Obstacle Detection using DBSCAN Clustering on Monocular SLAM Generated Map Points

MAE 199: Independent Study, End-of-Quarter Report

Gerardo Gonzalez and Shiyuan Huang

Spring 2016

Contents

1 Summary

The objective for this project is to develop a method for object detection using the ORB SLAM map point data. In order to meet this functionality, we decided on an algorithm that could process large spatial data sets and output clusters from it. These clusters can then be represented as obstacles by the robotic agents. Furthermore, we need to take into account the high noise data that ORB SLAM produces. Therefore, a large part of this project was focused on finding a reasonable algorithm that provided clustering in a high noise environment. After researching clustering algorithms, we decided to go with density-based spatial clustering of applications with noise (DBSCAN) because of its noise handling functionality and pseudocode availability. This quarter, we finalized the C++ and ROS development for the main algorithm. Furthermore, we focused on testing and optimization, resulting in the addition of features to the algorithm. In this paper, we will detail the testing process used, along with a performance analysis for the newly added features.

2 Big Picture

The goal of the lab is to develop and test distributed control algorithms on multi-agent robot systems for disaster response operations. One such operation is the navigation and search of unspecified indoor environments, which could be characterized by obstacles, including walls and doors. Thus, in order to avoid mission errors, such as collisions, it is necessary for these robot systems to be able to detect the obstacles in order to navigate around them.

The present method for achieving this is built around the premise that the obstacle is a static agent. Therefore a Voronoi cell is computed for the obstacle, in which case all other agents do not interfere with the obstacle's cell. However, with our current setup, the generation of an obstacle is limited by user input. The proposed idea is to identify the clusters found in the map points, already generated by ORB SLAM, as obstacles, therefore eliminating the need to integrate more expensive obstacle detection software with the system. One method which we have employed in this work is to use DBSCAN to first identify the clusters, and then find the vertices that define the convex hull of these clusters to ultimately define the obstacles.

3 Preliminaries

3.1 ORB-SLAM

ORB SLAM is the simultaneous localization and mapping (SLAM) algorithm that we will be using to localize our quadrotors. This SLAM algorithm is of interest to us due to its *survival of the fittest* approach to frame processing, resulting in robust and track-able maps that change only when the environment being mapped changes [3]. This characteristic is vital for our project, which requires the robot network to be able to search over areas of importance, including the inside of a building. Thus, the map (point cloud) ORB SLAM returns in a situation like this could be used to detect obstacles, in addition to helping the quadrotor localize. Moreover, ORB SLAM is specifically designed to work for monocular systems, which is the current camera setup for our quadrotors. Finally, ORB SLAM is fully open source, facilitating the development process for the project.

3.2 DBSCAN

DBSCAN is an algorithm whose acronym stands for density-based spatial clustering of applications with noise. This algorithm was proposed by Martin Ester, Hans-Peter Kriegel, Jörg Sander and Xiaowei Xu in 1996, see [1]. The main functionality of the DBSCAN algorithm is its ability to group spatial data points in proximity of each other, which we define as *neighbors*, and categorize them into a specific cluster, while categorizing outlier points (points not part of the clusters) as noise. DBSCAN has both its advantages and disadvantages. An important advantage is that it does not require any previous knowledge about the number of clusters in a group of data points. Also, DBSCAN only requires two parameters (ϵ and n_{min})

for controlling the outcome of the cluster arrangements (this will be explained later). In addition, arbitrary cluster shapes and noise handling are possible. There are, however, some disadvantages, one of which is the difficulty in choosing the parameter ϵ. If the data is not well understood, clusters with largely disparate densities may be difficult to cluster since only one parameter for density is used for the entire data set.

The following definitions will be of importance in understanding the DBSCAN algorithm.

Definition 1. The Eps-neighborhood, N_ϵ, of a point x in a data set, D, is shown below in equation 1, as defined in [1].

$$N_\epsilon(x) = \{s \in D | d(x, s) \leq \epsilon\} \tag{1}$$

.

Definition 2. We say that a point x is *directly-density-reachable* with respect to ϵ and n_{min} to another point s if both x is an element of the set of points which define the Eps-neighborhood of s and the number of points in the Eps-neighborhood of s is greater than or equal to n_{min}. To clarify n_{min} is simply the minimum number of points that can be considered a cluster, see [1].

Definition 3. We say that a point x is *density connected* to a point s with respect to ϵ and n_{min} if there exists another point, t, such that both x and s are each individually density-reachable to t (again with respect to ϵ and n_{min}), see [1].

Now we can now introduce the notion of a cluster in DBSCAN. A *cluster* in DBSCAN is simply a group of points that have at least n_{min} points and each of their respective points, from the data set of points, D, are connected by density connections as described in definition 3. Additionally, any points which are not considered a part of any cluster (those who either do not meet the n_{min} requirement or the Eps-neighborhood requirement) are considered noise, see [1].

Below is shown a pseudocode for the DBSCAN algorithm. Different sections of this algorithm will be explained in detail later in the report.

```
DBSCAN(D, eps, MinPts) {
   C = 0
   for each point P in dataset D {
      if P is visited
         continue next point
      mark P as visited
      NeighborPts = regionQuery(P, eps)
      if sizeof(NeighborPts) < MinPts
         mark P as NOISE
      else {
         C = next cluster
         expandCluster(P, NeighborPts, C, eps, MinPts)
      }
   }
}
expandCluster(P, NeighborPts, C, eps, MinPts) {
   add P to cluster C
   for each point P' in NeighborPts {
      if P' is not visited {
         mark P' as visited
         NeighborPts' = regionQuery. (P', eps)
         if sizeof( NeighborPts') >= MinPts
            NeighborPts = NeighborPts joined with NeighborPts'
      }
      if P’ is not yet member of any cluster
         add P’ to cluster C
   }
}
regionQuery(P, eps)
   return all points within P’s eps-neighborhood (including P)
```

3.3 Convex Hull

By using the DBSCAN to identify clusters, obstacles are able to be identified as the separate clusters themselves. However, it would be wasteful to pass along the entire set of points that belong to each cluster every time a cluster is found. Instead, to minimize the amount of information passed, the *convex hull* of the cluster is used to redefine the obstacle.

By definition, the convex hull is the smallest set for a particular set of points that is convex. For instance, consider the group or cluster of points which all together look like a triangle in figure 1. The convex hull of these points contains all the points while ensuring convexity. This in essence would return the perimeter of the triangle and the square as shown in figure 2.

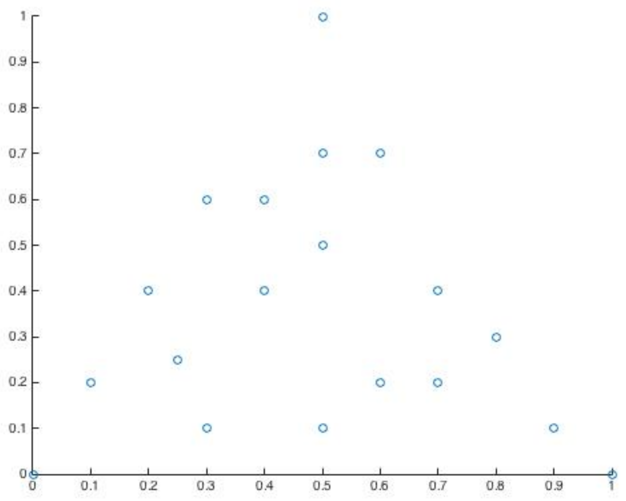


Figure 1: Cluster of Points in a Triangle

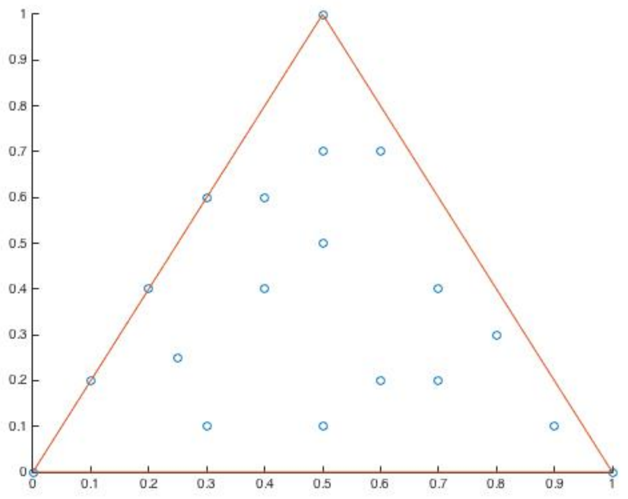


Figure 2: Convex Hull of Points in Triangle

However, it is unnecessary to define the line for the convex hull, and as such we can simply use the following three points to define the entire cluster, i.e by the edges of the convex hull as shown in figure 3

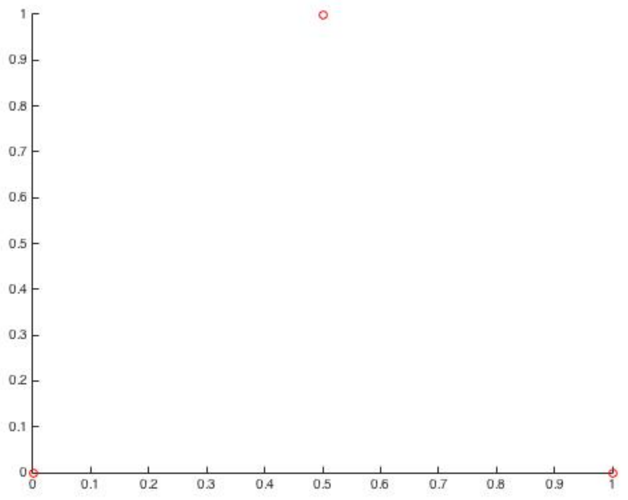


Figure 3: Edges of the Convex Hull

Thus, in order to reduce the information passed, it is only necessary to pass the edges of the cluster's convex hull perimeter. In the case of the triangle, we can reduce the information passed to define the obstacle with the points that define the edges of the perimeter. This describes conceptually the *Graham Scan* algorithm or any convex hull algorithm for that matter.

4 Our Contributions

4.1 ROS Integration for DBSCAN and Convex Hull - Gerardo Gonzalez

This quarter, I was able to finalize the ROS integration for the DBSCAN and convex hull algorithms. The obstacle detection algorithm is as follows:

```
int C = 0; // Cluster number
double epsilon = 0.06; // Distance threshold that classifies a "Neighbor"
int minPts = 20; // Size threshold that classifies a cluster

while (ros::ok())
{
  ros::spinOnce();
  if(CAM_STATE && MAP_STATE) // Check if ready to cluster
  {
    // Reset flags
    MAP_STATE = false;
    CAM_STATE = false;

    /* DBSCAN */
    double X[noMapPts][3];
    int IDX[noMapPts];
    init_X(X, noMapPts); // Initialize input array for DBSCAN

    cout << "Starting DBSCAN.." << endl;
    C = DBSCAN(X, epsilon, minPts, noMapPts, IDX); // cluster number
    cout << "DBSCAN complete." << endl;

    /* Convex Hull */
    geometry_msgs::PoseArray clusterVertices[C]; // holds vertices for all clusters

    cout << "Starting Convex Hull." << endl;
    getConvexHull(C, IDX, X, noMapPts, clusterVertices);
    cout << "Convex Hull complete." << endl;

    //printClusterVertices(clusterVertices, C);
    printToFile(X, IDX, clusterVertices, C); // Log output
    //vertices_pub.publish(TODO);
  }
```

The algorithm begins by initializing key parameters. The first one is the *cluster number*, which is the id used to designate a point to a cluster. The next two parameters are *epsilon* and *minPts*, where *epsilon* characterizes the distance threshold required for two points to be considered directly-density-reachable and *minPts* is the minimum number of points required. For now, both of these parameters are chosen by the user, with the current parameters being set after running multiple tests with different values and picking the ones that return the most accurate clusters.
The main ROS loop begins by doing a spin call. This spin call updates the *CAM_STATE* and *MAP_STATE* flags, along with the most up to date copy of the ORB SLAM map points. The *CAM_STATE* flag is set to true when the camera position is updated. Similarly, the *MAP_STATE* flag is set to true when the map points are updated. If both of these parameters are set to true, then DBSCAN and convex hull are executed. For DBSCAN, the 2D variable *X* is first initialized to the ORB SLAM map points and then passed into the DBSCAN method as a parameter, along with *epsilon* and *minPts*, and with the additional *noMapPts* and *IDX*, where *noMapPts* holds the total number of map points and *IDX* holds the cluster id for each map point. The DBSCAN method then updates *IDX* accordingly and returns the cluster number *C*.
This cluster number is then utilized to initialize *clusterVertices*, which is an array of type PoseArray that will be used to hold the convex hull vertices for every cluster. These vertices are computed in the *getConvexHull* method. Finally, the algorithm calls the *printToFile* method to output all of the data, including map points, clusters, and convex hull vertices, to a text file to be further processed by a MATLAB script.

At each spin call, the algorithm updates the current camera position and map points. For the camera position, the following callback is used:

```

/* This callback function receives camera position data from ORB SLAM */
void camPoseCallback(const tf2_msgs::TFMessage::ConstPtr& camPose)
{
  if(camPose->transforms[0].header.frame_id.compare("ORB_SLAM/World") == CAM_POSE_ID)
  {
    CAM_STATE = true;
    upperWin = camPose->transforms[0].transform.translation.y - epsWin;
    lowerWin = camPose->transforms[0].transform.translation.y + epsWin;
    cout << "Camera position updated!\n";
  }
  else cout << "Camera position not updated.\n";
}
```

This callback function is subscribed to the *tf* topic and receives messages of type *TFMESSAGE*. First, the frame id for the message is checked to make sure it matches the *CAM_POSE_ID*, which holds the id designated for the camera position. If this is true, then the *CAM_STATE* is set to true and the variables *upperWin* and *lowerWin* are updated. These variables are used to define the height window that filters the map point data.

4.1.1 Map Point Filter

The map point filter is one of the new features developed this quarter and is aimed at improving clustering accuracy and efficiency by reducing the number of map points based on the current position of the camera. Both of these variables are updated by taking the y-dimension (using OpenCV coordinate system for images) and adjusting it by *epsWin*, which is the distance from the midpoint to the perimeter along the y-dimension that defines the boundary of the height filter, also known as the height window. This *epsWin* is obtained using the following formula:

$$epsWin = \frac{\text{height of robot agent}}{2} \tag{2}$$

In other words, we can think of the height window as filtering out all points along the y-dimension that do not lie within the window. Analyzing the map point data that ORB SLAM generates, the features that define the map points are constantly spawned as the camera changes position and field of view. Thus, the map points that are returned are a combination of old and new map points, with a chance that the old ones might not be in the robot's field of view anymore, which are no longer useful in determining a best path. Therefore, the expectation is that this new feature will reduce the running time and improve accuracy of clustering by running only on a subset of the map points, those that constitute objects that would interfere with the robot's navigation.

Finally, a different callback function is used to update the map points. The following code illustrates the process:

```
/* This callback function receives map point messages from ORB SLAM. */
void mapPointsCallback(const visualization_msgs::Marker::ConstPtr& mapPtsMsg)
{
  if(mapPtsMsg->id == MAP_ID)
  {
    MAP_STATE = true; // set flag to update map points
    mapPts = *mapPtsMsg; // Store message
    fullMapPts = mapPts;

    // Filter map points based on window size
    for(int i = 0; i < mapPts.points.size(); )
    {
      if(mapPts.points[i].y < upperWin || mapPts.points[i].y > lowerWin)
        mapPts.points.erase(mapPts.points.begin() + i); // erase ith element
      else i++;
    }
    noMapPts = mapPts.points.size();

    cout << "Map points updated!\n" << endl;
  }
  else cout << "Map points not updated.\n";
}
```

This callback function subscribes to the topic *ORB_SLAM/Map* and takes in as an argument a message of type *visualization_msgs::Marker* and variable name *mapPtsMsg*. This message is first pre-processed to see if it contains map point data, since the *ORB_SLAM/Map* topic receives messages of the same type but with different encoded data. Thus, the message id is checked and if the conditional returns true, then the *MAP_STATE* flag is set to true and the message is copied to *mapPts* and *fullMapPts*, where *mapPts* holds the filtered map points and *fullMapPts* holds the original data in its entirety. Next, the map points are filtered by looping over each point and checking whether its y-dimension falls outside of the window range. If a point is found to do so, then it is erased from the collection of map points.

4.2 Parameter Tuning - Shiyuan Huang

When using DBSCAN to cluster points, it is important to decide on parameters, eps and MinPts, to get good clustering results. For this quarter, I've been working on parameter tuning by revising the DBSCAN algorithm. A good way to choose eps is to use k-distance method and VDBSCAN. The code is as follows:

```
MinPts = k;
[ID, D] = knnsearch(XZ,XZ,'k',k);

for i = 1 : size(XZ,1)
dist(i) = sum(D(i,:));
end

avgdist = sort(dist/(k-1));

for i = 1:1:length(avgdist)-1
   slope(i)= avgdist(i+1)-avgdist(i);
end

for i = 1:length(slope)-1
   if slope (i+1) >= 1.1*slope(i)
      epsilon(i) = avgdist(i);
   end
end
```

Before VDBSCAN, I generate a set of eps candidates by calculating the average distances from each point to its k nearest neighbors, where k is equal to MinPts chosen by the knowledge of domain. After sorting those distances, I calculate the slope between each two adjacent points. Epsilon candidates are set to be equal to the distances at which a "jump"(huge change of slope) occurs. Here I define the "jump" to be where the slope is increased by more than 10 percent. After I get eps candidates, I use VDBSCAN to assign cluster index to each point. VDBSCAN code is just a revision of DBSCAN. The revised part of the Matlab code is as follows:

```
function [IDX, isnoise,S,T]=DBSCAN(X,epsilon,MinPts)
    n=size(X,1);
    IDX=zeros(n,2);
    D=pdist2(X,X);
    isnoise=false(n,1);
    visited=false(n,1);
    epsC = 1;

    for m = 1:length(epsilon)
        C = 0;
        visited=false(n,1);
        isnoise=false(n,1);
        if epsilon(m)~=0
            for i=1:n

                if ~visited(i) && IDX(i,2)== 0
                    visited(i)=true;

                    Neighbors=RegionQuery(i,m);
                    if numel(Neighbors)<MinPts
                        % X(i,:) is NOISE
                        isnoise(i)=true;
                    else
                        C=C+1;
                        ExpandCluster(i,Neighbors,C,epsC,m);
                    end
                end
            end
            if size(find(IDX(:,2)== epsC),1) >= MinPts
                T(epsC) = size(find(IDX(:,2)== epsC),1);
                epsC = epsC + 1;
                S(epsC) = epsilon(m);
            end
        end
    end
```

Different from the original DBSCAN, each point is now assigned a two-index array. A point whose IDX is (i,j) means that it belongs to the i-th cluster that is characterized by the j-th eps. By jumping over the eps that form unqualified clusters whose size is less than MinPts, we will get a clustering with multi-densities.

5 Methodology

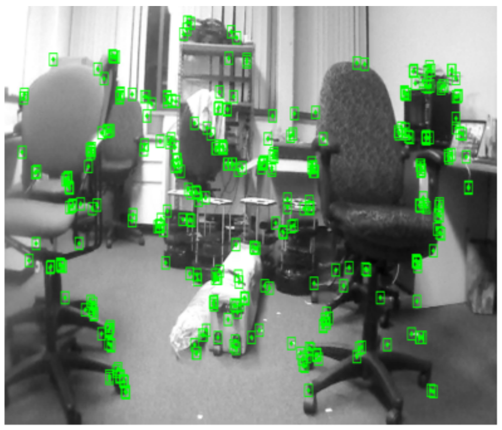
5.1 Test Guide

In order to test the algorithm, make sure to first install ROS, ORB SLAM, the ARDRONE autonomy package, and the Github repository for the lab (make sure to check our Github wiki for more information on these installations). Before continuing, make sure that to be on the *ObstacleDetection* branch. In this branch, there are two main directories used: *obstacle_detection* and *quadcopter/test_pkg* . The *obstacle_detection* directory contains scripts for visualizing clustered data and running the DBSCAN algorithm, both in MATLAB. In addition, this is where we store ORB SLAM map point datasets for future testing. On the other hand, the *quadcopter/test_pkg* is a ROS package that holds launch and source files for ROS nodes. As of now, the algorithm source is in a single ROS node. Its path, **from "ucsd_ros_project"**, is the following: *quadcopter/test_pkg/src/ObstacleDetection.cpp* . To run all of the necessary nodes, it is recommended to use the following launch file: *quadcopter/test_pkg/launch/obstacleDetection.launch.* This launch file will run the ardrone driver, ORB SLAM, and the obstacle detection node. **Before launching**, make sure that the machine being used to test is connected to the drone network and that the ROS parameters are appropriately configured. Now you are ready to test the clustering algorithm. Enjoy!

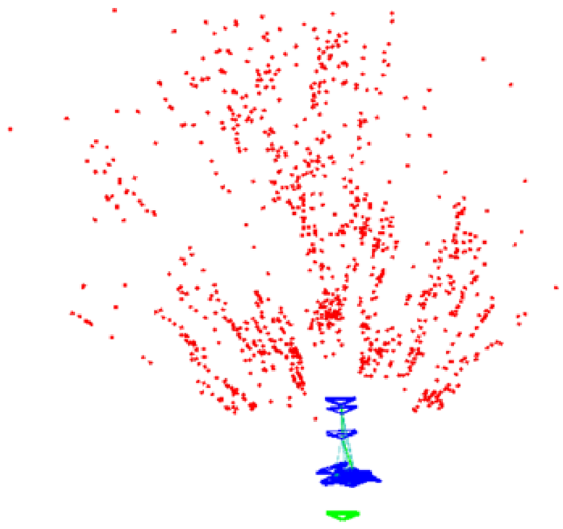
5.2 Test Results

Utilizing the setup described above, we tested the obstacle detection algorithm to promising results. One of our first testing environments was the lab room, which provided object rich frames to test on. The results are as follows:

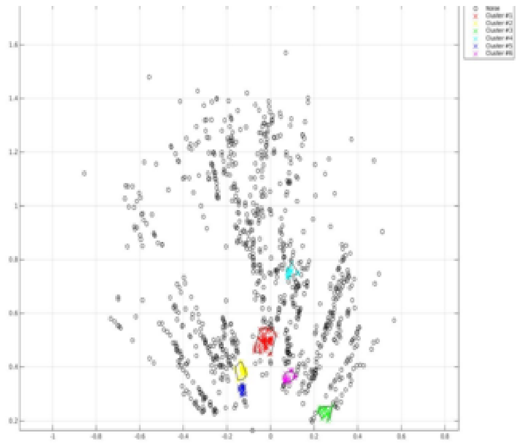
(a) Test Environment



(b) ORB SLAM Features



(c) RVIZ Plot

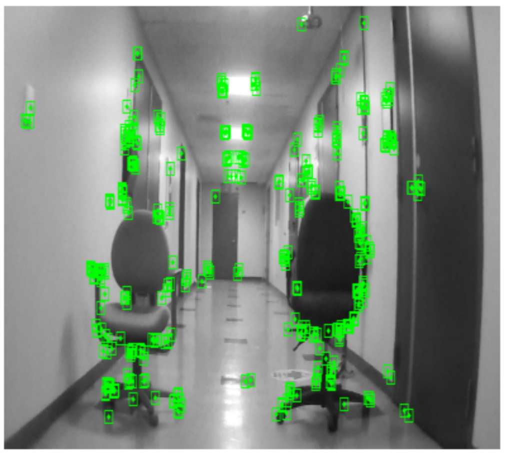


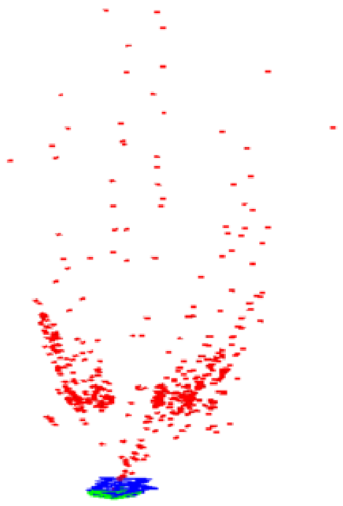
(d) Cluster and Object Visualization in Matlab

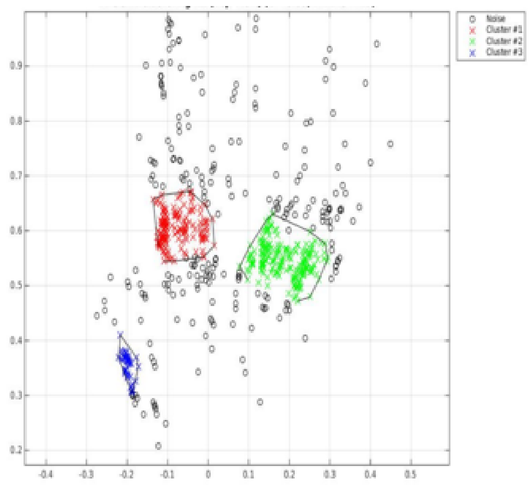
Figure 4: Test Results for Lab Environment

Figure 4a references the testing environment, which was the lab room for this test session. Next, figure 4b illustrates the features that ORB SLAM tracks in an image. These are again displayed in 4c, which shows a 3D model of the map points. Finally, figure 4d is the visualization of the final output for the DBSCAN and convex hull algorithms. Analyzing the results, we can see that, although the algorithm returns clusters and their convex hull as expected, the clusters are not representative of the rich set of objects that makeup the environment. In an effort to improve the accuracy of the clustering, we reduced the number of objects in the environment being mapped. Here are the results:

(a) Test Environment


(b) ORB SLAM Features


(c) RVIZ Plot



(d) Cluster and Object Visualization in Matlab

Figure 5: Test Results for Two Obstacle Environment

From 5d we can see that the algorithm accurately clusters the two objects in the frame while disregarding noise from the surrounding walls and floor. From this we can conclude that the algorithm works appropriately within certain constraints. It is important to consider the parameters and the process for choosing them because they will define the size and shape of the objects that will best be tracked by the algorithm. As a result, it failed to give any meaningful information in the first test case due to the differently sized and shaped objects resulting in map points with varying densities. Thus, the algorithm was not appropriate for this data set. Indeed, a major pitfall in the DBSCAN algorithm is in its need of a fixed parameter set. For this reason, we are investigating a variation of the DBSCAN, aptly named Varied-DBSCAN (VDBSCAN), to resolve this special case [6]. Nonetheless, the algorithm works as expected and we will move forward to study the conditions under which it will give us the best performance.

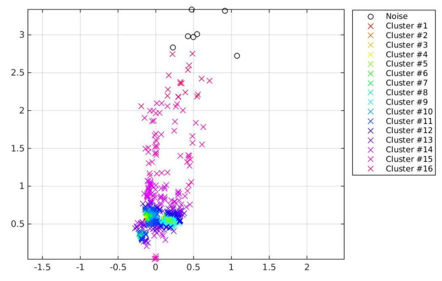
6 Tips

6.1 Using MATLAB to Visualize Data

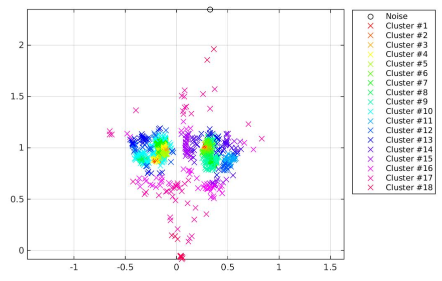
In order to test the performance of the algorithm, it was necessary for us to be able to visualize the map points generated by ORB SLAM, along with the clusters given by DBSCAN and the convex hull points returned by the Graham scan algorithm. For this reason, we developed a script in MATLAB that reads in the map points, the map point cluster IDs, and hull points and plots them. In order to use this script, make sure to first update the data being read by the script. For this purpose, we have conveniently created a Makefile that moves all the necessary files to the directory where the MATLAB script is held. After recording new data, change to the following directory: *ucsd_ros_project/quadcopter/test_pkg/src*. Once in this directory, enter "make" into the command line and you're good to go! The MATLAB script can be found in *ucsd_ros_project/obstacle_detection/dbscan_test.m* .

6.2 VDBSCAN on Some Data Sets

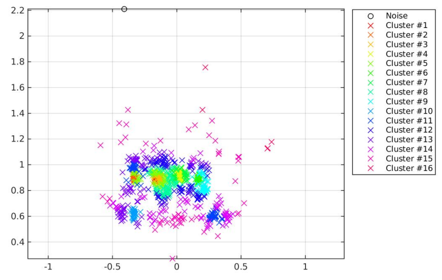
VDBSCAN gives multi-density clustering. We took the bird view data sets of two chairs, two turtlebots, and a desktop. Matlab visualization of the data sets are as follows:



(a) Two Chairs



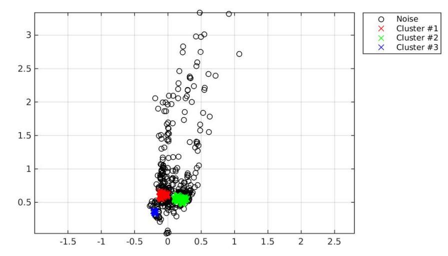
(b) Two Turtlebots



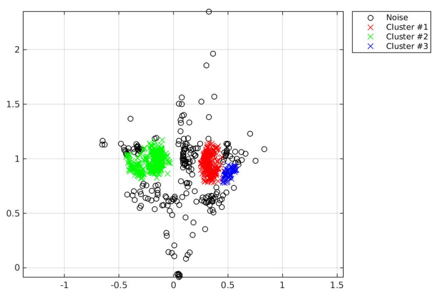
(c) Desktop

Figure 6: Multi-density Clustering

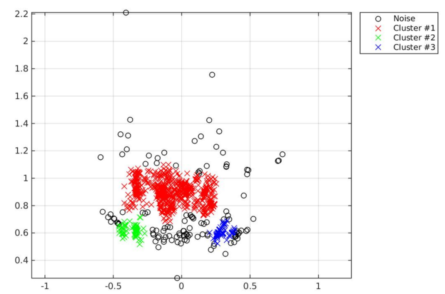
Almost each point belongs to some cluster. The green areas give approximately the shape of the obstacles. However, considering the fact that we need the convex hull coordinates for the most dense parts, it's better to choose one best eps to get only the obstacles being clustered. In order to achieve that, we draw a scatter point plot for the size of clusters at each eps . Use linear approximation to find the last eps that fits the same line with the previous points, we use this eps as our "best eps" parameter for DBSCAN. The matlab visualization for the clusters are as follows:



(a) Two Chairs, eps = 0.0343



(b) Two Turtlebots, eps = 0.0584



(c) Desktop, eps = 0.0739

Figure 7: Best eps Clustering

We can see that the eps chosen from the k-distance method gives a clear obstacle clustering.

7 Conclusions and Future Directions

7.1 Android Integration and Flight Test

Future directions include the testing of the algorithm in a flight simulation. To accomplish this, we will need to revisit and tune the autonomous controls for the quadrotors to make sure the flight is stable. Furthermore, we are interested in developing an Android feature that will allow users to visualize quadcopter-defined obstacles in real-time. The idea is to add a button in the application that, when pressed by the user, will run the algorithm and display the obstacles, represented as the polygons formed when the hull points for a cluster are connected.

References

[1] Martin Ester, Hans-Peter Kriegel, Jorg Sanders, and Xiaowei Xu. *A Density Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise*. Institute for Computer Science, Munich Germany, 1996.

[2] *DBSCAN* Wikipedia. https://en.wikipedia.org/wiki/DBSCAN

[3] Raúl Mur-Artal, J. M. M. Montiel and Juan D. Tardós. *ORB-SLAM: A Versatile and Accurate Monocular SLAM System.*. IEEE Transactions on Robotics, vol. 31, no. 5, pp. 1147-1163, October 2015.

[4] Yarpiz *DBSCAN Clustering Algorithm*. MATLAB File Exchange, 06 Sep 2015.

[5] Graham Scan Algorithm
https://github.com/kartikkukreja/blog-codes/blob/master/src/Graham%20Scan%20Convex%20Hull.cpp

[6] Peng Liu, Dong Zhou, and Naijun Wu. *VDBSCAN: Varied Density Based Spatial Clustering of Applications with Noise*. School of Information Management and Engineering, Shanghai University of Finance and Economics, Shanghai, 200433, China.

[7] Gerardo Gonzalez, Julio Martinez and Shiyuan Huang. *Cluster Assignment Using DBSCAN and ORB-SLAM for ObstacleDetection*. Jacobs School of Engineering, University of California, San Diego, March 2016.