PEDESTRIAN DETECTION TUTORIAL WITH SIMPLE ONLINE REALTIME TRACKING (SORT)

TECHNICAL REPORT

Robert Azarcon
University of Illinois at Urbana-Champaign
Champaign, IL 61820
azarcon3@illinois.edu

Lohit Muralidharan
University of Illinois at Urbana-Champaign
Champaign, IL 61820
lohitm2@illinois.edu

May 7, 2024

Contents

ABSTRACT

This paper revolves around the implementation of the SORT (simple online realtime tracking) algorithm. First, our team will present our background research on SORT displaying the comparisons made compared to other MOT (multi object tracking) algorithms. Second, we will dive into the different components that builds up SORT from the very beginning. This portion includes performing a mathematical and theoretical analysis on the different parts of SORT which includes a method for object detection, optimal assignment, as well as filters. Third, our team will develop code and explain the implementation of how we designed each algorithm. We will examine how exactly we translated pure math into code as well as extra necessary components added on top of our current algorithms. Lastly, we will perform an analysis on how exactly our SORT algorithm performs which includes showcasing faults as well as the strengths.

1 Introduction

The Multi-Object-Tracking (MOT) problem in computer vision is an emerging area of research over the past few years and is highly applicable to fields such as autonomous vehicles, surveillance and security, as well as sports analytics. This paper aims to provide a primer to MOT through the lens of an implementation of the Simple Online Realtime Tracking (SORT) algorithm and applying it to a pedestrian tracking scenario. The reason SORT was chosen is due to it being a precursor to modern MOT algorithms and its simplicity. A lot of current state of the art MOT algorithms build on top of this current algorithm such as StrongSORT, DeepSORT, and ByteTrack. This paper will start by discussing the components of the SORT pipeline from a theoretical point of view to provide a fundamental understanding of the underlying algorithms that many modern MOT solutions are inspired from as well as the implementation and testing of our algorithm.

2 SORT Background

SORT was not the first object tracking algorithm to be created. There were previous iterations of object tracking algorithms that were also developed. However, when SORT was developed, it beat almost every other algorithm in terms of MOTA(↑). MOTA is also known as multi object tracking accuracy. The diagram to the right [1] shows how other state of the art algorithms at the time compared with SORT. SORT had the highest accuracy compared to other algorithms where TDAM was a close second. However in terms of Speed (Hz), we can see that SORT while maintaining its accuracy, performed under realtime. In fact, the Speed Axis is scaled logarithmic. However, we believe that given proper hardware such as GPU, we can attain real time detection if not close to real time detection.

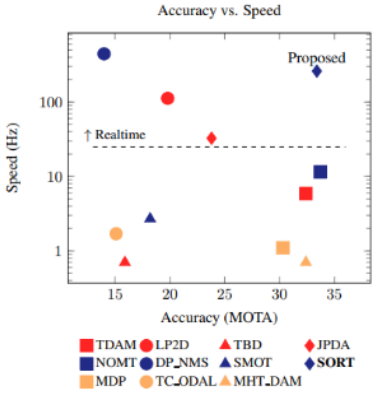

Figure 1: Mot Comparison

One other important thing we must note with object tracking algorithms is how detection algorithms play into this. In the original paper [1], it states that SORT is not used to help with improving the robustness of object detection as this can easily be solved by combining the ACF pedestrian detection with a version of CNN. They also note that in terms of object detection, they cannot make the algorithm niche towards pedestrians as CNN tends to generalize to multiple object classes.

There are three different parts to this algorithm mentioned within the paper. To build this algorithm, we require three components: an object detection model, a data association algorithm, and a method to filter measured and predicted results of a frame.

3 SORT Components

3.1 YOLOv1 Model

YOLOv1 follows a simple CNN object detection model that uses Resnet50. As complex as this model is, it still follows the fundamental neural network structure where it contains weights and biases that can be trained. These weights and biases can be trained using an optimizer and loss function. The loss function generates a loss after going through the forward pass of the neural network. Then, the weights and biases are altered during a backwards pass by the rules of the optimizer, which takes into account the loss that was generated before (the weights and biases are altered by a scaled version of the loss).

For weight updates, there is a choice between using either Adam optimizer or SGD optimizer. Adam optimizers tend to take a smoother path towards optimal min costs due to its characteristics of utilizing momentum. SGD optimizer takes a more choppy path toward optimal minimum.

In terms of the loss function, there are many characteristics to look at to compare different detections. For the YOLO model, we take a singular frame and split it into SxS amount of cells, and we perform a loss function on every cell within the SxS grid assuming we are predicting a box and we have a target box. There are 4 different parts to the YOLO Loss function shown below: regression loss, confidence loss, no object loss, and classification loss. Regression loss looks at the mean squared error between coordinate (x, y) and box size (widths and heights) for each cell (first and second double summations). No object loss looks at whether we are confident there is an object within a cell when there is not an object (fourth double summation). Classification loss looks at each cell and compares our classification predictions with a one hot vector representation (fifth double summation). In total, there are 20 different classes that our yolo model will use, meaning that there are 20 different class comparisons for our classification loss for every cell. Finally, confidence looks at given the IOU of a prediction and target box and compares it with our current confidence (third double summation). After the total loss, which is the sum of all of these losses, is computed and we run backpropagation over a certain amount of epochs, we have a trained Yolo Model which can predict overall bounding boxes around objects of the 20 classes.

$$
\begin{aligned}
&\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \\
&+ \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2 \\
&+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2
\end{aligned}
$$

Figure 2: YOLO Loss

3.2 Hungarian Algorithm

The Hungarian Algorithms is a more time efficient method for performing optimal assignment. In our case, let us assume we have 2 frames, each with 3 objects that were detected without bounding box coordinates. The main issue is the fact that we do not know which bounding box in both frames corresponds to the same object. Therefore if we were to place a unique identification on these boxes, we need to deploy some sort of algorithm that will pair bounding boxes across multiple frames to determine if they represent the same object or not. In order for us to determine optimal assignment, we need some sort of cost representation that will distinguish certain bounding boxes from others when choosing bounding boxes for ids. In the original research paper [1], it specifies that if we have n bounding boxes in both frames, then we set up a cost matrix that is n x n in which every cell in the matrix represents the IOU of specific box combinations. The naive way of approaching this problem is to go through every assignment possible and compute the total costs. Out of all the costs, we then choose the assignment that produces the minimal cost. However, the time complexity of this approach is O(n!) because if we are working with an n x n, we will have n assignments to choose

from, then n-1 assignments, and so on so forth (n * n-1 * n-2 ... 1 = n!). This brings us to the Hungarian Algorithms which can solve optimal assignment problems in O(n^3).

$$\begin{pmatrix} 15 & 20 & 35 \\ 25 & 10 & 15 \\ 5 & 15 & 30 \end{pmatrix} \tag{1}$$

Above is an example of an IOU cost matrix between bounding boxes across both frames on 3 objects. The first step is to find the minimum value across each row, and subtract the rows with that minimum value.

$$\begin{pmatrix} 0 & 5 & 20 \\ 15 & 0 & 5 \\ 0 & 10 & 25 \end{pmatrix} \tag{2}$$

The above matrix represents the new matrix after subtracting the min across each row. The next step is to subtract each column with the minimum value in each column. Below is the resulting matrix.

$$\begin{pmatrix} 0 & 5 & 15 \\ 15 & 0 & 0 \\ 0 & 10 & 20 \end{pmatrix} \tag{3}$$

Now after doing both subtractions, we need to find the minimum amount of horizontal or vertical lines across each row and column that will go through each zero. For our example, below shows a depiction of the minimum lines that will go through each zero.

$$\begin{pmatrix} 0 & 5 & 15 \\ 15 & 0 & 0 \\ 0 & 10 & 20 \end{pmatrix} \tag{4}$$

If the amount of horizontals and verticals equals the amount of rows and columns which is 3 in our case, we have found an optimal assignment for sure. Now in our case, we have an orange column and a red row meaning the amount of horizontals and verticals is 2 which is less than 3. If we run into this case, we need to perform one more step which is to find the smallest value that is not along any of our verticals or horizontals. In our case, we have coordinates (0, 1), (0, 2) (1, 2), and (2, 2) to choose from, and the coordinate that has the minimum value is (0, 1) with a corresponding minimum value of 5. We then subtract every uncovered value by 5 and add every intersection value by 5.

$$\begin{pmatrix} 0 & 0 & 10 \\ 20 & 0 & 0 \\ 0 & 5 & 15 \end{pmatrix} \tag{5}$$

After performing the addition and subtraction, we then re-calculate the verticals and horizontals. This calculation is shown below, and after this, we should be able to find an optimal assignment.

$$\begin{pmatrix} 0 & 0 & 10 \\ 20 & 0 & 0 \\ 0 & 5 & 15 \end{pmatrix} \tag{6}$$

Essentially to choose an optimal assignment, we choose the zeros in which every zero has its own row and column. The final optimal assignment is shown below.

$$\begin{pmatrix} 0 & 0 & 10 \\ 20 & 0 & 0 \\ 0 & 5 & 15 \end{pmatrix} \tag{7}$$

3.3 Kalman Filter

The Kalman Filter is a very necessary adjustment required for determining the final location of the bounding box. Note that the object detection model creates a bounding box and the Hungarian algorithms associate the bounding box with targets with specific ids. However, there is an issue in terms of the object detection model being slightly inaccurate. In order to account for this, we use a Kalman Filter. A Kalman Filter essentially takes measured values (in our case, the object bounding box parameters) and compares it with predicted bounding box values using a linear velocity model from the target bounding box in the previous frame. It then gives a new target value (in our case a new bounding box value that will represent the object within the screen). It is also important to note that we are assuming a linear velocity model meaning we need to use the bounding box representation and add more components to it. Within the original research paper [1], it states that the model representation they use to model the state of a target is represented by

$$X = [u \quad v \quad s \quad r \quad v_u \quad v_v \quad v_s] \tag{8}$$

where u and v represents the center of the bounding box, s and r represents the scale (area) and aspect ratio, and vu, vv, and vs represents the velocity of the center as well as the change in area over time. For our implementation, we will follow a more simplistic version meaning we will drop the aspect ratio and area of our targets and add additional features like bounding box width and height. Aspect Ratio and Area are not necessary for tracking objects.

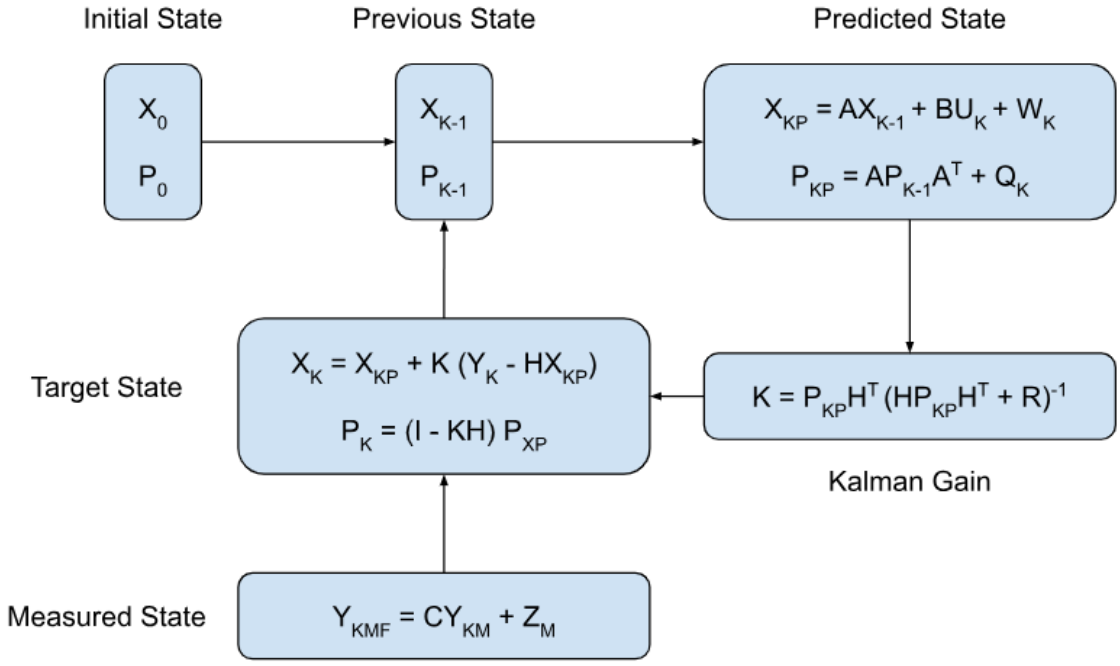

Figure 3: Kalman Filter

The Kalman Filter process is modeled above in the flow chart. From this chart, we can see that this process is iterative. Initially we start with an initial state X_0 and P_0. X_0 is modeled as:

$$X_0 = [x \quad y \quad v_x \quad v_y] \tag{9}$$

which are the initial center coordinate and the initial velocity of the center. In practice, we set the center velocity initially to 0, but will get updated in the next frames using simple kinematics. We also have a covariance matrix which in simple terms represents error or uncertainty in the state represented as P_0. We then transfer the initial state into the previous state ($X_0 \rightarrow X_{K-1}$, $P_0 \rightarrow P_{K-1}$). After we generate a previous state which is the same as the initial state, we then create a predicted state.

The predicted state uses simple kinematics in matrix form to generate the new X_{KP} and P_{KP}. Note that A can be modeled as a 4x4 matrix:

$$A = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{10}$$

If we matrix multiply A with X_{K-1}, we receive a new 4x1 matrix:

$$AX_{K-1} = [x + v_x\Delta t \quad y + v_y\Delta t \quad v_x \quad v_y] \tag{11}$$

Also note that B is a 4x2 matrix:

$$B = \begin{bmatrix} \frac{1}{2}\Delta t^2 & 0 \\ 0 & \frac{1}{2}\Delta t^2 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \tag{12}$$

and U_K is a 2x1 matrix representing:

$$U_k = \begin{bmatrix} a_x \\ a_y \end{bmatrix} \tag{13}$$

If both matrices are multiplied, we receive a new 4x1 matrix:

$$BU_k = \begin{bmatrix} \frac{1}{2}\Delta t^2 a_x \\ \frac{1}{2}\Delta t^2 a_y \\ a_x\Delta t \\ a_y\Delta t \end{bmatrix} \tag{14}$$

If we sum the 2 products AX_K-1 and BU_K, we receive the 4x1 matrix:

$$AX_{K-1} + BU_k = \begin{bmatrix} \frac{1}{2}\Delta t^2 a_x + x + v_x\Delta t \\ \frac{1}{2}\Delta t^2 a_y + y + v_y\Delta t \\ a_x\Delta t + v_x \\ a_y\Delta t + v_y \end{bmatrix} \tag{15}$$

These are textbook definitions of some fundamental kinematics equations. You may also notice that we add an additional term W_K in the flowchart which represents Gaussian Noise to best model the real world for X_{KP}. X_{KP} on a higher level uses kinematics to predict its future state in terms of position/velocity. P_{KP} also represents the predicted future state except for error and uncertainty. This is simply modeled as P_{KP} = AP_{K-1}AT + Q_K where Q_K also represents Gaussian Noise.

After creating a prediction for the target bounding box, we need to perform 2 operations: actually take measurements and to weight the comparisons on those measurements. This brings us to the Kalman Gain which actually does weight the comparisons on those measurements. Essentially, the formula for the Kalman Gain is modeled as K = P_{KP} H^T $(HP_{KP}H^T + R)^{-1}$. Note that we cannot divide matrices, but in matrix math, the inverse provides the same idea. In other textbooks, we may see it represented as well like HP_{KP} H^T $(HP_{KP}H^T + R)^{-1}$, which is more intuitive, but it is slower and provides the same results. Essentially, HP_{KP} H^T takes a chunk of the predicted covariance matrix and R represents the covariance matrix of the error in measured values. However, it is important to note for our implementation R takes into account the center coordinate and not velocity making it a 2x2 matrix:

$$R = \begin{bmatrix} \sigma_{xm}^2 & \sigma_{xm}\sigma_{ym} \\ \sigma_{ym}\sigma_{xm} & \sigma_{ym}^2 \end{bmatrix} \tag{16}$$

where the "m" represents the measured subscript. Therefore, HP_{KP} H^T needs to be a 2x2 matrix which also only looks at the centers. H in our situation is a 2x4 identity matrix:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{17}$$

If we were to compute HP_{KP} H^T using the previous calculation of P_{KP}, we would receive a 2x2 matrix:

$$HP_{KP}H^T = \begin{bmatrix} \sigma_{xp}^2 & \sigma_{xp}\sigma_{yp} \\ \sigma_{yp}\sigma_{xp} & \sigma_{yp}^2 \end{bmatrix} \tag{18}$$

where the "p" represents the predicted subscript. On a higher level, the Kalman Gain, compares the uncertainty between the prediction and the measured values which is a value between 0 and 1. If the Kalman gain is higher, then we are certain that the measured values are indeed accurate. If the Kalman Gain is lower, we are certain that the predicted values are more accurate.

After computing the Kalman Gain, we need to compute the Measured State. The Measured State can be measured by external sources such as cameras and sensors. In our case, our object detection model will perform the measurements. Note that for the measured state, we will only be working with positional values rather than both position and velocity. In the flowchart above, we depict that the measured state can be calculated as the following: Y_{KMF} = CY_{KM} + Z_m where Y_{KM} was the measured value which is a 4x1 matrix:

$$Y_{KM} = \begin{bmatrix} x_m \\ y_m \\ v_{xm} \\ v_{ym} \end{bmatrix} \tag{19}$$

In order to extract the center position, C must be the same exact matrix as H described above:

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{20}$$

Multiplying C and Y_{KM} will give us a new matrix:

$$CY_{KM} = \begin{bmatrix} x_m \\ y_m \end{bmatrix} \tag{21}$$

We can apply Gaussian noise to make the measurements more realistic which is depicted as Z_m. Up to this point, we have the Kalman Gain, Predicted State, and Measured State.

The final step is simply to calculate the final target state which represents the actual target prediction. The target covariance matrix can be calculated as follows: P_K = (1 - KH) P_{XP}. The target position/velocity vector can also be calculated as follows: X_K = X_{KP} + K(Y_K - HX_{KP}). This makes sense because the Kalman gain will take the scale of the difference between predicted and measured value appropriately such that either will perform a higher effect on X_K. This concludes 1 iteration of the Kalman Filter, but for our implementation, we need to run this filter for every frame to improve accuracy of our target bounding box.

4 SORT Code Implementation

Code Link: `https://github.com/Codaero/cs444-mot/tree/master`

4.1 Object Track Structure

For the SORT tracker, our team had to decide what information is necessary for every track and how exactly do we store each track. The code block below displays exactly what information is stored for each track.

```
'''
Object Attributes:
- id : int
- center : tuple(float, float)
- height : float
- width : float
- velocity: tuple(float, float)
- class : string
- prob : float
- last_frame_seen: int
- active_track: bool
- cov: array <= <dim : (4,4)>
'''
```

Note that each track is a python dictionary with these attributes as string parameters. "id" is the identification number of the track to distinguish between tracks. "center", "height", and "width" represents the center coordinate, height, and width of the bounding box associated with the track in a certain frame. The "velocity" parameter represents how fast and in what direction is the center coordinate traveling. Note that this is a 2d tuple just like "center". The "class" parameter represents what class the track is tracking, and the "prob" parameter represents the confidence score of the class. The last 3 parameters are "last_frame_seen", "active_track", and "cov". The "last_frame_seen" represents the frame number in which an object within a track was detected last. The "active_track" parameter shows whether a certain track is currently active or inactive. Lastly, the "cov" parameter represents the covariance matrix associated with a track specifically for the Kalman Filter. It is also important to note that we store each track within a python list called self.object_track which is created within our tracker class.

4.2 Tracker Implementation

For our implementation of the SORT tracker, our team decided it would be best if we implemented a class for the Tracker. Below is our constructor related to this class which sets a bunch of parameters and constants which will be used throughout the code base. This tracker is instantiated within our test script which takes a video and extracts multiple frames to be processed in our tracker. It is important to also note that all 3 components (Yolo Object Detector, Hungarian Algorithms, and Kalman Filter) are existing within this tracker class and will be used within process_frame.

```
def __init__(self, net, output_folder, frame_dim):
    self.frame_dim = frame_dim
    self.frame_count = 0
    self.frame_list = []
    self.net = net
    self.object_track = []
    self.output_folder = output_folder
    self.T_lost = 5
    self.IOU_min = 0.1
    self.start_assign = False
    self.kalman_filter = KalmanFilter()
```

process_frame is the most important function within our tracker class because it acts as a top level function that utilizes every other function and class. On a higher level, process_frame takes in a frame and annotates bounding boxes on it. However, there are a series of steps on how we actually process the frames. In lines 2-26, we start off by taking the frame and running it through self.net which is a YOLO Model that we passed into our tracker. We then take the results and remove predictions with a bounding box area of 0 because there are occasions where boxes with 0 width and 0 height are within the results with a random class and confidence score of 0. Once we filter these objects out, we convert these objects into a more usable format by using self.create_objects which is a function that is described later within this passage. Lines 30-34 then looks at whether we start to see objects or not. Essentially, there is a chance that there is

a certain amount of frames that will pass before we start to see our first objects. For improving runtime, we will not run the Hungarian algorithm and the Kalman filter until we start to see objects. Lines 37-99 executes if we do start to see objects entering the frame. In our case we are tracking pedestrians meaning this code will execute when we start seeing pedestrians enter the canvas. In these lines of code, we run the predict and update from the Kalman filter as well as generate a measurement using the Hungarian algorithm and YOLO result. Specifically, lines 41- 63 finds all of the objects that currently have an active track, an extracts the center, velocity, and covariance matrix of those objects. We then run those objects through predict to generate a prediction of where exactly those objects will be in the next frame. Lines 66-71 takes our results from the YOLO model and runs the hungarian algorithms on it to assign the objects detected to active tracks. Objects that are not associated with any track will then be added under a new track with a new id. Finally, lines 72-99, runs the update step where we take our predictions and our measurements to generate a target data, and update all of the tracks with this target bounding box data. Lines 102 then checks whether certain objects left the canvas, and if there are, we disable the tracks associated with these objects. Finally, lines 105-112 annotates the frames with our actively running tracks, appends it to self.frame_list and then returns the annotated frame.

```
def process_frame(self, frame):
    frame_filename = os.path.join(
        self.output_folder,
        f"frame_{self.frame_count:04d}.jpg"
    )
    cv2.imwrite(frame_filename, frame)

    # TASK: GENERATE RESULTS FROM YOLO MODEL
    result = self.net(frame)
    result = result.xyxy[0]

    # TASK: REMOVE BOUNDING BOXES RESULTING FROM ZERO AREA
    rm_idx = []
    for obj_idx, obj in enumerate(result):
        width = abs(obj[0] - obj[2])
        height = abs(obj[1] - obj[3])
        area = width * height
        if area == 0 or self.net.names[int(obj[5])] != 'person':
            rm_idx.append(obj_idx)

    # TASK: TRANSFORM OBJECT DATA WITH CENTER COORDINATE
    upd_result = [
        obj for idx, obj in enumerate(result)
        if idx not in rm_idx
    ]
    objects = self.create_objects(upd_result)

    # TASK: GET THE SET OF OBJECT TRACKS WE WANT
    # TO DISPLAY VIA HUNGARIAN ALGORITHM
    if not self.start_assign:
        # assign unique ids to every object
        self.object_track = objects
        if len(self.object_track) > 0:
            self.start_assign = True

    else:
        # check previous frame for object assignment
        print(self.object_track)
        print('\n\n\n')
        # get all predictions
        kalman_x, kalman_p, kalman_ids = [], [], []
        for obj in self.object_track:
            if obj['active_track']:
                x = obj['center'][0]
                y = obj['center'][1]
                vx = obj['velocity'][0]
                vy = obj['velocity'][1]
                kalman_x.append([x, y, vx, vy])
                kalman_ids.append(obj['id'])
                kalman_p.append(obj['cov'])
```

```
        kalman_x = torch.tensor(kalman_x).cpu().numpy()
        kalman_p = torch.tensor(kalman_p).cpu().numpy()

        pred_x, pred_p = [], []
        for idx in range(len(kalman_x)):
            x, p = self.kalman_filter.predict(
                kalman_x[idx],
                kalman_p[idx]
            )
            pred_x.append(x)
            pred_p.append(p)
        pred_x, pred_p = np.array(pred_x), np.array(pred_p)

        # get all measurements
        measured = object_assign(
            objects,
            self.object_track,
            self.frame_count,
            self.IOU_min
        )
        kalman_m = []
        for obj in measured:
            if obj['active_track'] and obj['id'] in kalman_ids:
                m_x = obj['center'][0]
                m_y = obj['center'][1]
                m_vx = obj['velocity'][0]
                m_vy = obj['velocity'][0]
                kalman_m.append([m_x, m_y, m_vx, m_vy])
        kalman_m = torch.tensor(kalman_m).cpu().numpy()

        self.object_track = measured

        # get all targets and save them
        for idx in range(len(kalman_m)):
            targ_x, targ_p = self.kalman_filter.update(
                kalman_m[idx],
                pred_x[idx],
                pred_p[idx]
            )
            self.object_track[kalman_ids[idx]]['cov'] = targ_p
            self.object_track[kalman_ids[idx]]['center'] = (
                targ_x[0],
                targ_x[1]
            )
            self.object_track[kalman_ids[idx]]['velocity'] = (
                targ_x[2],
                targ_x[3]
            )

    # TASK: BOUND CHECK FOR OBJECTS ENTERING AND LEAVING THE FRAME
    self.bound_check()

    # TASK: DISPLAY TRACKS
    frame = self.edit_frame(frame)

    cv2.imwrite(frame_filename, frame)

    self.frame_count += 1
    self.frame_list.append(frame)

    return frame
```

bound_check takes care of objects that leave the canvas. According to the original paper [1], if an object left the frame for T_lost amount of seconds, then if it were to reappear, it would appear under a new identity. Essentially, we would have to disable the track related to the object that leaves the frame by setting active_track to false. In the paper, it also mentions that in order for this to work, T_lost would need to be set to 1. The reason why we need to disable tracks is to prevent tracks from showing on the canvas when the object corresponding to the track is nonexistent.

```
def bound_check(self):
    for obj_idx, obj in enumerate(self.object_track):
        center = obj['center']
        width = obj['width']
        height = obj['height']
        velocity = obj['velocity']
        if self.frame_count - obj['last_frame_seen'] > self.T_lost:
            # Get corners of bounding box
            tp_lt = (int(center[0] - width / 2), int(center[1] - height / 2))
            bt_rt = (int(center[0] + width / 2), int(center[1] + height / 2))
            if (tp_lt[0] <= 5 and velocity[0] < 0) or (tp_lt[1] <= 5 and velocity
    [1] < 0) or (bt_rt[0] >= 635 and velocity[0] > 0) or (bt_rt[1] >= 355 and
    velocity[1] > 0):
                self.object_track[obj_idx]['active_track'] = False
```

edit_frame uses the tracks present in self.object_track to draw on the current frame. To determine if a track is present, we would just need to check the 'active_track' parameter for every object. The objects that have this parameter set to true will be drawn on the canvas of the frame that is passed into this function. Note that process_frame utilizes this function by passing in the current frame and recieving annotated frame. This annotated frame is then appended into self.frame_list.

```
def edit_frame(self, frame):
    for obj in self.object_track:
        center = obj['center']
        width = obj['width']
        height = obj['height']
        class_name = obj['class']
        id_val = obj['id']

        if obj['active_track'] and obj['last_frame_seen'] == self.frame_count:

            left_up = (int(center[0] - width / 2), int(center[1] - height / 2))
            right_bottom = (int(center[0] + width / 2), int(center[1] + height /
    2))
            color = COLORS[VOC_CLASSES.index(class_name)]
            cv2.rectangle(frame, left_up, right_bottom, color, 2)
            label = str(id_val)
            text_size, baseline = cv2.getTextSize(label, cv2.FONT_HERSHEY_SIMPLEX,
     0.4, 1)
            p1 = (left_up[0], left_up[1] - text_size[1])
            cv2.rectangle(frame, (p1[0] - 2 // 2, p1[1] - 2 - baseline), (p1[0] +
    text_size[0], p1[1] + text_size[1]), color, -1)
            cv2.putText(frame, label, (p1[0], p1[1] + baseline), cv2.
    FONT_HERSHEY_SIMPLEX, 0.4, (255, 255, 255), 1, 8)

    return frame
```

create_objects in the tracker class is used in process_frame to take the results of running a frame through the Yolo model and transform the bounding box data into a more organized fashion using a python dictionary. Note that these transformed objects are used throughout the rest of process_frame in the hungarian algorithms and the kalman filter.

```
def create_objects(self, result):

    object_transform = []

    if len(result) == 0:
        return []

    if not self.start_assign:
```

```
        obj_idx = 0
    else:
        obj_idx = self.object_track[-1]['id'] + 1

    for obj in result:

        center = ((obj[0] + obj[2]) / 2, (obj[1] + obj[3]) / 2)
        height = abs(obj[1] - obj[3])
        width = abs(obj[0] - obj[2])

        new_object = dict()
        new_object['id'] = obj_idx
        new_object['center'] = center
        new_object['height'] = height
        new_object['width'] = width
        new_object['velocity'] = (0, 0)
        new_object['prob'] = obj[4]
        new_object['class'] = self.net.names[int(obj[5])]
        new_object['last_frame_seen'] = self.frame_count
        new_object['active_track'] = True
        new_object['cov'] = np.identity(4)
        obj_idx += 1

        object_transform.append(new_object)

    return object_transform
```

get_frames in the last function in the tracker class. This function has a single purpose which is to return all of the frames that have been edited with bounding boxes. The is used after we completely mark a series of frames and we want to combine them into an mp4. Essentially we would use libraries and packages such as OpenCV VideoWriter to combine all of the frames.

```
def get_frames(self):
    return self.frame_list
```

4.3 YOLO Model Implementation

As mentioned within the different components of SORT, we explained that the object detector was necessary when determining the measured values for the Kalman Filter. We started off by developing our own Yolov1 model which was successful. However, when our team unit tested the Yolov1 model, we ran into some faults. While the Yolov1 model did detect pedestrians and drew bounding boxes around them, there would be occasional bounding boxes in random areas of the canvas. We then decided that it would be better if we upgraded our model to a pretrained Yolov5 which is 5 iterations of improvement from the base model. This required us to utilize open source code from an organization called ultralytics. Below is the instantiation of the Yolov5 model. It is important to note that Yolov5 does use loss functions described in the math implementation above as well as a resnet model.

```
net = torch.hub.load('ultralytics/yolov5', 'yolov5s')
```

One important key feature we can perform with the net in line 2 would be to run images through net by calling results = net(image). Results is a data structure that contains the results of passing the image through the Yolo model. One important feature of this data structure is results.xyxy[0] which will contain center, width, height, class, and confidence. Note that this is also the net that we pass into our SORT tracker class.

4.4 Hungarian Algorithm Implementation

object_assign is the main code block that performs the Hungarian Algorithm. This function also performs preprocessing and post processing of data. The functions essentially starts by performing 2 checks. In lines 7 and 8, we check if there are no objects detected in the current frame. If no objects were detected, we simply return the previous objects because we could not perform any assignments and update any track information. In lines 12-24, we check a second case where there were no objects in the previous frame that have an active track. This means there are new objects that are emerging into the canvas. If this is the case, we need to return a new set of tracks that correspond to the new objects that entered the frame with a new set of ids.

If both of these checks fail to occur, we start the actual Hungarian algorithm. The first step to this algorithm corresponds to lines 26-55. Essentially these lines creates what is called a cost matrix. For our cost matrix, every current object represents every row and every previous object represents every column. Every value in this matrix consists of 1 - IOU (intersection over union) of the current and previous object corresponding to a specific cell. The reason why we do 1 - IOU is because we are trying to maximize the sum of IOU of our final assignment. In other words, we are trying to minimize the sum of 1 - IOU of all of our assignments. To calculate the IOU we created a separate function called calculate_IOU which is described later. Lines 41 and 42 stash the row and column amount before we do more processing for our cost matrix. Lines 44 and 45 handle a special case which represents an object with a 1 - IOU of 1 for every object. This means that a certain object has no association with any other corresponding object within its own row meaning its an entirely new object that has been found. We then add additional rows and columns of 0s to convert our matrix into a square matrix just in case we have a rectangular cost matrix.

After the creation of the cost matrix, the first and second step is to subtract every row with the minimal value of the row and to subtract every column with the minimal value of the column. This is depicted in lines 58-64.

Lines 71-107 represents the last step of the Hungarian algorithms before post processing. Our team designed a function named get_strides which gets us an sub-optimal if not optimal assignment as well as the row and column strides associated with this set of assignments. As mentioned in the theory section of the Hungarian algorithms, if the sum of the amount of row and column strides is less than the length of the square matrix, we have to perform extra steps to modify our cost matrix. The first step is to find the minimum value that is not covered by a row or column stride. This is what lines 80-89 attempt to find. Once you find the minimum value, you must subtract every other value that is not covered by a row or column stride by this minimum value and add every intersection between row and column stride by this minimum value. The subtraction takes place in lines 92-99 and the addition takes place in lines 102-105. We repeat this entire process from lines 71-105 until the amount of row and column strides is equivalent to the length of our square matrix. Once we accomplish this check, we have received our optimal assignment. However there is one final issue. Our initial cost matrix before adding extra rows and columns of 0s could have been unbalanced. This means the initial cost matrix could have been a rectangular matrix. If this is the case, we have assignments of objects that do not exist. Line 107 performs the filtration of non-existing objects in our optimal assignments.

The final step of object_assign is to simply perform some post processing of our assignments. Our goal of object_assign is to assign new objects to previous track, or create new tracks for new objects. In lines 115-144, we traverse every assignment and if any assignment is within curr_unidentified_ids or prev_unidentified_ids which was instantiated in lines 44 and 45, we create new tracks out of these objects. Otherwise, in lines 136-144, we update the current tracks with the current object data that was assigned to it. Line 146-166 handles another case where we could have more new objects appearing on the canvas compared to previous objects on the canvas. If this is the case, we create new tracks out of these objects that are new. Lines 169-184 combines all of these new tracks created with the updates tracks and the non active tracks and returns this new set of tracks. The reason why we keep non active tracks is to create an easy method of assigning new ids in the future.

```
1  def object_assign(curr_objs, prev_objs, frame_num, IOU_min):
2      '''
3      curr_objs - [[center, height, width, cls, prob]]
4      prev_objs - [[ ids, center, height, width, cls, prob, last_frame_seen,
       active_track ], ...]
5      '''
6
7      if len(curr_objs) == 0:
8          return prev_objs
9
10     # STEP 1: CREATE COST MATRIX
11
12     prev_exist_objs = [obj for obj in prev_objs if obj['active_track']]
13
14     if len(prev_exist_objs) == 0:
15         hidden_return = copy.deepcopy(prev_objs)
16         last_id = prev_objs[-1]['id'] + 1
17
18         for obj in curr_objs:
19             new_obj = copy.deepcopy(obj)
20             new_obj['id'] = last_id
21             new_obj['last_frame_seen'] = frame_num
22             hidden_return.append(new_obj)
23             last_id += 1
```

```
            return hidden_return

    cost_matrix = np.zeros((len(curr_objs), len(prev_exist_objs)))


    for i, c_obj in enumerate(curr_objs):

        for j, p_obj in enumerate(prev_exist_objs):

            iou = calculate_IOU(c_obj, p_obj)

            if iou <= IOU_min:

                iou = 0

            cost_matrix[i][j] = 1 - iou
    # STEP 2: PERFORM ASSIGNMENT
    row_amt = cost_matrix.shape[0]
    col_amt = cost_matrix.shape[1]

    curr_unidentified_ids = np.where((cost_matrix == 1).all(axis=1))[0]
    prev_unidentified_ids = np.where((cost_matrix == 1).all(axis=0))[0]

    # reprocess cost matrix

    if row_amt > col_amt:
        extra_zeros = np.zeros((row_amt - col_amt, row_amt))
        cost_matrix = np.concatenate((cost_matrix, extra_zeros.T), axis=1)

    elif col_amt > row_amt:
        extra_zeros = np.zeros((col_amt - row_amt, col_amt))
        cost_matrix = np.concatenate((cost_matrix, extra_zeros), axis=0)


    # row subtract
    min_row_values = np.min(cost_matrix, axis=1)
    cost_matrix = (cost_matrix.T - min_row_values).T

    # column subtract
    min_col_values = np.min(cost_matrix, axis=0)
    cost_matrix = cost_matrix - min_col_values

    dim = cost_matrix.shape[0]

    row_strides = []
    col_strides = []

    while len(row_strides) + len(col_strides) < dim:

        assignment, row_strides, col_strides = get_strides(cost_matrix)

        min_value = float('inf')

        if len(row_strides) + len(col_strides) < dim:

            # get minimum value in unmarked values
            for row_num, row in enumerate(cost_matrix):

                if row_num not in row_strides:

                    for col_num, val in enumerate(row):

                        if col_num not in col_strides:

                            if min_value > cost_matrix[row_num][col_num]:
```

```
                                min_value = cost_matrix[row_num][col_num]

            # subtract minimum value to every unmarked values
            for row_num, row in enumerate(cost_matrix):

                if row_num not in row_strides:

                    for col_num, val in enumerate(row):

                        if col_num not in col_strides:
                            cost_matrix[row_num][col_num] -= min_value

            # add minimum value to intersections
            for row_num in row_strides:

                for col_num in col_strides:
                    cost_matrix[row_num][col_num] += min_value

    assignment = filter_detections(assignment, row_amt, col_amt)
    row_idx, col_idx = assignment[:, 0], assignment[:, 1]

    # STEP 3: PARSE ASSIGNMENT

    new_objs = copy.deepcopy(prev_exist_objs)
    next_new_id, curr_ids = prev_objs[-1]['id'] + 1, len(curr_objs)

    for row, col in zip(row_idx, col_idx):

        if row in curr_unidentified_ids or col in prev_unidentified_ids:
            obj = curr_objs[row]

            unidentified_object = dict()
            unidentified_object['id'] = next_new_id
            unidentified_object['center'] = obj['center']
            unidentified_object['height'] = obj['height']
            unidentified_object['width'] = obj['width']
            unidentified_object['class'] = obj['class']
            unidentified_object['prob'] = obj['prob']
            unidentified_object['last_frame_seen'] = frame_num
            unidentified_object['velocity'] = (0, 0)
            unidentified_object['active_track'] = True
            unidentified_object['cov'] = np.identity(4)

            next_new_id += 1
            new_objs.append(unidentified_object)

        else:
            new_objs[col]['height'] = curr_objs[row]['height']
            new_objs[col]['width'] = curr_objs[row]['width']
            new_objs[col]['class'] = curr_objs[row]['class']
            new_objs[col]['prob'] = curr_objs[row]['prob']
            new_objs[col]['cov'] = curr_objs[row]['cov']
            new_objs[col]['active_track'] = True
            new_objs[col]['velocity'] = curr_objs[row]['velocity']
            new_objs[col]['last_frame_seen'] = frame_num
            new_objs[col]['center'] = curr_objs[row]['center']

    if len(curr_objs) > len(prev_exist_objs):

        unidentified_ids = [x for x in range(curr_ids) if x not in row_idx]

        for id_val in unidentified_ids:
            obj = curr_objs[id_val]

            unidentified_object = dict()
```

```
            unidentified_object['id'] = next_new_id
            unidentified_object['center'] = obj['center']
            unidentified_object['height'] = obj['height']
            unidentified_object['width'] = obj['width']
            unidentified_object['class'] = obj['class']
            unidentified_object['prob'] = obj['prob']
            unidentified_object['last_frame_seen'] = frame_num
            unidentified_object['velocity'] = (0, 0)
            unidentified_object['active_track'] = True
            unidentified_object['cov'] = np.identity(4)

            new_objs.append(unidentified_object)

            next_new_id += 1

    final_objs, new_objs_cnt = [], 0

    for obj in prev_objs:

        if obj['active_track']:
            final_objs.append(new_objs[new_objs_cnt])
            new_objs_cnt += 1

        else:
            final_objs.append(obj)

    while new_objs_cnt < len(new_objs):
        final_objs.append(new_objs[new_objs_cnt])
        new_objs_cnt += 1

    return final_objs
```

get_strides attempts to get all of the strides. The strides can be row strides or column strides as discussed in the theoretical section of the Hungarian algorithms. This function also returns the assignment, which could be non-optimal, that corresponds to the row and column strides. In line 3, "assignment" will contain all of the assignment that corresponds to 0s in the modified cost matrix. In lines 5-7, all_rows contains a set of all row numbers, zero_rows contains a set of all row numbers that contain a 0 in the assignment, and unmarked rows contains a set of all of the row numbers that are not in zero_rows. Lines 15-25 essentially looks at every single row in unmarked_rows and finds if there are 0s in those rows. If there are, we get the column number and mark the column by appending it to another list called marked_cols. Now Lines 27-35 checks if there are any rows in the assignment which is already marked by marked_cols. If there are, you do not want to make a row for it or else you will be creating a row stride and column stride that attempts to cover the same exact 0. Therefore, you want to move that row number into unmarked_rows. You repeat both 15-25 and 27-35 until there are no more rows to add to unmarked_rows and marked_cols. Line 37 then gets the marked_rows that capture 0s that the marked_cols did not capture by finding all of the rows in all_rows that are not in unmarked_rows.

```
def get_strides(cost_matrix):

    assignment = search_optimum(copy.deepcopy(cost_matrix))

    all_rows = set(range(cost_matrix.shape[0]))
    zero_rows = set(assignment[:,1])
    unmarked_rows = list(all_rows - zero_rows)

    marked_cols, cond = [], True

    while cond:

        cond = False

        for row_num in unmarked_rows:

            zero_col = np.where(cost_matrix[row_num] == 0)[0]

```

```
            for col_num in zero_col:

                if col_num not in marked_cols:

                    marked_cols.append(col_num)

                    cond = True

        for coord in assignment:

            row_num, col_num = coord

            if row_num not in unmarked_rows and col_num in marked_cols:

                unmarked_rows.append(row_num)

                cond = True

    marked_rows = list(all_rows - set(unmarked_rows))

    return assignment, marked_rows, marked_cols
```

search_optimum was designed for the purpose of finding an assignment for a separate function named get_strides. Essentially, Line 4 converts our cost matrix into a boolean matrix where 0 represents True and any other number represents False. We then enter a while loop in Line 6 which will never break until the entire cost matrix is filled with False. Lines 8-16 goes through every row of the boolean matrix, and finds the row with at least 1 zero, but the least amount of zeros in comparison to every other row. This information is stored into a 2d array called solution which is created in line 8. The first element of solution represents the amount of zeros in that row, and the second element represents the actual row number. Lines 18 and 19 then attempts to get the column and row index of the first zero in the row with the least amount of zeros but at least one zero. We then append this coordinate into an array created in Line 2 called assignment. Since this coordinate is now a part of our assignment, we cannot create an assignment within the same row or column of this coordinate. Therefore in lines 23 and 24, we essentially set the entire row and column of our new assignment to False which helps attempt to break our while loop in line 6. On line 26, we return our array of our sub-optimal if not optimal assignments.

```
def search_optimum(cost_matrix):
    assignment = []

    bool_matrix = (cost_matrix == 0)

    while np.count_nonzero(bool_matrix == True):

        solution = [float('inf'), -1]

        for row_num, row in enumerate(bool_matrix):

            amount_zeros = np.count_nonzero(row == True)

            if amount_zeros > 0 and solution[0] > amount_zeros:
                solution[0] = amount_zeros
                solution[1] = row_num

        col_idx = np.where(bool_matrix[solution[1]] == True)[0][0]
        row_idx = solution[1]

        assignment.append([row_idx, col_idx])

        bool_matrix[:, col_idx] = False
        bool_matrix[row_idx, :] = False

    return np.array(assignment)
```

filter_detections is the final part of the actual Hungarian algorithm. Essentially, the assignment we received from performing the Hungarian Algorithms contains optimal assignment between the objects detected on the current frame

and the objects detected on the previous frame. However, there is a possibility where the amount of objects detected in the current frame and the previous frame are not the same which is why we add additional rows and columns of zeros. This causes extra non-existent assignments which will be removed by this specific function. row_amt and col_amt are the row and column amounts of the cost matrix before adding the additional rows and columns of 0s. In lines 4-11, we check if the initial amount of rows is greater than the initial amount of columns, and if so, we remove the assignment corresponding to the extra columns. If the initial amount of rows is less than the initial amount of columns, we perform the vice versa operation where we remove the assignments corresponding to the addition rows of 0s. If the initial amount of rows and columns are the same, we did not add any additional rows and columns of 0s meaning our current set of assignments do not require modification. We then return this new set of assignments in Line 25.

```
1  def filter_detections(assignment, row_amt, col_amt):
2      new_assignment = []
3
4      if row_amt > col_amt:  # unbalanced case 1
5
6          col_indexes = set(range(col_amt))
7
8          for coord in assignment:
9
10             if coord[1] in col_indexes:
11                 new_assignment.append(coord)
12
13     elif col_amt > row_amt:  # unbalanced case 2
14
15         row_indexes = set(range(row_amt))
16
17         for coord in assignment:
18
19             if coord[0] in row_indexes:
20                 new_assignment.append(coord)
21
22     else:
23         new_assignment = assignment
24
25     return np.array(new_assignment)
```

calculate_IOU essentially calculates the intersection over union between 2 different boxes. This is extremely important in the implementation of our design because our cost matrix in object_assign are essentially a bunch of 1 - IOU between 2 different boxes. Essentially lines 2-7 extracts the width and height of the box that represents the intersection between 2 bounding boxes. Then lines 10-13 finds both the intersection (I) and the union (U). The intersection can simply be calculated as the width times the height of the box that represents the intersection between 2 bounding boxes, and the union can be the area of 1 box summed with the area of the second box minus the overlap (or the intersection area). The reason why we need to subtract the overlap is because the overlap is summed twice (once per bounding box area calculation). We then return the intersection (I) divided by the union (U).

```
1  def calculate_IOU(box_1, box_2):
2      c1, c2 = box_1['center'], box_2['center']
3      h1, h2 = box_1['height'], box_2['height']
4      w1, w2 = box_1['width'], box_2['width']
5
6      w_i = min(c1[0] + w1, c2[0] + w2) - max(c1[0], c2[0])
7      h_i = min(c1[1] + h1, c2[1] + h2) - max(c1[1], c2[1])
8      if w_i <= 0 or h_i <= 0:  # No overlap
9          return 0
10     area_1 = w1 * h1
11     area_2 = w2 * h2
12     I = w_i * h_i
13     U = area_1 + area_2 - I  # Union = Total Area - I
14     return I / U
```

4.5 Kalman Filter Implementation

When designing the Kalman Filter, our team decided that constructing a class would be the most organized method. We would decided we would split the Kalman Filter process into 2 subparts: Predict and Update. In terms of Predict, we would determine the predicted next state as well as the predicted covariance matrix associated with this next state. The code is shown below as follows:

```
def predict(self, x_curr, P_curr):
    x_next = self.A @ x_curr.T
    p_next = self.A @ P_curr @ self.A.T + self.Q
    return x_next, p_next
```

In the predict function, we get the predicted state of the next frame for a certain object as well as the predicted covariance matrix for this object. As mentioned when discussing the math behind the Kalman Filter, we explained that the the next state is defined by X_{KP} = AX_{K-1} + BU_K + W_K. However for our implementation, we omitted the BU_K removing acceleration and assuming a constant velocity model for simplicity. We also removed W_K which is a Gaussian noise component for simplifying our implementation. Therefore, our equations are modified to X_{KP} = AX_{K-1} and P_{KP} = A P_{K-1} A^T + Q. Below displays code on how we created the matrix for A and Q with tunable parameters for the Gaussian Noise associated with Q.

```
# State Transition Matrix
self.A = np.array([[1, 0, self.deltaT, 0],
                   [0, 1, 0, self.deltaT],
                   [0, 0, 1, 0],
                   [0, 0, 0, 1]])
```

```
# Tune these
self.process_cv = {
    "wx": 1000,
    "wy": 1000,
    "ww": 1000,
    "wh": 1000,
}
# Process Noise Covariance
self.Q = np.array([
    [(self.deltaT ** 4) / 4 * self.process_cv["wx"], 0, (self.deltaT ** 3) / 2 *
    self.process_cv["wx"], 0],
    [0, (self.deltaT ** 4) / 4 * self.process_cv["wy"], 0, (self.deltaT ** 3) / 2
    * self.process_cv["wy"]],
    [(self.deltaT ** 3) / 2 * self.process_cv["wx"], 0, self.deltaT ** 2 * self.
    process_cv["wx"], 0],
    [0, (self.deltaT ** 3) / 2 * self.process_cv["wy"], 0, self.deltaT ** 2 * self
    .process_cv["wy"]],
])
```

The second part to the Kalman filter as mentioned above is the Update state. This includes utilizing the measurement, the kalman gain, and the predicted state and covariance matrix to determine the target state and covariance matrix. Firstly, as shown in line 2, we take our measurements and extract the positional data out of it using C which is saved into a new variable called y_k. y_k represents Y_{KMF} in our Kalman Filter math. Note that we did not include Gaussian Noise Z_M for simplicity. After retrieving our measured state, we then calculate the kalman gain in line 4 by applying K = P_{KP} H^T $(HP_{KP}H^T + R)^{-1}$. After determining the kalman gain, we calculate the target state as shown in line 6 and the target covariance matrix in line 8.

```
def update(self, measurement, x_curr, P_curr):
    y_k = self.C @ measurement

    kalman_gain = P_curr @ self.H.T @ np.linalg.inv(self.H @ P_curr @ self.H.T +
    self.R)

    new_state = x_curr + kalman_gain @ (y_k - self.H @ x_curr)

    p_new = (np.identity(4) - kalman_gain @ self.H) @ P_curr

    return new_state, p_new
```

Below shows the implementation of C, H, and R for our update step in the Kalman Filter.

```
# Positional Extracter
self.C = np.array([[1, 0, 0, 0],
                   [0, 1, 0, 0]])
```

```
# Measurement Matrix
self.H = np.array([[1, 0, 0, 0],
                   [0, 1, 0, 0]])
```

```
# Measurement Noise Covariance (diagonal elements are covariances)
self.R = np.array([[1, 0],
                   [0, 1]])
```

5 Test Analysis

The goal of this project was not to rigorously test our algorithm, but rather present a proof of concept. Therefore, we pulled a sample video of pedestrians walking from a street camera. Below shows the original video and the annotated video:

Original Video: `https://drive.google.com/file/d/159DRfPjfw7gG9U4OIhOj6E-o1h5YJvme/view?usp=drive_link`

Annotated Video: `https://drive.google.com/file/d/1T3DI7qxtchjWrwcqz6lDWGCZmAm98B3G/view?usp=drive_link`

The annotated video contains tracks of objects that are associated with ids. However, there are some small faults that our team noticed when testing our output. One extremely noticeable fault that occurred was the fact that the track associated with the biker changed ids midway. Below shows 3 separate frames of the same biker with 3 different ids.

Figure 4: Biker with id=1

Figure 5: Biker with id=2

Figure 6: Biker with id=4

The reason for this issue is because of the bounding box overlap from the previous and current frame. There is a parameter called self.IOU_min that is set in the tracker to essentially 0 out values that are below this minimum in our cost matrix within the Hungarian algorithms. The reason why we have this is due to the fact that if there are lots of people that are cluttered in a single frame, we need to be able to confidently say that a certain object belongs to a certain track. This however becomes an issue for faster moving object because as the distance between the position of an object increases between 2 consecutive frames, the bounding box IOU associated with this object will decrease by a much larger factor. Therefore, the IOU is bound to go below self.IOU_min. This means that a new track will be created for the same object.

Another more subtle issue with our tracker is the fact that certain id numbers may seem like they are being skipped when new objects enter the frame. One simple example was the last pedestrian that was walking with a track id of 15.

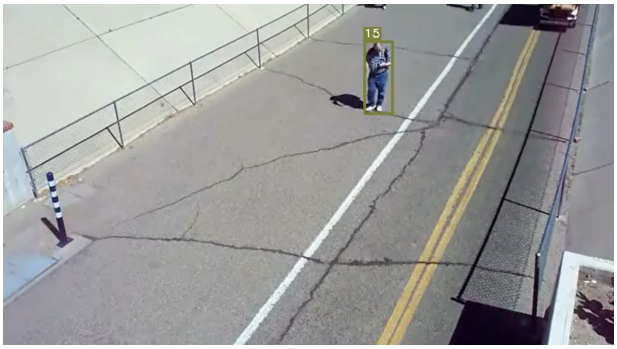

Figure 7: Pedestrian with id=15

Before this pedestrian entered the canvas, the last id that was visually spotted was an id of 12 where the Tracker accidentally mistook a pole for a pedestrian (false positive) which is shown below.

Figure 8: Pole with id=12

Overall though, the main issue is how exactly the track ids jumped from 12 to 15. This is due to condition placed on objects leaving the canvas. When an object appears on the canvas, a bounding box is placed around the canvas. However, the bounding box could be very unstable and could expand or compress. The method of how our team tracks if an object leaves the canvas is by checking the velocity of the center of the bounding box. The reason why we need to check for the velocity is because the bounding box does not expand outside of the frame, hence we cannot simply just check the center coordinate. If it position and velocity says that an object is leaving the canvas, we would assume that the track should be marked as inactive. However, if an object enters the frame and the box expands or compresses, the center is bound to have an unstable velocity meaning that there could accidentally be a scenario in which the position and velocity states that the object is leaving canvas. Therefore, unnecessary tracks are created and skipping of id numbers will occur.

Overall, our SORT tracker does accomplish the goal of tracking objects throughout multiple consecutive frames. Objects that are moving a medium pace will retain the same track id from when they enter till when they leave the canvas. Even objects that intersect with each other retained the same id as shown below.

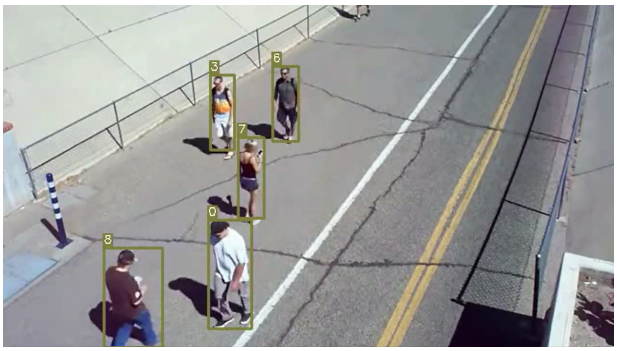

Figure 9: Pre Intersection

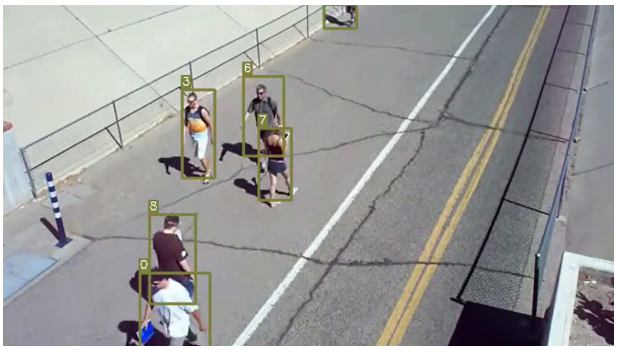

Figure 10: Intersection

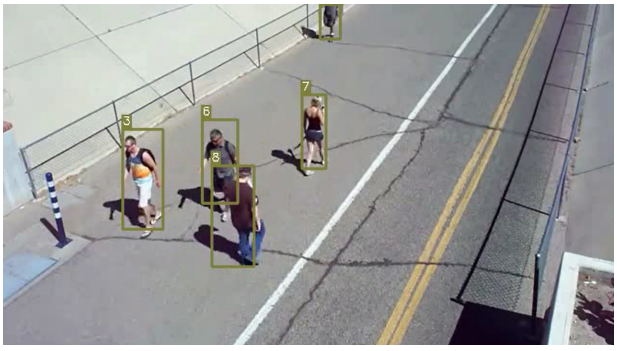

Figure 11: Post Intersection

In these sets of images, we can see that tracks 0 and 8 and tracks 6 and 7 intersect with each other. However, even after they intersect, they retain their respective ids and continue their current path. This means that the IOU calculation for assigning objects and SORT implementation was a success.

6 Conclusion

The goal of this project was to track objects throughout consecutive frames using a renowned algorithm known as SORT (simple online realtime tracking). Based off of the video demonstration of our tracker, our algorithm ran successful according to the goals of our project. Not only was this project a success, our team bolstered our skills by reading through various papers, decoding the math behind the algorithms that build up SORT, as well as have a deeper understanding on why MOT algorithms is necessary in this generation of technology. As mentioned before, various industries require MOT algorithms such as the autonomous vehicle industry in terms of a safety aspect. Self driving vehicles needs to be able to have its own vision and be able to detect and identify nearby objects to make informed decisions.

7 Group Contributions

Our team did not end up splitting work up and used our combined efforts to tackling each and every part of the SORT algorithm. We wrote the Kalman Filter, Hungarian Algorithms, Tester code, as well as the Tracker code together so that we could get multiple eyes on the same code base just in case one person finds an issue or problem that could arise.

References

[1]https://arxiv.org/pdf/1602.00763.pdf

[2]https://arxiv.org/pdf/1506.02640.pdf

[3]https://jonathan-hui.medium.com/real-time-object-detection-with-yolo-yolov2-28b1b93e2088

[4]https://stats.stackexchange.com/questions/287486/yolo-loss-function-explanation

[5]https://towardsdatascience.com/what-i-was-missing-while-using-the-kalman-filter-for-object-tracking-
gi=d9a10f39b4ce

[6]https://www.youtube.com/watch?v=CaCcOwJPytQ&list=PLX2gX-ftPVXU3oUFNATxGXY90AULiqnWT&
index=2

[7]https://thekalmanfilter.com/kalman-filter-explained-simply/

[8]https://www.youtube.com/watch?v=cQ5MsiGaDY8&t=1s

[9]https://thinkautonomous.medium.com/computer-vision-for-tracking-8220759eee85