Enhancing Energy Management: Predictive Modeling of Monthly Electricity Consumption

What is Time Series Analysis?

Time series analysis is a statistical technique used to analyze and interpret sequential data points, typically ordered by time. It involves studying the patterns of data points collected at regular intervals to forecast future trends or behaviors based on historical patterns.

Applications of Time Series Forecasting:

1. **Weather forecasting**
2. **Earthquake prediction**
3. **Astronomy**
4. **Statistics**
5. **Mathematical finance**
6. **Econometrics**
7. **Pattern recognition**
8. **Signal processing**
9. **Control engineering**

Time series forecasting utilizes various computational methods, including:

- **Machine learning**
- **Artificial neural networks**
- **Support vector machines**
- **Fuzzy logic**
- **Gaussian processes**
- **Hidden Markov models**

The primary objectives of time series analysis include identifying the underlying nature of observed phenomena and predicting future values of the time series variable. Achieving these goals involves recognizing patterns within the data and integrating these insights into predictive models.

Stages in Time Series Forecasting

Solving a time series problem requires specific stages, each serving a crucial role in the analysis and forecasting process.

1. Visualizing Time Series:

This initial step involves visually exploring the data to identify underlying patterns such as trends and seasonality. Understanding these patterns is essential before proceeding with further analysis.

2. Stationarizing Time Series:

A stationary time series exhibits consistent statistical properties over time, such as mean and variance. Transforming a series into a stationary form is often necessary for accurate forecasting, as many forecasting techniques assume stationarity.

3. Determining Model Parameters:

Using tools like Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF), optimal parameters for forecasting models (e.g., ARIMA) are determined. These parameters capture dependencies within the data that influence future predictions.

4. Fitting the Model:

With identified parameters, the chosen model (e.g., ARIMA) is applied to learn and capture the underlying patterns of the time series data. It aims to provide a mathematical representation of observed trends.

5. Making Predictions:

Finally, the fitted model is used to predict future values of the time series based on the established patterns. This step completes the forecasting process, allowing stakeholders to anticipate future trends and make informed decisions.

Each stage in time series analysis contributes to a structured approach in understanding and predicting sequential data patterns, ensuring robust and reliable forecasts for various applications.

In summary, time series analysis involves a systematic exploration of data patterns over time, leveraging statistical methods to forecast future outcomes based on historical observations.

In [33]:
```
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight') # special style template for matplotlib, highly useful for visualizing time se
from pylab import rcParams
rcParams['figure.figsize'] = 10, 7

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

```
/kaggle/input/electric-production/Electric_Production.csv
```

Load the dataset

In [34]:
```
df = pd.read_csv('../input/electric-production/Electric_Production.csv')
df
```

Out[34]:

	DATE	Value
0	01-01-1985	72.5052
1	02-01-1985	70.6720
2	03-01-1985	62.4502
3	04-01-1985	57.4714
4	05-01-1985	55.3151
...	...	...
392	09-01-2017	98.6154
393	10-01-2017	93.6137
394	11-01-2017	97.3359
395	12-01-2017	114.7212
396	01-01-2018	129.4048

397 rows × 2 columns

Now, Define column names, drop nulls, convert Date to DateTime format and make Date as an index column because it is not possible to plot the graph without index.

In [35]:
```
df.columns=['Date', 'Consumption']
df=df.dropna()
df['Date'] = pd.to_datetime(df['Date'])
df.set_index('Date', inplace=True) #set date as index
df.head()
```

Out[35]:

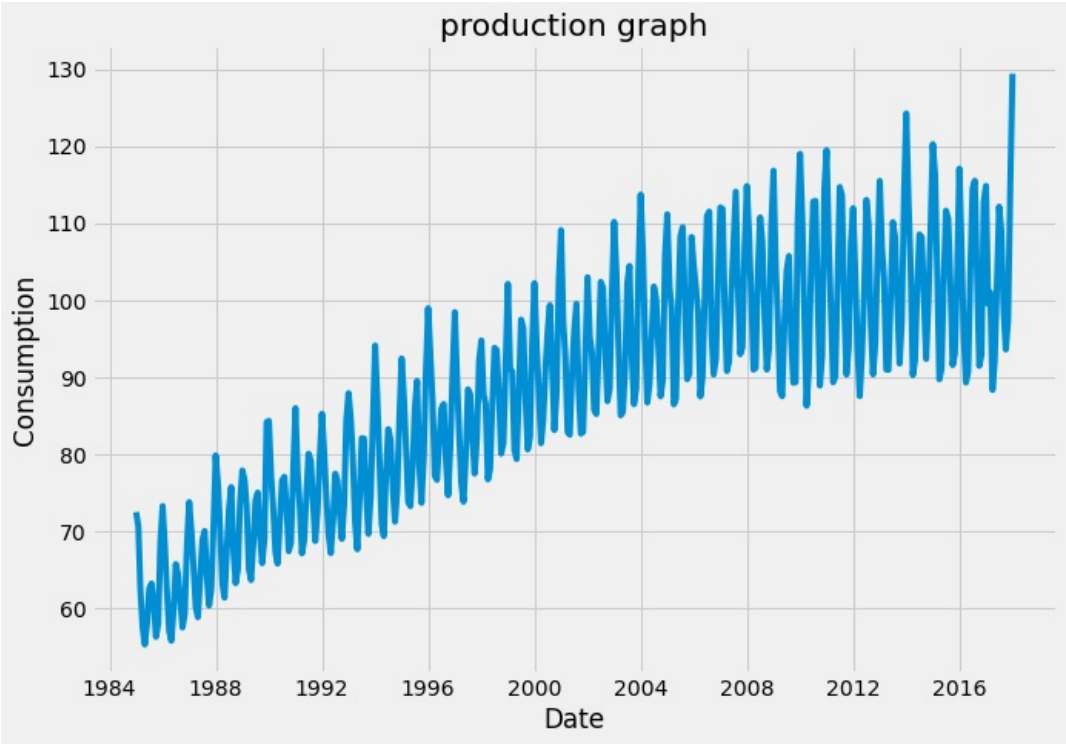
	Consumption
Date	
1985-01-01	72.5052
1985-02-01	70.6720
1985-03-01	62.4502
1985-04-01	57.4714
1985-05-01	55.3151

Now, let us start with our predefined steps:

1.Visualizing the time series.

In [36]:
```
plt.xlabel("Date")
plt.ylabel("Consumption")
plt.title("production graph")
plt.plot(df)
```

Out[36]:
```
[<matplotlib.lines.Line2D at 0x7ff6c808ad10>]
```

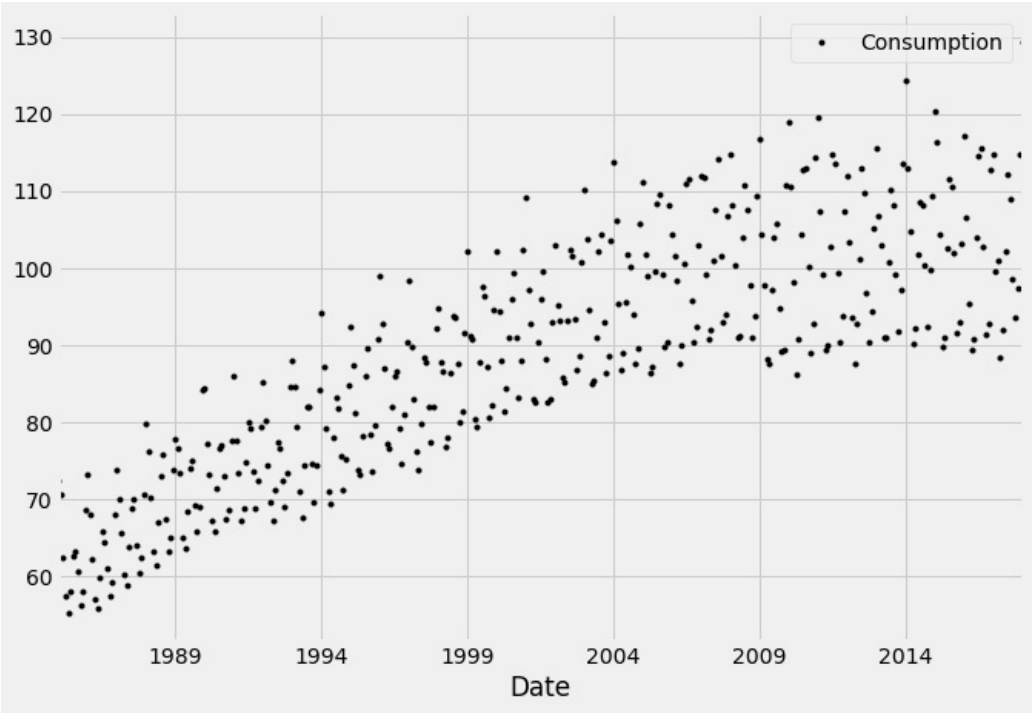


Note on Stationarity

- Remember that for time series forecasting, a series needs to be stationary. The series should have a constant mean, variance, and covariance.
- There are a few points to note here: the mean is not constant in this case as we can clearly see an upward trend.
- Hence, we have identified that our series is not stationary. We need to have a stationary series to do time series forecasting. In the next stage, we will try to convert this into a stationary series.

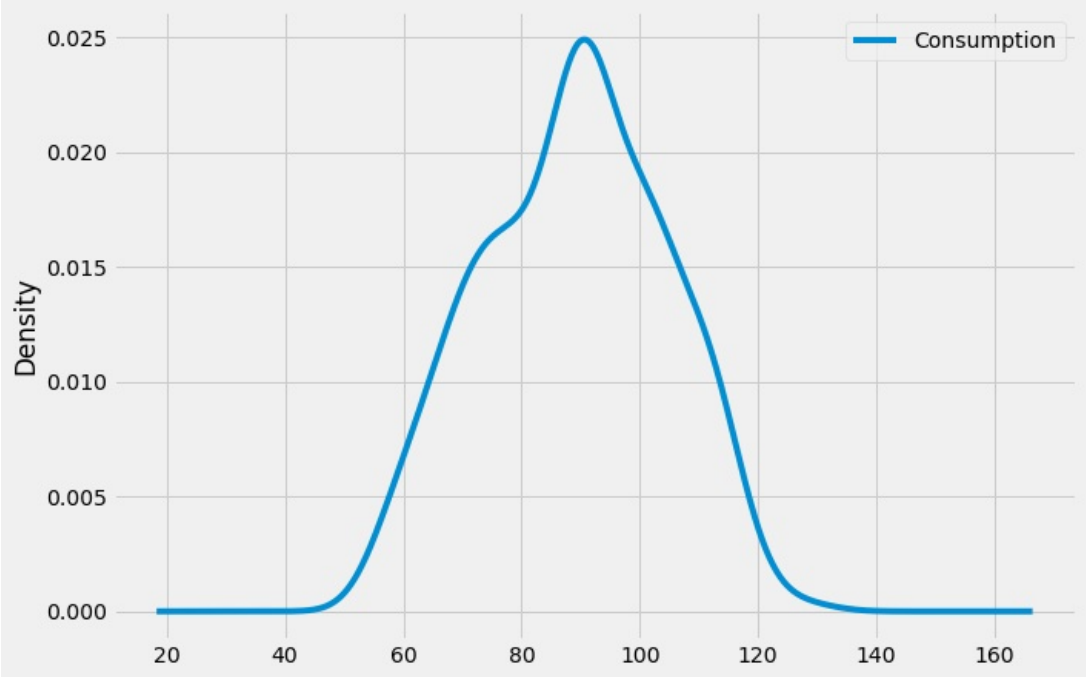
Let us plot the scatterplot:

In [37]:
```
df.plot(style='k.')
plt.show()
```



In [38]:
```
df.plot(kind='kde')
```

Out[38]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff6bf3bcfd0>
```



We can observe a near-normal distribution (bell-curve) over the consumption values.

- A given time series is thought to consist of three systematic components and one non-systematic component called noise. These components are defined as follows:

Level: The average value in the series.

Trend: The increasing or decreasing value in the series.

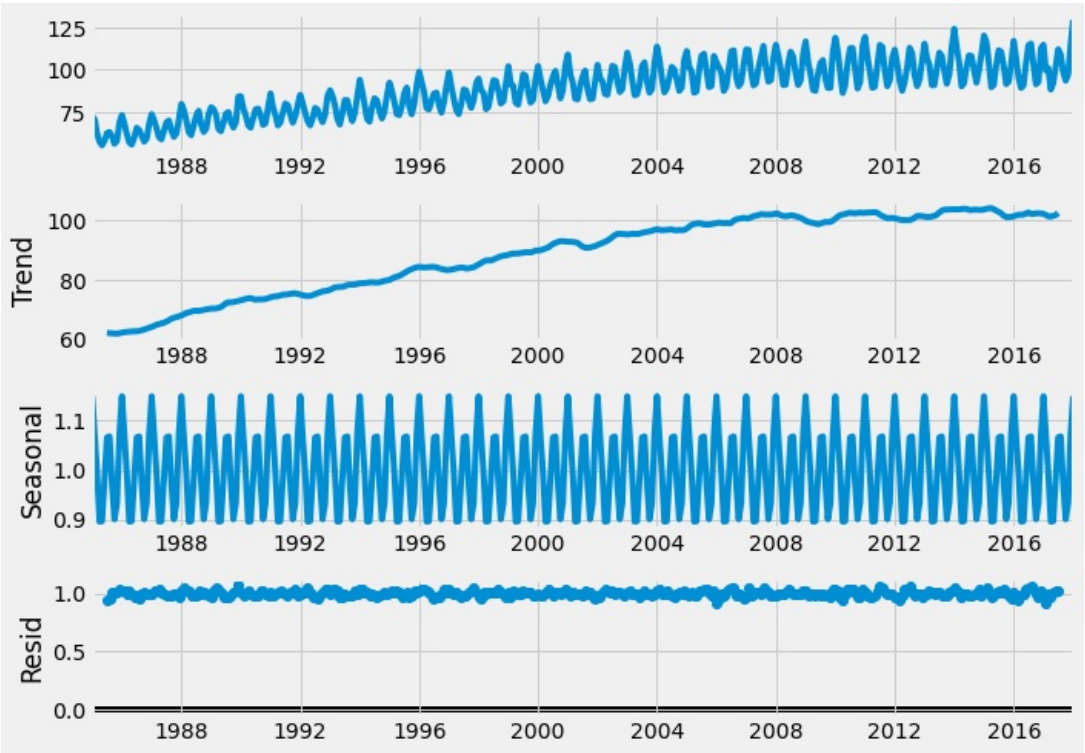
Seasonality: The repeating short-term cycle in the series.

Noise: The random variation in the series.

- To perform a time series analysis, it is often necessary to separate the seasonality and trend components from the series. This process helps in achieving a stationary series, which is essential for accurate time series forecasting.

Let us now proceed to separate the Trend and Seasonality from the time series.

In [39]:
```
from statsmodels.tsa.seasonal import seasonal_decompose
result = seasonal_decompose(df, model='multiplicative')
result.plot()
plt.show()
```



This gives us more insight into our data and real-world actions. Clearly, there is an upward trend and a recurring event where electricity

consumption shoots maximum every year.

2. Stationarising the Time Series

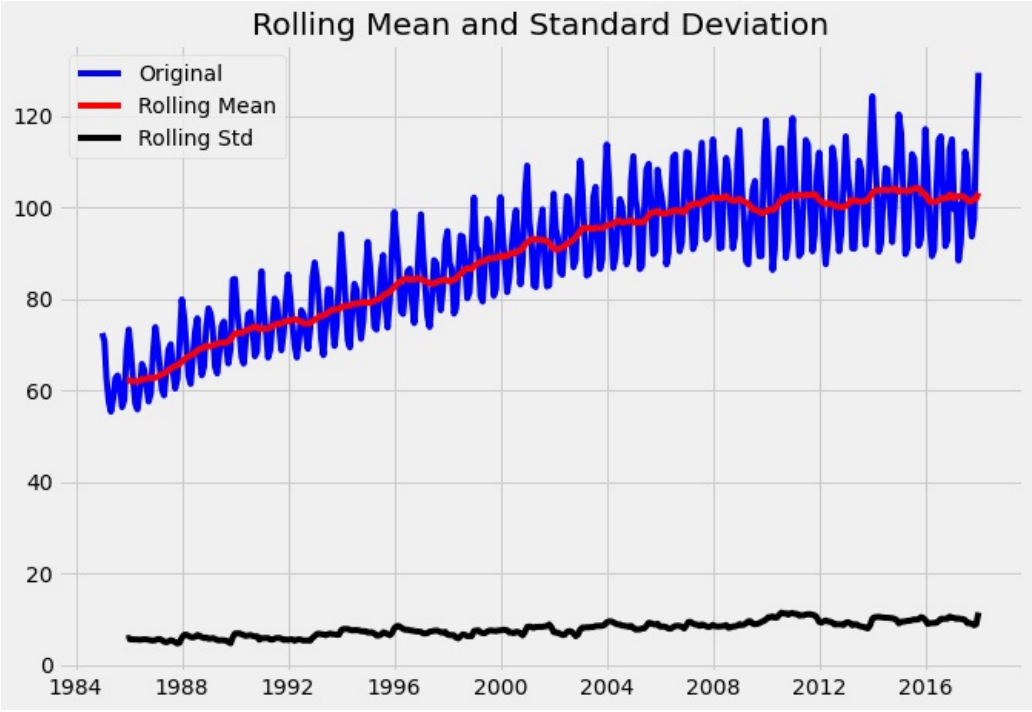
Before applying time series forecasting techniques, it's essential to determine if the series is stationary.

ADF (Augmented Dickey-Fuller) Test

- The Augmented Dickey-Fuller (ADF) test is a statistical hypothesis test used to determine whether a unit root is present in a time series dataset. A unit root test helps in understanding if the series is stationary or non-stationary. The hypotheses for the ADF test are:
- **Null Hypothesis:** The series has a unit root (non-stationary).
- **Alternate Hypothesis:** The series has no unit root (stationary).
- If we fail to reject the null hypothesis, it suggests that the series is non-stationary, meaning it may exhibit a trend or other non-random patterns over time. Stationarity typically requires that both the mean and variance of the series are constant over time.

To assess stationarity visually, we can plot the series along with its rolling mean and standard deviation.

In [40]:
```
from statsmodels.tsa.stattools import adfuller
def test_stationarity(timeseries):
    #Determing rolling statistics
    rolmean = timeseries.rolling(12).mean()
    rolstd = timeseries.rolling(12).std()
    #Plot rolling statistics:
    plt.plot(timeseries, color='blue',label='Original')
    plt.plot(rolmean, color='red', label='Rolling Mean')
    plt.plot(rolstd, color='black', label = 'Rolling Std')
    plt.legend(loc='best')
    plt.title('Rolling Mean and Standard Deviation')
    plt.show(block=False)

    #perform dickey fuller test
    print("Results of dickey fuller test")
    adft = adfuller(timeseries['Consumption'],autolag='AIC')
    # output for dft will give us without defining what the values are.
    #hence we manually write what values does it explains using a for loop
    output = pd.Series(adft[0:4],index=['Test Statistics','p-value','No. of lags used','Number of observations
    for key,values in adft[4].items():
        output['critical value (%s)'%key] =  values
    print(output)

test_stationarity(df)
```

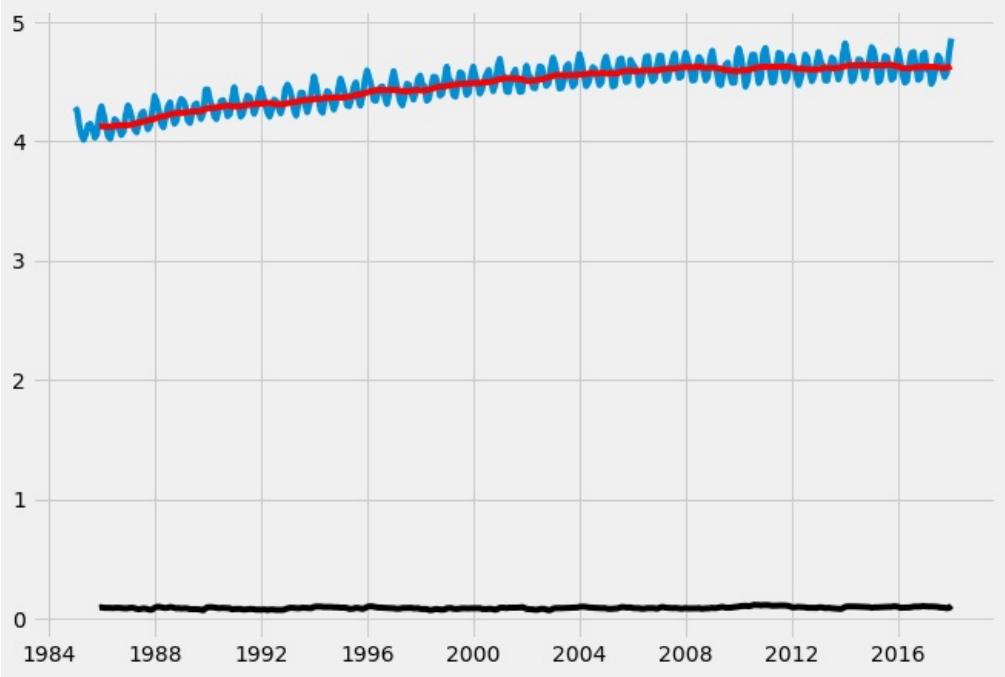


```
Results of dickey fuller test
Test Statistics                -2.256990
p-value                         0.186215
No. of lags used               15.000000
Number of observations used    381.000000
critical value (1%)             -3.447631
critical value (5%)             -2.869156
critical value (10%)            -2.570827
dtype: float64
```

- Through the above graph, we can see the increasing mean and standard deviation and hence our series is not stationary.

- We see that the p-value is greater than 0.05 so we cannot reject the Null hypothesis. Also, the test statistics is greater than the critical values. so the data is non-stationary.
- To get a stationary series, we need to eliminate the trend and seasonality from the series.
- We start by taking a log of the series to reduce the magnitude of the values and reduce the rising trend in the series. Then after getting the log of the series, we find the rolling average of the series. A rolling average is calculated by taking input for the past 12 months and giving a mean consumption value at every point further ahead in series.

In [41]:
```
df_log = np.log(df)
moving_avg = df_log.rolling(12).mean()
std_dev = df_log.rolling(12).std()
plt.plot(df_log)
plt.plot(moving_avg, color="red")
plt.plot(std_dev, color ="black")
plt.show()
```



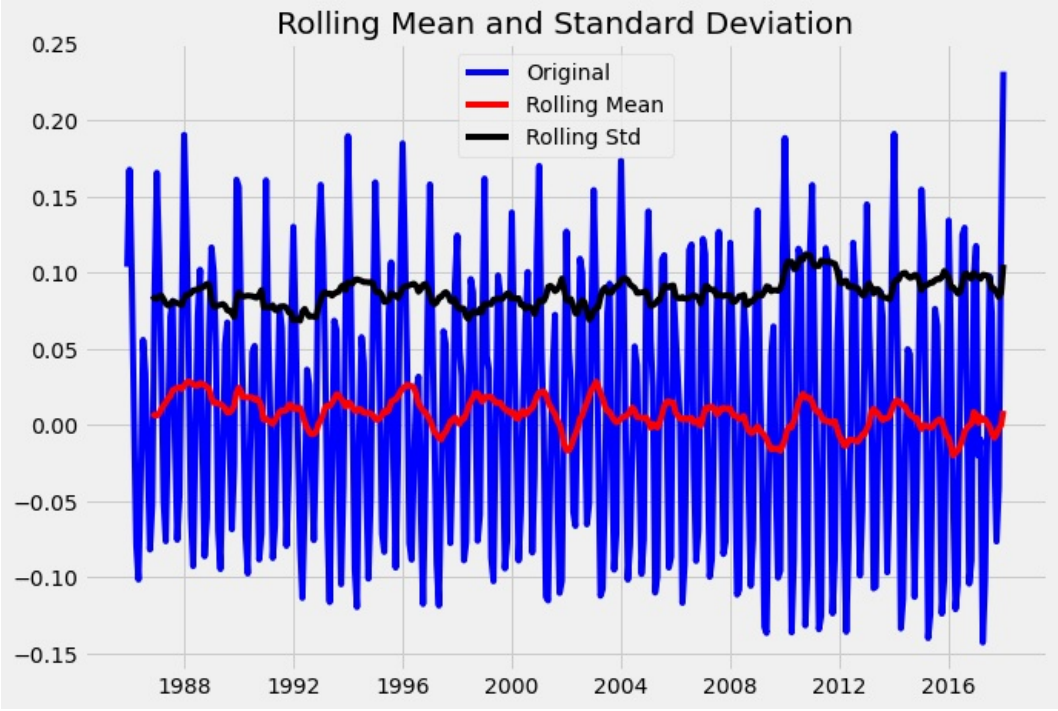
After finding the mean, we take the difference of the series and the mean at every point in the series.

This way, we eliminate trends out of a series and obtain a more stationary series.

In [42]:
```
df_log_moving_avg_diff = df_log-moving_avg
df_log_moving_avg_diff.dropna(inplace=True)
```

Perform the Dickey-Fuller test (ADFT) once again. We have to perform this function every time to check whether the data is stationary or not.

In [43]:
```
test_stationarity(df_log_moving_avg_diff)
```




```
Results of dickey fuller test
Test Statistics                  -5.211586
p-value                           0.000008
No. of lags used                 15.000000
Number of observations used     370.000000
critical value (1%)              -3.448148
critical value (5%)              -2.869383
critical value (10%)             -2.570948
dtype: float64
```

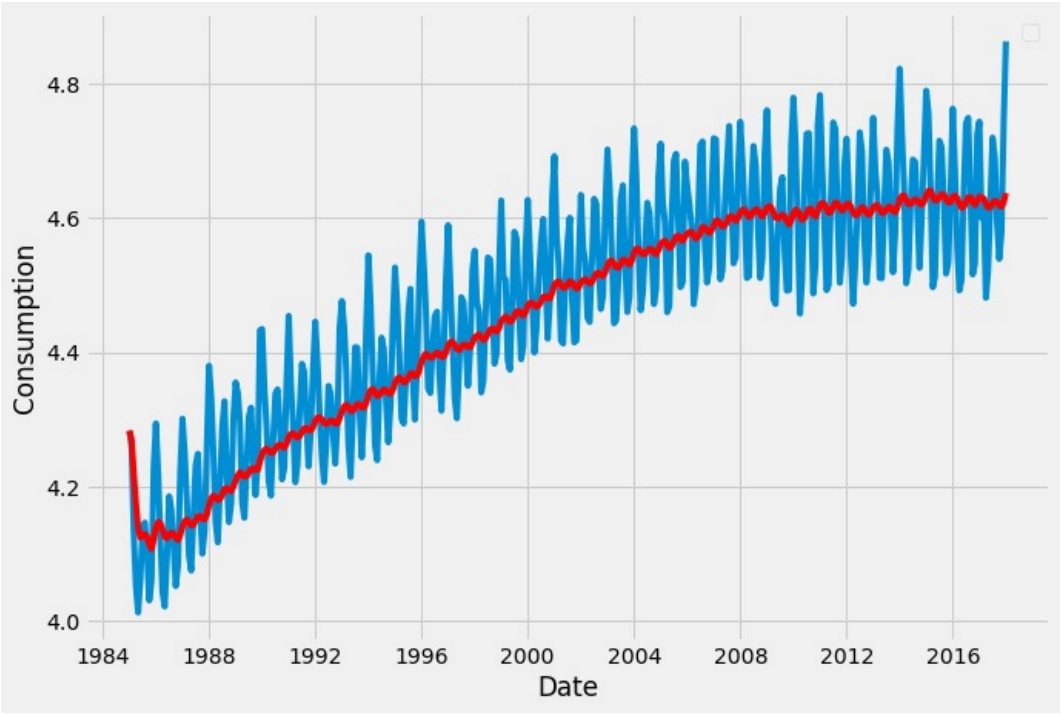
From the above graph, we observed that the data attained stationarity.

- One of the modules is completed as we came to a conclusion. We need to check the weighted average, to understand the trend of the data in time series. Take the previous log data and to perform the following operation.

In [44]:
```
weighted_average = df_log.ewm(halflife=12, min_periods=0,adjust=True).mean()
```

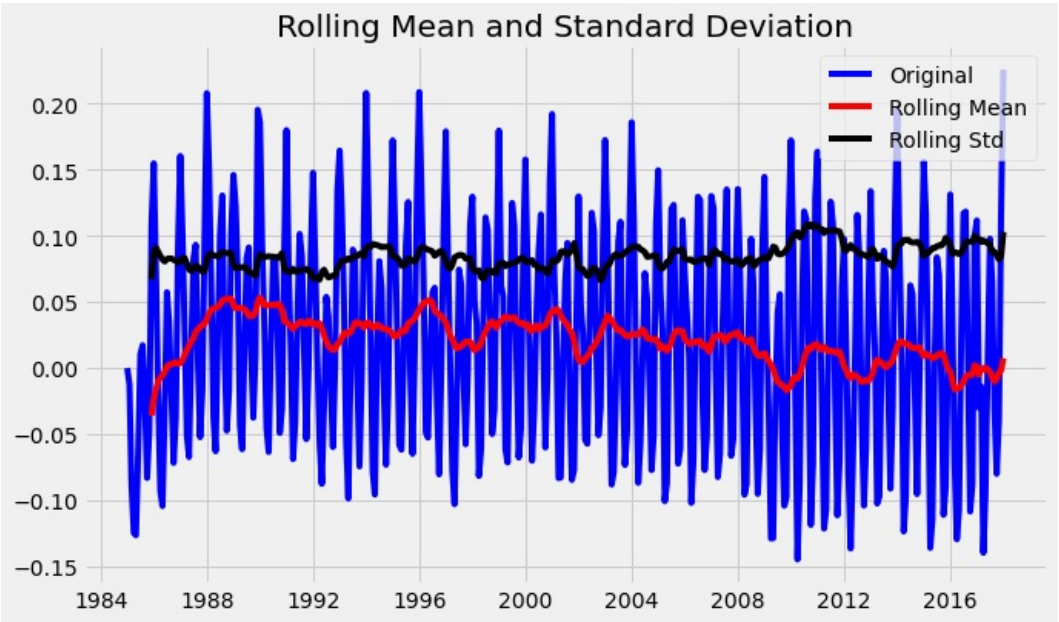
The exponential moving average (EMA) is a weighted average of the last n prices, where the weighting decreases exponentially with each previous price/period. In other words, the formula gives recent prices more weight than past prices.

In [45]:
```
plt.plot(df_log)
plt.plot(weighted_average, color='red')
plt.xlabel("Date")
plt.ylabel("Consumption")
from pylab import rcParams
rcParams['figure.figsize'] = 10,6
plt.legend()
plt.show(block =False)
```



Previously we subtracted df_log with moving average, now take the same df_log and subtract with weighted_average and perform the Dickey-Fuller test (ADFT) once again.

In [46]:
```
logScale_weightedMean = df_log-weighted_average
from pylab import rcParams
rcParams['figure.figsize'] = 10,6
test_stationarity(logScale_weightedMean)
```




```
Results of dickey fuller test
Test Statistics                  -3.251531
p-value                           0.017189
No. of lags used                 15.000000
Number of observations used     381.000000
critical value (1%)              -3.447631
critical value (5%)              -2.869156
critical value (10%)             -2.570827
dtype: float64
```

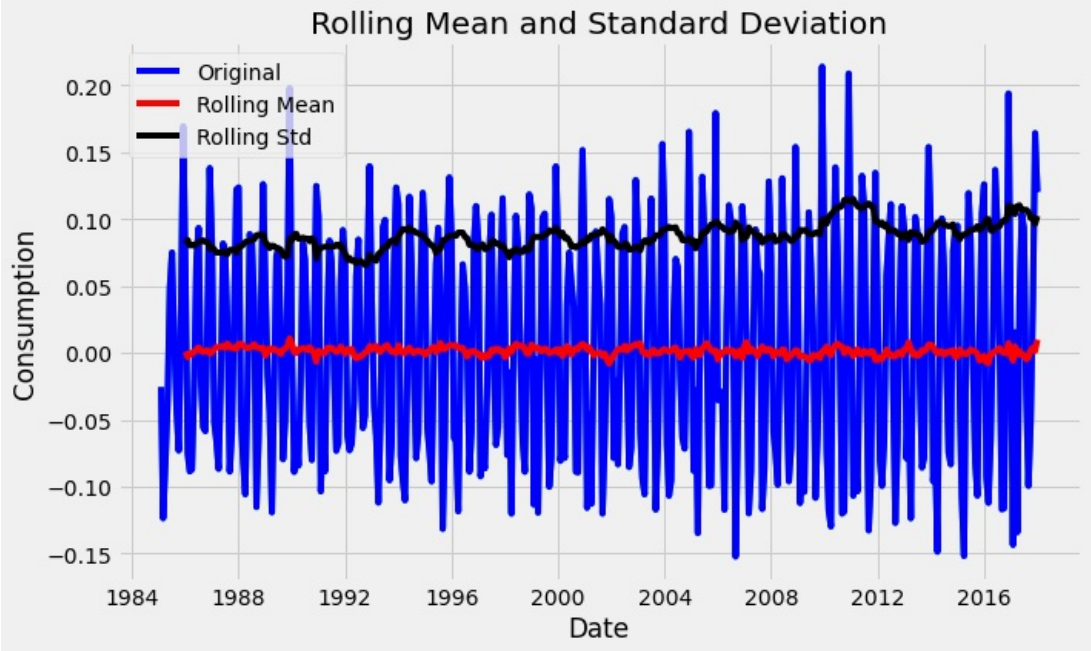
From the above graph, we observed that the data attained stationarity. We also see that the test statistics and critical value is relatively equal.

- There can be cases when there is a high seasonality in the data. In those cases, just removing the trend will not help much. We need to also take care of the seasonality in the series. One such method for this task is differencing.
- Differencing is a method of transforming a time series dataset.
- It can be used to remove the series dependence on time, so-called temporal dependence. This includes structures like trends and seasonality. Differencing can help stabilize the mean of the time series by removing changes in the level of a time series, and so eliminating (or reducing) trend and seasonality.

- Differencing is performed by subtracting the previous observation from the current observation.

Perform the Dickey-Fuller test (ADFT) once again.

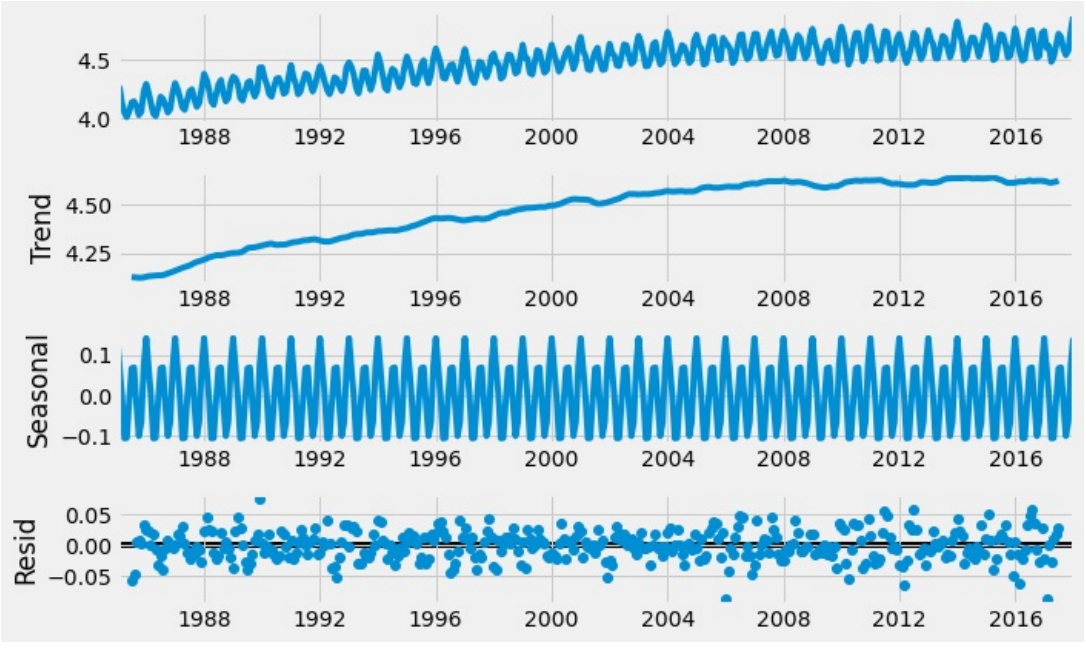
In [47]:
```
df_log_diff = df_log - df_log.shift()
plt.title("Shifted timeseries")
plt.xlabel("Date")
plt.ylabel("Consumption")
plt.plot(df_log_diff)

#Let us test the stationarity of our resultant series
df_log_diff.dropna(inplace=True)
test_stationarity(df_log_diff)
```




```
Results of dickey fuller test
Test Statistics                -6.748333e+00
p-value                         2.995161e-09
No. of lags used                1.400000e+01
Number of observations used     3.810000e+02
critical value (1%)            -3.447631e+00
critical value (5%)            -2.869156e+00
critical value (10%)           -2.570827e+00
dtype: float64
```

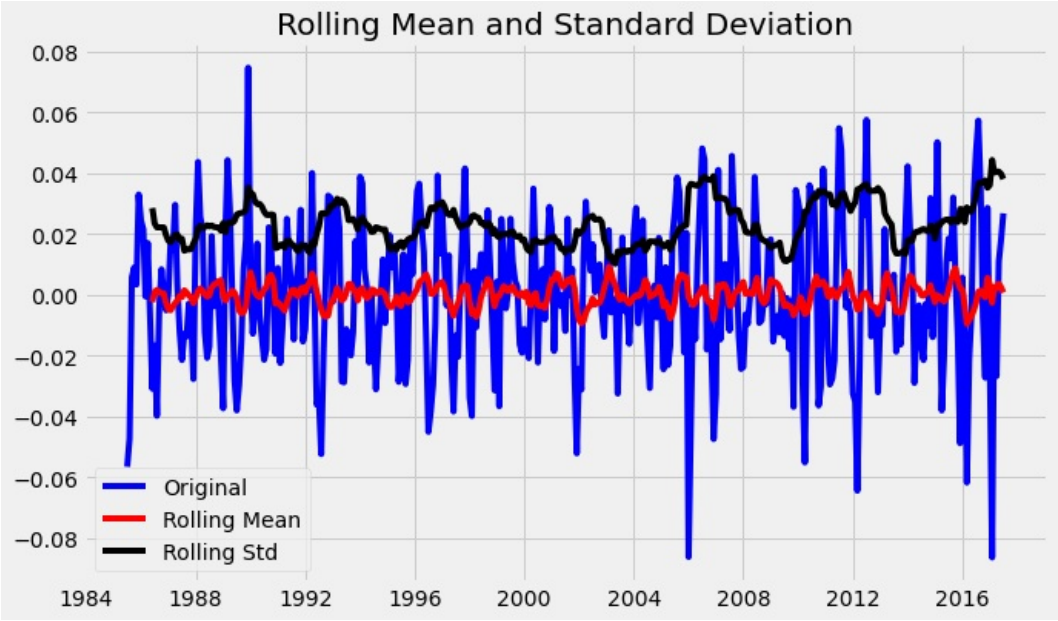
The next step is to perform decomposition which provides a structured way of thinking about a time series forecasting problem, both generally in terms of modeling complexity and specifically in terms of how to best capture each of these components in a given model.

In [50]:
```
from chart_studio.plotly import plot_mpl
from statsmodels.tsa.seasonal import seasonal_decompose
result = seasonal_decompose(df_log, model='additive', freq = 12)
result.plot()
plt.show()
```



Finally, perform the Dickey-Fuller test (ADFT) once again.

In [52]:
```
def test_stationarity_final(timeseries):
    #Determing rolling statistics
    rolmean = timeseries.rolling(12).mean()
    rolstd = timeseries.rolling(12).std()
    #Plot rolling statistics:
    plt.plot(timeseries, color='blue',label='Original')
    plt.plot(rolmean, color='red', label='Rolling Mean')
    plt.plot(rolstd, color='black', label = 'Rolling Std')
    plt.legend(loc='best')
    plt.title('Rolling Mean and Standard Deviation')
    plt.show(block=False)



trend = result.trend
trend.dropna(inplace=True)
seasonality = result.seasonal
seasonality.dropna(inplace=True)
residual = result.resid
residual.dropna(inplace=True)
test_stationarity_final(residual)
```



After the decomposition, if we look at the residual then we have clearly a flat line for both mean and standard deviation. We have got our stationary series and now we can move to find the best parameters for our model.

3. Finding the best parameters for our model

Before we go on to build our forecasting model, we need to determine optimal parameters for our model. For those optimal parameters, we need ACF and PACF plots.

- A nonseasonal ARIMA model is classified as an “ARIMA(p,d,q)” model, where:
- p → Number of autoregressive terms,
- d → Number of nonseasonal differences needed for stationarity, and
- q → Number of lagged forecast errors in the prediction equation.

Values of p and q come through ACF and PACF plots. So let us understand both ACF and PACF!

Autocorrelation Function(ACF)

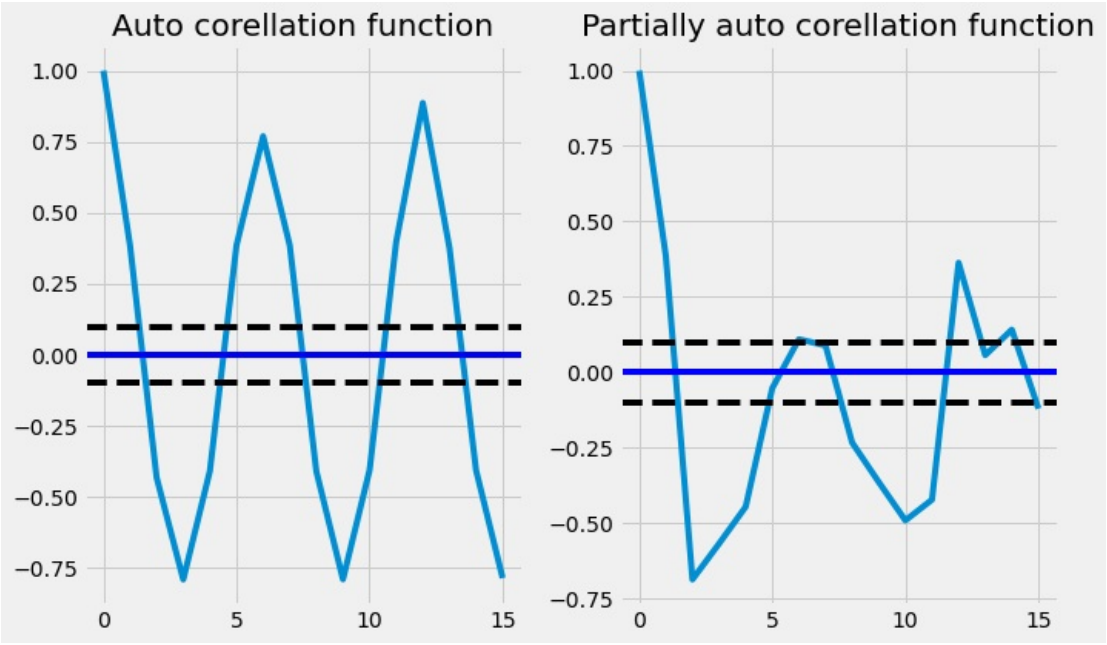
- Statistical correlation summarizes the strength of the relationship between two variables. Pearson’s correlation coefficient is a number between -1 and 1 that describes a negative or positive correlation respectively. A value of zero indicates no correlation.
- We can calculate the correlation for time series observations with previous time steps, called lags. Because the correlation of the time series observations is calculated with values of the same series at previous times, this is called a serial correlation, or an autocorrelation.
- A plot of the autocorrelation of a time series by lag is called the AutoCorrelation Function, or the acronym ACF. This plot is sometimes called a correlogram or an autocorrelation plot.

Partial Autocorrelation Function(PACF)

- A partial autocorrelation is a summary of the relationship between an observation in a time series with observations at prior time steps with the relationships of intervening observations removed.
- The partial autocorrelation at lag k is the correlation that results after removing the effect of any correlations due to the terms at shorter lags.
- The autocorrelation for observation and observation at a prior time step is comprised of both the direct correlation and indirect correlations. It is these indirect correlations that the partial autocorrelation function seeks to remove.

Below code plots, both ACF and PACF plots for us:

In [53]:
```
from statsmodels.tsa.stattools import acf,pacf
# we use d value here(data_log_shift)
acf = acf(df_log_diff, nlags=15)
pacf= pacf(df_log_diff, nlags=15,method='ols')#plot PACF
plt.subplot(121)
plt.plot(acf)
plt.axhline(y=0,linestyle='-',color='blue')
plt.axhline(y=-1.96/np.sqrt(len(df_log_diff)),linestyle='--',color='black')
plt.axhline(y=1.96/np.sqrt(len(df_log_diff)),linestyle='--',color='black')
plt.title('Auto corellation function')
plt.tight_layout()#plot ACF
plt.subplot(122)
plt.plot(pacf)
plt.axhline(y=0,linestyle='-',color='blue')
plt.axhline(y=-1.96/np.sqrt(len(df_log_diff)),linestyle='--',color='black')
plt.axhline(y=1.96/np.sqrt(len(df_log_diff)),linestyle='--',color='black')
plt.title('Partially auto corellation function')
plt.tight_layout()
```

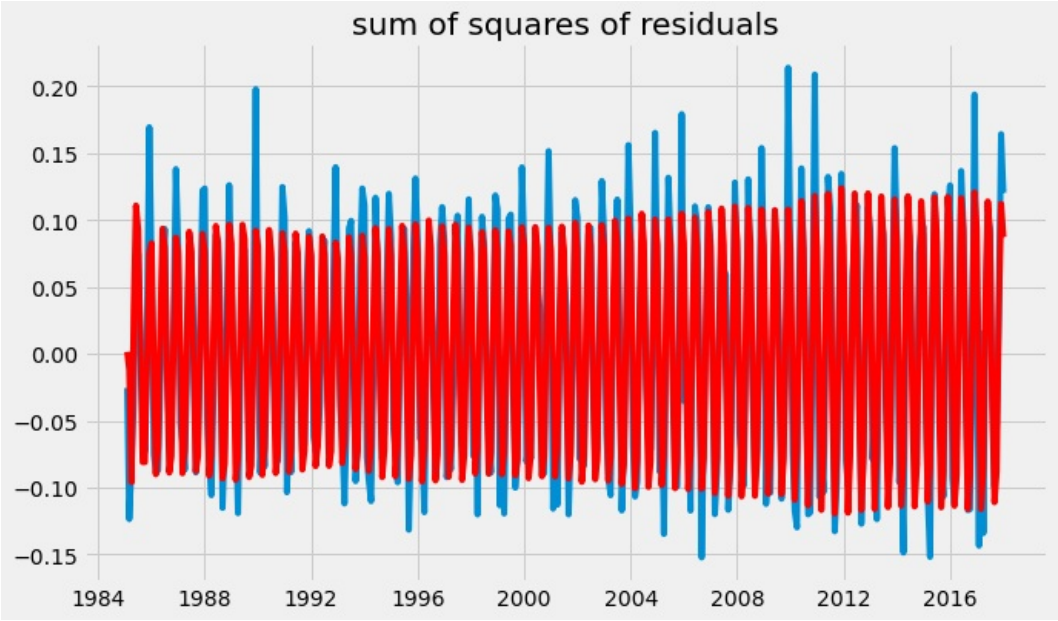


4. Fitting model

In order to find the p and q values from the above graphs, we need to check, where the graph cuts off the origin or drops to zero for the first time from the above graphs the p and q values are merely close to 3 where the graph cuts off the origin (draw the line to x-axis) now we have p,d,q values. So now we can substitute in the ARIMA model and let's see the output.

In [54]:
```
from statsmodels.tsa.arima_model import ARIMA
model = ARIMA(df_log, order=(3,1,3))
result_AR = model.fit(disp = 0)
plt.plot(df_log_diff)
plt.plot(result_AR.fittedvalues, color='red')
plt.title("sum of squares of residuals")
print('RSS : %f' %sum((result_AR.fittedvalues-df_log_diff["Consumption"])**2))
```

```
RSS : 0.522683
```

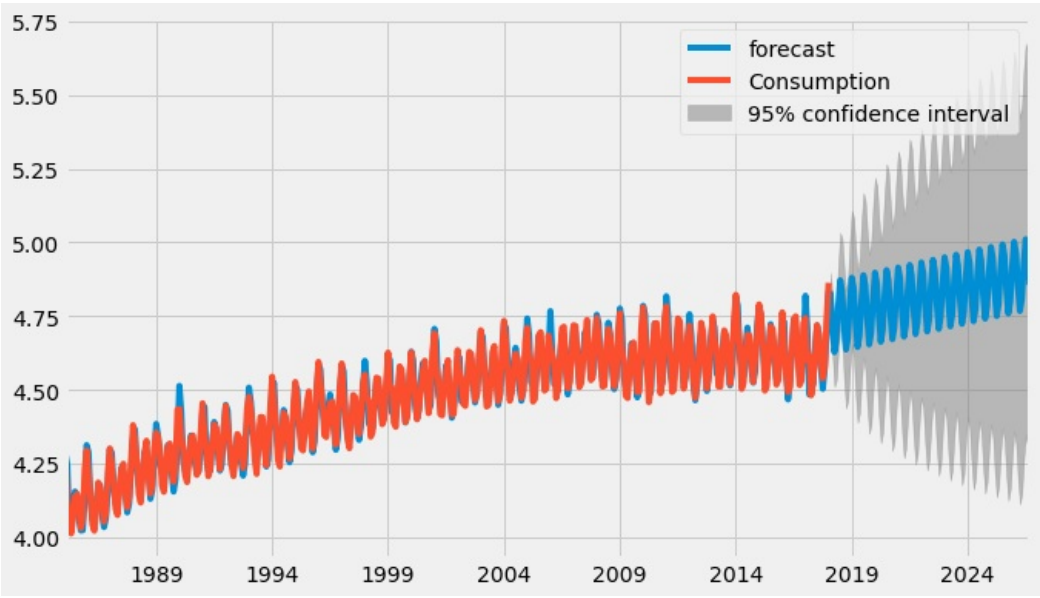


Less the RSS value, the more effective the model is. You check with (2,1,0),(3,1,1), etc to look for the smallest values of RSS.

5. Predictions

- to forecast shampoo sales for the next 6 years.

In [65]:
```
result_AR.plot_predict(1,500)
x=result_AR.forecast(steps=300)
```



From the above graph, we calculated the future predictions till 2024 the greyed out area is the confidence interval that means the predictions will not cross that area.