Tutorial report for AAE4011: ICP and LiDAR SLAM

Weisong Wen[*,1], Shaoting Qiu[1], and Xiwei Bai[*,1]

[1] Department of Aeronautical and Aviation Engineering, The Hong Kong Polytechnic University.
welson.wen@polyu.edu.hk

[*]corresponding author

Keywords: Iterative Closest Point (ICP), Simultaneously localization and mapping (SLAM), LiDAR SLAM, Artificial Intelligence (AI), Unmanned autonomous systems (UAS), Robotics.

Abstract. LiDAR-based Simultaneous Localization and Mapping (SLAM) is essential for autonomous robots to navigate unknown environments by constructing maps while estimating their position. The Iterative Closest Point (ICP) algorithm plays a central role in LiDAR SLAM, aligning sequential 3D point cloud scans to refine a robot's pose and map accuracy. By iteratively minimizing distances between corresponding points, ICP computes optimal rotations and translations between scans, enabling incremental mapping. However, challenges such as computational complexity, sensitivity to initial guesses, and performance issues in dynamic or feature-poor environments limit its reliability. To address these, modern implementations integrate sensor fusion (e.g., IMUs, odometry) for robustness, probabilistic ICP variants to handle noise, and optimizations like voxel downsampling for real-time efficiency. These advancements make ICP-based LiDAR SLAM viable for applications like self-driving cars, drones, and industrial automation. For undergraduate students, understanding ICP's principles—its strengths, limitations, and integration with complementary technologies—provides foundational insight into real-world robotic navigation systems. This abstract highlights ICP's role in bridging theoretical algorithms and practical autonomy, emphasizing its balance of precision and computational feasibility in resource-constrained robotics.

1 Introduction

Autonomous robotics relies on the ability to navigate and interact with unknown environments, a challenge addressed by Simultaneous Localization and Mapping (SLAM). This technology enables robots to build maps of their surroundings while estimating their own position within those maps. Among the sensors used for SLAM, Light Detection and Ranging (LiDAR) has become indispensable due to its precision in generating 3D point clouds, which capture spatial details with millimeter-level accuracy. LiDAR-based SLAM systems, particularly those using the Iterative Closest Point (ICP) algorithm, form the backbone of modern robotic navigation, enabling applications from self-driving cars to industrial automation. This introduction explores the historical evolution of ICP and SLAM, their integration into robotics, current applications, and the transformative potential of artificial intelligence (AI) in enhancing LiDAR SLAM systems. Simultaneous Localization and Mapping (SLAM) is a cornerstone of autonomous robotics, enabling machines to navigate and map unknown environments in real time. Among sensor modalities, Light Detection and Ranging (LiDAR) has emerged as a critical tool for SLAM due to its ability to generate high-resolution, accurate 3D point clouds, even in low-light conditions. LiDAR-based SLAM systems leverage these point clouds to simultaneously estimate a robot's trajectory and reconstruct its surroundings. A pivotal algorithm in this domain is the Iterative Closest Point (ICP), which aligns successive LiDAR scans to refine pose estimation and map consistency, forming the backbone of many LiDAR SLAM pipelines. ICP operates by iteratively

matching points between two scans, minimizing the distance between corresponding features to compute optimal transformations (rotation and translation). This process enables incremental map construction while correcting localization drift. However, ICP faces challenges: computational intensity from processing dense point clouds, susceptibility to local minima without accurate initial guesses, and performance degradation in dynamic or featureless environments. Noise, outliers, and varying scan densities further complicate convergence. Despite these limitations, ICP remains popular for its simplicity and effectiveness in structured settings, particularly when paired with optimizations like KD-trees for accelerated nearest-neighbor searches. Recent advancements address traditional ICP shortcomings through hybrid approaches. Sensor fusion with inertial measurement units (IMUs), wheel odometry, or cameras enhances robustness, while probabilistic ICP variants and machine learning improve correspondence matching and outlier rejection. Efficient implementations, such as voxel grid downsampling and parallel computing, enable real-time performance on resource-constrained platforms. Today, ICP-based LiDAR SLAM underpins diverse applications, from autonomous vehicles and drones to industrial robots and disaster-response systems. As innovations in hardware and algorithms continue, ICP's adaptability ensures its relevance in advancing autonomous navigation, balancing precision and computational efficiency to meet the demands of complex, dynamic environments.

Background and Historical Context: The concept of SLAM emerged in the 1980s as robotics researchers sought ways to enable machines to operate autonomously in unstructured environments. Early SLAM implementations relied on probabilistic methods, such as the Extended Kalman Filter (EKF) and particle filters, to fuse data from rudimentary sensors like sonar or 2D lasers. However, these approaches struggled with scalability and computational complexity in large or dynamic environments. The ICP algorithm, introduced in 1992 by Besl and McKay, revolutionized 3D point cloud registration. Originally designed for aligning geometric shapes in computer vision, ICP iteratively minimizes the distance between corresponding points in two scans to compute optimal transformations (rotation and translation). By the early 2000s, advances in LiDAR technology—such as faster scanning rates and higher resolution—made ICP a natural fit for SLAM. Researchers began integrating ICP into SLAM pipelines to align successive LiDAR scans, enabling robots to refine their pose estimates while incrementally constructing globally consistent maps.

ICP-Based LiDAR SLAM (Core Principles): ICP-based LiDAR SLAM operates in three key stages:

- Scan Matching: Sequential LiDAR scans are aligned using ICP to estimate relative motion between frames.
- Pose Graph Optimization: Loop closure detection corrects accumulated drift by identifying revisited locations, ensuring global map consistency.
- Map Integration: Aligned scans are fused into a unified 3D map, often represented as voxel grids or surfel-based models.

While ICP excels in structured environments (e.g., indoor corridors or urban streets), it faces challenges in featureless or dynamic settings. For instance, repetitive geometries (e.g., long hallways) or moving objects (e.g., pedestrians) can lead to incorrect correspondences, causing localization errors. To mitigate these issues, modern systems combine ICP with sensor fusion—integrating inertial measurement units (IMUs), wheel encoders, or cameras—to improve robustness. Additionally, optimizations like voxel grid downsampling reduce computational load, enabling real-time performance on embedded hardware.

Applications in Robotics: ICP-based LiDAR SLAM has enabled breakthroughs across robotics domains:

- Autonomous Vehicles: Self-driving cars use LiDAR SLAM to navigate complex urban environments, combining high-definition maps with real-time obstacle detection.
- Drones: UAVs leverage lightweight LiDAR systems for aerial mapping, disaster response, and precision agriculture.
- Industrial Automation: Warehouse robots rely on SLAM for inventory management and path planning in dynamic fulfillment centers.
- Search and Rescue: Ground robots map collapsed structures to locate survivors while avoiding hazards.

These applications highlight the versatility of ICP-based SLAM, but they also expose its limitations. For example, long-term autonomy in highly dynamic environments (e.g., crowded public spaces) remains an open challenge due to persistent localization drift.

The Role of AI in Advancing LiDAR SLAM: Recent advancements in AI and machine learning are reshaping ICP-based SLAM systems:

- Enhanced Correspondence Matching: Deep learning models, such as PointNet or 3D convolutional neural networks (CNNs), improve feature extraction from point clouds, enabling more accurate and faster ICP iterations.
- Dynamic Object Handling: AI-driven semantic segmentation identifies and filters moving objects (e.g., cars, pedestrians) from LiDAR data, reducing localization errors.
- Adaptive ICP Variants: Reinforcement learning (RL) optimizes ICP parameters (e.g., convergence thresholds) in real time, adapting to environmental changes.
- End-to-End SLAM: Neural architectures like Neural Radiance Fields (NeRFs) or Graph Neural Networks (GNNs) are being explored to replace traditional pipelines, learning mapping and localization directly from sensor data.

These AI-powered innovations address traditional ICP limitations, such as reliance on manual parameter tuning and vulnerability to outliers. For instance, Google's LOAM (Lidar Odometry and Mapping) algorithm, enhanced with learning-based loop closure, demonstrates superior performance in unstructured terrains.

2 Methodology and Theory for ICP based SLAM

The Simultaneous Localization and Mapping (SLAM) problem aims to estimate:

- Robot trajectory: $\mathcal{X} = \{\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_T\}, \mathbf{x}_t \in SE(3)$
- Environment map: $\mathcal{M} = \{\mathbf{m}_1, ..., \mathbf{m}_N\} \subset \mathbb{R}^3$

where $SE(3)$ is the special Euclidean group representing 3D poses:

$$\mathbf{x}_t = \begin{bmatrix} \mathbf{R}_t & \mathbf{t}_t \\ \mathbf{0}^\top & 1 \end{bmatrix}, \ \mathbf{R}_t \in SO(3), \ \mathbf{t}_t \in \mathbb{R}^3 \tag{1}$$

2.1 Introduction

The Iterative Closest Point (ICP) algorithm is a pivotal method for aligning two 3D point clouds through iterative refinement of a rigid transformation (rotation and translation). This document formalizes its mathematical foundation, variants, and applications.

2.2 Problem Formulation

Given a source point cloud $P = \{\mathbf{p}_i\}_{i=1}^{N}$ and a target point cloud $Q = \{\mathbf{q}_j\}_{j=1}^{M}$, ICP computes a rigid transformation $T = (R, \mathbf{t})$, where $R \in \mathbb{R}^{3\times3}$ is a rotation matrix ($R^T R = I$, $\det(R) = 1$) and $\mathbf{t} \in \mathbb{R}^3$ is a translation vector. The objective is to minimize:

$$E(R, \mathbf{t}) = \sum_{i=1}^{N} \min_{j} \|R\mathbf{p}_i + \mathbf{t} - \mathbf{q}_j\|^2. \tag{2}$$

2.3 Correspondence Estimation

At each iteration, correspondences are established via nearest neighbor search:

$$\mathbf{q}_i^{(k)} = \arg\min_{\mathbf{q}\in Q} \|R^{(k)}\mathbf{p}_i + \mathbf{t}^{(k)} - \mathbf{q}\|, \tag{3}$$

where $(R^{(k)}, \mathbf{t}^{(k)})$ is the current transformation estimate. Accelerated using KD-trees, this step has complexity $O(N \log M)$.

2.4 Transformation Estimation

Given correspondences $\{(\mathbf{p}_i, \mathbf{q}_i^{(k)})\}$, the optimal R and $\mathbf{t}$ minimize:

$$E(R, \mathbf{t}) = \sum_{i=1}^{N} \|R\mathbf{p}_i + \mathbf{t} - \mathbf{q}_i^{(k)}\|^2. \tag{4}$$

2.5 Centroid Calculation

Compute centroids of matched points:

$$\boldsymbol{\mu}_p = \frac{1}{N}\sum_{i=1}^{N} \mathbf{p}_i, \quad \boldsymbol{\mu}_q = \frac{1}{N}\sum_{i=1}^{N} \mathbf{q}_i^{(k)}. \tag{5}$$

2.6 Cross-Covariance Matrix

Center the points:

$$\mathbf{p}_i' = \mathbf{p}_i - \boldsymbol{\mu}_p, \quad \mathbf{q}_i' = \mathbf{q}_i^{(k)} - \boldsymbol{\mu}_q. \tag{6}$$

Compute the cross-covariance matrix:

$$H = \sum_{i=1}^{N} \mathbf{p}_i' \mathbf{q}_i'^T \in \mathbb{R}^{3\times3}. \tag{7}$$

2.7 Singular Value Decomposition (SVD)

Decompose H via SVD:

$$H = U\Sigma V^T, \tag{8}$$

where U and V are orthogonal matrices. The optimal rotation is:

$$R = VU^T. \tag{9}$$

To ensure $\det(R) = 1$, negate the last column of V if $\det(VU^T) = -1$.

2.8 Translation Vector

Compute the translation:

$$\mathbf{t} = \boldsymbol{\mu}_q - R\boldsymbol{\mu}_p. \tag{10}$$

2.9 Iterative Process

The algorithm iterates between correspondence and transformation steps:

Initialize $R^{(0)} = I$, $\mathbf{t}^{(0)} = \mathbf{0}$, $k = 0$;
while $\Delta E > \epsilon$ *or* $k < k_{max}$ **do**
    Correspondence Step: Update $\mathbf{q}_i^{(k)}$;
    Transformation Step: Compute $R^{(k+1)}$, $\mathbf{t}^{(k+1)}$;
    Update error $E^{(k+1)}$;
    $\Delta E = |E^{(k+1)} - E^{(k)}|$;
    $k = k + 1$;
end

Algorithm 1: ICP Algorithm

2.10 Convergence and Limitations

ICP converges to a local minimum. The stopping criteria include:

- Error threshold $\Delta E < \epsilon$.
- Maximum iterations k_{max}.

2.11 Limitations

- **Local Minima**: Requires a good initial guess.
- **Outliers**: Incorrect correspondences distort results.
- **Computational Cost**: $O(N \log M)$ per iteration.

2.12 Variants of ICP

2.13 Point-to-Plane ICP

Minimizes distance to local planes in Q:

$$E(R, \mathbf{t}) = \sum_{i=1}^{N} \left(\mathbf{n}_i^T (R\mathbf{p}_i + \mathbf{t} - \mathbf{q}_i)\right)^2, \tag{11}$$

where $\mathbf{n}_i$ is the normal vector at $\mathbf{q}_i$.

2.14 Robust ICP

Uses robust loss functions (e.g., Huber) to reduce outlier influence:

$$E(R, \mathbf{t}) = \sum_{i=1}^{N} \rho\left(\|R\mathbf{p}_i + \mathbf{t} - \mathbf{q}_i\|\right), \tag{12}$$

where ρ is a robust kernel.

2.15 Point-to-Plane Metric

Using surface normals $\mathbf{n}_i$ for better convergence:

$$E(\mathbf{R}, \mathbf{t}) = \sum_{i=1}^{N} w_i [(\mathbf{R}\mathbf{p}_i + \mathbf{t} - \mathbf{q}_i) \cdot \mathbf{n}_i]^2 \tag{13}$$

Linearizing for small angles $\boldsymbol{\omega} \in \mathfrak{so}(3)$:

$$\begin{bmatrix} \mathbf{p}_i \times \mathbf{n}_i \\ \mathbf{n}_i \end{bmatrix}^\top \begin{bmatrix} \boldsymbol{\omega} \\ \mathbf{t} \end{bmatrix} = -(\mathbf{p}_i - \mathbf{q}_i) \cdot \mathbf{n}_i \tag{14}$$

3 ICP-SLAM Mathematical Framework

3.1 SLAM as Nonlinear Least Squares

Joint optimization of map and trajectory:

$$\mathcal{X}^*, \mathcal{M}^* = \arg\min_{\mathcal{X}, \mathcal{M}} \underbrace{\sum_t \|\mathbf{f}(\mathbf{x}_t, \mathbf{u}_t)\|^2_{\Sigma_u}}_{\text{Odometry factors}} + \underbrace{\sum_{t,i} \|\mathbf{h}(\mathbf{x}_t, \mathbf{m}_i) - \mathbf{z}_t^i\|^2_{\Sigma_z}}_{\text{ICP factors}} \tag{15}$$

3.2 Error Jacobians

For ICP factor linearization:

$$\mathbf{J}_t = \frac{\partial \mathbf{e}_{ICP}}{\partial \delta \boldsymbol{\xi}} = \begin{bmatrix} \vdots & \\ -[\mathbf{R}\mathbf{p}_i]_\times & \mathbf{I}_{3\times 3} \\ \vdots & \end{bmatrix} \tag{16}$$

where $\delta\boldsymbol{\xi} \in \mathfrak{se}(3)$ is the Lie algebra perturbation.

3.3 Covariance Estimation

The Fisher information matrix:

$$\boldsymbol{\Sigma}^{-1} = \mathbf{J}^\top \mathbf{W} \mathbf{J}, \quad \mathbf{W} = \text{diag}(w_i) \tag{17}$$

4 Probabilistic Formulation

4.1 Measurement Model

For LiDAR measurement $\mathbf{z}_t^k$:

$$p(\mathbf{z}_t^k | \mathbf{x}_t, \mathcal{M}) = \mathcal{N}(\mathbf{z}_t^k | \mathbf{T}(\mathbf{x}_t)\mathbf{m}_{\pi(k)}, \boldsymbol{\Sigma}_z) \tag{18}$$

with $\boldsymbol{\Sigma}_z = \sigma_z^2 \mathbf{I} + \alpha \mathbf{n}\mathbf{n}^\top$ (anisotropic noise).

4.2 Motion Model

Odometry propagation with Lie algebra:

$$\mathbf{x}_t = \mathbf{x}_{t-1} \oplus \exp(\mathbf{u}_t + \boldsymbol{\epsilon}_u), \ \boldsymbol{\epsilon}_u \sim \mathcal{N}(0, \boldsymbol{\Sigma}_u) \tag{19}$$

where $\oplus$ is the SE(3) composition operator.

5 Implementation Details

5.1 Correspondence Search

Using KD-Trees with complexity $O(N \log N)$:

$$\pi(i) = \arg\min_{j \in \mathcal{M}} \|\mathbf{T}(\mathbf{x}_t)\mathbf{p}_i - \mathbf{m}_j\|_2 \tag{20}$$

5.2 Outlier Rejection

Adaptive thresholding with robust statistics:

$$w_i = \begin{cases} 1 & r_i < k\sigma \\ \frac{k\sigma}{r_i} & \text{otherwise} \end{cases}, \; r_i = \|\mathbf{e}_i\| \tag{21}$$

5.3 Pose Graph Optimization

Gauss-Newton iteration for SLAM:

$$\delta\boldsymbol{\xi}^* = \arg\min_{\delta\boldsymbol{\xi}} \sum_{i,j} \|\mathbf{e}_{ij} + \mathbf{J}_{ij}\delta\boldsymbol{\xi}\|^2_{\boldsymbol{\Sigma}_{ij}} \tag{22}$$

6 Challenges & Solutions

- **Local Minima**: Use robust kernels and initialization strategies
- **Computational Complexity**: Implement voxel grid downsampling
- **Dynamic Environments**: Employ moving object detection
- **Scale Drift**: Incorporate IMU or wheel odometry. Also, the GNSS can be incorporated to mitigate the overall drift over time.

7 Conclusion

ICP-based LiDAR SLAM represents a critical intersection of geometric algorithms and sensor technology, enabling robots to achieve autonomy in diverse environments. While challenges persist—particularly in scalability and dynamic settings—the integration of AI promises to unlock new capabilities, from adaptive mapping to lifelong learning. For undergraduate students, studying ICP-based SLAM offers a gateway to understanding the synergy between classical robotics and cutting-edge AI, preparing them to contribute to the next generation of autonomous systems. As LiDAR sensors become smaller, cheaper, and more powerful, and as AI continues to evolve, the future of robotic navigation is poised for unprecedented innovation [1].

Conflict of interests

The authors should declare here any potential conflicts of interests. This tutorial is derived based on the PCL library and also the ROS community.

Acknowledgments (optional)

The authors would like to thank Dr. Xiwei Bai and other colleagues, students for their helpful contributions and feedback.

Funding (optional)

This work was supported by the PolyU AAE4011 course.

Availability of data and software code (optional)

Our software code is available at the following URL: https://github.com/weisongwen/ToySLAM.
Our dataset is available at the following URL: https://github.com/weisongwen/UrbanLoco.

References

[1] Charles D Hansen, Min Chen, Christopher R Johnson, Arie E Kaufman, and Hans Hagen. *Scientific Visualization*. Springer, 2014.

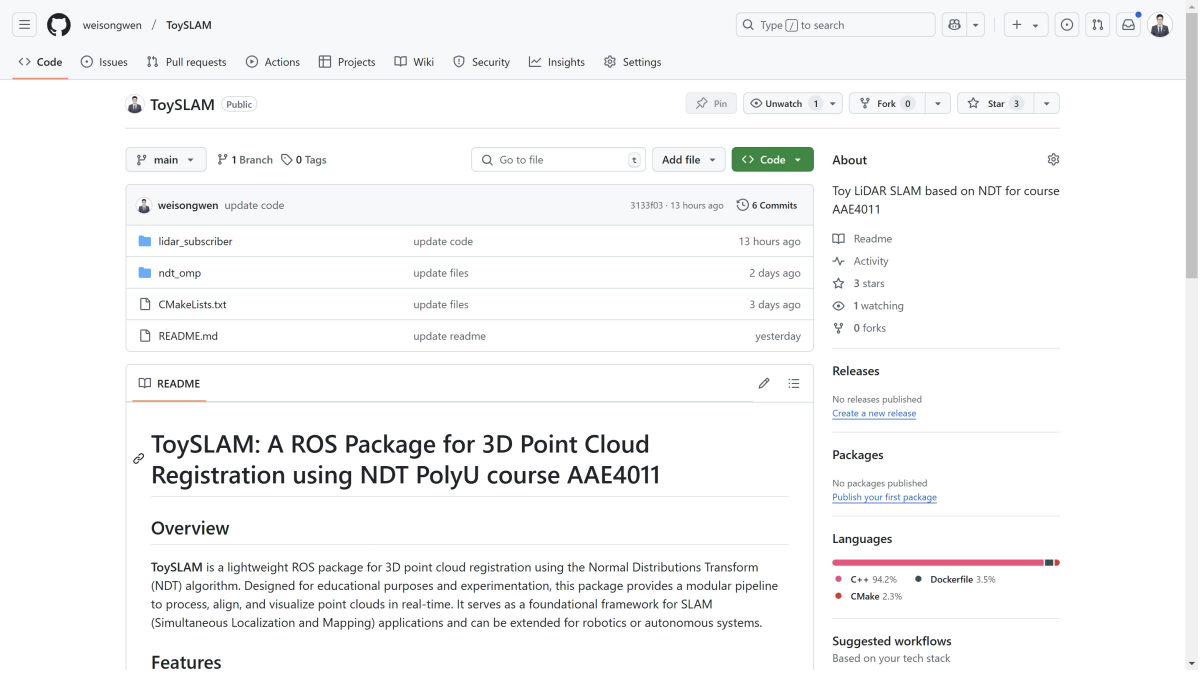

Figure 1: Overview of the ToySLAM package. The package is specifically designed for this course, focusing on designing the ICP implementation and LiDAR SLAM

8 Appendix: Application and practice for the ICP implementation with Python or C++

A collection of the Python/C++ code for the ICP and ICP based LiDAR SLAM can be found by link:

8.1 *Representation of the position and orientation. The implementation and example in Python*

```
import numpy as np

# Position in 3D space
position = np.array([1.0, 2.0, 3.0])  # x, y, z coordinates
print("Position:", position)

from scipy.spatial.transform import Rotation as R

# Euler angles (in radians)
euler_angles = np.array([np.pi / 4, np.pi / 6, np.pi / 3])  # Roll, Pitch
rotation = R.from_euler('xyz', euler_angles)
print("Euler Angles (radians):", euler_angles)
print("Rotation Matrix from Euler Angles:\n", rotation.as_matrix())

# Define a rotation matrix directly
rotation_matrix = np.array([
    [0.866, -0.5, 0.0],
    [0.5, 0.866, 0.0],
    [0.0, 0.0, 1.0]
])
```

```
print("Rotation Matrix:\n", rotation_matrix)

# Quaternion representation
quaternion = rotation.as_quat()  # Convert from Euler angles
print("Quaternion:", quaternion)

# Homogeneous transformation matrix
transformation_matrix = np.eye(4)
transformation_matrix[:3, :3] = rotation_matrix  # Rotation part
transformation_matrix[:3, 3] = position  # Translation part
print("Homogeneous Transformation Matrix:\n", transformation_matrix)
```

8.2 ***Apply a homogeneous transformation to a geometry. The implementation and example in Python***

```
import open3d as o3d
import numpy as np

def create_coordinate_frame(size=0.1, origin=[0, 0, 0]):
    """Create a coordinate frame for visualization."""
    return o3d.geometry.TriangleMesh.create_coordinate_frame(size=size, o

def apply_transformation(geometry, transformation):
    """Apply a homogeneous transformation to a geometry."""
    return geometry.transform(transformation)

def visualize_poses(poses):
    """Visualize multiple poses with coordinate frames."""
    geometries = []
    for pose in poses:
        # Create a coordinate frame for each pose
        frame = create_coordinate_frame()
        # Apply the transformation to the coordinate frame
        transformed_frame = apply_transformation(frame, pose)
        geometries.append(transformed_frame)

    # Visualize all coordinate frames
    o3d.visualization.draw_geometries(geometries, window_name="Homogeneou

# Define four different homogeneous transformation matrices
poses = []

# Pose 1: Identity (no transformation)
pose1 = np.eye(4)
poses.append(pose1)

# Pose 2: Translation along x-axis
pose2 = np.eye(4)
pose2[:3, 3] = [0.5, 0, 0]
poses.append(pose2)
```

```
# Pose 3: Rotation around z-axis
angle = np.pi / 4  # 45 degrees
pose3 = np.eye(4)
pose3[:3, :3] = [
    [np.cos(angle), -np.sin(angle), 0],
    [np.sin(angle), np.cos(angle), 0],
    [0, 0, 1]
]
poses.append(pose3)

# Pose 4: Translation and rotation
pose4 = np.eye(4)
pose4[:3, :3] = [
    [np.cos(angle), -np.sin(angle), 0],
    [np.sin(angle), np.cos(angle), 0],
    [0, 0, 1]
]
pose4[:3, 3] = [0.5, 0.5, 0]
poses.append(pose4)

# Visualize the poses
visualize_poses(poses)
```

8.3 *ICP implementation and example in Python*

```
import open3d as o3d
import numpy as np

def draw_registration_result(source, target, transformation, iteration):
    # Apply the transformation to the source point cloud
    source_temp = source.transform(transformation)

    # Set colors for visualization
    source_temp.paint_uniform_color([1, 0, 0])  # Red for source
    target.paint_uniform_color([0, 1, 0])       # Green for target

    # Visualize the point clouds
    o3d.visualization.draw_geometries([source_temp, target], window_name=

# Create a target point cloud
target = o3d.geometry.PointCloud()
target.points = o3d.utility.Vector3dVector(np.random.rand(10000, 3))

# Create a source point cloud by applying a known transformation to the t
source = o3d.geometry.PointCloud()
source.points = o3d.utility.Vector3dVector(np.random.rand(10000, 3))

# Apply a known transformation to simulate an initial misalignment
R = source.get_rotation_matrix_from_xyz((0.1, 0.1, 0.1))  # Rotation
```

```
t = np.array([0.5, -0.2, 0.3])  # Translation
transformation = np.eye(4)
transformation[:3, :3] = R
transformation[:3, 3] = t
source.transform(transformation)

# Initial transformation
trans_init = np.eye(4)

# Threshold for ICP
threshold = 0.1

# Number of iterations
max_iterations = 10

# Perform ICP registration manually to visualize each iteration
current_transformation = trans_init
for i in range(max_iterations):
    reg_p2p = o3d.pipelines.registration.registration_icp(
        source, target, threshold, current_transformation,
        o3d.pipelines.registration.TransformationEstimationPointToPoint()
        o3d.pipelines.registration.ICPConvergenceCriteria(max_iteration=1
    )

    # Update the transformation
    current_transformation = reg_p2p.transformation @ current_transformat

    print("current transformation matrix:")
    print(current_transformation)

    # Visualize the result of the current iteration
    draw_registration_result(source, target, current_transformation, i +

# Final transformation
print("Final transformation matrix:")
print(current_transformation)
```

8.4 ***Iterative Closest Point (ICP) SLAM example author: Atsushi Sakai (@Atsushi$_{t}wi), Göktuğ Karakaş***

```
"""
Iterative Closest Point (ICP) SLAM example
author: Atsushi Sakai (@Atsushi_twi), Göktuğ Karakaşlı, Shamil Gemuev
"""

import math

from mpl_toolkits.mplot3d import Axes3D  # noqa: F401 unused import
import matplotlib.pyplot as plt
import numpy as np
```

```
#  ICP parameters
EPS = 0.0001
MAX_ITER = 100

show_animation = True


def icp_matching(previous_points, current_points):
    """
    Iterative Closest Point matching
    - input
    previous_points: 2D or 3D points in the previous frame
    current_points: 2D or 3D points in the current frame
    - output
    R: Rotation matrix
    T: Translation vector
    """
    H = None  # homogeneous transformation matrix

    dError = np.inf
    preError = np.inf
    count = 0

    if show_animation:
        fig = plt.figure()
        if previous_points.shape[0] == 3:
           fig.add_subplot(111, projection='3d')

    while dError >= EPS:
        count += 1

        if show_animation:  # pragma: no cover
            plot_points(previous_points, current_points, fig)
            plt.pause(0.1)

        indexes, error = nearest_neighbor_association(previous_points, cu
        Rt, Tt = svd_motion_estimation(previous_points[:, indexes], curre
        # update current points
        current_points = (Rt @ current_points) + Tt[:, np.newaxis]

        dError = preError - error
        print("Residual:", error)

        if dError < 0:  # prevent matrix H changing, exit loop
            print("Not Converge...", preError, dError, count)
            break

        preError = error
```

```
        H = update_homogeneous_matrix(H, Rt, Tt)

        if dError <= EPS:
            print("Converge", error, dError, count)
            break
        elif MAX_ITER <= count:
            print("Not Converge...", error, dError, count)
            break

    R = np.array(H[0:-1, 0:-1])
    T = np.array(H[0:-1, -1])

    return R, T


def update_homogeneous_matrix(Hin, R, T):

    r_size = R.shape[0]
    H = np.zeros((r_size + 1, r_size + 1))

    H[0:r_size, 0:r_size] = R
    H[0:r_size, r_size] = T
    H[r_size, r_size] = 1.0

    if Hin is None:
        return H
    else:
        return Hin @ H


def nearest_neighbor_association(previous_points, current_points):

    # calc the sum of residual errors
    delta_points = previous_points - current_points
    d = np.linalg.norm(delta_points, axis=0)
    error = sum(d)

    # calc index with nearest neighbor assosiation
    d = np.linalg.norm(np.repeat(current_points, previous_points.shape[1]
                        - np.tile(previous_points, (1, current_points.shap
    indexes = np.argmin(d.reshape(current_points.shape[1], previous_point

    return indexes, error


def svd_motion_estimation(previous_points, current_points):
    pm = np.mean(previous_points, axis=1)
    cm = np.mean(current_points, axis=1)
```

```
    p_shift = previous_points - pm[:, np.newaxis]
    c_shift = current_points - cm[:, np.newaxis]

    W = c_shift @ p_shift.T
    u, s, vh = np.linalg.svd(W)

    R = (u @ vh).T
    t = pm - (R @ cm)

    return R, t


def plot_points(previous_points, current_points, figure):
    # for stopping simulation with the esc key.
    plt.gcf().canvas.mpl_connect(
        'key_release_event',
        lambda event: [exit(0) if event.key == 'escape' else None])
    if previous_points.shape[0] == 3:
        plt.clf()
        axes = figure.add_subplot(111, projection='3d')
        axes.scatter(previous_points[0, :], previous_points[1, :],
                     previous_points[2, :], c="r", marker=".")
        axes.scatter(current_points[0, :], current_points[1, :],
                     current_points[2, :], c="b", marker=".")
        axes.scatter(0.0, 0.0, 0.0, c="r", marker="x")
        figure.canvas.draw()
    else:
        plt.cla()
        plt.plot(previous_points[0, :], previous_points[1, :], ".r")
        plt.plot(current_points[0, :], current_points[1, :], ".b")
        plt.plot(0.0, 0.0, "xr")
        plt.axis("equal")


def main():
    print(__file__ + " start!!")

    # simulation parameters
    nPoint = 1000
    fieldLength = 50.0
    motion = [0.5, 2.0, np.deg2rad(-10.0)]  # movement [x[m],y[m],yaw[deg

    nsim = 3  # number of simulation

    for _ in range(nsim):

        # previous points
```

```
        px = (np.random.rand(nPoint) - 0.5) * fieldLength
        py = (np.random.rand(nPoint) - 0.5) * fieldLength
        previous_points = np.vstack((px, py))

        # current points
        cx = [math.cos(motion[2]) * x - math.sin(motion[2]) * y + motion[
              for (x, y) in zip(px, py)]
        cy = [math.sin(motion[2]) * x + math.cos(motion[2]) * y + motion[
              for (x, y) in zip(px, py)]
        current_points = np.vstack((cx, cy))

        R, T = icp_matching(previous_points, current_points)
        print("R:", R)
        print("T:", T)


def main_3d_points():
    print(__file__ + " start!!")

    # simulation parameters for 3d point set
    nPoint = 1000
    fieldLength = 50.0
    motion = [0.5, 2.0, -5, np.deg2rad(-10.0)]  # [x[m],y[m],z[m],roll[de

    nsim = 3  # number of simulation

    for _ in range(nsim):

        # previous points
        px = (np.random.rand(nPoint) - 0.5) * fieldLength
        py = (np.random.rand(nPoint) - 0.5) * fieldLength
        pz = (np.random.rand(nPoint) - 0.5) * fieldLength
        previous_points = np.vstack((px, py, pz))

        # current points
        cx = [math.cos(motion[3]) * x - math.sin(motion[3]) * z + motion[
              for (x, z) in zip(px, pz)]
        cy = [y + motion[1] for y in py]
        cz = [math.sin(motion[3]) * x + math.cos(motion[3]) * z + motion[
              for (x, z) in zip(px, pz)]
        current_points = np.vstack((cx, cy, cz))

        R, T = icp_matching(previous_points, current_points)
        print("R:", R)
        print("T:", T)


if __name__ == '__main__':
```

```
    main()
    main_3d_points()
```

8.5 *Continuous implementation of the ICP based on randomly generated 3D point clouds*

```
import open3d as o3d
import numpy as np

def generate_random_point_cloud():
    """Generate a random point cloud."""
    pc = o3d.geometry.PointCloud()
    pc.points = o3d.utility.Vector3dVector(np.random.rand(10000, 3))
    return pc

def apply_random_transformation(pc):
    """Apply a random transformation to a point cloud."""
    R = pc.get_rotation_matrix_from_xyz((np.random.rand() * 0.2, np.rando
    t = np.random.rand(3) * 1.5 # 0.2
    transformation = np.eye(4)
    transformation[:3, :3] = R
    transformation[:3, 3] = t
    return pc.transform(transformation)

def draw_map(map_cloud):
    """Visualize the accumulated map."""
    o3d.visualization.draw_geometries([map_cloud], window_name="Accumulat

# Initialize an empty point cloud for the map
map_cloud = o3d.geometry.PointCloud()

# Generate and accumulate 10 frames of point clouds
num_frames = 10
threshold = 0.1
trans_init = np.eye(4)

for i in range(num_frames):
    # Generate a new random point cloud and apply a random transformation
    source = generate_random_point_cloud()
    source = apply_random_transformation(source)

    # Color the source point cloud differently for each frame
    color = np.random.rand(3)
    source.paint_uniform_color(color)

    if i == 0:
        # Initialize the map with the first point cloud
        map_cloud += source
    else:
        # Perform ICP to align the source to the map
```

```
        reg_p2p = o3d.pipelines.registration.registration_icp(
            source, map_cloud, threshold, trans_init,
            o3d.pipelines.registration.TransformationEstimationPointToPoi
        )

        # Transform the source point cloud using the ICP result
        source.transform(reg_p2p.transformation)

        # Accumulate the transformed source into the map
        map_cloud += source

# Visualize the accumulated ma

draw_map(map_cloud)
```