

TSSL Lab 2 - Structural model, Kalman filtering and EM

Poss

We will continue to work with the Global Mean Sea Level (GMSL) data that we got acquainted with in lab 1. The data is taken from <https://climate.nasa.gov/vital-signs/sea-level/> and is available on LISAM in the file `sealevel.csv`.

In this lab we will analyse this data using a structural time series model. We will first set up a model and implement a Kalman filter to infer the latet states of the model, as well doing long-term prediction. We will then implement a disturbance smoother and an expectation maximization algorithm to tune the parameters of the model.

We load a few packages that are useful for solving this lab assignment.

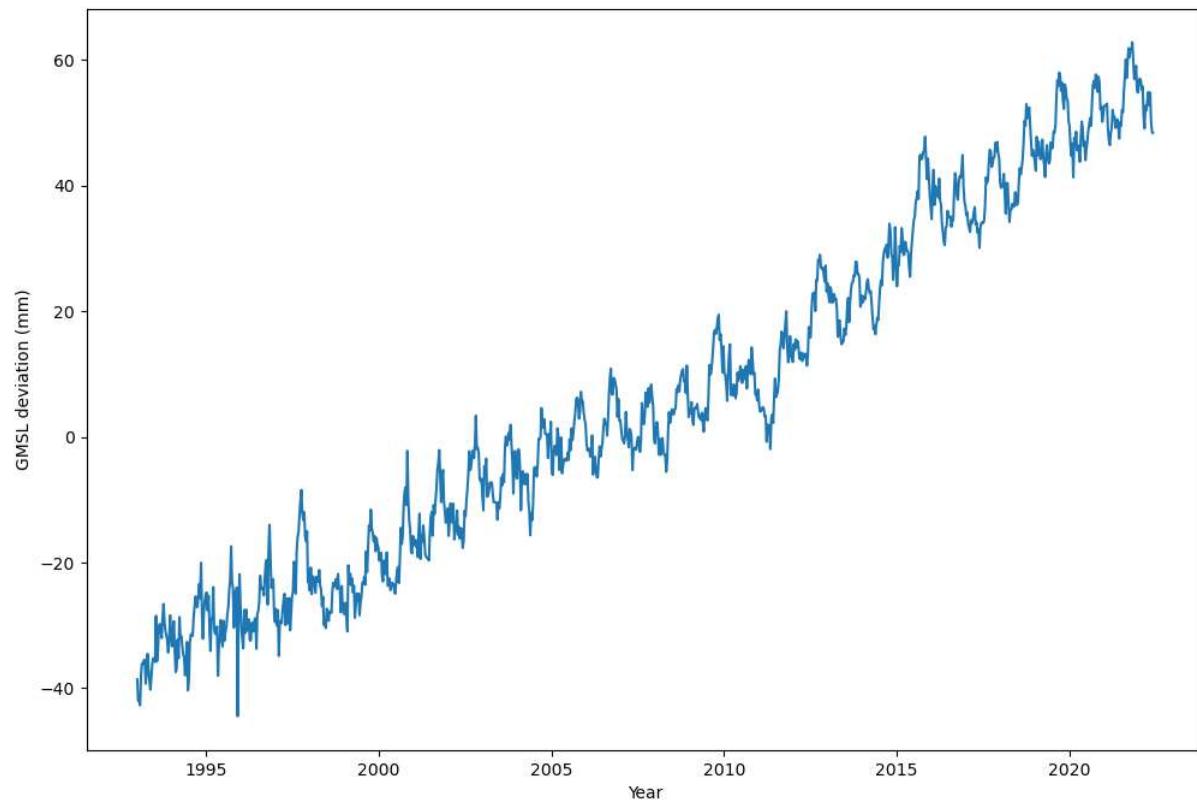
```
In [1]: import pandas # Loading data / handling data frames
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (12,8) # Increase default size of plots
import math
```

2.1 Setting up a structural state space model

We start by loading and plotting data to reming ourselves what it looks like.

```
In [2]: data=pandas.read_csv('sealevel.csv',header=0)
```

```
In [3]: y = data['GMSL'].values
u = data['Year'].values
ndata = len(y)
plt.plot(u,y)
plt.xlabel('Year')
plt.ylabel('GMSL deviation (mm)')
plt.show()
```



In this lab we will use a structural time series model to analys this data set. Specifically, we assume that the data $\{y_t\}_{t \geq 1}$ is generated by

$$y_t = \mu_t + \gamma_t + \varepsilon_t \quad (1)$$

where μ_t is a trend component, γ_t is a seasonal component, and ε_t is an observation noise. The model is expressed using a state space representation,

$$\begin{aligned} \alpha_{t+1} &= T\alpha_t + R\eta_t, & \eta_t &\sim N(0, Q), \\ y_t &= Z\alpha_t + \varepsilon_t, & \varepsilon_t &\sim N(0, \sigma_\varepsilon^2). \end{aligned} \quad (2) \quad (3)$$

Q0: Let $d = \dim(\alpha_t)$ denote the state dimension and $d_\eta = \dim(\eta_t)$ denote the dimension of the state noise. Then, what are the dimenisons of the matrices T , R , and Z of the state space model?

A: Dimensions are as follows:

T: $d \times d$ i.e. square matrix with the dimensions of the unobserved state variable

R: $d \times d_\eta$ i.e. state dimension by state noise dimension

Z: $1 \times d$ i.e. row vector of state dimension

ok

Q1: Create the state space matrices $T_{[\mu]}$, $R_{[\mu]}$, and $Z_{[\mu]}$ corresponding to the trend component μ_t . We assume a local linear trend (that is, of order $k = 2$).

Hint: Use **2-dimensional** numpy.ndarrays of the correct sizes to represent all the matrices.

```
In [5]: T_mu = np.zeros(shape = (2,2))
R_mu = np.zeros(shape = (2,1))
Z_mu = np.zeros(shape = (1,2))

for i in range(2):
    if i == 0:
        for j in range(2):
            T_mu[i,j] = (-1)**(j+2) * math.factorial(2)/math.factorial(j+1)/math.factorial(2-j-1)
    else:
        for j in range(2):
            T_mu[i,i-1] = 1

R_mu[0,0] = 1
Z_mu[0,0] = 1
```

```
In [6]: print("T: \n", T_mu)
print("R: \n", R_mu)
print("Z: \n", Z_mu)
```

```
T:
[[ 2. -1.]
 [ 1.  0.]]
R:
[[1.]
 [0.]]
Z:
[[1.  0.]]
```

Q2: There is a yearly seasonal pattern present in the data. What should we set the periodicity s of the seasonal component to, to capture this pattern?

Hint: Count the average number of observations per (whole) year and round to the closest integer.

```
In [7]: print(data['Year'])

0      1993.011526
1      1993.038692
2      1993.065858
3      1993.093025
4      1993.120191
...
1078   2022.304626
1079   2022.331793
1080   2022.358959
1081   2022.386125
1082   2022.413291
Name: Year, Length: 1083, dtype: float64
```

```
In [8]: year = data['Year'].astype(str).str[0:4]
peryearcount = year.value_counts()

print(peryearcount)
```

```
1993    37
2017    37
2015    37
2018    37
2013    37
2012    37
2011    37
2010    37
2009    37
1994    37
2007    37
2019    37
2005    37
2004    37
2003    37
2001    37
2000    37
1999    37
1998    37
1997    37
2021    37
1995    37
2016    37
2020    36
2008    36
2014    36
2006    36
2002    36
1996    36
2022    16
Name: Year, dtype: int64
```

All years have 36-37 observations, but the latest year i.e. 2022 only has 16 observations so far. So we ignore this partial year to set the periodicity.

```
In [9]: s = round(peryearcount[:-1].mean())
print("Periodicity should be set to: ", s)
```

Periodicity should be set to: 37

dr

Q3: What is the *state dimension* of a seasonal component with periodicity s ? That is, how many states are needed in the corresponding state space representation?

A: For $s=37$, state dimension of γ_t is 36×36 i.e. $(s-1) \times (s-1)$

dr

Q4: Create the state space matrices $T_{[\gamma]}$, $R_{[\gamma]}$, and $Z_{[\gamma]}$ corresponding to the seasonal component γ_t .

Hint: Use **2-dimensional** numpy.ndarrays of the correct sizes to represent all the matrices.

```
In [10]: T_gamma = np.zeros(shape = (s-1,s-1))
R_gamma = np.zeros(shape = (s-1,1))
Z_gamma = np.zeros(shape = (1,s-1))
```

```

T_gamma[0,] = -1
for i in range(s-2):
    T_gamma[i+1,i] = 1

R_gamma[0,0] = 1
Z_gamma[0,0] = 1

```

```
In [11]: print("T: \n", T_gamma)
print("R: \n", R_gamma)
print("Z: \n", Z_gamma)
```

```

T:
[[ -1. -1. -1. ... -1. -1. -1.]
 [ 1.  0.  0. ... 0.  0.  0.]
 [ 0.  1.  0. ... 0.  0.  0.]
 ...
 [ 0.  0.  0. ... 0.  0.  0.]
 [ 0.  0.  0. ... 1.  0.  0.]
 [ 0.  0.  0. ... 0.  1.  0.]]
R:
[[1.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
Z:
[[1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]


```

Q5: Using the matrices that you have constructed above, create the state space matrices for the complete structural time series model. Print out the shapes of the resulting system matrices and check that they correspond to what you expect (cf Q0).

Hint: Use `scipy.linalg.block_diag` and `numpy.concatenate`.

```
In [12]: import scipy.linalg

T = scipy.linalg.block_diag(T_mu, T_gamma)
R = scipy.linalg.block_diag(R_mu, R_gamma)
Z = np.concatenate((Z_mu, Z_gamma), axis = 1)
```

```
In [13]: print("dim(T): ", T.shape)
print("dim(R):", R.shape)
print("dim(Z):", Z.shape)

dim(T): (38, 38)
dim(R): (38, 2)
dim(Z): (1, 38)
```

$d = 38$, $d_\eta = 2$, which matches with what we mentioned in Q0.

We also need to specify the variances of the process noise η_t and measurement noise ε_t . Below, we will estimate (two of) these variances from data, but for now we set them arbitrarily to get an initial model to work with.

```
In [14]: # Some arbitrary noise values for now
sigma_trend = 0.01
sigma_seas = 1
sigma_eps = 1
Q = np.array([[sigma_trend**2, 0.], [0., sigma_seas**2]]) # Process noise covariance matrix
```

Finally, to complete the model we need to specify the distribution of the initial state. This encodes our *a priori* belief about the actual values of the trend and seasonality, i.e., before observing any data.

Q6: Set up the mean vector of the initial state $a_1 = \mathbb{E}[\alpha_1]$ such that:

- The trend component starts at the first observation, $\mathbb{E}[\mu_1] = y_1$,
- The slope of the trend is *a priori* zero in expectation, $\mathbb{E}[\mu_1 - \mu_0] = 0$,
- The initial mean of all states related to the seasonal component are zero.

Also, create an initial state covariance matrix $P_1 = \text{Cov}(\alpha_1)$ as an identity matrix of the correct dimension, multiplied with a large value (say, 100) to represent our uncertainty about the initial state.

```
In [15]: k = 2  
dim_a1 = k+s-1  
a1 = np.zeros(shape = (dim_a1,1))  
a1[0,0] = y[0]  
a1[1,0] = y[0]  
  
P1 = np.identity(dim_a1) * 100
```

```
In [16]: a1
```

```
Out[16]: array([[-38.61],  
                [-38.61],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ],  
                [ 0. ]])
```

✓

We have now defined all the matrices etc. that make up the structural state space model. For convenience, we can create an object of the class `LGSS` available in the module `tssltools_lab2` as a container for these quantities.

```
In [17]: from tssltools_lab2 import LGSS  
model = LGSS(T, R, Q, Z, sigma_eps**2, a1, P1)  
  
help(model.get_params)
```

Help on method get_params in module tssltools_lab2:
get_params() method of tssltools_lab2.LGSS instance
 Return all model parameters.

T, R, Q, Z, H, a1, P1 = model.get_params()

2.2 Kalman filtering for the structural model

Now we have the data and a model available. Next, we will turn our attention to the inference problem, which is a central task when analysing time series data using the state space framework.

State inference is the problem of estimating the unknown (latent) state variables given the data. For the time being we assume that the *model parameters* are completely specified, according to above, and only consider how to estimate the states using the Kalman filter.

In the questions below we will treat the first $n = 800$ time steps as training data and the remaining m observations as validation data.

```
In [18]: n = 800  
m = ndata-n
```

Q7: Complete the Kalman filter implementation below. The function should be able to handle missing observations, which are encoded as "not a number", i.e. `y[t] = np.nan` for certain time steps `t`.

Hint: The Kalman filter involves a lot of matrix-matrix and matrix-vector multiplications. It turns out to be convenient to store sequences of vectors (such as the predicted and filtered state estimates) as `(d,1,n)` arrays, instead of `(d,n)` or `(n,d)` arrays. In this way the matrix multiplications will result in 2d-arrays of the correct shapes without having to use a lot of explicit `reshape`. However, clearly, this is just a matter of coding style preferences!

```
In [19]: from tssltools_lab2 import kfs_res  
  
def kalman_filter(y, model: LGSS):  
    """Kalman filter for LGSS model with one-dimensional observation.  
  
    :param y: (n,) array of observations. May contain nan, which encodes missing observations.  
    :param model: LGSS object with the model specification.  
  
    :return kfs_res: Container class with member variables,  
                    alpha_pred: (d,1,n) array of predicted state means.  
                    P_pred: (d,d,n) array of predicted state covariances.  
                    alpha_filt: (d,1,n) array of filtered state means.  
                    P_filt: (d,d,n) array of filtered state covariances.  
                    y_pred: (n,) array of means of p(y_t | y_{1:t-1})  
                    F_pred: (n,) array of variances of p(y_t | y_{1:t-1})
```

```

"""
n = len(y)
d = model.d # State dimension
alpha_pred = np.zeros((d, 1, n))
P_pred = np.zeros((d, d, n))
alpha_filt = np.zeros((d, 1, n))
P_filt = np.zeros((d, d, n))
y_pred = np.zeros(n)
F_pred = np.zeros(n)

T, R, Q, Z, H, a1, P1 = model.get_params() # Get all model parameters (for brevity)

for t in range(n):
    # Time update (predict)
    # ADD CODE HERE
    if t==0: #initialize state at first time step for alpha_hat_{1|0}
        alpha_pred[:, :, t] = a1
        P_pred[:, :, t] = P1
    else: #alpha_hat_{t|t-1} for all other time steps
        alpha_pred[:, :, t] = np.dot(T, alpha_filt[:, :, t-1])
        P_pred[:, :, t] = np.dot(T, np.dot(P_filt[:, :, t-1], np.transpose(T))) + np.dot(R, np.dot(Q, np.transpose(R)))

    # Compute prediction of current output
    # ADD CODE HERE
    y_pred[t] = np.dot(Z, alpha_pred[:, :, t]) #y_hat_{t|t-1}
    F_pred[t] = np.dot(Z, np.dot(P_pred[:, :, t], np.transpose(Z))) + H #F_{t|t-1}

    # Measurement update (correct)
    # ADD CODE HERE
    if np.isnan(y[t]): #for missing values skip to prediction updates
        alpha_filt[:, :, t] = alpha_pred[:, :, t]
        P_filt[:, :, t] = P_pred[:, :, t]

    else:
        K = np.dot(P_pred[:, :, t], np.transpose(Z)) / F_pred[t] #Kalman Gain calculation
        alpha_filt[:, :, t] = alpha_pred[:, :, t] + np.dot(K, (y[t]-y_pred[t])) #update filter mean alpha_hat_{t|t}
        P_filt[:, :, t] = np.dot((np.identity(d) - np.dot(K, Z)), P_pred[:, :, t]) #update filter variance P_{t|t}

kf = kfs_res(alpha_pred, P_pred, alpha_filt, P_filt, y_pred, F_pred)
return kf

```

In [20]: y

Out[20]: array([-38.61, -42.01, -41.97, ..., 49.98, 48.49, 48.37])

Q8: Use the Kalman filter to infer the states of the structural time series applied to the sealevel data. Run the filter on the training data (i.e., first $n = 800$ time steps), followed by a long-range prediction of y_t for the remaining time points.

Generate a plot which shows:

1. The data $y_{1:n+m}$
2. The one-step predictions $\hat{y}_{t|t-1} \pm 1$ standard deviation for the training data, i.e., $t = 1, \dots, n$,
3. The long-range predictions $\hat{y}_{t|n} \pm 1$ standard deviation for the validation data, i.e., $t = n+1, \dots, n+m$,
4. A vertical line indicating the switch between training and validation data, using `plt.axvline(x=u[n])`.

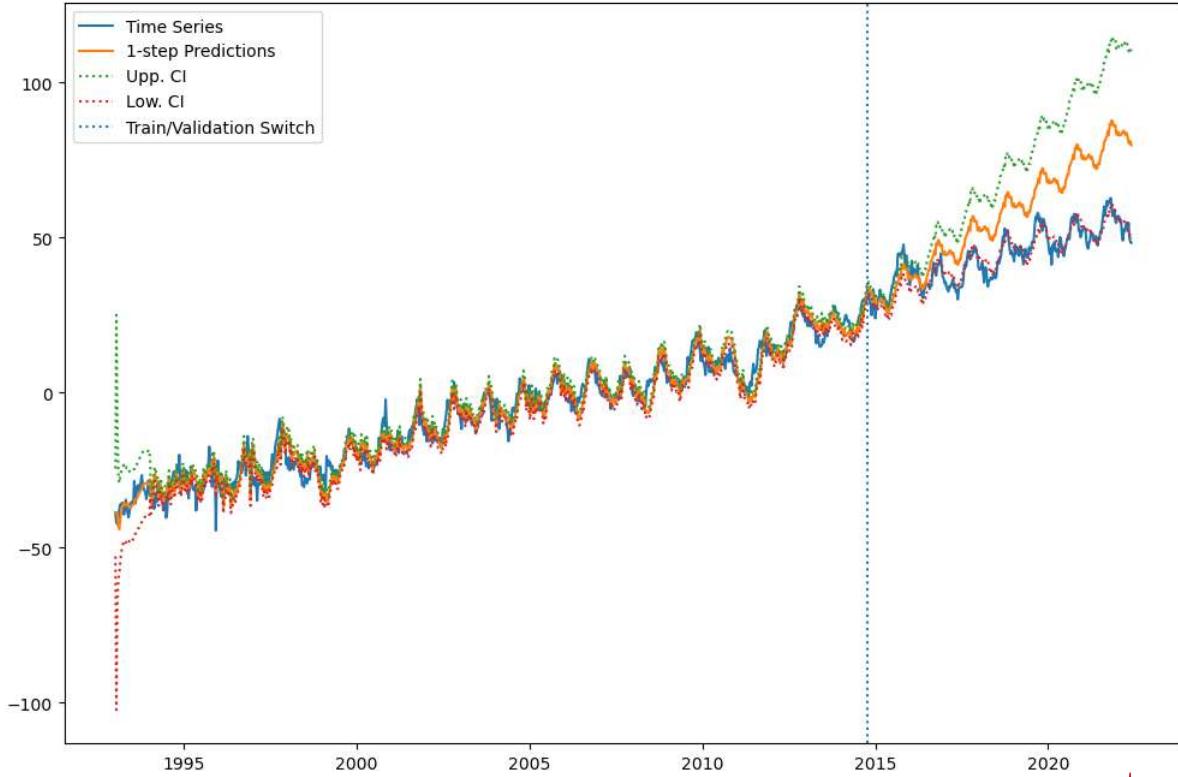
Hint: It is enough to call the `kalman_filter` function once. Make use of the missing data functionality!

```

In [21]: #apply kalman filter
y_train = y[:n]
y_valid = np.full(m, np.nan)
y_final = np.concatenate((y_train, y_valid))
KF = kalman_filter(y = y_final, model = model)

plt.plot(u, y, label = "Time Series")
plt.plot(u[:len(KF.y_pred)], KF.y_pred, label = "1-step Predictions")
plt.plot(u[:len(KF.y_pred)], KF.y_pred + np.sqrt(KF.F_pred), label = "Upp. CI", ls = ":")
plt.plot(u[:len(KF.y_pred)], KF.y_pred - np.sqrt(KF.F_pred), label = "Low. CI", ls = ":")
plt.axvline(x = u[n], ls = ":", label = "Train/Validation Switch")
plt.legend()
plt.show()

```



Q9: Based on the output of the Kalman filter, compute the training data log-likelihood $\log p(y_{1:n})$. dr

```
In [26]: # Log Likelihood of p(y | y_1:n)

def log_LL(n, y, y_hat, F):
    llhood = -0.5 * n * np.log(2*np.pi) - 0.5*np.sum(np.log(abs(F)) + np.transpose(y-y_hat)*(y-y_hat) / F)
    return(llhood)

# Log Likelihood of training data

LL = log_LL(n, y = y_train, y_hat = KF.y_pred[:n], F = KF.F_pred[:n])
print(LL)

-2986.425560280407
```

2.3 Identifying the noise variances using the EM algorithm

So far we have used fixed model parameters when running the filter. In this section we will see how the model parameters can be learnt from data using the EM algorithm. Specifically, we will try to learn the variance of the state noise affecting the seasonal component as well as the variance of the observation noise,

$$\theta = (\sigma_\gamma^2, \sigma_\varepsilon^2). \quad (4)$$

For brevity, the variance of the trend component σ_μ^2 is fixed to the value $\sigma_\mu^2 = 0.01^2$ as above. (See Appendix A below for an explanation.)

Recall that we consider $y_{1:n}$ as the training data, i.e., we will estimate θ using only the first $n = 800$ observations.

Q10: Which optimization problem is it that the EM algorithm is designed to solve? Complete the line below!

A:

$$\begin{aligned} \hat{\theta} &= \arg \max_{\theta} \mathbb{E} \left[\log p_{\theta}(\alpha_{1:n}, y_{1:n}) | y_{1:n}, \tilde{\theta} \right] \\ &= \arg \max_{\theta} \left(\text{const.} - \frac{1}{2} \sum_{t=1}^n \left[\log |\sigma_\varepsilon^2| + \log |Q| + \left\{ \hat{\epsilon}_{t|n}^\top + \text{Var}[\epsilon_t | y_{1:n}] \right\} \sigma_\varepsilon^{-2} + \text{tr} \left[\left\{ \hat{\eta}_{t|n} \hat{\eta}_{t|n}^\top + \text{Var}[\eta_t | y_{1:n}] \right\} Q^{-1} \right] \right] \right) \end{aligned}$$

where, $\hat{\epsilon}_{t|n}$, $\text{Var}[\epsilon_t | y_{1:n}]$, $\hat{\eta}_{t|n}$ and $\text{Var}[\eta_t | y_{1:n}]$ are the **smoothed** mean and variance of ϵ_t and η_t

Q11: Write down the updating equations on closed form for the M-step in the EM algorithm.

Hint: Look at Exercise Session 2

A:

$$\begin{aligned} \hat{\sigma}_\varepsilon^2 &= \frac{1}{n} \sum_{t=1}^n \left[\hat{\epsilon}_{t|n}^2 + \text{Var}[\epsilon_t | y_{1:n}] \right] \\ \hat{\sigma}_\gamma^2 &= \frac{1}{n} \sum_{t=1}^n \left[\hat{\eta}_{t|n} \hat{\eta}_{t|n}^\top + \text{Var}[\eta_t | y_{1:n}] \right] \end{aligned}$$

dr

To implement the EM algorithm we need to solve a *smoothing problem*. The Kalman filter that we implemented above is based only on a forward propagation of information. The *smoother* complements the forward filter with a backward pass to compute refined state estimates. Specifically, the smoothed state estimates comprise the mean and covariances of

$$p(\alpha_t | y_{1:n}), \quad t = 1, \dots, n \quad (5)$$

Furthermore, when implementing the EM algorithm it is convenient to work with the (closely related) smoothed estimates of the disturbances, i.e., the state and measurement noise,

$$\begin{aligned} p(\eta_t | y_{1:n}), & \quad t=1, \dots, n-1 \\ p(\varepsilon_t | y_{1:n}), & \quad t=1, \dots, n \end{aligned} \quad (6) \quad (7)$$

An implementation of a state and disturbance smoother is available in the `tsslttools_lab2` module. You may use this when implementing the EM algorithm below.

```
In [28]: from tsslttools_lab2 import kalman_smoother
help(kalman_smoother)

Help on function kalman_smoother in module tsslttools_lab2:

kalman_smoother(y, model=tsslttools_lab2.LGSS, kf=tsslttools_lab2.kfs_res)
    Kalman (state and disturbance) smoother for LGSS model with one-dimensional observation.

    :param y: (n,) array of observations. May contain nan, which encodes missing observations.
    :param model: LGSS object with the model specification.
    :param kf: kfs_res object with result from a Kalman filter forward pass.

    :return kfs_res: Container class. The original Kalman filter result is augmented with the following member variables,
        alpha_sm: (d,1,n) array of smoothed state means.
        V: (d,d,n) array of smoothed state covariances.
        eps_hat: (n,) array of smoothed means of observation disturbances.
        eps_var: (n,) array of smoothed variances of observation disturbances.
        eta_hat: (deta,1,n) array of smoothed means of state disturbances.
        eta_cov: (deta,deta,n) array of smoothed covariances of state disturbances.
```

```
In [37]: KS_test = kalman_smoother(y = y_final, model = model, kf = KF)

print(KS_test.eta_hat[:, :, 1])
[[ -9.45266338e-05]
 [ 2.32131823e-02]]
```

Q12: Implement an EM algorithm by completing the code below. Run the algorithm for 100 iterations and plot the traces of the parameter estimates, i.e., the values θ_r , for $r = 0, \dots, 100$.

Note: When running the Kalman filter as part of the EM loop you should only filter the *training data* (i.e. excluding the prediction for validation data).

```
In [55]: num_iter = 100

#define estimation variables
sigma_eps_hat = 1
sigma_seas_hat = 1

theta_tilde = np.zeros((num_iter, 2))
theta_tilde[0] = [sigma_eps_hat, sigma_seas_hat]

for r in range(1,num_iter):
    # E-step
    Q_tilde = np.array([[sigma_trend**2, 0], [0, sigma_seas_hat**2]])
    model_EM = LGSS(T, R, Q_tilde, Z, sigma_eps_hat**2, a1, P1)

    KF_EM = kalman_filter(y = y_train, model = model_EM)
    KS_EM = kalman_smoother(y = y_train, model = model_EM, kf = KF_EM)

    N = len(KS_EM.eps_hat)

    # M-step
    sigma_eps_hat = 1/N * np.sum(KS_EM.eps_hat**2 + KS_EM.eps_var)
    theta_tilde[r,0] = np.sqrt(sigma_eps_hat)

    data = KS_EM.eta_hat.shape[0]

    sigma_seas_est = 0

    for i in range(N):
        eta_smooth = KS_EM.eta_hat[:, :, i].dot(KS_EM.eta_hat[:, :, i].T) + KS_EM.eta_cov[:, :, i]
        sigma_seas_est = sigma_seas_est + eta_smooth[data-1, data-1]
        sigma_seas_hat = 1/N * sigma_seas_est

    theta_tilde[r,1] = np.sqrt(sigma_seas_hat)

    sigma_eps_hat = theta_tilde[r,0]
    sigma_seas_hat = theta_tilde[r,1]
```



```
In [58]: print("Est. Sigma_eps_hat: ", sigma_eps_hat)
print("Est. Sigma_seasonal_hat: ", sigma_seas_hat)

Est. Sigma_eps_hat: 2.89248866356765
Est. Sigma_seasonal_hat: 0.19558559811678197
```

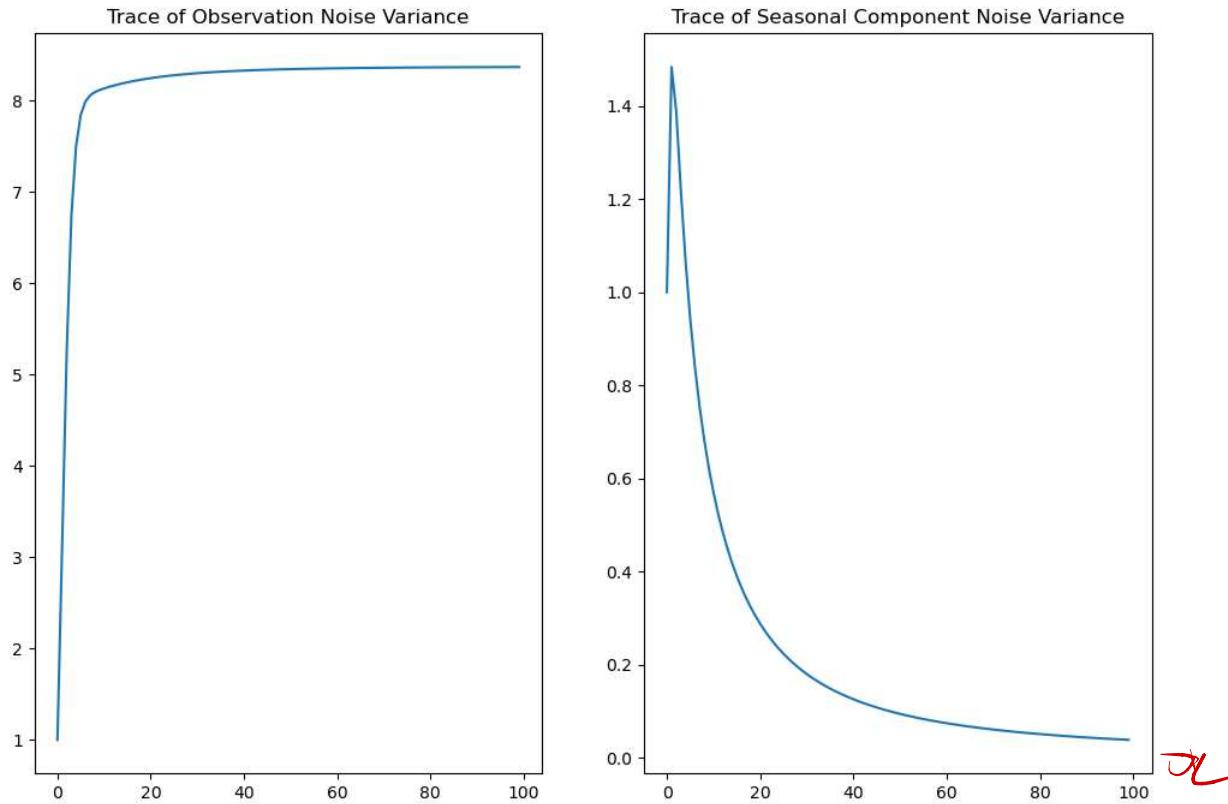
```
In [70]: fig, ax = plt.subplots(1,2)

ax[0].plot(theta_tilde[:,0]**2)
ax[0].set(title = "Trace of Observation Noise Variance")

ax[1].plot(theta_tilde[:,1]**2)
ax[1].set(title = "Trace of Seasonal Component Noise Variance")

[Text(0.5, 1.0, 'Trace of Seasonal Component Noise Variance')]
```

Out[70]:



2.4 Further analysing the data

We will now fix the model according to the final output from the EM algorithm and further analyse the data using this model.

Q13: Rerun the Kalman filter to compute a *long range prediction* for the validation data points, analogously to **Q8** (you can copy-paste code from that question). That is, generate a plot which shows:

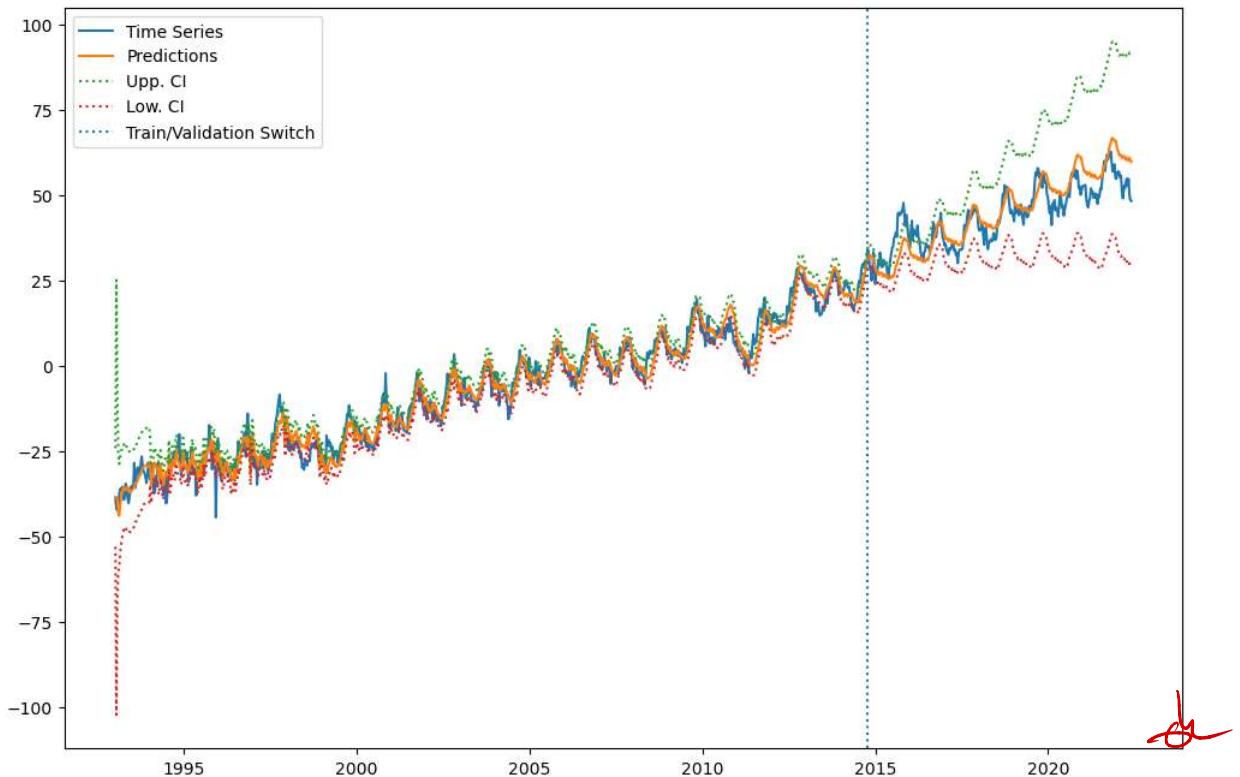
1. The data $y_{1:n+m}$
2. The one-step predictions $\hat{y}_{t|t-1} \pm 1$ standard deviation for the training data, i.e., $t = 1, \dots, n$,
3. The long-range predictions $\hat{y}_{t|n} \pm 1$ standard deviation for the validation data, i.e., $t = n+1, \dots, n+m$,
4. A vertical line indicating the switch between training and validation data, using `plt.axvline(x=u[n])`.

Furthermore, compute the training data log-likelihood $\log p(y_{1:n})$ using the estimated model (cf. **Q9**).

```
In [71]: # final output of EM optimization stored into Q
Q_final = np.array([[sigma_trend**2, 0., 0., sigma_seas_hat**2]])
# get the final model
model_final = LGSS(T, R, Q_final, Z, sigma_eps_hat**2, a1, P1)

#rerun Kalman filter on new model
KF_final = kalman_filter(y = y_final, model = model_final)

In [125... # plot
plt.plot(u, y, label = "Time Series")
plt.plot(u[:len(KF_final.y_pred)], KF_final.y_pred, label = "Predictions")
plt.plot(u[:len(KF_final.y_pred)], KF_final.y_pred + np.sqrt(KF_final.F_pred), label = "Upp. CI", ls = ":")
plt.plot(u[:len(KF_final.y_pred)], KF_final.y_pred - np.sqrt(KF_final.F_pred), label = "Low. CI", ls = ":")
plt.axvline(x = u[n], ls = ":", label = "Train/Validation Switch")
plt.legend()
plt.show()
```



```
In [87]: LL_train = log_LL(n, y = y_train, y_hat = KF_final.y_pred[:n], F = KF_final.F_pred[:n])
print("Training log Likelihood: ", LL_train)

LL_valid = log_LL(m, y = y[n:], y_hat = KF_final.y_pred[n:], F = KF_final.F_pred[n:])
print("Validation log Likelihood: ", LL_valid)

Training log Likelihood: -2147.333421985849
Validation log Likelihood: -1031.149588389554
```

Note that we can view the model for the data y_t as being comprised of an underlying "signal", $s_t = \mu_t + \gamma_t$ plus observation noise ε_t

$$y_t = s_t + \varepsilon_t \quad (8)$$

We can obtain refined, *smoothed*, estimates of this signal by conditioning on all the training data $y_{1:n}$.

Q14: Run a Kalman smoother to compute smoothed estimates of the signal, $\mathbb{E}[s_t|y_{1:n}]$, conditionally on all the *training data*. Then, similarly to above, plot the following:

1. The data $y_{1:n+m}$
2. The smoothed estimates $\mathbb{E}[s_t|y_{1:n}] \pm 1$ standard deviation for the training data, i.e., $t = 1, \dots, n$,
3. The predictions $\mathbb{E}[s_t|y_{1:n}] \pm 1$ standard deviation for the validation data, i.e., $t = n+1, \dots, n+m$,
4. A vertical line indicating the switch between training and validation data, using `plt.axvline(x=u[n])`.

Hint: Express s_t in terms of α_t . Based on this expression, compute the smoothed mean and variance of s_t based on the smoothed mean and covariance of α_t .

```
In [97]: KS_final = kalman_smoother(y = y_final, model = model_final, kf = KF_final)

signal = np.zeros(n+m)
signal_var = np.zeros(n+m)

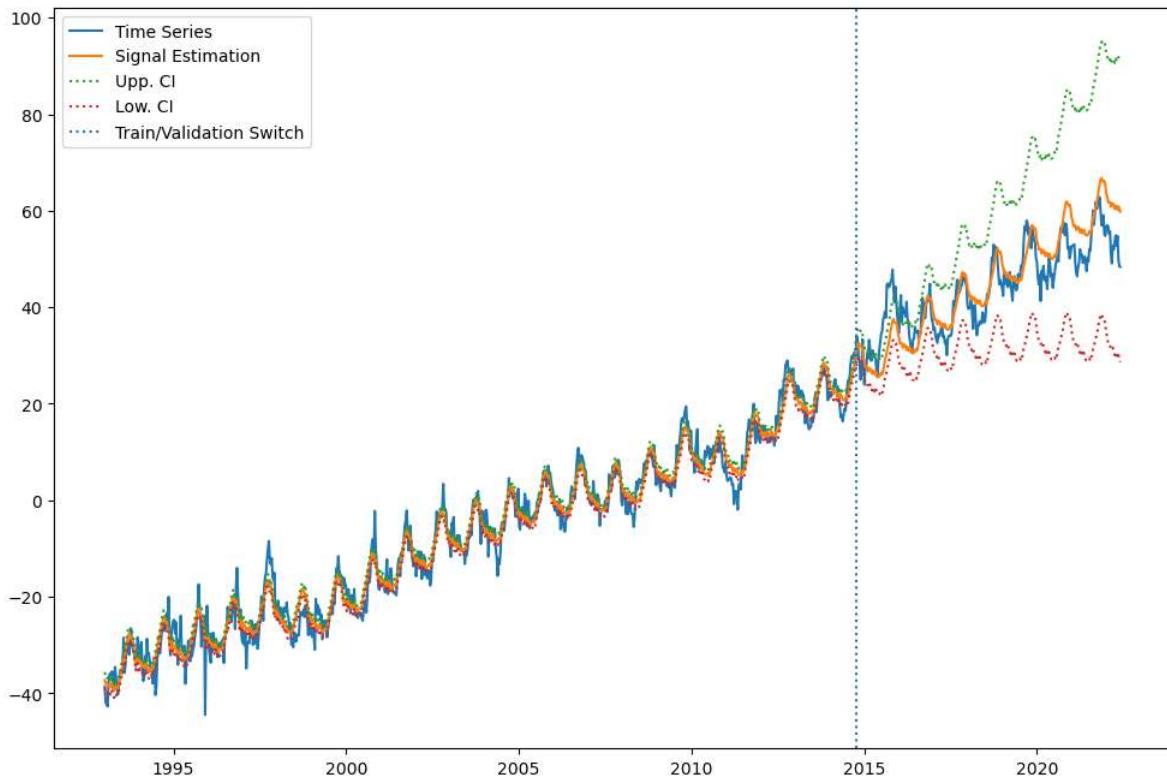
for i in range(n+m):
    signal[i] = np.dot( model_final.Z , KS_final.alpha_sm[:, :, i] )
    signal_var[i] = np.dot( model_final.Z , np.dot( KS_final.V[:, :, i] , model_final.Z.T ) ) + KS_final.sigmas_var[i]
```

should not be here
for y_t until

```
In [98]: plt.plot(u, y, label = "Time Series")
plt.plot(u, signal, label = "Signal Estimation")
plt.plot(u, signal + np.sqrt(signal_var), label = "Upp. CI", ls = ":")

plt.plot(u, signal - np.sqrt(signal_var), label = "Low. CI", ls = ":")

plt.axvline(x = u[n], ls = ":", label = "Train/Validation Switch")
plt.legend()
plt.show()
```



Q15: Explain, using a few sentences, the qualitative differences (or similarities) between the Kalman filter predictions plotted in **Q13** and the smoothed signal estimates plotted in **Q14** for,

1. Training data points, $t \leq n$
2. Validation data points, $t > n$

A: Because of smoothing we were able to observe a better pattern in the quality of prediction in the start of the time series. The CIs for the training predictions also were observed to be much tighter as compared to the predictions from the Kalman Filter. But, on the validation data, both the filter and smoother gave a very similar looking output. Neither was able to predict much better compared to the other, and the CIs also were similarly distanced.

We can shed additional light on the properties of the process under study by further decomposing the signal into its trend and seasonal components.

Q16: Using the results of the state smoother, compute and plot the *smoothed estimates* of the two signal components, i.e.:

1. Trend: $\hat{\mu}_{t|n} = \mathbb{E}[\mu_t | y_{1:n}]$ for $t = 1, \dots, n$
2. Seasonal: $\hat{\gamma}_{t|n} = \mathbb{E}[\gamma_t | y_{1:n}]$ for $t = 1, \dots, n$

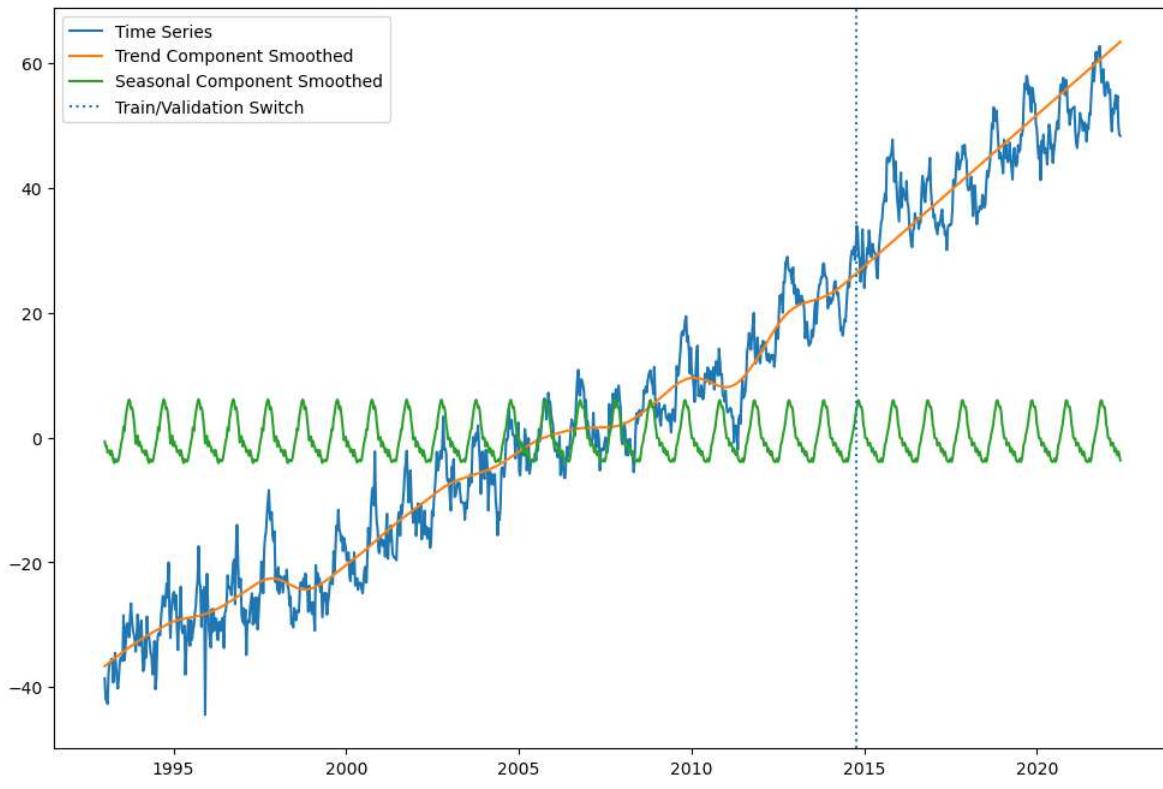
(You don't have to include confidence intervals here if don't want to, for brevity.)

```
In [101]: mu_smooth = np.zeros(n+m)
gamma_smooth = np.zeros(n+m)

mu_smooth[0] = np.dot( Z_mu , KS_final.alpha_sm[:, :, 0] )
gamma_smooth[0] = np.dot(Z_gamma , KS_final.alpha_sm[k :, :, 0] )

for i in range(n+m-1):
    mu_smooth[i+1] = np.dot( Z_mu , KS_final.alpha_sm[:, :, i+1] )
    gamma_smooth[i+1] = np.dot(Z_gamma , KS_final.alpha_sm[k :, :, i+1] )

plt.plot(u, y, label = "Time Series")
plt.plot(u, mu_smooth, label = "Trend Component Smoothed")
plt.plot(u, gamma_smooth, label = "Seasonal Component Smoothed")
plt.axvline(x = u[n], ls = ":" , label = "Train/Validation Switch")
plt.legend()
plt.show()
```



2.5 Missing data

We conclude this section by illustrating one of the key merits of the state space approach to time series analysis, namely the simplicity of handling missing data. To this end we will assume that a chunk of observations in the middle of the training data is missing.

Q17: Let the values y_t for $300 < t \leq 400$ be missing. Modify the data and rerun the Kalman filter and smoother. Plot,

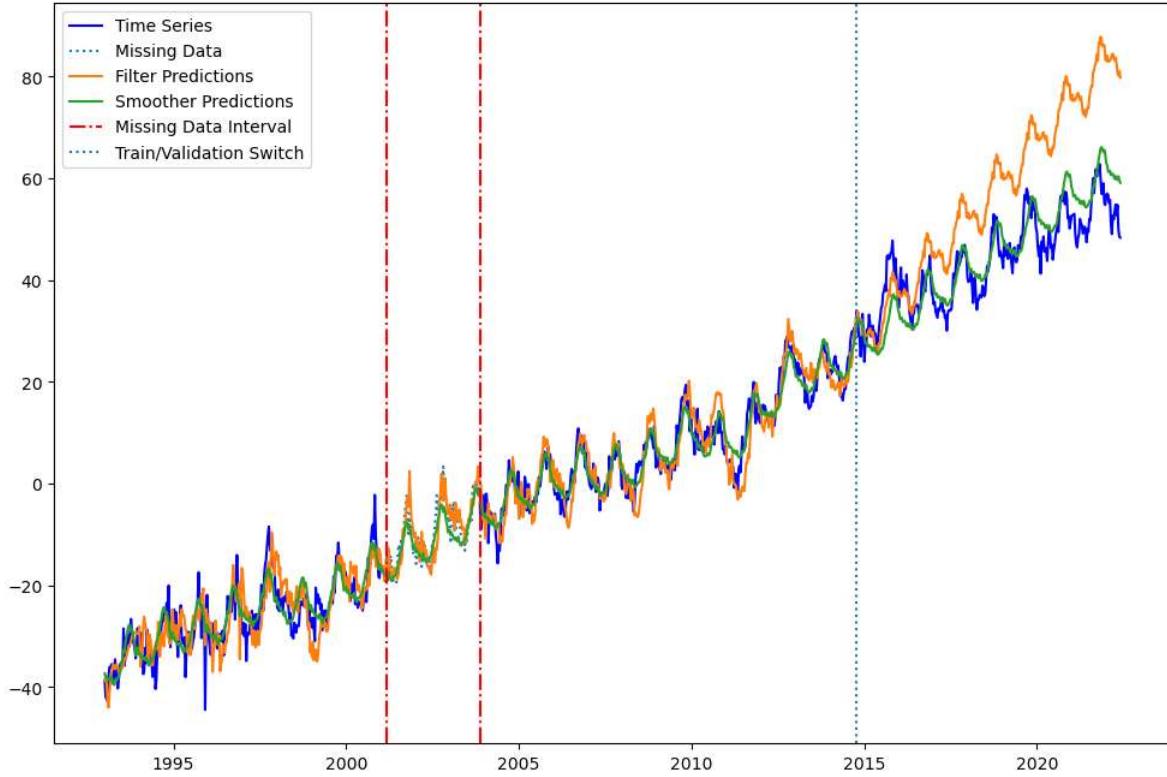
1. The Kalman filter predictions, analogously to **Q8**
2. The Kalman smoother predictions, analogously to **Q13**

Comment on the qualitative differences between the filter and smoother estimates and explain what you see (in a couple of sentences).

```
In [116]: y_missing = np.concatenate((y[:300], np.full(100, np.nan), y[400:n], y_valid))
KF_missing = kalman_filter(y = y_missing, model = model_final)
KS_missing = kalman_smoothen(y = y_missing, model = model_final, kf = KF_missing)
smoother_pred = np.zeros(n+m)
for i in range(n+m):
    smoother_pred[i] = np.dot(model_final.Z, KS_missing.alpha_sm[:, :, i])
```

```
In [129]: plt.plot(u[:300], y[:300], label = "Time Series", color = "blue")
plt.plot(u[300:400], y[300:400], label = "Missing Data", ls = ":")

plt.plot(u[400:], y[400:], color = "blue")
plt.plot(u, KF_y_pred, label = "Filter Predictions")
plt.plot(u, smoother_pred, label = "Smoothen Predictions")
plt.axvline(x = u[300], ls = "-.", label = "Missing Data Interval", color = "red")
plt.axvline(x = u[400], ls = "-.", color = "red")
plt.axvline(x = u[n], ls = ":" , label = "Train/Validation Switch")
plt.legend()
plt.show()
```



A: Re running the kalman filter on the new data with missing entries resulted in losing the goodness of the EM optimization and we see very inaccurate predictions for the validation data. However, for the missing data itself, we observed that the filter predictions were much closer to the actual data as compared to the predictions from the smoother.

ok

Appendix A. Why didn't we learn the trend noise variance as well?

In the assignment above we have fixed σ_μ to a small value. Conceptually it would have been straightforward to learn also this parameter with the EM algorithm. However, unfortunately, the maximum likelihood estimate of σ_μ often ends up being too large to result in accurate *long term predictions*. The reason for this issue is that the structural model

$$y_t = \mu_t + \gamma_t + \varepsilon_t \quad (9)$$

is not a perfect description of reality. As a consequence, when learning the parameters the mismatch between the model and the data is compensated for by increasing the noise variances. This results in a trend component which does not only capture the long term trends of the data, but also seemingly random variations due to a model misspecification, possibly resulting in poor *long range predictions*.

Kitagawa (Introduction to Time Series Modeling, CRC Press, 2010, Section 12.3) discusses this issue and proposes two solutions. The first is a simple and pragmatic one: simply fix σ_μ^2 to a value smaller than the maximum likelihood estimate. This is the approach we have taken in this assignment. The issue is of course that in practice it is hard to know what value to pick, which boil down to manual trial and error (or, if you are lucky, the designer of the lab assignment will tell you which value to use!).

The second, more principled, solution proposed by Kitagawa is to augment the model with a stationary AR component as well. That is, we model

$$y_t = \mu_t + \gamma_t + \nu_t + \varepsilon_t \quad (10)$$

where $\nu_t \sim \text{AR}(p)$. By doing so, the stationary AR component can compensate for the discrepancies between the original structural model and the "true data generating process". It is straightforward to include this new component in the state space representation (how?) and to run the Kalman filter and smoother on the resulting model. Indeed, this is one of the beauties with working with the state space representation of time series data! However, the M-step of the EM algorithm becomes a bit more involved if we want to use the method to estimate also the AR coefficients of the ν -component, which is beyond the scope of this lab assignment.