TTK4250 Assignment 5 Oystein Sethov

1 with the prior information regarding that the target has been detected affect the state distribution?

$$p(x|\delta) = \frac{p(\delta|x)\,p(x)}{p(\delta)} = \frac{p(\delta|x)\,p(x)}{\int p_D(x)\,p(x)\,dx}$$

$$= \frac{P_D(x)\,p(x)}{\int P_D(x)\,p(x)\,dx}$$

The question is whether this changes the distribution.

Assuming $P_D(x)$ constant, we get that

$$p(x|\delta) = \frac{P_D\,p(x)}{P_D \int p(x)\,dx} = p(x)$$

however, by assuming that $P_D(x)$ varying; we get that

$$p(x|\delta) = \frac{P_D(x)\,p(x)}{\int P_D(x)\,p(x)\,dx} \neq p(x)$$

assuming that $\frac{\partial}{\partial x} P_D(x) \neq P_D(x)\,p(x)$

in other words, prior information of existence changes the distribution

2 a) Sensor with N cells and a likelihood of false alarm given as P_{FA}

Assuming that there are m_k sensor cells being ~~shortest~~ used.

The number of false alarms is given as a binomial distribution:

$$P(\varphi_k = \varphi) = \binom{m_k}{\varphi} P_{FA}^{\varphi} (1-P_{FA})^{m_k-\varphi}$$

where $0 \leq \varphi \leq m_k \leq N$

b) Assuming a poisson-distribution with $\lambda = \frac{1}{V_k} m_k P_{FA}$

Supposed to use that

$$P_r\{a_k | Z_{1:k}\} \propto \begin{cases} (1-P_D)\, m_k \frac{\mu(m_k)}{\mu(m_k-1)}, & a_k = 0 \\ \frac{P_D}{c(z_k^{a_k})}, & a_k > 0 \end{cases}$$

I will be relying a bit on the lectures, as I would otherwise have been quite high to come up with this method.

we know that

$$I^{a_k} = \int dz\, (z_k^{a_k} \mid x_k)\, p_{k|k-1}(x_k)\, dx_k$$

Under gaussian-linear assumptions, we have that

$$I^{a_k} \hat{=} N(z_k^{a_k};\ \hat{z}_{k|k-1},\ S_k)$$

This gives us that (by also abusing the proportionality)

$$\Pr\{a_k \mid z_{1:k}\} \propto \begin{cases} (1-P_D)\,\mu(m_k) & , a_k = 0 \\ \dfrac{P_D\,\mu(m_k-1)}{m_k\, C(z_k)}\, N(z_k^{a_k};\ \hat{z}_{k|k-1},\ S_k) & , a_k > 0 \end{cases}$$

By using that this is a ~~det~~ poisson-distribution, we have that

$$\mu(\varphi) = \frac{\lambda^{\varphi}}{\varphi!} \exp(-\lambda)$$

which gives that

$$\Pr\{a_k \mid z_{1:k}\} \propto \begin{cases} (1-P_D)\,\dfrac{\lambda^{m_k}}{m_k!} \exp(-\lambda) & , a_k = 0 \\ \dfrac{P_D\,\lambda^{m_k-1}}{C(z_k)\,(m_k-1)!\,m_k} \exp(-\lambda)\, N(z_k^{a_k};\ \hat{z}_{k|k-1},\ S_k) & , a_k > 0 \end{cases}$$

$$\propto \begin{cases} (1-P_D)\,\lambda^{m_k} & , a_k = 0 \\ P_D\, N(z_k^{a_k};\ \hat{z}_{k|k-1},\ S_k) & , a_k > 0 \end{cases}$$

c) since we are to use real distribution from a), I assume that we are to use the binomial-distribution

This gives that

$$\mu(m_k) = \binom{m_k}{m_k} P_{TA}^{m_k} (1-P_{TA})^{m_k - m_k}$$

$$\Pr\{a_k \mid z_{1:k}\} \propto \begin{cases} (1-P_D) \binom{m_k}{m_k} P_{TA}^{m_k} (1-P_{TA})^{m_k - m_k} & , a_k = 0 \\ \dfrac{P_D \binom{m_k}{m_k - 1} P_{TA}^{m_k - 1} (1-P_{TA})^{m_k - m_k + 1}}{m_k\, C(z_k)}\, I^{a_k} & , a_k > 0 \end{cases}$$

By abusing the proportionality-sign, this gives us that

$$\propto cc \begin{cases} (1-P_D)\,\frac{m_k P_{FA}}{1-P_{FA}} \binom{M_k}{m_k} \cdot \frac{1}{\binom{M_k}{m_k-1}} & , a_k = 0 \\ P_D\, \mathcal{N}(z_k^{a_k};\, \hat{z}_{k|k-1},\, S_k) & , a_k > 0 \end{cases}$$

$$\binom{M_k}{m_k} \cdot \frac{1}{\binom{M_k}{m_k-1}} = \frac{M_k!}{m_k!(M_k-m_k)!} \cdot \frac{(m_k-1)!\,(M_k-m_k+1)!}{M_k!}$$

$$= \frac{(M_k-m_k+1)!}{m_k}$$

$$\Rightarrow \propto cc \begin{cases} (1-P_D)\, P_{FA}\, M_k\, \frac{(M_k-m_k+1)!}{m_k\,(1-P_{FA})} & , a_k = 0 \\ P_D\, \mathcal{N}(z_k^{a_k};\, \hat{z}_{k|k-1},\, S_k) & , a_k > 0 \end{cases}$$

I cannot see how one should simplify the top expression further

d) Assuming that M_k is large and P_{FA} small. This gives that

$$\lim_{P_{FA}\to 0} \lim_{M_k\to\infty} \Pr\{a_k \mid z_{1:k}\} \propto$$

$$cc \lim_{P_{FA}\to 0} \lim_{M_k\to\infty} (1-P_D)\, M \underbrace{\frac{1-\frac{m_k-1}{M_k}}{1-P_{FA}}}_{\to 1} = (1-P_D)\, M$$

As expected due to the properties of the poisson-distribution and the binomial-distribution, the approximation seems good.

3a) The linear-gaussian assumption assumes that all of the distributions are gaussians, such that the ~~det~~ posterior is ~~then~~ a gaussian.

The PDAF tries to combine all of the measurement-models into a single distribution by mixturing them. This is done to save computational power by only keeping track of one mixture, but also allows a weighting of the measurement models. Under the single-target tracking, it is impossible to know exactly which measurement has it origin in the target, due to clutter and/or misdetections.

By weighting the measurements, one could quantify how likely it is that a measurement originates from the target.

b) IPDA is a single target tracking algorithm that tries to track a target when there is no information whether the target exists or not. In other words, the IPDA is PDA (7) with the added estimation if the target is present or not.

Since PDA $\subset$ IPDA, there are no problems that PDA solves that IPDA cannot.

c) The IPDA will have a larger state-estimation compared to the PDA. It cannot just implicitely assume that a target exists, it has to find and maintain such a likelihood as well.

```python
    def get_mean(self) -> ndarray:
        """
        Return the mean of the gaussian mixture

        Hint: use what you did in mixturereductin.py assignment 4

        Returns:
            mean (ndarray): the mean
        """
        # I assume that the system should mixture self.gaussians
        means = np.zeros_like(self.weights)

        # Extract all of the data from the gaussians
        i = 0
        for gaussian in self.gaussians:
            means[i] = gaussian.mean
            i += 1

        # Multiply by the weights of the corresponding gaussians
        mean = self.weights.T @ means

        #mean = solution.gaussmix.GaussianMuxture.get_mean(self)
        return mean
```

```python
def get_cov(self) -> ndarray:
    """
    Return the covariance of the gaussian mixture

    Hint: use what you did in mixturereductin.py assignment 4

    Returns:
        cov (ndarray): the covariance
    """
    mean = self.get_mean()
    #diffs = means - mean[np.newaxis]

    cov_shape = np.shape(self.gaussians[0].cov)
    covs = np.zeros_like(cov_shape)
    #for i in range(len(self.gaussians)):
        # covs[i*cov_shape[0]:i]

    #cov_internal = np.average(covs, axis=0, weights=self.weights)
    #cov = 0

    # I was unable to actually solve this in task 4, and still haven't
    # understood how one should solve this

    cov = solution.gaussmix.GaussianMuxture.get_cov(self)
    return cov
```

```python
def reduce(self) -> MultiVarGaussian:
    """
    Reduce the gaussian mixture to a multivariate gaussian
    Hint: you can use self.get_mean and self.get_cov

    Returns:
        reduction (MultiVarGaussian): the reduction
    """
    mean = self.get_mean()
    cov = self.get_cov()

    reduction = np.random.multivariate_normal(mean, cov)

    #reduction = solution.gaussmix.GaussianMuxture.reduce(self)
    return reduction
```

```python
def predict_state(
                self,
                state_upd_prev_gauss: MultiVarGaussian,
                Ts: float
                ) -> MultiVarGaussian:
    """
    Prediction step
    Hint: use self.ekf

    Args:
        state_upd_prev_gauss (MultiVarGaussian): previous update gaussian
        Ts (float): timestep

    Returns:
        state_pred_gauss (MultiVarGaussian): predicted state gaussian
    """
    state_pred_gauss = self.ekf.predict_state(
            state_upd_prev_gauss=state_upd_prev_gauss,
            Ts=Ts)

    # state_pred_gauss = solution.pdaf.PDAF.predict_state(
    #     self, state_upd_prev_gauss, Ts)
    return state_pred_gauss
```

```python
def predict_measurement(
                        self,
                        state_pred_gauss: MultiVarGaussian
                        ) -> MultiVarGaussian:
    """
    Measurement prediction step
    Hint: use self.ekf

    Args:
        state_pred_gauss (MultiVarGaussian): predicted state gaussian

    Returns:
        z_pred_gauss (MultiVarGaussian): predicted measurement gaussian
    """

    z_pred_gauss = self.ekf.predict_measurement(state_pred_gauss=state_pred_gauss)

    # z_pred_gauss = solution.pdaf.PDAF.predict_measurement(
    #     self, state_pred_gauss)
    return z_pred_gauss
```

```python
def gate(
        self,
        z_pred_gauss: MultiVarGaussian,
        measurements: Sequence[ndarray]
        ) -> ndarray:
    """
    Gate the incoming measurements. That is remove the measurements
    that have a mahalanobis distance higher than a certain threshold.

    Hint: use z_pred_gauss.mahalanobis_distance_sq and self.gate_size_sq

    Args:
        z_pred_gauss (MultiVarGaussian): predicted measurement gaussian
        measurements (Sequence[ndarray]): sequence of measurements

    Returns:
        gated_measurements (ndarray[:,2]): array of accepted measurements
    """
    gated_measurements = np.empty_like([measurements[0]])
    is_initialized = False

    for m in measurements:
        if z_pred_gauss.mahalanobis_distance_sq(m) > self.gate_size_sq:
            continue
        if is_initialized:
            gated_measurements = np.append(gated_measurements, np.array(m))
            continue
        gated_measurements[:] = np.array(m)
        is_initialized = True

    # gated_measurements = solution.pdaf.PDAF.gate(
    #     self, z_pred_gauss, measurements)
    return gated_measurements
```

```python
def get_association_prob(
                        self,
                        z_pred_gauss: MultiVarGaussian,
                        gated_measurements: ndarray
                        ) -> ndarray:
    """
    Finds the association probabilities.

    associations_probs[0]: prob that no association is correct
    associations_probs[1]: prob that gated_measurements[0] is correct
    associations_probs[2]: prob that gated_measurements[1] is correct
    ...

    the sum of associations_probs should be 1

    Args:
        z_pred_gauss (MultiVarGaussian): predicted measurement gaussian
        gated_measurements (ndarray[:,2]): array of accepted measurements

    Returns:
        associations_probs (ndarray[:]): the association probabilities
    """
    P_D = self.detection_prob
    V = 1 / self.clutter_density

    m = len(gated_measurements)
    associations_probs = np.zeros((m,1))
    associations_probs = np.append(associations_probs, np.zeros((1,1)))

    # Calculating for i == 0
    associations_probs[0] = m/V * (1 - P_D)

    # Calculating for i > 0
    for i in range(1, m):
        I_i = z_pred_gauss.pdf(gated_measurements[i])
        associations_probs[i-1] = P_D*I_i

    associations_probs /= associations_probs.sum()

    # associations_probs = solution.pdaf.PDAF.get_association_prob(
    #     self, z_pred_gauss, gated_measurements)
    return associations_probs
```

```
    def get_cond_update_gaussians(
                                self,
                                state_pred_gauss: MultiVarGaussian,
                                z_pred_gauss: MultiVarGaussian,
                                gated_measurements: ndarray
                                ) -> Sequence[MultiVarGaussian]:
        """
```

```
# Allocating memory
n = len(gated_measurements)
update_gaussians = np.arange(n+1, dtype=MultiVarGaussian)

# Get the associated probabilities
associated_probs = self.get_association_prob(z_pred_gauss, gated_measurements)

# Iterate over the associated probabilities and calculate the posteriori state given
# that the associated measurement is correct
for i in range(len(associated_probs)):
    gaussian_mixture_class = GaussianMuxture(
            weights=np.array([associated_probs[i]]),
            gaussians=np.array([state_pred_gauss]))
    update_gaussians[i] = gaussian_mixture_class.reduce()

# Something is incorrect with this code, since it will not reduce properly
# for all

update_gaussians = solution.pdaf.PDAF.get_cond_update_gaussians(
    self, state_pred_gauss, z_pred_gauss, gated_measurements)
return update_gaussians
```

```python
def update(
        self,
        state_pred_gauss: MultiVarGaussian,
        z_pred_gauss: MultiVarGaussian,
        measurements: Sequence[ndarray]
        ) ->MultiVarGaussian:
    """
    Perform the update step of the PDA filter

    Args:
        state_pred_gauss (MultiVarGaussian): predicted state gaussian
        z_pred_gauss (MultiVarGaussian): predicted measurement gaussian
        measurements (Sequence[ndarray]): sequence of measurements

    Returns:
        state_upd_gauss (MultiVarGaussian): updated state gaussian
    """
    # Gate the measurements
    gated_measurements = self.gate(
            z_pred_gauss=z_pred_gauss,
            measurements=measurements)

    # Get the associated probabilities
    associated_probs = self.get_association_prob(
            z_pred_gauss=z_pred_gauss,
            gated_measurements=gated_measurements)

    # Get the conditional associated states
    cond_states = self.get_cond_update_gaussians(
            state_pred_gauss=state_pred_gauss,
            z_pred_gauss=z_pred_gauss,
            gated_measurements=gated_measurements)

    gaussian_mixture_class = GaussianMuxture(
            weights=associated_probs,
            gaussians=cond_states)
    state_upd_gauss = gaussian_mixture_class.reduce()

    # state_upd_gauss is here just an array of float64
    # and not a gaussian... Just fuck python! C++ all the way (except for plotting)

    state_upd_gauss = solution.pdaf.PDAF.update(
        self, state_pred_gauss, z_pred_gauss, measurements)
    return state_upd_gauss
```

```python
def step_with_info(
                self,
                state_upd_prev_gauss: MultiVarGaussian,
                measurements: Sequence[ndarray],
                Ts: float
                ) -> Tuple[MultiVarGaussian,
                           MultiVarGaussian,
                           MultiVarGaussian]:
    """
    Perform a full step and return usefull info

    Hint: you should not need to write any new code here,
    just use the methods you have implemented

    Args:
        state_upd_prev_gauss (MultiVarGaussian): previous updated gaussian
        measurements (Sequence[ndarray]): sequence of measurements
        Ts (float): timestep

    Returns:
        state_pred_gauss (MultiVarGaussian): predicted state gaussian
        z_pred_gauss (MultiVarGaussian): predicted measurement gaussian
        state_upd_gauss (MultiVarGaussian): updated state gaussian
    """

    state_pred_gauss = self.predict_state(state_upd_prev_gauss=state_upd_prev_gauss, Ts=Ts)
    z_pred_gauss = self.predict_measurement(state_pred_gauss=state_pred_gauss)
    state_upd_gauss = self.update(state_pred_gauss=state_pred_gauss, z_pred_gauss=z_pred_gauss, measurements=me

    # state_pred_gauss, z_pred_gauss, state_upd_gauss = solution.pdaf.PDAF.step_with_info(
    #     self, state_upd_prev_gauss, measurements, Ts)
    return state_pred_gauss, z_pred_gauss, state_upd_gauss
```

```
"""
Been raging a bit too hard on python and numpy, so
my bloodpressure is a bit too high right now...

Also depressed, such that I have no motivation on tuning the
filter well...

Perfect time to tune in other words!

I will try to rather get a feeling about the response, instead
of having a filter that is "perfect"....


Once my blood starts cooling down, pherhaps I try to get a more
optimized filter, but cannot guarantee that!


sigma_a and sigma_z is related to the EKF

sigma_z = R and is the noise belonging to the measurements. By
increasing this, we expect that the measurements will have a
larger noise, and thus trust the model more. For the PDA, this
means that values that are further away from the measurement is
more related to noise in measurements instead of the actual
process. Increasing this should in fact reduce number of samples
that will actively be considered

sigma_z = Q is the noise related to the model. By increasing
this, we say that the process can change rapidly. This allows
us to increase the likelihood for measurements further away
from the priori estimate. Thus, considering more estimates

Clutter density is number of measruerments per m^2. Increasing
this allows the system to generate more clutter. I cannot identify
if this exact value is reasonable or not, however the clutter
density will vary depending on the accuracy of the system as well
as of the desired scan-velocity.
For example would a radar have lower clutter intensity with a lower
rotation speed, with the downside that it updates relatively
infrequently. Increasing the rotation speed allwos it to scan the
area quicker, however will allow less time for the sensor cells
to acquire and identify a possible signal.

Detection probability should be in the range [0.5, 1)
Lower bounded by 0.5 since that value means that we are just
guessing yes/no if a detection has been made.
```

```
Note written couple of days later:

I should have tuned this a lot better. The difficult thing is to get
the system to follow the correct state without diverging too
rapidly. My experimentation showed that the system would either accept
too many measurements and therefore diverge, or that it would not value
the real system's measurements and thus diverge. I did experience that the
algortihm was able to follow the track somewhat, however the filter became
overconfident and started diverging at iteration 136. However, when running
the same value on my desktop (right when I am writing this shit), the system
diverged at iteration number 7 to 8. Idk why my laptop and desktop gave
different results....


I honestly started by trying to find the values that made the system stay
within a reasonable NEES, however that didn't quite work out.

After that, I tried to just tune after analysing the track... It went as well
as one could expect.

In other words, fuck my life!


"""
```