```python
"""

"""
# %% 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(
```

```python
        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(immsta
        # 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[:].mea

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

        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 
    """

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

    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[
    mode_loglikelihood = np.array([ekf_filter.loglikelihood(z, comp, sensor_state) for ek
    # 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, co
```

```python
        # 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.weight

        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)
        ]
```

```python
        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_prol
        - 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(sel

    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 parameter
    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]
```

```python
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_predict
    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
```