Lab 7 : Particle Filtering

Jeroen Olieslagers
Richard-John Lin

Center for Data Science

10/25/2022

Incorrect equations implemented

- Compartmentalize your code: turn big equations into sums of smaller equations
- Test your equations for a single iteration outside the class
- Do not copy paste lines, very small typos are inevitable
- If your vectorized equations are not working, turn them into nested for loops (slow but easier to implement)
- `np.linalg.inv` is a rather unstable method. Consider using Cholesky decomposition or using `np.linalg.solve`

Motivation

Assume the following problem :

$$z_{i+1} = f(z_i) + w_{i+1}$$
$$x_i = g(z_i) + v_i.$$

Kalman filtering does not work well under a general distribution.
Even the generalized Kalman filter does not necessary work well if the non linearities are important.
If we do have strong assumptions of a generative model behind, it seems wasteful not to consider this information.
Solution : Approximate the distributions with samples !

When particle filters are useful

- Nonlinear dynamics / measurements
- Multimodality : We want to track multiple objects simultaneously
- Multivariate : We want to track multiple variables
- Continuous : The state space can vary smoothly over time
- Non gaussian noise

Approximation by sampling

A distribution can be approximated by samples from it, when the number of samples is sufficient.

$$
\begin{aligned}
p(x|\mathcal{D}) &= \int p(x|\theta, \mathcal{D}) p(\theta, \mathcal{D}) \, \mathrm{d}\theta \\
&= \mathbb{E}_{p(\theta|\mathcal{D})} \left[p(x|\theta, \mathcal{D}) \right] \\
&\approx \frac{1}{S} \sum_{s=1}^{S} p(x|\theta_s, \mathcal{D}), \qquad \theta_s \sim p(\theta|\mathcal{D}).
\end{aligned}
$$

However, if the sampled distribution is "flat", we would have a very high variance !

Importance sampling

Importance sampling is a variance reduction technique.

$$\int f(x)p(x)\,\mathrm{d}x = \int f(x)\frac{p(x)}{q(x)}q(x)\,\mathrm{d}x$$
$$\approx \frac{1}{S}\sum_{s=1}^{S}\frac{p(x_j)}{q(x_j)}f(x_j),$$

where $x_j \sim q(x)$.
q is in general called an *importance density*, and verifies :

$$p(x) > 0 \implies q(x) > 0.$$

Principle

The approximated distributions are those written during the LDS derivations.

Prediction :

$$p(z_i|x_{1:i-1}) = \int p(z_i|z_{i-1})p(z_{i-1}|x_{1:i-1})\,\mathrm{d}z_{i-1}. \tag{1}$$

Update :

$$p(z_i|x_{1:i}) = \frac{p(x_i|z_i)p(z_i|x_{1:i-1})}{p(x_i|x_{1:i-1})}. \tag{2}$$

Sampling the distributions

- Sampling for the prediction distribution is easy : imagine we have samples from $z_{i-1}|x_{1:i-1}$. To sample from $z_i|x_{1:i-1}$, we just need to forward them through the latent dynamic.
- What about the update ?

$$p(z_i|x_{1:i}) = \frac{p(x_i|z_i)}{p(x_i|x_{1:i-1})} p(z_i|x_{1:i-1}).$$

We have samples from the wrong distribution $z_i|x_{1:i-1}$, but the density is weighted by some factor.
This means that we can resample the particles using those weights.

Expectations

- It is reasonable to take the mean from the samples to estimate the expectation. However, this may not be the most precise, due to the randomness of sampling, and may have a large variance.
- Another way to proceed is to express the quantities of interest with marginalisation such that we can use importance sampling.

Expectations

- **Prediction**:

$$
\begin{aligned}
p(z_i|x_{1:i-1}) &= \int p(z_i, z_{i-1}|x_{1:i-1})\,\mathrm{d}z_{i-1} \\
&= \int p(z_i|z_{i-1}, x_{1:i-1})p(z_{i-1}|x_{1:i-1})\,\mathrm{d}z_{i-1} \\
&= \int p(z_i|z_{i-1})p(z_{i-1}|x_{1:i-1})\,\mathrm{d}z_{i-1} \\
&\approx \frac{1}{N}\sum_{n=1}^{N} p(z_i^{(n)}|x_{1:i})p(z_i|z_{i-1}^{(n)}).
\end{aligned}
$$

Expectations

- **Update**:

$$\mathbb{E}[f(z_i)] = \int f(z_i)p(z_i|x_{1:i})\,\mathrm{d}z_i$$
$$= \int f(z_i)p(z_i|x_{1:i-1}, x_i)\,\mathrm{d}z_i$$
$$= \frac{\int f(z_i)p(x_i|z_i)p(z_i|x_{1:i-1})\,\mathrm{d}z_i}{\int p(x_i|z_i)p(z_i|x_{1:i-1})\,\mathrm{d}x_i\,\mathrm{d}z_i} \approx \sum_{n=1}^{N} w_i^{(n)} f(z_i^{(n)}),$$

where $w_i^{(n)} = \frac{p(x_i|z_i^{(n)})}{\sum_m p(x_i|z_i^{(m)})}$

A generic particle filter

1. Generate a random sample of particles
2. Predict the next state of particles
3. Update the particle based on the measurements : the closer to the measurements, the greater the weight
4. Resample if necessary : discard improbable samples and replace them with more likely ones
5. Compute the estimates, usually with weighted means and covariance

Example : Robot localisation

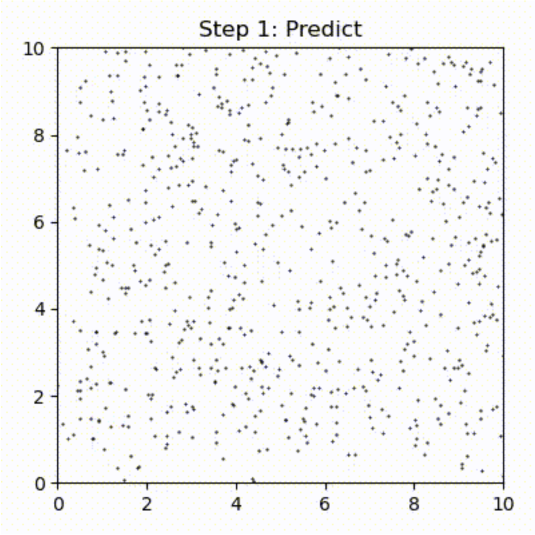


Figure 1: Robot localisation

Example : Robot localisation

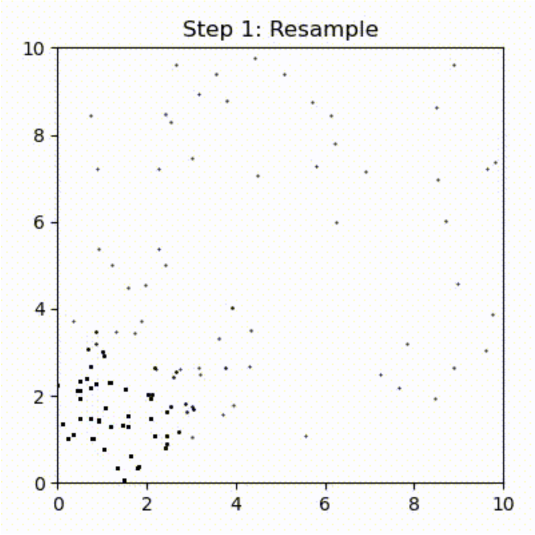


Figure 1: Robot localisation

Example : Robot localisation

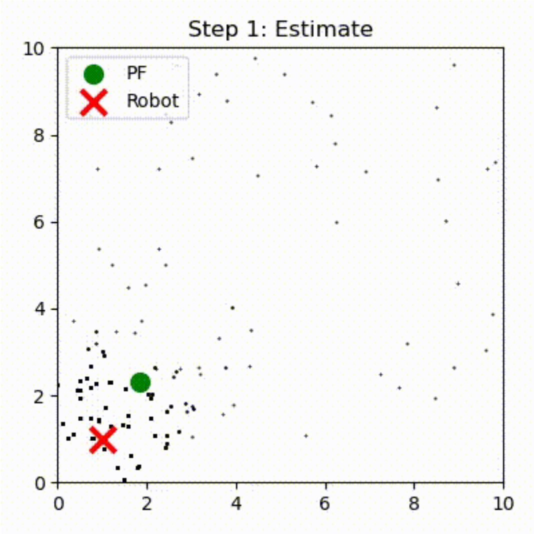


Figure 1: Robot localisation

Example : Robot localisation

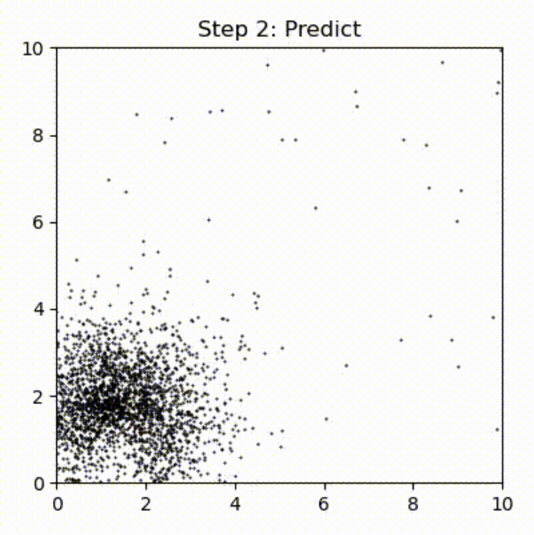


Figure 1: Robot localisation

Example : Robot localisation

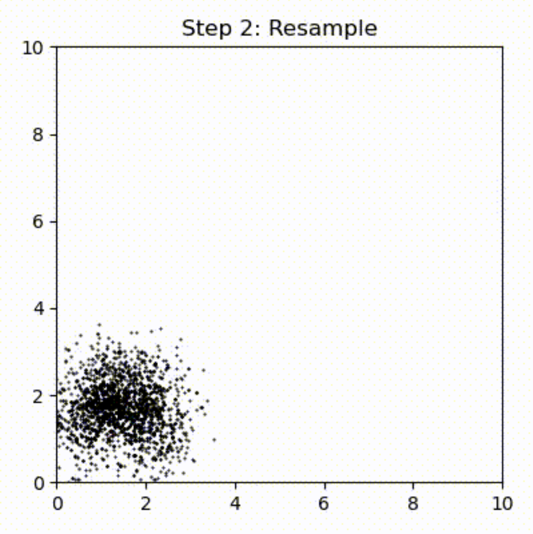


Figure 1: Robot localisation

Example : Robot localisation

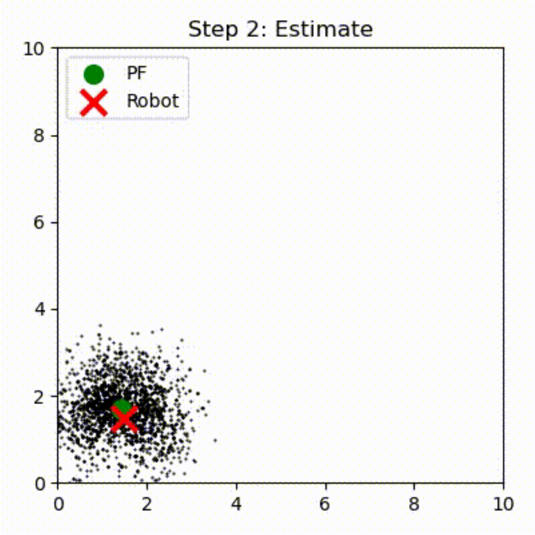


Figure 1: Robot localisation

Example : Robot localisation

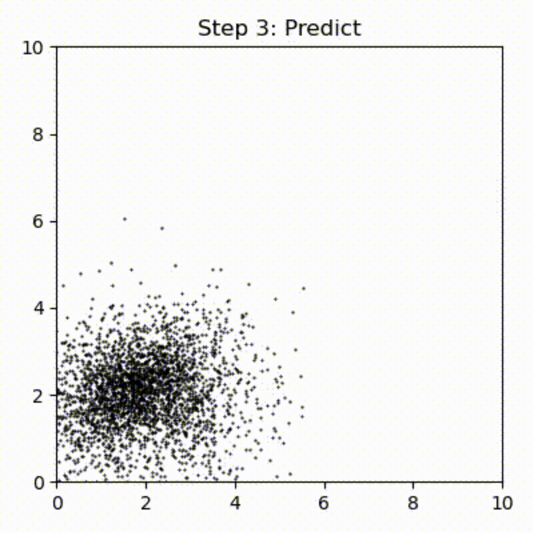


Figure 1: Robot localisation

Example : Robot localisation

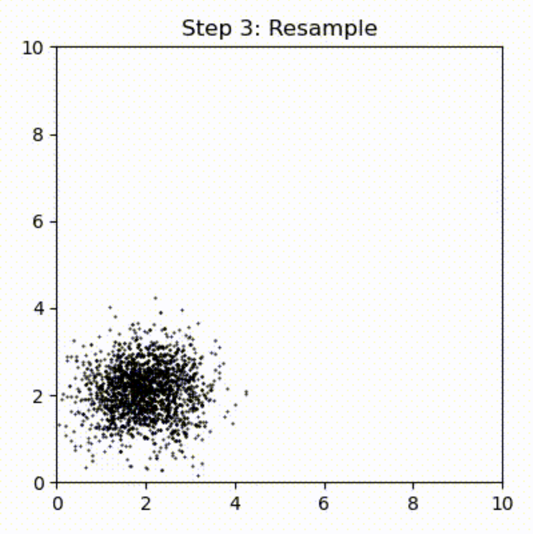


Figure 1: Robot localisation

Example : Robot localisation

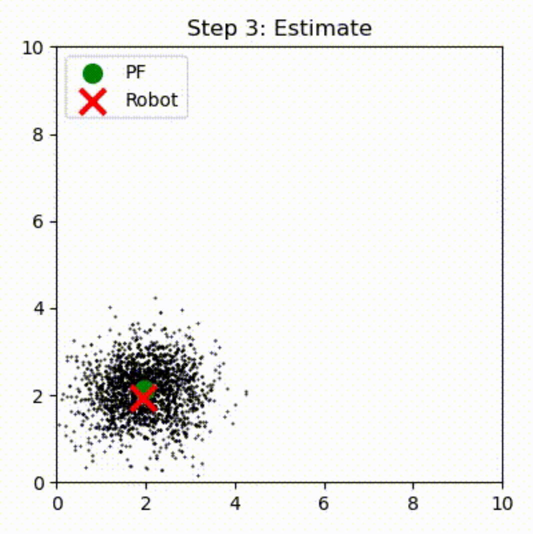


Figure 1: Robot localisation

Example : Robot localisation

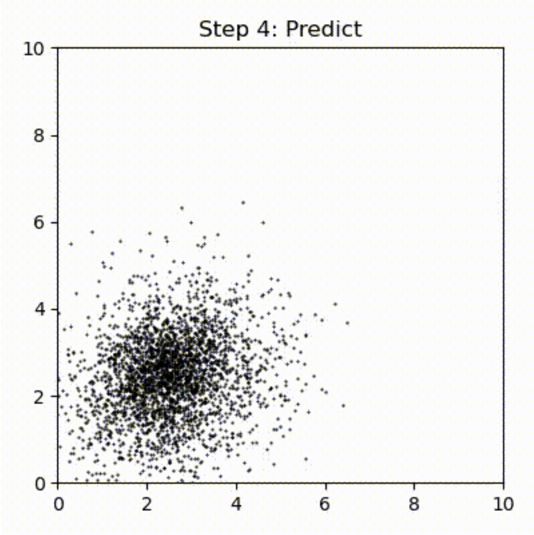


Figure 1: Robot localisation

Example : Robot localisation

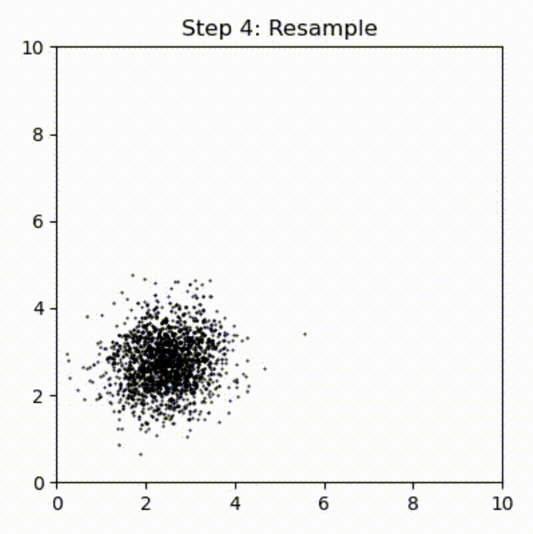


Figure 1: Robot localisation

Example : Robot localisation

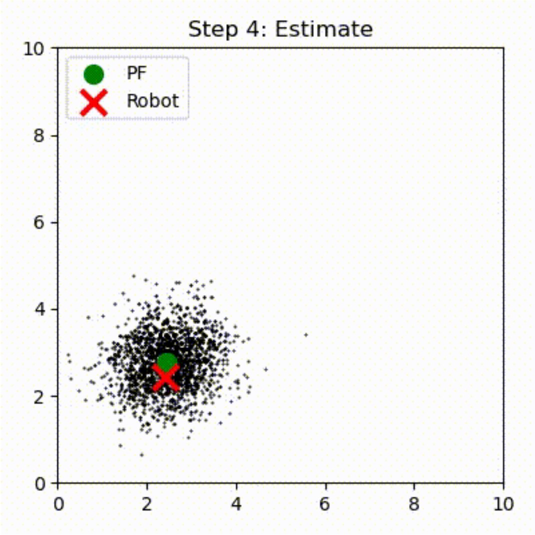


Figure 1: Robot localisation

Example : Robot localisation

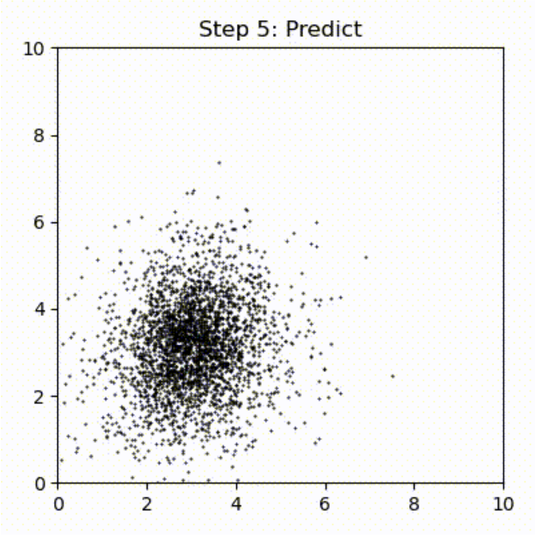


Figure 1: Robot localisation

Example : Robot localisation

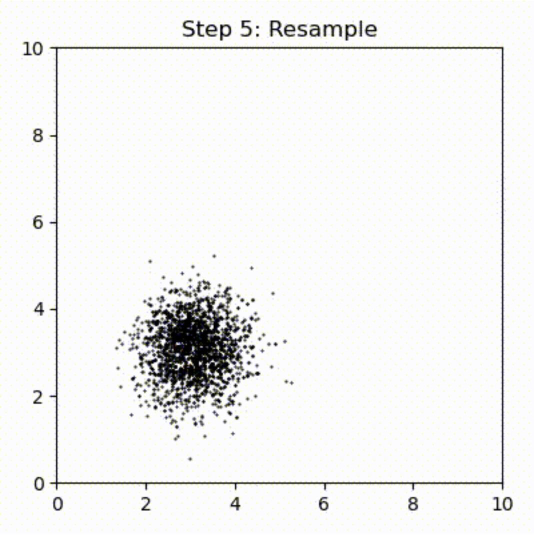


Figure 1: Robot localisation

Example : Robot localisation

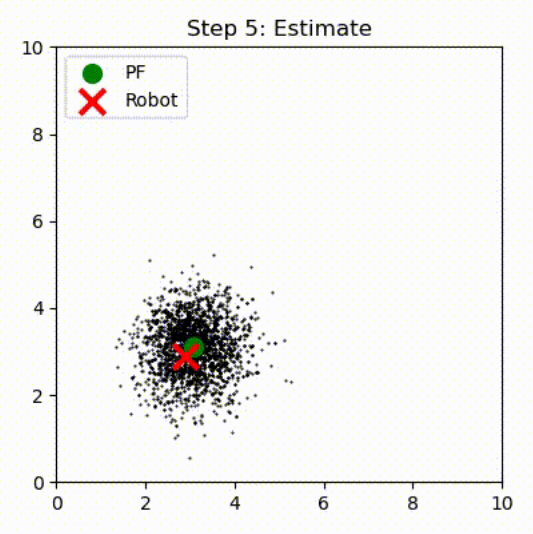


Figure 1: Robot localisation

Caveat : Small sensor noise

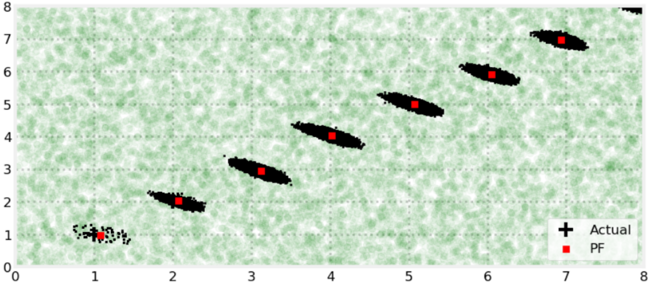


Figure 2: Small sensor noise

A very accurate sensor can lead to poor filter performance, because only a few particles will be representative of the actual distribution. Some possible fixes are :

- Artificially increase the sensor noise
- Increase the number of particles

Caveat : Sample degeneracy due to bad initial conditions

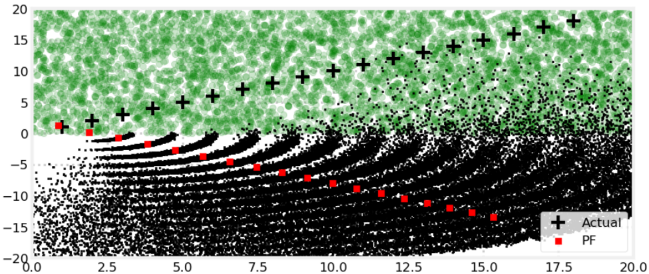


Figure 3: Sample degeneracy

The particle filter ends up resampling points that are not representative of the distribution ! This effect is called *sample impoverishment*

Caveat : Sample degeneracy due to bad initial conditions

It is always a good idea to create particles near the initial position (but not *too* near), if it is can be estimated.

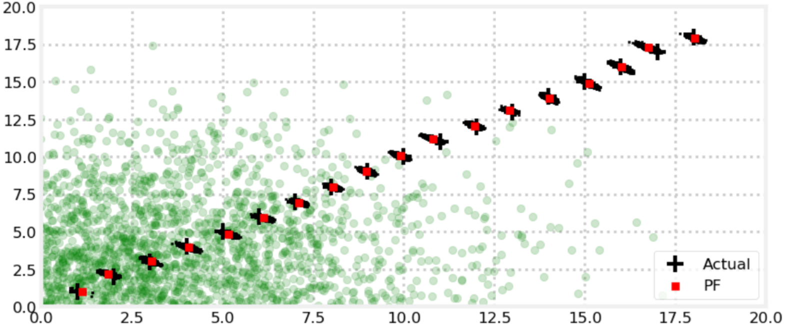


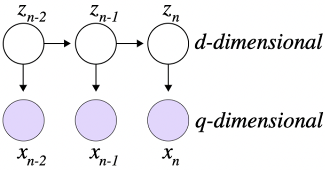
Figure 3: Better initial conditions

Particle filter (PF)

Particle Filtering: alternative inference

We know: data and parameters (A, C, Γ, Σ)

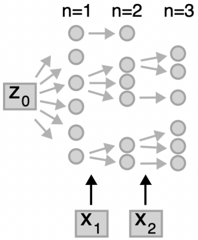
We assume: linear transformation in latent space, linear mapping from latent to observed space, Gaussian observations



We want: approximation of the posterior marginals $P(z_n | x_{1:t})$

Resampling

How: generate samples of $P(z_n^{(i)}|z_{n-1})$ through particle filtering, reweigh by observations, and average to obtain expected value



Initialization

A) initial samples for z_1

1) draw N_{samp} samples (=particles) given initial condition μ_0 and Γ_0

$$P(z_0^{(i)}|\mu_0, \Gamma_0)$$

where $i = 1, \ldots, N_{samp}$

2) propagate samples forward one time step ($n = 1$) through linear transformation A and adding noise with covariance Γ

$$P(z_1^{(i)}|z_0^{(i)})$$

Particle filter algorithm

B) for loop:

1) weigh samples for $z_n^{(i)}$ given observational evidence from x_n

2) compute the probability for the data for each sampled $z_n^{(i)}$:

$$P(x_n|z_n^{(i)})$$

3) compute the weights $w_n^{(i)}$ given $P(x_n|z_n^{(i)})$:

$$w_n^{(i)} = \frac{P(x_n|z_n^{(i)})}{\sum_i P(x_n|z_n^{(i)})}$$

2) produce new samples at $n + 1$

4) draw from multinomial distribution with probabilities w_n, which will give you class assignments $c_{(i)}$ that indicate which samples $z_n^{(i)}$ to use

5) $z_n^{(c_{(i)})}$ become your new priors form which you sample $z_{n+1}^{(i)}$

$$P\left(z_{n+1}^{(i)}|z_n^{(c_{(i)})}, \Gamma\right)$$

6) keep going WITHIN THE LOOP

PF initialization

```python
def particle_filter(self, data, Nsamp, seed=0):
    ###############################################
    # TODO: implementation of the particle filter #
    ###############################################
    np.random.seed(seed)
    # initial conditions:
    self.est_z_mean[0] = self.initial_state_mean.copy()
    self.est_z_var[0] = self.initial_state_covariance.copy()

    # placeholder
    self.z_samp = np.zeros([Nsamp, len(self.time)])
    self.w = np.zeros([Nsamp, len(self.time)])

    ### create samples from distribution with initial conditions
    # TODO: your code here!
    z_samp0 = np.repeat(0, Nsamp)
    ### propagate and create samples at time point n=1
    # TODO: your code here!
    z_samp = np.repeat(0, Nsamp)

    ### save those samples from n=1
    self.z_samp[:,0] = z_samp.copy()
```

PF loop

```
for nn in range(1, len(self.time)):

    ### compute the weights (implement function below)
    w = self.compute_w(data[nn-1, :], z_samp)

    ### keep track of mean and variance of the weighted samples
    # TODO: your code here:
    self.est_z_mean[nn-1] = 0
    self.est_z_var[nn-1] = 1

    ### compute class assignments
    # TODO: your code here:
    k = np.ones(Nsamp)

    ### particles according to class assignments (=reweighted particles)
    # TODO: your code here:
    z_samp_new = np.zeros(Nsamp)

    ### propagate and create samples at time point n+1 (using the reweighted particles)
    # TODO: your code here:
    z_samp = np.zeros(Nsamp) * np.nan

    # save particles and weights
    self.w[:, nn-1] = w
    self.z_samp[:, nn] = z_samp

# track for last sample:
self.est_z_mean[-1] = 0
self.est_z_var[-1] = 1
```

Weighting function

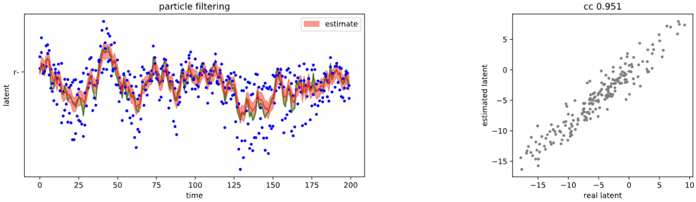
```
def compute_w(self, data_nn, z_samp, seed=0):
    np.random.seed(seed)
    ##############################################
    ####### function to compute weights ##########
    ##############################################
    # TODO: your code here:



    return np.ones(z_samp.shape[0])/z_samp.shape[0]
```

Inference

should look something like this ...



Less particles

should look something like this ...