Hackathon GDRA

Challenge 2 – Obstacle detection

Riccardo Bertoglio

AIRLab team

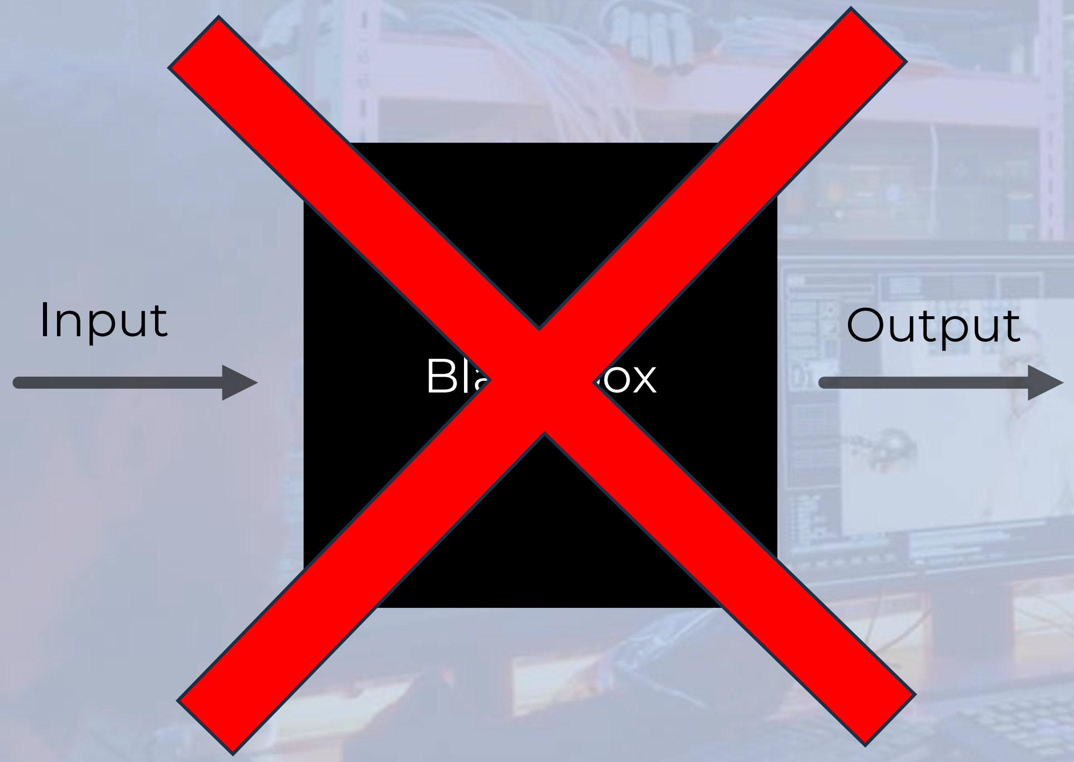
Outline

- System characteristics
- Elaboration pipeline
- Conclusions and future works

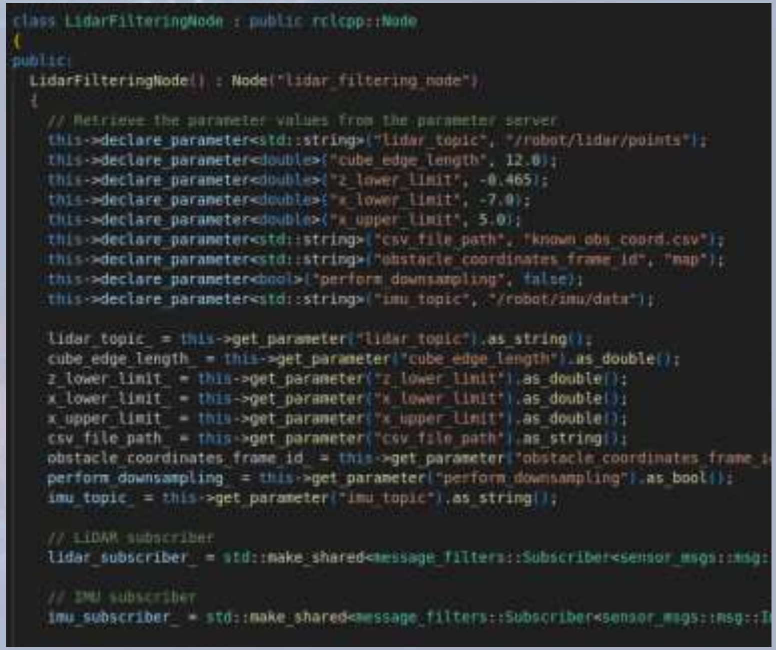
Outline

- System characteristics
- Elaboration pipeline
- Conclusions and future works

Safety in agricultural robotics needs explainable, reliable, and robust systems

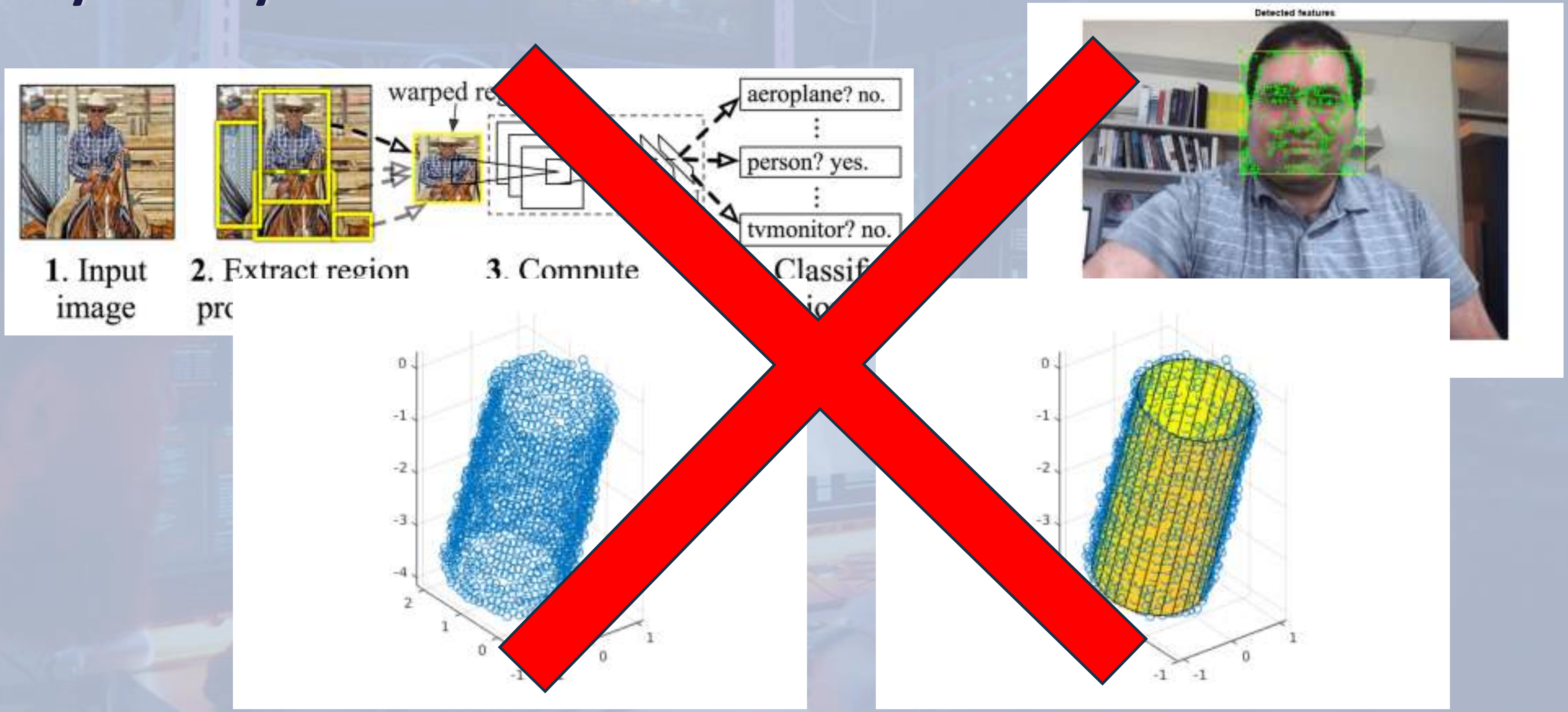


The code is totally transparent and explainable

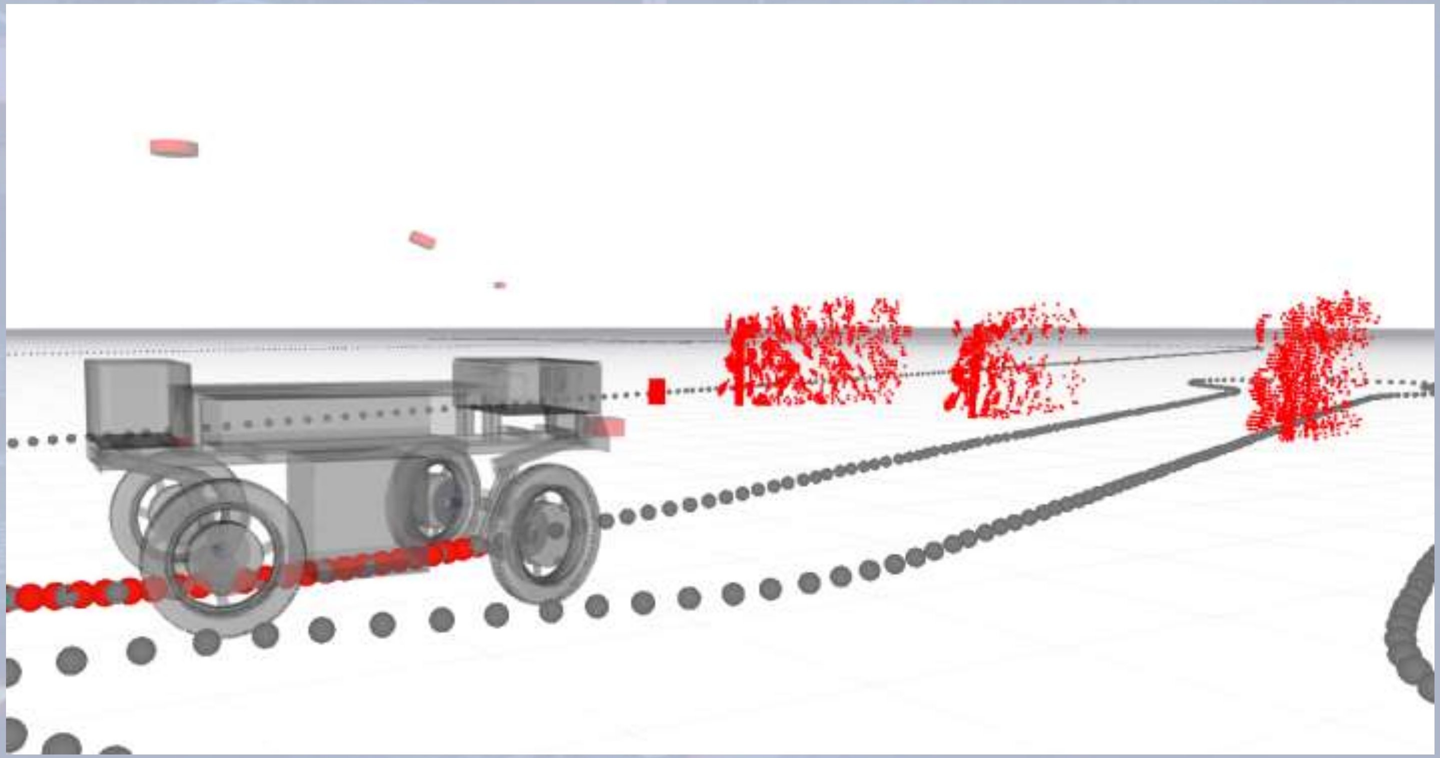


```
class LidarFilteringNode : public rclcpp::Node
{
public:
  LidarFilteringNode() : Node("lidar_filtering_node")
  {
    // Retrieve the parameter values from the parameter server
    this->declare_parameter<std::string>("lidar_topic", "/robot/lidar/points");
    this->declare_parameter<double>("cube_edge_length", 12.0);
    this->declare_parameter<double>("z_lower_limit", -0.465);
    this->declare_parameter<double>("x_lower_limit", -7.0);
    this->declare_parameter<double>("x_upper_limit", 5.0);
    this->declare_parameter<std::string>("csv_file_path", "known_obs_coord.csv");
    this->declare_parameter<std::string>("obstacle_coordinates_frame_id", "map");
    this->declare_parameter<bool>("perform_downsampling", false);
    this->declare_parameter<std::string>("imu_topic", "/robot/imu/data");

    lidar_topic_ = this->get_parameter("lidar_topic").as_string();
    cube_edge_length_ = this->get_parameter("cube_edge_length").as_double();
    z_lower_limit_ = this->get_parameter("z_lower_limit").as_double();
    x_lower_limit_ = this->get_parameter("x_lower_limit").as_double();
    x_upper_limit_ = this->get_parameter("x_upper_limit").as_double();
    csv_file_path_ = this->get_parameter("csv_file_path").as_string();
    obstacle_coordinates_frame_id_ = this->get_parameter("obstacle_coordinates_frame_i
    perform_downsampling_ = this->get_parameter("perform_downsampling").as_bool();
    imu_topic_ = this->get_parameter("imu_topic").as_string();

    // LiDAR subscriber
    lidar_subscriber_ = std::make_shared<message_filters::Subscriber<sensor_msgs::msg:

    // IMU subscriber
    imu_subscriber_ = std::make_shared<message_filters::Subscriber<sensor_msgs::msg::I
```

To build a versatile system that can detect any obstacle we did not rely on any kind of features



We use point clouds to detect aggregations of points that are near the robot



We use a 3D LiDAR to get point clouds since they are more accurate and robust than RGBD / stereo cameras

Depth Accuracy

Stereo vision uses triangulation to estimate depth from a disparity image, with the following formula describing how depth resolution changes over the range of a stereo camera:

`Dr=Z^2*alpha`, where `Dr` is depth resolution, `Z` the distance and `alpha` a constant.

Depth accuracy decreases quadratically over the z-distance, with a stereo depth accuracy of 1% of the distance in the near range to 9% in the far range. Depth accuracy can also be affected by outliers' measurements on homogenous and textureless surfaces such as white walls, green screens and specular areas. These surfaces usually generate temporal instability in the depth measurements.

Camera blindness

We use a 32 channels LiDAR with realistic configuration

```
type: 3D
minimal_azimut_angle: -125.0  # in deg
maximal_azimut_angle: 125.0  # in deg
azimut_angle_increment: [0.25, 0.125, 0.0625]  # in deg
azimut_angle_std: 0.0
samples: [1001, 2001, 4001]  # number of samples = (maxi
minimal_elevation_angle: -4.0  # in deg
maximal_elevation_angle: 8.09  # in deg
elevation_angle_increment: 0.39  # in deg
elevation_angle_std: 0.0
lasers: 32  # number of laser = (maximal_elevation_angle
minimal_range: 0.5  # in meter
maximal_range: 7.0  # in meter
range_std: 0.0
rate: 20
```

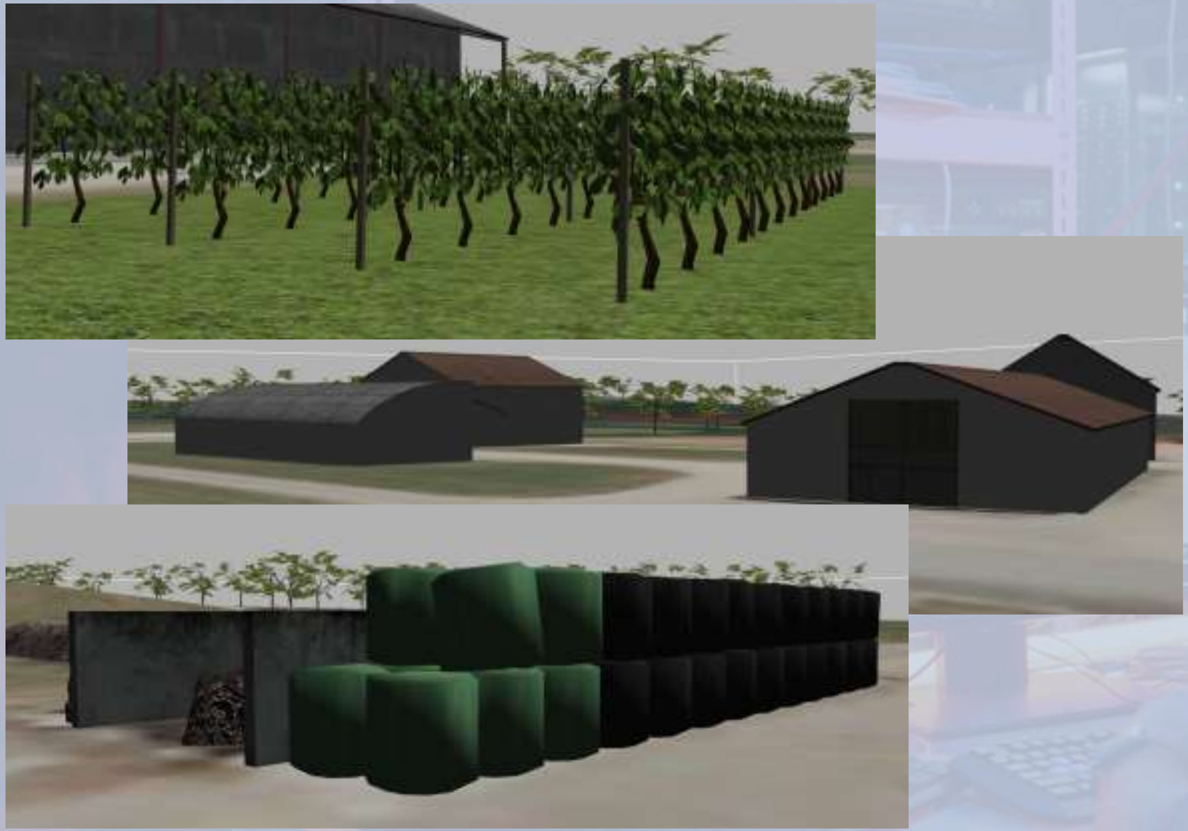
Specifications

- Channels: 32
- Measurement Range: 200 m
- Range Accuracy: Up to ±3 cm (Typical)[1]
- Horizontal Field of View: 360°
- Vertical Field of View: 40° (-25° to +15°)
- Minimum Angular Resolution (Vertical): 0.33° (non-linear distribution)
- Angular Resolution (Horizontal/Azimuth): 0.1° to 0.4°
- Rotation Rate: 5 Hz to 20 Hz
- Integrated Web Server for Easy Monitoring and Configuration

Velodyne VLP-32

We filter out known obstacles to focus only on unknown obstacles

Known obstacles

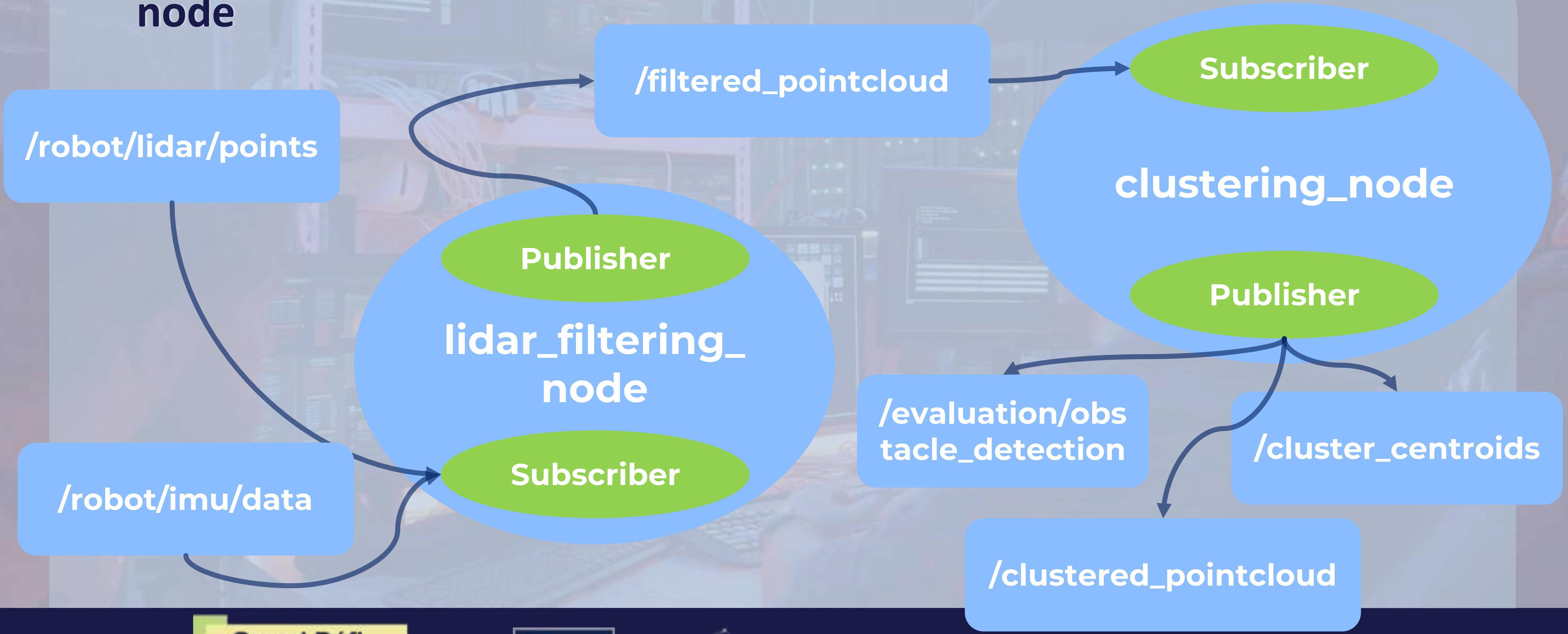


Unknown obstacles

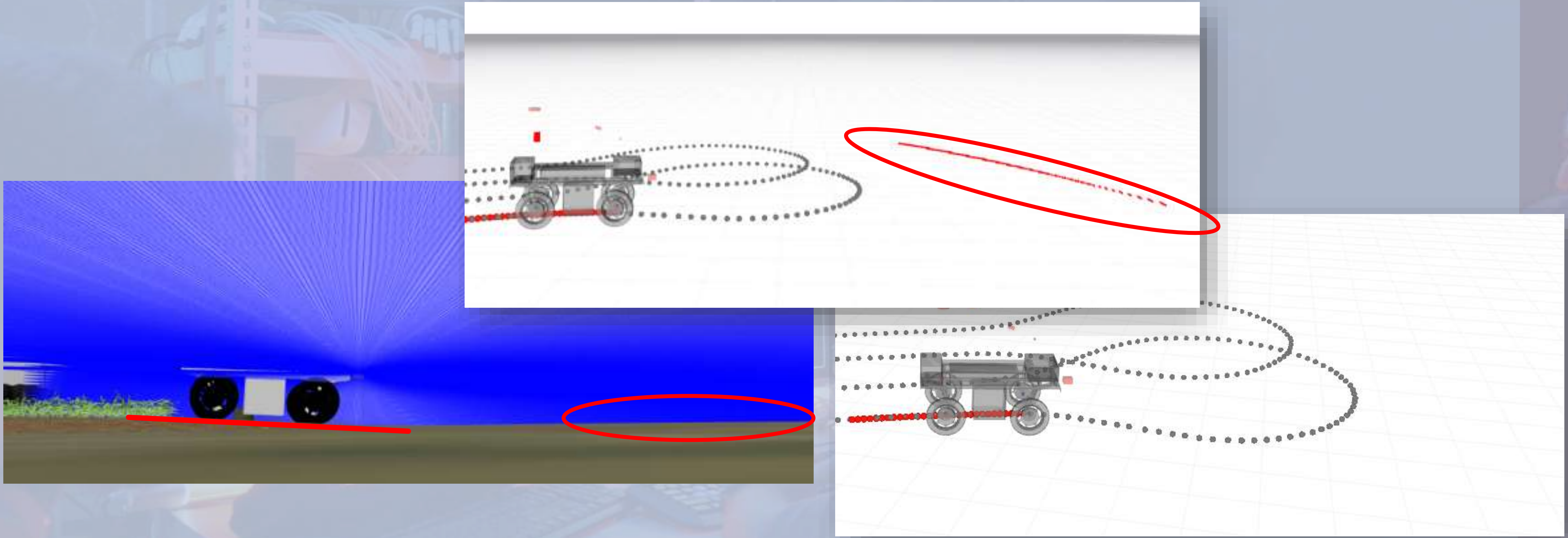
Outline

- System characteristics
- Elaboration pipeline
- Conclusions and future works

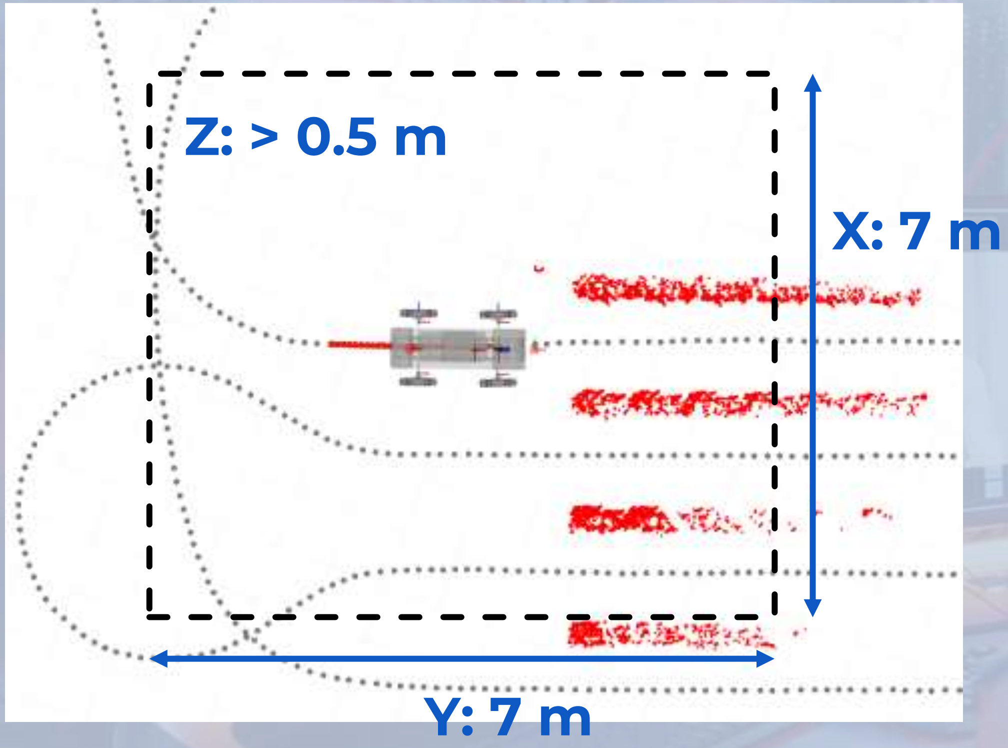
The system is composed of two nodes – a filtering and a clustering node



The point cloud is first rotated with IMU data so that it is parallel to the ground to avoid ground detection



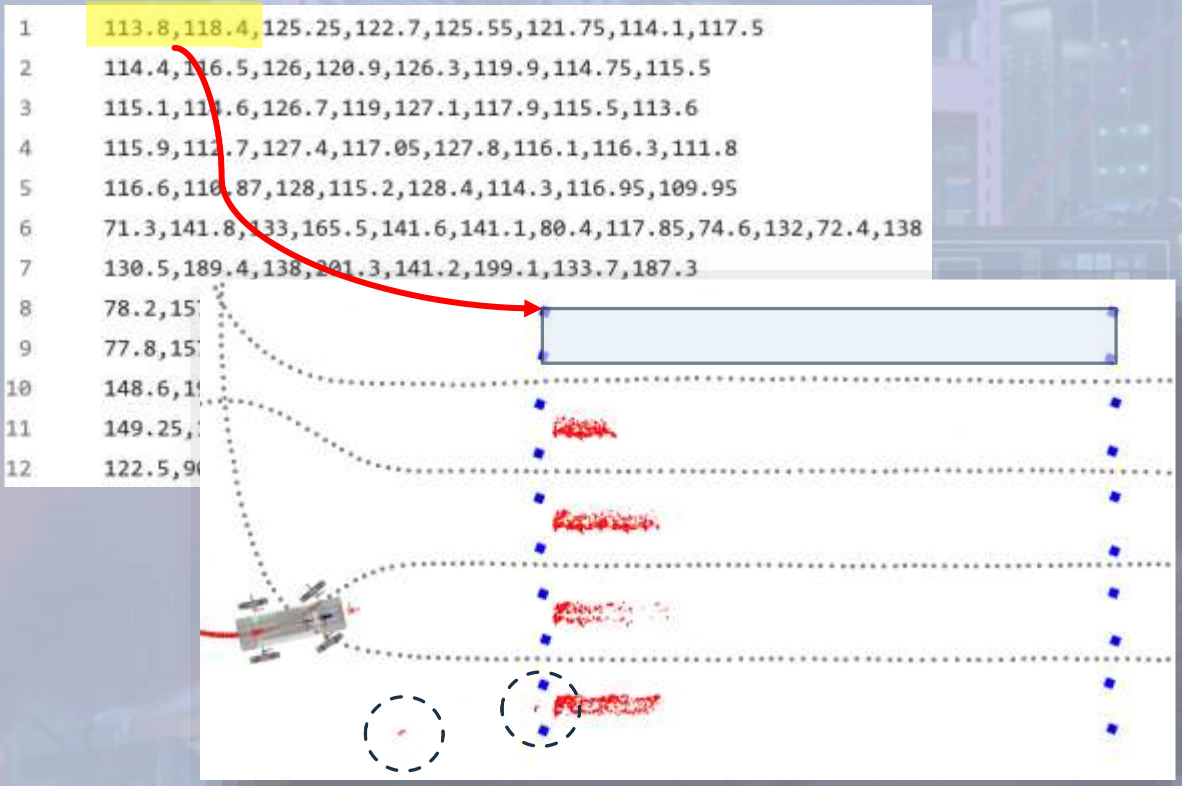
The point cloud is filtered on the three axes (X, Y, Z) and downsampled

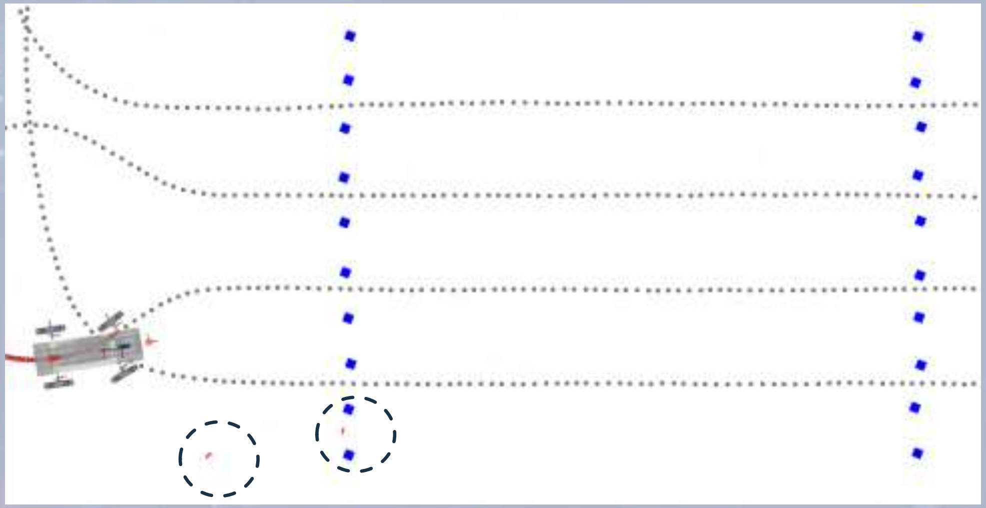


Optional downsampling

```
if (perform_downsampling_) {
    pcl::VoxelGrid<PointType> sor;
    sor.setInputCloud(filtered_cloud);
    sor.setLeafSize(0.1f, 0.1f, 0.1f);
    sor.filter(*downsampled_cloud);
```

We then filter out known obstacles by using polygons stored in a CSV file

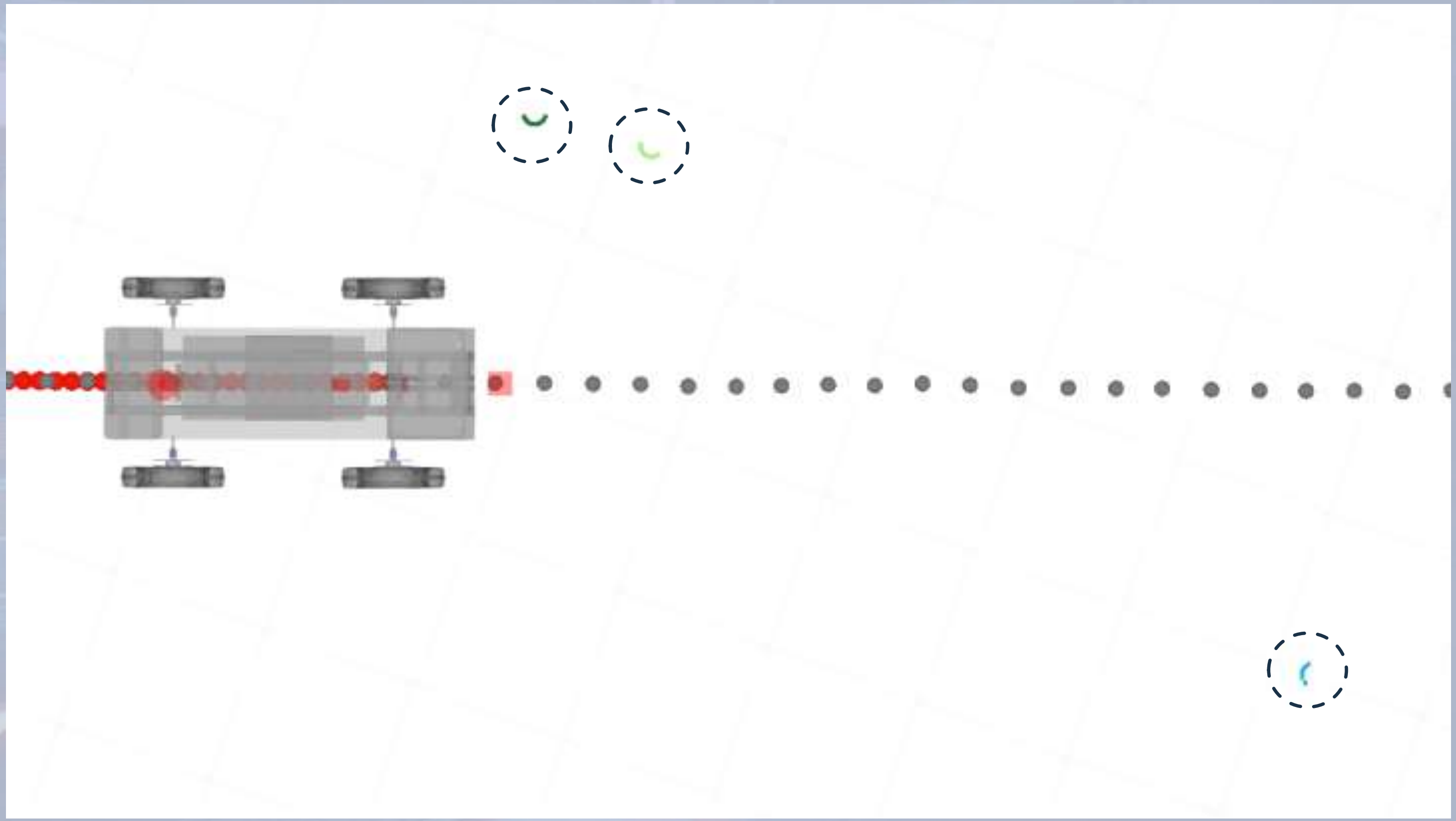




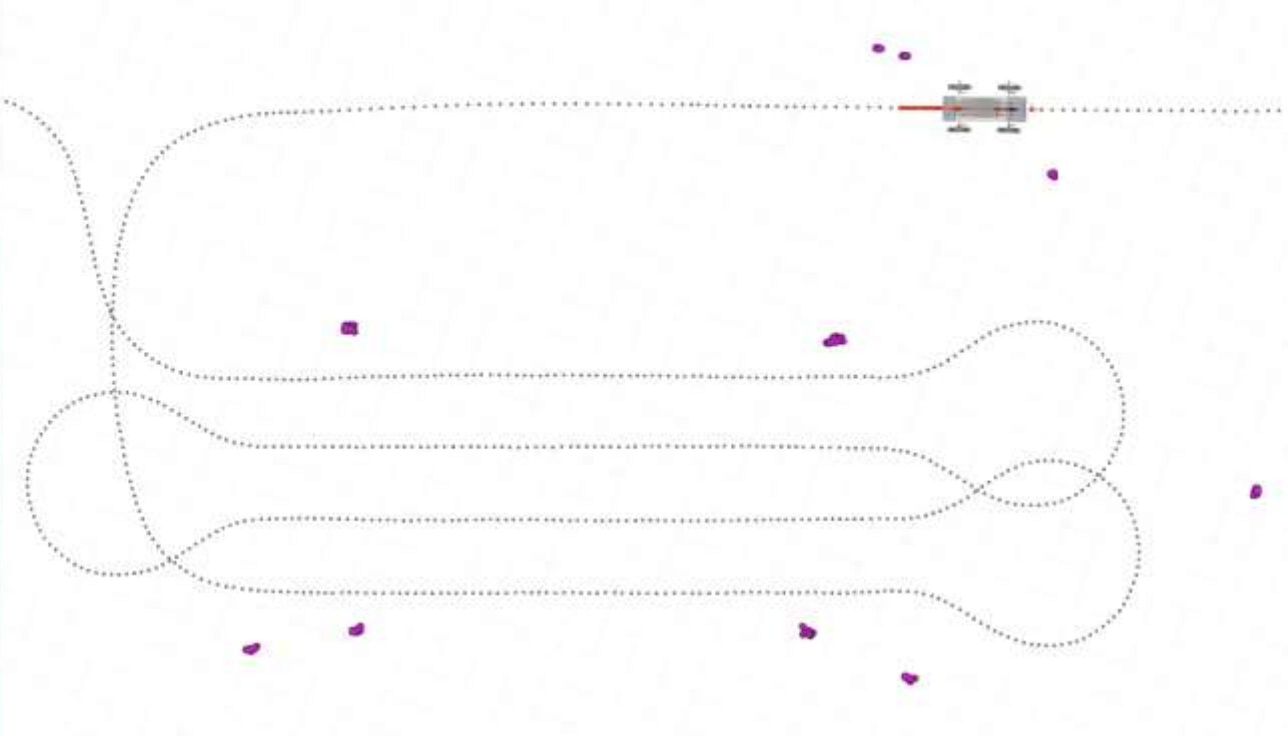
The clustering node first performs an optional outlier removal

```
if (perform_outlier_removal_){
    // Perform Statistical Outlier Removal
    pcl::StatisticalOutlierRemoval<PointType> sor;
    sor.setInputCloud(pcl_cloud);
    sor.setMeanK(sor_mean_K_);
    sor.setStddevMulThresh(1.0);    // Adjust as needed
    sor.filter(*pcl_cloud);
}
```

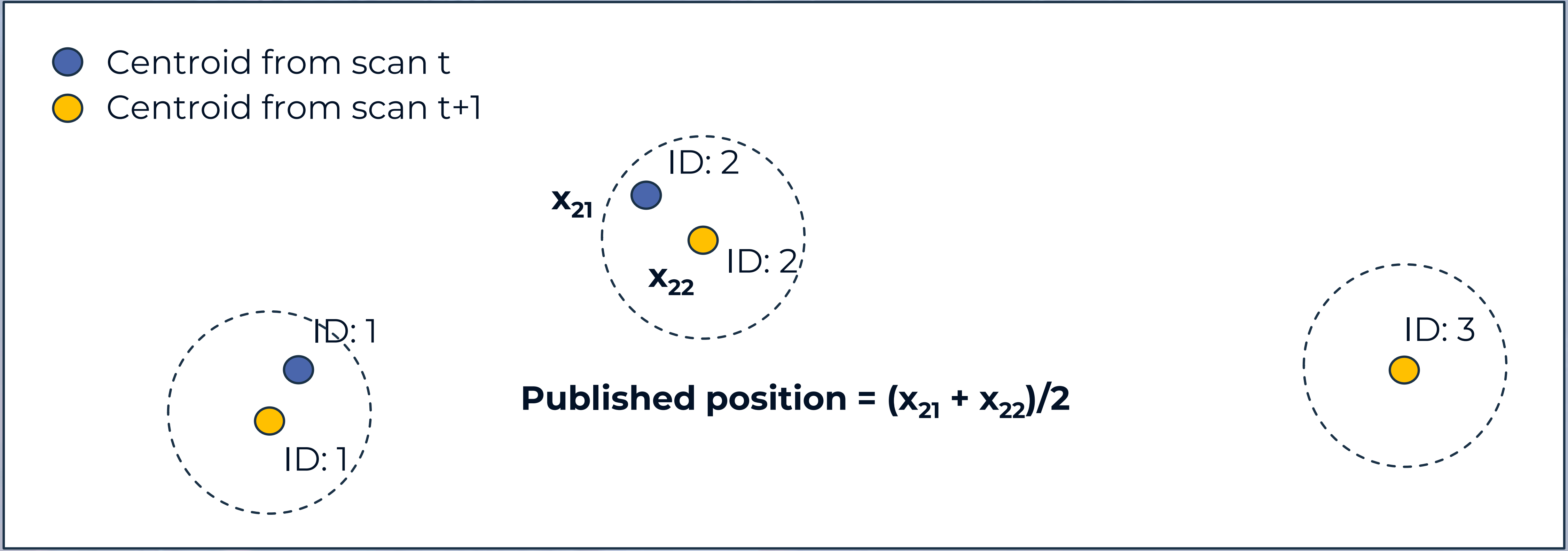
We then perform Euclidean Clustering to find aggregations of points that represent the obstacles



The centroid of each cluster represents the position of an unknown obstacle and is published on a PoinStamped topic for visualization



The position of previously detected obstacles is memorized and accumulated to return an average position on the evaluation topic



All the unique detected obstacle positions are memorized and dumped to a CSV file when a custom service is called

```
ros2 service call /dump_centroids ch2_msg_srv/srv/DumpCentroids "{centroid: true}"
```

```cpp
void dumpCentroidsCallback(
    const std::shared_ptr<ch2_msg_srv::srv::DumpCentroids::Request> request,
    const std::shared_ptr<ch2_msg_srv::srv::DumpCentroids::Response> response)
{
    // Convert geometry_msgs::msg::PointStamped to pcl::PointXYZ
    pcl::PointCloud<pcl::PointXYZ>::Ptr pcl_cloud(new pcl::PointCloud<pcl::PointXYZ>);
    for (const auto& point : unique_centroids_) {
        pcl::PointXYZ pcl_point;
        pcl_point.x = point.point.x;
        pcl_point.y = point.point.y;
        pcl_point.z = point.point.z;
        pcl_cloud->push_back(pcl_point);
    }

    // Create a KdTree object for the search method of the extraction
    auto kd_tree = std::make_shared<pcl::search::KdTree<pcl::PointXYZ>>();

    kd_tree->setInputCloud(pcl_cloud);
```

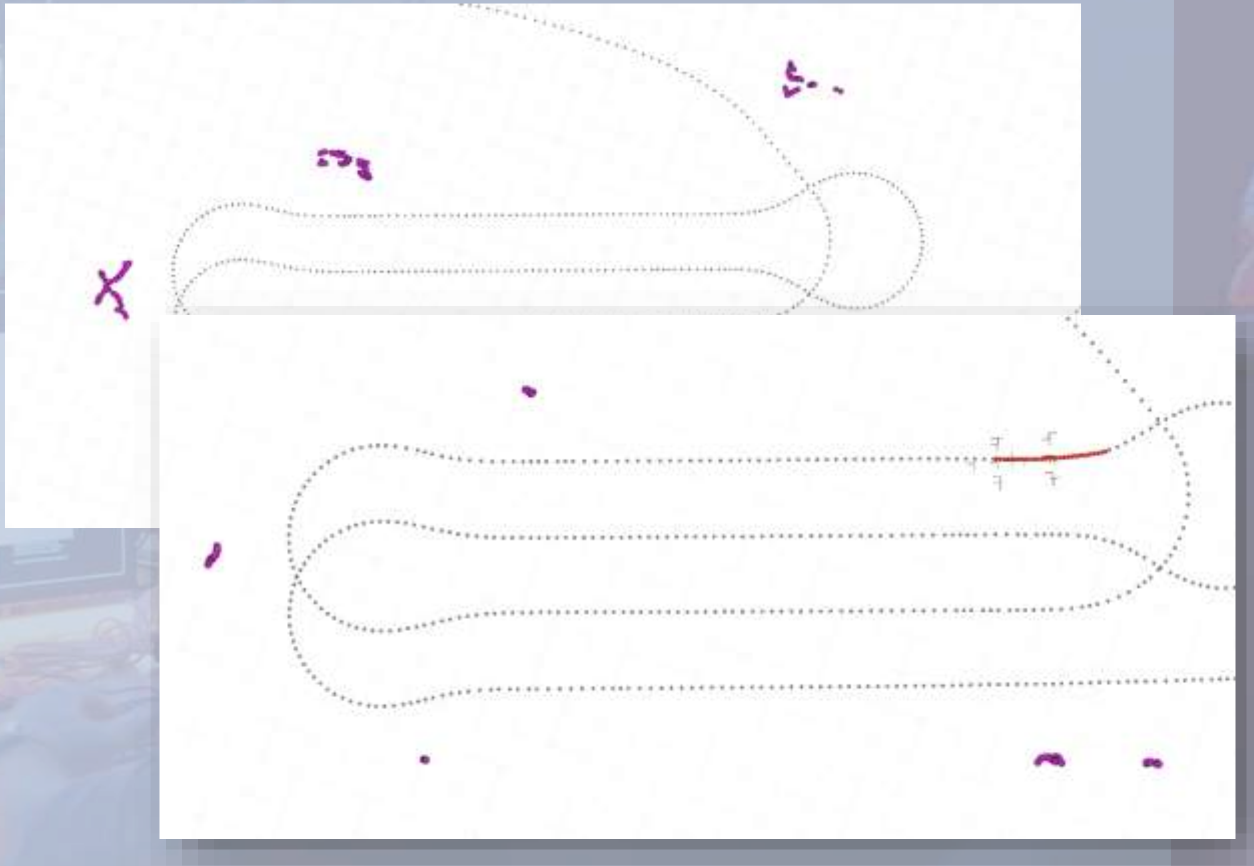
```
1 113.488,118.097,0.740213
2 130.261,128.981,0.687808
3 126.201,122.544,0.791468
4 138.616,122.776,0.625882
5 125.244,130.636,0.712
6 124.483,130.569,0.716286
7 142.631,140.841,0.751177
8 148.963,139.005,0.702564
9 158.297,139.722,0.668684
10 165.648,142.335,0.671026
11 167.567,152.043,0.772381
12 163.434,171.606,0.658395
13 152.436,177.395,0.743492
14 129.922,166.886,0.724242
15 123.184,164.12,0.712703
16 99.46,159.704,0.767879
```

Outline

- System characteristics
- Elaboration pipeline
- Conclusions and future works

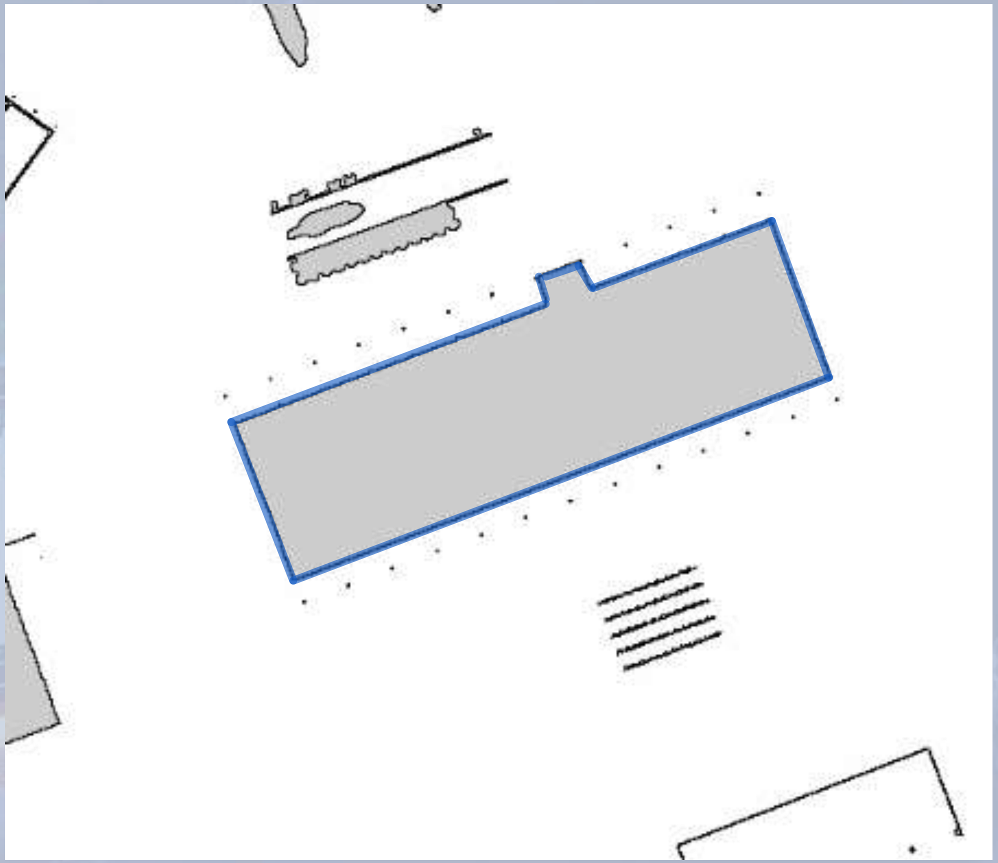
Minimal modifications introduced in the 2nd stage since the system was designed to be adaptable to environmental changes

- Fixed a bug causing wrong obstacle positions
- Added the possibility to filter the LiDAR points based on azimuth angle
- Return an obstacle positions if it is seen at least n times



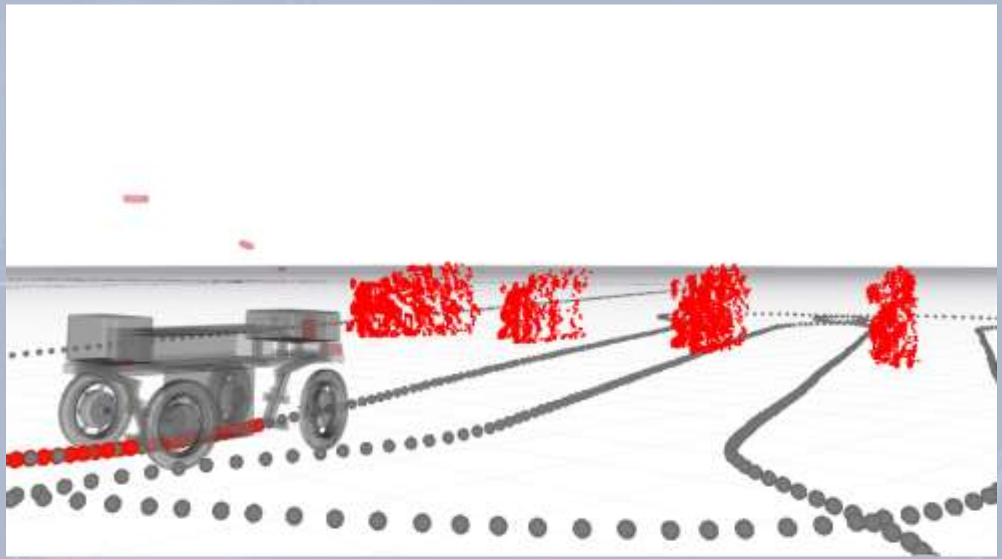
Future works

- Automatic polygons building from the map
- Introduce the possibility to use polyhedrons instead of polygons
- Retrieve obstacle measures for a possible classification



We have presented a versatile and explainable obstacle detection system based on 3D LiDAR data

- No black box algorithms
- No specific features
- Accurate and robust LiDAR data
- Obstacle positions at 6 Hz with LiDAR at 15-20 Hz



Riccardo Bertoglio

AIRLab team

Questions?

Grand Défi Robotique Agricole powered by RobAgri

OPAL-RT

4D VIRTUALIZ

TIRREX

INRAE