Scan Matching Localization (한국어버전 Writeup)

#1.프로젝트 목표 :

현지화(Localization)은 자율주행 차량이 자신의 현재위치와 자세(Pose:위치와 방향)을 정확히 파악하는 과정. 즉, 차량은 자신의 주변환경(맵데이터,map.pcd파일)을 기반으로 자신의 위치를 지속적으로 업데이트 합니다.

1-1.포즈 오차 1.2m 이하 유지: 차량이 주행하는 동안 현재 위치와 계산된 위치 간의 오차가 1.2m를 넘지 않아야 합니다.

1-2.170m 이상 주행: 차량을 중속으로 최소 170m 주행해야 합니다.

1-3.현지화 알고리즘 구현: Lidar 데이터를 필터링(복셀 필터), 매칭(ICP 또는 NDT)하여 차량의 위치를 계산하는 코드를 작성해야 합니다.

Lidar 데이터: 센서를 통해 차량 주변의 포인트 클라우드 데이터를 수집 (carla시뮬레이터 제공)
Map.pcd : 이 파일에 저장된 정적인 포인트 클라우드 맵(정적인 지도 데이터)과 수집한 라이다 데이터를 비교하여 차량의 위치 (포즈)를 계산

#2. C3-main.cpp :

##2-1 라이다 데이터 수신: Carla시뮬레이터에서 생성되는 라이다 데이터를 하기의 코드가 수신

```
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 resc
                        //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 CANDO:
            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
        }
    }
});
```

라이다 데이터는 pclCloud라는 변수에 저장되며, scanCloud로 복사되어 뒤에 사용됩니다.

Return : pclcloud를 scanCloud에 담아 리턴!

##2-2 라이다 데이터의 품질향상:

복셀필터(voxel filter)를 사용해 불필요한 노이즈를 제거, 필터링된 포인트를 사용해 계산수행 → 처리속도를 개선!

filterRes 값을 변경해가며 필터링 효과를 확인할 수 있습니다. 낮은 값(더 세밀한 필터링)은 더 많은 세부 정보를 유지하지만 처리 시간이 증가하고, 높은 값(더 거친 필터링)은 세부 정보를 줄이는 대신 처리 시간을 단축시킵니다.

(`filterRes` 값을 1.0에서 0.5로 줄이면 필터링된 점 데이터의 밀도가 증가해 더 정확한 매칭이 가능합니다. 반면, 값이 2.0으로 증가하면 점 데이터는 희소해지지만 계산 속도는 빨라집니다. 이러한 조정은 주행 환경에 따라 적절히 설정해야 합니다.)

```
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: scanCloud를 처리하여 cloudFiltered에 담아서 리턴

##2-3 정확한 스캔매칭기법: (NDT (Normal Distributions Transform)☺

라이다 데이터와 Map.pcd간의 비교를 통해서 현재위치를 추정합니다.

NDT와 ICP는 두 스캔(라이다 스캔과 맵 데이터) 간의 최적 변환을 찾는 데 사용되며 이 파일은 NDT만 사용합니다.

NDT로 맵 데이터를 확률 밀도 함수로 모델링하여 빠르고 안정적인 매칭을 제공합니다.

```
// 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 스캔변환 및 렌더링:

매칭된 변환 행렬을 사용해 라이다 데이터를 맵에 맞춰 변환해야 합니다.

변환된 데이터를 적절히 렌더링하여 차량의 추정 위치와 실제 위치를 시각적으로 비교합니다.

```
// 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: 계산된 변환행렬 transform을 이용하여 cloudFiltered 데이터를 변환하고, 결과를 scanCorrected에 저장

ScanCorrected에 저장된 데이터는 최종적으로 renderPointCloud를 통해 화면에 랜더링 됨.

##2-5 초기화와 반복의 차이점 및 이로인한 성능영향

포즈 초기화: 차량의 초기 위치와 자세를 설정하며, 이후 차량의 움직임을 추적하기 위한 참조 포즈를 정의합니다.

```
Pose pose(Point(0,0,0), Rotate(0,0,0));
```
초기값을 단순히 정의

```
    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));
```
초기값을 기준으로 실제 환경에서 차량의 초기 상태를 구체화.

poseRef : 차량의 **초기 위치와 자세(포즈)**를 기준으로 설정, 이후의 움직임이나 위치 변화는 이 poseRef를 기준으로 상대적으로 계산

```
// 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
```

while문안에서는 truePose가 추가되어 있어서 차량의 현재 위치를 매번 업데이트하며, 이는 초기화된 `poseRef`를 기준으로 상대적인 위치를 계산합니다. 이러한 반복 갱신은 차량의 실시간 추적과 정확한 위치 파악을 가능하게 합니다.

#3. 속도 조절 :

차량 속도가 너무 빠르면 라이다 데이터가 충분히 수집되지 않아 정확한 매칭이 어려워질 수 있습니다. 화살표 키를 적절히 사용해 중속을 유지!

4. 핵심 과제:

GPS와 같은 지구에서의 위치를 기준으로 한 것이 아니고, map.pcd에 저장된 정적인 맵과 라이다 데이터 간의 상대적인 일치를 기반으로 차량의 위치를 파악하는 것이 프로젝트의 핵심입니다.

따라서 이 프로젝트는:

1.라이다 데이터의 정제.

2.map.pcd와의 비교(스캔 매칭).

3.최적의 변환 계산(NDT 또는 ICP).:여기선 NDT함수만 구현

4.차량이 주행하면서도 지속적으로 위치를 추정할 수 있도록 시스템 안정성 유지.

5.Pose설정을 통한 차량위치 갱신

이 과정을 통해 실제 위치와 추정 위치 간의 오차를 최소화하는 것이 목표입니다.

#5. 실행 환경 :

이 프로젝트는 Udacity의 우분투 클라우드 환경을 사용합니다.

첫 번째 콘솔 창 :

먼저 우분투 환경에서는 git clone을 진행해야 합니다.

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

Clone된 디렉토리를 확인 후 아래의 경로로 이동합니다

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

ls명령어로 하기의 파일들이 있는지 확인합니다

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

이제 아래의 명령어를 실행합니다

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

그 다음 아래의 명령어를 실행합니다

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

C3-main.cpp파일의 코드를 Update해줍니다.

프롬프트에서 nano c3-main.cpp를 입력하여 nano에디터에서 작업합니다.

Nano 에디터에서는 Ctrl key + shift Key + K key를 눌러서 한 행씩 삭제 후 paste 해줍니다.

Paste도 code전체는 한번에 안되기에 나누어서 하셔야 합니다.

완료시 crtl+o를 눌러서 저장하고 enter눌러서 나옵니다.

이제 아래의 명령어를 실행합니다

```
cmake .
```

실행이 완료되면 아래의 명령어도 실행해 줍니다

```
make
```

두 번째 콘솔 창 :

Scan Matching Localisation을 사용하려면 Carla 시뮬레이터가 백그라운드에서 실행되어야 합니다.

새로운 터미널 창을 실행하고 아래와 같이 입력합니다

이 명령어는 Carla시뮬레이터를 실행하는 명령어입니다

```
./run_carla.sh
```

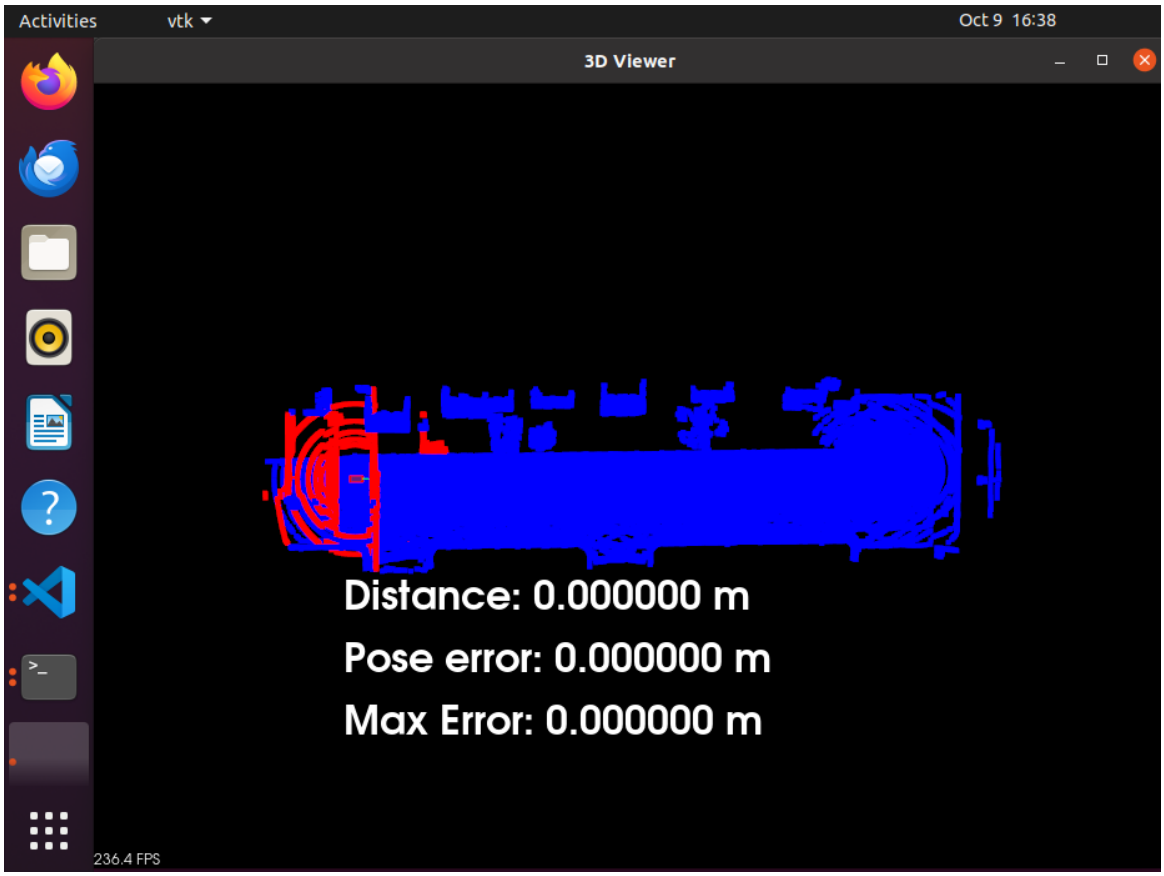
Disabling core dumps라는 메시지가 나온 후 몇분뒤에 세번째 콘솔창을 실행하셔야 합니다.

세번째 콘솔 창 :

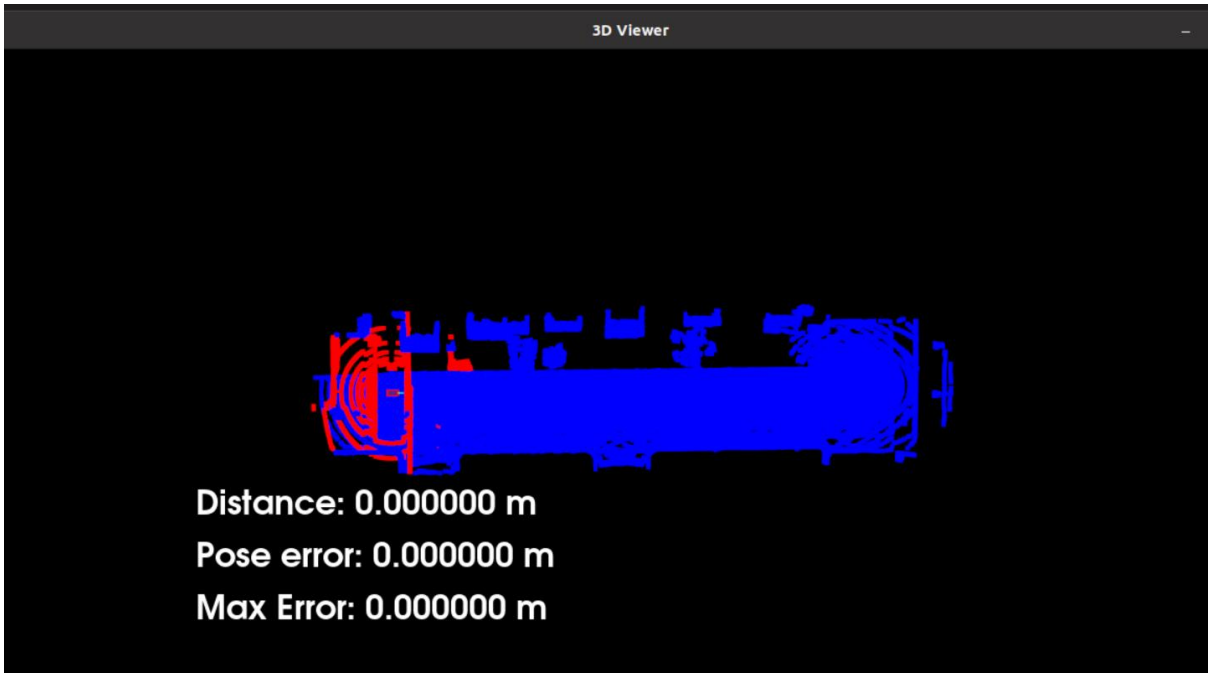
터미널창을 하나 더 열어줍니다. 그리고 아래와 같은 명령어를 입력합니다

```
./cloud_loc
```

이제 Carla 시뮬레이터가 실행되기 시작합니다

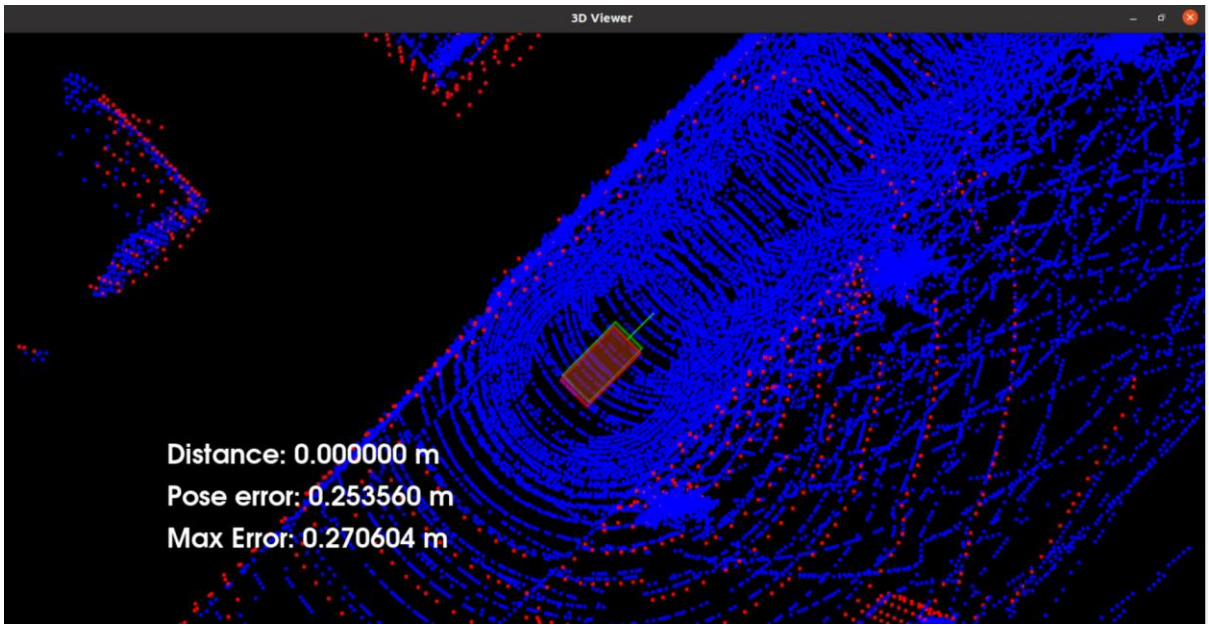


#6. 실행 결과 :

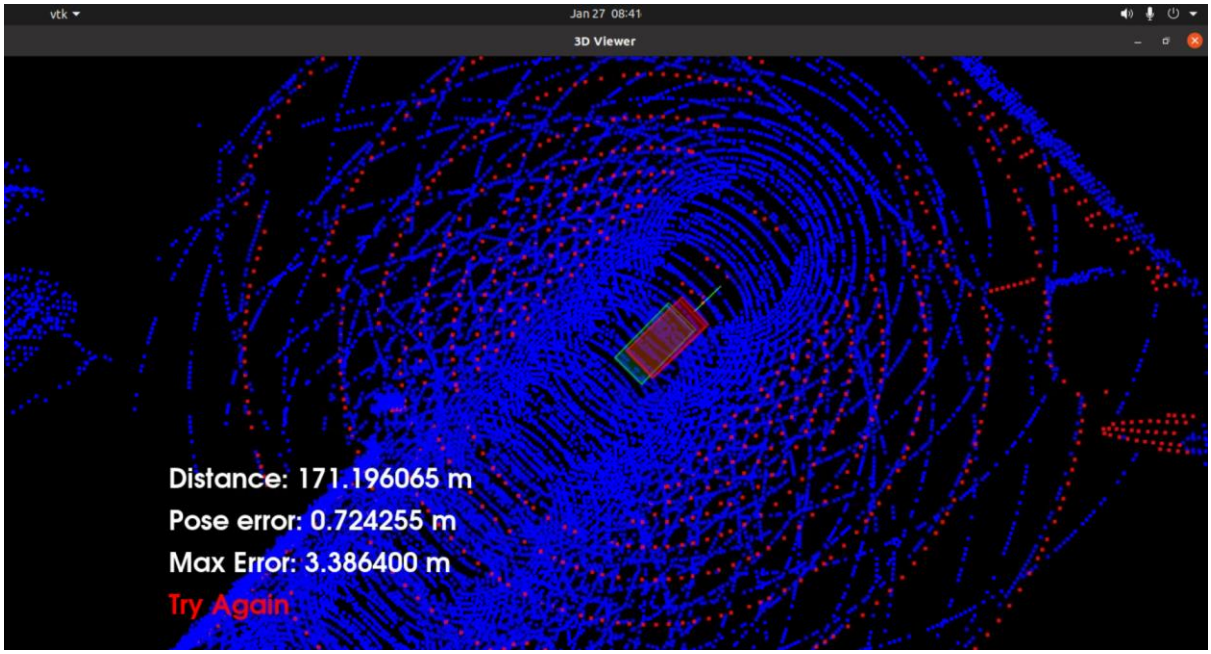


a키를 눌러서 top view 설정

-



화살표(up)키를 3번 눌러서 중간속도로 주행시작



170미터이상 주행시 오차는 3.38m임. 프로젝트 충족기준을 벗어남.

현재 시뮬레이터문제 또는 차량속도 문제인지 원인을 아직 파악하지 못했습니다.

향후 개선 방안을 찾아보려고 합니다!