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/ (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
```

2.1 Setting up a structural state space model

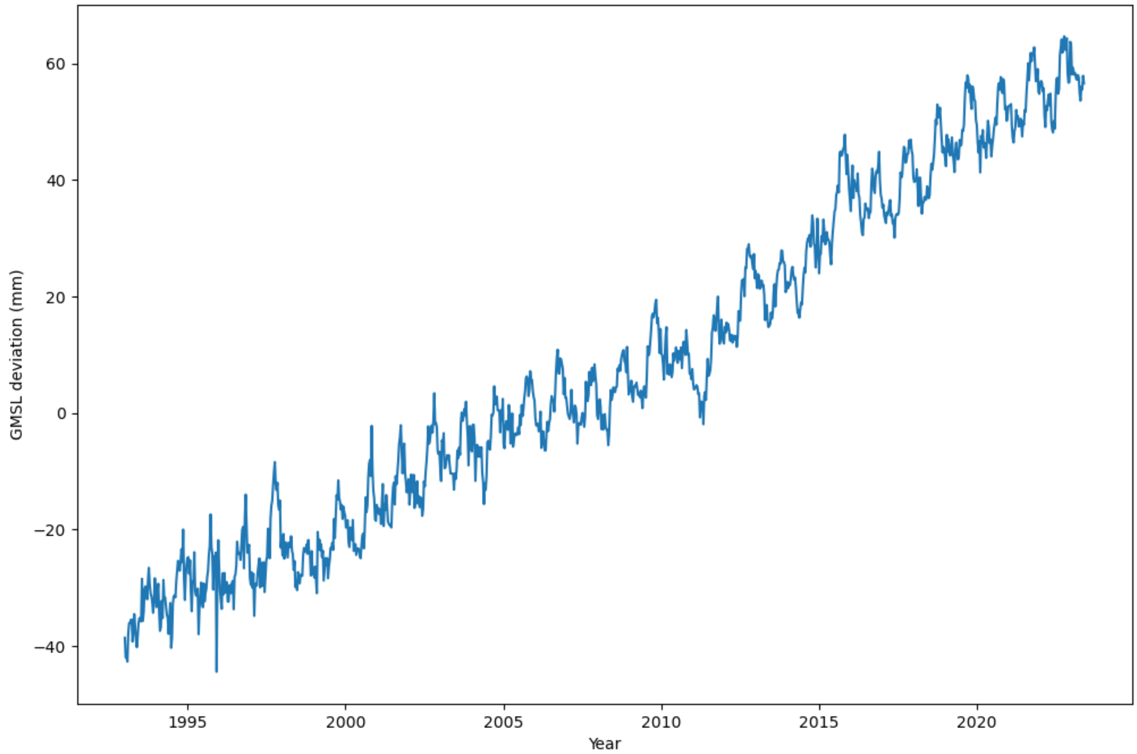
We start by loading and plotting data to reming ourselves what it looks like.

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

$$y_t = \mu_t + \gamma_t + \varepsilon_t$$

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,

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

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: dim(T) = $d * d$, dim(R) = $d * d_\eta$, dim(Z) = $1 * d$.

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.ndarray` s of the correct sizes to represent all the matrices.

In [4]:

```
T_mu = np.matrix([[2,-1],[1,0]])
R_mu = np.matrix([[1],[0]])
Z_mu = np.matrix([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 [5]:

```
np.round(u[u<2023].shape[0]/(np.floor(u[-1])-np.floor(u[0])))
```

Out[5]:

```
37.0
```

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: the state dimension of a seasonal component with periodicity s should be s - 1 = 36.

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.ndarray` s of the correct sizes to represent all the matrices.

In [6]:

```
T_gamma = np.eye(36,k=-1)
T_gamma[0,:]=-1
R_gamma = np.zeros((36,1))
R_gamma[0,0]=1
Z_gamma = np.zeros((1,36))
Z_gamma[0,0]=1
```

In [7]:

```
T_gamma
```

Out[7]:

```
array([[-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.]])
```

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 [8]:

```
import scipy
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),1)
```

In [9]:

```
print(T)

print(R)

print(Z)
```

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

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 [10]:

```
# 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 ma
```

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

In [11]:

```
a1=np.zeros((38,1))
a1[0:2,0]=[y[0],y[0]]

P1=np.eye(38)*100

print(P1)
print(a1)
```

```
[[100.   0.   0. ...   0.   0.   0.]
 [  0. 100.   0. ...   0.   0.   0.]
 [  0.   0. 100. ...   0.   0.   0.]
 ...
 [  0.   0.   0. ... 100.   0.   0.]
 [  0.   0.   0. ...   0. 100.   0.]
 [  0.   0.   0. ...   0.   0. 100.]]
[[-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.  ]
 [  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 [12]:

```
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 [13]:

```
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 convient 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 [14]:

```
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 observa
    :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:
            alpha_pred[:,:,t]=a1
            P_pred[:,:,t]=P1
        else:
            alpha_pred[:,:,t]=np.dot(T,alpha_filt[:,:,(t-1)])
            P_pred[:,:,t]=np.dot(np.dot(T,P_filt[:,:,(t-1)]),T.T)+np.dot(np.dot(R,Q),R.T

        # Compute prediction of current output
        # ADD CODE HERE
        y_pred[t]=np.dot(Z,alpha_pred[:,:,t])
        F_pred[t]=np.dot(np.dot(Z,P_pred[:,:,t]),(Z.T))+H

        # Measurement update (correct)
        # ADD CODE HERE
        if np.isnan(y[t]):
            alpha_filt[:,:,t]=alpha_pred[:,:,t].copy()
            P_filt[:,:,t]=P_pred[:,:,t].copy()
        else:
            K=np.dot(np.dot(P_pred[:,:,t],Z.T),pow(F_pred[t],-1))
            alpha_filt[:,:,t]=alpha_pred[:,:,t]+(K*(y[t]-y_pred[t]))
            P_filt[:,:,t]=np.dot((np.eye(d)-np.dot(K,Z)),P_pred[:,:,t])


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

```
    return kf
```

In [ ]:

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.

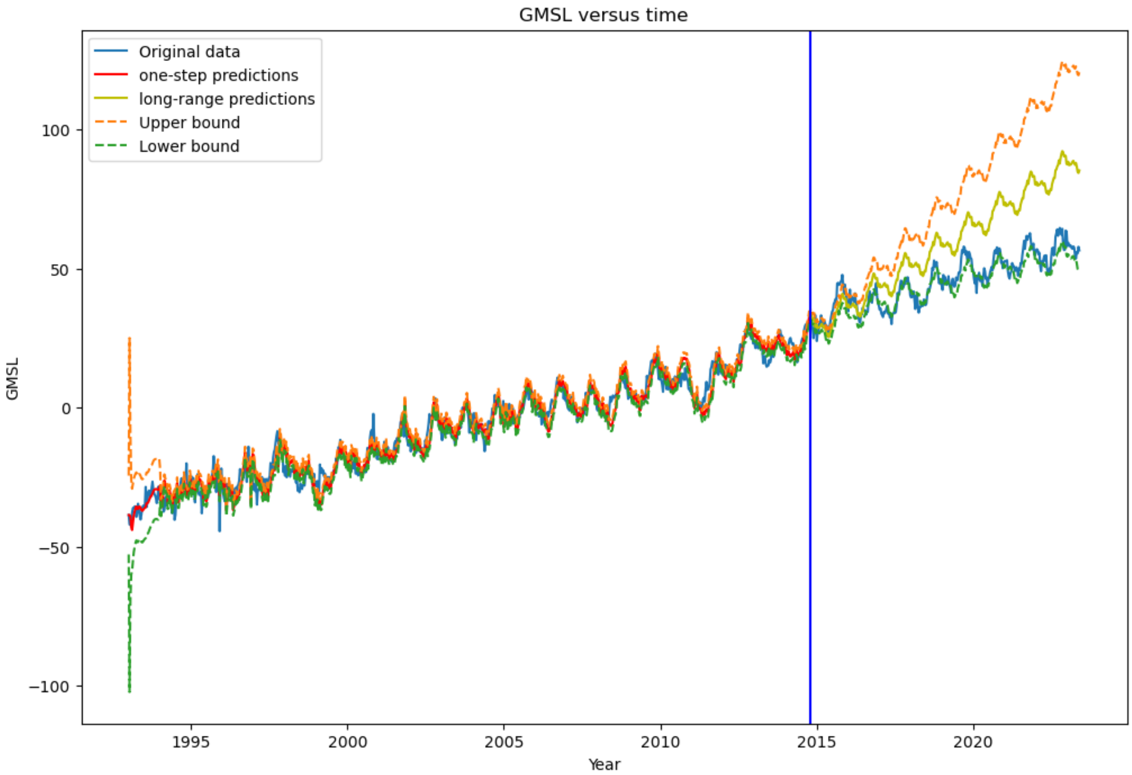
Generate a plot which shows:

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., $t = 1, \ldots, n$,
3. The long-range predictions $\hat{y}_{t|n} \pm 1$ standard deviation for the validation data, i.e., $t = n+1, \ldots, n+m$,
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 [15]:

```
train_data=y.copy()
train_data[(n+1):]=np.nan
kf=kalman_filter(train_data, model)

# Plot
plt.plot(u,y,label='Original data')
plt.plot(u[:801],kf.y_pred[:801],label='one-step predictions',c='r')
plt.plot(u[801:],kf.y_pred[801:],label='long-range predictions',c='y')
plt.plot(u,(kf.y_pred+np.sqrt(kf.F_pred)),linestyle='--',label='Upper bound')
plt.plot(u,(kf.y_pred-np.sqrt(kf.F_pred)),linestyle='--',label='Lower bound')
plt.axvline(x=u[801],c='b')
plt.title('GMSL versus time')
plt.xlabel('Year')
plt.ylabel('GMSL')
plt.legend()
plt.show()
```



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

A9:

$$p(y_{1:n}) = p(y_1)p(y_2|y_1)p(y_3|y_{1:2})\dots p(y_n|y_{1:n-1})$$

where in one-step process $p(y_t|y_{1:t-1}) \sim N(y_t|\hat{y}_{t|t-1}, F_{t|t-1})$

hence we can compute the training data log-likelihood $\log p(y_{1:n})$ via compute the logarithm of it's normal distribution's pdf.

In [16]:

```
F_pred=kf.F_pred
y_pred=kf.y_pred

#print(F_pred.shape)
#print(y_pred.shape)

loglik=0
for i in range(n):
    loglik=loglik - (((y[i]-y_pred[i])**2)/(2*F_pred[i])) - np.log(F_pred[i])/2 - np.log

print(loglik)
```

```
-2986.42556028042
```

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,

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

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: $\hat{\theta} = \arg\max_\theta E[log(p_\theta(\alpha_{1:n}, y_{1:n})|y_{1:n}, \tilde{\theta})]$

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

Hint: Look at Exercise Session 2

A: Compute the the partial derivative of $E[log(p_\theta(\alpha_{1:n}, y_{1:n})|y_{1:n}, \tilde{\theta})]$ with respect to σ_ϵ^2 and Q and set it to 0 :

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

,

and

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

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

$$p(\alpha_t \mid y_{1:n}), \qquad t = 1, \dots, n$$

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,

$$\begin{aligned} &p(\eta_t \mid y_{1:n}), \qquad t = 1, \dots, n-1 \\ &p(\varepsilon_t \mid y_{1:n}), \qquad t = 1, \dots, n \end{aligned}$$

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.

In [17]:

```
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-dimens
ional observation.

    :param y: (n,) array of observations. May contain nan, which encodes m
issing observations.
    :param model: LGSS object with the model specification.
    :parma kf: kfs_res object with result from a Kalman filter foward pas
s.

    :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 disturban
ces.
        eta_hat: (deta,1,n) array of smoothed means of state disturbances.
        eta_cov: (deta,deta,n) array of smoothed covariances of state dist
urbances.
```

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 [18]:

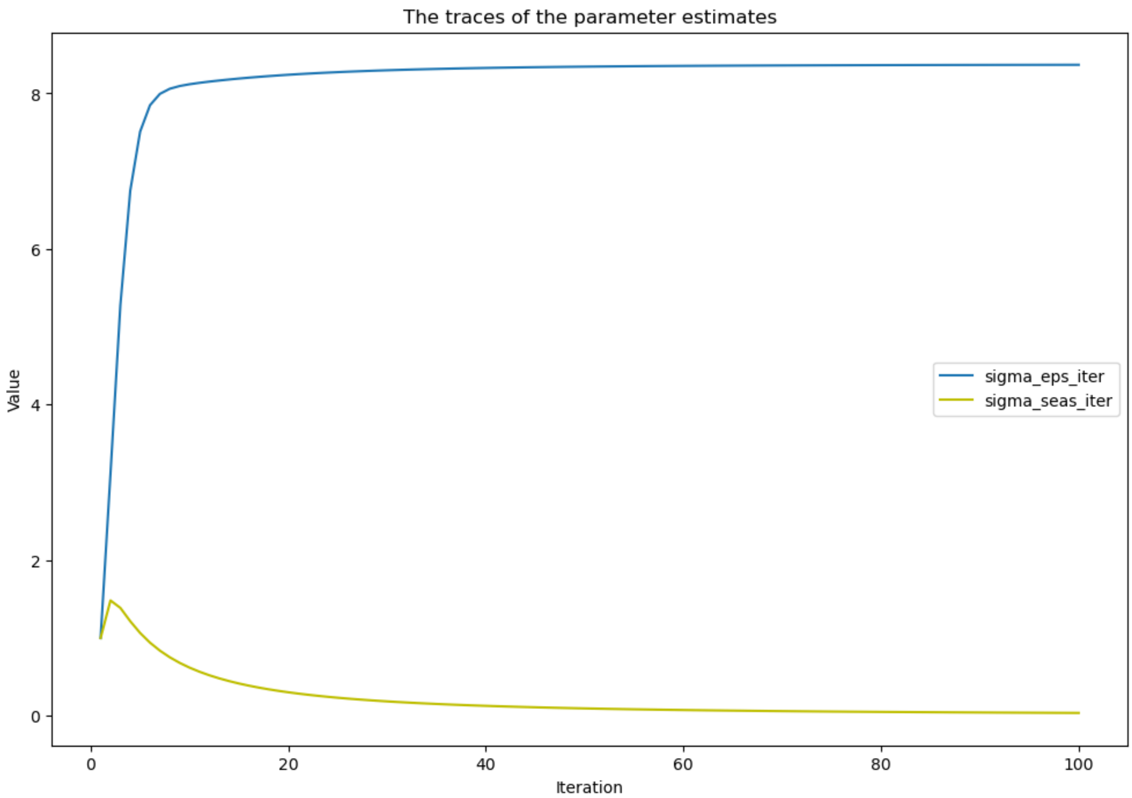
```
print("The log likelihood of each iteration:")
num_iter = 100

y_train=y[:(n+1)].copy()
Q_iter=Q.copy()

sigma_seas_iter=np.zeros(num_iter)
sigma_eps_iter=np.zeros(num_iter)
sigma_seas_iter[0]=Q[1,1]
sigma_eps_iter[0]=sigma_eps**2

for r in range(1,num_iter):
    # E-step

    #For brevity, the variance of the trend component  σ2_μ is fixed to the value  σ2_μ=
    Q_iter = np.array([[sigma_trend**2, 0.], [0., sigma_seas_iter[r-1]]])  # Process noi

    model = LGSS(T, R, Q_iter, Z, sigma_eps_iter[r-1], a1, P1)
    kf=kalman_filter(y_train, model)
    kfs_res_temp=kalman_smoother(y_train, model, kf)



    # Print the log likelihood
    F_pred=kf.F_pred
    y_pred=kf.y_pred
    loglik=0
    for i in range(n):
        loglik=loglik - (((y[i]-y_pred[i])**2)/(2*F_pred[i])) - np.log(F_pred[i])/2 - np
    print(loglik)



    # M-step
    eps_hat=kfs_res_temp.eps_hat
    eps_var=kfs_res_temp.eps_var
    eta_hat=kfs_res_temp.eta_hat
    eta_cov=kfs_res_temp.eta_cov

    sigma_eps_iter[r]=(1/n)*sum(eps_hat**2+eps_var)

    eta_hat_temp=0
    eta_cov_temp=0

    for k in range(n):
        eta_hat_temp=eta_hat_temp+np.dot(eta_hat[:,:,k],eta_hat[:,:,k].T)
        eta_cov_temp=eta_cov_temp+eta_cov[:,:,k]

    Q_iter=(1/n)*(eta_hat_temp+eta_cov_temp)
    sigma_seas_iter[r]=Q_iter[1,1]
```

```
The log likelihood of each iteration:
-2986.42556028042
-2353.6324430863183
-2249.430738177274
-2223.7284583360206
-2214.005706940207
-2207.8591677808345
-2202.9204888606623
-2198.6847997984114
-2194.9801607973263
-2191.7067168384415
-2188.7914638859693
-2186.1775745122723
-2183.8198364339423
-2181.681784346807
-2179.733669235528
-2177.950970698886
-2176.3132881948814
-2174.8035032941975
-2173.40713881174
-2172.11186291849
-2170.9071013650378
-2169.7837312698475
-2168.7338371149363
-2167.750514666213
-2166.8277121636556
-2165.9601007503534
-2165.142968028933
-2164.3721300522966
-2163.6438581144275
-2162.9548175042896
-2162.30201599178
-2161.682760278751
-2161.0946190061536
-2160.5353911868947
-2160.0030791521926
-2159.4958652704477
-2159.012091834405
-2158.55024362077
-2158.108932713362
-2157.6868852514617
-2157.282929821625
-2156.8959872577816
-2156.5250616523467
-2156.1692324120136
-2155.827647217899
-2155.4995157708577
-2155.1841042205815
-2154.8807301918423
-2154.5887583336767
-2154.3075963277424
-2154.03669130099
-2153.7755265948545
-2153.5236188502267
-2153.2805153719796
-2153.0457917421477
-2152.8190496544516
-2152.5999149462964
-2152.3880358073006
-2152.1830811460554
-2151.984739098757
```

```
-2151.792715665453
-2151.60673346131
-2151.426530571362
-2151.251859499131
-2151.0824861998863
-2150.918189190827
-2150.758758731004
-2150.6039960646035
-2150.453712722019
-2150.3077298734556
-2150.1658777306625
-2150.0279949924675
-2149.893928330479
-2149.7635319116293
-2149.6366669544154
-2149.5132013161074
-2149.3930091084753
-2149.275970339729
-2149.161970580554
-2149.050900652449
-2148.9426563365837
-2148.8371381016404
-2148.7342508492457
-2148.633903675606
-2148.536009648222
-2148.4404855965167
-2148.347251915501
-2148.2562323813722
-2148.1673539783537
-2148.0805467358873
-2147.9957435755655
-2147.9128801670395
-2147.8318947923303
-2147.75272821802
-2147.6753235747224
-2147.599626243432
-2147.5255837482414
-2147.453145655084
-2147.382263475981
```

In [19]:

```
plt.plot(range(1,101),sigma_eps_iter,label='sigma_eps_iter')
plt.plot(range(1,101),sigma_seas_iter,label='sigma_seas_iter',c='y')
plt.title('The traces of the parameter estimates')
plt.xlabel('Iteration')
plt.ylabel('Value')
plt.legend()
plt.show()
```



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.

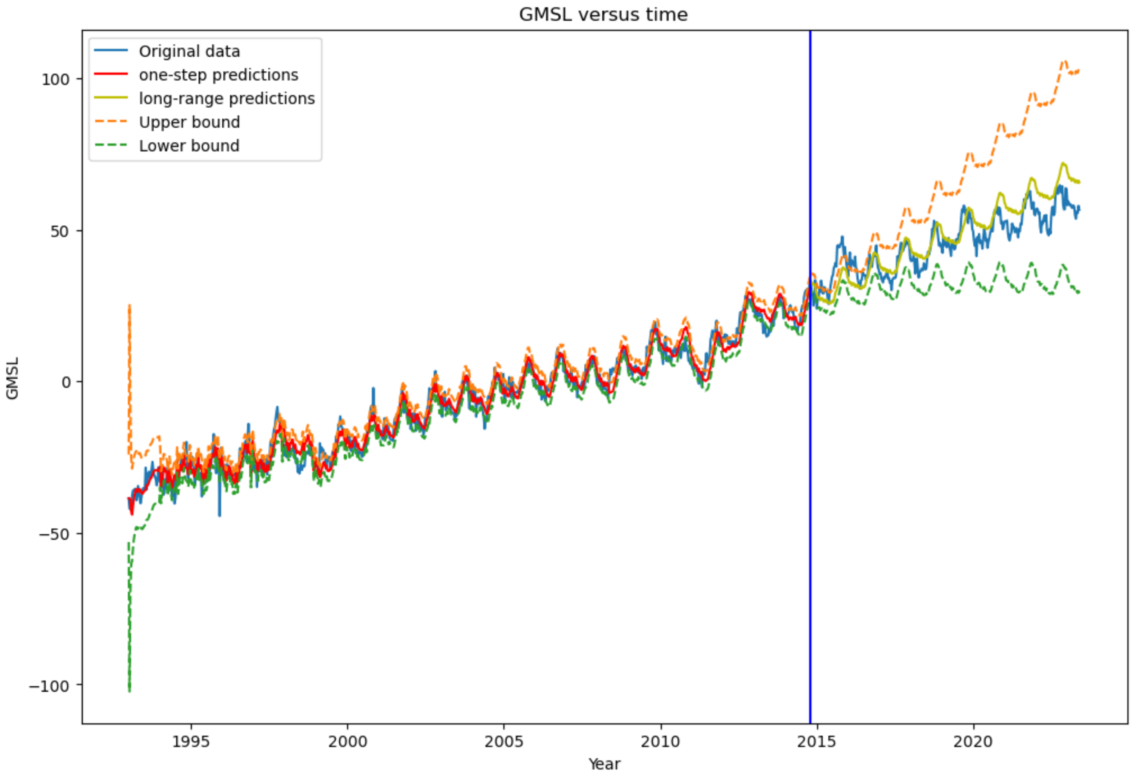
Q13: Rerun the Kalman filter to compute a *long range prediction for the validation data points,* analogously to **Q8** (you can copy-paste code from that question). That is, generate a plot which shows:

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., $t = 1, \ldots, n$,
3. The long-range predictions $\hat{y}_{t|n} \pm 1$ standard deviation for the validation data, i.e., $t = n + 1, \ldots, n + m$,
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 [20]:

```
sigma_eps=np.sqrt(sigma_eps_iter[-1])
Q = np.array([[sigma_trend**2, 0.], [0., sigma_seas_iter[r-1]]])
model = LGSS(T, R, Q, Z, sigma_eps**2, a1, P1)

train_data=y.copy()
train_data[(n+1):]=np.nan
kf=kalman_filter(train_data, model)

# Plot
plt.plot(u,y,label='Original data')
plt.plot(u[:801],kf.y_pred[:801],label='one-step predictions',c='r')
plt.plot(u[801:],kf.y_pred[801:],label='long-range predictions',c='y')
plt.plot(u,(kf.y_pred+np.sqrt(kf.F_pred)),linestyle='--',label='Upper bound')
plt.plot(u,(kf.y_pred-np.sqrt(kf.F_pred)),linestyle='--',label='Lower bound')
plt.axvline(x=u[801],c='b')
plt.title('GMSL versus time')
plt.xlabel('Year')
plt.ylabel('GMSL')
plt.legend()
plt.show()
```



In [21]:

```
F_pred=kf.F_pred
y_pred=kf.y_pred

loglik=0
for i in range(n):
    loglik=loglik - (((y[i]-y_pred[i])**2)/(2*F_pred[i])) - np.log(F_pred[i])/2 - np.log

print(loglik)
```

```
-2147.3822636561067
```

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

$$y_t = s_t + \varepsilon_t$$

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., $t = 1, \ldots, n$,
3. The predictions $\mathbb{E}[s_t | y_{1:n}] \pm 1$ standard deviation for the validation data, i.e., $t = n + 1, \ldots, n + m$,
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.

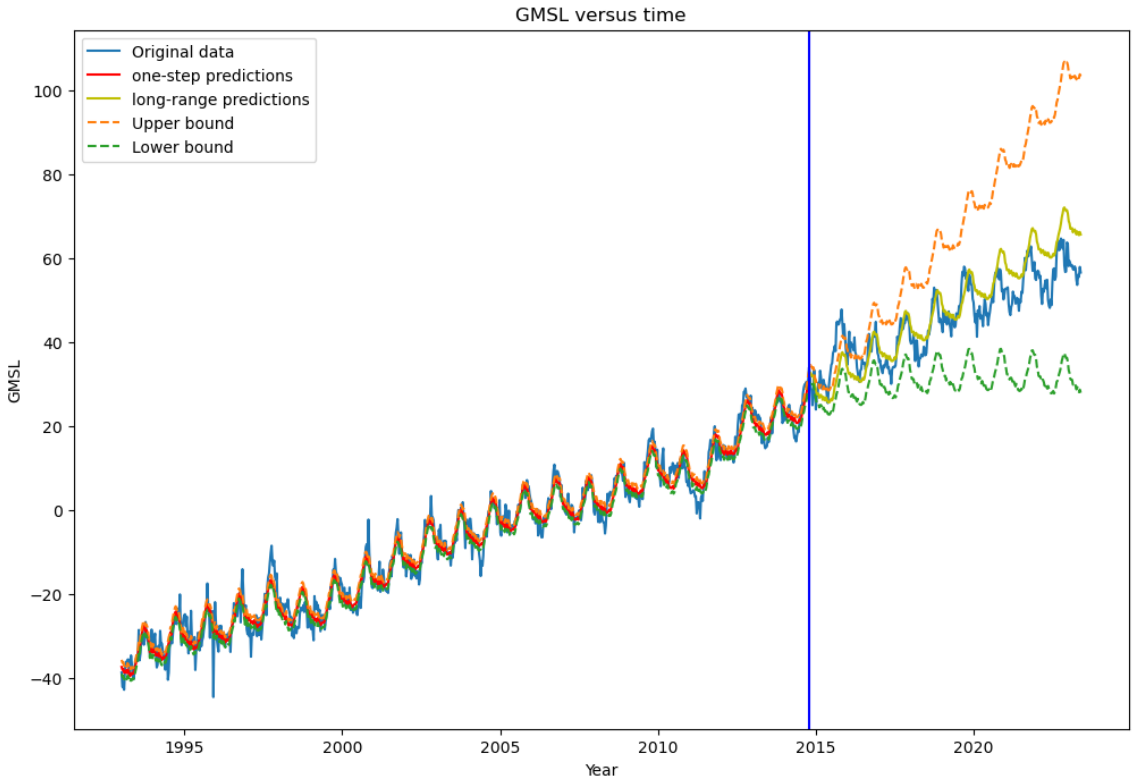
In [22]:

```
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-dimens
ional observation.

    :param y: (n,) array of observations. May contain nan, which encodes m
issing observations.
    :param model: LGSS object with the model specification.
    :parma kf: kfs_res object with result from a Kalman filter foward pas
s.

    :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 disturban
ces.
        eta_hat: (deta,1,n) array of smoothed means of state disturbances.
        eta_cov: (deta,deta,n) array of smoothed covariances of state dist
urbances.
```

In [23]:

```
smoothing_res=kalman_smoother(train_data, model, kf)
st=smoothing_res.alpha_sm[0,:,:]+smoothing_res.alpha_sm[2,:,:]
st=np.squeeze(st)
v=np.sqrt(smoothing_res.V[0,0,:])+np.sqrt(smoothing_res.V[2,2,:])

# Plot
plt.plot(u,y,label='Original data')
plt.plot(u[:801],st[:801],label='one-step predictions',c='r')
plt.plot(u[801:],st[801:],label='long-range predictions',c='y')
plt.plot(u,(st+v),linestyle='--',label='Upper bound')
plt.plot(u,(st-v),linestyle='--',label='Lower bound')
plt.axvline(x=u[801],c='b')
plt.title('GMSL versus time')
plt.xlabel('Year')
plt.ylabel('GMSL')
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:

1. In training data points, the variances of the predicted y_t made by smoother are smaller than that of Kalman filter. This is because the filter only refers to 1:t-1 states to make prediction, while the smoother refers to the whole states (1:n). Hence the smoother can make more accurate prediction.

2. In validation data points, the variances of the predicted y_t made by smoother are larger than that of

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

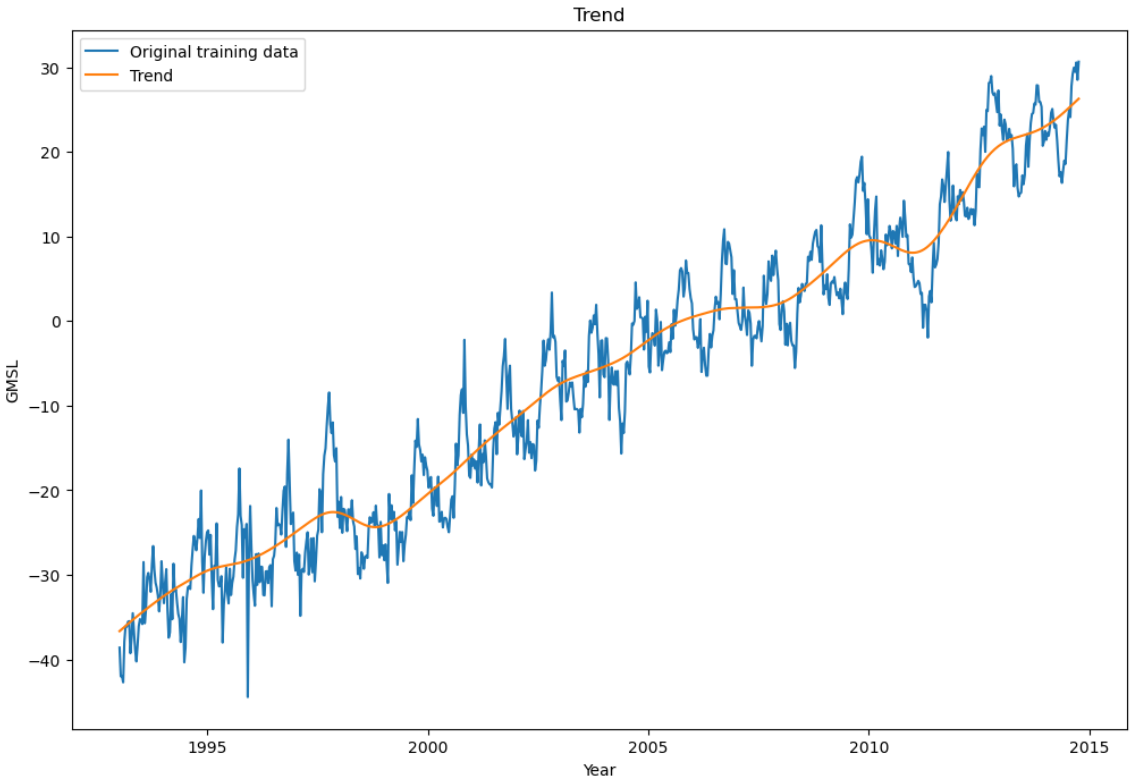
Q16: Using the results of the state smoother, compute and plot the *smoothed estimates* of the two signal components, i.e.:

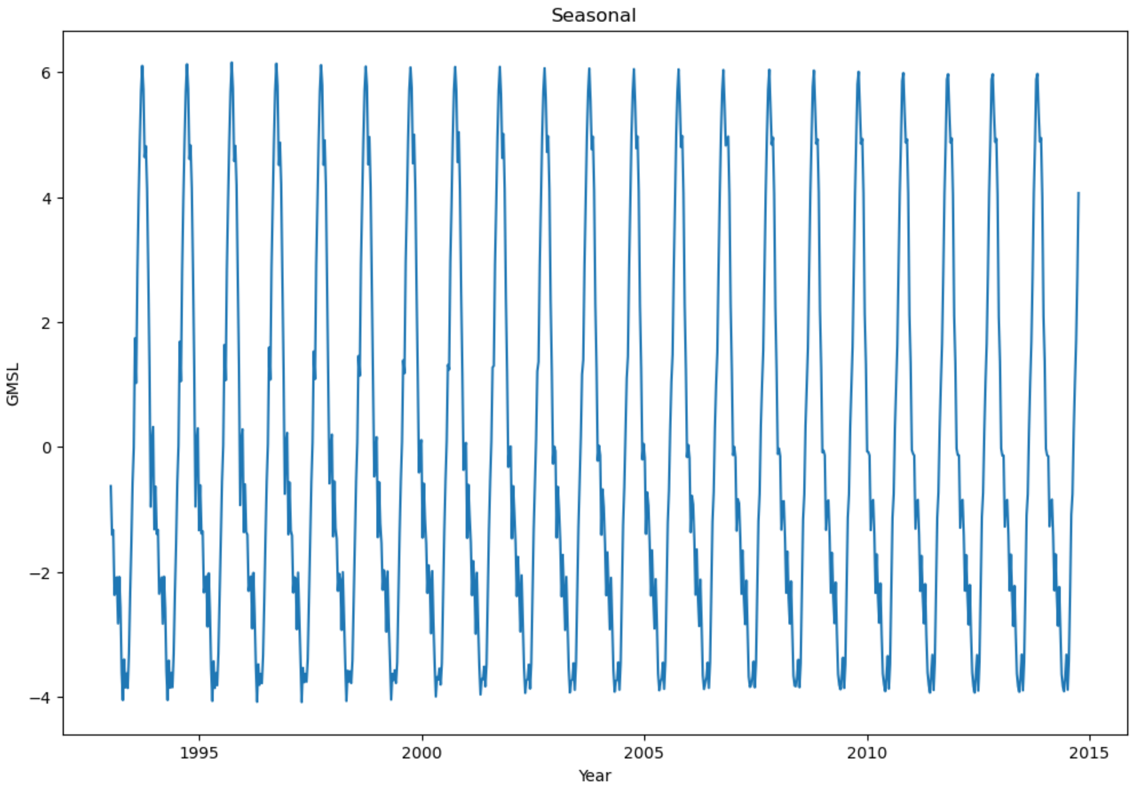
1. Trend: $\hat{\mu}_{t|n} = \mathbb{E}[\mu_t | y_{1:n}]$ for $t = 1, \dots, n$
2. Seasonal: $\hat{\gamma}_{t|n} = \mathbb{E}[\gamma_t | y_{1:n}]$ for $t = 1, \dots, n$

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

In [24]:

```
mu_t=np.squeeze(smoothing_res.alpha_sm[0,:,:])
gamma_t=np.squeeze(smoothing_res.alpha_sm[2,:,:])

# Plot Trend
plt.plot(u[:801],y[:801],label='Original training data')
plt.plot(u[:801],mu_t[:801],label='Trend')
plt.title('Trend')
plt.xlabel('Year')
plt.ylabel('GMSL')
plt.legend()
plt.show()

# Plot seasonal
plt.plot(u[:801],gamma_t[:801],label='Seasonal')
plt.title('Seasonal')
plt.xlabel('Year')
plt.ylabel('GMSL')
plt.show()
```





time series
unk of

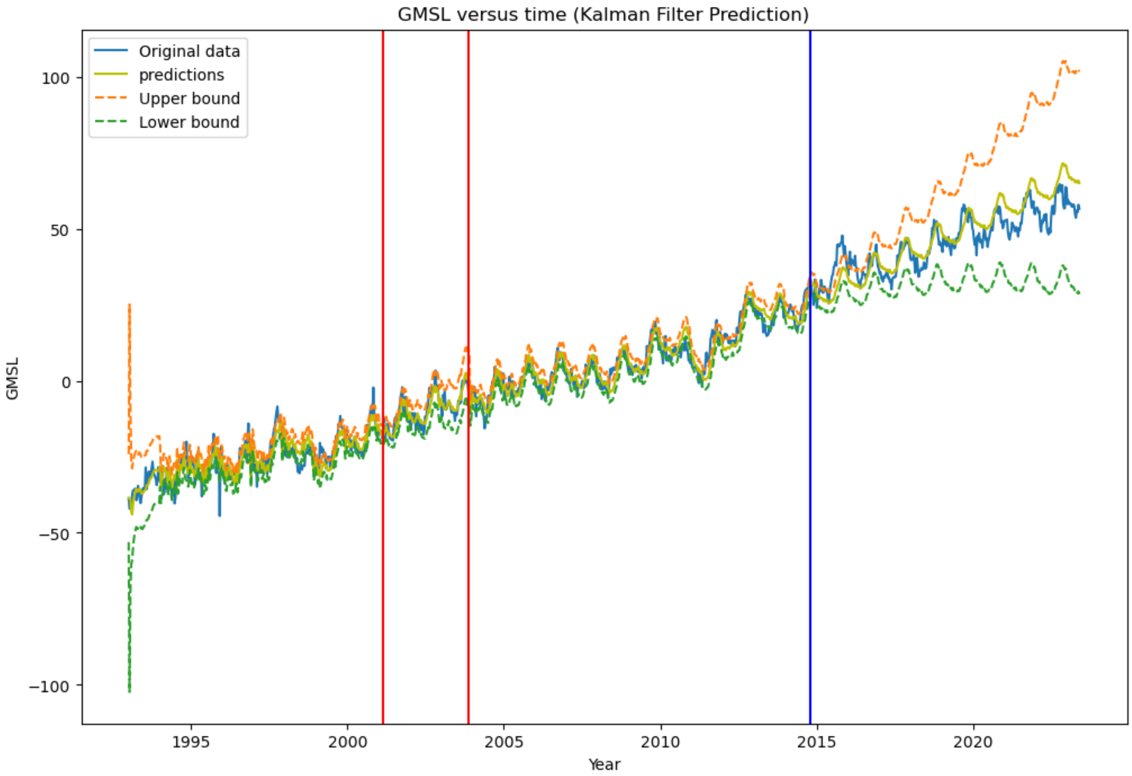
filter and

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

Comment In the Kalman filter prediction, the prediction between 300 < t < 400 has larger variance, and it's variance become larger as t approach 400. While the Smoother make the prediction with lower variance for 300 < t < 400. This is because the filter only infers to the past 1:300 observations to make prediction, while the smoother refers to the past and the future observations.

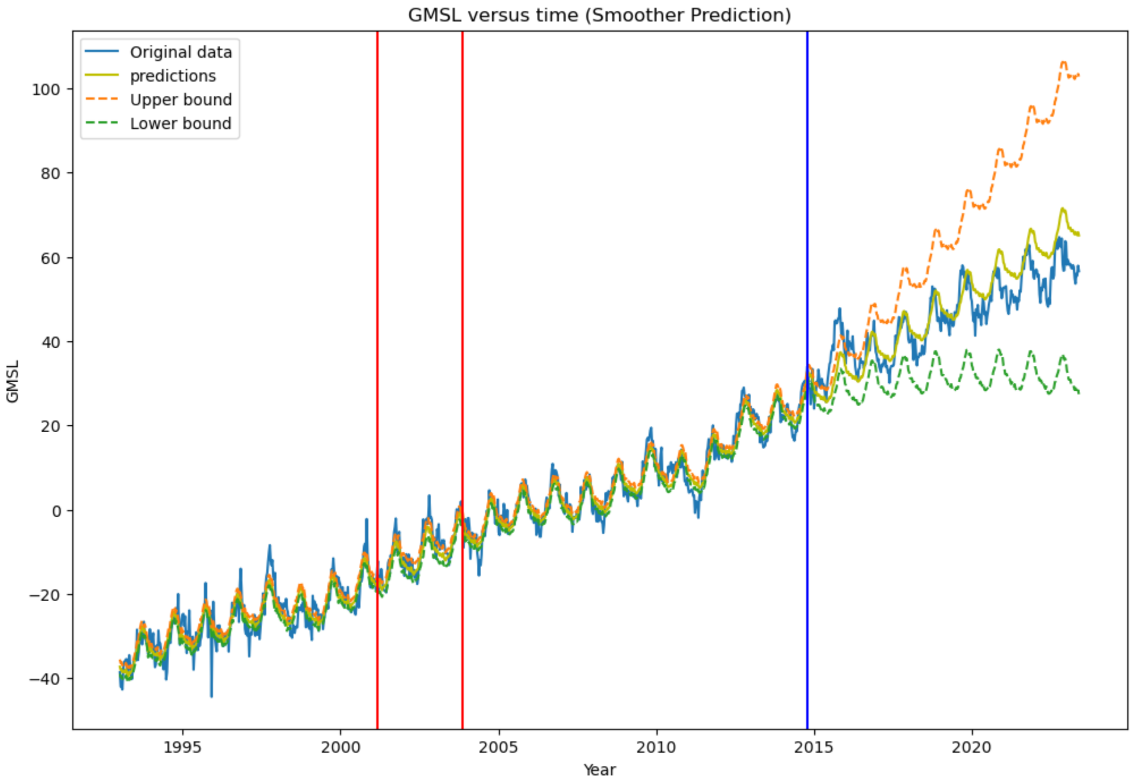
In [25]:

```
# Use the parameters which have already been estimated by smoother (Q,sigma_eps)
model = LGSS(T, R, Q, Z, sigma_eps**2, a1, P1)

train_data=y.copy()
train_data[(n+1):]=np.nan
train_data[300:401]=np.nan
kf=kalman_filter(train_data, model)

# Plot
plt.plot(u,y,label='Original data')
plt.plot(u,kf.y_pred,label='predictions',c='y')
plt.plot(u,(kf.y_pred+np.sqrt(kf.F_pred)),linestyle='--',label='Upper bound')
plt.plot(u,(kf.y_pred-np.sqrt(kf.F_pred)),linestyle='--',label='Lower bound')
plt.axvline(x=u[801],c='b')
plt.axvline(x=u[300],c='r')
plt.axvline(x=u[400],c='r')
plt.title('GMSL versus time (Kalman Filter Prediction)')
plt.xlabel('Year')
plt.ylabel('GMSL')
plt.legend()
plt.show()
```



In [26]:

```
smoothing_res=kalman_smoother(train_data, model, kf)
st=smoothing_res.alpha_sm[0,:,:]+smoothing_res.alpha_sm[2,:,:]
st=np.squeeze(st)
v=np.sqrt(smoothing_res.V[0,0,:])+np.sqrt(smoothing_res.V[2,2,:])

# Plot
plt.plot(u,y,label='Original data')
plt.plot(u,st,label='predictions',c='y')
plt.plot(u,(st+v),linestyle='--',label='Upper bound')
plt.plot(u,(st-v),linestyle='--',label='Lower bound')
plt.axvline(x=u[801],c='b')
plt.axvline(x=u[300],c='r')
plt.axvline(x=u[400],c='r')
plt.title('GMSL versus time (Smoother Prediction)')
plt.xlabel('Year')
plt.ylabel('GMSL')
plt.legend()
plt.show()
```



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

$$y_t = \mu_t + \gamma_t + \varepsilon_t$$

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 boild 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

$$y_t = \mu_t + \gamma_t + \nu_t + \varepsilon_t$$

where $\nu_t \sim \mathrm{AR}(p)$. 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