Lab 6 : Expectation Maximization (EM)

Jeroen Olieslagers
Richard-John Lin

Center for Data Science

10/18/2022

Prediction and variance of the prediction error

- We had an AR model with $p = 1$.
- Given $x_1, \dots x_n$, the best prediction at lag 1 we can do is :

$$\begin{aligned}\hat{x}_{n+1} := \mathbb{E}[x_{n+1}|x_{1:n}] &= \mathbb{E}[\phi x_n + w_{n+1}|x_{1:n}] \\ &= \phi x_n.\end{aligned}$$

More generally, at time $n + k$:

$$\hat{x}_{n+k} := \mathbb{E}[x_{n+k}|x_{1:n}] = \phi^k x_n.$$

Prediction and variance of the prediction error

- We had an AR model with $p = 1$.
- The variance of the error is expressed as :

$$\begin{aligned}\mathbb{V}[\hat{x}_{n+1} - x_{n+1}|x_{1:n}] &= \mathbb{V}[\phi x_n - (\phi x_n + w_{n+1})\,|x_{1:n}] \\ &= \mathbb{V}[w_{n+1}|x_{1:n}] \\ &= \sigma^2.\end{aligned}$$

For step 2 :

$$\begin{aligned}\mathbb{V}[\hat{x}_{n+2} - x_{n+2}|x_{1:n}] &= \mathbb{V}[\phi\,(\hat{x}_{n+1} - x_{n+1}) + w_{n+2}|x_{1:n}] \\ &= \phi^2\mathbb{V}[(\hat{x}_{n+1} - x_{n+1})\,|x_{1:n}] + \mathbb{V}[w_{n+2}|x_{1:n}] \\ &= \phi^2\sigma^2 + \sigma^2.\end{aligned}$$

Prediction and variance of the prediction error

- We had an AR model with $p = 1$.
- Similarly,

$$\begin{aligned}\mathbb{V}[\hat{x}_{n+k} - x_{n+k}|x_{1:n}] &= \sum_{i=0}^{k-1} \left(\phi^2\right)^i \sigma^2 \\ &= \frac{1-\phi^{2k}}{1-\phi^2}\sigma^2.\end{aligned}$$

- Checks :
  The variance is increasing in function of k (if the process is causal).
  For $k = 1$, we find a variance of σ^2, which is consistent with the earlier result.

Principle

When some parameters are unknown, one usual way to estimate them is to maximize the likelihood :

$$\mathcal{L}(\theta) = \log P(x|\theta).$$

From the graphical model, we can write out the joint distribution $P(x, z|\theta)$. With Kalman smoothing, we can compute partial statistics of $P(z|x, \theta)$.
Rewriting using Bayes' rule :

$$\mathcal{L}(\theta) = \log P(x, z|\theta) - \log P(z|x, \theta).$$

Principle

Since we do not know $P(z|x,\theta)$, one way to use known information is to take the expectation under this probability density :

$$\mathcal{L}(\theta) = \int P(z|,x,\theta)\log P(x,z|\theta)\,\mathrm{d}z - \int P(z|,x,\theta)\log P(z|x,\theta)\,\mathrm{d}z.$$

Let $q(z) = P(z|x,\theta)$. Then after maximizing $\mathcal{L}$, we have two objects :

$$\begin{cases} \theta = \arg\max \int q(z)\log P(x,z|\theta)\,\mathrm{d}z \\ q(z) = P(z|x,\theta). \end{cases}$$

EM is a natural iterative algorithm to find a fixed point for this condition.

Example

EM is often used with gaussian mixture models. However, it does not necessarily scale well in high dimensions as it can get stuck in local optima.

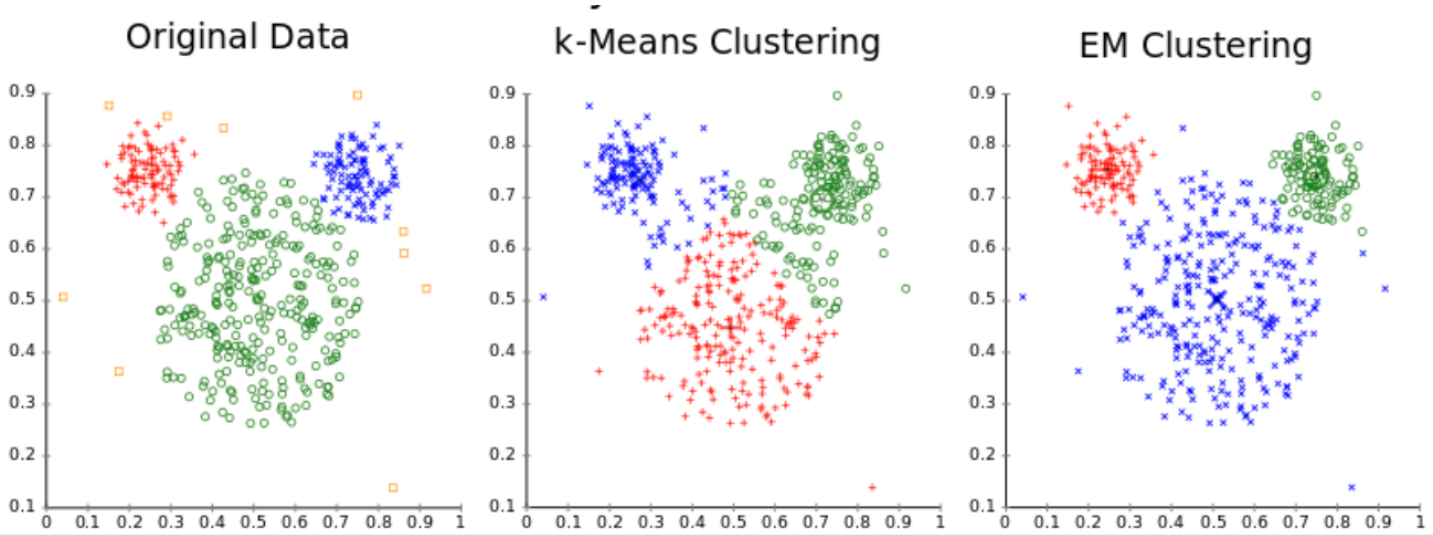


Figure 1: Comparison of EM and K-Means
From link

Application to parameter estimation in Kalman Filtering

Kalman smoothing doesn't tell us how to compute $q(z) = P(z|x, \theta)$, but rather how to compute partial statistics :

- One point marginals :

$$q(z_i) = P(z_i|x, \theta)$$

- Two points marginals for consecutive times

$$q(z_i, z_{i+1}) = P(z_i, z_{i+1}|x, \theta).$$

However, it is exactly the part we need for the calculations in the M step !

Expectation Maximization (EM)

EM algorithm

- want to maximize $logp(x|\theta)$
- need to marginalize out latent *(which is not tractable)*

$$log\left(p(x|\theta)\right) = log\left(\int p(x,z|\theta)dz\right)$$

- add a probability distribution $q(z)$ which will approximate the latent distribution

$$= \int_z q(z)logp(x|\theta)dz$$

- can be rewritten as

$$= \mathcal{L}(q,\theta) + KL\left(q(z)||p(z|x),\theta\right)$$

- $\mathcal{L}(q,\theta)$ contains the joint distribution of x and z
- $KL(q||p)$ contains the conditional distribution of $z|x$

EM steps

Expectation step

- parameters are kept fixed
- find a good approximation $q(z)$: maximize lower bound $\mathcal{L}(q, \theta)$ with respect to $q(z)$
- (already implemented Kalman filter+smoother)

Maximization step

- keep distribution $q(z)$ fixed
- change parameters to maximize the lower bound $\mathcal{L}(q, \theta)$

M step

M-step

(see Bishop, chapter 13.3.2 Learning in LDS)

Update parameters of the probability distribution

Initial parameters

$$\mu_0^{new} = E(z_1)$$
$$\Gamma_0^{new} = E(z_1 z_1^T) - E(z_1)E(z_1^T)$$

Latent parameters

$$A^{new} = \left(\sum_{n=2}^{N} E(z_n z_{n-1}^T)\right)\left(\sum_{n=2}^{N} E(z_{n-1} z_{n-1}^T)\right)^{-1}$$

$$\Gamma^{new} = \frac{1}{N-1}\sum_{n=2}^{N} E(z_n z_n^T) - A^{new} E(z_{n-1} z_n^T) - E(z_n z_{n-1}^T)A^{new} + A^{new} E(z_{n-1} z_{n-1}^T)(A^{new})^T$$

Observable space parameters

$$C^{new} = \left(\sum_{n=1}^{N} x_n E(z_n^T)\right)\left(\sum_{n=1}^{N} E(z_n z_n^T)\right)^{-1}$$

$$\Sigma^{new} = \frac{1}{N}\sum_{n=1}^{N} x_n x_n^T - C^{new} E(z_n) x_n^T - x_n E(z_n^T) C^{new} + C^{new} E(z_n z_n^T) C_{new}$$

For the updates in the M-step we will need the following posterior marginals obtained from the Kalman smoothing results $\hat{\mu}_n, \hat{V}_n$

$$E(z_n) = \hat{\mu}_n$$
$$E(z_n z_{n-1}^T) = J_{n-1}\hat{V}_n + \hat{\mu}_n \hat{\mu}_{n-1}^T$$
$$E(z_n z_n^T) = \hat{V}_n + \hat{\mu}_n \hat{\mu}_n^T$$

Kalman filter class

```
class MyKalmanFilter:
    """
    Class that implements the Kalman Filter
    """
    def __init__(self, n_dim_state=2, n_dim_obs=2):
        """
        @param n_dim_state: dimension of the laten variables
        @param n_dim_obs: dimension of the observed variables
        """
        self.n_dim_state = n_dim_state
        self.n_dim_obs = n_dim_obs
        self.transition_matrices = np.eye(n_dim_state)
        self.transition_covariance = np.eye(n_dim_state)
        self.observation_matrices = np.eye(n_dim_obs, n_dim_state)
        self.observation_covariance = np.eye(n_dim_obs)
        self.initial_state_mean = np.zeros(n_dim_state)
        self.initial_state_covariance = np.eye(n_dim_state)
```

Sampling

```python
def sample(self, n_timesteps, initial_state=None, random_seed=None):
    """
    Method that gives samples from the LDS
    @param initial_state: numpy array whose length == self.n_dim_state
    @param random_seed: an integer, for test purpose
    @output state: a 2d numpy array with dimension [n_timesteps, self.n_dim_state]
    @output observation: a 2d numpy array with dimension [n_timesteps, self.n_dim_obs]
    """

    latent_state = np.zeros([n_timesteps, self.n_dim_state])
    observed_state = np.zeros([n_timesteps, self.n_dim_obs])

    if random_seed is not None:
        np.random.seed(random_seed)

    ################
    ##### TODO #####
    ################


    return latent_state, observed_state
```

Filtering

```python
def filter(self, X):
    """
    Method that performs Kalman filtering
    @param X: a numpy 2D array whose dimension is [n_example, self.n_dim_obs]
    @output: filtered_state_means: a numpy 2D array whose dimension is [n_example, self.n_dim_state]
    @output: filtered_state_covariances: a numpy 3D array whose dimension is [n_example, self.n_dim_state,
    """

    # validate inputs
    n_example, observed_dim = X.shape
    assert observed_dim==self.n_dim_obs

    # create holders for outputs
    filtered_state_means = np.zeros( [n_example, self.n_dim_state] )
    filtered_state_covariances = np.zeros( [n_example, self.n_dim_state, self.n_dim_state] )

    ##############################################################################
    ################## insert your own filter here ###############################
    ##############################################################################
    # below: this is an alternative if you do not have an implementation of filtering
    kf = KalmanFilter(n_dim_state=self.n_dim_state, n_dim_obs=self.n_dim_obs)
    need_params = ['transition_matrices', 'observation_matrices', 'transition_covariance',
      'observation_covariance', 'initial_state_mean', 'initial_state_covariance']
    for param in need_params:
        setattr(kf, param, getattr(self, param))
    filtered_state_means, filtered_state_covariances = kf.filter(X)
    ##############################################################################

    return filtered_state_means, filtered_state_covariances
```

EM

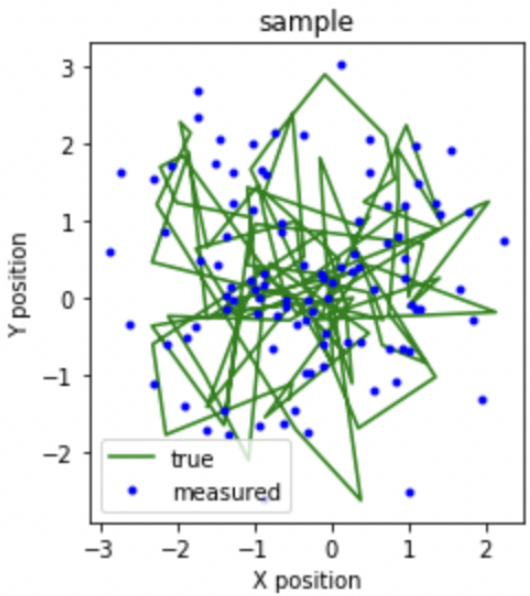
```python
def em(self, X, max_iter=10):
    """
    Method that perform the EM algorithm to update the model parameters
    Note that in this exercise we ignore offsets
    @param X: a numpy 2D array whose dimension is [n_example, self.n_dim_obs]
    @param max_iter: an integer indicating how many iterations to run
    """
    # validate inputs have right dimensions
    n_example, observed_dim = X.shape
    assert observed_dim==self.n_dim_obs

    # keep track of log posterior (use function calculate_posterior below)
    self.avg_em_log_posterior = np.zeros(max_iter)*np.nan

    ##############################
    #### TODO: EM iterations ####
    ##############################
```

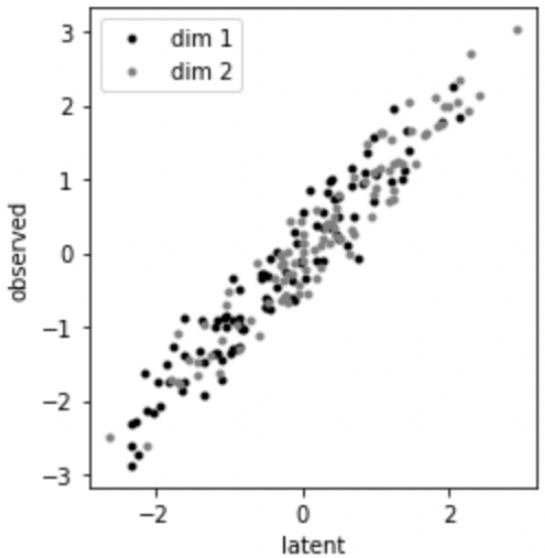
Testing sampling

test that your sampling works:

```
sampled_states, sampled_observations = kf.sample(100, initial_state=kf.initial_state_mean,
sampled_states_impl, sampled_observations_impl = my_kf.sample(100, initial_state=kf.initia
print('sampled states pykalman at t=2: ', sampled_states[2,:])
print('sampled states own implementation at t=2: ', sampled_states_impl[2,:])
fig = plot_kalman(sampled_states_impl[:,0],sampled_states_impl[:,1],sampled_observations_i
```

```
sampled states pykalman at t=2:  [1.43945741 0.96908939]
sampled states own implementation at t=2:  [1.43945741 0.96908939]
```



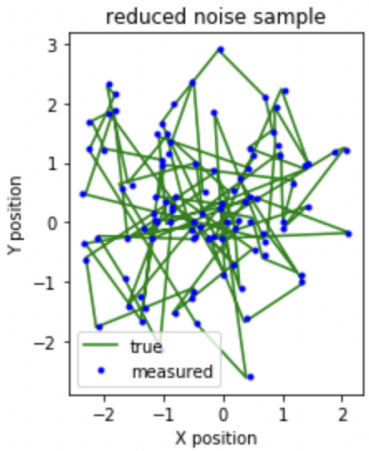


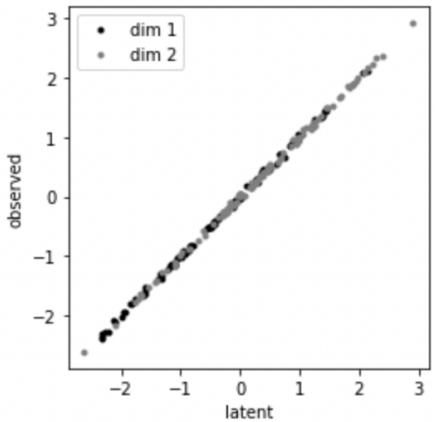
Reducing observation variance

reduce observation noise

What do you expect should happen?

```
#### reduce observation noise ####
obscov_old = my_kf.observation_covariance.copy()
my_kf.observation_covariance = my_kf.observation_covariance*.01

# plot
for nn in range(3):
    sampled_states_impl, sampled_observations_impl = my_kf.sample(100, initial_state=kf.initial_state_mean, random_seed
    fig = plot_kalman(sampled_states_impl[:,0],sampled_states_impl[:,1],sampled_observations_impl[:,0],sampled_observat
    plt.axis('square');
```



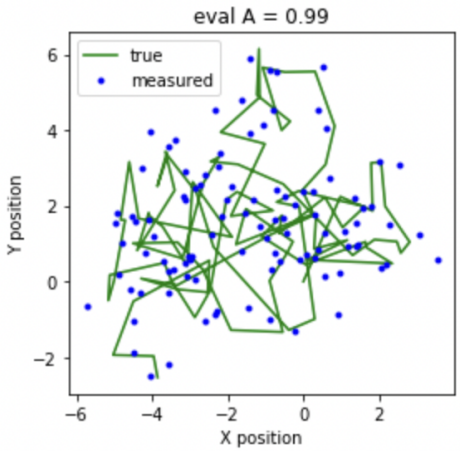


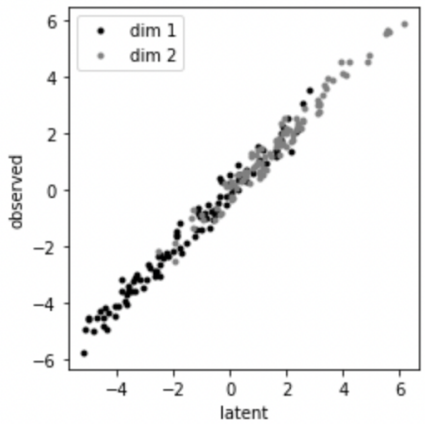
Increasing temporal gain

increase the respective temporal dyamics

What do you expect should happen?

```
#### increase latent temporal dependency ####
my_kf.observation_covariance = obscov_old.copy()
my_kf.transition_matrices = np.eye(n_dim_state)*.9

# plot
for nn in range(3):
    sampled_states_impl, sampled_observations_impl = my_kf.sample(100, initial_state=kf.initial_state_mean, random_seed
    fig = plot_kalman(sampled_states_impl[:,0],sampled_states_impl[:,1],
                      sampled_observations_impl[:,0],sampled_observations_impl[:,1], title='eval A = '+np.str(.99));
```





Initializing the model and getting samples

EM

data to use

```
kf_GT = KalmanFilter(n_dim_state=n_dim_state, n_dim_obs=n_dim_obs)
# set paramters
kf_GT.transition_matrices = np.eye(n_dim_state)*.9
kf_GT.transition_covariance = np.eye(n_dim_obs)
kf_GT.observation_matrices = np.eye(n_dim_state)
kf_GT.observation_covariance = np.eye(n_dim_obs)
kf_GT.initial_state_mean = np.zeros(n_dim_state)
kf_GT.initial_state_covariance = np.eye(n_dim_state)*.1
# import to your own kalman object
my_kf_GT = MyKalmanFilter(n_dim_state=n_dim_state, n_dim_obs=n_dim_obs)
my_kf_GT.import_param(kf_GT)
# print the parameters
print_parameters(my_kf_GT, evals=True)


# sample
latent, data = kf_GT.sample(1000, initial_state=kf_GT.initial_state_mean, random_state=np.random.RandomState(2))
_, _ = kf_GT.filter(data)
estlat,_ = kf_GT.smooth(data)
fig = plot_kalman(latent[:,0],latent[:,1],data[:,0],data[:,1], title='sample for EM');
```

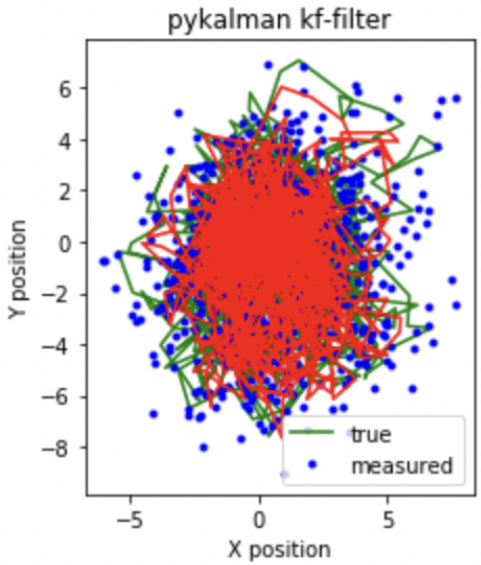
Running EM

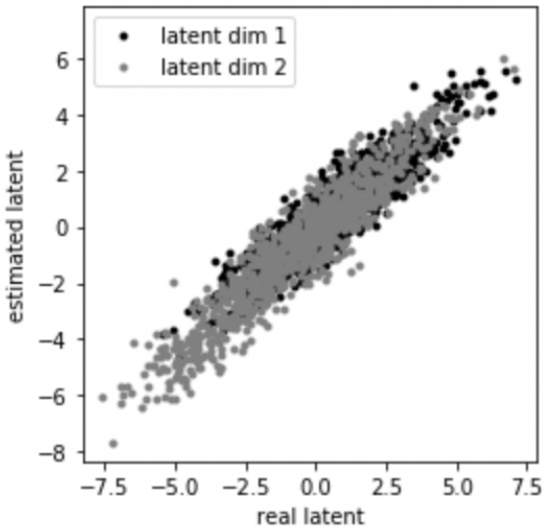
run EM

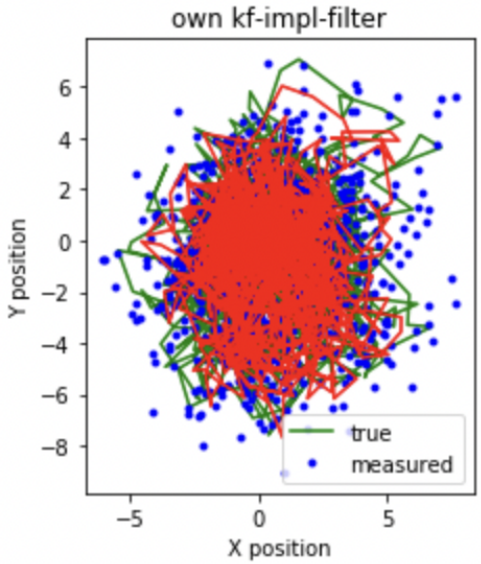
to learn parameters (M-step)

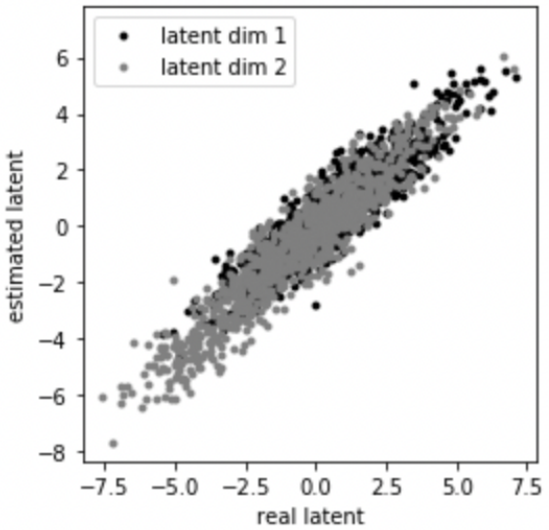
```
np.random.seed(0)

iters = 10
# perturb starting parameters
kf = KalmanFilter(n_dim_state=data.shape[1], n_dim_obs=data.shape[1],
                  transition_matrices= np.eye(data.shape[1]) * 0.95,
                  observation_matrices= np.eye(data.shape[1])+np.random.randn(data.shape[1])*.1,
                  transition_covariance= np.eye(data.shape[1]),
                  observation_covariance = np.eye(data.shape[1]),
                  initial_state_mean=np.random.randn(data.shape[1]),
                   initial_state_covariance = np.eye(data.shape[1]),
                  em_vars = ['transition_matrices', 'observation_matrices','transition_covariance','observation_covarian
                          'initial_state_mean', 'initial_state_covariance'])

my_kf = MyKalmanFilter(n_dim_state=data.shape[1], n_dim_obs=data.shape[1])
my_kf.import_param(kf)

kf.em(data, n_iter=iters)
my_kf.em(data, max_iter=iters)

print('           pykalman EM:')
print(' ')
print_parameters(kf, evals=True)
print('           own implementation EM:')
print(' ')
print_parameters(my_kf, evals=True)
```

Comparing filtering

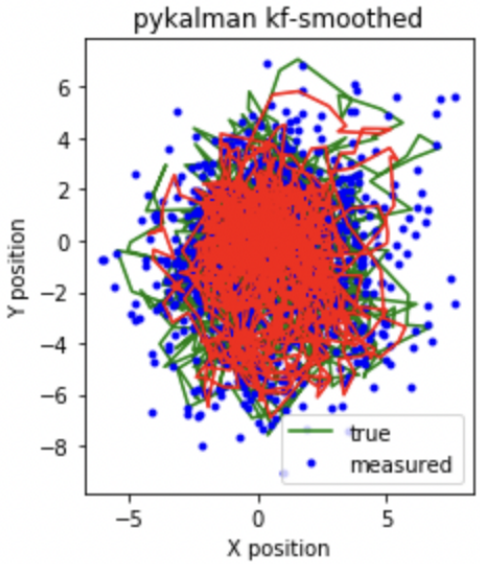


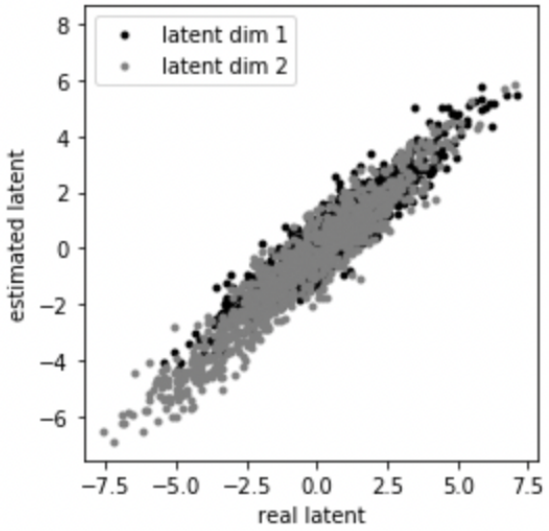


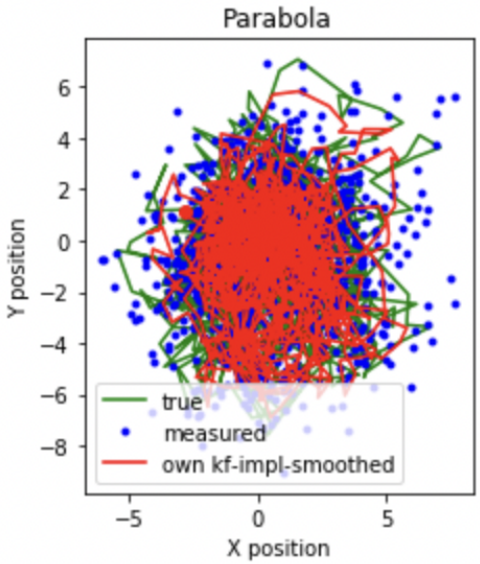


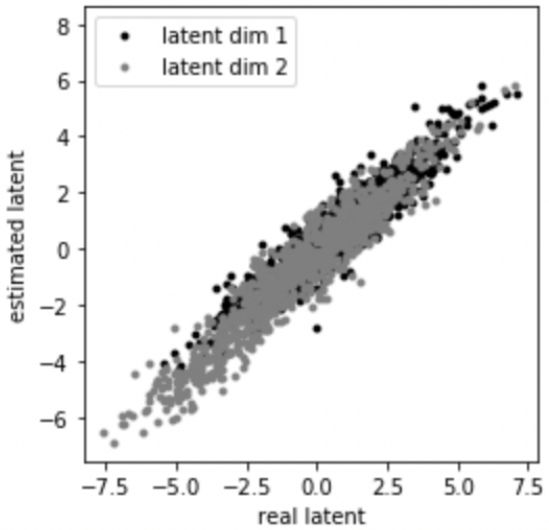


Comparing smoothing









Posterior log likelihood

```
# visualize the change of avg log posterior
visualize_line_plot(my_kf.avg_em_log_posterior, "# iter", "avg log posterior", "Log Posterior Progress")
```