Sensor 4

1b) It seems logical to merge models that have close μs and low weights (as they are less important).

i) 0 and 1, close μ

ii) 0 and 2 have lowest weights, but graph suggest that again 0 and 1 is best.

Regarding variance, it seems logical to merge pdfs with same covariance.

iii) 1,2

iv) Could be both 1,2 and 0,1

Either one prioritizes same mean or same cov, the graph suggests to prioritize cov., so 1,2

2a) Derive Meas. likelihood:

$$p(z_k \mid z_{1:k-1})$$

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

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

$p(x_k | s_k; z_{1:k-1})$?

$= \mathcal{N}(x_k, \mu^{\{s_k\}}; P^{\{s_k\}})$

$p(s_k | z_{1:k-1})$ does not depend on x_k can be pulled out of integral:

$\hookrightarrow = \sum p_r\{s_k | z_{1:k-1}\} \cdot \int p(z_k | x_k) \cdot p(x_k | s_k, z_{1:k-1}) \, dx_k$

$\Lambda^{(s_k)}$ (6.32)

$= \sum p_r\{s_k | z_{1:k-1}\} \cdot \Lambda\{s_k\}$

b) $p(z_k | z_{1:k-1}) = \sum p_r\{s_k | z_{1:k-1}\} \int p(z_k | x_k) \, dx$

$p(x_k | s_k, z_{1:k-1}) = \frac{p(x_k | z_{1:k-1}) \cdot p(s_k | z_{1:k-1}, x_k)}{p_r\{s_k | z_{1:k-1}\}}$

$$p(z_k | z_{1:k-1}) =$$

$$\int p(z_k | x_k) \cdot p(x_k | z_{k-1}) \, dx_k$$

$$= \sum \int w_k^i \delta(x_k - x_k^i) \cdot p(z_k | x_k) \, dx_k$$

$$= \sum_{i=1}^{M} w_k^i \cdot p(z_k | x_k^i) \quad ?$$

3a) $$p(x|y) = \frac{p(x_i \cap y)}{p(y)}$$

prior → conditional

$$= \frac{p(x_i) \cdot p(y_j | x_i)}{\sum_{i=1} p(x_i) \cdot p(y | x_i)}$$

$$= \frac{Pr\{s_k | s_{k-1}, z_{1:k-1}\} \cdot Pr\{s_{k-1} | z_{1:k-1}\}}{Pr\{s_k | z_{1:k-1}\}}$$

prior

$$pr = \begin{bmatrix} pr\{s_0\} \\ pr\{s_1\} \\ \vdots \\ p\{s_k\} \end{bmatrix}$$

$$cond = \begin{bmatrix} pr\{s_0|s_0\} & pr\{s_0|s_1\} & \cdots & pr\{s_k|s_k\} \\ & & & \\ & & & \\ & & & pr\{s_k|s_k\} \end{bmatrix}$$

$$cond = \begin{bmatrix} pr\{s_k=s_0|s_{k-1}=s_0\} & \cdots & pr\{s_k=s_0|s_{k-1}=s_n\} \\ \vdots & & \\ pr\{s_k=s_n|s_{k-1}=s_n\} & \cdots & pr\{s_k=s_n|s_{k-1}=s_n\} \end{bmatrix}$$

Mult ușoară element cu element.

$$pr = \begin{bmatrix} pr\{s_{k-1}=s_0\} \\ \vdots \\ pr\{s_{k-1}=s_1\} \end{bmatrix}$$

For marginal we want

$$\sum_{j=1}^{m} pr\{s_k = s_i \mid s_{k-1} = s_j\} \cdot pr\{s_{k-1} = s_j\}$$

All elements in one column of joint

so:

np.sum(joint, axis=1)

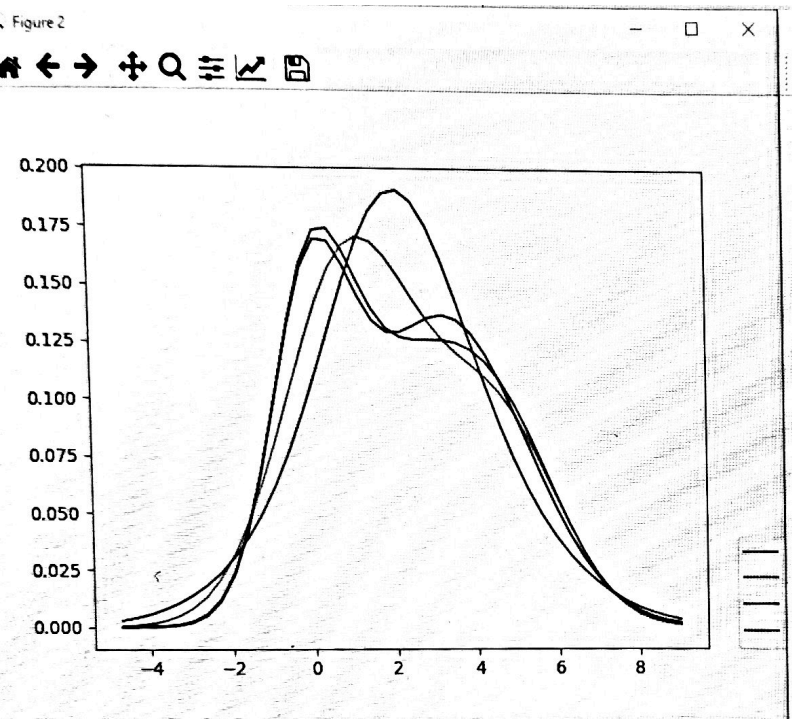
Figure 2
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
−4
−2
0
2
4
6
8

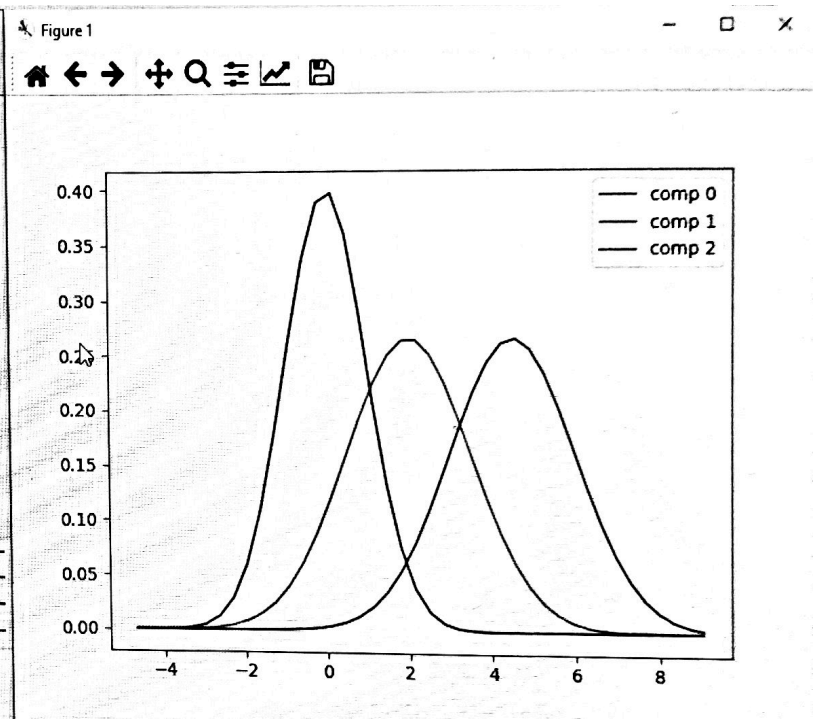
Figure 1
comp 0
comp 1
comp 2
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−4
−2
0
2
4
6
8

```python
# %% imports
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats

rng = np.random.default_rng()
# %% trajectory generation
# scenario parameters
x0 = np.array([np.pi / 2, -np.pi / 100])
Ts = 0.05
K = round(20 / Ts)

# constants
g = 9.81
l = 1
a = g / l
d = 0.5  # dampening
S = 5

# disturbance PDF
process_noise_sampler = lambda: rng.uniform(-S, S)

# dynamic function
def modulo2pi(x, idx=0):
    xmod = x
    xmod[idx] = (xmod[idx] + np.pi) % (2 * np.pi) - np.pi  # wrap to [-pi, pi]
    return xmod


def pendulum_dynamics(x, a, d=0):  # continuous dynamics
    xdot = np.array([x[1], -d * x[1] - a * np.sin(x[0])])
    return xdot


def pendulum_dynamics_discrete(xk, vk, Ts, a, d=0):
    xkp1 = modulo2pi(xk + Ts * pendulum_dynamics(xk, a, d))  # euler discretize
    xkp1[1] += Ts * vk  #  zero orde hold noise
    return xkp1


# sample a trajectory
x = np.zeros((K, 2))
x[0] = x0
for k in range(K - 1):
    v = process_noise_sampler()
    x[k + 1] = pendulum_dynamics_discrete(x[k], v, Ts, a, d)


# vizualize
fig1, axs1 = plt.subplots(2, sharex=True, num=1, clear=True)
axs1[0].plot(x[:, 0])
axs1[0].set_ylabel(r"$\theta$")
axs1[0].set_ylim((-np.pi, np.pi))

axs1[1].plot(x[:, 1])
axs1[1].set_xlabel("Time step")
axs1[1].set_ylabel(r"$\dot \theta$")
```

```
# %% measurement generation

# constants
Ld = 4
Ll = 5
r = 0.25

# noise pdf
measurement_noise_sampler = lambda: rng.triangular(-r, 0, r)

# measurement function
def h(x, Ld, l, Ll):  # measurement function
    lcth = l * np.cos(x[0])
    lsth = l * np.sin(x[0])
    z = np.sqrt((Ld - lcth) ** 2 + (lsth - Ll) ** 2)  # 2norm
    return z


Z = np.zeros(K)
for k, xk in enumerate(x):
    wk = measurement_noise_sampler()
    Z[k] = h(x[k], Ld, l, Ll) + wk


# vizualize
fig2, ax2 = plt.subplots(num=2, clear=True)
ax2.plot(Z)
ax2.set_xlabel("Time step")
ax2.set_ylabel("z")

# %% Task: Estimate using a particle filter

# number of particles to use
N = 1000 #Done


# initialize particles, pretend you do not know where the pendulum starts
px = np.array([
    rng.uniform(-np.pi, np.pi, N), # Done, this represents theta, an angle between -pi and pi. Can
    rng.normal(size=N)*np.pi/3 # Done this is theta dot, should probably start around zero so norma
    ]).T

# initial weights
w = np.full((N,1), 1/N) #weights must be in sum 1.

# allocate some space for resampling particles
pxn = np.zeros_like(px)

# PF transition PDF: SIR proposal, or something you would like to test
PF_dynamic_distribution = scipy.stats.uniform(loc=-S, scale=2 * S)
PF_measurement_distribution = scipy.stats.triang(c=0.5, loc=-r, scale=2 * r)

# initialize a figure for particle animation.
plt.ion()
fig4, ax4 = plt.subplots(num=4, clear=True)
plotpause = 0.01
```

```
sch_particles = ax4.scatter(np.nan, np.nan, marker=".", c="b", label=r"$\hat \theta^n$")
sch_true = ax4.scatter(np.nan, np.nan, c="r", marker="x", label=r"$\theta$")
ax4.set_ylim((-1.5 * l, 1.5 * l))
ax4.set_xlim((-1.5 * l, 1.5 * l))
ax4.set_xlabel("x")
ax4.set_ylabel("y")
th = ax4.set_title(f"theta mapped to x-y")
ax4.legend()

eps = np.finfo(float).eps
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[n]

    sum_weights = np.sum(w)
    w = np.array([weight/sum_weights for weight in w])

    # resample
    # TODO: some pre calculations?
    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
        uj = (n)/N +noise
        while  i != len(cumweights) and uj>cumweights[i]:
            i +=1
        # algorithm in the book, but there are other options as well
        pxn[n] = px[i]
    rng.shuffle(pxn, axis=0)  # shuffling because recommended

    #Recommended to replace weights with 1/N
    w.fill(1.0/N)

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

    # plot
    sch_particles.set_offsets(np.c_[l * np.sin(pxn[:, 0]), -l * np.cos(pxn[:, 0])])
    sch_true.set_offsets(np.c_[l * np.sin(x[k, 0]), -l * np.cos(x[k, 0])])

    fig4.canvas.draw_idle()
    plt.show(block=False)
    plt.waitforbuttonpress(plotpause)

#plt.waitforbuttonpress()
# %%
```

```
"""

"""
# %% Imports

# types
from typing import (
    Tuple,
    List,
    TypeVar,
    Optional,
    Dict,
    Any,
    Union,
    Sequence,
    Generic,
    Iterable,
)
from mixturedata import MixtureParameters
from gaussparams import GaussParams
#from estimatorduck import StateEstimator
from mixturereduction import gaussian_mixture_moments #done
# packages
from dataclasses import dataclass
from singledispatchmethod import singledispatchmethod
import numpy as np
from scipy import linalg
from scipy.special import logsumexp
from ekf import EKF as StateEstimator

# local
import discretebayes

# %% TypeVar and aliases
MT = TypeVar("MT")  # a type variable to be the mode type

# %% IMM
@dataclass
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."""
        # My comment: this step should implement step 1., 6.27

        predicted_mode_probabilities, mix_probabilities = discretebayes.discrete_bayes(immstate.wei
        # 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

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

        #My comment: Here we are implementing the step 2 of the algorithm, 6.29 and 6.30

        means = np.array([component.mean for component in immstate.components])
        covs = np.array([component.cov for component in immstate.components])

        mixed_states = gaussian_mixture_moments(mix_probabilities, immstate.components[:].mean, imm

        mixed_states = []
        for i in range(len(means)):
            mixed_states.append(gaussian_mixture_moments(mix_probabilities[i], means[i], covs[i]))

        mixed_states = np.array(mixed_states)

        return mixed_states

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

        #My comment, here we are doing step 3, mode mathed prediction
        modestates_pred = []
        for i in range(len(self.filters)):
            modestates_pred.append(self.filters[i].predict(mode_states[i], Ts))
```

```
        return np.array(modestates_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 each mode.
        """

        predicted_mode_probability, mixing_probability = self.mix_probabilities(immstate, Ts) #Done

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

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

        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."""

        #MY comment: This implements step 3 update
        updated_state = []
        for filt, mode_state in zip(self.filters, immstate.components):
            updated_state.append(filt.update(z, mode_state, sensor_state))
        return np.arrray(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"""
        # This function will update the mode probabilities pk(sk). Given in equation 6.33

        #mode_loglikelihood =  (z-h(immstate[:].mean))@np.inv(immstate[:].cov)@(z-h(immstate[:].mec
        mode_loglikelihood = np.array([ekf_filter.loglikelihood(z, comp, sensor_state) for ekf_filt
        # potential intermediate step logjoint =

        predicted_mode_probabilities = self.PI*immstate.weights #Done

        denominator = np.sum(np.exp(mode_loglikelihood)*predicted_mode_probabilities) #done
        log_pred_mode_probs = np.log(predicted_mode_probabilities)
```

```
    log_denominator = np.log(denominator)
    log_updated_mode_probs = mode_loglikelihood + log_pred_mode_probs - log_denominator


    updated_mode_probabilities = np.exp(log_updated_mode_probs)
    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.updated_mode_probabilities(z, immstate, sensor_state) #Done
    updated_states =  self.mode_matched_update(z, immstate, sensor_state)

    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_state"""

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

    return updated_immstate

def loglikelihood(
    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
    #components, is this p(X|SK), BUT WE WANT P(SK|X). Is mode_prob p(x)
    mode_prob, mode_conditioned_component_prob = discretebayes.discrete_bayes(weights, componer

    # 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

    means = []
    covs = []
    for mean, cov in immstate.components:
        means.append(means)
        covs.append(cov)

    means = np.array(means)
    covs = np.array(cov)
    means_reduced, covs_reduced = mixturereduction.GaussianMixtureMoments(immstate.weights, mea

    estimate = GaussParams(means_reduced, covs_reduced)
    return estimate

def gate(
    self,
    z: np.ndarray,
    immstate: MixtureParameters[MT],
    gate_size: float,
    sensor_state: Dict[str, Any] = None,
) -> bool:
    """Check if z is within the gate of any mode in immstate in sensor_state"""

    # THIS IS ONLY NEEDED FOR PDA. You can wait with implementation if you want
    gated_per_mode = None # TODO

    gated = None # TODO
    return gated

def NISes(
    self,
    z: np.ndarray,
    immstate: MixtureParameters[MT],
```

```
    *,
    sensor_state: Optional[Dict[str, Any]] = None,
) -> Tuple[float, np.ndarray]:
    """Calculate NIS per mode and the average"""
    NISes = np.array(
        [
            fs.NIS(z, ms, sensor_state=sensor_state)
            for fs, ms in zip(self.filters, immstate.components)
        ]
    )

    innovs = [
        fs.innovation(z, ms, sensor_state=sensor_state)
        for fs, ms in zip(self.filters, immstate.components)
    ]

    v_ave = np.average([gp.mean for gp in innovs], axis=0, weights=immstate.weights)
    S_ave = np.average([gp.cov for gp in innovs], axis=0, weights=immstate.weights)

    NIS = (v_ave * np.linalg.solve(S_ave, v_ave)).sum()
    return NIS, NISes

def NEESes(
    self,
    immstate: MixtureParameters,
    x_true: np.ndarray,
    *,
    idx: Optional[Sequence[int]] = None,
):
    NEESes = np.array(
        [
            fs.NEES(ms, x_true, idx=idx)
            for fs, ms in zip(self.filters, immstate.components)
        ]
    )
    est = self.estimate(immstate)

    NEES = self.filters[0].NEES(est, x_true, idx=idx)  # HACK?
    return NEES, NEESes

@singledispatchmethod
def init_filter_state(
    self,
    init,  # Union[
    #     MixtureParameters, Dict[str, Any], Tuple[Sequence, Sequence], Sequence
    # ],
) -> MixtureParameters:
    """
    Initialize the imm state to MixtureParameters.

    - If mode probabilities are not found they are initialized from self.initial_mode_probabili
    - If only one mode state is found, it is broadcasted to all modes.

    MixtureParameters: goes unaltered
    dict:
        ["weights", "probs", "probabilities", "mode_probs"]
            in this order can signify mode probabilities
```

```
        ["components", "modes"] signify the modes
    tuple: first element is mode probabilities and second is mode states
    Sequence: assumed to be only the mode states

    mode probabilities: array_like
    components:

    """  # TODO there are cases where MP unaltered can lead to trouble

    raise NotImplementedError(
        f"IMM do not know how to initialize a immstate from: {init}"
    )

@init_filter_state.register
def _(self, init: MixtureParameters[MT]) -> MixtureParameters[MT]:
    return init

@init_filter_state.register(dict)
def _(self, init: dict) -> MixtureParameters[MT]:
    # extract weights
    got_weights = False
    got_components = False
    for key in init:
        if not got_weights and key in [
            "weights",
            "probs",
            "probabilities",
            "mode_probs",
        ]:
            weights = np.asfarray([key])
            got_weights = True
        elif not got_components and key in ["components", "modes"]:
            components = self.init_components(init[key])
            got_components = True

    if not got_weights:
        weights = self.initial_mode_probabilities

    if not got_components:
        components = self.init_components(init)

    assert np.allclose(weights.sum(), 1), "Mode probabilities must sum to 1 for"

    return MixtureParameters(weights, components)

@init_filter_state.register(tuple)
def _(self, init: tuple) -> MixtureParameters[MT]:
    assert isinstance(init[0], Sized) and len(init[0]) == len(
        self.filters
    ), f"To initialize from tuple the first element must be of len(self.filters)={len(self.filt

    weights = np.asfarray(init[0])
    components = self.init_compontents(init[1])
    return MixtureParameters(weights, components)

@init_filter_state.register(Sequence)
def _(self, init: Sequence) -> MixtureParameters[MT]:
```

```
    weights = self.initial_mode_probabilities
    components = self.init_components(init)
    return MixtureParameters(weights, components)

@singledispatchmethod
def init_components(self, init: "Union[Iterable, MT_like]") -> List[MT]:
    """ Make an instance or Iterable of the Mode Parameters into a list of mode parameters"""
    return [fs.init_filter_state(init) for fs in self.filters]

@init_components.register(dict)
def _(self, init: dict):
    return [fs.init_filter_state(init) for fs in self.filters]

@init_components.register(Iterable)
def _(self, init: Iterable) -> List[MT]:
    if isinstance(init[0], (np.ndarray, list)):
        return [
            fs.init_filter_state(init_s) for fs, init_s in zip(self.filters, init)
        ]
    else:
        return [fs.init_filter_state(init) for fs in self.filters]

def estimate_sequence(
    self,
    # A sequence of measurements
    Z: Sequence[np.ndarray],
    # the initial KF state to use for either prediction or update (see start_with_prediction)
    init_immstate: MixtureParameters,
    # Time difference between Z's. If start_with_prediction: also diff before the first Z
    Ts: Union[float, Sequence[float]],
    *,
    # An optional sequence of the sensor states for when Z was recorded
    sensor_state: Optional[Iterable[Optional[Dict[str, Any]]]] = None,
    # sets if Ts should be used for predicting before the first measurement in Z
    start_with_prediction: bool = False,
) -> Tuple[List[MixtureParameters], List[MixtureParameters], List[GaussParams]]:
    """Create estimates for the whole time series of measurements. """

    # sequence length
    K = len(Z)

    # Create and amend the sampling array
    Ts_start_idx = int(not start_with_prediction)
    Ts_arr = np.empty(K)
    Ts_arr[Ts_start_idx:] = Ts
    # Insert a zero time prediction for no prediction equivalence
    if not start_with_prediction:
        Ts_arr[0] = 0

    # Make sure the sensor_state_list actually is a sequence
    sensor_state_seq = sensor_state or [None] * K

    init_immstate = self.init_filter_state(init_immstate)

    immstate_upd = init_immstate

    immstate_pred_list = []
```

```
immstate_upd_list = []
estimates = []

for z_k, Ts_k, ss_k in zip(Z, Ts_arr, sensor_state_seq):
    immstate_pred = self.predict(immstate_upd, Ts_k)
    immstate_upd = self.update(z_k, immstate_pred, sensor_state=ss_k)

    immstate_pred_list.append(immstate_pred)
    immstate_upd_list.append(immstate_upd)
    estimates.append(self.estimate(immstate_upd))

return immstate_pred_list, immstate_upd_list, estimates
```

```python
from typing import Tuple

import numpy as np


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 of the mixture shapes ((n,), (n, n))
    """Calculate the first two moments of a Gaussian mixture"""

    # mean
    mean_bar = np.average(mean, axis =0, weights=w)  # TODO: hint np.average using axis and weights

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

    # # spread of means
    # Optional calc: mean_diff =
    mean_diff = mean-mean_bar

    cov_ext = np.zeros(shape=cov[0,:,:].shape)
    M = np.size(mean, axis=0)
    for i in range(0, M):
        cov_ext += w[i]*(mean_diff[i, ...]).T@(mean_diff[i, ...])
    # # total covariance
    cov_bar = cov_int + cov_ext

    return mean_bar, cov_bar


def test_gaussian_mixture_moments():
    w = np.array([0.5, 0.5])
    mean = np.array([[5.0, 10.0], [3, 4]])
    cov = np.array([np.eye(2), np.eye(2)])

    mean_bar, cov_bar = gaussian_mixture_moments(w, mean, cov)
    print(mean_bar)
    print(cov_bar)

test_gaussian_mixture_moments()
```

```python
from typing import Tuple

import numpy as np


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# Done

    marginal = sum(joint, axis=1)# Done

    # Take care of rare cases of degenerate zero marginal,
    conditional = joint/marginal # TODO

    # 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
```