Probabilistic Machine Learning

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Exercise 12

Summer Term 2023

Full Name	Matriculation No
Batuhan Özcömlekci	6300476
Aakarsh Nair	6546577

Eberhards Karls University of Tübingen, July 23, 2023

Excercise-12

Batuhan, Oezcoemlekci (Matrikelnummer: 6300476)
Aakarsh, Nair (Matrikelnummer: 6546577)

16 June 2023

Exercise

Theory Question Parameter inference with exectation maximization.

Consider a linear Gaussian model with the following structure:

- $x_0 \sim \mathcal{N}(m_0(\theta), P_0(\theta))$
- $x_k \sim \mathcal{N}(A(\theta)x_{k-1}, Q(\theta))$
- $y_k \sim \mathcal{N}(H(\theta)x_k, R(\theta))$

To-do: Estimate θ for some data $y_{1:N}$ an alternate to maximizing the likelihood (discussed in lecture) is the EM-Algorithm. It works as follows:

1. Start from an initial guess $\theta^{(0)}$.
2. For $n = 0, 1, 2, \dots$ do:
   (a) **E-Step:** Compute:

$$\mathcal{Q}\left(\theta, \theta^{(n)}\right) := \int p(x_{0:N}|y_{0:N}, \theta^{(n)}) \log p(x_{0:N}, y_{0:N}|\theta) dx_{0:N} \tag{1}$$

   (b) **M-Step:** Compute: $\theta^{(n+1)} := \arg\max_\theta \mathcal{Q}(\theta, \theta^{(n)})$

In a considered case of linear Gaussian state space model, $\mathcal{Q}(\theta, \theta^{(n)})$ can be computed analytically. It is of the form:

$$\begin{aligned}
\mathcal{Q}(^{\theta,\theta^{(n)}}) = \\
&- \frac{1}{2}\log|2\pi P_0(\theta)| - \frac{N}{2}\log|2\pi Q(\theta)| - \frac{N}{2}\log|2\pi R(\theta)| \\
&- \frac{1}{2}\operatorname{tr}\left(P_0^{-1}(\theta)\left[P_0^{(s)} + (m_0^{(s)} - m_0(\theta))(m_0^{(s)} - m_0(\theta))^T\right]\right) \\
&- \frac{1}{2}\operatorname{tr}\left(Q^{-1}(\theta)\left[\Sigma - CA(\theta)^T - A(\theta)C^T + A(\theta)\Phi A(\theta)^T\right]\right) \\
&- \frac{1}{2}\operatorname{tr}\left(R^{-1}(\theta)\left[D - BH(\theta)^T - H(\theta)B^T + H(\theta)\Sigma H(\theta)^T\right]\right)
\end{aligned} \tag{2}$$

Where the following quantities are computed form the results of the Rauch-Tung-Striebel smoother run with parameter values $\theta^{(n)}$:

$$\Sigma = \frac{1}{N}\sum_{k=1}^{N}\left(P_k^{(s)} + m_k^s (m_k^s)^T\right)$$

$$\Phi = \frac{1}{N}\sum_{k=1}^{N}\left(P_{k-1}^{(s)} + m_{k-1}^s (m_{k-1}^s)^T\right)$$

$$B = \frac{1}{N}\sum_{k=1}^{N}\left(y_k (m_k^{(s)})^T\right)$$

$$C = \frac{1}{N}\sum_{k=1}^{N}\left(P_k^{(s)} G_{k-1}^T + m_k^{(s)} (m_{k-1}^s)^T\right)$$

$$D = \frac{1}{N}\sum_{k=1}^{N}\left(y_k y_k^T\right)$$

Exercise: Let $\theta = (A, Q)$, that is the model parameters are exactly the full transition model matrices. Derive closed form updates for both A and Q by computing the M-step.

(a) Compute A: We use the fact that trace is a linear operator.

$$\begin{aligned}
&\frac{\partial Q}{\partial A} = 0 \\
&\implies \frac{\partial}{\partial A}\left[-\frac{N}{2}\operatorname{tr}\left(Q^{-1}(\Sigma - CA^T - AC^T + A\Phi A^T)\right)\right] = 0 \\
&\implies \frac{N}{2}\frac{\partial}{\partial A}\operatorname{tr}\left[-Q^{-1}AC^T - Q^{-1}CA^T + Q^{-1}A\Phi A^T\right] = 0 \\
&\implies \frac{\partial}{\partial A}\operatorname{tr}\left[-Q^{-1}AC^T - Q^{-1}CA^T + Q^{-1}A\Phi A^T\right] = 0 \\
&\implies \frac{\partial}{\partial A}\operatorname{tr}\left[Q^{-1}A\Phi A^T\right] = \frac{\partial}{\partial A}\operatorname{tr}\left[Q^{-1}AC^T + Q^{-1}CA^T\right]
\end{aligned} \tag{3}$$

Now consider the LHS:

$$\frac{\partial}{\partial A}\operatorname{tr}\left[Q^{-1}A\Phi A^T\right] \tag{4}$$

This is known matrix differential form of the trace.

$$\frac{\partial}{\partial X_d}\operatorname{tr}\left[A_d X_d B_d X_d^T C_d\right] = A_d^T C_d^T X_d B_d^T + C_d A_d X_d B_d$$

Where $A_d = Q^{-1}$, $B_d = \Phi$, $C_d = I$ and $X_d = A$.

Thus we have we use the symmetry of the postive semi-definte matrix inverse $Q^{-1} = (Q^{-1})^T$ to simplify the LHS:

$$\begin{aligned}
\frac{\partial}{\partial A}\operatorname{tr}\left[Q^{-1}A\Phi A^T\right] &= Q^{-T} I^T A\Phi^T + IQ^{-1}A\Phi \\
&= Q^{-T}A\Phi^T + Q^{-1}A\Phi \\
&= Q^{-T}A\Phi^T + Q^{-1}A\Phi \\
&= 2Q^{-1}A\Phi
\end{aligned} \tag{5}$$

For the RHS we can we can simplify the first term using:

$$\frac{\partial}{\partial X_d} \operatorname{tr}\left[A_d X_d B_d\right] = A_d^T B_d^T \qquad (6)$$

With $A_d = (Q^{-1})$ and $X_d = A$ and $B_d = C^T$ for the first term we get:

$$\frac{\partial}{\partial A} \operatorname{tr}\left[Q^{-1} A C^T\right] = Q^{-T} C \quad = Q^{-1} C \quad (7)$$

And for the second term we can use the following matrix identity we get:

$$\frac{\partial}{\partial X_d} \operatorname{tr}\left[A_d X_d^T\right] = A_d \qquad (8)$$

With $A_d = (Q^{-1})C$ and $X_d = A$ the second term simplifies to:

$$\frac{\partial}{\partial A} \operatorname{tr}\left[Q^{-1} C A^T\right] = Q^{-1} C \quad (9)$$

Thus taken together , using the linearity of the trace operator we can separate the terms we get:

$$\begin{aligned} \frac{\partial}{\partial A} \operatorname{tr}\left[Q^{-1} A C^T + Q^{-1} C A^T\right] &= Q^{-1} C + Q^{-1} C \\ &= 2Q^{-1} C \end{aligned} \qquad (10)$$

Thus LHS and RHS together becomes:

$$\begin{aligned} 2Q^{-1} A \Phi &= 2Q^{-1} C \\ Q^{-1} A \Phi &= Q^{-1} C \\ A \Phi &= C \\ A &= C \Phi^{-1} \end{aligned} \qquad (11)$$

Thus we have $A^* = C\Phi^{-1}$.

(b) Compute Q:

We again compute the critical point by computing the matrix derivative with respect to Q and setting it to zero. Canceling the the constant from the linear operators for trace and derivative.

Let

$$Z = \left[\Sigma - CA^T - AC^T + A\Phi A^T\right]$$

$$
\begin{aligned}
\frac{\partial}{\partial Q}\left[-\frac{N}{2}\log|2\pi Q| - \frac{N}{2}\operatorname{tr}\left(Q^{-1}Z\right)\right] &= 0 \\
\frac{\partial}{\partial Q}\left[-\log|2\pi Q| - \frac{\partial}{\partial Q}\operatorname{tr}\left(Q^{-1}Z\right)\right] &= 0 \\
\frac{\partial}{\partial Q}\left[-\log|2\pi Q|\right] - \left[\frac{\partial}{\partial Q}\operatorname{tr}\left(Q^{-1}Z\right)\right] &= 0 \\
\frac{\partial}{\partial Q}\left[-\log|Q|\right] - \left[\frac{\partial}{\partial Q}\operatorname{tr}\left(Q^{-1}Z\right)\right] &= 0 \\
-\left[\frac{\partial}{\partial Q}\operatorname{tr}\left(Q^{-1}Z\right)\right] &= \frac{\partial}{\partial Q}\left[\log|Q|\right]
\end{aligned} \tag{12}
$$

Consider the LHS, we use the following matrix differential form of the trace.

$$
\frac{\partial}{\partial X_d}\operatorname{tr}(A_d X_d^{-1} B_d) = -X_d^{-T} A_d^T B_d^T X_d^{-T} \tag{13}
$$

With $A_d = I$, $B_d = Z$ and $X_d = Q$, and the fact that $Q^{-T} = Q^{-1}$ being inverse of PSD metrix we get:

$$
\frac{\partial}{\partial Q}(Q^{-1}Z) = -Q^{-T}Z^T Q^{-T} = -Q^{-1}Z^T Q^{-1} \tag{14}
$$

We also note that $Z = Z^T$ as we have that Σ and Φ are symmetric matrices.

$$
\begin{aligned}
Z^T &= (\Sigma - CA^T - AC^T + A\Phi A^T)^T & &= (\Sigma^T - AC^T - CA^T + A\Phi^T A^T) \\
&= (\Sigma - CA^T - AC^T + A\Phi A^T) & &= Z
\end{aligned} \tag{15}
$$

Thus we get the LHS to be :

$$
\frac{\partial}{\partial X_d}\operatorname{tr}(Q^{-1}Z) = -Q^{-1}ZQ^{-1} \tag{16}
$$

For the RHS we use the matrix identity:

$$
\frac{\partial}{\partial X_d}\log|X_d| = X_d^{-T} \tag{17}
$$

With $X_d = Q$ we get the RHS to be:

$$
\frac{\partial}{\partial Q}\log|Q| = Q^{-T} = Q^{-1} \tag{18}
$$

Taken together we get

$$
\begin{aligned}
-Q^{-1}ZQ^{-1} &= -Q^{-1} \\
\implies Q^{-1}ZQ^{-1} &= Q^{-1} \\
\implies ZQ^{-1} &= I
\end{aligned} \tag{19}
$$

Thus from the uniqueness of the inverse we get:

$$Q^{*} = Z = (\Sigma - CA^{T} - AC^{T} + A\Phi A^{T}) \tag{20}$$

References

Ex12_code

July 24, 2023

1 Probabilistic Machine Learning

University of Tübingen, Summer Term 2023 © 2023 P. Hennig

1.1 Exercise Sheet No. 12

Submission by:

- Batuhan, Oezcoemlekci, Matrikelnummer: 6300476
- Aakarsh, Nair, Matrikelnummer: 6546577

[ ]:
```
import numpy as np

from matplotlib import pyplot as plt
from numpy.typing import ArrayLike

import scipy.io as sio
import scipy.linalg as sla
import scipy.special as ssp

from tueplots import bundles
from tueplots.constants.color import rgb

# plt.rcParams.update(bundles.beamer_moml())
plt.rcParams.update({"figure.dpi": 200})
```

In this exercise, you will implement a Rauch-Tung-Striebel (RTS) smoother. We will work on the same data as before so you will recognize parts of the notebook from the previous exercise.

As before, only change code in cells where we explicitly ask you to.

2 I. The Data

[ ]:
```
DIM = 7
NUM_DERIV = 2
STATE_DIM = DIM * (NUM_DERIV + 1)
```

[ ]:
```
proj_position = np.eye(STATE_DIM)[:DIM, :]
proj_velocity = np.eye(STATE_DIM)[DIM:2*DIM, :]
proj_acceleration = np.eye(STATE_DIM)[2*DIM:, :]
```

[ ]:
```
def plot_data(axs, Y):
    assert len(axs) == 3
    N, d = Y.shape
    num_joints = d // 3
    xs = np.arange(N)
    positions = Y @ proj_position.T
    velocities = Y @ proj_velocity.T
    accelerations = Y @ proj_acceleration.T
    for i in range(num_joints):
        axs[0].scatter(xs, positions[:, i], marker="x", s=5, label="joint {}".
 ↪format(i), color="C{}".format(i))
        axs[1].scatter(xs, velocities[:, i], marker="x", s=5, label="joint {}".
 ↪format(i), color="C{}".format(i))
        axs[2].scatter(xs, accelerations[:, i], marker="x", s=5, label="joint␣
 ↪{}".format(i), color="C{}".format(i))
#     plt.legend()
    return axs
```

[ ]:
```
def plot_estimate(axs, kf_means, kf_covs, fctr=1.97):
    assert len(axs) == 3
    N, d = kf_means.shape
    num_joints = d // 3
    xs = np.arange(N)
    m_positions = kf_means @ proj_position.T
    m_velocities = kf_means @ proj_velocity.T
    m_accelerations = kf_means @ proj_acceleration.T

    kf_stds = np.array([fctr * np.sqrt(np.diag(C)) for C in kf_covs])
    s_positions = kf_stds @ proj_position.T
    s_velocities = kf_stds @ proj_velocity.T
    s_accelerations = kf_stds @ proj_acceleration.T

    for i in range(num_joints):
        axs[0].plot(xs, m_positions[:, i], color="C{}".format(i))
        axs[0].fill_between(xs, m_positions[:, i] - s_positions[:, i],␣
 ↪m_positions[:, i] + s_positions[:, i], color="C{}".format(i), alpha=0.4)
        axs[1].plot(xs, m_velocities[:, i], color="C{}".format(i))
        axs[1].fill_between(xs, m_velocities[:, i] - s_velocities[:, i],␣
 ↪m_velocities[:, i] + s_velocities[:, i], color="C{}".format(i), alpha=0.4)
        axs[2].plot(xs, m_accelerations[:, i], color="C{}".format(i))
        axs[2].fill_between(xs, m_accelerations[:, i] - s_accelerations[:, i],␣
 ↪m_accelerations[:, i] + s_accelerations[:, i], color="C{}".format(i),␣
 ↪alpha=0.4)
```

```
#     plt.legend()
    return axs
```

[ ]:
```
data = sio.loadmat('sarcos_inv.mat')["sarcos_inv"][:, :-7]
Y = data
```

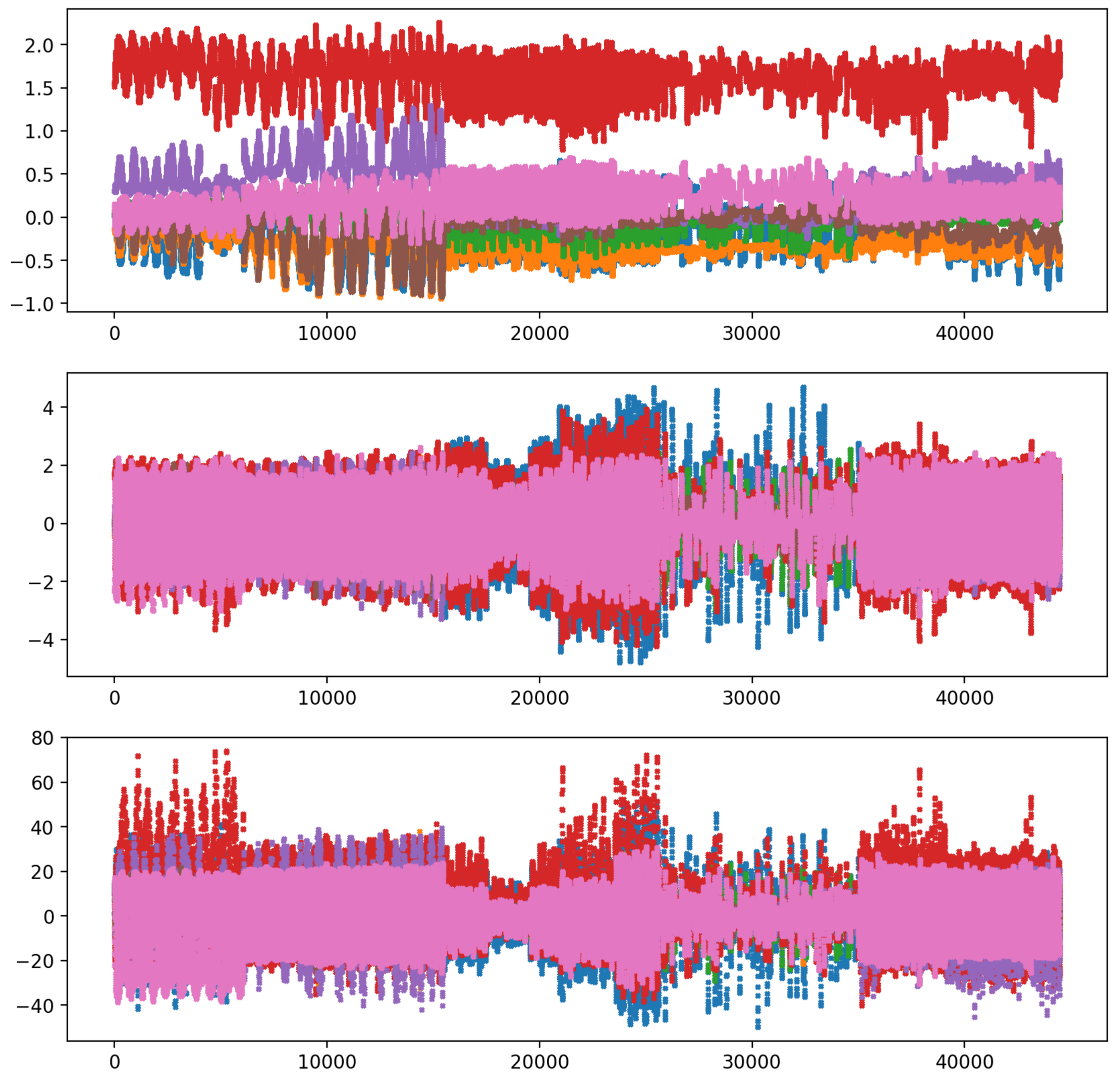
[ ]:
```
data.shape
```

[ ]:
```
(44484, 21)
```

2.1 Now, let's have a look at the entire time series.

[ ]:
```
fig, axs = plt.subplots(3,1, figsize=(10, 10))
plot_data(axs, data)
```

[ ]:
```
array([<Axes: >, <Axes: >, <Axes: >], dtype=object)
```

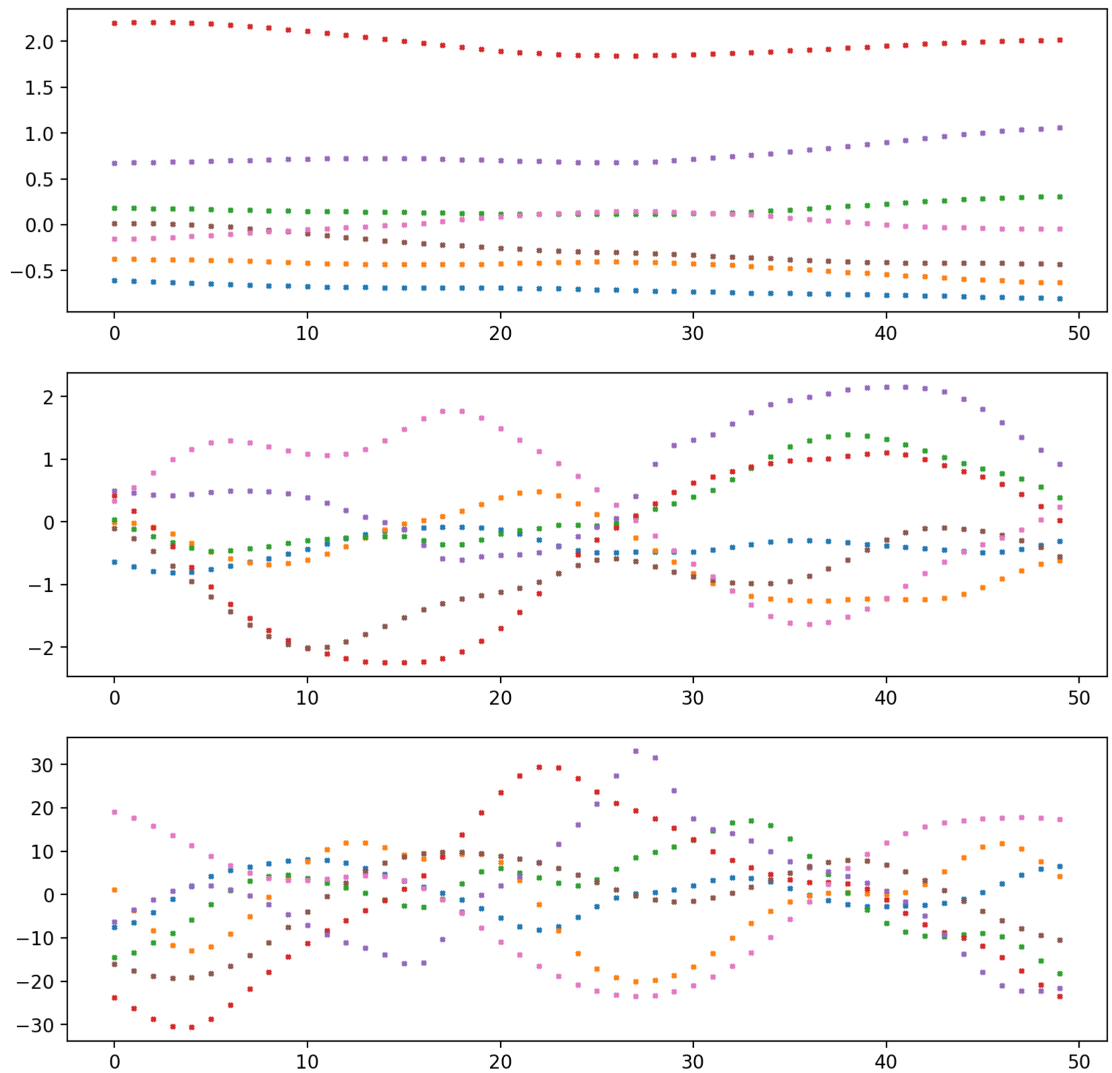


Pretty chaotic, huh... Well, it's over 44 thousand data points and our screen is only so wide... That's why we are going to look at a smaller, zoomed-in window from now on.

[ ]:
```
time_window_for_viz = slice(11000, 11050)
```

[ ]:
```
fig, axs = plt.subplots(3,1, figsize=(10, 10))
plot_data(axs, data[time_window_for_viz, :])
```

[ ]:
```
array([<Axes: >, <Axes: >, <Axes: >], dtype=object)
```



3 II. The Model

Ok, now that we have a feel for the data and what it looks like, we are going to set up a model. What kind of model this is, we are going to be secretive about for now. Perhaps, you will learn about it in one of the following lectures? (Perhaps not, let's see).

The following two cells ... ... are mysterious functions that create us two matrices A and Q, the transition matrix and process-noise covariance matrix of our linear, time-invariant Gaussian transition density.

You do not have to understand what these two functions do, just take them for granted! (I wouldn't try, anyway...)

The dynamics model (prior)

```
[ ]: def mysterious_operation(F, L, dt):
         dim = F.shape[0]

         if L.ndim == 1:
             L = L.reshape((-1, 1))

         Phi = np.block(
             [
                 [F, L @ L.T],
                 [np.zeros(F.shape), -F.T],
             ]
         )
         M = sla.expm(Phi * dt)

         Ah = M[:dim, :dim]
         Qh = M[:dim, dim:] @ Ah.T

         return Ah, Qh
```

```
[ ]: def create_mysterious_ssm(d, q, ell, dt, dff=1.0):
         F = np.diag(np.ones(q), 1)
         nu = q + 0.5
         D, lam = q + 1, np.sqrt(2 * nu) / ell
         F[-1, :] = np.array(
                 [-ssp.binom(D, i) * lam ** (D - i) for i in range(D)]
             )

         L = np.eye(q+1)[-1, :] * dff
         A_1d, Q_1d = mysterious_operation(F, L, dt)
         A = np.kron(A_1d, np.eye(d))
         Q = np.kron(Q_1d, np.eye(d))
         return A, Q
```

```
[ ]: dt = 1.0
```

```
[ ]: A, Q = create_mysterious_ssm(DIM, NUM_DERIV, 10.0, dt, 50.0)
```

The measurement model (likelihood)

```
[ ]: H = np.eye(STATE_DIM)  # We measure the entire state.
     R = np.kron(np.diag(np.array([0.01, 0.01, 1.0])), np.eye(DIM))  # a (not-quite␣
      ↪isotropic) sensor noise.
```

Finally, we set the initial moments to the first data point with some uncertainty.

```
[ ]: m0 = data[0, :]
     P0 = np.kron(np.diag(np.array([0.01, 0.01, 1.0])), np.eye(DIM))
```

4 III. Inference

4.1 Step 1: Filtering

[ ]:
```
def symmetrize(A):
    return 0.5 * (A + A.T) + (1e-8 * np.eye(A.shape[0]))
```

[ ]:
```
def kf_predict(m_filt, P_filt, A, Q):
    m_pred = A @ m_filt
    P_pred = A @ P_filt @ A.T + Q
    return m_pred, symmetrize(P_pred)
```

[ ]:
```
def kf_update(m_pred, P_pred, H, R, y):
    predicted_measurement = H @ m_pred
    innovation = (y - predicted_measurement)
    innovation_gramian = H @ P_pred @ H.T + R

    S_chol_fact = sla.cho_factor(symmetrize(innovation_gramian))
    cross_covariance = P_pred @ H.T
    kalman_gain = sla.cho_solve(S_chol_fact, cross_covariance.T).T

    mean_increment = kalman_gain @ innovation
    covariance_decrement = kalman_gain @ innovation_gramian @ kalman_gain.T

    m_filt = m_pred + mean_increment
    P_filt = P_pred - covariance_decrement
    return m_filt, symmetrize(P_filt)
```

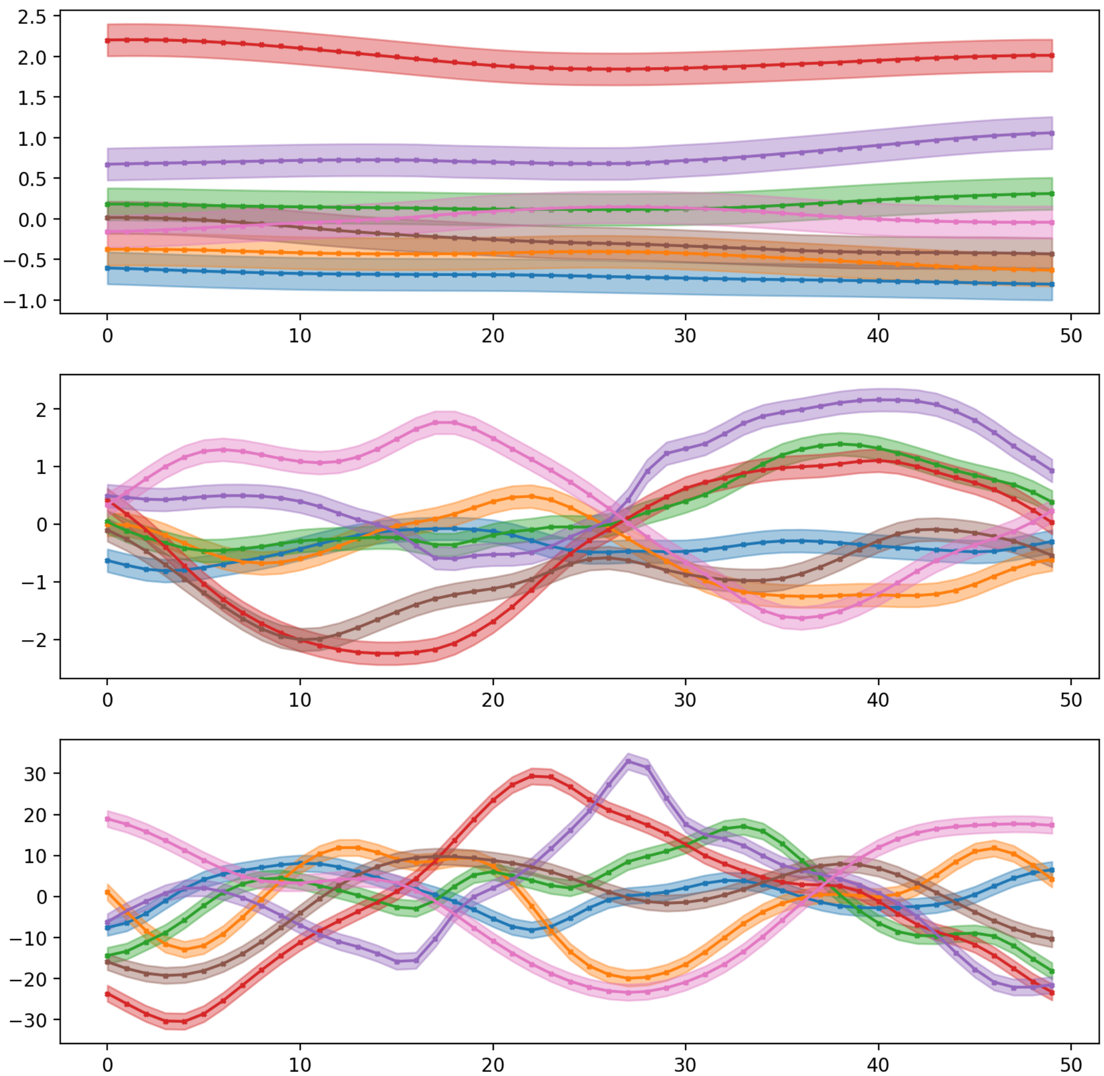
[ ]:
```
def filter_kalman(m0, P0, A, Q, H, R, Y):
    d, D = H.shape
    N = Y.shape[0]
    result_mean = [m0.copy()]
    result_cov = [P0.copy()]
    m = m0.copy()
    P = P0.copy()
    for n in range(1, N):
        m, P = kf_predict(m, P, A, Q)
        m, P = kf_update(m, P, H, R, Y[n, :])
        result_mean.append(m.copy())
        result_cov.append(P.copy())
    return np.array(result_mean), np.array(result_cov)
```

[ ]:
```
%%time
kf_means, kf_covs = filter_kalman(m0, P0, A, Q, H, R, Y)
```

```
CPU times: user 5.29 s, sys: 72 ms, total: 5.36 s
Wall time: 5.36 s
```

```
[ ]: fig, axs = plt.subplots(3,1, figsize=(10, 10))
     plot_estimate(axs, kf_means[time_window_for_viz, :],
      ↪kf_covs[time_window_for_viz, :, :])
     plot_data(axs, data[time_window_for_viz, :])
```

```
[ ]: array([<Axes: >, <Axes: >, <Axes: >], dtype=object)
```



4.2 Step 2: Smoothing

4.3 Task 1:

Implement a Rauch-Tung-Striebel smoother. ### (a) Fill the function body of the function `smoother_step` below. It takes as arguments - `filt_m`, `filt_P`: the filtering moments at time step $k-1$ - `pred_m`, `pred_P`: the predicted moments at time step k - `smooth_m`, `smooth_P`: the smoothed moments at time step k - `A`, `Q`: the parameters of the transition density

The function must return a tuple containing 1. `xi` the smoothing mean at time step $k-1$ 2. `Lambda` the smoothing covariance at time step $k-1$ 3. `G` the smoothing gain used to compute the above

[ ]:
```
def smoother_step(filt_m, filt_P, pred_m, pred_P, smooth_m, smooth_P, A, Q):
    pred_P_cho_factor, _ = sla.cho_factor(pred_P, lower=True)
    pred_P_inverse = sla.cho_solve((pred_P_cho_factor, True), np.eye(pred_P.
 ↪shape[0]))
    #pred_P_inverse = sla.inv(pred_P_inverse)
    G = filt_P @ A.T @ pred_P_inverse
    xi = filt_m + G @ (smooth_m - pred_m)
    Lambda = filt_P + G @ (smooth_P - pred_P) @ G.T
    return xi, Lambda, G
```

4.4 (b)

Fill the function body of the function `rts_smooth` below. It takes as arguments - `filter_means` - `filter_covs` - `A`, `Q`: the parameters of the transition density

4.4.1 IMPORTANT

This function must return a tuple of three arrays: - an $N \times D$-array containing the smoother means - an $N \times D \times D$-array containing the smoother covariances - AND an $N \times D \times D$-array containing the smoother gains G_k from every step. We will need the last one for later.

[ ]:
```
def rts_smooth(filter_means, filter_covs, A, Q):
    smoother_means = np.zeros_like(filter_means)
    smoother_covs = np.zeros_like(filter_covs)
    smoother_gains = np.zeros_like(filter_covs)

    for i in range(filter_means.shape[0], 0, -1):
        if i == filter_means.shape[0]:
            smoother_means[i-1, :] = filter_means[i-1, :]
            smoother_covs[i-1, :, :] = filter_covs[i-1, :, :]
        else:
            smoother_means[i-1, :], smoother_covs[i-1, :, :],␣
 ↪smoother_gains[i-1, :, :] = smoother_step(
                filter_means[i-1, :], filter_covs[i-1, :, :],
                filter_means[i, :], filter_covs[i, :, :],
                smoother_means[i, :], smoother_covs[i, :, :],
                A, Q
            )

    return smoother_means, smoother_covs, smoother_gains
```

5 Now we test your implementation.

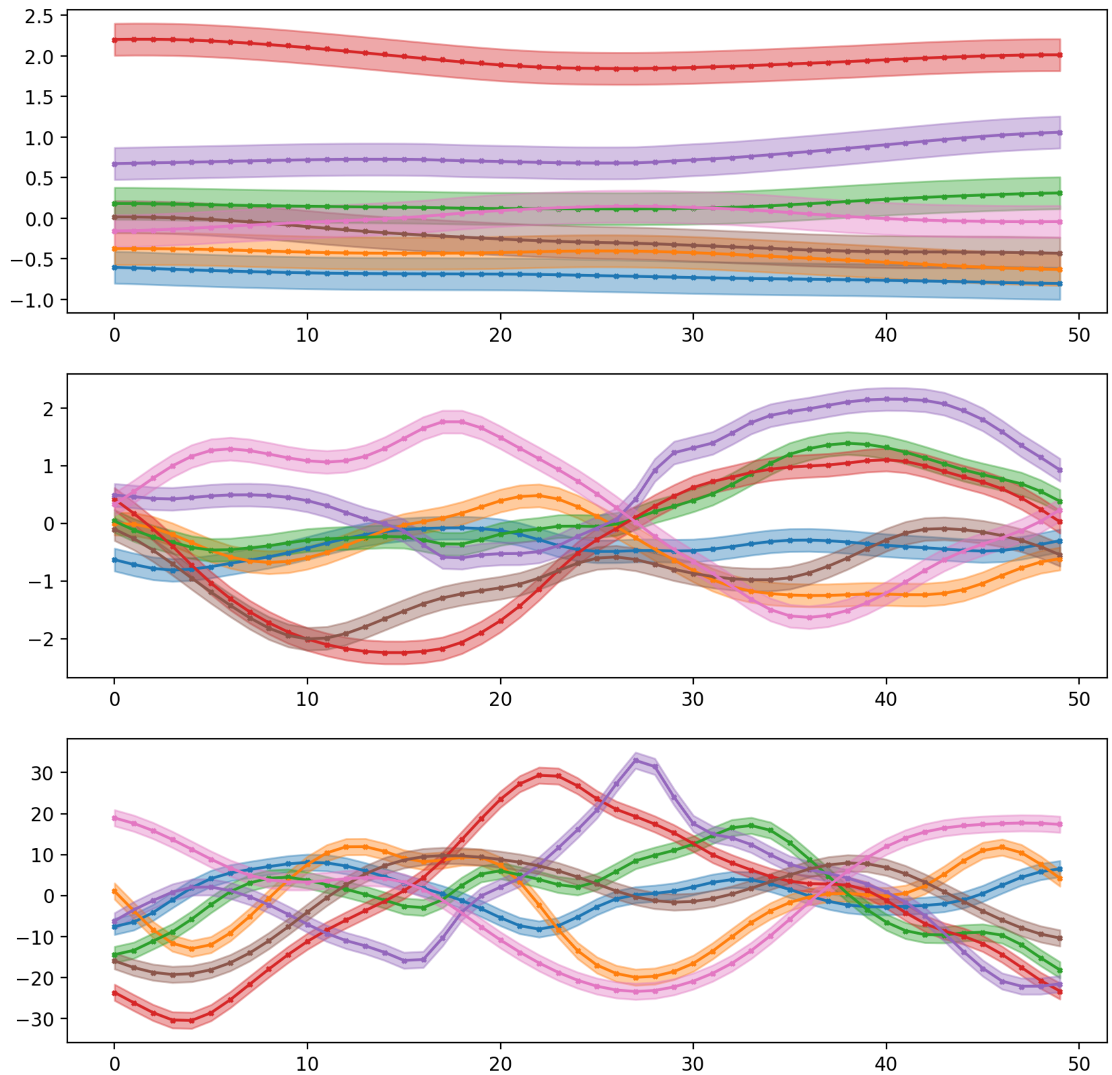
DO NOT CHANGE ANYTHING IN THESE CELLS

[ ]:
```
%%time
rts_means, rts_covs, _ = rts_smooth(kf_means, kf_covs, A, Q)
```

```
CPU times: user 2.76 s, sys: 60.2 ms, total: 2.83 s
Wall time: 2.84 s
```

[ ]:
```
fig, axs = plt.subplots(3,1, figsize=(10, 10))
plot_estimate(axs, rts_means[time_window_for_viz, :],␣
  ↪rts_covs[time_window_for_viz, :, :])
plot_data(axs, data[time_window_for_viz, :])
```

[ ]:
```
array([<Axes: >, <Axes: >, <Axes: >], dtype=object)
```



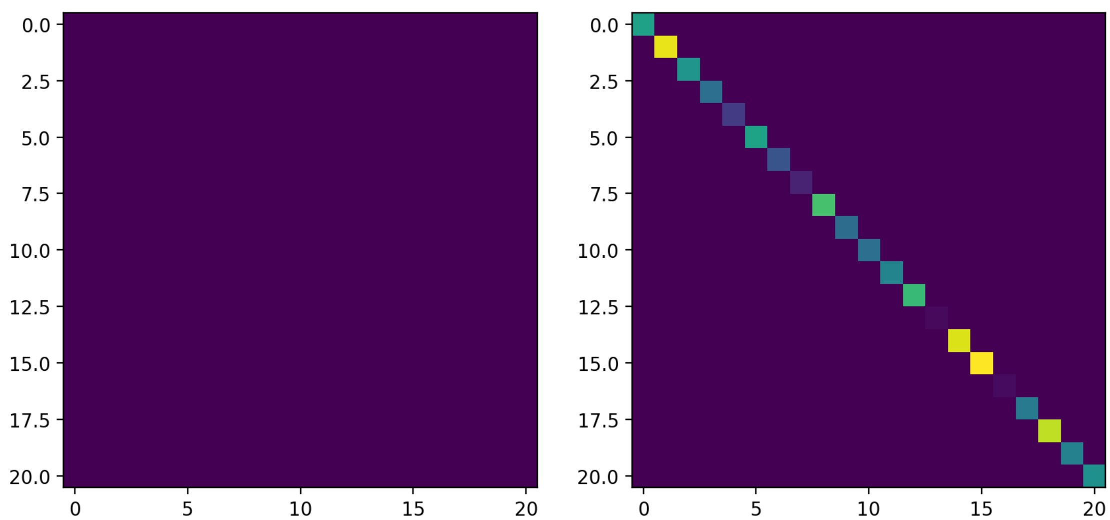
5.1 Task 2

All right, now - say - we do not have a good model for the data we see. Take for example this transition model here, which is really just random Gaussian white noise in every step.

[ ]:
```
A_init = np.zeros((STATE_DIM, STATE_DIM))
Q_init = 0.01*np.diag(np.random.rand(STATE_DIM))
```

[ ]:
```
fig, axs = plt.subplots(1, 2, figsize=(10, 10))
axs[0].imshow(A_init)
axs[1].imshow(Q_init)
```

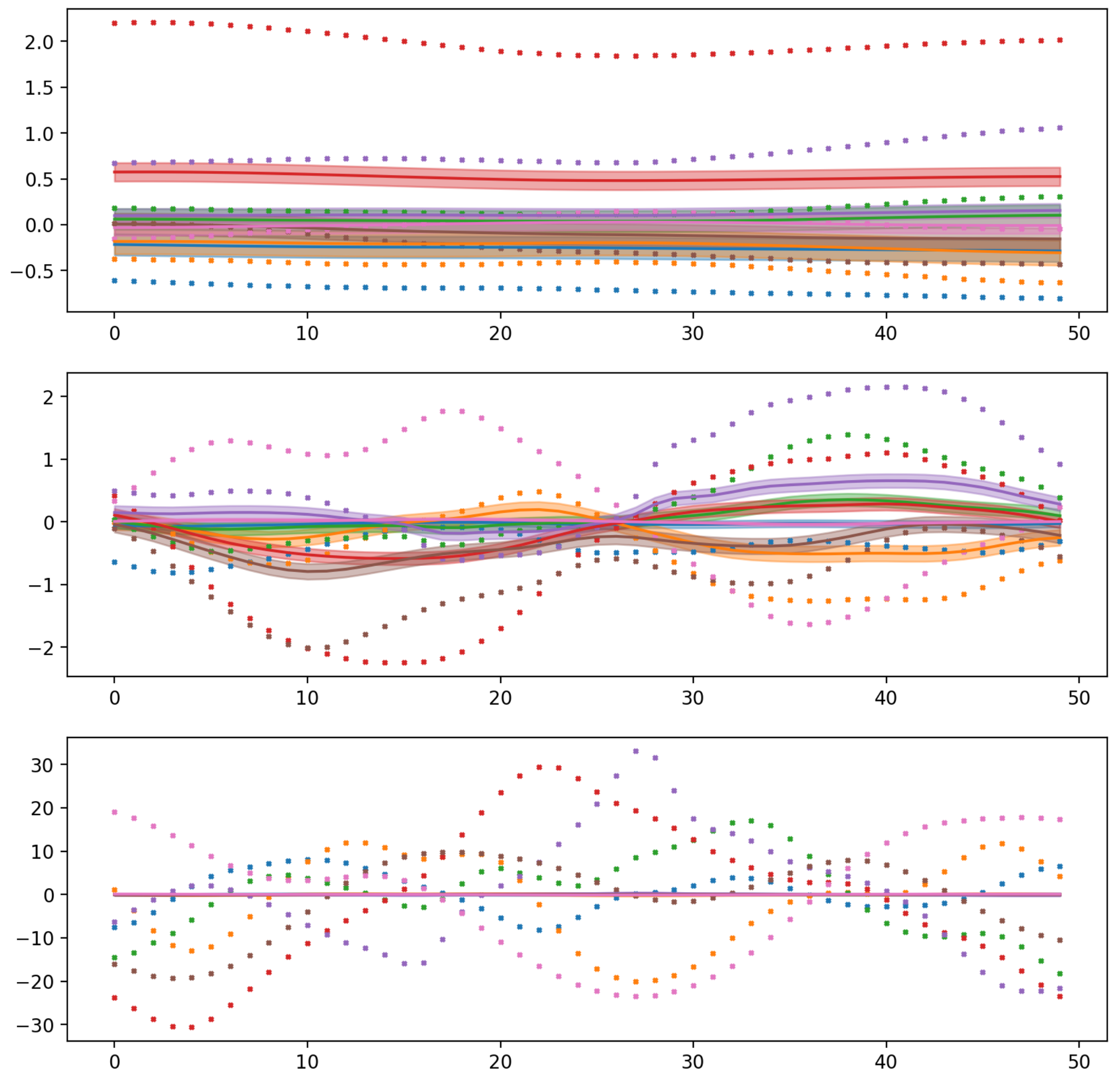
[ ]:
```
<matplotlib.image.AxesImage at 0x7f0b86305490>
```



[ ]:
```
init_kf_means, init_kf_covs = filter_kalman(m0, P0, A_init, Q_init, H, R, Y)
```

[ ]:
```
fig, axs = plt.subplots(3,1, figsize=(10, 10))
plot_estimate(axs, init_kf_means[time_window_for_viz, :],␣
 ↪init_kf_covs[time_window_for_viz, :, :])
plot_data(axs, data[time_window_for_viz, :])
```

[ ]:
```
array([<Axes: >, <Axes: >, <Axes: >], dtype=object)
```



Surprise, the fit is not so good. Luckily, you know a tool that might help.

6 The EM-Algorithm for linear Gaussian state-space models

6.0.1 (a) Implement the E-step of the EM algorithm

as given in the theory exercise. The function takes - `m0`, `P0`: initial moments - `A`, `Q`: a transition model - `H`, `R`: a measurement model - `Y`

The E-step computes a bunch of matrices

- `Sigma`
- `Phi`
- `B`
- `C`

- D

from the RTS smoother estimate given the current SSM and returns them.

```
[ ]: def E_step(m0, P0, A, Q, H, R, Y):
         N = Y.shape[0]
         _kf_means, _kf_covs = filter_kalman(m0, P0, A, Q, H, R, Y)

         for i in range(_kf_covs.shape[0] - 1, 0, -1):
             assert np.all(np.linalg.eigvals(_kf_covs[i, :, :]) > 0), f"matrix is␣
     ↪not psd {np.linalg.eigvals(_kf_covs[i, :, :])}"
             assert np.allclose(_kf_covs[i, :, :], _kf_covs[i, :, :].T), "covariance␣
     ↪is not symmetric"


         rts_means, rts_covs, rts_gains = rts_smooth(_kf_means, _kf_covs, A, Q)


         Sigma = (1.0 / N) * np.sum([rts_covs[k, : , :] + np.outer(rts_means[k, :] ,␣
     ↪rts_means[k, :].T) for k in range(N)], axis=0)
         Phi = (1.0 / N) * np.sum([rts_covs[k-1, : , :] + np.outer(rts_means[k-1, :]␣
     ↪, rts_means[k-1, :].T) for k in range(1, N)], axis=0)
         B = (1.0/ N) * np.sum([np.outer(Y[k, :] , rts_means[k, :].T)for k in␣
     ↪range(N)], axis=0)
         # C - uses previous gains unlike term in assignment.
         C = (1.0 / N) * np.sum([ rts_covs[k, :, :] @ rts_gains[k-1, :, :].T + np.
     ↪outer(rts_means[k, :] , rts_means[k-1, :].T) for k in range(N)], axis=0)
         D = (1.0 / N) * np.sum([np.outer(Y[k, :] , Y[k, :]) for k in range(N)],␣
     ↪axis=0)

         assert Sigma.shape == (STATE_DIM, STATE_DIM)
         assert Phi.shape == (STATE_DIM, STATE_DIM)
         assert B.shape == (STATE_DIM, STATE_DIM)
         assert C.shape == (STATE_DIM, STATE_DIM)
         assert D.shape == (STATE_DIM, STATE_DIM)

         return symmetrize(Sigma), symmetrize(Phi), B, C, D
```

6.0.2 (b) Compute the M-Step for the transition matrix A

based on the results of the E-Step

```
[ ]: def M_step_A(Sigma, Phi, B, C, D):
         # Your code goes here.
         Phi_cho_factor, _ = sla.cho_factor(symmetrize(Phi), lower=True)
         Phi_inv = sla.cho_solve((Phi_cho_factor, True), np.eye(Phi.shape[0]))
```

```
    assert np.allclose(symmetrize(Phi) @ Phi_inv, np.eye(Phi.shape[0]),␣
↪rtol=1e-4, atol=1e-6), "inverse is wrong, {}".format(np.linalg.norm(Phi @␣
↪Phi_inv - np.eye(Phi.shape[0])))
    return C @ Phi_inv
```

6.0.3 (c) Compute the M-Step for the transition noise covariance Q

based on the results of the E-Step

[ ]:
```
def M_step_Q(Sigma, Phi, B, C, D, A):
    # Your code goes here.
    return symmetrize(Sigma  - C @ A.T  - A @ C.T  + A @ Phi @ A.T)
```

7 From here on, DON'T CHANGE ANYTHING.

7.0.1 This might take a while

[ ]:
```
def EM_AQ(m0, P0, A_init, Q_init, H, R, Y, n_iter):
    A_star = A_init.copy()
    Q_star = Q_init.copy()
    for i in range(n_iter):
        print("EM-Step {}".format(i+1))
        Sigma, Phi, B, C, D = E_step(m0, P0, A, Q_star, H, R, Y)
        A_star = M_step_A(Sigma, Phi, B, C, D)
        Q_star = M_step_Q(Sigma, Phi, B, C, D, A)
    return A_star, Q_star
```

[ ]:
```
A_star, Q_star = EM_AQ(m0, P0, A_init, Q_init, H, R, Y, 100)
```

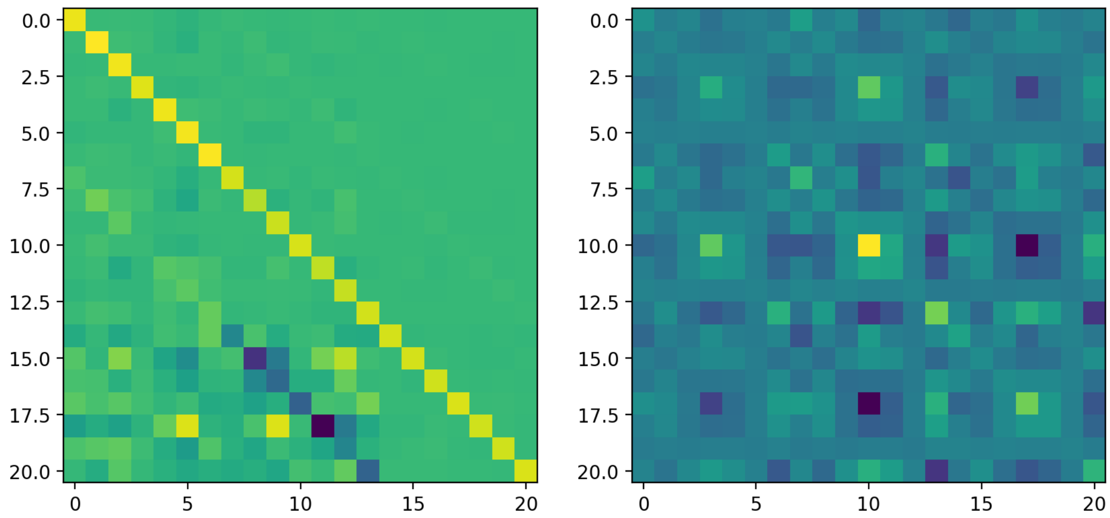
```
EM-Step 1
EM-Step 2
EM-Step 3
EM-Step 4
EM-Step 5
EM-Step 6
EM-Step 7
EM-Step 8
EM-Step 9
EM-Step 10
EM-Step 11
EM-Step 12
EM-Step 13
EM-Step 14
EM-Step 15
EM-Step 16
EM-Step 17
```

```
EM-Step 18
EM-Step 19
EM-Step 20
EM-Step 21
EM-Step 22
EM-Step 23
EM-Step 24
EM-Step 25
EM-Step 26
EM-Step 27
EM-Step 28
EM-Step 29
EM-Step 30
EM-Step 31
EM-Step 32
EM-Step 33
EM-Step 34
EM-Step 35
EM-Step 36
EM-Step 37
EM-Step 38
EM-Step 39
EM-Step 40
EM-Step 41
EM-Step 42
EM-Step 43
EM-Step 44
EM-Step 45
EM-Step 46
EM-Step 47
EM-Step 48
EM-Step 49
EM-Step 50
EM-Step 51
EM-Step 52
EM-Step 53
EM-Step 54
EM-Step 55
EM-Step 56
EM-Step 57
EM-Step 58
EM-Step 59
EM-Step 60
EM-Step 61
EM-Step 62
EM-Step 63
EM-Step 64
EM-Step 65
```

```
EM-Step 66
EM-Step 67
EM-Step 68
EM-Step 69
EM-Step 70
EM-Step 71
EM-Step 72
EM-Step 73
EM-Step 74
EM-Step 75
EM-Step 76
EM-Step 77
EM-Step 78
EM-Step 79
EM-Step 80
EM-Step 81
EM-Step 82
EM-Step 83
EM-Step 84
EM-Step 85
EM-Step 86
EM-Step 87
EM-Step 88
EM-Step 89
EM-Step 90
EM-Step 91
EM-Step 92
EM-Step 93
EM-Step 94
EM-Step 95
EM-Step 96
EM-Step 97
EM-Step 98
EM-Step 99
EM-Step 100
```

[ ]:
```
fig, axs = plt.subplots(1, 2, figsize=(10, 10))
axs[0].imshow(A_star)
axs[1].imshow(Q_star)
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f0b858dd490>
```

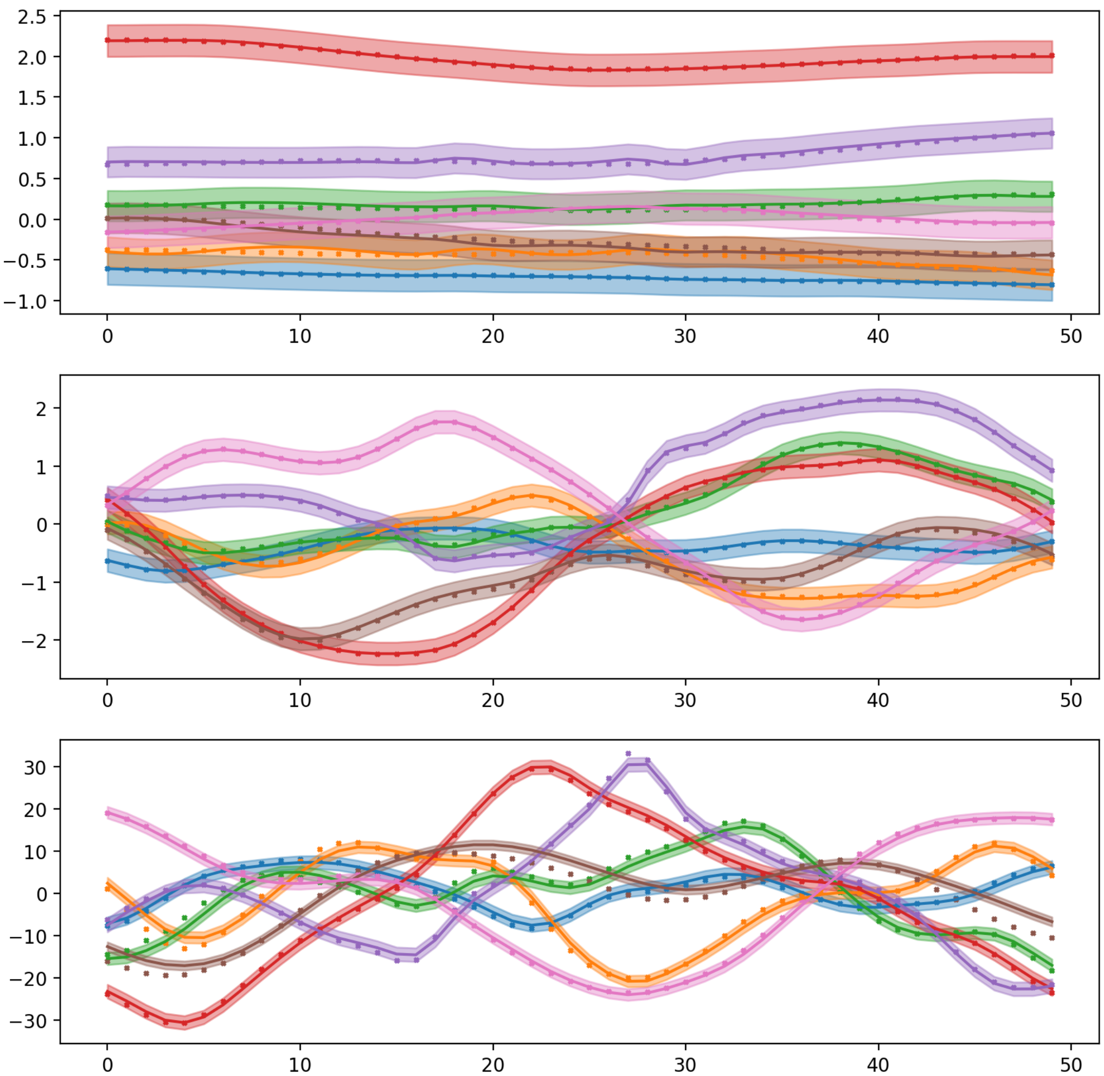


[ ]:
```
%%time
kf_means, kf_covs = filter_kalman(m0, P0, A_star, Q_star, H, R, Y)
```

```
CPU times: user 4.71 s, sys: 60 ms, total: 4.77 s
Wall time: 4.77 s
```

[ ]:
```
fig, axs = plt.subplots(3,1, figsize=(10, 10))
plot_estimate(axs, kf_means[time_window_for_viz, :],␣
    ↪kf_covs[time_window_for_viz, :, :])
plot_data(axs, data[time_window_for_viz, :])
```

[ ]:
```
array([<Axes: >, <Axes: >, <Axes: >], dtype=object)
```



7.1 If everything went as planned, ...

... you should see a good model with a fine fit now.

7.1.1 How to submit your work:

Export your answer into a pdf (for example using jupyter's `Save and Export Notebook as` feature in the `File` menu). Make sure to include all outputs, in particular plots. Also include your answer to the theory question, either by adding it as LaTeX code directly in the notebook, or by adding it as an extra page (e.g. a scan) to the pdf. Submit the exercise on Ilias, in the associated folder. **Do not forget to add your name(s) and matrikel number(s) above!)**