TTK4250 Sensor Fusion
Assignment 3

Hand in: *Wednesday 16. September 16.00* on Blackboard.

Tasks are to be solved on paper if you are not told otherwise, and you are supposed to show how you got to a particular answer. It is, however, encouraged to use Python, MATLAB, Maple, etc. to verify your answers. Rottmann's mathematical formula collection is allowed at both the exercises and the exam.

Task 1: *Bayesian estimation of an existence variable*

We are back at tracking the number of boats in a region. However, you now know that there is at most one boat in the region, and that it is there with probability $r_k \in (0, 1)$ at time step k. The boat will stay in the region with probability P_{S} and leave with probability $1 - P_{\mathrm{S}}$ if it is there. Otherwise, it may enter with probability P_E and not enter with probability $1 - P_E$ if it is not in the region. You have an unreliable measurement coming from a radar that detects a present boat with probability P_{D}, and may not detect a present boat with probability $1 - P_{\mathrm{D}}$. If it did not detect a present boat, it may declare that there is a boat present anyway, termed a false alarm, with probability P_{FA}. The task is to apply a Bayesian filter to this problem.

Hint: It might ease proper book keeping if you start by denoting the events with variables and use their (possibly conditional) pmfs before you insert the given probabilities.

(a) Apply the Bayes prediction step to get the predicted probability $r_{k+1|k}$ in terms of r_k, P_{S} and P_E.

Hint: Introducing the events at time step k as "in region" R_k,"staying" S_k and "entering" E_k, where $S_{k+1} = R_{k+1}|R_k$ and $E_{k+1} = R_{k+1}|\neg R_k$ with $\neg$ denoting negation, can be helpful.

(b) Apply the Bayes update step to get posterior probability for the boat being in the region, r_{k+1}, in terms of $r_{k+1|k}$, P_{D} and P_{FA}. That is, condition the probability on the measurement. There are two cases that needs to be considered; receiving a detection and not receiving a detection.

Hint: Introduce the events $M_{k+1} = D_{k+1} \cup F_{k+1}$ to denote the sensor declaring a present boat, with D_{k+1} denoting the event of detecting a present boat and F_{k+1} denoting the event of a false alarm. We then for instance have $\Pr(D_{k+1}|\neg R_{k+1}) = 0$ and $\Pr(F_{k+1}|D_{k+1}) = 0$ from the problem setup.

Task 2: *KF initialization of CV model without a prior*

The KF typically uses a prior for initializing the filter. However, in target tracking we often have no specific prior and would like to infer the initialization of the filter from the data. For the CV model (see chapter 4) with positional measurements, the position is observable with a single measurement, while the velocity needs two measurements to be observable (observable is here used in a statistical sense to mean that there is information about the state from the measurements).

With $x_k = \begin{bmatrix} p_k^T & u_k^T \end{bmatrix}^T$, where p_k is the position and u_k is the velocity at time step k, you should recognize the CV model as

$$x_{k+1} = \begin{bmatrix} p_{k+1} \\ u_{k+1} \end{bmatrix} = Fx_k + v_k,$$

with $v_k \sim \mathcal{N}(0, Q)$ and F and Q as defined in (4.64) in the book. The measurement model is given by $z_k = \begin{bmatrix} I_2 & 0_2 \end{bmatrix} x_k + w_k = p_k + w_k$ and $w_k \sim \mathcal{N}(0, R) = \mathcal{N}(0, \sigma_r^2 I_2)$.

Since the KF is linear, we would like to use a linear initialization scheme that uses two measurements and the model parameters. That is

$$\hat{x}_1 = \begin{bmatrix} \hat{p}_1 \\ \hat{u}_1 \end{bmatrix} = \begin{bmatrix} K_{p_1} & K_{p_0} \\ K_{u_1} & K_{u_0} \end{bmatrix} \begin{bmatrix} z_1 \\ z_0 \end{bmatrix}. \tag{1}$$

(a) Write z_1 and z_0 as a function of the noises, true position and speed, p_1 and u_1, using the CV model with positional measurements. Use v_k to denote the process disturbance and w_k to denote the measurement noise at time step k, and T for the sampling time between $k-1$ and k.

Hint: A discrete time transition matrix is always invertible, and it is easy to find for the CV model: remove the process noise and write out the transition as a system of linear equations, solve the system for the inverse and rewrite it as a matrix equation again.

(b) Show that to get an unbiased initial estimate, the initialization gain matrix must satisfy $K_{p_1} = I_2$, $K_{p_0} = 0_2$, $K_{u_1} = \frac{1}{T} I_2$ and $K_{u_0} = -\frac{1}{T} I_2$, where T is the sampling time. That is, find the $K_\times$ so that $E[\hat{x}_1] = x_1 = \begin{bmatrix} p_1 & u_1 \end{bmatrix}^T$

Note: To find estimator biases, one fixes the values to be estimated and do not treat them as random variables.

(c) What is the covariance of this estimate?

(d) You have used this initialization scheme for your estimator and found a mean and covariance. What distribution does the true state have after this initialization? What are its parameters.

Hint: From equations you have already used, you can write x_1 in terms of $\hat{x}$ and disturbances and noises. You should be able to see the result as a linear transformation of some random variables. Note that $\hat{x}$ is given since the measurements are given and thus can be treated as a constant. x_1 is now treated as a random variable as opposed to when finding the mean and variance of the estimator.

(e) In theory, would you say that this initialization scheme is optimal or suboptimal, given that the model and two measurements is all we have? What would you say about its optimality in practice?

Task 3: *Make CV dynamic model and position measurement model*

In Python: Finish the classes `WhitenoiseAcceleration` that implements the CV model and `CartesianMeasurement` that implements positional measurements, so that they can be used in an (E)KF. Even though these particular models are linear, we are going to implement them as if they were more general. The skeletons can be found in `dynamicmodels.py` and `measurementmodels.py` on Blackboard. You have the standard model for a KF

$$x_k = F_{k-1} x_{k-1} + v_{k-1}, \qquad v_{k-1} \sim \mathcal{N}(0; Q) \tag{4}$$
$$z_k = H_k x_k + w_k, \qquad w_k \sim \mathcal{N}(0, R) \tag{5}$$

Note: The skeleton is for a quite flexible classes, and therefor have more parameters than you need to think about. To be more explicit: you do not need to consider `dim, pos\_idx, vel\_idx, sensor\_state` (and `z` in `R`) if you do not want to.

(a) Implement the transition function $F_{k-1} x_{k-1}$ in

```
WhitenoiseAcceleration.f(self, x: np.ndarray, Ts: float) -> np.ndarray: ...
```

(b) Implement the Jacobian of f with respect to x (F_{k-1}) in

```
WhitenoiseAcceleration.F(self, x: np.ndarray, Ts: float) -> np.ndarray: ...
```

Implement the process noise covariance matrix as a function of x (Q) in

```
WhitenoiseAcceleration.Q(self, x: np.ndarray, Ts: float, ) -> np.ndarray: ...
```

(c) Implement the measurement prediction $H_k x_k$ in

```
CartesianMeasurement.h(self, x: np.ndarray, **kwargs) -> np.ndarray: ...
```

(d) Implement the Jacobian of h with respect to x H_k in

```
CartesianMeasurement.H(self, x: np.ndarray, **kwargs) -> np.ndarray: ...
```

(e) Implement the measurement noise covariance matrix R

```
CartesianPositionR(self, x: np.ndarray, **kwargs) -> np.ndarray: ...
```

Task 4: *Implement EKF i Python*

In Python: Finish the following functions in `ekf.py` skeleton found on blackboard. You do not need to worry about the sensors_state and other keyword only arguments (`**kwargs`). The EKF class will be initialized with a measurement model and a dynamic model of the form you made in the last task. You can therefore assume that the methods (or with the same interface) are available in `self.dynamic_models` and `self.measurement_model`.

You are free to change the prewritten code (for instance to optimize or make things clearer for your self), but the function definitions must be the same for evaluation purposes.

```
def predict(self, ekfstate: GaussParams, Ts: float) -> GaussParams:

def innovation_mean(self, z: np.ndarray, ekfstate: GaussParams, **kwargs) -> np.ndarray:

def innovation_cov(self, z: np.ndarray, ekfstate: GaussParams, **kwargs) -> np.ndarray:

def innovation(self, z: np.ndarray, ekfstate: GaussParams, **kwargs) -> GaussParams:

def update(self, z: np.ndarray, ekfstate: GaussParams, **kwargs) -> GaussParams:

def step(self, z: np.ndarray, ekfstate: GaussParams, Ts: float, **kwargs) -> GaussParams:

def NIS(self, z: np.ndarray, ekfstate: GaussParams, **kwargs) -> float:

@classmethod
def NEES(cls, ekfstate: GaussParams, x_true: np.ndarray) -> float:

def loglikelihood(self, z: np.ndarray, ekfstate: GaussParams) -> float:

def estimate_sequence(
        self,
        # A sequence of measurements
        Z: Sequence[np.ndarray],
        # the initial KF state to use for either prediction or update (see start_with_prediction)
        init_ekfstate: GaussParams,
        # Time difference between Z's. If start_with_prediction: also diff before the first Z
        Ts: Union[float, Sequence[float]],
        *,
        # An optional sequence of the sensor states for when Z was recorded
        sensor_state: Optional[Iterable[Optional[Dict[str, Any]]]] = None,
        # sets if Ts should be used for predicting before the first measurement in Z
        start_with_prediction: bool = False,
        ) -> Tuple[GaussParamList, GaussParamList]:
```

Task 5: *Tuning of KF*

Tune your CV KF implementation to fit the given data. The data is created by simulating a CT model with $\sigma_a = 0.25^2 = 0.0625$ and $\sigma_\omega = \left(\frac{\pi}{15}\right) = 0.0439$. You can also generate new data if you want to test more.

There is a Python script `runekf.py` on Blackboard to get you started. Feel free to change this as much as you like as long as it provides you the answers.

Hint: Section 4.6 – 4.8 will probably be helpful.

(a) Simply tune σ_a and σ_z until you are satisfied.

Note: Do not spend several hours doing this. It is for you to get a feel for the KF — which you will need — before we use it as building blocks of more complicated systems.

(b) Make a grid for σ_a and σ_z and plot NIS using `pyplot.Axes3D.plot_surface`. Use the distribution of NIS to find confidence intervals and plot the contour lines of your evaluated NIS for these intervals using `pyplot.Axes3D.contour`. Would you do some corrections to your tuning from a)? Do you expect NIS to be good for tuning a CV model to a CT trajectory? Why?

Hint: Write `help(<function name or object>)` in a Python console, or some key sequence in your IDE, to get information on it. `scipy.stats.chi2.interval` might come in handy.

(c) Do the same for NEES of predicted and updated state as you did with NIS. Would you tune differently now? Do you expect NEES to be good for tuning a CV model to a CT trajectory? What can you say from the plot where all the contour plots of NEES and NIS are superimposed?

(d) Run your tuned filter on some new data. Are the results as expected? Should the performance be the same? Why/why not?