TTK4250 Sensor Fusion

Assignment 5

Hand in: *Wednesday 7. October 16.00* on Blackboard (as single PDF-file) or to teaching assistants in the exercise class.

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: *State dependent detection probability*

Let us generalize the single target assumption S4 in the book and say we have a variable detection probability in the state space, ie. $\Pr(\delta|x) = P_{\mathrm{D}}(x)$. Let the state distribution be given by $f(x)$. Will knowing that the target has been detected (not the location of the measurement) change the state distribution, ie. $f(x|\delta) \neq f(x)$? In what cases does it seem reasonable to use such a model?

Hint: Use Bayes rule and the total probability theorem for the denominator; $\Pr(\delta) = \int P_{\mathrm{D}}(x) f(x) \, \mathrm{d}x$.

Task 2: *The PDAF event probabilities*

A sensor has N cells. Each cell has the same measurement volume and the same probability, P_{FA}, of giving a false alarm independent of each other (and any correct detections for the sake of simplicity). In addition, we have a track on an object known to be in the sensor volume with the standard linear-Gaussian assumptions on its state estimate.

(a) What is the distribution of the number of false alarms, φ, for a portion of the sensor volume containing M_k sensor cells?

Hint: Simple reasoning and some knowledge from chapter 2 should give the answer without any calculations.

(b) We assume that this can be well approximated by a Poisson distribution with parameter $\lambda V_k = M_k P_{FA}$,

where V_k is the volume of M_k sensor cells. Find the more specific formula for the posterior event probabilities in theorem 7.3.1 under these assumptions. That is, prove corollary 7.3.3 starting with theorem 7.3.1.

Hint: You can take advantage of the proportionality sign to get rid of factors independent of a_k present in both cases or move them from one case to the other.

(c) Use the real distribution under our assumptions (part (a)) as the distribution $\mu(\varphi_k)$ in theorem 7.3.1 and show that the posterior event probabilities can be given as

$$\Pr(a_k|Z_{1:k}) \propto \begin{cases} (1 - P_{\mathrm{D}}) \frac{P_{FA} M_k}{V_k} \frac{1 - \frac{m_k - 1}{M_k}}{(1 - P_{FA})}, & a_k = 0, \\ P_{\mathrm{D}} \mathcal{N}(z_k^{a_k}; \hat{z}_{k|k-1}, S_k), & a_k > 0. \end{cases}$$

(d) Compare the formula for the event probabilities using the true distribution under our assumptions with corollary 7.3.3. Does the Poisson approximation seem good when M_k is large and P_{FA} is small?

Hint: Look at $\frac{m_k - 1}{M_k} \approx 0$, and $P_{FA} \approx 0$.

Task 3: *IPDA vs PDAF*

(a) Briefly discuss why the posterior becomes a mixture in single target tracking. In particular, what is the interpretation of a component and its weight. What are the main complicating factors of this mixture?

(b) What problem does the IPDA try to solve that the PDA does not? Are there any problems that the PDA solves which the IPDA does not solve?

(c) What are some of the complicating factors in using an IPDA over a PDA?

Task 4: *Implement a parametric PDAF*

You are to implement a parametric PDA class in Python starting with the skeleton `pda.py` found on Blackboard. This class builds on the EKF or IMM class you made in assignment 3 or 4, and uses an instance of it as an initialization parameter. In addition to the EKF input it also takes clutter_rate, PD and gate_size — corresponding to λ, P_{D} and g respectively — as parameters.

Note: The structure is on purpose made general, so that it should be easy to use both EKF and IMM as a state estimator in it. It should also be fairly easy to extend it to their multi-target counterparts (IMM-)J(I)PDA.

(a) Implement the prediction step in the function

```python
def predict(self, filter_state: ET, Ts: float) -> ET:
```

(b) Implement gating in the function

```python
def gate(
    self,
    # measurements of shape=(M, m)=(#measurements, dim)
    Z: np.ndarray,
    filter_state: ET,
    *,
    sensor_state: Optional[Dict[str, Any]] = None,
) -> bool:  # gated (m x 1): gated(j) = true if measurement j is within gate
```

(c) Implement the log likelihood ratios (logarithm of corollary 7.3.3 in the book) in the function

```python
def loglikelihood_ratios(
    self,  # measurements of shape=(M, m)=(#measurements, dim)
    Z: np.ndarray,
    filter_state: ET,
    *,
    sensor_state: Optional[Dict[str, Any]] = None,
) -> np.ndarray:  # shape=(M + 1,), first element for no detection
```

(d) Implement the calculation of the association probabilities in the function

```python
def association_probabilities(
    self,
    # measurements of shape=(M, m)=(#measurements, dim)
    Z: np.ndarray,
    filter_state: ET,
    *,
    sensor_state: Optional[Dict[str, Any]] = None,
) -> np.ndarray:  # beta, shape=(M + 1,): the association probabilities (normalized likelihood r
```

(e) Implement the update of the state for all possible associations in the function

```python
def conditional_update(
    self,
    # measurements of shape=(M, m)=(#measurements, dim)
    Z: np.ndarray,
    filter_state: ET,
    *,
    sensor_state: Optional[Dict[str, Any]] = None,
) -> List[
    ET
]:  # Updated filter_state for all association events, first element is no detection
```

(f) Implement the mixture reduction step in the function. The functionality for this is actually to be implemented in EKF and IMM to make the PDA class general.

```python
def reduce_mixture(
    self, mixture_filter_state: MixtureParameters[ET]
) -> ET:  # the two first moments of the mixture
```

(g) Implement the combination of the earlier steps to get an overall update function,

```python
def update(
    self,
    # measurements of shape=(M, m)=(#measurements, dim)
    Z: np.ndarray,
    filter_state: ET,
    *,
    sensor_state: Optional[Dict[str, Any]] = None,
) -> ET:  # The filter_state updated by approximating the data association
```

(h) Combine the predict and update steps

```python
def step(
    self,
    # measurements of shape=(M, m)=(#measurements, dim)
    Z: np.ndarray,
    filter_state: ET,
    Ts: float,
    *,
    sensor_state: Optional[Dict[str, Any]] = None,
) -> ET:
```

(i) Implement the estimate function, which should rely on the estimate function of `self.state_filter`

```python
def estimate(self, filter_state: ET) -> GaussParams:
```

Task 5: *Tune your parametric PDAF*

Tune your parametric PDAF using a CV model with position measurements on the data given in the .mat file on Blackboard. The trajectory is the same as the one in assignment 2, so you should have some idea of where to start tuning σ_a and σ_z. You have to tune the process noise, the measurement noise, the detection probability, the false alarm rate and the gate size. We recommend looking at position and velocity RMSE, as well as total, position and velocity NEES.

You can initialize the position using the first measurement and the velocity as zero. The covariance for position can then be set to, say, $2R = 2\sigma_z^2 I_2$, while the velocity covariance can be set using that the maximum velocity is limited to 20.

Again, do not spend several hours testing everything and fine tuning the filter. The point is for you to get a feel for the algorithm and know what the different parameters will the the performance and how

the metrics will change.

Note: We have not enforced the computation of NIS here as there is a complicating factor in using NIS with are several measurements. However, one can, for instance, use the averaged innovation in the standard NIS calculation, $\bar{v}_k = \sum_{a_k=1}^{m_k} \beta_k^{a_k} (z_k^{a_k} - \hat{z}_{k|k-1}) = \sum_{a_k=1}^{m_k} \beta_k^{a_k} v_k^{a_k}$, if one wishes.