Sensors and object clustering

Nacho Sañudo
University of Modena and Reggio Emilia
Ignacio.sanudoolmedo@unimore.it

Brief introduction

- Ignacio (Nacho) Sañudo Olmedo
  - Postdoctoral researcher @ HiPeRT Lab
  - Computer science engineer degree 2014 @University of Cantabria
  - PhD thesis 2018 @ University of Modena and Reggio Emilia
- Research and teaching
  - Real-time (predictability, automotive, GPU architectures)
  - Research projects (Automotive)
    - Bosch
    - Ferrari
    - Many others...
  - Prof./collaborator in Platforms and Algorithms for Autonomous Driving
  - Prof. in Advanced informatics in Economy
    - Excel

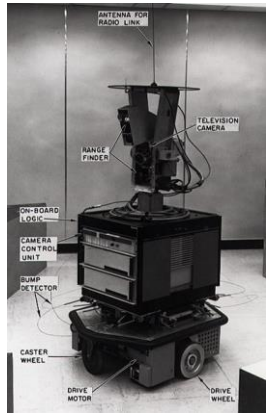
Self driving cars

- A self-driving car, also known as an autonomous vehicle (AV), is a vehicle that is capable of **sensing** its environment and **moving** safely with little or no human input
- SDC are robots that acts considering the environment
  - It's hard to model/interpret the world and be able to act intelligently
  - Easy to detect the color of a traffic light, not that easy if the sun is directly behind the traffic light
    - Moravec's paradox: "it is comparatively easy to make computers exhibit adult level performance on intelligence tests or playing checkers, and difficult or impossible to give them the skills of a one-year-old when it comes to perception and mobility"

https://youtu.be/s-8cYj_eh8E?t=1898

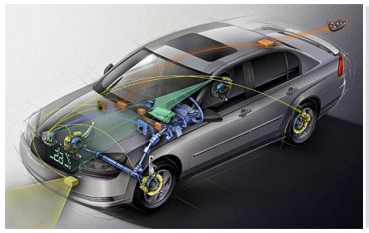
Self driving car history

- These research projects changed the course of the modern mobility
  - Shakey (1966-1972)
    - The first general-purpose mobile robot
      - A* search algorithm, the Hough transform, and the visibility graph method
  - VaMoRs Mercedes project (1983-2002)
    - One of the first autonomous car
      - More than 2000km semi-autonomous
      - Use of advanced computer vision techniques
  - DARPA Challenge (2005-2007)
    - The rebirth of the autonomous hype
      - The engine that pushed the industry to acknowledge the readiness of the technology



Source: IEEE Talk James Gowers May 12 2020

Automotive - Brain

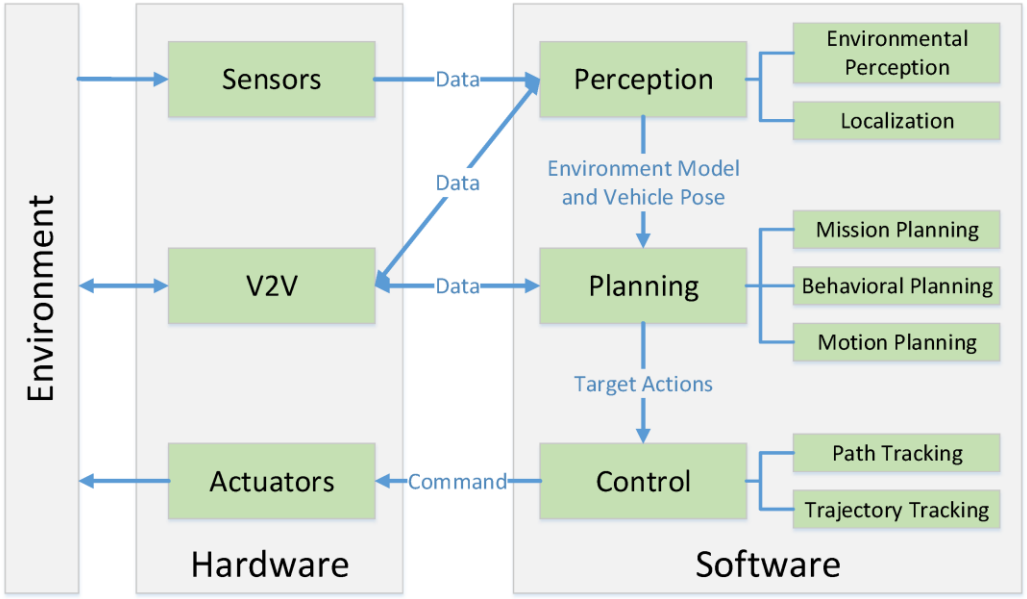
		Lines of Code (LoC)
2015		100M
2013		25M
1981		400k

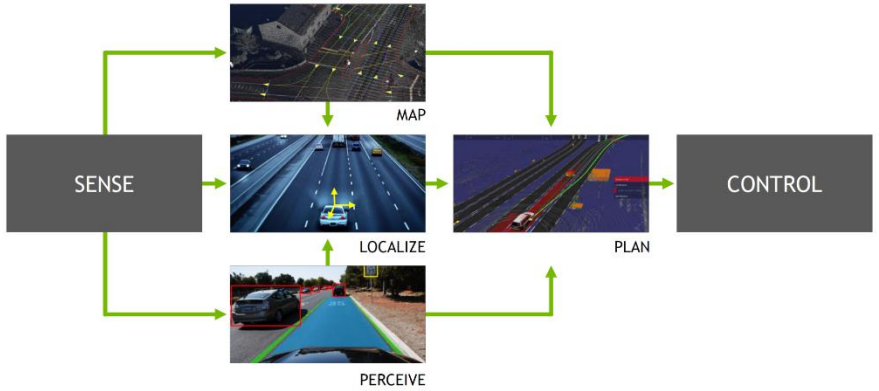
- A million lines of code, if printed, would be about 18,000 pages of text

Source: http://www.visualcapitalist.com/millions-lines-of-code/

Software stack

› The “brain” of the car is composed of three main components
  – Perception, Planning and control





Source: Perception, Planning, Control, and Coordination for Autonomous Vehicles. Scott Drew Pendleton et al.

Perception

- **Environmental perception:** ability to extract meaningful information from the environment by using sensors (Computer vision/AI)
  - Predominantly, the camera is the sensor used to perform object detection
    - Dynamic objects: Object classification, object tracking, motion prediction…
    - Static objects: traffic lights, signs, lanes….
- **Localization**: Is the capacity of the car to determine it's position in the world with low error
  - Different sensors can be used: GPS, IMU, radar, LiDAR, camera….

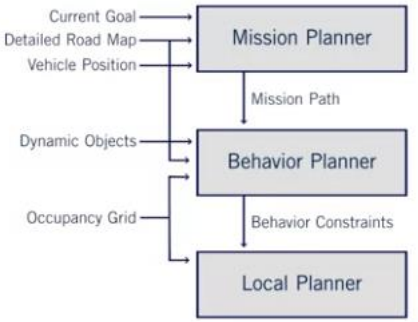
Source: Udacity

Object detection

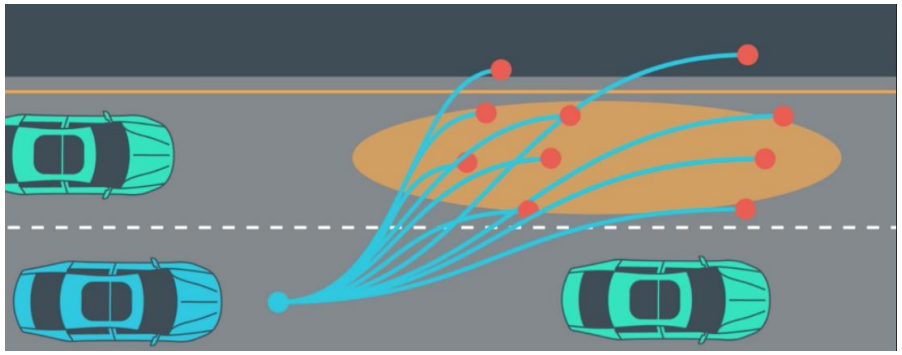
- “tkDNN is a Deep Neural Network library built with cuDNN and tensorRT primitives, specifically thought to work on NVIDIA Jetson Boards. It has been tested on TK1 (branch cudnn2), TX1, TX2, AGX Xavier, Nano and several discrete GPUs. The main goal of this project is to exploit NVIDIA boards as much as possible to obtain the best inference performance”
- https://www.youtube.com/watch?v=2WbtJBN6gzs

https://github.com/ceccocats/tkDNN

Path planning



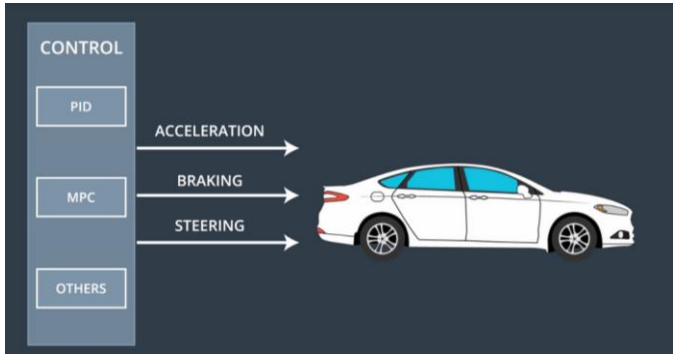
- **Path planning**: is computational problem to find a sequence of valid configurations or paths that moves the ego-vehicle from the source to destination
  - **Mission planning**: is the process of making the highest-level decisions with regard to the proposed destination, for instance, which road should we take if we would like to go from Modena to Bologna
  - **Behavioral planning**: responsible for decision making to ensure the vehicle follows any stipulated road rules and interacts with other agents in a safe manner
  - **Motion planning**: process of deciding on a sequence of actions to reach a specified goal, typically while avoiding collisions with obstacles (it defines the specific plan to drive)



Source: Perception, Planning, Control, and Coordination for Autonomous Vehicles. Scott Drew Pendleton et al.

Control

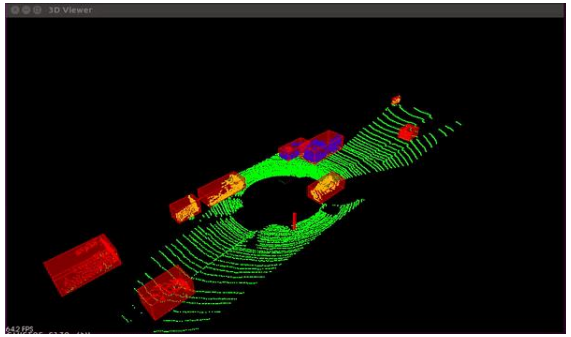
- **Control:** computes and sends commands to the actuators of the steering wheel, throttle and brakes
  - Path tracking: plan to move from a start pose to a goal pose (**pure pursuit**)
  - Trajectory tracking: plan to move from a start pose to a goal pose includes the velocity (and time) information of the motion (**MPC**)
- Controllers typically separates the problem in longitudinal and lateral control
  - Lateral control: steering angle
  - Longitudinal control: throttle and braking



https://www.youtube.com/watch?v=vgEyvazwrU8

Source: Perception, Planning, Control, and Coordination for Autonomous Vehicles. Scott Drew Pendleton et al.

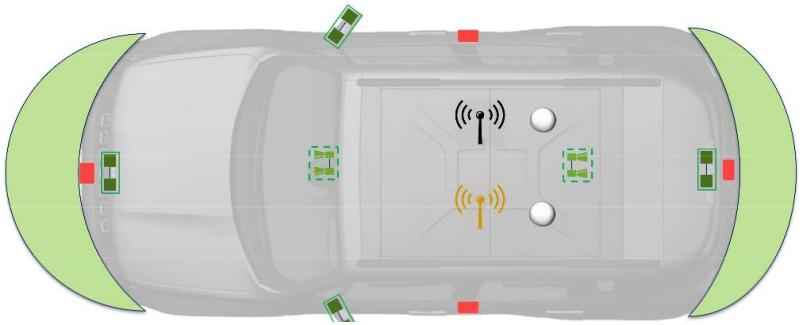
Euclidean clustering detection



- **Objective**: Find and segment the individual object point clusters lying on the plane
  1. Down sample the point cloud
  2. Segment the ground plane
  3. Create a KD-tree representation for the input point cloud dataset
  4. Compute Euclidean distance to build the cluster list
  5. The algorithm terminates when all points have been processed and are now part of the list of point clusters

sudo apt-get install pcl-tools
sudo apt install libpcl-dev

HiPeRT Car

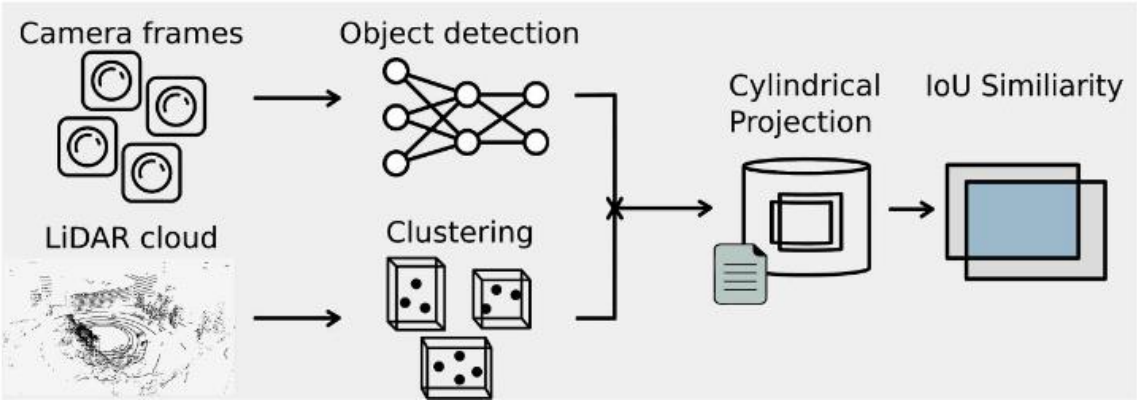


- **Maserati quattroporte**
  - 5 sekonix cameras (GMSL)
  - Ouster OS1 64/128-120m (Ethernet)
  - GPS (CAN)
  - Radar ARS301
  - We have different configurations
    - Pegasus
    - Drive PX2
    - Xavier
    - TX2

Perception HiPeRT SW stack

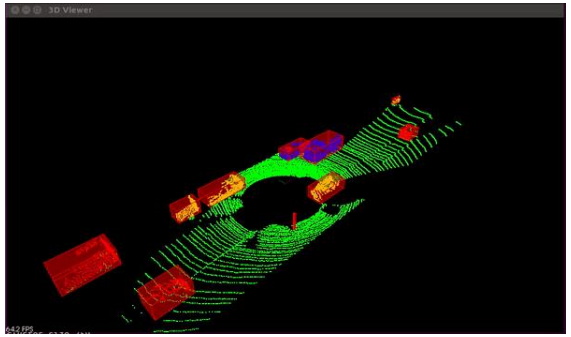
- LiDAR and camera detection (sensor fusion)
  - Clustering - LiDAR (VLP-32C) - **Euclidean clustering**
  - Object detection - Camera (4 sekonix/120ºFOV) - YOLO

https://www.youtube.com/watch?v=qkWqRSgUFmw



M. Verucchi, L. Bartoli, F. Bagni, F. Gatti, P. Burgio and M. Bertogna, "Real-Time clustering and LiDAR-camera fusion on embedded platforms for self-driving cars", in proceedings in IEEE Robotic Computing (2020)

Euclidean clustering detection

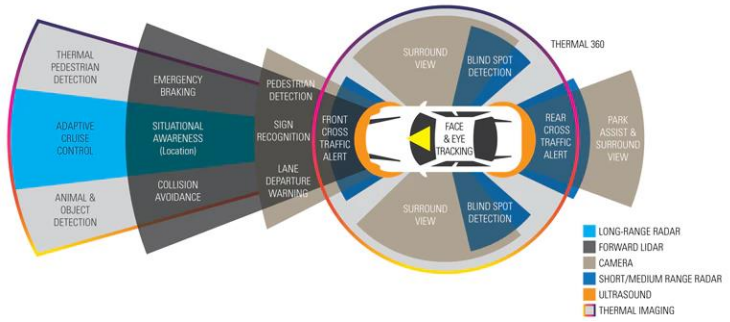


- **Objective**: Find and segment the individual object point clusters lying on the plane
  1. Down sample the point cloud
  2. Segment the ground plane
  3. Create a KD-tree representation for the input point cloud dataset
  4. Compute Euclidean distance to build the cluster list
  5. The algorithm terminates when all points have been processed and are now part of the list of point clusters

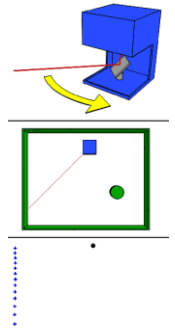
sudo apt-get install pcl-tools
sudo apt install libpcl-dev

Sensors

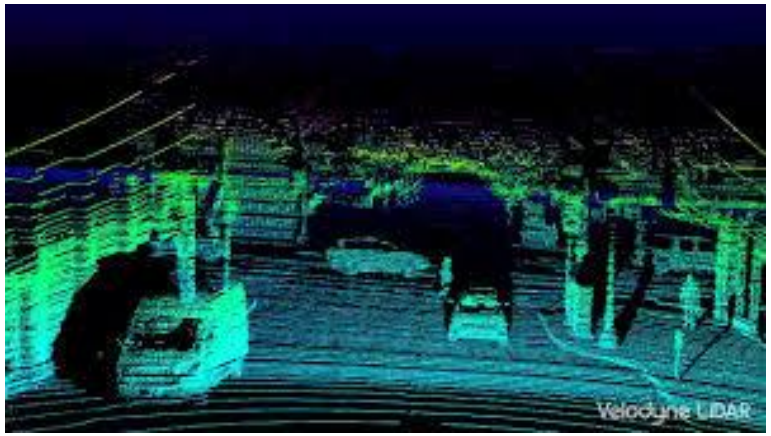
- Sensor device that measures a property of the environment
  - **Exteroceptive**: measure the propierties of the external environment
  - **Propioceptive**: measure the properties of the ego vehicle
- The number and complexity of the sensors has increased drastically over the past few years



LiDAR



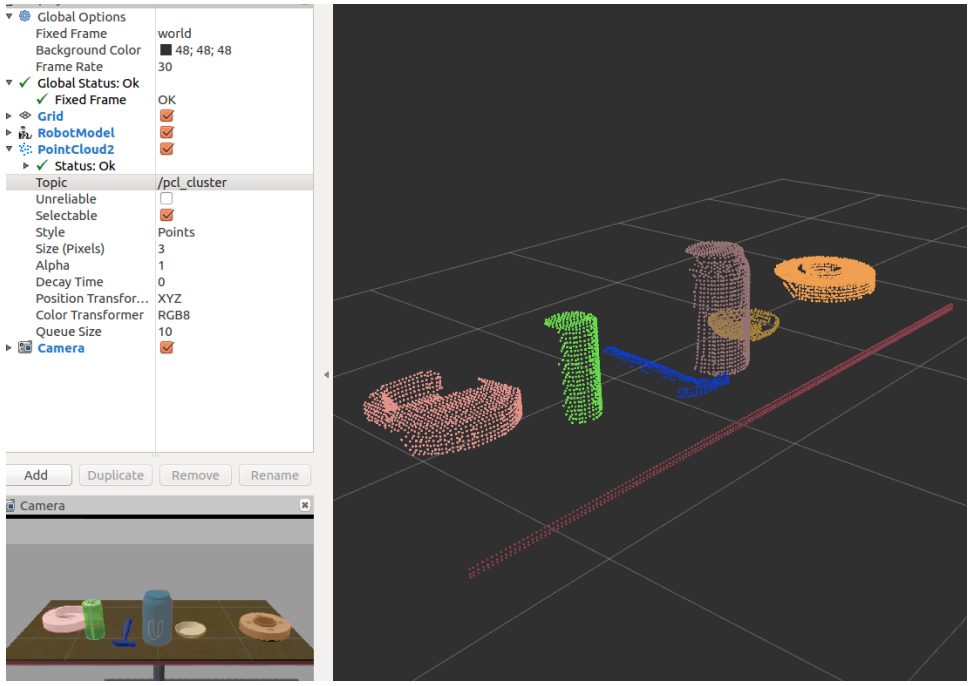
- Light Detection And Ranging (LiDAR) is a technology used to scan objects, **measure distances and height**
- A lidar sensor transmits **laser beams** that bounce off objects and return to the sensor
  - The distance of the objects is determined based on how long it takes the pulses of infrared light to travel back
- Using the information received, a lidar system produces a **point cloud** that looks like a shadow and reflects the object's shape and size
  - HDL-32E generates up to 1.4 million points per second
  - They are affected by bad weather conditions
  - Very expensive
- They are generally used to make high-resolution maps and localize the car in a map but also for **object classification**
  - Computer vision can help in the classification (semantic/classification)



Source: Velodyne

Point cloud exercise

- **Objective**: Find and segment the individual object point clusters lying on the plane (**Euclidean clustering**)
- A clustering method is used to organize an unorganized point cloud into smaller parts
  - identify which points in a point cloud belong to the same object
    1. Down sample the point cloud
    2. Segment the ground plane
    3. Create a KD-tree representation for the input point cloud dataset
    4. Compute Euclidean distance to build the cluster list
    5. the algorithm terminates when all points have been processed and are now part of the list of point clusters



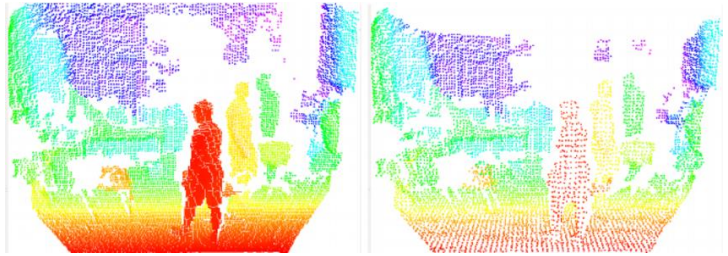
Source: https://pcl.readthedocs.io/projects/tutorials/

PCD data

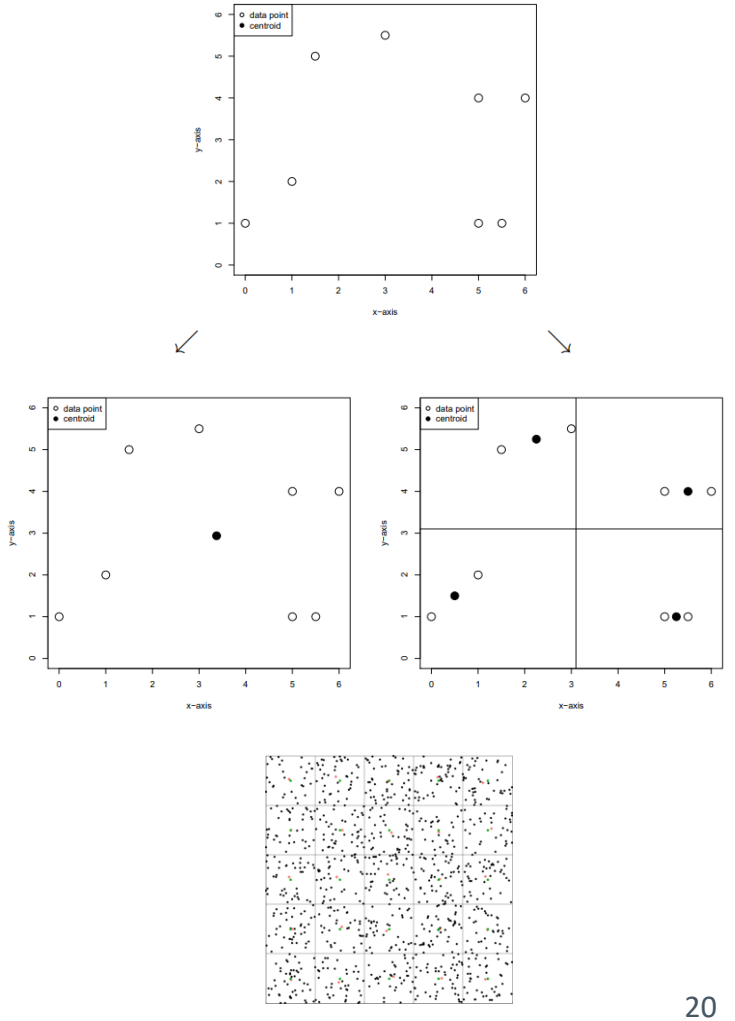
- PCD (Point Cloud Data) is a file format that supports 3D point cloud data
  - **FIELDS** - specifies the name of each dimension/field that a point can have
  - **SIZE** - specifies the size of each dimension in bytes
  - **POINTS** - specifies the total number of points in the cloud
  - **DATA** - specifies the data type that the point cloud data is stored in (ascii or binary)

```
# .PCD v.7 - Point Cloud Data file format
VERSION .7
FIELDS x y z rgb
SIZE 4 4 4 4
TYPE F F F F
COUNT 1 1 1 1
WIDTH 213
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS 213
DATA ascii
0.93773 0.33763 0 4.2108e+06
0.90805 0.35641 0 4.2108e+06
0.81915 0.32 0 4.2108e+06
0.97192 0.278 0 4.2108e+06
0.944 0.29474 0 4.2108e+06
0.98111 0.24247 0 4.2108e+06
```

Voxel filtering



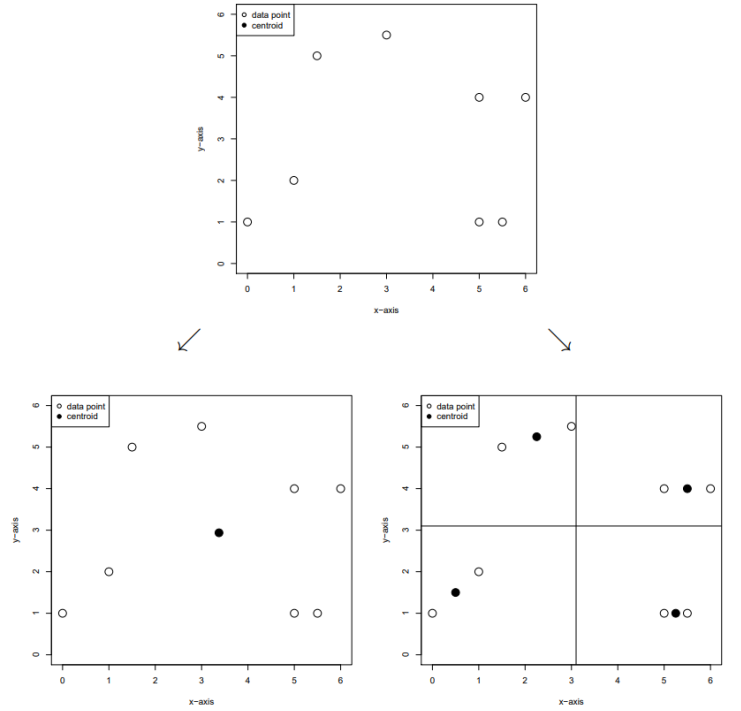
- Point clouds store a lot of detailed information about physical space
  - Not all the information is required
- **Voxel filtering**: technique used to reduce the number of points (also called downsample)
  - The voxel grid filter down-samples the data by taking a spatial average of the points in the cloud (i.e., using the centroid)



https://libpointmatcher.readthedocs.io/en/latest/DataPointsFilterDev/

Voxel filtering

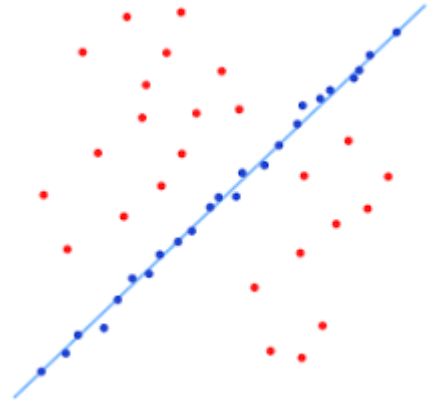
- Black dots are the centroids of each voxel
  - Left → big voxel size
  - Right → smaller voxel size
- A voxel grid filter down samples the data by averaging the points for each voxel grid in the point cloud,
- **setLeafSize(x,y,z) set the voxel grid leaf size**
  - **The bigger the voxel grid size the more the points filtered**



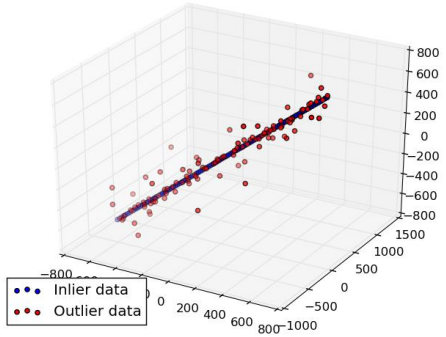

```
pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
sor.setInputCloud (cloud);
sor.setLeafSize (0.01f, 0.01f, 0.01f);
sor.filter (*cloud_filtered);
```

RANSAC

- **We need to separate the ground from the other objects**
- **Random sample consensus** (**RANSAC**) is an iterative method to estimate parameters of a mathematical model from a set of observed data that contains outliers
  - The algorithm groups the data into inliers and outliers
    - The algorithm collects the line with the maximum number of inliers
    - Inliers are the points that represent the table
    - Outliers are the points that represent the objects lying on the table
- 2D Ransac
  - Select two random points
  - Fit a line for those two points
  - Compute the points that are in/close to the line (score)
  - Repeat the process with other two random points
  - Choose the model which has the best “score”

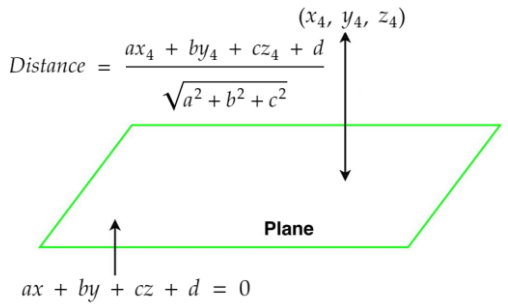


3D RANSAC



- Let us suppose that we want to get the objects (pedestrian, vehicles etc...) that are above the plane
- 3D RANSAC
  - Select three random points
  - Form a plane with these points (plane model)
  - Compute the number of points that fit into the plane model (we consider as **inliers** the points that lies close to the plane)
  - **Repeat** the process N times
  - Select the plane model that “collects” more points

```
// Create the segmentation object for the planar model and set all the parameters
pcl::SACSegmentation<pcl::PointXYZ> seg;
pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_plane (new pcl::PointCloud<pcl::PointXYZ> ());
pcl::PCDWriter writer;
seg.setOptimizeCoefficients (true);
seg.setModelType (pcl::SACMODEL_PLANE);
seg.setMethodType (pcl::SAC_RANSAC);
seg.setMaxIterations (100);
seg.setDistanceThreshold (0.02);
```



K-D Tree

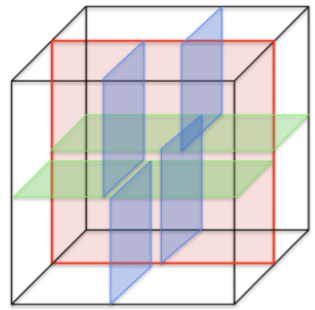
k-d tree

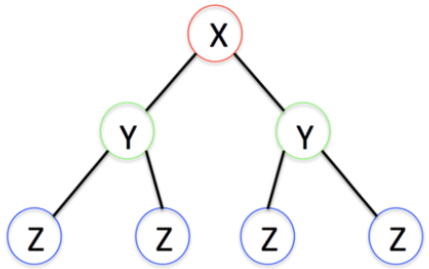
Type	Multidimensional BST
Invented	1975
Invented by	Jon Louis Bentley

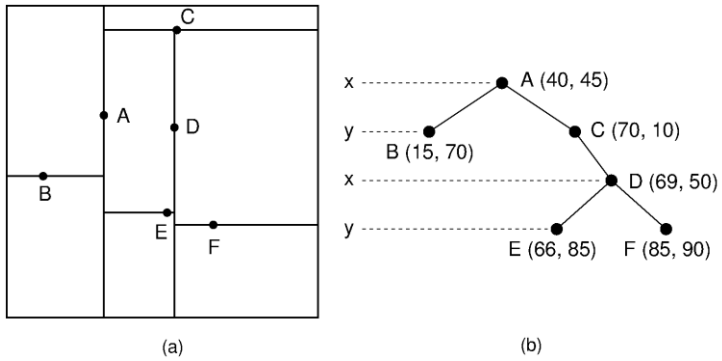
Time complexity in big O notation

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

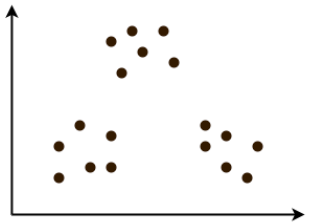
- How to store the point cloud?
  - Arrays? → complex to compute nearest neighbor
- In computer science, a K-D tree (short for k-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional space
  - **It is very useful to compute nearest neighbor points**



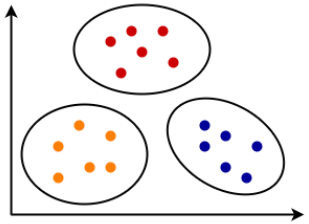




Clustering



Before K-Means



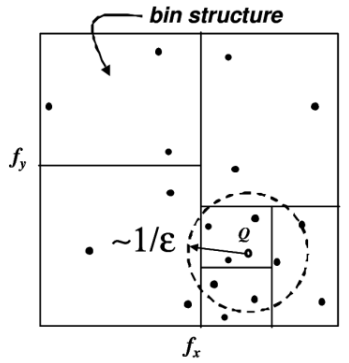
After K-Means

- We need a method to establish the points belong to a certain cluster
- **Euclidean clustering**: we use a K-D tree structure for finding the nearest neighbors
  - We use the Euclidean distance and **a search threshold**
    - Straight line distance between two points

$$searchpoint(x) + threshold > x > searchpoint(x) - threshold$$
$$searchpoint(y) + threshold > y > searchpoint(y) - threshold$$
$$searchpoint(z) + threshold > z > searchpoint(z) - threshold$$

$$Distance_{(searchpoint\ ,\ currentpoint\ )} = \sqrt{(x - searchpoint(x))^2 + (y - searchpoint(y))^2 + (z - searchpoint(z))^2}$$

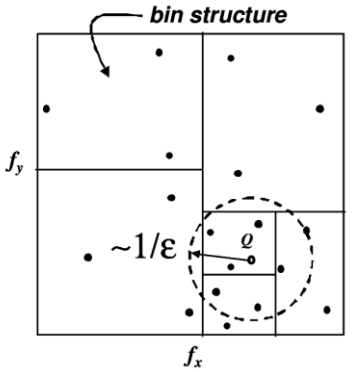
Clustering



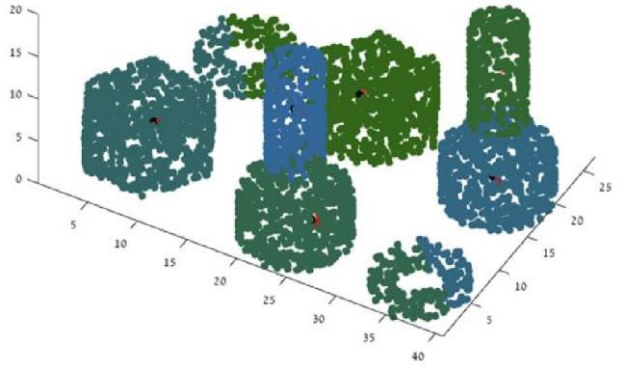
- We use the Euclidean distance and the KD-tree to determine clusters
- **A search radius is defined**
  - *setClusterTolerance* **defines the maximum distance between points when defining the clusters**

```
// Creating the KdTree object for the search method of the extraction
pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZ>);
tree->setInputCloud (cloud_filtered);

// Here we are creating a vector of PointIndices, which contain the actual index information in a vector<int>. The indices of each detected
cluster are saved here.
std::vector<pcl::PointIndices> cluster_indices;
pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;

//Set the spatial tolerance for new cluster candidates
//If you take a very small value, it can happen that an actual object can be seen as multiple clusters. On the other hand, if you set the
value too high, it could happen, that multiple objects are seen as one cluster
ec.setClusterTolerance (0.02); // 2cm

//We impose that the clusters found must have at least setMinClusterSize() points and maximum setMaxClusterSize() points
ec.setMinClusterSize (100);
ec.setMaxClusterSize (25000);
ec.setSearchMethod (tree);
ec.setInputCloud (cloud_filtered);
ec.extract (cluster_indices);
```

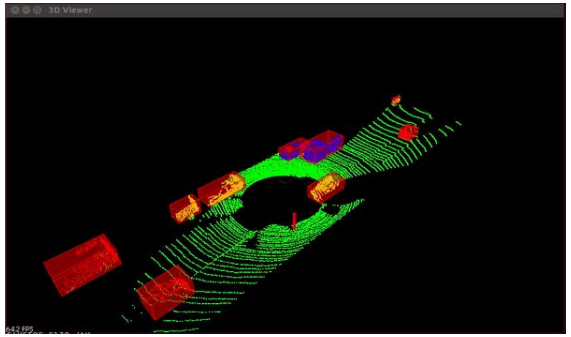
K-D tree nearest neighbors



1. create a kd-tree representation for the input point cloud dataset $\mathcal{P}$;
2. set up an empty list of clusters C, and a queue of the points that need to be checked Q;
3. then for every point $\boldsymbol{p}_i \in \mathcal{P}$, perform the following steps:
   - add $\boldsymbol{p}_i$ to the current queue Q;
   - for every point $\boldsymbol{p}_i \in Q$ do:
     - search for the set $\mathcal{P}_i^k$ of point neighbors of $\boldsymbol{p}_i$ in a sphere with radius $r < d_{th}$;
     - for every neighbor $\boldsymbol{p}_i^k \in \mathcal{P}_i^k$, check if the point has already been processed, and if not add it to Q;
   - when the list of all points in Q has been processed, add Q to the list of clusters C, and reset Q to an empty list
4. the algorithm terminates when all points $\boldsymbol{p}_i \in \mathcal{P}$ have been processed and are now part of the list of point clusters C.



Euclidean clustering detection



- **Objective**: Find and segment the individual object point clusters lying on the plane
  1. Down sample the point cloud
  2. Segment the ground plane
  3. Create a KD-tree representation for the input point cloud dataset
  4. Compute Euclidean distance to build the cluster list
  5. The algorithm terminates when all points have been processed and are now part of the list of point clusters

```
sudo apt-get install pcl-tools
sudo apt install libpcl-dev
```