Name: Abijith J. Kamath
Student Id: 17788

E1 244: Detection and Estimation

February-May 2021

Solution – Final Project

Introduction

Automating a driving test can be done using a vehicle tracking algorithm from some measurements of the vehicle and analysing the path. A naive approach is to use the path to ensure the driver has maintained the path and avoided crashes. This does not ensure aspects like following lane rules, traffic signs, maintaining optimal speed, and other important driving ethics are being followed, but it is sufficient to make sure the driver is able to follow a given path.

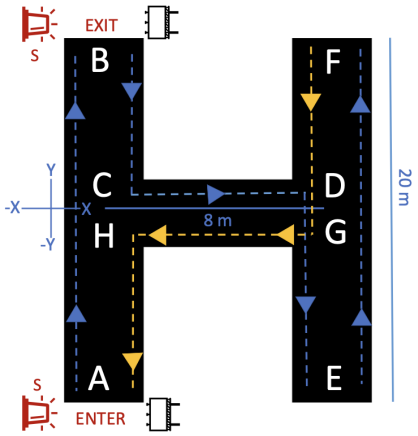


Figure 1: Track defined for the driving test.

Consider the track shown in Figure 1. The path to be taken by the vehicle is defined to be A-B-C-D-E-F-G-H-A. The vehicle position and velocities are measured using a RADAR, that helps to continuously track the vehicle in real time. The RADAR switched ON only when the detector at the entry point A detects a vehicle, and it remains ON until the detector at the exit point B detects a vehicle. The detectors are designed using light-dependent diodes (LDR) that are: excited by a source when there is no vehicle blocking the source, which can help in differentiating between states of vehicle being present and not present. The detection problem is to design appropriate detectors at the entry and the exit, and the estimation problem is to track the vehicle using the noisy RADAR measurements.

1 Part A: Vehicle Detection

1.1 Derivation

The detectors are built using LDRs, where a sensor measures voltages across the LDR. Voltage measurements $x_m[n],\ n = 0, 1, \cdots, N-1$ are made at M such sensors with labels $m = 1, 2, \cdots, M$. Hence, the output of

the sensors are:

$$x_m[n] = \begin{cases} A + B_t + w_m[n], & \text{vehicle absent,} \\ B_t + w_m[n], & \text{vehicle present,} \end{cases} \tag{1}$$

where A is the voltage due to the source, B_t is the voltage due to ambient light at some time t, and $w_m[n]$ are i.i.d zero mean, white Gaussian noise with variance σ^2, at the mth sensor. Therefore, detection of the vehicle is a hypothesis testing problem between two hypothesis:

$$\begin{aligned} &\text{(vehicle absent) } \mathcal{H}_0 : x_m[n] = A + B_t + w_m[n], \\ &\text{(vehicle present) } \mathcal{H}_1 : x_m[n] = B_t + w_m[n], n = 0, 1, \cdots, N-1; m = 1, 2, \cdots, M. \end{aligned} \tag{2}$$

Let $\mathbf{x}$ be the vectorised measurements of dimension NM (for simplicity, the derivations further take $M = 1$). The joint distribution of the measurements $p_X(\mathbf{x})$ is a Gaussian distribution with variance σ^2 and mean $A + B_t$ under $\mathcal{H}_0$, and B_t under $\mathcal{H}_1$. We will assume that we know some knowledge of B_t for some times during the day.

1.1.1 Detector at the entry for a given B_t

The detector at the entry is designed such that the probability of false alarm is the least, given the probability of detection is set to a constant. Such detectors maybe termed constant detection rate (CDR) detectors. Let the detector divide $\mathbb{R}$ into regions of decision R_0 and R_1 corresponding hypothesis $\mathcal{H}_0$ and $\mathcal{H}_1$, respectively. The regions are mutually exclusive and exhaustive, i.e., $R_0 \cup R_1 = \mathbb{R}$ and $R_0 \cap R_1 = \emptyset$. The CDR detector with probability of detection $P_D = \int_{R_1} p_X(\mathbf{x}; \mathcal{H}_1) \mathrm{d}\mathbf{x}$ and $P_{FA} = \int_{R_1} p_X(\mathbf{x}; \mathcal{H}_0) \mathrm{d}\mathbf{x}$ solves:

$$\begin{aligned} &\underset{R_1}{\text{minimise}}\ P_{FA}, \\ &\text{subject to } P_D = \beta. \end{aligned} \tag{3}$$

Consider the Lagrangian with multiplier λ:

$$\begin{aligned} \mathcal{L} &= P_{FA} + \lambda(P_D - \beta) \\ &= \int_{R_1} \left(p_X(\mathbf{x}; \mathcal{H}_0) + \lambda p_X(\mathbf{x}; \mathcal{H}_1) \right) \mathrm{d}\mathbf{x} - \lambda\beta. \end{aligned} \tag{4}$$

The minimiser of $\mathcal{L}$ also minimises P_{FA} and includes points $\mathbf{x}$ in R_1 such that $p_X(\mathbf{x}; \mathcal{H}_0) + \lambda p_X(\mathbf{x}; \mathcal{H}_1) < 0$. This gives the ratio test for detection: decide on $\mathcal{H}_1$ if:

$$L_{ent}(\mathbf{x}) = \frac{p_X(\mathbf{x}; \mathcal{H}_0)}{p_X(\mathbf{x}; \mathcal{H}_1)} < -\lambda = \gamma, \tag{5}$$

which is a ratio test that compares the ratio of likelihoods of the two hypothesis to some threshold γ that is decided using the condition $P_D = \beta$. Such a ratio test gives simple regions R_0 and R_1 that is parametrised by a threshold γ. This is similar to the Neymann-Pearson detector, where P_{FA} is held constant and P_D is maximised, and a similar ratio test is obtained.

Consider the test under the hypothesis in (2). The likelihood ratio admits:

$$\begin{aligned} L_{ent}(\mathbf{x}) &= \frac{p_X(\mathbf{x}; \mathcal{H}_0)}{p_X(\mathbf{x}; \mathcal{H}_1)}, \\ &= \frac{\frac{1}{(2\pi\sigma^2)^{N/2}} \exp\left(-\frac{1}{2\sigma^2} \sum_{n=0}^{N-1} (x_m[n] - A - B_t)^2 \right)}{\frac{1}{(2\pi\sigma^2)^{N/2}} \exp\left(-\frac{1}{2\sigma^2} \sum_{n=0}^{N-1} (x_m[n] - B_t)^2 \right)}, \\ \implies \ln L_{ent}(\mathbf{x}) &= -\frac{1}{2\sigma^2} \left(-2A \sum_{n=0}^{N-1} x_m[n] + NA^2 + 2NAB_t \right). \end{aligned} \tag{6}$$

Since $\ln(\cdot)$ is an increasing function, the ratio test is equivalent to $\ln L_{ent}(\mathbf{x}) < \ln \gamma$, which gives the test statistic:

$$T(\mathbf{x}) = \frac{1}{N}\sum_{n=0}^{N-1} x_m[n] < \frac{2\sigma^2 \ln\gamma + NA^2 + 2NAB_t}{2NA} = \gamma'. \tag{7}$$

The test statistic is the sample mean of the measurements. Since the measurements are i.i.d, the test statistic also has a Gaussian distribution:

$$T(\mathbf{x}) \sim \begin{cases} \mathcal{N}(A + B_t, \frac{\sigma^2}{N}), & \text{under } \mathcal{H}_0, \\ \mathcal{N}(B_t, \frac{\sigma^2}{N}), & \text{under } \mathcal{H}_1. \end{cases} \tag{8}$$

The constant probability of detection condition $\beta = \Pr[T(\mathbf{x}) < \gamma'; \mathcal{H}_1] = 1 - Q\left(\frac{\gamma' - B_t}{\sqrt{\sigma^2/N}}\right)$ gives:

$$\gamma' = \sqrt{\frac{\sigma^2}{N}} Q^{-1}(1-\beta) + B_t, \tag{9}$$

where $Q(\cdot)$ is the survival function of the standard normal distribution. The theoretical probability of false alarm $P_{FA} = \Pr[T(\mathbf{x}) < \gamma'; \mathcal{H}_0] = 1 - Q\left(\frac{\gamma' - A - B_t}{\sqrt{\sigma^2/N}}\right)$.

1.1.2 Detector at the entry with unknown B_t

The CDR detector is completely described by its threshold, and the threshold derived in (9) that minimises the probability of false alarm, depends on the knowledge of B_t. If B_t is not exactly known, the probability of detection may not remain a constant. In such scenarios, methods of composite hypothesis testing like generalised likelihood ratio test are used. However, in this case, since the range of possible values that B_t takes is known, the solution can be inferred from the threshold derived in (9).

Note that, for any choice of B_t, the theoretical probability of false alarm:

$$\begin{aligned} P_{FA} &= \Pr[T(\mathbf{x}) < \gamma'; \mathcal{H}_0], \\ &= 1 - Q\left(\frac{\gamma' - A - B_t}{\sqrt{\sigma^2/N}}\right), \\ &= 1 - Q\left(Q^{-1}(1-\beta) - \sqrt{\frac{NA^2}{\sigma^2}}\right), \end{aligned} \tag{10}$$

is independent of B_t, and hence, the choice of B_t does not change the optimal solution. Since the threshold still depends on B_t, the equality constraint $P_D = \beta$ cannot be maintained. Under the conditions of the problem, it is sufficient to maintain $P_D \geq \beta$. This can be ensured by picking the largest possible choice for the threshold, where, the probability of detection may increase, but the probability of false alarm remains the same, since it is independent of B_t. This is achieved by setting the threshold:

$$\gamma' = \sqrt{\frac{\sigma^2}{N}} Q^{-1}(1-\beta) + B_{\text{max}}, \tag{11}$$

where it is known that $B_t < B_{\text{max}}$ over the entire duration when testing is allowed. The detector at the entry (D_{entry}) is completely specified by the threshold given in (11), which constrains the probability of detection to be bounded below by β, and the probability of false alarm, as given by (10), to be the least with that constraint.

1.1.3 Detector at the exit for a given B_t

The detector at the exit is designed such that the probability of detection is the maximum, given the probability of false alarm is set to a constant. Such a detector is called a constant false-alarm (CFAR) detector, and the optimal detector is the Neymann-Pearson detector. Let the detector divide $\mathbb{R}$ into regions of decision R_0 and R_1 corresponding hypothesis $\mathcal{H}_0$ and $\mathcal{H}_1$, respectively. The regions are mutually exclusive and exhaustive, i.e., $R_0 \cup R_1 = \mathbb{R}$ and $R_0 \cap R_1 = \emptyset$. The CFAR detector with probability of detection $P_D = \int_{R_1} p_X(\mathbf{x}; \mathcal{H}_1)\mathrm{d}\mathbf{x}$ and $P_{FA} = \int_{R_1} p_X(\mathbf{x}; \mathcal{H}_0)\mathrm{d}\mathbf{x}$ solves:

$$\begin{aligned} &\underset{R_1}{\text{maximise}}\ P_D, \\ &\text{subject to } P_{FA} = \alpha. \end{aligned} \tag{12}$$

Consider the Lagrangian with multiplier λ:

$$\begin{aligned} \mathcal{L} &= P_D + \lambda(P_{FA} - \alpha) \\ &= \int_{R_1} \left(p_X(\mathbf{x}; \mathcal{H}_1) + \lambda p_X(\mathbf{x}; \mathcal{H}_0)\right) \mathrm{d}\mathbf{x} - \lambda\alpha. \end{aligned} \tag{13}$$

The maximiser of $\mathcal{L}$ also maximises P_D and includes points $\mathbf{x}$ in R_1 such that $p_X(\mathbf{x}; \mathcal{H}_1) + \lambda p_X(\mathbf{x}; \mathcal{H}_0) > 0$. This gives the ratio test for detection: decide on $\mathcal{H}_1$ if:

$$L_{ext}(\mathbf{x}) = \frac{p_X(\mathbf{x}; \mathcal{H}_1)}{p_X(\mathbf{x}; \mathcal{H}_0)} > -\lambda = \xi, \tag{14}$$

which is a ratio test that compares the ratio of likelihoods of the two hypothesis to some threshold ξ that is decided using the condition $P_{FA} = \alpha$. This is also a ratio rest that gives simple regions R_0 and R_1 that is parametrised by ξ. This is the Neymann-Pearson detector. The CDR detector in (5) and CFAR detector in (14) are similar, but the thresholds are decided using different conditions, that yield different thresholds.

Consider the test under the hypothesis in (2). The likelihood ratio admits:

$$\begin{aligned} L_{ext}(\mathbf{x}) &= \frac{p_X(\mathbf{x}; \mathcal{H}_1)}{p_X(\mathbf{x}; \mathcal{H}_0)}, \\ &= \frac{\frac{1}{(2\pi\sigma^2)^{N/2}} \exp\left(-\frac{1}{2\sigma^2}\sum_{n=0}^{N-1}(x_m[n] - B_t)^2\right)}{\frac{1}{(2\pi\sigma^2)^{N/2}} \exp\left(-\frac{1}{2\sigma^2}\sum_{n=0}^{N-1}(x_m[n] - A - B_t)^2\right)}, \\ \implies \ln L_{ext}(\mathbf{x}) &= -\frac{1}{2\sigma^2}\left(2A\sum_{n=0}^{N-1} x_m[n] - NA^2 - 2NAB_t\right). \end{aligned} \tag{15}$$

Since $\ln(\cdot)$ is an increasing function, the ratio test is equivalent to $\ln L_{ext}(\mathbf{x}) > \ln \xi$, which gives the same test statistic as in (7):

$$T(\mathbf{x}) = \frac{1}{N}\sum_{n=0}^{N-1} x_m[n] < \frac{-2\sigma^2 \ln \xi + NA^2 + 2NAB_t}{2NA} = \xi'. \tag{16}$$

The test statistic is the sample mean of the measurements and has the same distribution as in (8). The constant false-alarm condition $\alpha = \Pr[T(\mathbf{x}) < \xi'; \mathcal{H}_0] = 1 - Q\left(\frac{\xi' - A - B_t}{\sqrt{\sigma^2/N}}\right)$ gives:

$$\xi' = \sqrt{\frac{\sigma^2}{N}} Q^{-1}(1 - \alpha) + A + B_t. \tag{17}$$

The theoretical probability of detection $P_D = \Pr[T(\mathbf{x}) < \gamma'; \mathcal{H}_1] = 1 - Q\left(\frac{\gamma' - B_t}{\sqrt{\sigma^2/N}}\right)$.

1.1.4 Detector at the exit with unknown B_t

The CFAR detector is completely described by its threshold, and the threshold derived in (17) that maximises the probability of detection, depends on the knowledge of B_t. If B_t is not exactly known, similar to the case described in Section 1.1.2 the probability of false alarm may not remain a constant.

Note that, for any choice of B_t, the theoretical probability of detection:

$$\begin{aligned} P_D &= \Pr[T(\mathbf{x}) < \xi'; \mathcal{H}_1], \\ &= 1 - Q\left(\frac{\xi' - B_t}{\sqrt{\sigma^2/N}}\right), \\ &= 1 - Q\left(Q^{-1}(1-\alpha) - \sqrt{\frac{NA^2}{\sigma^2}}\right), \end{aligned} \tag{18}$$

is independent of B_t, and hence, the choice of B_t does not change the optimal solution. Since the threshold still depends on B_t, the equality constraint $P_{FA} = \alpha$ cannot be maintained. Under the conditions of the problem, it is sufficient to maintain $P_{FA} \leq \alpha$. This can be ensured by picking the smallest possible choice for the threshold, where, the probability of false alarm may decrease, but the probability of detection remains the same, since it is independent of B_t. This is achieved by setting the threshold:

$$\xi' = \sqrt{\frac{\sigma^2}{N}} Q^{-1}(1-\alpha) + A + B_{\text{min}}, \tag{19}$$

where it is known that $B_t > B_{\text{min}}$ over the entire duration when testing is allowed. The detector at the exit (D_{exit}) is completely specified by the threshold given in (19), which constrains the probability of false alarm to be bounded above by α, and the probability of detection, as given by (**??**), to be the highest with that constraint.

1.1.5 Locally Most Powerful detector at the exit

Section 1.1.3 derives the Neymann-Pearson detector for hypothesis testing, where the hypothesis are as in (2). Since the noise is i.i.d Gaussian distributed, the two hypotheses can be reformulated as mean-detection problem with $\mathbf{x} \sim \mathcal{N}(\mu, \sigma^2)$ and:

$$\begin{aligned} \text{(vehicle absent) } \mathcal{H}_0 &: \mu = B_t, \\ \text{(vehicle present) } \mathcal{H}_1 &: \mu > B_t, \end{aligned} \tag{20}$$

since it is known that the voltage drop due to the source $A > 0$. This is a one-sided hypothesis test, and is amenable to the locally most-powerful (LMP) test. The data has a Gaussian distribution $p_X(\mathbf{x}; \mu)$ that is parametrised by the mean, depending on the hypothesis. The LMP uses the test statistic:

$$T_{\text{LMP}}(\mathbf{x}) = \frac{1}{\sqrt{I(B_t)}} \frac{\partial}{\partial \mu} \ln p_X(\mathbf{x}; B_t), \tag{21}$$

where $I(\cdot)$ is the Fisher information of μ on p_X. The derivatives of the log-likelihood function are computed as:

$$\begin{aligned} \ln p_X(\mathbf{x}; \mu) &= -\frac{N}{2} \ln(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{n=0}^{N-1} \left(x_m[n] - \mu\right)^2, \\ \implies \frac{\partial}{\partial \mu} \ln p_X(\mathbf{x}; \mu) &= \frac{1}{\sigma^2} \sum_{n=0}^{N-1} \left(x_m[n] - \mu\right), \\ \implies \frac{\partial^2}{\partial \mu^2} \ln p_X(\mathbf{x}; \mu) &= -\frac{N}{\sigma^2}. \end{aligned} \tag{22}$$

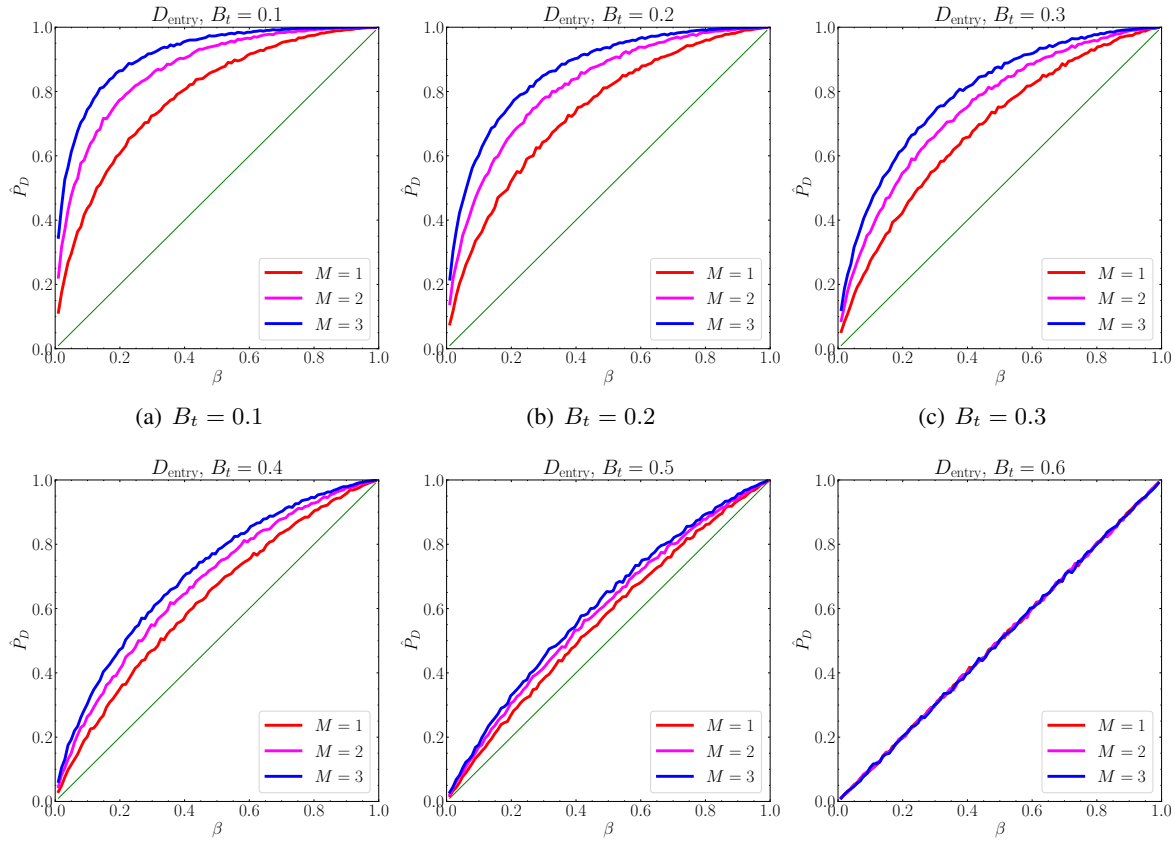
The Fisher information $I(\mu) = -\mathbb{E}[\frac{\partial^2}{\partial \mu^2} \ln p_X(\mathbf{x}; \mu)] = \frac{N}{\sigma^2}$. Using this in (21), the test statistic $T_{\text{LMP}}(\mathbf{x}) = \frac{1}{\sqrt{N\sigma^2}} \sum_{n=0}^{N-1} (x_m[n] - B_t)$. The LMP detector decides on the hypothesis $\mathcal{H}_1$ if $T_{\text{LMP}}(\mathbf{x}) > \eta$, where η is some threshold chosen using the problem constraints, for example, constant false-alarm rate as in Section 1.1.3. The test statistic $T_{\text{LMP}}(\mathbf{x})$ depends on the data only in the sample mean, as in (16), i.e., the two detectors have identical test statistics. If the constraints are the same, the LMP detector is identical to the Neymann-Pearson detector.

1.2 Implementation

We simulate the source with $A = 1$, and consider detection during six time instants during the day where the ambient light gives $B_t \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6\}$. We consider $N = 5$ measurements from each sensor for both detectors and consider the effect of varying the number of sensors from $M = 1, 2, 3$.

1.2.1 Monte-Carlo simulations for D_{entry}

We implement D_{entry} using the threshold (11), and perform Monte-Carlo simulations averaged over 10000 realisations to estimate the probability of detection $\hat{P}_D$. We repeat the experiment for the six values of B_t considered and vary $M = 1, 2, 3$ in each case.



(a) $B_t = 0.1$ (b) $B_t = 0.2$ (c) $B_t = 0.3$

(d) $B_t = 0.4$ (e) $B_t = 0.5$ (f) $B_t = 0.6$

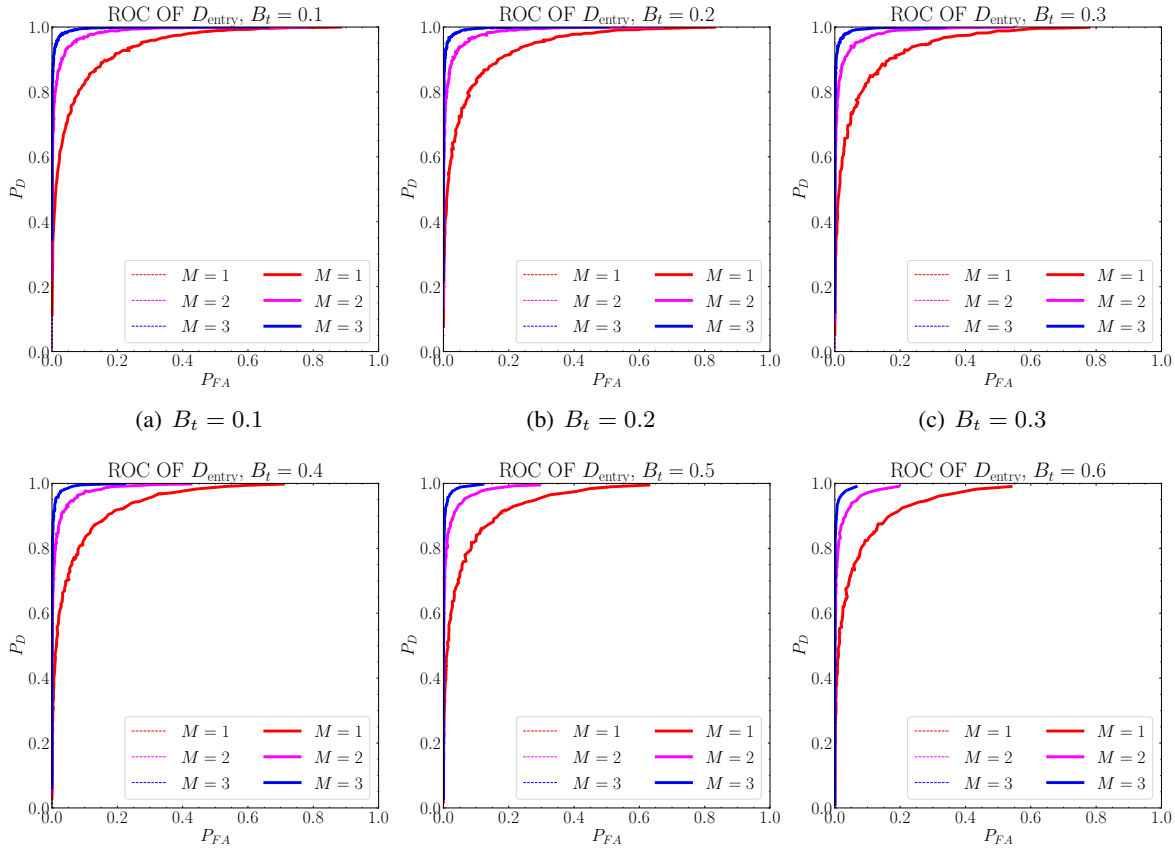
Figure 2: [Colour online] Variation of estimated probability of detection with varying bounds β and number of channels M, for each value of parameter B_t. The green line shows the equality constraint $P_D = \beta$.

Figure 2 shows the variation of the estimated probability of detection $\hat{P}_D$ with the bound β. We observe, in all cases, the probability of detection is bounded below by the parameter β. Since the parameter matches the

true value for $B_t = 0.6$, we see that $\hat{P}_D = \beta$, and for values of B_t diverging from 0.6, the estimated probability of detection moves further away from the bound. For any fixed value of B_t, and fixed value of β, we observe the estimated probability of detection increases with the increase in the number of channels M. This is as expected using the law of large numbers.

1.2.2 ROC of D_{entry}

Using the Monte-Carlo simulations in Section 1.2.1, we plot the ROC for D_{entry}, for each of the six values of B_t considered. We also compare it to the true ROC using (10).



(a) $B_t = 0.1$ (b) $B_t = 0.2$ (c) $B_t = 0.3$

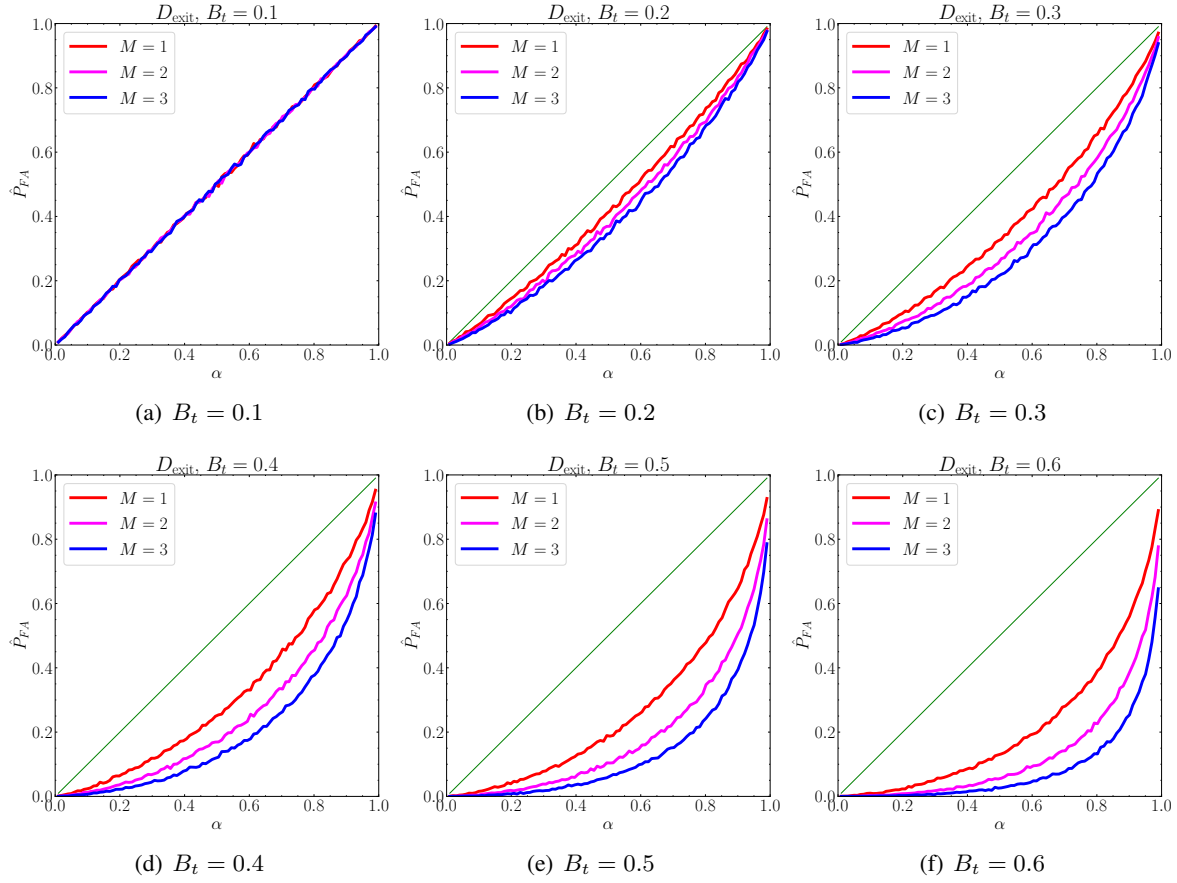
(d) $B_t = 0.4$ (e) $B_t = 0.5$ (f) $B_t = 0.6$

Figure 3: [Colour online] ROC for the detector for varying number of channels M, for each value of parameter B_t. Solid lines show the estimated ROC and dashed lines show the theoretical ROC.

Figure 3 shows the ROC for D_{entry} with varying number of channels M, and for each value of B_t. We observe that the ROC is identical for each setting M, over varying B_t. This shows that the detector is independent of B_t, as designed. For any given value of B_t, we observe the ROC is concave, i.e., achieves high probability of detection for low probability of false alarm, and matches the theoretical ROC, in all cases with varying M. Between the ROC for varying M, we see that the ROC with a larger value for M is more concave. This is, again, as expected as more channels implies more measurements.

1.2.3 Monte-Carlo simulations for D_{exit}

We implement D_{exit} using the threshold (19), and perform Monte-Carlo simulations averaged over 10000 realisations to estimate the probability of false alarm $\hat{P}_{FA}$. We repeat the experiment for the six values of B_t considered and vary $M = 1, 2, 3$ in each case.



(a) $B_t = 0.1$ (b) $B_t = 0.2$ (c) $B_t = 0.3$

(d) $B_t = 0.4$ (e) $B_t = 0.5$ (f) $B_t = 0.6$

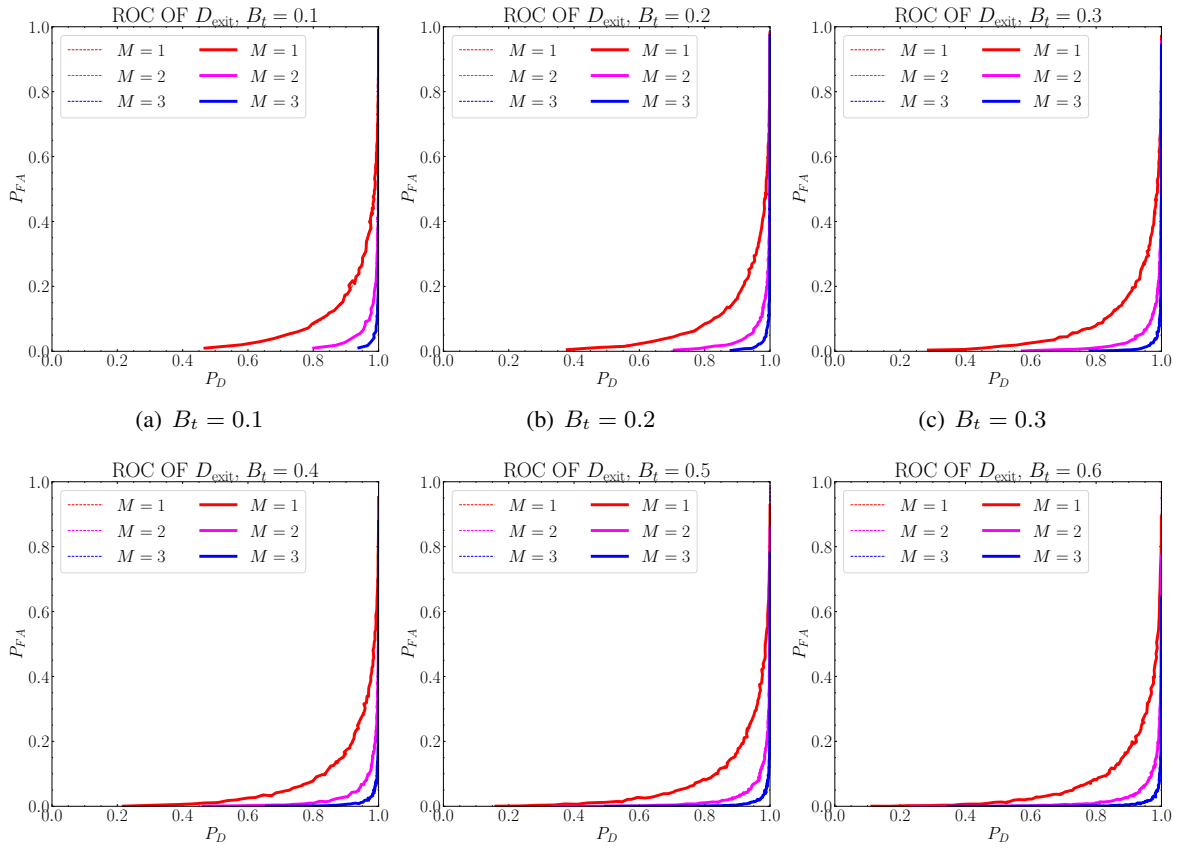
Figure 4: [Colour online] Variation of estimated probability of false alarm with varying bounds α and number of channels M, for each value of parameter B_t.

Figure 4 shows the variation of the estimated probability of false alarm $\hat{P}_{FA}$ with the bound α. We observe, in all cases, the probability of false alarm is bounded above by the parameter α. Since the parameter matches the true value for $B_t = 0.1$, we see that $\hat{P}_{FA} = \alpha$, and for values of B_t diverging from 0.1, the estimated probability of false alarm moves further away from the bound. For any fixed value of B_t, and fixed value of α, we observe the estimated probability of false alarm decreases with the increase in the number of channels M. This is as expected using the law of large numbers.

1.2.4 ROC of D_{exit}

Using the Monte-Carlo simulations in Section 1.2.3, we plot the ROC for D_{exit}, for each of the six values of B_t considered. We also compare it to the true ROC using (18).

Figure 5 shows the ROC for D_{exit} with varying number of channels M, and for each value of B_t. We observe that the ROC is identical for each setting M, over varying B_t. This shows that the detector is independent of B_t, as designed. For any given value of B_t, we observe the ROC is convex (axes are flipped, as compared to Figure 3), i.e., achieves high probability of detection for low probability of false alarm, and matches the theoretical ROC, in all cases with varying M. Between the ROC for varying M, we see that the ROC with a larger value for M is more convex. This is, again, as expected as more channels implies more measurements.



(a) $B_t = 0.1$ (b) $B_t = 0.2$ (c) $B_t = 0.3$

(d) $B_t = 0.4$ (e) $B_t = 0.5$ (f) $B_t = 0.6$

Figure 5: [Colour online] ROC for the detector for varying number of channels M, for each value of parameter B_t. Solid lines show the estimated ROC and dashed lines show the theoretical ROC.

2 Part B: Vehicle Tracking

2.1 Derivation

The deployed RADAR provides measurements of the position and velocity of the vehicle in 2 dimensions. We assume the vehicle moves with constant speed at the instants where the position and velocity are measured. Since the track has paths that mostly only have horizontal-only or vertical-only motions, constant speed implies constant velocity at each segment after which either the sign or the coordinate or both changes. The position and velocity maybe assumed to be unknown and maybe incorporated into the state model.

2.1.1 State model for Kalman filter

The vehicle dynamics, under constant speed, is fully described by its position vector $[p_x[n] \;\; p_y[n]]^\mathsf{T}$ and velocity vector $[\dot{p}_x[n] \;\; \dot{p}_y[n]]^\mathsf{T}$, at every time step $n = 0, 1, \cdots, N-1$. Hence, let the state variable be

$\mathbf{x}[n] = [p_x[n]\ p_y[n]\ \dot{p}_x[n]\ \dot{p}_y[n]]^{\mathsf{T}}$. The equations of kinematics with constant motion, in general, are:

$$\begin{aligned} p_x[n] &= p_x[n-1] + \dot{p}_x[n-1]\Delta + w_1[n], \\ p_y[n] &= p_y[n-1] + \dot{p}_y[n-1]\Delta + w_2[n], \\ \dot{p}_x[n] &= \dot{p}_x[n-1] + w_3[n], \\ \dot{p}_y[n] &= \dot{p}_y[n-1] + w_4[n], \\ \text{i.e. } \mathbf{x}[n] &= \underbrace{\begin{bmatrix} 1 & 0 & \Delta & 0 \\ 0 & 1 & 0 & \Delta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{A}[n-1]} \mathbf{x}[n-1] + \mathbf{w}[n], \end{aligned} \tag{23}$$

where $\mathbf{A}[n]$ is the state-transition matrix and $\mathbf{w}[n]$ is the noise vector at time step n that accounts for model mismatch. We assume the noise vector is white Gaussian distributed with covariance matrix $\mathbf{Q}_w[n]$. However, the state-transition matrix changes with the time step as the vehicle turns at the corners. Therefore, (23) is not valid for all (or maybe simplified at some) time steps.

Let n_X denote the time step when the vehicle reaches the node X, where $X \in$ A,B,...,H. From A-B, B-C, D-E, E-F, F-G, H-A, i.e., $n = 0, 1, \cdots, n_B, n_B+1, n_B+2, \cdots, n_C-1, n_D+1, n_D+2, \cdots, n_E-1, n_E+1, n_E+2, \cdots, n_F-1, n_F+1, n_F+2, \cdots, n_G-1, n_H+1, n_H+2, \cdots, N-1$, we can set $\dot{p}_x[n] = 0$. At B, E, F, i.e., $n = n_B, n_E, n_F$, $\dot{p}_y[n] = -\dot{p}_y[n-1] + w_4[n]$. At C, G, i.e., $n = n_C, n_G$, we have $\dot{p}_x[n] = -\dot{p}_y[n-1]$ and $\dot{p}_y[n] = 0$. From C-D, G-H, i.e., $n = n_C+1, n_C+2, \cdots, n_D-1, n_G+1, n_G+2, \cdots, n_H-1$, we can set $\dot{p}_y[n] = 0$ in (23). At D, H, i.e., $n = n_D, n_H$, we have $\dot{p}_y[n] = -\dot{p}_x[n-1]$ and $\dot{p}_x[n] = 0$. This gives the following as the state matrix:

$$\mathbf{A}[n] = \begin{cases} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & \Delta \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} & ; n < n_C, n_D < n < n_F, n_H < n, \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & \Delta \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix} & ; n = n_C, n_G, \\ \begin{bmatrix} 1 & 0 & \Delta & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} & ; n = n_D, n_H, \\ \begin{bmatrix} 1 & 0 & \Delta & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} & ; \text{otherwise.} \end{cases} \tag{24}$$

2.1.2 Kalman filter using velocity measurements

Consider the vehicle with the state equation $\mathbf{x}[n] = \mathbf{A}[n-1]\mathbf{x}[n-1] + \mathbf{w}[n]$, with the state variable measured using:

$$\mathbf{y}[n] = \underbrace{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{C}[n]} \mathbf{x}[n-1] + \mathbf{v}[n],\ n = 0, 1, \cdots, N-1, \tag{25}$$

where $\mathbf{v}[n]$ is the measurement noise. We assume the measurement noise is i.i.d zero mean Gaussian distributed with covariance matrix $\mathbf{Q}_v[n]$. The Kalman filter updates for the state variable and the error covariance are then given by:

$$\begin{aligned} \hat{\mathbf{x}}[n|n-1] &= \mathbf{A}[n-1]\hat{\mathbf{x}}[n-1|n-1], \\ \mathbf{P}[n|n-1] &= \mathbf{A}[n-1]\mathbf{P}[n-1|n-1]\mathbf{A}^{\mathsf{T}}[n-1] + \mathbf{Q}_w[n], \end{aligned} \tag{26}$$

and the Kalman filter predictions are given by:

$$\begin{aligned} \hat{\mathbf{x}}[n|n] &= \hat{\mathbf{x}}[n|n-1] + \mathbf{K}[n]\left(y[n] - \mathbf{C}[n]\hat{\mathbf{x}}[n|n-1]\right), \\ \mathbf{P}[n|n] &= \left(\mathbf{I} - \mathbf{K}[n]\mathbf{C}[n]\right)\mathbf{P}[n|n-1], \end{aligned} \tag{27}$$

where the Kalman gain is given by $\mathbf{K}[n] = \mathbf{P}[n|n-1]\mathbf{C}^{\mathsf{T}}[n]\left(\mathbf{Q}_v + \mathbf{C}[n]\mathbf{P}[n|n-1]\mathbf{C}^{\mathsf{T}}[n]\right)^{-1}$.

2.1.3 Kalman filter using position and velocity measurements

Consider the vehicle with the state equation $\mathbf{x}[n] = \mathbf{A}[n-1]\mathbf{x}[n-1] + \mathbf{w}[n]$, with the state variable measured using:

$$\mathbf{y}[n] = \mathbf{x}[n-1] + \mathbf{v}[n],\ n = 0, 1, \cdots, N-1, \tag{28}$$

where $\mathbf{v}[n]$ is the measurement noise. We assume the measurement noise is i.i.d zero mean Gaussian distributed with covariance matrix $\mathbf{Q}_v[n]$. The Kalman filter updates are identical to (26) and predictions are identical to (27) with $\mathbf{C}[n] = \mathbf{I}$. The Kalman filter predictions:

$$\begin{aligned} \hat{\mathbf{x}}[n|n] &= \hat{\mathbf{x}}[n|n-1] + \mathbf{K}[n]\left(y[n] - \hat{\mathbf{x}}[n|n-1]\right), \\ \mathbf{P}[n|n] &= \left(\mathbf{I} - \mathbf{K}[n]\right)\mathbf{P}[n|n-1], \end{aligned} \tag{29}$$

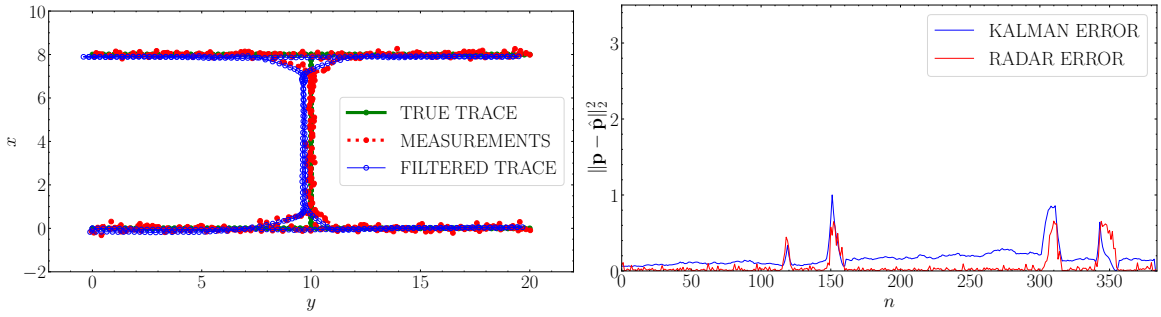
where the Kalman gain is given by $\mathbf{K}[n] = \mathbf{P}[n|n-1]\left(\mathbf{Q}_v + \mathbf{P}[n|n-1]\right)^{-1}$.

2.2 Implementation

2.2.1 Kalman filter with velocity only measurements

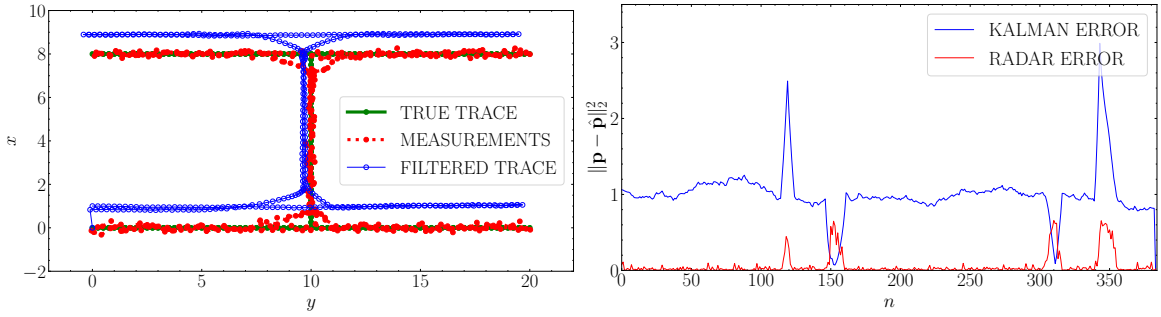
We implement Kalman filter with velocity-only measurements using the state model as in (23) and measurement model as in (25). We consider RADAR measurements of velocity with measurement noise $\mathbf{Q}_v = 0.1\mathbf{I}$. We set the model noise $\mathbf{Q}_w = 0.1\mathbf{I}$ and consider two initialisations: first with the correct initial position $\hat{x}[0|0] = \mathbf{0}$, and second with an offset to the position with $\hat{x}[0|0] = [1\ 0\ 0\ 0]^{\mathsf{T}}$. We determine the tracking performance using the squared-error between the Kalman filter predictions of position ($\hat{\mathbf{p}}[n]$) and the true trace ($\mathbf{p}[n]$), and compare it to the squared error between the RADAR measurements and the true trace.

Figure 6 shows the trace and the errors for the two cases of initialisation. We observe that the tracking in case of initialisation with $\hat{x}[0|0] = \mathbf{0}$ is fairly good. The error of the Kalman filter predictions matches the error in the RADAR measurements with an offset. The offset is much larger in the second case when the initialisation is set to $\hat{x}[0|0] = [1\ 0\ 0\ 0]^{\mathsf{T}}$. The trace shows that the tracking is off by the same amount the initialisation is off, throughout the trace. This is a feature of any position tracking algorithm from velocity measurements.



(a) Trace $\mathbf{Q}_w = 0.1\mathbf{I}, \hat{\mathbf{x}}(0|0) = \mathbf{0}$

(b) Error, $\mathbf{Q}_w = 0.1\mathbf{I}, \hat{\mathbf{x}}(0|0) = \mathbf{0}$



(c) Trace $\mathbf{Q}_w = 0.1\mathbf{I}, \hat{\mathbf{x}}(0|0) \neq \mathbf{0}$

(d) Error, $\mathbf{Q}_w = 0.1\mathbf{I}, \hat{\mathbf{x}}(0|0) \neq \mathbf{0}$

Figure 6: [Colour online] Figures show the Kalman filter prediction and RADAR measurements compared to the true trace. The first column shows the traces and the second column shows the errors, for different initialisations and model noise.

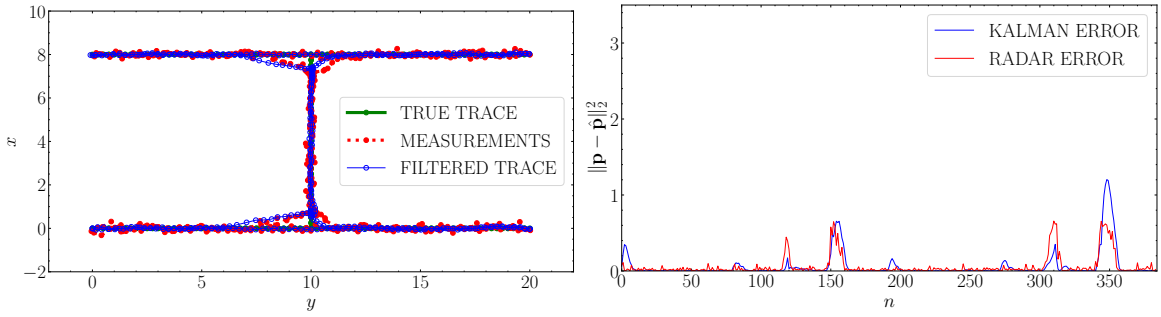
Estimation of position from velocity measurements needs solving first order system. The solutions are accurate up to a constant, if initial values are not available or are incorrect. In the first case, when the initial values are correct, the tracking is fairly decent. However, the second case, the tracking is poor, but the solution, as expected, is offset by the initialisation. Further, note that the Kalman filter predictions find the turns the vehicle takes, even though such motions are not allowed according to the state model. This is because the model noise $\mathbf{Q}_w = 0.1\mathbf{I}$ allows some model mismatch. However, since only velocity measurements are available, the tracking at turns also has additional bias.

Suppose the measurements are clean ($\mathbf{Q}_v = \mathbf{0}$) and the vehicle always moves with a constant speed ($\mathbf{Q}_w = \mathbf{0}$), it is possible to accurately track the position of the vehicle only up to a constant offset. This offset maybe removed by correctly initialising the Kalman filter.

2.2.2 Kalman filter with position and velocity measurements

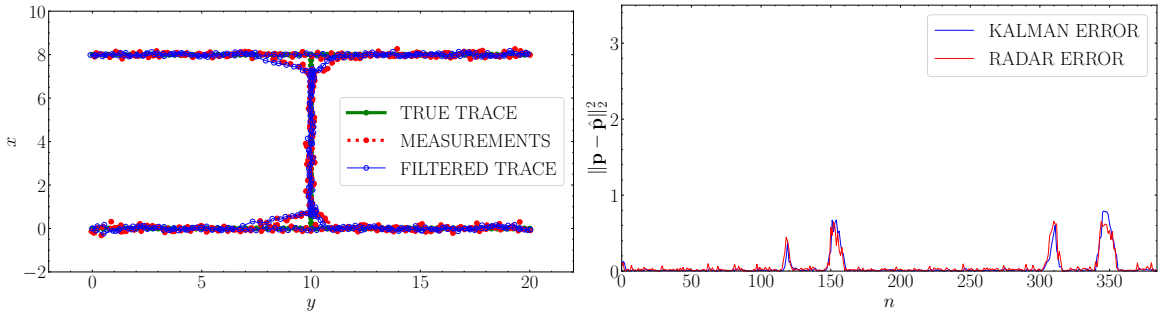
We implement the Kalman filter using both position and velocity measurements using the state model in (23) and measurement model as in (28). We consider RADAR measurements of both position and velocity with measurement noise $\mathbf{Q}_v = 0.1\mathbf{I}$. We vary the model noise $\mathbf{Q}_w \in \{0.01\mathbf{I}, 0.1\mathbf{I}, \mathbf{I}\}$ and initialise the Kalman filter with $\hat{x}[0|0] = \mathbf{0}$. We determine the tracking performance using the squared-error between the Kalman filter predictions of position ($\hat{\mathbf{p}}[n]$) and the true trace ($\mathbf{p}[n]$), and compare it to the squared error between the RADAR measurements and the true trace.

Figure 7 shows the trace and errors for the three cases of model noise. We observe the Kalman filter predictions perform well in all three cases, at tracking when the model is accurately followed, i.e., at all points that do not include a turn. However, at turns, we see that the error with lower values of model noise is higher. This is because the measurements do not strictly follow the model in (23), where smooth turns are not allowed. When the model noise is low, the tracking error is high wherever the model mismatch is high. We see that



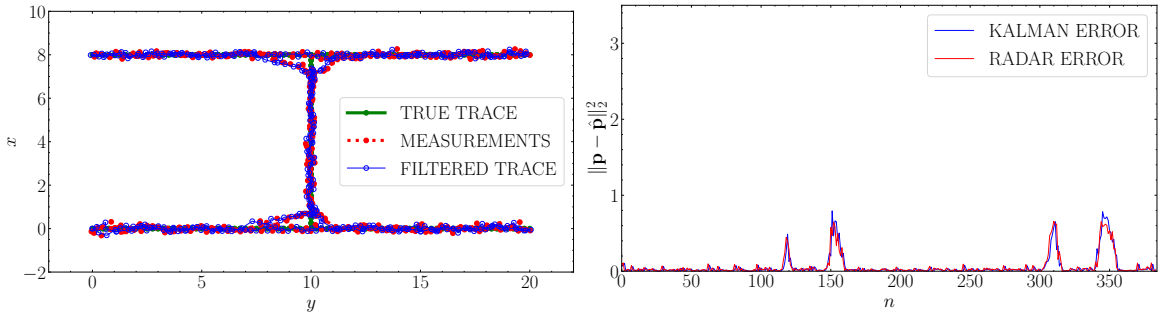
(a) Trace $\mathbf{Q}_w = 0.01\mathbf{I}$

(b) Error, $\mathbf{Q}_w = 0.01\mathbf{I}$



(c) Trace $\mathbf{Q}_w = 0.1\mathbf{I}$

(d) Error, $\mathbf{Q}_w = 0.1\mathbf{I}$



(e) Trace $\mathbf{Q}_w = \mathbf{I}$

(f) Error, $\mathbf{Q}_w = \mathbf{I}$

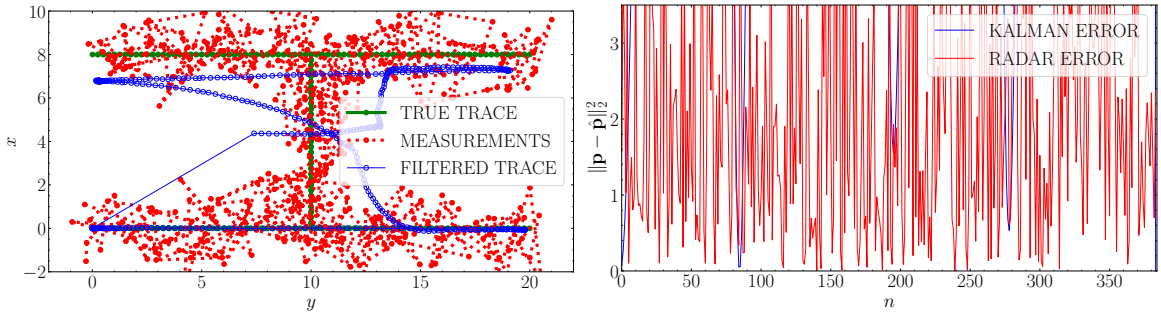
Figure 7: [Colour online] Figures show the Kalman filter prediction and RADAR measurements compared to the true trace. The first column shows the traces and the second column shows the errors, for different model noise, under measurements with $\mathbf{Q}_v = 0.1\mathbf{I}$.

the error at turns in the Kalman filter predictions matches the error in the RADAR measurements as the model noise increases.

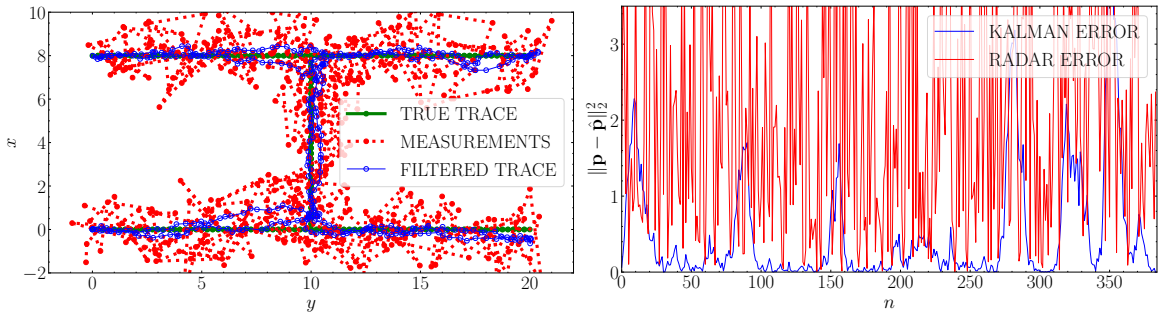
2.2.3 Kalman filter with noisy measurements

We implement the Kalman filter using both position and velocity measurements using the state model in (23) and measurement model as in (28). We consider RADAR measurements of both position and velocity with measurement noise $\mathbf{Q}_v = \mathbf{I}$. We vary the model noise $\mathbf{Q}_w \in \{0.0001\mathbf{I}, 0.01\mathbf{I}, 0.1\mathbf{I}, \mathbf{I}\}$ and initialise the Kalman filter with $\hat{x}[0|0] = \mathbf{0}$. We determine the tracking performance using the squared-error between the Kalman filter predictions of position ($\hat{\mathbf{p}}[n]$) and the true trace ($\mathbf{p}[n]$), and compare it to the squared error between the RADAR measurements and the true trace.

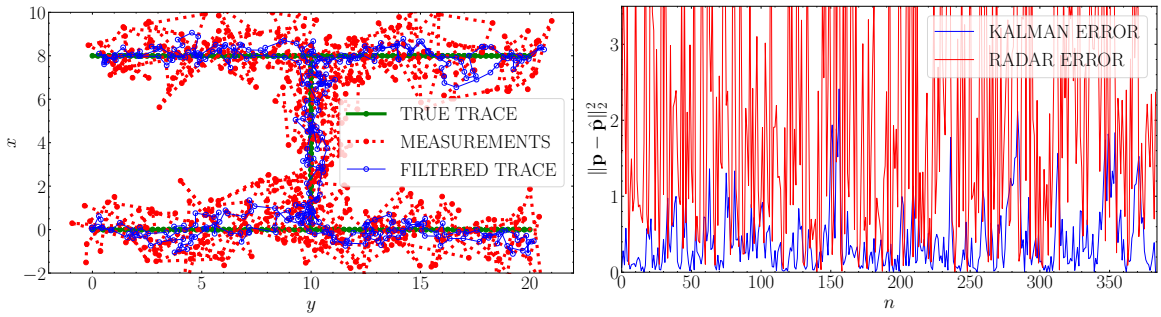
Figure 8 shows the trace and errors for the three cases of model noise. We observe the effect of model noise is stronger when the measurement noise is large. In the case where $\mathbf{Q}_v = 0.0001\mathbf{I}$, the Kalman filter



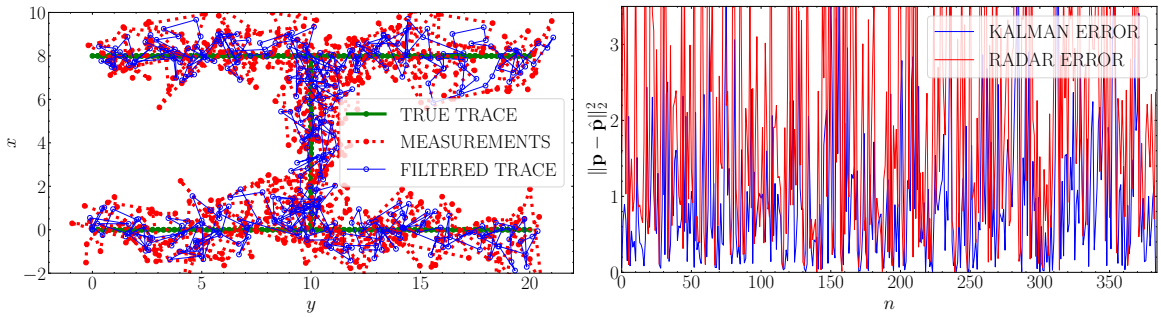
(a) Trace $\mathbf{Q}_w = 0.0001\mathbf{I}$ (b) Error, $\mathbf{Q}_w = 0.0001\mathbf{I}$



(c) Trace $\mathbf{Q}_w = 0.01\mathbf{I}$ (d) Error, $\mathbf{Q}_w = 0.01\mathbf{I}$



(e) Trace $\mathbf{Q}_w = 0.1\mathbf{I}$ (f) Error, $\mathbf{Q}_w = 0.1\mathbf{I}$



(g) Trace $\mathbf{Q}_w = \mathbf{I}$ (h) Error, $\mathbf{Q}_w = \mathbf{I}$

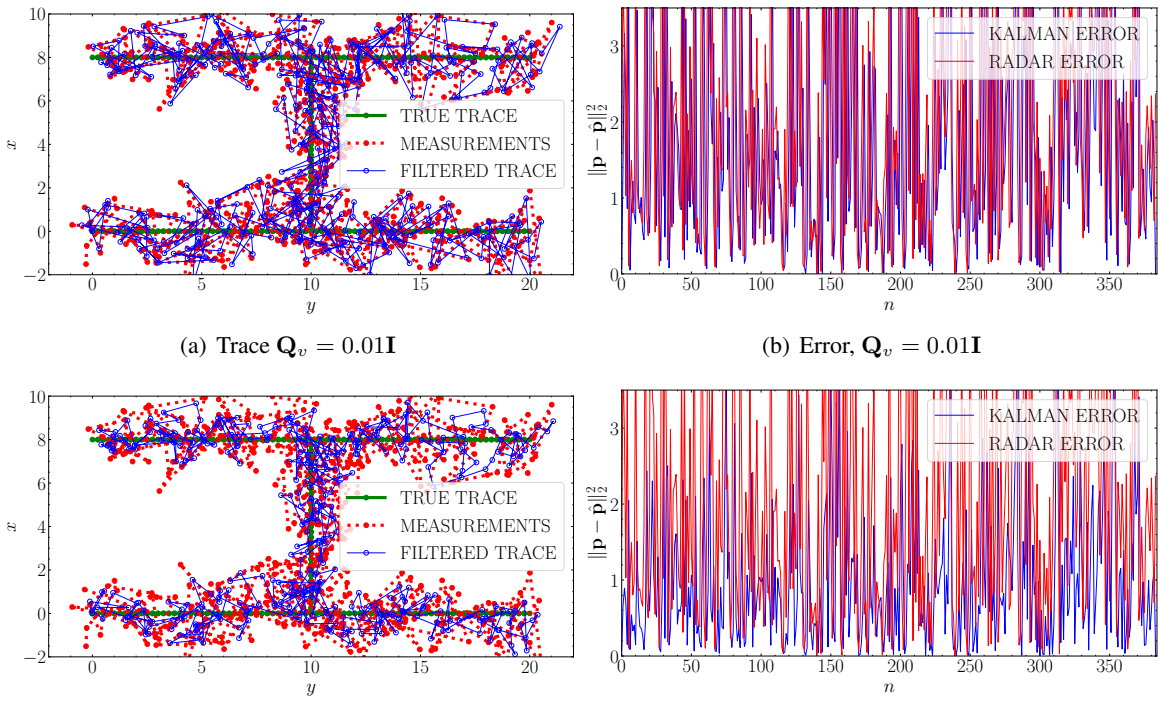
Figure 8: [Colour online] Figures show the Kalman filter prediction and RADAR measurements compared to the true trace. The first column shows the traces and the second column shows the errors, for different model noise, under measurements with $\mathbf{Q}_v = \mathbf{I}$.

attempts to predict paths that are parallel to the axes, as not much model mismatch is allowed. Hence, due to the high measurement noise, the tracking is poor. This is clear with the Kalman filter error being much higher

than the RADAR error. With $\mathbf{Q}_v = 0.01\mathbf{I}, 0.1\mathbf{I}$, the tracking improves as some model mismatch is allowed. The errors in the Kalman filter predictions are reasonable compared to the error in the RADAR measurements. Large errors are seen at time instants where the vehicle takes a turn as smooth turns are not accounted in the model. With $\mathbf{Q}_v = \mathbf{I}$, the model error is large and accommodates measurements including the noise in the measurements. The trace follows the measurements and captures the noise in the measurements. The errors in the Kalman filter predictions compare similar to the errors in the RADAR measurements. This is complete opposite to the case where $\mathbf{Q}_v = 0.0001\mathbf{I}$. The model noise trades between fitting the measurements and obeying the model. The choice $\mathbf{Q}_v = 0.01\mathbf{I}$ is a good choice for model noise as it trades well between obeying the model and denoising the measurements. This is clear by comparing the error in the Kalman filter predictions to the error in the RADAR measurements. In all cases, the error at time instants where the vehicle turns is large, as the model does not account for such motions. The choice $\mathbf{Q}_v = 0.01\mathbf{I}$ gives the least error amongst these choices, for all other time instants.

2.2.4 Kalman filter with parameter mismatch

We implement the Kalman filter using both position and velocity measurements using the state model in (23) and measurement model as in (28). We consider RADAR measurements of both position and velocity with measurement noise $\mathbf{Q}_v = \mathbf{I}$. We fix the model noise with $\mathbf{Q}_w = 0.1\mathbf{I}$ and consider Kalman filter predictions with model mismatch, taking $\mathbf{Q}_v \in \{0.01\mathbf{I}, 0.1\mathbf{I}\}$. We initialise the Kalman filter with $\hat{x}[0|0] = \mathbf{0}$, and determine the tracking performance using the squared-error between the Kalman filter predictions of position ($\hat{\mathbf{p}}[n]$) and the true trace ($\mathbf{p}[n]$), and compare it to the squared error between the RADAR measurements and the true trace.



(a) Trace $\mathbf{Q}_v = 0.01\mathbf{I}$

(b) Error, $\mathbf{Q}_v = 0.01\mathbf{I}$

(c) Trace $\mathbf{Q}_v = 0.1\mathbf{I}$

(d) Error, $\mathbf{Q}_v = 0.1\mathbf{I}$

Figure 9: [Colour online] Figures show the Kalman filter prediction and RADAR measurements compared to the true trace. The first column shows the traces and the second column shows the errors, for different measurement noise (mismatched with the true parameter), under model with $\mathbf{Q}_w = 0.1\mathbf{I}$.

Figure 9 shows the trace and errors for the two cases of measurement noise. We observe that the tracking is poor in both cases. The Kalman filter updates, in Kalman gain, depend on the measurement statistics. The case

where $\mathbf{Q}_v = 0.1\mathbf{I}$ has lower error in the Kalman filter predictions as the mismatch in the parameter is smaller than the case where $\mathbf{Q}_v = 0.01\mathbf{I}$.

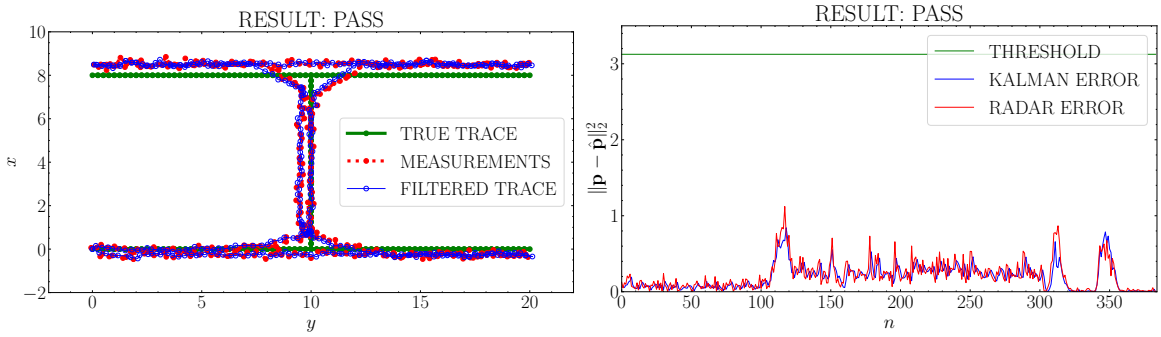
As seen in Section 2.2.3, the Kalman filter gives reliable tracking performance in presence of high measurement noise, if the model parameters are tuned well and the measurement statistics are known. If the measurement statistics are not known well, as seen in Figure 9, the tracking performance will not be reliable, even in the case where the right model noise parameter is chosen from the results in Section 2.2.3.

2.2.5 Kalman filter for automated driving test

We consider automating driving test using a Kalman filter with state model in (23) and measurement model as in (28). We consider the test measurements and assume measurement noise $\mathbf{Q}_v = 0.1\mathbf{I}$, and choose the Kalman filter model with $\mathbf{Q}_w = 0.1\mathbf{I}$. We track the vehicle using the Kalman filter predictions and flag the vehicle to fail the test if the Kalman filter prediction has an error larger than some threshold.

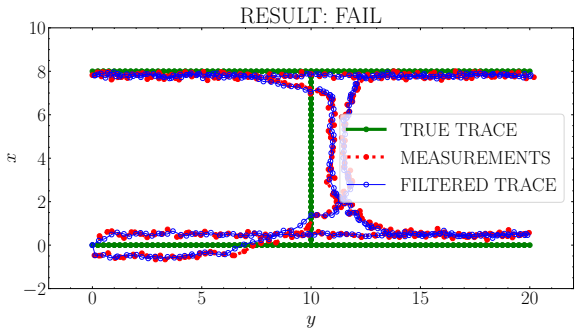
We want the vehicle to keep within the width of the track, i.e., we want $|p_x[n] - \hat{p}_x[n]| < 1.25$ and $|p_y[n] - \hat{p}_y[n]| < 1.25, \ \forall n$. This gives, $\|\hat{\mathbf{p}}[n] - \mathbf{p}[n]\|_2^2 \leq 3.125$. Therefore, the test flags "FAIL" if the Kalman filter prediction error is more than $\tau = 3.125$ at any time instant, otherwise the test says "PASS".

Figure 10 shows the trace and errors for the four test cases. We observe the first, third and fourth cases "PASS", but the second case "FAILS". The error in the "PASS" cases never exceeds the threshold at any time instant, but the error in the second case exceeds the threshold over some intervals. This is also seen in the trace where the vertical section of the path is wrongly driven while the other cases drive approximately well.

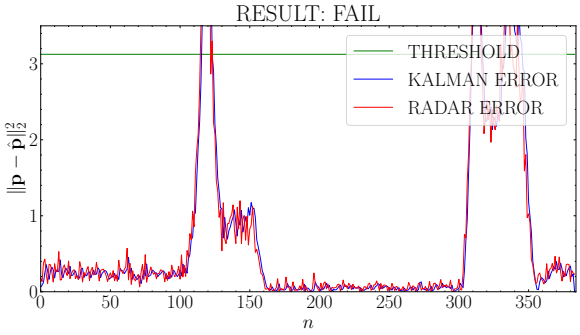


(a) Trace, Data 1

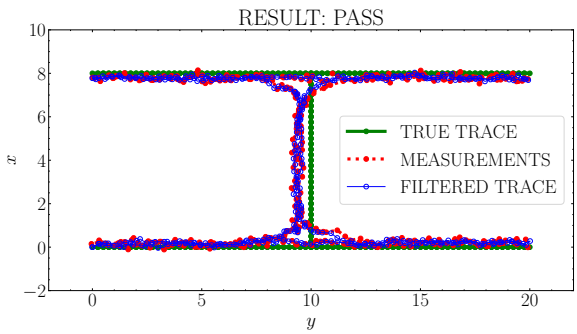
(b) Error, Data 1



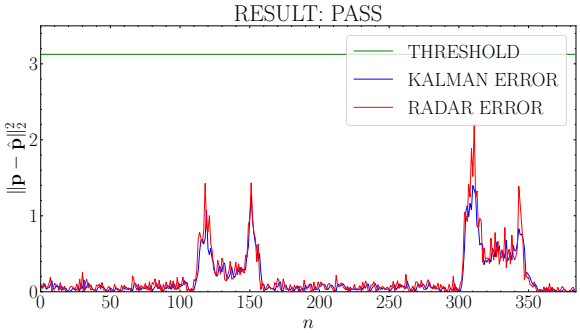
(c) Trace, Data 2



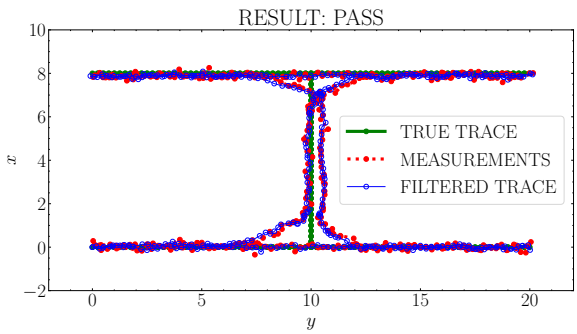
(d) Error, Data 2



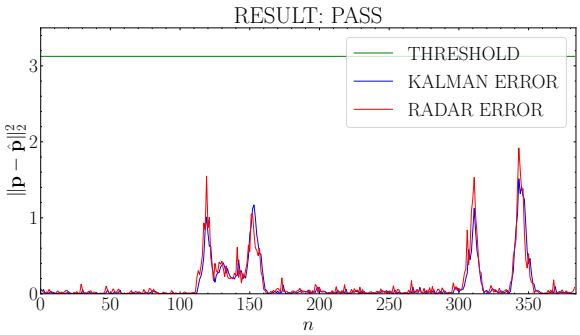
(e) Trace, Data 3



(f) Error, Data 3



(g) Trace, Data 4



(h) Error, Data 4

Figure 10: [Colour online] Figures show the Kalman filter prediction and RADAR measurements compared to the true trace. The first column shows the traces and the second column shows the errors, for different test cases. The green line in the error plots depicts the threshold on the error to flag the test. The titles of the plot gives the result.

Scripts

The Python3 scripts to generate all figures can be downloaded from the GitHub repository `https://github.com/kamath-abhijith/Vehicle_Tracking`. Use `requirements.txt` to install all dependencies, and the bash scripts to run the Python3 codes. Also, see the following code snippets for reference.

Implementation of D_{entry}

The following script generates Figure 2, 3. The relevant functions are in `utils.py`.

```
'''

CONSTANT DETECTION RATE NEYMANN-PEARSON
DETECTOR FOR VEHICLE DETECTION BASED ON
SIGNAL MEAN

AUTHOR: ABIJITH J. KAMATH
abijithj@iisc.ac.in, kamath-abhijith.github.io

'''

# %% LOAD LIBRARIES

import os
import argparse
import numpy as np

from tqdm import tqdm

from scipy.stats import norm

from matplotlib import style
from matplotlib import rcParams
from matplotlib import pyplot as plt

import utils

# %% PARSE ARGUMENTS
parser = argparse.ArgumentParser(
    description = "CDAR DETECTOR AT THE ENTRY"
)

parser.add_argument('--Bt', help="ambient light", type=float, default=0.1)

args = parser.parse_args()
ambient_mean = args.Bt

# %% PLOT SETTINGS

plt.style.use(['science','ieee'])

plt.rcParams.update({
    "font.family": "serif",
    "font.serif": ["cm"],
    "mathtext.fontset": "cm",
    "font.size": 24})

# %% PARAMETERS

N = 5
```

```
M = 1
NUM_STATS = 10000

noise_var = 1
# ambient_mean = 0.1
est_ambient_mean = 0.6
dc = 1

# %% MONTE CARLO SIMULATIONS

Ms = np.arange(1,4)
true_PD = np.linspace(0.01, 0.99, 100)
true_PFA = np.zeros((len(true_PD), len(Ms)))
est_PFA = np.zeros((len(true_PD), len(Ms)))
est_PD = np.zeros((len(true_PD), len(Ms)))

for itr, PD in tqdm(enumerate(true_PD)):
    for _, M in enumerate(Ms):
        threshold = np.sqrt(noise_var/(N*M))*norm.isf(1-PD) + est_ambient_mean

        stats_H0 = utils.mean_stat_H0(NUM_STATS, ambient_mean=ambient_mean, M=M)
        stats_H1 = utils.mean_stat_H1(NUM_STATS, ambient_mean=ambient_mean, M=M)

        false_alarms = sum(stats_H0 < threshold)
        detections = sum(stats_H1 < threshold)

        est_PFA[itr, M-1] = false_alarms / NUM_STATS
        est_PD[itr, M-1] = detections / NUM_STATS

        true_PFA[itr, M-1] = 1-norm.sf((threshold-est_ambient_mean-dc)/np.sqrt(noise_var
    /(N*M)))

# %% PLOTS

os.makedirs('./../results/CDR/', exist_ok=True)
path = './../results/CDR/'

plt.figure(figsize=(8,8))
ax = plt.gca()
utils.plot_signal(true_PD, true_PD, ax=ax, plot_colour='green', show=False)
utils.plot_signal(true_PD, est_PD[:,0], ax=ax, plot_colour='red', line_width=4,
    legend_label=r'$M=1$', show=False)
utils.plot_signal(true_PD, est_PD[:,1], ax=ax, plot_colour='magenta', line_width=4,
    legend_label=r'$M=2$', show=False)
utils.plot_signal(true_PD, est_PD[:,2], ax=ax, plot_colour='blue', line_width=4,
    title_text=r'$D_{\text{entry}}$, $B_t=%.1f$'%(ambient_mean), legend_loc='lower right'
    ,
    legend_label=r'$M=3$', yaxis_label=r'$\hat{P}_{D}$', xaxis_label=r'$\beta$',
    show=False, save=path+'CDR_PD_Bt_'+str(ambient_mean))

plt.figure(figsize=(8,8))
ax = plt.gca()
utils.plot_signal(true_PFA[:,0], true_PD, ax=ax, plot_colour='red',
    line_style='--', legend_label=r'$M=1$', show=False)
utils.plot_signal(true_PFA[:,1], true_PD, ax=ax, plot_colour='magenta',
    line_style='--', legend_label=r'$M=2$', show=False)
utils.plot_signal(true_PFA[:,2], true_PD, ax=ax, plot_colour='blue',
    line_style='--', legend_label=r'$M=3$', show=False)
utils.plot_signal(est_PFA[:,0], est_PD[:,0], ax=ax, plot_colour='red', line_width=4,
    legend_label=r'$M=1$', show=False)
utils.plot_signal(est_PFA[:,1], est_PD[:,1], ax=ax, plot_colour='magenta', line_width=4,
```

```
    legend_label=r'$M=2$', show=False)
utils.plot_signal(est_PFA[:,2], est_PD[:,2], ax=ax, plot_colour='blue', line_width=4,
    legend_label=r'$M=3$', legend_loc='lower right', legend_ncol=2,
    title_text=r'ROC OF $D_{\text{entry}}$, $B_t=%.1f$'%(ambient_mean),
    yaxis_label=r'$P_{D}$', xaxis_label=r'$P_{FA}$', show=False,
    save=path+'CDR_ROC_Bt_'+str(ambient_mean))

# %%
```

Implementation of D_{exit}

The following script generates Figure 4, 5. The relevant functions are in `utils.py`.

```
'''

CONSTANT FALSE_ALARM RATE NEYMANN-PEARSON
DETECTOR FOR VEHICLE DETECTION BASED ON
SIGNAL MEAN

AUTHOR: ABIJITH J. KAMATH
abijithj@iisc.ac.in, kamath-abhijith.github.io

'''

# %% LOAD LIBRARIES

import os
import argparse
import numpy as np

from tqdm import tqdm

from scipy.stats import norm

from matplotlib import style
from matplotlib import rcParams
from matplotlib import pyplot as plt

import utils

# %% PARSE ARGUMENTS
parser = argparse.ArgumentParser(
    description = "CFAR DETECTOR AT THE EXIT"
)

parser.add_argument('--Bt', help="ambient light", type=float, default=0.1)

args = parser.parse_args()
ambient_mean = args.Bt

# %% PLOT SETTINGS

plt.style.use(['science','ieee'])

plt.rcParams.update({
    "font.family": "serif",
    "font.serif": ["cm"],
    "mathtext.fontset": "cm",
    "font.size": 24})

```

```
# %% PARAMETERS

N = 5
M = 1
NUM_STATS = 10000

noise_var = 1
# ambient_mean = 0.6
est_ambient_mean = 0.1
dc = 1

# %% MONTE CARLO SIMULATIONS

Ms = np.arange(1,4)
true_PFA = np.linspace(0.01, 0.99, 100)
true_PD = np.zeros((len(true_PFA), len(Ms)))
est_PFA = np.zeros((len(true_PFA), len(Ms)))
est_PD = np.zeros((len(true_PFA), len(Ms)))

for itr, PFA in tqdm(enumerate(true_PFA)):
    for _, M in enumerate(Ms):
        threshold = np.sqrt(noise_var/(N*M))*norm.isf(1-PFA) + est_ambient_mean + dc

        stats_H0 = utils.mean_stat_H0(NUM_STATS, ambient_mean=ambient_mean, M=M)
        stats_H1 = utils.mean_stat_H1(NUM_STATS, ambient_mean=ambient_mean, M=M)

        false_alarms = sum(stats_H0 < threshold)
        detections = sum(stats_H1 < threshold)

        est_PFA[itr, M-1] = false_alarms / NUM_STATS
        est_PD[itr, M-1] = detections / NUM_STATS

        true_PD[itr, M-1] = 1 - norm.sf((threshold-est_ambient_mean)/np.sqrt(noise_var/(N
    *M)))

# %% PLOTS

os.makedirs('./../results/CFAR/', exist_ok=True)
path = './../results/CFAR/'

plt.figure(figsize=(8,8))
ax = plt.gca()
utils.plot_signal(true_PFA, true_PFA, ax=ax, plot_colour='green', show=False)
utils.plot_signal(true_PFA, est_PFA[:,0], ax=ax, plot_colour='red', line_width=4,
    legend_label=r'$M=1$', show=False)
utils.plot_signal(true_PFA, est_PFA[:,1], ax=ax, plot_colour='magenta', line_width=4,
    legend_label=r'$M=2$', show=False)
utils.plot_signal(true_PFA, est_PFA[:,2], ax=ax, plot_colour='blue', line_width=4,
    title_text=r'$D_{\text{exit}}$, $B_t=%.1f$'%(ambient_mean), legend_loc='upper left',
    legend_label=r'$M=3$', yaxis_label=r'$\hat{P}_{FA}$', xaxis_label=r'$\alpha$',
    show=False, save=path+'CFAR_PFA_Bt_'+str(ambient_mean))

plt.figure(figsize=(8,8))
ax = plt.gca()
utils.plot_signal(true_PD[:,0], true_PFA, ax=ax, plot_colour='red',
    line_style='--', legend_label=r'$M=1$', show=False)
utils.plot_signal(true_PD[:,1], true_PFA, ax=ax, plot_colour='magenta',
    line_style='--', legend_label=r'$M=2$', show=False)
utils.plot_signal(true_PD[:,2], true_PFA, ax=ax, plot_colour='blue',
    line_style='--', legend_label=r'$M=3$', show=False)
utils.plot_signal(est_PD[:,0], est_PFA[:,0], ax=ax, plot_colour='red', line_width=4,
```

```
    legend_label=r'$M=1$', show=False)
utils.plot_signal(est_PD[:,1], est_PFA[:,1], ax=ax, plot_colour='magenta', line_width=4,
    legend_label=r'$M=2$', show=False)
utils.plot_signal(est_PD[:,2], est_PFA[:,2], ax=ax, plot_colour='blue', line_width=4,
    legend_label=r'$M=3$', legend_loc='upper left', legend_ncol=2,
    title_text=r'ROC OF $D_{\text{exit}}$, $B_t=%.1f$'%(ambient_mean),
    yaxis_label=r'$P_{FA}$', xaxis_label=r'$P_{D}$', show=False,
    save=path+'CFAR_ROC_Bt_'+str(ambient_mean))

# %%
```

Implementation of Kalman filter with velocity-only measurements

The following script generates Figure 6. The relevant functions are in `utils.py`.

```
'''

KALMAN FILTER FOR POSITION ESTIMATION
USING VELOCITY MEASUREMENTS

AUTHOR: ABIJITH J. KAMATH
abijithj@iisc.ac.in

'''

# %% LOAD LIBRARIES

import os
import argparse
import numpy as np

from tqdm import tqdm

from scipy import io
from scipy.stats import norm

from matplotlib import style
from matplotlib import rcParams
from matplotlib import pyplot as plt

import utils

# %% PARSE ARGUMENTS
parser = argparse.ArgumentParser(
    description = "KALMAN FITLER WITH VEOLCITY MEASUREMENTS"
)

parser.add_argument('--init', help="initial position", default='bias')
parser.add_argument('--noise_std', help="initial position", type=float, default=0.1)

args = parser.parse_args()
init = args.init
noise_std = args.noise_std

# %% PLOT SETTINGS

plt.style.use(['science','ieee'])

plt.rcParams.update({
    "font.family": "serif",
```

```
    "font.serif": ["cm"],
    "mathtext.fontset": "cm",
    "font.size": 24})

# %% LOAD DATA

true_data = io.loadmat('./../dataset/trace_ideal.mat')
sample_data = io.loadmat('./../dataset/trace_1.mat')
radar_data = io.loadmat('./../dataset/Radar_med.mat')

true_trace = true_data['true_trace']
sample_trace = sample_data['x']
measurements = radar_data['y']

# %% TRACKING PARAMETERS

# Initialise variables
state_dim, num_points = true_trace.shape
update_state = np.zeros((state_dim, num_points))
update_statecov = np.zeros((state_dim, state_dim, num_points))
predict_state = np.zeros((state_dim, num_points))
predict_statecov = np.zeros((state_dim, state_dim, num_points))

# Define transition matrices
time_step = 0.1
state_mat = np.matrix([[1, 0, time_step, 0],
                       [0, 1, 0, time_step],
                       [0, 0, 1, 0],
                       [0, 0, 0, 1]])
meas_mat = np.eye(state_dim)[2:,:]

# Define noise covariances
# noise_std = 0.0
state_noise_cov = noise_std*np.eye(state_dim)
meas_noise_cov = 0.1*np.eye(state_dim-2)

# %% TRACKING

# Initialisation
if init == 'bias':
    update_state[:,0] = np.array([1.0, 0.0, 0.0, 0.0])
elif init == 'exact':
    update_state[:,0] = np.array([0.0, 0.0, 0.0, 0.0])
update_statecov[:,:,0] = 0.1*np.eye(state_dim)

# Kalman filtering
for idx in tqdm(range(num_points-1)):

    predict_state[:,idx], predict_statecov[:,:,idx] = utils.kf_predict(\
        update_state[:,idx], state_mat, update_statecov[:,:,idx],
        state_noise_cov)

    update_state[:,idx+1], update_statecov[:,:,idx+1] = utils.kf_update(\
        measurements[2:,idx], predict_state[:,idx], predict_statecov[:,:,idx],
        meas_mat, meas_noise_cov)

# %% ERROR

kalman_error = np.linalg.norm(predict_state[:2,:] - true_trace[:2,:], axis=0)**2
radar_error = np.linalg.norm(measurements[:2,:] - true_trace[:2,:], axis=0)**2

```

```
# %% PLOTS

os.makedirs('./../results/KF/ex1/', exist_ok=True)
path = './../results/KF/ex1/'

plt.figure(figsize=(12,6))
ax = plt.gca()
utils.plot_trace(true_trace, ax=ax, plot_colour='green', line_style='-',
    legend_label=r'TRUE TRACE', show=False)
utils.plot_trace(measurements, ax=ax, plot_colour='red', line_style='dotted',
    legend_label=r'MEASUREMENTS', show=False)
utils.plot_trace(predict_state, ax=ax, plot_colour='blue', line_style=None,
    line_width=1, fill_style='none', legend_label=r'FILTERED TRACE',
    show=False, save=path+'KF_Vel_Trace_Init_'+str(init)+'_noise_'+str(noise_std))

plt.figure(figsize=(12,6))
ax = plt.gca()
utils.plot_signal(np.arange(num_points), kalman_error, ax=ax,
    plot_colour='blue', legend_label=r'KALMAN ERROR', show=False)
utils.plot_signal(np.arange(num_points), radar_error, ax=ax,
    xlimits=[0,num_points], ylimits=[0,3.5], plot_colour='red',
    xaxis_label=r'$n$', yaxis_label=r'$\Vert\mathbf{p}-\hat{\mathbf{p}}\Vert_2^2$',
    legend_label=r'RADAR ERROR', show=False,
    save=path+'KF_Vel_Errors_Init_'+str(init)+'_noise_'+str(noise_std))

# %%
```

Implementation of Kalman filter with position and velocity measurements

The following script generates Figure 7, 8. The relevant functions are in `utils.py`.

```
'''

KALMAN FILTER FOR POSITION ESTIMATION
USING POSITION AND VELOCITY MEASUREMENTS

AUTHOR: ABIJITH J. KAMATH
abijithj@iisc.ac.in

'''

# %% LOAD LIBRARIES

import os
import argparse
import numpy as np

from tqdm import tqdm

from scipy import io
from scipy.stats import norm

from matplotlib import style
from matplotlib import rcParams
from matplotlib import pyplot as plt

import utils

# %% PARSE ARGUMENTS
parser = argparse.ArgumentParser(
```

```
    description = "KALMAN FITLER WITH POSITION AND VELOCITY MEASUREMENTS"
)

parser.add_argument('--data', help="dataset", default='med')
parser.add_argument('--noise_std', help="model noise", type=float, default=0.1)

args = parser.parse_args()
dataset = args.data
noise_std = args.noise_std

# %% PLOT SETTINGS

plt.style.use(['science','ieee'])

plt.rcParams.update({
    "font.family": "serif",
    "font.serif": ["cm"],
    "mathtext.fontset": "cm",
    "font.size": 24})

# %% LOAD DATA

true_data = io.loadmat('./../dataset/trace_ideal.mat')
sample_data = io.loadmat('./../dataset/trace_1.mat')
if dataset == 'med':
    radar_data = io.loadmat('./../dataset/Radar_med.mat')
    meas_noise = 0.1
if dataset == 'high':
    radar_data = io.loadmat('./../dataset/Radar_high.mat')
    meas_noise = 1.0

true_trace = true_data['true_trace']
sample_trace = sample_data['x']
measurements = radar_data['y']

# %% TRACKING PARAMETERS

# Initialise variables
state_dim, num_points = true_trace.shape
update_state = np.zeros((state_dim, num_points))
update_statecov = np.zeros((state_dim, state_dim, num_points))
predict_state = np.zeros((state_dim, num_points))
predict_statecov = np.zeros((state_dim, state_dim, num_points))

# Define transition matrices
time_step = 0.1
state_mat = np.zeros((state_dim, state_dim, num_points))
for idx in range(num_points):
    if idx<120 or (idx>152 and idx<312) or idx>344:
        state_mat[:,:,idx] = np.matrix([[1, 0, 0, 0],
                                        [0, 1, 0, time_step],
                                        [0, 0, 0, 0],
                                        [0, 0, 0, 1]])
    elif idx==120 or idx==312:
        state_mat[:,:,idx] = np.matrix([[1, 0, 0, 0],
                                        [0, 1, 0, time_step],
                                        [0, 0, 0, -1],
                                        [0, 0, 0, 0]])
    elif idx==152 or idx==344:
        state_mat[:,:,idx] = np.matrix([[1, 0, time_step, 0],
                                        [0, 1, 0, 0],
```

```
                                        [0, 0, 0, 0],
                                        [0, 0, -1, 0]])
    elif (idx>120 and idx<152) or (idx>312 and idx<344):
        state_mat[:,:,idx] = np.matrix([[1, 0, time_step, 0],
                                        [0, 1, 0, 0],
                                        [0, 0, 1, 0],
                                        [0, 0, 0, 0]])
meas_mat = np.eye(state_dim)

# Define noise covariances
# noise_std = 0.0001
state_noise_cov = noise_std*np.eye(state_dim)
meas_noise_cov = meas_noise*np.eye(state_dim)

# %% TRACKING

# Initialisation
update_state[:,0] = np.array([0.0, 0.0, 0.0, 0.0])
update_statecov[:,:,0] = 0.0*np.eye(state_dim)

# Kalman filtering
for idx in tqdm(range(num_points-1)):

    predict_state[:,idx], predict_statecov[:,:,idx] = utils.kf_predict(\
        update_state[:,idx], state_mat[:,:,idx], update_statecov[:,:,idx],
        state_noise_cov)

    update_state[:,idx+1], update_statecov[:,:,idx+1] = utils.kf_update(\
        measurements[:,idx], predict_state[:,idx], predict_statecov[:,:,idx],
        meas_mat, meas_noise_cov)

# %% ERROR

kalman_error = np.linalg.norm(predict_state[:2,:] - true_trace[:2,:], axis=0)**2
radar_error = np.linalg.norm(measurements[:2,:] - true_trace[:2,:], axis=0)**2

# %% PLOTS

if dataset == 'med':
    os.makedirs('./../results/KF/ex2/', exist_ok=True)
    path = './../results/KF/ex2/'
if dataset == 'high':
    os.makedirs('./../results/KF/ex3/', exist_ok=True)
    path = './../results/KF/ex3/'

plt.figure(figsize=(12,6))
ax = plt.gca()
utils.plot_trace(true_trace, ax=ax, plot_colour='green', line_style='-',
    legend_label=r'TRUE TRACE', show=False)
utils.plot_trace(measurements, ax=ax, plot_colour='red', line_style='dotted',
    legend_label=r'MEASUREMENTS', show=False)
utils.plot_trace(predict_state, ax=ax, plot_colour='blue', line_style=None,
    line_width=1, fill_style='none', legend_label=r'FILTERED TRACE', show=False,
    save=path+'KF_Pos_Trace_Data_'+str(dataset)+'_noise_'+str(noise_std))

plt.figure(figsize=(12,6))
ax = plt.gca()
utils.plot_signal(np.arange(num_points), kalman_error, ax=ax,
    plot_colour='blue', legend_label=r'KALMAN ERROR', show=False)
utils.plot_signal(np.arange(num_points), radar_error, ax=ax,
    xlimits=[0,num_points], ylimits=[0,3.5], plot_colour='red',
```

```
    xaxis_label=r'$n$', yaxis_label=r'$\Vert\mathbf{p}-\hat{\mathbf{p}}\Vert_2^2$',
    legend_label=r'RADAR ERROR', show=False,
    save=path+'KF_Pos_Errors_Data_'+str(dataset)+'_noise_'+str(noise_std))

# %%
```

Implementation of Kalman filter with mismatched parameters

The following script generates Figure 9. The relevant functions are in `utils.py`.

```
'''

KALMAN FILTER FOR POSITION ESTIMATION
USING POSITION AND VELOCITY MEASUREMENTS
WITH MISMATCHED PARAMETERS

AUTHOR: ABIJITH J. KAMATH
abijithj@iisc.ac.in

'''

# %% LOAD LIBRARIES

import os
import argparse
import numpy as np

from tqdm import tqdm

from scipy import io
from scipy.stats import norm

from matplotlib import style
from matplotlib import rcParams
from matplotlib import pyplot as plt

import utils

# %% PARSE ARGUMENTS
parser = argparse.ArgumentParser(
    description = "KALMAN FITLER WITH POSITION AND VELOCITY MEASUREMENTS"
)

parser.add_argument('--meas_std', help="measurement noise", type=float, default=0.1)

args = parser.parse_args()
meas_noise = args.meas_std

# %% PLOT SETTINGS

plt.style.use(['science','ieee'])

plt.rcParams.update({
    "font.family": "serif",
    "font.serif": ["cm"],
    "mathtext.fontset": "cm",
    "font.size": 24})

# %% LOAD DATA

```

```
true_data = io.loadmat('./../dataset/trace_ideal.mat')
sample_data = io.loadmat('./../dataset/trace_1.mat')
radar_data = io.loadmat('./../dataset/Radar_high.mat')
# meas_noise = 1.0

true_trace = true_data['true_trace']
sample_trace = sample_data['x']
measurements = radar_data['y']

# %% TRACKING PARAMETERS

# Initialise variables
state_dim, num_points = true_trace.shape
update_state = np.zeros((state_dim, num_points))
update_statecov = np.zeros((state_dim, state_dim, num_points))
predict_state = np.zeros((state_dim, num_points))
predict_statecov = np.zeros((state_dim, state_dim, num_points))

# Define transition matrices
time_step = 0.1
state_mat = np.zeros((state_dim, state_dim, num_points))
for idx in range(num_points):
    if idx<120 or (idx>152 and idx<312) or idx>344:
        state_mat[:,:,idx] = np.matrix([[1, 0, 0, 0],
                                        [0, 1, 0, time_step],
                                        [0, 0, 0, 0],
                                        [0, 0, 0, 1]])
    elif idx==120 or idx==312:
        state_mat[:,:,idx] = np.matrix([[1, 0, 0, 0],
                                        [0, 1, 0, time_step],
                                        [0, 0, 0, -1],
                                        [0, 0, 0, 0]])
    elif idx==152 or idx==344:
        state_mat[:,:,idx] = np.matrix([[1, 0, time_step, 0],
                                        [0, 1, 0, 0],
                                        [0, 0, 0, 0],
                                        [0, 0, -1, 0]])
    elif (idx>120 and idx<152) or (idx>312 and idx<344):
        state_mat[:,:,idx] = np.matrix([[1, 0, time_step, 0],
                                        [0, 1, 0, 0],
                                        [0, 0, 1, 0],
                                        [0, 0, 0, 0]])
meas_mat = np.eye(state_dim)

# Define noise covariances
noise_std = 0.1
state_noise_cov = noise_std*np.eye(state_dim)
meas_noise_cov = meas_noise*np.eye(state_dim)

# %% TRACKING

# Initialisation
update_state[:,0] = np.array([0.0, 0.0, 0.0, 0.0])
update_statecov[:,:,0] = 0.0*np.eye(state_dim)

# Kalman filtering
for idx in tqdm(range(num_points-1)):

    predict_state[:,idx], predict_statecov[:,:,idx] = utils.kf_predict(\
        update_state[:,idx], state_mat[:,:,idx], update_statecov[:,:,idx],
        state_noise_cov)
```

```
    update_state[:,idx+1], update_statecov[:,:,idx+1] = utils.kf_update(\
        measurements[:,idx], predict_state[:,idx], predict_statecov[:,:,idx],
        meas_mat, meas_noise_cov)

# %% ERROR

kalman_error = np.linalg.norm(predict_state[:2,:] - true_trace[:2,:], axis=0)**2
radar_error = np.linalg.norm(measurements[:2,:] - true_trace[:2,:], axis=0)**2

# %% PLOTS

os.makedirs('./../results/KF/ex4/', exist_ok=True)
path = './../results/KF/ex4/'

plt.figure(figsize=(12,6))
ax = plt.gca()
utils.plot_trace(true_trace, ax=ax, plot_colour='green', line_style='-',
    legend_label=r'TRUE TRACE', show=False)
utils.plot_trace(measurements, ax=ax, plot_colour='red', line_style='dotted',
    legend_label=r'MEASUREMENTS', show=False)
utils.plot_trace(predict_state, ax=ax, plot_colour='blue', line_style=None,
    line_width=1, fill_style='none', legend_label=r'FILTERED TRACE', show=False,
    save=path+'KF_Pos_Trace'+'_noise_'+str(meas_noise))

plt.figure(figsize=(12,6))
ax = plt.gca()
utils.plot_signal(np.arange(num_points), kalman_error, ax=ax,
    plot_colour='blue', legend_label=r'KALMAN ERROR', show=False)
utils.plot_signal(np.arange(num_points), radar_error, ax=ax,
    xlimits=[0,num_points], ylimits=[0,3.5], plot_colour='red',
    xaxis_label=r'$n$', yaxis_label=r'$\Vert\mathbf{p}-\hat{\mathbf{p}}\Vert_2^2$',
    legend_label=r'RADAR ERROR', show=False,
    save=path+'KF_Pos_Errors'+'_noise_'+str(meas_noise))

# %%
```

Implementation of Kalman filter to automate driving tests

The following script generates Figure 10. The relevant functions are in `utils.py`.

```
'''

KALMAN FILTER FOR POSITION ESTIMATION
USING POSITION AND VELOCITY MEASUREMENTS
FOR AUTOMATED DRIVING TEST

AUTHOR: ABIJITH J. KAMATH
abijithj@iisc.ac.in

'''

# %% LOAD LIBRARIES

import os
import argparse
import numpy as np

from tqdm import tqdm

```

```
from scipy import io
from scipy.stats import norm

from matplotlib import style
from matplotlib import rcParams
from matplotlib import pyplot as plt

import utils

# %% PARSE ARGUMENTS
parser = argparse.ArgumentParser(
    description = "KALMAN FITLER WITH POSITION AND VELOCITY MEASUREMENTS"
)

parser.add_argument('--data', help="dataset", type=int, default=1)

args = parser.parse_args()
dataset = args.data

# %% PLOT SETTINGS

plt.style.use(['science','ieee'])

plt.rcParams.update({
    "font.family": "serif",
    "font.serif": ["cm"],
    "mathtext.fontset": "cm",
    "font.size": 24})

# %% LOAD DATA

true_data = io.loadmat('./../dataset/trace_ideal.mat')
sample_data = io.loadmat('./../dataset/trace_1.mat')
radar_data = io.loadmat('./../dataset/Test'+str(dataset)+'.mat')
meas_noise = 0.1

true_trace = true_data['true_trace']
sample_trace = sample_data['x']
measurements = radar_data['radar_measurement']

# %% TRACKING PARAMETERS

# Initialise variables
state_dim, num_points = true_trace.shape
update_state = np.zeros((state_dim, num_points))
update_statecov = np.zeros((state_dim, state_dim, num_points))
predict_state = np.zeros((state_dim, num_points))
predict_statecov = np.zeros((state_dim, state_dim, num_points))

# Define transition matrices
time_step = 0.1
state_mat = np.zeros((state_dim, state_dim, num_points))
for idx in range(num_points):
    if idx<120 or (idx>152 and idx<312) or idx>344:
        state_mat[:,:,idx] = np.matrix([[1, 0, 0, 0],
                                        [0, 1, 0, time_step],
                                        [0, 0, 0, 0],
                                        [0, 0, 0, 1]])
    elif idx==120 or idx==312:
        state_mat[:,:,idx] = np.matrix([[1, 0, 0, 0],
                                        [0, 1, 0, time_step],
```

```
                                        [0, 0, 0, -1],
                                        [0, 0, 0, 0]])
    elif idx==152 or idx==344:
        state_mat[:,:,idx] = np.matrix([[1, 0, time_step, 0],
                                        [0, 1, 0, 0],
                                        [0, 0, 0, 0],
                                        [0, 0, -1, 0]])
    elif (idx>120 and idx<152) or (idx>312 and idx<344):
        state_mat[:,:,idx] = np.matrix([[1, 0, time_step, 0],
                                        [0, 1, 0, 0],
                                        [0, 0, 1, 0],
                                        [0, 0, 0, 0]])
meas_mat = np.eye(state_dim)

# Define noise covariances
noise_std = 0.1
state_noise_cov = noise_std*np.eye(state_dim)
meas_noise_cov = meas_noise*np.eye(state_dim)

# %% TRACKING

# Initialisation
update_state[:,0] = np.array([0.0, 0.0, 0.0, 0.0])
update_statecov[:,:,0] = 0.0*np.eye(state_dim)

# Kalman filtering
for idx in tqdm(range(num_points-1)):

    predict_state[:,idx], predict_statecov[:,:,idx] = utils.kf_predict(\
        update_state[:,idx], state_mat[:,:,idx], update_statecov[:,:,idx],
        state_noise_cov)

    update_state[:,idx+1], update_statecov[:,:,idx+1] = utils.kf_update(\
        measurements[:,idx], predict_state[:,idx], predict_statecov[:,:,idx],
        meas_mat, meas_noise_cov)

# %% ERROR

kalman_error = np.linalg.norm(predict_state[:2,:] - true_trace[:2,:], axis=0)**2
radar_error = np.linalg.norm(measurements[:2,:] - true_trace[:2,:], axis=0)**2

threshold = 3.125
if sum(kalman_error>threshold)==0:
    result = r'PASS'
elif sum(kalman_error>threshold)>0:
    result = r'FAIL'


# %% PLOTS

os.makedirs('./../results/KF/ex5/', exist_ok=True)
path = './../results/KF/ex5/'

plt.figure(figsize=(12,6))
ax = plt.gca()
utils.plot_trace(true_trace, ax=ax, plot_colour='green', line_style='-',
    legend_label=r'TRUE TRACE', show=False)
utils.plot_trace(measurements, ax=ax, plot_colour='red', line_style='dotted',
    legend_label=r'MEASUREMENTS', show=False)
utils.plot_trace(predict_state, ax=ax, plot_colour='blue', line_style=None,
    line_width=1, fill_style='none', legend_label=r'FILTERED TRACE', show=False,
```

```
    title_text=r'RESULT: '+result, save=path+'KF_Pos_Trace_Data_'+str(dataset))

plt.figure(figsize=(12,6))
ax = plt.gca()
utils.plot_signal(np.arange(num_points), threshold*np.ones(num_points), ax=ax,
    plot_colour='green', legend_label=r'THRESHOLD', show=False)
utils.plot_signal(np.arange(num_points), kalman_error, ax=ax,
    plot_colour='blue', legend_label=r'KALMAN ERROR', show=False)
utils.plot_signal(np.arange(num_points), radar_error, ax=ax,
    xlimits=[0,num_points], ylimits=[0,3.5], plot_colour='red',
    xaxis_label=r'$n$', yaxis_label=r'$\Vert\mathbf{p}-\hat{\mathbf{p}}\Vert_2^2$',
    legend_label=r'RADAR ERROR', title_text=r'RESULT: '+result, show=False,
    save=path+'KF_Pos_Errors_Data_'+str(dataset))

# %%
```

utils.py

Use the `make trace video` to generate the video as shown in the presentation.

```
'''

TOOLS FOR VEHICLE DETECTION AND TRACKING

AUTHOR: ABIJITH J. KAMATH
abijithj@iisc.ac.in, kamath-abhijith.github.io

'''

# %% LOAD LIBRARIES

import numpy as np

from matplotlib import pyplot as plt
from celluloid import Camera

from scipy.stats import multivariate_normal

from tqdm import tqdm

# %% PLOTTING FUNCTIONS

def plot_signal(x, y, ax=None, plot_colour='blue', xaxis_label=None,
    yaxis_label=None, title_text=None, legend_label=None, legend_show=True,
    legend_loc='upper right', legend_ncol=1, line_style='-', line_width=None,
    show=False, xlimits=[0,1], ylimits=[0,1], save=None):
    '''
    Plots signal with abscissa in x and ordinates in y

    '''
    if ax is None:
        fig = plt.figure(figsize=(12,6))
        ax = plt.gca()

    plt.plot(x, y, linestyle=line_style, linewidth=line_width, color=plot_colour,
        label=legend_label)
    if legend_label and legend_show:
        plt.legend(ncol=legend_ncol, loc=legend_loc, frameon=True, framealpha=0.8,
            facecolor='white')
    plt.xlabel(xaxis_label)
```

```
    plt.ylabel(yaxis_label)

    plt.xlim(xlimits)
    plt.ylim(ylimits)
    plt.title(title_text)

    if save:
        plt.savefig(save + '.pdf', format='pdf')

    if show:
        plt.show()

    return

def plot_trace(state_var, ax=None, plot_colour='blue', marker='o',
    fill_style='full', xaxis_label=r'$y$', yaxis_label=r'$x$',
    title_text=None, legend_label=None, legend_show=True,
    legend_loc='center right', line_style='-', line_width=4, show=False,
    xlimits=[-2,22], ylimits=[-2,10], save=None):
    '''
    Plots trace using the state variable

    '''
    if ax is None:
        fig = plt.figure(figsize=(12,6))
        ax = plt.gca()

    plt.plot(state_var[1,:], state_var[0,:], linestyle=line_style,
        marker=marker, fillstyle=fill_style, linewidth=line_width,
        color=plot_colour, label=legend_label)

    if legend_label and legend_show:
        plt.legend(loc=legend_loc, frameon=True, framealpha=0.8, facecolor='white')
    plt.xlabel(xaxis_label)
    plt.ylabel(yaxis_label)

    plt.xlim(xlimits)
    plt.ylim(ylimits)
    plt.title(title_text)

    if save:
        plt.savefig(save + '.pdf', format='pdf')

    if show:
        plt.show()

    return

def make_trace_video(true, measurements, filter, xlimits=[-2,22],
    ylimits=[-2,10],
    save=None):
    '''
    Makes video with static true trace and dynamic
    measurements and filter output

    '''

    fig = plt.figure(figsize=(12,6))
    ax = plt.gca()
    camera = Camera(fig)

```

```
    _, num_points = true.shape
    for i in tqdm(range(num_points)):
        ax.plot(true[1,:], true[0,:], c='green', linewidth=4, linestyle='-',
            marker='o')
        ax.plot(measurements[1,:i], measurements[0,:i], c='red', linewidth=4,
            linestyle='dotted', marker='o')
        ax.plot(filter[1,:i], filter[0,:i], c='blue', linewidth=1,
            linestyle=None, marker='o', fillstyle='none')
        camera.snap()

    plt.xlim(xlimits)
    plt.ylim(ylimits)

    animation = camera.animate()
    animation.save('trace.mp4')

    fig = plt.figure(figsize=(12,6))
    ax = plt.gca()
    camera = Camera(fig)

    _, num_points = true.shape
    for i in tqdm(range(num_points)):
        ax.plot(true[2,:], c='green', linewidth=4, linestyle='-',
            marker='o')
        ax.plot(measurements[2,:i], c='red', linewidth=4,
            linestyle='dotted')
        ax.plot(filter[2,:i], c='blue', linewidth=1,
            linestyle=None, fillstyle='none')
        camera.snap()

    plt.xlim([0,num_points])
    plt.ylim([-3,3])

    animation = camera.animate()
    animation.save('xvel.mp4')

    fig = plt.figure(figsize=(12,6))
    ax = plt.gca()
    camera = Camera(fig)

    _, num_points = true.shape
    for i in tqdm(range(num_points)):
        ax.plot(true[3,:], c='green', linewidth=4, linestyle='-',
            marker='o')
        ax.plot(measurements[3,:i], c='red', linewidth=4,
            linestyle='dotted')
        ax.plot(filter[3,:i], c='blue', linewidth=1,
            linestyle=None, fillstyle='none')
        camera.snap()

    plt.xlim([0,num_points])
    plt.ylim([-3,3])

    animation = camera.animate()
    animation.save('yvel.mp4')

    return

# %% STATISTICS

def mean_stat_H0(num_stats, ambient_mean, dc=1, N=5, M=1, noise_var=1):
```

```
    '''
    Generates signal mean statistic for no vehicle hypothesis

    :param num_stats: number of realisations
    :param ambient_mean: mean due to ambient light
    :param dc: dc value due to source
    :param N: number of measurements
    :param M: number of LEDs
    :param noise_var: variance of AWGN

    :return: mean statistics

    '''

    stats = np.zeros(num_stats)
    for itr in range(num_stats):
        noise = np.sqrt(noise_var)*np.random.randn(N*M)

        stats[itr] = np.mean(dc + ambient_mean + noise)

    return stats

def mean_stat_H1(num_stats, ambient_mean, dc=1, N=5, M=1, noise_var=1):
    '''
    Generates signal mean statistic for vehicle present hypothesis

    :param num_stats: number of realisations
    :param ambient_mean: mean due to ambient light
    :param dc: dc value due to source
    :param N: number of measurements
    :param M: number of LEDs
    :param noise_var: variance of AWGN

    :return: mean statistics

    '''

    stats = np.zeros(num_stats)
    for itr in range(num_stats):
        noise = np.sqrt(noise_var)*np.random.randn(N*M)

        stats[itr] = np.mean(ambient_mean + noise)

    return stats

# %% KALMAN FILTER

def kf_predict(state, state_mat, state_cov, noise_cov):
    '''
    Predict state, given past states

    :param state: past state variable
    :param state_mat: past state matrix
    :param state_cov: past covariance of state
    :param noise_var: noise variance

    :returns: predicted state variable, predicted error covariance

    '''

    prediction = np.dot(state_mat, state)
```

```
    covariance = np.dot(np.dot(state_mat, state_cov), state_mat.T) + noise_cov

    return prediction, covariance

def kf_update(meas, state, state_cov, meas_mat, meas_cov):
    '''
    Update the state variable given measurements

    :param meas: measurements
    :param state: state variable
    :param state_cov: error covariance in state
    :param meas_mat: measurement matrix
    :param meas_cov: error covariance in measurements

    :returns: updated state variable, error covariance

    '''

    S = np.dot(np.dot(meas_mat, state_cov), meas_mat.T) + meas_cov
    gain = np.dot(np.dot(state_cov, meas_mat.T), np.linalg.inv(S))
    updated = state + np.dot(gain, meas-np.dot(meas_mat, state))
    covariance = np.dot((np.eye(meas_mat.shape[1]) - np.dot(gain, meas_mat)), state_cov)

    return updated, covariance
```