R.D. & S. H. National College & S. W.A. Science College

Bandra, Mumbai - 400050.

Department of Computer Science

CERTIFICATE

This is to certify that ~~Mr~~./Ms.____YASH RAJESH SHAH____of M.Sc. Data Science class (Second Semester) has satisfactorily completed ____10____ Practicals, in the subject of ____Time Series Analysis and Forecasting____as a part of M.Sc. in Data Science Program during the academic year 2023- 2024.

Date of Certification: 31/07/2024

Subject Incharge

Co-ordinator,
Department Computer Science

Signature of Examiner

INDEX

R.D & S.H National College & S.W.A Science College

Bandra, Mumbai – 400050.

DEPARTMENT OF COMPUTER SCIENCE

M.Sc. Data Science – **Semester II**

Course Code: **PSDS513**

Course Name: **Time Series Analysis and Forecasting**

Practical Journal

2023-2024

Seat No: CS – DS - 23017

Practical 1

Aim: Fitting and plotting of modified exponential curve.

Theory:

A modified exponential curve is a mathematical model used to describe data that follows an exponential growth or decay pattern but includes an additional constant term to account for shifts or offsets in the data. This type of model is particularly useful in various fields such as finance, biology, and physics, where exponential relationships are observed but data does not start at zero.

The General Form

The general form of the modified exponential function can be expressed as:

$$y = a \cdot e^{b \cdot x}$$

where:

- y is the dependent variable (e.g., gold price).
- x is the independent variable (e.g., time).
- a is a scaling factor that adjusts the amplitude of the curve.
- b is the growth rate (if $b > 0$) or decay rate (if $b < 0$).
- c is a constant that shifts the entire curve vertically.

Applications

The modified exponential curve is useful in various scenarios:

- Finance: Modeling the growth of investments or prices of commodities, like gold prices, over time.
- Biology: Describing population growth with a carrying capacity or decay in the presence of a constant offset.
- Physics: Modeling radioactive decay processes or other phenomena with a constant background level.

Fitting the Curve

To fit a modified exponential curve to a dataset, numerical methods such as nonlinear least squares can be used. The goal is to find the parameters a, b, and c that minimize the difference between the observed data points and the values predicted by the model.

The modified exponential curve is a powerful tool for modeling data that exhibits exponential growth or decay with an offset. By incorporating a constant term, this model provides a more accurate fit for many real-world scenarios, making it invaluable in fields like finance, biology, and physics. Understanding the components and interpretation of the parameters helps in applying this model effectively to analyze and predict trends in various datasets.

Code:

```
import pandas as pd
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt
from io import StringIO

# Read the data into a DataFrame
df = pd.read_csv("Gold.csv", parse_dates=['Date'])

# Convert the Date column to datetime format and create a numerical representation
df['Days'] = (df['Date'] - df['Date'].min()).dt.days
```

# Extract the days and values for fitting
x = df['Days'].values
y = df['Value'].values

# Define the modified exponential function
def modified_exponential(x, a, b, c):
    return a * np.exp(b * x) + c

Fit the curve
params, _ = curve_fit(modified_exponential, x, y)

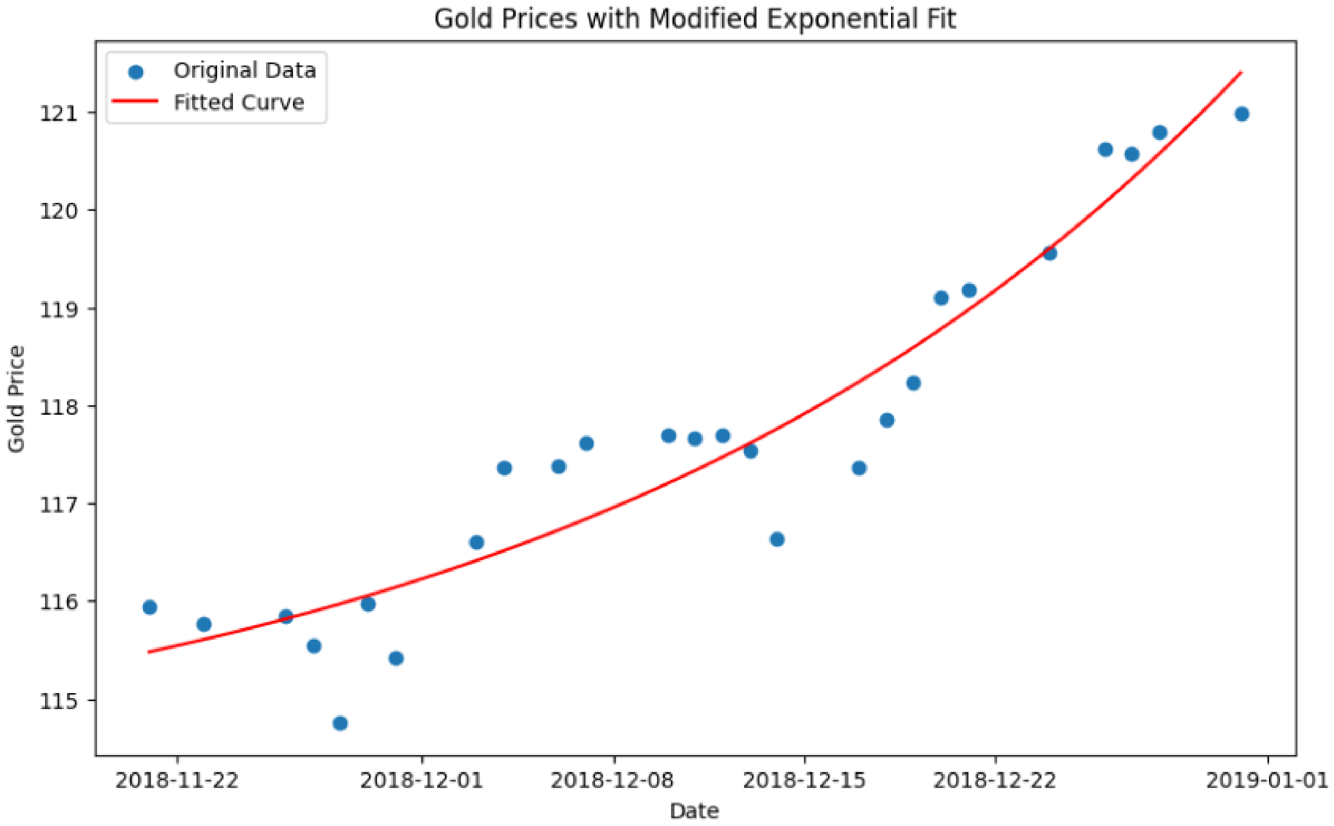
Extract the parameters
a, b, c = params
print(f"Fitted parameters: a = {a}, b = {b}, c = {c}")

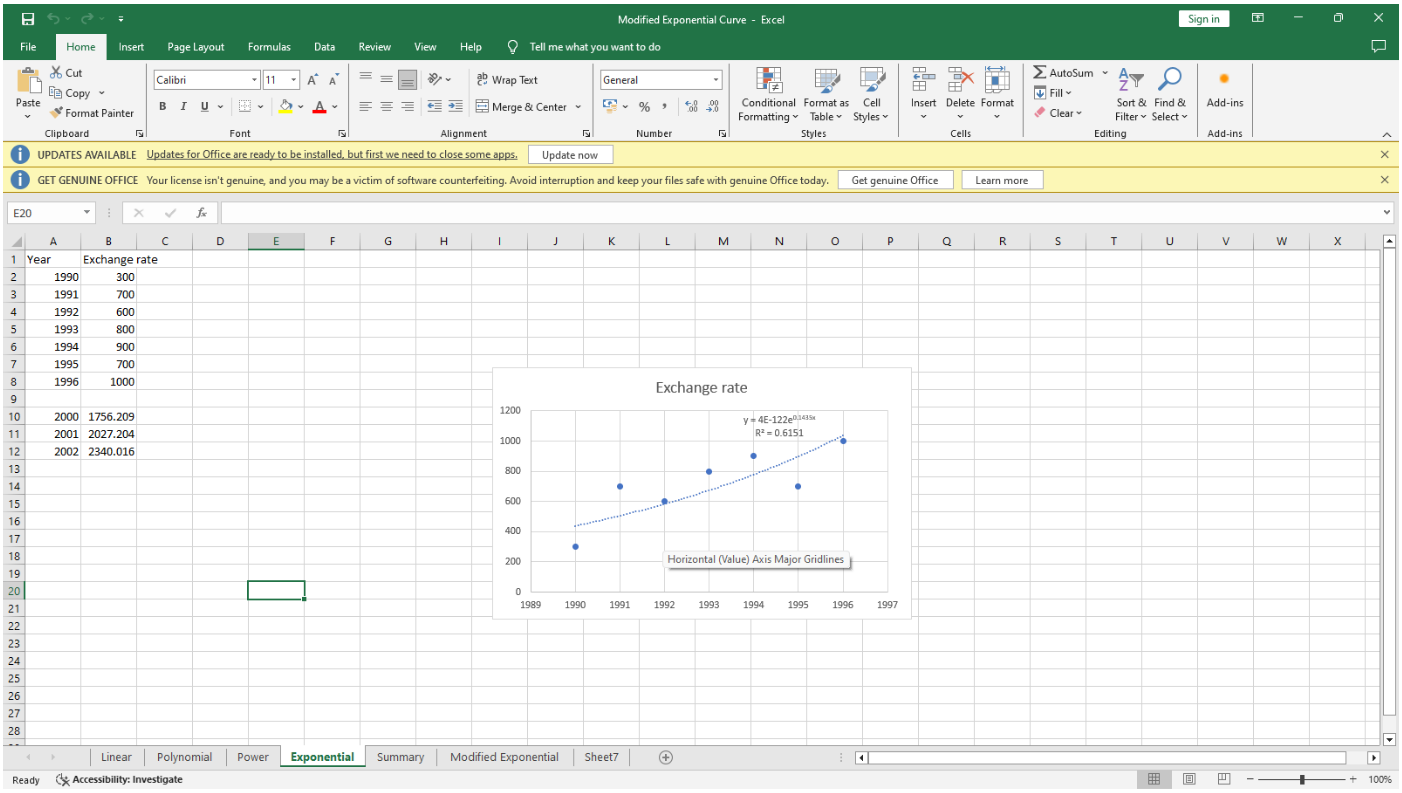
# Generate x values for the fitted curve
x_fit = np.linspace(x.min(), x.max(), 1000)
y_fit = modified_exponential(x_fit, a, b, c)

# Plot the original data and the fitted curve
plt.figure(figsize=(10, 6))
plt.scatter(df['Date'], y, label='Original Data')
plt.plot(np.array(df['Date'].min() + pd.to_timedelta(x_fit, unit='D')), y_fit, color='red', label='Fitted Curve')
plt.xlabel('Date')
plt.ylabel('Gold Price')
plt.title('Gold Prices with Modified Exponential Fit')
plt.legend()
plt.show()

Output:

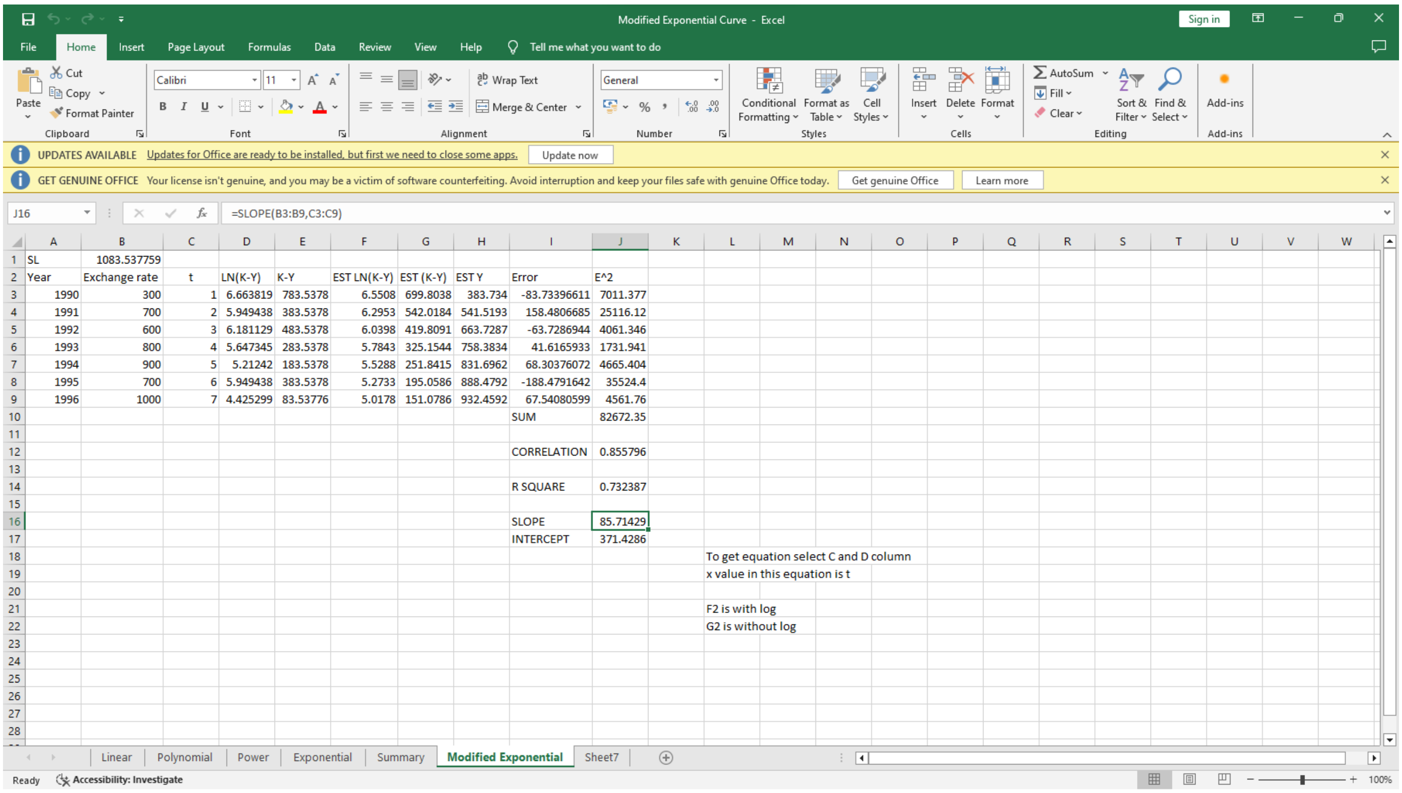
```
Fitted parameters: a = 1.5499561995062028, b = 0.03932190360335689, c = 113.92908624837918
```





E20

	A	B
1	Year	Exchange rate
2	1990	300
3	1991	700
4	1992	600
5	1993	800
6	1994	900
7	1995	700
8	1996	1000
9		
10	2000	1756.209
11	2001	2027.204
12	2002	2340.016



J16 =SLOPE(B3:B9,C3:C9)

	A	B	C	D	E	F	G	H	I	J
1	SL	1083.537759								
2	Year	Exchange rate	t	LN(K-Y)	K-Y	EST LN(K-Y)	EST (K-Y)	EST Y	Error	E^2
3	1990	300	1	6.663819	783.5378	6.5508	699.8038	383.734	-83.73396611	7011.377
4	1991	700	2	5.949438	383.5378	6.2953	542.0184	541.5193	158.4806685	25116.12
5	1992	600	3	6.181129	483.5378	6.0398	419.8091	663.7287	-63.7286944	4061.346
6	1993	800	4	5.647345	283.5378	5.7843	325.1544	758.3834	41.6165933	1731.941
7	1994	900	5	5.21242	183.5378	5.5288	251.8415	831.6962	68.30376072	4665.404
8	1995	700	6	5.949438	383.5378	5.2733	195.0586	888.4792	-188.4791642	35524.4
9	1996	1000	7	4.425299	83.53776	5.0178	151.0786	932.4592	67.54080599	4561.76
10									SUM	82672.35
12									CORRELATION	0.855796
14									R SQUARE	0.732387
16									SLOPE	85.71429
17									INTERCEPT	371.4286

To get equation select C and D column
x value in this equation is t

F2 is with log
G2 is without log

Practical 2

Aim: Fitting and plotting of Gompertz curve.

Theory:

The Gompertz curve is a type of sigmoid function that is often used to describe growth processes, such as the growth of populations, the spread of diseases, and the progression of certain types of cancers. It is named after Benjamin Gompertz, who introduced it in 1825 as a mathematical model for human mortality. The Gompertz function is characterized by its asymmetrical S-shape, reflecting rapid growth at the beginning that slows down as it approaches an upper limit.

The General Form: The general form of the modified exponential function can be expressed as:

$$y = a \cdot e^{-b \cdot e^{-c \cdot x}}$$

where:

- y is the dependent variable (e.g., population size, price level).
- x is the independent variable (e.g., time).
- a is the upper asymptote or the maximum value that y can reach.
- b and c are parameters that control the displacement along the x-axis and the growth rate, respectively.

Fitting the Gompertz Curve
To fit a Gompertz curve to empirical data, nonlinear regression techniques are typically used. This involves finding the parameters a, b, and c that minimize the difference between the observed data points and the values predicted by the Gompertz function. The curve_fit function from the scipy.optimize module in Python is commonly used for this purpose.

The Gompertz curve is a powerful and versatile tool for modeling a wide range of growth processes. Its asymmetrical S-shape and the flexibility of its parameters allow it to accurately represent growth phenomena in biology, demography, economics, and beyond. Understanding the underlying theory and properties of the Gompertz curve helps in effectively applying this model to analyze and predict trends in various datasets.

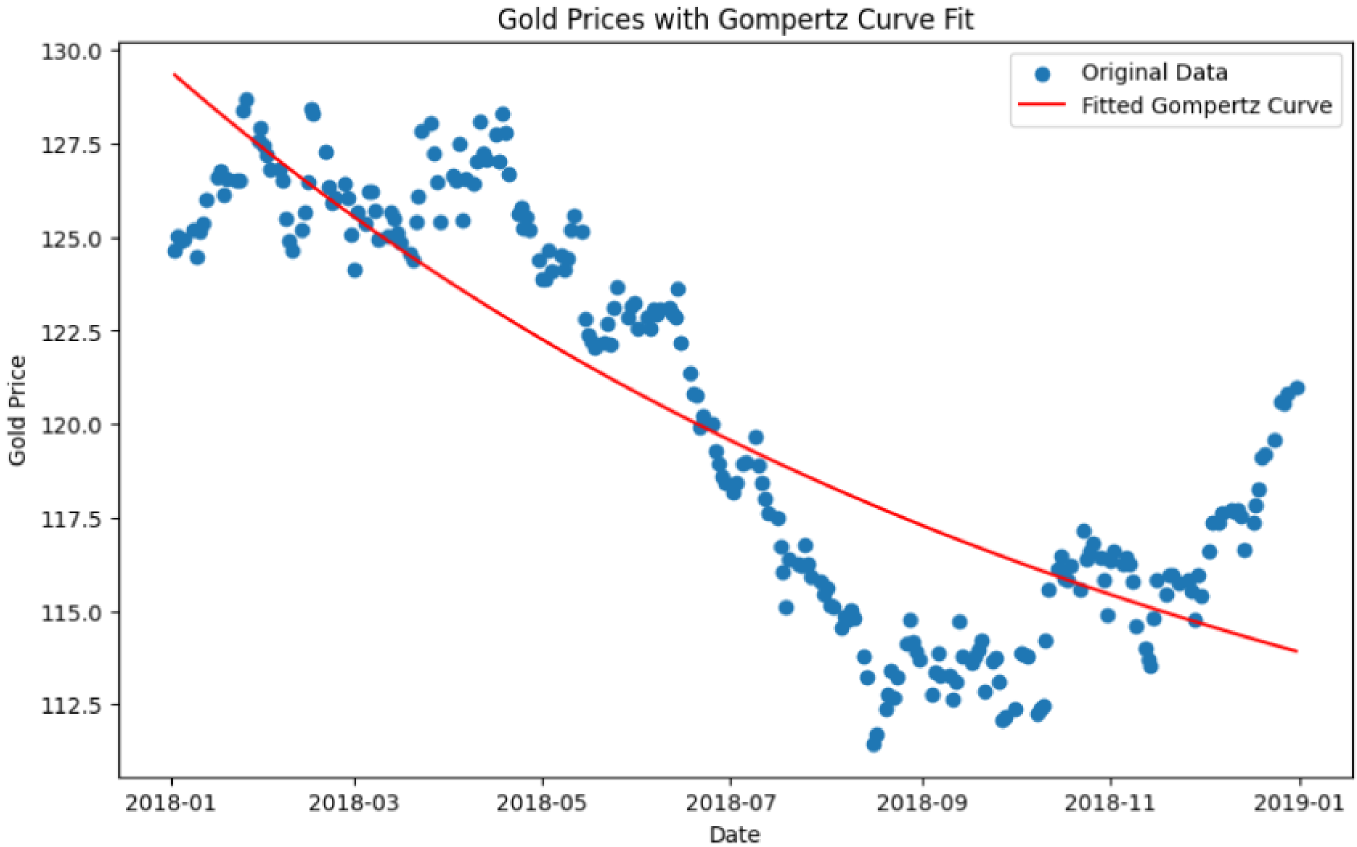
Code:

```
import pandas as pd
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

# Read the data into a DataFrame
df = pd.read_csv("Gold.csv", parse_dates=['Date'])

# Convert the Date column to datetime format and create a numerical representation
df['Days'] = (df['Date'] - df['Date'].min()).dt.days
# Extract the days and values for fitting
x = df['Days'].values
y = df['Value'].values

# Scale the data
x_scaled = x / max(x)
y_scaled = y / max(y)
```

```
# Define the Gompertz function
def gompertz(x, a, b, c):
    return a * np.exp(-b * np.exp(-c * x))

# Fit the curve
params, _ = curve_fit(gompertz, x_scaled, y_scaled, p0=[1, 1, 1], maxfev=100000)

# Extract the parameters
a, b, c = params
print(f"Fitted parameters: a = {a}, b = {b}, c = {c}")

# Generate x values for the fitted curve
x_fit = np.linspace(x.min(), x.max(), 1000)
x_fit_scaled = x_fit / max(x)
y_fit_scaled = gompertz(x_fit_scaled, a, b, c)
y_fit = y_fit_scaled * max(y)

# Plot the original data and the fitted curve
plt.figure(figsize=(10, 6))
plt.scatter(df['Date'], y, label='Original Data')
plt.plot(np.array(df['Date'].min() + pd.to_timedelta(x_fit, unit='D')), y_fit, color='red', label='Fitted
Gompertz Curve')
plt.xlabel('Date')
plt.ylabel('Gold Price')
plt.title('Gold Prices with Gompertz Curve Fit')
plt.legend()
plt.show()
```

Output:

```
Fitted parameters: a = 0.8231607891784759, b = -0.19964817505794002, c = 1.0099071252726064
```



Practical 3

Aim: Fitting and plotting of logistic curve.

Theory:

The logistic curve, or logistic function, is a sigmoidal mathematical model that describes growth processes that initially accelerate and then decelerate as they approach an upper limit. It is widely used in various fields, including biology, economics, epidemiology, and more, to model phenomena where growth saturates over time.

Mathematical Formulation: The logistic curve is defined by the following equation:

$$y = \frac{L}{1 + e^{-k(x-x_0)}}$$

where:

- y is the dependent variable (e.g., population size, price level).
- x is the independent variable (e.g., time).
- L is the curve's maximum value, also known as the carrying capacity or saturation level.
- b is the growth rate parameter that determines the steepness of the curve.
- x_0 is the x-value of the sigmoid's midpoint, representing the point where the curve reaches half of its maximum value L.

Fitting a Logistic Curve
To fit a logistic curve to empirical data, nonlinear regression techniques are typically employed. The curve_fit function from the scipy.optimize module in Python is commonly used for this purpose. It iteratively adjusts the parameters L, k, and x_0 to minimize the difference between the observed data points and the values predicted by the logistic function.

Plotting the Logistic Curve
Once the logistic curve is fitted, it can be plotted along with the original data to visualize how well the model fits the empirical observations. This graphical representation helps in understanding the growth dynamics and making predictions about future trends based on the fitted model.

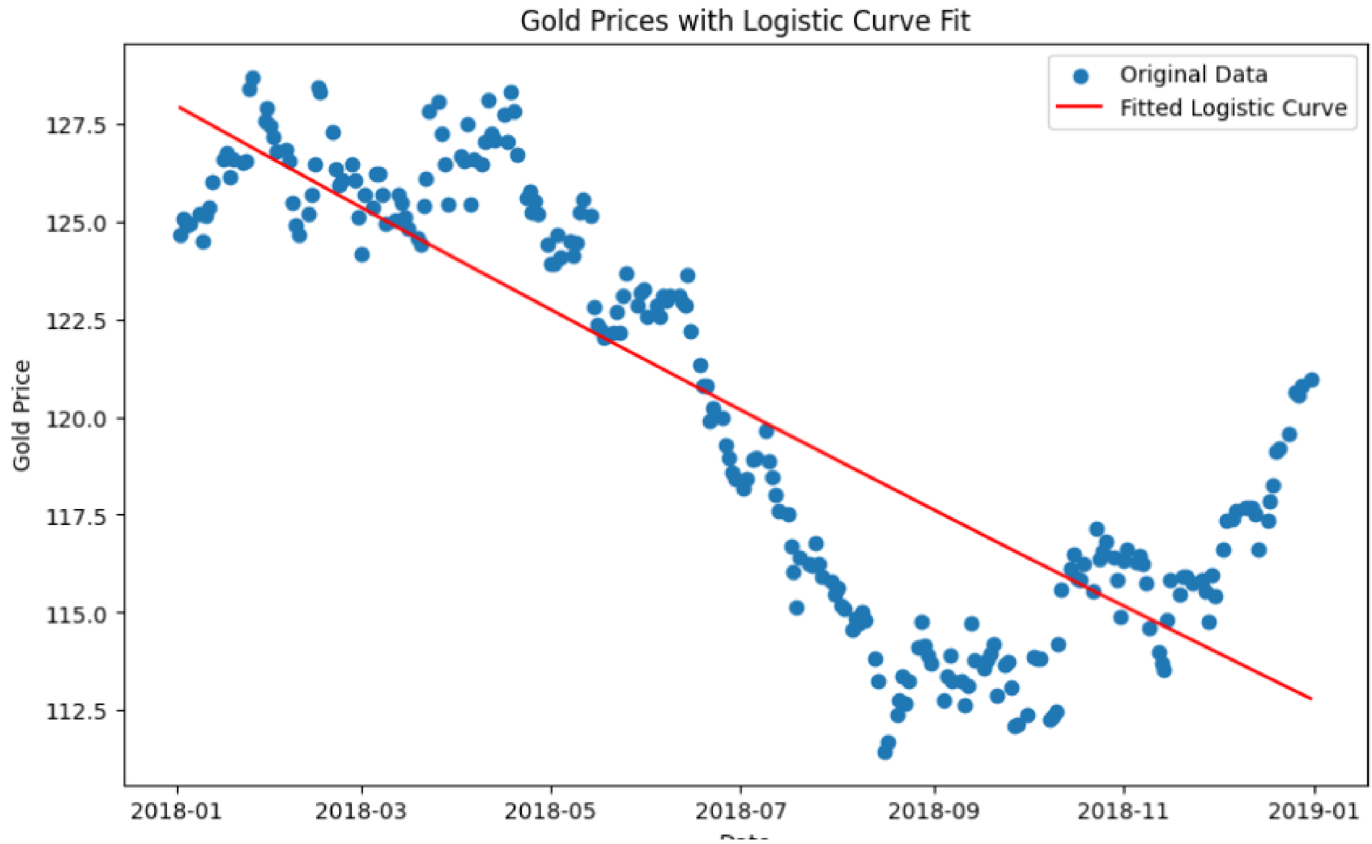
Code:

```
import pandas as pd
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

# Read the data into a DataFrame
df = pd.read_csv("Gold.csv", parse_dates=['Date'])

# Convert the Date column to datetime format and create a numerical representation
df['Days'] = (df['Date'] - df['Date'].min()).dt.days

# Extract the days and values for fitting
x = df['Days'].values
y = df['Value'].values

# Define the logistic function
def logistic(x, L, k, x_0):
```

```
    return L / (1 + np.exp(-k * (x - x_0)))

# Fit the curve
params, _ = curve_fit(logistic, x, y, p0=[max(y), 1, np.median(x)], maxfev=10000)

# Extract the parameters
L, k, x_0 = params
print(f"Fitted parameters: L = {L}, k = {k}, x_0 = {x_0}")

# Generate x values for the fitted curve
x_fit = np.linspace(x.min(), x.max(), 1000)
y_fit = logistic(x_fit, L, k, x_0)

# Plot the original data and the fitted curve
plt.figure(figsize=(10, 6))
plt.scatter(df['Date'], y, label='Original Data')
plt.plot(np.array(df['Date'].min() + pd.to_timedelta(x_fit, unit='D')), y_fit, color='red', label='Fitted
Logistic Curve')
plt.xlabel('Date')
plt.ylabel('Gold Price')
plt.title('Gold Prices with Logistic Curve Fit')
plt.legend()
plt.show()
```

Output:

```
Fitted parameters: L = 203792.18432658273, k = -0.00034632491672900446, x_0 = -21289.160118364893
```



Practical 4

Aim: Fitting of trend by Moving Average Method

Theory:

The Moving Average Method is a statistical technique used to analyze time series data by calculating averages of subsets of data points over a specified period. This method is particularly useful for smoothing out short-term fluctuations to identify underlying trends in the data. Here's a detailed explanation of the theory behind fitting a trend using the Moving Average Method:

Purpose of Moving Average Method

The primary goal of the Moving Average Method is to reduce the noise or random fluctuations in time series data, thereby making it easier to identify the underlying trend or pattern. By averaging out short-term variations, the method highlights longer-term trends that may be obscured by noise.

Mathematical Formulation

Given a time series $\{y_t\}$ where t represents time, the moving average MA_tat time with a window size n is calculated as:

$$MA_t = \frac{1}{n} \Sigma_{i=t-n+1}^{t} y_i$$

where:

- y_i are the observed values at time i
- n is the window size, representing the number of periods over which to average.

Advantages:

1. *Smoothing Out Fluctuations:* One of the primary advantages of moving averages is their ability to smooth out short-term fluctuations and noise in the data. This is particularly useful in financial markets where prices can be volatile on a daily basis. By averaging out these fluctuations, moving averages provide a clearer picture of the underlying trend over time.

2. *Highlighting Trends:* Moving averages help to identify and visualize trends within the data. By averaging data points over a specified window, they emphasize long-term patterns and cycles that may not be immediately apparent from raw data. This is crucial for understanding the overall direction and behavior of the time series.

3. *Simplifying Data Interpretation:* They simplify complex time series data into a smoother curve, making it easier to interpret and analyze trends visually. This simplification enhances the ability to communicate insights to stakeholders and decision-makers who may not be familiar with the technical details of the data.

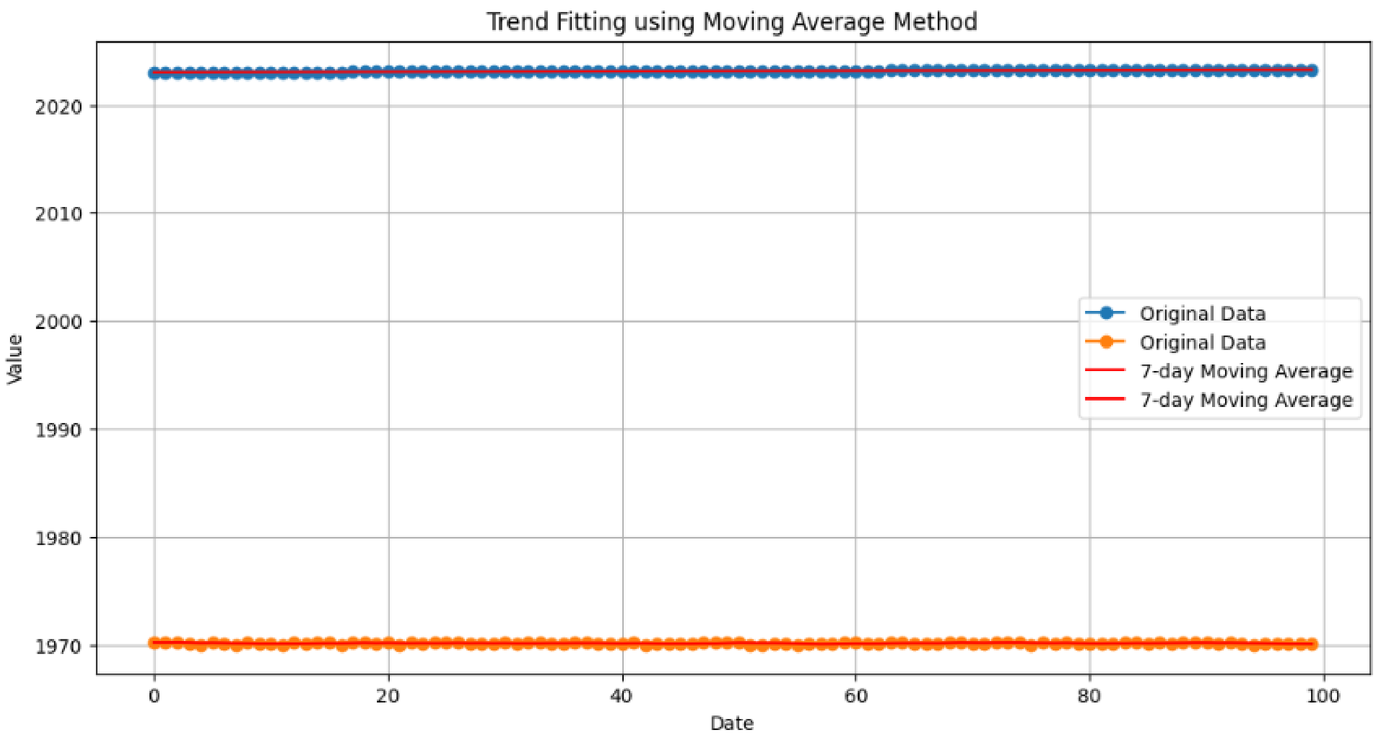
4. *Forecasting and Prediction:* Moving averages can be used to forecast future values based on historical trends. Once the underlying trend is identified through moving averages, extrapolating future values becomes more reliable, making them valuable tools for predictive modeling and forecasting.

Code:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Sample data (replace with your actual dataset)
```

```
data = {
    'Date': pd.date_range(start='2023-01-01', periods=100),
    'Value': np.random.rand(100) * 100  # Random values for demonstration
}

df = pd.DataFrame(data)

def moving_average(data, window_size):
    ma = data['Value'].rolling(window=window_size, min_periods=1).mean()
    return pd.DataFrame({'Date': data['Date'], 'Moving Average': ma})

window_size = 7  # Example: 7-day moving average

# Calculate moving average
df_ma = moving_average(df, window_size)

# Plotting
plt.figure(figsize=(12, 6))
plt.plot(df, label='Original Data', marker='o')
plt.plot(df_ma, color='red', label=f'{window_size}-day Moving Average')
plt.xlabel('Date')
plt.ylabel('Value')
plt.title('Trend Fitting using Moving Average Method')
plt.legend()
plt.grid(True)
plt.show()
```

Output:



Practical 5

Aim: Measurement of Seasonal indices Ratio-to-Trend method.

In [1]:
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```python
# Create sample data
date_range = pd.date_range(start='2018-01-01', end='2022-12-31', freq='M')
data = pd.DataFrame({
    'date': date_range,
    'values': np.random.rand(len(date_range)) * 100 + 500  # random values between 500 and 600
})
date_range
```

Out[2]:
```
DatetimeIndex(['2018-01-31', '2018-02-28', '2018-03-31', '2018-04-30',
               '2018-05-31', '2018-06-30', '2018-07-31', '2018-08-31',
               '2018-09-30', '2018-10-31', '2018-11-30', '2018-12-31',
               '2019-01-31', '2019-02-28', '2019-03-31', '2019-04-30',
               '2019-05-31', '2019-06-30', '2019-07-31', '2019-08-31',
               '2019-09-30', '2019-10-31', '2019-11-30', '2019-12-31',
               '2020-01-31', '2020-02-29', '2020-03-31', '2020-04-30',
               '2020-05-31', '2020-06-30', '2020-07-31', '2020-08-31',
               '2020-09-30', '2020-10-31', '2020-11-30', '2020-12-31',
               '2021-01-31', '2021-02-28', '2021-03-31', '2021-04-30',
               '2021-05-31', '2021-06-30', '2021-07-31', '2021-08-31',
               '2021-09-30', '2021-10-31', '2021-11-30', '2021-12-31',
               '2022-01-31', '2022-02-28', '2022-03-31', '2022-04-30',
```

In [3]:
```python
# Set the date column as the index
data.set_index('date', inplace=True)
```

In [4]:
```python
# Calculate the trend
trend = data.rolling(window=12).mean()
trend.head()
```

Out[4]:

date	values
2018-01-31	NaN
2018-02-28	NaN
2018-03-31	NaN
2018-04-30	NaN

In [6]:
```
# Calculate the seasonal indices
seasonal_indices = data / trend
seasonal_indices.head()
```

Out[6]:

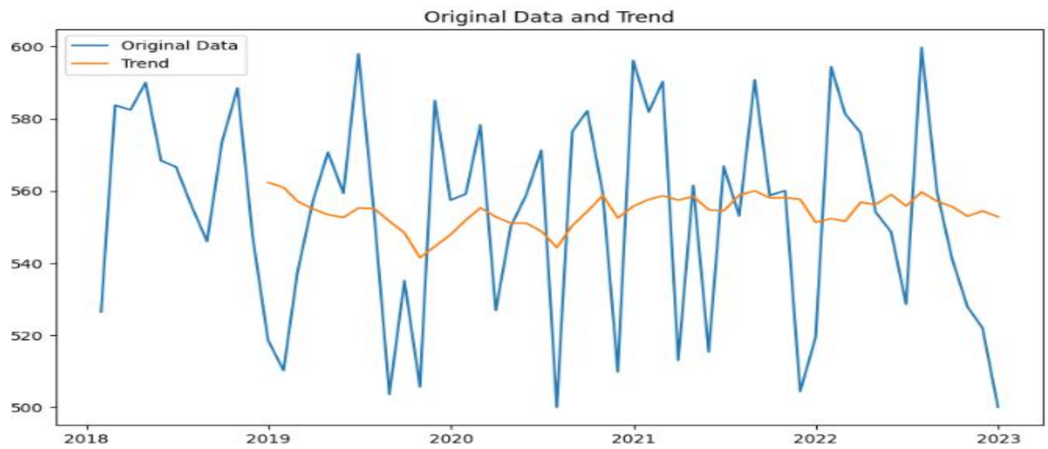
date	values
2018-01-31	NaN
2018-02-28	NaN
2018-03-31	NaN
2018-04-30	NaN
2018-05-31	NaN

In [8]:
```
# Calculate the average seasonal index for each month
average_seasonal_indices = seasonal_indices.groupby(seasonal_indices.index.month).mean()
average_seasonal_indices.head()
```

Out[8]:

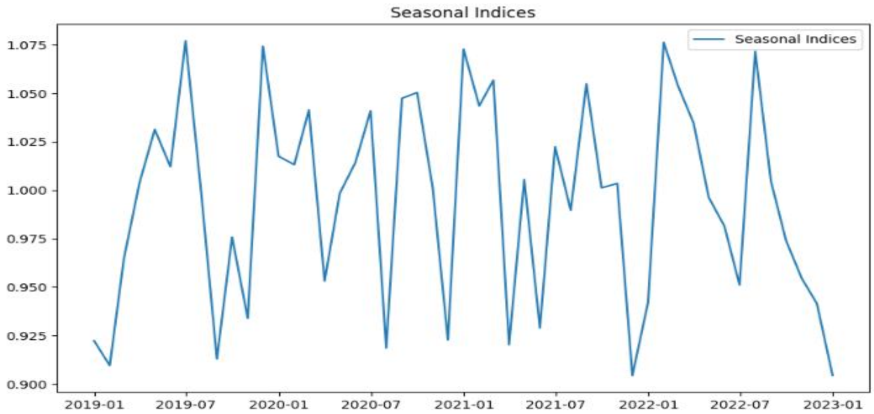
date	values
1	1.010588
2	1.029198
3	0.977996
4	1.007861
5	0.984194

In [9]:
```
# Plot the original data and trend
plt.figure(figsize=(10, 6))
plt.plot(data.index, data.values, label='Original Data')
plt.plot(trend.index, trend.values, label='Trend')
plt.legend()
plt.title('Original Data and Trend')
plt.show()
```



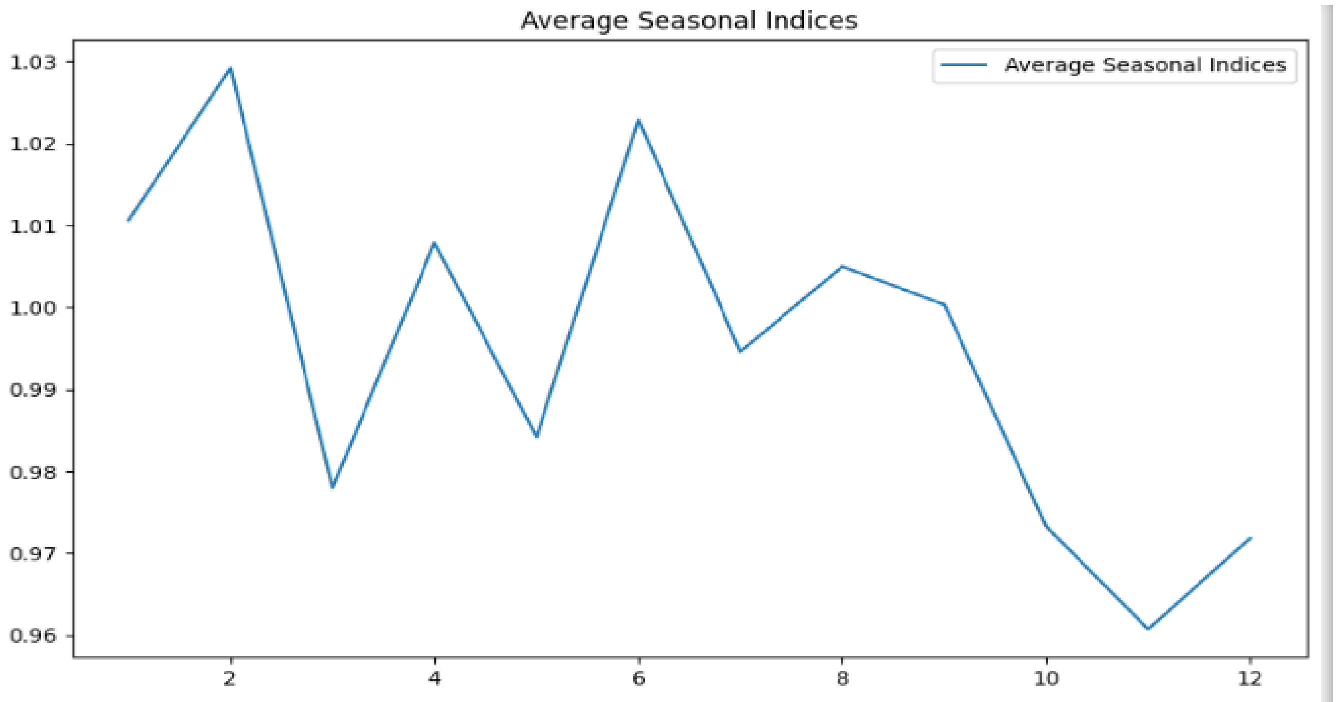
In [11]:

```python

# Plot the average seasonal indices
plt.figure(figsize=(10, 6))
plt.plot(average_seasonal_indices.index, average_seasonal_indices.values, label='Average Seasonal I
plt.legend()
plt.title('Average Seasonal Indices')
plt.show()
```



In [11]:

```python

# Plot the average seasonal indices
plt.figure(figsize=(10, 6))
plt.plot(average_seasonal_indices.index, average_seasonal_indices.values, label='Average Seasonal
plt.legend()
plt.title('Average Seasonal Indices')
plt.show()
```

Practical 6

Aim: Measurement of Seasonal indices Ratio-to-Moving Average method.

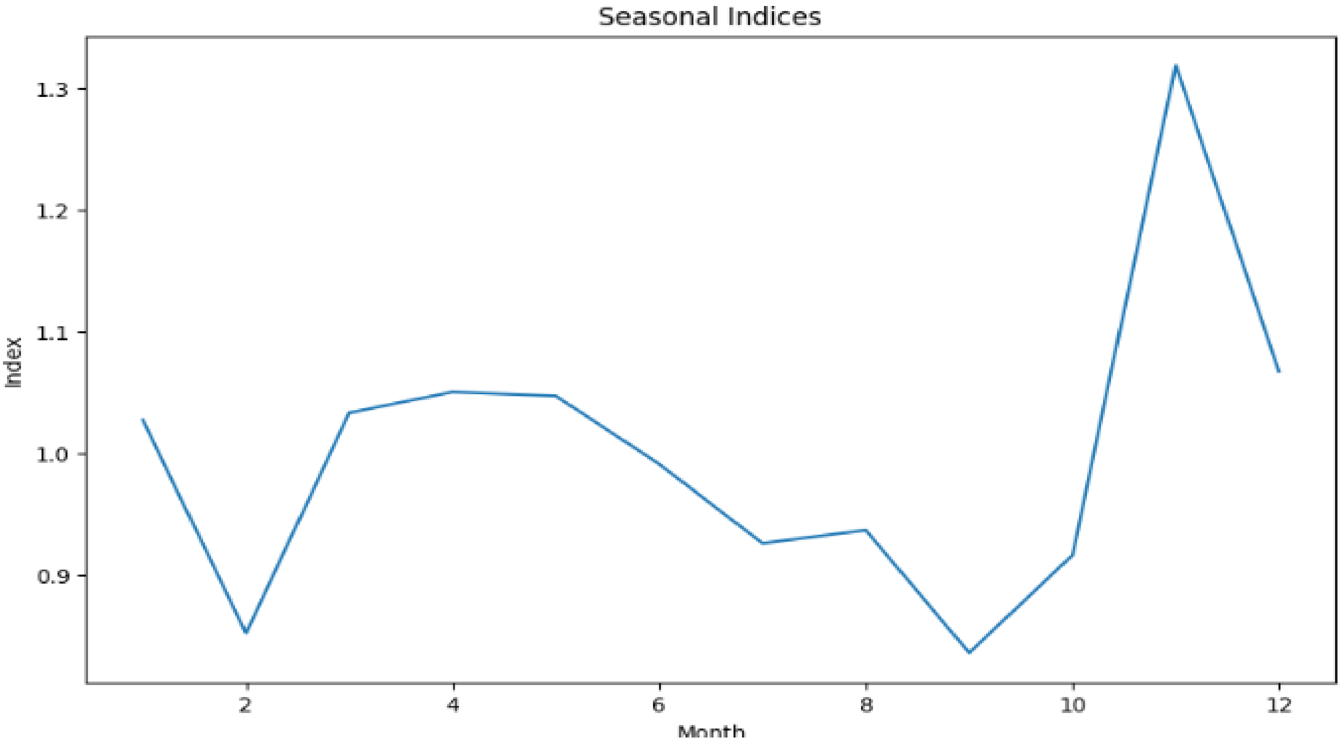
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Create example data
dates = pd.date_range(start='2020-01-01', periods=36, freq='M')
sales = np.random.randint(100, 200, size=len(dates))
data = pd.DataFrame({'Date': dates, 'Sales': sales})
data.set_index('Date', inplace=True)
```

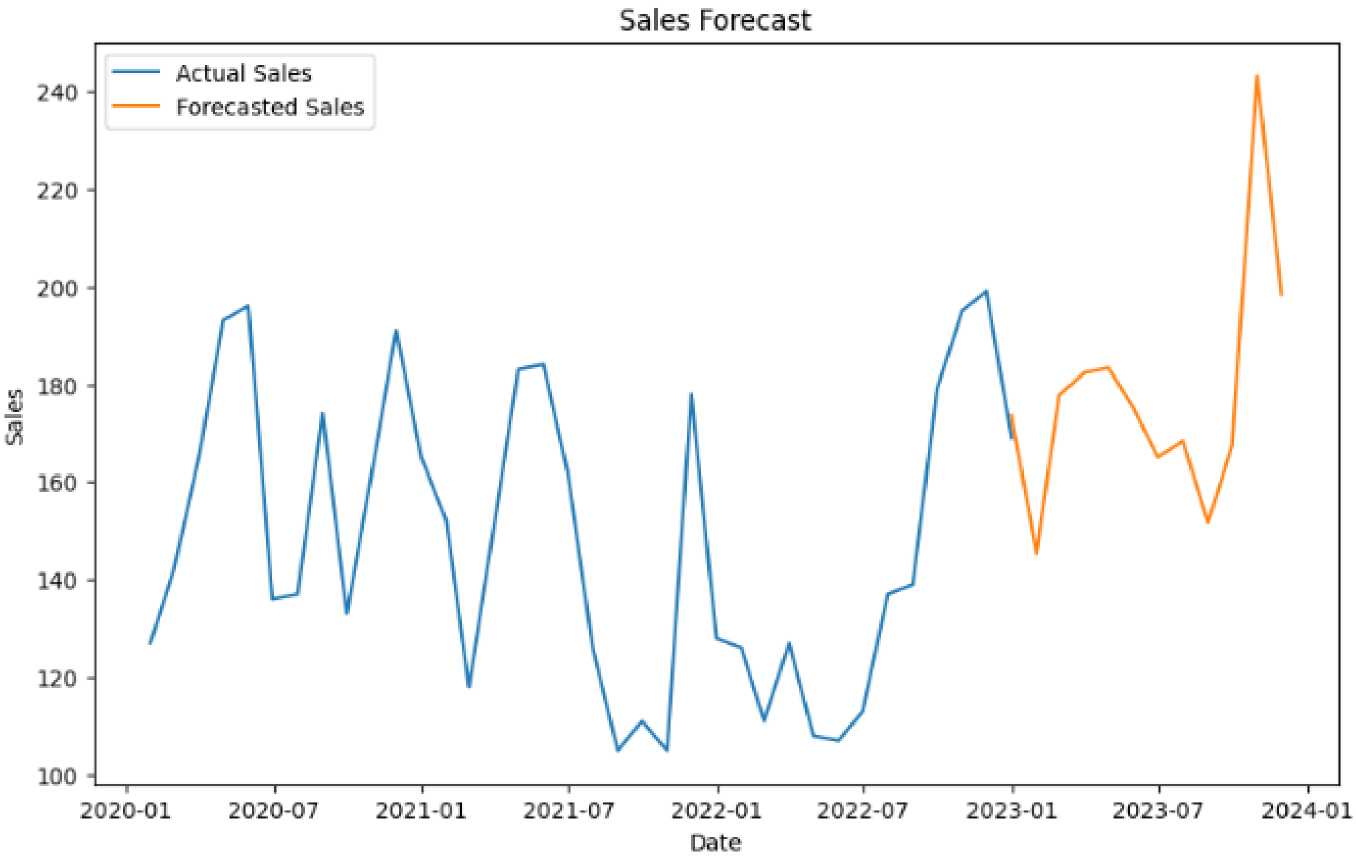
```
# Calculate Moving Averages
data['Moving_Average'] = data['Sales'].rolling(window=12, center=True).mean()

# Calculate Seasonal Indices
data['Ratio'] = data['Sales'] / data['Moving_Average']
monthly_avg = data.groupby(data.index.month)['Ratio'].mean()
seasonal_indices = monthly_avg / monthly_avg.mean()
```

```
# Plot Seasonal Indices
plt.figure(figsize=(10, 6))
seasonal_indices.plot(kind='line')
plt.title('Seasonal Indices')
plt.xlabel('Month')
plt.ylabel('Index')
plt.show()

# Forecasting
trend_forecast = np.linspace(start=data['Sales'].iloc[-1], stop=data['Sales'].iloc[-1]*1.1, num=12
seasonal_forecast = trend_forecast * seasonal_indices.values
```

```
plt.figure(figsize=(10, 6))
plt.plot(data.index, data['Sales'], label='Actual Sales')
plt.plot(pd.date_range(start=data.index[-1], periods=12, freq='M'), seasonal_forecast, label='Fore
plt.title('Sales Forecast')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.legend()
plt.show()
```

Output:


Seasonal Indices
Index
Month
1.3
1.2
1.1
1.0
0.9
2
4
6
8
10
12


Sales Forecast
Actual Sales
Forecasted Sales
Sales
Date
240
220
200
180
160
140
120
100
2020-01
2020-07
2021-01
2021-07
2022-01
2022-07
2023-01
2023-07
2024-01

Year	Month	Sales	Moving Avg	Centered M	Seasonal Relative
2009	Jan	500			
	Feb	600			
	Mar	650			
	Apr	750			
	May	800			
	Jun	800	825		
	Jul	850	829.16667	#########	102.7707809
	Aug	900	825	827.08333	108.8161209
	Sep	900	820.83333	822.91667	109.3670886
	Oct	950	812.5	816.66667	116.3265306
	Nov	1100	804.16667	808.33333	136.0824742
	Dec	1100	795.83333	800	137.5
2010	Jan	550	787.5	791.66667	69.47368421
	Feb	550	775	781.25	70.4
	Mar	600	762.5	768.75	78.04878049
	Apr	650	750	756.25	85.95041322
	May	700	733.33333	741.66667	94.38202247
	Jun	700	725	729.16667	96
	Jul	750	720.83333	722.91667	103.7463977
	Aug	750	725	722.91667	103.7463977
	Sep	750	720.83333	722.91667	103.7463977
	Oct	800	716.66667	718.75	111.3043478
	Nov	900	712.5	714.58333	125.9475219
	Dec	1000	716.66667	714.58333	139.941691
2011	Jan	500	716.66667	716.66667	69.76744186
	Feb	600	725	720.83333	83.23699422
	Mar	550	737.5	731.25	75.21367521
	Apr	600	754.16667	745.83333	80.44692737
	May	650	770.83333	762.5	85.24590164
	Jun	750	787.5	779.16667	96.25668449
	Jul	750	795.83333	791.66667	94.73684211
	Aug	850	800	797.91667	106.5274151
	Sep	900	808.33333	804.16667	111.9170984
	Oct	1000	820.83333	814.58333	122.7621483
	Nov	1100	833.33333	827.08333	132.9974811
	Dec	1200	845.83333	839.58333	142.9280397
2012	Jan	600	866.66667	856.25	70.0729927
	Feb	650	879.16667	872.91667	74.46300716
	Mar	650	891.66667	885.41667	73.41176471
	Apr	750	900	895.83333	83.72093023
	May	800	908.33333	904.16667	88.47926267
	Jun	900	912.5	910.41667	98.85583524
	Jul	1000			
	Aug	1000			
	Sep	1050			
	Oct	1100			
	Nov	1200			
	Dec	1250			

	2009	2010	2011	2012	median	Seasonal Indices
Jan		69.474	69.767	70.073	69.7674	70.0106127
Feb		70.4	83.237	74.463	74.463	74.72254415
Mar		78.049	75.214	73.412	75.2137	75.47582862
Apr		85.95	80.447	83.721	83.7209	84.01273524
May		94.382	85.246	88.479	88.4793	88.78765261
Jun		96	96.257	98.856	96.2567	96.59218223
Jul	102.77	103.75	94.737		102.771	103.1289831
Aug	108.82	103.75	106.53		106.527	106.898711
Sep	109.37	103.75	111.92		109.3671	109.748282
Oct	116.33	111.3	122.76		116.327	116.7319808
Nov	136.08	125.95	133		132.997	133.461037
Dec	137.5	139.94	142.93		139.942	140.4294506
Total	710.86	688.43	711.87		1195.83	1200
Average	118.48	114.74	118.64		99.6527	100

Practical 7

Aim: Measurement of seasonal indices Link Relative method.

Code:

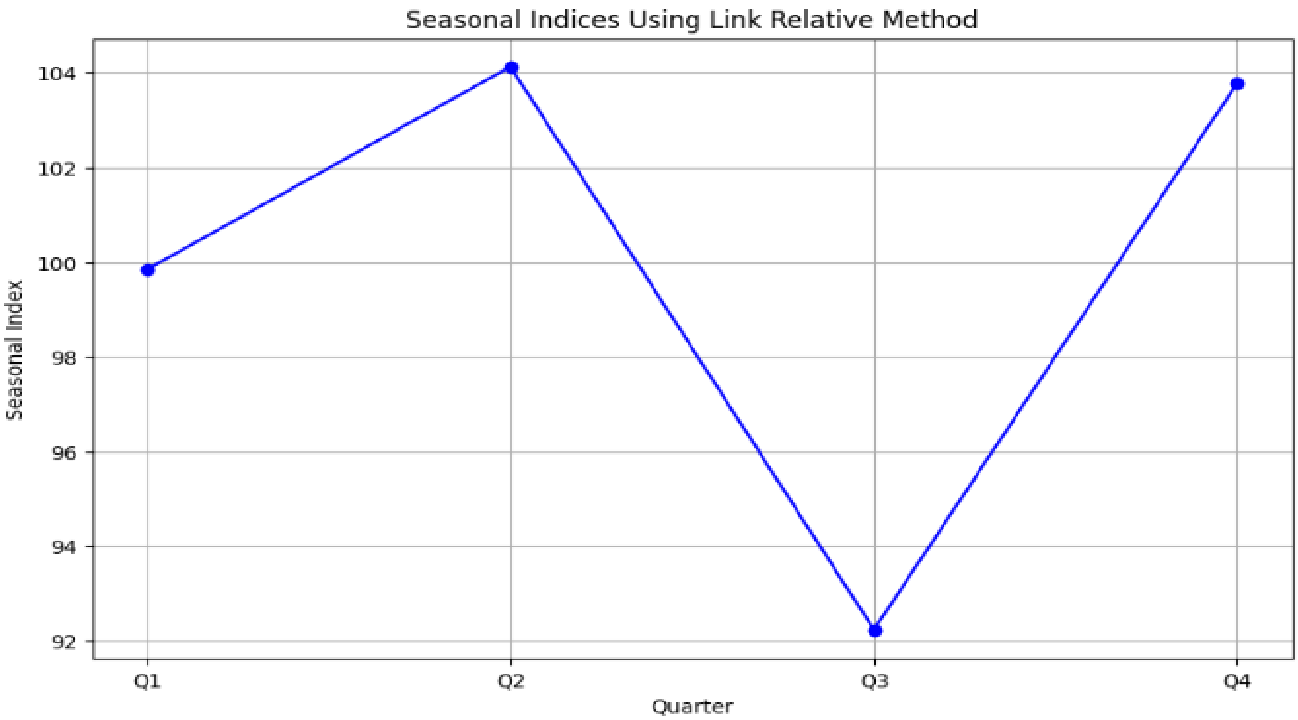
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Sample data: quarterly sales data for 3 years
data = {
    'Year': [2021, 2021, 2021, 2021, 2022, 2022, 2022, 2022, 2023, 2023, 2023, 2023],
    'Quarter': ['Q1', 'Q2', 'Q3', 'Q4', 'Q1', 'Q2', 'Q3', 'Q4', 'Q1', 'Q2', 'Q3', 'Q4'],
    'Sales': [200, 220, 210, 230, 240, 260, 250, 270, 280, 300, 290, 310]
}

df = pd.DataFrame(data)
```

```python
# Calculate link relatives
df['Link_Relative'] = df['Sales'].pct_change() * 100 + 100

```

```python
# Calculate average link relatives for each quarter
average_link_relatives = df.groupby('Quarter')['Link_Relative'].mean()

# Convert to seasonal indices
seasonal_indices = average_link_relatives / average_link_relatives.mean() * 100
```

```python
# Plot the seasonal indices
plt.figure(figsize=(10, 6))
plt.plot(seasonal_indices.index, seasonal_indices.values, marker='o', linestyle='-', color='b')
plt.title('Seasonal Indices Using Link Relative Method')
plt.xlabel('Quarter')
plt.ylabel('Seasonal Index')
plt.grid(True)
plt.show()

```

Output:



Practical 8

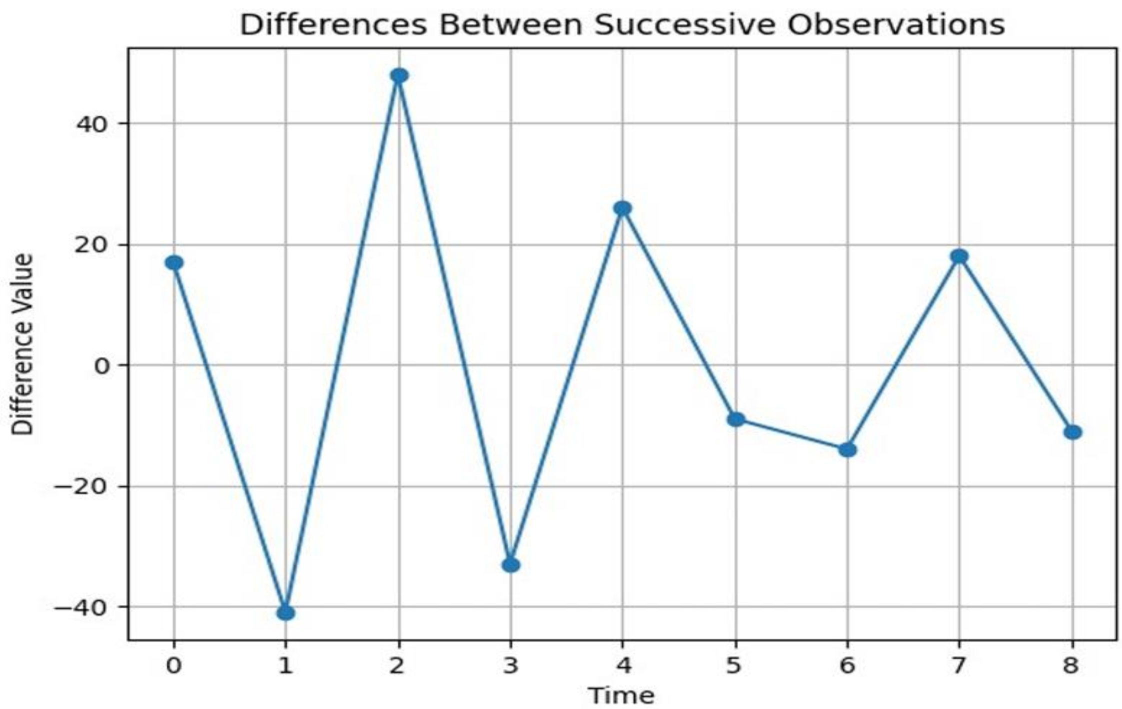
Aim: Calculation of variance of random component by variate difference method.

Code:

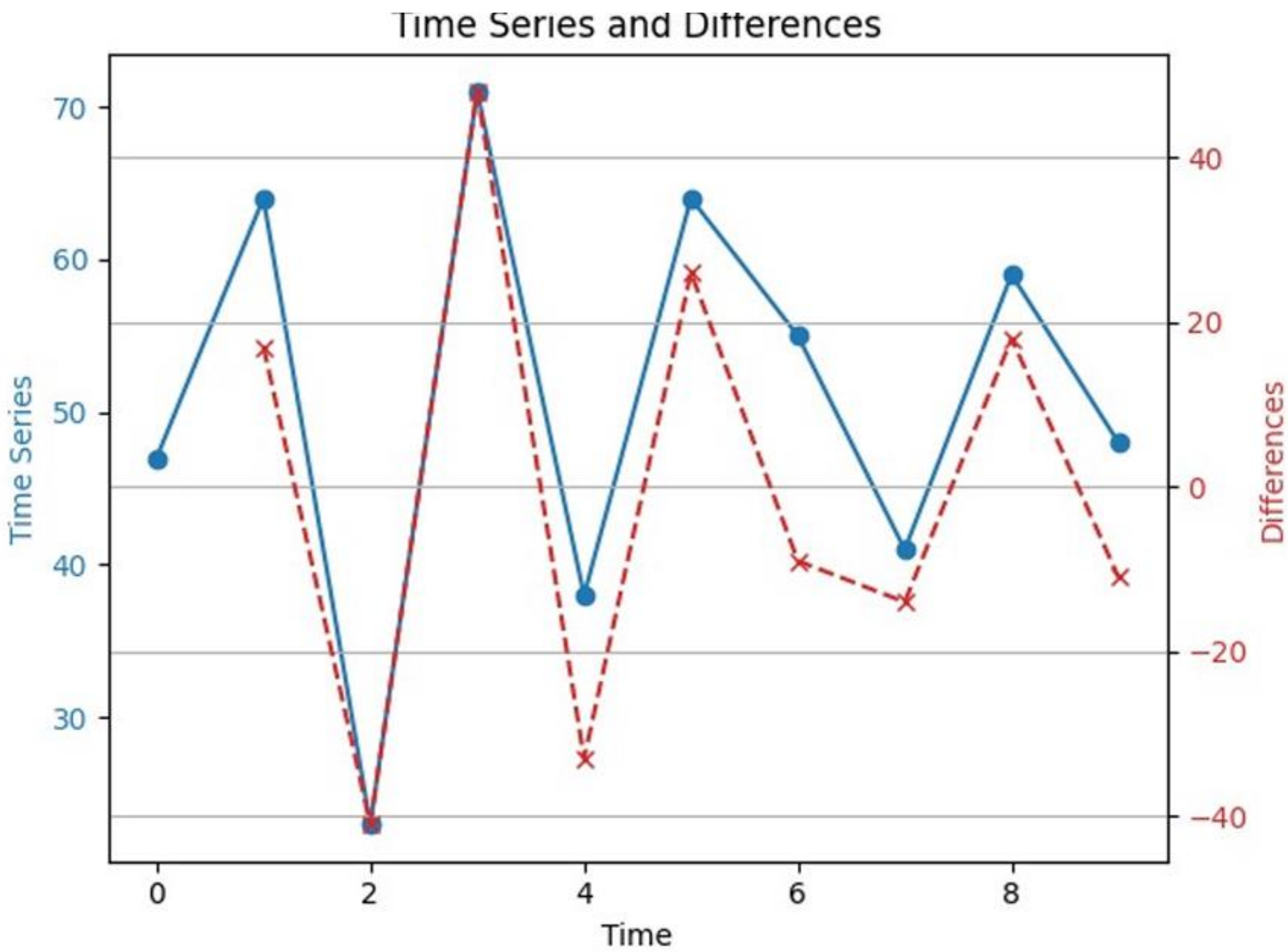
```
import numpy as np
import matplotlib.pyplot as plt

# Given time series data
time_series = [47, 64, 23, 71, 38, 64, 55, 41, 59, 48]

# Calculate differences
differences = np.diff(time_series)

# Calculate mean of differences
mean_diff = np.mean(differences)

# Calculate variance of differences
var_diff = np.var(differences, ddof=1)

# Calculate variance of random components
var_random = var_diff / 2

# Print results
print(f"Mean of Differences: {mean_diff}")
print(f"Variance of Differences: {var_diff}")
print(f"Variance of Random Components: {var_random}")
```

```
Mean of Differences: 0.1111111111111111
Variance of Differences: 845.1111111111112
Variance of Random Components: 422.5555555555556
```

```
# Plot the differences
plt.plot(differences, marker='o')
plt.title('Differences Between Successive Observations')
plt.xlabel('Time')
plt.ylabel('Difference Value')
plt.grid(True)
plt.show()
```



```python
# Create a figure and axis
fig, ax1 = plt.subplots()

# Plot the time series on the primary y-axis
color = 'tab:blue'
ax1.set_xlabel('Time')
ax1.set_ylabel('Time Series', color=color)
ax1.plot(time_series, marker='o', color=color, label='Time Series')
ax1.tick_params(axis='y', labelcolor=color)

# Create a secondary y-axis for the differences
ax2 = ax1.twinx()
color = 'tab:red'
ax2.set_ylabel('Differences', color=color)
ax2.plot(range(1, len(time_series)), differences, marker='x',
linestyle='--', color=color, label='Differences')
ax2.tick_params(axis='y', labelcolor=color)

# Add title and grid
plt.title('Time Series and Differences')
fig.tight_layout()  # Adjust layout to make room for both y-axes
plt.grid(True)
plt.show()
```



Practical 9

Aim: Forecasting by exponential smoothing.

In [20]:
```
import pandas as pd
from statsmodels.tsa.api import SimpleExpSmoothing, Holt
import matplotlib.pyplot as plt

```

In [21]:
```
# Example oil production data
data = [446.6565, 454.4733, 455.663, 423.6322, 456.2713, 440.5881, 425.3325, 485.1494, 506.0482, 5
index = pd.date_range(start="1996", end="2008", freq="Y")
oildata = pd.Series(data, index)
```

In [23]:
```
# Fit the SES model
model = SimpleExpSmoothing(oildata).fit(smoothing_level=0.2, optimized=False)
model
```

Out[23]:
```
<statsmodels.tsa.holtwinters.results.HoltWintersResultsWrapper at 0x1a1b021b040>
```

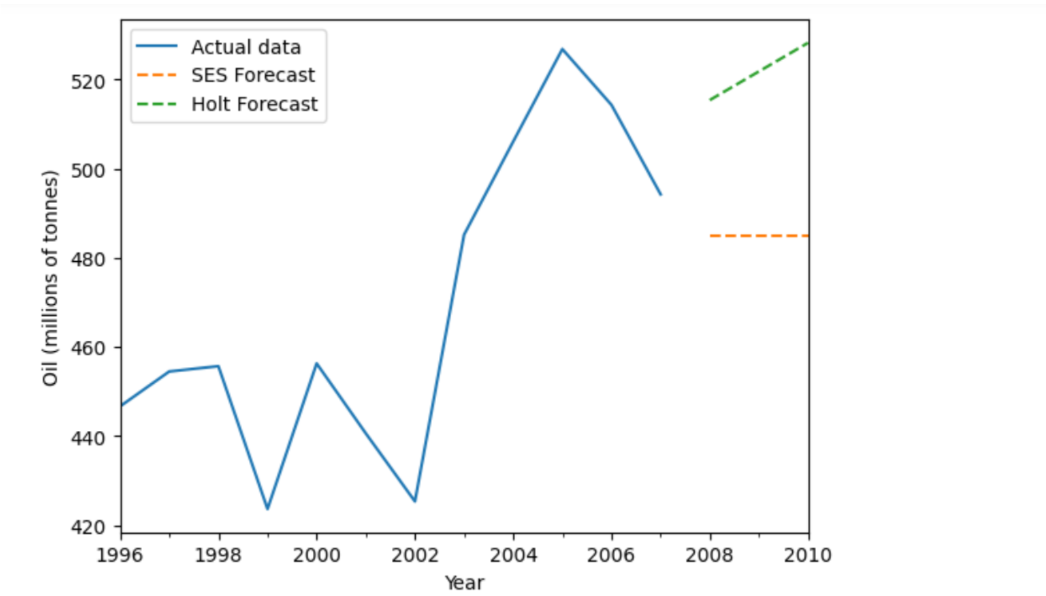
In [25]:
```

# Fit Holt's model
model_holt = Holt(oildata).fit(smoothing_level=0.6, smoothing_slope=0.2, optimized=False)
model_holt
```

```
C:\Users\sanga\AppData\Local\Temp\ipykernel_18852\2628540625.py:2: FutureWarning: the 'smoothing_slop
e' keyword is deprecated, use 'smoothing_trend' instead.
  model_holt = Holt(oildata).fit(smoothing_level=0.6, smoothing_slope=0.2, optimized=False)
```

Out[25]:
```
<statsmodels.tsa.holtwinters.results.HoltWintersResultsWrapper at 0x1a1b05e6170>
```

In [27]:
```
# Forecast 3 steps ahead
forecast_ses = model.forecast(steps=3)
forecast_holt = model_holt.forecast(steps=3)
forecast_holt
forecast_ses
```

Out[27]:
```
2008-12-31    484.802465
2009-12-31    484.802465
2010-12-31    484.802465
Freq: A-DEC, dtype: float64
```



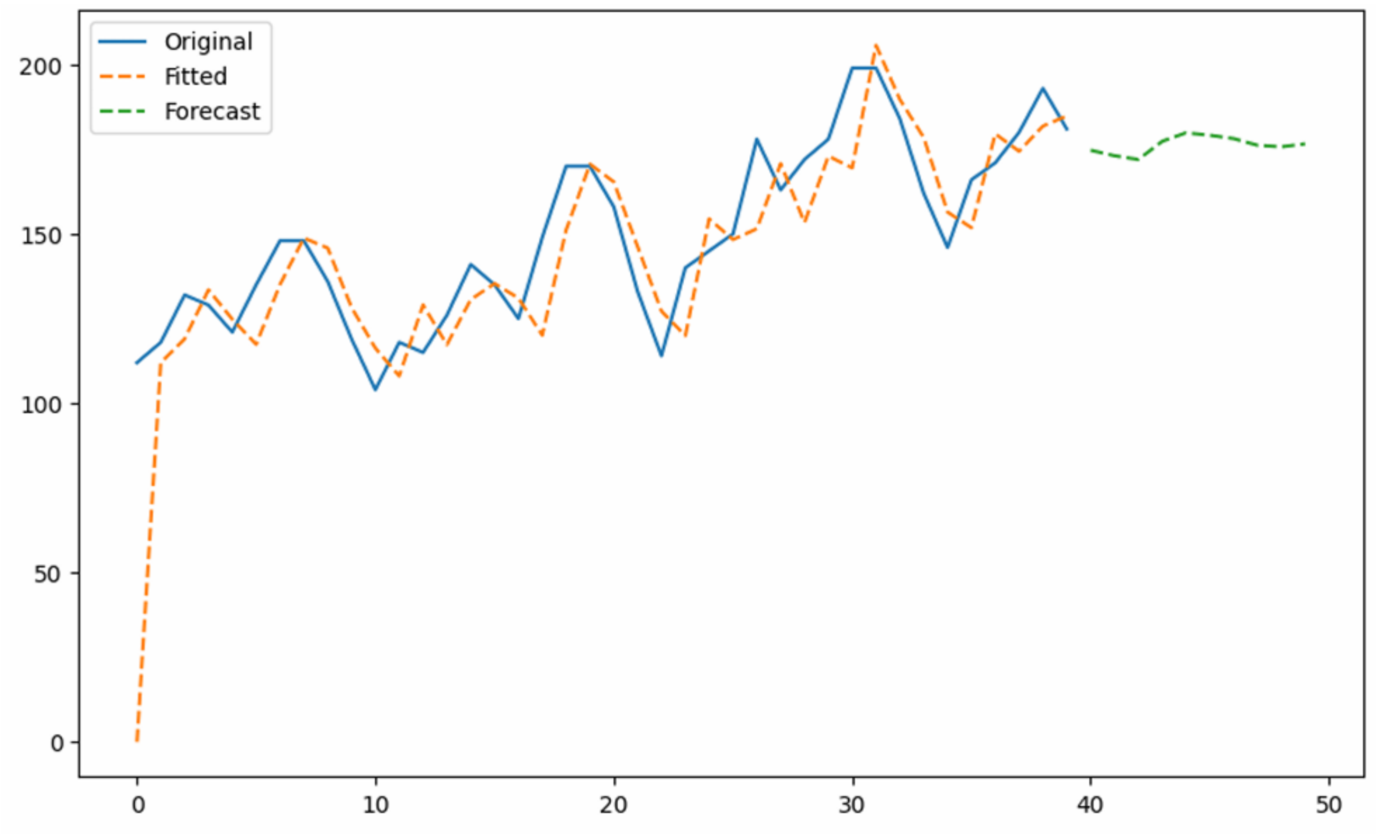
Practical 10

Aim: Forecasting by short term forecasting methods.

ARIMA: ARIMA models are more sophisticated and take into account the autocorrelations in the data.

]:

```
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA

# Sample data
data = [112, 118, 132, 129, 121, 135, 148, 148, 136, 119, 104, 118, 115, 12
        158, 133, 114, 140, 145, 150, 178, 163, 172, 178, 199, 199, 184, 16

# Convert data to pandas DataFrame
df = pd.DataFrame(data, columns=['value'])

# Fit the ARIMA model
model = ARIMA(df['value'], order=(5, 1, 1))  # (p,d,q) order
fit = model.fit()

# Forecast future values
forecast = fit.forecast(steps=10)  # Forecast next 10 periods

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(df['value'], label='Original')
plt.plot(fit.fittedvalues, label='Fitted', linestyle='--')
plt.plot(range(len(df), len(df) + 10), forecast, label='Forecast', linestyl
plt.legend()
plt.show()
```




```
40    174.739132
41    173.090693
42    171.980484
43    177.340726
44    179.873947
45    179.150980
46    178.149276
47    176.194698
48    175.731596
49    176.627380
Name: predicted_mean, dtype: float64
```

1. p (AutoRegressive part): If p=5, the model uses the previous 5 values to predict the current value.
2. d (Differencing part): If d=1, the model uses the difference of the observations (e.g., y2-y1) to make the series stationary.
3. q (Moving Average part): If q=0, no lagged forecast errors are included.