TTK4250 Sensor Fusion
Solution to Assignment 4

Task 1: *Gaussian mixture reduction*

(a) In the file `mixturereduction.py`, implement the Python function

```python
def gaussian_mixture_moments(
    w: np.ndarray,  # the mixture weights shape=(N,)
    x: np.ndarray,  # the mixture means shape(N, n)
    P: np.ndarray,  # the mixture covariances shape (N, n, n)
) -> Tuple[
    np.ndarray, np.ndarray
]:  # the mean and covariance of of the mixture shapes ((n,), (n, n))
```

This should implement equations 6.19–6.21 in the book.

Solution: The code can be written as

```python
# mean
xbar = np.average(x, axis=0, weights=w)

# covariance
# # internal covariance
Pint = np.average(P, axis=0, weights=w)

# # spread of means
xdiff = x - xbar[None]
Pext = np.average(xdiff[:, :, None] * xdiff[:, None, :], axis=0, weights=w)

# # total
Pbar = Pint + Pext
```

(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?

The script `perform_mixture_reduction.py` 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 $w_1 p_1(x) + w_2 p_2(x) + w_3 p_3(x) = w_1 p_1(x) + (w_2 + w_3)\left(\frac{w_2}{w_2+w_3} p_2(x) + \frac{w_3}{w_2+w_3} p_3(x)\right)$. There is not

necessarily a one true answer here, and it depends on what you emphasize.

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 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. Merging 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" caused 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 while only slightly overestimating the second mode. Again, merging 1 and 3 is naturally a catastrophe 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 while only slightly overestimating the right "shoulder". Again, merging 1 and 3 is naturally a catastrophe compared to the others. Thus merging 2 and 3 is still the the best option, even with the the means of 1 and 2 coinciding. 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,

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

in terms of the mode likelihood, $\Lambda_k^{s_k}$, and mode probabilities, $\Pr(s_k|z_{1:k-1})$, assuming $p\left(z_k|x_k, s_k\right) = p\left(z_k|x_k\right)$.

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 Python skeleton for an IMM class that is available at Blackboard. The function and class definitions 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

```
def mix_probabilities(
    self,
    immstate: MixtureParameters[MT],
    # sampling time
    Ts: float,
) -> Tuple[
    np.ndarray, np.ndarray
]:  # predicted_mode_probabilities, mix_probabilities: shapes = ((M, (M ,M))).
    # mix_probabilities[s] is the mixture weights for mode s
```

which should return the predicted mode probability and the mixing probabilities based on the transition matrix and the previous mode probabilities.

This is simplified by completing the file `discretebayes.py` and making the function

```
def discrete_bayes(
    # the prior: shape=(n,)
    pr: np.ndarray,
    # the conditional/likelihood: shape=(n, m)
    cond_pr: np.ndarray,
) -> Tuple[
    np.ndarray, np.ndarray
]:  # the new marginal and conditional: shapes=((m,), (m, n))
```

Solution: The mix probabilities can be done as

```
predicted_mode_probabilities, mix_probabilities = discretebayes.discrete_bayes(
    immstate.weights, self.PI
)
```

while discrete Bayes can be done as

```
joint = cond_pr * pr[:, None]

marginal = joint.sum(axis=0)

# Take care of rare cases of degenerate zero marginal,
conditional = np.divide(
    joint,
    marginal[None],
    out=np.repeat(pr[:, None], joint.shape[1], 1),
    where=marginal[None] > 0,
)

# flip axes
conditional = conditional.T
```

(b) Implement step 2 in the function

```
def mix_states(
    self,
    immstate: MixtureParameters[MT],
    # the mixing probabilities: shape=(M, M)
    mix_probabilities: np.ndarray,
) -> List[MT]:
```

which should return the mixed mean and covariance based on the mixing probabilities and the previous means and covariances.

To make this generalize and reuse code for the upcoming PDA you can implement the function to do the work in the EKF class

```
def reduce_mixture(
    self, ekfstate_mixture: MixtureParameters[GaussParams]
) -> GaussParams:
    """Merge a Gaussian mixture into single mixture"""
```

Solution:

```
mixed_states = [
    fs.reduce_mixture(MixtureParameters(mix_pr_s, immstate.components))
    for fs, mix_pr_s in zip(self.filters, mix_probabilities)
]
```

The EKF mixture reduction code can be implemented as

```
def reduce_mixture(
    self, ekfstate_mixture: MixtureParameters[GaussParams]
) -> GaussParams:
    """Merge a Gaussian mixture into single mixture"""
    w = ekfstate_mixture.weights
    x = np.array([c.mean for c in ekfstate_mixture.components], dtype=float)
    P = np.array([c.cov for c in ekfstate_mixture.components], dtype=float)
    x_reduced, P_reduced = mixturereduction.gaussian_mixture_moments(w, x, P)
    return GaussParams(x_reduced, P_reduced)
```

(c) Implement the prediction part of step 3 in the function

```
def mode_matched_prediction(
    self,
    mode_states: List[MT],
    # The sampling time
    Ts: float,
) -> List[MT]:
```

which should output the EKF predictions according to the mode.

Solution:

```
modestates_pred = [
    fs.predict(cs, Ts) for fs, cs in zip(self.filters, mode_states)
]
```

(d) Combine the above into a overall predict function

```
def predict(
    self,
    immstate: MixtureParameters[MT],
    # sampling time
    Ts: float,
) -> MixtureParameters[MT]:
```

Solution:

```
predicted_mode_probability, mixing_probability = self.mix_probabilities(
    immstate, Ts
)

mixed_mode_states: List[MT] = self.mix_states(immstate, mixing_probability)

predicted_mode_states = self.mode_matched_prediction(mixed_mode_states, Ts)
```

```
predicted_immstate = MixtureParameters(
    predicted_mode_probability, predicted_mode_states
)
```

(e) Implement the update part of step 3 in the function

```
def mode_matched_update(
    self,
    z: np.ndarray,
    immstate: MixtureParameters[MT],
    sensor_state: Optional[Dict[str, Any]] = None,
) -> List[MT]:
```

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:

```
updated_state = [
    fs.update(z, cs, sensor_state=sensor_state)
    for fs, cs in zip(self.filters, immstate.components)
]
```

(f) Implement step 4 in

```
def update_mode_probabilities(
    self,
    z: np.ndarray,
    immstate: MixtureParameters[MT],
    sensor_state: Dict[str, Any] = None,
) -> np.ndarray:
```

which should update the mode probabilities and calculate the overall measurement log likelihood based on the prior mode probabilities and the mode measurement log likelihood.

For generalization and for use with PDA you might want to implement

```
def loglikelihood(
    self,
    z: np.ndarray,
    ekfstate: GaussParams,
    sensor_state: Optional[Dict[str, Any]] = None,
) -> float:
    """Calculate the log likelihood of ekfstate at z in sensor_state"""
```

It might be more numerically stable to do this in logarithms and only exponentiate to get probabilities in the end, but not strictly necessary. However, a function `logsumexp`[1] has been imported for you.

Solution:

[1] Calculates the logarithm of a sum of exponentiated terms in an array in a more numerically stable way than the naive approach

```
loglikelihood = np.array(
    [
        fs.loglikelihood(z, cs, sensor_state=sensor_state)
        for fs, cs in zip(self.filters, immstate.components)
    ]
)

logjoint = loglikelihood + np.log(immstate.weights)

updated_mode_probabilities = np.exp(logjoint - logsumexp(logjoint))
```

The EKF log-likelihood is implemented as

```
        for fs, mode_s_cond_comp_prob, mode_s_comp in zip(
            self.filters,
            mode_conditioned_component_prob,
            zip(*immstate_mixture.components),
        )
    ]

    immstate_reduced = MixtureParameters(mode_prob, mode_states)

    return immstate_reduced

def estimate(self, immstate: MixtureParameters[MT]) -> GaussParams:
    """Calculate a state estimate with its covariance from immstate"""

    # ! assuming all the modes have the same reduce and estimate
    dataRed = self.filters[0].reduce_mixture(immstate)
    return self.filters[0].estimate(dataRed)

def gate(
    self,
```

(g) Combine mode matched update and probability update into an overall update function

```
def update(
    self,
    z: np.ndarray,
    immstate: MixtureParameters[MT],
    sensor_state: Dict[str, Any] = None,
) -> MixtureParameters[MT]:
```

Solution:

```
updated_weights = self.update_mode_probabilities(
    z, immstate, sensor_state=sensor_state
)
updated_states = self.mode_matched_update(
    z, immstate, sensor_state=sensor_state
)
```

```
updated_immstate = MixtureParameters(updated_weights, updated_states)
```

(h) Implement the function that gives the state estimate of the IMM, in terms of a mean and covariance, in the function

```
def estimate(self, immstate: MixtureParameters[MT]) -> GaussParams:
```

Solution:

```
# ! assuming all the modes have the same reduce and estimate
dataRed = self.filters[0].reduce_mixture(immstate)
return self.filters[0].estimate(dataRed)
```

(i) Take a look at the functions `NISes` and `NEESes` see that you understand what it is doing. This function might come in handy for tuning the filter. Also have a look at `estimate_sequence`.

Solution: They calculate an averaged innovation/difference and then calculates a NIS/NEES based on that, along with the NIS/NEES for each mode. These are now not necessarily chi squared anymore as the filter is using 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 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 functionality as 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 assignment 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, although it should at least be close. Quantities of interest 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 both the CV model and CT model, which is between 1.68 and 2.33 corresponding to the chi squared confidence interval $[0.05, 0.95]$.

```
# %% tune single filters
sigma_z = 2.25
```

```
sigma_a_CV = 0.22
sigma_a_CT = 0.07
sigma_omega = 0.0016 * np.pi
```

(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 single filters.

Note: Again, do not spend several hours on this. You should get a feel for the algorithm and see what ambiguity it can give to not have ground truth.

Solution: The parameters in the code under makes NIS consistent and seem to give smooth results along with decent model switching 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 probability for switching gives more fluctuating mode estimates.

```
sigma_z = 2.25
sigma_a_CV = 0.05
sigma_a_CT = 0.07
sigma_omega = 0.007
PI = np.array([[0.95, 0.05], [0.05, 0.95]])
```

(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 4.82 which is outside the 90% chi squared confidence interval at $[3.54, 4.48]$. 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.

```
sigma_z = 2.25
sigma_a_CV = 0.05
sigma_a_CT = 0.07
sigma_omega = 0.0225
PI = np.array([[0.95, 0.05], [0.05, 0.95]])
```

Task 5: *Implement a SIR particle filter for a pendulum*

We want to estimate the position of a pendulum influenced by some random disturbance, with a suboptimal measuring device.

The pendulum dynamic equation is given by $\ddot{\theta} = \frac{g}{l}\sin(\theta) - d\dot{\theta} + \alpha$, where g is the gravitational acceleration, l is the length from the center of rotation to the pendulum, and d is the dampening coefficient. The pendulum is otherwise assumed to be a point mass attached to a massless rod. α represents other forces that acts on the pendulum in the form of a stochastic angular acceleration process. θ is 0 when the rod is at the bottom.

A measuring device is set up L_d below the rotation point, and L_l to the left so that $z_k = h(\theta_k) = \sqrt{(L_d - \cos(\theta_k))^2 + (\sin(\theta_k) - L_l)^2} + w_k$, where $w_k \sim \text{triangle}(E[w_k], a)$, a symmetric triangle distribution with width $2a$, height $\frac{1}{a}$ and peak at $E[w_k]$.

A skeleton that creates data and provides some plot aid and so on can be found on Blackboard.

(a) Finish the particle filter. To finish it you need to implement particle prediction, weight update and resampling (algorithm 3). How does the filter perform?

Solution: It seems to perform OK with around 300 particles. The estimate is naturally bi modal. More particles seems slightly better, less seems to (erroneously) loose the multimodality and by chance pick one of them. At the bottom the measurements are not very sensitive to the position, so the estimate becomes wide.

```
# number of particles to use
N = 300

# initialize particles, pretend you do not know where the pendulum starts
px = np.array([rng.uniform(-np.pi, np.pi, N), rng.normal(0, np.pi / 4, N)]).T

# initial weights
w = np.ones(N) / N

...

for k in range(K):
    print(f"k = {k}")
    # weight update
    for n in range(N):
        w[n] = PF_measurement_distribution.pdf(Z[k] - h(px[n], Ld, l, Ll))
    w = w / w.sum()

    # resample
    cumw = np.cumsum(w)
    i = 0
    for n in range(N):
        u = (n + 1) / N - 10 * eps  # remove some eps for convergence
        while u > cumw[i]:
            i += 1

        pxn[n] = px[i]

    # trajecory sample prediction
    for n in range(n):
        vkn = PF_dynamic_distribution.rvs()
        px[n] = pendulum_dynamics_discrete(pxn[n], vkn, Ts, a, d)
```

(b) Do not resample the simulated pendulum, but vary L_l and generate new measurements to see how it changes the estimate. Does it remove any problems in the estimation you found above?

Solution: Already at $L_l = 0.1$ the estimate seems to find the right mode after a little while, so this setup seems much better. This becomes more prominent for higher values of L_l.

(c) When would you choose to try a PF instead of a EKF and why?

Solution: If the model is highly nonlinear with high noise, multimodality and/or highly skewed we can assume that an EKF will struggle and a PF might do better. High noise in a highly nonlinear model means large covariances and large probability for the linearization at the mean to give poor description of the truth. A mean and covariance is not a very good descriptor of something that is multi modal. High skewness along with nonlinearities is an issue due to the EKF being symmetric in the sense that it only estimates mean and covariance.

In this case we only had a minor nonlinearity but the estimation problem was inherently bi modal so we can assume that an EKF would have struggled, clinging onto one of the modes at random.