Cluster Assignment Using DBSCAN and ORB-SLAM for Obstacle Detection

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

Julio Martinez and Gerardo Gonzalez

Fall 2015

Contents

1 Summary

The objective for this project was 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 online, we decided to go with density-based spatial clustering of applications with noise (DBSCAN) because of its noise handling functionality and pseudocode availability. Finally, we worked on the development of the algorithm in C++.

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) 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(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)
```

4 Our Contributions

The following algorithm was designed based on our interpretation of the paper in [1] and the pseudocode shown in section 3.2. The main source code that will be shown is programmed in C++ and will be function based.

4.1 DBSCAN

The algorithm shown in listing 1, which will be referred to as the umbrella algorithm (since this algorithm incorporates all other functions in the code), will be discussed and explained in this section. The umbrella function is named DBSCAN and will utilize other functions named expandClusters, pdist2, and RegionQuery, hence the umbrella term. The DBSCAN algorithm takes three user inputs. These inputs are the data set X (the point cloud), epsilon, minPts and n, which is the number of points in X. The second and third parameters were discussed earlier. Here, epsilon = ϵ as shown in equation 1 under definition 1 and minPts = n_{min} as shown in definition 2.

```
1  void DBSCAN(double X[][3], double epsilon, int minPts, int n){
2      int C = 0; //cluster number
3      int IDX[n]; //ID for X
4      // Pairwise distance matrix
5      vector<vector<double> > D(n, std::vector<double>(n));
6      pdist2(D, X, X);
7      to_screen_matrix(D);
8      // Vector of all neighbors
9      vector<int> Neighbors;
10     //Bookkeeping arrays
11     bool visited[n];
12     bool isnoise[n];
13     //Initialize IDX to all noise (zero cluster)
14     for (int i = 0; i < n; i++){
15         IDX[i] = 0;
16         visited[i] = false;
17         isnoise[i] = false;
18     }
19     cout << "Initialized IDX, visited, and isnoise." << endl << endl;
20     cout << "Starting Main DBSCAN for loop..." << endl;
21     //Find all neighbors for cluster
22     for (int i = 0; i < n; i++){
23         if (!visited[i]){
24             visited[i] = true;
25             RegionQuery(D, Neighbors, i, epsilon);
26             cout << "Loop " << i << ": RegionQuery complete.\n";
27             if (sizeof(Neighbors)/sizeof(Neighbors[0]) < minPts) {
28                 isnoise[i] = true;
29             }
30             else {
31                 C = C + 1;
32                 cout << "Loop " << i << ": Starting ExpandCluster...\n";
33                 expandCluster(D, IDX, i, Neighbors, C, visited, epsilon, minPts);
34             }
35         }
36     }
37     to_screen_array(IDX,n);
38 }
```

Listing 1: DBSCAN Umbrella Aglorithm

In line 2, the integer variable C is set to zero, where C denotes the assignment number of the cluster and any points with the C=0 label are categorized as noise (with this scheme, you can think of noise as a type of cluster). Furthermore, on line 3, an array with size n and label *IDX* is initialized. Here, *IDX* will ultimately hold the ID of the points, or *observations*, in the data set X. The input X is a data set with 3 columns where the first column denotes the x-coordinate, the second the y-coordinate, and the third the z-coordinate. With this scheme, each row is an observation. Thus, *IDX* will hold values 0, 1, 2, and so on such that the cluster for the ith row of X will be held in the ith element of IDX, i.e. if the 3rd point in X belongs to the 7th found cluster, the 3rd element of IDX will hold an integer 7, or if the point is an outlier (noise) then IDX will hold an integer 0. Since the data set will contain n number of observations, both X and IDX will contain n rows.

In line 6, a function pdist2 is called. This function takes in the data set X and returns a 2-dimensional vector (a vector of vectors), D, with n rows and n columns. After this function call, D will contain all the pairwise distances for all observations in X. Thus it will be nxn and symmetric. This function will be discussed in section 4.2. These pairwise distances will be needed to see which points fall within the Eps-neighborhood of other points and to further qualify and label clusters.

Next we declare two helping integer arrays of size n (for each observation in X) in lines 11 and 12. These are isnoise and visited. In lines 14-18, both will be initialized to false for every element. The array isnoise will be false, meaning that all elements are not considered noise. All elements in isvisited are also initialized to false, meaning that none of the elements have been visited. Lastly, all elements in IDX are initialized to zero, categorizing everything as noise. Of course all these elements will be changed according to what is actually found, but this initializing helps save some time.

This next part of the algorithm is where the assignment of the clusters actually takes place. In line 22, a for loop is set in place to run through every observation's pairwise distance to other observations. The first step in the for loop is to include a conditional statement to check whether or not an observation has been visited. If visited[i] is false, then we have not visited the ith observation, and so we continue only if !visited[i] is true. If this is the case, then before we continue, we make sure we mark visited[i] as true, to note that after this loop, the ith observation has been visited. Next, in line 31, we run the RegionQuery function. This function will be described in detail in section 4.4. However, to understand DBSCAN, it suffices to know that RegionQuery will check the pairwise distances for the ith observation that are stored in the ith row of D. It will check for and return an array *Neighborhood* that contains all the column indices in D that have a pairwise distance less than epsilon to the ith observation. That is, it will check for and record all points that are in the Eps-neighborhood of the ith observation as described in definition 1 in section 3.2.

The cluster is defined by checking whether the minimum number of observations required are in the vector Neighbors. This is done in line 27 by checking its size. However, if this requirement is not met, then the ith observation is considered noise, otherwise it is not noise and we found a new cluster. As shown in line 31, we increment C to denote the new cluster number found and then run the expandCluster function.

The expandCluster function will be explained in section 4.3. However, a brief description will be given here to show how DBSCAN works. As described above, when a cluster is found using the pairwise distances of the ith observation (i.e. both minPts and epsilon requirements are satisfied), the variable C is incremented to denote a new cluster label number. When expandCluster is called, the ith observation is assigned to the Cth cluster found. Then a subsequent search for all observations in its own cluster is made and these observation numbers (index of the observations) are assigned to its same cluster number C by using definitions 2 and 3 in section 3.2. This is done to have directly reachable and density connected observations. Thus, the end result will be a concatenation of Neighbor vectors that belong to the same cluster. All corresponding indices of IDX will be labeled with the number C, indicating the these observations belong to cluster C.

4.2 Pairwise Distance

The source code in listing 2 refers to the "pairwise comparison of distances" function.

```
2  // This function computes the pairwise distances between two arrays.
   void pdist2(vector<vector<double> > &D, double X[][3], double Y[][3])
4  {
       for(int i=0; i<D.size(); i++)
6      {
           for(int j=i; j<D.at(i).size(); j++) // Start at i because of matrix symmetry
8          {
               D.at(i).at(j) = getDistance(X[i][0],X[i][1],Y[j][0],Y[j][1]);
10             D.at(j).at(i) = D.at(i).at(j); // Matrix is symmetric. Set corresponding transpose of entry
           }
12     }
   }
14
   // Find Euclidean distance between two points
16 double getDistance(double point1_x, double point1_y, double point2_x, double point2_y)
   {
18     double distance = sqrt(pow((point1_x - point2_x),2) + pow(point1_y - point2_y,2));
       return distance;
20 }
```

Listing 2: Pairwise Distance function

The inputs for this function are two 2D arrays, where the number of rows represent the number of points (observations) in the point cloud, and the number columns represent dimensions in space, Also, a vector of vectors is used to denote the matrix that will hold all of the calculated distances. The function works by iterating over the 2D arrays and computing their corresponding pairwise distances. In the code, we make

use of the fact that the matrix is symmetric by setting the column index (j) equal to the row index (i) and then setting every entry equal to its corresponding transpose. By doing so, we reduce the number of computations needed by a factor of 1/2. Lastly, this function requires a helper function that computes the Euclidean distance between two points.

4.3 Expand Clusters

The source code shown in listing 3 is the expandCluster function utilized to find all remaining points of a cluster once a qualified Neighborhood cluster has been found.

```
1  void expandCluster(vector<vector<double> > D, int IDX[], int i, vector<int> Neighbors, int C, bool visited[],
       double epsilon, double minPts){
2      IDX[i] = C;
3      vector<int> Neighbors2;
4      int k = 0; int j;
5      while (true){
6          cout << "Loop " << i << ": In while loop...\n";
7          int j = Neighbors.at(k); // FIXME: ArrayOutOfBounds
8          //if have not visited
9          if (!visited[j]){
10             visited[j] = true;
11             //Get Points in Eps-neighborhood
12             RegionQuery(D,Neighbors2, j, epsilon);
13             //Check for minPts qualification of clusters
14             if (Neighbors2.size() >= minPts){
15                 //concatenate vectors into Neighbors
16                 cout << "Loop " << i << ": Starting MergeVector...\n";
17                 mergeVector(Neighbors,Neighbors2);
18                 cout << "Loop " << i << ": End MergeVector.\n";
19             }
20         }
21         //If that corresponding observation is zero
22         if(IDX[j] == 0){
23             //assign it to cluster C
24             IDX[j] = C;
25         }
26         //move in to next element in Neighbors
27         k = k+1;
28         //unless we exceed its size, in which case we exit loop
29         if (k > Neighbors.size()-1){
30             break;
31         }
32     }
33     cout << "Loop " << i << ": End while loop.\n";
34     cout << "Loop " << i << ": End ExpandCluster.\n";
35 }
```

Listing 3: expandCluster function

First, in line 2, the ith observation is assigned to cluster number C (C is passed in as an argument), by setting the ith element in IDX equal to C. Next, a variable k is initialized to zero. I this scheme, k will represent the element index of the vector Neighbors. In line 7, j is initialized to the kth element of k for clarity.

Next, in line 5 we start with a while loop. The while loop will run until all elements in Neighbors have been exhausted (since we do not previously know the size of Neighbors and it may continue to expand from concatenations as will be explained, it makes sense to use a while loop that is terminated after exceeding its size). When we enter the loop, the first thing needed is a conditional statement in line 9 to check whether the ith observation has been visited. If so, we do nothing. However, if this is not the case, then we set visited[i] to true (so that this is not computed again) and call the function RegionQuery using the jth observation and store its Eps-neigborhood indices in a vector Neighborhood2. Again, as in the umbrella algorithm DBSCAN, Neighbors2 is checked so that it satisfies the minPts condition (that Neighbors2 vector has at least minPts number of elements). If so, Neighbors and Neighbors2 are concatenated (merged together) to form a new vector labeled Neighbors that contains both vectors, meaning that the cluster has been expanded or that both observations in Neighbors and Neighbors2 are part of the same cluster. If, however, Neighbors2 does not have the minPts for a cluster, then this part is skipped and the Neighbors vector is unchanged. This is all shown in lines 13 - 19.

Lines 22-25 are to make sure that if IDX has not already been assigned a cluster number, then it is assigned by making the jth component of IDX equal to C. Of course, this only occurs when RegionQuery is called. In line 27, k is incremented to move onto the next element of Neighbors, starting the process all over again, and not exiting until we have exhausted all the Neighbors of the Neighbors, of the Neighbors, and so on. This way we get a complete cluster. If, however, the k is incremented above the element length of Neighbors, then the while loop is terminated and we know all observations in cluster number C have been exhausted.

4.4 Region Query

The source code in listing 4 refers to the Region Query function.

```
1
// This function iterates over a row in the pairwise distance matrix and outputs a vector that
3 // contains all the indexes for that row whose corresponding entries are less than the specified
// epsilon value
5 void RegionQuery(vector<vector<double> > D, vector<int> &Neighbors, int row, double epsilon)
{
7     Neighbors.clear();
      for(int j=0; j<D.at(row).size(); j++)
9     {
          if(D.at(row).at(j) < epsilon)
11        {
              Neighbors.push_back(j);
13        }
      }
15 }
```

Listing 4: Region Query function

This function iterates over a row in the pairwise distance matrix and outputs a vector that contains all the indexes for that row whose corresponding entries are less than the specified epsilon value. The inputs for this function are: the vector of vectors matrix of pairwise distances, a vector for all the indexes in the row that satisfy the condition, an integer that indexes a row of interest in the matrix, and a double with label *epsilon*, which holds the minimum distance value for two points to be considered neighbors.

5 Methodology

5.1 ORB-SLAM Point Cloud

The following section outlines the procedure used to extract the point cloud data from ORB SLAM. First, run the quadrotor and ORB SLAM in ROS. From this, create a sample map using the quadrotor's monocular camera. In order to visualize the point cloud data, use the "echo" tool in ROS to output the data in the /ORB_SLAM/Map topic to the terminal screen. A series of messages containing information used in the construction of ORB SLAM's map will be outputted. The message that holds the point cloud will have the header "Map Points". The data can then be copied to a text file for future testing. We used this method for testing the Matlab and initial C++ version of DBSCAN.

5.2 MATLAB Testing

After obtaining a sample point cloud from the ORB-SLAM algorithm, x and y data points were plotted using Matlab to get an idea of what the data looks like. Since the z coordinate is ignored, the x and y coordinates give a top view (a view from above) of the points, as shown in the scatter plot in figure 1.

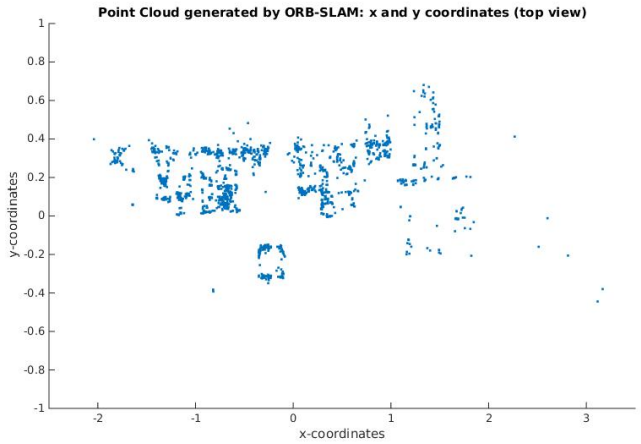


Figure 1: Scatter Plot of ORB-SLAM Generated Point Cloud

As you can see, the data itself must be well understood to determine which points are considered clusters and how many clusters should be made. To see the differences in choosing different parameters epsilon (ϵ) and minPts (n_{min}) in the C++ code, scatter plots of the clusters defined by colors were made using Matlab and will be shown below.

The first plot will be our reference case since this case gives reasonable results. This case was made using $\epsilon = 0.1$ and $n_{min} = 10$. The plot is shown below in figure 2.

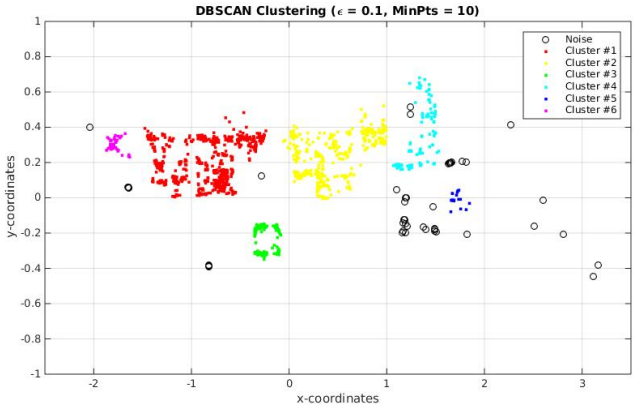
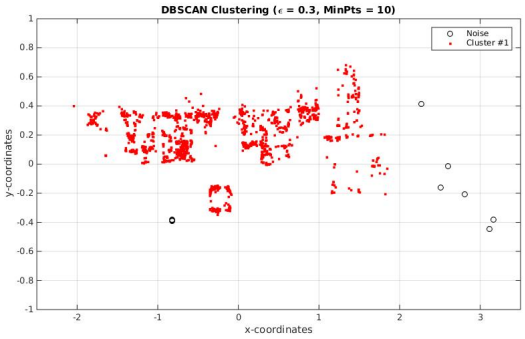


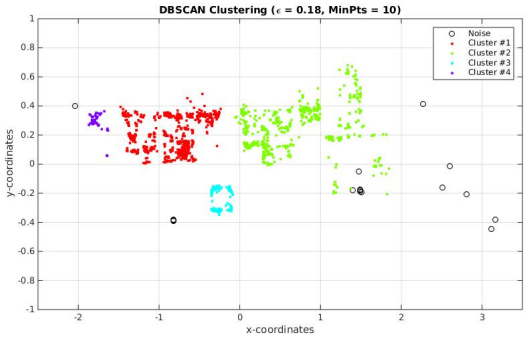
Figure 2: DBSCAN Matlab Scatter Plot

By using these two parameters on this data set, 6 clusters are created. Is is clear from the figure that all outlier points are considered noise (the black label), as well as any small clusters (clusters with less than 10 points).

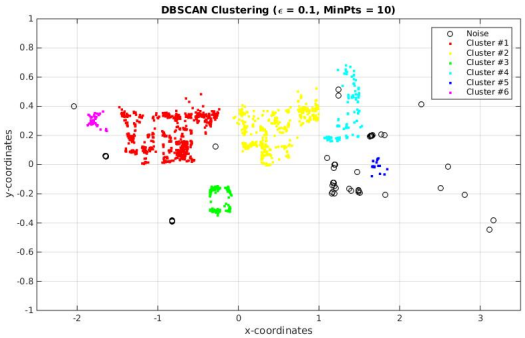
Next, we take a look at changing these parameters. Taking our reference case, $\epsilon = 0.1$ and $n_{min} = 10$, if we hold n_{min} constant at 10 but test $\epsilon = 0.3, 0.13, 0.1, 0.05$ as shown in figures 3a, 3b, 3c, and 3d we are able to see the changes caused by varying ϵ.



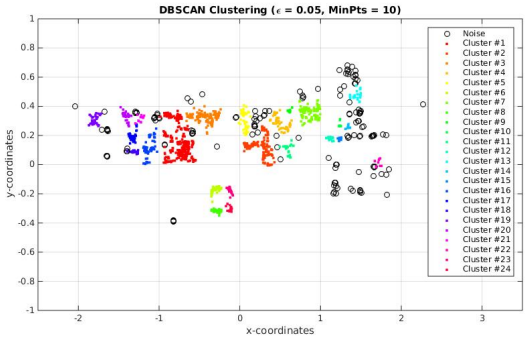
(a) $\epsilon = 0.3$, $n_{min} = 10$



(b) $\epsilon = 0.18$, $n_{min} = 10$



(c) $\epsilon = 0.1$, $n_{min} = 10$

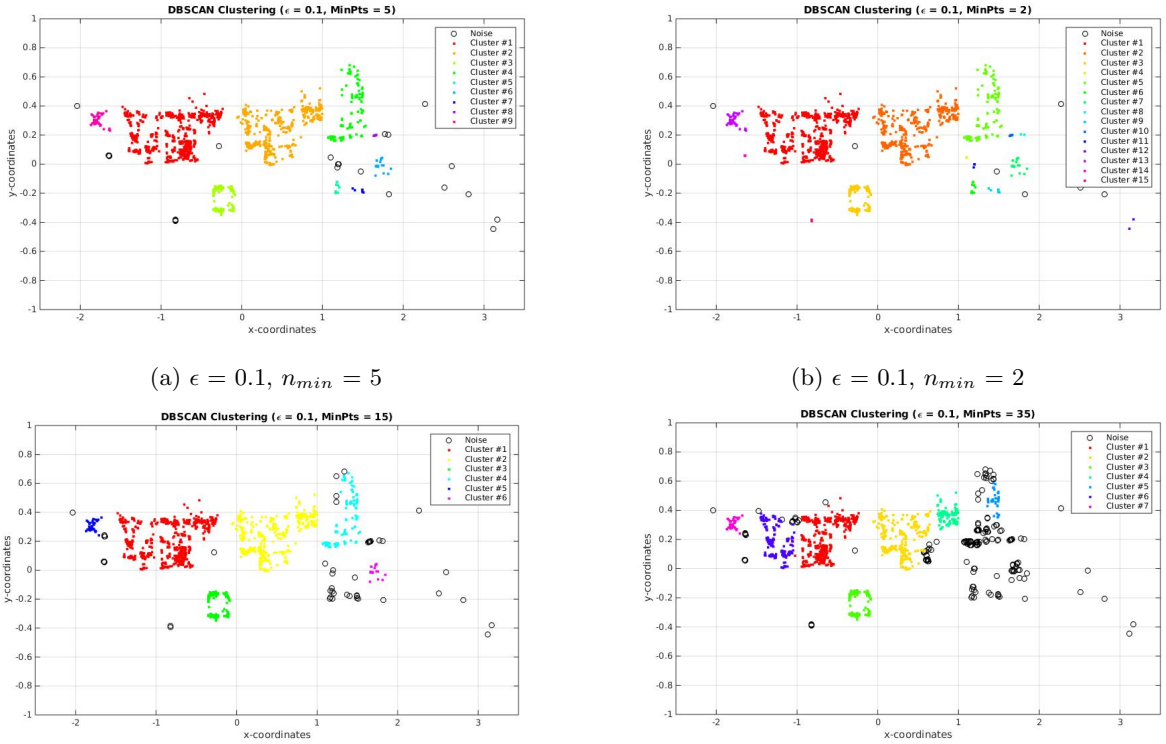


(d) $\epsilon = 0.05$, $n_{min} = 10$

Figure 3: Varying ϵ

As can be see in these figures, starting at a relatively large epsilon for the data set (which may be small for other data sets) will cause less clustering. In this case, $\epsilon = 0.3$ results in only one cluster. However, decreasing ϵ to 0.18 disconnects some of the clusters. The data is further broken up as the parameter is decreased to 0.1 (our reference case). However, because the data is broken up, some clusters are now considered noise, which results in more noise than before. Finally, with $\epsilon = 0.05$, the data is broken into 24 clusters and the noise is consequently increased.

Next, going back to our base case, we will hold $\epsilon = 0.1$ constant and allow n_{min} to increase from 2, 5, 15, to 35 points as shown in figures 4a, 4b, 4c, and 4d. The changes seen here are the difference in number of clusters, which seem to converge to around 6 to 7 clusters. However, the data starts with 15 clusters. Another significant change is in the amount of noise. When the n_{min} increases, noise also increases. This makes sense since more clusters that do not meet the minimum requirement will be left out as outliers.



(a) $\epsilon = 0.1$, $n_{min} = 5$

(b) $\epsilon = 0.1$, $n_{min} = 2$

(c) $\epsilon = 0.1$, $n_{min} = 15$

(d) $\epsilon = 0.1$, $n_{min} = 35$

Figure 4: Varying n_{min}

6 Tips

6.1 Parameters ϵ and n_{min}

From the observations made by changing the parameters ϵ and n_{min} it is concluded that a good understanding of the data is needed before defining what the ideal parameters should be. In this project, a good rule of thumb will be to use the following parameters shown below.

ϵ	n_{min}
0.1	10

However, it should be understood that this is only a rule of thumb and not a hard rule. Instead, starting with these parameters may be useful only to find the best parameters by proceeding as in section 5.2, where one parameter was fixed while the other was tested for results and vice versa, until a good match is found.

7 Pitfalls

7.1 Problems with ORB-SLAM

Unfortunately, there are problems with ORB SLAM's tracking algorithm. We noticed that under certain circumstances, the visualization scaling for the map is severely distorted. A possible explanation for this erroneous behavior is that false "outlier" data points (data points that are perceived to be far away from the camera) cause the visualization to zoom out in order to fit those points. To address this problem, we changed the calibration parameters for the camera and updated to the most recent ORB SLAM version. Since doing so, we have not experienced the error. If the error were to return, a possible solution would be

to add a filter to the tracking node of the algorithm that checks for extreme outliers, eliminating them once detected.

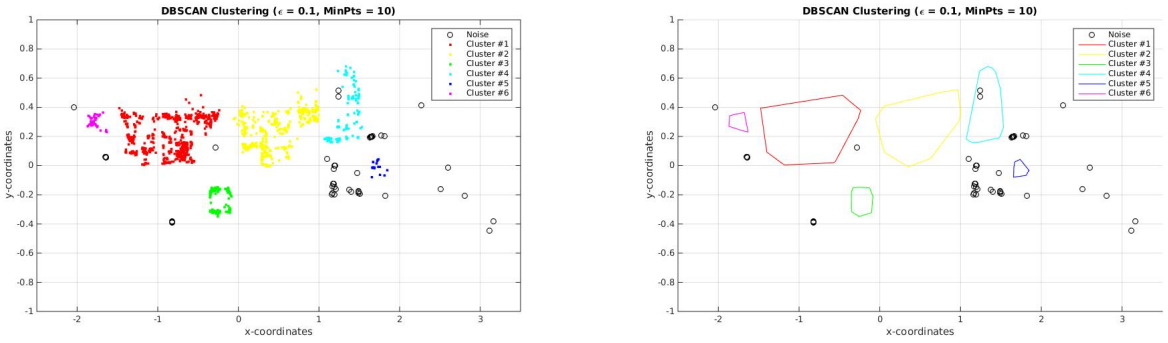
8 Conclusions and Future Directions

8.1 ROS Implementation

The next step for this project is to move the C++ code into the ROS platform and to test it out using the Android application. Future work to do includes: creating a DBSCAN node, a Graham scan node, and a main node that processes obstacle related information from the quadrotor and communicates it to the Android app. We hope that our quadrotor network will be able to utilize this obstacle information to identify potential obstacles and move around them in real-time. Testing using quadrotors will also be a main goal for next quarter.

8.2 Graham Scan

The major idea behind using DBSCAN was to be able to identify objects by using the clusters that define them. However, because of noise, and the vast amount of points that may define a cluster, it is not efficient to pass all points in a cluster. Instead, our future work in defining obstacles will be to take the fewest points possible from the cluster that nonetheless define the obstacle (or cluster) as a whole. To do this, the convex hull of the cluster, which is the smallest convex set that contains all of the points in the cluster, will be computed. Then, once the the convex hull is known, the vertices that define this convex hull will be used to precisely define the obstacle itself. Thus, only a small subset of the actual data will be used in defining the obstacle. To further help illustrating this, refer to figures 5a and 5b. Here you will see again our reference case shown previously in section 5.2, as well as the convex hulls' that define the corresponding clusters, shown in the figure to the right. The clusters themselves are removed, but the hulls are left. One could now imagine an agent making its way across the region moving between the obstacles and without interfering with them.



(a) $\epsilon = 0.1$, $n_{min} = 10$

(b) Matlab Convex Hull Illustration

Figure 5: Obstacles from Cluster Convex Hulls

The way we plan to do this is by employing the Graham-Scan algorithm, which works similar to the Gift-Wrapping algorithm but with a few improvements. Nevertheless, the end result is the same; the final vertices will be returned for their corresponding convex hull.

9 Index

9.1 Source Code

```
#include <iostream>
#include <math.h>
#include <vector>
#include <iomanip>      /* Nice output formating */
#include <stdio.h>      /* printf, scanf, puts, NULL */
#include <stdlib.h>     /* srand, rand */
#include <time.h>       /* time */

using namespace std;

const int N = 100; // Size of X (# of observations)

void mergeVector(vector<int>& A, vector<int> B);
void DBSCAN(double X[][3], double epsilon, int minPts, int n);
void expandCluster(vector<vector<double> > D, int IDX[], int i, vector<int> Neighbors, int C,bool visited[], double epsilon, double minPts);
void RegionQuery(vector<vector<double> > D, vector<int> &Neighbors, int row, double epsilon);
void pdist2(vector<vector<double> > &D, double X[][3], double Y[][3]);
double getDistance(double point1_x, double point1_y, double point2_x, double point2_y);
void init_X(double ar[][3], int length);
void to_screen_X(double ar[][3], int length);
void to_screen_matrix(vector<vector<double> > vec);
void to_screen_array(int ar[], int length);


// Test DBSCAN
int main(void) {

    double X[N][3];
    init_X(X,N); // Create sample point cloud
    to_screen_X(X,N);
    double epsilon = 0.5;
    int minPts = 7;
    DBSCAN(X,epsilon,minPts,N);
    return 0;
}




//DBSCAN
void DBSCAN(double X[][3], double epsilon, int minPts, int n){
    int C = 0; //cluster number
    int IDX[n]; //ID for X
    // Pairwise distance matrix
    vector<vector<double> > D(n, std::vector<double>(n));
    pdist2(D, X, X);
    to_screen_matrix(D);
    // Vector of all neighbors
    vector<int> Neighbors;
    //Bookkeeping arrays
    bool visited[n];
    bool isnoise[n];
    //Initialize IDX to all noise (zero cluster)
    for (int i = 0; i < n; i++){
        IDX[i] = 0;
        visited[i] = false;
        isnoise[i] = false;
    }
    cout << "Initialized IDX, visited, and isnoise." << endl << endl;
    cout << "Starting Main DBSCAN for loop..." << endl;
    //Find all neighbors for cluster
    for (int i = 0; i < n; i++){
        if (!visited[i]){
            visited[i] = true;
            RegionQuery(D, Neighbors, i, epsilon);
            cout << "Loop " << i << ": RegionQuery complete.\n";
            if (sizeof(Neighbors)/sizeof(Neighbors[0]) < minPts) {
                isnoise[i] = true;
            }
            else {
                C = C + 1;
                cout << "Loop " << i << ": Starting ExpandCluster...\n";
                expandCluster(D, IDX, i, Neighbors, C, visited, epsilon, minPts);
            }
        }
    }
    to_screen_array(IDX,n);
}

void expandCluster(vector<vector<double> > D, int IDX[], int i, vector<int> Neighbors, int C, bool visited[], double epsilon, double minPts){
    IDX[i] = C;
    vector<int> Neighbors2;
    int k = 0; int j;
    while (true){
        cout << "Loop " << i << ": In while loop...\n";
        int j = Neighbors.at(k); // FIXME: ArrayOutOfBounds
        //if have not visited
        if (!visited[j]){
            visited[j] = true;
            //Get Points in Eps-neighborhood
            RegionQuery(D,Neighbors2, j, epsilon);
            //Check for minPts qualification of clusters
            if (Neighbors2.size() >= minPts){
                //concatenate vectors into Neighbors
                cout << "Loop " << i << ": Starting MergeVector...\n";
                mergeVector(Neighbors,Neighbors2);
```

```
                cout << "Loop " << i << ": End MergeVector.\n";
            }
        }
        //If that corresponding observation is zero
        if(IDX[j] == 0){
            //assign it to cluster C
            IDX[j] = C;
        }
        //move in to next element in Neighbors
        k = k+1;
        //unless we exceed its size, in which case we exit loop
        if (k > Neighbors.size()-1){
            break;
        }
    }
    cout << "Loop " << i << ": End while loop.\n";
    cout << "Loop " << i << ": End ExpandCluster.\n";
}


//MERGE VECTOR
void mergeVector(vector<int>& A, vector<int> B){

    A.insert( A.end(), B.begin(), B.end() );

}

// This function iterates over a row in the pairwise distance matrix and outputs a vector that
// contains all the indexes for that row whose corresponding entries are less than the specified
// epsilon value
void RegionQuery(vector<vector<double> > D, vector<int> &Neighbors, int row, double epsilon)
{
    Neighbors.clear();
    for(int j=0; j<D.at(row).size(); j++)
    {
        if(D.at(row).at(j) < epsilon)
        {
            Neighbors.push_back(j);
        }
    }
}

// This function computes the pairwise distances between two arrays.
void pdist2(vector<vector<double> > &D, double X[][3], double Y[][3])
{
    for(int i=0; i<D.size(); i++)
    {
        for(int j=i; j<D.at(i).size(); j++) // Start at i because of matrix symmetry
        {
            D.at(i).at(j) = getDistance(X[i][0],X[i][1],Y[j][0],Y[j][1]);
            D.at(j).at(i) = D.at(i).at(j); // Matrix is symmetric
        }
    }
}

// Find Euclidian distance between two points
double getDistance(double point1_x, double point1_y, double point2_x, double point2_y)
{
    double distance = sqrt(pow((point1_x - point2_x),2) + pow(point1_y - point2_y,2));
    return distance;
}

// Initialize 2D array of sample points
void init_X(double ar[][3], int length)
{
    // initialize random seed
    srand (time(NULL));

    for(int i=0; i<length; i++)
    {
        for(int j=0; j<3; j++)
        {
            ar[i][j] = (double) (rand() % 1000 + 1);
        }
    }
}

// Print to screen 2D array of sample points
void to_screen_X(double ar[][3], int length)
{
    cout << "X = \n";
    for(int i=0; i<length; i++)
    {
        for(int j=0; j<3; j++)
        {
            cout << setw(3) << ar[i][j];
        }
        cout << endl;
    }
    cout << endl;
}

// Print the 2D pairwise matrix to the screen
void to_screen_matrix(vector<vector<double> > vec)
{
    cout << "D = \n";
    for(int i=0; i<vec.size(); i++)
    {

        for(int j=0; j<vec.at(i).size(); j++)
        {
            cout << setw(9) << setprecision(4) << vec.at(i).at(j);
        }
        cout << endl;
    }
    cout << endl;
}
```

```
206
// Print to screen 1D array
208 void to_screen_array(int ar[], int length)
{
210     cout << "IDX = \n";
        for(int i=0; i<length; i++)
212     {
            cout << "Point " << i+1 << ":" << setw(3) << ar[i] << endl;
214     }
        cout << endl;
216 }
```

Listing 5: DBSCAN Complete Function

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.