

# EECS 465/ROB 422: Introduction to Algorithmic Robotics Fall 2024 Final Project Report

## Title: Point Cloud Processing using Point Feature Histograms

Students: Mingrui Li, Xintong Liu

Date: December 15, 2024

## Introduction

The alignment and registration of point clouds is a fundamental problem in the fields of robotics, computer vision, and 3D modeling. Accurate point cloud alignment is essential for a variety of applications, including object recognition, scene reconstruction, and navigation. Traditional methods, such as the Point Feature Histogram (PFH), have been widely used for these tasks, but they often suffer from high computational complexity and inefficiencies, particularly when dealing with large datasets.

This project introduces a series of optimizations to the traditional PFH method, aimed at enhancing its efficiency and scalability. Our approach addresses the primary challenges associated with PFH—high computational cost and slow convergence—by implementing five innovative optimizations. These improvements not only maintain the accuracy of the traditional method but also significantly reduce the computational burden, making our approach more suitable for real-time applications and large-scale point cloud processing.

## Implementation

### Overview

The project involves implementing the PFH method to align two partially overlapping point clouds. The approach involves computing a histogram of features for each point in the source cloud, comparing these histograms with the target cloud, and iteratively refining the alignment using the Iterative Closest Point (ICP) algorithm.

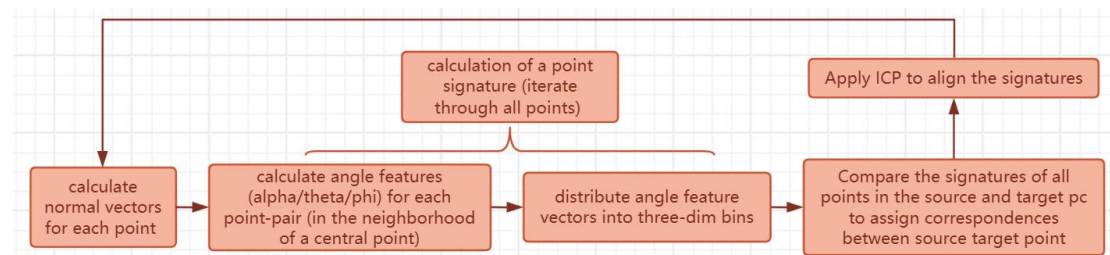


Figure 1. Standard version of PFH-based ICP.

## 1. Point Cloud Feature Extraction (PFH)

### 1.1 Neighborhood Selection

For each point  $p_i$  in the point cloud, select its  $k$ -nearest neighbors or all points within a certain distance threshold to form a local neighborhood.

## 1.2 Compute Surface Normal

For each point  $p_i$  and its neighborhood with a radius  $r$ , compute the surface normal  $n$ .

## 1.3 Compute Eigenvectors and Eigenvalues

Construct a matrix  $U$  using the neighborhood points relative to  $p_i$ . Compute the eigenvectors and eigenvalues of  $UU^T$  to determine the curvature and principal directions of the surface patch.

## 1.4 Compute Angle statistics

Represent the point-pair geometric relationship with three angle statistics  $\alpha$ ,  $\phi$ , and  $\theta$ .

## 1.5 PFH Signature

For each point, create a feature signature based on the statistics of how the surface normal changes within the neighborhood. This involves computing angular features for each pair of points in the neighborhood, capturing the change in surface normal between these points.

# 2. PFH-Based ICP Algorithm

## 2.1 Initialization

Start with an initial guess for the transformation between the source point cloud  $P$  and the target point cloud  $P'$ .

## 2.2 Correspondence Establishment

For each point in  $P$ , find the closest point in  $P'$  and compute the PFH feature signature for both points. Establish correspondences based on the similarity of these signatures.

## 2.3 Transformation Estimation

Using the established correspondences, estimate the transformation (rotation  $R$  and translation  $t$ ) that minimizes the distance between corresponding points. This is done by singular value decomposition (SVD) on the covariance matrix of the corresponding points.

## 2.4 Transformation Application

Apply the estimated transformation to the source point cloud  $P$  to obtain a new transformed point cloud  $P''$ .

## 2.5 Convergence Check

Repeat steps 2.2 to 2.5. If the change in the transformation between iterations is below a certain threshold or a maximum number of iterations is reached, the algorithm converges, and the final transformation is returned.

# Methodology

Based on the standard PFH-ICP algorithm, we iteratively upgrade our version to achieve better performance and less elapsed time, which means the latter optimization algorithms inherit the strategies from the previous.

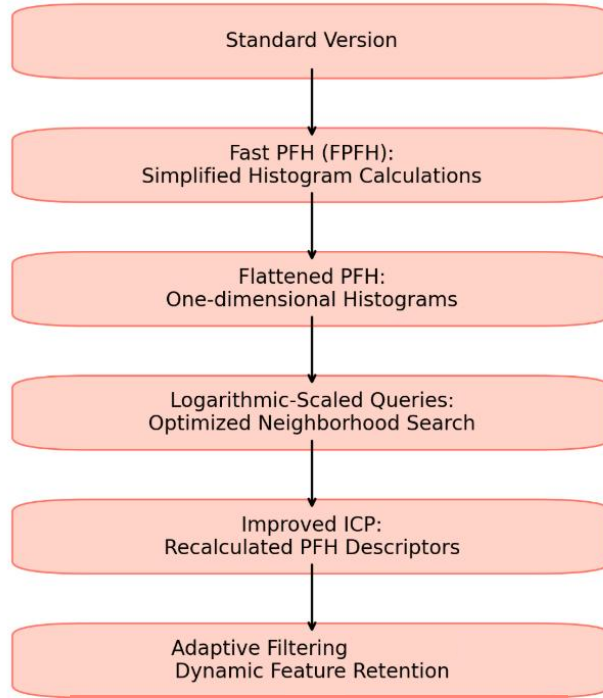


Figure 2. Flowchart of iterative optimizations.

## 1. Initial PFH Implementation:

- The initial approach involves transforming all points into a KD-tree with a complexity of  $O(n \log n)$ , followed by searching within the KD-tree to find all neighborhood points within a distance  $r$ , resulting in a complexity of  $O(k^2 n)$  where  $k$  is the number of neighbors within the radius distance of  $r$ . The KD-Tree is a space-partitioning data structure that divides space into orthogonal regions, which is more suitable for multi-dimensional data like point clouds compared to a traditional BST.
- For each point, a signature is computed based on the geometric relationships within its neighborhood. First, we calculate the normal vectors of each point by finding the eigenvector corresponding to the smallest eigenvalue of the covariance matrix  $C$  of the neighborhood points.

$$C = \frac{1}{n-1} \sum_{i=1}^n (neighbors_i - centroid)(neighbors_i - centroid)^T$$

The final normal vector for a point  $p$  is given by:

$$n_p = eigvecs[:, \arg \arg (eigvals) ]$$

where  $eigvals$  and  $eigvecs$  are the eigenvalues and eigenvectors of the covariance matrix of the neighborhood points of  $p$ .

- Then, we calculate three angle statistics  $\alpha$ ,  $\phi$ , and  $\theta$  to parametrize each relationship between two points.  $\alpha$  is the angle between the normal vector of the source point and the direction vector from the source point to the target point.  $\phi$  is the angle between the normal vector of the target point and the direction vector pointing from the source point to the target point.  $\theta$  is the angle between the normal vectors of the source and target points.

$$\begin{aligned}\alpha &= \arccos(n_s \cdot dp_{normalized}) \\ \phi &= \arccos(n_t \cdot dp_{normalized}) \\ \theta &= \arccos(n_s \cdot n_t)\end{aligned}$$

- Finally, we distribute the three-dimension vectors into the three-dimensional grids with  $m$  bins in each dimension ( $m^3$  grids in total) to form a histogram. The normal vector for each point in the point cloud is computed by finding the eigenvector corresponding to the smallest eigenvalue of the covariance matrix of the neighborhood points. This process effectively captures the local surface orientation at each point.
- The error used to measure the level of alignment is the mean value of the euclidean distances between the corresponding points in the source and target point cloud. Since the input source file and the target file will be first normalized and then processed with PFH-based ICP, the error is referenceable and scalable for different sizes of point clouds.

## 2. Drawbacks of the initial approach

- The PFH construction is inefficient, for a large number of points in the neighborhood, it costs  $O(k^2n)$  time to construct a PFH for all points. Thus it is extremely time-consuming.
- The radius and the threshold should be hand-set. This may require repeated trials and error for the model to reach a good performance with respect to certain tests.
- The histogram is a sparse tensor, where many elements are 0. This makes the ICP much slower and is memory inefficient.

## 3. Optimization One: Efficient Signature Calculation “fast PFH” (FPFH):

- To avoid the quadratic time complexity of signature computation, we introduce an optimization that reduces the complexity of signature computation to  $O(nk)$ . For each central point, instead of computing relationships among all pairs of neighbors, we only calculate the relationships that radiate from the central point to form a  $histogram_0$ . This  $histogram_0$  is then dynamically combined with the weighted  $histogram_i$ , from the surrounding neighbor  $i$ , effectively using dynamic programming to store and combine histograms. The weight of each neighbor histogram is computed as follows, where the *distance* indicates the Euclidean distance between the central point and the neighbor, and *maxDist* is the maximum distance among all neighbors to the central point:

$$weight = 1 - 4 * \left( \frac{distance - maxDist/2}{maxDist} \right)^2$$

$$PFH_{final} = histogram_0 + \sum_{i=1}^n weight_i \cdot histogram_i$$

- This optimization significantly reduces the computational burden by focusing on the immediate neighborhood's contribution to the histogram, and memorization of each point's histogram.
- The weight design is based on the following idea: For the points closest to the central point, the error, which can be regarded as a gaussian noise, extremely matters to the parameters in the histogram, which are angles. In the meanwhile, for the points at the edge of the neighborhood, the surface condition may be

different and cannot indicate the surface condition around the central point well. So both points far from and close to the central point are given low significance, while the points with medium distance are given higher weight.

#### 4. Optimization Two: Efficient Histogram Distance Calculation “flatten PFH”:

- The second optimization uses a flattened vector (size  $3m$ ) to store the histogram and adjusts the distance metric between two histograms. To be specific, we transform the Euclidean distance between  $n$  three-dimensional arrays of histograms (with  $m^3$  bins) into the Euclidean distance between three one-dimensional histograms. Each of these one-dimensional histograms represents the distribution along one axis, simplifying the distribution process of the point-pair relationships and distance calculation and further reducing computational cost.

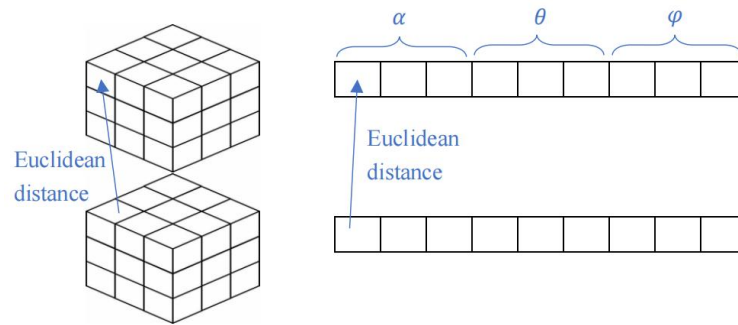


Figure 3. Example of flattening histogram with bin number  $m = 3$ . The 27-grid matrix histogram is transformed into size  $3m = 9$  vector.

#### 5. Optimization Three: Fast Neighborhood Search with Parallel Calculation “logarithmic scaled query PFH” (LSQ-PFH):

- The third optimization focuses on quickly finding neighborhood points by limiting the search to approximately  $c \sim 8 \log n$  nearest points for each central point instead of all the neighbor points within a radius of  $r$ , aiming to maintain the overall algorithm complexity at  $O(n \log n)$ , and independent of radius setting. This is achieved by using a more efficient nearest neighbor search algorithm and parallelizing the calculations to leverage multi-core processing capabilities.
- Such optimization makes the number of neighbors scalable with the point cloud size, avoiding exploration of query-ball radius  $r$  which may not be adaptable to future point clouds.
- Also, this optimization contributes to the conditions where the point cloud is not uniformly distributed.

#### 6. Optimization Four: Recalculating PFH for ICP iterations (improved ICP):

- The fourth optimization focuses on the recalculation of PFH (Point Feature Histograms) descriptors for each iteration of the Iterative Closest Point (ICP) algorithm. Recalculating the PFH descriptors of the transformed point cloud in each ICP iteration ensures that the registration process is based on the most up-to-date information about the point cloud's features. This can lead to more accurate alignments, particularly in cases where the initial estimate is far from the optimal solution.

## 7. Optimization Five: Feature Filtering with Adaptive Retainment Ratios:

### 7.1 Overview

The fifth optimization concentrates on enhancing the efficiency of feature selection in point cloud processing by dynamically adjusting the retention of features, or specifically, the signatures of the points based on adaptive ratios. This approach reduces the algorithm's complexity while adapting to the varying densities and characteristics of different point clouds.

### 7.2 Evaluator

Firstly, we denote the importance evaluator of a point as below. Here  $\alpha_i$ ,  $\varphi_i$ ,  $\theta_i$  indicates all the elements in the  $\alpha$ ,  $\varphi$ ,  $\theta$  parts of the  $3m$  ( $m$  is the number of bins for each dimension) length vector in the histogram, and the evaluator is the sum of the ratio between the maximum in the group and the sum of the group. As the larger ratio indicates more points have the same normal directions, the larger evaluator means less geometrically unique and significant.

$$evaluator = \frac{\max(\alpha_i)}{\sum(\alpha_i)} + \frac{\max(\varphi_i)}{\sum(\varphi_i)} + \frac{\max(\theta_i)}{\sum(\theta_i)}$$

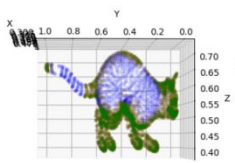
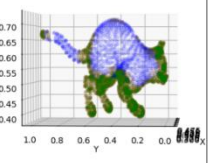
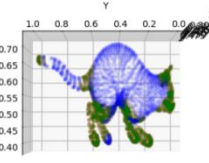
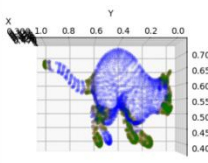
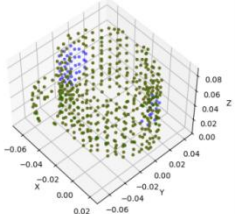
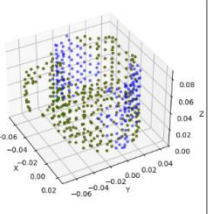
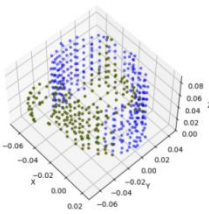
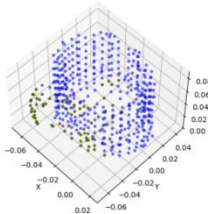
Threshold	1.1	1	0.9	0.8
Testcase 1				
Testcase 2				

Table 1. Retained points (green) after filtering at different thresholds for two cases (the cat and the cup)

This evaluator evaluates the “wrinkling” around the central point. If the surface around the central point is relatively flat, it is less necessary to include the point to depict the surface around the point, while if the surface around the central point is relatively wrinkling, the importance of the central point increases to depict the surface condition with rich geometric information.

Then the evaluator works fine to describe the surface condition around the evaluation point. Here is a small experiment that shows the impact of different thresholds on the evaluator.

In this picture, the evaluators of blue points are larger than the threshold, while those of green points are smaller. This indicates that the back and tail of the cat is of least wrinkling and contain less geometric information of the point cloud, while the head, foot and tail tip is of most information and importance, which corresponds to our intuition. Focusing on the important geometric parts may improve the efficiency and even performance of alignment.

### 7.3 Filtering

- To determine which points signatures to retain in the cat, we need a filter function that outputs the possibility that the point is sampled for alignment. For

those important points we want them to remain in the process, while for the less important points we only want to retain a few to indicate the smooth surface.

- For this we construct 3 different filter functions, 3 different retainment ratios for each filter function and 4 clip modes.

### 7.3.1 filter functions

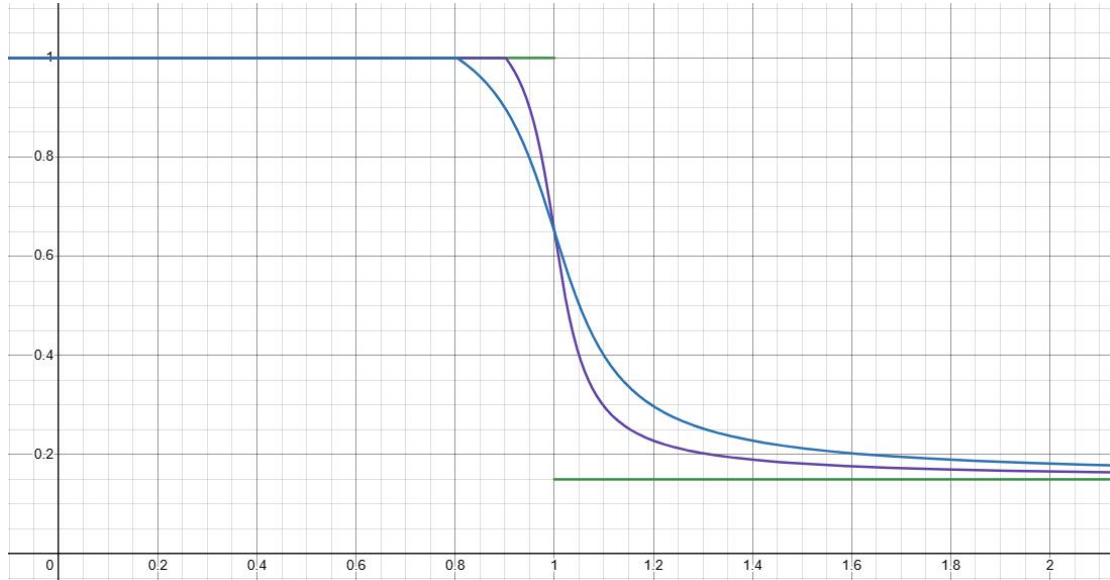


Figure 4. Green: clip,  $p = \text{step}(1 - x)$ ; Blue: smooth,  $p = \arctan(10 - 10x)/\pi + 0.65$ ; Purple: arctan,  $p = \arctan(20 - 20x)/\pi + 0.65$ . All clipped to the range of (0.15,1).

- The filter functions are the main part for the filtering. From the test shown above, we hope the filter functions can keep most points with evaluators smaller than 0.8, and dispose of a large ratio of points with evaluators larger than 1.1.
- The most direct way is using the step function, denoted as “clip” in filter.py. This function retains all points with evaluator smaller than a threshold, while disposes 85% of points larger than that threshold
- Another idea is a smoother version of the step function. So, we use the arctan function, which is a widely-adopted function to map a real number to (0,1) in machine learning, e.g., in the LSTM model. In this part a smoother and a steeper version are introduced for comparison. The arctan function is multiplied by  $1/\pi$  to limit its change to 1 and add 0.65 to ensure basic possibility. The Figure 4. shows the shapes of different filter functions.

### 7.3.2 thresholds design

There are three types of thresholds in the filter function, which is arranged around 1 (dense, as shown in Table 1.), 0.8 (medium) and 0.6(sparse). They keep about 75%, 50% and 25% data respectively.

### 7.3.3 clip mode design

In some conditions the points are so dense that we can also dispose of some points despite their importance. And there are some other conditions that we want to avoid disposing of too many points even on a smooth surface. Clipping means when the values exceed certain threshold, replace the values with the threshold value. There are 4 clip modes: normal, protective, light and protective-light. At normal mode the filter function is clipped to (0.15, 1). For protective and protective-light modes, the lower limit is clipped to 0.25 and for light and protective-light modes, the upper limit is clipped to 0.6.



#### 7.4 Advantages

- The optimization's adaptive retention ratios make the number of features scalable with the size and complexity of the point cloud, eliminating the need for empirical settings that may not be suitable for all point clouds. This dynamic adjustment ensures that the most relevant features are retained, which is crucial for the subsequent stages of point cloud alignment.
- In summary, feature filtering improves the efficiency by providing a scalable, adaptable, and efficient method for retaining only the most significant features from point clouds, thus enhancing the performance of downstream algorithms such as ICP.

#### 8. ICP Alignment:

- With the optimized PFH features, the Iterative Closest Point (ICP) algorithm is applied to align the source and target point clouds. The transformation that minimizes the distance metric between the signatures of corresponding points in the two clouds is iteratively refined until convergence or a maximum number of iterations is reached. The convergence is defined by  $tolerance = 1e - 6$  and  $patience = 5$  (*sometimes 3 or 2 based on the test cases*). When the ICP process has done at least 5 iterations of alignment with the error changes less than the tolerance, the alignment is deemed as converged.

## Results and Analysis

We have run three tests with small, medium, and large point cloud size to evaluate the time and performance of the six versions of PFH based ICP (one standard and five optimized). For all the point cloud images, the blue points indicate the source point cloud, the red for target, and the green for transformed point cloud generated by our algorithms.

### 1 Comparison among original PFH, fast PFH, flattened PFH, and LSQ-PFH with small point cloud.

#### 1.1 Experimental setup

Shape: mug cup

Size: 495 points

Data process: rotation and translation from source to generate target

Query radius  $r$ : 0.02

Query neighbor number:  $8\log(n)$

Bin number:  $8/\text{dimension}$

#### 1.2 Experiment data and images



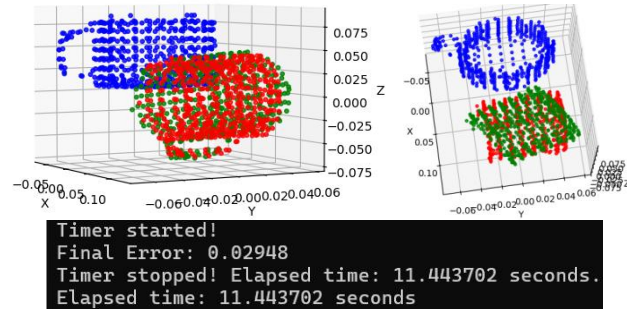


Figure 5. Time and performance of original PFH w.r.t. small point cloud

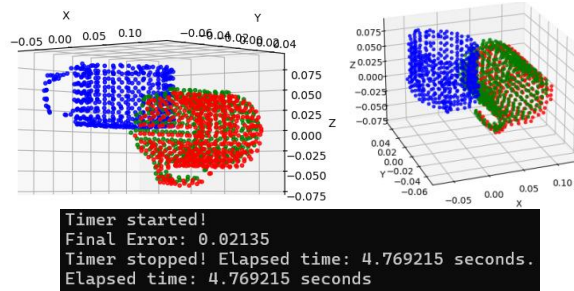


Figure 6. Time and performance of fast PFH w.r.t. small point cloud

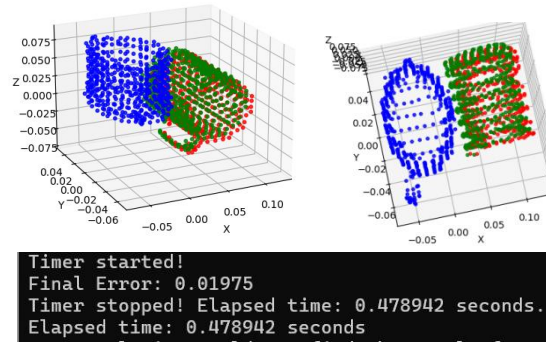


Figure 7. Time and performance of flattened PFH w.r.t. small point cloud

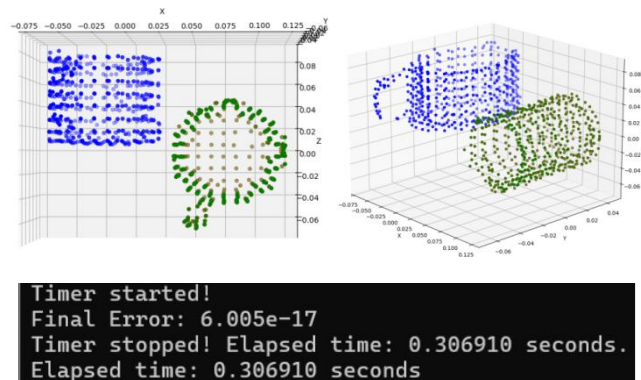


Figure 8. Time and performance of LSQ-PFH w.r.t. small point cloud

### 1.3 Analysis

From the elapsed time of three versions, we can conclude that fast PFH is faster than the original version, while flattened PFH can be much faster than fast PFH. Besides, the alignment of point clouds is much better by fast PFH than by original

PFH. The error has been cut by about 1/3. For the flattened PFH, the error is further reduced. For the LSQ-PFH, the error is almost zero and the elapsed time is also reduced. Compared with the original PFH, the LSQ-PFH has reduced the time by about 30 times, and improved performance to almost perfection.

In conclusion, the most up-to-date version LSQ-PFH has been a great improvement compared with the original PFH both in terms of time and performance.

## 2 Comparison between LSQ-PFH with filtering and without filtering with medium point cloud.

### 2.1 Experimental setup

Shape: cat

Size: 3400 points

Data process: source point cloud is normalized; rotation and translation from source to generate target; random noise  $\sim \text{Normal}(\text{std}=0.001)$  is added to source and target points separately

Query neighbor number:  $8\log(n)$

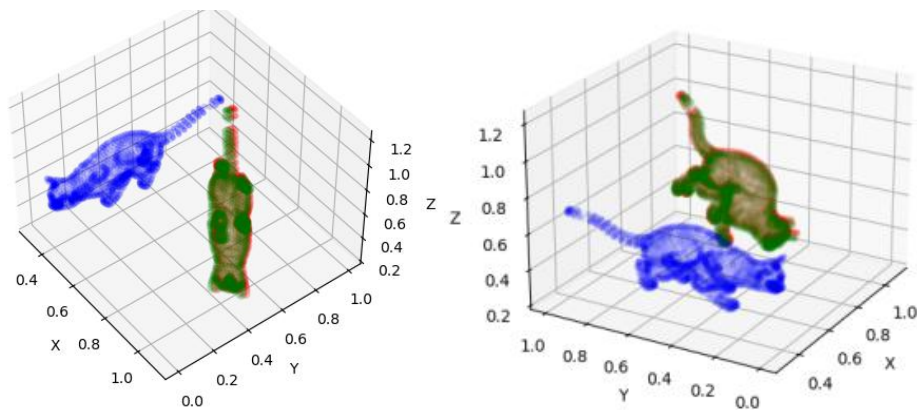
Bin number: 10/dimension

Filter\_function="arctan"

Retainment\_ratio="sparse"

Clip\_mode="light"

### 2.2 Experiment data and images



```
Timer started!
Normal calculation time is: 0.5008
SPFH TIME      4.901881217956543
SPFH TIME      5.13112211227417
PFH calculation time is: 6.1180
Final Error: 0.1001
ICP calculation time is: 5.7669
Timer stopped! Elapsed time: 12.385998 seconds.
Elapsed time: 12.3860 seconds
```

Figure 9. Time and performance of LSQ-PFH without filtering w.r.t. medium point cloud

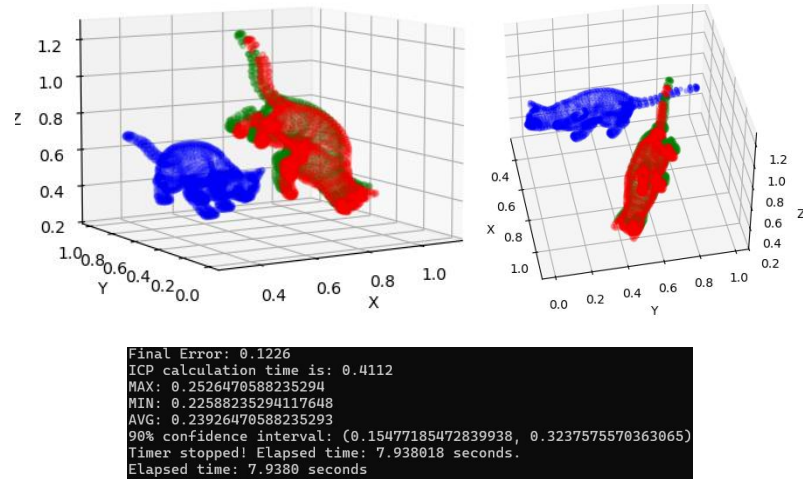


Figure 10. Time and performance of LSQ-PFH with filtering w.r.t. medium point cloud

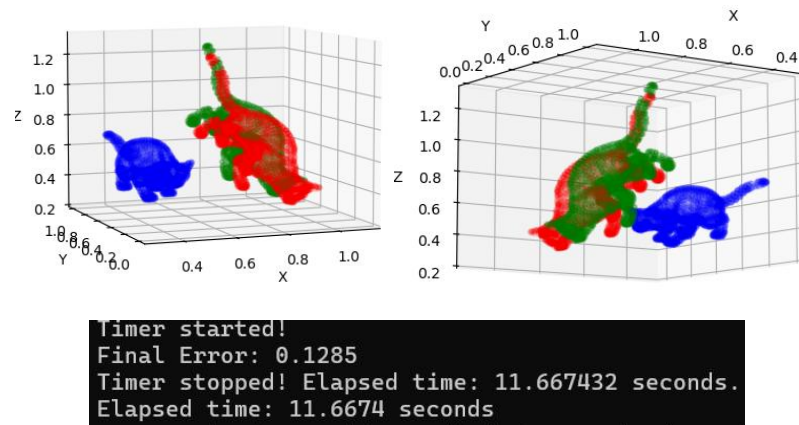


Figure 11. Time and performance of LSQ-PFH without filtering w.r.t. medium point cloud with Gaussian noise

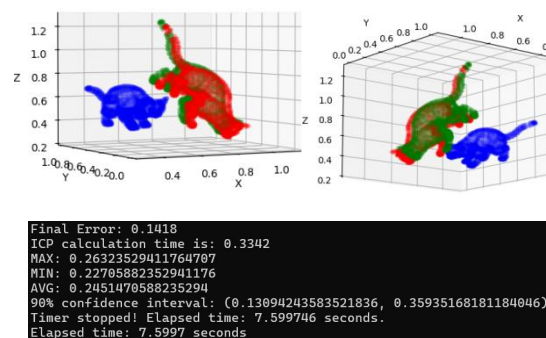


Figure 12. Time and performance of LSQ-PFH with filtering w.r.t. medium point cloud with Gaussian noise

## 2.3 Analysis

The elapsed time for LSQ-PFH with filter has been reduced by  $\frac{1}{3}$ , compared with the algorithm without filter, and the trade-off in final error is still relatively acceptable.

Also, the filtering makes the algorithm more robust to random noise. From the point cloud visualization in Figure 9 and 10, we can observe that the LSQ-PFH with filtering makes the transformed cloud more aligned with the target. The larger error may be due to the wrong correspondences between the source and target points.

In conclusion, filtering speeds up the algorithm to a large extent with acceptable trade-off in performance and makes the algorithm more robust to random noise in both source and target point clouds.

### 3 Comparison between filtered LSQ-PFH with improved ICP and the one without recalculation with medium point cloud.

#### 3.1 Experimental setup

Shape: cat

Size: 3400 points

Data process: source point cloud is normalized; rotation and translation from source to generate target; random noise  $\sim \text{Normal}(\text{std}=0.001)$  is added to source and target points separately

Query neighbor number:  $8\log(n)$

Bin number: 10/dimension

Filter\_function="arctan"

Retainment\_ratio="sparse"

Clip\_mode="light"

#### 3.2 Experiment data and images

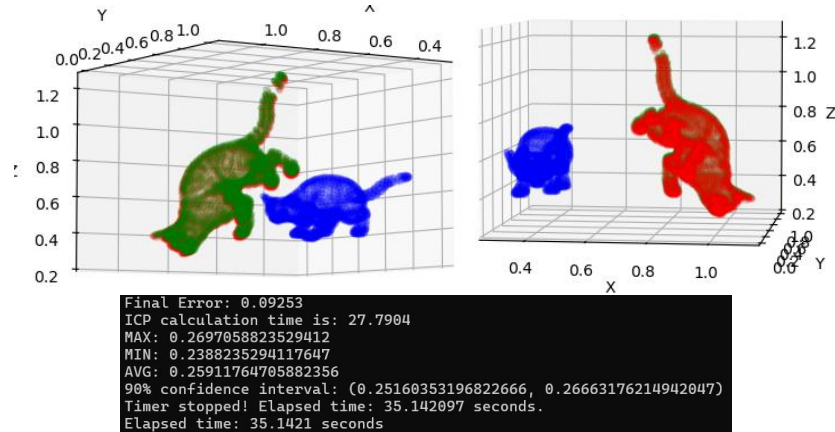


Figure 13. Time and performance of filtered LSQ-PFH with improved ICP w.r.t. medium point cloud with Gaussian noise

#### 3.3 Analysis

From the visualization, we can observe that the point cloud is much better aligned than in the last section (Figure 12), with the error being reduced by  $\frac{1}{3}$ . Yet, the time cost has risen significantly, which makes it not so suitable for real-time aligning of large samples. However, with the great alignment of complex objects it

achieves, the algorithm can be used for applications which demand high precision with less strict time limit.

## 4 Retainment ratio v.s. final error.

### 4.1 Retainment ratio

Retainment ratio is the ratio of the number of points before and after the filtering. It can be used to evaluate the strength of the filter and estimate the reduction of working time. It is hoped that a filter achieves as smaller retainment ratio as possible while maintaining a relatively good performance, as smaller retainment ratio leads to fewer points to process and faster alignment.

### 4.2 Experimental setup

Shape: cat

Size: 3400 points

Data process: source point cloud is normalized; rotation and translation from source to generate target; random noise  $\sim \text{Normal}(\text{std}=0.001)$  is added to source and target points separately

Query neighbor number:  $8\log(n)$

Bin number: 10/dimension

Filter\_function="arctan"/"smooth"/"clip"

Retainment\_ratio="sparse"/"dense"/"medium"

Clip\_mode="light"/"normal"

### 4.3 Experiment data and images

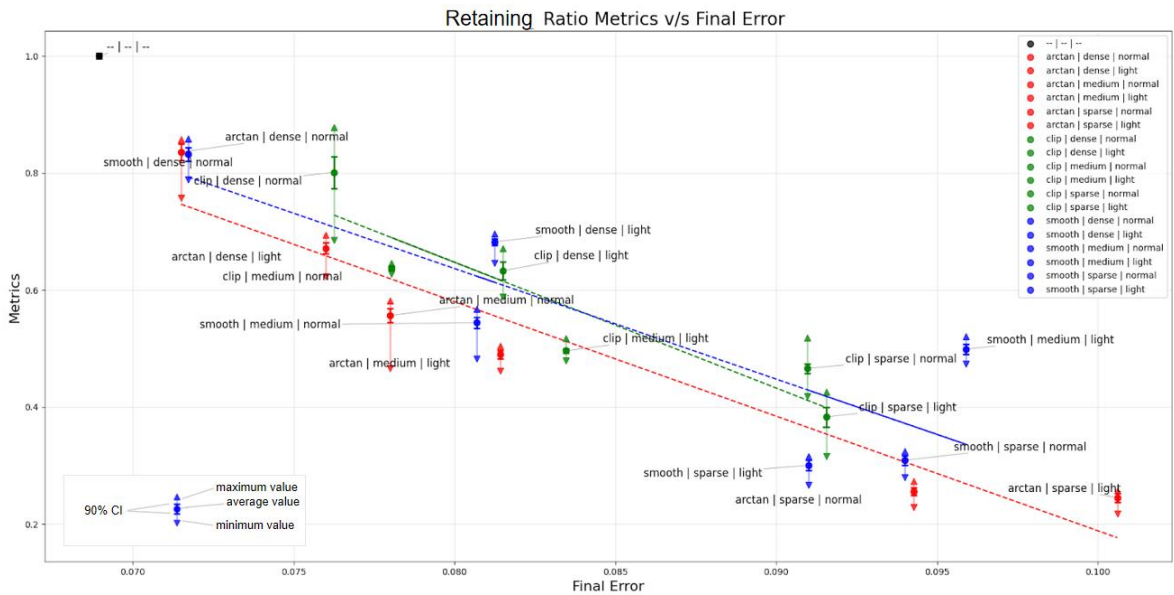


Figure 14. The relationship between the retainment ratio and final error of alignment with hue of different filtering function types.

## 4.4 Analysis

From Figure 14, there are strong linear relationships between the retainment ratio and final error of alignment for all three types of filtering function (  $R^2 = 0.9160$  for *arctan*;  $R^2 = 0.8458$  for *clip*;  $R^2 = 0.7258$  for *smooth* ). It indicates that more points used in alignment lead to lower error and higher precision of alignment, that is, there exists a tradeoff between error and retainment ratio. Besides, the red line is consistently lower than the two other lines, which means to achieve the same level of precision, arctan function can be more efficient in sampling significant points with richer geometric information.

## 5 Retainment ratio and mean error v.s. bin number.

### 5.1 Experimental setup

Shape: cat

Size: 3400 points

Data process: source point cloud is normalized; rotation and translation from source to generate target; random noise  $\sim \text{Normal}(\text{std}=0.001)$  is added to source and target points separately

Query neighbor number:  $8\log(n)$

Bin number: from 3 to 25

max\_iters=1000,

patience=3,

filter\_function="arctan",

retainment\_ratio="sparse",

clip\_mode="light",

### 5.2 Experiment data and images

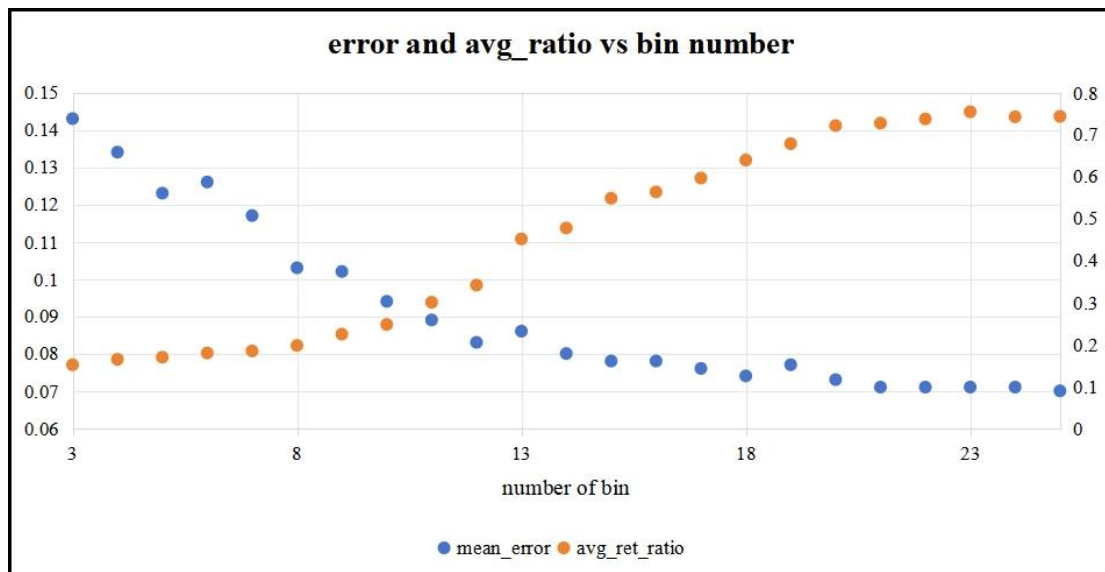




Figure 15. The mean error and average retainment ratio for different numbers of bins per dimension ( $m$ ). The left Y-axis corresponds to the range of the mean error, while the right Y2-axis corresponds to the range of the average retainment ratio during iterations of ICP.

### 5.3 Analysis

From Figure 15, it shows that with the increase of the bin number, the average ratio increases and the mean error decreases. After there are more than 20 bins, both average ratio and mean error become stable. It remains not known whether it's the more delicate division of angle by more bins that reduces the mean error or it is just plainly caused by increased retainment ratio.

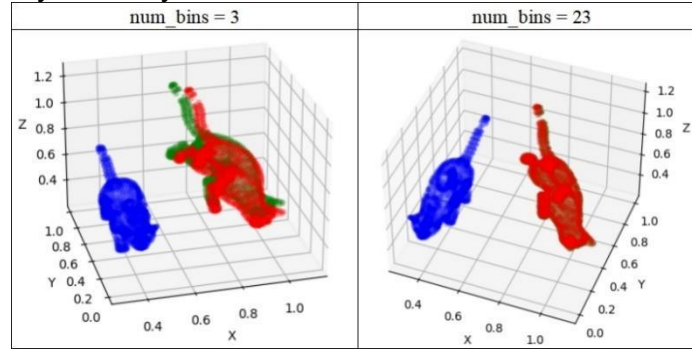


Figure 16. The performance of alignment with different bin numbers (left  $m = 3$  v.s. right  $m = 23$ ).

## 6 Real-life room scan

### 6.1 Experimental setup

Shape: room with person

Size: 112586 points

Data process: source point cloud is randomly sampled for 1/6 of the original points and then normalized; rotation and translation from source to generate target; random noise  $\sim \text{Normal}(\text{std}=0.001)$  is added to source and target points separately

Query neighbor number:  $8\log(n)$

Bin number: 10/dimension

max\_iters=1000,

patience=5,

filter\_function="arctan",

retainment\_ratio="sparse",

clip\_mode="light",



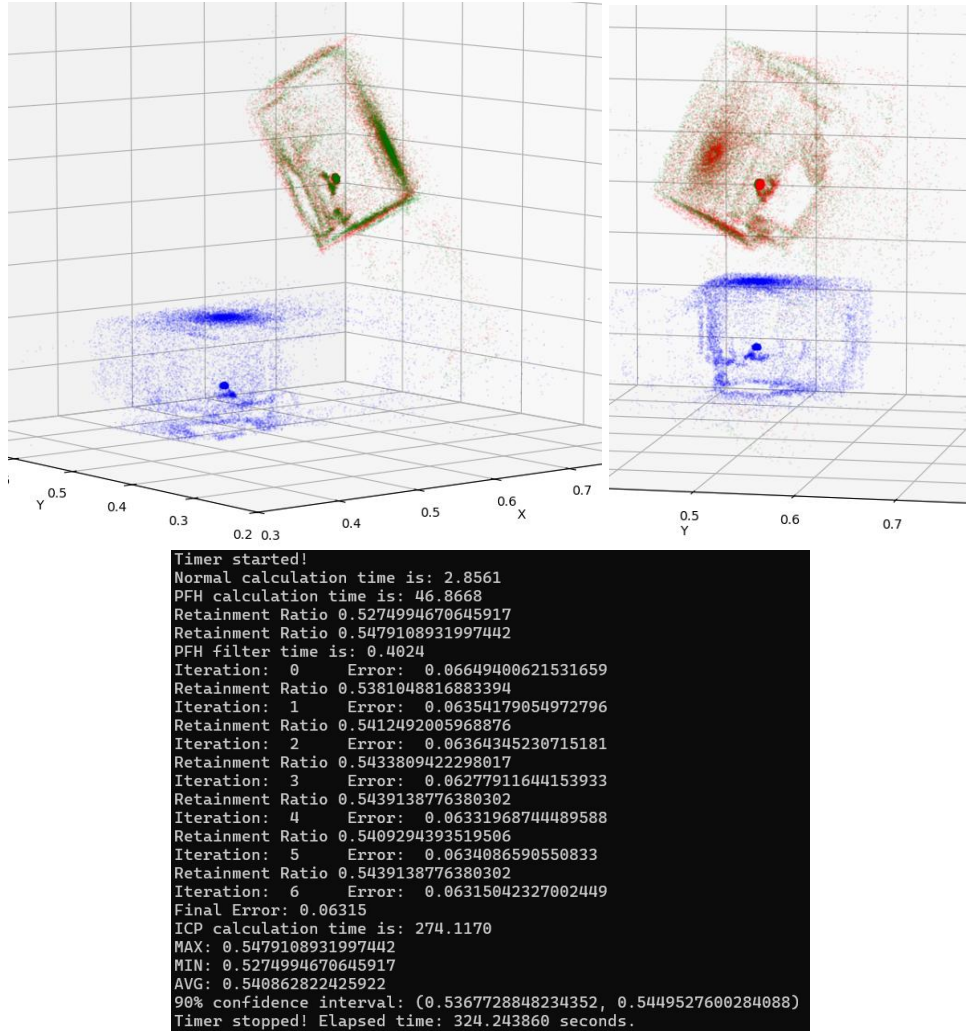


Figure 17. The time and performance of alignment of the room with our most-updated version of PFH-ICP.

## 6.2 Analysis

From the visualization in Figure 17., we can observe that the transformed room scan by our algorithm aligns quite well with the target room scan point cloud. Though only 1/6 of the points are involved in the calculation for the transformation and visualization to increase computation efficiency and clarity of the point cloud shape, we can predict that the whole point cloud is already aligned to the target in general by observation. The elapsed time is 324.24 seconds, which means the algorithm is quite efficient given that the original point cloud data are more than 0.1 million.

## Conclusion

In this project, we have successfully developed and optimized a Point Feature Histogram (PFH)-based algorithm for point cloud alignment, overcoming the inherent limitations of the traditional PFH method. By introducing a series of innovative optimizations, including fast PFH computation, flattened histograms, logarithmic-scaled queries, adaptive filtering, and improved ICP recalculations, our approach has achieved substantial improvements in both computational efficiency and alignment accuracy.

Key outcomes include:

1. **Efficiency Gains:** The optimized PFH algorithm has reduced computational complexity from quadratic to near-logarithmic, enabling faster processing even with large and complex datasets. For instance, in our tests, LSQ-PFH with filtering has decreased processing time by up to 30 times while maintaining alignment precision.
2. **Scalability:** Dynamic neighborhood selection and adaptive feature retention ratios have ensured the algorithm's adaptability to varying point cloud densities, making it suitable for diverse real-world applications.
3. **Robustness:** The use of advanced filtering techniques has not only accelerated processing but also enhanced robustness to noise, which is crucial in scenarios involving real-world noisy data.
4. **Accuracy:** The integration of recalculated PFH descriptors in ICP iterations has further improved alignment precision, demonstrating the potential of our approach for high-accuracy applications.

These advancements have made our algorithm particularly advantageous for real-time applications (LSQ-PFH), large-scale point cloud processing (improved ICP), and tasks requiring high precision (improved ICP), such as robotics, 3D modeling, and autonomous navigation. Future directions include exploring more sophisticated query strategies and integrating machine learning techniques to further enhance feature selection and alignment performance.

## Code Structure

1. `query_improved.py`: This module contains the main functions for computing PFH descriptors and performing ICP alignment with query optimization and parallel computation. It includes the `icp_with_fpfh_within_query` function, which is the core of the optimized alignment process.
2. `utility.py`: Utility functions for reading point clouds, computing normals, and visualizing results. It includes functions like `read_pcd_to_numpy`, `compute_normals`, and `view_pc` for handling point cloud data. **\*view\_pc function is adapted from the starter codes for homework4.**
3. `timer.py`: A simple timer class for measuring the performance of different stages of the algorithm. It helps in tracking the time taken for normal calculation, PFH calculation, and ICP calculation.
4. `filter.py`: Functions related to feature filtering, such as `arct_func`, `clip_func`, and `smooth_func`, which are used to dynamically adjust the retention of features based on adaptive ratios.
5. `proj_bin_improved.py`: Implements the computation of FPFH histograms in one dimension, which is part of the optimization process to reduce computational complexity.
6. `proj_fpfh.py`: Original implementation of the FPFH computation, which uses three-dimensional histograms for point cloud feature extraction.
7. `proj_pfh.py`: Houses the original PFH computation method, which is the baseline for the optimizations implemented in the project.

8. `demo.py`: A demonstration script that loads point cloud data, applies transformations, and utilizes the optimized ICP alignment method to align source and target point clouds of a room scan.