# Point Cloud Processing with Point Feature Histograms

Alexander Groh
grohbot@umich.edu

Steven Liu
stvnliu@umich.edu

***Abstract*— This report describes our final project implementation and results for EECS 498 - Introduction to Algorithmic Robotics. We chose the Point Cloud Processing topic.**

## I. Motivation

3D Point cloud is a common type of data that robotic sensors such as a laser rangefinder returns. This data enables the robot to localize and map itself for navigation purposes, reason about its surroundings via geometric primitives, and identify or segment objects in the environment that it should care about.

The Iterative Closest Point algorithm is a key component to realizing these tasks by providing the ability to estimate pose change between two consecutive data frames. Yet, the ICP algorithm does not always converge to the correct solution due to its myopic nature. That is, it determines point to point correspondence based on minimum euclidean distance and does not incorporate the fact that most points are interconnected.

In this project, we tackle the problem of formulating a discriminative feature descriptor which considers the local geometry in a point's neighborhood and evaluate the improvements to ICP performance using this feature descriptor. We expect this work to be useful in technologies such as Virtual/Augmented Reality, 3D Reconstruction, self driving cars, and many other mobile robotics applications.

## II. Implementation

We have implemented classic ICP 1 as part of the course homework and found that there are challenges to finding the correct matches. For this project, our goal is to implement the Point Feature Histogram descriptor [1] in order to improve the correspondence search between two point clouds. A diagram showing our pipeline is in figure 1.

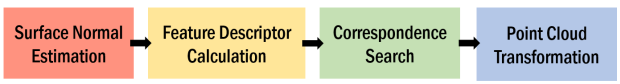



Fig. 1: Pipeline of our Point Feature Histogram implementation, given a source and target point cloud

### A. Vanilla Iterative Closest Point (ICP)

Our starting point is a vanilla variant of ICP algorithm (Algorithm 1 below) which computes point to point correspondences using minimum euclidean distance as a similarity metric. Principle Component Analysis (PCA) is used to incrementally compute transformations and move the source point cloud towards the target point cloud. The algorithm terminates when the sum of squared error between the transformed point cloud and the target point cloud is less than a threshold $\epsilon$. We additionally add terminating conditions due to lack of progress, and reaching maximum iterations.

**Algorithm 1** Vanilla Iterative Closest Point.

**Inputs**: P and Q, **Outputs**: R and t
**while** not Done **do**
  // Compute correspondences
  $C = \emptyset$
  **for all** $p_i \in P$ **do**
    find the closest $q_i \in Q$
    $C = C \cup \{p_i, q_i\}$
  **end for**
  // Compute Transforms
  $R, t \leftarrow$ **GetTransform**$(C_{pq}, Cq)$
  **if** $\sum_{i=1}^{n} \|RC_{pi} + t) - C_{qi}\|^2 < \epsilon$ **then**
    **return** $R, t$
  **end if**
  // Update all P
  **for all** $p_i \in P$ **do**
    $p_i = Rp_i + t$
  **end for**
**end while**

### B. Feature Descriptors

**PFH: Point Feature Histogram**

The vanilla ICP algorithm's weakness lies in its point to point matching, which does not consider the underlying geometry of a point's neighborhood. The Point Feature Histogram from [1] improves on this weakness by formulating a pose (position and orientation) invariant feature descriptor which generalizes a

point's neighboring surface geometric properties. The vanilla ICP method can then be modified to use the distance between histogram signatures as the similarity metric for matching, which should improve on the original metric. Of course, this comes at a computation cost of computing the histogram feature.

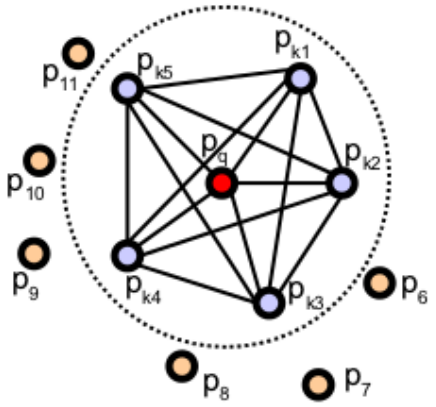



Fig. 2: PFH neighborhood visualization courtesy of [1], where $p_q$ has k neighbors $p_k$ within radius $r$.

To formulate the PFH descriptor, each 3D point must have an estimated surface normal associated with it. To estimate the surface normal, we find the $k$ nearest neighbors of the query point, build a 3x3 covariance matrix, and determine the normal by selecting the right singular vector associated with the smallest singular value. However, it is ambiguous as to which direction the normal vector points. To keep it consistent over the entire point cloud, we chose to orient all surface normals towards the viewpoint.

Now, let us consider a set of points $\mathcal{P}$ comprised of the query point $p_i$ and its $k$ neighbors $p_j$ within a given radius $r$. Each of these points has an associated surface normal which was estimated using PCA. For every unique pair of points in that set, features are defined. These combinations are shown in Figure 2 with a line connecting each pair. We will select a source $p_s$ and a target $p_t$ point, the source being the one having the smaller angle between the associated normal and the line connecting the points:

$$\begin{cases} p_s = p_i & \text{if } acos(\langle n_i, p_j - p_i \rangle) \leq acos(\langle n_j, p_i - p_j \rangle) \\ p_s = p_j & \text{otherwise} \end{cases}$$

Then, the Darboux frame [1] can be defined with the origin as the source point $p_s$:

$$\begin{aligned} u &= n_s \\ v &= (p_t - p_s) \times u \\ w &= u \times v \end{aligned} \tag{1}$$

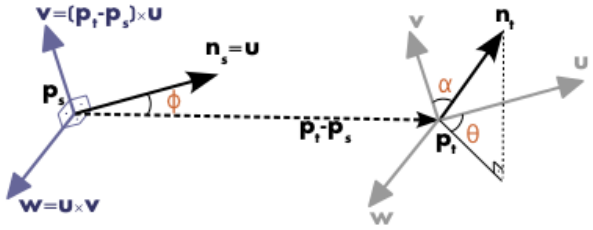



Fig. 3: Visualization courtesy of [1], where $\alpha, \phi, \theta$ represent $f_1, f_3$, and $f_4$.

Finally, the features $f = (f_1, f_2, f_3, f_4)$ can be defined and binned into a histogram of $div^4$ bins, where $div$ represents the number of subdivisions of each feature's value range. Figure 3 visualizes these features. In our implementation, we omit $f_2$ when binning the histogram because the distance between points increases as it gets farther from the sensor, rendering it uninformative. Thus our histogram is actually $div^3$ bins.

$$\begin{aligned} f_1 &= v \cdot n_t \\ f_2 &= \|p_t - p_s\| \\ f_3 &= u \cdot (p_t - p_s)/f_2 \\ f_4 &= atan(w \cdot n_t, u \cdot n_t) \end{aligned} \tag{2}$$

Another nuance of the histogram binning is that we are creating a 1-dimensional histogram from a multidimensional feature vector $f$. Our histogram will have $div^3$ bins, and Rusu provides an equation for determining which index of the histogram's bins to increment based on $f$'s divisions in [2]. The equation is as follows for $div = 2$:

$$idx = \sum_{i=1}^{i \leq 4} step(s_i, f_i) \cdot 2^{i-1} \tag{3}$$

$$step(s, f) = \begin{cases} 0 & \text{if} f < s \\ 1 & \text{otherwise} \end{cases} \tag{4}$$

**Similarity Metric for Correspondence Search**

The L2 norm between vectorized histogram signatures was used as a simple similarity metric for the correspondence search step of ICP. We found that this

produced accurate results on our test data. There are other distance metrics that could be used, but we did not explore these.

### C. Reducing Computational Complexity

The complexity of computing the PFH descriptor is $O(k^2)$. To reduce computation time of the PFH descriptor, we modified the classic PFH algorithm to reduce the number of points considered when computing the histogram signatures. This will improve the run time at the cost of less discriminative power. We explored two approaches, one method from our own intuition (Named Simple PFH) and one method from Rusu's work known as Fast PFH [3].

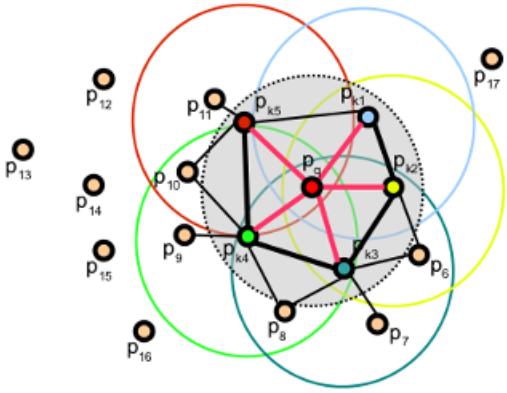



Fig. 4: Fast PFH neighborhood visualization, courtesy of [1].

#### Simple Point Feature Histogram

The classic point feature histogram descriptor looks at all combinations of pairs of points in $\mathcal{P}$. We decided to only consider the query point and its neighbors in order to reduce computation time. Computational complexity improves from $O(k^2)$ to $O(k)$, but since we have computed less feature vectors within a patch of points, the descriptor is less discriminative. In Figure 4, the red lines illustrate the pairs of points used to compute the feature descriptor.

#### Fast Point Feature Histogram

Our intuition to decrease the number of pairs of points used to calculate the feature descriptor to just the query point $p_q$'s neighbors actually aligns with Rusu's work in [3]. According to Rusu, doing just SPFH loses too much discriminative power. He further improves on the SPFH descriptor, while retaining complexity of $O(Nk)$ by introducing a weighting scheme as follows:

$$FPFH(p_q) = SPFH(p_q) + \frac{1}{k}\sum_{i=1}^{k}\frac{1}{w_k}\cdot SPFH(p_k)$$

Where $w_k$ represents the distance between query point $p_q$ and neighboring point $p_k$ in the given metric space. Figure 4 visualizes this weighting scheme, where the red lines represents SPFH of $p_q$, and the colored circles represent SPFH of $p_q$'s neighbors.

## III. Dataset

We used a combination of real and synthetic data for testing our algorithms. We started with the provided cup example with four different rotations and translations from homework as the synthetic dataset.

To test out some real world datasets, we used the 3DCOMET dataset [4]. We selected a plant data as seen in Figure 5 because of the complex geometries of its flowers and leaves. The plant was originally formatted as 73,354 point .pcd file. We downsampled using the PointCloudLibrary's VoxelGrid filter [5] to 959 points to reduce runtime. We then used the PyPCD library's [6] functions to import the .pcd files as numpy arrays.

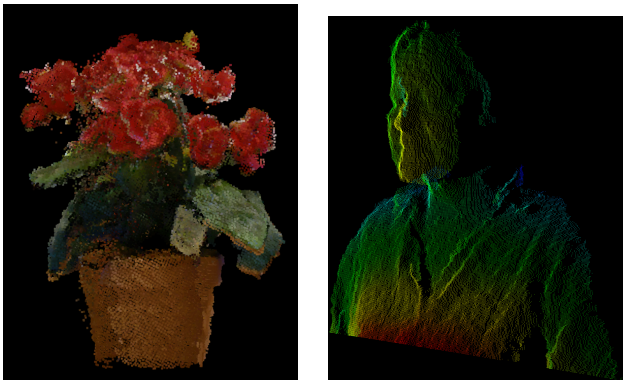
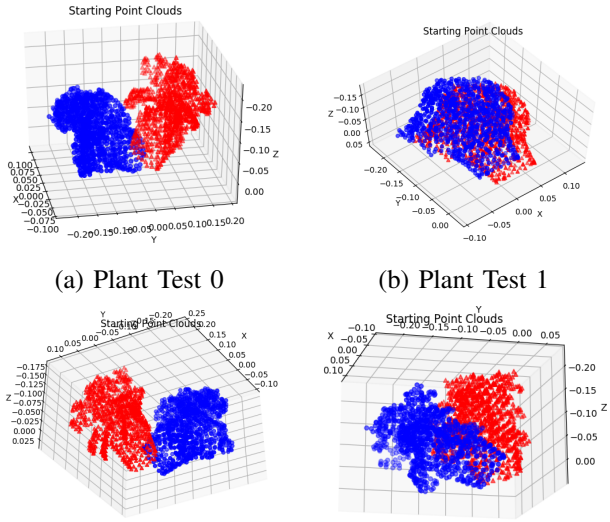


Fig. 5: 3D Pointcloud Plant and Face

Additionally, we selected some pointcloud data generated using a Microsoft Kinect sensor created for the ETH BIWI Facepose dataset [7]. We also rotated and translated the face data so that the target and source faces are facing each other and do not overlap. This will force the algorithm to produce a 180°rotation which we consider to be very challenging since it is much easier to mismatch correspondences. We similarily downsampled and converted this data to numpy array format. The starting point for ICP of the plant and face is displayed in Figure 6 and 7.




(a) Plant Test 0 (b) Plant Test 1

(c) Plant Test 2 (d) Plant Test 3

Fig. 6: Starting point of plant data

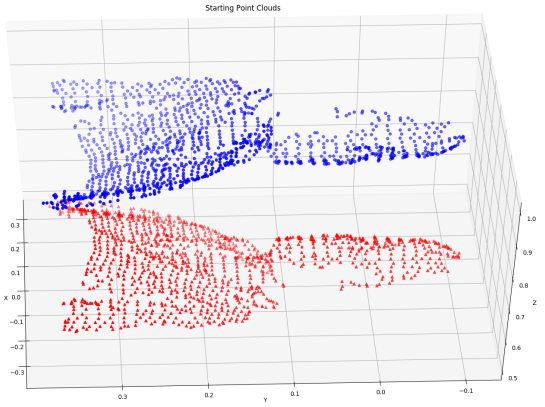



Fig. 7: Starting point of face data

## IV. Results

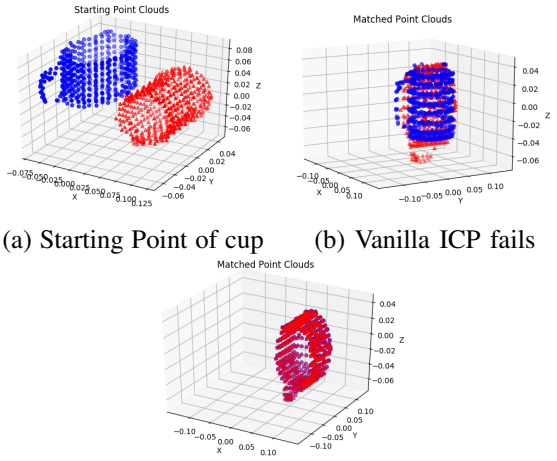
Our experiments were performanced on a 2017 Macbook Pro, using Ubuntu 16.04 in Parallels Virtual Machine. Four Cores and eight GB of RAM were allocated to the virtual machine. The software environment used to run is Python 2.7.12, numpy 1.11.0, matplotlib 2.2.3, scipy 0.17.0. The synthetic dataset used was the cup example provided in .csv format and did not require any preprocessing. The real world datasets were provided by the 3DCOMET [4] and the BIWI facepose dataset [7], and did require preprocessing. We used PointCloudLibrary 1.7.2 and PyPCD to pre-process our datasets by downsampling .pcd files and store them as numpy arrays.

### A. Parameter tuning

The parameters that must be tuned are $k$, $r$, $div$, and $\epsilon$. Where $k$ is the number of neighbors for estimating surface normals and for computing the feature histogram, $r$ is the radius of the sphere within which we find the $k$ neighbors, $div$ is the number of divisions for histogram binning, and $\epsilon$ is the error upon which ICP terminates. We referenced [1] to get a starting point, and then largely tuned these parameters by trial and error since they heavily depend on the dataset. Generally, the parameters for surface normal estimation depends on the overall size and density of the data. The histogram parameters depend on how precisely we wish to capture the surface geometry and there is a tradeoff between accuracy and computation time. $\epsilon$ determines the termination criteria, where there is a low error result. If the error does not change more than a fraction of $\epsilon$, we also terminate due to lack of progress. Overall, we found that $div = 5$ worked well in general for histogram feature descriptor, and $k = 12$ worked well for number of neighbors to consider for both surface normal estimation and feature descriptor calculation.

### B. Test on synthetic point clouds

We used the provided cup dataset as an initial test of our algorithm to prove that it works better than classic ICP. From the homework assignment, ICP was not able to correctly match Target 3 due to incorrect correspondence search. Figure 8 shows that our algorithm correctly matches Target 3 and is an improvement on classic ICP.



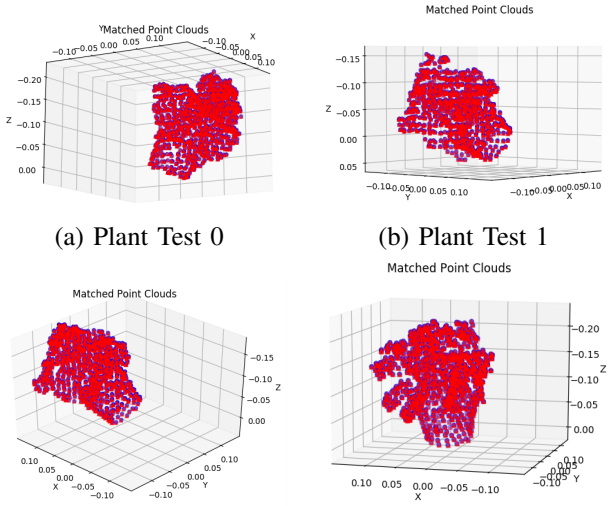

(a) Starting Point of cup (b) Vanilla ICP fails

(c) ICP with FPFH succeeds

Fig. 8: Comparing vanilla ICP with FPFH-ICP

### C. Test on real data

We evaluated the performance of classic PFH, SPFH, and FPFH quantitatively on the plant dataset [4] in terms of computation time per iteration, accuracy, and number of iterations. We additionally evaluate FPFH on the face dataset as a more realistic example. The starting point of plant dataset and face dataset can be seen in Figure 6 and Figure 7. The results using FPFH are visualized in Figure 9 and Figure 10.




(a) Plant Test 0 (b) Plant Test 1

(c) Plant Test 2 (d) Plant Test 3

Fig. 9: Output Poses of Plant using FPFH

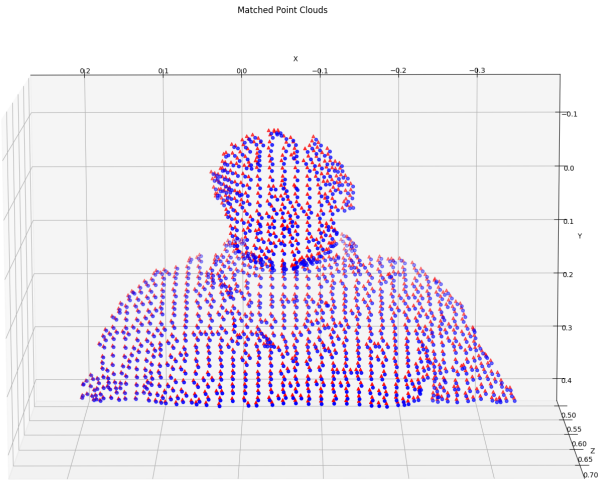



Fig. 10: Face matched

### D. Algorithm Performance

We evaluate the computation complexity, number of iterations, and final error for three different algorithms using the plant pointcloud dataset [4] with varying histogram bin sizes and number of neighbors. The results are reported in Table I below.

| **PFH** | time/iter. (s) | final error | iterations |
|---|---|---|---|
| 2 divs, 8 neighbors | 22.34 | 3.404 | MAX |
| 3 divs, 8 neighbors | 19.10 | 0.651 | 6 |
| 3 divs, 12 neighbors | 22.95 | 0.061 | 10 |
| 5 divs, 8 neighbors | 19.68 | 0.002 | 16 |
| 5 divs, 12 neighbors | 23.33 | 0.027 | 4 |
| **SPFH** | | | |
| 2 divs, 8 neighbors | 16.62 | 9.509 | MAX |
| 3 divs, 8 neighbors | 16.62 | 7.902 | MAX |
| 3 divs, 12 neighbors | 17.80 | 3.016 | MAX |
| 5 divs, 8 neighbors | 17.07 | 0.177 | MAX |
| 5 divs, 12 neighbors | 19.53 | 0.008 | 9 |
| **FPFH** | | | |
| 2 divs, 8 neighbors | 17.25 | 2.879 | 4 |
| 3 divs, 8 neighbors | 16.55 | 2.711 | 3 |
| 3 divs, 12 neighbors | 17.17 | .976 | 4 |
| 5 divs, 8 neighbors | 17.48 | 0.074 | 3 |
| 5 divs, 12 neighbors | 17.06 | 0.00687 | 2 |

TABLE I: Table showing performance on plant data

The face dataset represents a real world application in 3D facial recognition, similiar to Apple FaceID. For this reason, we also evaluated the face dataset [7] with only FPFH. The results are in below Table II. The error and number of iterations improve when increasing the number of $divs$ and neighbors.

| **FPFH** | time/iter. (s) | final error | Iterations |
|---|---|---|---|
| 2 divs, 8 neighbors | 31.48 | 27.334 | 11 |
| 3 divs, 8 neighbors | 28.09 | 42.267 | 4 |
| 3 divs, 12 neighbors | 29.13 | 39.558 | 5 |
| 4 divs, 8 neighbors | 29.01 | 18.116 | 6 |
| 4 divs, 12 neighbors | 30.05 | 9.318 | 5 |

TABLE II: Table showing FPFH performance on face data

We found that in terms of computation time, SPFH and FPFH outperforms PFH. This makes sense because PFH is $O(k^2)$ while SPFH and FPFH are both linear time. In terms of error, PFH had the best performance as it had low error with as few as 3 $divs$ and 8 neighbors. As we increase $divs$ and neighbors, SPFH and FPFH catch up to PFH. In terms of number of iterations, SPFH had the worst performance which shows it is the least effective feature descriptor. For FPFH, the iterations are low because ICP terminates due to lack of progress while PFH does not. Thus it's hard to rank PFH and FPFH in terms of the number of iterations. Generally, the fewer the bins and number of neighbors, the more likely to reach max

number of iterations without finding optimal solution. Also, varying the number of bins and neighbors for a given algorithm does not seem to significantly affect computation time on our dataset.

Overall, the complexity of PFH is $O(k^2)$ and the complexity of SPFH and FPFH are both $O(Nk)$ where $N$ represents the total number of points in the point cloud. The nearest neighborhood search is the most time consuming part in each algorithm, and can be improved by k-d tree search in the future. This will be very important if the dataset is larger, and the search space increases. For context, modern laser rangefinders can generate in excess of 2 million points per second, the datasets we were operating on were in the scale of roughly 1000 points.

## V. Conclusion

We have implemented the Point Feature Histogram, Fast Point Feature Histogram, Simple Point Feature Histogram, in hopes of improving upon the vanilla ICP algorithm's accuracy. We evaluated our implementation on several datasets and found that all 3 variants were successful in improving the algorithm's performance. We believe point cloud processing will play a key role in robotics with recent technology advancements in depth sensors and as the cost of such sensors decreases. We would like to thank Vince Gong and Professor Dmitry Berenson for their guidance and enthusiasm in EECS 498.

## References

[1] R. B. Rusu, "Semantic 3d object maps for everyday manipulation in human living environments," *KI - Künstliche Intelligenz*, vol. 24, pp. 345–348, 2009.

[2] R. B. Rusu, Z. C. Marton, and N. Blodow, "Persistent point feature histograms for 3 d point clouds," 2008.

[3] R. B. Rusu, N. Blodow, and M. Beetz, "Fast point feature histograms (fpfh) for 3d registration," *2009 IEEE International Conference on Robotics and Automation*, pp. 3212–3217, 2009.

[4] J. Navarrete, V. Morell, M. Cazorla, D. Viejo, J. Garca-Rodrguez, and S. Orts-Escolano, "3dcomet: 3d compression methods test dataset," *Robotics and Autonomous Systems*, vol. 75, p. 550557, 2016.

[5] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, (Shanghai, China), May 9-13 2011.

[6] D. Maturana, "Python pcd." https://github.com/dimatura/pypcd, 2015-2017.

[7] G. Fanelli, M. Dantone, J. Gall, A. Fossati, and L. Van Gool, "Random forests for real time 3d face analysis," *Int. J. Comput. Vision*, vol. 101, pp. 437–458, February 2013.