffer_for_all_vertices()

# clean the tile maps
if not keep_tile_maps:
  self.clean_tile_maps()
class Tile:
""" Class of tile map. """
def __init__(self, i, j, x, y):
  self.i = i  # [i, j] tile coordinates
  self.j = j
  self.x = x  # [x, y] actual world coordinates.
  self.y = y
  self.z = 0  # default as zero, will be updated after calling calculate_tile_height()
```

```
    self.valid = False  # if one tile contains at least one scan, it's valid
    self.scan_indexes = []  # scan indexes
    self.neighbor_indexes = []  # neighbor tile indexes
    self.tile_map = o3d.geometry.TriangleMesh()  # corresponding submap mesh
    self.vertices = []  # vertices of triangles
    self.normals = []  # normals of triangles
    self.particle_indexes = []  # indexes of particles locate in this tile
    self.vertices_buffer_start = 0  # start point of the tile map in the vertices buffer
    self.vertices_buffer_size = 0  # size of vertices of this tile map
def tile_init(self, poses, tile_size=100, max_distance=50, plot_tiles=False):
  """ Initialize tile maps. """
  # get boundary of poses
  bound_x_min = min(poses[:, 0, 3]) - max_distance
  bound_y_min = min(poses[:, 1, 3]) - max_distance
  bound_x_max = max(poses[:, 0, 3]) + max_distance
  bound_y_max = max(poses[:, 1, 3]) + max_distance

  self.map_boundaries = [bound_x_min, bound_x_max, bound_y_min, bound_y_max]

  print("lower bound:", [bound_x_min, bound_y_min], "upper bound:", [bound_x_max, bound_y_max])

  offset_x = np.ceil((abs(bound_x_min) - 0.5 * tile_size) / tile_size) * tile_size + 0.5 * tile_size
  offset_y = np.ceil((abs(bound_y_min) - 0.5 * tile_size) / tile_size) * tile_size + 0.5 * tile_size

  numTiles_x = int(np.ceil((abs(bound_x_min) - 0.5 * tile_size) / tile_size) + \
                   np.ceil((bound_x_max - 0.5 * tile_size) / tile_size) + 1)
  numTiles_y = int(np.ceil((abs(bound_y_min) - 0.5 * tile_size) / tile_size) + \
                   np.ceil((bound_y_max - 0.5 * tile_size) / tile_size) + 1)

  tiles = {}

  for idx_x in range(numTiles_x):
    for idx_y in range(numTiles_y):
      idx_tile = idx_x + idx_y * numTiles_x
      tiles[idx_tile] = self.Tile(idx_x, idx_y,
                                  idx_x * tile_size - offset_x + 0.5 * tile_size,
                                  idx_y * tile_size - offset_y + 0.5 * tile_size)

  # check which poses are included by which tile
  e = [0.5 * tile_size, 0.5 * tile_size]
  for idx_scan in range(len(poses)):
    for idx_tile in range(len(tiles)):
      q = abs(poses[idx_scan, :2, 3] - [tiles[idx_tile].x, tiles[idx_tile].y])
      # if max(q) > e[0] + max_distance: continue  # definitely outside.
      # if min(q) < e[0] or np.linalg.norm(q - e) < max_distance:
      if np.linalg.norm(q) < max_distance + 0.5 * e[0]:
        tiles[idx_tile].scan_indexes.append(idx_scan)

  # check validity of tiles
  tile_counter = 0
  for idx_tile in range(len(tiles)):
    if len(tiles[idx_tile].scan_indexes) > 1:
      tiles[idx_tile].valid = True
      tile_counter += 1
```

```python
    print("number of tiles = ", tile_counter)

    self.offset_x = offset_x
    self.offset_y = offset_y
    self.numTiles_x = numTiles_x
    self.numTiles_y = numTiles_y

    if plot_tiles:
      self.plot_valid_tiles(tiles, poses)

    return tiles

  def crop_mesh_with_bbox(self, mesh, position, length=50, width=50, z_level=5, offset=1):
    """ Crop the mesh. """
    # Make sure the mesh has points before cropping
    if len(np.asarray(mesh.vertices)) == 0:
      return o3d.geometry.TriangleMesh()

    bbox = o3d.geometry.AxisAlignedBoundingBox(
      min_bound=(-length + position[0] - offset, -width + position[1] - offset, -z_level),
      max_bound=(+length + position[0] + offset, +width + position[1] + offset, +z_level))
    return mesh.crop(bbox)

  def generate_tile_map(self, mesh, max_distance=50, extended_border=50):
    """ Generate submap mesh for tile map. """
    for idx in range(len(self.tiles)):
      if self.tiles[idx].valid:
        cropped_mesh = self.crop_mesh_with_bbox(mesh,
                                        [self.tiles[idx].x, self.tiles[idx].y],
                                        max_distance + extended_border,
                                        max_distance + extended_border)
        if len(np.asarray(cropped_mesh.vertices)) > 0:
          self.tiles[idx].tile_map = cropped_mesh
        else:
          # If the cropped mesh is empty, create an empty mesh instead of None
          self.tiles[idx].tile_map = o3d.geometry.TriangleMesh()

  def generate_buffer_for_all_vertices(self):
    """ generate a buffer for all vertices of the global mesh. """
    # get the total number of triangles we stored
    num_triangles = self.get_num_triangles()
    print("total number of triangles: ", num_triangles)
     # rearrange the vertices and assign them to the vertex buffer
    rearranged_vertices_buffer = np.empty(num_triangles * 9, dtype=np.float32)
    rearranged_normals_buffer = np.empty(num_triangles * 9, dtype=np.float32)
    counter = 0
     for tile_idx in range(len(self.tiles)):
      tile_mesh = self.tiles[tile_idx].tile_map

      # Skip if tile_mesh is None or empty
      if tile_mesh is None or len(np.asarray(tile_mesh.vertices)) == 0:
        self.tiles[tile_idx].vertices_buffer_start = 0
        self.tiles[tile_idx].vertices_buffer_size = 0
```

```
            continue

        vertices = np.asarray(tile_mesh.vertices, dtype=np.float32)
        normals = np.asarray(tile_mesh.vertex_normals, dtype=np.float32)
        triangles = np.asarray(tile_mesh.triangles, dtype=np.int32)

        # Skip if there are no triangles
        if len(triangles) == 0:
            self.tiles[tile_idx].vertices_buffer_start = 0
            self.tiles[tile_idx].vertices_buffer_size = 0
            continue

        rearranged_vertices = vertices[triangles]
        rearranged_normals = normals[triangles]

        num_triangles_tile = len(rearranged_vertices)

        if num_triangles_tile > 0:
            rearranged_vertices_buffer[counter * 9:(counter + num_triangles_tile) * 9] = rearranged_vertices.reshape(-1)
            rearranged_normals_buffer[counter * 9:(counter + num_triangles_tile) * 9] = rearranged_normals.reshape(-1)

            # assign start point and vertices size of the tile map
            self.tiles[tile_idx].vertices_buffer_start = counter * 9
            self.tiles[tile_idx].vertices_buffer_size = num_triangles_tile * 9
            counter += num_triangles_tile
        else:
            self.tiles[tile_idx].vertices_buffer_start = 0
            self.tiles[tile_idx].vertices_buffer_size = 0

        # clean the tile maps
        if not self.keep_tile_maps:
            self.tiles[tile_idx].tile_map = None
      mesh = Mesh()
    mesh._buf_vertices.assign(rearranged_vertices_buffer)
    mesh._buf_normals.assign(rearranged_normals_buffer)
      return mesh

def generate_tile_map_vertex(self, rearranged_vertices, rearranged_normals, max_distance=50):
    """ Old version, we save submap vertices"""
    for idx in tqdm(range(len(rearranged_vertices))):
        # get tile index of each vertex of the triangle
        tile_idx_A = self.get_tile_idx([rearranged_vertices[idx, 0, 0], rearranged_vertices[idx, 0, 1]])  # vertex A(x, y)
        tile_idx_B = self.get_tile_idx([rearranged_vertices[idx, 1, 0], rearranged_vertices[idx, 1, 1]])  # vertex B(x, y)
        tile_idx_C = self.get_tile_idx([rearranged_vertices[idx, 2, 0], rearranged_vertices[idx, 2, 1]])  # vertex C(x, y)

        # add vertices to the corresponding tile map
        self.tiles[tile_idx_A].vertices.append(rearranged_vertices[idx])
        self.tiles[tile_idx_A].normals.append(rearranged_normals[idx])

        # for the same triangle we store only once in one tile map
        if tile_idx_B != tile_idx_A:
            self.tiles[tile_idx_B].vertices.append(rearranged_vertices[idx])
            self.tiles[tile_idx_B].normals.append(rearranged_normals[idx])
```

```python
      if tile_idx_C != tile_idx_A and tile_idx_C != tile_idx_B:
        self.tiles[tile_idx_C].vertices.append(rearranged_vertices[idx])
        self.tiles[tile_idx_C].normals.append(rearranged_normals[idx])
    # get the total number of vertices we stored
    num_triangles = self.get_num_triangles()
    print("total number of triangles: ", num_triangles)
    # rearrange the vertices and assign them to the vertex buffer
    rearranged_vertices_buffer = np.empty(num_triangles * 9, dtype=np.float32)
    rearranged_normals_buffer = np.empty(num_triangles * 9, dtype=np.float32)
    counter = 0
    for tile_idx in range(len(self.tiles)):
      num_triangles_tile = len(self.tiles[tile_idx].vertices)
      if num_triangles_tile > 0:
        rearranged_vertices_buffer[counter * 9:(counter + num_triangles_tile) * 9] = np.array(
          self.tiles[tile_idx].vertices).reshape(-1)
        rearranged_normals_buffer[counter * 9:(counter + num_triangles_tile) * 9] = np.array(
          self.tiles[tile_idx].normals).reshape(-1)
        counter += num_triangles_tile
    return rearranged_vertices_buffer, rearranged_normals_buffer

  def get_local_map(self, tile_idx):
    """ Get the tile map sub-mesh. """
    if self.tiles[tile_idx].tile_map is None or len(np.asarray(self.tiles[tile_idx].tile_map.vertices)) == 0:
      return None
    return self.tiles[tile_idx].tile_map

  def get_global_map(self):
    """ Get the global mesh map. """
    global_map = o3d.geometry.TriangleMesh()
    for tile_idx in range(len(self.tiles)):
      if self.tiles[tile_idx].tile_map is not None and len(np.asarray(self.tiles[tile_idx].tile_map.vertices)) > 0:
        global_map += self.tiles[tile_idx].tile_map
    return global_map

  def get_particles(self, tile_idx):
    """ Get the indexes of particles of give tile. """
    return self.tiles[tile_idx].particle_indexes

  def get_tile_idx(self, position):
    """ Get the index of a tile of give position. """
    # world coordinates to tile index
    i = round((position[0] + self.offset_x - 0.5 * self.tile_size) / self.tile_size)
    j = round((position[1] + self.offset_y - 0.5 * self.tile_size) / self.tile_size)
    tile_idx = int(round(i + j * self.numTiles_x))

    # Make sure the tile_idx is valid
    if tile_idx < 0 or tile_idx >= len(self.tiles):
      # Return a default tile index or handle the error
      return 0
    return tile_idx

  def get_num_triangles(self, use_tile_map=True):
    """ Get the number of triangles. """
    num_triangles = 0
```

```
    if use_tile_map:
      # print('use cropped tile map to rearrange vertices buffer.')
      for tile_idx in range(len(self.tiles)):
        tile_mesh = self.tiles[tile_idx].tile_map

        # Skip if tile_mesh is None or empty
        if tile_mesh is None or len(np.asarray(tile_mesh.vertices)) == 0:
          continue

        vertices = np.asarray(tile_mesh.vertices, dtype=np.float32)
        triangles = np.asarray(tile_mesh.triangles, dtype=np.int32)

        # Skip if there are no triangles
        if len(triangles) == 0:
          continue

        rearranged_vertices = vertices[triangles]
        num_triangles += len(rearranged_vertices)
    else:
      # print('use vertices directly.')
      for tile_idx in range(len(self.tiles)):
        num_triangles += len(self.tiles[tile_idx].vertices)
        print(len(self.tiles[tile_idx].vertices))
    return num_triangles

  def clean_tile_maps(self):
    """ Release tile maps in CPU. """
    for tile_idx in range(len(self.tiles)):
      self.tiles[tile_idx].tile_map = None

  def calculate_tile_height(self):
    """ Calculate the height for each tile. """
    for tile_idx in range(len(self.tiles)):
      if len(self.tiles[tile_idx].scan_indexes) == 0: continue
      poses = self.poses[self.tiles[tile_idx].scan_indexes]
      self.tiles[tile_idx].z = np.mean(poses[:, 2, 3])

  def plot_valid_tiles(self, tiles, poses):
    """ Visualize supmaps together with trajectory. """
    fig = plt.figure()
    ax = fig.add_subplot(111)
    currentAxis = plt.gca()
    plt.plot(poses[:, 0, 3], poses[:, 1, 3])
    for idx in range(len(tiles)):
      if tiles[idx].valid:
        currentAxis.add_patch(Rectangle((tiles[idx].x - self.max_distance, tiles[idx].y - self.max_distance),
                                        self.tile_size, self.tile_size, alpha=1, fill=None))
    ax.set_aspect('equal', adjustable='box')
    plt.xlabel("X [m]")
    plt.ylabel("Y [m]")
    plt.title("Visualization of trajectory and submaps")
    plt.show()

  def vis_mesh(self, mesh, crop_mesh=False):
```

```
    """ Visualize mesh. """
    if mesh is None or len(np.asarray(mesh.vertices)) == 0:
      print("Empty mesh, nothing to visualize")
      return

    if crop_mesh:
      bbox = o3d.geometry.AxisAlignedBoundingBox(min_bound=(-50, -50, -5),
                                              max_bound=(+50, +50, +5))
      mesh = mesh.crop(bbox)
     mesh.compute_vertex_normals()
    o3d.visualization.draw_geometries([mesh])

  def vis_mesh_traj(self, mesh):
    """ Visualize mesh together with trajectory. """
    if mesh is None or len(np.asarray(mesh.vertices)) == 0:
      print("Empty mesh, nothing to visualize")
      return

    pose_points = self.poses[:, :3, 3]
    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(np.asarray(pose_points))
    origin = o3d.geometry.TriangleMesh.create_coordinate_frame(size=50.0)
    # pcd.estimate_normals()
    o3d.visualization.draw_geometries([mesh, pcd, origin])

  def cleanup(self):
    """Clean up resources properly"""
    # Clean up the mesh if it exists
    if hasattr(self, 'mesh') and self.mesh is not None:
      # Clear buffers
      self.mesh._buf_vertices = None
      self.mesh._buf_normals = None
      self.mesh = None

    # Clean up the window reference
    if hasattr(self, 'window') and self.window is not None:
      self.window = None


# Add a cleanup method to OffscreenWindow class in map_renderer.py
def cleanup_window(window):
  """Helper function to clean up a window instance"""
  try:
    # Call any cleanup methods needed
    pass
  except:
    pass


# DEBUG: Add a __del__ method to Mesh class to handle OpenGL buffer cleanup
def safe_delete_buffer(buf):
  """Safely delete OpenGL buffer with error handling"""
  try:
    import OpenGL.GL as gl
```

```
    if bool(gl.glDeleteBuffers) and buf is not None and buf.id_ is not None:
      gl.glDeleteBuffers(1, [buf.id_])
  except Exception as e:
    pass  # Silently ignore errors during cleanup


# debugging
def test_tile_map_vertex():
  mesh_file = r'C:\Users\kavin\OneDrive\Documents\GitHub\range-mcl\data\mesh_kitti_07.ply'
  pose_file = r'C:\Users\kavin\OneDrive\Documents\GitHub\range-mcl\data\kitti-07\07\poses.txt'
  poses = load_poses(pose_file)
   window = OffscreenWindow(show=False)  # Keep window reference
  submap_test = MapModule(poses, mesh_file)
   for idx in range(len(submap_test.tiles)):
    tile = submap_test.tiles[idx]

    if len(tile.vertices) == 0:
      continue

    pcd = o3d.geometry.PointCloud()
    vertices = np.array(tile.vertices).reshape((-1, 3))
    pcd.points = o3d.utility.Vector3dVector(vertices)

    o3d.visualization.draw_geometries([pcd])

  # Clean up
  submap_test.cleanup()
  cleanup_window(window)


def test_get_map():
  mesh_file = r'/Users/supriyakommini/range-mcl-main/data/07/mesh_kitti_07.ply'
  pose_file = r'/Users/supriyakommini/range-mcl-main/data/07/poses.txt'
  poses = load_poses(pose_file)
   window = OffscreenWindow(show=False)  # Keep window reference
  submap_test = MapModule(poses, mesh_file, keep_tile_maps=True)
   # get global map
  global_mesh = submap_test.get_global_map()
  if len(np.asarray(global_mesh.vertices)) > 0:
    o3d.visualization.draw_geometries([global_mesh])
   # get local map
  for idx in range(len(submap_test.tiles)):
    local_mesh = submap_test.get_local_map(idx)
    if local_mesh is not None and len(np.asarray(local_mesh.vertices)) > 0:
      o3d.visualization.draw_geometries([local_mesh])

  # Clean up
  submap_test.cleanup()
  cleanup_window(window)


def test_get_rearranged_vertices_buffer():
  mesh_file = r'/Users/supriyakommini/range-mcl-main/data/07/mesh_kitti_07.ply'
  pose_file = r'/Users/supriyakommini/range-mcl-main/data/07/poses.txt'
```

```python
    poses = load_poses(pose_file)
     window = OffscreenWindow(show=False)  # Keep window reference
    submap_test = MapModule(poses, mesh_file)
     # Check if global_vertices attribute exists
    if hasattr(submap_test, 'global_vertices'):
      global_vertices = np.array(submap_test.global_vertices).reshape((-1, 3))

      pcd = o3d.geometry.PointCloud()
      pcd.points = o3d.utility.Vector3dVector(global_vertices)
      o3d.visualization.draw_geometries([pcd])
    else:
      print("global_vertices attribute not found")

    # Clean up
    submap_test.cleanup()
    cleanup_window(window)


def test_average_height():
    mesh_file = r'/Users/supriyakommini/range-mcl-main/data/07/mesh_kitti_07.ply'
    pose_file = r'/Users/supriyakommini/range-mcl-main/data/07/poses.txt'
    poses = load_poses(pose_file)
     window = OffscreenWindow(show=False)  # Keep window reference
    submap_test = MapModule(poses, mesh_file)
    # get local map
    for idx in range(len(submap_test.tiles)):
      print("tile_idx: ", idx, " z: ", submap_test.tiles[idx].z)

    # Clean up
    submap_test.cleanup()
    cleanup_window(window)


if __name__ == '__main__':
    try:
      # test_tile_map_vertex()
      test_get_map()
      # test_get_rearranged_vertices_buffer()
      # test_average_height()
      # pass
    except Exception as e:
      print(f"Error: {e}")
```

sensor_model.py

```python
#!/usr/bin/env python3
# This file is covered by the LICENSE file in the root of this project.
# Brief: this is the sensor model for correlation-based Monte Carlo localization.

import os
import numpy as np
import OpenGL.GL as gl
import matplotlib.pyplot as plt
```

```python
from map_renderer import MapRenderer_instanced
from utils import load_files, load_vertex, range_projection, rotation_matrix_from_euler_angles

class SensorModel():
 """
 Brief: This class is the implementation of using correlation of range images as the sensor model for
       localization. In this sensor model we discretize the environment and generate a virtual frame for each grid
       after discretization. We estimate the similarity between the current frame and the grid virtual frames using
       the correlation.
 Initialization Input:
    mapsize: The size of the given map
    grid_coords: coordinates of virtual frames
    depth_image_paths: paths of range images
    grid_res: The resolution of the grids, default as 0.2 meter
 """
  def __init__(self, map_module, scan_folder, params):
   # load the map module.
   self.map_module = map_module

   # initialize the map renderer with the appropriate parameter.
   self.params = params
   self.max_instance = params['max_instance']
   self.renderer = MapRenderer_instanced(self.params)
   self.renderer.set_mesh(self.map_module.mesh)
    # specify query scan paths
   self.scan_paths = load_files(scan_folder)

   self.is_converged = False
  def update_weights(self, particles, frame_idx):
   """ This function update the weight for each particle using the difference
   between current range image and the synthetic rendering for each particle.
   Old version where we render range image for each particle individually
   To use old version one need to import MapRenderer from renderer.py
   Input:
      particles: each particle has four properties [x, y, theta, weight]
      frame_idx: the index of the current frame
   Output:
      particles ... same particles with updated particles(i).weight
   """
   # load current scan and compute the histogram
   current_path = self.scan_paths[frame_idx]
   current_vertex = load_vertex(current_path)
   current_range, _, _, _ = range_projection(current_vertex,
                                             fov_up=self.params["fov_up"],
                                             fov_down=self.params["fov_down"],
                                             proj_H=self.params["height"],
                                             proj_W=self.params["width"],
                                             max_range=self.params["max_range"])
   # self.save_depth_image('current_frame', current_range, frame_idx, 0)

   scores = np.ones(len(particles)) * 0.00001

   tiles_collection = []
```

```
for idx in range(len(particles)):
  particle = particles[idx]

  # first check whether the particle is inside the map or not
  if particle[0] < self.map_module.map_boundaries[0] or \
      particle[0] > self.map_module.map_boundaries[1] or \
      particle[1] < self.map_module.map_boundaries[2] or \
      particle[1] > self.map_module.map_boundaries[3]:
    continue

  # get tile index given particle position
  tile_idx = self.map_module.get_tile_idx([particle[0], particle[1]])
  if not self.map_module.tiles[tile_idx].valid:
    continue
  if tile_idx not in tiles_collection:
    tiles_collection.append(tile_idx)

  # get tile vertices start point and size
  start = self.map_module.tiles[tile_idx].vertices_buffer_start
  size = self.map_module.tiles[tile_idx].vertices_buffer_size

  # particle pose
  particle_pose = np.identity(4)  # init
  particle_pose[0, 3] = particle[0]  # particle[0]
  particle_pose[1, 3] = particle[1]  # particle[1]
  particle_pose[2, 3] = self.map_module.tiles[tile_idx].z  # use tile z
  particle_pose[:3, :3] = rotation_matrix_from_euler_angles(particle[2], degrees=False)  # rotation

  # generate synthetic range image
  self.renderer.render_with_tile(particle_pose, start, size)
  particle_depth = self.renderer.get_depth_map()

  # update the weight
  diff = abs(particle_depth - current_range)
  scores[idx] = np.exp(-0.5 * np.mean(diff[current_range > 0]) ** 2 / (2.0 ** 2))

# normalization
particles[:, 3] = particles[:, 3] * scores
particles[:, 3] = particles[:, 3] / np.max(particles[:, 3])

# check convergence using supporting tile map idea
if len(tiles_collection) < 2 and not self.is_converged:
  self.is_converged = True
  print('Converged!')
  # cutoff redundant particles and leave only num of particles
  idxes = np.argsort(particles[:, 3])[::-1]
  particles = particles[idxes[:100]]

return particles, len(particles)
def update_weights_instanced(self, particles, frame_idx):
""" This function update the weight for each particle using the difference
between current range image and the synthetic rendering for each particle.
Here, we use instance rendering to accelerate the sensor model
Input:
```

```
    particles: each particle has four properties [x, y, theta, weight]
    frame_idx: the index of the current frame
  Output:
    particles ... same particles with updated particles(i).weight
  """

  # load current scan and compute the histogram
  current_path = self.scan_paths[frame_idx]
  current_vertex = load_vertex(current_path)
  current_range, _, _, _ = range_projection(current_vertex,
                                            fov_up=self.params["fov_up"],
                                            fov_down=self.params["fov_down"],
                                            proj_H=self.params["height"],
                                            proj_W=self.params["width"],
                                            max_range=self.params["max_range"])
  # self.save_depth_image('current_frame', current_range, frame_idx, 0)
  scores = np.ones(len(particles)) * 0.00001

  tiles_collection = []  # for counter number of tiles
  tiles_mask = np.ones(len(particles)) * -1  # for clustering

  for idx in range(len(particles)):
    particle = particles[idx]

    # first check whether the particle is inside the map or not
    if particle[0] < self.map_module.map_boundaries[0] or \
        particle[0] > self.map_module.map_boundaries[1] or \
        particle[1] < self.map_module.map_boundaries[2] or \
        particle[1] > self.map_module.map_boundaries[3]:
      continue

    # get tile index given particle position
    tile_idx = self.map_module.get_tile_idx([particle[0], particle[1]])
    if not self.map_module.tiles[tile_idx].valid:
      continue
    tiles_mask[idx] = tile_idx
    if tile_idx not in tiles_collection:
      tiles_collection.append(tile_idx)

  # we render all particles lies in the same tile instancely once
  for tile_idx in tiles_collection:
    # get tile vertices start point and size
    start = self.map_module.tiles[tile_idx].vertices_buffer_start
    size = self.map_module.tiles[tile_idx].vertices_buffer_size

    # collect poses of particles in the same tile
    mask = np.argwhere(tiles_mask == tile_idx)
    particles_in_tile = particles[mask]
    num_particles_in_tile = len(particles_in_tile)

    for interval_idx in range(int(num_particles_in_tile / self.max_instance) + 1):
      particles_in_tile_ = particles_in_tile[interval_idx * self.max_instance:
                                             (interval_idx + 1) * self.max_instance]
      num_particles_in_tile_ = len(particles_in_tile_)
```

```
      particle_poses = []
      for particle_idx in range(num_particles_in_tile_):
        particle = particles_in_tile_[particle_idx, 0]
        particle_pose = np.identity(4)
        particle_pose[0, 3] = particle[0]  # particle[0]
        particle_pose[1, 3] = particle[1]  # particle[1]
        particle_pose[2, 3] = self.map_module.tiles[tile_idx].z  # use tile z
        particle_pose[:3, :3] = rotation_matrix_from_euler_angles(particle[2], degrees=False)  # rotation
        particle_poses.append(particle_pose)

      # generate synthetic range image
      self.renderer.render_instanced(particle_poses, start, size)
      particle_depth = self.renderer.get_instance_depth_map()
      # update the weight
      scores_ = []
      for particle_idx in range(num_particles_in_tile_):
        diff = abs(particle_depth[particle_idx] - current_range)
        scores_.append(np.exp(-0.5 * np.mean(diff[current_range > 0]) ** 2 / (2.0 ** 2)))

      indices = mask[interval_idx * self.max_instance:(interval_idx + 1) * self.max_instance]
      if len(indices) > 1:
        scores[indices.squeeze()] = scores_

    # normalization
    particles[:, 3] = particles[:, 3] * scores
    particles[:, 3] = particles[:, 3] / np.max(particles[:, 3])

    # check convergence using supporting tile map idea
    if len(tiles_collection) < 2 and not self.is_converged:
      self.is_converged = True
      print('Converged!')
      # cutoff redundant particles and leave only num of particles
      idxes = np.argsort(particles[:, 3])[::-1]
      particles = particles[idxes[:100]]

    return particles, len(particles)
   def save_depth_image(self, folder_name, current_range, frame_idx, idx):
    """ Saving renderings for debugging """
    fig = plt.figure(frameon=False)  # frameon=False, suppress drawing the figure background patch.
    fig.set_size_inches(9, 0.64)
    ax = plt.Axes(fig, [0., 0., 1., 1.])
    ax.set_axis_off()
    fig.add_axes(ax)

    ax.imshow(current_range, aspect='equal')
    fig.savefig(os.path.join(folder_name, str(frame_idx).zfill(6) + '_' + str(idx) + '.png'))
    plt.close()


def test_map_render():
  """ debugging """
  map_file = '/path/to/scan/map'
  scan_folder = '/path/to/scan/folder'
  correlation_sensor = SensorModel(map_file, scan_folder)
```

```
 particles = np.zeros((100, 4))
particles[:, 0] = np.arange(100) - 50
particles[:, 3] = np.ones(len(particles))
 for frame_id in range(1):
  correlation_sensor.update_weights(particles, frame_id)


if __name__ == '__main__':
 # test_map_render()
 pass

if __name__ == '__main__':
  from map_module import MapModule
  from utils import load_poses_kitti

  scan_folder = 'data/07/velodyne'
  mesh_file = 'data/07/mesh_kitti_07.ply'
  pose_file = 'data/07/poses.txt'
  calib_file = 'data/07/calib.txt'

  print("🟢 Running standalone SensorModel test...")

  map_poses = load_poses_kitti(pose_file, calib_file)
  map_module = MapModule(map_poses, mesh_file)

  sensor_params = {
      "fov_up": 3.0,
      "fov_down": -25.0,
      "height": 64,
      "width": 1024,
      "max_range": 80.0,
      "min_range": 2.0,
      "render_instanced": False,
      "max_instance": 128
  }

  sensor_model = SensorModel(map_module, scan_folder, sensor_params)
  print("✅ SensorModel initialized successfully.")
```

motion_model.py

```
#!/usr/bin/env python3
# This file is covered by the LICENSE file in the root of this project.
# Brief: this is the motion model for overlap-based Monte Carlo localization.

from utils import *


def motion_model(particles, u, real_command=False, duration=0.1):
 """ MOTION performs the sampling from the proposal.
 distribution, here the rotation-translation-rotation motion model

 input:
```

```
    particles: the particles as in the main script
    u: the command in the form [rot1 trasl rot2] or real odometry [v, w]
    noise: the variances for producing the Gaussian noise for
    perturbating the motion,  noise = [noiseR1 noiseTrasl noiseR2]

  output:
    the same particles, with updated poses.

  The position of the i-th particle is given by the 3D vector
  particles(i).pose which represents (x, y, theta).

  Assume Gaussian noise in each of the three parameters of the motion model.
  These three parameters may be used as standard deviations for sampling.
  """
  num_particles = len(particles)
  if not real_command:
    # noise in the [rot1 trasl rot2] commands when moving the particles
    MOTION_NOISE = [0.01, 0.05, 0.01]
    r1Noise = MOTION_NOISE[0]
    transNoise = MOTION_NOISE[1]
    r2Noise = MOTION_NOISE[2]

    rot1 = u[0] + r1Noise * np.random.randn(num_particles)
    tras1 = u[1] + transNoise * np.random.randn(num_particles)
    rot2 = u[2] + r2Noise * np.random.randn(num_particles)

    # update pose using motion model
    particles[:, 0] += tras1 * np.cos(particles[:, 2] + rot1)
    particles[:, 1] += tras1 * np.sin(particles[:, 2] + rot1)
    particles[:, 2] += rot1 + rot2

  else:  # use real commands with duration
    # noise in the [v, w] commands when moving the particles
    MOTION_NOISE = [0.05, 0.05]
    vNoise = MOTION_NOISE[0]
    wNoise = MOTION_NOISE[1]

    # use the Gaussian noise to simulate the noise in the motion model
    v = u[0] + vNoise * np.random.randn(num_particles)
    w = u[1] + wNoise * np.random.randn(num_particles)
    gamma = wNoise * np.random.randn(num_particles)

    # update pose using motion models
    particles[:, 0] += - v / w * np.sin(particles[:, 2]) + v / w * np.sin(particles[:, 2] + w * duration)
    particles[:, 1] += v / w * np.cos(particles[:, 2]) - v / w * np.cos(particles[:, 2] + w * duration)
    particles[:, 2] += w * duration + gamma * duration

  return particles


def gen_commands(poses):
  """ Create commands out of the ground truth with noise.
  input:
    ground truth poses
```

```
    output:
      commands for each frame.
    """
    # compute noisy-free commands
    # set the default command = [0,0,0]'
    commands = np.zeros((len(poses), 3))
    # compute relative poses
    rela_poses = []
    headings = []
    last_pose = poses[0]
    for idx in range(len(poses)):
      rela_poses.append(np.linalg.inv(last_pose).dot(poses[idx]))
      headings.append(euler_angles_from_rotation_matrix(poses[idx][:3, :3])[2])
      last_pose = poses[idx]
    rela_poses = np.array(rela_poses)
    dx = (poses[1:, 0, 3] - poses[:-1, 0, 3])
    dy = (poses[1:, 1, 3] - poses[:-1, 1, 3])
    direct = np.arctan2(dy, dx)  # atan2(dy, dx), 1X(S-1) direction of the movement
    r1 = []
    r2 = []
    distance = []
    for idx in range(len(rela_poses) - 1):
      r1.append(direct[idx] - headings[idx])
      r2.append(headings[idx + 1] - direct[idx])
      distance.append(np.sqrt(dx[idx] * dx[idx] + dy[idx] * dy[idx]))
    r1 = np.array(r1)
    r2 = np.array(r2)
    distance = np.array(distance)
    # add noise to commands
    commands_ = np.c_[r1, distance, r2]
    commands[1:] = commands_ + np.array([0.01 * np.random.randn(len(commands_)),
                                         0.01 * np.random.randn(len(commands_)),
                                         0.01 * np.random.randn(len(commands_))]).T

    return commands


def gen_motion_reckon(commands):
    """ Generate motion reckon only for comparison.
    """

    particle = [0, 0, 0, 1]
    motion_reckon = []
    for cmmand in commands:
      # use the Gaussian noise to simulate the noise in the motion model
      rot1 = cmmand[0]
      tras1 = cmmand[1]
      rot2 = cmmand[2]
      # update pose using motion model
      particle[0] = particle[0] + tras1 * np.cos(particle[2] + rot1)
      particle[1] = particle[1] + tras1 * np.sin(particle[2] + rot1)
      particle[2] = particle[2] + rot1 + rot2
```

```
    motion_reckon.append([particle[0], particle[1]])
  return np.array(motion_reckon)


if __name__ == '__main__':
  pass
```

initialization.py

```
#!/usr/bin/env python3
# This file is covered by the LICENSE file in the root of this project.
# Brief: some functions for MCL initialization

import numpy as np
import open3d as o3d
from utils import euler_angles_from_rotation_matrix

np.random.seed(0)


def init_particles_uniform(map_size, numParticles):
  """ Initialize particles uniformly.
    Args:
      map_size: size of the map.
      numParticles: number of particles.
    Return:
      particles.
  """
  [x_min, x_max, y_min, y_max] = map_size
  particles = []
  rand = np.random.rand
  for i in range(numParticles):
    x = (x_max - x_min) * rand(1) + x_min
    y = (y_max - y_min) * rand(1) + y_min
    # theta = 2 * np.pi * rand(1)
    theta = -np.pi + 2 * np.pi * rand(1)
    weight = 1
    particles.append([x, y, theta, weight])
  return np.array(particles)


def gen_coords_given_poses(poses, resolution=0.2, submap_size=2):
  """ Generate the road coordinates given the map poses.
    Args:
      poses: poses used to build the map.
      resolution: size of the grids for the initialization.
      submap_size: size of the submap for the initialization.
    Return:
      coords: coordinates of road grids for initialize particles.
  """
  submap_coords = []
  for x_coord in np.arange(-submap_size, submap_size, resolution):
    for y_coord in np.arange(-submap_size, submap_size, resolution):
```

```
      submap_coords.append([x_coord, y_coord])
   coords = []
  for pose in poses:
    center = pose[:2, 3]
    coords.append(submap_coords + center)
   coords = np.array(coords).reshape(-1, 2)
  coords_3d = np.zeros((coords.shape[0], coords.shape[1] + 1))
  coords_3d[:, :2] = coords
   pcd = o3d.geometry.PointCloud()
  pcd.points = o3d.utility.Vector3dVector(coords_3d)
  downpcd = pcd.voxel_down_sample(voxel_size=resolution).points
  coords = np.array(downpcd)[:, :2]
   min_x = int(np.round(np.min(coords[:, 0])))
  max_x = int(np.round(np.max(coords[:, 0])))
  min_y = int(np.round(np.min(coords[:, 1])))
  max_y = int(np.round(np.max(coords[:, 1])))
   return [min_x, max_x, min_y, max_y], coords


def init_particles_given_coords(numParticles, coords, init_weight=1.0):
  """ Initialize particles uniformly given the road coordinates.
    Args:
      numParticles: number of particles.
      coords: road coordinates.
      init_weight: initialization weight.
    Return:
      particles.
  """
  particles = []
  rand = np.random.rand
  args_coords = np.arange(len(coords))
  selected_args = np.random.choice(args_coords, numParticles)
   for i in range(numParticles):
    x = coords[selected_args[i]][0]
    y = coords[selected_args[i]][1]
    # theta = 2 * np.pi * rand(1)
    theta = -np.pi + 2 * np.pi * rand(1)
    particles.append([x, y, theta, init_weight])
   return np.array(particles, dtype=float)


def init_particles_pose_tracking(numParticles, init_pose, noises=[10.0, 10.0, np.pi/3.0], init_weight=1.0):
    """Initialize particles with a noisy initial pose."""
    particles = []
    init_x = init_pose[0, 3]
    init_y = init_pose[1, 3]
    init_yaw = euler_angles_from_rotation_matrix(init_pose[:3, :3])[2]

    for _ in range(numParticles):
        x = float(init_x + noises[0] * (np.random.rand() - 0.5))
        y = float(init_y + noises[1] * (np.random.rand() - 0.5))
        theta = float(init_yaw + noises[2] * (np.random.rand() - 0.5))
        particles.append([x, y, theta, init_weight])
```

```
    return np.array(particles, dtype=np.float32)
```

pso_pose_optimizer.py

```
import numpy as np
import time

class PSOOptimizer:
    def __init__(self, omega=0.5, phi_p=1.5, phi_g=1.5, max_iters=15, max_velocity=0.3, verbose=False):
        self.omega = omega
        self.phi_p = phi_p
        self.phi_g = phi_g
        self.max_iters = max_iters
        self.max_velocity = max_velocity
        self.verbose = verbose

    @staticmethod
    def range_diff_score(real, synth):
        mask = real > 0
        if np.sum(mask) == 0:
            return np.inf
        return np.mean(np.abs(real[mask] - synth[mask]))

    def optimize(self, real_range, particles, render_func):
        particles = particles.copy()
        N = len(particles)
        v = np.random.randn(N, 3) * 0.1
        p_best = particles.copy()
        p_scores = np.array([self.range_diff_score(real_range, render_func(p)) for p in particles])
        g_idx = np.argmin(p_scores)
        g_best = p_best[g_idx].copy()
        g_score = p_scores[g_idx]
        history = [g_score]

        for iter in range(self.max_iters):
            for i in range(N):
                r_p, r_g = np.random.rand(3), np.random.rand(3)
                v[i] = (self.omega * v[i] +
                        self.phi_p * r_p * (p_best[i] - particles[i]) +
                        self.phi_g * r_g * (g_best - particles[i]))
                v[i] = np.clip(v[i], -self.max_velocity, self.max_velocity)
                particles[i] += v[i]
                particles[i][2] = np.arctan2(np.sin(particles[i][2]), np.cos(particles[i][2]))

                score = self.range_diff_score(real_range, render_func(particles[i]))
                if score < p_scores[i]:
                    p_best[i], p_scores[i] = particles[i], score
                    if score < g_score:
                        g_best, g_score = particles[i], score
            history.append(g_score)
            if self.verbose:
                print(f"[PSO] Iter {iter+1}, best score = {g_score:.5f}")
        return g_best
```

refine_trajectory_with_pso.py

```python
import numpy as np
from utils import load_vertex, range_projection
from pso_pose_optimizer import PSOOptimizer


def refine_trajectory(est_poses, scan_paths, sensor_model):
    """
    Refine an estimated trajectory using PSO.

    Parameters:
    - est_poses:     (N_frames, 3) array of [x, y, theta]
    - scan_paths:    list of LiDAR .bin file paths
    - sensor_model:  instance of SensorModel with a render_scan() method that accepts [x, y, theta]

    Returns:
    - refined_poses: (N_frames, 3) array of PSO-refined poses
    """

    H = sensor_model.params["height"]
    W = sensor_model.params["width"]

    refined_poses = []

    for i, (x0, y0, t0) in enumerate(est_poses):
        print(f"🔧 Refining frame {i}...")

        # --- Load real scan and generate range image ---
        vertex = load_vertex(scan_paths[i])
        real_range, *_ = range_projection(
            vertex,
            fov_up=sensor_model.params["fov_up"],
            fov_down=sensor_model.params["fov_down"],
            proj_H=H,
            proj_W=W
        )

        # --- Define render function for optimizer ---
        def render_func(pose_3d):
            return sensor_model.render_scan(pose_3d)

        # --- PSO optimization (no keyword args) ---
        pso = PSOOptimizer(x0, y0, t0)
        best_pose = pso.optimize(real_range, render_func)  # Pass args positionally

        refined_poses.append(best_pose)

    return np.array(refined_poses)
```

build_mesh_map.py (Using fuzzy systems)

```python
#!/usr/bin/env python3
```

```python
# This file is covered by the LICENSE file in the root of this project.
# Brief: This script can be used to create mesh maps using LiDAR scans with GT poses.

import os
import yaml
import numpy as np
import open3d as o3d
from tqdm import tqdm
from copy import deepcopy

from utils import load_files, load_poses, load_calib, load_vertex
from map_building.simplify_ground_mesh import pcd_ground_seg_fuzzy, mesh_simplify
from map_building.compute_normals import compute_normals_range


def preprocess_cloud(pcd, voxel_size=0.1,
                     crop_x=30, crop_y=30, crop_z=5,
                     downsample=False):
  """ preprocess the point cloud, including downsampling and cropping.
  """
  # downsample the point cloud if needed
  cloud = pcd.voxel_down_sample(voxel_size) if downsample else deepcopy(pcd)
  # crop point cloud with a box
  bbox = o3d.geometry.AxisAlignedBoundingBox(min_bound=(-crop_x, -crop_y, -crop_z),
                                             max_bound=(+crop_x, +crop_y, +crop_z))
  return cloud.crop(bbox)


def run_poisson(pcd, depth, min_density):
  """ run Poisson reconstruction on a local point cloud to get a local mesh.
  """
  if not pcd.has_normals():
    print("PointCloud doesn't have normals")
  o3d.utility.set_verbosity_level(o3d.utility.VerbosityLevel.Debug)
  mesh, densities = o3d.geometry.TriangleMesh.create_from_point_cloud_poisson(
    pcd, depth=depth)
  # Post-process the mesh
  if min_density:
    vertices_to_remove = densities < np.quantile(densities, min_density)
    mesh.remove_vertices_by_mask(vertices_to_remove)
  # Return mesh
  mesh.compute_vertex_normals()
  return mesh


def main(config):
  """ This script can be used to create mesh maps using LiDAR scans with GT poses.
  It assumes you have the data in the kitti-like format like:

  data
  └── sequences
      └── 00
          ├── calib.txt
          ├── poses.txt
```

```
    └── velodyne
        ├── 000000.bin
        ├── 000001.bin
        └── ...

How to run it and check a quick example:
$ ./build_gt_map.py /path/to/config.yaml
"""
# load scans and poses
scan_folder = config['scan_folder']
scan_paths = load_files(scan_folder)

# load poses
pose_file = config['pose_file']
poses =load_poses(pose_file)
inv_frame0 = np.linalg.inv(poses[0])

# load calibrations
# Note that if your poses are already in the LiDAR coordinate system, you
# just need to set T_cam_velo as a 4x4 identity matrix
calib_file = config['calib_file']
T_cam_velo = load_calib(calib_file)
T_cam_velo = np.asarray(T_cam_velo).reshape((4, 4))
T_velo_cam = np.linalg.inv(T_cam_velo)

# convert poses into LiDAR coordinate system
new_poses = []
for pose in poses:
  new_poses.append(T_velo_cam.dot(inv_frame0).dot(pose).dot(T_cam_velo))
new_poses = np.array(new_poses)
gt_poses = new_poses
 # Use the whole sequence if -1 is specified
n_scans = len(scan_paths) if config['n_scans'] == -1 else config['n_scans']
 # init mesh map
mesh_file = config['mesh_file']
if os.path.exists(mesh_file):
  exit(print('The mesh map already exists at:', mesh_file))
global_mesh = o3d.geometry.TriangleMesh()
cloud_map = o3d.geometry.PointCloud()
 # counter for local map
count = 1
local_map_size = config['local_map_size']
 # config for range images
range_config = config['range_image']

for idx in tqdm(range(n_scans)):
  # load the point cloud
  curren_points = load_vertex(scan_paths[idx])

  # get rid of invalid points
  dist = np.linalg.norm(curren_points[:, :3], 2, axis=1)
  curren_points = curren_points[(dist < range_config['max_range']) & (dist > range_config['min_range'])]

  # convert into open3d format and preprocess the point cloud
```

```python
    local_cloud = o3d.geometry.PointCloud()
    local_cloud.points = o3d.utility.Vector3dVector(curren_points[:, :3])

    # estimated normals
    local_cloud = compute_normals_range(local_cloud, range_config['fov_up'], range_config['fov_down'],
                                        range_config['height'], range_config['width'], range_config['max_range'])

    # preprocess point clouds
    local_cloud = preprocess_cloud(local_cloud, config['voxel_size'],
                                   config['crop_x'], config['crop_y'], config['crop_z'],
                                   downsample=True)

    # integrate the local point cloud
    local_cloud.transform(gt_poses[idx])
    cloud_map += local_cloud

    if idx > 0:
      # if the car stops, we don't count the frame
      relative_pose = np.linalg.inv(gt_poses[idx - 1]).dot(gt_poses[idx])
      traj_dist = np.linalg.norm(relative_pose[:3, 3])
      if traj_dist > 0.2:
        count += 1

      # build a local mesh map
      if count % local_map_size == 0:
        # segment the ground
        ground, rest = pcd_ground_seg_fuzzy(cloud_map)

        # build the local poisson mesh
        mesh = run_poisson(ground+rest, depth=config['depth'], min_density=config['min_density'])

        # simply the ground to save space
        mesh = mesh_simplify(mesh, config)
        mesh.compute_vertex_normals()
        mesh.compute_triangle_normals()

        # integrate the local mesh into global mesh
        global_mesh += mesh

        # re-init cloud map
        cloud_map = o3d.geometry.PointCloud()

  # save the mesh map
  print("Saving mesh to " + mesh_file)
  o3d.utility.set_verbosity_level(o3d.utility.VerbosityLevel.Error)
  o3d.io.write_triangle_mesh(mesh_file, global_mesh)

  # visualize the mesh map
  if config['visualize']:
    o3d.visualization.draw_geometries([global_mesh])


if __name__ == "__main__":
  # load config file
```

```
config_filename = '/Users/supriyakommini/range-mcl-main/config/build_map.yml'
 if yaml.__version__>='5.1':
  config = yaml.load(open(config_filename), Loader=yaml.FullLoader)
else:
  config = yaml.load(open(config_filename))
 o3d.utility.set_verbosity_level(o3d.utility.VerbosityLevel.Info)
main(config)
```

ground_segmentation_module.py (called in mesh map building code)

```
#!/usr/bin/env python3
# This file is covered by the LICENSE file in the root of this project.
# Brief: functions to simplify the ground mesh.

import copy
import numpy as np

from time_utils import timeit


@timeit
def pcd_ground_seg_pca(scan, th=0.80, z_offset=-1.1):
 """ Perform PCA over PointCloud to segment ground.
 """
 pcd = copy.deepcopy(scan)
 _, covariance = pcd.compute_mean_and_covariance()
 eigen_vectors = np.linalg.eig(covariance)[1]
 k = eigen_vectors.T[2]

 # magnitude of projecting each face normal to the z axis
 normals = np.asarray(scan.normals)
 points = np.asarray(scan.points)
 mag = np.linalg.norm(np.dot(normals, k).reshape(-1, 1), axis=1)
 ground = pcd.select_by_index(np.where((mag >= th) & (points[:, 2] < z_offset))[0])
 rest = pcd.select_by_index(np.where((mag >= th) & (points[:, 2] < z_offset))[0], invert=True)

 # Also remove the faces that are looking downwards
 up_normals = np.asarray(ground.normals)
 orientation = np.dot(up_normals, k)
 ground = ground.select_by_index(np.where(orientation > 0.0)[0])

 ground.paint_uniform_color([1.0, 0.0, 0.0])
 rest.paint_uniform_color([0.0, 0.0, 1.0])
  return ground, rest



@timeit
def pcd_ground_seg_fuzzy(scan):
 """ Segment ground using fuzzy logic on height and normal angle. """
 import skfuzzy as fuzz
 from skfuzzy import control as ctrl
```

```python
    pcd = copy.deepcopy(scan)
    points = np.asarray(pcd.points)
    normals = np.asarray(pcd.normals)

    z_vals = points[:, 2]  # height
    up_vector = np.array([0, 0, 1])
    angles = np.degrees(np.arccos(np.clip(np.dot(normals, up_vector), -1, 1)))  # angle to Z axis

    # Fuzzy membership rules
    z_low = fuzz.interp_membership([-5, 0, 1], [1, 1, 0], z_vals)  # low height = ground
    angle_low = fuzz.interp_membership([0, 10, 30], [1, 1, 0], angles)  # small angle = flat

    # Combine: fuzzy AND (min)
    ground_score = np.minimum(z_low, angle_low)
    ground_idx = np.where(ground_score > 0.5)[0]
    rest_idx = np.setdiff1d(np.arange(len(points)), ground_idx)

    ground = pcd.select_by_index(ground_idx)
    rest = pcd.select_by_index(rest_idx)

    ground.paint_uniform_color([1.0, 0.0, 0.0])
    rest.paint_uniform_color([0.0, 0.0, 1.0])

    return ground, rest

@timeit
def mesh_simplify(mesh, config):
    """ simplify the ground meshes using simplify_vertex_clustering and filter_smooth_simple.
    """
    mesh_gnd = copy.deepcopy(mesh)
    mesh_rest = copy.deepcopy(mesh)
     # triangles.shape = n_t x 3 x 3,
    # where n_t is the number of triangles,
    # the first 3 is the three vertices
    # and the second three is the 3d coordinates of the vertices
    triangles = np.asarray(mesh.triangles, dtype=np.int32)
     # colors.shape = n_v x 3, where n_v is the number of vertices, 3 channel contain RGB
    colors = np.asarray(mesh.vertex_colors)
    rearranged_colors = colors[triangles]
     gnd_idx = np.argwhere((rearranged_colors[:, 0, 0] > 0.5) |
                      (rearranged_colors[:, 1, 0] > 0.5) |
                      (rearranged_colors[:, 2, 0] > 0.5))
     rest_idx = np.ones(len(rearranged_colors), np.bool)
    rest_idx[gnd_idx] = 0
     mesh_gnd.remove_triangles_by_index(rest_idx)
    mesh_rest.remove_triangles_by_index(gnd_idx)

    mesh_gnd = mesh_gnd.simplify_vertex_clustering(config['simplify_resolution'])
    mesh_gnd = mesh_gnd.filter_smooth_simple(number_of_iterations=config['number_of_iterations'])
     mesh = mesh_gnd + mesh_rest
    mesh = mesh.remove_duplicated_triangles()
     return mesh

def get_mesh_size(mesh):
```

```python
    """ functions to compute the size of mesh.
    """
    size = -1
    triangles = np.array(mesh.triangles)
    vertices = np.array(mesh.vertices)
    size += triangles.size * triangles.itemsize
    size += vertices.size * vertices.itemsize
    return size

def get_mesh_size_kb(mesh):
    """ functions to compute the size of mesh in KB.
    """
    return np.floor(get_mesh_size(mesh) / 1024.0)

def get_mesh_size_mb(mesh):
    """ functions to compute the size of mesh in MB.
    """
    return np.floor(get_mesh_size_kb(mesh) / 1024.0)

if __name__ == '__main__':
    pass
```