Tutorial report for AAE4011: NDT-Based LiDAR SLAM-Theory and Methodology

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

Abstract. This paper presents a comprehensive analysis of Normal Distributions Transform (NDT)-based LiDAR SLAM, a robust approach for simultaneous localization and mapping in robotic systems. Unlike traditional Iterative Closest Point (ICP) methods, NDT employs statistical representations of point clouds to achieve superior computational efficiency and robustness in complex environments. The document details the mathematical foundations of NDT, its integration into SLAM pipelines, and emerging synergies with artificial intelligence. Key topics include probabilistic surface modeling, optimization strategies, and comparative advantages over point-to-point registration methods. Applications in autonomous vehicles, industrial robotics, and planetary exploration are discussed, along with challenges and future directions.

1 Introduction

1.1 SLAM in Robotics

Simultaneous Localization and Mapping (SLAM) forms the perceptual backbone of autonomous systems, enabling operation in unknown environments through concurrent map construction and ego-motion estimation. LiDAR sensors have become pivotal for SLAM due to their high angular resolution and immunity to lighting conditions.

1.2 Limitations of ICP

Traditional ICP-based approaches face three fundamental challenges:

- Computational complexity in dense point clouds ($O(n^2)$ for naive implementations)
- Sensitivity to initial pose estimates and local minima
- Performance degradation in unstructured environments

1.3 The NDT Paradigm

The Normal Distributions Transform (Biber & Straßer, 2003) addresses these limitations through probabilistic surface modeling:

$$\mathcal{P}(x) \sim \sum_{c=1}^{N} \frac{1}{\sqrt{(2\pi)^3|\Sigma_c|}} e^{-\frac{1}{2}(x-\mu_c)^T\Sigma_c^{-1}(x-\mu_c)} \tag{1}$$

where μ_c and Σ_c are the mean and covariance of voxel c. This representation enables efficient registration while providing inherent noise tolerance.

2 Methodology and Theory

2.1 NDT Registration Framework

2.1.1 Voxelization Process

The target point cloud $Q = \{q_j\}_{j=1}^{M}$ is discretized into voxels with resolution δ. For each occupied voxel c:

$$\mu_c = \frac{1}{|c|} \sum_{q_j \in c} q_j \tag{2}$$

$$\Sigma_c = \frac{1}{|c| - 1} \sum_{q_j \in c} (q_j - \mu_c)(q_j - \mu_c)^T \tag{3}$$

2.1.2 Transformation Model

The source cloud $P = \{p_i\}_{i=1}^{N}$ is transformed via:

$$T(p_i; \theta) = Rp_i + t, \quad \theta = (t_x, t_y, t_z, \phi, \psi, \omega) \tag{4}$$

where $R \in SO(3)$ parametrized by Euler angles (ϕ, ψ, ω).

2.1.3 Optimization Objective

Maximize the log-likelihood of transformed points under the NDT distribution:

$$\theta^* = \arg\max_{\theta} \sum_{i=1}^{N} \log \mathcal{P}(T(p_i; \theta)) \tag{5}$$

2.2 Newton Optimization

The objective function is maximized using Newton's method with Hessian modification:

$$H = J^T \Lambda J + \lambda I \tag{6}$$

$$\Delta\theta = H^{-1} J^T \Lambda r \tag{7}$$

where J is the Jacobian matrix, Λ the information matrix, and r the residual vector.

Input: Source P, Target NDT $\{\mu_c, \Sigma_c\}$;
Initialize $\theta^{(0)}$, $k = 0$;
while $\|\Delta\theta\| > \epsilon$ **do**
  Evaluate $T(P; \theta^{(k)})$;
  Compute μ_c, Σ_c for transformed points;
  Build J and H using analytic derivatives;
  Solve $H\Delta\theta = -J^T r$;
  $\theta^{(k+1)} = \theta^{(k)} + \Delta\theta$;
  $k = k + 1$;
end

Algorithm 1: NDT Registration

Criterion	**NDT**	**ICP**
Representation	Probabilistic surface	Point-to-point
Convergence Basin	Large	Small
Time Complexity	$O(n+m)$	$O(n \log m)$
Outlier Rejection	Built-in	Requires heuristics
Memory Usage	High (voxel grid)	Low

2.3 NDT vs ICP: Theoretical Comparison

2.4 NDT-SLAM Architecture

2.4.1 Local Mapping

- Voxel size δ_l (typically 1-2m)
- Constant-time insertion via spatial hashing
- Incremental covariance update:

$$\Sigma_c^{(k+1)} = \frac{k\Sigma_c^{(k)} + (q_{new} - \mu_c^{(k)})(q_{new} - \mu_c^{(k)})^T}{k+1} \tag{8}$$

2.4.2 Global Consistency

- Pose graph optimization with NDT constraints:

$$E(\Theta) = \sum_{(i,j)} \|T_{ij} - T_{ij}^{NDT}\|_{\Omega_{ij}}^2 \tag{9}$$

- Multi-resolution NDT for loop closure detection

2.5 AI-Enhanced NDT

2.5.1 Deep NDT

Neural networks predict optimal voxel parameters:

$$(\mu_c, \Sigma_c) = f_\Theta(Q_c) \tag{10}$$

where f_Θ is a 3D CNN processing voxel contents.

2.5.2 Dynamic Environment Handling

Spatio-temporal NDT models:

$$\mathcal{P}(x,t) = \sum_{c=1}^{N} w_c(t)\mathcal{N}(x|\mu_c(t), \Sigma_c(t)) \tag{11}$$

with LSTM-based weight prediction $w_c(t)$.

3 Conclusion

NDT-based LiDAR SLAM represents a significant advancement in robotic perception, combining the efficiency of statistical modeling with the precision of laser scanning. While challenges remain in dynamic environments and computational demands, integration with modern AI techniques promises to unlock new capabilities for autonomous systems operating in complex real-world scenarios.

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

4 Appendix: Application and practice for the NDT implementation with C++

A collection of the C++ code for the NDT based LiDAR SLAM can be found by link:

4.1 *Subscribe the 3D LiDAR point cloud data from a given rosbag file and save the 3D point cloud data to the PCD*

```
#include <ros/ros.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl_conversions/pcl_conversions.h>
#include <string>
#include <boost/filesystem.hpp>

class PointCloudSaver
{
public:
    PointCloudSaver() : nh_("~"), save_count_(0)
    {
        // Initialize parameters
        nh_.param<std::string>("save_directory", save_directory_, "/home/
        nh_.param<std::string>("file_prefix", file_prefix_, "cloud_");
        nh_.param<std::string>("input_topic", input_topic_, "/velodyne_po

        // Create directory if it doesn't exist
        boost::filesystem::path dir(save_directory_);
        if(!boost::filesystem::exists(dir)) {
            boost::filesystem::create_directories(dir);
            ROS_INFO("Created directory: %s", save_directory_.c_str());
        }

        // Initialize subscriber
        sub_ = nh_.subscribe<sensor_msgs::PointCloud2>(
```

```
            input_topic_, 1, &PointCloudSaver::cloudCallback, this);

        ROS_INFO("Point Cloud Saver initialized");
        ROS_INFO("Saving to: %s", save_directory_.c_str());
    }

private:
    void cloudCallback(const sensor_msgs::PointCloud2ConstPtr& msg)
    {
        // Convert ROS message to PCL point cloud
        pcl::PointCloud<pcl::PointXYZI>::Ptr cloud(new pcl::PointCloud<pc
        pcl::fromROSMsg(*msg, *cloud);

        // Generate filename
        std::string filename = save_directory_ + "/" + file_prefix_ +
                               std::to_string(++save_count_) + ".pcd";

        // Save to PCD file
        if(pcl::io::savePCDFileBinary(filename, *cloud) == 0) {
            ROS_INFO_ONCE("Saving point clouds...");
            ROS_DEBUG("Saved %s", filename.c_str());
            ROS_INFO("Saved %s", filename.c_str());
        }
        else {
            ROS_ERROR("Failed to save %s", filename.c_str());
        }
    }

    ros::NodeHandle nh_;
    ros::Subscriber sub_;
    std::string save_directory_;
    std::string file_prefix_;
    std::string input_topic_;
    unsigned int save_count_;
};

int main(int argc, char** argv)
{
    ros::init(argc, argv, "pointcloud_saver_node");
    PointCloudSaver saver;
    ros::spin();
    return 0;
}

// rosrun lidar_subscriber lidar_subscriber_node \ _save_directory:=/home
```

4.2 *Read the 3D point cloud data from the PCD files, and perform the NDT for point cloud registration. Output the transformation*

```
#include <ros/ros.h>
#include <nav_msgs/Path.h>
#include <geometry_msgs/PoseStamped.h>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/filters/voxel_grid.h>
#include <pclomp/ndt_omp.h>
#include <tf2_ros/transform_broadcaster.h>
#include <geometry_msgs/TransformStamped.h>
#include <boost/filesystem.hpp>
#include <algorithm>

class NDT_OMP_Processor {
public:
    NDT_OMP_Processor() : nh_("~"), current_index_(1) {
        initialize_parameters();
        load_pointclouds();
        initialize_ndt();
        initialize_publishers();
    }

    void process_clouds() {
        if (clouds_.size() < 2) {
            ROS_WARN("Need at least 2 clouds for registration");
            return;
        }

        ros::Rate rate(1); // Processing rate (1Hz)
        while (ros::ok() && current_index_ < clouds_.size()) {
            auto& target_cloud = clouds_[current_index_ - 1];
            auto& source_cloud = clouds_[current_index_];

            Eigen::Matrix4f transformation = perform_registration(target_
            publish_transform(transformation);

            ROS_INFO_STREAM("Transform " << current_index_-1 << " to " <<
                            << transformation);

            current_index_++;
            rate.sleep();
        }
    }

private:
    void initialize_parameters() {
        nh_.param<std::string>("pcd_directory", pcd_directory_, "/home/ww
```

```
        nh_.param<double>("resolution", ndt_resolution_, 1.0);
        nh_.param<double>("step_size", step_size_, 0.1);
        nh_.param<double>("epsilon", epsilon_, 0.01);
        nh_.param<int>("max_iterations", max_iterations_, 64);
        nh_.param<int>("num_threads", num_threads_, 4);
        nh_.param<double>("voxel_leaf_size", voxel_leaf_size_, 0.5);

        transformationSum = Eigen::Matrix4f::Identity();
    }

    void initialize_publishers() {
        path_pub_ = nh_.advertise<nav_msgs::Path>("/ndt_trajectory", 10);
        // map_pub_ = nh_.advertise<sensor_msgs::PointCloud2>("/global_ma
        // aligned_cloud_pub_ = nh_.advertise<sensor_msgs::PointCloud2>("
    }

    void load_pointclouds() {
        namespace fs = boost::filesystem;
        std::vector<fs::path> files;

        // Get and sort PCD files
        for (const auto& entry : fs::directory_iterator(pcd_directory_))
            if (entry.path().extension() == ".pcd") {
                files.push_back(entry.path());
            }
        }

        std::sort(files.begin(), files.end(), [](const fs::path& a, const
            return std::stoi(a.stem().string().substr(6)) <
                   std::stoi(b.stem().string().substr(6));
        });

        // Load clouds
        for (const auto& file : files) {
            pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud
            if (pcl::io::loadPCDFile<pcl::PointXYZ>(file.string(), *cloud
                ROS_ERROR("Failed to load %s", file.string().c_str());
                continue;
            }

            // Downsample cloud
            pcl::VoxelGrid<pcl::PointXYZ> voxel_filter;
            voxel_filter.setLeafSize(voxel_leaf_size_, voxel_leaf_size_,
            voxel_filter.setInputCloud(cloud);

            pcl::PointCloud<pcl::PointXYZ>::Ptr filtered_cloud(new pcl::P
            voxel_filter.filter(*filtered_cloud);
```

```
            clouds_.push_back(filtered_cloud);
            ROS_INFO("Loaded and filtered %s (%ld points)",
                    file.filename().c_str(), filtered_cloud->size());
        }
    }

    void initialize_ndt() {
        ndt_omp_.setResolution(ndt_resolution_);
        ndt_omp_.setStepSize(step_size_);
        ndt_omp_.setTransformationEpsilon(epsilon_);
        ndt_omp_.setMaximumIterations(max_iterations_);
        ndt_omp_.setNumThreads(num_threads_);
        ndt_omp_.setNeighborhoodSearchMethod(pclomp::DIRECT7);
    }

    Eigen::Matrix4f perform_registration(
        const pcl::PointCloud<pcl::PointXYZ>::Ptr& target,
        const pcl::PointCloud<pcl::PointXYZ>::Ptr& source)
    {
        ndt_omp_.setInputTarget(target);
        ndt_omp_.setInputSource(source);

        pcl::PointCloud<pcl::PointXYZ>::Ptr aligned(new pcl::PointCloud<p
        ndt_omp_.align(*aligned);

        if (!ndt_omp_.hasConverged()) {
            ROS_WARN("NDT failed to converge");
            return Eigen::Matrix4f::Identity();
        }

        return ndt_omp_.getFinalTransformation();
    }

    void publish_transform(const Eigen::Matrix4f& transformation) {
        static tf2_ros::TransformBroadcaster br;

        geometry_msgs::TransformStamped transform_msg;
        transform_msg.header.stamp = ros::Time::now();
        transform_msg.header.frame_id = "map";
        transform_msg.child_frame_id = "robot";

        Eigen::Affine3f affine(transformation);
        Eigen::Vector3f translation = affine.translation();
        Eigen::Quaternionf rotation(affine.linear());

        transform_msg.transform.translation.x = translation.x();
        transform_msg.transform.translation.y = translation.y();
        transform_msg.transform.translation.z = translation.z();
```

```
        transform_msg.transform.rotation.x = rotation.x();
        transform_msg.transform.rotation.y = rotation.y();
        transform_msg.transform.rotation.z = rotation.z();
        transform_msg.transform.rotation.w = rotation.w();

        br.sendTransform(transform_msg);

        transformationSum *= transformation;
        Eigen::Affine3f affineSum(transformationSum);
        Eigen::Vector3f translationSum = affineSum.translation();
        Eigen::Quaternionf rotationSum(affineSum.linear());

        geometry_msgs::PoseStamped pose;
        pose.header.stamp = ros::Time::now();
        pose.header.frame_id = "map";
        pose.pose.position.x = translationSum.x();
        pose.pose.position.y = translationSum.y();
        pose.pose.position.z = translationSum.z();
        pose.pose.orientation.x = rotationSum.x();
        pose.pose.orientation.y = rotationSum.y();
        pose.pose.orientation.z = rotationSum.z();
        pose.pose.orientation.w = rotationSum.w();
        trajectory_poses_.poses.push_back(pose);

        if (trajectory_poses_.poses.empty()) return;

        trajectory_poses_.header.stamp = ros::Time::now();
        trajectory_poses_.header.frame_id = "map";
        path_pub_.publish(trajectory_poses_);

        ROS_INFO_STREAM("TransformSum " << current_index_-1 << " to " <<
                        << transformationSum);

    }

    ros::NodeHandle nh_;
    ros::Publisher path_pub_;
    ros::Publisher map_pub_;
    ros::Publisher aligned_cloud_pub_;
    nav_msgs::Path trajectory_poses_;
    std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> clouds_;
    pclomp::NormalDistributionsTransform<pcl::PointXYZ, pcl::PointXYZ> nd
    size_t current_index_;

    // Parameters
    std::string pcd_directory_;
    double ndt_resolution_, step_size_, epsilon_, voxel_leaf_size_;
```

```
    int max_iterations_, num_threads_;

    Eigen::Matrix4f transformationSum;
};

int main(int argc, char** argv) {
    ros::init(argc, argv, "ndt_omp_node");
    NDT_OMP_Processor processor;
    processor.process_clouds();
    return 0;
}
```

4.3 *Read the 3D point cloud data from the PCD files, and perform the NDT for point cloud registration. Output the transformation and also output the 3D point cloud map*

```
#include <ros/ros.h>
#include <nav_msgs/Path.h>
#include <geometry_msgs/PoseStamped.h>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/filters/voxel_grid.h>
#include <pclomp/ndt_omp.h>
#include <tf2_ros/transform_broadcaster.h>
#include <boost/filesystem.hpp>
#include <Eigen/Geometry>
#include <pcl/common/transforms.h>

#include <sensor_msgs/PointCloud2.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl_conversions/pcl_conversions.h>

class NDTMapper {
public:
    NDTMapper() : nh_("~"), current_index_(1), global_map_(new pcl::Point
        initialize_parameters();
        initialize_publishers();
        initialize_ndt();
        load_initial_clouds();
    }

    void run() {
        ros::Rate rate(1); // Processing rate (1Hz)
        while (ros::ok()) {
            process_available_clouds();
            publish_trajectory();
            publish_global_map();
            rate.sleep();
```

```
        }
    }

private:
    void initialize_parameters() {
        nh_.param<std::string>("pcd_directory", pcd_directory_, "/home/ww
        nh_.param<double>("resolution", ndt_resolution_, 1.0);
        nh_.param<double>("step_size", step_size_, 0.1);
        nh_.param<double>("epsilon", epsilon_, 0.01);
        nh_.param<int>("max_iterations", max_iterations_, 64);
        nh_.param<int>("num_threads", num_threads_, 40);
        nh_.param<double>("voxel_leaf_size", voxel_leaf_size_, 0.5);
        nh_.param<double>("map_voxel_size", map_voxel_size_, 0.2);
    }

    void initialize_publishers() {
        path_pub_ = nh_.advertise<nav_msgs::Path>("/ndt_trajectory", 10);
        map_pub_ = nh_.advertise<sensor_msgs::PointCloud2>("/global_map",
        aligned_cloud_pub_ = nh_.advertise<sensor_msgs::PointCloud2>("/al
    }

    void initialize_ndt() {
        ndt_omp_.setResolution(ndt_resolution_);
        ndt_omp_.setStepSize(step_size_);
        ndt_omp_.setTransformationEpsilon(epsilon_);
        ndt_omp_.setMaximumIterations(max_iterations_);
        ndt_omp_.setNumThreads(num_threads_);
        ndt_omp_.setNeighborhoodSearchMethod(pclomp::DIRECT7);
    }

    void load_initial_clouds() {
        process_new_clouds(true);
        if (!clouds_.empty()) {
            add_to_trajectory(Eigen::Matrix4f::Identity());
            update_global_map(clouds_[0], Eigen::Matrix4f::Identity());
        }
    }

    void process_available_clouds() {
        size_t previous_count = clouds_.size();
        process_new_clouds();

        while (current_index_ < clouds_.size()) {
            auto result = align_consecutive_clouds(
                clouds_[current_index_ - 1],
                clouds_[current_index_]
            );
```

```
            if (result.hasConverged()) {
                Eigen::Matrix4f transform = result.getFinalTransformation

                ROS_INFO_STREAM("Transform " << current_index_-1 << " to
                           << transform);

                Eigen::Matrix4f global_transform;
                if(trajectory_.size())
                {
                    global_transform = trajectory_.back() * transform;
                }
                else
                {
                    global_transform = transform;
                }

                update_trajectory(global_transform);
                update_global_map(clouds_[current_index_], global_transfo
            }

            current_index_++;

            // repeat the publishment
            publish_trajectory();
            publish_global_map();
        }
    }

    void process_new_clouds(bool initial_load = false) {
        namespace fs = boost::filesystem;
        std::vector<fs::path> new_files;

        for (const auto& entry : fs::directory_iterator(pcd_directory_))
            if (entry.path().extension() == ".pcd") {
                int file_number = extract_file_number(entry.path().stem()
                if (file_number >= static_cast<int>(clouds_.size()) + 1)
                    new_files.push_back(entry.path());
                }
            }
        }

        std::sort(new_files.begin(), new_files.end(), [](const fs::path&
            return extract_file_number(a.stem().string()) <
                   extract_file_number(b.stem().string());
        });

        for (const auto& file : new_files) {
            auto cloud = load_and_filter_cloud(file.string());
```

```
            if (cloud && !cloud->empty()) {
                clouds_.push_back(cloud);
                ROS_INFO("Loaded %s (%zu points)",
                        file.filename().c_str(), cloud->size());
            }
        }
    }

    pcl::PointCloud<pcl::PointXYZ>::Ptr load_and_filter_cloud(const std::
        pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pcl
        if (pcl::io::loadPCDFile<pcl::PointXYZ>(path, *cloud) == -1) retu

        pcl::VoxelGrid<pcl::PointXYZ> voxel_filter;
        voxel_filter.setLeafSize(voxel_leaf_size_, voxel_leaf_size_, voxe
        voxel_filter.setInputCloud(cloud);

        pcl::PointCloud<pcl::PointXYZ>::Ptr filtered(new pcl::PointCloud<
        voxel_filter.filter(*filtered);
        return filtered;
    }

    pclomp::NormalDistributionsTransform<pcl::PointXYZ, pcl::PointXYZ>
    align_consecutive_clouds(
        const pcl::PointCloud<pcl::PointXYZ>::Ptr& target,
        const pcl::PointCloud<pcl::PointXYZ>::Ptr& source)
    {
        ndt_omp_.setInputTarget(target);
        ndt_omp_.setInputSource(source);

        pcl::PointCloud<pcl::PointXYZ>::Ptr aligned(new pcl::PointCloud<p
        ndt_omp_.align(*aligned);

        sensor_msgs::PointCloud2 msg;
        pcl::toROSMsg(*aligned, msg);
        msg.header.frame_id = "map";
        aligned_cloud_pub_.publish(msg);

        // return ndt_omp_.getResult();
        return ndt_omp_;
    }

    void update_trajectory(const Eigen::Matrix4f& global_transform) {
        trajectory_.push_back(global_transform);
        add_to_trajectory(global_transform);
    }

    void add_to_trajectory(const Eigen::Matrix4f& transform) {
        Eigen::Affine3f affine(transform);
```

```
        Eigen::Vector3f position = affine.translation();
        Eigen::Quaternionf rotation(affine.linear());

        geometry_msgs::PoseStamped pose;
        pose.header.stamp = ros::Time::now();
        pose.header.frame_id = "map";
        pose.pose.position.x = position.x();
        pose.pose.position.y = position.y();
        pose.pose.position.z = position.z();
        pose.pose.orientation.x = rotation.x();
        pose.pose.orientation.y = rotation.y();
        pose.pose.orientation.z = rotation.z();
        pose.pose.orientation.w = rotation.w();

        trajectory_poses_.poses.push_back(pose);
    }

    void update_global_map(const pcl::PointCloud<pcl::PointXYZ>::Ptr& clo
                           const Eigen::Matrix4f& transform) {
        pcl::PointCloud<pcl::PointXYZ>::Ptr transformed_cloud(new pcl::Po
        pcl::transformPointCloud(*cloud, *transformed_cloud, transform);

        *global_map_ += *transformed_cloud;

        // Downsample global map
        pcl::VoxelGrid<pcl::PointXYZ> voxel_filter;
        // voxel_filter.setLeafSize(map_voxel_size_, map_voxel_size_, map
        voxel_filter.setLeafSize(0.5, 0.5, 0.5);
        voxel_filter.setInputCloud(global_map_);

        pcl::PointCloud<pcl::PointXYZ>::Ptr filtered(new pcl::PointCloud<
        voxel_filter.filter(*filtered);
        global_map_.swap(filtered);
    }

    void publish_trajectory() {
        if (trajectory_poses_.poses.empty()) return;

        trajectory_poses_.header.stamp = ros::Time::now();
        trajectory_poses_.header.frame_id = "map";
        path_pub_.publish(trajectory_poses_);
    }

    void publish_global_map() {
        if (global_map_->empty()) return;

        sensor_msgs::PointCloud2 msg;
        pcl::toROSMsg(*global_map_, msg);
```

```
        msg.header.stamp = ros::Time::now();
        msg.header.frame_id = "map";
        map_pub_.publish(msg);
    }

    static int extract_file_number(const std::string& filename) {
        size_t underscore = filename.find_last_of('_');
        if (underscore != std::string::npos) {
            try {
                return std::stoi(filename.substr(underscore + 1));
            } catch (...) {}
        }
        return -1;
    }

    ros::NodeHandle nh_;
    ros::Publisher path_pub_;
    ros::Publisher map_pub_;
    ros::Publisher aligned_cloud_pub_;
    nav_msgs::Path trajectory_poses_;
    std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> clouds_;
    pcl::PointCloud<pcl::PointXYZ>::Ptr global_map_;
    pclomp::NormalDistributionsTransform<pcl::PointXYZ, pcl::PointXYZ> nd
    std::vector<Eigen::Matrix4f> trajectory_;
    size_t current_index_;

    // Parameters
    std::string pcd_directory_;
    double ndt_resolution_, step_size_, epsilon_, voxel_leaf_size_, map_v
    int max_iterations_, num_threads_;
};

int main(int argc, char** argv) {
    ros::init(argc, argv, "ndt_mapping_node");
    NDTMapper mapper;
    mapper.run();
    return 0;
}
```

4.4 ***Read the 3D point cloud data from the rosbag files, and perform the NDT for point cloud registration. Output the transformation and also output the 3D point cloud map***

```
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl_conversions/pcl_conversions.h>

#include <ros/ros.h>
#include <rosbag/bag.h>
#include <rosbag/view.h>
```

```
#include <nav_msgs/Path.h>
#include <geometry_msgs/PoseStamped.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl_conversions/pcl_conversions.h>
#include <pclomp/ndt_omp.h>
#include <pcl/filters/voxel_grid.h>
#include <tf2_ros/transform_broadcaster.h>
#include <Eigen/Geometry>

class NDTMappingNode {
public:
    NDTMappingNode() : global_map_(new pcl::PointCloud<pcl::PointXYZ>) {
        initialize_parameters();
        initialize_publishers();
        initialize_ndt();
    }

    void process_bag(const std::string& bag_path) {
        rosbag::Bag bag;
        try {
            bag.open(bag_path, rosbag::bagmode::Read);
        } catch (rosbag::BagException& e) {
            ROS_ERROR("Failed to open bag file: %s", e.what());
            return;
        }

        std::vector<std::string> topics{input_topic_};
        rosbag::View view(bag, rosbag::TopicQuery(topics));

        Eigen::Matrix4f pose = Eigen::Matrix4f::Identity();
        pcl::PointCloud<pcl::PointXYZ>::Ptr prev_cloud;

        for (const rosbag::MessageInstance& msg : view) {
            if (!ros::ok()) break;

            sensor_msgs::PointCloud2::ConstPtr cloud_msg =
                msg.instantiate<sensor_msgs::PointCloud2>();
            if (!cloud_msg) continue;

            pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud
            pcl::fromROSMsg(*cloud_msg, *cloud);

            // Preprocess cloud
            pcl::PointCloud<pcl::PointXYZ>::Ptr filtered = downsample_clo

            if (!prev_cloud) {
```

```
                prev_cloud = filtered;
                update_global_map(filtered, pose);
                continue;
            }

            // Perform registration
            Eigen::Matrix4f transform = perform_registration(prev_cloud,
            pose = pose * transform;

            // Update tracking
            update_trajectory(pose);
            update_global_map(filtered, pose);

            prev_cloud = filtered;

            // Publish intermediate results
            publish_global_map();
            publish_trajectory();
        }

        bag.close();
    }

private:
    void initialize_parameters() {
        nh_.param<std::string>("input_topic", input_topic_, "/velodyne_po
        nh_.param<double>("ndt_resolution", ndt_resolution_, 1.0);
        nh_.param<double>("ndt_step_size", ndt_step_size_, 0.1);
        nh_.param<double>("ndt_epsilon", ndt_epsilon_, 0.01);
        nh_.param<int>("ndt_max_iterations", ndt_max_iterations_, 64);
        nh_.param<int>("ndt_threads", ndt_threads_, 40);
        nh_.param<double>("voxel_leaf", voxel_leaf_, 0.3); // 0.5 is unst
        nh_.param<double>("map_voxel", map_voxel_, 0.6);
    }

    void initialize_publishers() {
        path_pub_ = nh_.advertise<nav_msgs::Path>("/ndt_trajectory", 10);
        map_pub_ = nh_.advertise<sensor_msgs::PointCloud2>("/global_map",
    }

    void initialize_ndt() {
        ndt_omp_.setResolution(ndt_resolution_);
        ndt_omp_.setStepSize(ndt_step_size_);
        ndt_omp_.setTransformationEpsilon(ndt_epsilon_);
        ndt_omp_.setMaximumIterations(ndt_max_iterations_);
        ndt_omp_.setNumThreads(ndt_threads_);
        ndt_omp_.setNeighborhoodSearchMethod(pclomp::DIRECT7);
    }
```

```
pcl::PointCloud<pcl::PointXYZ>::Ptr downsample_cloud(
    const pcl::PointCloud<pcl::PointXYZ>::Ptr& cloud)
{
    pcl::VoxelGrid<pcl::PointXYZ> voxel_filter;
    voxel_filter.setLeafSize(voxel_leaf_, voxel_leaf_, voxel_leaf_);
    voxel_filter.setInputCloud(cloud);

    pcl::PointCloud<pcl::PointXYZ>::Ptr filtered(new pcl::PointCloud<
    voxel_filter.filter(*filtered);
    return filtered;
}

Eigen::Matrix4f perform_registration(
    const pcl::PointCloud<pcl::PointXYZ>::Ptr& target,
    const pcl::PointCloud<pcl::PointXYZ>::Ptr& source)
{
    ndt_omp_.setInputTarget(target);
    ndt_omp_.setInputSource(source);

    auto t1 = ros::WallTime::now();
    pcl::PointCloud<pcl::PointXYZ>::Ptr aligned(new pcl::PointCloud<p
    ndt_omp_.align(*aligned);
    auto t2 = ros::WallTime::now();
    std::cout << "single (t2-t1) : " << (t2 - t1).toSec() * 1000 << "
    std::cout << "fitness (t2-t1): " << ndt_omp_.getFitnessScore() <<
    // ndt_omp_.align(*aligned);
    // auto t3 = ros::WallTime::now();
    // std::cout << "single (t3-t2) : " << (t3 - t2).toSec() * 1000 <
    // std::cout << "fitness (t3-t2): " << ndt_omp_.getFitnessScore()


    if (ndt_omp_.hasConverged()) {
        return ndt_omp_.getFinalTransformation();
    }
    return Eigen::Matrix4f::Identity();
}

void update_global_map(const pcl::PointCloud<pcl::PointXYZ>::Ptr& clo
                       const Eigen::Matrix4f& transform) {
    pcl::PointCloud<pcl::PointXYZ>::Ptr transformed(new pcl::PointClo
    pcl::transformPointCloud(*cloud, *transformed, transform);

    *global_map_ += *transformed;

    // Downsample global map
    pcl::VoxelGrid<pcl::PointXYZ> voxel_filter;
    voxel_filter.setLeafSize(map_voxel_, map_voxel_, map_voxel_);
```

```
        voxel_filter.setInputCloud(global_map_);

        pcl::PointCloud<pcl::PointXYZ>::Ptr filtered(new pcl::PointCloud<
        voxel_filter.filter(*filtered);
        global_map_.swap(filtered);
    }

    void update_trajectory(const Eigen::Matrix4f& pose) {
        Eigen::Affine3f affine(pose);
        Eigen::Vector3f position = affine.translation();
        Eigen::Quaternionf rotation(affine.linear());

        geometry_msgs::PoseStamped pose_msg;
        pose_msg.header.stamp = ros::Time::now();
        pose_msg.header.frame_id = "map";
        pose_msg.pose.position.x = position.x();
        pose_msg.pose.position.y = position.y();
        pose_msg.pose.position.z = position.z();
        pose_msg.pose.orientation.x = rotation.x();
        pose_msg.pose.orientation.y = rotation.y();
        pose_msg.pose.orientation.z = rotation.z();
        pose_msg.pose.orientation.w = rotation.w();

        trajectory_.poses.push_back(pose_msg);
    }

    void publish_trajectory() {
        trajectory_.header.stamp = ros::Time::now();
        trajectory_.header.frame_id = "map";
        path_pub_.publish(trajectory_);
    }

    void publish_global_map() {
        sensor_msgs::PointCloud2 msg;
        pcl::toROSMsg(*global_map_, msg);
        msg.header.stamp = ros::Time::now();
        msg.header.frame_id = "map";
        map_pub_.publish(msg);
    }

    ros::NodeHandle nh_;
    ros::Publisher path_pub_;
    ros::Publisher map_pub_;
    nav_msgs::Path trajectory_;
    pcl::PointCloud<pcl::PointXYZ>::Ptr global_map_;
    pclomp::NormalDistributionsTransform<pcl::PointXYZ, pcl::PointXYZ> nd

    // Parameters
```

```
    std::string input_topic_;
    double ndt_resolution_, ndt_step_size_, ndt_epsilon_;
    double voxel_leaf_, map_voxel_;
    int ndt_max_iterations_, ndt_threads_;
};

int main(int argc, char** argv) {
    ros::init(argc, argv, "ndt_mapping_node");

    if (argc < 2) {
        ROS_ERROR("Usage: rosrun your_package ndt_mapping_node path_to_ba
        return 1;
    }

    NDTMappingNode mapper;
    mapper.process_bag(argv[1]);

    return 0;
}
```