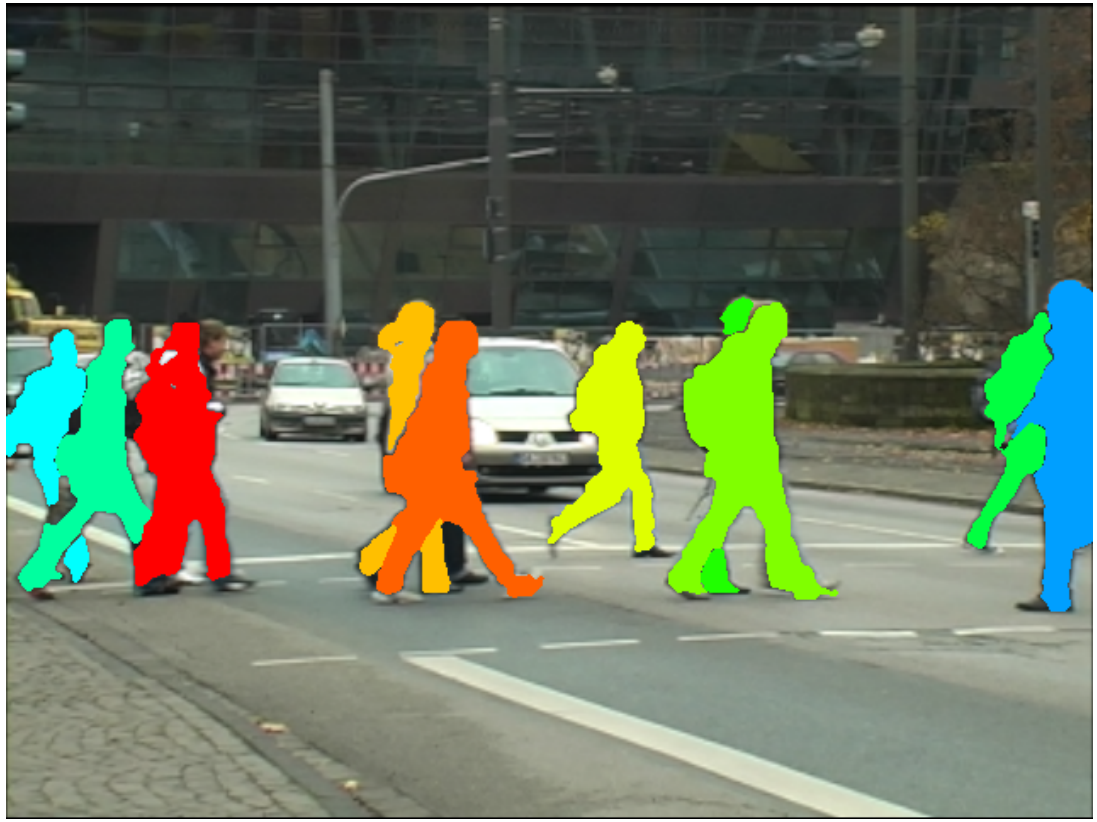


> *In Computer Vision, one of the most interesting area of research is obstacle detection using Deep Neural Networks. A lot of papers went out, all achieving SOTA (State of the Art) in detecting obstacles with a really high accuracy. The goal of these algorithms is to predict a list of bounding boxes from an input image. Machine Learning has evolved really well into localising and classifying obstacles in real-time in an image. However, none of these*

URL of post to be deleted (one per line) *

https://mc.ai/computer-vision-for-tracking/

Comment *

please state the reason for your request

☐ **I understand, that this form is for "requests for deletion" only. ***

Recaptcha

☐ I'm not a robot reCAPTCHA Privacy - Terms

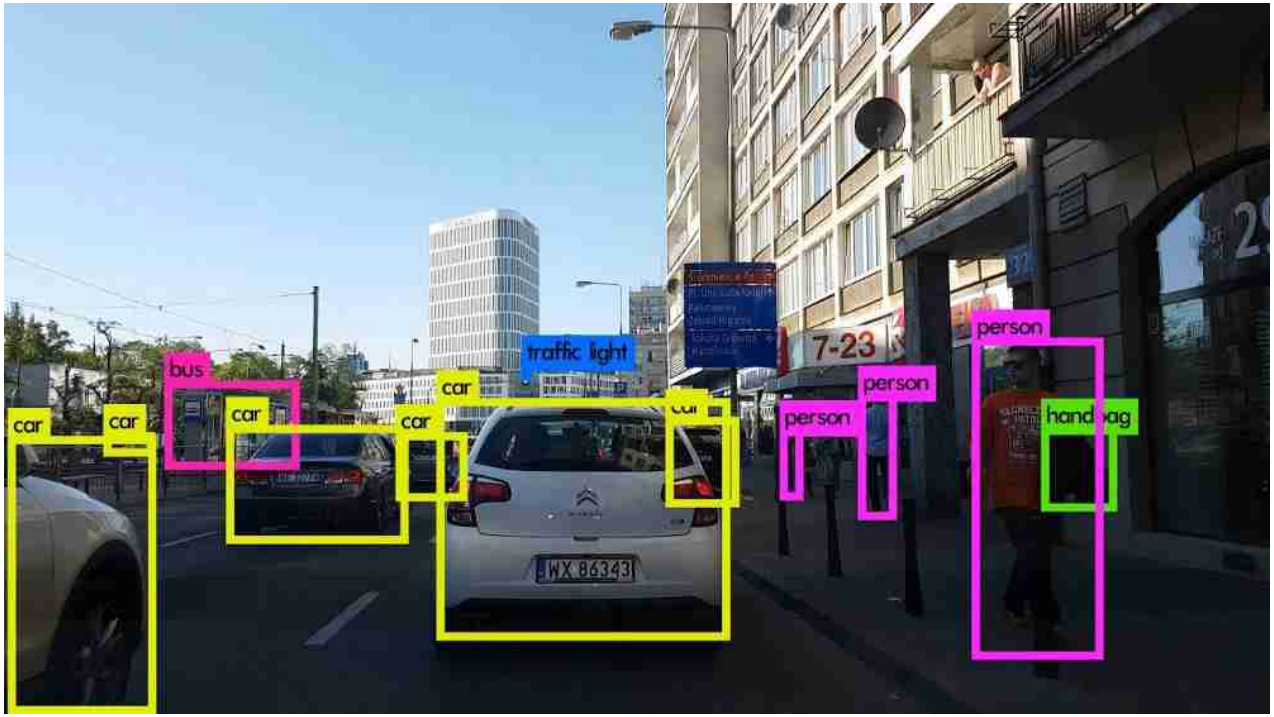
> *algorithm include the notion of time and continuity. When detecting an obstacle, these algorithms assume it's a new obstacle every time.*

SUBMIT

Here's one of the most popular object detection algorithm, called **YOLO (You Only Look Once)**.

Yolov3 demo

I won't go into the details of the algorithm here, but you can have a look at this video from Siraj Raval that explains it very well.



The output of the algorithm is a **list of bounding box, in format [class, x, y, w, h, confidence]**. The class is an id related to a number in a txt file (0 for car , 1 for pedestrian, …). x, y, w and h represent the parameters of the bounding box. x and y are the coordinates of the center while w and h are its size (width and height). The confidence is a number expressed in %.

How can we use these bounding boxes ?

So far, bounding boxes have been used to count the number of obstacles of the same class in a crowd, in self-driving cars, drones, surveillance cameras, autonomous robots, and all sorts of

systems using Computer Vision. However, they all share the same limit : same class obstacles are from the same color and cannot be set apart.

Deep SORT (Deep Simple Online Realtime Tracking)

So how could we define these bounding boxes as independant and how can we track them through time ?

Deep SORT demo

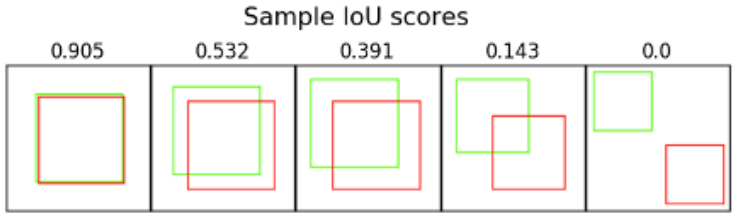
> *This implementation uses an object detection algorithm, such as YOLOv3 and a system to track obstacle. As you can see, it works with occlusion as well. How ? The reason is the use of* ***a Kalman Filter*** *and* ***The Hungarian Algorithm****.*

- 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.
- A Kalman Filter is an algorithm that can **predict future positions based on current position**. It can also **estimate current position better** than what the sensor is telling us. It will be used to have better association.

The Hungarian Algorithm (Kuhn-Munkres)

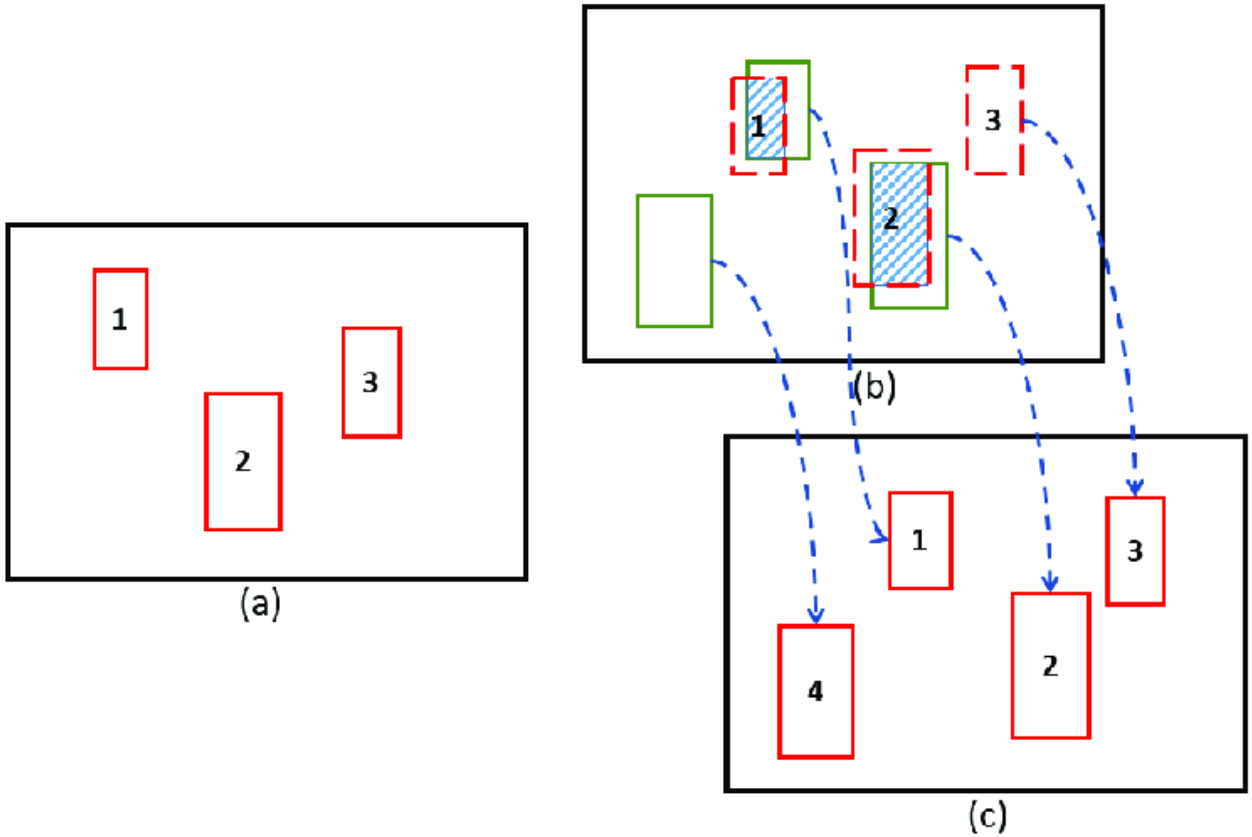
The hungarian algorithm, also known as Kuhn-Munkres algorithm, can associate an obstacle from one frame to another, based on a score. We have many scores we can think of :

- **IOU** (Intersection Over Union); meaning that if the bounding box is overlapping the previous one, it's probably the same.



- **Shape Score** ; if the shape or size didn't vary too much during two consecutives frames; the score increases.
- **Convolution Cost** ; we could run a CNN (Convolutional Neural Network) on the bounding box and compare this result with the one from a frame ago. If the convolutional features are the same, then it means the objects looks the same. If there is a partial occlusion, the convolutional features will stay partly the same and association will remain.

How to do the association ?



Association example

In this example, from frame a to frame b, we are tracking two obstacles (with id 1 and 2), adding one new detection (4) and keeping a track (3) in case it's a false negative.

The process for obstaining this is the following :

1. We have two lists of boxes from YOLO : a **tracking list (t-1)** and a **detection list (t)**.
2. Go through tracking and detection list, and calculate IOU, shape, convolutional score. **Store the scores in a matrix** …

Detection/Tracking	Tracking 1	Tracking 2	Tracking 3
Detection A	IOU = 0	IOU = 0	IOU = 0
Detection B	IOU = 0.56	IOU = 0	IOU = 0
Detection C	IOU = 0	IOU = 0.77	IOU = 0

Matrix of IOU scores

3. In some cases, we can have two matches for one bounding box. In this case, we **set the maximum IOU value to 1 and all the others to 0**.

Detection/Tracking	Tracking 1	Tracking 2	Tracking 3
Detection A	0	0	0
Detection B	1	0	0
Detection C	0	1	0

unmatched detection

unmatched tracking

Match Matrix

We have a matrix that tells us matching between Detection and Trackings. The next thing is to call a **sklearn function called *linear_assignment()*** that implements the Hungarian Algorithm. The algorithm minimizes the matrix of cost we give (it is made so it can match the lowest cost); since we want to match the highest score, we call it with a minus.

Here is the code to interprete this.

```
    matched_idx = linear_assignment(-IOU_mat)

    unmatched_trackers, unmatched_detections = [], []
    for t,trk in enumerate(trackers):
        if(t not in matched_idx[:,0]):
            unmatched_trackers.append(t)

    for d, det in enumerate(detections):
        if(d not in matched_idx[:,1]):
            unmatched_detections.append(d)

    matches = []

    # For creating trackers we consider any detection with an
    # overlap less than iou_thrd to signifiy the existence of
    # an untracked object

    for m in matched_idx:
        if(IOU_mat[m[0],m[1]]<iou_thrd):
            unmatched_trackers.append(m[0])
            unmatched_detections.append(m[1])
        else:
            matches.append(m.reshape(1,2))

    if(len(matches)==0):
        matches = np.empty((0,2),dtype=int)
    else:
        matches = np.concatenate(matches,axis=0)

    return matches, np.array(unmatched_detections), np.array(unmatched_trackers)
```

(source)

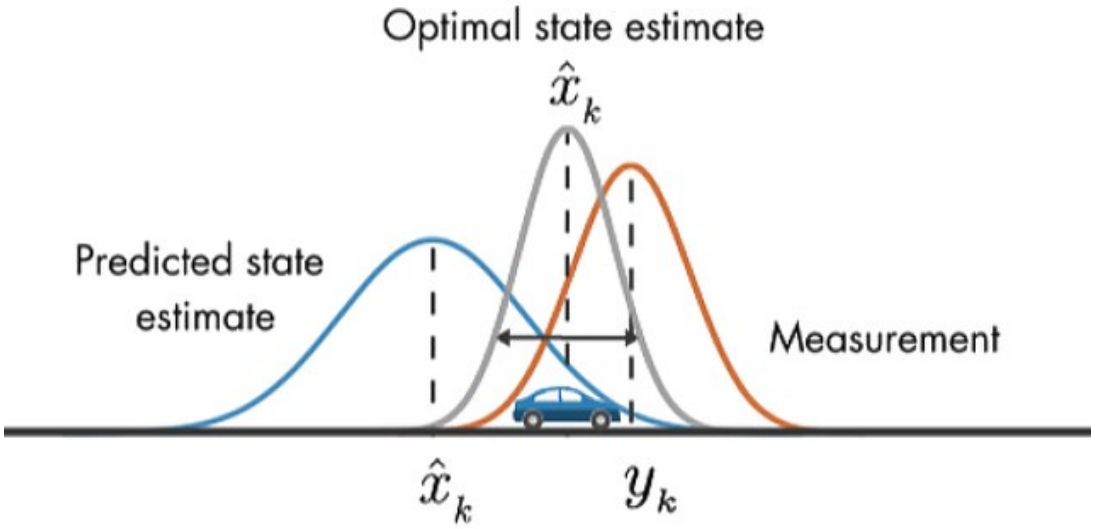
> *What we get from that is matrix of what element in detection matches what element in tracking.*

> *From that, we can output **matched detections**, unmatched detections and unmatched trackings.*

The Kalman Filter

Kalman Filters are very popular for tracking obstacles and predicting current and future positions. It is used in all sort of robots, drones, self-flying planes, self-driving cars, multi-sensor fusion, …

> *→ For an understanding on Kalman Filters logic, go check my Sensor Fusion article.*



Kalman Filter

A Kalman Filter is used on every bounding box, so it comes after a box has been matched. When the association is made, **predict** and **update** functions are called. These functions implement the

math of Kalman Filters composed of formulas for determining state mean and covariance.

State Mean and Covariance

Mean and Covariance are what we want to estimate. Mean is the coordinates of the bounding box, Covariance is our uncertainty on this bounding box having these coordinates.

Mean (x) is a state vector. It is composed by 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.

$$x = [\ cx \quad cy \quad w \quad h \quad vx \quad vy \quad vw \quad vh\ ]$$

Mean for a bounding box

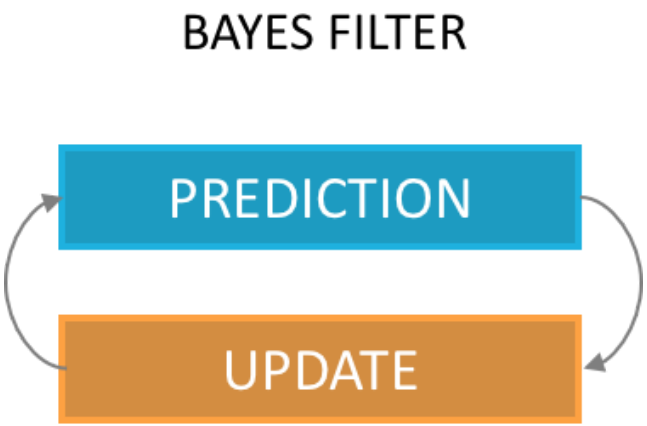
When we initialize this parameter, we set velocities to 0. They will then be estimated by the Kalman Filter.

Covariance (P) is our uncertainty matrix in the estimation. We will set it to an arbitrary number and tweak it to see results. A larger number means a larger uncertainty.

$$P = \begin{bmatrix} \mathbf{10} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{10} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{10} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{10} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{10} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{10} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{10} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{10} \end{bmatrix}$$

> *That's all we need to estimate : a state and an uncertainty ! There is two steps for a Kalman Filter to work : prediction and update. Prediction will predict future positions, update will correct them and enhance the way we predict by changing uncertainty. With time, a Kalman Filter gets better and better to converge.*

How to use ?



Kalman Filter cycle

> ***At time t=0, we have a measurement of 3 bounding boxes**. The Hungarian Algorithm defines them at 3 new detections. We therefore only have 3 detections in our system. **For each box, we inialize Kalman Matrices with coordinates of the bounding boxes.***

> ***At time t=1, we have 3 bounding boxes, of the same object.** The Hungarian Algorithm matches them with the 3 former boxes and we can start calling predict and update. We **predict the actual bounding boxes at time t from** the bounding boxes at **time t-1 a**nd then **update our prediction with the measurement at time t.***

Prediction

> *Prediction phase is matrix multiplication that will tell us* ***the position of our bounding box at time t based on its position at time t-1****.*

$$x' = Fx + u$$

$$P' = FPF^T + Q$$

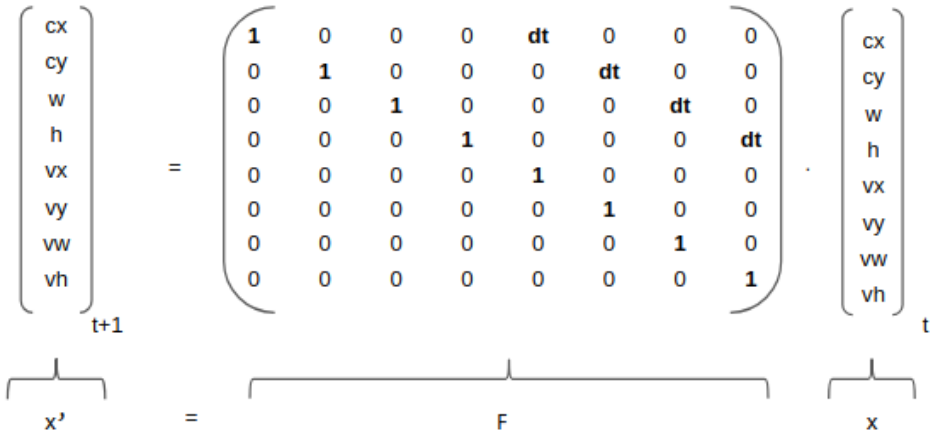
Prediction equations

So imagine we have a function called **predict()** in a class Kalman Filter that implements these maths. All we need to do is to have correct matrices F, Q and u. We will not use the **u vector** as it is used to **estimate external forces,** which we can't really do easily here.

How to set up the matrices correctly ?

State Transition : F

F is the core implementation of what we will define. What we put here is important because when we will multiply x by F, **we will change our x and have a new x, called x'**.



x Predict formula

As you can see, F [8×8] matrix contains a time value : **dt** is the difference between current frame and former frame timestamp. We will end up having **cx' = cx + dt*vx** for the first line, **cy' = cy + dt*vy,** and so on…

> *In this case, it means that we consider the velocity to be constant. We therefore set up F to implement the* ***Constant Velocity model.*** *There are a lot of models we can use depending on the problem we want to solve.*

Process Covariance Matrix : Q

Q [8×8] is our noise matrix. It is how much confidence we give in the system. Its definition is very important and can change a lot of things. Q will be added to our covariance and will then define our

global uncertainty. We can put very small values (0.01) and change it with time.

> *Based on these matrices, and our measurement, we can now make a prediction that will give us x' and P'. This can be used to predict future or actual positions. Having a matrix of confidence is useful for our filter that can model uncertainty of the world and of the algorithm to get better results.*

Update

> *Update phase is a correction step. It includes the new measurement (z) and helps improve our filter.*

$$y = z - Hx'$$

$$S = HP'H^T + R$$

$$K = P'H^T S^{-1}$$

$$x = x' + Ky$$

$$P = (I - KH)P'$$

Update equations

The process of **update** is to start by measuring an error between measurement (z) and **predicted mean**. The second step is to calculate a **Kalman Gain (K)**. The Kalman gain is used to estimate the

importance of our error. We use it as a multiplication factor in the final formula to estimate a new x.

Again, How to set up the matrices correctly ?

Measurement Vector: Z

z is the measurement at time t. We don't input velocities here as it is not measured, simply measured values.

$$Z = \begin{bmatrix} cx & cy & w & h \end{bmatrix}$$

Measurement Vector

Measurement Matrix : H

H [4×8] is our measurement matrix, it simply makes the math work between all or different matrices. We put the ones according to how we defined our state, and its dimension highly depends on how we define our state.

$$H = \begin{bmatrix} \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \end{bmatrix}$$

Measurement Matrix

Measurement Noise : R

R is our measurement noise; it's the noise from the sensor. For a LiDAR or RADAR, it's usually given by the constructor. Here, we need to define a noise for YOLO algorithm, in terms of pixels. It will be arbitrary, we can say that the noise in terme of the center is about 1 or 2 pixels while the noise in the width and height can be bigger, let's say 10 pixels.

$$R = \begin{matrix} \mathbf{1} & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 \\ 0 & 0 & \mathbf{10} & 0 \\ 0 & 0 & 0 & \mathbf{10} \end{matrix}$$

Measurement Noise Matrix

> *Matrix multiplications are made; and we have a prediction and update loop that gets us better results than the yolo algorithm. The filter can also be used to predict at time t+1 (prediction with no update) from time t. For that, it needs to be good enough and have a low uncertainty.*

Conclusion

> *We have now understand how to track an obstacle through time. From that, we could use Machine Learning to predict future behavior or trajectories, we could be able to estimate what an obstacle has been doing the last 10 seconds. This tool is powerful and tracking become not only possible, but also very accurate. Velocities can be estimated and a huge set of possibilites becomes available.* ***The general process is to detect obstacles using an object detection algorithm, match these bounding box with former bounding boxes we have using The Hungarian Algorithm; and then predict future bounding box positions or actual positions using Kalman Filters.***

> *Jeremy Cohen.*

→ Connect on LinkedIn and be sure to follow !

> Discover this article in French

References and Githubs