Coursework - Riccardo Petrella

1) Download the Nile data, available in the R dataset: and fit a LLM to the data. Compare your results with Table 2.1.

Remark:

Python has been used to perform this time series analysis. There is no problem to retrieve the Nile dataset in Python since there exists a function named "get_rdataset" that allows us to get all the datasets from R-studio.

List of required libraries

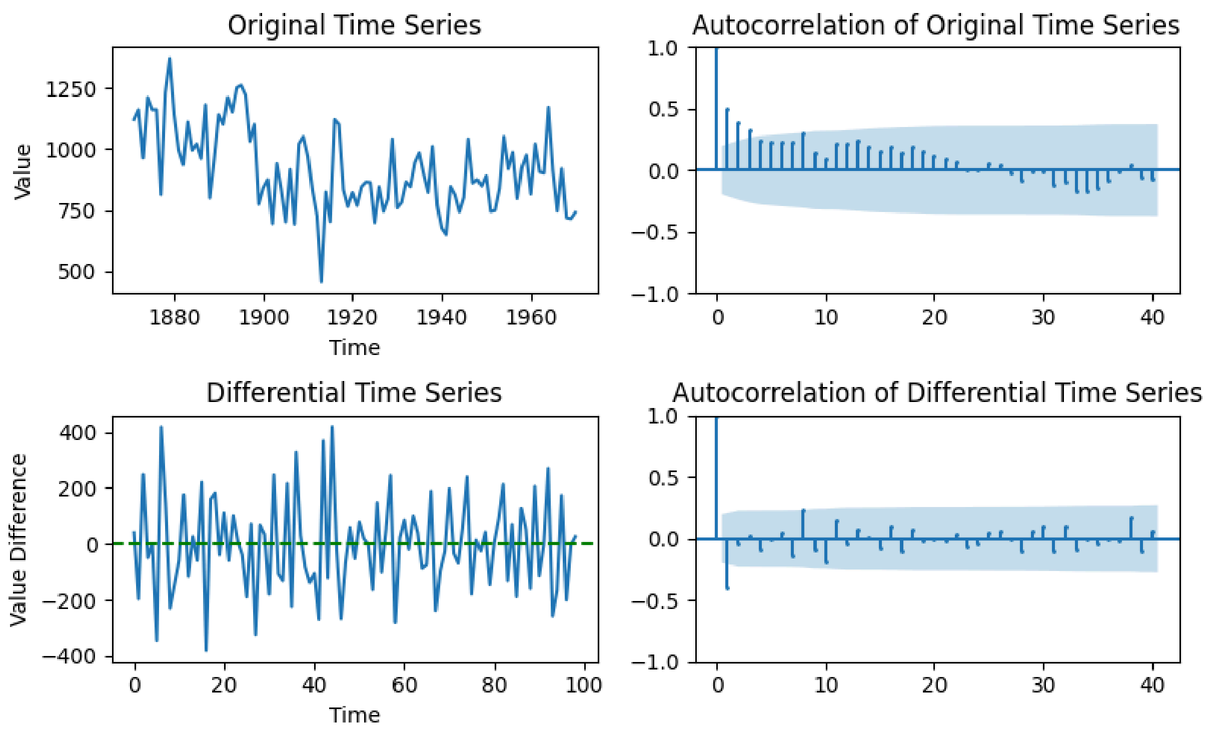
We import the need libraries

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import warnings
import statsmodels.api as sm

from scipy.optimize import minimize
from math import sqrt
from statsmodels.datasets import get_rdataset
from statsmodels.graphics.tsaplots import plot_acf
```

Dowload of the dataset and its graph

In [2]:
```
# to get the Nile dataset from R's dataset repository
data = get_rdataset('Nile').data

# Convert the 'value' column of the data into a numpy array
y = np.array(data['value'])
n = len(y)

# Plotting the original time series
plt.figure(figsize=(8, 5))
plt.subplot(221)
plt.plot(data['time'], y)
plt.title('Original Time Series')
plt.xlabel('Time')
plt.ylabel('Value')

# Plotting the autocorrelation of the original time series
plt.subplot(222)
plot_acf(y, ax=plt.gca(), lags=40, markersize = 1)
plt.title('Autocorrelation of Original Time Series')
```

```python
# Calculating and plotting the differential time series
y_diff = np.diff(y)
plt.subplot(223)
plt.plot(y_diff)
plt.axhline(0, color='green', linestyle='--')
plt.title('Differential Time Series')
plt.xlabel('Time')
plt.ylabel('Value Difference')

# Plotting the autocorrelation of the differential time series
plt.subplot(224)
plot_acf(y_diff, ax=plt.gca(), lags=40, markersize=1)
plt.title('Autocorrelation of Differential Time Series')

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



1. **Original Time Series Analysis**

- Plot Observations: The original time series plot shows fluctuations within a certain range, but there is no clear trend or seasonality apparent from the plot itself. The values oscillate around what appears to be a constant mean level.

- Autocorrelation: The autocorrelation plot reveals that the autocorrelation values decrease gradually as the lag increases, but there are significant autocorrelations at specific lags. This slow decay of autocorrelation suggests some level of long-term dependency in the data.

- Stationarity: The absence of a visible trend or seasonality in the original time series plot suggests that the mean might be stationary. However, the presence of significant autocorrelation at higher lags could also imply non-stationarity.

- Variance: The plot doesn't show any clear signs of changing variance, since there are not periods of increased or decreased fluctuations in the time series amplitude.

2. **Differential Time Series Analysis** $(1-L) \cdot y_t$

- Plot Observations: The differential time series, which plots the differences $y_t - y_{t-1}$, appears more erratic with what looks to be an approximate zero mean. This transformation is used to remove trends and achieve stationarity.

- Autocorrelation: The autocorrelation of the differential series is markedly lower than the original, with all autocorrelation coefficients close to zero beyond the immediate first lag. This suggests that differencing has successfully removed autocorrelations, pointing toward stationarity.

- Stationarity: The differential time series is likely to be stationary, with mean and autocorrelations that do not depend on time, indicated by the autocorrelation plot where correlations quickly fall to near zero.

- Variance: The variance in the differential time series appears to be constant over time, although the series itself is noisy, indicating frequent and significant shifts from one value to the next.

- Gaussianity: The more random and noisy appearance of the differential time series compared to the original could imply a more Gaussian-like behavior, but further statistical testing would be necessary to confirm this (for instance the Q-Q plot).

Conclusion:

The original Nile time series exhibits some characteristics of stationarity in mean but shows dependency over time in autocorrelations. Differencing the data results in a series that seems to be stationary in both mean and variance, with minimal autocorrelation, suggesting that it could be well-suited for modeling using ARIMA-type models if the goal is forecasting or further analysis.

Functions

Before applying working directly on the dataset, first, we are going to define the necessary functions to perform the time series analysis, listed as follows:

- the Kalman filter for a LLM

- the likelihood

- the estimator function.

Kalman filter

As introduced at 2.15 in the book, the Kalman filter for a LLM appears as follows:

$$\begin{aligned} v_t &= y_t - a_t, & F_t &= P_t + \sigma_\varepsilon^2, \\ a_{t|t} &= a_t + K_t v_t, & P_{t|t} &= P_t\left(1 - K_t\right), \\ a_{t+1} &= a_t + K_t v_t, & P_{t+1} &= P_t\left(1 - K_t\right) + \sigma_\eta^2, \end{aligned}$$

where F_t is the variance of the prediction error v_t and K_t is known as the Kalman gain $K_t = P_t/F_t$, P_t as the state variance, $a_{t|t}$ as the filtered estimator of the state α_t and $a_t + 1$ as the one-step ahead predictor of α_{t+1}.

Please notice that the function has step-by-step comments.

In [3]:
```
# Set the random seed to ensure reproducible results
np.random.seed(0)

def KF(y, q):
    # Initialize the Kalman filter's initial state mean and variance
    m10 = y[0]  # Initial state estimate taken as the first observation
    P10 = 1 + q  # Initial state variance, adjusted by signal-to-noise ratio q

    # Initialize variables
    n = len(y)  # Number of observations
    mu_pred = np.empty(n)  # Array to store the predicted state estimates
    P = np.zeros(n)  # Array to store the predicted state variances
    v = np.empty(n)  # Array to store the prediction error
    K = np.zeros(n)  # Array to store the Kalman gain values
    F = np.zeros(n)  # Array to store the variances of the prediction error
    dllk = np.empty(n)  # Array with components of log-likelihood increment
    llk = 0  # Variable to accumulate log-likelihood

    # Set initial predictions
    mu_pred[1] = m10
    P[1] = P10

    # Kalman filter recursion from the second observation to the second-last
    for t in range(1, n-1):
        v[t] = y[t] - mu_pred[t]  # Measurement residual
        F[t] = P[t] + 1  # Prediction error variance
        K[t] = P[t] / F[t]  # Kalman gain
        P[t+1] = P[t] * (1 - K[t]) + q  # Update estimate variance
        mu_pred[t+1] = mu_pred[t] + K[t] * v[t]  # Update state estimate
        dllk[t] = -0.5 * np.log(F[t])  # Contribution to the log-likelihood



    # In order to define sigma_e_hat and the log likelihood
    # we retrieve the last elements of the for loop

    F[n-1] = P[n-1] + 1
    K[n-1] = P[n-1] / F[n-1]
    dllk[n-1] = -0.5 * np.log(F[n-1])
    v[n-1] = y[n-1] - mu_pred[n-1]

    # Estimate measurement noise variance
    sigma_e_hat = np.sum(v[1:] ** 2 / F[1:]) / (n - 1)

    # Total log-likelihood
    llk = -0.5 * (n-1) * np.log(sigma_e_hat) + np.sum(dllk[1:])

    return {'mu_pred': mu_pred, 'llk': llk, 'v': v, 'F': F, 'P': P,
            'sigma_e_hat': sigma_e_hat}
```

Loglikelihood

From the book it is defined in 2.58 as shown here:

$$\log L = \log p\left(Y_n\right) = -\frac{n}{2}\log(2\pi) - \frac{1}{2}\sum_{t=1}^{n}\left(\log F_t + \frac{v_t^2}{F_t}\right).$$

The function "loglikelihood", calculates the negative log-likelihood of a model parameterized by "par" given the data "y".

In [4]:
```
def loglikelihood(par, y):
    q_new = par[0]
    obj = KF(y, q_new)['llk']
    return -obj
```

Estimator

In the next chunk, the estimator/optimisation function is defined. This function defines an estimator to find the optimal model parameters.

Remark:

The function which performs the minimisation "minimize" used in the book, works with the 'BFGS' algorithm. Since after some trials the method 'BFGS' has yielded worse results, I have used the 'Nelder-Mead' method which yields closer results to the ones from the table 2.1. If in the argument 'method' we put 'BFGS', the algorithm converges but not with the same result of the table.

In [5]:
```
def estimator(y, par):
    n = len(y)  # Length of the dataset y
    q_0 = par   # Initial guess for the parameter q

    # Use the `minimize` function to find the parameter that minimizes
    # the negative loglikelihood.
    # The minimize function is called with the initial guess q_0, the data y,
    # and using the Nelder-Mead algorithm
    hat = minimize(loglikelihood, q_0, args=(y,), method='Nelder-Mead')
```

```python
    # set method='BFGS' to comply with the book

    # Extract the optimized parameter value from the optimization result
    q_hat = hat.x[0]

    # Collect and return results including the estimated parameter 'q_hat'
    # and the number of iterations 'nit'
    theta_list = {'q_hat': q_hat, 'iter': hat.nit}  # Store the estimated
    # parameter and iteration count
    out = {'theta_list': theta_list}  # Structure the output
    return out
```

Apply the functions

In [6]:
```python
# Suppress warnings that might arise during execution
warnings.filterwarnings('ignore')

# random seed to ensure reproducible results
np.random.seed(0)


# Set an initial guess for the parameter to be estimated
q_0 = 1

# Call the estimator function with the dataset and initial guess
# to find the optimal parameter
est = estimator(y, q_0)

# Extract the estimated parameter q_hat from the results
q_hat = est['theta_list']['q_hat']

# Apply the Kalman Filter with the estimated parameter
filter_output = KF(y, q_hat)

# Calculate sigma_e_hat, an estimate of the noise variance
sigma_e_hat = filter_output['sigma_e_hat']
#sigma_e_hat = np.sum(v[1:] ** 2 / F[1:]) / (n - 1) from the kalman filter


# Calculate sigma_eta_hat, an estimate of the signal variance
sigma_eta_hat = q_hat * sigma_e_hat


print("The value of sigma_e_hat is ", sigma_e_hat)
print("The value of sigma_eta_hat is ", sigma_eta_hat)
print("The value of q_hat is ", q_hat)
print("The value of llk is", filter_output['llk'])
```

```
The value of sigma_e_hat is  15099.46118346705
The value of sigma_eta_hat is  1468.65852917315
The value of q_hat is  0.09726562499999919
The value of llk is -492.0707103998211
```

Comments

From the out put, we notice that the log likelihood is the same as shown in table 2.1 as well as q_hat. Both sigma_e_hat and sigma_eta_hat comply with the initialised values from the book.

2) Reproduce the Simulation example on Section 2.6.

To reproduce the example on section 2.6 according to the book, we need to define a list of functions:

- data generating process (dgp)
- a brand new kalman Filter.

Function: dgp

This Python function, dgp, is designed to simulate a data generating process (DGP) for a simple time series model, where the observations are generated from a latent (hidden) state that evolves over time. The model includes both state noise (eta) and observation noise (e).

In [7]:
```python
def dgp(n, sigma_e, sigma_eta):

    # Generate state noise 'eta' from a normal distribution, scaled by
    # the square root of sigma_eta
    eta = np.sqrt(sigma_eta) * np.random.normal(0, 1, n)

    # Generate observation noise 'e' from a normal distribution, scaled by
    # the square root of sigma_e
    e = np.sqrt(sigma_e) * np.random.normal(0, 1, n)

    # Initialize arrays for the latent state 'alpha' and the observed series 'y'
    alpha = np.zeros(n)
    y = np.zeros(n)

    # Set initial values for 'alpha' and 'y' at time 0
    alpha[0] = 1120  # Initial latent state
    y[0] = 1120      # Initial observed value

    # Recursive equations to generate the time series as a LLM
    for t in range(0, n-1):
        alpha[t+1] = alpha[t] + eta[t]  # Update state with state noise
        y[t+1] = alpha[t+1] + e[t+1]   # Obs. value includes state and obs. noise

    # Collect results into a dictionary for output
    out = {'y': np.asarray(y), 'alpha': np.asarray(alpha), 'e': e}

    return out

# results
result = dgp(n, sigma_e_hat, sigma_eta_hat)
```

```
y_plus = result['y']  # Simulated observed data
e_plus = result['e']  # Simulated observation noise
```

Function: KFL

To find the Kalaman filter vectors we need to define again a function that performs a Kalman filter iteration. It is worth mentioning here the expression in 2.30 of the book where L_t is defined:

$$L_t = 1 - K_t = \sigma_\varepsilon^2 / F_t.$$

This Python function, KFL, extends the Kalman Filter by introducing an additional output term L that represents the adjustment factor (1 minus the Kalman gain). This version of the filter calculates predictions, updates, and variances in the presence of both state noise (sigma_eta) and measurement noise (sigma_e).

In [8]:
```
def KFL(y, sigma_e, sigma_eta):

    # Initialize starting values
    m10 = y[0]  # initial predicted state equal to the first obs
    p10 = sigma_e  # Initial predicted variance

    n = len(y)  # Number of observations
    # Preallocate arrays
    mu_pred = np.zeros(n)  # Predicted mean (state estimate)
    P = np.zeros(n)  # Variance of the predicted state
    v = np.zeros(n)  # prediction error
    K = np.zeros(n)  # Kalman gain
    F = np.zeros(n)  # variance of prediction error
    L = np.zeros(n)  # Adjustment factor L

    # Set initial predictions
    mu_pred[0] = m10
    P[0] = p10

    # Kalman filter recursion
    for t in range(0, n-1):
        v[t] = y[t] - mu_pred[t]  # Calculate forecast error
        F[t] = P[t] + sigma_e  # Update var of pred. error with meas. noise
        K[t] = P[t] / F[t]  # Update Kalman gain
        P[t+1] = P[t] + sigma_eta - K[t]*F[t]*K[t]  # Update state variance
        mu_pred[t+1] = mu_pred[t] + K[t]*v[t]  # Update state prediction
        L[t] = 1 - K[t]  # Calculate L factor as 1 minus Kalman gain

    # Compute final values for the last time point
    F[n-1] = P[n-1] + sigma_e
    K[n-1] = P[n-1] / F[n-1]
    v[n-1] = y[n-1] - mu_pred[n-1]
    L[n-1] = 1 - K[n-1]

    # Output dictionary containing all relevant arrays
    out = {'mu_pred': mu_pred, 'v': v, 'F': F, 'K': K, 'P': P, 'L': L}
    return out
```

Application of the functions to Nile

Now we apply the Kalman filter with L to the values of the Nile dataset and we name it "filter2".

In [9]:
```
filter2 = KFL(y, sigma_e_hat, sigma_eta_hat)
```

Computation of the recursion for r_t and u_t.

The backwards recursion r_t is defined in expression 2.36, namely:

$$r_{t-1} = \frac{v_t}{F_t} + L_t \cdot r_t \quad r_n = 0$$

for $\quad t = n, n-1, \ldots, 1.$

In [10]:
```
# Retrieve elements from the result of
# the filtering from previous steps

v = filter2['v']  # prediction error
F = filter2['F']  # Prediction error variances
L = filter2['L']  # Adjustment factors (1 - Kalman gain)

# Initialize the backward recursion array
r2 = np.zeros(n)

# Backwards recursion to compute r_t
# The loop starts from the second to last observation and
# goes backwards to the first
for t in range(n-2, 0, -1):
    r2[t-1] = v[t] / F[t] + L[t] * r2[t]

# r2 now contains the series resulting from the backward recursion
```

Here we define the recursion of u_t (smoothing error) as in expression 2.45, namely:

$$u_t = \frac{v_t}{F_t} - K_t \cdot r_t$$

In [11]:
```
# Retrieve the Kalman gain from the previous filtering

K = filter2['K']  # Kalman gains

# Initialize the array for smoothing error
u2 = np.zeros(n)

# Recursion of the smoothing error u_t
for t in range(1, n-1):
    u2[t] = v[t] / F[t] - K[t] * r2[t]
```

```
# u2 now contains the series of smoothing errors
```

Now it's possible to define the smoothed observation disturbance $\hat{\varepsilon}_t = \mathrm{E}\left(\varepsilon_t \mid Y_n\right)$ is computed as follows (2.44):

$$\hat{\varepsilon}_t = \sigma_{\varepsilon}^2 \cdot u_t, \quad t = n, \ldots, 1$$

where σ_{ε}^2 is the "sigma_e_hat" that we have found with the first Kalman filter.

In [12]:
```
# e_hat is the smoothed observation disturbance
# e_hat = smoothing error * Estimate measurement noise variance
e_hat = u2 * sigma_e_hat
```

As before we apply the Kalman filter with L to the sample $Y_n^+ = \left(y_1^+, \ldots, y_n^+\right)'$ with the estimated noises and we call it "filter3".

In [13]:
```
filter3 = KFL(y_plus, sigma_e_hat, sigma_eta_hat)
```

- KFL Function: This function is a modified version of the Kalman Filter that additionally computes and returns an adjustment factor L, along with other typical outputs such as predicted states, variances, and residuals. The function uses parameters for the process noise and measurement noise.
- y_plus: This is the dataset obtained from the data generating process (dgp) which simulates new observations based on specified noise characteristics and model dynamics. y_plus is generated to test the behavior of the filter under controlled conditions, using the noise variance parameters sigma_e_hat and sigma_eta_hat.

Now we proceed to compute r_t and u_t.

For r_t we use once again the formula (2.37) applied to "filter3":

In [14]:
```
v = filter3['v']
F = filter3['F']
L = filter3['L']
r3 = np.zeros(n)

for t in range(n-2, 0, -1):
    r3[t-1] = v[t] / F[t] + L[t] * r3[t]
```

Same for u_t from (2.45):

In [15]:
```
v = filter3['v']
F = filter3['F']
K = filter3['K']
u3 = np.zeros(n)
```

```
for t in range (1, n-1):
    u3[t] = v[t]/F[t] - K[t] * r3[t]
```

Defining the following variables as shown below:

- $$\varepsilon_t^+ \sim \mathrm{N}\left(0, \sigma_\varepsilon^2\right), \quad \eta_t^+ \sim \mathrm{N}\left(0, \sigma_\eta^2\right), \quad t = 1, \ldots, n \quad (2.51)$$
- $$y_t^+ = \alpha_t^+ + \varepsilon_t^+, \quad \alpha_t^+ = y_t^+ - \varepsilon_t^+, \quad t = 1, \ldots, n \quad (2.52)$$
- $$\tilde{\varepsilon}_t = \varepsilon_t^+ - \hat{\varepsilon}_t^+ + \hat{\varepsilon}_t, \quad (2.53)$$
- $$\hat{\alpha} = a_t + P_t \cdot r_{t-1} \quad (2.33)$$
- $$\tilde{\alpha}_t = y_t - \tilde{\varepsilon}_t \quad (2.53)$$

In [16]:
```
e_hat_plus = u3 * sigma_e_hat  # estimated observation error scaled with
# the measurement noise variance sigma_e_hat.

e_tilde = e_plus - e_hat_plus + e_hat # Adjusted Observation Error

alpha_tilde = y - e_tilde    # Adjusted State Estimate

# since in the simulation y (and not y_plus) is used, we
# retrieve the values of mu_pred from filter2 # (and not filter3 with y_plus)

alpha = pd.Series(filter2['mu_pred'])

# initialisation of the smoothed state
alpha_hat = np.zeros(n)
alpha_hat[0] = alpha[0]
alpha_hat[1] = alpha[1]

# recursion of the smoothed state
for t in range(1, n):
    alpha_hat[t] = alpha[t] + filter2['P'][t] * r2[t-1]

# true state alpha_plus with the recursion
alpha_plus = np.zeros(n)
alpha_plus[0] = 1120  # first known value of the series
for t in range(1, n):
    alpha_plus[t] = y_plus[t] - e_plus[t]
```

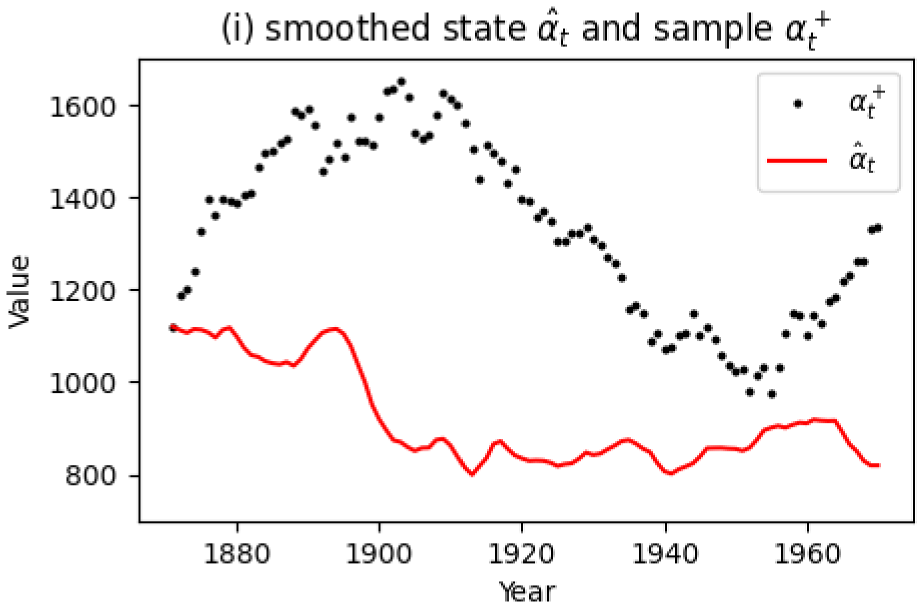
Finally here are defined as:

- $$\hat{\eta}_t = \hat{\alpha}_{t+1} - \hat{\alpha}_t$$
- $$\tilde{\eta}_t = \tilde{\alpha}_{t+1} - \tilde{\alpha}_t \quad (2.53)$$

In [17]:
```
eta_hat = np.diff(alpha_hat)
eta_tilde = np.diff(alpha_tilde)
```

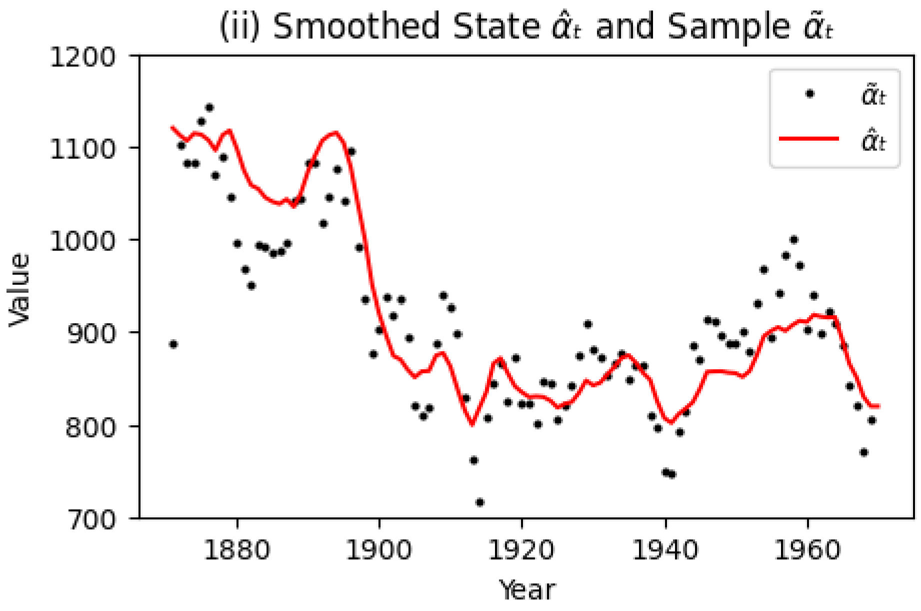
Comparison with graphs from Fig. 2.4 from Simulation 2.6

```
In [18]:
# graph i (alpha_hat and alpha_plus)
plt.figure(figsize=(5, 3))
plt.plot(data['time'], alpha_plus, 'ko', label = '$α_t^+$', markersize = 2)
plt.ylim(700, 1700)

plt.plot(data['time'], alpha_hat, color='red', label = '$\hat{α}_t$')

plt.xlabel('Year')
plt.ylabel('Value')

plt.title('(i) smoothed state $\hat{α}_t$ and sample $α_t^+$')
plt.legend()

plt.show()
```



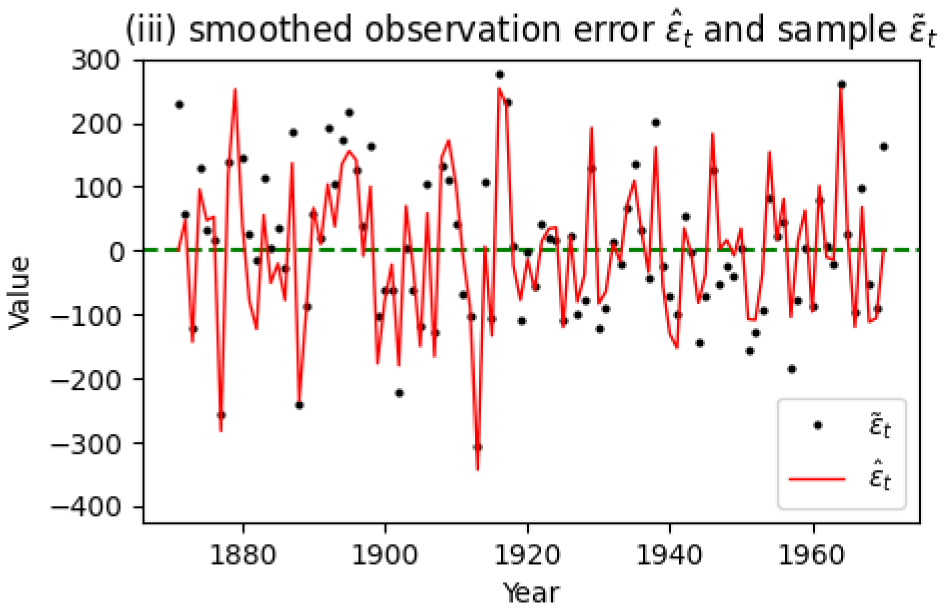
By comparing graph (i) with the one from the book, we clearly notice that $\hat{\alpha}_t$ is identical and α_t^+ is very close to the simulation since the pattern is the same but around 1910 the peak is a bit higher in my experiment.

```
In [19]:
# Graph ii (alpha_tilde and alpha_hat)
plt.figure(figsize=(5,3))
# Your plotting code


plt.plot(data['time'], alpha_tilde, 'ko', label='$αₜ$', markersize = 2)
plt.ylim(700, 1200)
plt.plot(data['time'], alpha_hat, color='red', label='$αₜ$')
plt.xlabel('Year')
plt.ylabel('Value')

plt.title('(ii) Smoothed State $αₜ$ and Sample $αₜ$')
plt.legend()
plt.show()
```



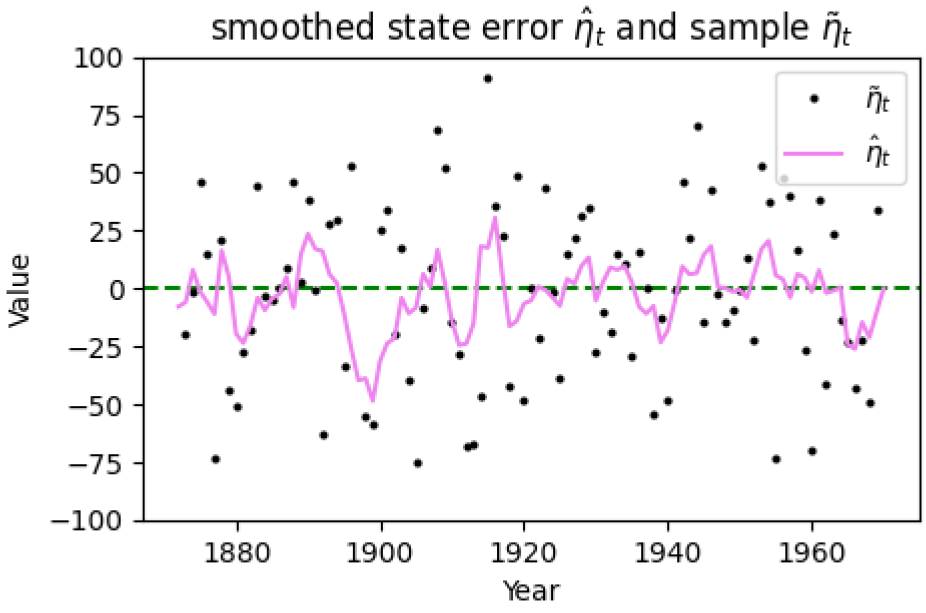
There is not much to say other than the fact that (ii) looks pretty much the same. The only exception is a higher peak of $\tilde{\alpha}_t$ in 1960 with respect to the one from the book.

In [20]:
```
# graph iii (e_hat and e_tilde)

plt.figure(figsize=(5,3))
plt.axhline(np.mean(e_hat), color = 'green', linestyle='--')
plt.plot(data['time'], e_tilde, 'ko', markersize = 2, label='$ε˜_t$')
plt.ylim(-425, 300)

plt.plot(data['time'], e_hat, color='red', linewidth = 1, label='$\hat{ε}_t$')
plt.xlabel('Year')
plt.ylabel('Value')

plt.title('(iii) smoothed observation error $\hat{ε}_t$ and sample $ε˜_t$')

plt.legend()
plt.show()
```



In (iii) there is no detectable difference by eye.

In [21]:
```
# Graph iiii (eta_hat and eta_tilde)
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(5,3))
plt.axhline(0, color='green', linestyle='--')

plt.plot(data.iloc[1:,0], eta_tilde, 'ko', markersize = 2, label = '$η˜_t$')
plt.ylim(-100, 100)

plt.plot(data.iloc[1:,0], eta_hat, color='violet', label = '$\hat{η}_t$')
plt.xlabel('Year')
plt.ylabel('Value')
plt.legend()
plt.title('smoothed state error $\hat{η}_t$ and sample $η˜_t$')
# Displaying the plot
plt.show()
```



Finally, (iiii) is also identical to the one from the simulation.

We can conclude that the simulation has been correctly reproduced.

3) Choose a time series over a period of your choice.

Name of the Time Series:

Average Sales Price of Houses Sold for the United States (ASPUS). (The data are expressed in Dollars).

Period:

From 1963-01-01 to 2024-01-01

Frequency:

Quarterly

Sources:

U.S. Census Bureau and U.S. Department of Housing and Urban Development, Average Sales Price of Houses Sold for the United States [ASPUS], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/ASPUS, May 6, 2024.

4) Analyse the series fitting an appropriate linear Gaussian state space model.

Remark:

After uploading the time series and visualising the data (please see below), an upwards trend is clearly noticeable. Thus, as a first attempt, a Local Linear Trend model has been implemented. The LLT in the State Space Form appears as follows (as a special version of 2.31):

$$y_t = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} \mu_t \\ \nu_t \end{pmatrix} + \varepsilon_t,$$
$$\begin{pmatrix} \mu_{t+1} \\ \nu_{t+1} \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} \mu_t \\ \nu_t \end{pmatrix} + \begin{pmatrix} \xi_t \\ \zeta_t \end{pmatrix},$$

where:

$$\varepsilon_t \sim \mathrm{N}\left(0, \sigma_\varepsilon^2\right)$$
$$\xi_t \sim \mathrm{N}\left(0, \sigma_\xi^2\right)$$
$$\zeta_t \sim \mathrm{N}\left(0, \sigma_\zeta^2\right)$$

Below there is the code that executes the model fit and plots the series.

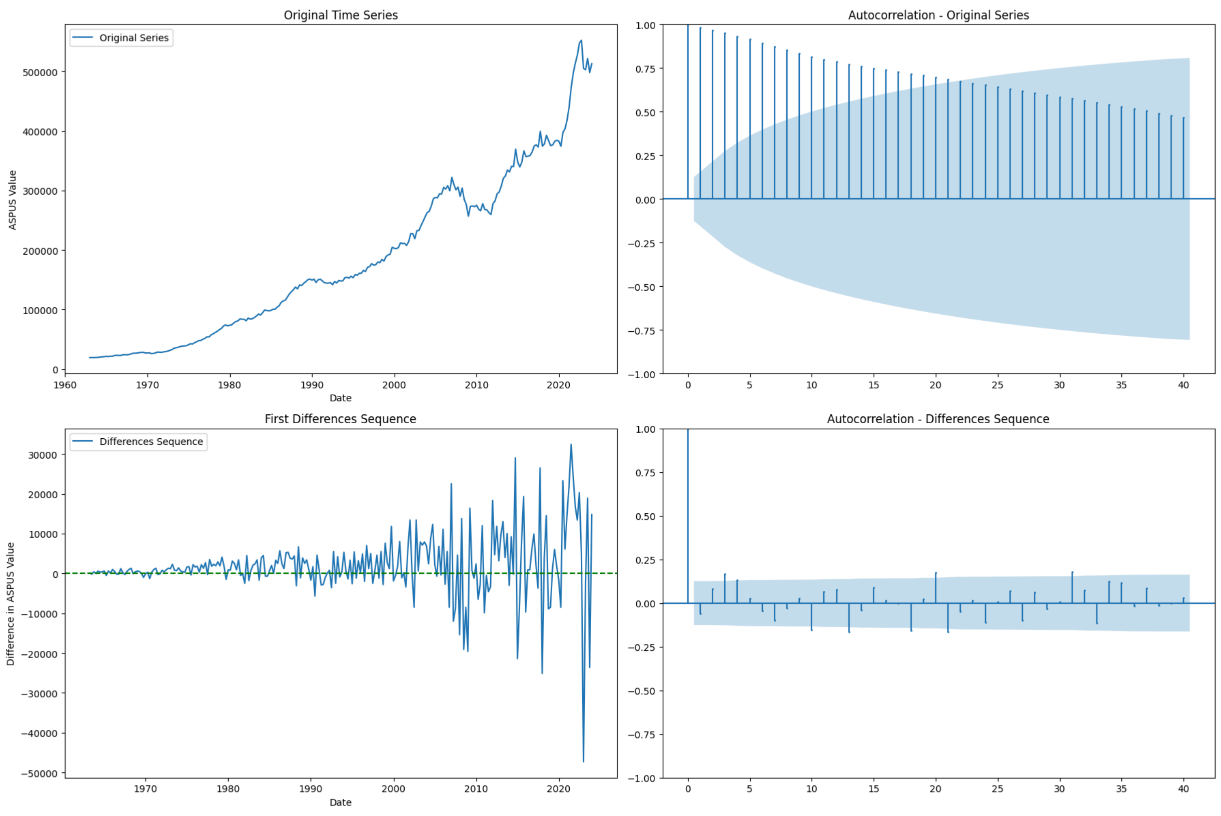
Code with the following functions:

- "load_and_preprocess_data": to load and convert the data into a dataframe.
- "plot_time_series": pretty self explainatory.
- "fit_llt_model": to fit the LLT and yield the estimated parameters and the negative llk.

In [22]:
```python
warnings.filterwarnings('ignore')


def load_and_preprocess_data(filepath):
    # Load the data
    data = pd.read_csv(filepath)
    # Convert DATE to datetime and set as index
    data['DATE'] = pd.to_datetime(data['DATE'])
    data.set_index('DATE', inplace=True)
    return data

def plot_series_and_acf(data):
    # Prepare figure for subplots
    fig, axes = plt.subplots(2, 2, figsize=(18, 12))

    # Plot original series
    # axes are al alternative way of (subplots)
    axes[0, 0].plot(data.index, data['ASPUS'], label='Original Series')
    axes[0, 0].set_title('Original Time Series')
    axes[0, 0].set_xlabel('Date')
    axes[0, 0].set_ylabel('ASPUS Value')
    axes[0, 0].legend()

    # Plot ACF of original series
    plot_acf(data['ASPUS'], ax=axes[0, 1], lags=40,
             title='Autocorrelation - Original Series', markersize = 1)

    # Calculate differences and plot differences sequence
    differences = data['ASPUS'].diff().dropna()

    axes[1, 0].plot(differences.index, differences, label='Differences Sequence')
    axes[1, 0].axhline(0, color='green', linestyle='--')
    axes[1, 0].set_title('First Differences Sequence')
    axes[1, 0].set_xlabel('Date')
    axes[1, 0].set_ylabel('Difference in ASPUS Value')
    axes[1, 0].legend()

    # Plot ACF of differences sequence
    plot_acf(differences, ax=axes[1, 1], lags=40,
             title='Autocorrelation - Differences Sequence', markersize = 1)

    plt.tight_layout()
    plt.show()



def fit_llt_model(data):
    # Fit a Local Linear Trend model
    llt_model = sm.tsa.UnobservedComponents(data['ASPUS'], 'local linear trend')
    llt_results = llt_model.fit()
    return llt_results
```

Graphs and comments

In [23]:
```
filepath = r'C:\Users\ricky\Downloads\ASPUS.csv'
data = load_and_preprocess_data(filepath)
plot_series_and_acf(data)
```



1. **Original Time Series**

- Trend and Stationarity: The original series exhibits a clear upward trend over time, which is a strong indication of non-stationarity in the mean.
- Autocorrelation Function: The ACF of the original series shows a gradual decline but remains positive over many lags, which reinforces the presence of a trend and suggests long-term positive autocorrelation. This pattern is typical for non-stationary data where past values have a prolonged influence on future values.

2. **Differential Time Series** $(1 - L) \cdot y_t$

- Stationarity: Differencing the series, results in a plot where the mean appears to hover around zero without any discernible trend. The visual absence of a trend suggests improved stationarity in the mean.
- Variance: The variance in the differenced series obviously appears non-constant over time. We can see that it is "exploding" from 2000 on. There is heteroschedasticity and Gaussianity should be excluded.
- Autocorrelation Function: The ACF for the differenced series quickly drops to near zero and fluctuates around this level. This rapid decay and low level of autocorrelation post-differencing suggest that the differencing process has effectively addressed the autocorrelation due to the trend, supporting the hypothesis of stationarity in the differenced series.

LLT model fit and summary

```
In [24]: llt_results = fit_llt_model(data)
         print("Summary of the LLT Model Fit:")
         print(llt_results.summary())
         print("\nLog Likelihood of the model:", llt_results.llf)  # Log Likelihood
         # Displaying the estimated variances of noise components
         print("\nEstimated Noise Components:")
         print("Irregular Variance (sigma2.irregular): {:.4f}".format(
             llt_results.params['sigma2.irregular']))
         print("Level Variance (sigma2.level): {:.4f}".format(
             llt_results.params['sigma2.level']))
         print("Trend Variance (sigma2.trend): {:.4f}".format(
             llt_results.params['sigma2.trend']))
```

```
Summary of the LLT Model Fit:
                        Unobserved Components Results
==============================================================================
Dep. Variable:                  ASPUS   No. Observations:                  245
Model:             local linear trend   Log Likelihood               -3101.571
Date:                Sun, 12 May 2024   AIC                           6209.141
Time:                        13:07:39   BIC                           6219.620
Sample:                    01-01-1963   HQIC                          6213.362
                         - 01-01-2024
Covariance Type:                  opg
====================================================================================
                       coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------------
sigma2.irregular  1.314e+08   1.39e+10      0.009      0.992   -2.72e+10    2.74e+10
sigma2.level      7.586e+06   1.15e+10      0.001      0.999   -2.26e+10    2.26e+10
sigma2.trend      1.849e+10    6.3e+10      0.294      0.769   -1.05e+11    1.42e+11
===================================================================================
Ljung-Box (L1) (Q):                  71.81   Jarque-Bera (JB):               364.77
Prob(Q):                              0.00   Prob(JB):                         0.00
Heteroskedasticity (H):              34.84   Skew:                            -0.61
Prob(H) (two-sided):                  0.00   Kurtosis:                         8.88
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-ste
p).

Log Likelihood of the model: -3101.57052228224

Estimated Noise Components:
Irregular Variance (sigma2.irregular): 131437586.9258
Level Variance (sigma2.level): 7586232.3839
Trend Variance (sigma2.trend): 18494651308.2033
```

The LLT model, applied to 245 observations from January 1963 to January 2024, assesses the components of variance in the time series data of ASPUS. The reported log likelihood of -3101.570 suggests the model's explanatory power given its parameters, though a more direct comparison with alternative models via AIC or BIC is beneficial for assessing model selection.

Noise Component Estimates

- Irregular Variance ($\sigma^2_{irregular}$): Estimated at approximately 131,437,587. This substantial variance indicates significant random fluctuations or noise at each observation, suggesting the data points vary widely from their trend and level components. The high standard error relative to the estimate suggests considerable uncertainty in this parameter's estimate.

- Level Variance (σ^2_{level}): Estimated at 7,586,232, which quantifies the noise in the underlying level or baseline of the series. While lower than the irregular variance, its large standard error similarly indicates high uncertainty in estimating this component, suggesting potential instability or shifts in the baseline level over time.

- Trend Variance (σ^2_{trend}): At approximately 18,494,651,308, this is a measure of the variability in the series' trend. This high variance suggests that the trend component is not stable and fluctuates significantly, which could be reflective of underlying structural changes affecting the series.

Diagnostics

- Ljung-Box Test: The statistic of 71.81 with a p-value near zero indicates significant autocorrelation in the residuals at lag 1, suggesting that the model does not capture all the autocorrelation structures within the data, which could potentially be improved by adding seasonal or other cyclic components.

- Jarque-Bera Test: The very high value of 364.77 with a p-value of 0 indicates that the residuals do not follow a normal distribution. They exhibit negative skew (-0.61) and high kurtosis (8.88), suggesting the presence of outliers or an asymmetric distribution of residuals.

- Heteroskedasticity: A value of 34.84 with a p-value of 0 indicates significant changes in the variance of residuals over time, which reveals that the series has a form of conditional heteroskedasticity.

Overall Assessment

The model reveals significant irregularities and trends in the data, with substantial variance components that indicate a complex underlying structure not captured by the current model configuration. The presence of significant autocorrelation, non-normality in residuals, and heteroskedasticity suggests that further refinement of the model is necessary.