VIDEO-BASED HUMAN TRACKING

A dissertation submitted for the degree of BEng with Honours in Electrical and Electronic Engineering

Student Name: **Jokubas Ziziunas**

Student ID: **170432405**

Supervisor: **Dr Mohsen Naqvi**

Second Examiner: **Prof Satnam Dlay**

Date: **01/05/2020**

Abstract

Greatly rising amount of video footage produced by cameras require means of processing it automatically. Therefore, the project is dedicated for reviewing and implementing some of the methods that are used for tracking object from video footage. We shall be concerned with human tracking only throughout this project, although similar techniques may be applied for any other kind of object tracking.

Aims and objectives of the project were met successfully as we used two video sequences to track people within. A background subtraction method has been implemented in order to obtain measurements for tracking. It is known that these measurements are quite often noisy or absent. And sometimes we would like predict the future state of the system. We have implemented following tracking techniques in order to deal with that — Kalman filter, extended Kalman filter and particle filter. It has been all done using Python programming language and OpenCV image processing framework.

The background theory for these filters has been introduced. The work goes through steps of implementing all three of the filters and discusses strengths, weaknesses, limitations, design choices and other considerations. Also, performance of the techniques has been evaluated.

In practice, more complicated systems are used for human tracking that take into consideration many more variables, however all the following work can be considered as a founding block for the application and can be expanded upon to be able to track multiple and varying number of targets and work under any conditions.

Acknowledgements

I would like to thank my thesis supervisor Dr Mohsen Naqvi for opportunity to work on this project and help, guidance and feedback throughout the project, my second marker and Image Processing lecturer Prof Satnam Dlay for valuable information in the lectures and drawing my interest towards image processing field.

Table of Contents

1. Introduction

1.1. Background and Motivation

Number of CCTV cameras is on a steady rise, number estimated to be from 4.9 million to 5.9 million in the UK alone [1]. Such a large amount of footage cannot be processed manually. This increased the demand for techniques that help us process the footage automatically, as effectively and as efficiently as possible. Video-based human tracking is a difficult process and there are number of things to consider (**figures 1, 2, 3**) [2][3].

Figure 1. Background Clutter

Figure 2. Multiple objects of interest

Figure 3. Occlusions

Although we are concerned only with human tracking, techniques used throughout the project are also suitable for tracking other objects. Because of this, in recent years, there have been major breakthroughs in development of autonomous vehicles, public safety or national security systems and assisted living and healthcare technology [4][5][6].

The ultimate goal is to estimate the state of an object given some measurements. In video-based tracking we are interested in kinematics only, meaning the state of object is typically represented by position, speed, acceleration [7]. There are generally three distinct steps in the process of object surveillance:

- Detection. Obtaining measurements from the sensors, such as radars or as in our case — video cameras. Although important, not the main interest of the project.
- Recognition. Recognising what is the detected object or whether we are interested in it. Usually requires deep learning and data association techniques [8][9]. Not an interest of this project.
- Tracking. Predicts future of the system, reduces noise introduced by inaccurate detections and associates the object to its track. Focal point of the project. For different problems, different approaches shall be used.

Object tracking techniques require strong fundamentals of statistics, especially Bayesian statistics. In dynamic state estimation, we shall use Bayesian inference methods to construct posterior probability density function of the state given the information we have [10]. State space representation of the models will make it convenient to handle multivariate data and handle non-linear processes [11].

Depending on a problem, we shall employ different techniques for tracking. For most basic problems that assume linear movement and normally distributed (Gaussian) noise, Kalman filter will be suited. For systems that are not linear but still possess Gaussian noise, extended Kalman filter may be used. For highly non-linear and non-Gaussian systems particle filter shall be used. Some techniques might be better than others, however, in computing better quality usually comes at a cost of reduced efficiency and increased complexity [10][11].

Throughout the project we shall be using Python programming language. Open-source computer vision framework OpenCV is very convenient for handling problems related to object tracking and

image processing. Although MATLAB does provide Image Processing Toolbox, it is not free for everyone, hence, usually less support available. Support for computer vision is widely available for other languages such as C++, however, Python has simple syntax and offers “readability”, which allows to focus on the design rather than the programming itself, which is why it is becoming prevalent in the computer vision field.

1.2. Aim

We are interested in tracking moving human in the video footage. In order to detect moving person, background subtraction method will be used. To predict future state of the system, reduce noise introduced by inaccurate measurements and associate object to its tracks various tracking techniques will be used [12]. Results obtained throughout project shall be used to evaluate suitability of mentioned techniques for different kinds of problems.

1.3. Objectives

- Extensive background study and literature review.
- Human detection using background subtraction technique.
- Human tracking using multiple different methods: Kalman filter, extended Kalman filter, particle filter.
- Evaluation of the mentioned techniques.

2. Literature Review

It is worth noting that in this field research, people are usually interested in tracking larger variety of objects. Yet, most of the literature mentioned below is still relevant even if some of it may not be used exclusively for people tracking. Human movement is in a sense barely different from movement of an aircraft, vehicle or any other object that we may want to track using video footage.

In real world problems, previously mentioned filters in its simplest form usually are not that useful. Practical solutions to our problems tend to be more complicated and usually are a combination of multiple detection, recognition and tracking techniques, hence, following section is dedicated for a brief review of methods that are used in field of computer vision.

2.1. Object Detection

Methods that are used to detect objects usually fall into two categories: machine learning-based (in a broader sense) or deep learning-based.

For approaches that use machine learning, it is needed to define certain features of object in interest and use some sort of classification technique. P.Viola and M.Jones introduced object detection framework that is based on Haar features (**figure 4**). Training is done using many positive and negative images and then cascade of classifiers are applied throughout the image [13][14]. Another method uses an idea that local shape and appearance of an object in question is related to the distribution of intensity gradients [15].

For deep learning approaches, one is known as region-based convolutional neural networks. Selective search algorithm is applied throughout the image to produce separate region that potentially contain object in interest. After that, given any region, a forward propagation through neural network creates a feature vector which is then consumed by a binary support-vector machine that performs classification [16]. Another model called YOLO (You Only Look Once) instead of classification looks into object detection as a regression problem and has been used more extensively due to superior speed and reduced number of false positives [17].

2.2. Object Tracking

A. Yilmaz et al. categorizes object tracking methods as shown in **figure 4** [18].

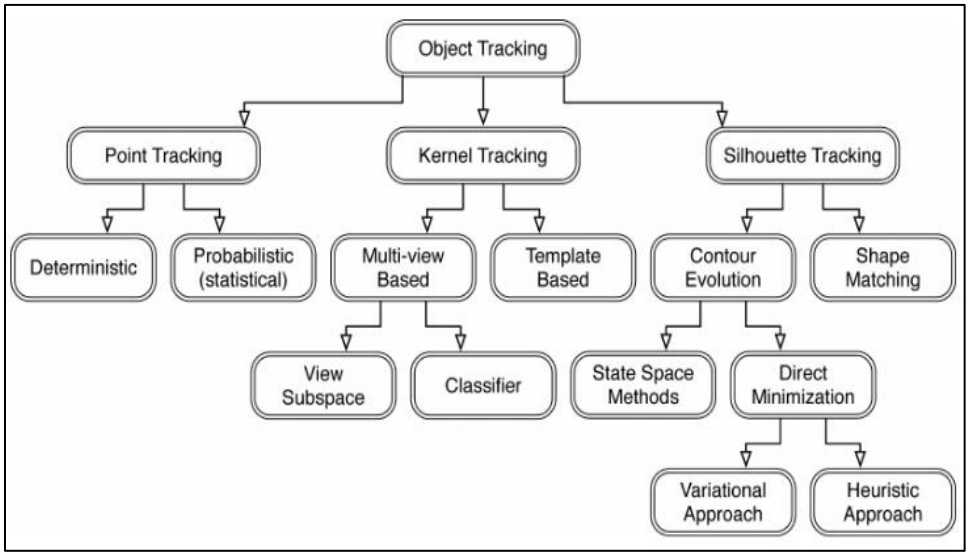


Figure 4. Taxonomy of target tracking methods

Many different techniques may be applied for object tracking, methods that are used may vary depending on what kind of tracking we would like to perform. It is a very broad topic and all the tracking methods that are going to be used in this work will fall into category of probabilistic (statistical) point tracking. Besides Kalman, extended Kalman and particle filters that shall be discussed later on, following is brief review of the filters that may be used for object tracking.

Alpha-beta filter (sometimes g-h filter) is a simplified version of Kalman filter, easy to understand for someone who is just getting familiarized with these kind of tracking methods. While Kalman filter varies its coefficients during every time step in order to make the next estimate [10], alpha-beta filter uses fixed coefficients or they also be changed on demand within the program [19].

Unscented Kalman filter – in a sense improvement of extended Kalman filter. Extended Kalman filter is as good as linearization is, and for highly non-linear systems it might be problematic. Unscented Kalman filter works around this problem by producing a few sample (Sigma) points around the current estimate of the state. These points are propagated through the function that is non-linear and new, more accurate estimates of mean and covariance are obtained [20][21].

Mentioned techniques fall into category of filters that are used for single target tracking exclusively. However, in real life scenarios it is rarely useful. Following type of filters are able to deal with more complicated scenarios. They build upon techniques that are used for single target tracking and implement ability to keep track of multiple and variable number of targets. Probability Hypothesis Density (PHD) filter – uses finite set statistics for the target tracking. Can be seen as a set of filters that communicate to oracle that makes decisions on which filter should be attached to which target [22]. One technique, for example uses Gaussian mixture model as means to implement multiple object tracking [23], other uses sequential Monte Carlo method through the set of weighted random particles [24]. These techniques are generally more effective and robust as they are not restricted by non-linearities, noise and other problems that may arise and may be used almost under any circumstances.

3. Object Detection Implementation

In this project, background subtraction will be used as a means of detection. There are quite a few ways of performing background subtraction, however all of them follow a very similar algorithm [25]:

- Obtain a current and a background frame.
- Convert both images into greyscale images.
- Convert background and current frames into a binary image, depending on their grey levels.
- Perform XOR operation between corresponding pixels of background and current frames.

Resultant will be a binary image, that has two regions – background and foreground. Then we may apply thresholds to relegate objects that are not of interest back into the background.

The algorithm described above is not very practical on its own as it assumes static background, also does not represent colours very accurately and does not take illumination into consideration. OpenCV library provides with a background subtractor class MOG2 (Mixture of Gaussians). It uses a technique of modelling each background pixel with a mixture of Gaussian distributions. Weight of each of the distributions is proportional to the amount of time certain colour is present on the pixel. Higher weights then can be assigned to background, as in the background same colours stay in the same place longer allowing for dynamic background [26]. Dataset from ChangeDetection.net research group library has been used for this experiment [27] and background subtraction will be performed. OpenCV background subtractor also takes in an optional argument – *historyTime*. *It* defines how fast objects who previously moved get assigned to background once it stops moving [28]. The results of varying *historyTime* are shown below (**figure 5**).

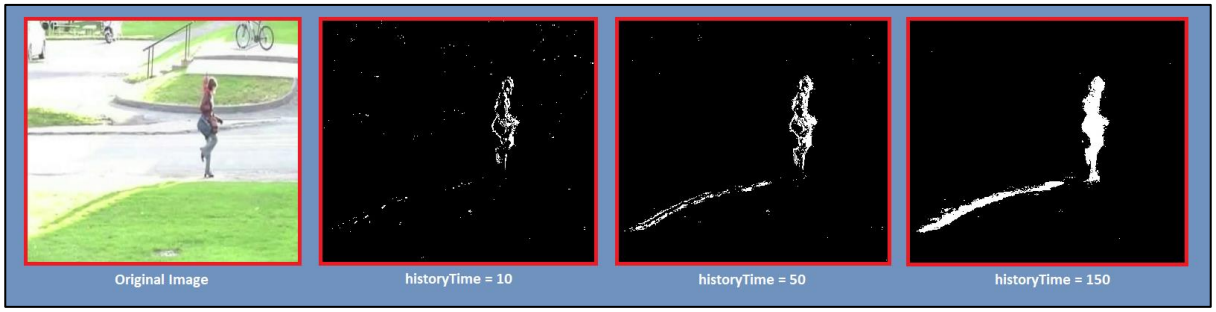


Figure 5. Detected foreground with different historyTime values

In our case person that was walking stopped for a second. You would want to use higher *historyTime* in cases where object moves very slowly or even stops sometimes. However, too high values would make background subtraction more sensitive to sudden changes, which is better in busy traffic scenarios or for cases where object appears in the frame for a short period of time [26]. In this case, higher value also detects shadows and we do not want it, hence we will use intermediate value of 50. By performing background subtraction, we have obtained image that contains background and foreground, but it is not very useful on its own. In order to track object, we need its coordinates. For this purpose, a *detect* function (line 16) has been written, that returns coordinates of every object detected in the frame and draws a bounding rectangle around it. It uses an OpenCV functions *findContours* and *boundingRect* in order to do this. We saw that produced foreground mask contains some noise, hence, before calling *detect* function, we are going to apply median filter, because finding contours is quite lengthy operation and we do not want to use it on every piece of noise [29]. Also, *detect* function contains a piece of code that finds an area of every detected object and returns

coordinates of center of an object if it passes minimum area threshold. **Figure 6** shows a piece of code that detects an object given a foreground mask, while full code is provided in the **appendix**.

```
def detect(self):
    '''centers are measurements for filtering, Coordiantes are trivial, for
    rectangle drawing on a picture, not actually used for tracking. Here we
    find countours of every shape, find centroid of this shape, and if it
    passes the minimumAreaThreshold, it is then counted as a detected human.'''
    centers, coordinates = [], []
    self.frame = cv2.medianBlur(self.frame, 5)
    contours, hierarchy = cv2.findContours(self.frame, cv2.RETR_EXTERNAL,
                                           cv2.CHAIN_APPROX_SIMPLE)
    boundingRectangles = [None]*len(contours)

    for i in range(len(contours)):
        boundingRectangles[i] = cv2.boundingRect(contours[i])
    for i in range(len(boundingRectangles)):
        x1 = (int(boundingRectangles[i][0]))
        y1 = (int(boundingRectangles[i][1]))
        x2 = (int(boundingRectangles[i][0]+boundingRectangles[i][2]))
        y2 = (int(boundingRectangles[i][1]+boundingRectangles[i][3]))
        center = [((x2 + x1) / 2), ((y2 + y1) / 2)]

        if ((abs(x2 - x1) * abs(y2 - y1)) > self.minimumDetectionArea):
            centers.append(center)
            coordinates.append(np.array([x1, y1, x2, y2]))
     return centers, coordinates
```

Figure 6. Code for detecting object in a foreground mask

MinimumDetectionArea has to be chosen carefully and separately for every video, because sometimes person may be close to the camera and sometimes – further away. **Figure 7** shows what impact varying *minimumDetectionArea* has for two different videos. In the first video, because person is further away from camera, value of 6000 is not enough to detect a person, but in second video it is enough for detection. Also worth noting that small values introduce a few false detections.

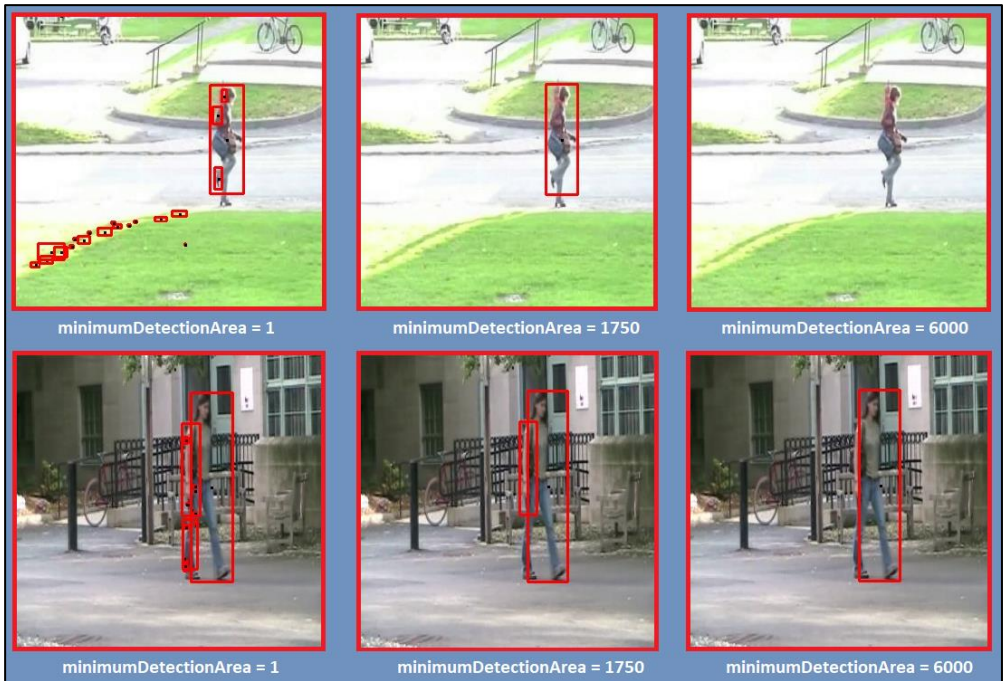


Figure 7. Detected people with different minimumDetectionArea values

Result of whole background subtraction procedure is shown in **figure 8**. **Figures 9** and **10** show a scatter plot of detections throughout the first and second videos as people walk through the clip.

Figure 8. Background Subtraction and Detection

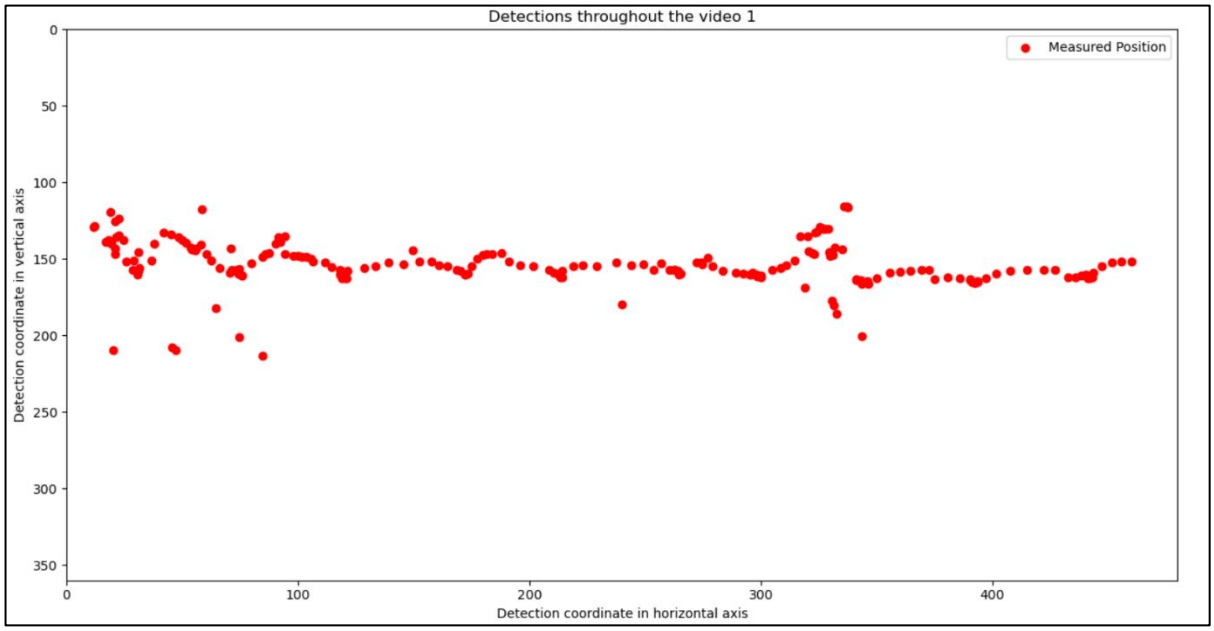
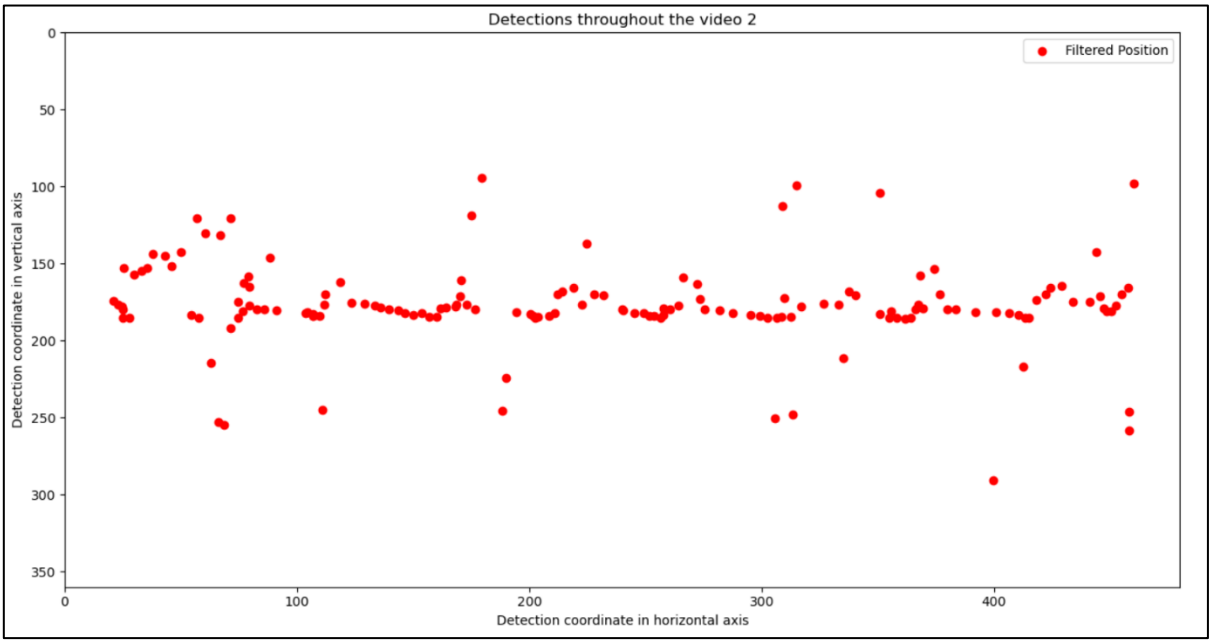


Figure 9. Detection throughout the video 1.



Note that both in **figures 9** and **10** there are periodic peaks in vertical axis. This happens, because person has to raise the leg every time the step is taken, hence shifting the centre of detection upwards (**figure 11**). This could be overcome by using a different detection technique, for example, detecting only face using previously discussed method based on Haar Cascade Classifiers.

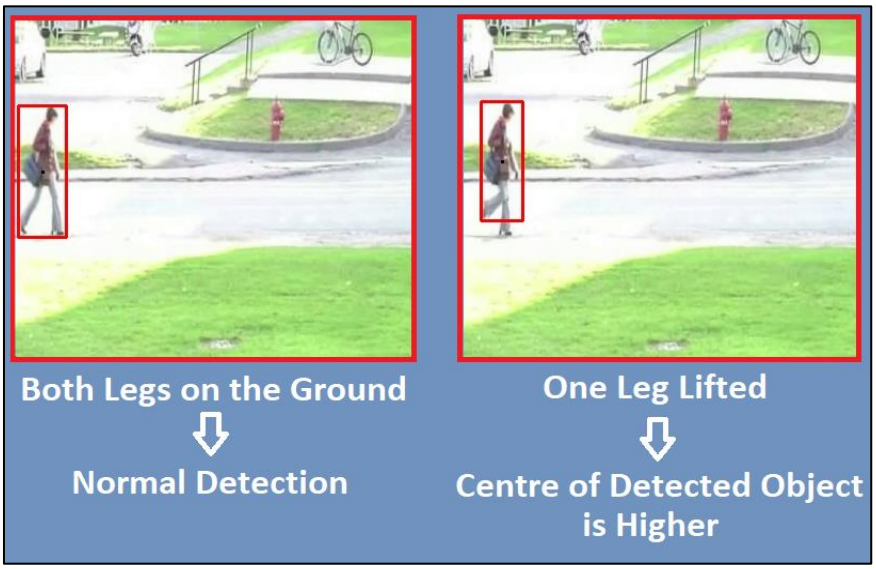


Figure 11. Explanation of periodic peaks occurring in detections in vertical axis

4. Object Tracking Implementation

4.1. Theory Behind Bayesian Filtering

Kalman, extended Kalman and particle filters all fall into category of Bayesian filters. For tracking problems, we shall assume that state vector $\boldsymbol{x_k}$ is a random variable, whose particular value we are trying to estimate [10]. This is known as stochastic process and the state we are tracking evolves according to the discrete-time stochastic model [11]:

$$\boldsymbol{x_k = f_{k-1}(x_{k-1}, v_{k-1})} \qquad \underline{\textit{Equation 1}}$$

Here $\boldsymbol{f_{k-1}}$ is a function of the state describing the evolution of state vector $\boldsymbol{x_k}$ and $\boldsymbol{v_{k-1}}$ stands for the process noise, both of which may be time-dependent and non-linear. Here $\boldsymbol{x_k}$ is assumed to be unobservable or latent. Observable are the measurements that we obtain. We denote measurement vector $\boldsymbol{z_k}$ which is also a random variable corrupted by noise.

$$\boldsymbol{z_k = h_k(x_k, w_k)} \qquad \underline{\textit{Equation 2}}$$

Where $\boldsymbol{h_k}$ is a function of current state $\boldsymbol{x_k}$ and the measurement noise $\boldsymbol{w_k}$, both of which may also be time-dependent and non-linear.

The filtering problem requires us to estimate $\boldsymbol{x_k}$ given system model and all the measurements of data $\boldsymbol{z_k}$ up to and including discrete-time step $\boldsymbol{k}$ [30](**figure 12**).

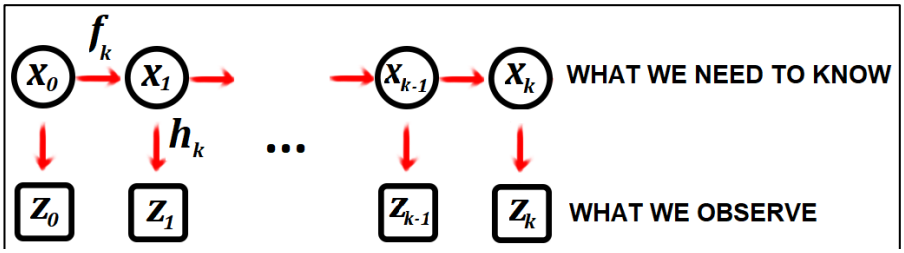


Figure 12. Bayesian Filtering

Since it is not deterministic model, initial state is assumed to have a known probability density function (pdf) $\boldsymbol{p(x_0)}$, transitional density $\boldsymbol{p(x_k|x_{k-1})}$ and likelihood function $\boldsymbol{p(z_k|x_k)}$**.** This information will be used to recursively estimate $\boldsymbol{x_k}$ in two steps: prediction (prior) and update (posterior) [31].

For prediction, in order to estimate $\boldsymbol{p(x_k)}$ we assume that $\boldsymbol{p(z_{k-1}|x_{k-1})}$ is available. We use system model to obtain prior pdf using Chapman-Kolomogorov equation [11]:

$$\boldsymbol{p(x_k|z_{k-1}) = \int p(x_k|x_{k-1})p(x_{k-1}|z_{k-1})\, dx_k} \qquad \underline{\textit{Equation 3}}$$

Update step incorporates the measurements by applying Bayes' theorem in order to obtain posterior pdf [11]:

$$\begin{aligned} \boldsymbol{p(x_k|z_k)} &= \boldsymbol{p(x_k|z_k,\, z_{k-1})} \\ &= \frac{\boldsymbol{p(z_k|x_k,\, z_{k-1})p(x_k|z_{k-1})}}{\boldsymbol{p(z_k|z_{k-1})}} \\ &= \frac{\boldsymbol{p(z_k|x_k)p(x_k|z_{k-1})}}{\boldsymbol{p(z_k|z_{k-1})}} \end{aligned} \qquad \underline{\textit{Equation 4}}$$

Knowledge of posterior pdf $\boldsymbol{p(x_k|z_k)}$ allows us to find the solution for $\boldsymbol{x_k}$**.** One of the ways to do it would be a minimum mean-square estimation (MMSE), which finds the expected value of $\boldsymbol{x_k}$ [10]:

$$\boldsymbol{\hat{x}_{k|k}^{MMSE}} \triangleq \mathbb{E}\{\boldsymbol{x_k|z_k}\} = \int \boldsymbol{x_k\, p(x_k|z_k)dx_k} \qquad \underline{\textit{Equation 5}}$$

Another way would be to find maximum of the posterior pdf (MAP) [10]:

$$\boldsymbol{\hat{x}_{k|k}^{MAP}} \triangleq arg\ max\ \boldsymbol{p(x_k|z_k)} \qquad \underline{\textit{Equation 6}}$$

In addition to this, we are also able to quantify accuracy of the estimate, obtaining, for example, covariance from $\boldsymbol{p(x_k|z_k)}$. It is worth noting that recursive estimation using prediction and update steps is only a conceptual solution, because generally, equations 3 and 4 cannot be solved analytically, except for a set of restrictive cases [11].

4.2. Kalman Filter

One such exception is known as Kalman filter. Here we assume that posterior pdf may be expressed by a Gaussian $\boldsymbol{N(\mu, \sigma^2)}$, where $\boldsymbol{\mu}$ is mean and $\boldsymbol{\sigma^2}$ is variance, meaning that only two parameters completely characterize $\boldsymbol{x_k}$ [11]. If $\boldsymbol{p(x_{k-1}|z_{k-1})}$ is Gaussian, it can proved that $\boldsymbol{p(x_k|z_k)}$ will also be a Gaussian if certain conditions are met [32]:

- Noise terms $\boldsymbol{v_{k-1}}$ and $\boldsymbol{w_k}$ are random Gaussian variables.
- $\boldsymbol{f_{k-1}(x_{k-1}, v_{k-1})}$ is a known linear function.
- $\boldsymbol{h_k(x_k, w_k)}$ is a known linear function.

In order to implement Kalman filter we will have to introduce some definitions and notation [32]:

- State vector $\boldsymbol{x_k}$ of dimensions $\boldsymbol{x_{dim} \times 1}$, where $\boldsymbol{x_{dim}}$ is number of state variables.
- Measurement vector $\boldsymbol{z_k}$ of dimensions $\boldsymbol{z_{dim} \times 1}$**,** where $\boldsymbol{z_{dim}}$ is number of measurements.
- Previously $\boldsymbol{f_k}$**,** now $\boldsymbol{F_k}$ — state transition matrix of dimensions $\boldsymbol{x_{dim} \times x_{dim}}$.
- Previously $\boldsymbol{h_k}$**,** now $\boldsymbol{H_k}$ — measurement matrix of dimensions $\boldsymbol{z_{dim} \times x_{dim}}$.
- Process noise covariance matrix $\boldsymbol{Q_k}$ of dimensions $\boldsymbol{x_{dim} \times x_{dim}}$.
- Measurement noise covariance matrix $\boldsymbol{R_k}$ of dimensions $\boldsymbol{z_{dim} \times z_{dim}}$.
- Process covariance matrix $\boldsymbol{P_k}$ of dimensions $\boldsymbol{x_{dim} \times x_{dim}}$

Steps for implementing Kalman filter are as follows [32]:

- Initialization:
  - Initialize state of the system, $\boldsymbol{x_k}$.
  - Initialize belief in accuracy of the system, $\boldsymbol{P_k}$.
- Prediction:
  - Use state transition matrix $\boldsymbol{F_{k-1}}$ to propagate the $\boldsymbol{x_k}$ to next time step.
  - Adjust $\boldsymbol{P_k}$ to take the uncertainty of prediction into consideration.
- Update:
  - Obtain the measurement vector $\boldsymbol{z_k}$ and belief about its accuracy, $\boldsymbol{R_k}$.
  - Compute residual between $\boldsymbol{x_k}$ and $\boldsymbol{z_k}$ in measurement space $\boldsymbol{H_k}$ and belief about its accuracy.
  - Compute scaling factor (Kalman gain, $\boldsymbol{K_k}$) based on whether $\boldsymbol{x_k}$ or $\boldsymbol{z_k}$ is more accurate.
  - Update $\boldsymbol{x_k}$ based on $\boldsymbol{K_k}$, in between last prediction and measurement.
  - Update $\boldsymbol{P_k}$ based on how certain we are in measurement.

An informative flow chart regarding these steps are provided in **figure 13** [33].

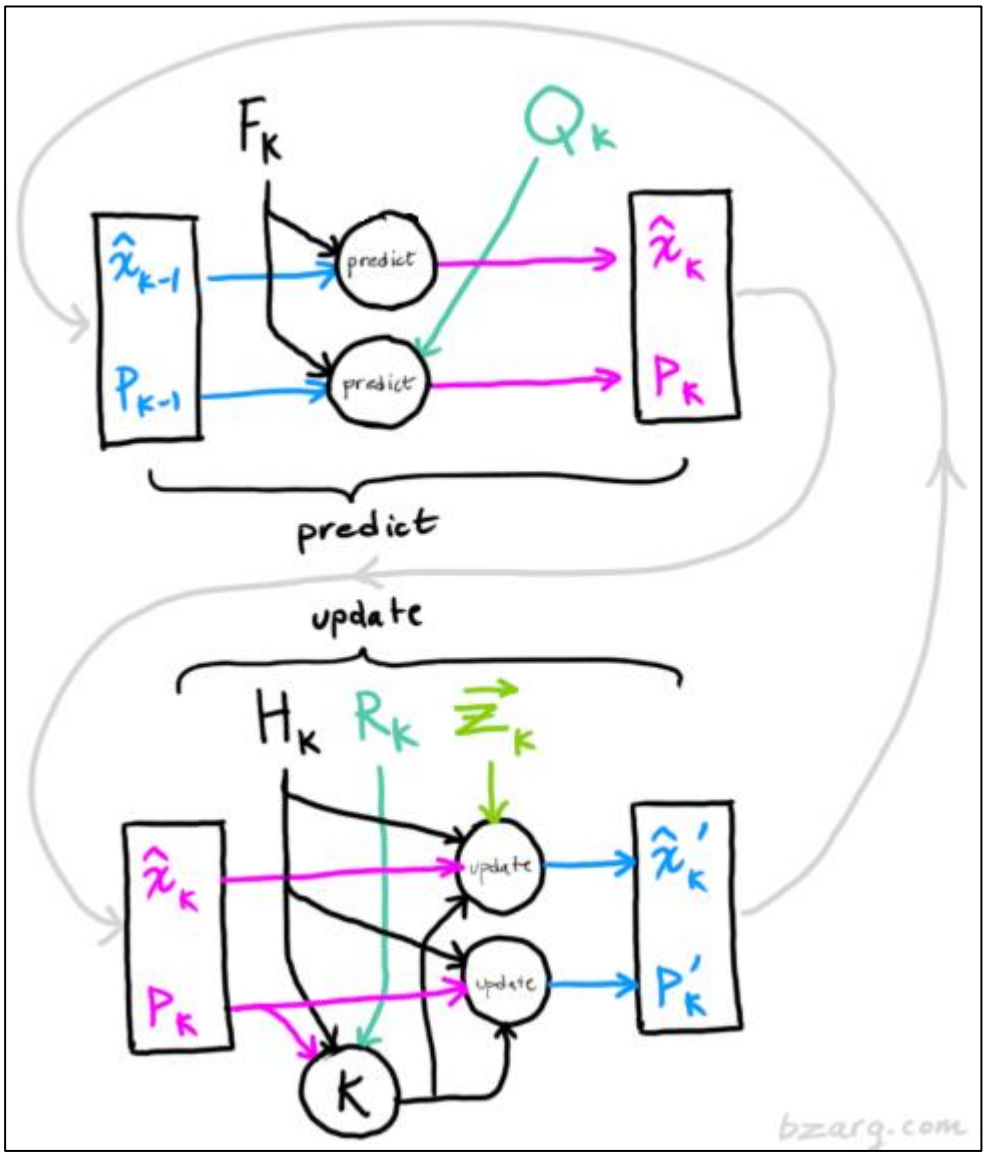


Figure 13. Flow chart of prediction and update steps in Kalman filter

Derivation of each step is provided in [11] and [32]. After we initialize our state and belief about it, we want to propagate it using system model:

$$\hat{\boldsymbol{x}}_{\boldsymbol{k|k-1}} = \boldsymbol{F}_{\boldsymbol{k-1}}\hat{\boldsymbol{x}}_{\boldsymbol{k-1|k-1}} \qquad \text{Equation 7}$$

$$\boldsymbol{P}_{\boldsymbol{k|k-1}} = \boldsymbol{F}_{\boldsymbol{k-1}}\boldsymbol{P}_{\boldsymbol{k-1|k-1}}\boldsymbol{F}_{\boldsymbol{k-1}}^{\boldsymbol{T}} + \boldsymbol{Q}_{\boldsymbol{k-1}} \qquad \text{Equation 8}$$

Where superscript **T** stands for transpose operation. After obtaining measurements, we will "transfer" $\boldsymbol{x_k}$, $\boldsymbol{z_k}$ and $\boldsymbol{P_k}$ into measurement space $\boldsymbol{H_k}$ in order to compute residual vector $\boldsymbol{y_k}$ of dimensions of dimensions $\boldsymbol{z_{dim}} \times \boldsymbol{1}$ between $\boldsymbol{x_k}$ and $\boldsymbol{z_k}$, and covariance matrix $\boldsymbol{S_k}$ of dimensions $\boldsymbol{z_{dim}} \times \boldsymbol{z_{dim}}$ associated with measurements. After this, Kalman gain, $\boldsymbol{K_k}$, would be calculated and previous prediction would be modified accordingly:

$$\boldsymbol{y_k} = \boldsymbol{z_k} - \boldsymbol{H_k}\hat{\boldsymbol{x}}_{\boldsymbol{k|k-1}} \qquad \text{Equation 9}$$

$$\boldsymbol{S_k} = \boldsymbol{H_k}\boldsymbol{P}_{\boldsymbol{k|k-1}}\boldsymbol{H_k^T} + \boldsymbol{R_k} \qquad \text{Equation 10}$$

$$\boldsymbol{K_k} = \boldsymbol{P}_{\boldsymbol{k|k-1}}\boldsymbol{H_k^T}\boldsymbol{S_k^{-1}} \qquad \text{Equation 11}$$

$$\hat{\boldsymbol{x}}_{\boldsymbol{k|k}} = \hat{\boldsymbol{x}}_{\boldsymbol{k|k-1}} + \boldsymbol{K_k}\boldsymbol{y_k} \qquad \text{Equation 12}$$

$$\boldsymbol{P}_{\boldsymbol{k|k}} = (\boldsymbol{I} - \boldsymbol{K_k}\boldsymbol{H_k})\,\boldsymbol{P}_{\boldsymbol{k|k-1}} \qquad \text{Equation 13}$$

Here $\boldsymbol{I}$ stand for identity matrix of dimensions $\boldsymbol{x_{dim}} \times \boldsymbol{x_{dim}}$ and superscript **-1** indicates matrix inversion operation.

We can use all of this to make an optimal prediction about future state of the system, reduce the process noise and noise that was introduced by inaccurate measurements. It helps us to associate object to its tracks as well (**figure 14**) [34].

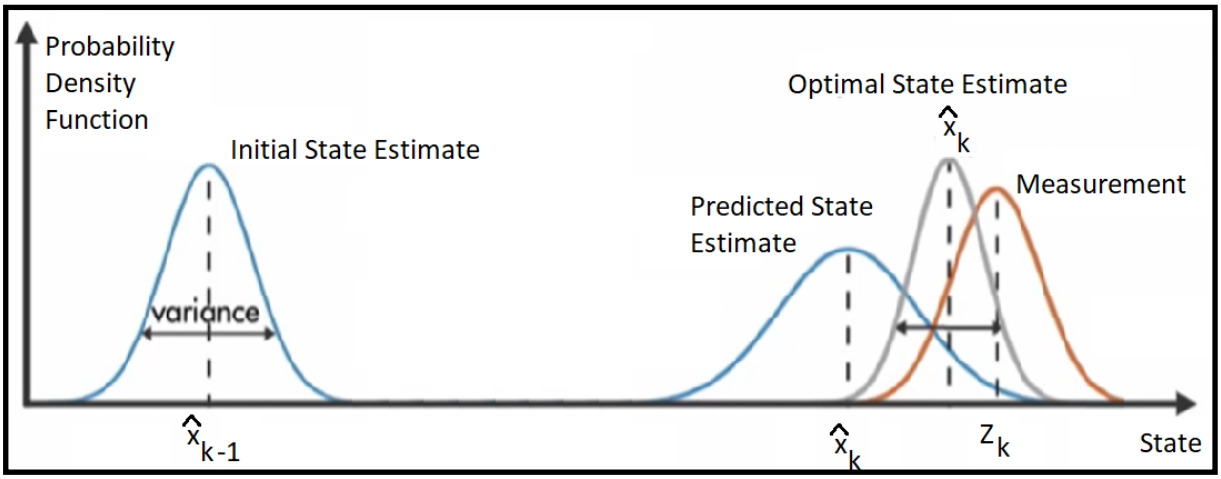


Figure 14. Combination of prediction and measurement

We want to track people that are in video footage. Movement of objects can be described by Newtonian kinematics equations, which will be used to create $\boldsymbol{F_k}$. In single dimension (assuming constant speed) these equations would be:

$$\boldsymbol{p_k} = \boldsymbol{p_{k-1}} + \boldsymbol{v_{k-1}}\boldsymbol{dt} \qquad \text{Equation 14}$$

$$\boldsymbol{v_k} = \boldsymbol{v_{k-1}} \qquad \text{Equation 15}$$

Here $\boldsymbol{p_k}$ (position) and $\boldsymbol{v_k}$ (velocity) are state variables that we are trying to estimate, $\boldsymbol{x_k} = [\boldsymbol{p_k}, \boldsymbol{v_k}]^T$ and measurement we obtain is $\boldsymbol{z_k} = [\boldsymbol{z_{p(k)}}]$, where p denotes that it is position measurement and k denotes time step. This shall be used to make the prediction step:

$$\begin{bmatrix} \boldsymbol{p_k} \\ \boldsymbol{v_k} \end{bmatrix} = \begin{bmatrix} \mathbf{1} & dt \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{p_{k-1}} \\ \boldsymbol{v_{k-1}} \end{bmatrix} \qquad \underline{\textit{Equation 16}}$$

This applies for every kinematic system. We might also use acceleration or take control inputs into account to model the state [32], in this case this will not be necessary.

To account for uncertainty in estimates and measurements, we use covariance matrices, which quantifies joint variability between two variables that are random [32]. $\boldsymbol{P_k}$ is a square matrix and entry P_{ij} of this matrix is amount of covariance i-th state variable has with j-th state variable. Along the diagonal, where *i = j*, the quantity represents the variance of state variable. Again, in one-dimensional case:

$$\boldsymbol{P_k} = \begin{bmatrix} \sigma^2_{\boldsymbol{p_k p_k}} & \sigma^2_{\boldsymbol{p_k v_k}} \\ \sigma^2_{\boldsymbol{v_k p_k}} & \sigma^2_{\boldsymbol{v_k v_k}} \end{bmatrix} \qquad \underline{\textit{Equation 17}}$$

Using all of this information we will simulate a point moving throughout two-dimensional space, with a constant (noisy) speed, obtain (noisy) measurements and implement Kalman filter to obtain optimal estimate. Code is trivially simple, as the only thing it does implements equations 7-13 in a loop, hence it is left out in **appendix**. Results of simulation are shown in **figure 15**

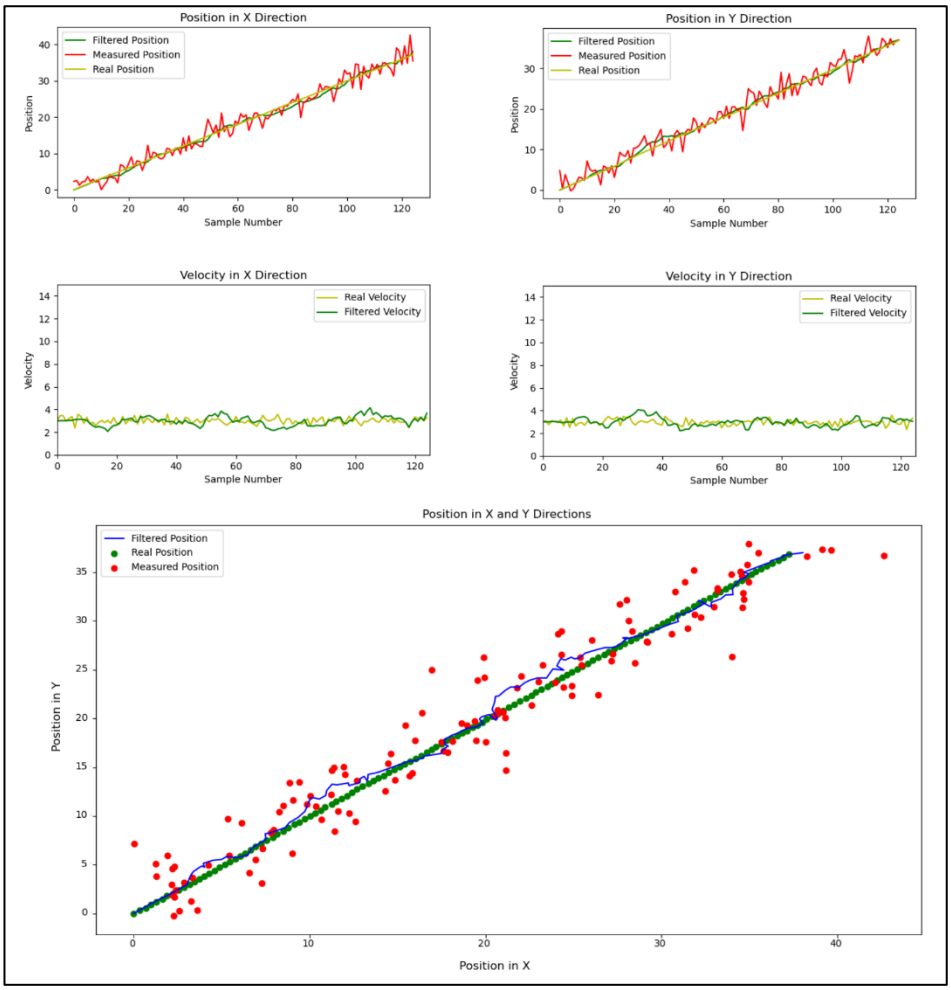


Figure 15. Simulation of moving object and implementation of Kalman filter

4.3. Extended Kalman Filter

As mentioned before, for Kalman filter to work, assumptions must hold. However, in real world, many systems are non-linear and linearity of the systems was one of those assumptions. Trying to implement Kalman filter on a non-linear system would not yield optimal result. One of the ways to work around it would be to use extended Kalman filter. Here, we do not alter the equations, but linearize them at mean of the current estimate [32]. However, we still have to assume noise to be Gaussian. Equations 7 and 9 now replaced with non-linear equivalents:

$$\hat{\boldsymbol{x}}_{\boldsymbol{k|k-1}} = \boldsymbol{f}_{\boldsymbol{k-1}}(\boldsymbol{x}_{\boldsymbol{k-1}}) + \boldsymbol{v}_{\boldsymbol{k-1}} \qquad \textit{Equation 18}$$

$$\boldsymbol{y}_{\boldsymbol{k}} = \boldsymbol{z}_{\boldsymbol{k}} - \boldsymbol{h}_{\boldsymbol{k}}(\hat{\boldsymbol{x}}_{\boldsymbol{k|k-1}}) + \boldsymbol{w}_{\boldsymbol{k}} \qquad \textit{Equation 19}$$

Linearization of $\boldsymbol{f}_{\boldsymbol{k-1}}(\boldsymbol{x}_{\boldsymbol{k-1}})$ and $\boldsymbol{h}_{\boldsymbol{k}}(\hat{\boldsymbol{x}}_{\boldsymbol{k|k-1}})$ involves finding a matrix of partial derivatives, the Jacobian [10], that will be used to substitute the $\boldsymbol{F}_{\boldsymbol{k-1}}$ and $\boldsymbol{H}_{\boldsymbol{k}}$ in regular Kalman filter:

$$\boldsymbol{F}_{\boldsymbol{k-1}} = \left.\frac{\partial \boldsymbol{f}_{\boldsymbol{k-1}}(\boldsymbol{x}_{\boldsymbol{k-1}})}{\partial \boldsymbol{x}_{\boldsymbol{k-1}}}\right|_{\boldsymbol{x}_{\boldsymbol{k-1}}} \qquad \textit{Equation 20}$$

$$\boldsymbol{H}_{\boldsymbol{k}} = \left.\frac{\partial \boldsymbol{h}_{\boldsymbol{k}}(\hat{\boldsymbol{x}}_{\boldsymbol{k|k-1}})}{\partial \hat{\boldsymbol{x}}_{\boldsymbol{k}}}\right|_{\boldsymbol{x}_{\boldsymbol{k}}} \qquad \textit{Equation 21}$$

Using this knowledge we will simulate a process, a point that moves in a non-linear motion following equations 22 and 23:

$$\boldsymbol{p}_{\boldsymbol{x}}(\boldsymbol{t}) = \boldsymbol{sin}(\boldsymbol{t}) + \boldsymbol{t} \qquad \textit{Equation 22}$$

$$\boldsymbol{p}_{\boldsymbol{y}}(\boldsymbol{t}) = \boldsymbol{cos}(\boldsymbol{t}) + \boldsymbol{t} \qquad \textit{Equation 23}$$

Where $\boldsymbol{p}_{\boldsymbol{x}}(\boldsymbol{t})$ and $\boldsymbol{p}_{\boldsymbol{y}}(\boldsymbol{t})$ are positions in x and y axes. From this, knowing that speed is derivative of displacement, we obtain equations 24 and 25 for speed in x and y directions. And the simulation result is shown in **figures 16** and **17**:

$$\frac{\boldsymbol{d}(\boldsymbol{p}_{\boldsymbol{x}}(\boldsymbol{t}))}{\boldsymbol{dt}} = \boldsymbol{cos}(\boldsymbol{t}) + \boldsymbol{1} \qquad \textit{Equation 24}$$

$$\frac{\boldsymbol{d}(\boldsymbol{p}_{\boldsymbol{y}}(\boldsymbol{t}))}{\boldsymbol{dt}} = -\boldsymbol{sin}(\boldsymbol{t}) + \boldsymbol{1} \qquad \textit{Equation 25}$$

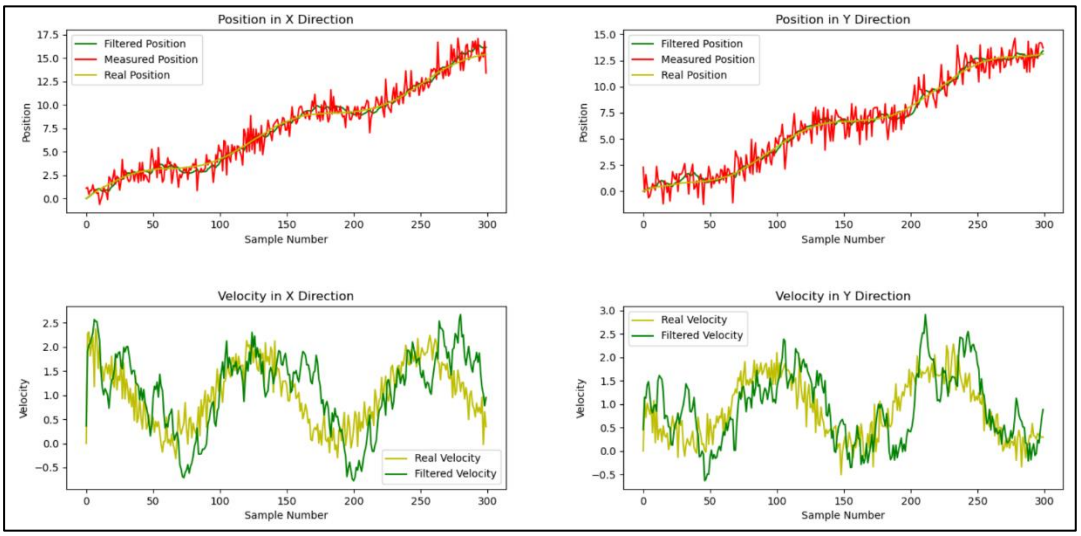


Figure 16. Simulation of moving object and implementation of extended Kalman filter

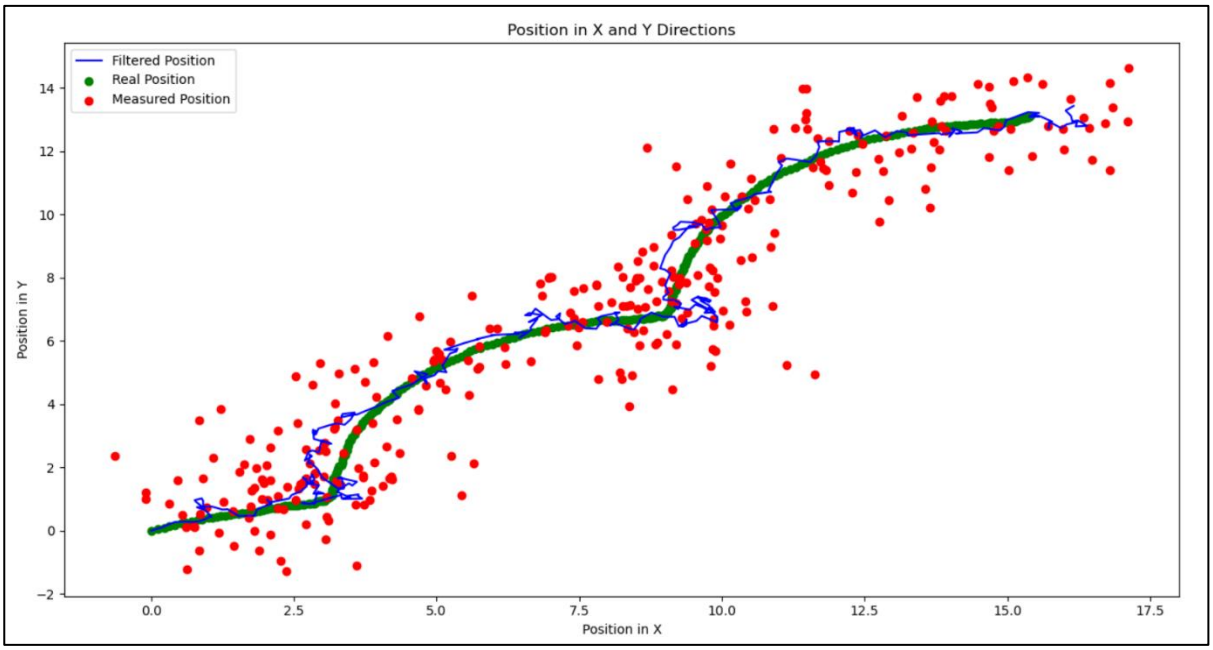


Figure 17. Simulation of moving object and implementation of extended Kalman filter

4.4. Particle Filter

For some tracking applications, Kalman filter is pretty reliable and efficient. However, it is limited to a relatively restricted class of linear Gaussian problems. To solve problems beyond this class, particle filter has proved to be pretty effective for stochastic dynamic state estimation.

Problem with human tracking is that people not always move in straight lines, especially in areas with many obstacles. It may be difficult to describe their motion with any sort of equations, people can abruptly change directions in a way that is impossible to predict.

Particle filters perform sequential Monte Carlo estimation based on particle representation of probability densities to recursively estimate state of the system. It is superior to Kalman filters, because they are able to handle heavy non-linearity, non-Gaussian noise and even unknown process models [11]. Most notable drawback, however, is that requires more computational effort, because here we are calculation full pdf.

Key idea behind implementation of particle filters are to represent posterior pdf by a set of randomly generated samples (particles) together with their associated weights, where each particle represent a *possible* state of the system. General idea for implementation of particle filter goes as follows [32]:

- Randomly generate particles. As many as possible, as this yields a better result. However, it is worth to note that this comes at a computational cost. Particles can represent anything from position, to speed, to heading.
- Predict next state of the particles. Provided that model is known. We move all the particles based on our belief in behaviour of the system.
- Incorporate measurements into prediction. Update weights of particles, assign higher weights to those that match measurements.
- Resample. Discard particles that are highly unlikely, replace them with copies with particles that did well.
- Compute estimate. This can be done by finding mean and variance of every point.

As mentioned, we will use sequential Monte Carlo estimation, therefore, let us have a multidimensional integral that we wish to evaluate:

$$\boldsymbol{I} = \int \boldsymbol{g}(\boldsymbol{x})\boldsymbol{dx} \qquad \text{Equation 26}$$

Monte Carlo method for numerical integration factorizes $\boldsymbol{g}(\boldsymbol{x})$ such that $\boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{\pi}(\boldsymbol{x})\boldsymbol{f}(\boldsymbol{x})$[pf, 35], where $\boldsymbol{\pi}(\boldsymbol{x})$ is effectively a pdf such that $\int \boldsymbol{\pi}(\boldsymbol{x})\boldsymbol{dx} = \mathbf{1}$, where we draw $\boldsymbol{N}$ samples $\{\boldsymbol{x^i}; \boldsymbol{i} = \mathbf{1}, \dots, \boldsymbol{N}\}$ distributed according to $\boldsymbol{\pi}(\boldsymbol{x})$, where $\boldsymbol{N >> 1}$ and our integral turns into [11].

$$\boldsymbol{I} = \int \boldsymbol{g}(\boldsymbol{x})\boldsymbol{dx} = \int \boldsymbol{\pi}(\boldsymbol{x})\boldsymbol{f}(\boldsymbol{x})\boldsymbol{dx} \qquad \text{Equation 27}$$

Here our integral in equation 27 represents all possible states, hence we can calculate mean of all samples:

$$\boldsymbol{I_N} = \frac{\mathbf{1}}{\boldsymbol{N}}\sum_{\boldsymbol{i=1}}^{\boldsymbol{N}} \boldsymbol{f}(\boldsymbol{x^i}) \qquad \text{Equation 28}$$

And according to law of large numbers, $\boldsymbol{I_N}$ is very likely to approach $\boldsymbol{I}$ as $\boldsymbol{N}$ approaches infinity, provided that samples $\boldsymbol{x^i}$ are independent [11].

To represent the state as accurately as possible, ideally we would like draw samples from the distribution $\boldsymbol{\pi}(\boldsymbol{x})$**.** However, it is not always possible to know the distribution of interest. To work around this, we may draw samples from another distribution, $\boldsymbol{q}(\boldsymbol{x})$, which is similar to $\boldsymbol{\pi}(\boldsymbol{x})$. Appropriately weighted distribution $\boldsymbol{q}(\boldsymbol{x})$ makes the Monte Carlo estimation still possible. This part of the particle filter is called importance sampling, turning equation 27 into [11]:

$$\boldsymbol{I} = \int \boldsymbol{\pi}(\boldsymbol{x})\boldsymbol{f}(\boldsymbol{x})\boldsymbol{dx} = \int \boldsymbol{f}(\boldsymbol{x})\boldsymbol{q}(\boldsymbol{x})\frac{\boldsymbol{\pi}(\boldsymbol{x})}{\boldsymbol{q}(\boldsymbol{x})}\boldsymbol{dx} \qquad \text{Equation 29}$$

Now we do not draw samples directly from $\boldsymbol{\pi}(\boldsymbol{x})$, however resulting integral remains the same, we draw from the distribution $\boldsymbol{q}(\boldsymbol{x})$ and remainder of the integral represents aforementioned weights [11].

$$\widetilde{\boldsymbol{w}}(\boldsymbol{x^i}) = \frac{\boldsymbol{\pi}(\boldsymbol{x})}{\boldsymbol{q}(\boldsymbol{x})} \qquad \text{Equation 30}$$

Hence, we are able to calculate integral of interest in equation 28 which is a weighted sum [11]:

$$\boldsymbol{I_N} = \frac{\mathbf{1}}{\boldsymbol{N}}\sum_{\boldsymbol{i=1}}^{\boldsymbol{N}} \boldsymbol{f}(\boldsymbol{x^i})\,\widetilde{\boldsymbol{w}}(\boldsymbol{x^i}) \qquad \text{Equation 31}$$

Problem here is that normalizing factor of the distribution $\boldsymbol{\pi}(\boldsymbol{x})$ is likely to be unknown, therefore we need to perform normalization in order to evaluate $\boldsymbol{I_N}$ [11].

$$\boldsymbol{I_N} = \frac{\mathbf{1}}{\boldsymbol{N}}\sum_{\boldsymbol{i=1}}^{\boldsymbol{N}} \boldsymbol{f}(\boldsymbol{x^i})\,\widetilde{\boldsymbol{w}}(\boldsymbol{x^i}) = \frac{\frac{\mathbf{1}}{\boldsymbol{N}}\sum_{\boldsymbol{i=1}}^{\boldsymbol{N}} \boldsymbol{f}(\boldsymbol{x^i})\,\widetilde{\boldsymbol{w}}(\boldsymbol{x^i})}{\frac{\mathbf{1}}{\boldsymbol{N}}\sum_{\boldsymbol{j=1}}^{\boldsymbol{N}} \widetilde{\boldsymbol{w}}(\boldsymbol{x^j})} \qquad \text{Equation 32}$$

Also we shall normalize weights as follows [11]:

$$\boldsymbol{w}(\boldsymbol{x^i}) = \frac{\widetilde{\boldsymbol{w}}(\boldsymbol{x^i})}{\sum_{\boldsymbol{j=1}}^{\boldsymbol{N}} \widetilde{\boldsymbol{w}}(\boldsymbol{x^j})} \qquad \text{Equation 33}$$

Implement equations above recursively and end up with what is known as sequential importance sampling. The process helps us to approximate the true posterior density $\boldsymbol{p(x_k|Z_k)}$. **Figure 18** provide a meaningful visualisation on how the process goes [35].

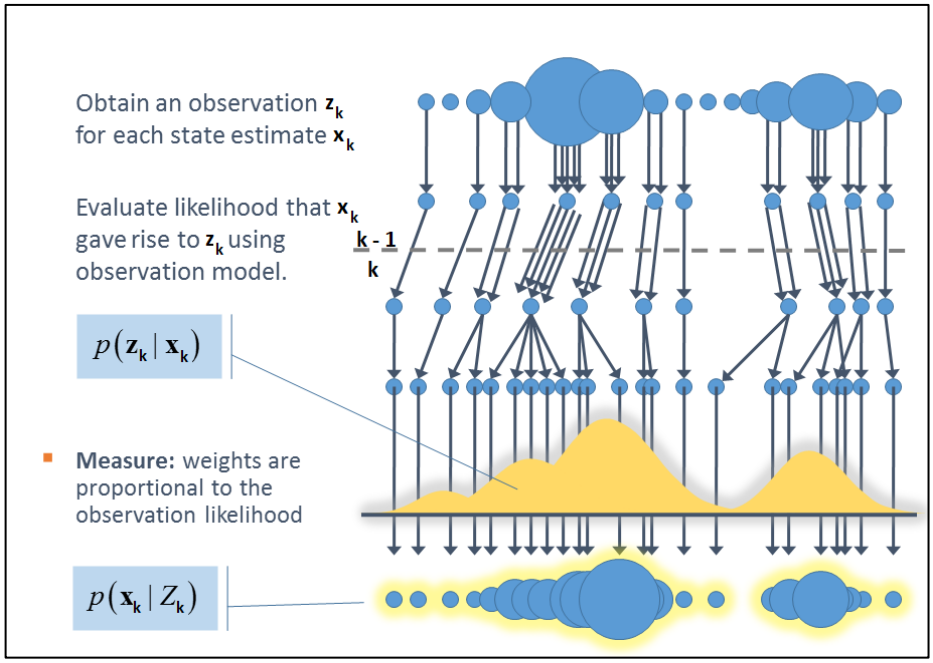


Figure 18. Visualisation behind the process of particle filter

For particle filter we have created a ***ParticleFilter*** class in Python and in the constructor ***__init__*** of this class we will call functions that create particles and assign weights to them. We will consider 480x360 image. If we have any initial knowledge about the location of human we are tracking, we will generate random particles around that spot, if we do not have any initial knowledge about initial position, we shall generate particles uniformly distributed throughout the picture. Initially, we shall assign every particle equal weight, as we have received no measurements and every randomly generated particle is as likely to represent true system as others [32]. **Figures 19**, **20** and **21**. Code is included in the **appendix**.

Here we are tracking position in x and y axes and since we are passively tracking people (no control input), we also need to track speed of the object, as we are going to need to make predictions in one way or another. However, generally as we increase number of dimensions, more particles are required to make an estimate, therefore we always trying to keep as small amount of dimensions as possible [32].

```
N = 5000 # Number of particles
initialPosition = np.array([100, 100]) # Initial known position, it can be any
# x and y, used only as example here
initialSpeed = np.array([20, 0]) # Initial known speed
initialStdDev = np.array([10, 10]) # Belief in position (Standard deviation)
initialConditions = np.array([initialPosition,
                              initialSpeed,
                              initialStdDev]) # Set of initial conditions

imageSize = np.array([480, 360])# Size of the image

# Creating instance of ParticleFilter object
pf = ParticleFilter(N, initialConditions, imageSize)
```

Figure 19. Initial conditions for the particle filter

```
    '''Particles 0 and 1 stand for position in x and y, and particles
    2 and 3 stand for velocity in x and y directions.'''

    def create_gaussian_particles(self):
        self.particles = np.empty((N, 4))
        self.particles[:, 0] = self.initialCoordinates[0] +
                               (np.random.randn(N) * self.initialStdDev[0])
        self.particles[:, 1] = self.initialCoordinates[1] +
                               (np.random.randn(N) * self.initialStdDev[1])
        self.particles[:, 2] = self.initialSpeed[0] +
                               (np.random.randn(N) * self.initialStdDev[0])
        self.particles[:, 3] = self.initialSpeed[1] +
                               (np.random.randn(N) * self.initialStdDev[1])

    def create_uniform_particles(self):
        self.particles = np.empty((N, 4))
        self.particles[:, 0] = np.random.uniform(0, self.imageSize[0], N)
        self.particles[:, 1] = np.random.uniform(0, self.imageSize[1], N)
        # -10 and 10 are trivial values for limits of speed particles
        self.particles[:, 2] = np.random.uniform(-10, 10, N)
        self.particles[:, 3] = np.random.uniform(-10, 10, N)
```

Figure 20. Generating random particles

```
    def create_weights(self):
        self.weights = np.ones(N) / N
```

Figure 21. Assigning weights for particles

After this, we are predicting next position of particles. We can use previous measurements to imply current speed, i.e., ***current speed = position in current frame – position in previous frame***. We are not going to move particles of speed themselves (except add noise), because to track changes in speed would require to track acceleration. Position particles are going to be moved amount that is equal to ***current speed***. Since we are tracking speed in pixels per frame and our ***dt*** is 1 frame, means that ***next position = current position + current speed*** and that is how we move particles (**figure 22**).

```
    def predict(self, speed, noise):
        self.particles[:, 0] += speed[0] + noise[0]
        self.particles[:, 1] += speed[1] + noise[1]
        self.particles[:, 2] += noise[2]
        self.particles[:, 3] += noise[3]
```

Figure 22. Predicting new position of particles

Next step is to incorporate measurements into our prediction. We calculate how far is every particle (in position domain and in speed domain) from measurement and use this information to assign new weights for particles using Bayes' theorem (equation 4). The Further the particle is away from measurement, the lower weight it gets. Old weight here was ***prior*** and newly generated weight is ***posterior***. Then weights are normalized. Implementation of the code is seen in **figure 23**.

Then we discard weights that are not very likely to represent the system through resampling step and create more copies of particles that represent system well. There are few resampling methods, which will be discussed later. Implementation of systematic resampling that is going to be used is shown in **figure 24**.

```
def update(self, measurements):
    # Find how far are distance particles from measurement
    distance_x = np.abs(self.particles[:, 0] -  measurements[0])
    distance_y = np.abs(self.particles[:, 1] -  measurements[1])
    distance = self.euclidean_distance(distance_x, distance_y)

    # Find how far are speed particles from measurements
    speed_x = np.abs(self.particles[:, 2] - measurements[2])
    speed_y = np.abs(self.particles[:, 3] - measurements[3])
    speed = self.euclidean_distance(speed_x, speed_y)

    # Combine them
    total_difference = distance + speed

    # If measurement is far away from particle, it means its weight is low
    new_weights = 1/total_difference

    # Implementing Bayes Theorem
    self.weights *= new_weights
    self.weights += 1.e-300      # avoid round-off to zero
    self.weights /= sum(self.weights)
```

Figure 23. Updating weights of particles

```
def resample(self):
    # Obtain indexes of particles we want to keep
    indexes = self.return_indexes()
    # Make new particles from those indexes
    self.resampled_particles(indexes)

def return_indexes(self):
    sampling_positions = (np.arange(N) + np.random.uniform(0, 1/N)) / N
    indexes = np.zeros(N, 'i')
    cumulative_sum = np.cumsum(self.weights)
    i, j = 0, 0
    while i < N:
        if sampling_positions[i] < cumulative_sum[j]:
            indexes[i] = j
            i += 1
        else:
            j += 1
    return indexes

def resampled_particles(self, indexes):
    resampled_weights = []
    resampled_particles = np.empty((N, 4))

    j = 0
    for i in indexes:
        resampled_weights.append(self.weights[i])
        resampled_particles[j, 0:4] = self.particles[i, 0:4]
        j += 1

    resampled_weights /= sum(resampled_weights)
    self.weights = resampled_weights
    self.particles = resampled_particles
```

Figure 24. Resampling step of the filter

Last piece that is left is to make an estimate using particles and weights. NumPy has a function that allows us to calculate a weighted average using weights. Code in **figure 25** returns mean and variance of weighted particles.

```
def estimate(self):
    particles_position = self.particles[:, 0:2]
    particles_speed = self.particles[:, 2:4]
    position_mean, position_var = self.mean_and_variance(particles_position)
    speed_mean, speed_var = self.mean_and_variance(particles_speed)
    return position_mean, position_var, speed_mean, speed_var

def mean_and_variance(self, the_array):
    mean = np.average(the_array, weights=self.weights, axis=0)
    var = np.average((the_array - mean)**2, weights=self.weights, axis=0)
    return mean, var
```

Figure 25. Making an estimate from particles

There are many more things to take account when designing a particle filter, some of which will be discussed later, but this is basic principle behind particle filter that is going to be implemented. The filter is passed through a loop and resulting particles look like in **figure 26**.

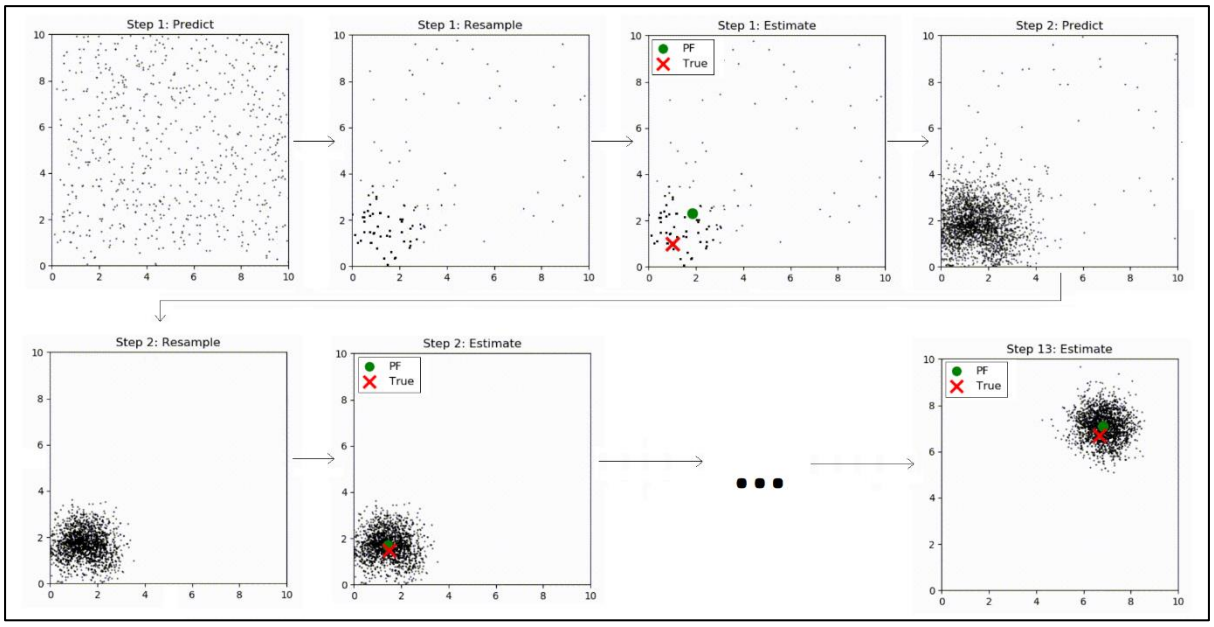


Figure 26. Few cycles of a particle filter simulation

Using this information, particle filter has been implemented to track objects in two previously used videos. Results of this are shown in **figures 27** and **28** and the entire code is left out in the **appendix**. For both cases we used 5000 particles for each state variable and initial conditions were set equal to first detection. As for Kalman filter, we were able to track state variables that are not directly measured, such as speed.

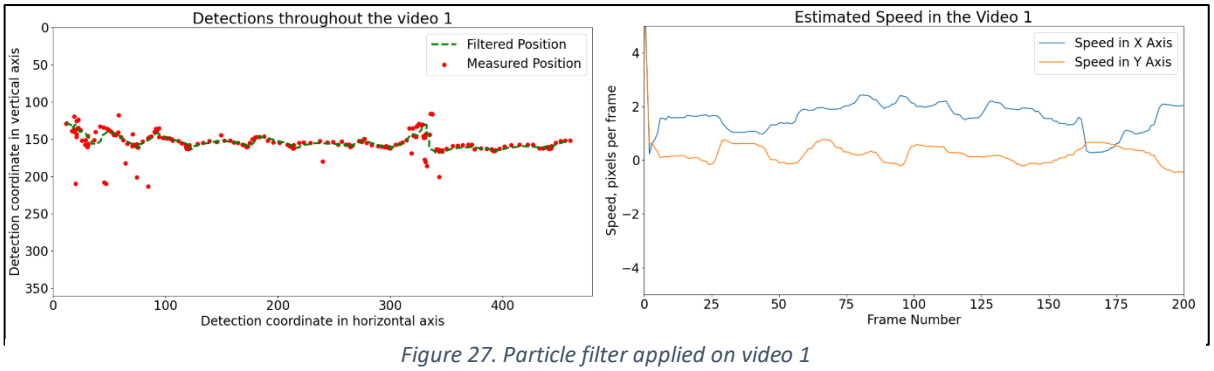


Figure 27. Particle filter applied on video 1

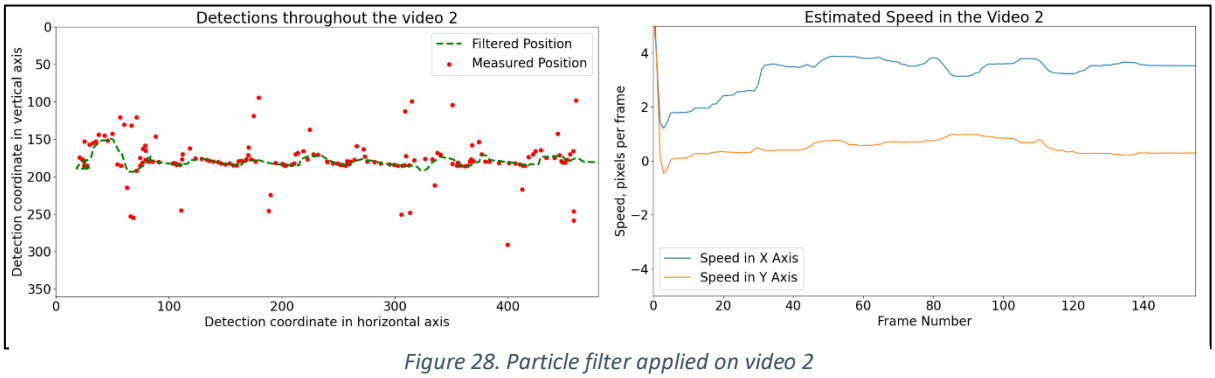


Figure 28. Particle filter applied on video 2

5. Discussion

5.1. Design of Kalman and Extended Kalman Filters

Cases that we saw in simulations are 'textbook' examples, that are easy to define and implement. Usually it is not the case and we need to take many things into account. For instance, we could apply Kalman filter for measurements obtained in **figure 10**, even though, the measurement noise was not normally distributed and it might produce seemingly good results. However, it has been proven that for optimal estimate our system has to satisfy previously mentioned conditions. Therefore, it is important to design filter carefully and not lie to it by providing false parameters, who seem to make filter work nicely, even though it does not produce optimal estimate. If filter produces good results in during the span of observation, it does not mean that it will not diverge after a longer period of time [11][32].

Parameter withing the Kalman filter that directly impacts state output $\hat{\boldsymbol{x}}_{\boldsymbol{k}|\boldsymbol{k}}$ is Kalman gain $\boldsymbol{K_k}$ (equation 12). We can think of it as a ratio, defining how much we trust prediction versus measurement. Equation 11 was given for calculation of $\boldsymbol{K_k}$ and it uses $\boldsymbol{P_k}$ which accounts for uncertainty and $\boldsymbol{S_k^{-1}}$ which accounts for uncertainty [32]. Note that although division for matrices is not defined, it is useful to think of it of matrix inverse operation as finding reciprocal:

$$\boldsymbol{K_k \approx \frac{P_{k|k-1}}{S_k} H_k^T \approx \frac{Uncertainty_{PREDICTION}}{Uncertainty_{MEASUREMENT}} H_k^T} \qquad \text{Equation 34}$$

$\boldsymbol{K_k}$ is a ratio between 0 and 1, for example 0.9 means that we take 90% of prediction and 10% of measurement for our new estimate. Note that in **figure 15** the produced estimate was a pretty straight line, because in that case, we have set measurement variance to be $\mathbf{2^2}$ and speed variance $\mathbf{0.25^2}$, which means that measurement noise is way higher compared to process noise, i.e. we are telling our filter, that measurements are not very precise, so we should mostly ignore them and trust our system model.

In simulation shown in **figure 29** we have increased speed variance to $\mathbf{2^2}$, telling filter that our measurements are not that bad compared to process model, which produces a result that is not such

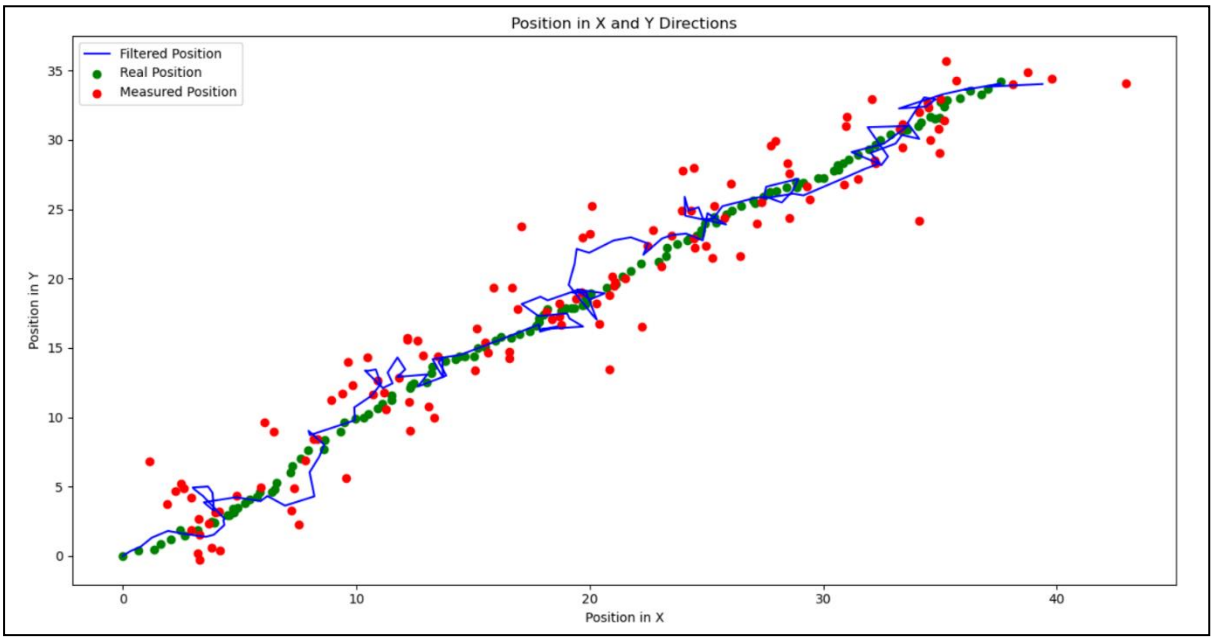


Figure 29. Kalman filter simulation with higher speed variance

a straight line anymore [32].

Another thing that is worth considering is an initial estimate. In previous simulations, our estimate matched the actual behaviour. Initial speed estimate in x and y directions was to be defined to be equal to **3** and initial position estimate – **0**. This have been changed to **10** and **7** respectively (**figure 30**).

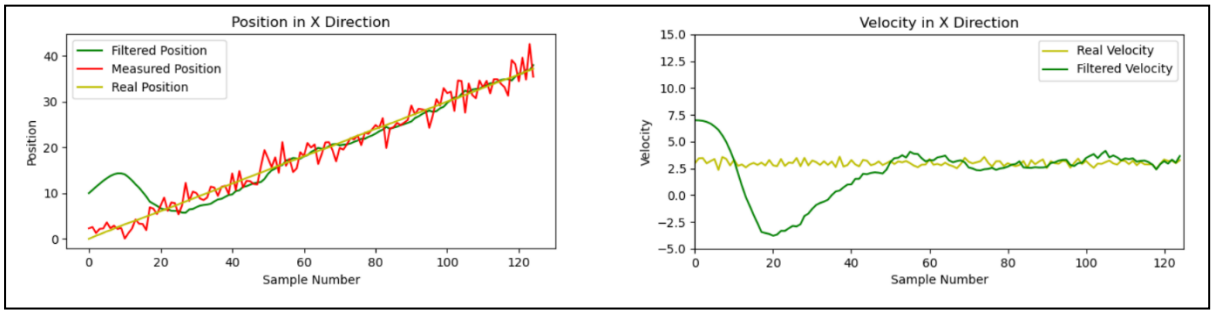


Figure 30. Kalman filter simulation with poor initial estimate.

Although filter does recover after some time, it is not ideal, as we might need to track object as effectively as possible for the entire duration of observation. That is why good initial estimate is essential. For human tracking, we know that average person walks at about 5 km/h, therefore it is relatively easy to make a reasonable initial estimate.

Extended Kalman filters, in the meantime, are a little bit more complicated. As they are just an extension of Kalman filter, same principles from above apply, however, difference here being the linearization of functions. It is worth noting, that this filter is not optimal and is as good as linearization is. If non-linearities are too high, this method also deems to be useless, as in **figure 31**, where amplitude and frequency of oscillations have been increased 5 times, filter cannot adapt to changes fast enough – linearization is not good enough [10].

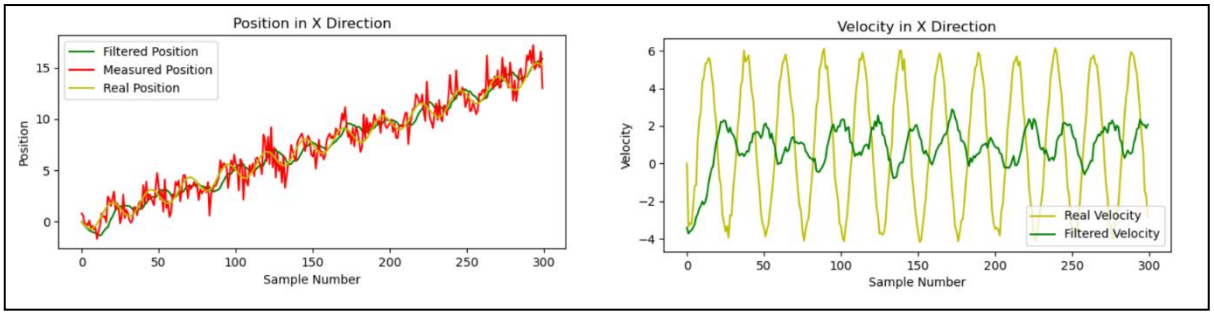


Figure 31. Case where extended Kalman filter turns out to be useless

5.2. Features of Kalman Filter

Kalman filters are able to associate object to its tracks and track it even if the measurements are missing. This requires as accurate system model as possible, because this is the way estimates are made. To make sure of this, in our simulation, in short periods of time we get rid of measurements and result of experiment is shown in **figure 32**.

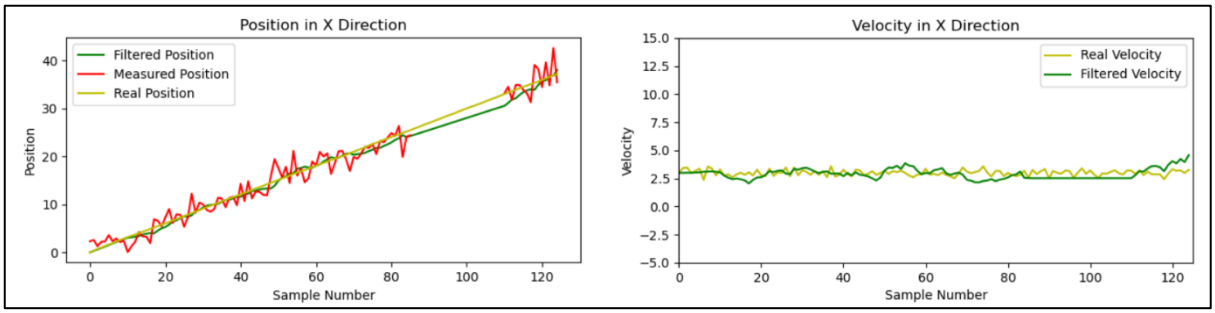


Figure 32. Kalman filter works even when measurements are absent

However, if during the time measurements are missing and system is behaving in some way that is not anticipated by the model, it will not be possible to make a good estimate (**figure 33**) [36].

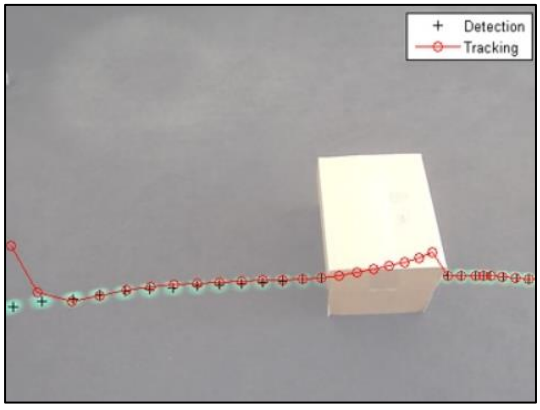


Figure 33. When measurements are not available, we can only trust the system model

It also worth noting that Kalman filter is able to make estimates of state variables that are not being measured directly, as seen above, we do not need to obtain speed measurements to make an estimate. However, since the system model is the only piece of information we are using to make this prediction, it will not be as accurate, i.e. we do not have anything additional to reinforce our prediction as it is the case for position tracking. Note that combining measurements and predictions of the system model makes our belief in the current state gets stronger (**figure 14**). Even though information from separate sources might not be too accurate, when everything is combined, estimate is very decent. Ideally, the more information, the better estimate is.

5.3. Design of Particle Filter

Although particle filters deal well with most of the systems, it has its own problems. For filter to work well, we must generate as many particles that represent system as accurately. As mentioned, if initial position is known, we might generate particles that are around that position. However, we should not generate particles too densely around initial position, as we might miss non-linearities. As for unknown initial conditions, we would like more particles, because when they are randomly generated over large area (entire picture), it is likely that none of the particles will be good enough representatives of the system [32]. **Figure 34** shows how could distribution of the particles look like for both cases.

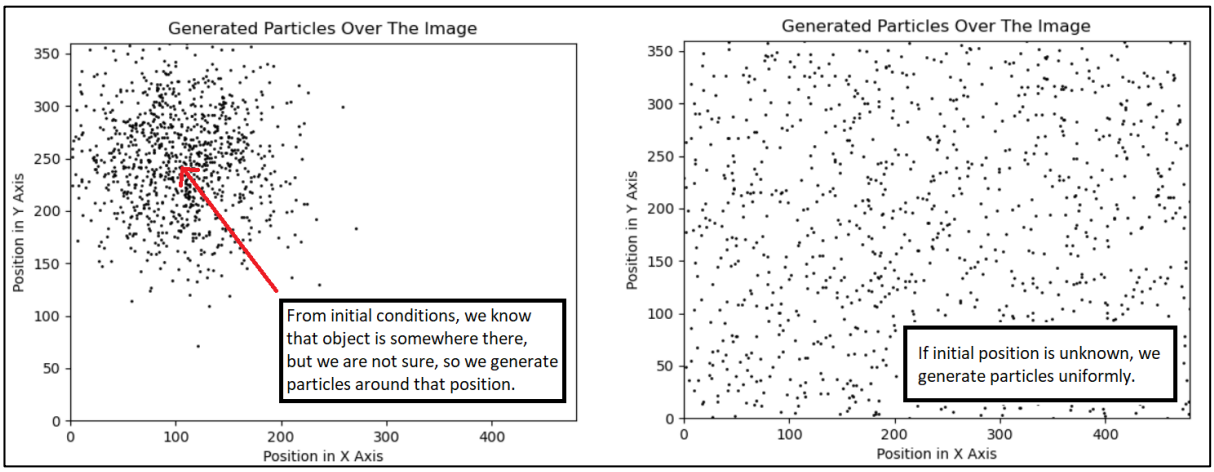
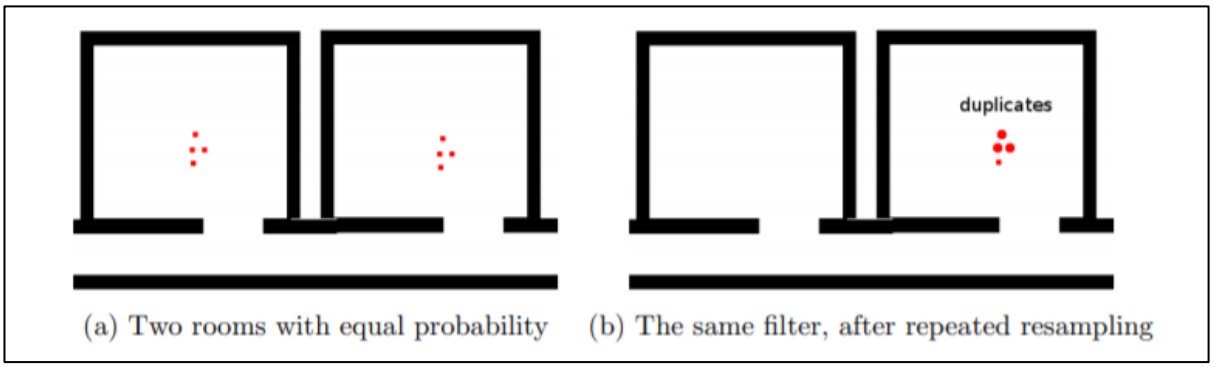


Figure 34. Generated particles over the image

Poor sampling results in lack of diversity. Over a period of time, especially, when measurements are uninformative, particles tend to congregate [11][32][37]. Consider following:

If we do not gain any new information, after a few resampling steps, the particle filter will converge onto one of the rooms. This happens, because particle filters are non-deterministic, at resampling step, one room is more likely to gain particles than other and filter will eventually converge. Once a state loses particles, it is hard to get them back, if no information is reinforcing possibility in this state, zero particles would mean zero probability, even if it may not be the case. There are few ways of



overcoming such a problem. We might add more particles, as it will take more time to congregate onto one state. Usually useful, however, comes at a computational cost. Also, we can skip the resampling step if we are not receiving any information [32][37]. For this, we might consider such metrics such as entropy or $\mathbf{max}(\boldsymbol{w}_k^i)\,/\boldsymbol{min}(\boldsymbol{w}_k^i)$ [37].

Also, we might use different sampling techniques. More complicated ones tend to generate better results. More common ones would be [32]:

- Multinomial resampling. Compute the cumulative sum of normalized weights at every step of the weight array. This would provide us with new array of values that increase from 0 to 1. We generate a random number selected from uniform distribution starting at 0 and ending at 1. Use binary search to find its position within the array. Large weights occupy more space than small ones, meaning they are more likely to be selected.
- Residual resampling. Very efficient. Normalized weights are multiplied by ***N*** and the integer value of these weights are chosen to calculate how many samples of that particle will be taken. Let us have particle whose weight is equal to 0.003 and ***N*** equal to 2000. In this case we are guaranteed to have 6 new particles representing this particle. However, if weight of the particle is equal to 0.0003, then 0.0003 × 2000 = 0.6 (round down to 0), meaning that we would discard this particle, since it carries no significant weight.
- Stratified resampling. Attempts to select new particles uniformly across old particles. Divides cumulative sum of all weights into ***N*** equal sections and selects one randomly out of every section.
- Systematic resampling. Again, divide cumulative sum of all weights into ***N*** equal sections, choose a random offset for all of those divisions and sample, ensuring ***1/N*** distance between all weights.

See all of the resampling techniques in **figure 36**. Coloured bars represent the cumulative sum, black bars indicate N equal subdivisions and black dots are selected new particles.

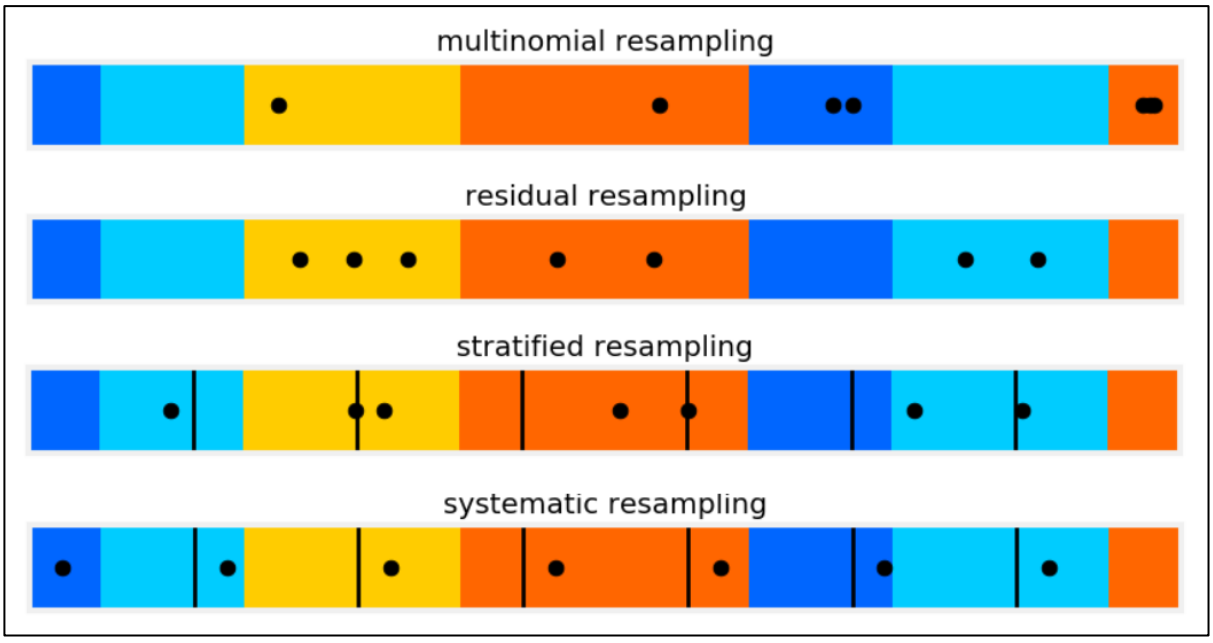


Figure 36. Illustration of different resampling techniques

However, we do not need to resample at every step. If, for instance, we have received no measurements there is no information from which resampling could benefit. A metric $\widehat{N}_{eff}$ which approximates the number of particles that provide meaningful contribution to probability distribution. Equation for it is given as follows [11]:

$$\widehat{N}_{eff} = \frac{1}{\sum_{i=1}^{N}(w_k^i)^2} \qquad \text{Equation 35}$$

Small $\widehat{N}_{eff}$ indicates a severe sample degeneracy and high value indicates that particles are doing fine. Knowing that particle filter implementation is computationally expensive it would be good to skip some steps if they are not necessary, hence we implement code that performs resampling only when $\widehat{N}_{eff}$ falls below a certain threshold value. Good starting point is ***N/2***, however, it varies for different applications. Failure to have enough adequate samples may lead to inaccurate estimates and system may diverge very quickly [11][32].

```
def neff(self):
    return 1 / np.sum(np.square(self.weights))

if pf.neff() < pf.N/2:
        pf.resample()
```

Figure 37. Implementing resampling only when there are too few effective particles

Also, in order to have enough adequate samples, knowledge of initial conditions is also very important. The further away initial particles are generated from true state, the worse the results are. If particles are too far away from true state, particles may not be able to find true state at all, because there will be no particles generated close enough to the true state, such as in **figure 38**.

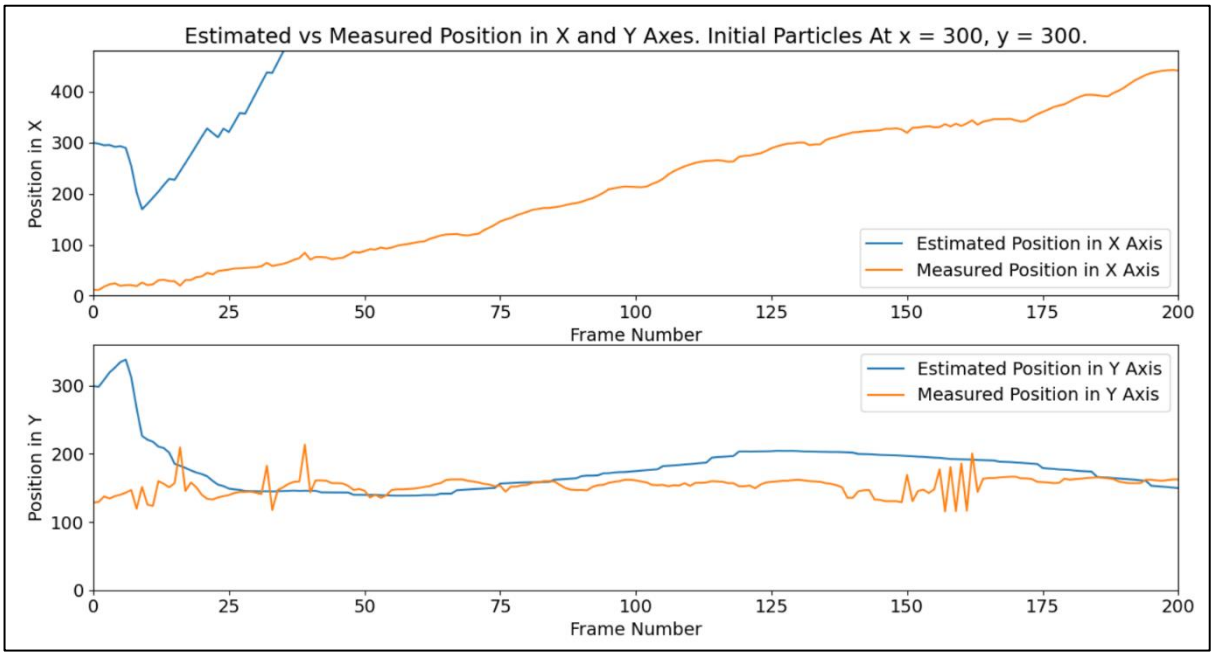


Figure 38. Implementing particle filter with bad initial conditions

Results here were very poor. If we do not know anything about initial conditions, we may generate particles that are distributed uniformly across the image. In this way we guarantee that some of particles will end up near true state of the system. However, it will take longer time for system to converge around the true state as seen in **figure 39**.

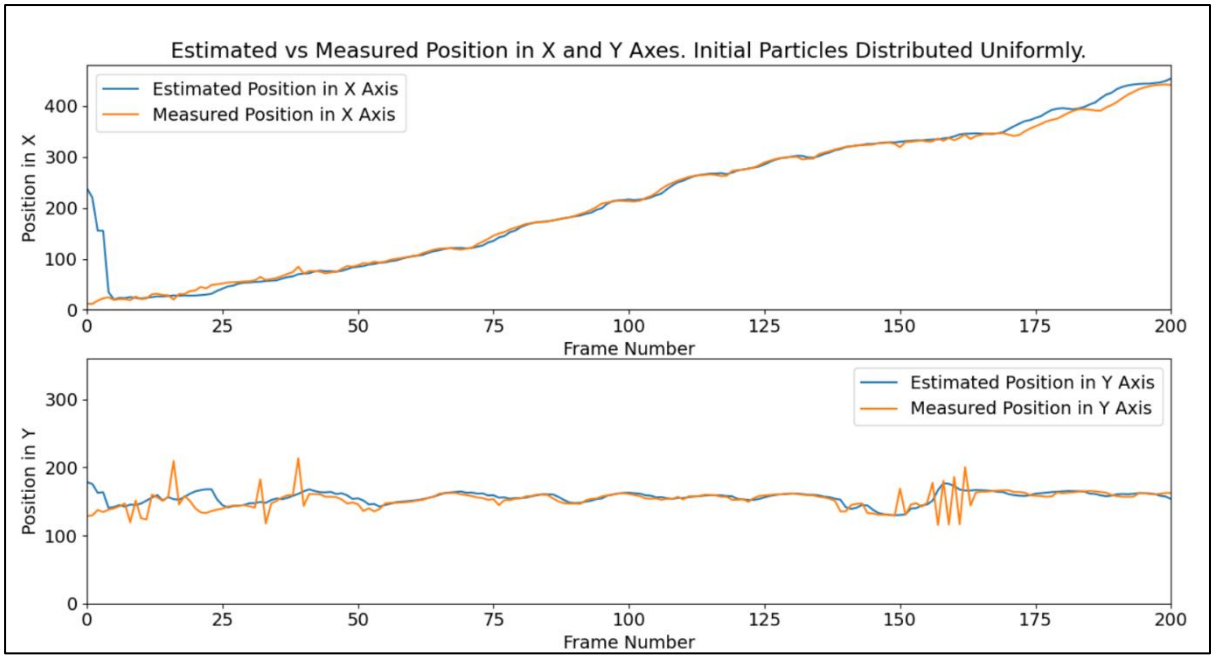


Figure 39. Implementing particle filter with uniformly distributed particles.

Here we see that in x axis is took about 5 epochs of particle filter to converge to true state of the system. In our case, where video is running at 30 frames per second, means that it took approximately 0.17 seconds for particle filter to find the target. Whether this is good or not, mostly depends on the application. If you are trying to implement particle filter that is used by self-driving vehicle that needs to detect people, this amount of time is not something you would want. That is why in computer vision systems that are used by vehicles, people also implement methods that anticipate areas where people show up from (determine initial conditions) [7]. For y axis results appear to be better, but that is only the case because mean of uniformly distributed particles ended up being almost the same as true state. Since our image was 480x360, uniformly generated particles will end up having initial mean of x = 240 and y = 180, which was lucky enough for system to be able to converge to a true value relatively fast, however this will not always be true.

5.4. Performance Evaluation

5.4.1. Kalman and Extended Kalman Filters

In these type of filters, there are ways to analytically evaluate performance of the filter. We may vary different constants and variables and observe what impact does it have, however, we would like to be able to quantify that. If the conditions for Kalman filter are met it is considered an optimal filter and its estimation errors will have following properties [10]:

- Mean of the estimation error will be zero over time.
- Covariance of estimation error is the covariance matrix $\boldsymbol{P_k}$.

Using this method to evaluate performance is not always possible, because we need to know the ground truth [32]. In our simulations we have access to it meaning, we can evaluate performance of our filter. Knowing that in Gaussian distributed random variables, 95% of values fall within two standard deviations, we can check for this and if it is not the case in our filter, it means, that something may be wrong.

```
good_counts = 0 # amount of times value falls withing 2 standard deviations
i = 0 # total amount of estimations

"""For every mean and covariance estimate compare estimate +- 2 standard
deviations against the real value, if it falls withing, then it is good
estimate."""

for mean, cov in zip(means,covs):
    if mean[0] - 2*np.sqrt(cov[0,0]) < real_position_x[i] and \
       mean[0] + 2*np.sqrt(cov[0,0]) > real_position_y[i]:
        good_counts += 1
    i += 1

good_percentage = good_counts / i

# Repeat for every estimated state after this
```

Figure 40. Estimating performance of the Kalman filter

Tables below shows the results of implemented code above for every state estimate. Amount of time steps for the experiment: 1000. Experiment was repeated 10 times and the average was taken. Ratios that are way lower than 0.95 mean that filter is not implemented correctly and may be diverging from true value.

Kalman Filter Simulation	
State Variable	**Ratio of Estimates That Fall Within 2 Standard Deviations**
X Position	0.964
Y Position	0.969
X Speed	0.988
Y Speed	0.991
Extended Kalman Filter Simulation	
State Variable	**Ratio of Estimates That Fall Within 2 Standard Deviations**
X Position	0.952
Y Position	0.941
X Speed	0.976
Y Speed	0.978

Table 1. Evaluating performance of filters within simulations

Also note that Kalman filters are computationally very effective. Since there are no loops within the code (except the main loop itself), program runtime is very low, because all we need to do is to implement equations that perform matrix addition, multiplication, subtraction and sometimes inversion. Kalman filter we implemented for simulation was able to run 100000 iterations in 15 seconds and 31 seconds for extended Kalman filter. Latter obviously runs slower, because of the need to linearize function every time step by finding derivatives, which complicates things a little bit.

5.4.2. Particle Filters

In our video-based human tracking cases there is a small problem that we do not have 'ground truth' and we would like to be able to compare true and estimated values. Even then, it is difficult to agree on what this 'ground truth' would be since one might be using human head to track the person, other may be tracking entire body.

One thing we know that the more particles we have, the better estimate is. We have varied amount of particles that are used for tracking from 100 to 100000 and results are shown in **figure 41**. We may find that estimates are way off for 100 and 1000 particles. For 10000 estimate is pretty good and there is no much of a difference between having 10000 and 100000 particles, hence we may assume that former choice would be safe enough to obtain a decent estimate.

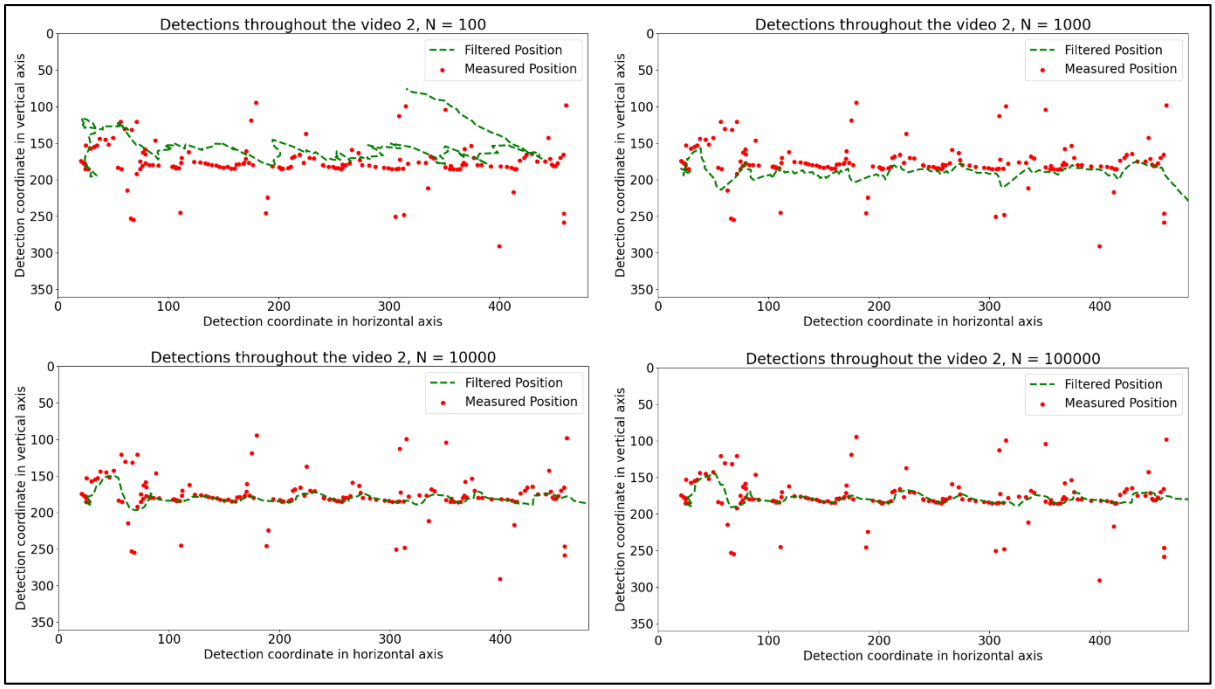


Figure 41. Effect of varying number of particles for a filter

However, we know that larger amount of particles mean larger computational burden, but in certain scenarios we want our estimates to be extremely precise as well. To maximize performance, state of the art algorithms are able to dynamically adjust number of particles, for example, one of the methods is based on a sequential comparison between the observations and predictive pdfs [38].

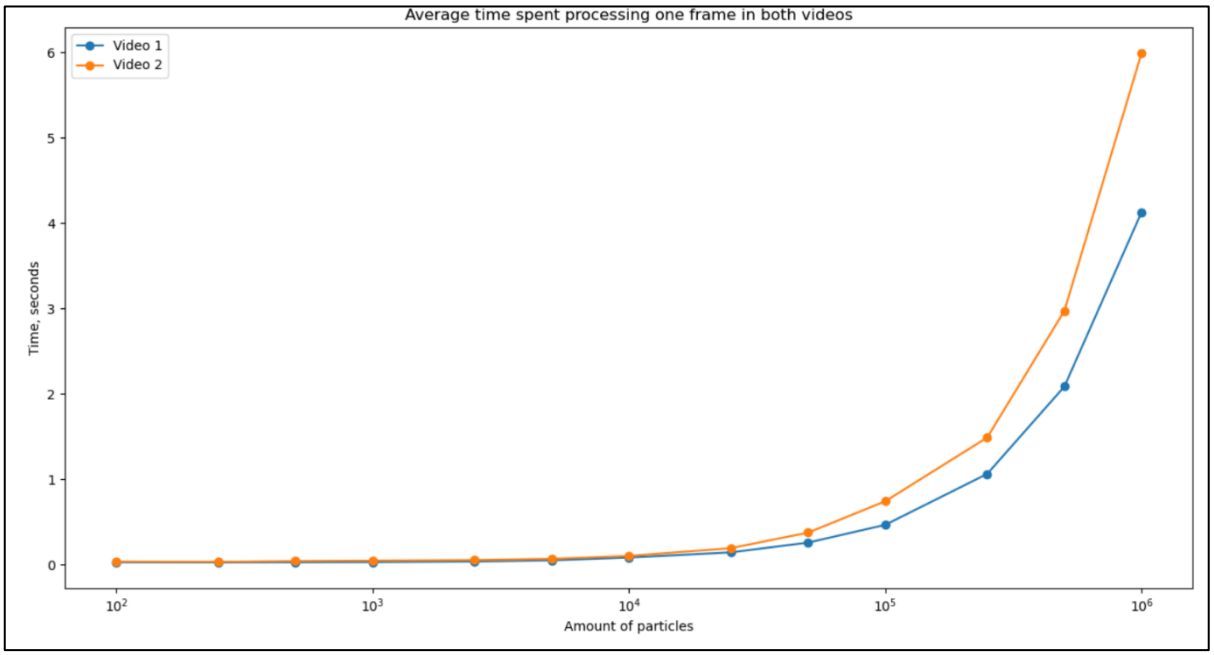


Figure 42. Average time spent processing one frame for given number of particles

We have tracked for both videos how number of particles affect runtime of the program. Results are shown **figure 42**. It is worth noting that frame processing time for second video may be higher due to several factors. Firstly, we may be performing resampling more often on one video compared to another. Remember, that we do not always resample, but only when number of effective particles is below the threshold. In addition to this, timing may not be consistent due to background tasks performed by the computer during the time experiment was conducted. Lastly, size of the image is also a factor. Here we used 480x360 resolution, however, by the current standards, this is pretty low and for better results we would consider image of higher resolution which would significantly increase the processing time. It is worth noting that for our both cases we used videos running at 30 frames per second, or 0.033 seconds for one frame. If we wanted to run the particle filter real-time on a video, we would not be able to use more than 2500 particles, because one frame processing time after that becomes greater than 0.033 seconds.

However, not only number of particles amount the runtime of the filter. Time required for particle filter increases exponentially with number of the states tracked. Imagine an example, where we are trying to estimate ***position x***. If we are intending to use 100 particles and probability of randomly generated particle to have true state is equal to 0.01 then we would expect that one particle would represent true state. Add another variable ***position y*** that would also have 0.01 probability that randomly generated particle represents true state. Here, probability to generate a particle that has a true state of ***position x*** and ***position y*** would be 0.01 × 0.01 = 0.0001, meaning that now in order to generate at least one particle that represents true state we would be expected to use 10000 particles. This is exponential growth and it increases computational complexity very fast. Some may think that tracking, for example, acceleration would make estimates better. However, higher order estimates are not always better and cost of including another two variables that need to be tracked would greatly exceed the accuracy of the estimate (or not at all) [32].

5.5. Other Considerations

5.5.1. Data From Multiple Sources

Multiple data sources are always better even if some turns out to be worse than the others. Consider an example where we have sensor A and sensor B trying to obtain the position of an object. Sensor A detects object at position of 160 meters that is accurate within 1 meter and sensor B detects same object at position of 170 meters that is accurate with 9 meters. Using this information, we can deduce that object is at position of 161 meters since this is the only position that would be possible (**figure 43**) [32].

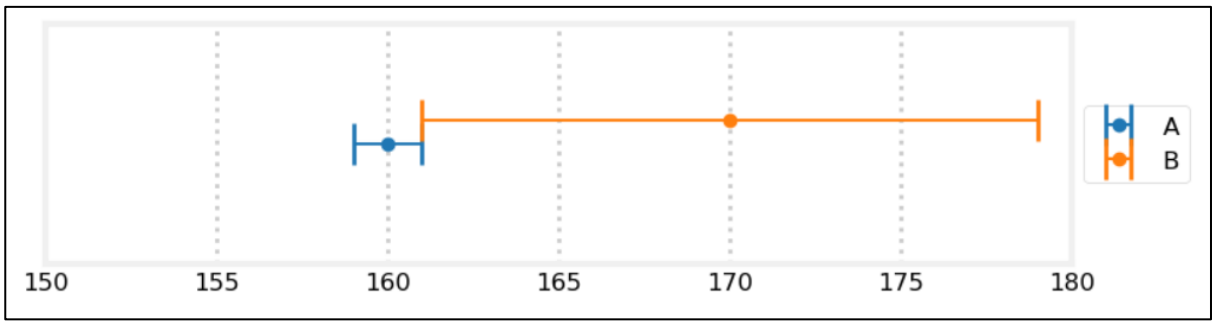


Figure 43. Example of using multiple measurements to detect true position

Kalman filter gives us a convenient way to fuse together measurements from multiple sources through the measurement function $\boldsymbol{H_k}$. If we are tracking state $\boldsymbol{x_k} = [\boldsymbol{p_k}, \boldsymbol{v_k}]^\mathsf{T}$ and we have a measurement vector $\boldsymbol{z_k} = [\boldsymbol{z_1}, \boldsymbol{z_2}]^\mathsf{T}$ where both of the variables attempt to measure position of the same object. Then we may design $\boldsymbol{H_k}$ as follows:

$$\boldsymbol{H_k} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \qquad \textit{Equation 36}$$

In this case equations 9-13 incorporate measurements from multiple sensor and combine their results to obtain the best estimate.

However, we may have to establish confidence in our sensors. This is done through the measurement noise covariance matrix $\boldsymbol{R_k}$:

$$\boldsymbol{R_k} = \begin{bmatrix} \sigma^2_{z_1 z_1} & \sigma^2_{z_2 z_1} \\ \sigma^2_{z_1 z_2} & \sigma^2_{z_2 z_2} \end{bmatrix} = \begin{bmatrix} Variance\ Sensor\ 1 & 0 \\ 0 & Variance\ Sensor\ 2 \end{bmatrix} \qquad \textit{Equation 37}$$

Here it is useful to think of these variances as ratio of one another. If $\frac{Variance\ of\ sensor\ 1}{Variance\ of\ sensor\ 2}$ is very high it means that we consider sensor 2 to be more accurate, meaning that deviation in sensor 1 will be taken seriously by an estimator only if it is supported by data from sensor 2. Kalman filter equations helps to find the best combination to take combine all measurements and all predictions. Again, there is no bad information, unless it is completely rubbish.

In a very same way we shall combine measurements for the particle filter to get the best results. If we may think that measurement one is five times better than measurement two, then in function ***update*** where we incorporate measurements to obtain new weights for the particles we may define exactly that:

```
# Combine both measurements to find how far they are from true state
difference_from_both_measurements = 5 * measurement1 + \
                                    measurement2

# If measurement is far away from particle, it means its weight is low
new_weights = 1 / difference_from_both_measurements

# Implementing Bayes Theorem
self.weights *= new_weights
self.weights += 1.e-50      # avoid round-off to zero
self.weights /= sum(self.weights)
```

Figure 44. Combining multiple measurements for particle filter

5.5.2. Data Association and Gating

These two components of object tracking system are created with an objective to identify origin of the measurements. In video 1 we had the case where for a brief period of time shadow would be also identified as a human.

Figure 45. Case of a false detection in video 1

In filters we usually have some measure that establishes confidence level in our estimates. Using this information we may determine whether received new measurements are accurate enough and reject poor measurements. This process is called gating. In our example we have simply overcome the problem 'manually', because we knew that this was a false positive and we knew the location of it so we could just ignore it. However, this will not always be the case. When we are tracking object, statistics are telling us that 99% of the measurements will not be more than three standard deviations away from the state estimate, hence, this can be used identify measurements that are not the object we are trying to track. For good results we may want to calculate something called Mahalanobis distance which is given by equation below [39]:

$$D_m = \sqrt{(x-\mu)^T S^{-1}(x-\mu)} \qquad \textit{Equation 38}$$

This Mahalanobis distance basically quantifies how many standard deviations away from the mean is the given value. You would apply such a gate for the measurement, if its distance is larger than 3 standard deviations it would be rejected. Problem here, however, is that Mahalanobis distance may increase computational time. In this case we might introduce rectangular gates that are less effective but is also requires computational effort and easier to implement. For middle of the road solution we might want to use double gates: rectangular for rejecting measurements that are obviously bad, if these gates are passed then we calculate Mahalanobis distance to confirm the quality of measurement [32]. In case below, orange X measurement would be rejected. Once origin of the measurement is confirmed, it may be subjected to track management techniques. Here we are concerned with multiple objects, where we are identifying each one of them and assigning a track to it [11] which may be concern of Joint Probability Data Association Filters (JPDAF) [40] or any type of PHD filter.

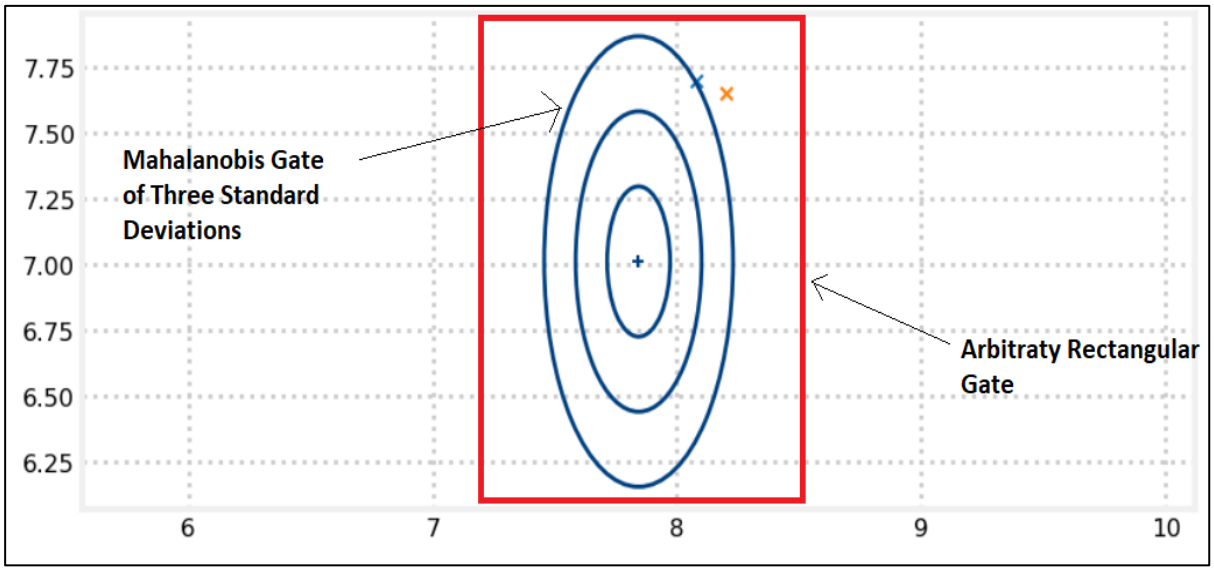


Figure 46. Example of implemented gates to reject bad measurements

6. Conclusions

We shall conclude that aims and objectives of this project were met successfully. We have briefly reviewed different kinds of techniques that are used for object tracking: alpha-beta filters, unscented Kalman filters and PHD filters. A concept of background subtraction was introduced in order to perform detection of a human on the screen. Although this is not the most reliable method to detect people, since background subtraction can detect any moving object, this was good enough for our cause, because this was not the main focus of the project.

We have implemented three different techniques that may be used in order to track a person in a video sequence: Kalman filter, extended Kalman filter and particle filter. Recursive Bayesian state estimation framework was introduced which is the basis for all the aforementioned filters. Since the object detection method we used provided results whose noise is not drawn from Gaussian distribution, we implemented first two filters for simulations only. Both filters have yielded great results as state of the system was tracked very accurately. We were able to disregard noise and make future estimates even in the absence of measurements. Extended Kalman filter was able to deal with non-linearities which is very important for majority of applications.

However, since both of these filters have limitations, our attention shifted towards particle filter which is able to deal with systems that are non-linear and whose noise is non-Gaussian. Particle filter was successfully implemented on two video sequences and it was used to track people within. We were able to track position and speed in x and y directions with high precision and even higher order estimates. The filter was tested with varying initial conditions and varying number of particles. We went through some aspects of the design of particle filters, such as initialization and resampling and demonstrated how it is important to select correct parameters, because otherwise filter may be deemed ineffective of inefficient.

In the end we have discussed how accuracy of tracking may be improved by introducing multiple sources of information and implementing gating and data association techniques. As mentioned, all of these filters are rarely used on their own and they are just founding blocks for more complex tracking systems that take into account many variables such as amount of state variables, noise, linearity and others. Using finite set statistics, any of these filters may be expanded into a PHD filter that is capable of tracking multiple and variable number of targets and this may be used under most circumstances and take into consideration various factors that may disturb smooth object tracking such as poor weather conditions, bad lighting, low framerate, shadows, occlusions and etc. Even more complicated methods may deal with cases where camera is non-static or turbulence is observed.

Every step of object tracking process has its intricate details and would deserve a book on its own, therefore the project was just a familiarization with techniques that lie behind the process and the goal was achieved without a doubt.

7. References

[1] Barrett, D., 2020. *One Surveillance Camera For Every 11 People In Britain, Says CCTV Survey*. [online] Telegraph.co.uk. Available at: <https://www.telegraph.co.uk/technology/10172298/One-surveillance-camera-for-every-11-people-in-Britain-says-CCTV-survey.html> [Accessed: 9 May 2020].

[2] “Multimodal Surveillance – Intelligent Sensing”, Intelsensing.com, 2020 [online]. Available at: <http://www.intelsensing.com/research/information-processing/multimodal-surveillance/> [Accessed: 9 May 2020].

[3] I.stack.imgur.com, 2020 [online]. Available: https://i.stack.imgur.com/v9zhy.jpg [Accessed: 9 May 2020].

[4] "Artificial intelligence-based security camera introduced by Umbo Computer Vision", *Vision Systems Design*, 2020. [Online]. Available: https://www.vision-systems.com/boards-software/article/16752078/artificial-intelligencebased-security-camera-introduced-by-umbo-computer-vision. [Accessed: 09 May 2020].

[5] Murrison M., "AEye unveils iDAR advanced perception and planning for driverless cars | Internet of Business", *Internet of Business*, 2020. [Online]. Available: https://internetofbusiness.com/aeye-unveils-idar-advanced-perception-driverless-cars/. [Accessed: 09 May 2020].

[6] Gao J., Yang Y., Lin P., Park D.S., “Computer Vision in Healthcare Applications”, Journal of Healthcare Engineering, Hindawi, Volume 2018.

[7] Janai J., Fatma G., Aseem B., Geiger A., “Computer Vision for Autonomous Vehicles: Problems, Datasets and State of the Art”, Autonomous Vision Group, MPI for Intelligent Systems, University of Tubingen, Germany, Koc University, Turkey, 2019.

[8] Socher R., Huval B., Bhat B., Manning C. D., Ng A. Y., “Convolutional-Recursive Deep Learning for 3D Object Classification”, Computer Science Departament, Stanford University, 2012.

[9] Zhang L., Li Y., Nevatia R., “Global Data Association for Multi-Object Tracking Using Network Flows”, Institute of Robotics and Intelligent Systems, University of Sourthern California, 2008.

[10] Kay S., “*Fundamentals of statistical signal processing”*, Volume 1, Englewood Cliffs, NJ: Prentice-Hall, 1993.

[11] Ristic B., Arulampalam S., Gordon N., “*Beyond the Kalman filter. Particle Filters for Tracking Applications”*. Boston, Mass.: Artech House, 2004.

[12] Blekhman A., “An Intuitive Introduction to Kalman Filter”, [online] Available: <https://www.mathworks.com/matlabcentral/fileexchange/13479-an-intuitive-introduction-to-kalman-filter>, [Accessed: 09 May 2020]

[13] Viola P., Jones M., “Rapid Object Detection Using a Boosted Cascade of Simple Features”, Mitsubishi Electric Research Laboratory, 2004.

[14] "OpenCV: Cascade Classifier", *Docs.opencv.org*, 2020. [Online]. Available: https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html. [Accessed: 09 May 2020].

[15] Dalal N., Triggs B., “Histograms of Oriented Gradients for Human Detection”. International Conference on Computer Vision & Pattern Recognition (CVPR ’05), Jun 2005, San Diego, United States. pp.886–893, ff10.1109/CVPR.2005.177ff, ffinria-00548512f.

[16] Ren S., He K., Girshick R., Sun J., “Faster R-CNN: Towards Real-Time Object Detection with Regional Proposal Networks”, Microsoft Research, 2015.

[17] Redmon J., Divvala S., Girshick R., Farhadi A., “You Only Look Once: Unified, Real-Time Object Detection”, University of Washington, Allen Institute for AI, Facebook AI Research, 2016.

[18] Yilmaz, A., Javed, O., and Shah, M., “Object tracking: A survey”. ACM Comput. Surv. 38, 4, Article 13, 2006

[19] Penoyer R., “The Alpha-Beta filter”, C User Journal, 1993.

[20] Yi Cao (2020). Learning the Unscented Kalman Filter [online] Available: <https://www.mathworks.com/matlabcentral/fileexchange/18217-learning-the-unscented-kalman-filter>, MATLAB Central File Exchange. [Accessed: 09 May 2020].

[21] Wan A. E., van der Merve R., “The Unscented Kalman Filter for Nonlinear Estimation”, Oregon Graduate Institute of Science and Technology, 2014.

[22] Si W., Wang L., Qu Z., “Multi-Target State Extraction for the SMC-PHD filter”, Harbin Engineering University, China, 2016.

[23] Vo B.-N., Ma W.-K., “The Gaussian Mixture Probability Hypothesis Density Filter”, IEEE Transactions of Signal Processing, Vol. 54, No. 11, 2006

[24] B.-N. Vo, S. Singh and D. Arnaud, "Sequential Monte Carlo methods for multitarget filtering with random finite sets", *IEEE Trans. Aerosp. Electron. Syst.*, vol. 51, no. 4, pp., 2005.

[25] "OpenCV: How to Use Background Subtraction Methods", *Docs.opencv.org*, 2020. [Online]. Available: https://docs.opencv.org/master/d1/dc5/tutorial_background_subtraction.html. [Accessed: 09 May 2020].

[26] Macromini L. A., Cuhna A. L., “A Comparison between Background Modelling Methods for Vehicle Segmentation in Highway Traffic Videos”, 2018.

[27] Y. Wang, P.-M. Jodoin, F. Porikli, J. Konrad, Y. Benezeth, and P. Ishwar, “*CDnet 2014: An Expanded Change Detection Benchmark Dataset”*, in Proc. IEEE Workshop on Change Detection (CDW-2014) at CVPR-2014, pp. 387-394. 2014.

[28] "OpenCV: cv::BackgroundSubtractorMOG2 Class Reference", *Docs.opencv.org*, 2020. [Online]. Available: https://docs.opencv.org/3.4/d7/d7b/classcv_1_1BackgroundSubtractorMOG2.html. [Accessed: 09 May 2020].

[29] Suzuki, S. and Abe, K., *Topological Structural Analysis of Digitized Binary Images by Border Following*. CVGIP 30 1, pp 32-46, 1985.

[30] Orhan E., “Bayesian Inference: Particle Filtering”, Departament of Brain and Cognitive Sciences, University of Rochester, 2012.

[31] Chen Z., “Bayesian Filtering: From Kalman Filters to Particle Filters, and Beyond”, Communications Research Laboratory, McMaster University, Canada, 2003.

[32] Labbe R. R., “Kalman Filters in Python”, [online] Available at: <https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python>, 2015, [Accessed: 09 May 2020].

[33] "How a Kalman filter works, in pictures | Bzarg", *Bzarg.com*, 2020. [Online]. Available: https://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/. [Accessed: 09 May 2020].

[34] Understanding Kalman Filters, Part 3: Optimal State Estimator Video", *Mathworks.com*, 2020. [Online]. Available: https://www.mathworks.com/videos/understanding-kalman-filters-part-3-optimal-state-estimator--1490710645421.html. [Accessed: 09 May 2020].

[35] "Particle Filter : A hero in the world of Non-Linearity and Non-Gaussian", *Medium*, 2020. [Online]. Available: https://towardsdatascience.com/particle-filter-a-hero-in-the-world-of-non-linearity-and-non-gaussian-6d8947f4a3dc?gi=fa48f15a1968. [Accessed: 09 May 2020].

[36] "Kalman Filter", *Mathworks.com*, 2020. [Online]. Available: https://www.mathworks.com/discovery/kalman-filter.html. [Accessed: 09 May 2020].

[37] Bagnell D., “Good, Bad and Ugly of Particle Filters”, Lecture in Statistical Techniques in Robotics, Carnegie Mellon University.

[38] Elvira V., Miguez J., Djuric P. M., “Adapting The Number of Particles in Sequential Monte Carlo Methods through an Online Scheme for Convergence Assessment”, IEEE Transactions on Signal Processing, vol. 65, no. 7, pp. 1781-1794, 2017

[39] De Maesschalck R., Jouan-Rimbaud D., Massart D. L., “The Mahalanobis Distance”, Chemometrics and Intelligent Laboratory Systems, Volume 50, Issue 1, 2000, Pages 1-18.

[40] Gorji A., Shiry S., Menhaj M. B., “Multiple target tracking for mobile robots using the JPDAF algorithm”, 2009

8. Appendix

8.1. Object Detection

```
import cv2
import matplotlib.pyplot as plt
import numpy as np

"""Creating object detector class with a detect method"""
class detector(object):

    # Creating the constructor of detector class
    def __init__(self, frame, minimumDetectionArea, image_no, captured_frame):
        self.frame = frame
        self.minimumDetectionArea = minimumDetectionArea
        self.image_no = image_no
        self.captured_frame = captured_frame

    # This function is for rectangle drawing on the picture
    def draw_rectangles(self, coordinates, color):

        for i in coordinates:
            cv2.rectangle(self.captured_frame, (i[0], i[1]), (i[2], i[3]), /
                                                    color, 2)

    def detect(self):
        '''centers are measurements we obtain for filtering, i.e. point
        tracking. Coordiantes are trivial, for rectangle drawing
        on a picture, not actually used for tracking.
        Here we find countours of every shape, find center of
        this shape, and if it passes the minimumDetectionArea,
        it is then counted as a detected human.'''
        centers, coordinates = [], []
        self.frame = cv2.medianBlur(self.frame, 3) # Median Filter

        # Find Countours
        contours, hierarchy = cv2.findContours(self.frame, cv2.RETR_EXTERNAL,
                                               cv2.CHAIN_APPROX_SIMPLE)
        boundingRectangles = [None]*len(contours)

        # Find Coordinates of every bounding rectangle of every countour
        for i in range(len(contours)):
            boundingRectangles[i] = cv2.boundingRect(contours[i])
        for i in range(len(boundingRectangles)):
            x1 = (int(boundingRectangles[i][0]))
            y1 = (int(boundingRectangles[i][1]))
            x2 = (int(boundingRectangles[i][0]+boundingRectangles[i][2]))
            y2 = (int(boundingRectangles[i][1]+boundingRectangles[i][3]))
            center = [((x2 + x1) / 2), ((y2 + y1) / 2)]
            # Apply Threshold
            if ((abs(x2 - x1) * abs(y2 - y1)) > self.minimumDetectionArea):
                centers.append(center)
                coordinates.append(np.array([x1, y1, x2, y2]))
        if len(centers) == 0:
            return None, None
        return centers, coordinates
```

Figure 47. Code for object detection. Part 1

```
# MAIN BODY OF THE PROGRAM

minimumDetectionArea = 2000
imageSize = (480, 360)
history_time = 50

#creating an instance for VideoCapture object
capture = cv2.VideoCapture("Video1.mp4")

#creating an instance of BackgroundSubtractor object
bg_sub = cv2.createBackgroundSubtractorMOG2(history = history_time,
                                            detectShadows=False)

#permanent loop while the frame is available
while(1):
    #reading the current frame
    ret, captured_frame = capture.read()

    #break the loop if no input is received
    if captured_frame is None:
        break

    #resizing the frame received
    captured_frame = cv2.resize(captured_frame, (imageSize))

    #applying background subtraction method to get foreground mask
    fg_mask = bg_sub.apply(captured_frame)

    #creating an instance of detector object
    d = detector(fg_mask,
                 minimumDetectionArea,
                 image_no,
                 captured_frame)

    # Obtain measurements
    centers, coordinates = d.detect()

    # Draws boxes where object detected, if no measurements, do nothing
    try:
        d.draw_rectangles(coordinates, (256, 0, 0))
    except:
        pass

    cv2.imshow('Detection on the Original Image', captured_frame)
    cv2.imshow('Detected Background After Filtering', fg_mask)
    # Write images into desired folder
    # cv2.imwrite(f'pics\org_box\{image_no}.jpg', captured_frame)
    # cv2.imwrite(f'pics\c_bg_bf\{image_no}.jpg', fg_mask)

    #OpenCV syntax for display
    cv2.waitKey(1)

# Commands for plotting of the data follows after this
```

Figure 48. Code for object detection. Part 2

8.2. Kalman Filter

Note that implementation for filter is the same for both Kalman and extended Kalman, equations are the same, all we do is pass a different function.

```
import numpy as np

class KalmanFilter(object):
    def __init__(self, initial_state, initial_covariance, process_model,
                 process_noise, measurement_function, measurement_covariance,
                 no_of_state_variables, no_of_state_measurements):

        self.x = initial_state
        self.P = initial_covariance
        self.F = process_model
        self.Q = process_noise
        self.H = measurement_function
        self.R = measurement_covariance
        self.dim_x = no_of_state_variables
        self.dim_z = no_of_state_measurements

    def predict(self, dt):
        # KALMAN FILTER EQUATIONS:
        # x = F * x
        # P = F * P * F^T + Q, where ^T is equivalent to transposing matrix

        self.x = self.F.dot(self.x)
        self.P = self.F.dot(self.P).dot(self.F.T) + self.Q

    def update(self, state_measurement):

        z = state_measurement

        # Handling case where no measurements are received
        if np.all(z == np.array([0,0])):
            z = np.array([self.x[0], self.x[2]])

        # y = z - H * x
        # S = H * P * H^T + R
        # K = P * H^T
        # x = x + K * y
        # P = (I - K * H) * P, where I is an identity matrix

        y = z - self.H.dot(self.x)
        S = self.H.dot(self.P).dot(self.H.T) + self.R
        K = self.P.dot(self.H.T).dot(np.linalg.inv(S))

        self.x = self.x + K.dot(y)
        self.P = (np.eye(self.dim_x) - K.dot(self.H)).dot(self.P)

    @property
    def covs(self) -> np.array:
        return self.P

    @property
    def mean(self) -> np.array:
        return self.x
```

Figure 49. Code for Kalman filter. Part 1

```
# MAIN BODY OF THE CODE

import numpy as np
import math
from scipy.misc import derivative

np.random.seed(2) # Defining random seed so results are reproducable
dt = 0.1
STEPS = 100
time = np.linspace(0, dt*STEPS, STEPS) # Defining time array

meas_variance = 2 ** 2; acceleration_variance = 0.25 ** 2
meas_value_x = 0.0; meas_value_y = 0.0

# Defining initial true state
real_pos_x = 0.0; real_pos_y = 0.0
initial_speed_x = 3.0; initial_speed_y = 3.0

# Defining initial estimates
initial_pos_x_est = 0.0; initial_pos_y_est = 0.0
initial_speed_x_est = 3.0; initial_speed_y_est = 3.0

no_of_state_variables = 4
no_of_state_measurements = 2

# Defining state Matrices x, P, F, Q, H, R
x = np.array([initial_pos_x_est, initial_speed_x_est,
              initial_pos_y_est, initial_speed_y_est])

P = np.diag([0, acceleration_variance, 0, acceleration_variance])

# F Defined by regular kinematics equations:
# x = x + dt * v
# v = v

F = np.array([[1, dt, 0,  0],
              [0,  1, 0,  0],
              [0,  0, 1, dt],
              [0,  0, 0,  1]])

Q = np.diag([0, acceleration_variance, 0, acceleration_variance])

H = np.array([[1, 0, 0, 0],
              [0, 0, 1, 0]])

R = np.array([[meas_variance, 0],
              [0, meas_variance]])

# Initializing Kalman Filter
kf = KalmanFilter(initial_state = x,
                  initial_covariance = P,
                  process_model = F,
                  process_noise = Q,
                  measurement_function = H,
                  measurement_covariance = R,
                  no_of_state_variables = no_of_state_variables,
                  no_of_state_measurements = no_of_state_measurements)
```

Figure 50. Code for Kalman filter. Part 2

```
step = 0 # step counter, used for simulation of abscent measurements only

# Loop that runs throughout the time array defined previously
for t in time:
    #Simulate real speed
    real_speed_x = initial_speed_x + \
                   np.random.randn() * np.sqrt(acceleration_variance)
    real_speed_y = initial_speed_y + \
                   np.random.randn() * np.sqrt(acceleration_variance)

    # FOR EXTENDED KALMAN FILTER ONLY

    # def speed_of_x(t):
    #    return math.sin(t) + t

    # def speed_of_y(t):
    #    return math.cos(t) + t

    # x[1] = derivative(speed_of_x, t)
    # x[3] = derivative(speed_of_y, t)

    # Simulate real position
    real_pos_x = real_pos_x + dt * real_speed_x
    real_pos_y = real_pos_y + dt * real_speed_y

    # Simulate measurements with noise
    meas_value_x = real_pos_x + np.random.randn() * np.sqrt(meas_variance)
    meas_value_y = real_pos_y + np.random.randn() * np.sqrt(meas_variance)

    # Simulation of abscent measurements
    if step > 70 and step < 110:
        meas_value_x, meas_value_y = 0, 0

    # Assign obtained measurements to measurement array
    z = np.array([meas_value_x, meas_value_y])

    kf.predict(dt = dt)
    kf.update(state_measurement = z)
    step += 1 # increment step

# Commands for plotting of the data follows after this
```

Figure 51. Code for Kalman filter. Part 3

8.3. Particle Filter

```
import numpy as np
np.random.seed(0)

class ParticleFilter(object):
    def __init__(self, N, initialConditions, imageSize):
        '''__init__ constructor is a special python function that is
        called automatically every time when memory is allocated
        for a new object. Here we initialize object attributes and
        call class methods that create particles and weights for
        those particles.'''
        self.imageSize = imageSize
        self.N = N
        self.initialConditions = initialConditions
        if self.initialConditions is not None:
            self.initialCoordinates = np.array([self.initialConditions[0, 0],
                                                self.initialConditions[0, 1]])
            self.initialSpeed = np.array([self.initialConditions[1, 0],
                                          self.initialConditions[1, 1]])
            self.initialStdDev = np.array([self.initialConditions[2, 0],
                                           self.initialConditions[2, 1]])
        self.create_particles()
        self.create_weights()

    def create_particles(self):
        '''If we know about the initial state of the system
        we create normally distributed particles around that
        state, if we do not know anything, we create uniform
        particles throughout.'''
        if self.initialConditions is None:
            self.create_uniform_particles()
        else:
            self.create_gaussian_particles()

    '''Particles 0 and 1 stand for position in x and y, and particles
    2 and 3 stand for velocity in x and y directions.'''
    def create_gaussian_particles(self):
        self.particles = np.empty((self.N, 4))
        self.particles[:, 0] = self.initialCoordinates[0] + \
                               (np.random.randn(self.N) * self.initialStdDev[0])
        self.particles[:, 1] = self.initialCoordinates[1] + \
                               (np.random.randn(self.N) * self.initialStdDev[1])
        self.particles[:, 2] = self.initialSpeed[0] + \
                               (np.random.randn(self.N) * self.initialStdDev[0])
        self.particles[:, 3] = self.initialSpeed[1] + \
                               (np.random.randn(self.N) * self.initialStdDev[1])

    def create_uniform_particles(self):
        self.particles = np.empty((self.N, 4))
        self.particles[:, 0] = np.random.uniform(0, self.imageSize[0], self.N)
        self.particles[:, 1] = np.random.uniform(0, self.imageSize[1], self.N)
        # -10 and 10 are trivial values for limits of speed particles
        self.particles[:, 2] = np.random.uniform(-10, 10, self.N)
        self.particles[:, 3] = np.random.uniform(-10, 10, self.N)
```

Figure 52. Code for particle filter. Part 1

```
        # create weights for initial particles
    def create_weights(self):
        self.weights = np.ones(self.N) / self.N

        # find number of effective particles
    def neff(self):
        return 1 / np.sum(np.square(self.weights))

        # find mean and variance of weighted particles
    def estimate(self):
        particles_position = self.particles[:, 0:2]
        particles_speed = self.particles[:, 2:4]
        position_mean,position_var = self.mean_and_variance(particles_position)
        speed_mean, speed_var = self.mean_and_variance(particles_speed)
        return position_mean, position_var, speed_mean, speed_var

    def mean_and_variance(self, the_array):
        mean = np.average(the_array, weights=self.weights, axis=0)
        var  = np.average((the_array - mean)**2, weights=self.weights, axis=0)
        return mean, var

        # We move particles according to how we think they move
    def predict(self, speed):
        self.particles[:, 0] += speed[0] + (np.random.randn(self.N) * 10)
        self.particles[:, 1] += speed[1] + (np.random.randn(self.N) * 10)
        self.particles[:, 2] += (np.random.randn(self.N) * 0.1)
        self.particles[:, 3] += (np.random.randn(self.N) * 0.1)

    def update(self, measurements):
        # Find how far are distance particles from measurement
        distance_x = np.abs(self.particles[:, 0] - measurements[0])
        distance_y = np.abs(self.particles[:, 1] - measurements[1])
        distance = self.euclidean_distance(distance_x, distance_y)

        # Find how far are speed particles from measurements
        # As measurement for speed we use difference of previous and current
        # position
        speed_x = np.abs(self.particles[:, 2] - measurements[2])
        speed_y = np.abs(self.particles[:, 3] - measurements[3])
        speed = self.euclidean_distance(speed_x, speed_y)

        # Combine them
        total_difference = distance + speed

        # If measurement is far away from particle, it means its weight is low
        new_weights = 1/total_difference

        # Implementing Bayes Theorem
        self.weights *= new_weights
        self.weights += 1.e-50  # avoid round-off to zero
        self.weights /= sum(self.weights) # normalize

        # Function that calculates distance of particle from measurement
    def euclidean_distance(self, dist_x, dist_y):
        return np.sqrt(dist_x ** 2 + dist_y ** 2)
```

Figure 53. Code for particle filter. Part 2

```
    def resample(self):
        # Obtain indexes of particles we want to keep
        indexes = self.return_indexes()
        # Make new particles from those indexes
        self.resampled_particles(indexes)

    def return_indexes(self):
        sampling_positions = (np.arange(self.N) + \
                              np.random.uniform(0, 1/self.N)) / self.N
        indexes = np.zeros(self.N, 'i')
        cumulative_sum = np.cumsum(self.weights)
        i, j = 0, 0
        while i < self.N:
            if sampling_positions[i] < cumulative_sum[j]:
                indexes[i] = j
                i += 1
            else:
                j += 1
        return indexes

    def resampled_particles(self, indexes):
        resampled_weights = []
        resampled_particles = np.empty((self.N, 4))
        j = 0
        for i in indexes:
            resampled_weights.append(self.weights[i])
            resampled_particles[j, 0:4] = self.particles[i, 0:4]
            j += 1
        resampled_weights /= sum(resampled_weights)
        self.weights = resampled_weights
        self.particles = resampled_particles

# MAIN BODY OF THE CODE
N = 5000 # Number of particles
initialPosition = np.array([12, 185]) # Initial known position
initialSpeed = np.array([0, 0]) # Initial known speed
initialStdDev = np.array([50, 50]) # Belief in position (Standard deviation)
initialConditions = np.array([initialPosition,
                              initialSpeed,
                              initialStdDev]) # Set of initial conditions
# Creating instance of ParticleFilter object
pf = ParticleFilter(N, initialConditions, imageSize)
"""permanent loop while the file is open. IN THIS LOOP WE ALSO USE OBJECT
DETECTION METHOD SHOWN PREVIOUSLY"""
while(1):
    pf.predict(speed_mean)
    centers, coordinates = d.detect()
    if centers is not None:
        pf.update(centers)
    if pf.neff() < pf.N/2:
        pf.resample()
    pos_mean, pos_var, speed_mean, speed_var = pf.estimate()
```

Figure 54. Code for particle filter. Part 3

8.4. Data for Figure 42

	Time needed to process one frame, seconds	
Number of particles	Video 1	Video 2
100	0.027	0.035
250	0.026	0.032
500	0.027	0.041
1000	0.029	0.046
2500	0.035	0.052
5000	0.050	0.067
10000	0.082	0.101
25000	0.143	0.192
50000	0.257	0.374
100000	0.465	0.741
250000	1.062	1.488
500000	2.085	2.972
1000000	4.127	5.991

8.5. Gantt Chart