Sensorfusion assignment 4

Bjørn Kåre Sæbø

Task 1

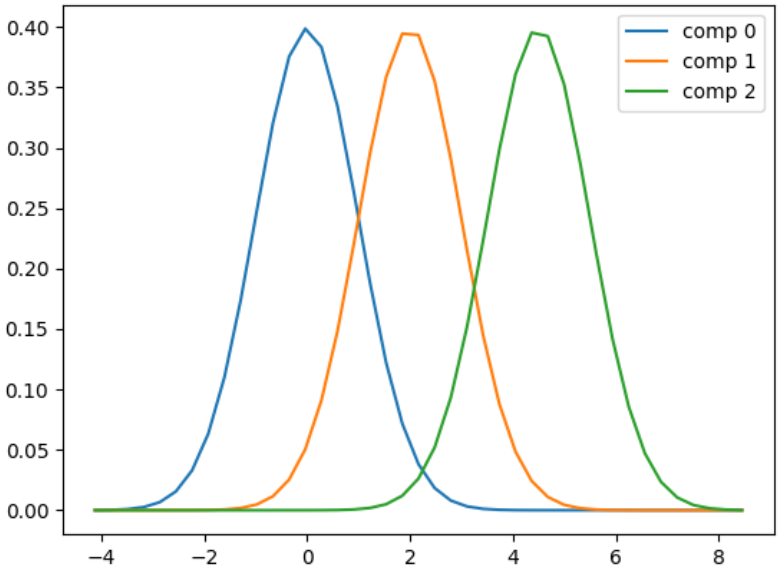
A)

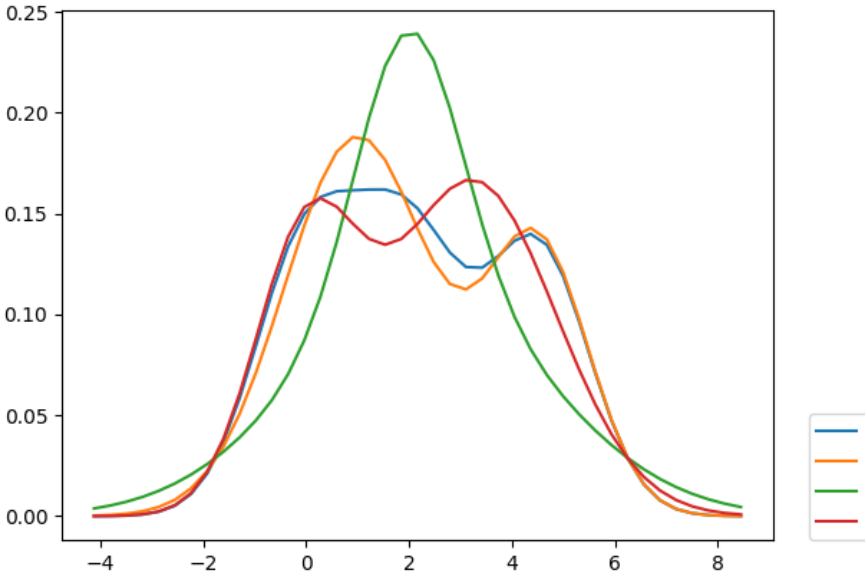
Implemented based on 6.19 - 6.22 in book

```
"""Calculate the first two moments of a Gaussian mixture"""

# mean
mean_bar = np.average(mean, weights=w, axis=0)

# covariance
# # internal covariance
cov_int = np.average(cov, weights=w, axis=0)

# # spread of means
# Optional calc: mean_diff =
mean_diff = np.diagonal((mean - mean_bar)@(mean - mean_bar).T)
cov_ext = np.average(mean_diff, weights=w, axis=0)

# # total covariance
cov_bar = cov_int + cov_ext

return mean_bar, cov_bar
```

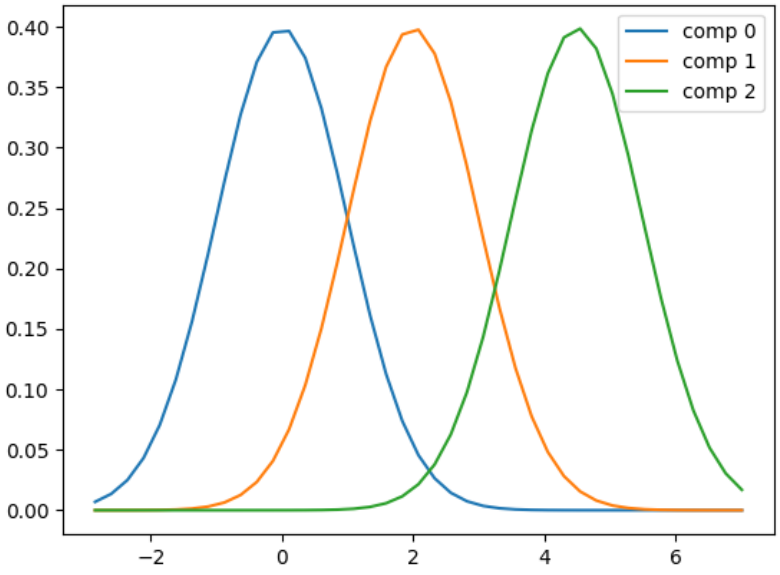
B)

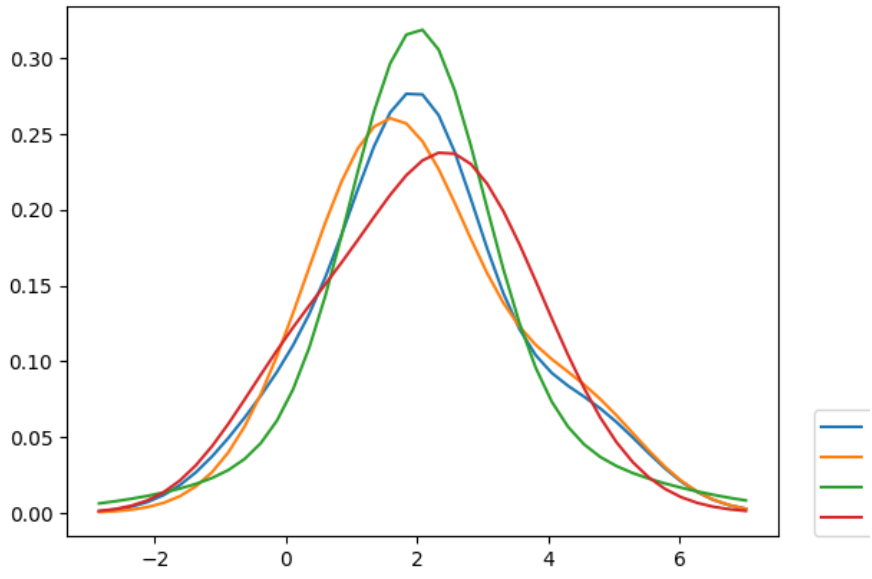
I) Would merge the outer distributions to get a wider distribution with mean in the middle of the three



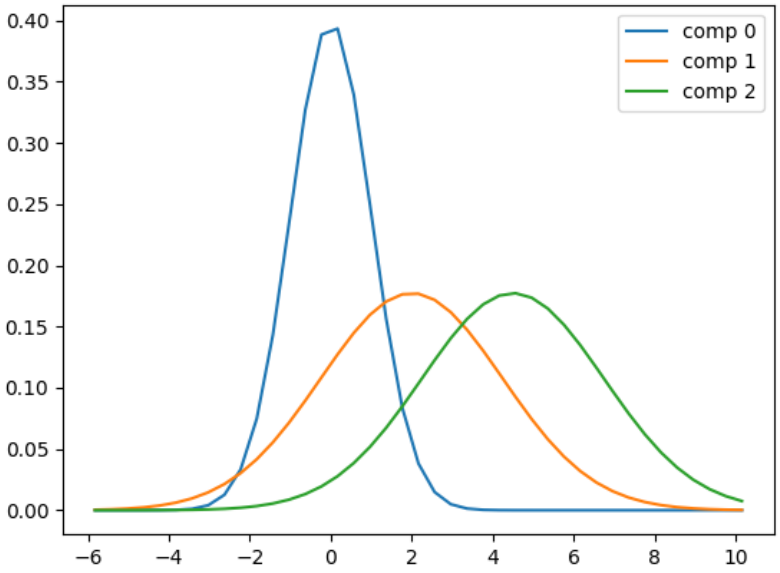

0.25
0.20
0.15
0.10
0.05
0.00
−4
−2
0
2
4
6
8

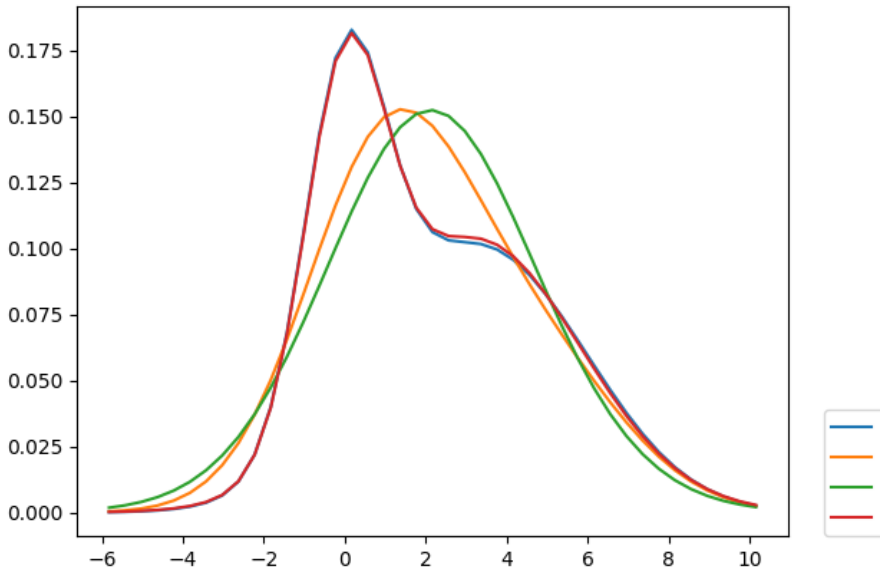
II) Doesn't make too much difference, as the weighting ensures that the middle distribution dominates the result anyway



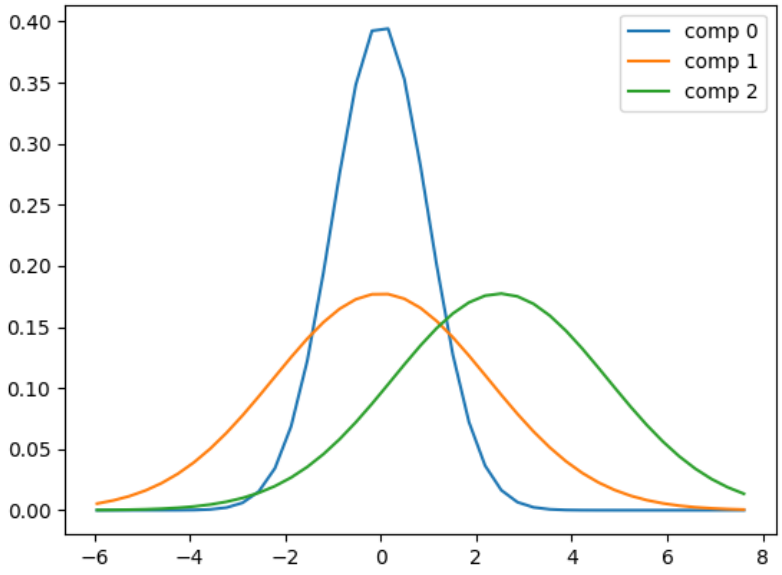

0.30
0.25
0.20
0.15
0.10
0.05
0.00
−2
0
2
4
6

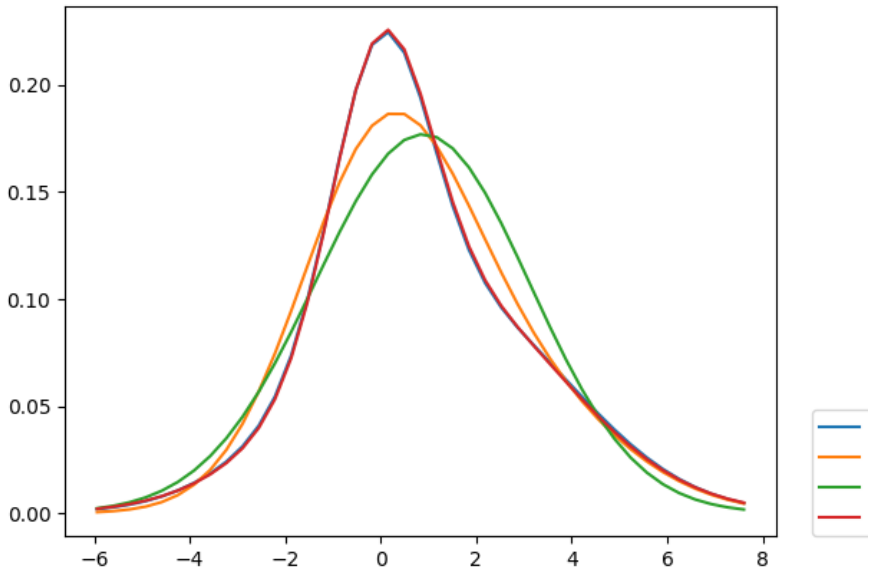
III) Would merge the two wide rightmost distributions, as they contribute much of the same info



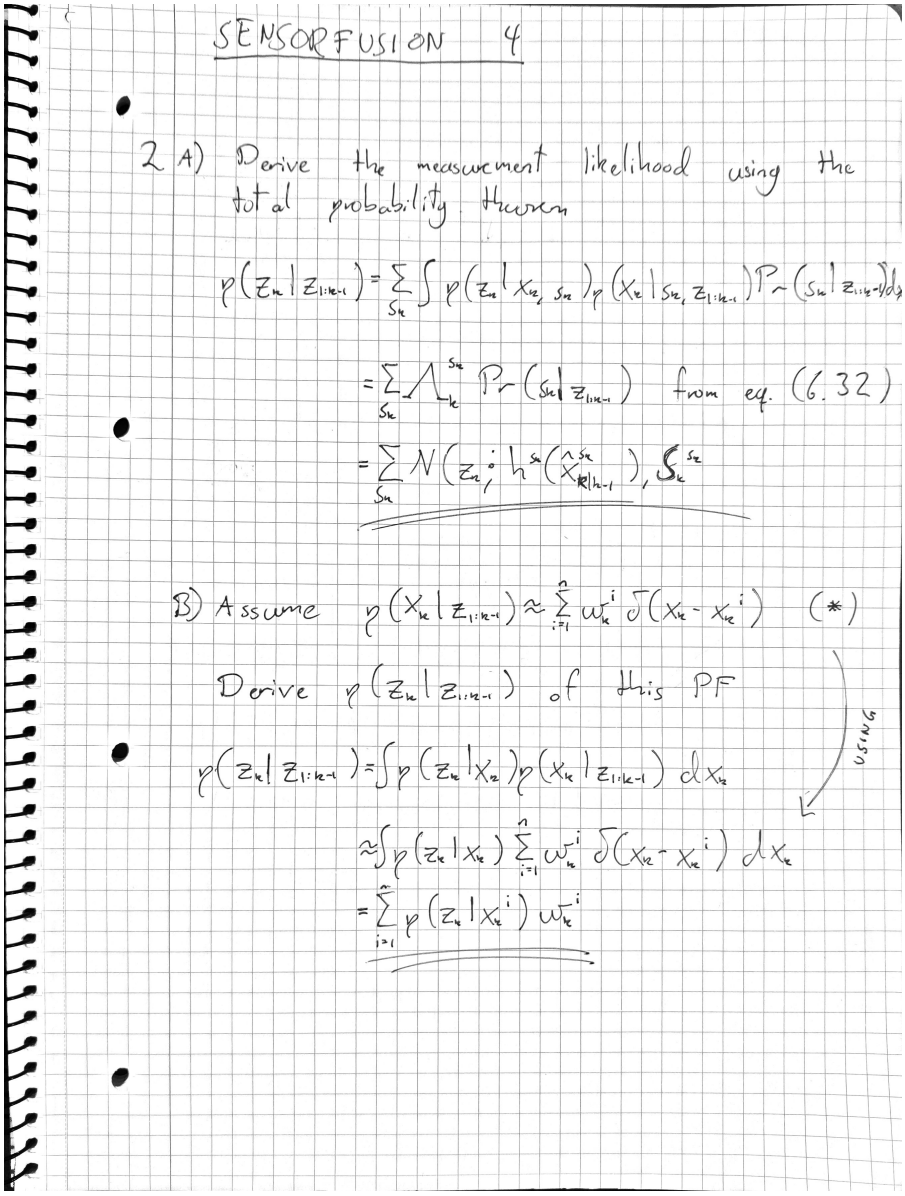

0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
−6
−4
−2
0
2
4
6
8
10

IV)Would merge the distributions with mean 0, and get a distribution with mean 0 and variance somewhere between the two originals




0.20
0.15
0.10
0.05
0.00
−6
−4
−2
0
2
4
6
8

Task 2



SENSORFUSION 4

2 A) Derive the measurement likelihood using the total probability theorem

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

$$= \sum_{s_k} \Lambda_k^{s_k}\, Pr(s_k \mid z_{1:k-1}) \quad \text{from eq. (6.32)}$$

$$= \sum_{s_k} N\left(z_k;\, h^{s_k}(\hat{x}_{k|k-1}^{s_k}),\, S_k^{s_k}\right)$$

B) Assume $p(x_k \mid z_{1:k-1}) \approx \sum_{i=1}^{n} w_k^i\, \delta(x_k - x_k^i)$ (*)

Derive $p(z_k \mid z_{1:k-1})$ of this PF

using

$$p(z_k \mid z_{1:k-1}) = \int p(z_k \mid x_k)\, p(x_k \mid z_{1:k-1})\, dx_k$$

$$\approx \int p(z_k \mid x_k) \sum_{i=1}^{n} w_k^{-i}\, \delta(x_k - x_k^i)\, dx_k$$

$$= \sum_{i=1}^{n} p(z_k \mid x_k^i)\, w_k^{-i}$$

Task 3 & 4

Spent a lot of time trying to get this to work, eventually gave up. Not available on friday, so can't bother the student assistant any more than I already have. Including code from IMM.py for reference

```python
class IMM(Generic[MT]):
    # The M filters the IMM relies on
    filters: List[StateEstimator[MT]]
    # the transition matrix. PI[i, j] = probability of going from model i to j: shape (M, M
    PI: np.ndarray
    # init mode probabilities if none is given
    initial_mode_probabilities: Optional[np.ndarray] = None

    def __post_init__(self):
        # This have to be satisfied!
        if not np.allclose(self.PI.sum(axis=1), 1):
            raise ValueError("The rows of the transition matrix PI must sum to 1.")

        # Nice to have a reasonable initial mode probability
        if self.initial_mode_probabilities is None:
            eigvals, eigvecs = linalg.eig(self.PI)
            self.initial_mode_probabilities = eigvecs[:, eigvals.argmax()]
            self.initial_mode_probabilities = (
                self.initial_mode_probabilities / self.initial_mode_probabilities.sum()
            )

    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,self.PI)
        )

        # 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

    def mix_states(
        self,
        immstate: MixtureParameters[MT],
```

```
    # the mixing probabilities: shape=(M, M)
    mix_probabilities: np.ndarray,
) -> List[MT]:
    mixed_states = [fl.reduce_mixture(MixtureParameters(mp, immstate.components))
                    for fl, mp in zip(self.filters, mix_probabilities)]
    return mixed_states

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

    pred = []
    for m in mode_states:
        for fl in self.filters:
            pred.append(fl.predict(m, Ts))

    return np.array(pred)

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 eac
    """

    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
    )
    return predicted_immstate

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 = np.array([fl.update(z, m, sensor_state) for fl, m in zip(self.filte
    return updated_state

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(
        [filt.loglikelihood(z, comp, sensor_state) \
         for filt, comp in zip(self.filters, immstate.components)])

    # potential intermediate step logjoint =

    predicted_mode_probabilities = immstate.weights # shape (M,1)
    normalization = \
        np.sum(np.exp(mode_loglikelihood) * predicted_mode_probabilities) # scalar floa

    log_pred_mode_probs = np.log(predicted_mode_probabilities) # shape (M,1)
    log_norm = np.log(normalization) # scalar float
    # (6.33)
    # mode_loglikelihood * predicted_mode_probabilities / normalization
    log_updated_mode_probs = \
        mode_loglikelihood + log_pred_mode_probs - log_norm

    updated_mode_probabilities = np.exp(log_updated_mode_probs)

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

    return updated_mode_probabilities

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."""

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

    updated_immstate = MixtureParameters(updated_weights, updated_states)
    return updated_immstate

def step(
    self,
    z,
    immstate: MixtureParameters[MT],
    Ts: float,
    sensor_state: Dict[str, Any] = None,
) -> MixtureParameters[MT]:
    """Predict immstate with Ts time units followed by updating it with z in sensor_sta

    predicted_immstate = self.predict(immstate, Ts)
    updated_immstate = self.update(z, immstate, Ts)

    return updated_immstate

def loglikelihood(  # Postponed until required
    self,
    z: np.ndarray,
    immstate: MixtureParameters,
    *,
    sensor_state: Dict[str, Any] = None,
) -> float:

    # THIS IS ONLY NEEDED FOR IMM-PDA. You can therefore wait if you prefer.

    mode_conditioned_ll = None # TODO in for IMM-PDA

    ll = None # TODO

    return ll

def reduce_mixture(
    self, immstate_mixture: MixtureParameters[MixtureParameters[MT]]
) -> MixtureParameters[MT]:
    """Approximate a mixture of immstates as a single immstate"""
```

```
        # extract probabilities as array
        weights = immstate_mixture.weights
        component_conditioned_mode_prob = np.array(
            [c.weights.ravel() for c in immstate_mixture.components]
        )

        # flip conditioning order with Bayes
        mode_prob, mode_conditioned_component_prob = discretebayes.discrete_bayes(weights, c

        # Hint list_a of lists_b to list_b of lists_a: zip(*immstate_mixture.components)
        mode_states = None # TODO:

        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"""

        # ! You can assume all the modes have the same reduce and estimate function
        # ! and use eg. self.filters[0] functionality
        data_reduced = [gaussian_mixture_moments(immstate.weights, comp.mean, comp.cov) for

        estimate = self.filters[0].estimate(data_reduced)
        return estimate
```

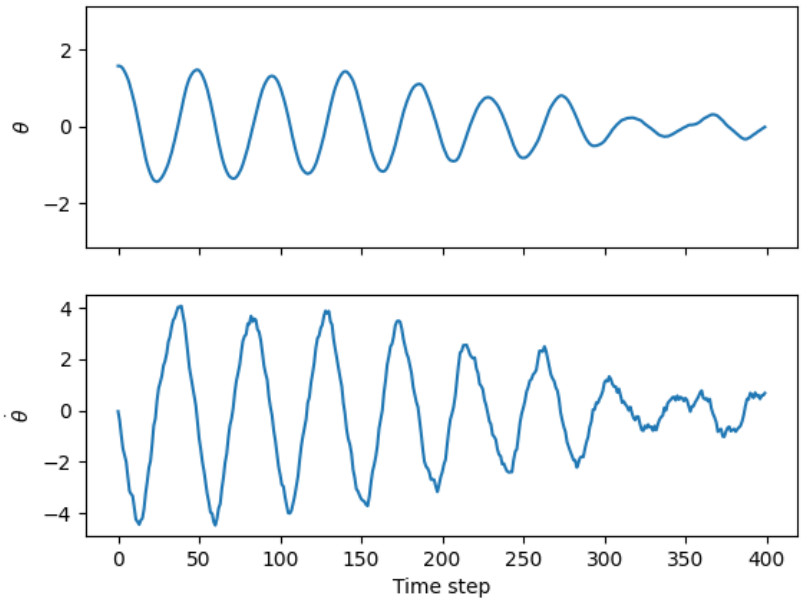
Task 5

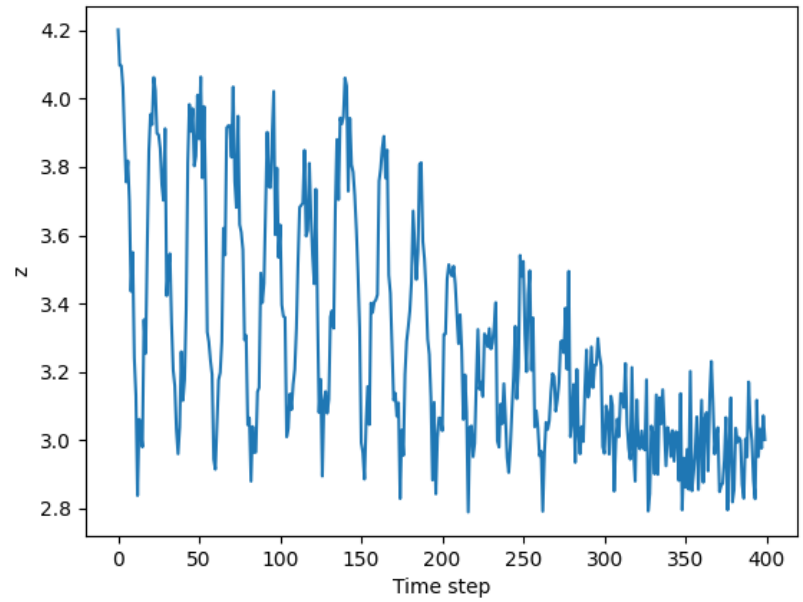
A) Finished code

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

    # resample
    # TODO: some pre calculations?
    # N_eff = 1 / np.sum(w**2)
    # if N_eff <= N/2:
    # resample using algorithm 3 p. 90
    cumweights = np.cumsum(w)
```

```
indicesout = np.zeros((N, 1))
noise = rng.random((1,1)) / N

i = 0
for n in range(N):
    # find a particle 'i' to pick
    # algorithm in the book, but there are other options as well
    u_n = n/N + noise
    while u_n > cumweights[i]:
        i += 1
    pxn[n] = px[i]
# indicesout = indicesout[np.randperm(np.size(indicesout))]
rng.shuffle(pxn,axis=0)

w.fill(1/N)
# else:
#     pxn = px[:]

N_eff = 1 / np.sum(w**2)

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

This worked decently enough with 100 filter, but became very noisy towards the end, when the pendulum slowed


θ
2
0
−2
θ̇
4
2
0
−2
−4
0
50
100
150
200
250
300
350
400
Time step


z
4.2
4.0
3.8
3.6
3.4
3.2
3.0
2.8
0
50
100
150
200
250
300
350
400
Time step

B)

Larger L_l seems to give better results, which makes some sense, based on intuition from the bearings-only example in the lecures.

C)

PF seems like a good alternative to EKFs when there is a lot of nonlinearity in process models and measurement models