Scan Matching Localization

#1. Project Goal:

Localization is the process by which an autonomous vehicle accurately identifies its current location and pose (position and direction). In other words, the vehicle continuously updates its location based on its surroundings (map data, map.pcd file).

1-1. Maintain pose error of 1.2m or less: The error between the current location and the calculated location should not exceed 1.2m while the vehicle is driving.

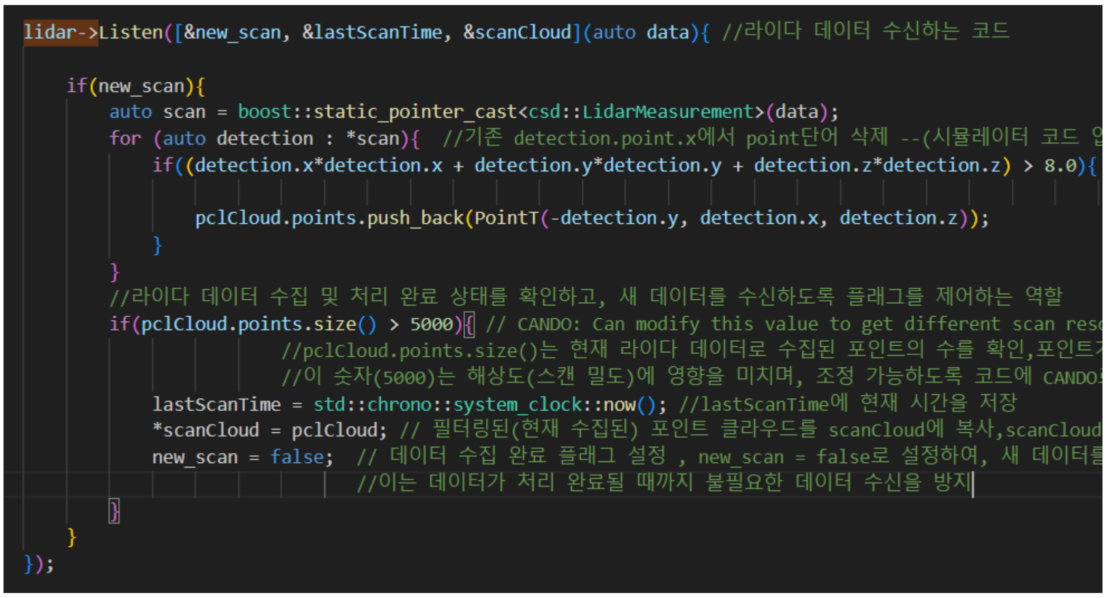
1-2. Drive more than 170m: The vehicle must drive at least 170m at medium speed.

1-3. Implement localization algorithm: You must write a code that calculates the location of the vehicle by filtering (voxel filter) and matching (ICP or NDT) Lidar data.

Lidar data: Collect point cloud data around the vehicle through sensors (provided by carla simulator) Map.pcd: Calculate the location (pose) of the vehicle by comparing the static point cloud map (static map data) stored in this file with the collected Lidar data.

#2. C3-main.cpp :

##2-1 Receiving Lidar Data: The following code receives the Lidar data generated from the Carla simulator.



```cpp
lidar->Listen([&new_scan, &lastScanTime, &scanCloud](auto data){ //라이다 데이터 수신하는 코드

    if(new_scan){
        auto scan = boost::static_pointer_cast<csd::LidarMeasurement>(data);
        for (auto detection : *scan){  //기존 detection.point.x에서 point단어 삭제 --(시뮬레이터 코드 업
            if((detection.x*detection.x + detection.y*detection.y + detection.z*detection.z) > 8.0){

                pclCloud.points.push_back(PointT(-detection.y, detection.x, detection.z));
            }
        }
        //라이다 데이터 수집 및 처리 완료 상태를 확인하고, 새 데이터를 수신하도록 플래그를 제어하는 역할
        if(pclCloud.points.size() > 5000){ // CANDO: Can modify this value to get different scan reso
                        //pclCloud.points.size()는 현재 라이다 데이터로 수집된 포인트의 수를 확인,포인트가
                        //이 숫자(5000)는 해상도(스캔 밀도)에 영향을 미치며, 조정 가능하도록 코드에 CANDO로
            lastScanTime = std::chrono::system_clock::now(); //lastScanTime에 현재 시간을 저장
            *scanCloud = pclCloud; // 필터링된(현재 수집된) 포인트 클라우드를 scanCloud에 복사,scanCloud
            new_scan = false;  // 데이터 수집 완료 플래그 설정 , new_scan = false로 설정하여, 새 데이터를
                                //이는 데이터가 처리 완료될 때까지 불필요한 데이터 수신을 방지
        }
    }
});
```

Lidar data is stored in a variable called pclCloud, copied to scanCloud and used later.

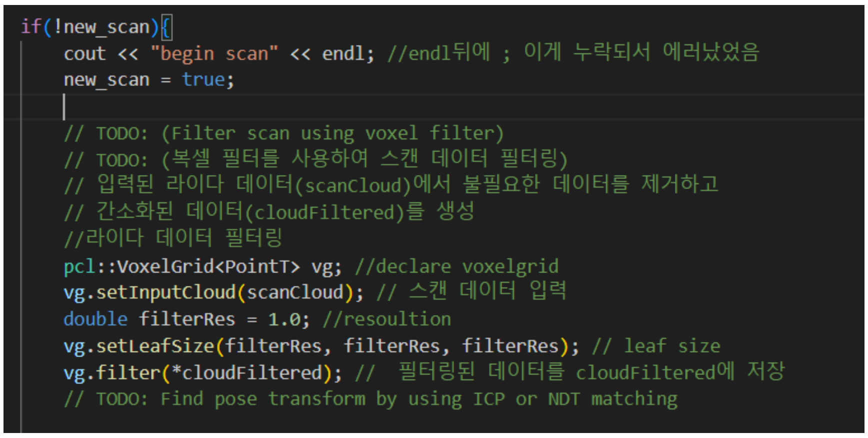
Return: Put pclcloud in scanCloud and return it!

##2-2 Improving the quality of lidar data:

Remove unnecessary noise using a voxel filter, perform calculations using filtered points improve processing speed!

You can check the filtering effect by changing the filterRes value. A lower value (more detailed filtering) maintains more details but increases processing time, while a higher value (more coarse filtering) reduces details but shortens processing time.

(Reducing the `filterRes` value from 1.0 to 0.5 increases the density of filtered point data, enabling more accurate matching. On the other hand, increasing the value to 2.0 makes the point data sparse but increases the calculation speed. This adjustment should be set appropriately depending on the driving environment.)



```
if(!new_scan){
    cout << "begin scan" << endl; //endl뒤에 ; 이게 누락되서 에러났었음
    new_scan = true;

    // TODO: (Filter scan using voxel filter)
    // TODO: (복셀 필터를 사용하여 스캔 데이터 필터링)
    // 입력된 라이다 데이터(scanCloud)에서 불필요한 데이터를 제거하고
    // 간소화된 데이터(cloudFiltered)를 생성
    //라이다 데이터 필터링
    pcl::VoxelGrid<PointT> vg; //declare voxelgrid
    vg.setInputCloud(scanCloud); // 스캔 데이터 입력
    double filterRes = 1.0; //resoultion
    vg.setLeafSize(filterRes, filterRes, filterRes); // leaf size
    vg.filter(*cloudFiltered); //  필터링된 데이터를 cloudFiltered에 저장
    // TODO: Find pose transform by using ICP or NDT matching
```

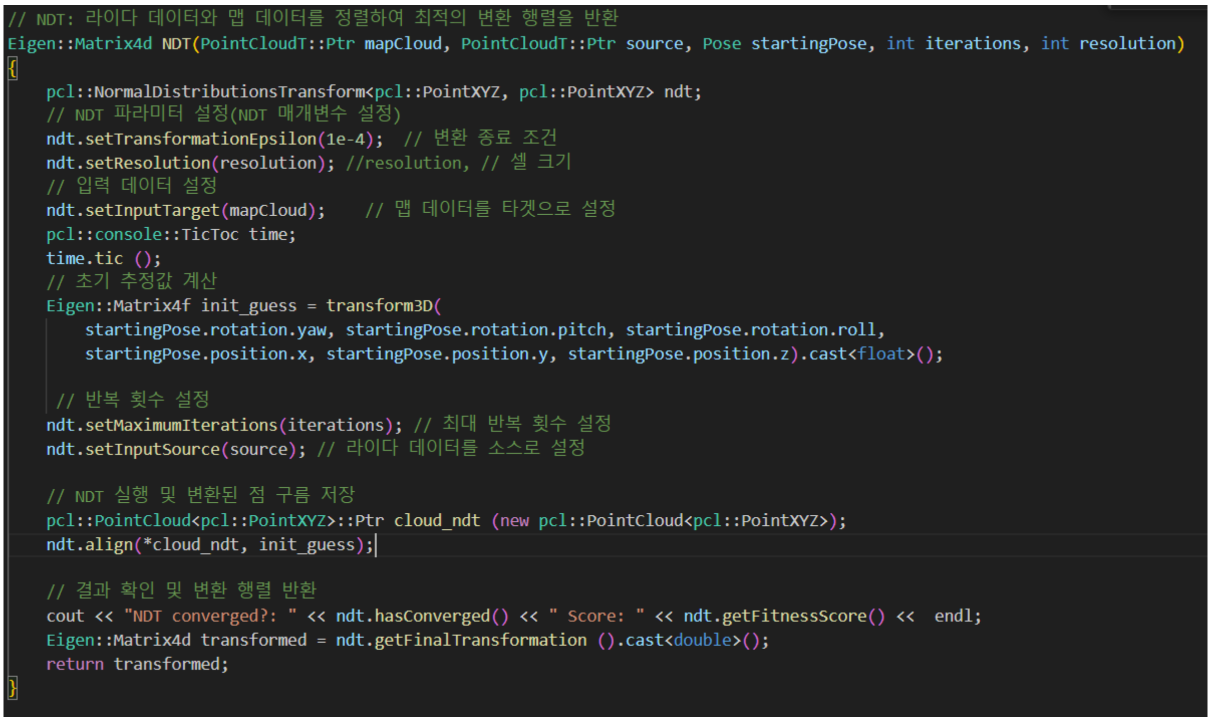
Return: Process scanCloud and return it in cloudFiltered.

##2-3 Accurate Scan Matching Technique: (NDT (Normal Distributions Transform)

The current location is estimated by comparing the lidar data with Map.pcd.

NDT and ICP are used to find the optimal transformation between the two scans (lidar scan and map data), and this file only uses NDT.

NDT models the map data as a probability density function, providing fast and stable matching.

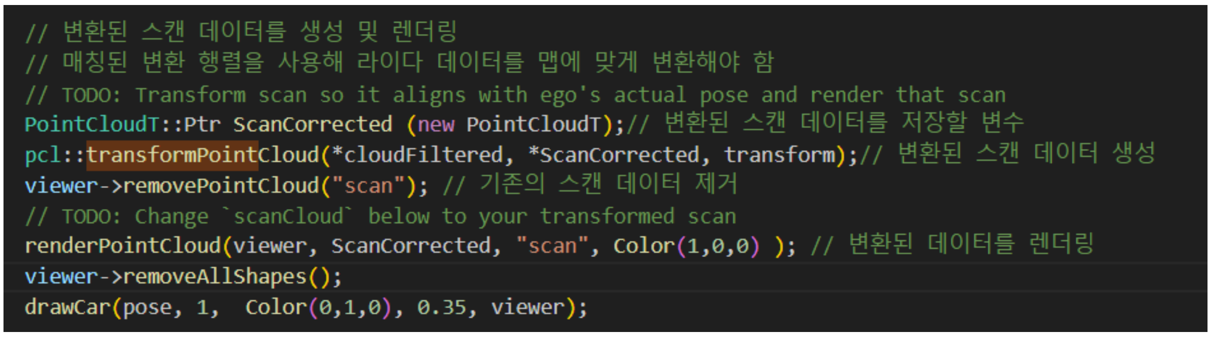


```
// NDT: 라이다 데이터와 맵 데이터를 정렬하여 최적의 변환 행렬을 반환
Eigen::Matrix4d NDT(PointCloudT::Ptr mapCloud, PointCloudT::Ptr source, Pose startingPose, int iterations, int resolution)
{
    pcl::NormalDistributionsTransform<pcl::PointXYZ, pcl::PointXYZ> ndt;
    // NDT 파라미터 설정(NDT 매개변수 설정)
    ndt.setTransformationEpsilon(1e-4);  // 변환 종료 조건
    ndt.setResolution(resolution); //resolution, // 셀 크기
    // 입력 데이터 설정
    ndt.setInputTarget(mapCloud);    // 맵 데이터를 타겟으로 설정
    pcl::console::TicToc time;
    time.tic ();
    // 초기 추정값 계산
    Eigen::Matrix4f init_guess = transform3D(
        startingPose.rotation.yaw, startingPose.rotation.pitch, startingPose.rotation.roll,
        startingPose.position.x, startingPose.position.y, startingPose.position.z).cast<float>();

     // 반복 횟수 설정
    ndt.setMaximumIterations(iterations); // 최대 반복 횟수 설정
    ndt.setInputSource(source); // 라이다 데이터를 소스로 설정

    // NDT 실행 및 변환된 점 구름 저장
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_ndt (new pcl::PointCloud<pcl::PointXYZ>);
    ndt.align(*cloud_ndt, init_guess);

    // 결과 확인 및 변환 행렬 반환
    cout << "NDT converged?: " << ndt.hasConverged() << " Score: " << ndt.getFitnessScore() <<  endl;
    Eigen::Matrix4d transformed = ndt.getFinalTransformation ().cast<double>();
    return transformed;
}
```

##2-4 Scan Transformation and Rendering:

The lidar data needs to be transformed to fit the map using the matching transformation matrix.

The transformed data is appropriately rendered to visually compare the estimated and actual positions of the vehicle.



```
// 변환된 스캔 데이터를 생성 및 렌더링
// 매칭된 변환 행렬을 사용해 라이다 데이터를 맵에 맞게 변환해야 함
// TODO: Transform scan so it aligns with ego's actual pose and render that scan
PointCloudT::Ptr ScanCorrected (new PointCloudT);// 변환된 스캔 데이터를 저장할 변수
pcl::transformPointCloud(*cloudFiltered, *ScanCorrected, transform);// 변환된 스캔 데이터 생성
viewer->removePointCloud("scan"); // 기존의 스캔 데이터 제거
// TODO: Change `scanCloud` below to your transformed scan
renderPointCloud(viewer, ScanCorrected, "scan", Color(1,0,0) ); // 변환된 데이터를 렌더링
viewer->removeAllShapes();
drawCar(pose, 1,  Color(0,1,0), 0.35, viewer);
```

Return: Transform the cloudFiltered data using the calculated transformation matrix transform, and save the result in scanCorrected. The data saved in ScanCorrected is finally rendered on the screen via renderPointCloud.

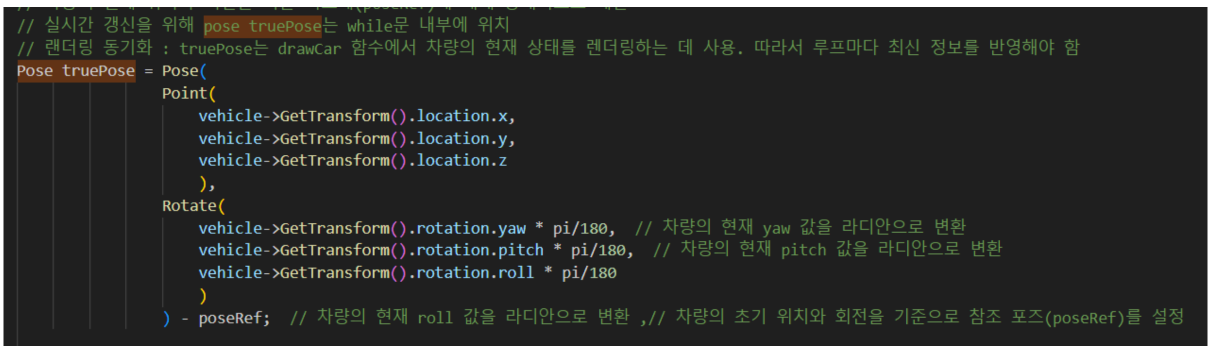
##2-5 Differences between Initialization and Repeat and Their Performance Impact

Pose Initialization: Sets the initial position and pose of the vehicle, and defines a reference pose for tracking the vehicle's movements.

```
Pose pose(Point(0,0,0), Rotate(0,0,0)); //초기포즈 정의
```
Simply define the initial value

```
Pose poseRef(Point(vehicle->GetTransform().location.x,
                vehicle->GetTransform().location.y,
                vehicle->GetTransform().location.z),
                Rotate(vehicle->GetTransform().rotation.yaw * pi/180,
                vehicle->GetTransform().rotation.pitch * pi/180,
                vehicle->GetTransform().rotation.roll * pi/180));
```
Based on the initial value, specify the initial state of the vehicle in the real environment.

poseRef: Set based on the **initial position and posture (pose)** of the vehicle, and subsequent movements or changes in position are calculated relatively based on this poseRef.


```
// 실시간 갱신을 위해 pose truePose는 while문 내부에 위치
// 랜더링 동기화 : truePose는 drawCar 함수에서 차량의 현재 상태를 렌더링하는 데 사용. 따라서 루프마다 최신 정보를 반영해야 함
Pose truePose = Pose(
                Point(
                    vehicle->GetTransform().location.x,
                    vehicle->GetTransform().location.y,
                    vehicle->GetTransform().location.z
                    ),
                Rotate(
                    vehicle->GetTransform().rotation.yaw * pi/180,  // 차량의 현재 yaw 값을 라디안으로 변환
                    vehicle->GetTransform().rotation.pitch * pi/180,  // 차량의 현재 pitch 값을 라디안으로 변환
                    vehicle->GetTransform().rotation.roll * pi/180
                    )
                ) - poseRef;  // 차량의 현재 roll 값을 라디안으로 변환 ,// 차량의 초기 위치와 회전을 기준으로 참조 포즈(poseRef)를 설정
```

In the while loop, truePose is added to update the current position of the vehicle each time, which calculates the relative position based on the initialized `poseRef`. This repeated update enables real-time tracking of the vehicle and accurate positioning.

#3. Speed Control:

If the vehicle speed is too fast, the lidar data may not be collected enough, making accurate matching difficult. Use the arrow keys appropriately to maintain a medium speed!

4. Core Tasks:

The core of the project is to determine the vehicle's position based on the relative match between the static map stored in map.pcd and the lidar data, not based on the position on the Earth such as GPS.

Therefore, this project is:

1.Refining the lidar data.

2.Comparison with map.pcd (scan matching).

3.Calculating the optimal transformation (NDT or ICP).:Only the NDT function is implemented here.

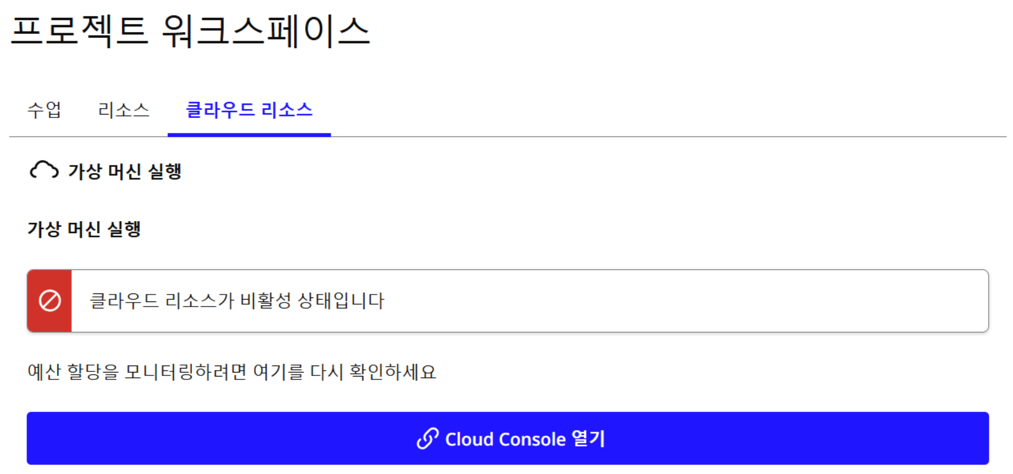
4.Maintaining the system stability so that the vehicle can continuously estimate the position while driving.

5.Updating the vehicle position through pose setting

The goal of this process is to minimize the error between the actual position and the estimated position.

#5. Execution Environment:

This project uses Udacity's Ubuntu Cloud environment.


프로젝트 워크스페이스

가상 머신 실행

클라우드 리소스가 비활성 상태입니다

예산 할당을 모니터링하려면 여기를 다시 확인하세요

First console window:

First, you need to do git clone in Ubuntu environment.

Git clone https://github.com/udacity/nd0013_cd2693_Exercise_Starter_Code

Check the cloned directory and move to the path below.

```
cd nd0013_cd2693_Exercise_Starter_Code/Lesson_7_Project_Scan_Matching_Localization/c3-project
```

Use the ls command to check if the following files exist:

```
├──── CMakeLists.txt
├──── README.md
├──── c3-main.cpp
├──── helper.cpp
├──── helper.h
├──── make-libcarla-install.sh
├──── map.pcd
├──── map_loop.pcd
├──── rpclib
└──── run_carla.sh
```

Now run the command below

```
chmod +x make-libcarla-install.sh
```

Then run the command below

```
./make-libcarla-install.sh
```

Update the code of the C3-main.cpp file.

Enter nano c3-main.cpp at the prompt to work in the nano editor.

In the nano editor, press Ctrl key + Shift Key + K key to delete one line at a time and paste.

Since you cannot paste the entire code at once, you have to do it in parts.

When finished, press ctrl+o to save and press enter to exit.

Now run the command below

```
cmake .
```

Once the execution is complete, also run the command below.

```
make
```

Second console window:

To use Scan Matching Localisation, the Carla simulator must be running in the background.

Launch a new terminal window and type the following:

This is the command to run the Carla simulator.

```
./run_carla.sh
```

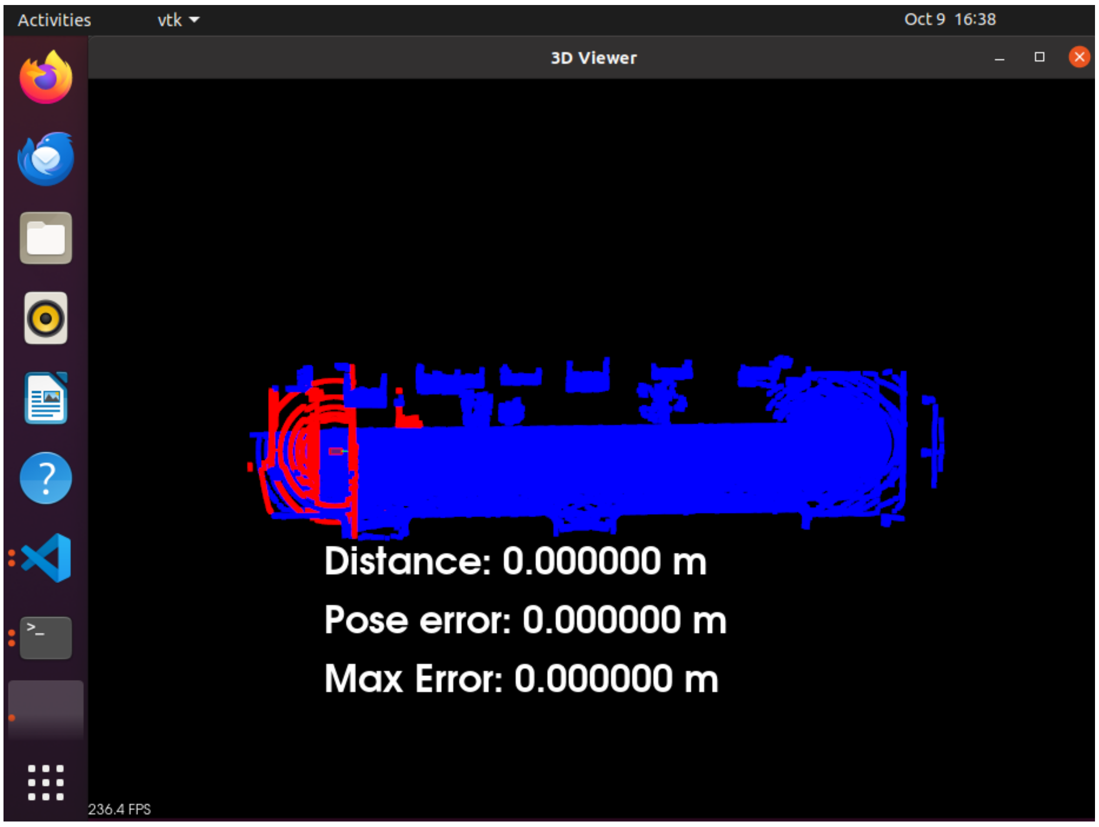
After a few minutes you should see the message Disabling core dumps, you should be able to launch a third console window.

Third console window:

Open another terminal window. Then type the following command:

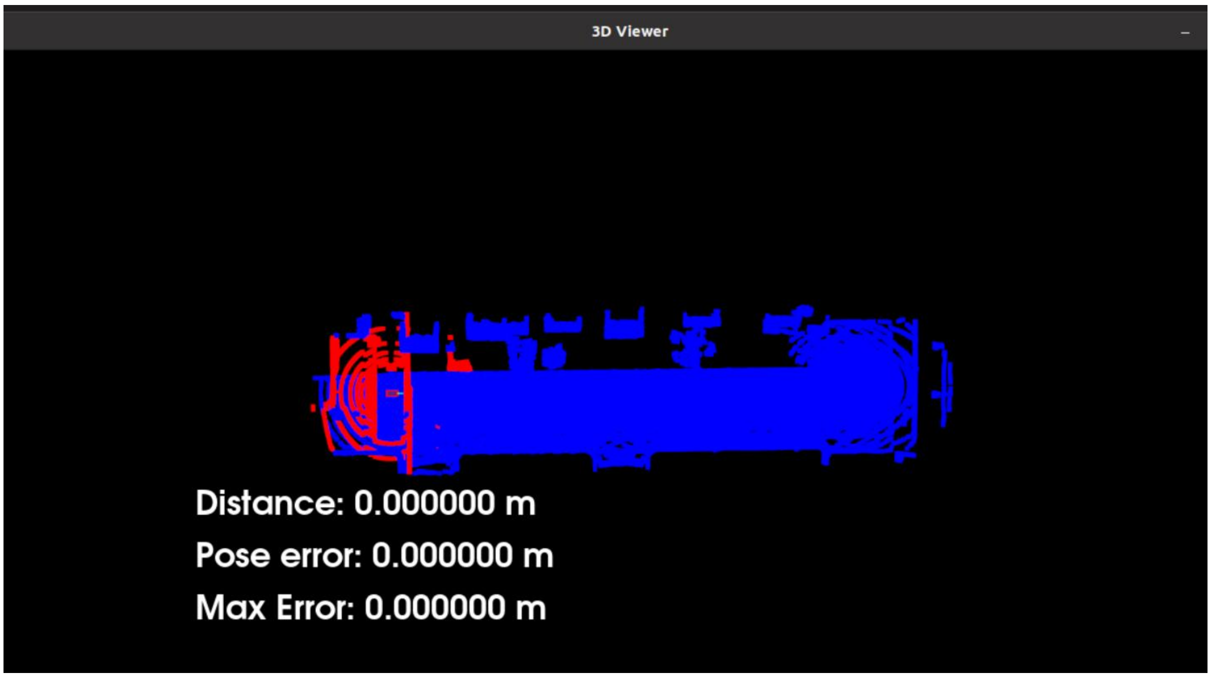
```
./cloud_loc
```

The Carla simulator will now start running



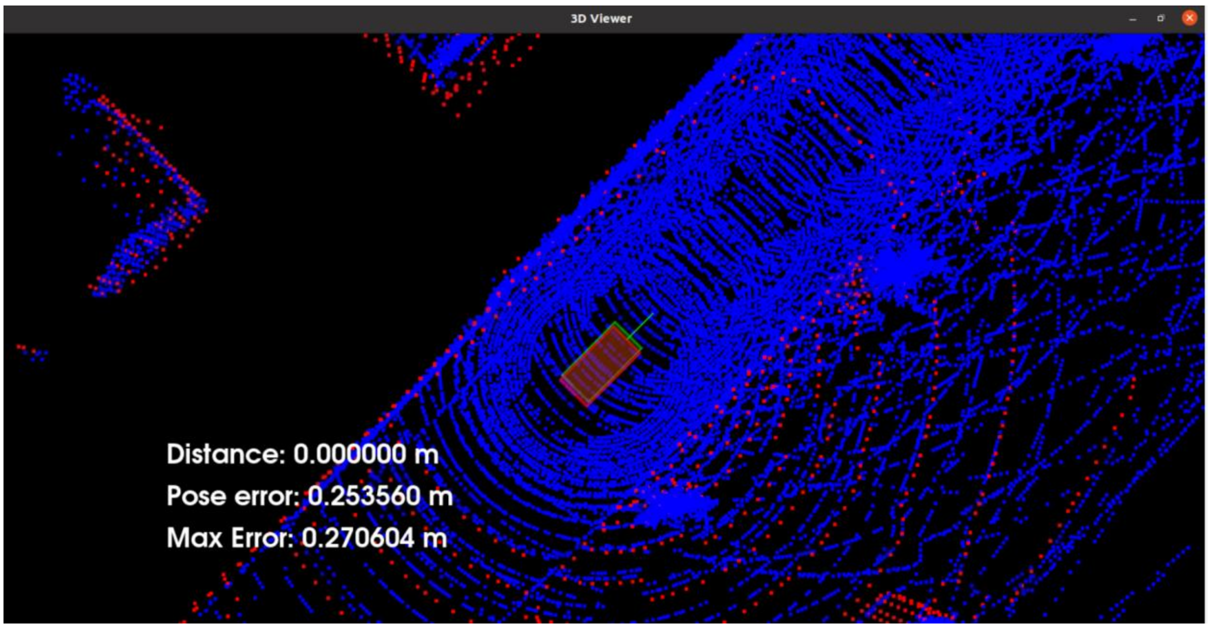
Distance: 0.000000 m
Pose error: 0.000000 m
Max Error: 0.000000 m

#6. Execution results:



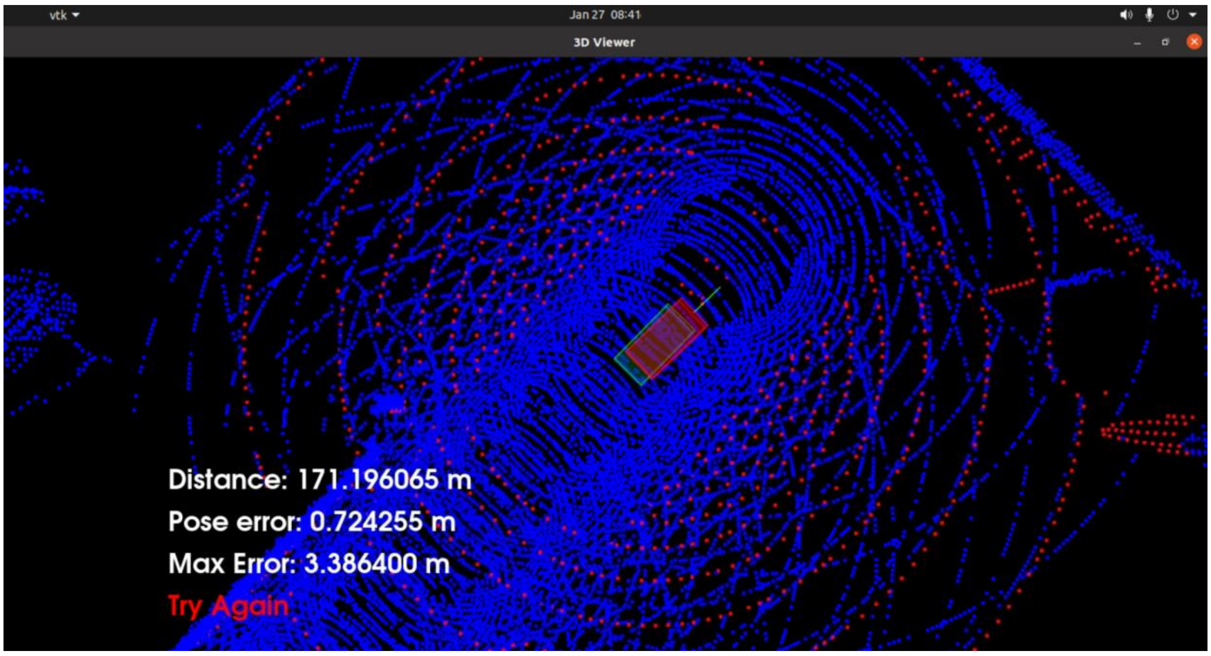
Distance: 0.000000 m
Pose error: 0.000000 m
Max Error: 0.000000 m

Press the a key to set the top view



Distance: 0.000000 m
Pose error: 0.253560 m
Max Error: 0.270604 m

Press the up arrow key three times to start driving at medium speed



Distance: 171.196065 m
Pose error: 0.724255 m
Max Error: 3.386400 m
Try Again

The error is 3.38m when driving over 170m. It is out of the project compliance criteria.

I have not yet identified the cause, whether it is a simulator problem or a vehicle speed problem.

I will try to find ways to improve it in the future!