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 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.

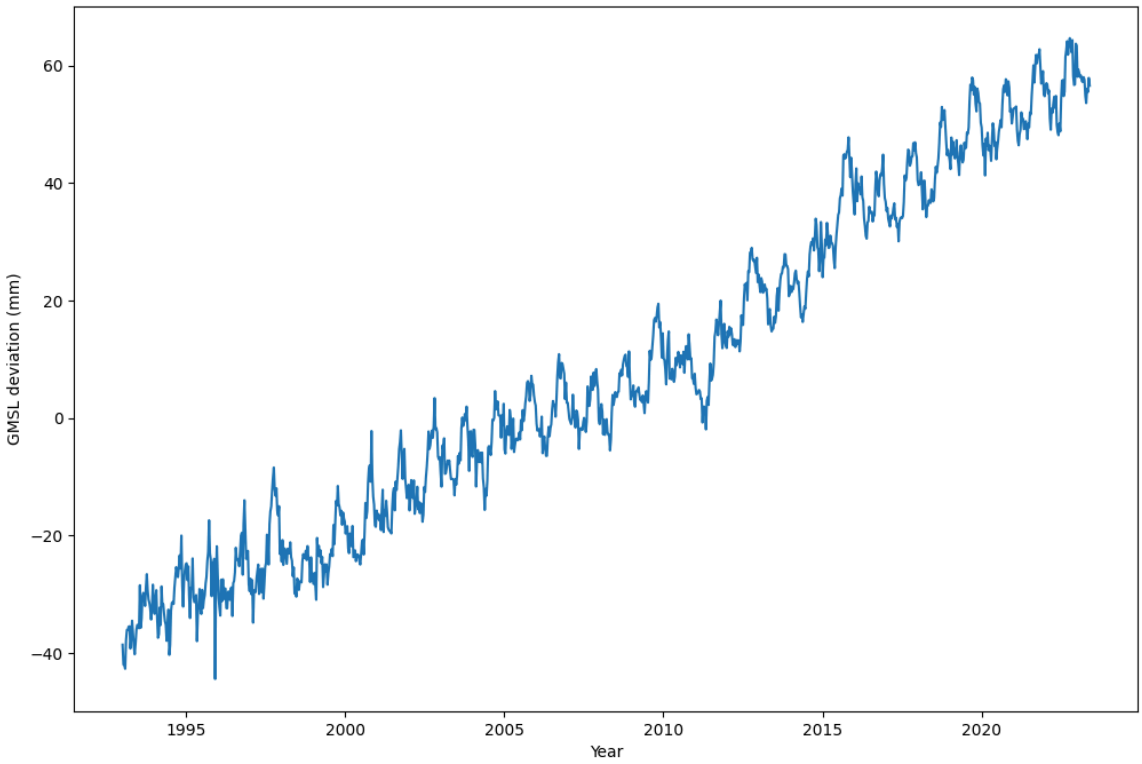
In [1]:
```
import pandas  # Loading data / handling data frames
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import block_diag
plt.rcParams["figure.figsize"] = (12,8)  # Increase default size of plots
```

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

$$y_t = \mu_t + \gamma_t + \varepsilon_t \tag{1}$$

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,

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

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?

A0:

State Transition Matrix (T) has dimension $(d \times d)$

(Square matrix of the dimensions of the unobserved state variable)

State Noise Transition Matrix (R) has dimension $(d \times d_\eta)$

(Matrix of dimensions of state dimension by state noise dimension)

Observation Matrix (Z) has dimension $(1 \times d)$

(Row vector of state dimension)

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.array([[2, -1], [1, 0]])
R_mu = np.array([[1], [0]])
Z_mu = np.array([[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]:
```
print(data['Year'])
```

```
0       1993.011526
1       1993.038692
2       1993.065858
3       1993.093025
4       1993.120191
           ...
1114    2023.282608
1115    2023.309774
1116    2023.336940
1117    2023.364106
1118    2023.391272
Name: Year, Length: 1119, dtype: float64
```

In [6]:
```
year = data['Year'].astype(int)
year_count = year.value_counts()

print(year_count)
print("Average number of observations per (whole) year is", round(year_count[:-1
```

```
1993    37
2018    37
2016    37
2015    37
2019    37
2013    37
2012    37
2011    37
2010    37
2009    37
1994    37
2007    37
2005    37
2004    37
2003    37
2021    37
2001    37
2000    37
1999    37
1998    37
1997    37
2022    37
1995    37
2017    37
2020    36
2008    36
2014    36
2006    36
2002    36
1996    36
2023    15
Name: Year, dtype: int64
Average number of observations per (whole) year is 37
```

As you can see, most of the years has 37 observations and some of them has 36 observations except year 2023. We can ignore 2023 and set the periodicity $s = 37$

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?

A3: State Transition Matrix (T) -> $(s-1) \times (s-1) = 36x36$

State Noise Transition Matrix (R) -> $(s-1) \times 1 = 36x1$

Observation Matrix (Z) -> $1 \times (s-1) = 1x36$

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 [7]:
```
s = 37
T_gamma = np.zeros((s - 1, s - 1))
T_gamma[0, :] = -1
T_gamma[1:, :-1] = np.eye(s - 1 - 1)
```

```
R_gamma = np.zeros((s - 1, 1))
R_gamma[0, :] = 1

Z_gamma = np.zeros((1, s - 1))
Z_gamma[:, 0] = 1

print("T_gamma:\n", T_gamma)
print("R_gamma:\n", R_gamma)
print("Z_gamma:\n", Z_gamma)
```

```
T_gamma:
 [[-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_gamma:
 [[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.]]
Z_gamma:
 [[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.]]
```

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

print("Shape of T:", T.shape)
print("Shape of R:", R.shape)
print("Shape of Z:", Z.shape)
```

```
Shape of T: (38, 38)
Shape of R: (38, 2)
Shape of Z: (1, 38)
```

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 [9]:
```
# 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 covar
```

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 [10]:
```
k = 2
total_dim = k + s - 1 # Total dimension of the state, also can be use T.shape[0]
a1 = np.zeros((total_dim, 1))

a1[0] = y[0]  # Set E[mu_1] to the first observation
a1[1] = y[0]  # Set the slope E[mu_1 - mu_0] to 0

P1 = np.eye(total_dim) * 100 # Identity matrix scaled by 100
```

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 [11]:
```
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 [12]:
```
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 [13]:
```
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
    :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

    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(T,np.dot(P_filt[:,:,t-1],np.transpose(T))) +


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

        # Measurement update (correct)
        # ADD CODE HERE
        if np.isnan(y[t]):
            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]
            alpha_filt[:,:,t] = alpha_pred[:,:,t] + np.dot(K,(y[t]-y_pred[t]))
            P_filt[:,:,t] = np.dot((np.identity(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
```

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:

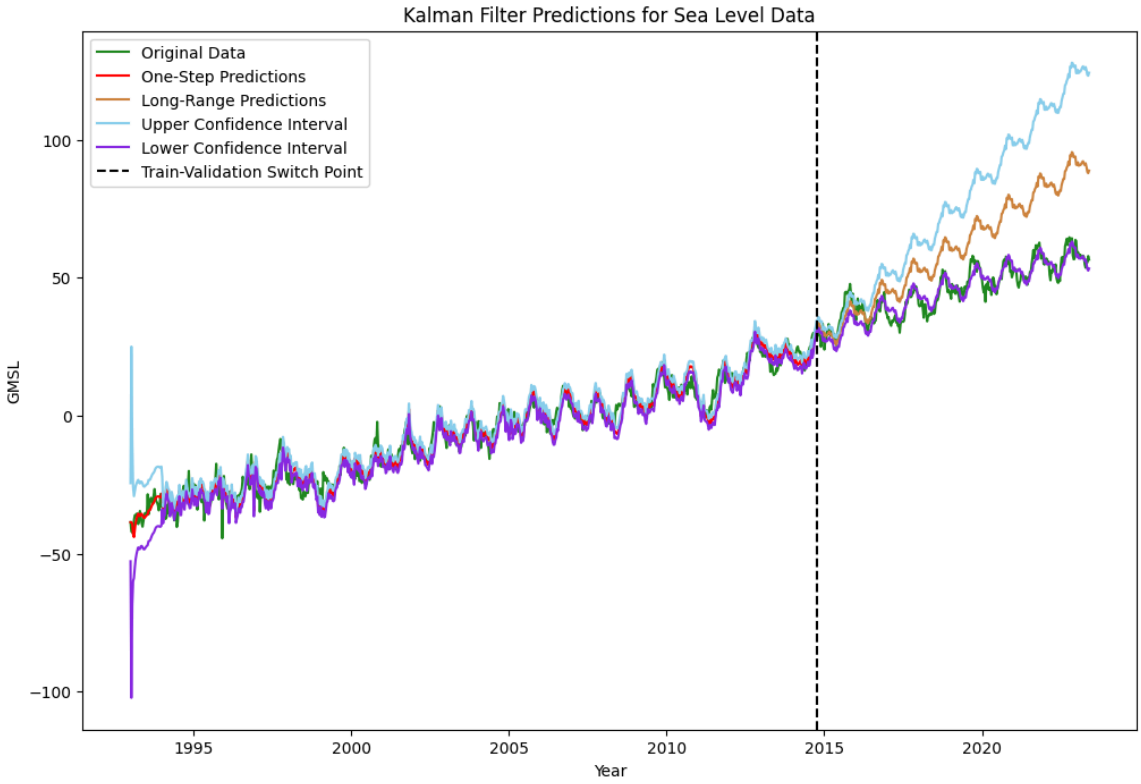
1. The data $y_{1:n+m}$,
2. The one-step predictions $\hat{y}_{t|t-1} \pm 1$ standard deviation for the training data, i.e., $t = 1, \ldots, n$,

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 [14]:
```
y_kalman = np.copy(y)
y_kalman[n:] = np.nan

# Kalman filter
kalman = kalman_filter(y_kalman, model)

# Confidence Interval
upper_CI = kalman.y_pred + np.sqrt(kalman.F_pred)
lower_CI = kalman.y_pred - np.sqrt(kalman.F_pred)

# Plot
plt.plot(u, y, label = "Original Data", color = "forestgreen")
plt.plot(u[:n], kalman.y_pred[:n], label = "One-Step Predictions", color = "red"
plt.plot(u[n:], kalman.y_pred[n:], label = "Long-Range Predictions", color = "pe
plt.plot(u, upper_CI, label = "Upper Confidence Interval", color = "skyblue")
plt.plot(u, lower_CI, label = "Lower Confidence Interval", color = "blueviolet")
plt.axvline(x = u[n], color="black", label="Train-Validation Switch Point", line
plt.title("Kalman Filter Predictions for Sea Level Data")
plt.xlabel("Year")
plt.ylabel("GMSL")
plt.legend()
plt.show()
```

```
C:\Users\sfurk\AppData\Local\Temp\ipykernel_14716\1836612437.py:42: Deprecation
Warning: Conversion of an array with ndim > 0 to a scalar is deprecated, and wi
ll error in future. Ensure you extract a single element from your array before
performing this operation. (Deprecated NumPy 1.25.)
  y_pred[t] = np.dot(Z,alpha_pred[:,:,t])
C:\Users\sfurk\AppData\Local\Temp\ipykernel_14716\1836612437.py:43: Deprecation
Warning: Conversion of an array with ndim > 0 to a scalar is deprecated, and wi
ll error in future. Ensure you extract a single element from your array before
performing this operation. (Deprecated NumPy 1.25.)
  F_pred[t] = np.dot(Z,np.dot(P_pred[:,:,t],np.transpose(Z))) + H
```



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

In [15]:
```
# Normalization constant for Gaussian distribution
constant = -n/2 * np.log(2 * np.pi)
#Sum of log of the prediction variances
var = np.sum(np.log(kalman.F_pred[:n]))

# Difference between observed values and one-step predictions
residuals = y[:n] - kalman.y_pred[:n]

exponential = np.sum((residuals ** 2) / kalman.F_pred[:n])

loglikelihood = constant - 1/2 * (var + exponential)
print("Training Data Log-Likelihood:", loglikelihood)
```

```
Training Data Log-Likelihood: -2986.4255602804183
```

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,

$$\theta = (\sigma_\gamma^2, \sigma_\varepsilon^2). \tag{4}$$

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

$= \arg\max_\theta(const - \frac{1}{2}\sum_{t=1}^{n} log(|\sigma_\epsilon^2| + log(|Q|) + \theta_{t|n}^2 + Var[\epsilon|y_{1:n}]\sigma_\epsilon^{-2} + tr[\hat{\eta}_{t|n}\hat{\eta}_{t|n}^T +$

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

Hint: Look at Exercise Session 2

A:

$\hat{\sigma}_\epsilon^2 = \frac{1}{n}\sum_{t=1}^{n}\left[\hat{\epsilon}_{t|n}^2 + Var[\epsilon_t|y_{1:n}]\right]$

$\hat{\sigma}_\gamma^2 = \frac{1}{n}\sum_{t=1}^{n}\left[\hat{\eta}_{t|n}\hat{\eta}_{t|n}^T + Var[\eta_t|y_{1:n}]\right]$

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

$$p(\alpha_t \mid y_{1:n}), \qquad t = 1, \ldots, n \tag{5}$$

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,

$$p(\eta_t \mid y_{1:n}), \qquad t = 1, \ldots, n-1 \tag{6}$$
$$p(\varepsilon_t \mid y_{1:n}), \qquad t = 1, \ldots, n \tag{7}$$

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.

In [16]:
```
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 missin
g observations.
    :param model: LGSS object with the model specification.
    :parma kf: kfs_res object with result from a Kalman filter foward pass.

    :return kfs_res: Container class. The original Kalman filter result is augm
ented 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 disturban
ces.
```

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 $r = 0, \ldots, 100$.

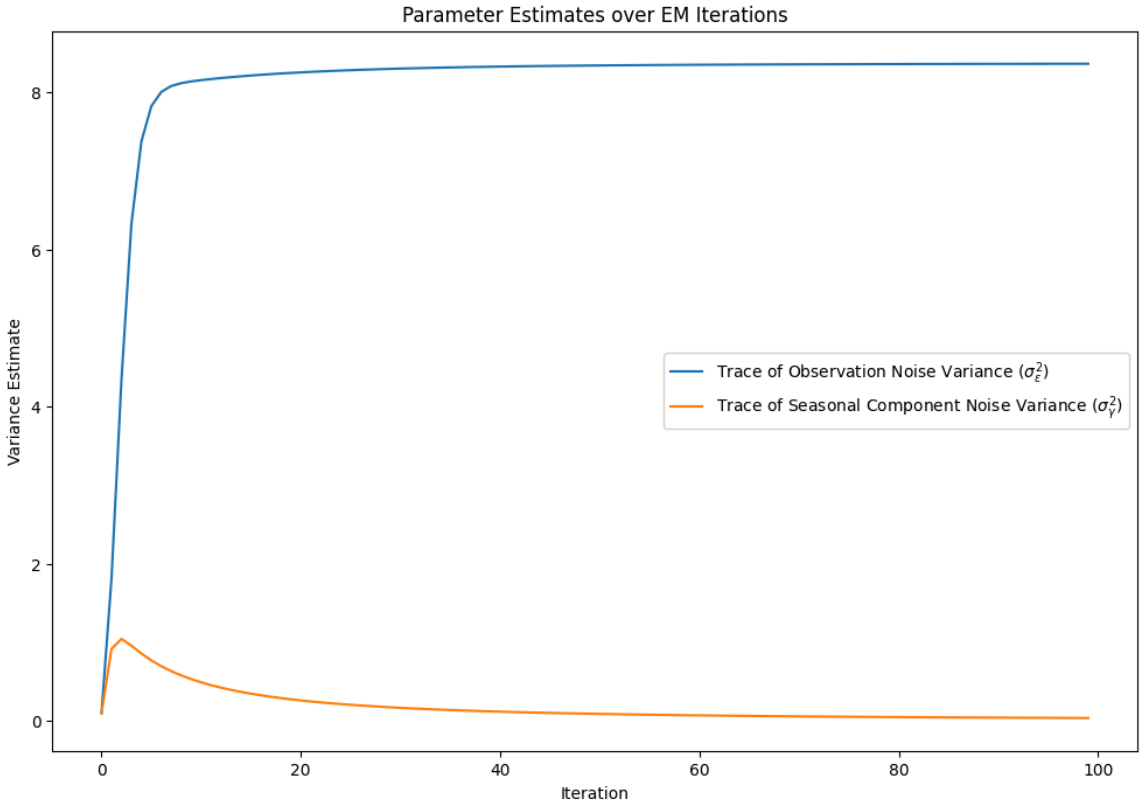
Note: When running the Kalman filter as part of the EM loop you should only filter the *training data* (i.e. excluding the prediction for validation data).

In [17]:
```
num_iter = 100
sigma_eps = 0.1
sigma_seas = 0.1

theta = np.zeros((num_iter, 2))
theta[0, 0] = sigma_eps
theta[0, 1] = sigma_seas

for r in range(1,num_iter):
    # E-step
    Q = np.array([[sigma_trend**2, 0], [0, sigma_seas]])
    model = LGSS(T, R, Q, Z, sigma_eps, a1, P1)
    kalman = kalman_filter(y_kalman[:n], model)
    smooth = kalman_smoother(y_kalman[:n], model, kalman)

    # M-step
    sigma_eps = 1/n * sum(smooth.eps_hat**2 + smooth.eps_var)
    theta[r,0] = sigma_eps

    result = 0
    for j in range(n):
        smooth_hat = np.dot(smooth.eta_hat[:,:,j], smooth.eta_hat[:,:,j].T)
        smooth_cov = smooth.eta_cov[:,:,j]
        result += smooth_hat + smooth_cov

    result = result[1,1]
    sigma_seas = result / n
    theta[r,1] = sigma_seas
```

```
print("Sigma Epsilon Estimation: ", round(sigma_eps, 5))
print("Sigma Seasonal Component Estimation: ", round(sigma_seas, 5))
```

```
C:\Users\sfurk\AppData\Local\Temp\ipykernel_14716\1836612437.py:42: Deprecation
Warning: Conversion of an array with ndim > 0 to a scalar is deprecated, and wi
ll error in future. Ensure you extract a single element from your array before
performing this operation. (Deprecated NumPy 1.25.)
  y_pred[t] = np.dot(Z,alpha_pred[:,:,t])
C:\Users\sfurk\AppData\Local\Temp\ipykernel_14716\1836612437.py:43: Deprecation
Warning: Conversion of an array with ndim > 0 to a scalar is deprecated, and wi
ll error in future. Ensure you extract a single element from your array before
performing this operation. (Deprecated NumPy 1.25.)
  F_pred[t] = np.dot(Z,np.dot(P_pred[:,:,t],np.transpose(Z))) + H
Sigma Epsilon Estimation:  8.36675
Sigma Seasonal Component Estimation:  0.03737
```

In [18]:
```
plt.plot(theta[:, 0], label="Trace of Observation Noise Variance ($\sigma_\epsil
plt.plot(theta[:, 1], label="Trace of Seasonal Component Noise Variance ($\sigma
plt.xlabel("Iteration")
plt.ylabel("Variance Estimate")
plt.title("Parameter Estimates over EM Iterations")
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.

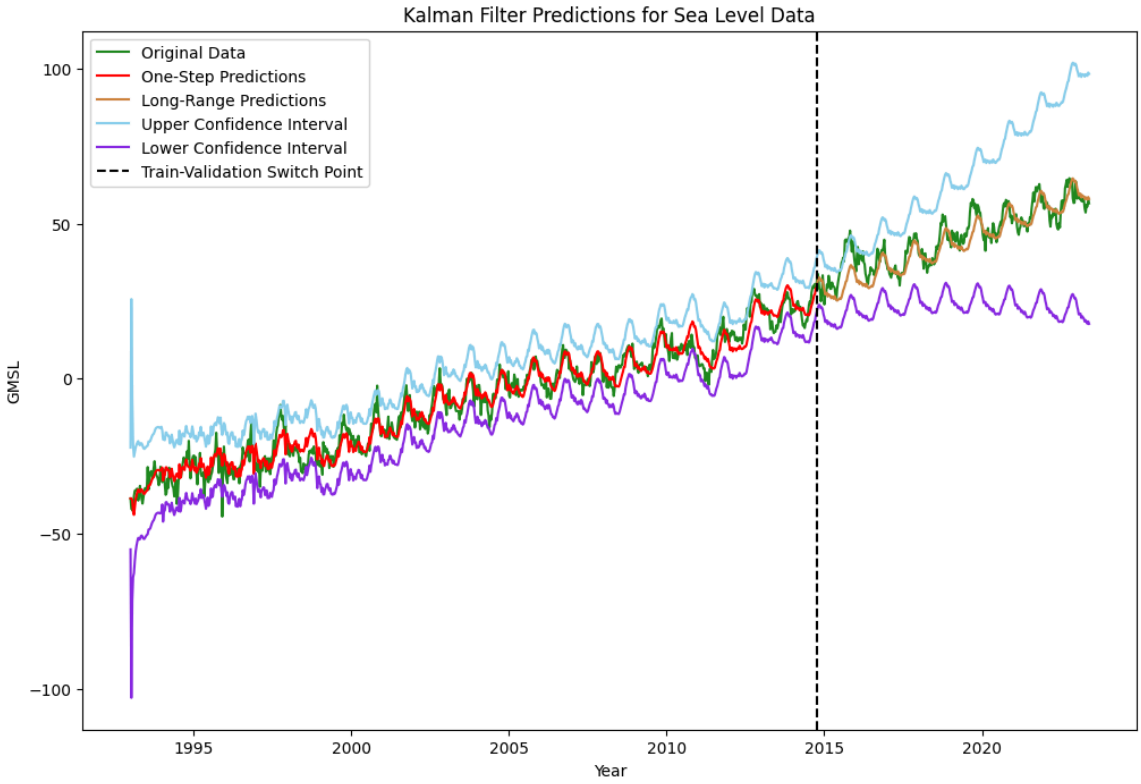
Q13: Rerun the Kalman filter to compute a *long range prediction for the validation data points,* analogously to **Q8** (you can copy-paste code from that question). That is, generate a plot which shows:

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 [19]:
```
y_kalman = np.copy(y)
y_kalman[n:] = np.nan

Q_new = np.array([[sigma_trend**2, 0.], [0., sigma_seas**2]])
model = LGSS(T, R, Q_new, Z, sigma_eps**2, a1, P1)

# Perform Kalman filtering on the entire dataset
kalman = kalman_filter(y_kalman, model)

# Confidence Interval
upper_CI = kalman.y_pred + np.sqrt(kalman.F_pred)
lower_CI = kalman.y_pred - np.sqrt(kalman.F_pred)

# Plot
plt.plot(u, y, label = "Original Data", color = "forestgreen")
plt.plot(u[:n], kalman.y_pred[:n], label = "One-Step Predictions", color = "red"
plt.plot(u[n:], kalman.y_pred[n:], label = "Long-Range Predictions", color = "pe
plt.plot(u, upper_CI, label = "Upper Confidence Interval", color = "skyblue")
plt.plot(u, lower_CI, label = "Lower Confidence Interval", color = "blueviolet")
plt.axvline(x = u[n], color="black", label="Train-Validation Switch Point", line
plt.title("Kalman Filter Predictions for Sea Level Data")
plt.xlabel("Year")
plt.ylabel("GMSL")
plt.legend()
plt.show()
```

```
C:\Users\sfurk\AppData\Local\Temp\ipykernel_14716\1836612437.py:42: Deprecation
Warning: Conversion of an array with ndim > 0 to a scalar is deprecated, and wi
ll error in future. Ensure you extract a single element from your array before
performing this operation. (Deprecated NumPy 1.25.)
  y_pred[t] = np.dot(Z,alpha_pred[:,:,t])
C:\Users\sfurk\AppData\Local\Temp\ipykernel_14716\1836612437.py:43: Deprecation
Warning: Conversion of an array with ndim > 0 to a scalar is deprecated, and wi
ll error in future. Ensure you extract a single element from your array before
performing this operation. (Deprecated NumPy 1.25.)
  F_pred[t] = np.dot(Z,np.dot(P_pred[:,:,t],np.transpose(Z))) + H
```



In [20]:
```
residuals = y[:n] - kalman.y_pred[:n]
log_likelihood = -n/2 * np.log(2 * np.pi) - 0.5 * np.sum(np.log(kalman.F_pred[:n
                 0.5 * np.sum((residuals ** 2) / kalman.F_pred[:n])

print("Training Data Log-Likelihood:", log_likelihood)
```

```
Training Data Log-Likelihood: -2584.0909588755294
```

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

$$y_t = s_t + \varepsilon_t \tag{8}$$

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:

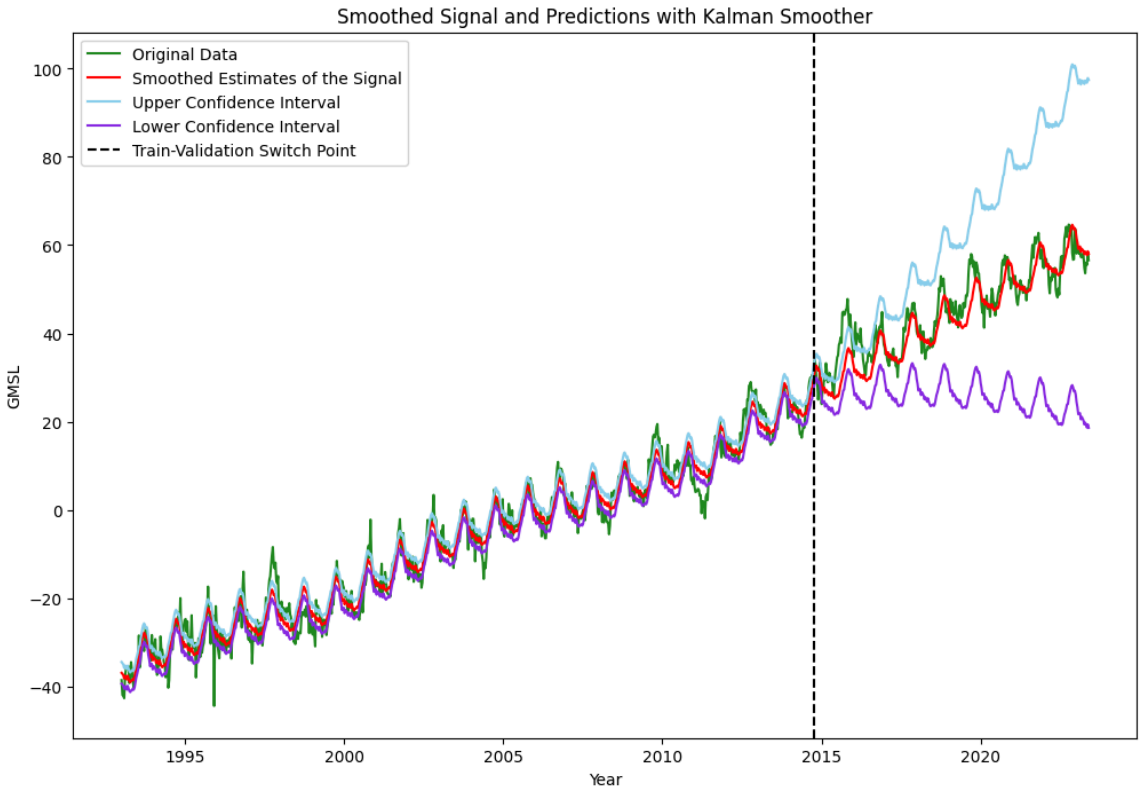
1. The data $y_{1:n+m}$,
2. The smoothed estimates $\mathbb{E}[s_t|y_{1:n}] \pm 1$ standard deviation for the training data, i.e., $t = 1, \ldots, n$,
3. The predictions $\mathbb{E}[s_t|y_{1:n}] \pm 1$ standard deviation for the validation data, i.e., $t = n+1, \ldots, n+m$,
4. A vertical line indicating 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 [21]:
```
smooth_result = kalman_smoother(y_kalman, model, kalman)
```

```
res = np.zeros(n + m)
res_var = np.zeros(n + m)

for i in range(n + m):
    res[i] = np.dot(model.Z , smooth_result.alpha_sm[:,:,i] )
    res_var[i] = np.dot(model.Z, np.dot(smooth_result.V[:,:,i], model.Z.T))

# Plot
plt.plot(u, y, label = "Original Data", color = "forestgreen")
plt.plot(u, res, label = "Smoothed Estimates of the Signal", color = "red")
plt.plot(u, res + np.sqrt(res_var), label = "Upper Confidence Interval", color =
plt.plot(u, res - np.sqrt(res_var), label = "Lower Confidence Interval", color =
plt.axvline(x = u[n], color="black", label="Train-Validation Switch Point", line
plt.title("Smoothed Signal and Predictions with Kalman Smoother")
plt.xlabel("Year")
plt.ylabel("GMSL")
plt.legend()
plt.show()
```

```
C:\Users\sfurk\AppData\Local\Temp\ipykernel_14716\2795167245.py:7: DeprecationW
arning: Conversion of an array with ndim > 0 to a scalar is deprecated, and wil
l error in future. Ensure you extract a single element from your array before p
erforming this operation. (Deprecated NumPy 1.25.)
  res[i] = np.dot(model.Z , smooth_result.alpha_sm[:,:,i] )
C:\Users\sfurk\AppData\Local\Temp\ipykernel_14716\2795167245.py:8: DeprecationW
arning: Conversion of an array with ndim > 0 to a scalar is deprecated, and wil
l error in future. Ensure you extract a single element from your array before p
erforming this operation. (Deprecated NumPy 1.25.)
  res_var[i] = np.dot(model.Z, np.dot(smooth_result.V[:,:,i], model.Z.T))
```



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:

For Training data points, $t \leq n$:

Kalman Filter Prediction (Q13): These are one-step-ahead predictions. This means that the prediction for y_t at time t is based on all available data up to $t-1$. Since the Kalman filter does not use future data, these predictions are more uncertain and recursive, as they only rely on past observations.

Smoothed Signal Estimates (Q14): These estimates incorporate all available data up to $t = n$, including both past and future data points. This makes the estimates more accurate with smaller uncertainty. Because future data is also considered, the smoothing gives more reliable estimates for each time step.

Conclusion: The smoothed estimates are more accurate for the training data because they use more information (both past and future), making them more stable and providing narrower confidence intervals compared to the one-step-ahead predictions from the Kalman filter.

For Validation data points, $t > n$:

Kalman Filter Prediction (Q13): These are long-range predictions, which means they are based on the last available data point at $t = n$. Over time, the uncertainty increases because there is no new data to update the estimates.

Smoothed Signal Estimates (Q14): In this case, the smoothed estimates also act as long-range predictions beyond $t = n$. Since there is no future data available for these points, the smoothing and Kalman filter predictions become similar. Both methods are essentially making predictions based only on the past data.

Conclusion: For validation data points, both the Kalman filter and smoothed estimates provide similar predictions because both rely on past information without any new data to improve accuracy. Therefore, the uncertainty grows similarly for both methods in the validation period.

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.:

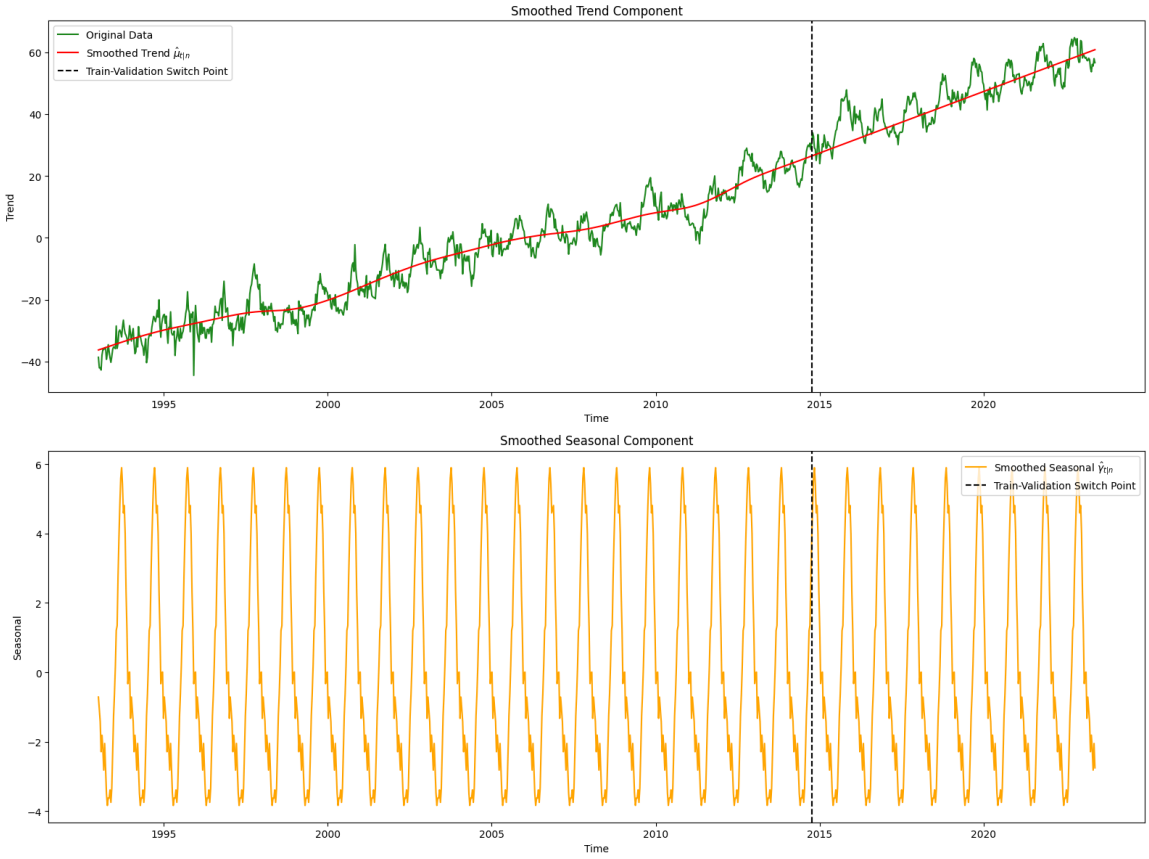
1. Trend: $\hat{\mu}_{t|n} = \mathbb{E}[\mu_t | y_{1:n}]$ for $t = 1, \ldots, n$
2. Seasonal: $\hat{\gamma}_{t|n} = \mathbb{E}[\gamma_t | y_{1:n}]$ for $t = 1, \ldots, n$

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

In [22]:
```
# Initialize arrays for storing smoothed trend (mu) and seasonal (gamma) compone
smooth_mu = np.zeros(n + m)
```

```
smooth_gamma = np.zeros(n + m)

# Loop over all time steps to compute smoothed estimates
for i in range(n + m):
    smooth_mu[i] = np.dot(Z_mu, kalman.alpha_sm[:k, :, i])     # Smoothed trend
    smooth_gamma[i] = np.dot(Z_gamma, kalman.alpha_sm[k:, :, i])  # Smoothed sea


plt.figure(figsize=(16, 12))

# Plot smoothed trend component
plt.subplot(2, 1, 1)
plt.plot(u, y, label="Original Data", color="forestgreen")
plt.plot(u, smooth_mu, label=r"Smoothed Trend $\hat{\mu}_{t|n}$", color='red')
plt.axvline(x=u[n], color="black", linestyle="--", label="Train-Validation Switc
plt.title('Smoothed Trend Component')
plt.xlabel('Time')
plt.ylabel('Trend')
plt.legend()

# Plot smoothed seasonal component
plt.subplot(2, 1, 2)
plt.plot(u, smooth_gamma, label=r"Smoothed Seasonal $\hat{\gamma}_{t|n}$", color
plt.axvline(x=u[n], color="black", linestyle="--", label="Train-Validation Switc
plt.title('Smoothed Seasonal Component')
plt.xlabel('Time')
plt.ylabel('Seasonal')
plt.legend()

plt.tight_layout()
plt.show()
```

```
C:\Users\sfurk\AppData\Local\Temp\ipykernel_14716\1321108367.py:7: DeprecationW
arning: Conversion of an array with ndim > 0 to a scalar is deprecated, and wil
l error in future. Ensure you extract a single element from your array before p
erforming this operation. (Deprecated NumPy 1.25.)
  smooth_mu[i] = np.dot(Z_mu, kalman.alpha_sm[:k, :, i])     # Smoothed trend
C:\Users\sfurk\AppData\Local\Temp\ipykernel_14716\1321108367.py:8: DeprecationW
arning: Conversion of an array with ndim > 0 to a scalar is deprecated, and wil
l error in future. Ensure you extract a single element from your array before p
erforming this operation. (Deprecated NumPy 1.25.)
  smooth_gamma[i] = np.dot(Z_gamma, kalman.alpha_sm[k:, :, i])  # Smoothed seas
onal
```



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**

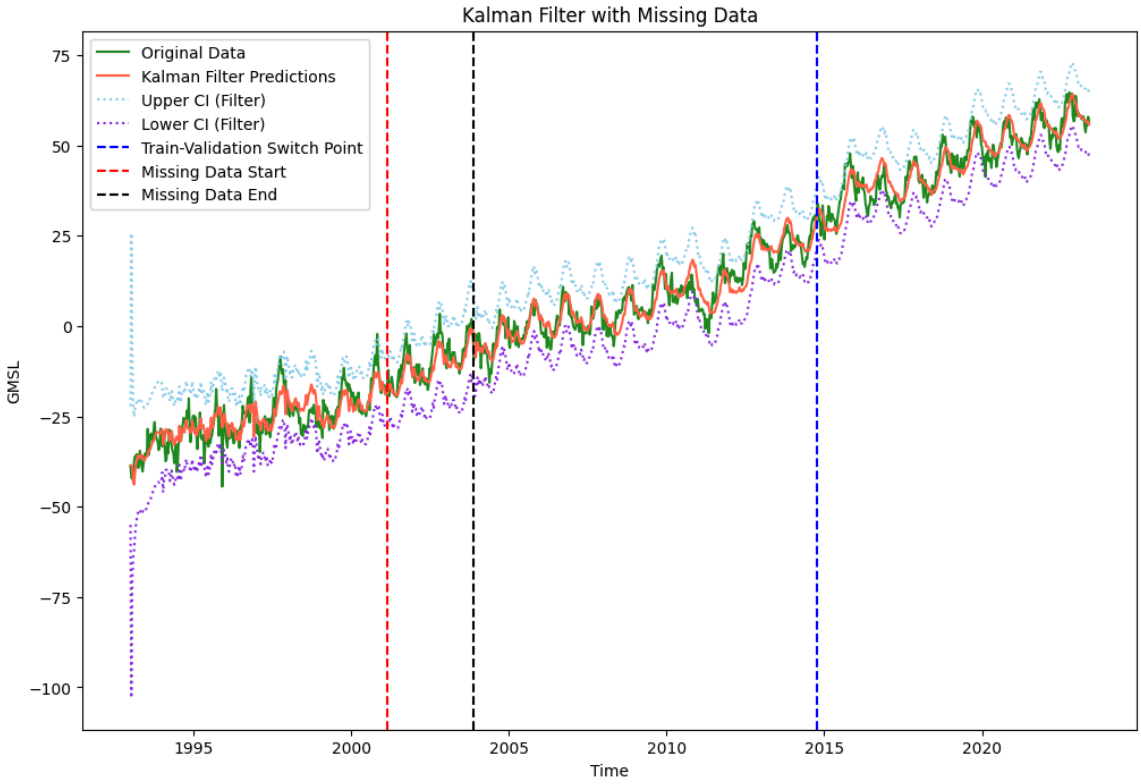
Comment on the qualitative differences between the filter and smoother estimates and explain what you see (in a couple of sentences).

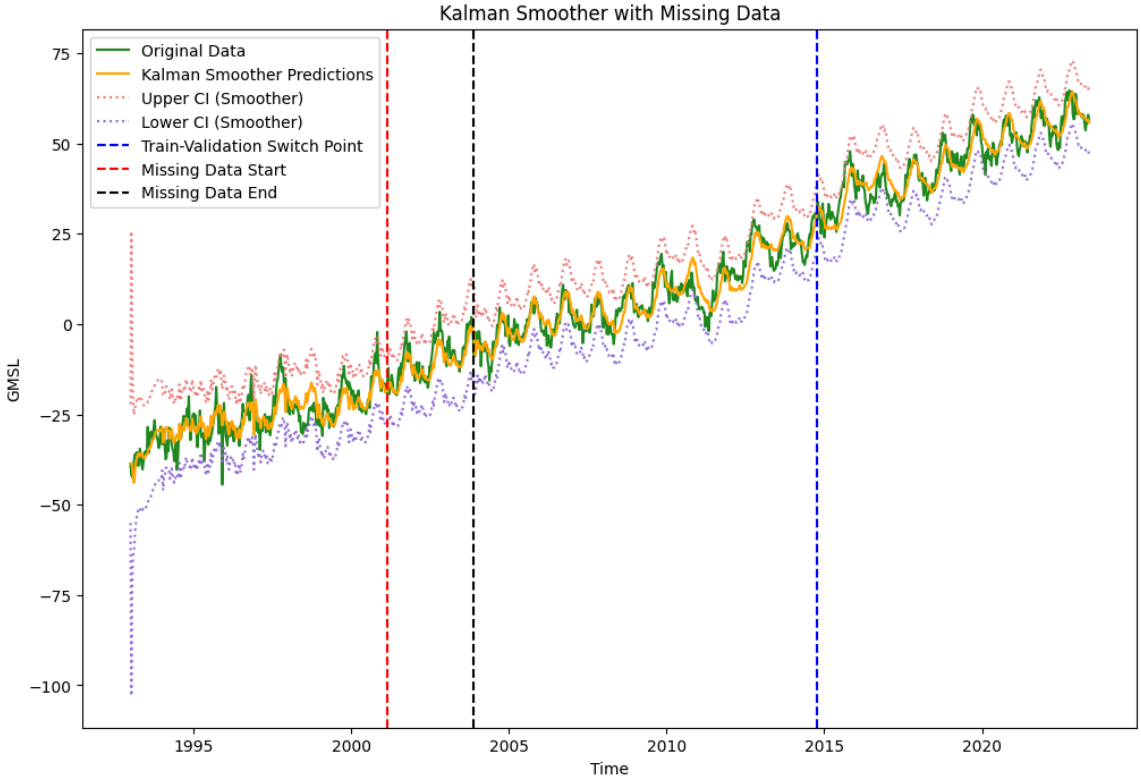
A17: It is expected that the predictions and uncertainty from both the Kalman filter and Kalman smoother will look almost identical since neither has future data to update predictions.

In [23]:
```
# Modify the data
y_missing = np.copy(y)
y_missing[300:401] = np.nan  # Set the specified range to missing

# Kalman filter
kalman_missing = kalman_filter(y_missing, model)

# Confidence Intervals for Kalman Filter predictions
kf_upper_CI = kalman_missing.y_pred + np.sqrt(kalman_missing.F_pred)
kf_lower_CI = kalman_missing.y_pred - np.sqrt(kalman_missing.F_pred)
```

```
# Plot Kalman Filter predictions with missing data
plt.plot(u, y, label="Original Data", color="forestgreen")
plt.plot(u, kalman_missing.y_pred, label="Kalman Filter Predictions", color="tom
plt.plot(u, kf_upper_CI, label="Upper CI (Filter)", color="skyblue", linestyle="
plt.plot(u, kf_lower_CI, label="Lower CI (Filter)", color="blueviolet", linestyl
plt.axvline(x=u[n], color="blue", linestyle="--", label="Train-Validation Switch
plt.axvline(x=u[300], color="red", linestyle="--", label="Missing Data Start")
plt.axvline(x=u[400], color="black", linestyle="--", label="Missing Data End")
plt.title("Kalman Filter with Missing Data")
plt.xlabel("Time")
plt.ylabel("GMSL")
plt.legend()
plt.show()

# Kalman smoother
kalman_smooth_missing = kalman_smoother(y_missing, model, kalman_missing)

# Confidence Intervals for Kalman Smoother predictions
ks_upper_CI = kalman_smooth_missing.y_pred + np.sqrt(kalman_smooth_missing.F_pre
ks_lower_CI = kalman_smooth_missing.y_pred - np.sqrt(kalman_smooth_missing.F_pre

# Plot Kalman Smoother predictions with missing data
plt.plot(u, y, label="Original Data", color="forestgreen")
plt.plot(u, kalman_smooth_missing.y_pred, label="Kalman Smoother Predictions", c
plt.plot(u, ks_upper_CI, label="Upper CI (Smoother)", color="lightcoral", linest
plt.plot(u, ks_lower_CI, label="Lower CI (Smoother)", color="mediumpurple", line
plt.axvline(x=u[n], color="blue", linestyle="--", label="Train-Validation Switch
plt.axvline(x=u[300], color="red", linestyle="--", label="Missing Data Start")
plt.axvline(x=u[400], color="black", linestyle="--", label="Missing Data End")
plt.title("Kalman Smoother with Missing Data")
plt.xlabel("Time")
plt.ylabel("GMSL")
plt.legend()
plt.show()
```

```
C:\Users\sfurk\AppData\Local\Temp\ipykernel_14716\1836612437.py:42: Deprecation
Warning: Conversion of an array with ndim > 0 to a scalar is deprecated, and wi
ll error in future. Ensure you extract a single element from your array before
performing this operation. (Deprecated NumPy 1.25.)
  y_pred[t] = np.dot(Z,alpha_pred[:,:,t])
C:\Users\sfurk\AppData\Local\Temp\ipykernel_14716\1836612437.py:43: Deprecation
Warning: Conversion of an array with ndim > 0 to a scalar is deprecated, and wi
ll error in future. Ensure you extract a single element from your array before
performing this operation. (Deprecated NumPy 1.25.)
  F_pred[t] = np.dot(Z,np.dot(P_pred[:,:,t],np.transpose(Z))) + H
```





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

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

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 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

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

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 the method to estimate also the AR coefficients of the ν-component, which is beyond the scope of this lab assignment.