LiDAR Odometry for Beginner

Seungwon Choi
csw3575@snu.ac.kr

Tae-Wan Kim
taewan@snu.ac.kr

Preface

As I release this small document into the world, I humbly extend a greeting to those taking their first steps into the world of LiDAR Odometry.

I wrote this guide remembering the vast gap between complex theory and working code. This document is for those who have studied the theory for LiDAR Odometry but find it difficult when trying to implement it. Countless papers tell us 'what' needs to be done, but a kind guide on 'how' to do it always felt lacking.

This guide's primary goal is not to delve into deep mathematical proofs but to answer the practical question, "So, how do I implement this?" To lower the barrier to entry, all examples use the approachable `Python` language, helping you focus on the core principles. Of course, `Python` has speed limitations compared to C++, so the code in this tutorial should be seen as a 'stepping stone' to bridge theory and implementation.

To get the most out of this guide, I strongly encourage you to review the provided code alongside each chapter.

I truly hope that this small guide can become a lantern on your learning journey, lowering the barrier to entry into the fascinating world of LiDAR Odometry, even if just a little.

All the `Python` code covered in this tutorial can be found in the GitHub repository below:

- **Tutorial Python Code:** https://github.com/93won/lidar_odometry_for_beginner

Furthermore, after you have mastered the principles with `Python`, I recommend you continue your learning by analyzing the C++ code below, which was implemented with performance and scalability in mind.

- **Advanced C++ Implementation:** https://github.com/93won/lidar_odometry

Chapter 1: How Can a Robot Navigate Without GPS?

Autonomous cars, robotic vacuums, delivery robots. What do they have in common? They all perceive their location and understand their surroundings to move. But what if they are indoors or in an underground parking lot where GPS is unavailable? How can a robot track its movement and not get lost?

In this chapter, we will paint the big picture of how **LiDAR Odometry** is the core of the answer to this fundamental question and why we need to learn it from the ground up.

The Problem: "Where Am I?"

The broader technology for a mobile robot to create a map and track its position within it is called **SLAM (Simultaneous Localization and Mapping)**. A SLAM system is typically divided into two main parts:

- **Front-end (Odometry)**: This is the part that continuously processes sensor data to estimate the robot's movement from one moment to the next. It's like taking one step at a time by only comparing your immediate surroundings. This process is also known as **Odometry**. However, relying only on the front-end causes small errors to accumulate over time—a process known as **drift**.
- **Back-end (Mapping & Correction)**: This is the part that corrects the accumulated drift from the front-end to create a globally consistent map. It does this by recognizing previously visited places (**Loop Closure**) and optimizing the entire trajectory. Imagine realizing you've returned to a familiar fountain; the back-end uses this information to correct your entire path.

The Front-end, which estimates motion frame-by-frame, is the core of robot navigation and is our focus. In this tutorial, we will concentrate on building a complete and robust **Front-end (LiDAR Odometry)** from the ground up. While we will explain the role of the Back-end and Loop Closure for context, its implementation is beyond the scope of this series.

The Key to the Solution: The LiDAR Sensor

So, what does a robot "see" to calculate its own movement? While robots can use various sensors like cameras or radar, this tutorial will focus on one of the most powerful and common tools for this task: the **LiDAR (Light Detection and Ranging)** sensor. LiDAR emits thousands of laser beams in a 360-degree pattern to precisely measure the distance to surrounding objects.

This collection of measurements creates a 3D dataset called a **Point Cloud**. As you can see in Figure 1, it is literally a **3D snapshot** of the environment represented by a multitude of points. Each point in this cloud holds **3D coordinate information (x, y, z)** measured from the sensor. LiDAR can accurately perceive the shape of the surroundings even in darkness, giving the robot **eyes** to see the world.

Now, the robot is ready to use the information from its **eyes** to calculate its own movement.

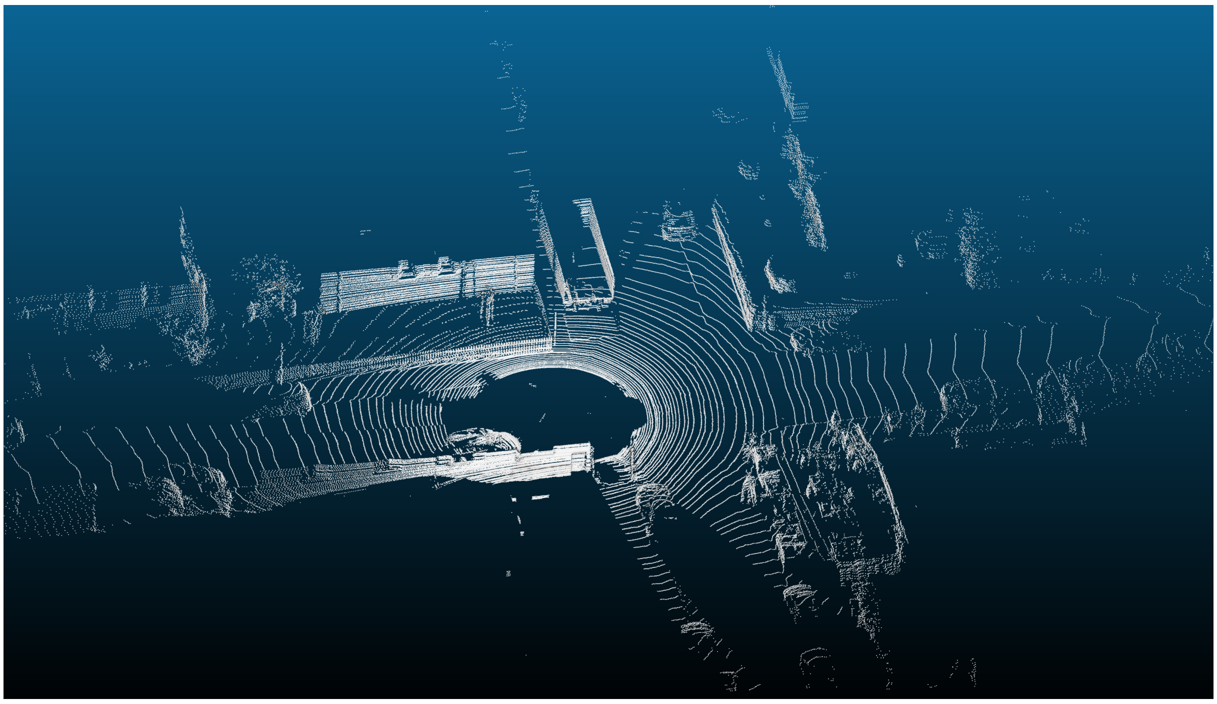

Figure 1: A view of the surrounding environment measured by a LiDAR sensor (Point Cloud)

The Core Idea: Estimating Movement by Overlapping Two Clouds

The core idea of LiDAR Odometry is surprisingly intuitive: it compares the **scenery** from a moment ago with the current **scenery**.

Imagine you have two sheets of semi-transparent tracing paper with the same landscape drawn on them, but one is slightly misaligned. If you rotate and shift the misaligned paper to perfectly overlap with the one underneath, you can figure out exactly **how much you rotated and moved** it. Odometry works in the exact same way.

Figure 2 illustrates this process well. The robot scanned its surroundings twice with a short interval. The **green point cloud** represents the scenery from the previous moment, while the **yellow point cloud** is the current view. Our goal is to determine the exact rotation and translation needed to perfectly align the yellow point cloud with the green one.

The process can be summarized in the following steps:

1. At **'Time 1'**, the robot acquires the **first point cloud (Green)**.
2. The robot moves a small amount.
3. At **'Time 2'**, it acquires the **second point cloud (Yellow)**.
4. We calculate the rotation and translation needed to make the second point cloud perfectly overlap with the first one.
5. This calculated **rotation and translation** is the **estimated amount the robot moved** between 'Time 1' and 'Time 2'.

By repeating this process very quickly, the robot can track its trajectory in real-time. This is the essence of LiDAR Odometry.

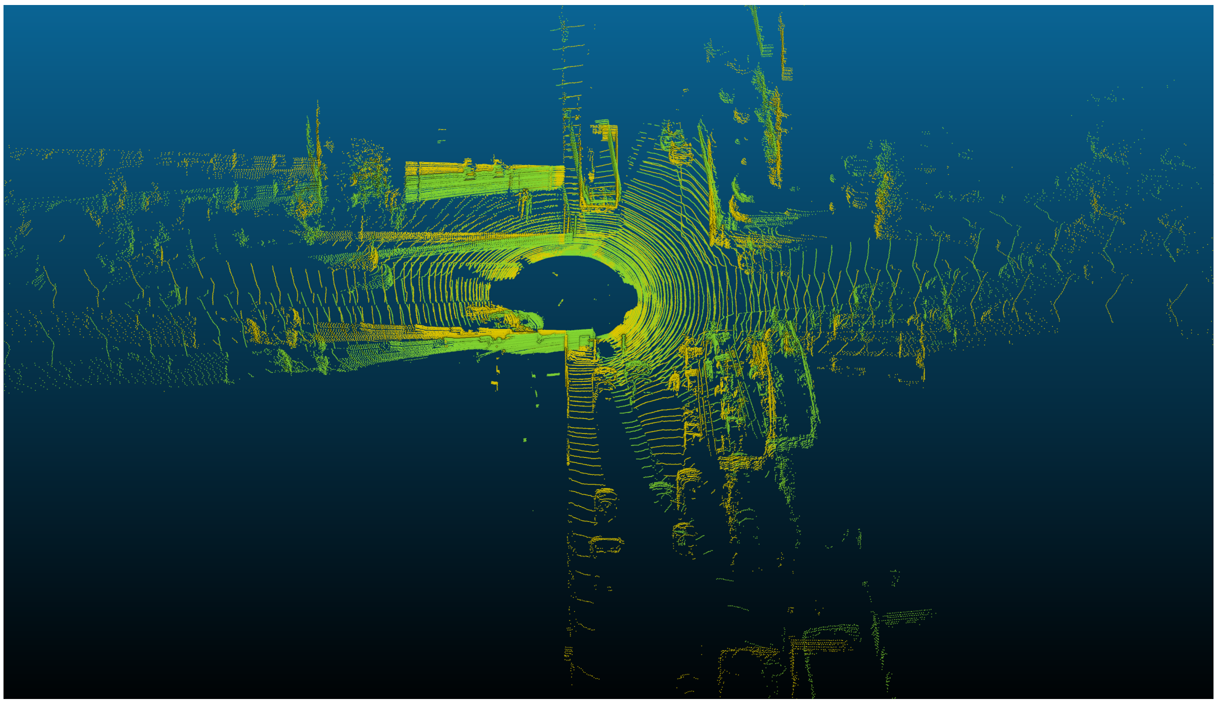

Figure 2: Two Point Clouds measured with a short interval. Green represents the data from the previous time step, and yellow represents the current data.

Practical Tip: Using a Local Map to Improve Odometry

The method we just described, comparing two point clouds (**scan-to-scan**), is the core of odometry. However, in real-world environments, such as a featureless corridor, errors can easily accumulate.

To build a more robust odometry system and minimize these immediate errors, practical implementations often use a concept called a **Local Map**. Instead of comparing the current scan to just the single previous scan, this method compares it to a small, dense local map created by combining the last few scans (**scan-to-map**). This provides richer environmental information, leading to a much more stable and accurate motion estimation.

The Journey Ahead: What Will We Build?

So far, we have looked at the big picture of why LiDAR Odometry is necessary and how it works. Now, here is a brief roadmap of what we will learn to implement this idea in actual code.

- **1. Handling Point Cloud Data**: We will learn how to represent and manipulate LiDAR data on a computer.
- **2. Point Cloud Registration**: We will dive into the principles of the core algorithm for precisely aligning two point clouds, **ICP (Iterative Closest Point)**, and implement it ourselves.
- **3. Transformation**: We will learn how to represent and calculate the robot's rotation and translation using mathematical tools like matrices.

By the end of this tutorial, you will have combined these three key elements to build a complete **LiDAR Odometry** system with your own hands. This system will form the robust front-end of any potential SLAM system.

Chapter 1 in Practice: Visualizing Our Data

Theory is essential, but seeing is believing. Before we can begin to align point clouds to build our odometry system, our very first step is to load this data and visualize it. In this practical exercise, we will use the popular **Open3D** library in Python to bring the point clouds from Figure 2 to life on our screen.

Preparing the Environment

First, you'll need to install the necessary Python library. Open your terminal and run the following command:

```
pip3 install open3d numpy
```

You will also need the sample data files (‘00000.pcd‘ and ‘00010.pcd‘) placed in a ‘data‘ folder.

Understanding the Code

The provided Python script performs a few simple but important tasks: loading the two ‘.pcd‘ files, assigning them different colors (green and yellow) for clarity, merging them into a single object, and launching an interactive 3D viewer.

Execution and Expected Outcome

Run the script from your terminal:

```
python3 visualize_lidar.py
```

A window will appear displaying the two point clouds. You should see the green and yellow clouds slightly misaligned, just as depicted in our earlier figures. This visualization perfectly captures the problem our odometry algorithm needs to solve.

Chapter 2: The Mathematics of Alignment

The Goal: From Intuition to Derivation

In Chapter 1, we developed the intuition for point cloud alignment. Now, we will build the mathematical foundation from the ground up. This chapter will rigorously dissect the ICP algorithm, focusing on the derivation of its core components. We will answer the fundamental questions: How do we mathematically formulate the alignment problem as a non-linear least squares optimization? How do we linearize this problem to find a solution? And how can we make this solution robust to the noise and outliers inherent in real-world data?

The ICP Algorithm: An Intuitive Overview

ICP is a simple and powerful idea: start with a guess, check how well it fits, make a small correction, and repeat. Think of it as carefully turning a key in a lock.

1. **Initial Guess:** Place the key (Source cloud) into the lock (Target cloud). Initially, this is just a guess that they are at the same position.
2. **Find Contact Points (Correspondences):** Wiggle the key slightly and feel where it makes contact with the lock's pins. This is like finding pairs of points that should match.
3. **Calculate Correction (Minimize Error):** Based on the contact points, decide the best small twist and push (ΔT) needed to make the key fit better.
4. **Apply Correction (Update):** Apply this small correction to the key's position.
5. **Repeat:** Continue this process of feeling for contact, calculating a correction, and applying it. With each iteration, the key slides deeper into the lock until it fits perfectly.

This iterative refinement is the heart of ICP. Now let's build the mathematical machinery for each step.

—

Step 1: The Language of Motion - SE(3) and its Tangent Space

To calculate a ‘correction‘, we need a robust way to represent the robot's pose (its rotation and translation).

The Challenge of Direct Optimization

A 3D pose can be described by a 3x3 rotation matrix (R) and a 3x1 translation vector (t). Optimizing these 12 parameters directly is a minefield for three key reasons:

- **The Constraint Problem:** The parameters are not independent. The 9 elements of a rotation matrix R are tightly coupled by the strict constraints of the **Special Orthogonal Group (SO(3))**: orthogonality ($R^T R = I$) and a determinant of $+1$ ($\det(R) = 1$). These constraints define a complex, curved mathematical surface, not a simple flat space.

- **The Update Problem:** Standard optimization involves adding a small update to the current estimate. However, adding an arbitrary matrix to a valid rotation matrix R will almost certainly result in a new matrix that is no longer in SO(3) (i.e., it is not a valid rotation). The updated matrix falls 'off' the SO(3) surface.

- **The Correction Problem:** This means that after every single iteration, a complex and computationally expensive correction step (e.g., projection via SVD) is required to force the updated matrix back onto the SO(3) manifold.

The Solution: Optimizing on a "Flat Map" (Tangent Space)

To avoid these problems, we use a more elegant approach from Lie theory. Imagine trying to navigate on the curved surface of the Earth. For short distances, we can use a simple, flat map. The math is much easier on the flat map than on the curved globe.

This is exactly what we do for pose optimization. The set of all valid poses (SE(3)) forms a curved manifold. Instead of working directly on this 'curved globe', we perform our calculations on a local, flat approximation called the **tangent space**.

The vector space associated with a Lie group is its Lie algebra. For SE(3), this is **'se(3)'**, which serves as the tangent space at the identity element (the 'no transformation' origin).

- **Simple Representation:** In this flat tangent space, any small motion (a 'twist') can be represented by a simple, unconstrained 6D vector $\xi = [\text{translation}, \text{rotation}]^T$.

- **Simple Math:** Because this is a standard vector space, updates are just vector additions. We can find the optimal update step $\Delta\xi$ without worrying about constraints.

- **The Bridge Back:** The **Exponential Map** is the function that elegantly 'wraps' our flat tangent space vector $\Delta\xi$ back onto the curved SE(3) manifold. This guarantees that when we apply our update, the result is always a physically valid pose.

This is why we use SE(3) and 'se(3)': they provide a principled way to use simple, unconstrained vector math in the tangent space to navigate the complex, constrained world of 3D rotations and translations. For a deeper treatment, we recommend the paper by Solà et al.[1]

—

Step 2: Formulating the Optimization Problem

Finding Correspondences and Defining the Error

As before, for each source point $\mathbf{p}_i$, we find its corresponding plane in the target cloud, defined by a point $\mathbf{q}_i$ and a normal vector $\mathbf{n}_i$. Our goal is to find the transformation $\mathbf{T} \in SE(3)$ that minimizes the sum of squared point-to-plane distances.

Let the current estimated transformation be $\mathbf{T}$. A point $\mathbf{p}_i$ is transformed to $\mathbf{T}\mathbf{p}_i = \mathbf{R}\mathbf{p}_i + \mathbf{t}$. The residual error r_i is:

$$r_i(\mathbf{T}) = \mathbf{n}_i^T(\mathbf{R}\mathbf{p}_i + \mathbf{t} - \mathbf{q}_i)$$

This leads to the **Non-Linear Least Squares** objective function, which we want to minimize:

$$E(\mathbf{T}) = \sum_{i=1}^{N} r_i(\mathbf{T})^2$$

Our task is to find a small update $\Delta\xi \in se(3)$ to our current pose $\mathbf{T}$ such that the new pose $\mathbf{T}_{\text{new}} = \mathbf{T} \cdot \exp(\Delta\xi)$ minimizes this error function.

—

Step 3: Solving the Problem - The Gauss-Newton Method

Because the error function $E(\mathbf{T})$ is non-linear with respect to rotation, we cannot solve for the optimal pose in one step. The Gauss-Newton algorithm is an iterative method that solves a sequence of linear approximations to the original problem.

Derivation 1: Linearizing the Error Function

The core idea is to approximate the residual function r_i for a small update $\Delta\xi$ using a first-order Taylor expansion around the current estimate $\mathbf{T}$:

$$r_i(\mathbf{T} \cdot \exp(\Delta\xi)) \approx r_i(\mathbf{T}) + \mathbf{J}_i \Delta\xi$$

Here, $\mathbf{J}_i$ is the **Jacobian** of the residual function with respect to the motion parameters ξ. It is a 1x6 matrix that tells us how the error r_i changes for an infinitesimal change in each of the 6 components of the pose update.

For right multiplication parameterization $\mathbf{T}' = \mathbf{T} \cdot \exp(\Delta\xi)$, the Jacobian is:

$$\mathbf{J}_i = \left.\frac{\partial r_i}{\partial \xi}\right|_{\xi=0} = \begin{bmatrix} \mathbf{n}_i^T \mathbf{R} & -\mathbf{n}_i^T \mathbf{R} [\mathbf{p}_i]_\times \end{bmatrix}$$

where:

- Translation part: $\frac{\partial r_i}{\partial \mathbf{t}} = \mathbf{n}_i^T \mathbf{R}$ (rotation is applied to the translation direction)
- Rotation part: $\frac{\partial r_i}{\partial \boldsymbol{\omega}} = -\mathbf{n}_i^T \mathbf{R} [\mathbf{p}_i]_\times$ (negative sign for right perturbation with skew of original point)

Derivation 2: From Linearization to the Normal Equations

Now we substitute our linearized residual back into the objective function. We want to find the $\Delta\xi$ that minimizes this simplified, quadratic cost function:

$$E(\Delta\xi) \approx \sum_{i=1}^{N} (r_i + \mathbf{J}_i \Delta\xi)^2$$

To find the minimum, we take the derivative of $E(\Delta\xi)$ with respect to $\Delta\xi$ and set it to zero.

$$\begin{aligned}
\frac{\partial E}{\partial \Delta\xi} &= \sum_{i=1}^{N} 2(r_i + \mathbf{J}_i \Delta\xi)^T \frac{\partial}{\partial \Delta\xi}(r_i + \mathbf{J}_i \Delta\xi) \\
&= \sum_{i=1}^{N} 2(r_i + \mathbf{J}_i \Delta\xi)^T \mathbf{J}_i \\
&= \sum_{i=1}^{N} 2(\mathbf{J}_i^T r_i + \mathbf{J}_i^T \mathbf{J}_i \Delta\xi)
\end{aligned}$$

Setting this derivative to zero gives:

$$\sum_{i=1}^{N} (\mathbf{J}_i^T r_i + \mathbf{J}_i^T \mathbf{J}_i \Delta\xi) = 0$$

Rearranging the terms, we arrive at the famous **Normal Equations**:

$$\left(\sum_{i=1}^{N} \mathbf{J}_i^T \mathbf{J}_i \right) \Delta\xi = - \sum_{i=1}^{N} \mathbf{J}_i^T r_i$$

This is a standard linear system of the form $\mathbf{H}\Delta\xi = \mathbf{b}$, where:

- $\mathbf{H} = \sum \mathbf{J}_i^T \mathbf{J}_i$ is the 6x6 Hessian approximation.
- $\mathbf{b} = \sum \mathbf{J}_i^T r_i$ is the 6x1 gradient vector.

By solving this linear system for $\Delta\xi$, we find the optimal step for the current iteration. We then update our pose $\mathbf{T}_{\text{new}} = \mathbf{T} \cdot \exp(\Delta\xi)$ and repeat the process.

A Note on Levenberg-Marquardt (LM)

The Gauss-Newton method works well when the problem is well-conditioned. However, if the Hessian approximation $\mathbf{H}$ is singular or nearly singular (e.g., in a featureless corridor), the solution for $\Delta\xi$ can be unstable and huge.

The **Levenberg-Marquardt (LM)** algorithm improves upon this by adding a damping term:

$$(\mathbf{J}^T \mathbf{J} + \lambda \mathbf{I})\Delta\xi = -\mathbf{J}^T \mathbf{r}$$

The damping parameter λ is adjusted adaptively.

- If λ is small, LM acts like Gauss-Newton, taking confident steps.
- If λ is large, LM acts like Gradient Descent, taking small, safe steps in the steepest direction.

This makes LM more robust than GN and the method of choice for most production-grade SLAM/odometry systems, though we focus on GN in this tutorial for its clear derivation.

—

Step 4: Robustness with Huber Loss and IRLS

The standard least-squares objective $E = \sum r_i^2$ is highly sensitive to outliers. A single bad correspondence can create a massive squared error, corrupting the solution.

The Robust Objective Function

To mitigate this, we replace the squared error with a robust loss function $\rho(\cdot)$, like the Huber loss. The new objective is:

$$E_{\text{robust}}(\mathbf{T}) = \sum_{i=1}^{N} \rho(r_i(\mathbf{T}))$$

The Huber loss function $\rho(r) = L_\delta(r)$ is defined as:

$$L_\delta(r) = \begin{cases} \frac{1}{2}r^2 & \text{for } |r| \leq \delta \\ \delta|r| - \frac{1}{2}\delta^2 & \text{for } |r| > \delta \end{cases}$$

This function penalizes small errors quadratically but large errors only linearly, preventing outliers from dominating.

Iteratively Reweighted Least Squares (IRLS)

How do we minimize this new objective? We can use a clever technique called **Iteratively Reweighted Least Squares (IRLS)**. It allows us to solve the robust problem by iteratively solving a *weighted* version of the standard least-squares problem.

The normal equations are modified to include a weight w_i for each correspondence:

$$\left(\sum_{i=1}^{N} w_i \mathbf{J}_i^T \mathbf{J}_i\right) \Delta\xi = -\sum_{i=1}^{N} w_i \mathbf{J}_i^T r_i$$

The crucial part is that the weight w_i is derived directly from the robust loss function based on the current residual r_i. For Huber loss, this weight is:

$$w_i(r_i) = \begin{cases} 1 & \text{for } |r_i| \leq \delta \\ \frac{\delta}{|r_i|} & \text{for } |r_i| > \delta \end{cases}$$

In each iteration of our Gauss-Newton solver, we first calculate all residuals r_i, then use them to compute the weights w_i. These weights are then used to build and solve the weighted linear system. This effectively tells our solver to pay close attention to the inliers (where $w_i = 1$) and largely ignore the outliers (where $w_i \to 0$).

Chapter 2 in Practice: A Guided Tour of the Code

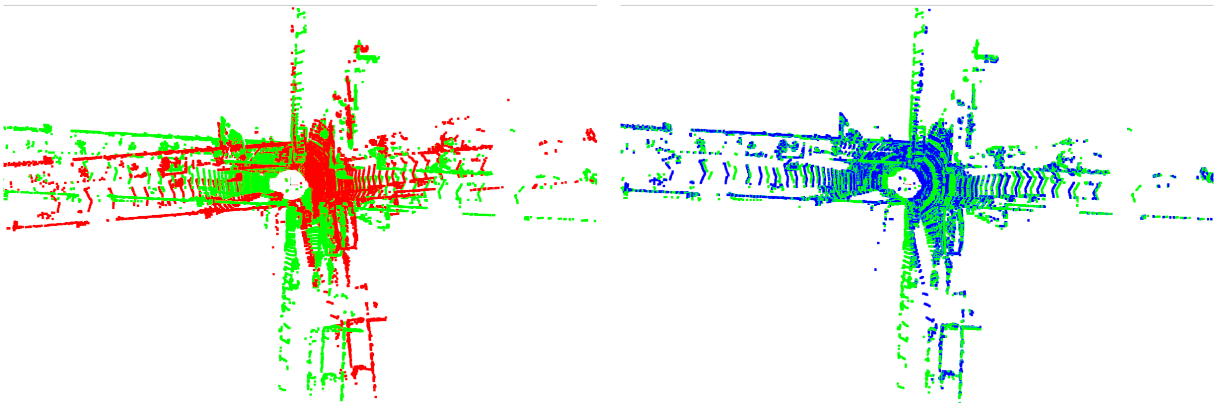
(a) Initial Alignment (Before ICP) (b) Final Alignment (After ICP)

Figure 3: Point cloud alignment results before and after applying the ICP algorithm (practice chapter2). The goal is to align the source cloud (Red in (a), transformed to Blue in (b)) with the target cloud (Green).

Theory provides the 'what' and the 'why', but the code reveals the 'how'. In this section, we will act as code detectives, taking a guided tour through our key Python files. We will dissect the implementation of each major theoretical concept—from finding a plane in a cloud of points to building and solving the robust optimization problem—to see the elegant mathematics come to life.

Code Deep Dive 1: Finding a Plane ('point_cloud_utils.py')

Before we can optimize, we must first establish correspondences and define the error. The cornerstone of the Point-to-Plane metric is the ability to fit a plane to a small cluster of points. This crucial task is handled by the 'fit_plane_to_points' function.

```
# Inside point_cloud_utils.py
import numpy as np

def fit_plane_to_points(points: np.ndarray):
    """
    Fit a plane to a set of points using SVD
    """
    if len(points) < 3:
        return None, None # A plane requires at least 3 points

    # Step 1: Calculate the centroid (mean) of the neighbor points.
    # This becomes the representative point on the plane, q_t.
    centroid = np.mean(points, axis=0)

    # Step 2: Center the points by subtracting the centroid.
    # This prepares the data for PCA/SVD.
    centered_points = points - centroid
    # Step 3: Perform SVD on the centered points matrix.
    # This is a numerically stable way to find the principal components.
    _, _, Vt = np.linalg.svd(centered_points)

    # Step 4: The normal vector, n_t, is the last row of Vt.
    # This vector corresponds to the smallest singular value and
    # represents the direction of minimum variance.
    normal = Vt[-1, :]

    return normal, centroid
```

Let's map this code directly to the theory of Principal Component Analysis (PCA):

- **Line 11: ‘centroid = np.mean(points, axis=0)‘**
  This is the direct implementation of finding the center of mass of the neighbor points. In our theory, this becomes the representative point $\mathbf{q}_t$ that lies on the plane. ‘np.mean‘ with ‘axis=0‘ calculates the mean for each coordinate (x, y, z) across all points.

- **Line 15: ‘centered_points = points - centroid‘**
  This line performs the centering step. By subtracting the mean, we are shifting the entire point cluster so that its new center is at the origin (0,0,0). This is a required preprocessing step for PCA/SVD.

- **Line 18: ‘_, _, Vt = np.linalg.svd(centered_points)‘**
  This is the mathematical core of the function. We call NumPy's Singular Value Decomposition function on our centered data. SVD decomposes the matrix into three other matrices, U, S, V^T. We are interested in ‘Vt‘ (which is V^T in math notation). The rows of this matrix are the principal axes (eigenvectors) of our point distribution, sorted from the direction of most variance to the least.

- **Line 22: ‘normal = Vt[-1, :]‘**
  This final line extracts our desired normal vector. Since the principal axes in ‘Vt‘ are sorted by the magnitude of variance they represent, the last row (‘[-1, :]‘ in Python) corresponds to the principal axis with the *least* variance. As we established in our theory, this is the very definition of the normal vector $\mathbf{n}_t$ for our best-fit plane.

This compact function is incredibly powerful. It is called by ‘find_correspondences_point_to_plane‘ for every source point to determine the local geometry of the target cloud, forming the foundation of our error metric.

Code Deep Dive 2: The Optimization Engine (‘gauss_newton_solver.py‘)

With the ability to find planes and calculate residuals, we can now build the optimization engine. The ‘_build_linear_system‘ function constructs the Normal Equations.

```
# Inside the GaussNewtonSolver class
def _build_linear_system(self, correspondences, pose):
    H = np.zeros((6, 6))  # Hessian approximation, H = J^T * W * J
    b = np.zeros(6)       # Gradient vector, b = J^T * W * r
    for corr in correspondences:
        # ... (code to get source_point, plane_normal, weight, etc.)

        transformed_point = pose * source_point
        residual = np.dot(plane_normal, transformed_point - corr['target_point'])

        # Right multiplication Jacobian: T' = T * exp(xi)
        jac_translation = plane_normal.T @ pose.rotation
        p_skew = skew_symmetric(source_point)
        jac_rotation = -plane_normal.T @ pose.rotation @ p_skew
        jacobian = np.concatenate([jac_translation, jac_rotation])

        H += weight * np.outer(jacobian, jacobian)
        b += weight * jacobian * residual

    return H, b
```

Here is the detailed breakdown:

- **Line 12-15: 'jacobian = ...'**
  This block computes the 1x6 Jacobian matrix $\mathbf{J}_i = \begin{bmatrix}\mathbf{n}_i^T\mathbf{R} & -\mathbf{n}_i^T\mathbf{R}[\mathbf{p}_i]_\times\end{bmatrix}$ as derived in our theory for right multiplication parameterization. 'jac_translation' is $\mathbf{n}_i^T\mathbf{R}$ (derivative with respect to translation), and 'jac_rotation' is $-\mathbf{n}_i^T\mathbf{R}[\mathbf{p}_i]_\times$ (derivative with respect to rotation), created using the skew-symmetric matrix of the original source point.

- **Line 17: 'H += weight * np.outer(jacobian, jacobian)'**
  This line is the implementation of building the Hessian approximation: $\mathbf{H} = \sum w_i \mathbf{J}_i^T \mathbf{J}_i$. For a single 1x6 'jacobian' vector, the 'np.outer' product computes the 6x6 matrix $\mathbf{J}_i^T \mathbf{J}_i$. This is then scaled by the correspondence's robust 'weight' and added to the total.

- **Line 18: 'b += weight * jacobian * residual'**
  This implements the construction of the gradient vector: $\mathbf{b} = \sum w_i \mathbf{J}_i^T r_i$. It scales the jacobian (the direction of change) by the residual (the magnitude of error) and the robust weight, then adds it to the total gradient vector.

Code Deep Dive 3: The Robust Weighting Engine ('RobustGaussNewtonSolver')

Finally, how does each correspondence get its 'weight'? This is handled by the '_update_robust_weights' function, which is the engine of the Iteratively Reweighted Least Squares (IRLS) process.

```
# Inside the RobustGaussNewtonSolver class
def _update_robust_weights(self, correspondences, pose):
    for corr in correspondences:
        if not corr['valid']:
            continue
```

```
# 1. Calculate the current residual for this correspondence
transformed_point = pose * corr['source_point']
residual = np.dot(corr['plane_normal'],
                  transformed_point - corr['target_point'])

# 2. Pass the residual to the loss function to get the weight
_, weight = self.loss_function.compute_loss_and_weight(residual)

# 3. Store the updated weight back in the correspondence dictionary
corr['weight'] = weight
```

This function is called at the **beginning** of each Gauss-Newton iteration. Its job is to "pre-judge" each correspondence before it's used to build the linear system.

1. It computes the latest residual for each point based on the most recent pose estimate.
2. It passes this raw residual to our ‘HuberLoss‘ object.
3. The ‘HuberLoss‘ function acts as a gatekeeper: if the residual is small (an inlier), it returns a weight of ‘1.0‘. If the residual is large (an outlier), it returns a weight less than 1.0 that shrinks as the residual grows, according to the formula $w_i(r_i) = \delta/|r_i|$.
4. This freshly calculated weight is then stored, ready to be used by ‘_build_linear_system‘.

The critical insight is this: by the time ‘_build_linear_system‘ is called, every correspondence has been assigned a trust score. The inliers are trusted fully, while the outliers have had their influence neutralized. This effectively cleans the data that feeds into the optimization, making the calculated step $\Delta\xi$ far more accurate and stable.

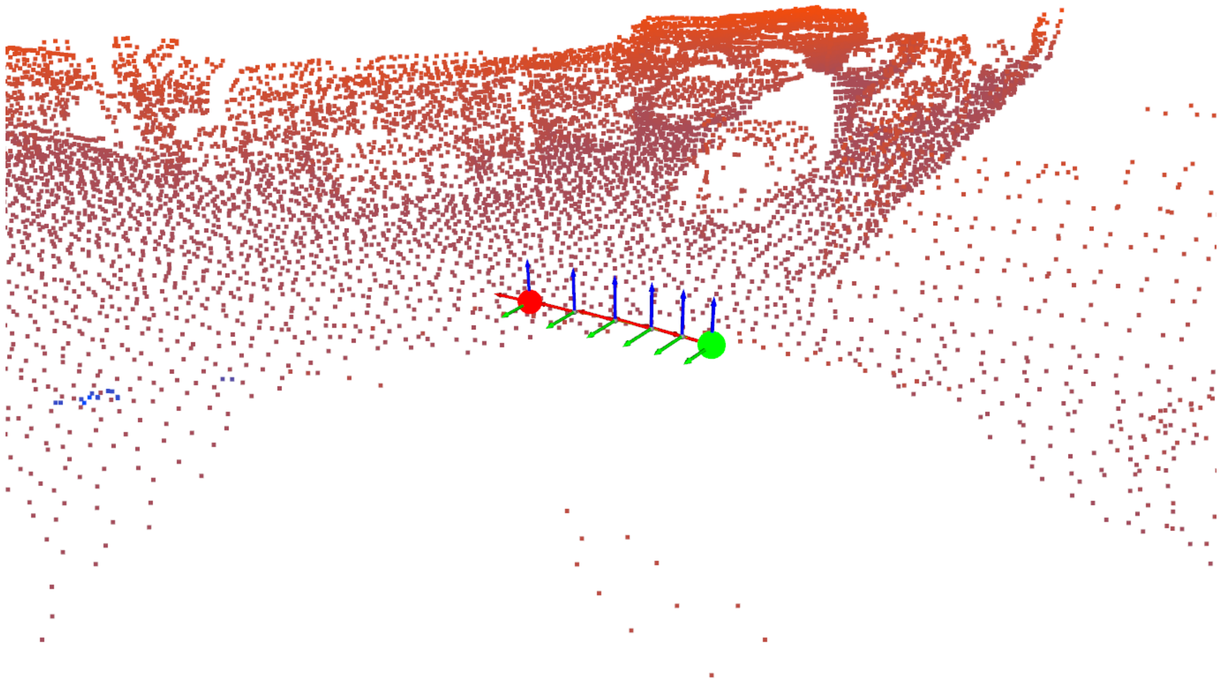

Figure 4: Sequential Local Mapping Results LiDAR odometry visualization showing accumulated feature map, estimated trajectory.

Chapter 3: LiDAR Odometry - ICP and Local Mapping

From Scan-to-Scan to Scan-to-Map

In Chapter 2, we explored how to align two consecutive point clouds. This **Scan-to-Scan** method is a fundamental building block of odometry. However, it has a notable weakness: it is forgetful. By only considering the immediate past frame, it can be susceptible to drift, especially in geometrically ambiguous environments like long, featureless corridors. If one alignment is slightly off, that error is carried forward and amplified over time.

To build a more robust and stable odometry system, we can give our algorithm a better memory. Instead of comparing the current scan to just the single previous scan, we will compare it to a richer, denser, and more stable reference: a **local map**. This upgraded approach is called **Scan-to-Map** odometry.

Imagine navigating a hallway. Scan-to-Scan is like looking only at your feet to take the next step. Scan-to-Map is like looking at the last few meters of the hallway you just walked through. The latter provides a far more stable reference to determine your current position.

The Concept of a Local Map

A local map is a short-term memory of the robot's recent environment. It is a point cloud created by stitching together the last several scans (called **keyframes**) into a single, consistent reference frame (the ‘world‘ frame).

Using a local map as the target for ICP provides two significant advantages:

- **Richer Geometric Information:** The map contains structural details from multiple viewpoints, such as corners, edges, and walls, that might not be fully visible in a single scan. This drastically reduces the ambiguity of the alignment problem.
- **Temporal Averaging:** By accumulating points over time, the map naturally averages out sensor noise, providing a cleaner and more stable registration target.

However, this ‘memory‘ must be carefully managed. If the map grows indefinitely, it will become too large to process in real-time. Therefore, a ‘local‘ map is maintained by two key operations:

1. **Downsampling:** The map's point density is kept in check using a voxel grid filter. This prevents the map from becoming infinitely dense as new scans are added.
2. **Cropping:** To keep the map relevant to the robot's current location, we only keep points within a certain radius of the robot's current pose. Points that are too far behind are discarded.

The Scan-to-Map Odometry Workflow

The introduction of a local map and a motion model changes our odometry pipeline. The new workflow is a continuous cycle of prediction, matching, and updating.

Algorithm 1 Scan-to-Map Odometry Workflow

```
Initialize pose $\mathbf{T}_{k-1}$, velocity $\mathbf{V}_{k-1}$, and empty LocalMap $\mathcal{M}$
for each new Frame $k$ do
    Preprocess current scan $S_k$
                                                    ▷ Step 1: Predict
    Predict current pose: $\mathbf{T}_{\text{pred}} \leftarrow \mathbf{T}_{k-1} \cdot \mathbf{V}_{k-1}$

                                                    ▷ Step 2: Prepare Target
    Get map points in world frame: $\mathcal{M}_{\text{world}} \leftarrow \text{GetMap}(\mathcal{M})$
    Transform map to predicted frame: $\mathcal{M}_{\text{pred}} \leftarrow \mathbf{T}_{\text{pred}}^{-1} \cdot \mathcal{M}_{\text{world}}$

                                                    ▷ Step 3: Match
    Find correction via ICP: $\Delta\mathbf{T} \leftarrow \text{ICP}(S_k, \mathcal{M}_{\text{pred}})$

                                                    ▷ Step 4: Update
    Update final pose: $\mathbf{T}_k \leftarrow \mathbf{T}_{\text{pred}} \cdot \Delta\mathbf{T}$
    Update velocity model: $\mathbf{V}_k \leftarrow \text{smooth}(\mathbf{T}_{k-1}^{-1} \cdot \mathbf{T}_k, \mathbf{V}_{k-1})$

                                                    ▷ Step 5: Extend Map
    Add current scan as keyframe: $\mathcal{M} \leftarrow \text{AddKeyframe}(\mathcal{M}, S_k, \mathbf{T}_k)$
```

—

Chapter 3 in Practice: A Guided Tour of the Code

Let's now transition from the conceptual workflow to the actual Python code. We will see how each step of our Scan-to-Map odometry is implemented in the 'LocalMap' and 'SequentialLocalMapper' classes.

Code Deep Dive 1: 'LocalMap' - The Short-Term Memory

The 'LocalMap' class is responsible for maintaining our short-term environmental memory. Its two most important jobs are adding new information and providing a coherent picture for the current ICP task.

```
# Inside the LocalMap class
def add_keyframe(self, cloud: PointCloud, pose: SE3) -> None:
    # 1. Transform the new scan to the global 'world' frame
    world_cloud = cloud.copy()
    world_cloud.transform(pose.to_matrix())

    # 2. Add the new points to the existing map
    self.feature_map.add_points_from(world_cloud)

    # ... (store keyframe info) ...

    # 3. Downsample the entire map to manage point density
    self.feature_map = self.feature_map.downsample_voxel(self.map_voxel_size)

    # 4. Crop the map to keep it 'local'
    current_pos = pose.translation
    min_bound = current_pos - self.max_range
    max_bound = current_pos + self.max_range
    self.feature_map = self.feature_map.crop_box(min_bound, max_bound)
```

The 'add_keyframe' method is the heart of map management.

- **Line 4-5:** The incoming 'cloud' is in its own local coordinate system. To add it to the map, it must first be transformed into the global 'world' frame using its calculated 'pose'.
- **Line 11:** The 'downsample_voxel' call is crucial for performance. Without it, the map would grow denser with every keyframe, and the ICP step would become progressively slower. This ensures the map maintains a consistent point density.
- **Lines 14-17:** The 'crop_box' call ensures the map remains 'local'. It discards points that are too far away from the robot's current position, keeping the map size manageable and relevant.

```
# Inside the LocalMap class
def get_local_features_in_frame(self, current_pose: SE3) -> PointCloud:
    if self.feature_map.empty():
        return PointCloud()
    # Transform map points from the 'world' frame to the 'current' frame
    local_features = self.feature_map.copy()
    T_lw = current_pose.inverse().to_matrix() # T_local_world = T_world_local^-1
    local_features.transform(T_lw)

    return local_features
```

This method prepares the target for ICP. ICP needs both the source (current scan) and target (map) to be in the same coordinate system. Since the current scan is in its own local frame, the easiest approach is to bring the map into that same frame. This is achieved by applying the **inverse** of the current pose estimate: $\mathcal{M}_{\text{pred}} \leftarrow \mathbf{T}_{\text{pred}}^{-1} \cdot \mathcal{M}_{\text{world}}$.

Code Deep Dive 2: 'SequentialLocalMapper' - The Conductor

This class orchestrates the entire odometry process, implementing the workflow from our pseudocode within its 'process_frame' method.

```
# Inside the SequentialLocalMapper class, process_frame method
# Step 1: Motion Prediction
predicted_pose = self.current_pose * self.velocity

# Step 2: Prepare Target Map
local_map_features = self.local_map.get_local_features_in_frame(predicted_pose)

# Step 3: Match (ICP Alignment)
success, relative_pose, icp_stats = self._run_icp(feature_cloud, local_map_features)

# Step 4: Update Pose
if not success:
    final_pose = predicted_pose
else:
    final_pose = predicted_pose * relative_pose # T_new = T_pred * d_T

# Step 5: Update Velocity Model (with exponential smoothing)
previous_pose = self.trajectory[-1]
current_velocity = previous_pose.inverse() * final_pose
# ... (smoothing logic using alpha) ...
self.velocity = smoothed_velocity

# Step 6: Extend Map
self._add_keyframe(feature_cloud)
```

Let's examine the key steps:

- **Motion Prediction:** We use a constant velocity model as a simple but effective motion prior. The line 'predicted_pose = self.current_pose * self.velocity' gives us a strong initial guess for where the robot currently is, which helps ICP converge faster and more reliably.

- **ICP Alignment:** The call to '_run_icp' is where the magic from Chapter 2 happens. The key difference is the target: we are now matching against the rich 'local_map_features' instead of just a single previous scan. The output, 'relative_pose', is a *small correction* to our initial prediction.
- **Pose Update:** The line 'final_pose = predicted_pose * relative_pose' is the mathematical embodiment of our workflow: $\mathbf{T}_k \leftarrow \mathbf{T}_{\text{pred}} \cdot \Delta\mathbf{T}$. We take our initial guess and refine it with the correction found by ICP.
- **Velocity Update:** We calculate the most recent velocity and then apply exponential smoothing. This acts as a simple low-pass filter, preventing the velocity estimate from being too noisy or jerky, leading to smoother predictions for the subsequent frames.
- **Keyframe Addition:** Finally, the current scan, now correctly positioned at 'final_pose', is added back into the local map. This enriches the map with new information, making it an even better target for the next frame's alignment. This completes the virtuous cycle of our odometry algorithm.

References

[1] Joan Solà, Jérémie Deray, and Dinesh Atchuthan. A micro lie theory for state estimation in robotics. *ArXiv*, abs/1812.01537, 2018.