```python
mean = weights.T @ means

# Internal covariance
# I assume that we are talking about \Sigma_{i=1}^{M} w_i P_i
cov_internal = np.average(covs, axis=0, weights=weights)

# External covariance
# Assuming that we are talking about P_tilde
mean_diff = np.array(means - mean[np.newaxis])

# I couldn't quite figure out how to find the cov_external
# cov_external = np.average(mean_diff @ mean_diff.T, axis=0, weights=weights) - mean @ mean.T
cov_external = 0

# Total covariance
cov = cov_internal + cov_external

mean, cov = solution.mixturereduction.mixture_moments(weights, means, covs)
return mean.T, cov
```

"""
Discussion about which gaussians to merge together

Initial (and general) discussion, is that merging the gaussians depend quite a lot about the problem. merging two gaussians might be better in one case, however could be totally infeasable in another case.

We would like to maintain as much probability as possible, and ideally it is better to have a too wide probabilistic model compared to a too narrow, as a too narrow model will cause us to lose too much probability

i)

I would say that it is best to merge 0 and 1

This is due to the fact that it follows the original probability model, such that the probabilistic estimate would be proportional to the actual probability.

By mergin either 0 and 2 or 1 and 2, the combined model would be increasing or decreasing where the original model is decreasing or increasing. This will lead to a likely too large shift in the probability

ii)

This is a bit more tricky. All of the combinations will cover the probability of the original function, at least when one is inside a domain D.

The combination of 0 and 2 is too narrow, which causes the function to lose a lot of probability outside when outside of the expected value.

Combining 0 and 1 gives a proibability model that follows the original with a good margine. It gives a slightly bias to lower values, and thus loses some probability for higher values.

Combining 1 and 2 cause a slight bias towards higher values, however it generates a probability density with a larger variance.

I am a bit unsure which I think is best, however it would either be combining 0 and 1 or 1 and 2. The first is due to taking the probability at around 4.5 into account, and generally following the original probability for higher values reasonable well. The latter due to having a generally higher variance.

However it is also important to take the system into account. If this was my friend(s) I hadn't trusted them to arrive until a year had passed. In that regard, both are valid.

iii)

Combining 1 and 2 would be best. This is due to the combination following the original almost perfectly.

iv)

Similar as in iii)

TTK 4250 Assignment 4 Øystein Solbø

2a) Given that

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

assuming that

$$p(z_k|x_k,s_k) = p(z_k|x_k)$$

From the book we know that

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

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

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

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

$$= \sum_{s_k} \Lambda_{s_k}^{(s_k)}\, Pr(s_k|z_{1:k-1})$$

where one could also use that

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

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

which gives that

$$p(z_k|z_{k-1}) = \sum_{s_k} \Lambda^{(s_k)} \frac{\pi^{s_{k-1}s_k}\, p_{k-1}^{(s_{k-1})}}{\mu_{s_{k-1}|s_k}}$$

even though this was not what the task asked, but nice relationship nonetheless..

b) Assuming that

$$p(x_n \mid z_{1:n-1}) \approx \sum_{i=1}^{N} w_n^{(i)} \delta(x_n - x_n^{(i)})$$

$$p(z_n \mid z_{n-1}) = \int p(z_n \mid x_n)\, p(x_n \mid z_{1:n-1})\, dx_n$$

$$\approx \int p(z_n \mid x_n) \sum_{i=1}^{N} w_n^{(i)} \delta(x_n - x_n^{(i)})\, dx_n$$

$$= \sum_{i=1}^{N} w_n^{(i)} \int p(z_n \mid x_n)\, \delta(x_n - x_n^{(i)})\, dx_n$$

Using the fact that $\int f(x)\, \delta(x)\, dx = f(x)$

$$= \sum_{i=1}^{N} w_n^{(i)}\, p(z_n \mid x_n^{(i)})$$

```python
def init_PF(rng: np.random):
    """
    Initialize particles.

    Args:
        rng: a random number generator

    Returns:
        N (int): number of particles
        px (ndarray): particle states shape=(N, dim(state))
        weights (ndarray): normalized weights. shape = (N,)
    """
    # Number of particles to use
    N = 200

    # Initialize particles
    Ld, Ll, r = get_measurement_parameters()

    px = 4*Ll*np.random.random_sample((N,2)) - (Ll + r)

    # initial weights
    w = np.random.random_sample((N,))
    w /= w.sum()

    # N, px, w = solution.SIR_PF_pendulum.init_PF(rng)
    assert np.isclose(sum(w), 1), "w must be normalized"
    assert len(px) == N and len(w) == N, "Number of particles must be N"

    return N, px, w
```

```python
def weight_update(
            zk: float,
            px: np.ndarray,
            w: np.ndarray,
            h: Callable,
            meas_noise_dist: scipy.stats.distributions.rv_frozen
        ):
    """
    Update the weights.

    Args:
        zk: measurement
        px: particles, shape = (N, dim(state))
        w: weights in, shape = (N, )
        h: heasurement funciton that takes the state as args.
        meas_noise_dist: the measurement distribution (a numpy random vari

    Returns:
        updated_weights: shape = (N,) must be normalized
    """
    w_upd = np.empty_like(w)
    for n, pxn in enumerate(px):
        # print(meas_noise_dist.pdf(zk - h(pxn)))
        w_upd[n] = meas_noise_dist.pdf(zk - h(pxn)) + w[n]
    w_upd = w_upd / np.linalg.norm(w_upd)
    # print(w_upd)

    # w_upd = solution.SIR_PF_pendulum.weight_update(
    #     zk, px, w, h, meas_noise_dist)

    return w_upd
```

```python
def resample(
        px: np.ndarray,
        w: np.ndarray,
        rng: np.random.Generator
    ) -> np.ndarray:
    """
    Resample particles

    Args:
        px: shape = (N, dim(state)), the particles
        w: shape = (N,), particle weights
        rng: random number generator.
            Must be used in order to be able to make predictable b

    Returns:
        pxn: the resampled particles
    """
    N = len(w)
    pxn = np.zeros_like(px)

    total_weight = np.cumsum(w, axis=0)
    noise = rng.normal(loc=1, scale=1)

    i = 0
    for n in range(N):
        u_n = n / N + noise
        while u_n > total_weight[i]:
            i += 1
        pxn[n] = px[i]

    # pxn = solution.SIR_PF_pendulum.resample(px, w, rng)

    return pxn
```

```python
def particle_prediction(
        px: np.ndarray,
        Ts: float,
        f: Callable,
        proc_noise_dist: scipy.stats.distributions.rv_frozen
    ) -> np.ndarray:
    """
    Predict particles some time units ahead sampling the process noise

    Args:
        px: shape = (N. dim(state)), the particles
        Ts: Time step size
        f: process function taking the state, noise realization and ti
        dyn_dist: a distribution that can create process noise realiza

    Returns:
        px_pred: the predicted particles
    """
    px_pred = zeros_like(px)
    for n, pxn in enumerate(px):
        vkn = proc_noise_dist.rvs()
        px_pred[n] = f(pxn, vkn, Ts)

    # px_pred = solution.SIR_PF_pendulum.particle_prediction(
    #     px, Ts, f, proc_noise_dist)

    return px_pred
```

```
# Task 5a)
# The particle filter generates an estimate that looks like it is constantly
# lagging behind the actual state. This was developed using resampling, but
# the estimated (resampled) state looks like it is lagging pi/2 rad behind the
# actual response. Since the system responds so quickly, this is totally
# unacceptable to accheive any form of control, unless you utilize feed-forward
# terms heavily

# I retried using 1000 particles instead of 100, and one can see that the
# response is better, however now I have the problem that the estimated
# angle's phase is switching between being ahead and behing the actual angle.
# At around 200-250 timestamps somewhere, I also noticed that the particle
# estimate starts oscillating around zero and completely disregards the
# measurements/actual angle.

# Could I have possibly done something wrong during the assignment?



# Task 5b)
# Based on a), I was honestly not expecting any change by just modifying L1 to
# 0.5, however it did. The particle filter was far better into following the
# actual measurement, and did not get out of phase as in a)

# Trying to set L1 = 1 makes the particle filter overshoot, and it looks like
# is is slightly out of phase/has a higher frequency compared to the actual
# angle. This implies - assuming that I have not done a stupid error when
# programming - that there is a nice value between 0 and 1 that makes the
# estimate be relatively equal to the actual angle.

# This is likely because of that increasing L1 reduces the effect from
# the noise compared to the measurements, having a too large value makes it
# difficult for the camera to measure the changes in the maximum and minimum
# in both x and y-direction.

# I will not bother finding a better value for L1, as I am actually quite
# pleased with L1 = 0.5
```

```
# Task 5c)
# From the lectures it was mentioned that EKF (or other KF) will struggle when
# the system is no longer gaussian. Either through linearization, or that the
# noise is a complex structure, the assumption that the noise is gaussian will
# likely be incorrect. In reality, most systems will suffer with bias and other
# skewness as well. Even though these could be approximated using ESKF or ESEKF,
# it would complicate the matter, and would likely be better to just use a
# particle filter instead.
```