TTK4250 Sensor Fusion

Solution to Assignment 3

Task 1: *Gaussian mixture reduction*

(a) Implement the MATLAB function `[xmix, Pmix] = reduceGaussMix(w, x, P)`, that implements equations 6.18 - 6.19 in the book.

Solution: The code can be written as

```
% mean
for i = 1:M
    xmix = xmix + x(:, i) * w(i);
end

% covariance
for i = 1:M
    mixInnovation = x(:, i) - xmix;
    Pmix = Pmix + (P(:, :, i) + mixInnovation * mixInnovation') *
        w(i);
end
```

(b) You are waiting for a friend who is driving to you on a particular route. While you are waiting, you deduce a distribution over how far you believe he has come from a couple of messages from him. The messages were a bit unclear, so you ended up with a univariate Gaussian mixture with three components as the distribution. However, you think it is a bit much with three components and believe that you should get a good enough representation with only two Gaussians. So, you decide to merge two of them by making these two into a new Gaussian that matches the mean and covariance of the mixture of the two. To keep it simple you neglect any time/speed aspect.

In the given Gaussian mixtures, which would you merge by moment matching if you were to merge two components, why would you merge these, and what would the resulting components be?

A script is given if you want to visualize the problem and use your solution to (a) for calculations.

Hint: Section 6.3 in the book can provide some insight. For merging you can use the fact that $w_1p_1(x) + w_2p_2(x) + w_3p_3(x) = w_1p_1(x) + (w_2 + w_3)\left(\frac{w_2}{w_2+w_3}p_2(x) + \frac{w_3}{w_2+w_3}p_3(x)\right)$. There is not neccessarily a one true answer here, and it depends on what you emphasise.

i. $w = \{\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\}$, $\mu = \{0, 2, 4.5\}$ and $P = \{1^2, 1^2, 1^2\}$.

Solution: From intuition it seems that merging the two closest, namely 1 and 2, will be appropriate since the other parameters are the same. A quick plot confirms that this actually keeps the modes, although having a to high first mode and a to low valley between the modes. Merging 2 and 3 shifts the modes quite a bit, and overall seem like a worse option than merging 1 and 2. Merging 1 and 3 gives a single mode and does not really resemble the original distribution at all. Thus merging 1 and 2 is the best option, giving $w_{1,2} = \frac{2}{3}$, $\mu_{1,2} = 1$ and $\sigma^2_{1,2} = \sqrt{2}^2$.

ii. $w = \{\frac{1}{6}, \frac{4}{6}, \frac{1}{6}\}$, $\mu = \{0, 2, 4.5\}$ and $P = \{1^2, 1^2, 1^2\}$.

Solution: Intuition gives seems that merging the two closest, namely 1 and 2, or the one with lowest weights, 1 and 3, will be best. Again, a quick plot confirms this. Mergining 1 and 2 only slightly changes the mode while keeping the small bump caused by 3. Merging 1 and 3 keeps the mode at the right location, although overestimating its height, at the expense of loosing the "shoulders" casued by 1 and 3 alone along with a slightly heavier tail. Merging 2 and 3 seem like the worst option, totally missing the shape. If keeping the exact shape is most important, merging 1 and 2 seem like the better option, while merging 1 and 3 seem better if the mode is more important. We have $w_{1,2} = \frac{5}{6}$, $\mu_{1,2} = 1.6$ and $\sigma^2_{1,2} \approx 1.28$, and $w_{1,3} = \frac{2}{6}$, $\mu_{1,3} = 2.25$ and $\sigma^2_{1,3} \approx 2.46$.

iii. $w = \{\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\}$, $\mu = \{0, 2, 4.5\}$ and $P = \{1^2, 1.5^2, 1.5^2\}$.

Solution: This seem very similar to the mixture in i., although the variances are different and can change the situation. The plot confirms this. Merging 1 and 2 now stretches out and shifts the first mode to much, while merging 2 and 3 keeps the modes whlie only slightly overestimating the second mode. Again, merging 1 and 3 is naturally a catastrophy compared to the others. Thus merging 2 and 3 is the best option, with $w_{2,3} = \frac{2}{3}$, $\mu_{2,3} = 3.25$ and $\sigma^2_{2,3} \approx 1.95$.

iv. $w = \{\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\}$, $\mu = \{0, 0, 2.5\}$ and $P = \{1^2, 1.5^2, 1.5^2\}$.

Solution: This is very similar to the mixture in iii., only with the two last means shifted to the left so that the two first coincide. Merging 1 and 2 now only stretches out the (only) mode to much lowering its peak, while merging 2 and 3 keeps the mode whlie only slightly overestimating the right "shoulder". Again, merging 1 and 3 is naturally a catastrophy compared to the others. Thus merging 2 and 3 is still the the best option, even with the the means of 1 and 2 coninciding. The merge gives $w_{2,3} = \frac{2}{3}$, $\mu_{2,3} = 1.25$ and $\sigma^2_{2,3} = 1.95$.

Task 2: *Measurement likelihood of the interacting multiple model filter and particle filter*

Hint: These should not end up being very complicated.

(a) Derive the measurement likelihood, $p(z_k|z_{1:k-1})$, of an IMM filter by using of the total probability theorem,

$$p(z_k|z_{1:k-1}) = \sum_{s_k} \int p(z_k|x_k, s_k)\, p(x_k|s_k, z_{1:k-1})\, \Pr(s_k|z_{1:k-1})\, \mathrm{d}x_k,$$

in terms of the mode likelihood, $\Lambda_k^{s_k}$, and mode probabilities, $\Pr(s_k|z_{1:k-1})$.

Hint: Use terms and equations from section 6.4.

Solution: Simply inserting equation (6.31) directly gives

$$p\left(z_k|z_{1:k-1}\right) = \sum_{s_k} \int p\left(z_k|x_k, s_k\right) p\left(x_k|s_k, z_{1:k-1}\right) \Pr(s_k|z_{1:k-1})\, \mathrm{d}x_k = \sum_{s_k} \Lambda_k^{s_k} \Pr(s_k|z_{1:k-1})\,.$$

(b) Assume that the PF state distribution is $p(x_k|z_{1:k-1}) \approx \sum_{i=1}^{N} w_k^i \delta(x_k - x_k^i)$, where $\delta(x)$ is the dirac delta function, w_k^i are the normalized weights and x_k^i are the particles. Derive the measurement likelihood, $p\left(z_k|z_{1:k-1}\right)$, of this PF.

Hint: You can use the total probability theorem to be able to use the given approximate PF state distribution.

Solution: Using the hint and the given approximation

$$\begin{aligned} p\left(z_k|z_{1:k-1}\right) &= \int p\left(z_k|x_k\right) p\left(x_k|z_{1:k-1}\right) \mathrm{d}x_k \\ &\approx \int p\left(z_k|x_k\right) \sum_{i=1}^{N} w_k^i \delta(x_k - x_k^i)\, \mathrm{d}x_k = \sum_{i=1}^{N} p\left(z_k|x_k^i\right) w_k^i. \end{aligned}$$

Task 3: *Implement an IMM class*

Use the MATLAB skeleton for an IMM class that is available at Blackboard. The function and class definition has to be as given, but you are free to change any other prewritten code, if you want. You have to implement steps 1 through 4 of the IMM algorithm as given in the book, in addition to a 5th estimation step. The 5th step consists of calculating the overall mean and covariance of the state. Step 3, mode matched filtering, will further be divided into a mode matched prediction and mode matched update step. You will then combine these steps into the higher level functions of predict and update.

Hint: For step 2 and 5 you can use the function reduceGaussianMixture you made in task 1.

(a) Implement step 1 in the function `[spredprobs, smixprobs] = mixProbabilities(obj, sprobs)`, which should return the predicted mode probability and the mixing probabilities based on the transistion matrix and the previous mode probabilities.

Solution:

```
% Joint probability for this model and next
spsjointprobs = obj.PI .* (ones(obj.M, 1) * sprobs(:)');

% marginal probability for next model
spredprobs = sum(spsjointprobs, 2);

% conditionional probability for model at this time step on the
   next.
smixprobs = spsjointprobs ./ (spredprobs * ones(1, obj.M));
```

(b) Implement step 2 in the function `[xmix, Pmix] = mixStates(obj, smixprobs, x, P)`, which should return the mixed mean and covariance based on the mixing probabilities and the previous means and covariances.

Solution:

```
for s = 1:obj.M
   [xmix(:, s), Pmix(:, :, s)] = reduceGaussMix(smixprobs(s, :),
      x, P);
end
```

(c) Implement the prediction part of step 3 in the function `[xpred, Ppred] = modeMatchedPrediction(obj, x, P, Ts)`, which should output the EKF predictions according to the mode.

Solution:

```
for s = 1:obj.M
   [xpred(:, s), Ppred(:, :, s)] = ...
      obj.modeFilters{s}.predict(x(:, s), P(:, :, s), Ts);
end
```

(d) Combine the above into a overall predict function `[sprobspred, xpred, Ppred] = predict(obj, sprobs, x, P, Ts)`.

Solution:

```
% step 1
[sprobspred, smixprobs] = obj.mixProbabilities(sprobs);

% step 2
[xmix, Pmix] = obj.mixStates(smixprobs, x, P);

% prediction part of step 3
[xpred, Ppred] = obj.modeMatchedPrediction(xmix, Pmix, Ts);
```

(e) Implement the update part of step 3 in the function `function [xupd, Pupd, logLambdas] = modeMatchedUpdate(obj, z, sprobs, x, P)`, which should update the mean and covariance along with calculation of the measurement log likelihood, $\log(\Lambda_k^{s_k})$, according to the mode.

Solution:

```
for s = 1:obj.M
   filter = obj.modeFilters{s};
   [xupd(:, s), Pupd(:, :, s)] = filter.update(z, x(:, s), P(:,
      :, s));
   logLambdas(s) = filter.loglikelihood(z, x(:, s), P(:, :, s));
end
```

(f) Implement step 4 in `[supdprobs, loglikelihood] = updateProbabilities(obj, logLambdas, sprobs)`, which should update the mode probabilities and calculate the overal measurement log likelihood based on the prior mode probabilities and the mode measurement log likelihood.

It is more numerically stable to do this in logarithms and only exponentiate to get probabilities in the end, but not strictly neccesarry. However, a function logSumExp (calculates the logarithm of a sum of exponentiated terms in an array in a more numerically stable way than the naive approach) has been given in the bottom of the file to help with this.

Solution:

```
supdprobs = logLambdas(:) + log(sprobs(:));
loglikelihood = logSumExp(supdprobs);
supdprobs = exp(supdprobs - loglikelihood);
```

(g) Combine mode matched update and probability update into an overal update function `[supdprobs, xupd, Pupd, loglikelihood] = update(obj, z, sprobs, x, P)`.

Solution:

```
[xupd, Pupd, logLambdas] = obj.modeMatchedUpdate(z, x, P);
[supdprobs, loglikelihood] = obj.updateProbabilities(logLambdas,
   sprobs);
```

(h) Implement the function that gives the state estimate of the IMM, in terms of a mean and covariance, in the function `function [xest, Pest] = estimate(obj, sprobs, x, P)`

Solution:

```
[xest, Pest] = reduceGaussMix(sprobs, x, P);
```

(i) Take a look at the function NIS and see that you understand what it is doing. This function might come in handy for tuning the filter.

Solution: It calculates an averaged innovation and then calculates a NIS based on that, along with the NIS for each mode. These are now not necessarily chi squared anymore as the filter is using an approximations.

Task 4: *Tune an IMM*

You are to tune an IMM consisting of a CV and CT model with position measurement on the data in the given .mat file. The data has been created by simulating these two models with deterministic switching times and setting a new turn rate at the same time. You are, however, to estimate these switching times along with the trajectory by tuning both filters along with the switching probabilities. The initial state is sampled randomly, and the first measurement is from this initial position. A live script can be found on Blackboard to get you started.

Both the CV model and CT models can be found on Blackboard. The CV model is the same as the one you made in the previous assignment except for having added some zero padding to be able to combine it with the 5 states of the CV model. The EKF module is the one you were supposed to implement in assigment 2.

You can try to use NIS and/or NEES, but an optimally tuned filter will not necessarily be within the confidence bounds given by the χ^2 distribution, altough it should at least be close. Quantities of intereset are low RMSE/MSE, low peak deviation, and good maneuver start/end time estimates.

Hint: The models in IMM must have sufficiently different noise properties if one are to estimate maneuver times. If the models are sufficiently different, the transition probabilities determine how ”willing” the filter is to switch to a new model, which gives a tradeoff between maneuver detection and precision. The IMM is a suboptimal filter so the tuning process might have to compensate for that (the process with much noise will inflate the covariance of the process with low noise etc. when mixing).

(a) Perform a crude tuning of a CV model to the data by getting a consistent NIS, without calculating errors to the ground truth. Then do the same for the CT model. Briefly discuss.

Hint: This is very similar to what you did in assignment 2.

Solution: Using the parameters under gives ANIS of 2.24 for the CV model and 2.23 for the CT model, which is between 1.63 and 2.41 corresponding to the chi squared confidence intervall $[0.025, 0.975]$.

```
% [...]
r = 5;
qCV = 0.05;
qCT = [0.005, 0.000025];

%[...]

% initialize filter
xbar(:, 1) = init.x;
Pbar(:, : ,1) = init.P;

% filter
for k = 1:K
   NISes(k) = models{s}.NIS(Z(:, k), xbar(:, k), Pbar(:, :, k));
   [xhat(:, k), Phat(:, :, k)] = models{s}.update(Z(:, k), xbar
      (:, k), Pbar(:, :, k));
   if k < K
      [xbar(:, k+1), Pbar(:, :, k+1)] = models{s}.predict(xhat(:,
          k), Phat(:, :, k), Ts);
   end
end

%[...]

% consistency
chi2inv([0.025, 0.975], K*2)/K
ANIS = mean(NISes)
```

(b) Tune the IMM without comparing to ground truth. Briefly describe your tuning process and why you chose the parameters. Specifically mention how the differences in parameters are changed from having sinlge filters.

Solution: The parameters in the code under makes NIS consistent and seem to give smooth results along with decent model swithcing estimates. The noise in the low noise CV model can be greatly reduced compared to when it was used as a single model as it does not have to take the maneuvering into account. We can also increase the process noise somewhat on the CT model as it does not need to be as precise in the straight line motion regions. The

π matrix decides how "willing" the estimator is to switch models. Keeping the probability for not switching high gives clearer specific jumps between the mode estimates, whereas higher proability for switching gives more fluctuating mode estimates.

```
r = 5;
qCV = 0.0025;
qCT = [0.005, 0.00005];
PI = [0.95, 0.05; 0.05, 0.95]; assert(all(sum(PI, 1) == [1, 1]),'
    columns of PI must sum to 1')

% make model
models =  cell(2,1);
models{1} = EKF(discreteCVmodel(qCV, r));
models{2} = EKF(discreteCTmodel(qCT, r));
imm = IMM(models, PI);

%[...]

% initialize
xbar(:, :, 1) = repmat([0; 0; 2; 0; 0], [1, 2]);
Pbar(:, : ,:, 1) = repmat(diag([25, 25, 3, 3, 0.0005].^2)
    ,[1,1,2]);
probbar(:, 1) = [0.9; 0.1];

% filter
for k=1:100
    [NIS(k), NISes(:, k)] = imm.NIS(Z(:, k), probbar(:,k), xbar
        (:,:,k), Pbar(:,:,:,k));
    [probhat(:, k), xhat(:, :, k), Phat(:, :, :, k)] = ...
        imm.update(Z(:, k), probbar(:, k), xbar(:, :, k), Pbar(:,
            :, :, k));
    [xest(:, k), Pest(:, :, k)] = imm.estimate(probhat(:, k), xhat
        (: ,:, k), Phat(:, :, :, k));
    if k < 100
        [probbar(:, k+1), xbar(:, :, k+1), Pbar(:, :, :, k+1)] =
            ...
            imm.predict(probhat(:, k), xhat(:, :, k), Phat(:, :, :,
                k), Ts);
    end
end

% consistency
chi2inv([0.025, 0.975], K*2)/K
ANIS = mean(NIS)
```

(c) Now tune by comparing to the ground truth. Did you have to change your parameters? Was it easier than without ground truth?

Solution: The parameters from the last task gave a bit high NEES, at 4.99 which is outside the 95% chi squared confidence interval at [3.4648 4.5731]. Increasing the turn rate noise to the value given below made both NEES and NIS inside the confidence bounds while also decreasing the "RMSE" somewhat at the expense of slightly higher peak deviations. Tuning using only the measurements gives good enough results, but it is hard to know how good the state estimates are without knowing ground truth. Knowing ground truth makes the evaluation easier and thus parameter selection more certain.

```
r = 5;
qCV = 0.0025;
qCT = [0.005, 0.0005];
PI = [0.95, 0.05; 0.05, 0.95]; assert(all(sum(PI, 1) == [1, 1]),'
    columns of PI must sum to 1')

%[...]

% initialize
xbar(:, :, 1) = repmat([0; 0; 2; 0; 0], [1, 2]);
Pbar(:, : ,:, 1) = repmat(diag([25, 25, 3, 3, 0.0005].^2)
    ,[1,1,2]);
probbar(:, 1) = [0.9; 0.1];

% filter
for k=1:100
    [NIS(k), NISes(:, k)] = imm.NIS(Z(:, k), probbar(:,k), xbar
        (:,:,k), Pbar(:,:,:,k));
    [probhat(:, k), xhat(:, :, k), Phat(:, :, :, k)] = ...
    i  mm.update(Z(:, k), probbar(:, k), xbar(:, :, k), Pbar(:, :,
        :, k));
    [xest(:, k), Pest(:, :, k)] = imm.estimate(probhat(:, k), xhat
        (: ,:, k), Phat(:, :, :, k));
    NEES(k) = (xest(1:4,k) - Xgt(1:4,k))' * (Pest(1:4, 1:4, k) \ (
        xest(1:4,k) - Xgt(1:4,k)));
    if k < 100
        [probbar(:, k+1), xbar(:, :, k+1), Pbar(:, :, :, k+1)] =
            ...
            imm.predict(probhat(:, k), xhat(:, :, k), Phat(:, :, :,
                k), Ts);
    end
end

% errors
poserr = sqrt(sum((xest(1:2,:) - Xgt(1:2,:)).^2, 1));
posRMSE = sqrt(mean(poserr.^2)); % not true RMSE (which is over
    monte carlo simulations)
velerr = sqrt(sum((xest(3:4, :) - Xgt(3:4, :)).^2, 1));
velRMSE = sqrt(mean(velerr.^2)); % not true RMSE (which is over
    monte carlo simulations)
peakPosDeviation = max(poserr);
peakVelDeviation = max(velerr);

% consistency
chi2inv([0.025, 0.975], K*2)/K
ANIS = mean(NIS)
```

```
chi2inv([0.025, 0.975], K*4)/K
ANEES = mean(NEES)
```

(d) Generate some new data a couple of times with `K = 100; Ts = 2.5; r = 5; q = [0.005, 1e-6*pi];` and run your tuned filter on it. How does it perform? Does it seem to generalize well or does it seem to be tuned for the particular scenario?

Solution: It does OK on new data. The NEES is usually not consistent, whereas the NIS is. The peak deviations is fluctuating a lot, and the model switching times is also not as good as one would hope for. It can seem that the tuning is slightly biased toward the data set, but not overly so. This is not investigated further, but could be done by performing several Monte Carlo simulations and analysing the resulting statistics.