

# tssl\_lab2

October 14, 2023

## 1 TSSL Lab 2 - Structural model, Kalman filtering and EM

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 latent 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.

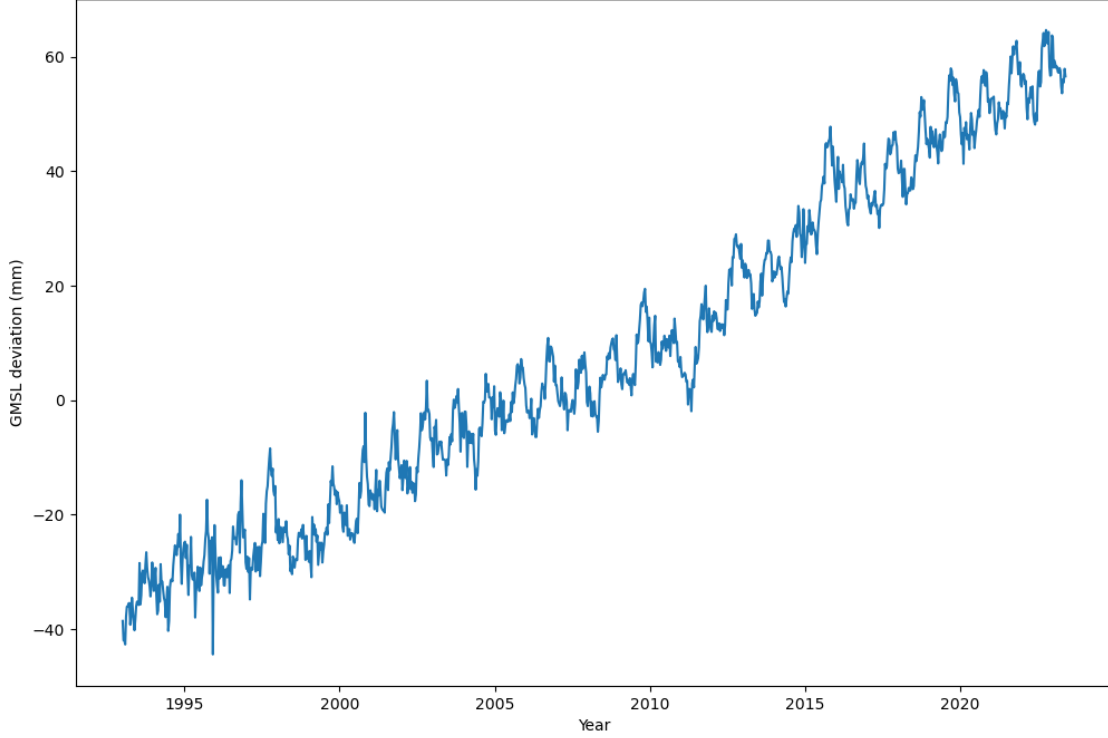
```
[1]: import pandas # Loading data / handling data frames
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import block_diag, inv
plt.rcParams["figure.figsize"] = (12,8) # Increase default size of plots
```

### 1.1 2.1 Setting up a structural state space model

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

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

```
[99]: 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 analyse 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  $\mu_t$  is a trend component,  $\gamma_t$  is a seasonal component, and  $\varepsilon_t$  is an observation noise. The model is expressed using a state space representation,

$$\alpha_{t+1} = T\alpha_t + R\eta_t, \quad \eta_t \sim N(0, Q), \quad (2)$$

$$y_t = Z\alpha_t + \varepsilon_t, \quad \varepsilon_t \sim N(0, \sigma_\varepsilon^2). \quad (3)$$

**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 dimensions of the matrices  $T$ ,  $R$ , and  $Z$  of the state space model?

**A:**

**R dimension corrected**

If  $\alpha_t$  dimension is  $(d, 1)$ , then  $\alpha_{t+1}$  has to be of the same dimensions, because of these  $T$  dimension has to be  $(d, d)$  and  $R$  dimension is  $(d, d_{\eta})$ . Because  $Q$  is of dimension  $(d_{\eta} \times d_{\eta})$ .  $y_t$  is a scalar, so the dimension of  $Z$  is  $(1, d)$

**Q1:** Create the state space matrices  $T_{[\mu]}$ ,  $R_{[\mu]}$ , and  $Z_{[\mu]}$  corresponding to the trend component  $\mu_t$ . 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.

```
[100]: T_mu = np.array([[2,-1], [1, 0]]) # 2x2
R_mu = np.array([[1],[0]])# 2x1
Z_mu = np.array([[1,0]]) # 1x2
print("Shapes:", "\n", T_mu.shape, R_mu.shape, Z_mu.shape)
print("Matrices", "\n", T_mu, "\n", R_mu, "\n", Z_mu)
```

Shapes:

(2, 2) (2, 1) (1, 2)

Matrices

```
[[ 2 -1]
 [ 1  0]]
[[1]
 [0]]
[[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.

```
[125]: import math
from datetime import datetime, timedelta
#Function to change decimal years to YYYY-MM-DD format

def decimal_year_to_dt(decimal):
    year = int(decimal)
    base = datetime(year, 1, 1)
    remainder = decimal - year
    result = base + timedelta(seconds=(base.replace(year=base.year + 1) - base).
    ↪total_seconds() * remainder)
    return result

dates = data['Year'].apply(decimal_year_to_dt)

year= dates.dt.year.unique() # All years with measurements

complete_years= np.zeros(len(year)) # Array to store complete years
w=0
for y2 in year:
    if len(dates[dates.dt.year==y2].dt.month.unique())==12 : # If a year have
    ↪measurements taken from the 12 month it is a complete year
        complete_years[w] = y2
    else:
        complete_years[w] = 0
```

```

w =w+1

whole_years=np.count_nonzero(complete_years) # Numbers of complete years

S = math.ceil(len(dates[dates.dt.year!=2023])/whole_years) # Divide the total
↳ number of measurements by the amount of complete years
print("Periodicity S to capture the pattern:",S)

```

Periodicity S to capture the pattern: 37

**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?

```

[126]: State_dim= math.floor(S-1)
print("Number of states needed:",State_dim)

```

Number of states needed: 36

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

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

```

[127]: T_gamma = np.empty(shape=(S-1,S-1))
T_gamma[0,:] = -1
T_gamma[1:,35] =0
T_gamma[1:,:35] = np.identity(35)
R_gamma = np.zeros(shape=(S-1,1))
R_gamma[0,0]= 1
Z_gamma = np.zeros(shape=(1,S-1))
Z_gamma[0,0] =1
print("T_gamma:",T_gamma.shape,"R_gamma:", R_gamma.shape, "Z_gamma:",Z_gamma.
↳ shape)

```

T\_gamma: (36, 36) R\_gamma: (36, 1) Z\_gamma: (1, 36)

**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`.

```

[128]: from scipy.linalg import block_diag
T = block_diag(T_mu,T_gamma)
R = block_diag(R_mu,R_gamma)
Z = np.concatenate((Z_mu,Z_gamma),axis=1)

print(T.shape, R.shape, Z.shape)

```

(38, 38) (38, 2) (1, 38)

We also need to specify the variances of the process noise  $\eta_t$  and measurement noise  $\varepsilon_t$ . 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.

```
[129]: # 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
Q.shape
```

```
[129]: (2, 2)
```

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.

```
[137]: a1 = np.zeros(shape=(1,38)) # K=2+Seasonal_dimension=36=38
a1[0,1] = y[0]
a1[0,0] = y[0]

e_mu1_mu0 = a1[0,1] - a1[0,0]
P1 = np.identity(38)*100 # identity matrix of 38x38
P1.shape
e_mu1_mu0 # Condition fulfill
```

```
[137]: 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.

```
[138]: 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()
```

## 1.2 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.

```
[139]: n = 800
       m = ndata-n
```

```
[122]: np.isnan(y).any() # Check if there is any nan values in the data set
```

```
[122]: False
```

**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.  $y[t] = \text{np.nan}$  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!

```
[150]: 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 38(S) in this case
#print(d)
alpha_pred = np.zeros((d, 1, n)) # predict at - Time update (predict)
P_pred = np.zeros((d, d, n)) # predict at - Time update (predict)

alpha_filt = np.zeros((d, 1, n)) # update filter - Measurement update
↪(correct)
P_filt = np.zeros((d, d, n)) # update filter - Measurement update
↪(correct)

y_pred = np.zeros(n) # predict yt - Compute prediction of current output
F_pred = np.zeros(n) # predict Ft - Compute prediction of current output
↪Variance

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

for t in range(n):
    # Time update (predict)
    if t == 0: # use for first iteration
        alpha_pred[:,0,t] = a1
        P_pred[:, :, t] = P1
    else:
        alpha_pred[:,0,t] = T@alpha_filt[:,0,t-1]
        P_pred[:, :, t] = (T@P_filt[:, :, t-1]@T.T) + (R@Q@R.T)

    # Compute prediction of current output
    y_pred[t] = Z@alpha_pred[:,0,t]
    F_pred[t] = Z@P_pred[:, :, t]@Z.T + H # H is the variance of the error

    k = P_pred[:, :, t]@Z.T*(1/F_pred[t]) # Kalman filter gain
    # Measurement update (correct)
    if not np.isnan(y[t]):
        alpha_filt[:,0,t] = alpha_pred[:,0,t] + np.
↪squeeze(k*(y[t]-y_pred[t])) # use squeeze to pass from (38,1,1)
        P_filt[:, :, t] = (np.identity(d)-k@Z)@P_pred[:, :, t]
    else:
        alpha_filt[:,0,t] = alpha_pred[:,0,t]
        P_filt[:, :, t] = P_pred[:, :, t]

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

```

**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!

```
[151]: # y array. To make predictions the first 800 values of the training are from
        our y original array, and then from 801 to 1119 this are nan values.
y_kalman = np.zeros(len(y))
y_kalman[:800] = y[:800]
y_kalman[800:] = np.nan

result=kalman_filter(y_kalman,model)

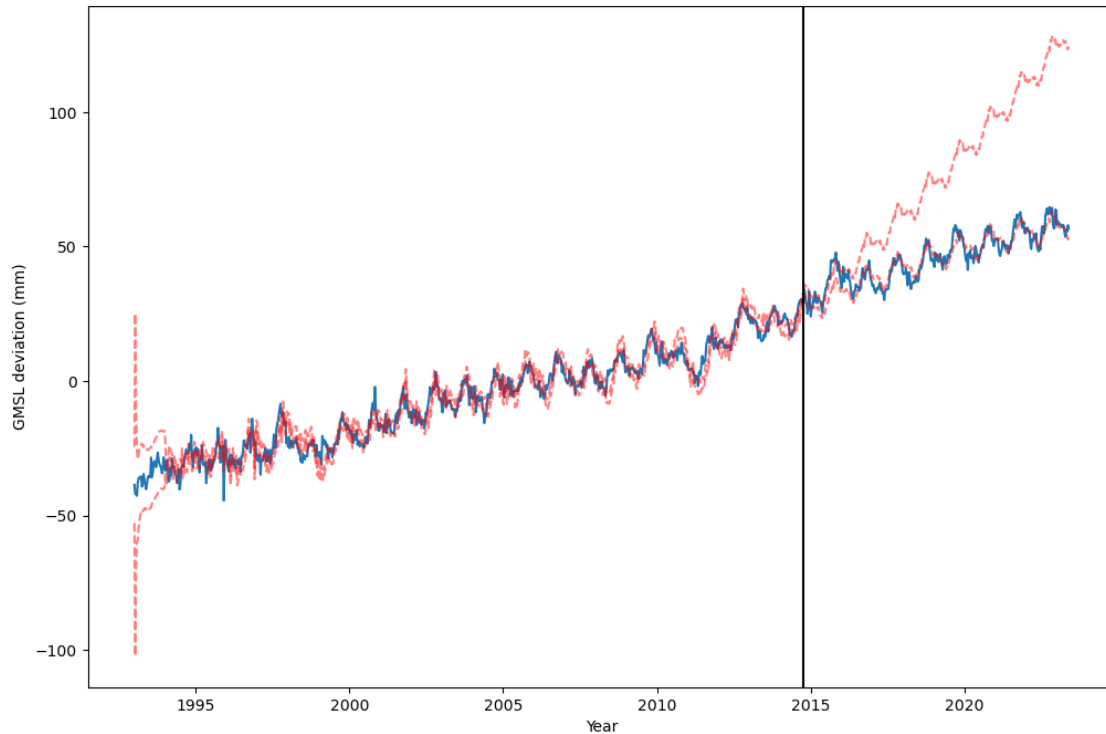
y_sdp_t = result.y_pred[:800] - np.sqrt(result.F_pred[:800])
y_sdn_t = result.y_pred[:800] + np.sqrt(result.F_pred[:800])

y_sdp_v = result.y_pred[800:] - np.sqrt(result.F_pred[800:])
y_sdn_v = result.y_pred[800:] + np.sqrt(result.F_pred[800:])

my = (result.y_pred)
plt.plot(u,y)
#plt.plot(u,result.y_pred)
plt.plot(u[:800], y_sdp_t, alpha=0.5, linestyle="--", color= "red")
plt.plot(u[:800], y_sdn_t, alpha=0.5,linestyle="--" ,color= "red")
plt.plot(u[800:], y_sdp_v, alpha=0.5, linestyle="--" ,color= "red")
plt.plot(u[800:], y_sdn_v, alpha=0.5, linestyle="--" ,color= "red")
plt.axvline(x=u[n], color="black")

plt.xlabel('Year')
plt.ylabel('GMSL deviation (mm)')
plt.show()
```





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

```
[152]: def log_likelihood(yt,kf_out):

    F_pred = kf_out.F_pred
    y_pred = kf_out.y_pred
    n = len(yt)
    log_likelihood = 0
    for t in range(len(yt)):
        log_likelihood = log_likelihood + ((0.5)*(F_pred[t] + ((yt[t]-y_pred[t]).
↪T*(1/F_pred[t])*(yt[t]-y_pred[t]))))
        log_likelihood = -(0.5*2*np.log(2*np.pi)) - log_likelihood
    return log_likelihood
```

```
[153]: log_likelihood(y_kalman[:800], result)
```

```
[153]: -8011.205462800837
```

### 1.3 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  $\sigma_\mu^2$  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  $\theta$  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:**  $\hat{\theta} = \arg \max_{\theta} Q(\theta, \tilde{\theta})$  where  $Q(\theta, \tilde{\theta}) = E[\log p_{\theta}(\alpha_{1:n}, y_{1:n}) | y_{1:n}, \tilde{\theta}]$

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

*Hint: Look at Exercise Session 2*

**A:** Here is the closed form of the equations in the M- Step

$$\sigma_\varepsilon^2 = \frac{1}{n} \sum_{t=1}^n [\hat{\varepsilon}_{t|n}^2 + \text{Var}(\varepsilon_t | y_{1:n})]$$

$$Q^* = \frac{1}{n} \sum_{t=1}^n [\hat{\eta}_{t|n} \hat{\eta}_{t|n}^T + \text{Var}(\eta_t | y_{1:n})]$$

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,

$$p(\eta_t | y_{1:n}), \quad t = 1, \dots, n-1 \quad (6)$$

$$p(\varepsilon_t | y_{1:n}), \quad t = 1, \dots, n \quad (7)$$

An implementation of a state and disturbance smoother is available in the `tssltools_lab2` module. You may use this when implementing the EM algorithm below.

```
[154]: from tssltools_lab2 import kalman_smoother
      help(kalman_smoother)
```

Help on function kalman\_smoother in module tssltools\_lab2:

```
kalman_smoother(y, model: tssltools_lab2.LGSS, kf: tssltools_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.

```

**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  $\theta_r$ , 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).

```

[155]: num_iter = 100

# Lists to store parameter estimates for plotting
sigma_seasonal_noise = np.zeros(num_iter)
sigma_obs_noise = np.zeros(num_iter)
var_obs_noise = 0
Qcopy = Q.copy()
sigma_obs_noise[0] = sigma_eps # We set the first values as sigmas we set above
sigma_seasonal_noise[0] = sigma_seas

ytrain = y[:800]
for r in range(1, num_iter):
    # E-step
    model1 = LGSS(T, R, Qcopy, Z, sigma_obs_noise[r-1]**2, a1, P1) #Model
    res = kalman_filter(ytrain, model1) #Kalman Filter
    res_sm = kalman_smoother(ytrain, model1, res) #Kalman Smother

    # M-step
    sigma_obs_noise[r] = np.sqrt(np.mean((res_sm.eps_hat**2 + res_sm.eps_var)))
    # Epsilon

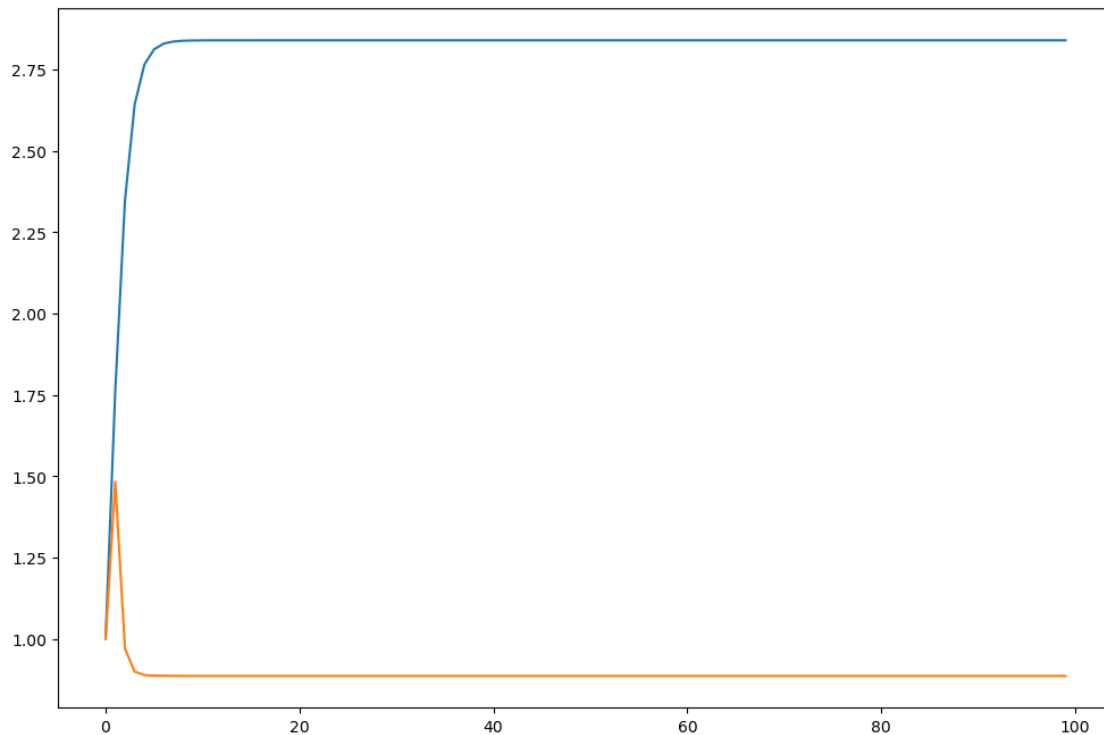
    sigma_seasonal_noise[r] = np.mean(res_sm.eta_hat[1, :, :] * res_sm.eta_hat[1, :, :
    ] + res_sm.eta_cov[1, 1, :]) # Q

```

```
[156]: print(sigma_obs_noise.shape, sigma_seasonal_noise.shape , res_sm.eta_hat.shape,
↳ res_sm.eta_hat.T.shape, Qcopy.shape, res_sm.eta_cov.shape,
      res_sm.alpha_sm.shape, res_sm.V.shape)
res_sm.eta_hat[:, :, 1]
#sum_eta_hat.shape
plt.plot(range(0,100), sigma_obs_noise)
plt.plot(range(0,100), sigma_seasonal_noise)
plt.show
```

```
(100,) (100,) (2, 1, 800) (800, 1, 2) (2, 2) (2, 2, 800) (38, 1, 800) (38, 38,
800)
```

```
[156]: <function matplotlib.pyplot.show(close=None, block=None)>
```



## 1.4 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**).

```
[157]: # y array. To make predictions the first 800 values of the training are from
        ↪ our y original array, and then from 801 to 1119 this are nan values.
y_kalman = np.zeros(len(y))
y_kalman[:800] = y[:800]
y_kalman[800:] = np.nan

result=kalman_filter(y_kalman,model)

y_sdp_t = result.y_pred[:800] - np.sqrt(result.F_pred[:800])
y_sdn_t = result.y_pred[:800] + np.sqrt(result.F_pred[:800])

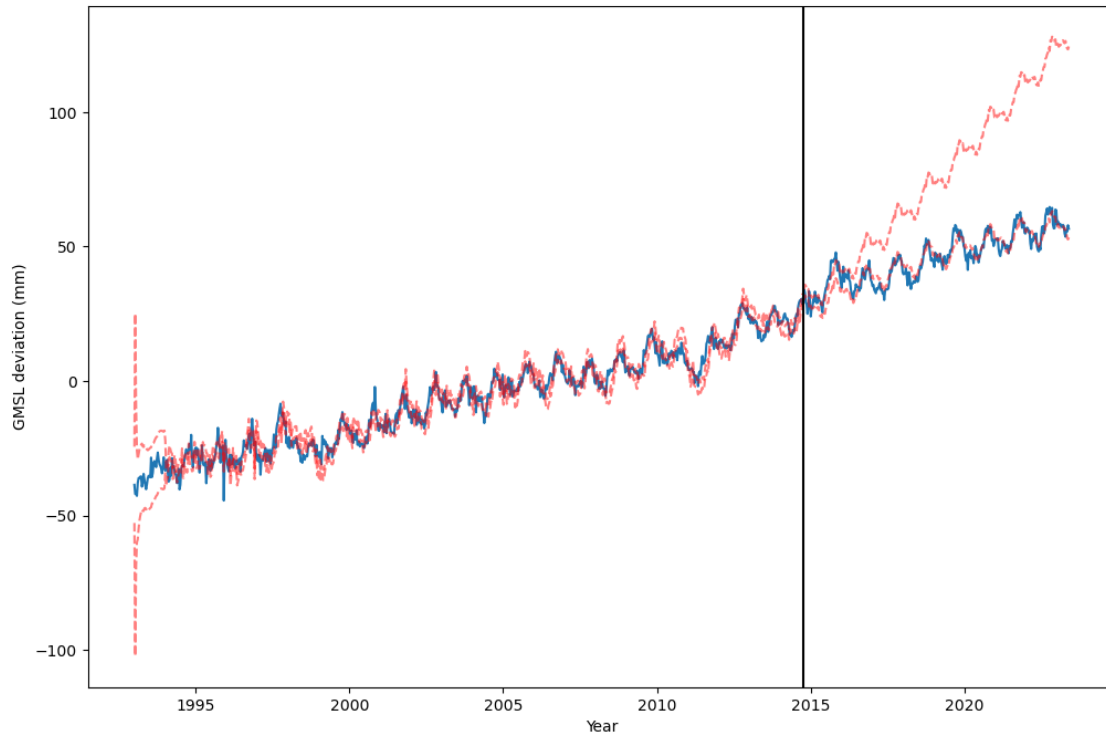
y_sdp_v = result.y_pred[800:] - np.sqrt(result.F_pred[800:])
y_sdn_v = result.y_pred[800:] + np.sqrt(result.F_pred[800:])

print("Loglikelihood:",log_likelihood(y_kalman[:800], result))
my = (result.y_pred)

plt.plot(u,y)
plt.plot(u[:800], y_sdp_t, alpha=0.5, linestyle="--", color= "red")
plt.plot(u[:800], y_sdn_t, alpha=0.5,linestyle="--" ,color= "red")
plt.plot(u[800:], y_sdp_v, alpha=0.5, linestyle="--" ,color= "red")
plt.plot(u[800:], y_sdn_v, alpha=0.5, linestyle="--" ,color= "red")
plt.axvline(x=u[n], color="black")

plt.xlabel('Year')
plt.ylabel('GMSL deviation (mm)')
plt.show()
```

Loglikelihood: -8011.205462800837



```
[158]: log_likelihood(y_kalman[:800], result)
```

```
[158]: -8011.205462800837
```

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  $\varepsilon_t$

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

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  $\alpha_t$ . Based on this expression, compute the smoothed mean and variance of  $s_t$  based on the smoothed mean and covariance of  $\alpha_t$ .

```
[159]: res_sm = kalman_smoother(y_kalman, model, result) #Kalman Smother

e_st_y1n = np.zeros(len(y_kalman))
```

```

v_st_y1n = np.zeros(len(y_kalman))
for q in range(0,len(y_kalman)):
    e_st_y1n[q] = Z@res_sm.alpha_sm[:, :, q]
    v_st_y1n[q] = Z@res_sm.V[:, :, q]@Z.T

y_kspt = e_st_y1n[:800] - np.sqrt(v_st_y1n[:800])
y_ksnt = e_st_y1n[:800] + np.sqrt(v_st_y1n[:800])

y_kspv = e_st_y1n[800:] - np.sqrt(v_st_y1n[800:])
y_ksnv = e_st_y1n[800:] + np.sqrt(v_st_y1n[800:])

```

```

[160]: plt.plot(u,y) # 1 data training and validation
plt.plot(u[:800], y_kspt,alpha=0.5 ,color="red") # 2
plt.plot(u[:800], y_ksnt, alpha=0.5 ,color="red") # 2
plt.plot(u[800:], y_kspv,alpha=0.5 ,color="red") # 3
plt.plot(u[800:], y_ksnv, alpha=0.5 ,color="red") # 3

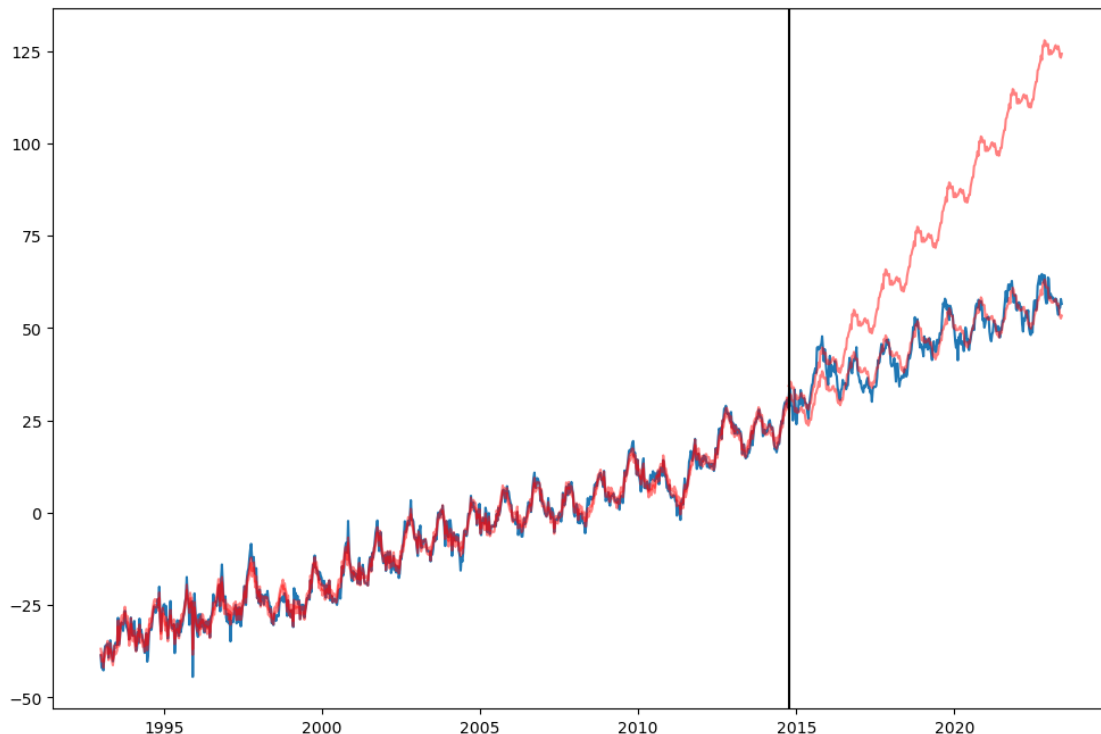
plt.axvline(x=u[n], color="black") # 4 Boundary Training and Validation data
plt.show

```

```

[160]: <function matplotlib.pyplot.show(close=None, block=None)>

```



**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:** 1. The predictions from the Kalman filter follows the data but with at lag. Meanwhile, the predictions of the Kalman smoother appears to follow better the trend and seasonality of the data but has peaks smaller than the ones on the Kalman filter.

2. Both predictions split apart from the true trajectory of the data. The kalman filter predictions have the seasonality pattern but it is of smaller magnitude and visually appears to became smaller. In contrast, the kalman smoother predictions have a higher magnitude and it can be seen in the peaks of data, which are greater and it maintain the magnitude, as the trend appears to maintain the size and shape.

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.:

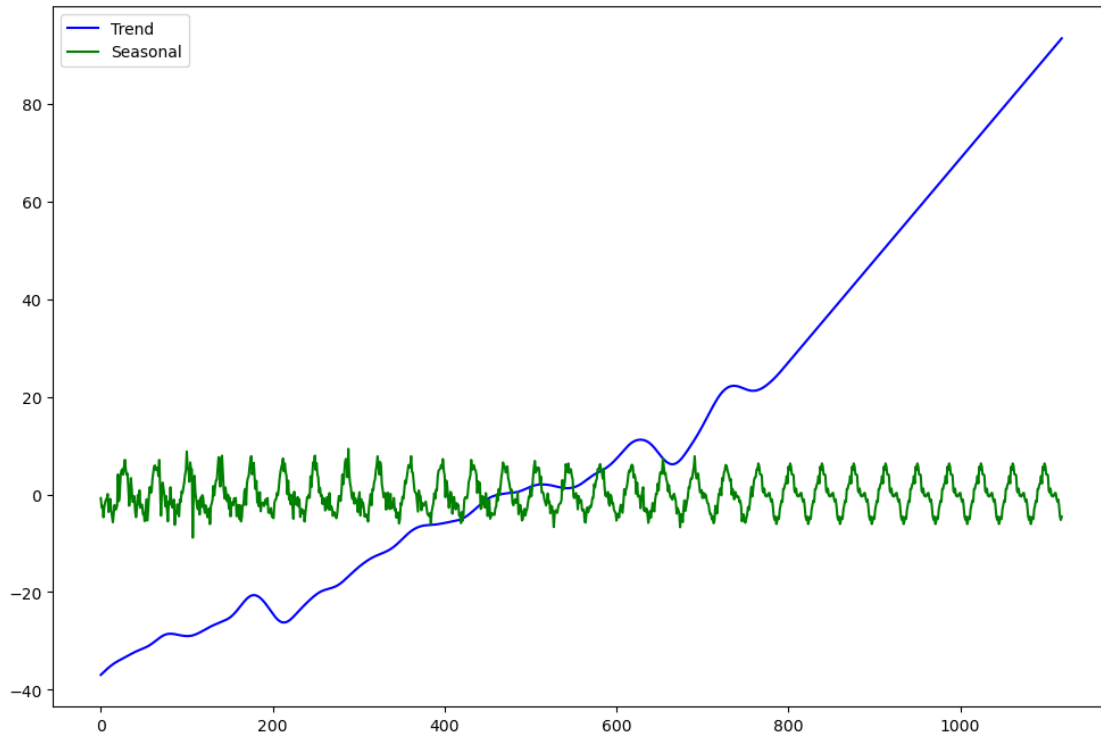
1. Trend:  $\hat{\mu}_{t|n} = \mathbb{E}[\mu_t | y_{1:n}]$  for  $t = 1, \dots, n$
2. Seasonal:  $\hat{\gamma}_{t|n} = \mathbb{E}[\gamma_t | y_{1:n}]$  for  $t = 1, \dots, n$

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

```
[161]: e_mu_y1n = np.zeros(len(y_kalman))
e_ga_y1n = np.zeros(len(y_kalman))
for w in range(0, len(y_kalman)):
    e_mu_y1n[w] = Z_mu @ res_sm.alpha_sm[:2, :, w] # Trend component
    e_ga_y1n[w] = Z_gamma @ res_sm.alpha_sm[2: :, :, w] # Seasonal component

plt.plot(range(0, len(y_kalman)), e_mu_y1n, color="blue", label= "Trend")
plt.plot(range(0, len(y_kalman)), e_ga_y1n, color= "green", label= "Seasonal")
plt.legend()
plt.show()
```





## 1.5 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).

```
[163]: # Y vector preparation
y_missing = np.zeros(len(ytrain))
y_missing[:300] = ytrain[:300]
y_missing[301:401] = np.zeros(100)
y_missing[401:] = ytrain[401:800]

# Kalman Filter and Kalman Smoother computation
result5_kf = kalman_filter(y_missing, model)
result5_kf_sm = kalman_smoother(y_missing, model, result5_kf)
# Standard deviations
ym_sdp_t_kf = result5_kf.y_pred - np.sqrt(result5_kf.F_pred)
```

```

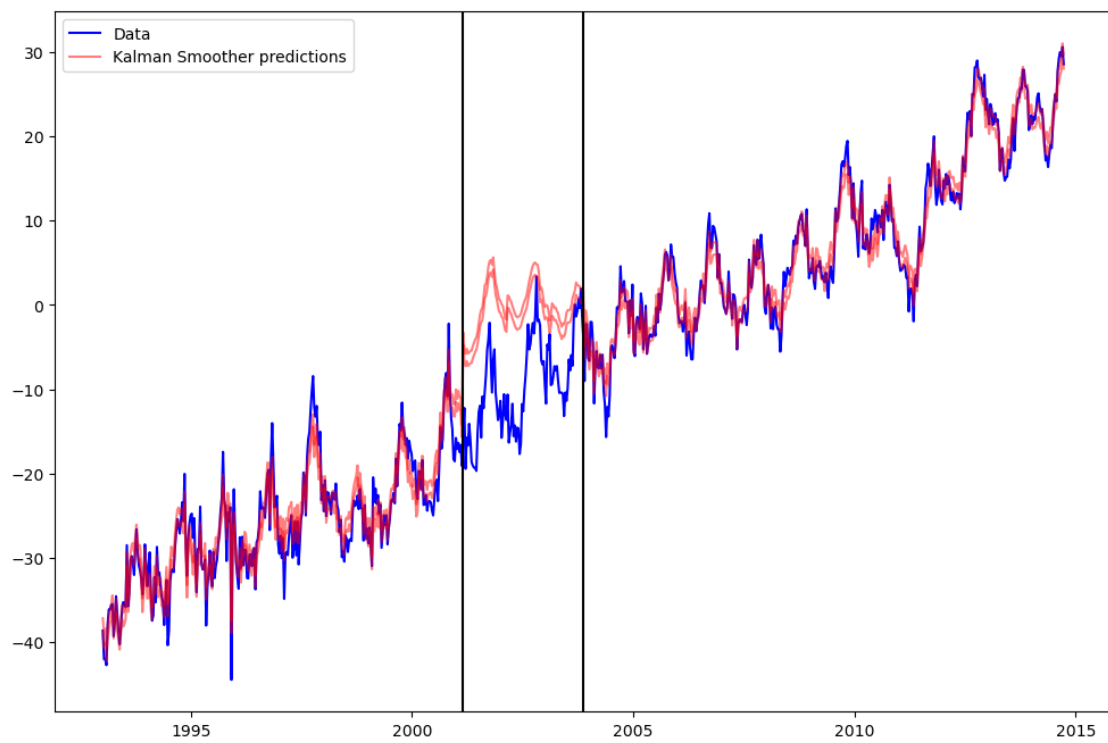
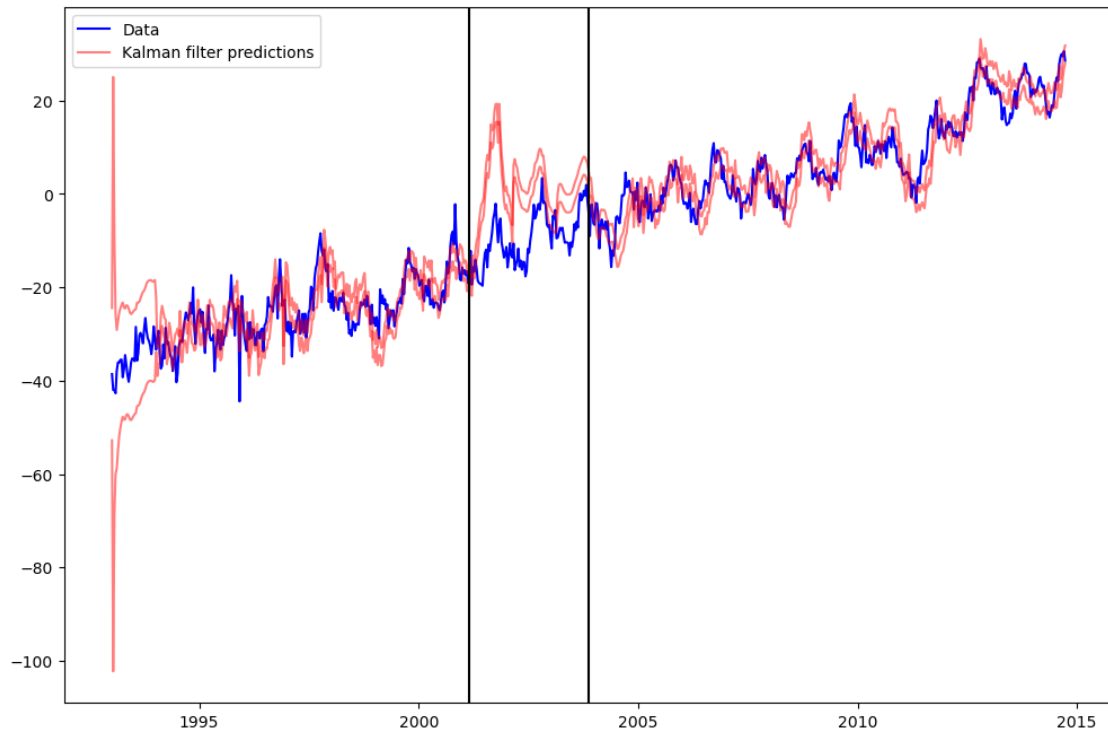
ym_sdn_t_kf = result5_kf.y_pred + np.sqrt(result5_kf.F_pred)

# Computation as in Q14
e5_st_y1n = np.zeros(len(ytrain))
v5_st_y1n = np.zeros(len(ytrain))
for l in range(0, len(ytrain)):
    e5_st_y1n[l] = Z@result5_kf_sm.alpha_sm[:, :, l]
    v5_st_y1n[l] = Z@result5_kf_sm.V[:, :, l]@Z.T
# Standard deviations
ym_kspt = e5_st_y1n - np.sqrt(v5_st_y1n)
ym_ksnt = e5_st_y1n + np.sqrt(v5_st_y1n)

# Kalman filter
plt.plot(u[:800], ytrain, color="blue", label="Data")
plt.plot(u[:800], ym_sdp_t_kf, alpha=0.5, color="red", label="Kalman filter_
↳ predictions")
plt.plot(u[:800], ym_sdn_t_kf, alpha=0.5, color="red")
plt.axvline(x=u[300], color="black")
plt.axvline(x=u[400], color="black")
plt.legend()
plt.show()

# Kalman Smoother
plt.plot(u[:800], ytrain, color="blue", label="Data")
plt.plot(u[:800], ym_kspt, alpha=0.5, color="red", label="Kalman Smoother_
↳ predictions")
plt.plot(u[:800], ym_ksnt, alpha=0.5, color="red")
plt.axvline(x=u[300], color="black")
plt.axvline(x=u[400], color="black")
plt.legend()
plt.show()

```



**A:**

Kalman Filter predictions on missing data follow the pattern of the original data but it has a lag. And this predictions are of greater magnitude.

Kalman Smoother predictions on missing data are of greater magnitude than the predictions of the Kalman Filter. The predictions shape appear to be shrinked but maintain the pattern of the original data. It look almost like a curve line and the original data has a two peaks in form of the letter M. It could be that as his name implies, the predictions are smoothed so that is why the values do not have the fall so pronounce after the peaks as in the original data.

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

In the assignment above we have fixed  $\sigma_\mu$  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  $\sigma_\mu$  often ends up being too large to result in accurate *long term predictions*. The reason for this issue is that the structural model

$$y_t = \mu_t + \gamma_t + \varepsilon_t \tag{9}$$

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  $\sigma_\mu^2$  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 build 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

$$y_t = \mu_t + \gamma_t + \nu_t + \varepsilon_t \tag{10}$$

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  $\nu$ -component, which is beyond the scope of this lab assignment.