ENPM818Z

On-Road Automated Vehicles

L2C: SLAM for the Real World

The Frontend

v2.0

Lecturer: Z. Kootbally

Semester/Year: Fall/2025

MARYLAND APPLIED
GRADUATE ENGINEERING

Table of Contents

Prerequisites

- Follow the **installation** instructions.
- Datasets can be provided on USB stick.

Learning Objectives

By the end of this session, you will be able to:

- **LiDAR SLAM Frontend Architecture**
  - Understand the complete SLAM pipeline from raw sensor data to odometry estimates
  - Explain coordinate frames, extrinsic calibration, and pose representation
  - Understand motion compensation and its critical importance for geometric consistency
- **Data Processing and Feature Extraction**
  - Describe preprocessing steps: outlier removal, motion deskewing, and downsampling
  - Contrast curvature-based and PCA-based feature extraction methods
  - Understand why feature extraction reduces data while preserving geometric information
- **Scan Matching with ICP**
  - Explain the Iterative Closest Point algorithm and its convergence properties
  - Compare point-to-point, point-to-plane, and point-to-line ICP variants
  - Understand why point-to-plane ICP is superior for structured environments
  - Solve for optimal transformation using SVD-based methods
  - Implement correspondence finding, outlier rejection, and convergence checking
- **Practical SLAM Systems**
  - Compare landmark systems: LOAM, LeGO-LOAM, LIO-SAM, and FAST-LIO2
  - Understand the keyframe strategy for scalable, efficient mapping
  - Recognize failure modes and mitigation strategies for robust real-world operation

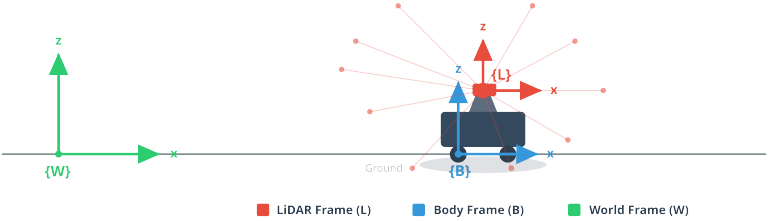
Coordinate Frames

A **coordinate frame** (or reference frame) is a coordinate system used to specify positions and orientations in 3D space. It consists of:

- An **origin** — a fixed point in space
- Three **orthogonal axes** — X, Y, Z directions
- Units of measurement (typically meters)

Coordinate Frames

Why do we need multiple frames? Different sensors and objects have their own natural coordinate systems. We need transformations to relate measurements between them.



- **LiDAR Frame (L):**
  - Sensor's local coordinates
  - Where raw points are measured
- **Body Frame (B):**
  - Robot center
  - Fixed to the robot
- **World Frame (W):**
  - Global map coordinates
  - Fixed in the environment

The Transformation Chain: ${}^{W}\mathbf{p} = {}^{W}\mathbf{T}_{B}\,{}^{B}\mathbf{T}_{L}\,{}^{L}\mathbf{p}$

- ${}^{L}\mathbf{p}$ — 3D point as measured by LiDAR sensor
- ${}^{B}\mathbf{T}_{L}$ — Extrinsic calibration (LiDAR mounting position/orientation, **fixed**)
- ${}^{W}\mathbf{T}_{B}$ — Robot pose in world coordinates (**what SLAM estimates!**)
- ${}^{W}\mathbf{p}$ — Point in global map coordinates

Rigid-Body Transformation: A transformation **T** that preserves distances and angles between points. It consists of a rotation **R** and translation **t**, transforming a point **s** as: $\mathbf{s}' = \mathbf{R}\mathbf{s} + \mathbf{t}$.

Homogeneous Form

$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0\,0\,0 & 1 \end{bmatrix} \in \mathbb{R}^{4\times4}$$

- $\mathbf{R} \in SO(3)$ — 3×3 rotation matrix
- $\mathbf{t} \in \mathbb{R}^3$ — 3×1 translation vector

Applying a Transform

$$\begin{bmatrix} \mathbf{p}' \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0\,0\,0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix}$$

Or simply: $\mathbf{p}' = \mathbf{R}\mathbf{p} + \mathbf{t}$

LiDAR mounted on Robot

- 0.3m forward
- 0.1m to the right
- 0.2m up
- No rotation

$$^{B}\mathbf{T}_{L} = \begin{bmatrix} 1 & 0 & 0 & 0.3 \\ 0 & 1 & 0 & 0.1 \\ 0 & 0 & 1 & 0.2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- $\mathbf{R} = \mathbf{I}$ (identity, no rotation)
- $\mathbf{t} = [0.3, 0.1, 0.2]^\top$

Coordinate Frames

Extrinsic Calibration

- ${}^{B}\mathbf{T}_{L}$ — LiDAR to Body
  - **Fixed** throughout operation
  - Determined **once before SLAM runs**
  - Describes sensor mounting
- **How to obtain:**
  - CAD model measurements
  - Manual measurement with ruler
  - 3D scanning (laser tracker)
  - Data-driven calibration (advanced)

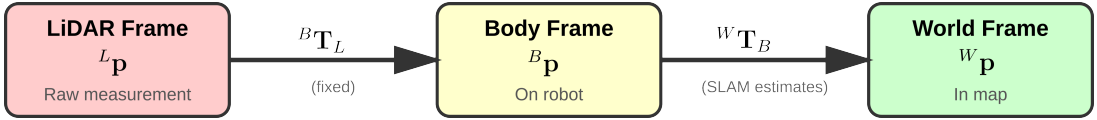
Assumption: We have accurate ${}^{B}\mathbf{T}_{L}$ before SLAM starts. Poor calibration → poor SLAM.

Robot Pose

- ${}^{W}\mathbf{T}_{B}$ — Body to World
  - **Changes** as robot moves
  - **What SLAM estimates!**
  - One pose per time step
- **Initialization:**
  - At $t = 0$, robot defines world origin
  - ${}^{W}\mathbf{T}_{B}(t_0) = \mathbf{I}_{4\times4}$
  - Starting position = $(0, 0, 0)$
  - Starting orientation = identity

SLAM's job: Estimate ${}^{W}\mathbf{T}_{B}(t)$ for all time steps t.

From LiDAR Measurement to World Coordinates



$$ {}^{W}\mathbf{p} = {}^{W}\mathbf{T}_{B}(t) \cdot {}^{B}\mathbf{T}_{L} \cdot {}^{L}\mathbf{p} $$

- **Start**: LiDAR measures point ${}^{L}\mathbf{p} = [2.5, 0.3, -0.1]^{\top}$ (in sensor coordinates)
- **Step 1**: Transform to body using calibration $\rightarrow {}^{B}\mathbf{p}$
- **Step 2**: Transform to world using current pose estimate $\rightarrow {}^{W}\mathbf{p}$
- **Result**: Point location in global map

Why This Matters for SLAM

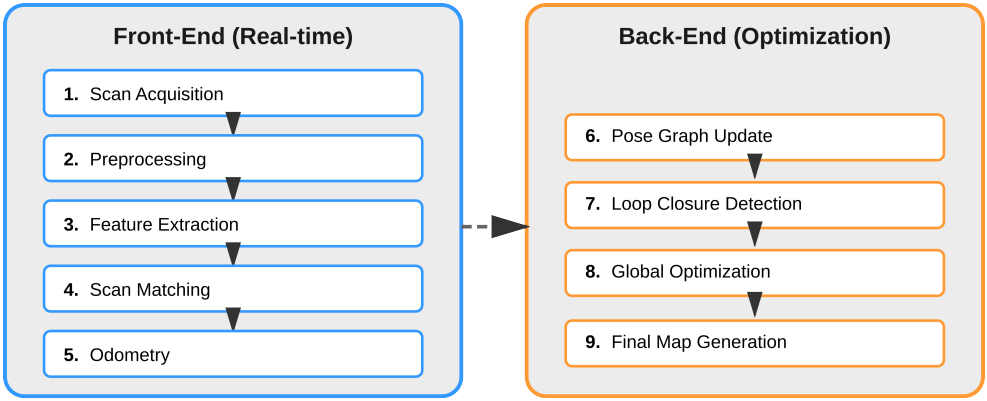
1. **Motion Compensation (Preprocessing):**
   - Each point captured at different time → different ${}^{W}\mathbf{T}_{B}(t_i)$
   - Must transform all to common reference time
2. **Scan Matching:**
   - Find $\Delta\mathbf{T}$ between consecutive scans
   - Update pose: ${}^{W}\mathbf{T}_{B}(t+1) = {}^{W}\mathbf{T}_{B}(t) \cdot \Delta\mathbf{T}$
3. **Odometry:**
   - Chain of transformations: ${}^{W}\mathbf{T}_{B}(t_k) = {}^{W}\mathbf{T}_{B}(t_0) \cdot \Delta\mathbf{T}_1 \cdot \Delta\mathbf{T}_2 \cdots \Delta\mathbf{T}_k$
4. **Mapping:**
   - Transform scan to world: ${}^{W}\mathbf{p}_i = {}^{W}\mathbf{T}_{B}(t) \cdot {}^{B}\mathbf{T}_{L} \cdot {}^{L}\mathbf{p}_i$
5. **Pose Graph Optimization:**
   - Optimize all poses $\{{}^{W}\mathbf{T}_{B}(t_0), {}^{W}\mathbf{T}_{B}(t_1), \ldots, {}^{W}\mathbf{T}_{B}(t_N)\}$ simultaneously

Key Insight: Every step in SLAM manipulates or estimates transformations. Understanding this notation is essential for the rest of the lecture.

Notation Summary

Symbol	Meaning	Description
${}^{L}\mathbf{p}$	Point in LiDAR frame	Raw sensor measurement
${}^{B}\mathbf{p}$	Point in body frame	After applying calibration
${}^{W}\mathbf{p}$	Point in world frame	In global map
${}^{B}\mathbf{T}_{L}$	LiDAR to body transform	Fixed extrinsic calibration
${}^{W}\mathbf{T}_{B}$	Body to world transform	Robot pose (SLAM output)
$\Delta\mathbf{T}$	Incremental transform	Motion between frames
$\mathbf{R}$	Rotation matrix	3×3, describes orientation
$\mathbf{t}$	Translation vector	3×1, describes position
$\mathbf{T}$	Homogeneous transform	4×4, combines $\mathbf{R}$ and $\mathbf{t}$

The SLAM Pipeline



10-200 Hz • Perception & Estimation

1-5 Hz • Global Consistency

Modern LiDAR SLAM Systems

LOAM, LeGO-LOAM, LIO-SAM, and FAST-LIO2

Four landmark systems that define the state-of-the-art in LiDAR SLAM

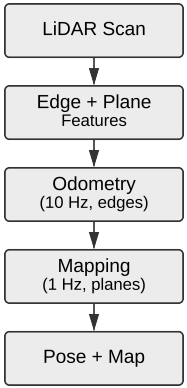
LOAM (LiDAR Odometry and Mapping) [5]

Key Innovation:

- Two-algorithm framework
- Odometry (10 Hz) + Mapping (1 Hz)
- Edge features for speed
- Planar features for accuracy

Architecture:

- Curvature-based feature extraction
- Point-to-line ICP (edges)
- Point-to-plane ICP (planes)
- LiDAR only, no IMU
- No loop closure



Performance:

- Speed: 10 Hz
- Drift: 1-5% (no loops)
- Accuracy: Excellent locally

Strengths:

- ✔ Established modern approach
- ✔ Real-time capable
- ✔ Conceptually clear

Limitations:

- ✘ No IMU fusion
- ✘ Accumulates drift
- ✘ No global consistency

Legacy: LOAM established the foundation for feature-based LiDAR SLAM and inspired virtually all subsequent systems. Its two-stage approach remains influential today.

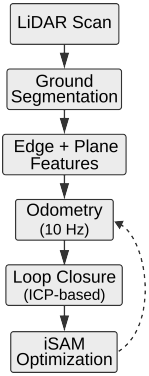
LeGO-LOAM: Lightweight and Ground-Optimized LOAM [2]

Key Innovation:

- Ground plane segmentation
- Lightweight feature extraction
- ICP-based loop closure
- Two-step optimization

Architecture:

- Ground point removal
- Point cloud segmentation
- Feature extraction (edge + plane)
- Two-stage scan matching
- Loop closure via ICP
- Pose graph optimization (iSAM)



Performance:

- Speed: 10 Hz
- Drift (no loops): 0.5-3%
- Drift (with loops): 0.2-1%
- Optimized for ground vehicles

Strengths:

- ✔ Adds loop closure to LOAM
- ✔ Ground-optimized for vehicles
- ✔ Lightweight and efficient
- ✔ Good for outdoor environments

Limitations:

- ✘ No IMU fusion
- ✘ ICP-only loop closure (slow)
- ✘ Less robust than modern systems
- ✘ Relies on ground assumption

Legacy: LeGO-LOAM bridged the gap between LOAM and modern systems by adding loop closure and backend optimization, while introducing ground segmentation for improved performance on ground vehicles.

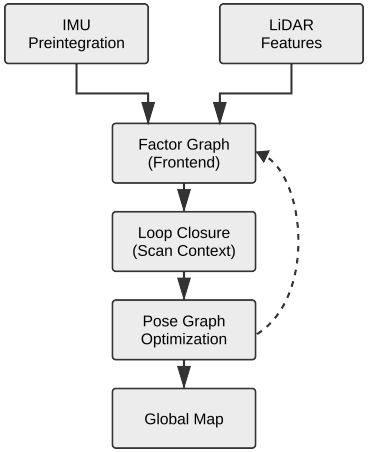
LIO-SAM: LiDAR Inertial Odometry via Smoothing and Mapping [3]

Key Innovation:

- Tight LiDAR-IMU fusion
- Factor graph framework (GTSAM)
- Full backend optimization
- Scan Context loop closure

Architecture:

- IMU pre-integration
- Feature-based ICP
- Factor graph optimization
- Loop closure detection
- Pose graph optimization
- Optional GPS integration



Performance:

- Speed: 10-20 Hz
- Drift: 0.1-0.5% (with loops)
- Final: 0.05-0.2% (after backend)

Strengths:

- ✔ Complete SLAM pipeline.
- ✔ Globally consistent maps.
- ✔ Robust to aggressive motion.
- ✔ Open-source, well-documented.

Limitations:

- ✘ More complex than LOAM.
- ✘ Higher computational cost.
- ✘ Requires good IMU.

Impact: LIO-SAM represents the state-of-the-art in complete SLAM systems, combining tight sensor fusion with global optimization for production-ready performance.

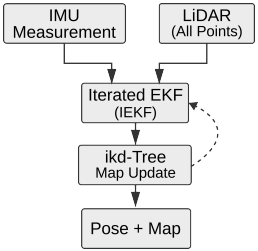
FAST-LIO2: Fast Direct LiDAR-Inertial Odometry [4]

Key Innovation:

- Direct method (no features!)
- Uses ALL points in cloud
- Iterated Extended Kalman Filter
- Incremental k-d tree (ikd-Tree)

Architecture:

- IEKF state estimation
- Point-to-plane residuals
- Dynamic map management
- Measurement-level fusion
- No feature extraction overhead
- No loop closure (odometry only)



Performance:

- Speed: 100+ Hz capable
- Drift: 0.05-0.3% (odometry)
- 10-100× faster than LOAM

Strengths:

- ✔ Extremely fast (100+ Hz)
- ✔ No feature extraction
- ✔ Dense point usage
- ✔ Excellent odometry accuracy

Limitations:

- ✘ No loop closure
- ✘ No backend optimization
- ✘ Odometry only (not full SLAM)
- ✘ Requires good IMU

Philosophy: FAST-LIO2 focuses on doing one thing exceptionally well: ultra-fast, accurate odometry. For full SLAM, combine with separate loop closure and backend modules.

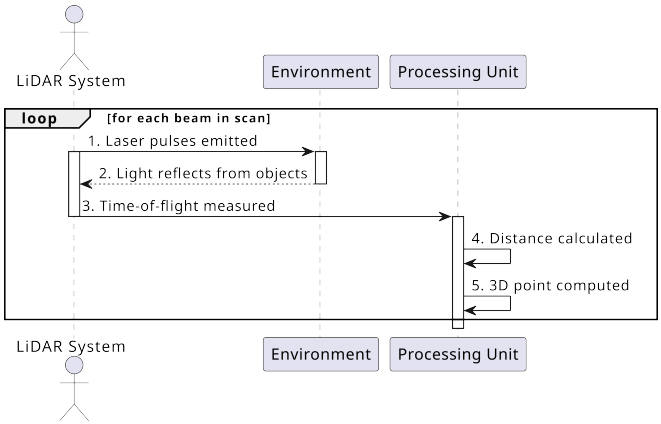
Comprehensive Comparison: Four Landmarks

Feature	LOAM (2014)	LeGO-LOAM (2018)	LIO-SAM (2020)	FAST-LIO2 (2022)
Core Architecture				
Sensors	LiDAR only	LiDAR only	LiDAR + IMU	LiDAR + IMU
IMU Fusion	✘	✘	✔ Tight	✔ IEKF
Features	Edge + Plane	Edge + Plane	Edge + Plane	All points (direct)
Ground Segmentation	✘	✔	✔	✘
SLAM Pipeline				
Frontend	2-stage ICP	2-stage ICP	Factor Graph	IEKF
Loop Closure	✘	✔ ICP-based	✔ Scan Context	✘
Backend	✘	✔ iSAM	✔ GTSAM	✘
Global Consistency	✘	✔	✔	✘
Performance				
Speed	10 Hz	10 Hz	10-20 Hz	100+ Hz
Drift (no loops)	1-5%	0.5-3%	0.1-0.5%	0.05-0.3%
Drift (with loops)	N/A	0.2-1%	0.05-0.2%	N/A
Computational Load	Medium	Medium-High	High	Medium
Best Use Case				
Primary Use	Learning baseline	Outdoor robots	Complete SLAM	Fast odometry
Strength	Foundational	Ground-optimized	Full pipeline	Maximum speed
Limitation	No fusion/loops	No IMU	Complexity	No loops

1. Scan Acquisition

Scan acquisition is the process of capturing raw 3D data from the LiDAR sensor to create a point cloud representation of the environment.

- **Point Cloud:** $P = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n\}$
- Each point: $\mathbf{p}_i = [x_i, y_i, z_i, \mathbf{a}_i]^\top$ in LiDAR frame L, where $\mathbf{a}_i$ contains attributes such as:
  - **Intensity (reflectivity)** — useful for feature extraction and loop closure.
  - **Timestamp** — important when points in a single scan are not captured simultaneously.
  - **Ring or channel ID** — identifies the laser beam in multi-beam LiDARs.
  - **Color (RGB)** — available if fused with a camera or RGB-D sensor.
  - **Confidence or range noise** — in advanced sensors or simulated data.



Key Characteristics:

- 30k-120k points per scan
- 10-20 scans per second
- 100ms acquisition time
- Points measured in LiDAR frame L

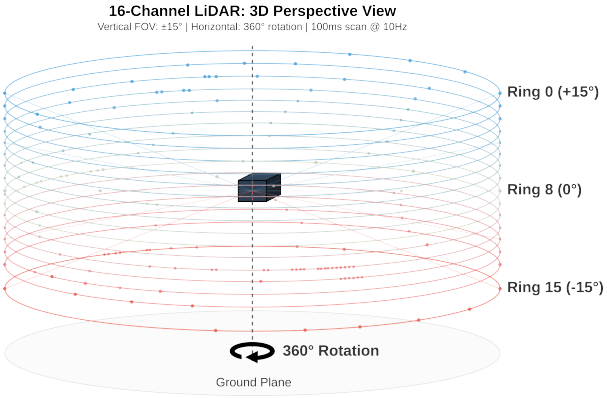
Sensor	Beams	Points/Scan	Rate
Velodyne VLP-16	16	30,000	10-20 Hz
Ouster OS1-64	64	65,000	10-20 Hz
Velodyne HDL-64E	64	120,000	10-20 Hz

LiDAR Rings:

- Each laser beam forms a horizontal ring of points
- 16-channel LiDAR = 16 vertically stacked rings
- Each ring scans 360° horizontally
- Vertical spacing: ~2° between adjacent rings
- Typical vertical FOV: ±15° for VLP-16

Scan Pattern:

- All 16 beams fire simultaneously.
- Rotate at 10-20 Hz (600-1200 RPM).
- Each ring contains ~1,875 points (100ms @ 10Hz).
- Total: ~30,000 points per scan.



Demo

ROS 2 KITTI Data Loader Node

2. Preprocessing

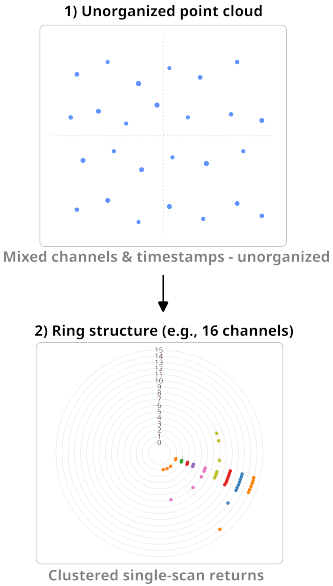
Clean and prepare raw LiDAR data for feature extraction (next step).

1. Data organization (ring structure)
2. Outlier removal (range, intensity filters)
3. **Motion compensation** (**critical!**)
4. Ground removal (optional)
5. Downsampling (optional)

Preprocessing: Preparing Raw Data

Key Steps:

1. **Data organization (ring structure):** Arrange raw packets into ordered scans according to laser channel IDs to preserve geometric consistency.
2. **Outlier removal (range, intensity filters):** Cleans LiDAR point clouds by eliminating points that are physically implausible or inconsistent with sensor behavior. Such points arise from noise, reflections, moving objects, or weather effects.
3. **Motion compensation (critical!):** Correct for sensor or vehicle movement during the scan so that all points represent the same spatial instant.
4. **Ground removal (optional):** Separate ground points using geometric constraints or height thresholds to simplify environment mapping.
5. **Downsampling (optional):** Reduce point density with a voxel-grid filter to decrease computational load for later processing steps.



Preprocessing: Preparing Raw Data

Key Steps:

1. **Data organization (ring structure):** Arrange raw packets into ordered scans according to laser channel IDs to preserve geometric consistency.
2. **Outlier removal (range, intensity filters):** Cleans LiDAR point clouds by eliminating points that are physically implausible or inconsistent with sensor behavior. Such points arise from noise, reflections, moving objects, or weather effects.
3. **Motion compensation (critical!):** Correct for sensor or vehicle movement during the scan so that all points represent the same spatial instant.
4. **Ground removal (optional):** Separate ground points using geometric constraints or height thresholds to simplify environment mapping.
5. **Downsampling (optional):** Reduce point density with a voxel-grid filter to decrease computational load for later processing steps.

- **Range filtering:** Removes points outside the sensor's valid range or those showing abrupt depth jumps caused by reflections or transparent surfaces.
- **Intensity filtering:** Discards points with very low or high reflectivity, which often indicate weak signals or overly reflective objects such as mirrors or plates.

Preprocessing: Preparing Raw Data

Key Steps:

1. **Data organization (ring structure):** Arrange raw packets into ordered scans according to laser channel IDs to preserve geometric consistency.
2. **Outlier removal (range, intensity filters):** Cleans LiDAR point clouds by eliminating points that are physically implausible or inconsistent with sensor behavior. Such points arise from noise, reflections, moving objects, or weather effects.
3. **Motion compensation (critical!):** Correct for sensor or vehicle movement during the scan so that all points represent the same spatial instant.
4. **Ground removal (optional):** Separate ground points using geometric constraints or height thresholds to simplify environment mapping.
5. **Downsampling (optional):** Reduce point density with a voxel-grid filter to decrease computational load for later processing steps.

Each LiDAR point has its own timestamp t_i. Because the vehicle moves during a scan, points are captured from slightly different poses. A motion model or odometry data estimates the pose $\mathbf{T}(t_i)$ at each instant to correct these distortions:

$$^{W}\mathbf{p}_i = \mathbf{T}(t_i)\,{}^{L}\mathbf{p}_i$$

- ${}^{L}\mathbf{p}_i$ — raw point in the LiDAR frame at time t_i
- $\mathbf{T}(t_i)$ — estimated sensor pose from odometry/IMU/GNSS
- ${}^{W}\mathbf{p}_i$ — corrected point in a common reference frame

Preprocessing: Preparing Raw Data

Key Steps:

1. **Data organization (ring structure):** Arrange raw packets into ordered scans according to laser channel IDs to preserve geometric consistency.
2. **Outlier removal (range, intensity filters):** Cleans LiDAR point clouds by eliminating points that are physically implausible or inconsistent with sensor behavior. Such points arise from noise, reflections, moving objects, or weather effects.
3. **Motion compensation (critical!):** Correct for sensor or vehicle movement during the scan so that all points represent the same spatial instant.
4. **Ground removal (optional):** Separate ground points using geometric constraints or height thresholds to simplify environment mapping.
5. **Downsampling (optional):** Reduce point density with a voxel-grid filter to decrease computational load for later processing steps.

Fusion Methods:

- **Linear interpolation:** approximate pose between scan start and end.
- **Continuous-time estimation:** use spline or B-spline trajectory fitting.

System	Deskewing Method
LOAM	IMU/Odometry
LeGO-LOAM	IMU
FAST-LIO2	Tightly-coupled IMU
LIO-SAM	IMU pre-integration

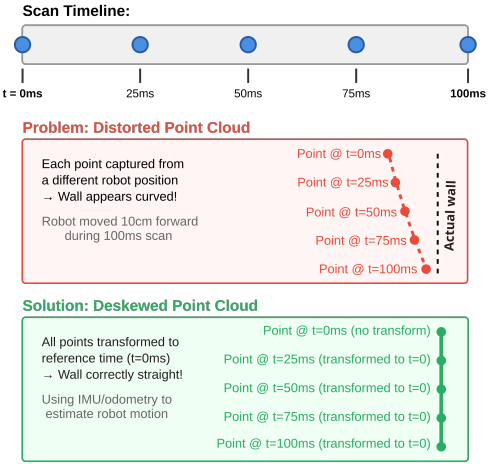
Impact:

- Without compensation — curved walls, misaligned scans, inconsistent maps.
- With compensation — straightened structures, better registration, reliable SLAM input.

Preprocessing: Preparing Raw Data

Key Steps:

1. **Data organization (ring structure):** Arrange raw packets into ordered scans according to laser channel IDs to preserve geometric consistency.
2. **Outlier removal (range, intensity filters):** Cleans LiDAR point clouds by eliminating points that are physically implausible or inconsistent with sensor behavior. Such points arise from noise, reflections, moving objects, or weather effects.
3. **Motion compensation (critical!):** Correct for sensor or vehicle movement during the scan so that all points represent the same spatial instant.
4. **Ground removal (optional):** Separate ground points using geometric constraints or height thresholds to simplify environment mapping.
5. **Downsampling (optional):** Reduce point density with a voxel-grid filter to decrease computational load for later processing steps.



Preprocessing: Preparing Raw Data

Key Steps:

1. **Data organization (ring structure):** Arrange raw packets into ordered scans according to laser channel IDs to preserve geometric consistency
2. **Outlier removal (range, intensity filters):** Cleans LiDAR point clouds by eliminating points that are physically implausible or inconsistent with sensor behavior. Such points arise from noise, reflections, moving objects, or weather effects
3. **Motion compensation (critical!):** Correct for sensor or vehicle movement during the scan so that all points represent the same spatial instant
4. **Ground removal (optional):** Separate ground points using geometric constraints or height thresholds to simplify environment mapping
5. **Downsampling (optional):** Reduce point density with a voxel-grid filter to decrease computational load for later processing steps

Ground points (e.g., roads, floors) dominate LiDAR scans, especially for mobile robots or vehicles. Removing them helps:

- Focus on obstacles and structures above the ground
- Simplify segmentation, clustering, and mapping
- Improve efficiency in SLAM and object detection pipelines

Preprocessing: Preparing Raw Data

Key Steps:

1. **Data organization (ring structure):** Arrange raw packets into ordered scans according to laser channel IDs to preserve geometric consistency
2. **Outlier removal (range, intensity filters):** Cleans LiDAR point clouds by eliminating points that are physically implausible or inconsistent with sensor behavior. Such points arise from noise, reflections, moving objects, or weather effects
3. **Motion compensation (critical!):** Correct for sensor or vehicle movement during the scan so that all points represent the same spatial instant
4. **Ground removal (optional):** Separate ground points using geometric constraints or height thresholds to simplify environment mapping
5. **Downsampling (optional):** Reduce point density with a voxel-grid filter to decrease computational load for later processing steps

Common Approaches:

- **Height Thresholding:** Remove points near a fixed reference height (e.g., $|z_i - z_{\text{ref}}| < \tau_z$). Fast but unreliable on sloped terrain
- **Grid Elevation:** Divide space into 2D cells; the lowest z in each defines ground. Points above a threshold are non-ground. Handles mild unevenness
- **Plane Fitting (RANSAC):** Fit a plane $ax + by + cz + d = 0$ to low points; those close to it are ground. Accurate but more computationally demanding
- **Slope-based Growing:** Expand ground regions where local slope $\tan(\theta) = \frac{|z_j - z_i|}{\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}}$ is below a threshold. Works well on varied terrain.

Preprocessing: Preparing Raw Data

Key Steps:

1. **Data organization (ring structure):** Arrange raw packets into ordered scans according to laser channel IDs to preserve geometric consistency
2. **Outlier removal (range, intensity filters):** Cleans LiDAR point clouds by eliminating points that are physically implausible or inconsistent with sensor behavior. Such points arise from noise, reflections, moving objects, or weather effects
3. **Motion compensation (critical!):** Correct for sensor or vehicle movement during the scan so that all points represent the same spatial instant
4. **Ground removal (optional):** Separate ground points using geometric constraints or height thresholds to simplify environment mapping.
5. **Downsampling (optional):** Reduce point density with a voxel-grid filter to decrease computational load for later processing steps

Light reduction of very dense raw LiDAR data so later stages (feature extraction, segmentation) run in real time. Keeps geometric structure intact.

- **Example**: Simple voxel-grid filter (large voxel size, e.g., 0.2 m).
  - Using a larger voxel size (e.g., 0.2 m) means nearby points are merged together, reducing point density and noise while preserving overall structure (ideal for faster processing and smoother maps).

Demo

ROS 2 Voxel Grid Downsampler Node

3. Feature Extraction

LiDAR scans produce thousands of points per frame. **Feature extraction** identifies the most informative ones (edges and planar regions) to simplify **registration** and **mapping**.

$$P = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n\}, \quad \mathbf{p}_i = [x_i,\ y_i,\ z_i]^\top$$

From P, feature extraction identifies:

$$\mathcal{F} = \mathcal{F}_{\text{edge}} \cup \mathcal{F}_{\text{plane}}$$

The system reduces the raw scan P to a compact set of structural features, improving speed and stability of the SLAM process. These selected features capture the structure of the environment while keeping the data compact and efficient for downstream SLAM algorithms.

Approaches

Two main approaches exist:

1. **Curvature-Based Feature Selection:** Direct geometric analysis.
   - Compute local curvature from ring neighborhoods
   - Select edge (high curvature) and planar (low curvature) points
   - Fast, ring-structured LiDARs
2. **Voxel+PCA:** Downsampling with normal estimation.
   - Voxelize point cloud
   - Compute centroids for downsampling
   - Extract surface normals via PCA
   - Better for unstructured clouds, robust plane fitting

We will explore both methods as they represent different philosophies in LiDAR SLAM.

Approach 1: Curvature-Based Feature Selection

Goal: Identify which points are most informative for scan matching by classifying local geometry.

Key Objectives:

- **Feature Selection:** Identify edge points (corners, boundaries) and planar points (smooth surfaces).
- **Data Reduction:** ~100k points → ~4k-6k features (96% reduction).
- **Computational Efficiency:** Fast distance-based computation (no Eigen decomposition).
- **Matching Reliability:** Complementary geometric constraints from edges and planes.

This approach answers: *"Which points should I pay attention to?"*

Curvature Computation

For each point $\mathbf{p}_i$, compute curvature from neighboring points on the same LiDAR ring.

Original LOAM formula:

$$c_i = \frac{1}{|\mathcal{S}| \cdot \|\mathbf{p}_i\|} \left\| \sum_{j \in \mathcal{S}} (\mathbf{p}_i - \mathbf{p}_j) \right\|$$

- $\mathcal{S}$ — $2m$ consecutive neighbors on the same ring (typ. m=5 ⇒ $|\mathcal{S}|$=10)
- $\|\mathbf{p}_i\|$ — range normalization
- c_i — smoothness/roughness score

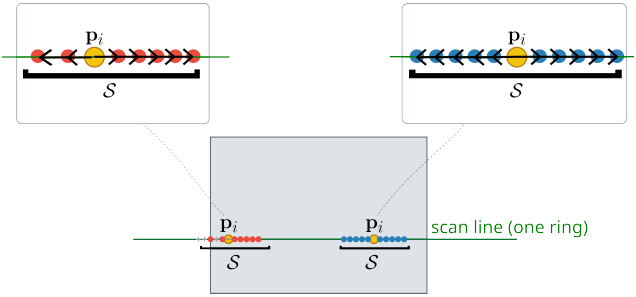
Intuition: Measures how far $\mathbf{p}_i$ deviates from its local neighborhood centroid; symmetric neighborhoods ⇒ small c_i.

Note: Points near occlusions or with insufficient valid neighbors are filtered out before scoring.

☰ Curvature → Feature Selection

Classification (per ring):

- **Edge features:** largest c_i (top ~2%)
  - Corners, depth discontinuities
  - Used in point-to-line matching
- **Planar features:** smallest c_i (bottom ~2%)
  - Smooth walls, floors, roads
  - Used in point-to-plane matching
- **Mid-range** c_i: discarded for efficiency/stability



One ring: planar segment (small c_i) vs. edge (large c_i).

- Planar: $c_i \approx 10^{-3}$
- Edge/corner: $c_i \approx 5 \times 10^{-1}$
- Occlusion boundary: $c_i \approx 1.0$ (often filtered)

Output: Sparse, labeled features (~4–6k from ~100k points) for real-time scan matching.

One ring with 1,800 points after computing curvature

- **Step 1: Sort by curvature**

$\mathbf{p}_i$	c_i	Class
$\mathbf{p}_1$	0.001	Planar
$\mathbf{p}_2$	0.002	Planar
⋮	⋮	⋮
$\mathbf{p}_{36}$	0.020	Planar
$\mathbf{p}_{37}$	0.035	Ignored
$\mathbf{p}_{38}$	0.042	Ignored
⋮	⋮	⋮
$\mathbf{p}_{1764}$	0.158	Ignored
$\mathbf{p}_{1765}$	0.892	Edge
$\mathbf{p}_{1766}$	0.905	Edge
⋮	⋮	⋮
$\mathbf{p}_{1800}$	1.567	Edge

- **Step 2: Apply thresholds**
  - **Bottom 2%:**
    - $1800 \times 0.02 = 36$ points
    - Select: $\mathbf{p}_1$ to $\mathbf{p}_{36}$
    - These are **planar features**
    - Low c_i → smooth surfaces
  - **Top 2%:**
    - $1800 \times 0.02 = 36$ points
    - Select: $\mathbf{p}_{1765}$ to $\mathbf{p}_{1800}$
    - These are **edge features**
    - High c_i → sharp geometry
  - **Result for this ring:**
    - 36 planar + 36 edge = 72 features
    - 1,728 points discarded (96%)

Approach 2: Voxel + PCA

Goal: Transform dense, irregular point clouds into a compact representation with geometric structure.

Three-Step Process:

1. **Voxelization**: Divide 3D space into a uniform grid of cubic cells (voxels). Group all points that fall within each voxel together.
2. **Compute Centroids**: For each voxel, calculate a single representative point by averaging all points inside. This drastically reduces the number of points while maintaining spatial coverage.
3. **Extract Surface Normals**: Analyze how points are distributed within each voxel to determine the local surface orientation. The normal vector points perpendicular to the underlying surface.

Surface normals enable robust plane fitting and point-to-plane matching, which are critical for algorithms such as LOAM, LeGO-LOAM, and FAST-LIO2.

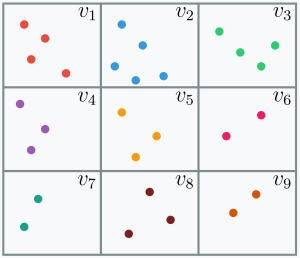
Voxelization: Divide 3D space into a uniform grid of cubic cells (voxels). Group all points that fall within each voxel together.

$$V = \{v_k\}_{k=1}^{N_v}, \quad v_k = \text{set of points in voxel } k$$

Original Point Cloud

Dense, non-uniform distribution

Voxel Grid

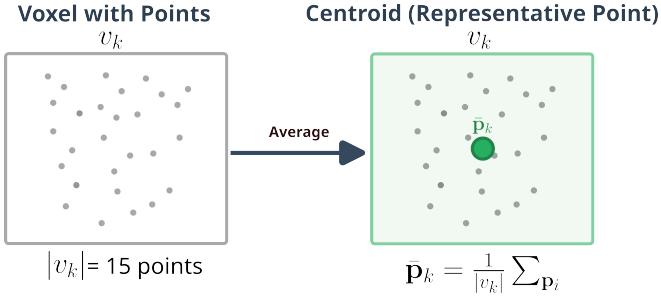


$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}$

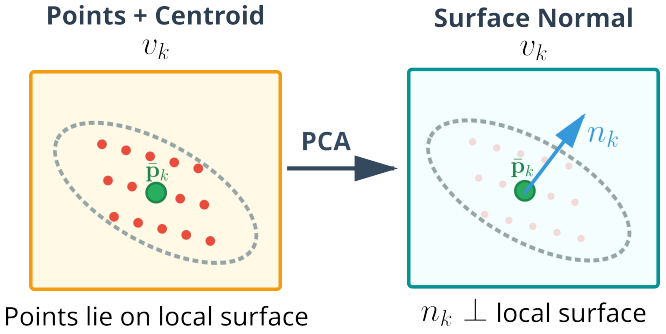
Points grouped by spatial proximity

Compute Centroids: For each voxel v_k, calculate a single representative point $\bar{\mathbf{p}}_k$ by averaging all points inside. This drastically reduces the number of points while maintaining spatial coverage.

$$\bar{\mathbf{p}}_k = \frac{1}{|v_k|} \sum_{\mathbf{p}_i \in v_k} \mathbf{p}_i$$



Extract Surface Normals: Analyze how points are distributed within each voxel to determine the local surface orientation. The normal vector points perpendicular to the underlying surface.



One approach used to extract surface normals is Principal Component Analysis (PCA).

Principal Component Analysis (PCA): PCA is a mathematical technique that finds the main directions of variation in a dataset (point cloud).

- **The Process:**
  1. **Center the Data**: Compute centroid $\bar{\mathbf{p}}_k$ and measure all points relative to it
  2. **Covariance Matrix**: Captures how points spread in all directions

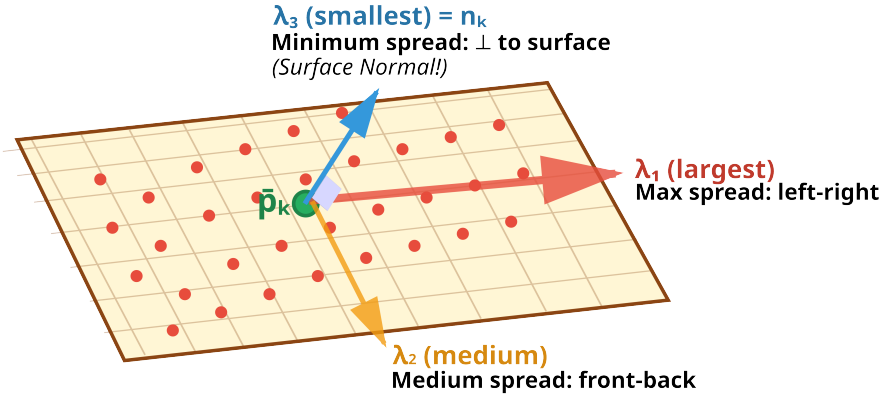
$$\mathbf{C}_k = \frac{1}{|\mathcal{V}_k|} \sum_{\mathbf{p}_i \in \mathcal{V}_k} (\mathbf{p}_i - \bar{\mathbf{p}}_k)(\mathbf{p}_i - \bar{\mathbf{p}}_k)^\top \quad \text{(3×3 matrix)}$$

  3. **Eigenvalue Decomposition**: Find principal directions

$$\mathbf{C}_k = \mathbf{V}\Lambda\mathbf{V}^\top, \quad \Lambda = \text{diag}(\lambda_1, \lambda_2, \lambda_3), \quad \lambda_1 \geq \lambda_2 \geq \lambda_3$$

- **Eigenvalues & Eigenvectors:**
  - λ_1 (largest): Maximum variance direction — spread *along* surface
  - λ_2 (medium): Second variance direction — spread *along* surface
  - λ_3 (smallest): Minimum variance direction — spread *perpendicular* to surface

Key Insight: For planar surfaces, the smallest eigenvalue $\lambda_3 \approx 0$. Its corresponding eigenvector, which points in the direction of minimum variance, is the **surface normal** $\mathbf{n}_k$.



λ_1 (Largest Eigenvalue)
Points spread widely left-right across table
High variance

λ_2 (Medium Eigenvalue)
Points spread moderately front-back across table
Medium variance

λ_3 (Smallest Eigenvalue)
Points barely spread perpendicular to table
≈ 0 variance → Normal!

Feature Extraction: Summary

- **Curvature-Based (LOAM)**
  - Compute c_i for all points.
  - Organize by LiDAR rings.
  - Select top/bottom 2% per ring.
  - Output: Edge + planar point labels.
  - Fast, simple computation.
  - 96% data reduction.
- **Best for:**
  - Point-to-point matching.
  - Point-to-line matching.
  - Real-time odometry.
  - Structured LiDAR data.

- **Voxel + PCA (LeGO-LOAM, FAST-LIO2)**
  - Voxelize point cloud.
  - Compute centroids.
  - Extract surface normals via PCA.
  - Output: Points + normal vectors.
  - Robust geometric representation.
  - Uniform spatial density.
- **Best for:**
  - Point-to-plane ICP.
  - Plane fitting and segmentation.
  - Unstructured point clouds.
  - Dense mapping.

Key Takeaway: Both methods transform raw sensor data (~100k points) into information-rich representations suitable for scan matching and odometry (the next steps in the SLAM pipeline).

Demo

ROS 2 LOAM Feature Extraction Node

3. Registration

Align two geometric datasets (point clouds) by finding the optimal spatial transformation between them.

- **Goal:** Given two point clouds, find the rigid-body transformation $(\mathbf{R}, \mathbf{t})$ that brings them into alignment.
- **Input:** Source point cloud $\mathcal{S} = \{\mathbf{s}_1, \mathbf{s}_2, \dots, \mathbf{s}_n\}$ and target point cloud $\mathcal{T} = \{\mathbf{t}_1, \mathbf{t}_2, \dots, \mathbf{t}_m\}$.
- **Output:** Rotation matrix $\mathbf{R} \in SO(3)$ and translation vector $\mathbf{t} \in \mathbb{R}^3$ that minimize the alignment error.
- Registration is fundamental to scan matching, map merging, and loop closure in SLAM.

4. Scan Matching

Estimate robot motion by aligning current LiDAR scan with previous scan or a map.

- We have a source point cloud (current scan, $\mathcal{S}$) and a target point cloud (previous scan or map, $\mathcal{T}$).
- We want to find the registration between these two point clouds. That is, the rigid-body transformation (a rotation $\mathbf{R}$ and translation $\mathbf{t}$) that best aligns the source scan onto the target.
- The result of scan matching is an estimate of the robot's motion (its odometry). This transformation is used to update the robot's pose and extend the map.

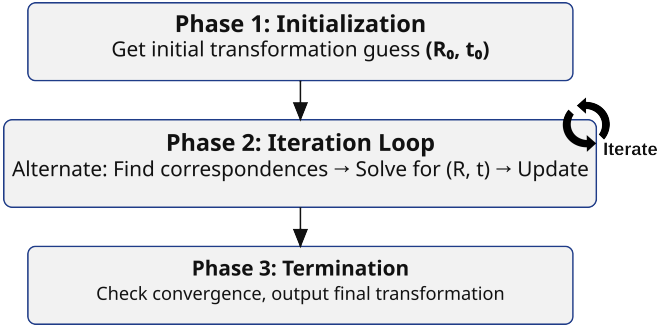
Correspondence: A correspondence is a pair of points $(\mathbf{s}_i, \mathbf{t}_j)$ where $\mathbf{s}_i$ belongs to the source point cloud $\mathcal{S}$ and $\mathbf{t}_j$ belongs to the target point cloud $\mathcal{T}$. The point $\mathbf{t}_j$ is chosen as the best match for $\mathbf{s}_i$. Finding these correspondences is the essential step for estimating the transformation $(\mathbf{R}, \mathbf{t})$ that aligns the two scans.

Iterative Closest Point (ICP) [1] is the classic and most fundamental algorithm for scan matching. It is used for the fine alignment of two point clouds.

Core Idea: Given an initial guess of the robot's motion, ICP iteratively refines the transformation (rotation $\mathbf{R}$ and translation $\mathbf{t}$) by finding corresponding points between the current and reference scans and minimizing their distance.

The Chicken-and-Egg Problem: To find the best transformation ($\mathbf{R}, \mathbf{t}$), you need to know which points correspond. **But...** to figure out which points correspond, you need to know the transformation!

The Complete ICP Algorithm



Key Insight: ICP solves the “chicken-and-egg” problem by iterating. Each iteration refines both the correspondences AND the transformation, gradually improving the alignment.

ICP Variants

Different Error Metrics

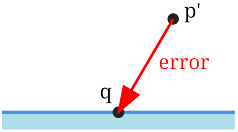
The “classic” ICP uses point-to-point error. However, modifying the error metric leads to variants that are often far more effective in SLAM.

Three Main Variants

The key difference: How we measure distance between corresponding points

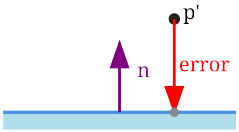
Variant	Error Type	Best For
Point-to-Point	3D Euclidean	Teaching, simple cases
Point-to-Plane	Perpendicular only	**SLAM Standard**
Point-to-Line	Distance to edge	Edge features

Point-to-Point



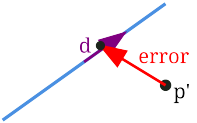
Full 3D distance

Point-to-Plane



Perpendicular only

Point-to-Line



Distance to edge

Variant 1: Point-to-Point ICP

Error Metric: Minimizes direct Euclidean distance.

$$E_{\text{point-to-point}} = \sum_{i=1}^{N} w_i \left\| (\mathbf{R}\mathbf{p}_i + \mathbf{t}) - \mathbf{q}_i \right\|^2$$

- **Intuition:**
  - Force each source point exactly onto target
  - Measures full 3D displacement
  - Penalizes all motion equally
- **Characteristics:**
  - ✔ Simplest to understand
  - ✔ Closed-form SVD solution
  - ✘ Slow convergence (50-100 iter)
  - ✘ Poor on planar surfaces
- **Behavior Example:** Sliding along a wall.
  - Robot moves 10cm along wall
  - Points still on surface
  - But large point-to-point error!
  - Algorithm wastes iterations
  - Trying to fix non-problem
- **Use Cases:**
  - Educational purposes
  - Quick prototyping
  - Very sparse point clouds
  - When normals unavailable

Variant 2: Point-to-Plane ICP (Industry Standard)

Error Metric: Minimizes distance to tangent plane.

$$E_{\text{point-to-plane}} = \sum_{i=1}^{N} w_i \left[((\mathbf{R}\mathbf{p}_i + \mathbf{t}) - \mathbf{q}_i) \cdot \mathbf{n}_i \right]^2$$

where $\mathbf{n}_i$ is the surface normal at target point $\mathbf{q}_i$

- **Intuition:**
  - Allow "sliding" along surface
  - Only penalize motion perpendicular
  - Understands surface structure
- **Why It's Better:**
  - Sliding along wall = zero error ✔
  - Moving through wall = large error ✔
  - Focuses on what matters
  - Larger convergence basin
  - Much fewer iterations needed
- **Characteristics:**
  - ✔ Fast convergence (15-30 iter)
  - ✔ 2-3× faster than point-to-point
  - ✔ More robust initialization
  - ✔ Better for structured environments
  - ✘ Requires normal estimation
  - ✘ Slightly more complex math
- **Requirements:**
  - Need surface normals $\mathbf{n}_i$
  - Compute via PCA (covered earlier)
  - From local neighborhoods
  - One-time preprocessing

Variant 3: Point-to-Line ICP

Error Metric: Minimizes distance to line feature

$$E_{\text{point-to-line}} = \sum_{i=1}^{N} w_i \left\| ((\mathbf{R}\mathbf{p}_i + \mathbf{t}) - \mathbf{q}_i) \times \mathbf{d}_i \right\|^2$$

where $\mathbf{q}_i$ is a point on the line and $\mathbf{d}_i$ is the line's direction vector

- **Intuition:**
  - Constrains perpendicular to edges
  - Allows sliding along edges
  - Specialized for linear features
- **Characteristics:**
  - ✔ Fast convergence for edges
  - ✔ Complements point-to-plane
  - ✘ Requires edge detection
  - ✘ Less general purpose
- **When to Use:**
  - Strong edge features present
  - Corners, poles, building edges
  - Combined with planar features
  - Feature-based SLAM systems
- **LOAM's Approach:**
  - Extract edge AND planar features
  - Point-to-line for edges
  - Point-to-plane for planes
  - Combine in joint optimization
  - Best of both worlds!

Comparison: Convergence Speed

Same scenario, different metrics:

Iteration	Point-to-Point	Point-to-Plane	Improvement
0	1247.3	1247.3	-
5	342.1	15.3	22×
10	156.8	1.8	87×
15	78.4	0.3	261×
20	34.2	**0.08**	**Converged!**
30	12.1	-	-
50	2.3	-	-
80	**0.09**	-	Converged

Key Observations:

- Point-to-plane: 20 iterations → converged
- Point-to-point: 80 iterations → converged
- **4× fewer iterations with point-to-plane**
- **4× faster in wall-time**

Solving the Transformation

The Point-to-Point Solution

This is the core mathematical step for the **point-to-point** metric. It finds the optimal transformation when we have a set of known correspondences.

Problem Setup

Given Information:

- **Two Point Clouds:**
  - **Source cloud** (current scan): $\mathcal{P} = \{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_N\}$
  - **Target cloud** (reference scan/map): $\mathcal{Q} = \{\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_N\}$
- **Known Correspondences:**
  - We have N corresponding pairs: $\{(\mathbf{p}_1, \mathbf{q}_1), (\mathbf{p}_2, \mathbf{q}_2), \dots, (\mathbf{p}_N, \mathbf{q}_N)\}$
    - Each $\mathbf{p}_i$ is matched to exactly one $\mathbf{q}_i$

Goal: Find the optimal rotation $\mathbf{R}$ and translation $\mathbf{t}$ that best aligns the source to the target.

Key Assumption: We are assuming correspondences are already known for one iteration of ICP.

The Weighted Error Metric

Objective Function: We want to minimize the weighted sum of squared distances between corresponding points (the point-to-point error):

$$E(\mathbf{R}, \mathbf{t}) = \sum_{i=1}^{N} w_i \left\| (\mathbf{R}\mathbf{p}_i + \mathbf{t}) - \mathbf{q}_i \right\|^2 \rightarrow \min$$

Components:

- $\mathbf{R}\mathbf{p}_i + \mathbf{t}$ — transformed source point
- $\mathbf{q}_i$ — corresponding target point
- $\| \cdots \|^2$ — squared Euclidean distance
- w_i — weight for point pair i

Goal: Find $(\mathbf{R}^*, \mathbf{t}^*)$ that minimizes $E(\mathbf{R}, \mathbf{t})$

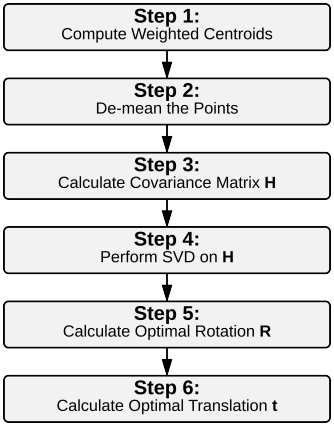
Understanding Weights w_i

Weights allow us to assign varying importance to different point pairs.

Common Weight Strategies:

1. **Uniform Weighting (Standard ICP):**
   - All $w_i = 1$
   - Every correspondence treated equally
   - Simplest approach
2. **Measurement Confidence:**
   - Higher weight for points with lower measurement uncertainty
   - Example: Closer points often have better accuracy
3. **Geometric Significance:**
   - Higher weight for distinctive features (corners, edges)
   - Lower weight for planar regions with less constraint
4. **Outlier Handling:**
   - Down-weight potential outliers
   - Robust M-estimators assign weights based on residual size

Closed-Form Solution via SVD



This solution is exact and efficient. No iterative optimization needed for this step!

Step 1: Compute Weighted Centroids

Calculate the weighted center of mass for both point clouds:

$$\bar{\mathbf{p}} = \frac{\sum_{i=1}^{N} w_i \mathbf{p}_i}{\sum_{i=1}^{N} w_i} \quad \text{and} \quad \bar{\mathbf{q}} = \frac{\sum_{i=1}^{N} w_i \mathbf{q}_i}{\sum_{i=1}^{N} w_i}$$

Where:

- $\bar{\mathbf{p}}$ is the weighted centroid of the source cloud
- $\bar{\mathbf{q}}$ is the weighted centroid of the target cloud
- If all $w_i = 1$, these reduce to simple arithmetic means
- Centroids serve as reference points for alignment

Why? Centering the data simplifies the rotation calculation by removing translation effects.

Step 2: De-mean the Points

Center both point clouds by subtracting their centroids:

$$\mathbf{p}'_i = \mathbf{p}_i - \bar{\mathbf{p}} \quad \text{and} \quad \mathbf{q}'_i = \mathbf{q}_i - \bar{\mathbf{q}}$$

Result:

- Centered source cloud: $\mathcal{P}' = \{\mathbf{p}'_1, \mathbf{p}'_2, \ldots, \mathbf{p}'_N\}$
- Centered target cloud: $\mathcal{Q}' = \{\mathbf{q}'_1, \mathbf{q}'_2, \ldots, \mathbf{q}'_N\}$
- Both clouds now have their weighted centroids at the origin

Key Property: The optimal rotation between centered point clouds is the same as between the original clouds. This separation of rotation and translation simplifies the math significantly.

Step 3: Calculate the Covariance Matrix

Construct the weighted cross-covariance matrix:

$$\mathbf{H} = \sum_{i=1}^{N} w_i \mathbf{p}'_i (\mathbf{q}'_i)^\top$$

Properties:

- $\mathbf{H}$ is a 3×3 matrix
- Each term $\mathbf{p}'_i (\mathbf{q}'_i)^\top$ is an outer product (column vector × row vector)
- $\mathbf{H}$ captures the correlation between the centered point clouds
- The weights w_i modulate the contribution of each correspondence

Intuition: $\mathbf{H}$ encodes how the source and target point clouds are related. The optimal rotation can be extracted from this matrix via Singular Value Decomposition (SVD).

Step 4: Singular Value Decomposition (SVD)

Perform SVD on the covariance matrix:

$$\mathbf{H} = \mathbf{U}\Sigma\mathbf{V}^\top$$

SVD Components:

- $\mathbf{U}$ — 3×3 orthogonal matrix (left singular vectors)
- Σ — 3×3 diagonal matrix (singular values $\sigma_1, \sigma_2, \sigma_3 \geq 0$)
- $\mathbf{V}$ — 3×3 orthogonal matrix (right singular vectors)
- Orthogonal means: $\mathbf{U}^\top\mathbf{U} = \mathbf{I}$ and $\mathbf{V}^\top\mathbf{V} = \mathbf{I}$

Why SVD? It decomposes $\mathbf{H}$ into orthogonal components that directly reveal the optimal rotation. This is a fundamental result from linear algebra known as the Orthogonal Procrustes Problem.

Step 5: Calculate Optimal Rotation

Extract the rotation matrix from SVD components:

$$\mathbf{R} = \mathbf{V}\mathbf{U}^\top$$

Handling Reflections (Critical Check):

1. Compute $\det(\mathbf{R})$ (determinant of $\mathbf{R}$)
2. **If** $\det(\mathbf{R}) = +1$: ✔ Valid rotation matrix
3. **If** $\det(\mathbf{R}) = -1$: ✘ This is a reflection, not a rotation!
   - Flip the sign of the last column of $\mathbf{V}$:

$$\mathbf{V}' = \mathbf{V}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

   - Recompute: $\mathbf{R} = \mathbf{V}'\mathbf{U}^\top$

Why? Physical rotations must preserve orientation ($\det(\mathbf{R}) = +1$). Reflections flip orientation and are not valid rigid-body transformations.

Step 6: Calculate Optimal Translation

Once we have the optimal rotation, translation is straightforward:

$$\mathbf{t} = \bar{\mathbf{q}} - \mathbf{R}\bar{\mathbf{p}}$$

Interpretation:

- $\mathbf{R}\bar{\mathbf{p}}$ — rotate the source centroid
- $\bar{\mathbf{q}} - \mathbf{R}\bar{\mathbf{p}}$ — translation needed to align centroids after rotation
- This ensures the transformed source centroid matches the target centroid

Final Transformation:

For any source point $\mathbf{p}_i$, the transformed point is:

$$\mathbf{p}_i^{\text{transformed}} = \mathbf{R}\mathbf{p}_i + \mathbf{t}$$

Result: We now have the complete transformation ($\mathbf{R}, \mathbf{t}$) that optimally aligns the source to the target!

Complete Algorithm Summary

Input: Source $\mathcal{P}$, Target $\mathcal{Q}$, Correspondences, Weights $\{w_i\}$

Step 1: Weighted Centroids

$$\bar{\mathbf{p}} \leftarrow \frac{\sum_{i=1}^{N} w_i \mathbf{p}_i}{\sum_{i=1}^{N} w_i}$$

$$\bar{\mathbf{q}} \leftarrow \frac{\sum_{i=1}^{N} w_i \mathbf{q}_i}{\sum_{i=1}^{N} w_i}$$

Step 2: De-mean

for $i = 1$ to N **do**

$\mathbf{p}'_i \leftarrow \mathbf{p}_i - \bar{\mathbf{p}}$

$\mathbf{q}'_i \leftarrow \mathbf{q}_i - \bar{\mathbf{q}}$

end for

Step 3: Covariance Matrix

$$\mathbf{H} \leftarrow \sum_{i=1}^{N} w_i \mathbf{p}'_i (\mathbf{q}'_i)^\top$$

Step 4: SVD

$$[\mathbf{U}, \Sigma, \mathbf{V}] \leftarrow \text{SVD}(\mathbf{H})$$

Step 5: Rotation

$$\mathbf{R} \leftarrow \mathbf{V}\mathbf{U}^\top$$

if $\det(\mathbf{R}) < 0$ **then**

Flip last column of $\mathbf{V}$ and recompute $\mathbf{R}$

end if

Step 6: Translation

$$\mathbf{t} \leftarrow \bar{\mathbf{q}} - \mathbf{R}\bar{\mathbf{p}}$$

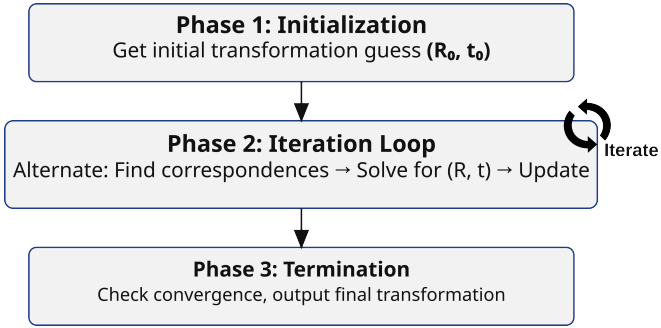
Output: $(\mathbf{R}, \mathbf{t})$

The Complete ICP Algorithm

With Unknown Correspondences

This is the complete algorithm as used in practice, where correspondences are not known in advance and must be estimated iteratively.

Algorithm Overview: Three Phases



Each iteration refines both the correspondences AND the transformation, gradually improving the alignment. Typically converges in 10-30 iterations.

Phase 1: Initialization

Goal: Provide a starting point for the iterative process

Step 1: Get Initial Transformation Guess (R_0, t_0)

Common Sources:

- **Motion Model:**
  - Wheel odometry
  - IMU integration
  - Constant velocity model
- **Identity Transform:**
  - $R_0 = I, t_0 = \mathbf{0}$
  - Assumes small motion
  - Only works for slow movement
- **Previous Frame:**
  - Use last transformation
  - Assume smooth motion

Step 2: Apply Initial Transform

Apply (R_0, t_0) to source cloud:

$$\mathscr{P}_0 = \{R_0\mathbf{p}_i + t_0 \mid \mathbf{p}_i \in \mathscr{P}\}$$

In SLAM Systems:

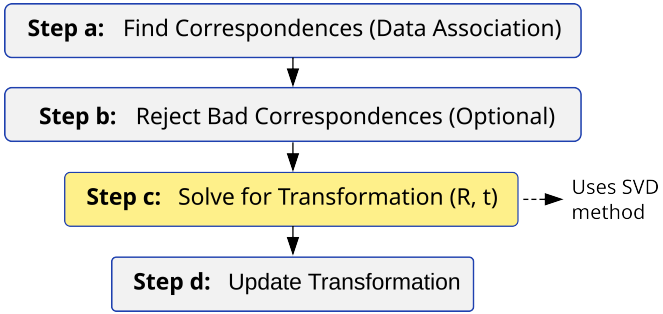
- **LOAM:** Constant velocity
- **FAST-LIO2:** IMU pre-integration
- **LIO-SAM:** IMU odometry

Critical: Good initialization is essential!

- Poor guess → wrong local minimum
- Rule of thumb: $< 30^\circ$ rotation, < 1m translation
- Bad init is #1 cause of ICP failure

Phase 2: The Iteration Loop

Repeat until convergence



The “Solve for Transformation” step uses the appropriate method for the chosen error metric (e.g., SVD for point-to-point).

Step a: Find Correspondences

Goal: Match each source point to its closest target point

The Process:

1. For each point $\mathbf{p}_i$ in transformed source cloud
2. Find nearest neighbor $\mathbf{q}_j$ in target cloud: $\mathbf{q}_j = \arg\min_{\mathbf{q} \in \mathcal{Q}} \|\mathbf{p}_i - \mathbf{q}\|$
3. Store correspondence pair: $(\mathbf{p}_i, \mathbf{q}_j)$

Computational Complexity:

Naive Approach:

- For each of N source points
- Check all M target points
- Complexity: $O(NM)$
- For 50k points: 2.5 billion comparisons!
- Too slow for real-time

K-d Tree Acceleration:

- Build k-d tree from target cloud (once)
- Query for each source point
- Complexity: $O(N \log M)$
- For 50k points: 800k operations
- 50,000× faster!

K-d tree is essential for real-time ICP. Without it, ICP would be impractical.

K-d Tree: A spatial data structure for efficient nearest neighbor search

How it works:

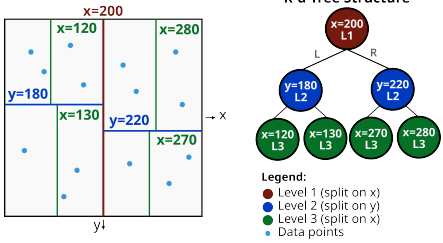
- Binary tree that partitions space
- Each level splits along a different axis (x, y, z, x, y, z, ...)
- Points stored in leaf nodes
- Allows pruning of search space

Operations:

- **Build:** $O(N \log N)$ - done once
- **Query:** $O(\log N)$ - per point
- **Memory:** $O(N)$

Libraries:

- PCL: `pcl::KdTreeFLANN`
- Open3D: `KDTreeFlann`
- SciPy: `scipy.spatial.KDTree`



Query point? Traverse tree by comparing coordinates, prune branches that can't contain nearest neighbor.

Step b: Reject Bad Correspondences

Problem: Not all nearest-neighbor matches are correct.

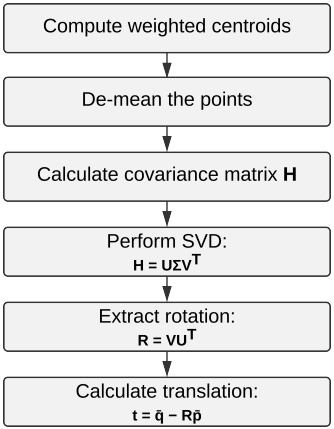
- **Sources of Bad Correspondences:**
  - Moving objects in scene
  - Occlusions and boundaries
  - Measurement noise
  - Insufficient overlap between scans
  - Still-poor alignment from previous iteration

Common Filtering Strategies

1. **Distance Thresholding:**
   - Reject if $\|\mathbf{p}_i - \mathbf{q}_j\| > d_{max}$
   - Typical: $d_{max} = 0.5\text{-}2.0$ meters
2. **Trimmed ICP:**
   - Sort pairs by distance
   - Keep best 90-95%
   - Discard worst 5-10%
3. **Normal Compatibility:**
   - Reject if surface normals differ significantly
   - Check: $\mathbf{n_p} \cdot \mathbf{n_q} > \cos(\theta_{max})$
   - Typical: $\theta_{max} = 30°\text{-}45°$
4. **Statistical Outlier Removal:**
   - Compute mean and std of distances
   - Reject if beyond 2-3 std deviations

Step c: Solve for Transformation

Given the filtered correspondences $\{(\mathbf{p}_i, \mathbf{q}_i)\}_{i=1}^{N}$:



Output: Incremental transformation $(\Delta R, \Delta t)$ for this iteration

Step d: Update Transformation

Two things to update:

1. **Transform the Source Cloud:**
   - Apply the newly computed transformation to all source points:

$$\mathbf{p}_i^{\text{new}} = \Delta R\, \mathbf{p}_i^{\text{current}} + \Delta t$$

2. **Compose Total Transformation:**
   - Update the cumulative transformation from the initial guess:

$$T_{k-1} \longrightarrow \times \longrightarrow \Delta T_k \longrightarrow = \longrightarrow T_k$$

Previous total transform | This iteration's transform | Updated total transform

   - In homogeneous form:

$$T_k = \begin{bmatrix} R_k & t_k \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R_{k-1} & t_{k-1} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \Delta R_k & \Delta t_k \\ 0 & 1 \end{bmatrix}$$

Phase 3: Termination

Convergence Criteria (stop if ANY condition is met):

1. **Error Change is Small:**

$$|E_k - E_{k-1}| < \epsilon_E \quad \text{(e.g., } \epsilon_E = 10^{-6}\text{)}$$

   where $E_k = \sum_i w_i \|\mathbf{p}_i - \mathbf{q}_i\|^2$

2. **Transformation Change is Small:**

$$\|R_k - R_{k-1}\|_F + \|t_k - t_{k-1}\| < \epsilon_T \quad \text{(e.g., } \epsilon_T = 10^{-4}\text{)}$$

3. **Maximum Iterations Reached:**

$$k \geq k_{max} \quad \text{(e.g., } k_{max} = 50 \text{ or } 100\text{)}$$

Typical Convergence:

- Well-initialized: 10-30 iterations
- Poor initialization: 50+ iterations or failure
- Each iteration: 10-100ms (depending on point count)

Putting It All Together: Pseudocode

Require: Source $\mathcal{P}$, Target $\mathcal{Q}$, Initial guess (R_0, t_0), Max iterations k_{max}
Ensure: Final transformation (R^*, t^*)

Initialize:
$k \leftarrow 0$, $R \leftarrow R_0, t \leftarrow t_0$
Build k-d tree from target $\mathcal{Q}$

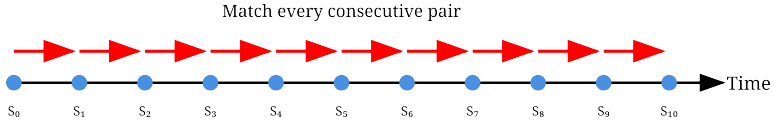
while not converged and $k < k_{max}$ **do**
  // **Step a: Find Correspondences**
  for each $\mathbf{p}_i \in \mathcal{P}$ **do**
    $\mathbf{p}'_i \leftarrow R\mathbf{p}_i + t$ //Transform
    $\mathbf{q}_i \leftarrow$ NearestNeighbor($\mathbf{p}'_i$, k-d tree)
  end for
  // **Step b: Reject Bad Correspondences**
  Filter pairs by distance threshold and/or trimming
  // **Step c: Solve for Transform**
  $(\Delta R, \Delta t) \leftarrow$ Solve_Method($\{(\mathbf{p}_i, \mathbf{q}_i)\}$, error_metric)
  // **Step d: Update**
  $R \leftarrow \Delta R \cdot R, t \leftarrow \Delta R \cdot t + \Delta t$
  $k \leftarrow k + 1$
end while
return (R, t)

Practical Considerations

Making ICP Work in Real SLAM Systems

The Problem: Scan-to-Scan Matching

Naive Approach: Match every new scan to the previous one.



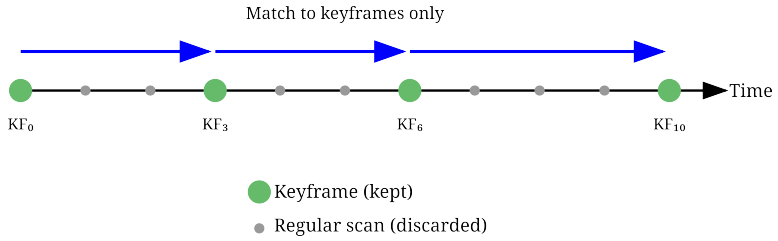
Problems with This Approach:

- **Computationally Expensive:**
  - ICP every 100ms (10 Hz)
  - Processing all points every time
  - Wastes computation on similar scans
- **Rapid Drift Accumulation:**
  - Small errors in each match
  - Errors compound over time
  - No chance for correction
- **Dense, Unwieldy Maps:**
  - Storing every single scan
  - Millions of redundant points
  - Memory explosion
- **Weak Constraints:**
  - Adjacent scans are very similar
  - Poor for loop closure
  - Minimal geometric diversity

The Solution: Keyframe Strategy

Keyframe: A strategically chosen LiDAR scan (and its corresponding robot pose) that is stored for longer-term use in map building and optimization.

Key Idea: Instead of constant stream of data, build map from sparser, more significant set of keyframes.



Why Use Keyframes?

1. Computational Efficiency:

- Reduces ICP calls
- Process only significant scans
- Real-time feasible

- Without: 100 scans → 99 ICP calls
- With: 100 scans → 10 keyframes → 9 ICP calls
- 90% reduction!

2. Map Sparsity:

- Prevents memory explosion
- Manageable map size
- Easier optimization

- 1km path, 10 Hz scanning
- Without: 100k scans stored
- With: 200 keyframes
- 500× reduction!

3. Better Constraints:

- Distinctive viewpoints
- Strong loop closures
- Better optimization

- Adjacent scans: 95% overlap
- Keyframes: 30-50% overlap
- More geometric diversity
- Stronger constraints

Keyframes are the foundation of scalable SLAM. Almost all modern systems use them: LOAM, LeGO-LOAM, LIO-SAM, FAST-LIO2, ORB-SLAM.

When to Create a Keyframe?

A new scan becomes a keyframe when the robot has moved significantly.

Common Triggering Criteria

1. Distance Threshold:

- Create keyframe if moved $> d_{threshold}$
- Typical values: 0.5m - 2.0m
- Ensures spatial separation

- Threshold: 1.0m
- Robot moves: 0.3m, 0.4m, 0.5m
- Total: 1.2m → Create keyframe!

2. Rotation Threshold:

- Create keyframe if rotated $> \theta_{threshold}$
- Typical values: 10° - 30°
- Captures viewpoint change

- Threshold: 15°
- Robot turns: 5°, 6°, 8°
- Total: 19° → Create keyframe!

Combined Criterion (Most Common):

Create keyframe if **EITHER** distance **OR** rotation threshold exceeded:

$$\text{distance} > 1.0\text{m} \quad \textbf{OR} \quad \text{rotation} > 15^\circ$$

In Practice: Systems like LIO-SAM use: 1m distance OR 15° rotation. LOAM uses similar values.

Strengths of ICP

1. **Conceptually Simple**
   - Easy to understand and implement
   - Well-documented with many libraries
   - Good starting point for learning
2. **Accurate for Fine Alignment**
   - Sub-centimeter accuracy possible
   - Excellent when properly initialized
   - Industry standard for 30+ years
3. **Guaranteed Convergence**
   - Always converges to *a* local minimum
   - Error monotonically decreases (when properly implemented)
   - Predictable behavior

Note: "Guaranteed to converge to a local minimum" means it will find *some* solution, but not necessarily the *correct* global solution if poorly initialized.

Summary: Key Takeaways

1. **ICP is the workhorse** of scan matching in LiDAR SLAM
2. **Iterative solution:** Alternates finding correspondences and computing transformation
3. **Good initialization is critical:** Use IMU, odometry, or motion model
4. **Keyframes are essential** for scalable, efficient SLAM
5. **Point-to-plane ICP** is the industry standard variant
6. **Robust strategies needed:** Outlier rejection, convergence monitoring
7. **Aware of failure modes:** Local minima, featureless scenes, dynamics

What We have Learned

ICP Algorithm

- Iterative approach
- Unknown correspondences
- K-d tree acceleration
- Outlier rejection
- Convergence criteria

ICP Variants

- Keyframe strategy
- Failure modes
- Error metric variants
- Point-to-plane superiority
- Real-world considerations

The Math (Point-to-Point)

- SVD-based closed-form solution
- Given correspondences
- Weighted least squares
- Optimal transformation
- 6-step algorithm

Skills Gained:

- Understand ICP deeply
- Implement from scratch
- Debug failures
- Choose right variant
- Deploy in SLAM system

Demo

ROS 2 LOAM Scan Matching Node

Next Class

- L2D: SLAM for the Real World: The Backend.

References I

[1] Paul J Besl and Neil D McKay. “A method for registration of 3-D shapes”. In: **IEEE Transactions on Pattern Analysis and Machine Intelligence** 14.2 (1992), pp. 239–256.

[2] Tixiao Shan and Brendan Englot. “LeGO-LOAM: Lightweight and Ground-Optimized Lidar Odometry and Mapping on Variable Terrain”. In: **IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)**. IEEE. 2018, pp. 4758–4765.

[3] Tixiao Shan et al. “LIO-SAM: Tightly-coupled Lidar Inertial Odometry via Smoothing and Mapping”. In: **IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)**. IEEE. 2020, pp. 5135–5142.

[4] Wei Xu et al. “FAST-LIO2: Fast Direct LiDAR-Inertial Odometry”. In: **IEEE Transactions on Robotics** 38.4 (2022), pp. 2053–2073.

[5] Ji Zhang and Sanjiv Singh. “Low-drift and real-time lidar odometry and mapping”. In: **Autonomous Robots** 41.2 (2017), pp. 401–416.