Cluster Assignment Using DBSCAN and ORB-SLAM for Obstacle Detection

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

Julio Martinez, Gerardo Gonzalez, and Shiyuan Huang

Winter 2016

Contents

1 Summary

The objective for this project is to develop a method for object detection using the ORB SLAM point cloud 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 worked on finishing the development of the algorithm in C++ and ROS. By the end of the quarter, we were able to test the C++ version of the code on obstacle data, with promising results.

2 Big Picture

The present method for moving around an obstacle is built around the premise that the obstacle is a static agent. Therefore a Voronoi cell is computed for the obstacle as is the case for any agent, in which case, all other agents do not interfere with the static agent's (the obstacle's) cell. However, currently this is limited to input by the user. Because of the 3-dimensional point cloud generated by ORB SLAM, user input may be significantly reduced if not eliminated if detection of the objects is enabled through the use of the image data generated by ORB SLAM. The proposed idea is to identify the clusters found in the data as obstacles. 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(The code for the callback is sown below:

\begin{lstlisting}
/* This callback function subscribes to an ORB SLAM topic and receives point cloud data messages. */
void mapPointCloudCallback(const visualization_msgs::Marker::ConstPtr& mapPointCloudMsg)
{
    if(mapPointCloudMsg->id==MAP_POINT_CLOUD_ID)
        MAP_FLAG=true; // set flag to update point cloud

    if(MAP_FLAG)
```

```
    {
      MAP_FLAG = false; // reset flag
      CLUSTER_FLAG = true; // cluster the point cloud
      pointCloud = *mapPointCloudMsg; // Store message
      ROS_INFO("Wrote to file... Size = %d\n", pointCloud.points.size()); // Stdout Message
    }
}
\end{lstlisting}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 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.

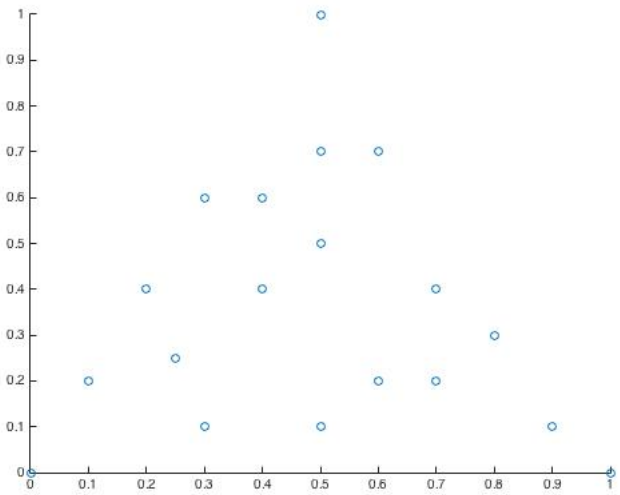


Figure 1: Cluster of Points in a Triangel

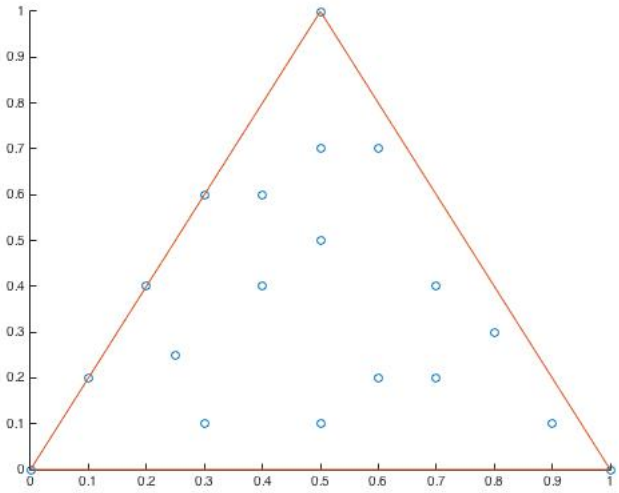


Figure 2: Convex Hull of Points in Triangle

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

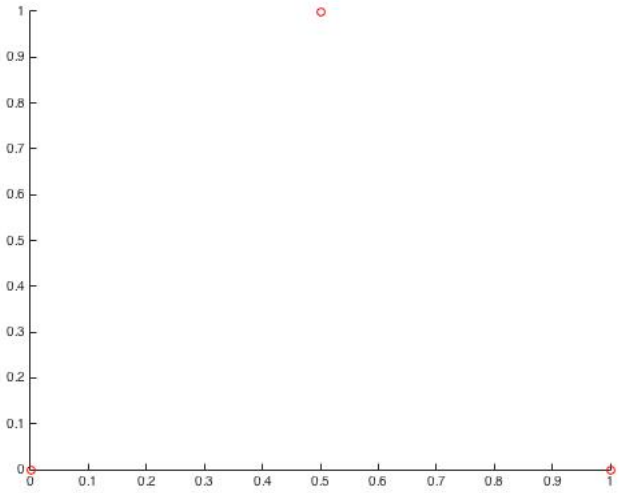


Figure 3: Edges of the Convex Hull

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 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 Development - Gerardo Gonzalez

This quarter, I focused on integrating our C++ version of the DBSCAN algorithm with the ROS platform. The first thing to figure out for this project was how the ORB SLAM point cloud data was going to be transferred to the obstacle detection node. In the ORB SLAM github repository, it is specified that the point cloud data is published to the topic "ORB_SLAM/Map" using a visualization_msgs::Marker message type with message id 6. The code for the callback is shown below:

```
/* This callback function subscribes to an ORB SLAM topic and receives point cloud data
    messages. */
void mapPointCloudCallback(const visualization_msgs::Marker::ConstPtr& mapPointCloudMsg)
{
    if(mapPointCloudMsg->id==MAP_POINT_CLOUD_ID)
        MAP_FLAG=true; // set flag to update point cloud

    if(MAP_FLAG)
    {
      MAP_FLAG = false; // reset flag
      CLUSTER_FLAG = true; // cluster the point cloud
      pointCloud = *mapPointCloudMsg; // Store message
      ROS_INFO("Wrote to file... Size = %d\n", pointCloud.points.size()); // Stdout Message
    }
}
```

In this callback function, we first check to make sure that the message has the correct ID type and set a flag for it. If the flag is set to true, then we copy the message over to a global variable and output a checkpoint message to stdout. By the end of this callback function, we will have the most up to date point cloud.
The next step is designing the main ROS loop, outlined in the code below:

```
  while (ros::ok())
  {
    ros::spinOnce();
    loop_rate.sleep();
    if(CLUSTER_FLAG) // Check if ready to cluster
    {
      CLUSTER_FLAG=false;
      double X[pointCloud.points.size()][3];
      int IDX[pointCloud.points.size()];
      init_X(X, pointCloud.points.size()); // Initialize input array for DBSCAN
      C = DBSCAN(X, epsilon, minPts, pointCloud.points.size(), IDX);
      printToFile(X, IDX); // Log output

      /* TODO: Uncomment these when ready to test Convex Hull */
      //int noObservations = NUM_CIRCLES*SIZE_CIRCLE;
      //int noHullPoints[C];
      //Point HullPoints[];
      //geometry_msgs::PoseArray clusterVertices[C]; // holds vertices for all clusters
      // get convex hull vertices
      //getConvexHull(C, IDX, X, noObservations, clusterVertices);
      //sortVertices(clusterVertices, C);
      //vertices_pub.publish(TODO);
    }
    loop_rate.sleep();
  }
```

The spin function checks for new point cloud data. If found, a flag is set which allows for the execution of the DBSCAN algorithm. This loop keeps iterating until the node terminates. In comments is the function that calculates the convex hull of the clusters and its publisher. They will be uncommented when we begin testing the convex hull.

4.2 Graham Scan Algorithm - Julio Martinez

Although there are competing algorithms that find the defining edges of the convex hull of a set of points such as the Gift Wrapping algorithm and the Graham Scan algorithm, the Graham Scan Algorithm was chosen due to efficiency. The gift Wrapping algorithm, which outputs the same information, is not as efficient with run time at $\mathcal{O}(nh)$ where n denotes the number of samples and h denotes the number of points in the hull (aka number of edges that define the hull). However, the Graham Scan algorithm has a run time of $\mathcal{O}(n \log n)$ which despite being more complex as an algorithm is much more efficient, especially as the data set grows with samples. In our case, this is of special importance since ORB SLAM may return a large number of observations with a large number of clusters, each of which contain a large number of samples.

The following source code implements the Graham Scan Algorithm for the cluster found via DBSCAN. An explanation is found in the following paragraphs. The Graham Scan algorithm itself was found on github and is cited in [5]

```
/* run ConvexHull */
void getConvexHull(int C, int IDX[], double X[][DIMENSIONS_IN_SPACE],
                   int noObservations, geometry_msgs::PoseArray clusterVertices[]){
    //array to contain number of points in each cluster
    //int clusterCount[C+1];
    //clusterCount[0] = 0;
    int noPoints;
    //for C clusters starting at 1 (since 0 is noise)
    for (int n = 1; n < C+1; n++){
        noPoints = std::count(IDX,IDX + noObservations ,n);
        //make array to contain all points in cluster number n
        Point cluster[noPoints];
        cout<<"-----Starting Loop: "<<n<<endl;
        int j = 0;
        // Take these points out of the X data when IDX the correct cluster index
        //int noPoints =  NUM_CIRCLES*SIZE_CIRCLE;
        for (int i = 0; i < noObservations; i++){
            //Checking for correct cluster index
            if (IDX[i] == n && j < noPoints)
            {
                cluster[j].x = X[i][0];
                cluster[j].y = X[i][1];
                j = j+1;
            }
        }
        //Get the Convex Hull from Graham Scan Algorithm
        stack<Point> hull = grahamScan(cluster, noPoints);
        cout<<"grahamScan call "<< n <<endl; // test output
      // used count #vertices/cluster for printing/testing
        int noHullPoints = 0;
        int verticeIndex=0; // used to access vertice for printing/testing
      // temp PoseArray that will hold the vertices for the current cluster
      geometry_msgs::PoseArray currClusterVert;
        char buffer [50]; // buffer that will hold the frame id of the cluster
      // Loop over the hull stack and retrieve all vertices for currentCluster
      while (!hull.empty()) {
      // parse frame id
      // frame id string = "n C" where n=currentCluster, C=totalClusters
        sprintf(buffer, "%d %d", n, C);
      currClusterVert.header.frame_id=buffer; // set frame id
        Point p = hull.top(); // get vertice
        geometry_msgs::Pose vertice;
        vertice.position.x = p.x; // store x dimension of vertice
        vertice.position.y = p.y; // store y dimension of vertice
        currClusterVert.poses.push_back(vertice); // store vertice
        hull.pop();
      // store vertices for current cluster
      clusterVertices[n-1] = currClusterVert;
        noHullPoints = noHullPoints + 1;
        verticeIndex++;
        }
        cout << "no. Hull points = " << noHullPoints << endl << endl; // test output
```

```
53      }
}
```

In order to utilize the Graham Scan algorithm for DBSCAN a function was created named getConvexHull as shown in the source code listing above on line 2. The main inputs for this function are the cluster number C, the ID arrray, IDX, and the data itself, X. This is enough information to find the clusters of all the found clusters with DBSCAN.

In line 5 an array is created to contain all the number of points that belong to each cluster. This information is necessary for the for loops that will go through all the data in each cluster to find the edges. In line 7 the noPoints variable is created and reassigned every loop and placed in the corresponding index of the clusterCount array.

In line 9 the outer most loop is created. This loop cycles through every cluster. In line 17, a nested loop cycles through the number of observations. Thus for each cluster, all samples will be checked to see if they belong to the current cluster (i.e the C count) and if so they will be recorded in an array.

After the nested loop is finished looping through all samples, a complete cluster is stored and subsequently used as the input to call the Graham Scan algorithm, as shown in line 28. The array that holds the values is shown in line 12, the variable cluster. This variable is written over each loop which is why it is declared inside the loop iterating over the cluster numbers. Thus, it is renewed and wiped clean each loop. Line 30 initialized the variable to count the number of points (edges in each cluster), which will be useful in the ROS implementation shown below.

4.3 Sorting - Shiyuan Huang

The following code in the Graham Scan algorithm works to eliminate potential duplicates in the input point array so that the sort function works as expected:

```
// returns square of Euclidean distance between two points
int sqrDist(Point a, Point b) {
    int dx = a.x - b.x, dy = a.y - b.y;
    return dx * dx + dy * dy;
}

// used for sorting points according to polar order w.r.t the pivot
bool POLAR_ORDER(Point a, Point b) {
    int order = ccw(pivot, a, b);
    if (order == 0)
        return sqrDist(pivot, a) < sqrDist(pivot, b);
    return (order == -1);
}

stack<Point> grahamScan(Point *Oripoints, int M){
    stack<Point> hull;
    /*Eliminate duplicates in the input points*/
    int N = 1,diff=1;
    Point points[M];
    points[0]=Oripoints[0];          //Initialize a new point array to contain the unique
    points.
    for (int i=1;i<M;i++){
      for(int j=0;j<N;j++)
        if(sqrDist(points[j],Oripoints[i])<0.05){
                diff=0;   // Define a minimal distance limit=0.05. Points within the
    distance limit is considered to be identical
        }

      if(diff==1){
        points[N]=Oripoints[i];
       // cout<<points[N].x<<  points[N].y<<endl;
        N++;
      }
      diff=1;
    }
```

```
34
    sort(&points[0] + 1, &points[0] + N, POLAR_ORDER);
36 }
```

sort() requires a comparison function, in this case the POLAR_ORDER, to define a strict weak ordering. That is, POLAR_ORDER(a,b) and POLAR_ORDER(b,a) must be different. If there are two points in the array are identical, sort() breaks because it cannot give a strict order between them. Due to the limit of precision, two different points can be recognized as identical by POLAR_ORDER if they are too close and have approximately the same sqrDist from the pivot point. So in the grahamScan we need to first create a new point array that contains points that are recognized as unique.
In lines 19-21, a new point array is initialized. The first point is set to be the first point of the input array. N is the new length. A parameter, diff, is used to compare whether two points are equal. Lines 23-34 is a loop that eliminates the duplicates. The nested for loop will compare each two points in the input point array(Oripoints) and update the new unique array(points). Line 24 sets a minimal distance to the sqrDist, which is 0.05. A point whose distance square from the points stored in the unique array is below the limit would change diff to be zero. In lines 28-29, the unique array is concatenated with a new point if diff is still one. So the loop finally will return a modified unique array from the original points.
Line 35 is the sort function. It starts from the second position in the point array(the first point is the one with least Y coordinate) to the last position. It pairwise compares the points in this range following the ordering defined by POLAR_ORDER.

5 Methodology

5.1 DBSCAN Testing

We now outline the procedure we used to test the C++ and ROS integrated version of DBSCAN. The following code describes a function used to log the output of the algorithm:

```
/* This function writes to a file all of the coordinates for each
2    point in the point cloud */
void printToFile(double X[][3], int IDX[])
4 {
  ofstream myfile; // initialize print stream
6  myfile.open ("sample_point_cloud.txt"); // open file

8  // Loop over every point in the point cloud and write their
  // coordinates to the file
10 for(int i=0; i<pointCloud.points.size(); i++)
  {
12   myfile << setw(14) << pointCloud.points[i].x  << setw(14)
         << pointCloud.points[i].y  << setw(14) << pointCloud.points[i].z << endl;
14 }
  myfile.close(); // close file
16
  //ofstream myfile; // initialize print stream
18 myfile.open ("X.txt"); // open file

20 // Loop over every point in the point cloud and write their
  // coordinates to the file
22 for(int i=0; i<pointCloud.points.size(); i++)
  {
24   myfile << setw(14) << X[i][0]  << setw(14)
         << X[i][1]  << setw(14) << X[i][2] << endl;
26 }
  myfile.close(); // close file
28
  //ofstream myfile; // initialize print stream
30 myfile.open ("IDX.txt"); // open file

32 // Loop over every point in the point cloud and write their
```

```
  // coordinates to the file
  for(int i=0; i<pointCloud.points.size(); i++)
  {
    myfile << IDX[i] << endl;
  }
  myfile.close(); // close file

}
```

This function prints to a file the contents of three important program variables: pointCloud, X, and IDX. The variable pointCloud is a copy of the unprocessed point cloud storing message published by ORB SLAM. The variable X holds an array of all the points tracked in the ORB SLAM point cloud and is the array used by DBSCAN to compute the clusters. Finally, the variable IDX is the array that holds cluster labels for each point in X and is computed by DBSCAN. The reason why we print this data to a file is that we want to visualize it using MATLAB. This is accomplished using the following MATLAB script:

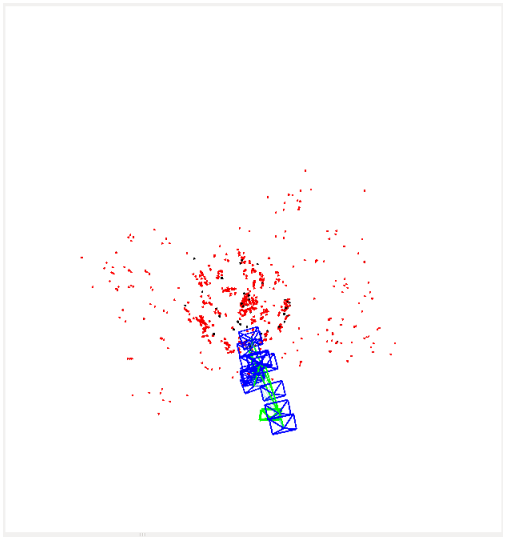
```
clc
clear

load('sample_point_cloud')
load('IDX')

X = sample_point_cloud(:, 1);
Y = sample_point_cloud(:, 2);
Z = sample_point_cloud(:, 3);

epsilon=0.09;
MinPts=20;
XY = [X, Y];

PlotClusterinResult(XY, IDX);
title(['DBSCAN Clustering XY (Frontal View) (\epsilon = ' num2str(epsilon) ', MinPts = '
    num2str(MinPts) ')']);

figure
XZ = [X, Z];
PlotClusterinResult(XZ, IDX);
title(['DBSCAN Clustering XZ (Top View) (\epsilon = ' num2str(epsilon) ', MinPts = ' num2str
    (MinPts) ')']);
```

The function PlotClusterinResult can be found in [4].

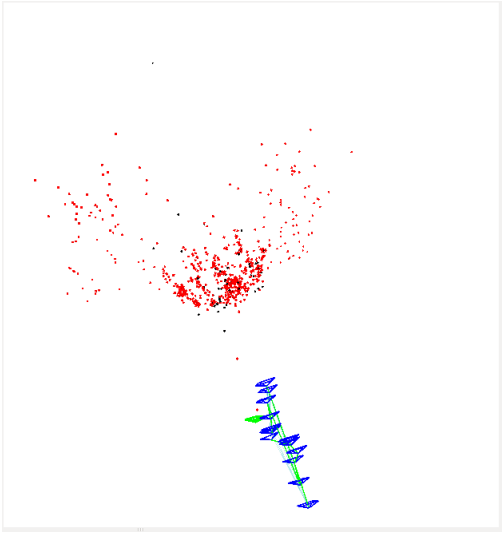
Before the end of the quarter, we were able to finalize the ROS integration and test it using actual ORB SLAM point cloud data. For this first test, we wanted to setup a testing environment with reduced noise in order to quickly fix the clustering parameters and test the functionality of the algorithm. Furthermore, we wanted to test the algorithm on a single object, for similar reasons. Thus, we decided to use a soccer ball as an obstacle and place it on the ground. Then, we faced the camera towards the floor where the ball was placed, and ran the algorithm. With this setup, we ensured that there was only one obstacle in the frame with some noise being picked up by the texture of the lab floor. The following is a picture illustrating the test environment setup and how the camera was positioned:

Figure 4: Test Object, Frontal View

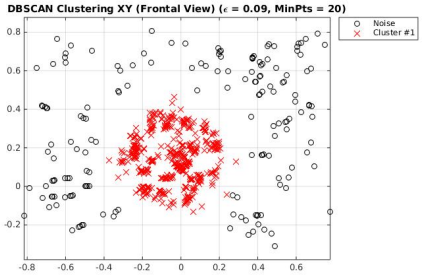
Here are the results of this test:



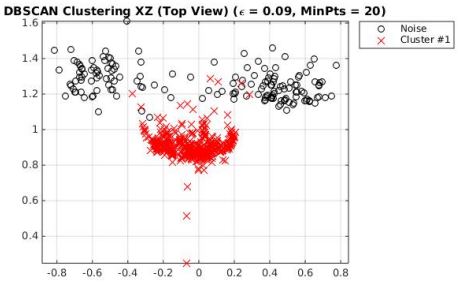
(a) RVIZ, Frontal View



(b) RVIZ, Top View



(c) DBSCAN Plot, Frontal View



(d) DBSCAN Plot, Top View

Figure 5: Test Results

The first two images were taken from RVIZ, which is a tool in ROS that is commonly used to plot visual data. These images illustrate the point cloud that is being generated by ORB SLAM, shown in red.

Moreover, the camera frame is also tracked, which is represented by the blue markers. The bottom two images were generated using the Matlab visualization script. These images demonstrate the correctness and effectiveness of the DBSCAN algorithm, at least for the case of one object. It is important to note that only through careful tuning of the clustering parameters were we able to get these results. Thus, an important challenge for this project will be to device decision rules for the setting of these parameters.

6 Tips

6.1 Using MATLAB to Visualize Data

In order to test the output of DBSCAN, we developed a script that plots point cloud data and color codes the points using their corresponding labels. This script utilizes plotting tools described in [4]. Furthermore, the script requires point cloud data and label data, which we load into Matlab from a text file. In our obstacle detection node, we declared a function that does this and generates the following text files: sample_point_cloud.txt, X.txt, and IDX.text. These files are created in the directory where the node is launched from. To use the visualization script, create a directory and copy the script there, along with the plotting script in [4]. Finally, run the obstacle detection node and move the newly created files from the current directory to the directory with the visualization script. You are now ready to run the script and test the DBSCAN output.

7 Pitfalls

7.1 Sorting

In the grahamScan a minimal distance = 0.05 is defined manually to eliminate possible duplicates. When the minimal distance is too small, there may exist points that are read as identical by POLAR_ORDER. (When I change the minimal distance = 0.01, sort() breaks). Sort() works well in the testing example where epsilon = 2. But if epsilon, the distance threshold that classifies a Neighbor, is smaller than 0.05, sort() may break, which is a problem because ORB SLAM gives small coordinates within [-1,1], so the choice of epsilon tends to be smaller than 0.05.

A potential solution is to scale up the coordinates in grahamScan to increase the sqrDist between points, so that POLAR_ORDER can tell the difference between them.

8 Conclusions and Future Directions

8.1 DBSCAN Testing

The DBSCAN testing this quarter was done in a very controlled environment, with only one obstacle allowed in the frame. Although results using this setup look very promising, it would be best practice to consider more robust testing scenarios for the algorithm. Therefore, one of our future goals is to scale the number of obstacles in the frame and improve the algorithm parameters for such environments.

8.2 Graham Scan Testing

So far, we have not been able to test the C++ version of our Convex Hull algorithm using ORB SLAM data due to the sorting problem previously discussed. Therefore, a future goal is to test the Convex Hull algorithm. From there, we would move to more robust testing using quadrotors in actual flight and more complex obstacle configurations.

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