Ausdruck

Sunday, December 8, 2019 12:28 AM

Intro to Sensor Fusion

Wednesday, November 13, 2019 7:55 PM

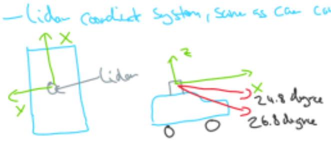
- Different Sensors
  - active
  - passive
- Lidar
- Radar -> Doppler, Fourier, ...
- Camera
- Kalman Filters -> EKF, UKF
- Projects for each
- Fusion from 3D to 2D, Lidar, Radar & camera
- Sensor Strength & weakness
  - -> Fusion for great Perception -> uncertainty
- Many Sensors: Odometry, Lidar, Camera, Radar, ...
- Fusion -> Reduce uncertainty, more precise
- Kalman Filters are used for fusion, more advanced concepts like EKF, UKF for non linearity

LIDAR COURSE

- MBDRNA -> Sensor Fusion MB USA
- Lidar: sends lasers, scanning, photodetector for detection
- Point cloud: set of all Lidar beam reflections, 100 MB/s is called here
- PCL Data:

$$\text{256k Points} \left\{ \begin{array}{l} (x, y, z, I) \quad I: \text{Intensity} \\ (x, y, z, I) \end{array} \right.$$

  - -> Velodyne VLP64 - Lidar
- Lidar coordinate system, same as car coordinate system

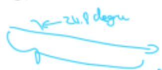


Car

Top Down View

Side View

Exercise:



33.35 ns @ $3 \cdot 10^8 \frac{m}{s}$ => 10.005 m

=> $z = \sin(24.8°) \cdot 10.005 \approx 4m$

- PCL Library
  - -> mainly Segmentation, Clustering, also OpenCV
  - -> open source for C++, Rendering, ...
- mount lidar on the roof for max view -> SUVs are good bc of height
- 2° in 60m -> pedestrian is invisible => more layers for further view
- PCL viewer
  - for graphics
  - render points & shapes
  - init camera & init Highway and use it
  - explore camera options

- init camera & initHighway and use it
- explore camera options

– Lidar models simulated:
  - ray tracing
  - simulate materials, environment, ...

- PCL datatype:
  pcl::PointCloud<pcl::PointXYZ>::Ptr
- Templates -> bc there are different PCL types
- is pcl::PointCloud<PointT>::PTR a Value or type? -> type
  ↳ needs to be passed with: typename pcl:: ..., bc the compiler needs to know whether it's a Value or type, it will assume a Value when not specified
- for realistic Lidar: more Layers, noise, set min Distance, ...

Segmentation

- Segment Ground & Obstacle Plane
- Detect Objects in Obstacle Plane
- Detect lanes / free space in ground plane

Create Point Processor
- has all the methods: filters, segmentation, clustering, ...
- uses templates

PCL to Segment Planes
- function returns std::pair<typename pcl::PointCloud<PointT>::Ptr, ...>
  ↳ 2 Clouds as tuple: ground plane & Obstacle Plane
- Separate Clouds with Separate Clouds function
- use it inside of SegmentPlane with calculated inliers & cloud
- to generate the obstacle cloud we use the extract object
  ↳ substract plane cloud from input cloud

RANSAC

- method to find best plane / line
- no LR: bc all points are considered
- find model with outliers
- randomly chooses Subset, fits line to these points
- looks for most inliers (based on distance) => best model criteria
- other methods that only consider e.g. 20% of total points for sampling
  ↳ calc error -> lowest error line is the best one -> might be better bc not all points need to be considered

RANSAC for lines

- randomly sample 2 points
- fit a line
- calculate inliers

Equation: (2D, 2 points)

$Ax + By + C = 0 \Rightarrow (y_1 - y_2)x + (x_2 - x_1)y + (x_1 \cdot y_2 - x_2 \cdot y_1) = 0$

Point (x, y)

Distance $d = |Ax + By + C| / \sqrt{A^2 + B^2}$

Extending RANSAC to planes

- 3 points needed

$Ax + By + Cz + D = 0$

$p_1 = (x_1, y_1, z_1)$ $p_2 = (x_2, y_2, z_2)$

$p_3 = (x_3, y_3, z_3)$

Vector 1: von p_1 zu p_2

Vector 2: von p_1 zu p_3

$p_3 = (x_3, y_3, z_3)$

Vector 1: von p_1 zu p_2

Vector 2: Von p_1 zu p_3

$v_1 = \langle x_2 - x_1, y_2 - y_1, z_2 - z_1 \rangle$

$v_2 = \langle x_3 - x_1, y_3 - y_1, z_3 - z_1 \rangle$

Find Normal vector of plane (cross product)

$v_1 \times v_2 = \langle (y_2 - y_1)(z_3 - z_1) - (z_2 - z_1)(y_3 - y_1),$
$(z_2 - z_1)(x_3 - x_1) - (x_2 - x_1)(z_3 - z_1),$
$(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1) \rangle$

Simplification: $v_1 \times v_2 = \langle i, j, k \rangle$

then:

$i(x - x_1) + j(y - y_1) + k(z - z_1) = 0$

$ix + jy + kz - (ix_1 + jy_1 + kz_1) = 0$

$A = i$

$B = j$

$C = k$

$D = -(ix_1 + jy_1 + kz_1)$

Distance point to plane:

$d = |A \cdot x + B \cdot y + C \cdot z + D| / \mathrm{sqrt}(A^2 + B^2 + C^2)$

Std::tuple <float, float, float> can be used for vector representation, not necessary though

Clustering

- Group points by how close they are to each other
- K-D tree speeds up NN search
- Cluster points to objects

Euclidean Clustering with PCL

- fill in cluster function in Point Processor
- Render the different clusters in environment.cpp
- uses a distance tolerance, min & max for points that represent a cluster
  - ↳ small clusters: could be noise
  - ↳ v. large ones: overlapping of clusters
  - ↳ tolerance helps resolve this
- KD tree is built using input cloud (obstacle cloud)

KD Tree

- first point is the root
- all the other points will be left of prev. point if x value is smaller than the root & right otherwise

Visualization:

1st point (-6.2, 7)

2nd point (-6.3, 8.4) goes left

red line: splitting y-dimension

after 2 points:

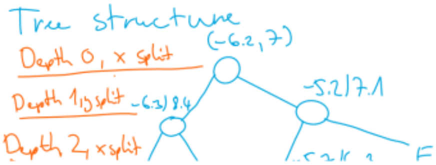
if depth is even : x-split

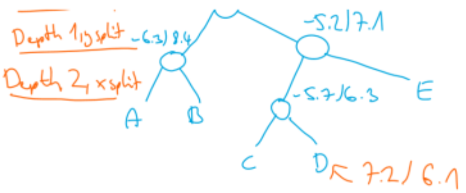
is odd : y-split

(-5.2, 7.1) &

(-5.7, 6.3) → inserted 1st

Tree structure





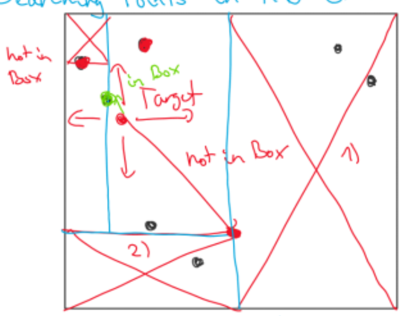
→ always traverse from root when inserting new value

Improving the tree

- insert points that alternate between each splitting region
- Median values will split the regions more evenly
- 2D example: insert x median, then y median, then x median, ...

→ improves search time

Searching Points in KD tree



→ Split x-Region
↳ don't have to check right region bc value is less than x of root
→ then split y-Region
→ Do for all points
→ a big computational advantage, especially with large PCs

- Box Square is 2× distanceTol in length, positioned around target point
- if other point is inside the Box, the Id is added to the list of nearby Ids

Clustering over KD Tree

```
Proximity(point, cluster):
  if point has not been processed
    mark point as processed
    add point to cluster
    nearby points = tree(point)
    iterate through nearby points
      Proximity(nearby point, cluster)

EuclideanCluster():
  list of clusters  //cluster is represented by list of ids
  iterate through points
    if point has not been processed
      create cluster
      Proximity(point, cluster)
      cluster add clusters
  return clusters
```

Filtering and replaying real PCD

Voxel Grid Presentation

→ Downsample for faster processing

Voxel Grid Filtering

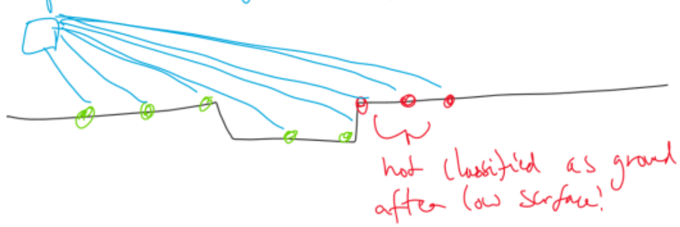
1. Reduce num. of points in cloud to process faster
2. Cubic Grid that filters cloud leaving single point per cell
3. Filter inside box removing data outside box (ROI based)

2. Cubic Grid that filters cloud leaving single point per cell
3. Filter inside box removing data outside box (ROI based)
(4. Remove roof points)
5. Fill in Filter function in Point Processor

Voxel: 3D pixel => Minecraft Block

IBEO Ground Segmentation

- ground points are not always a plane geometry
  -> even in flat environment, problem in longer distances
- weighted average based on neighbor distances to compute the normal
- problem with geometric features



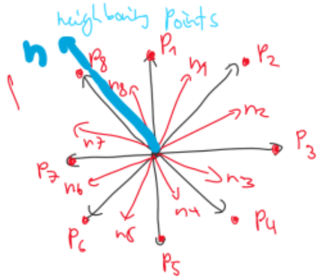
Proposed:

1. Local Feature Extraction
   - computes local geometric features & feature qualities for each point individually
     -> find possible Ground Point candidates

LFA:

1.) Surface normal estimation n
2.) Surface normal quality measurement n_p
3.) Ground candidate quality measurement g_p

- n for each point is calculated based on surface normals between adjacent neighboring points



Direction vector d_i is given by: $d_i = P_i - P$

$$n_i = \begin{cases} d_i \times d_{i-1} & \text{if } 2 \leq i \leq 8 \\ d_1 \times d_8 & \text{otherwise} \end{cases}$$

-> flat horizontal normals point straight up: $[0,0,1]^T$

$\angle\, d_1 \times d_2$ [illegible]

=> flat horizontal normals point straight up: $[0,0,1]^T$

once all n_i are computed, the surface normal n is computed by:

$$n = \frac{\sum_{i=1}^{8} n_i \cdot \frac{1}{|d_i| + |d_{i-1}|}}{\left| \sum_{i=1}^{8} n_i \cdot \frac{1}{|d_i| + |d_{i-1}|} \right|}$$

-> weighted scheme focuses more on closer points
-> Lidar sensors are usually tilted
-> reduces noise

N_p

- calculated based on cosine similarity btw n & n_i
- the more similar the n_i s are to mean n, the higher n_p

$$n_p = \frac{1}{8} \cdot \sum_{i=1}^{8} \left(\frac{\langle n, n_i \rangle}{|n| \cdot |n_i|} \right)$$

G_p

- Local feature
- ego vehicle is on the ground, cloud given in car coordinates

$$g_p = n \cdot [0,0,1]^T \cdot n_p$$

Clustering & Classification

- High g_p for points w/ upwards n vectors
- to discard ground points, they are clustered based on distance & g_p
- PCL EC is used, Ground = largest cluster in area
-> if smaller clusters fall into same area -> non ground points
- other large clusters are also identified as ground

-> Faster Runtime than RANSAC (2x as fast on HW)

-> 97% & 93% of GP detected

-> handle step scenarios w/o excluding points behind targets