# Apollo Obstacle detection module

## Code Location :

1. Both deploy.pt(model for camera perception) and deploy.md can be found in /apollo/modules/perception/production/data/perception/camera/models/yolo\_obstacle\_detector/3d\_yolo
2. Location of the YOLO obstacle detector used for object detection:

<https://github.com/ApolloAuto/apollo/blob/master/modules/perception/camera/lib/obstacle/detector/yolo/yolo_obstacle_detector.cc#L340>

1. Location of the 3d YOLO model (deploy.pt file for 3D YOLO):

/apollo/modules/perception/production/data/perception/camera/models/yolo\_obstacle\_detector/3d\_yolo

1. The Yolo used for training are found in modules/perception/production/data/perception/camera/models/yolo\_obstacle\_detector,

While that for inference are found in modules/perception/inference/inference\_test\_data/yolo/deploy.pt

1. Look at multicue\_obstacle\_transformer.h in modules/perception/camera/lib/obstacle/transformer/multicue/multicue\_obstacle\_transformer.h

## Introduction :

In Apollo, YOLO [1][2] was used as a base network of object and lane segment detection. The object has vehicle, truck, cyclist, and pedestrian categories and represented by a 2-D bounding box with orientation information. The lane lines are detected by segmentation using the same network with some modification.

It should be noted that traffic light detection is also a subset of object detection. For traffic light detection, we use vanilla YOLO, where as for 3D-obstacle detection, specially depth estimation, we have to add a post-processing step to convert the 2D bounding box output from the YOLO operation into 3D. We will discuss this post-processing step in detail under the chapter “YOLO - 2D TO 3D Conversion”.

[YoloObstacleDetector](https://github.com/ApolloAuto/apollo/blob/master/modules/perception/camera/lib/obstacle/detector/yolo/yolo_obstacle_detector.cc) is the wrapper for YOLO detector. [In 'Detect' method](https://github.com/ApolloAuto/apollo/blob/master/modules/perception/camera/lib/obstacle/detector/yolo/yolo_obstacle_detector.cc#L287) this class calls the inference.[Source : Apollo Github issue #6556].

[The inference class](https://github.com/ApolloAuto/apollo/blob/master/modules/perception/inference/inference.h) is a base abstract class for inference.

[Here](https://github.com/ApolloAuto/apollo/blob/master/modules/perception/inference/caffe/caffe_net.cc) is the caffe implementation of base inference class.

**Most relevant GitHub issue answering the 2D-to-3D conversion :**

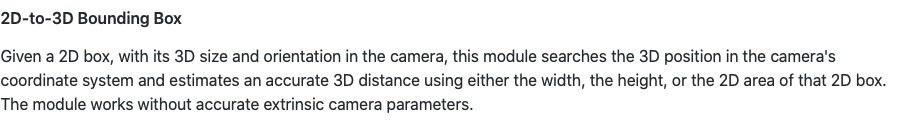
<https://github.com/ApolloAuto/apollo/issues/5819>

**Input of YOLO3D** : Camera images which are are divided into SXS matrix by YOLO.

**Output** : The Apollo's Yolo detector **outputs** 2D detections and 3D dimensions of obstacles as well as yaw orientation.

**2D-TO-3D conversion** : Based on the output data the [Transformer](https://github.com/ApolloAuto/apollo/blob/master/modules/perception/camera/lib/obstacle/transformer/multicue/multicue_obstacle_transformer.cc#L215) calculates a 3D position of object in the map frame. Inside the Transformer code, to be found under /apollo/modules/perception/camera/lib/obstacle/transformer/multicue/multicue\_obstacle\_transformer.cc , the **Solve3dBbox** function carries out the 2D to 3D conversion.

The rest of this report will be dedicated to explaining the following in greater detail :



Source :

<https://github.com/ApolloAuto/apollo/blob/master/docs/specs/perception_apollo_3.0.md>

## 

## YOLO - 2D TO 3D Conversion after Monocular camera object detection

Main sources :

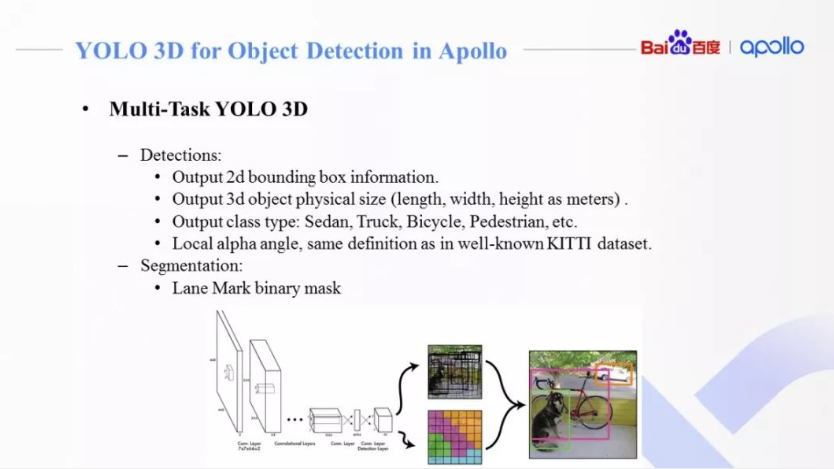
[1].<http://www.programmersought.com/article/4188546438/;jsessionid=60F21BA6B5C33F987DEE0177B4E13811>

[2]. Zoox's paper "3D Bounding Box Estimation Using Deep Learning and Geometry";

[3]. GitHub implementation of the above paper : : <https://github.com/smallcorgi/3D-Deepbox>

[4] Supplementary material to [2] - Solving X,Y through SVD - <http://bit.ly/2oYMpuw>

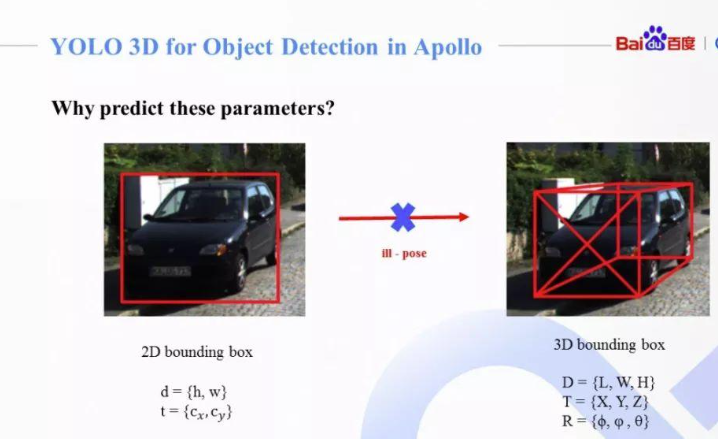
[5] Medium Article explaining the paper[2] and the math- <https://towardsdatascience.com/geometric-reasoning-based-cuboid-generation-in-monocular-3d-object-detection-5ee2996270d1>

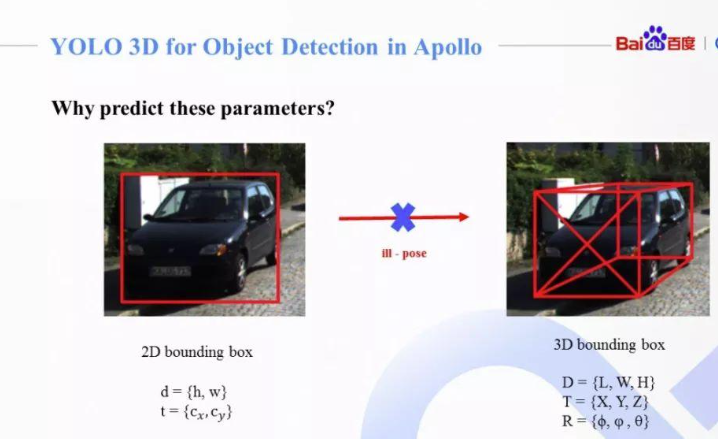


**Output of YOLO 3D :**

The final output of the object detection includes a 2D frame (in pixels), a 3D real object size (in meters), an obstacle category and an obstacle relative deflection angle (Alpha Angle, consistent with the KITTI data set definition).

Our objective is to obtain 3-D box dimensions (Dimension D ={L,W,H}), Translation(Coordinates of centre of 3DBbox) T={X,Y,Z} and Orientation R = {φ, φ, θ}, from 2-D dimensions. The following diagram is self-explanatory :





Figures taken from [1].

**What is an ill-pose ?**

An ill-pose is where you have to predict the 9-dimensions of a 3-D pose from 4-Dimensions of 2-D, a seemingly impossible task. However, with the help of a few real-world assumptions, we will convert it into a well-pose problem, where we will only have to predict 6 of the 9 dimensions.

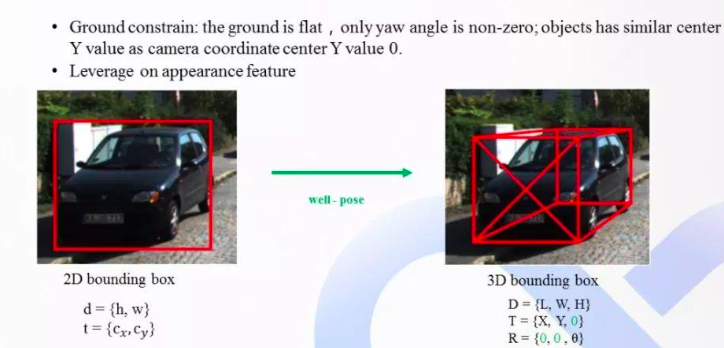


Fig. 3 - Cited from [1]

To predict D(dimension) and R(orientation), we use CNN[2], while we obtain T={X,Y} through SVD Optimisation[4].

It may be noted here that we choose to regress the box dimensions D rather than translation T because the variance of the dimension estimate is typically smaller (e.g. cars tend to be roughly the same size) and does not vary as the object orientation changes: a desirable property if we are also regressing orientation parameters.

It may be added here that after obtaining {X,Y} through SVD, we may calculate distance of the obstacle from the car(monocular camera) with the following formula : Distance=sqrt(X^2+Y^2). So by getting X and Y, we can naturally get the real distance of the obstacle from the camera, which is one of the final requirements of monocular ranging.

This operation can also be found in the Apollo code base under the path modules/perception/obstacle/transformer/multicue/obj\_mapper.cc - Look for the line : *“float dist\_rough = sqrtf(common::ISqr(center[0]) + common::ISqr(center[2]));”*

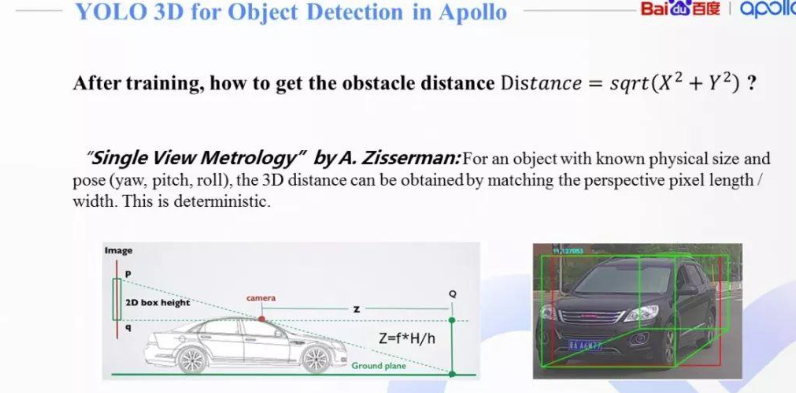
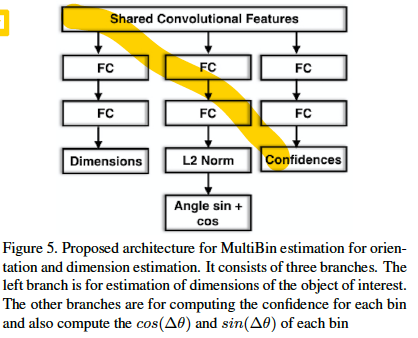


Fig taken from [1].

### Architecture of CNN which is used for predicting D and R :



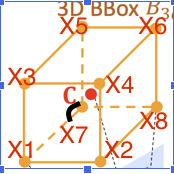
Assumptions based on which we can reduce the number of 3D params from 9 to 6:

1. We assume that the 3D obstacle only rotates along the axis of the vertical ground, while the other two directions do not rotate, that is, only the yaw offset angle changes with time, and the remaining Pitch Rolls are all zero.
2. The height of the obstacle center is equal to the height of the camera, so it is possible to take Z=0 .
3. We can assume that D ={L,W,H} has very low variance(because the physical size of the obstacles of the same category generally does not appear in magnitude), so we can train a CNN to predict it.

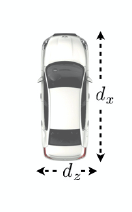
### 3-D Bounding Box Estimation

Note : This part is taken directly from [2], Section 3, and deals with finding out

Assuming that the dimensions of 3D box are D=[dx; dy; dz], the coordinates of the 3D bounding box are [+-dx/2,+-dy/2,+-dz/2] .



For example, from the above figure, its apparent that if the centre of the 3D BBox be C(0,0,0), then X1 coordinates should be (dx/2,-dz/2,-dy/2) and so on, where dx,dz are along the length and breath of the car respectively, as is shown in the following Fig



X1 =

[dx=2; dy=2; dz=2]T , X2 = [-dx=2; dy=2; dz=2]T , : : : ,

X8 = [-dx=2; -dy=2; -dz=2]T . The constraint that the

3D bounding box fits tightly into 2D detection window requires

that each side of the 2D bounding box to be touched

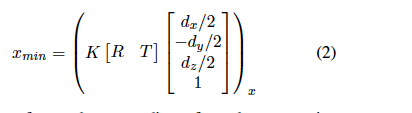
by the projection of at least one of the 3D box corners.

For example, consider the projection of one 3D corner

X0 = [dx=2; -dy=2; dz=2]T that touches the left side of

the 2D bounding box with coordinate xmin. This point-toside

correspondence constraint results in the equation:



where (:)x refers to the x coordinate from the perspective

projection. Similar equations can be derived for the remaining

2D box side parameters xmax ; ymin; ymax . In total the

sides of the 2D bounding box provide four constraints on

the 3D bounding box. This is not enough to constrain the

nine degrees of freedom (DoF) (three for translation, three

for rotation, and three for box dimensions).

Details of how to solve for translation, ie the coordinates of the centre of the 3DBBox[tx,ty,tz], using SVD are included in

the supplementary material [4].

### Regressing the 3D bounding box orientation and dimension

Note : This part is taken directly from Section4 of [2].

Orientation : As has been stated above, the rotation R(theta) is parametrized only by azimuth(theta) (yaw).

Although

the global orientation R(theta) of the car (its 3D bounding

box) does not change, its local orientation theta-l with respect to the ray through the crop center does, and generates

changes in the appearance of the cropped image - Look at Fig.3 and Fig.4 of [2] .

Thus, instead of regressing the whole yaw angle theta, we simply train our CNN to regress theta-l.

the local orientation angle theta-l and the

ray angle change in such a way that their combined effect

is a constant global orientation of the car.

Given intrinsic

camera parameters, the ray direction at a particular pixel is

trivial to compute. At inference time we combine this ray

direction at the crop center with the estimated local orientation

in order to compute the global orientation of the object.

Now we will talk a little about the loss-function used here. The authors of the paper have named the approach “MultiBin Orientation”.

Here is a brief description of the MultiBin architecture approach for Orientation Estimation :

We first discretize the orientation

angle and divide it into n overlapping bins. For each

bin, the CNN network estimates both a confidence probability

ci that the output angle lies inside the ith bin and the

residual rotation correction that needs to be applied to the

orientation of the center ray of that bin in order to obtain

the output angle. The residual rotation is represented by

two numbers, for the sine and the cosine of the angle. This

results in 3 outputs for each bin i: (ci; cos(i); sin(i)).

Valid cosine and sine values are obtained by applying an L2

normalization layer on top of a 2-dimensional input. The

total loss for the MultiBin orientation is thus:

**L-theta = Lconf + wX Lloc**

**The confidence loss Lconf is equal to the softmax loss of**

**the confidences of each bin. Lloc is the loss that tries to**

**minimize the difference between the estimated angle and**

**the ground truth angle in each of the bins that covers the**

**ground truth angle, with adjacent bins having overlapping**

**coverage.**

**Note** : Please read Section 4 and specially Fig 5. of [2] for further details.

**Summary of loss functions used :**

1. L2 loss function for dimension D{l,b,h} since the distribution is believed to be low-variance.

The loss for dimension estimation Ldims is computed as follows:



D\* are the ground truth dimensions of the box, D are the mean dimensions for objects of a certain category and

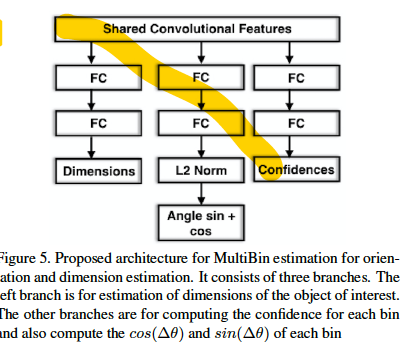
δ is the estimated residual with respect to the mean that the

network predicts. So δ is predicted by our CNN.

1. MultiBin loss for Orientation estimation Loss L-theta



The whole CNN architecture for both Dimension and Orientation is depicted below :



There are three branches: two

branches for orientation estimation and one branch for dimension

estimation. All of the branches are derived from

the same shared convolutional features and the total loss is

the weighted combination of L = α X Ldims + L-theta.

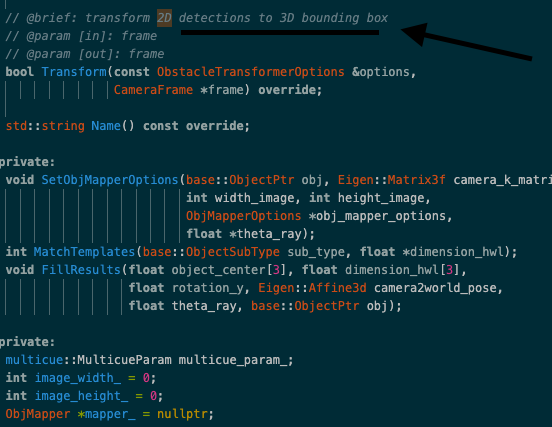
It should be noted here that the Loss - Functions mentioned here are used for training, not for inference.

**Apollo code location of above stated loss functions** : Look at train.pt under modules/perception/production/data/perception/camera/models/yolo\_obstacle\_detector/3d-r4-half - Cntrl-F “Loss Layers”.

**Note1** : The Yolo used for training are found in modules/perception/production/data/perception/camera/models/yolo\_obstacle\_detector,

While that for inference are found in modules/perception/inference/inference\_test\_data/yolo/deploy.pt

**Note2 :** Look at multicue\_obstacle\_transformer.h in modules/perception/camera/lib/obstacle/transformer/multicue/multicue\_obstacle\_transformer.h - Look at the comment on the 2D-to-3D conversion process :



This is exactly as described in [2].

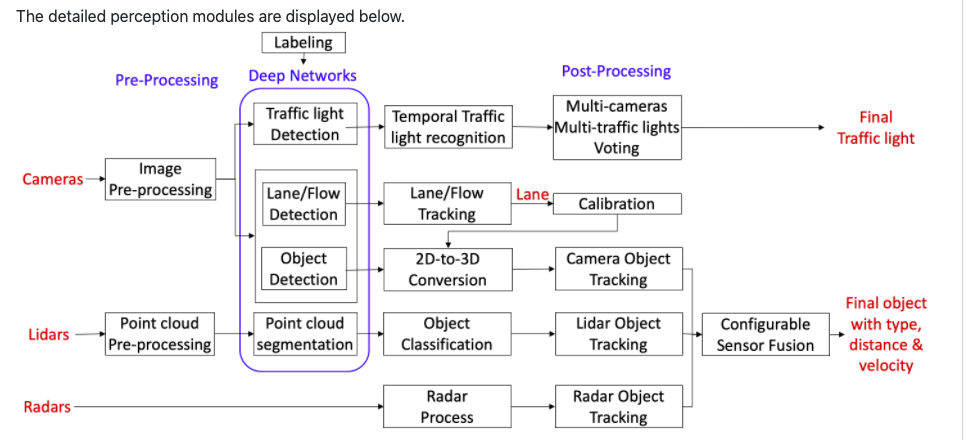
## Future Work :

1. Read the paper “Single View Metrology” by A. Zisserman.
2. Watch the video “How Tesla trains neural networks to perceive depth” (Youtube).
3. The above video cites the papers 1.Unsupervised Learning of Depth and Ego-Motion from Video(<https://people.eecs.berkeley.edu/~tinghuiz/projects/SfMLearner/cvpr17_sfm_final.pdf>) , 2.Depth from Videos in the Wild: Unsupervised Monocular Depth Learning from Unknown Cameras(<https://arxiv.org/pdf/1904.04998.pdf>)
4. Watch the video - “Can self-driving cars learn depth - 2 minute video” (Youtube)

# Obstacle Tracking

## Introduction :

We use Kalman Filter for obstacle tracking. Kalman filters are used in all 3 sensor tracking, namely Camera , Lidar and Radar, but we wont bother ourselves with LIDAR and RADAR at the moment. We will deal exclusively with Camera Object tracking, and specifically the use of Kalman Filter in it. Kalman Filters are also used in sensor Fusion, and the best results from the 3 sensors are chosen to calculate the final output of the perception module - Object type, distance and velocity.



Code Paths :

1. modules/perception/camera/lib/obstacle/tracker/common/kalman\_filter.cc(for Camera Obstacle tracking)
2. modules/perception/fusion/common/kalman\_filter.h
3. <https://github.com/ApolloAuto/apollo/blob/master/modules/perception/fusion/lib/data_fusion/motion_fusion/kalman_motion_fusion/kalman_motion_fusion.cc> (Kalman filter for motion fusion)

Source :

1. Kalman Filter for tracking (Note : doesnt involve camera) : :<https://medium.com/intro-to-artificial-intelligence/extended-kalman-filter-simplified-udacitys-self-driving-car-nanodegree-46d952fce7a3>
2. Medium article on camera obstacle tracking(Note : This one is the main article) :

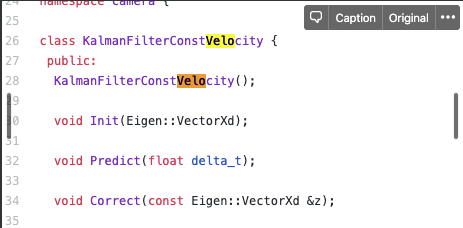
<https://towardsdatascience.com/computer-vision-for-tracking-8220759eee85>

Types of Kalman Filter used for multi-sensor fusion for Apollo, as well as camera obstacle tracking - EKF(Extended Kalman), while that for radar object tracking is Adaptive Kalman Filter**.**

<https://github.com/ApolloAuto/apollo/issues/10957>

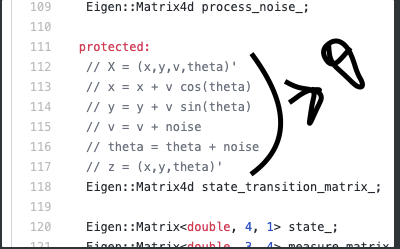
## Camera Obstacle Tracking

Camera Tracking using Kalman Filter uses “ constant velocity “ model which is evident from the snapshot taken from <https://github.com/ApolloAuto/apollo/blob/master/modules/perception/camera/lib/obstacle/tracker/common/kalman_filter.h>



It wasnt apparent from the documentation itself if the Camera Obstacle tracking module uses Kalman Filter, so in the following section I will furnish some proof in its favour.

The following snippet is also taken from <https://github.com/ApolloAuto/apollo/blob/master/modules/perception/camera/lib/obstacle/tracker/common/kalman_filter.h> (ie the code on camera obstacle tracking) :

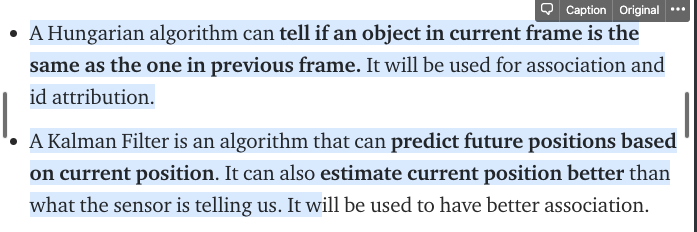


As the snippet suggests, the inputs to the Kalman Filter x,y(coordinates of the centre) as well as theta are taken from the output of the 2D-to-3D conversion module - found in multicue\_obstacle\_transformer.cc in /apollo/modules/perception/camera/lib/obstacle/transformer/multicue\_obstacle\_transformer.cc



The Medium article <https://towardsdatascience.com/computer-vision-for-tracking-8220759eee85> also buttresses this idea.

The algorithm used for "association" is called Hungarian Algo(more on the Algo in the medium article - Computer Vision for tracking).



We have two lists of boxes from YOLO : a tracking list (t-1) and a detection list (t) - Calculate association rule between this 2 lists using Hungarian Algo - What we get from that is matrix of what element in detection matches what element in tracking - from that we can output matched detections, unmatched detections and unmatched trackings .

The matched detections give us the state space variable X consisting of the coordinates of the center of the bounding box (cx,cy), size of the box (width, height) and the change of each of these parameters, velocities. So Kalman filter is initialised with the x, y, w, h that we receive from the Hungarian Algorithm, and the velocities are set to 0. They will then be estimated by the Kalman Filter.

So using the bounding box at time t as input to the Kalman filter , we can predict the bounding boxes at time t+1 and then update our prediction with the measurement at time t+1.

So normal workflow of Camera Object tracking is :

detect obstacles using object detection Algo(YOLO v3) → match the bounding box with former bounding boxes(Hungarian Algo(A Hungarian algorithm can tell if an object in current frame is the same as the one in previous frame. It will be used for association and id attribution.) → predict future bounding box or actual position using Kalman Filter.

Hungarian Algo implemented in modules/perception/common/graph

## 

## Sensor-Fusion based Multi-Object Tracking

Various methods for MOT(Moving Object Tracking) exist in the literature. They can be mainly categorized into six classes: traditional, model based, stereo vision based, grid map based, sensor fusion based, and deep learning based - Great Survey of various methods of MOT -<https://arxiv.org/pdf/1901.04407.pdf> (Section 3.4 onwards).

Apollo uses Sensor Fusion based MOVING OBJECT TRACKING(MOT). [<https://arxiv.org/pdf/1901.04407.pdf>] - "Sensor fusion-based methods fuse data from various kinds of sensors (e.g., LIDAR, RADAR, and camera) in order to explore their individual characteristics and improve environment perception."

Sensor fusion-based methods fuse data from various kinds of sensors (e.g., LIDAR, RADAR, and camera) in order to explore their individual characteristics and improve environment perception.

The MOT subsystem is divided into two layers. The sensor layer extracts features from sensor data that may be used to describe a moving obstacle hypothesis according to either a point model or a box model. The sensor layer also attempts to associate features with currently predicted hypotheses from the fusion layer(using Hungarian Algorithm). Features that cannot be associated to an existing hypothesis are used to generate new proposals. Based on proposals and observations provided by the sensor layer, the fusion layer selects the best tracking model for each hypothesis and estimates (or updates the estimation of) the hypothesis state using a Kalman Filter - see the Kalman filter code under<https://github.com/ApolloAuto/apollo/blob/master/modules/perception/fusion/lib/data_fusion/motion_fusion/kalman_motion_fusion/kalman_motion_fusion.cc>

Since we have limited the scope of our current discussion to camera-based obstacle tracking, I wont go into greater detail on Sensor-fusion based obstacle tracking here.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*