Report perception task

Giovanni Todesco

September 2025

1 Introduction

L'intenzione di questo elaborato è quella di spiegare tutti i passaggi e con quali strumenti è stata risolta la task proposta dalla seguente repository Github: https://github.com/eagletrt/recruiting-sw/tree/master/driverless/perception

In aggiunta verranno anche spiegate le eventuali difficoltà che sono state riscontrate durante la risoluzione della task.

Gli strumenti che sono stati utilizzati sono:

- C++ language
- CMake
- PCL (Point Cloud Library)
- OpenMP

2 Level 1: Load and Display Captured Data

In questa prima fase viene chiesto di visualizzare una nuvola 3D prodotta da un sensore LiDAR. Questa fase non ha creato particolari problemi, è bastato solamente includere il file.

```
1 if (pcl::io::loadPCDFile<pcl::PointXYZ>("../data/cones.pcd",
      *raw_cloud) == -1) {
2     PCL_ERROR("File␣mancante␣o␣formato␣file␣sbagliato\n");
3     return -1;
4 }
```

Successivamente basta inizializzare il viewer 3D della libreria pcl, scegliendo la nuvola da visualizzare, implementando i colori dei punti e la posizione della "telecamera". Infine visualizzare la finestra usando un ciclo while finchè essa non viene chiusa.

```
1 pcl::visualization::PCLVisualizer::Ptr viewer(new pcl::
    visualization::PCLVisualizer("Visualizzatore␣PCL␣raw"));
2 viewer->addPointCloud<pcl::PointXYZ>(raw_cloud, "sample␣
    cloud");
3
```

```
pcl::visualization::PointCloudColorHandlerGenericField<pcl::
    PointXYZ> color_handler(raw_cloud, "z");
viewer->addPointCloud<pcl::PointXYZ>(raw_cloud,
    color_handler, "cloud_z");
viewer->setPointCloudRenderingProperties(pcl::visualization
    ::PCL_VISUALIZER_POINT_SIZE, 2, "cloud_z");
viewer->setCameraPosition(
-5, 0, 0,
0, 0, 0,
0, 0, 1
);
```

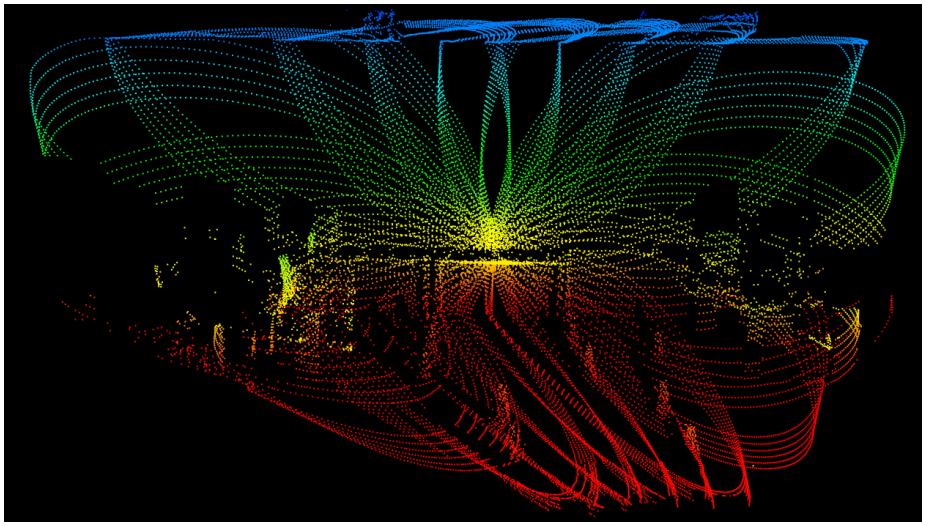

Figure 1: Raw cloud

3 Level 2: Cone Detection

La fase successiva consiste nell'estrarre dalla nuvola tutti i punti appartenenti a ostacoli o coni, per poi classificarli. L'obiettivo è quindi quello di pulire il più possibile la nuvola da punti indesiderati (soffitto, muri, pavimenti e rumore).

La difficoltà principale di questo livello è stata individuare la pipeline più efficace, evitando di rimuovere punti essenziali per la classificazione degli oggetti.

Dopo svariate prove, il miglior risultato è stato ottenuto dalla seguente pipeline.

3.1 Filtraggio asse z

Il primo passo è usare un filtro pass-through ($pcl :: PassThrough$) sull'asse Z per rimuovere punti troppo alti o troppo bassi. In questo caso essendo il sensore

a $z = 0$ si eliminano tutti i punti $z < -1$ e $z > 1$, rimuovendo automaticamente il soffitto.

3.2 Rimozione dei piani (pavimento e muri)

Per evitare che superfici come pavimenti o muri interferiscano con la suddivisione degli oggetti, è stata effettuata una rimozione dei piani mediante RANSAC, utilizzando una soglia di distanza di $0.05m$ così anche eventuale rumore viene rimosso.

Il riconoscimento dei piani è stato fatto utilizzando $pcl :: SACSegmentation$ mentre la rimozione dei piani dalla nuvola è stata effettuata con $pcl :: ExtractIndices$.

3.3 Rimozione degli outlier

Per migliorare la qualità della nuvola e ridurre il numero di punti isolati che potrebbero influenzare la classificazione, è stato applicato il filtro Statistical Outlier Removal ($pcl :: StatisticalOutlierRemoval$). Questo algoritmo calcola, per ogni punto, la distanza media rispetto ai suoi vicini più prossimi e rimuove quelli che si discostano significativamente dalla distribuzione complessiva. Il risultato migliore è stato ottenuto con $meank = 20$ e $StddevMulThresh = 1$.

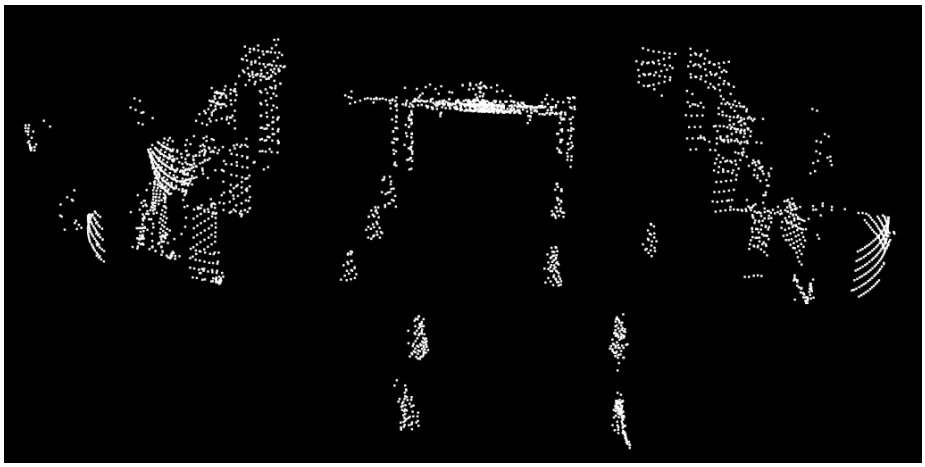

Figure 2: Dopo pipline di filtraggio

4 Level 3: Object Classification

Il terzo livello richiedeva di stabilire quali degli oggetti rilevati fossero effettivamente coni e quali semplici ostacoli. Questa è stata la parte più laboriosa del progetto. La difficoltà è stata decidere l'algoritmo di classificazione più efficace e trovare i giusti parametri. Dopo aver provato RANSAC, che ha prodotto risultati mediocri (7 coni su 10 correttamente riconosciuti), si è deciso di adottare

l'algoritmo ICP che, nonostante sia più lento, risulta più preciso, in quanto consente di costruire un modello personalizzato da confrontare con ciascun cluster.

Il primo passo è stato dividere i cluster data una nuvola di punti usando *pcl* :: *EuclideanClusterExtraction* con questi parametri:

- Tolleranza del cluster: 0.15 m
- Dimensioni minime dei cluster: 15 punti
- Dimensioni massime dei cluster: 2000 punti

Una volta inseriti in un vettore si controllano uno ad uno. Per velocizzare la classificazione (oltre ad usare OpenMP), dato che ICP è un algoritmo relativamente lento, vengono preliminarmente scartati i gruppi di punti che non presentano caratteristiche compatibili con un cono, mediante semplici controlli su dimensione del cluster, posizione e altezza.

```
if (cluster->size() < 20){
    #pragma omp critical
    {
        for (const auto &p : cluster->points){
        pcl::PointXYZRGB q{p.x, p.y, p.z, 255, 0, 0};
        colored_final_cloud->points.push_back(q);
        }
        cout<<"Cluster #" <<i <<" scartato: pochi punti\n";
        }
    continue
}
```

```
if (height <= 0.0f || base_radius <= 0.0f || height > 0.3f
    || min_pt.z > -0.3f){
    #pragma omp critical
    {
        for (const auto &p : cluster->points){
        pcl::PointXYZRGB q{p.x, p.y, p.z, 255, 0, 0};
        colored_final_cloud->points.push_back(q);
        }
        cout<<"Cluster #" <<i <<" scartato: posizione o
            altezza irrealistici\n";
    }
continue;
}
```

Se nessuna di queste condizioni è soddisfatta, si procede con l'ICP, che registra o allinea due nuvole di punti cercando la trasformazione rigida (rotazione e traslazione) che minimizza la distanza tra i punti corrispondenti. Per aumentare la precisione è stato creato un modello personalizzato del cono, poiché con un sensore LiDAR non è possibile "vedere" l'intero oggetto, ma solo la parte esposta al fascio laser. Questa scelta consente di migliorare il riconoscimento dei coni.

```
//funzione che costruisce un modello di cono personalizzato
```

```
2  pcl::PointCloud<pcl::PointXYZ>::Ptr makeConeModel(float
       height, float radius, int slices) {
3
4      float fov_deg = 180.0f; //angolo visibile dal LiDAR
5      auto model = std::make_shared<pcl::PointCloud<pcl::
           PointXYZ>>();
6      model->points.reserve(slices * 50);
7
8      const float fov_rad = fov_deg * M_PI / 180.0f;
9      const float start_angle = -fov_rad / 2.0f;
10     const float end_angle   =  fov_rad / 2.0f;
11     const int vertical_steps = 50;
12
13     for (int i = 0; i < slices; ++i) {
14         float angle = start_angle + (end_angle - start_angle
               ) * i / (slices - 1);
15
16         for (int k = 0; k <= vertical_steps; ++k) {
17             float z = height * k / vertical_steps;
18             float r = radius * (1.0f - z / height);
19
20             pcl::PointXYZ p;
21             p.x = r * cos(angle);
22             p.y = r * sin(angle);
23             p.z = z;
24
25             model->points.push_back(p);
26         }
27     }
28
29     model->width = static_cast<uint32_t>(model->points.size
           ());
30     model->height = 1;
31     model->is_dense = true;
32     return model;
33 }
34
35
36 bool isConeICP(const pcl::PointCloud<pcl::PointXYZ>::Ptr &
       cluster, double max_corresp, double score_threshold,
       double base_radius){
37     if (cluster->empty()) return false;
38
39     //stima bounding box del cluster
40     pcl::PointXYZ min_pt, max_pt;
41     pcl::getMinMax3D(*cluster, min_pt, max_pt);
42     float height = 0.28f;
43
44     //genera modello teorico e lo centra al cluster
45     Eigen::Vector4f centroid;
46     pcl::compute3DCentroid(*cluster, centroid);
```

```
47      auto cone_model = makeConeModel(height, base_radius, 50)
            ;
48      for (auto &p : cone_model->points) {
49          p.x += centroid[0];
50          p.y += centroid[1];
51          p.z += centroid[2];
52      }
53
54      //icp
55      pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ>
             icp;
56      icp.setInputSource(cone_model);
57      icp.setInputTarget(cluster);
58      icp.setMaximumIterations(20);
59      icp.setMaxCorrespondenceDistance(max_corresp);
60      icp.setTransformationEpsilon(1e-6);
61
62      pcl::PointCloud<pcl::PointXYZ> aligned;
63      icp.align(aligned);
64
65      if (!icp.hasConverged())
66          return false;
67
68      double score = icp.getFitnessScore(); //media distanze^2
69      cout <<"ICP fitness score: " <<score <<endl;
70
71      return score < score_threshold;
72  }
```

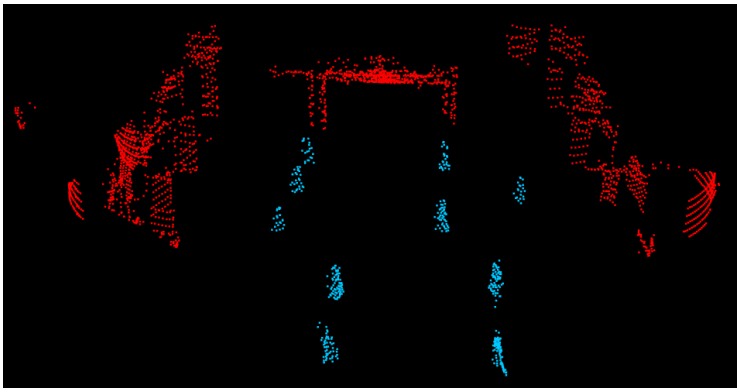

Figure 3: Dopo la classificazione

5 Level 4: Extract Track Edges

Questo livello richiedeva di estrarre e visualizzare i bordi del tracciato. Durante la classificazione è stato popolato un vettore che contiene i centroidi dei coni.

Per costruire i bordi del tracciato dobbiamo suddividere i coni in due gruppi, che sono rispettivamente il bordo destro e il bordo sinistro. Questo viene fatto con una semplice separazione assumendo che l'origine del sistema di riferimento si trovi al centro della pista

```
for (const auto &c : cone_centers) {
    if (c.y() <= 0)
        right_cones.push_back(c);
    else
        left_cones.push_back(c);
}
```

I coni di ciascun lato vengono successivamente ordinati in base alla vicinanza dell'origine utilizzando un algoritmo nearest-neighbor implementato dalla funzione order_by_nn.

```
order_by_nn(left_cones);
order_by_nn(right_cones);
```

Per rappresentare graficamente i bordi, il centroide di ogni cono viene rappresentato da una sfera gialla e tra coni consecutivi viene generato un segmento verde.

```
track_viewer->addSphere(center, 0.03, 1, 1, 0, id);
track_viewer->addLine(a, b, 0, 1, 0, line_id);
```

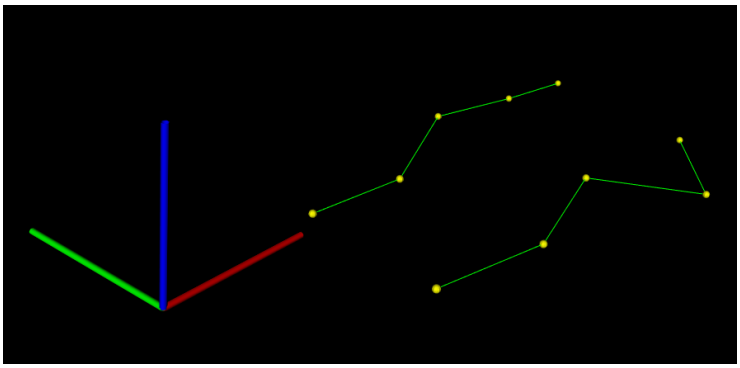

Figure 4: Visualizzazione del tracciato

6 Level 5: Odometry

L'ultimo livello della task richiede di implementare una stima della trasformazione tra due nuvole acquisite ad intervalli di tempo diversi, così permettendo di capire di quanto il soggetto si sia spostato e in che direzione. L'algoritmo usato è sempre ICP. Per ottenere una stima precisa le nuvole vengono preprocessate con la stessa pipline utilizzata in precedenza per il Cone Detection:

- Filtro pass-through sull'asse Z

- Rimozione dei piani
- Rimozione degli outlier
- Downsampling tramite VoxelGrid, per ridurre il numero di punti e accelerare l'allineamento.

Le nuvole vengono poi confrontate con ICP che calcola la rotazione e traslazione minimizzando la distanza tra i punti corrispondenti. I parametri utilizzati sono:

- Numero massimo di iterazioni: 50
- Massima distanza di corrispondenza: 1 metro

Se l'ICP converge si ottiene una matrice $4x4$ che rappresenta lo spostamento del veicolo

```
void odometry(const pcl::PointCloud<pcl::PointXYZ>::Ptr &
    first,
        const pcl::PointCloud<pcl::PointXYZ>::Ptr &second,
        int max_iteration,
        float max_correspond_distance){

    pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ>
        icp;
    icp.setMaximumIterations(max_iteration);
    icp.setMaxCorrespondenceDistance(max_correspond_distance
        );
    icp.setTransformationEpsilon(1e-8);
    icp.setEuclideanFitnessEpsilon(1e-6);
    icp.setInputSource(first);
    icp.setInputTarget(second);

    pcl::PointCloud<pcl::PointXYZ> aligned;
    icp.align(aligned);

    if(icp.hasConverged()){
        cout<<"ICP converged! Fitness score: " <<icp.
            getFitnessScore() <<endl;
        Eigen::Matrix4f transform = icp.
            getFinalTransformation();
        cout <<"Trasformazione stimata (odometria):\n" <<
            transform <<endl;
    }else{
        cout<<"ICP has not converged\n";
    }

}
```

Come possiamo dell'ultima colonna della matrice il soggetto si è spostato di $+1.13m$ sull'asse x, $-0.46m$ sull'asse y e $+1.03m$ sull'asse z.

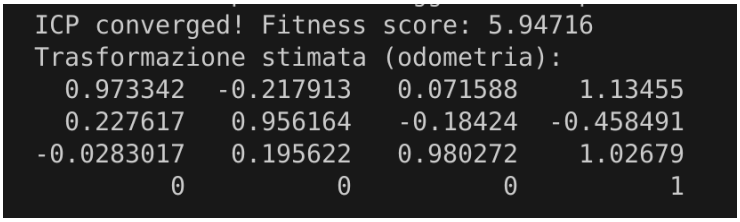

Figure 5: Matrice spostamento