Sensor Ø4

1 Gaussian mixture reduction: gaussian_mixture_moments

a) In the file mixturereduction.py, implement 6.19 - 6.21

$$\bar{\mu} = \sum_{i=1}^{M} w^i \mu^i \qquad (6.19) \quad \checkmark$$

The expectation of $\vec{x}$ over the mixture $f(\vec{x})$

$$\bar{P} = \sum_{i=1}^{M} w^i P^i + \tilde{P} \qquad (6.20) \quad \checkmark$$

The covariance of the Gaussian mixture $f(\vec{x})$ in (6.18)

$$\tilde{P} = \sum_{i=1}^{M} w^i(\mu^i - \bar{\mu})(\mu^i - \bar{\mu})^T = \sum_{i=1}^{M} w^i \mu^i (\mu^i)^T - \bar{\mu}\bar{\mu}^T \quad (6.21) \quad \checkmark$$

The spread-of-the-innovations term

$$f(\vec{x}) = \sum_{i=1}^{M} w^i \mathcal{N}(\vec{x}; \mu^i, P^i), \text{ where } \sum_{i=1}^{M} w^i = 1,\ w^i \geq 0\ \forall i \quad (6.18)$$

A Gaussian mixture

b) Perfor_mixture_reduction for visualization. Which mixtures 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?

$$w_1 P_1(x) + w_2 P_2(x) + w_3 P_3(x) = w_1 P_1(x) + (w_2 + w_3)\left(\frac{w_2}{w_2 + w_3} P_2(x) + \frac{w_3}{w_2 + w_3} P_3(x)\right)$$

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

Seems like mixture 1 and 2 would be the best merge. They have the most similar means. 2 and 3 would also work, but their means are a bit further apart. 1 and 3 results in a single mode distribution, and that is wheale.
The components resulting from merging 1 and 2 are:

$w_{1,2} = 2/3$, $\mu_{1,2} = 1$, $\sigma_{1,2}^2 = \sqrt{2}^2 = 2$

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

Same argument as before, I choose the merger of the mixtures with the closest means. The plots also suggest merging 1 and 2 is the best option:

$w_{1,2} = 5/6$, $\mu_{1,2} = 4/3$, $\sigma_{1,2}^2 = 1.42$

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

From the plot it is clear to see that merging 2 and 3 is the best option:

$w_{2,3} = \underline{2/3}$, $\mu_{2,3} = \underline{3.25}$, $\sigma^2_{2,3} = \underline{1.95}$

iv) $w = \{1/3, 1/3, 1/3\}$, $\mu = \{0, 0, 2.5\}$, $P = \{1^2, 1.5^2, 1.5^2\}$

Merging 2 and 3 looks to be the best option despite 1 and 2 having equal means:

$w_{2,3} = \underline{2/3}$, $\mu_{2,3} = \underline{5/6}$, $\sigma^2_{2,3} = \underline{1.95}$

2. Measurement likelyhood of the interactive model filter and particle filter

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

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

in terms of the mode likelyhood, $\Lambda_k^{s_k}$, and mode probabilities, $\Pr(s_k | z_{1:k-1})$, assuming $p(z_k | x_k, s_k) = p(z_k | x_k)$. Section 6.4 gives us the answer:

(6.32)

$$p(z_k | z_{1:k-1}) = \sum_{s_k} \Lambda_k^{s_k} \Pr(s_k | z_{1:k-1})$$ (the $\Pr(\ldots)$ expression does not depend on x_k)

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 likelyhood, $p(z_k | z_{1:k-1})$ of this PF.

We have that $p(z_k | z_{1:k-1}) = \int p(z_k | x_k) p(x_k | z_{1:k-1}) dx_k$

$$\overset{(*)}{\approx} \int p(z_k | x_k) \sum_{i=1}^{N} w_k^i \delta(x_k - x_k^i) dx_k$$

$$= \sum_{i=1}^{N} p(z_k | x_k^i) w_k^i$$

3. Implement an IMM class

The IMM method (6.4.1)

1. Calculation of mixing probabilities

mixing probabilities $\mu_{s_{k-1}|s_k}$, are the probabilities that model s_{k-1} was the true mode at time step $k-1$, given that model s_k is the true model at time step k:

$$\mu_{s_{k-1}|k} = \pi^{s_{k-1} s_k} p_{k-1}^{(s_k-1)}$$

See figs 2-11.

4 Tune an IMM

a) I could not get the entire script to run, and the plot for the confidence interval did not show up. However, I heard a rumor that ANIS needed to be on the interval $\{1.68, 2.33\}$. After some trial and error, I arrived at the values

sigma_z = 2.5

sigma_a_CV = 0.2

sigma_a_CT = 0.05

sigma_omega = 0.0012 * np.pi

Resulting in ANISes of 2.07 and 2.22.

b)

5. Implement a SIR filter for a pendulum

$$\ddot{\theta} = \frac{g}{l}\sin\theta - d\dot{\theta} + \alpha$$

$$z_k = h(\theta_k) = \sqrt{(L_d - \cos\theta_k)^2 + (\sin\theta_k - L_L)^2} + w_k$$

$w_k \sim \text{triangle}(E[w_k], a)$ is a symmetric triangle distribution with width $2a$, height $1/a$ and peak at $E[w_k]$

a) Used 400 particles, works just fine.

b) Chose an L_L of 0.15, works nice.

c) A linear model not subject to too much noise will work well for an EKF. However, a very nonlinear model subject to high noise and/or several modes will work better with a PF.

```python
def gaussian_mixture_moments(
    w: np.ndarray,  # the mixture weights shape=(N,)
    mean: np.ndarray,  # the mixture means shape(N, n)
    cov: np.ndarray,  # the mixture covariances shape (N, n, n)
) -> Tuple[
    np.ndarray, np.ndarray
]:  # the mean and covariance of the mixture shapes ((n,), (n, n))
    """Calculate the first two moments of a Gaussian mixture"""

    # mean
    mean_bar = w.dot(mean)  # TODO: hint np.average using axis and weights argument

    # covariance
    # # internal covariance
    cov_int = 0# TODO: hint, also an average
    for i in range(0, len(w)):
        cov_int = cov_int + w[i]*cov[i]

    # # spread of means
    # Optional calc:
    sum_term = 0
    for i in range(0, len(w)):
        sum_term = sum_term + w[i]*mean[i]*mean[i]
    mean_diff = mean_bar.dot(np.transpose(mean_bar))
    cov_ext = sum_term - mean_diff  # TODO: hint, also an average

    # # total covariance
    cov_bar = cov_int + cov_ext  # TODO

    return mean_bar, cov_bar
```

```python
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
    """Calculate the predicted mode probability and the mixing probabilities."""

    predicted_mode_probabilities, mix_probabilities = (
        discretebayes.discrete_bayes(immstate.weights, np.pi)
    )  # TODO hint: discretebayes.discrete_bayes

    # Optional assertions for debugging
    assert np.all(np.isfinite(predicted_mode_probabilities))
    assert np.all(np.isfinite(mix_probabilities))
    assert np.allclose(mix_probabilities.sum(axis=1), 1)

    return predicted_mode_probabilities, mix_probabilities
```

```python
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))
    """Swap which discrete variable is the marginal and conditional."""

    joint = cond_pr*pr[:, None]

    marginal = joint.sum(axis=0)

    # Take care of rare cases of degenerate zero marginal,
    conditional = np.divide(joint, marginal)

    # flip axes?? (n, m) -> (m, n)
    conditional = conditional.T

    # optional DEBUG
    assert np.all(
        np.isfinite(conditional)
    ), f"NaN or inf in conditional in discrete bayes"
    assert np.all(
        np.less_equal(0, conditional)
    ), f"Negative values for conditional in discrete bayes"
    assert np.all(
        np.less_equal(conditional, 1)
    ), f"Value more than on in discrete bayes"

    assert np.all(np.isfinite(marginal)), f"NaN or inf in marginal in discrete bayes"

    return marginal, conditional
```

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

    mixed_states = []
    for filters, mix in zip(self.filters, mix_probabilities): #Todo: simplify this using zip(filters, mix)
        mixed_states = [filters.reduce_mixture(MixtureParameters(mix, immstate.components))] #TODO
    return mixed_states
```

```python
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)
```

```python
def mode_matched_prediction(
    self,
    mode_states: List[MT],
    # The sampling time
    Ts: float,
) -> List[MT]:
    modestates_pred = [filters.predict(c, Ts) for filters, c in zip(self.filters, mode_states)]# TODO
    return modestates_pred
```

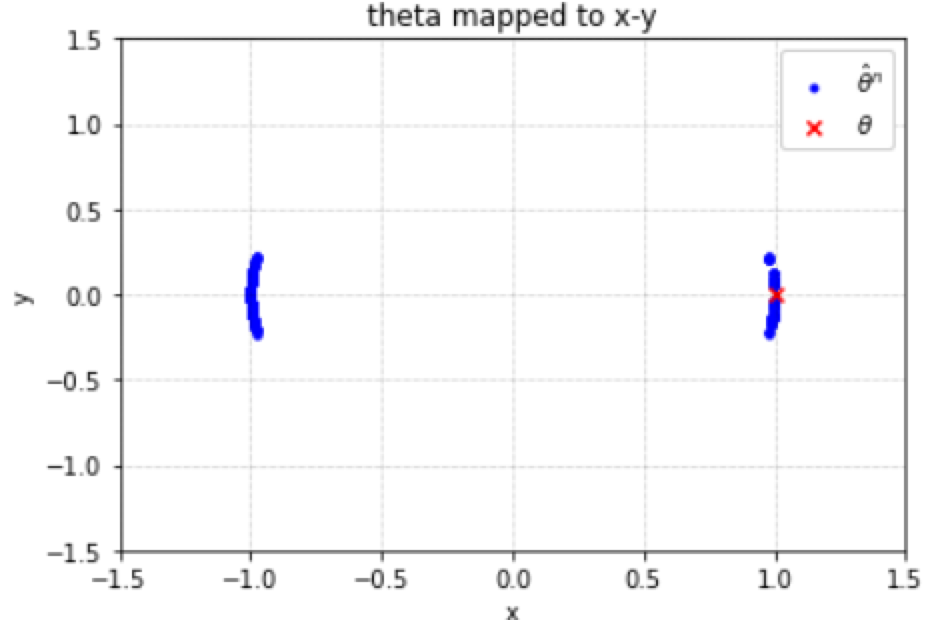
```python
def predict(
    self,
    immstate: MixtureParameters[MT],
    # sampling time
    Ts: float,
) -> MixtureParameters[MT]:
    """
    Predict the immstate Ts time units ahead approximating the mixture step.

    Ie. Predict mode probabilities, condition states on predicted mode,
    appoximate resulting state distribution as Gaussian for each mode, then predict each mode.
    """

    # TODO: proposed structure
    predicted_mode_probability, mixing_probability = self.mix_probabilities(immstate, Ts)# TODO

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

    predicted_mode_states = self.mode_matched_prediction(mixed_mode_states, Ts)# TODO

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

```python
def mode_matched_update(
    self,
    z: np.ndarray,
    immstate: MixtureParameters[MT],
    sensor_state: Optional[Dict[str, Any]] = None,
) -> List[MT]:
    """Update each mode in immstate with z in sensor_state."""

    updated_state = [filters.update(z, c, sensor_state=sensor_state) for filters, c in zip(self.filters, immstate.components)]# TODO

    return updated_state
```

```python
def update_mode_probabilities(
    self,
    z: np.ndarray,
    immstate: MixtureParameters[MT],
    sensor_state: Dict[str, Any] = None,
) -> np.ndarray:
    """Calculate the mode probabilities in immstate updated with z in sensor_state"""

    mode_loglikelihood = np.array([filters.loglikelihood(z, c, sensor_state=sensor_state)
                                   for filters, c in zip(self.filters, immstate.components)]) # TODO

    # potential intermediate step
    joint = mode_loglikelihood + np.log(immstate.weights)

    updated_mode_probabilities = np.exp(joint - logsumexp(joint))# TODO: will this work?

    # Optional debuging
    assert np.all(np.isfinite(updated_mode_probabilities))
    assert np.allclose(np.sum(updated_mode_probabilities), 1)

    return updated_mode_probabilities
```

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

    # ! You can assume all the modes have the same reduce and estimate function
    # ! and use eg. self.filters[0] functionality
    data_reduced = self.filters[0].reduce_mixture(immstate)# TODO
    estimate = self.filters[0].estimate(data_reduced)# TODO
    return estimate
```

```python
def update(
        self,
        z: np.ndarray,
        immstate: MixtureParameters[MT],
        sensor_state: Dict[str, Any] = None,
) -> MixtureParameters[MT]:
        """Update the immstate with z in sensor_state."""
        u_w = self.update_mode_probabilities(z, immstate, sensor_state=sensor_state)# TODO
        u_s = self.mode_matched_update(z, immstate, sensor_state=sensor_state)# TODO
        updated_immstate = MixtureParameters(u_w, u_s)
        return updated_immstate
```



theta mapped to x-y
$\hat{\theta}^n$
θ
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
y
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
x