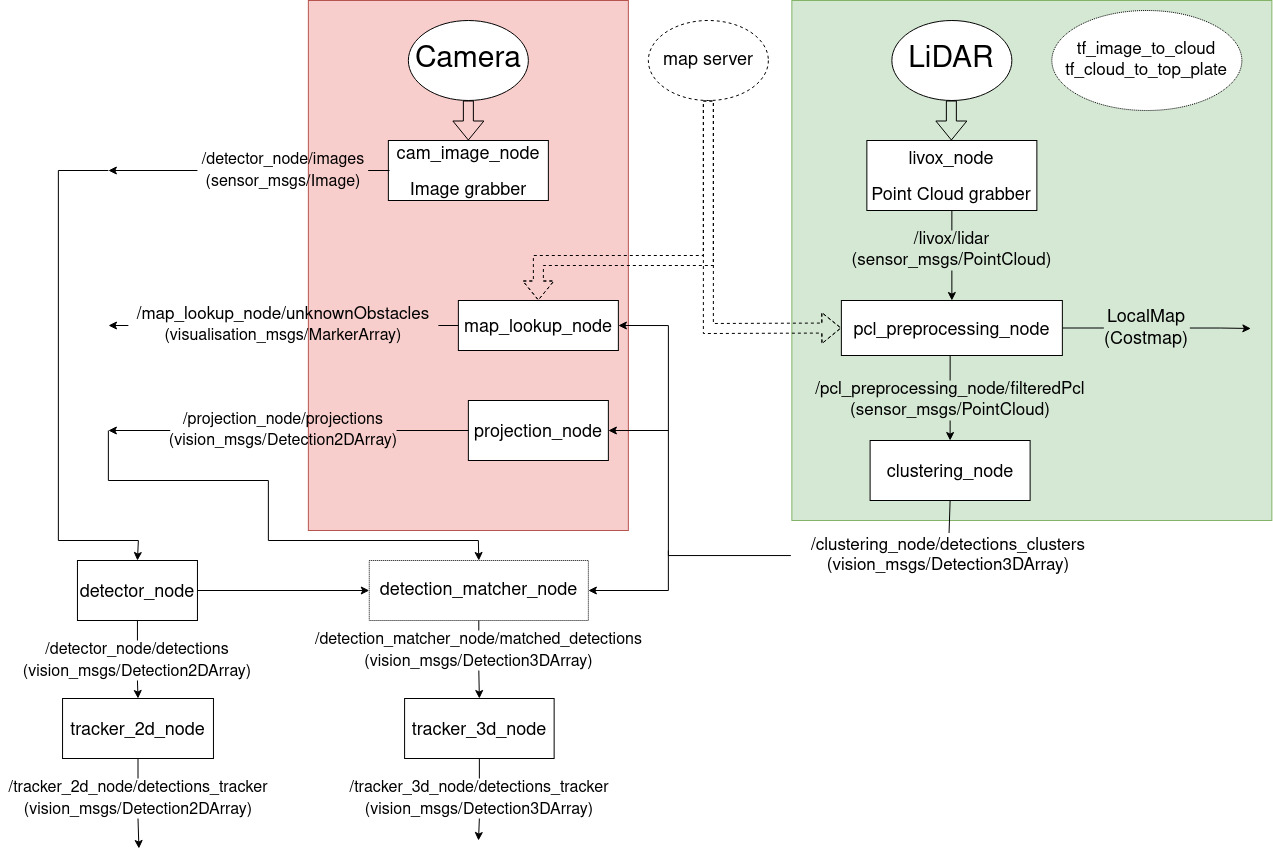
AI4DI documentation vision group

# Overview

The vision part of the robot takes camera and LiDAR data in order to ascertain the presence of obstacles around the robot (or in front of the robot in testing). An independent system of detections in 2D and 3D was developed, and the independent detections are merged to create 3D detections with information about the position, orientation and class of the obstacle. The system is aware of the robot’s position on the map and is able to filter detected obstacles in zones with known obstacles.

The basic system information:

* 14 packages (10 in diagram, 2 YOLOv3, 2 utilities for saved data)
* ROS2-FOXY
* GIT: <https://github.com/LukasKratochvila/ros2-ws-ai4di>
* three devices: 2 Raspberry Pi, 1 Jetson nano



An overview of the pipeline as realized

# Arrangement (HW)

The current HW configuration is as follows:



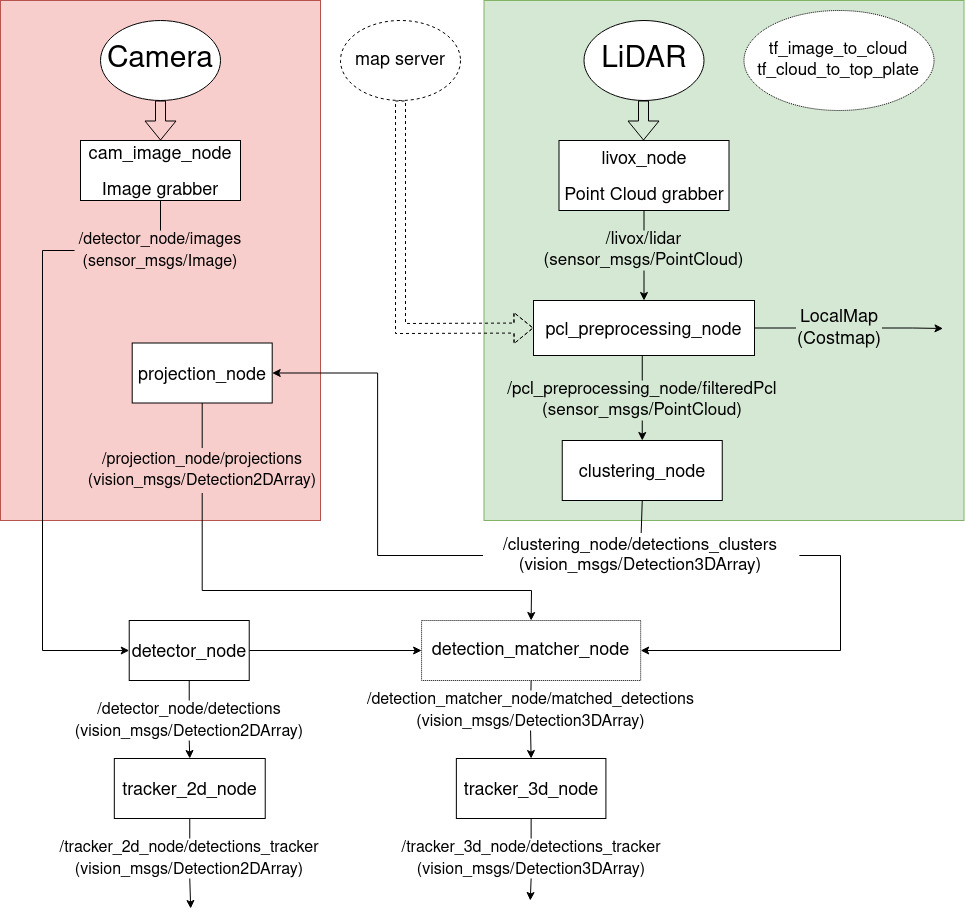
There are three main threads in the diagram (only logical threads, nothing to do with computational threads):

* **camera thread** - The camera + lens feeds raw image data into the RPi - cam Raspberry Pi.
* **LiDAR thread -** feeds raw pointcloud data from the MID-70 into the RPi - LiDAR Raspberry Pi
* **image processing thread** - the Jetson nano performs the more challenging image processing operations, and is not connected directly to the camera to save computational power purely for the neural network inference

The computational devices consist of two Raspberry Pi 4B units, and a Jetson nano unit. The SW is split among these units in order to use the best performance for each unit, and to ensure that time-critical operations are performed without delays:

* **RPi - LiDAR (in green)** - handles the LiDAR configuration and readout via the livox\_ros2\_driver node, and then performs the preprocessing (pcl\_preprocessing\_node) and clustering (clustering\_node) operations
* **RPi - cam** **(in red)** - performs camera reading operations (cam\_image\_node), and then performs some auxiliary operations (such as projecting obstacles detected in the 3D data into the image plane to be matched)
* **Jetson nano -** performs the neural network inference. The performance of this unit is limited, and ideally it should be replaced with the more powerful Jetson models to achieve better performance, and to be able to handle more image sources.

The diagram below demonstrated the split of tasks between the HW units. The nodes in the green field are handled by the RPi - LiDAR, the nodes in the red field by the RPi - cam. The detector\_node is handled by the Jetson nano. The detection\_matcher\_node is currently run externally, but could be run on the more powerful Jetson if it becomes available. The remaining unassigned nodes are also run externally.



## Raspberry Pi

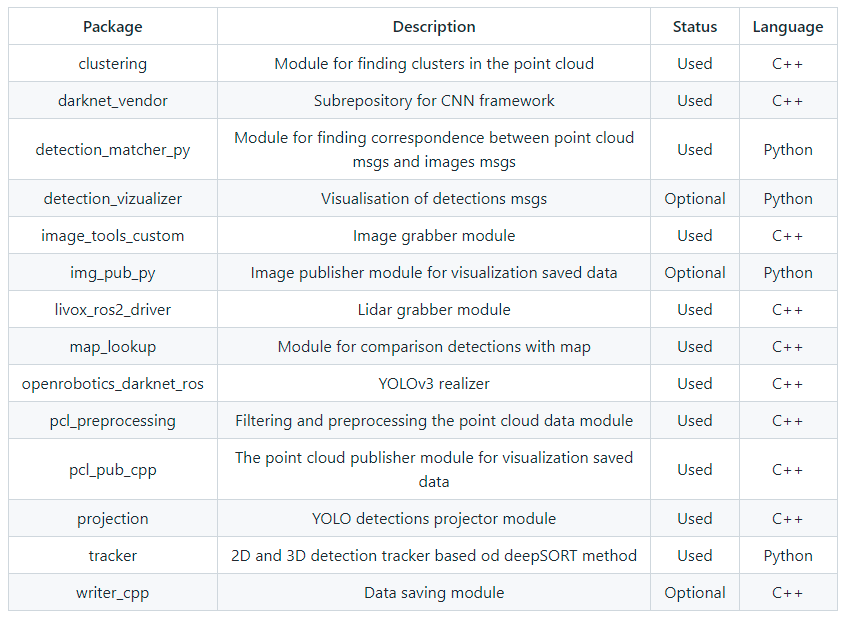
Both the RPi-LiDAR and RPi-cam are Raspberry Pi 4B 4GB variants with added power supply and cooling boards. Both are running the Ubuntu Mate 20.04 OS. Each unit has installed a ROS2-Foxy environment with a specific workspace as mentioned before. For the RPi-LiDAR it is necessary to install LivoxSDK.

## Jetson nano

The Jetson nano also has 4GB RAM, but beside the CPU: Quad-core ARM Cortex-A57, has GPU: NVIDIA Maxwell architecture with 128 NVIDIA CUDA® cores. Jetson running on JetPack SDK4.6.1 from NVIDIA. It is able to run even neural networks at sufficient speed. We use docker image from GIT: <https://github.com/dusty-nv/jetson-containers>. In this container is installed a ROS2-Foxy environment with a specific workspace.

# Packages (SW)

The following packages have been created:



Below the packages are described as far as their interface (inputs and outputs), internal workings, and algorithms used (in alphabetical order, for the pipeline arrangement see the images above):

## clustering

* IN: **preprocessed point cloud** sensor\_msgs/msg/PointCloud2
* OUT: **clusters** vision\_msgs/msg/Detection3DArray

Node takes the preprocessed point cloud and finds clusters using the [Euclidean clustering method](https://pointclouds.org/documentation/classpcl_1_1_euclidean_cluster_extraction.html). The found clusters (if any) are represented as bounding boxes. Either AABB (axis-aligned bounding boxes), or OBB (oriented bounding boxes) can be selected. The node operates as follows:

1. Receive **preprocessed point cloud** msgs =>
2. check content (if empty republish empty msg) =>
3. use pcl::search::KdTree and pcl::EuclideanClusterExtraction =>
4. find clusters (if none found pushlish empty msg) =>
5. find AABB or OBB with inner function (OBB tries find the AABB with minimum footprint by rotating the cluster point cloud, mode and angle increments are set in the launch parameters) =>
6. publish **clusters** msg

| clustering\_node:  ros\_\_parameters:  cluster\_tolerance: 0.2 #in meters  min\_cluster\_size: 30  max\_cluster\_size: 5000  input\_topic: "/pcl\_preprocessing\_node/filteredPcl"  output\_topic: "~/detections\_clusters"  OBB: true  step: 30  debug: false | **Clustering node parameters**  How far apart points are still one cluster (in meters)  Minimum cluster size (in points)  Maximum cluster size (in points)  Input topic identifier  Output topic identifier  Bounding box mode switch (true=OBB, false=AABB)  Angle step for OBB search  Debug switch (true for verbose execution) |
| --- | --- |

## darknet\_vendor (non-original)

* Taken over from: <https://github.com/ros2/darknet_vendor>
* CMake wrapper around darknet, an open source neural network framework.

## detection\_matcher\_py

* IN: **clusters** vision\_msgs/msg/Detection3DArray and **2Ddetections, 2Dprojections** both vision\_msgs/msg/Detection2DArray
* OUT: **matched detections** vision\_msgs/msg/Detection3DArray

Node takes 2D detection (from YOLO) and 2D projection (from Clusters) and tries to match to 3D detections (clusters) for label transfer (from 2D YOLO detection to 3D cluster detection). The result then should be groups of corresponding 2D and 3D detections, published as 3D detections with the associated class.

1. Receive 3 approximate synchronize msgs **clusters, 2Ddetections, 2Dprojections** =>
2. control sizes of projection and 3D detection (should be same) =>
3. one by one match 3D detection with the highest IOU ([Intersection over Union](https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/#:~:text=predicted%20bounding%20boxes.-,What%20is%20Intersection%20over%20Union%3F,the%20popular%20PASCAL%20VOC%20challenge.)) candidate and associate 3D detection with its label =>
4. publish **matched detections** msg

| detection\_matcher\_node:  ros\_\_parameters:  Project2D\_topic: "/projection\_node/projections"  Detect2D\_topic: "/detector\_node/detections"  Detect3D\_topic: "/clustering\_node/detections\_clusters"  Output3D\_topic: "~/matched\_detections"  queue\_size: 100  time\_toll: 0.1  tresh: 0.5  debug: false | **detection\_matcher\_node parameters**  Project2D topic identifier  Detect2D topic identifier  Detect3D topic identifier  Output topic identifier  Queue size for message synchronization  Time toll parameter for message synchronization  Threshold parameter for IoU metric  Debug switch (true for verbose execution) |
| --- | --- |

## detection\_vizualizer

Part taken over from: <https://github.com/ros2/detection_visualizer>

Package for visualization 2D detections (from YOLO), 3D detections and image decompression.

### DetectionVisualizerNode

* Node for visualization 2D detection.
* IN: **image** sensor\_msgs/msg/Image and **2Ddetections** vision\_msgs/msg/Detection2DArray
* OUT: **image with detections** sensor\_msgs/msg/Image

### Det3dVisualizerNode

* Node for visualization 3D detection. Detection 3D to MarkerArray converter.
* IN: **3Ddetection** vision\_msgs/msg/Detection3DArray
* OUT: **3Ddetection** visualization\_msgs/msg/MarkerArray

### Decompressor

* Node for decompression of compressed images. Compressed Image to Image converter.
* IN: **compressed image** sensor\_msgs/msg/CompressedImage
* OUT: **decompressed image** sensor\_msgs/msg/Image

Only for debugging and presentation.

## image\_tools\_custom (non-original)

Taken over from: <https://github.com/ros2/demos/tree/foxy/image_tools>. Edited node for image publishing. (Added properties, see the param file example)

* OUT: **raw image** sensor\_msgs/msg/Image and **compressed image** sensor\_msgs/msg/CompressedImage

| cam\_image\_node:  ros\_\_parameters:  reliability: 'reliable'  history: 'keep\_last'    depth: 10  frequency: 30.0  burger\_mode: false  show\_camera: false  width: 640  height: 480  frame\_id: image  output\_topic: '/detector\_node/images'  device\_id: 0  api\_id: 0  cap\_mode: 0  debug: false | **image\_tools\_custom node parameters**  Reliability QoS setting. Either 'reliable' (default) or 'best\_effort'  History QoS setting. Either 'keep\_last' (default) or 'keep\_all'. If 'keep\_last', then up to N samples are stored where N is the depth  Depth of the publisher queue. Only honored if history QoS is 'keep\_last'. Default value is 10  Publish frequency in Hz. Default value is 30  From the original node - unused  From the original node - unused  Width component of the camera stream resolution. Default is 320  Height component of the camera stream resolution. Default is 240  ID of the sensor frame. Default value is 'camera\_frame'  Output ROS topic. Default value is 'image'  ID of the camera source. Default value is '0'  ID of the opencv api. Default value is '0'  Capture mode (only for libv4l). Default value is '0'  Debug switch (true for verbose execution) |
| --- | --- |

## img\_pub\_py (utility)

Node for publishing saved images from folder at a given frequency.

* OUT: **images** sensor\_msgs/msg/Image

## livox\_ros2\_driver (non-original)

Taken over from: <https://github.com/Livox-SDK/livox_ros2_driver>. Livox device driver under Ros2, support Lidar Mid-40, Mid-70, Tele-15, Horizon, Avia. Node for reading and publishing raw point cloud from Livox sensor. Enables configure measurements parameters.

* OUT: **raw point cloud** sensor\_msgs/msg/PointCloud2

| livox\_node:  ros\_\_parameters:  xfer\_format: 0  multi\_topic: 0  data\_src: 0  publish\_freq: 10.0  output\_data\_type: 0  frame\_id: "cloud"  user\_config\_path: "path\_default"  cmdline\_input\_bd\_code: "000000000000001"  lvx\_file\_path: "/home/livox/livox\_test.lvx" | **livox\_ros2\_driver node parameters**  0-Pointcloud2(PointXYZRTL), 1-customized pointcloud format  0-All LiDARs share the same topic, 1-One LiDAR one topic  0-lidar,1-hub  frequency of publishing,1.0,2.0,5.0,10.0,etc freq from (0.1, 100)  Output to ros: 0, Output to bagfile: = 1  Output frame id  User config file path  Default BD code  LVX file path |
| --- | --- |

## 

## map\_lookup

* IN: **candidate detections** visualization\_msgs/msg/MarkerArray
* OUT: **new detections** nav\_msgs/msg/OccupancyGrid

The node takes detection candidates from the detection pipeline, and looks in the map whether the detection falls into an area of known obstacles, or whether this is actually an unknown obstacle. The algorithm for each candidate is as follows:

1. transform the candidate detection into map coordinates =>
2. generate a grid (5x5 by default) of points describing the footprint of the bounding box =>
3. check the occupancy grid in all the points (25 by default) (the box might not be entirely in the occupied/clear area), and increase a counter for each point in the occupied area =>
4. compare the accumulator to a threshold (threshold by default 0.2) =>
5. publish **new detections**, which satisfy the threshold condition

The node is not currently used in the pipeline, instead pcl\_preprocessing map crop feature is used. If map\_lookup is to be used, changes are needed.

| map\_lookup\_node:  ros\_\_parameters:  input\_topic: "/clustering\_node/detections"  output\_boxes\_topic: "~/unknownObstacles"  output\_map\_topic: "~/ObstacleMap"  synchronised: false  map\_server: "/map\_server/map"  debug: false | **map\_lookup\_node parameters**  Input topic identifier  Output box topic identifier  Output map topic identifier  Synchronisation switch (default false)  Map topic identifier  Debug switch (true for verbose execution) |
| --- | --- |

## openrobotics\_darknet\_ros (non-original)

This is a ROS 2 wrapper around [darknet](https://pjreddie.com/darknet), an open source neural network framework.

* Taken over from: <https://github.com/ros2/openrobotics_darknet_ros>
* IN: **image** sensor\_msgs/msg/Image
* OUT: **2Ddetections** vision\_msgs/msg/Detection2DArray

The node runs the selected model (in this case YOLOv3) on any received image message, and returns the detections in form of a 2D detection message, including the position, size, class and confidence of the detection.

The CNN model is loaded from a file and inference is performed according to the model. Training is performed externally either as a retraining of an already trained model on a given dataset, or from scratch on a given dataset.

| detector\_node:  ros\_\_parameters:  network:  config: "./model/yolov3-tiny.cfg"  weights: "./model/yolov3-tiny.weights"  class\_names: "./model/coco.names"  detection: threshold: 0.25  nms\_threshold: 0.45 | **map\_lookup\_node parameters**  Config file path  Model weights file path  Class name legend file path  Detection confidence threshold (above is accepted, below rejected)  Non Maximum Supression threshold (to weed out overdetections) |
| --- | --- |

## pcl\_preprocessing

* IN: **raw input cloud** sensor\_msgs/msg/PointCloud2
* OUT: **filtered cloud** sensor\_msgs/msg/PointCloud2
* OUT (optional): **local map** nav\_msgs/msg/OccupancyGrid

Node reads raw pointcloud data provided by the livox\_ros2\_driver node, and performs preprocessing necessary for further operations such as clustering. The work flow is as described below:



Upon receiving the pointcloud message, following operations are performed (particular configuration of the node depends on the parameters at launch:

* **cloud crop** - crops the raw cloud by a fixed (non-smart) crop box (dimensions specified in the parameters at launch). This removes unwanted points - too far ahead, to the side, above and below. The purpose is to reduce the cloud size for performance
* **detect floor** - Detects the largest near horizontal surface that is described by the cloud, then removes the points representing that surface
* **downsample** - downsampled the cloud using the [voxelgrid](https://pointclouds.org/documentation/classpcl_1_1_voxel_grid.html) method. This removes a significant amount of points reducing computational power requirements, it also reduces the point cloud density in areas close to the sensor, and retains density in less dense areas
* **map crop** - removes points in places of already known obstacles (from the map). Obstacle detections are then not generated in areas already marked as obstacles
* **frame aggregator** - optionally (not currently used) the frame aggregator concatenates pointcloud frames transformed by the corresponding position and orientation of the sensor in order to generate a dense point cloud - the aggregation has problems with faster moving objects if the aggregation periods are longer

A filtered cloud is then output as a basis for further processing. Optionally a local map occupancy grid is also published.

| pcl\_preprocessing\_node:  ros\_\_parameters:  crop: true  detectFloor: false  downSample: true  mapCrop: true  enableAgg: false  createLocalMap: false  input\_topic: "/livox/lidar"  output\_cloud\_topic: "~/filteredPcl"  output\_map\_topic: "~/localMap"  filter\_x\_min: 0.1  filter\_y\_min: -10.0  filter\_z\_min: -0.2  filter\_x\_max: 25.0  filter\_y\_max: 10.0  filter\_z\_max: 2.0  voxel\_grid\_x\_res: 0.025  voxel\_grid\_y\_res: 0.025  voxel\_grid\_z\_res: 0.025  mapTreshold: 200  map\_frame: "map"  agg\_window: 2e8  costmap\_node\_name: "pcl\_preprocessing\_costmap"  local\_costmap\_namespace: "pcl\_preprocessing\_costmap"  debug: false | **pcl\_preprocessing\_node parameters**  Cloud crop switch  Floor detect switch  Downsample switch  Map crop switch  Frame aggregator switch  Local map switch  Input topic identifier  Output cloud topic identifier  Output map topic identifier  Crop parameters (XYZ in Livox MID-70 frame, all in meters)  Voxel Grid resolution in X and Y, keep them equal  Voxelgrid resolution in Z  Map threshold for occupied / clear  Map frame identifier  Frame aggregator time span in ns  Costmap identifier  Local map identifier  Debug switch (true for verbose execution) |
| --- | --- |

An [example](#_nqsb85ernnx2) of the pcl\_preprocessing.

## pcl\_pub\_cpp (utility)

Node for publishing saved point clouds (from .pcd files) at a given frequency.

* OUT: **point cloud** sensor\_msgs/msg/PointCloud2

## projection

* IN: **clusters** visualization\_msgs/msg/MarkerArray
* OUT: **2Dprojections** vision\_msgs/msg/Detection2DArray
* OUT (optional): **img\_publisher** sensor\_msgs/msg/Image
* OUT (optional): **pcl\_publisher** visualization\_msgs/msg/MarkerArray

Node takes clusters and projects them to camera space (image coordinates) using a known transformation. The projection allows for mapping of 3D detections onto the 2D detections. This is an inverse perspective projection problem which requires camera calibrations (or more precisely intrinsic camera parameters), the camera was calibrated externally and intrinsic parameter matrix is used as parameter. Any camera change, lens change (or zoom change) requires a camera calibration. The workflow is as follows:

1. Receive **clusters** msg =>
2. for each cluster try to transform marker to image frame =>
3. project front side and center of 3D object to 2D => (here problems may be encountered using the OBB representation, the selected face may not be the best representation (requires some changes to code))
4. publish **2Dprojections** msg

| projection\_node:  ros\_\_parameters:  input\_topic: "/clustering\_node/detections"  output\_topic: "~/projections"  output\_pcl\_topic: "~/pcl"  output\_img\_topic: "~/img"  output\_frame: "image"  debug: false | **map\_lookup\_node parameters**  Input topic identifier  Output projections topic identifier  Output cloud topic identifier  Output image topic identifier  Output topic frame identifier  Debug switch (true for verbose execution) |
| --- | --- |

## tracker

* IN: **detections** vision\_msgs/msg/Detection3DArray or vision\_msgs/msg/Detection2DArray
* OUT: **tracked** **detections** vision\_msgs/msg/Detection3DArray or vision\_msgs/msg/Detection2DArray
* Part taken over from: <https://github.com/nwojke/deep_sort>

Node realizes tracking of already detected potential obstacles. It uses the [DeepSORT](https://learnopencv.com/understanding-multiple-object-tracking-using-deepsort/#:~:text=DeepSORT%20is%20a%20computer%20vision,Simple%20Online%20Realtime%20Tracking)%20algorithm.) method to achieve tracking using Kalman filtration, and deep association between instances using neural networks. Kalman filtration predicts position of tracked object and deep association uses a deep appearance descriptor (only works with images in this implementation). Would work better with higher detection framerate.

Node works in three modes: 3D or 2D tracker and the 2D tracker with or without deep features. It takes detections, converts them to deep\_sort format and tracks them with a kalman filter. The workflow is as follows:

1. Receive **detections** msg =>
2. if deep features are enabled find it =>
3. convert detections into deep\_sort format=>
4. predict new positions (use deep\_sort) =>
5. update positions =>
6. convert detections to vision\_msgs format =>
7. publish **tracked** **detections** msg

In 2D mode

| tracker\_2d\_node:  ros\_\_parameters:  input\_det\_topic: '/detector\_node/detections'  output\_det\_topic: '~/detections\_tracker'  dim: 2  model\_path: "model/tracker-mars-small128.pb"  metric: "euclidean"  matching\_threshold: 0.2  budget: 10  max\_iou\_distance: 0.9  max\_age: 30  n\_init: 3  debug: false | **tracker\_2d\_node parameters**  Input topic identifier  Output detection topic identifier  Dimensions to track (2D in this case)  Path to the DeepSORT model  Distance metric  Threshold of matching  Budget - how many latest samples to remember  How many frames can a sample be old  Debug switch for verbose execution |
| --- | --- |

In 3D mode

| tracker\_3d\_node:  ros\_\_parameters:  input\_det\_topic: '/clustering\_node/detections\_clusters'  output\_det\_topic: '~/detections\_tracker'  dim: 3  model\_path: None  metric: "euclidean"  matching\_threshold: 0.2  budget: 10  max\_iou\_distance: 0.9  max\_age: 3  n\_init: 3  debug: false | **tracker\_3d\_node parameters**  Input topic identifier  Output detection topic identifier  Dimensions to track (3D in this case)  Path to the DeepSORT model  Distance metric  Threshold of matching  Budget - how many latest samples to remember  How many frames can a sample be old  Debug switch for verbose execution |
| --- | --- |

## 

## writer\_cpp (utility)

Node for saving images and/or point clouds into .pcd files. Merely a utility.

* IN: **image** sensor\_msgs/msg/Image
* IN: **point cloud** sensor\_msgs/msg/PointCloud2

| writer\_node:  ros\_\_parameters:  save\_dir: "./"  img\_sub\_dir: "image/compressed"  pcl\_sub\_dir: "pcl/"  input\_img\_topic: "/image"  input\_pcl\_topic: "/livox/lidar"  mode: "both"  synchronized: false  time\_toll: 0.01  start\_img\_counter: 0  start\_pcl\_counter: 0  img\_file\_ext: "jpg"  pcl\_file\_ext: "pcd"  debug: true | **writer\_node parameters**  Save directory path  Image subdirectory name  Clout subdirectory name  Image input topic name  Cloud input topic name  Save mode: “img”, “pcl” or “both”  Synchronised switch  Time toll for synchronization  Filename counter start for image  Filename counter start for cloud  Image file extension  Cloud file extension  Debug switch for verbose execution |
| --- | --- |

# Startup

The launch of the perception pipeline is realized via ROS2 launch files ([particular launch file examples for this project](https://github.com/LukasKratochvila/ros2-ws-ai4di/tree/main/launch)). For each hardware unit a custom launch file exists launching the particular nodes that are to be run on said unit. A common params file exists for all the units ([an example for this project](https://github.com/LukasKratochvila/ros2-ws-ai4di/tree/main/params)), excerpts from which are shown above. This params file dictates the configuration of all the nodes including their parameters.

Currently the HW is set up for automatic launch of all nodes in the pipeline.

# Datasets

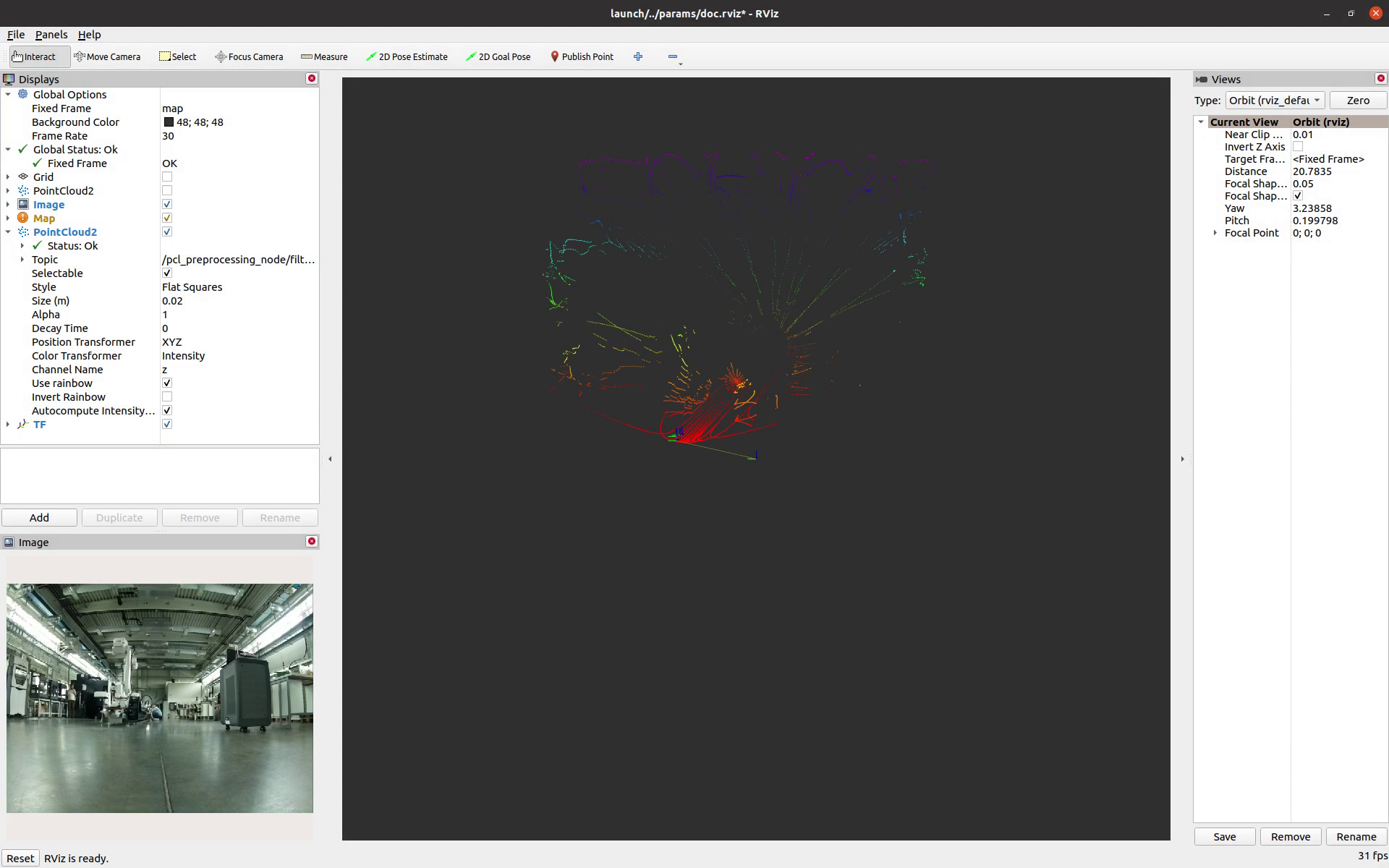
During testing we collected 3 different datasets. The first one was in an industrial environment with different situations such as planning way, obstacle detours and emergency stopping. This one is the closest to the real situation. Unfortunately it contains several errors. Second one was developed in the laboratory at the University. In the lab there are many different objects. The last dataset was created in a small room testbed on CEITEC. It is the most complete dataset. Only small error is bad calibration of map origin.

# Visualization

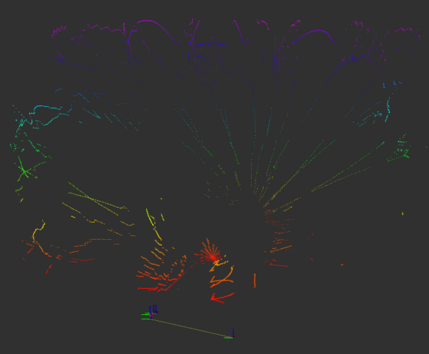
## Lidar preprocessing pipeline

# 

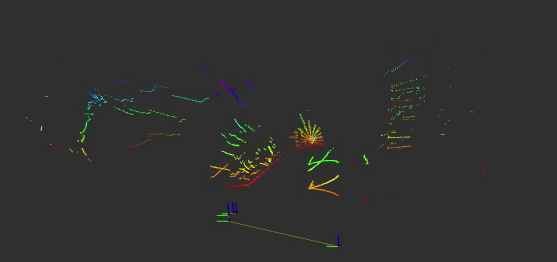
Raw point cloud



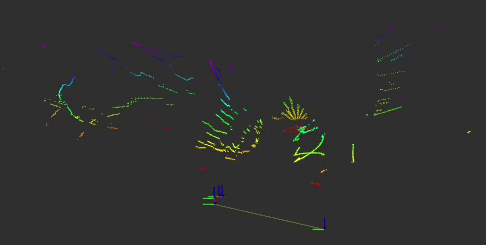
Associated Image



Floor detected and removed

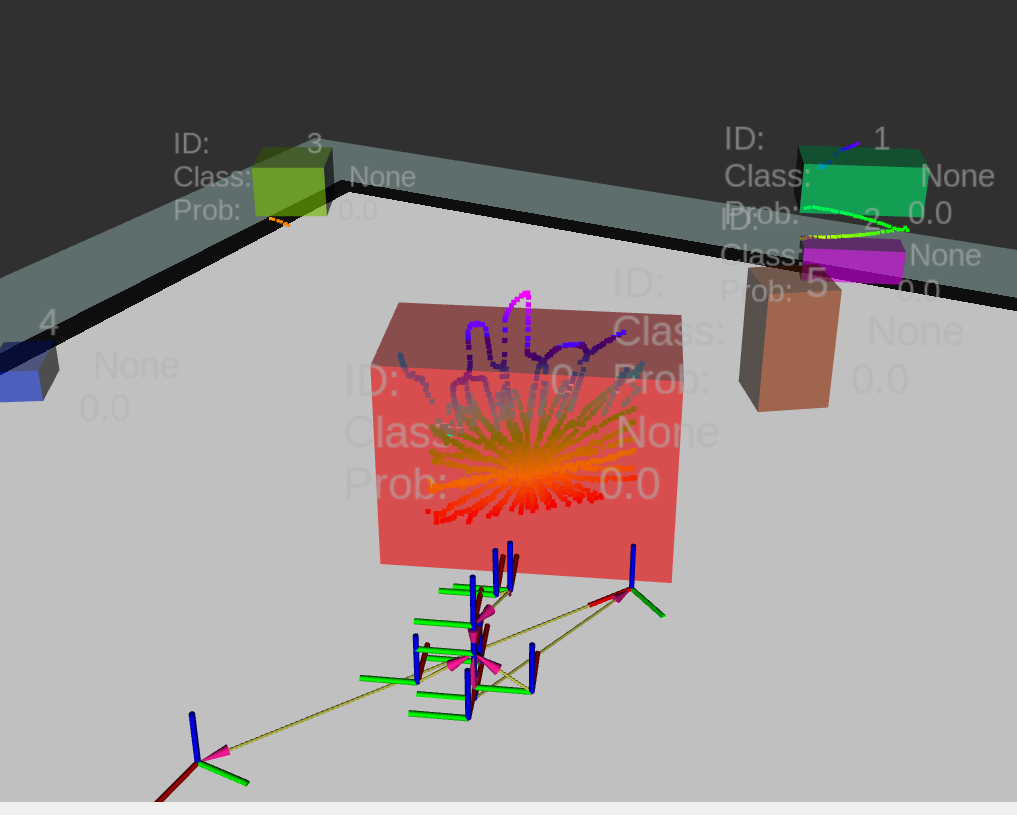


Cropped by a fixed box (extra points too far removed)



Cropped by a fixed box and downsampled

## Clustering

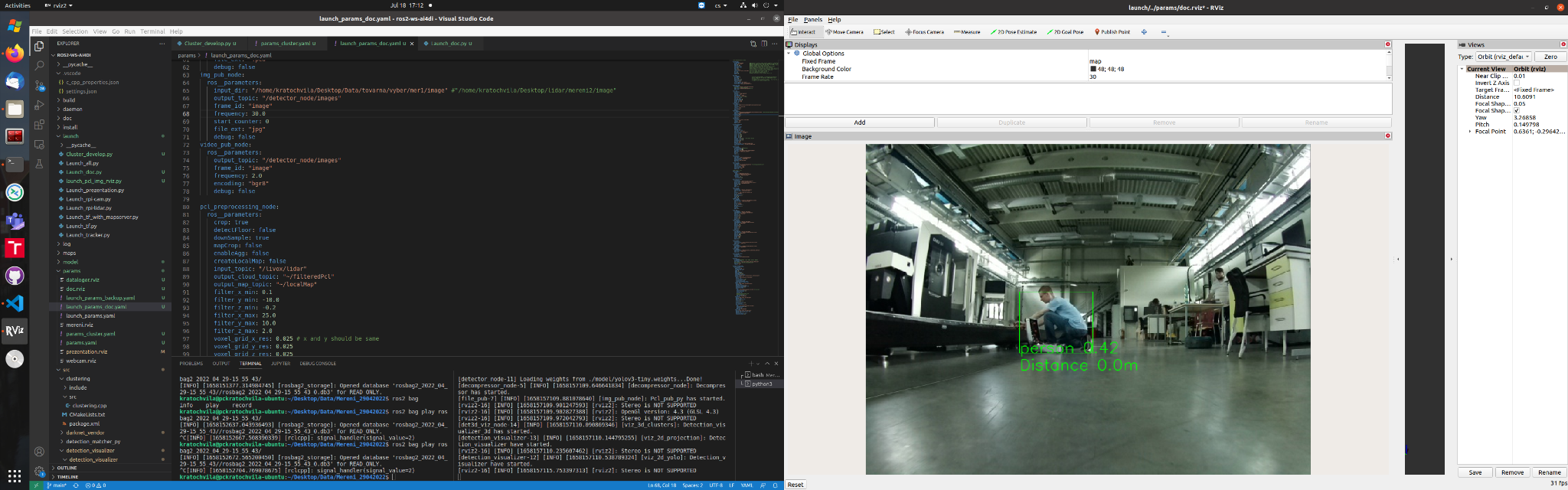


## Projection



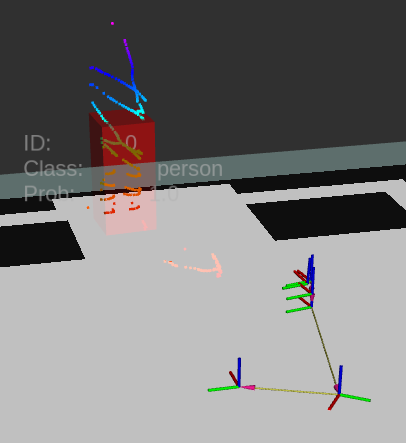
3D detections projected into the image (note the missing class information and confidence, but the present distance from the robot (assumed from the LiDAR sensor origin)

## YOLO detection



Visualized YOLO detection

## Matcher



Visualization of the 3D detection from matcher node with associated label and probabilities (label: person, confidence: 1.0)