In [1]:
```
import pandas as pd
import pandas.plotting
import matplotlib.pyplot as plt
import numpy as np
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import adfuller
%matplotlib inline
```

To analyze the behavior of a multivariate time series and identify any trends or patterns, plotting the data is a useful approach. By visualizing the data, you can gain insights into its characteristics and identify any underlying patterns or relationships.

In [2]:
```
file = r'C:\Users\Sidharth\Desktop\798Q\.ipynb_checkpoints\data.csv.csv'
df=pd.read_csv(file,index_col=0)
```

In [3]:
```
data=df.copy()
```

In [4]:
```
data=data[:-3]
data
```

Out[4]:

Sn	From	to	pm10	pm2.5	NO	NO2	NOX	CO	SO2	NH3	Ozone	Benzene
1	01-02-2023 00:00	01-02-2023 00:15	95.0	35.0	NaN	90.1	56.2	0.31	NaN	17.7	28.1	0.4
2	01-02-2023 00:15	01-02-2023 00:30	95.0	35.0	NaN	88.0	55.1	0.33	NaN	18.3	27.1	0.4
3	01-02-2023 00:30	01-02-2023 00:45	95.0	35.0	NaN	87.7	55.2	0.38	NaN	19.7	24.9	0.4
4	01-02-2023 00:45	01-02-2023 01:00	122.0	34.0	NaN	88.9	55.7	0.38	NaN	21.3	21.9	0.4
5	01-02-2023 01:00	01-02-2023 01:15	122.0	34.0	NaN	90.0	55.8	0.38	NaN	22.3	16.7	0.4
...	...	...	...	...	...	...	...	...	...	...	...	...
8636	01-05-2023 22:45	01-05-2023 23:00	19.0	11.0	17.9	100.0	67.8	0.63	10.0	10.7	26.1	0.1
8637	01-05-2023 23:00	01-05-2023 23:15	19.0	11.0	17.9	100.0	67.7	0.57	10.0	10.4	30.9	0.1
8638	01-05-2023 23:15	01-05-2023 23:30	19.0	11.0	19.6	100.2	69.2	0.58	9.9	10.5	29.6	0.1
8639	01-05-2023 23:30	01-05-2023 23:45	19.0	11.0	20.8	100.2	70.2	0.58	9.5	10.8	30.0	0.1
8640	01-05-2023 23:45	02-05-2023 00:00	32.0	6.0	21.8	98.8	70.3	NaN	NaN	11.0	33.5	0.1

8640 rows × 12 columns

Analyzing multivariate time series data that contains NaN (missing) values replaced with zero values requires special attention to ensure accurate interpretation and meaningful analysis. Here's an overview of the theory behind analyzing such multivariate time series:

Consider the nature of missing data: Missing data can occur in multivariate time series due to various reasons such as measurement errors, sensor failures, or data collection issues. Replacing missing values with zeros assumes that the missing values indicate zero measurements rather than true zero values. It is essential to understand the context and implications of replacing missing values with zeros for accurate analysis.

Data preprocessing: Before analyzing the multivariate time series, it is crucial to preprocess the data properly. This includes handling missing values, transforming the data if necessary, and ensuring consistency in the data format. Replacing NaN values with zeros is just one approach to handle missing data, but it may not always be the most appropriate. Consider

alternative methods such as interpolation or imputation techniques based on the specific characteristics of your data.

Interpretation of zero values: When missing values are replaced with zeros, it is important to interpret zero values correctly. Zero values in the time series can represent either actual zero measurements or missing values that were replaced with zeros. zero values may impact the analysis results, particularly in calculations involving averages, trends, and statistical measures.

Visual exploration: Visualizing the multivariate time series is an effective way to understand its behavior. By plotting the data, you can observe patterns, trends, and relationships between variables. However, be cautious when interpreting the zero values in the plots, as they may not reflect true zero measurements.

Analysis techniques: When analyzing multivariate time series with replaced zero values, be aware of the potential biases introduced by the replacement. The presence of zero values can affect various analysis techniques such as correlation analysis, regression modeling, or time series forecasting. Consider the implications of zero values in the specific analysis techniques used and adjust the interpretation accordingly.

Sensitivity analysis: It is important to perform sensitivity analysis to assess the impact of replacing missing values with zeros on the analysis results. This involves comparing the outcomes obtained with zero replacements to alternative approaches, such as imputation methods or excluding the affected time periods altogether. Sensitivity analysis helps evaluate the robustness of the findings and provides insights into the potential biases introduced by the zero replacements.

Overall, analyzing multivariate time series data with NaN values replaced by zero values requires careful consideration of the implications. It is recommended to explore alternative methods for handling missing data and perform sensitivity analysis to ensure the validity of the analysis results.

In [5]:
```
NO='NO'
data[NO].fillna(0,inplace=True)
NO2='NO2'

data[NO2].fillna(0, inplace=True)
NOX='NOX'
avg_NOX = data[NOX].mean()
data[NOX].fillna(0, inplace=True)
CO='CO'
avg_CO = data[CO].mean()
data[CO].fillna(0, inplace=True)
SO2='SO2'
avg_SO2 = data[SO2].mean()
data[SO2].fillna(0, inplace=True)
NH3='NH3'
avg_NH3 = data[NH3].mean()
data[NH3].fillna(0, inplace=True)
Ozone='Ozone'
avg_Ozone = data[Ozone].mean()
data[Ozone].fillna(0, inplace=True)
Benzene='Benzene'
avg_Benzene = data[Benzene].mean()
```

```
data[Benzene].fillna(0, inplace=True)
pm10='pm10'
avg_pm10 = data[pm10].mean()
data[pm10].fillna(0, inplace=True)
pm25='pm2.5'
avg_pm25 = data[pm25].mean()
data[pm25].fillna(0, inplace=True)
```

In [6]:
```
 data.drop(['to'], axis=1, inplace=True)
```

In [7]:
```
data.columns
```

Out[7]:
```
Index(['From', 'pm10', 'pm2.5', 'NO', 'NO2', 'NOX', 'CO', 'SO2', 'NH3',
       'Ozone', 'Benzene'],
      dtype='object')
```

In [8]:
```
data['From'] = pd.to_datetime(data['From'], format = '%d-%m-%Y %H:%M')
```

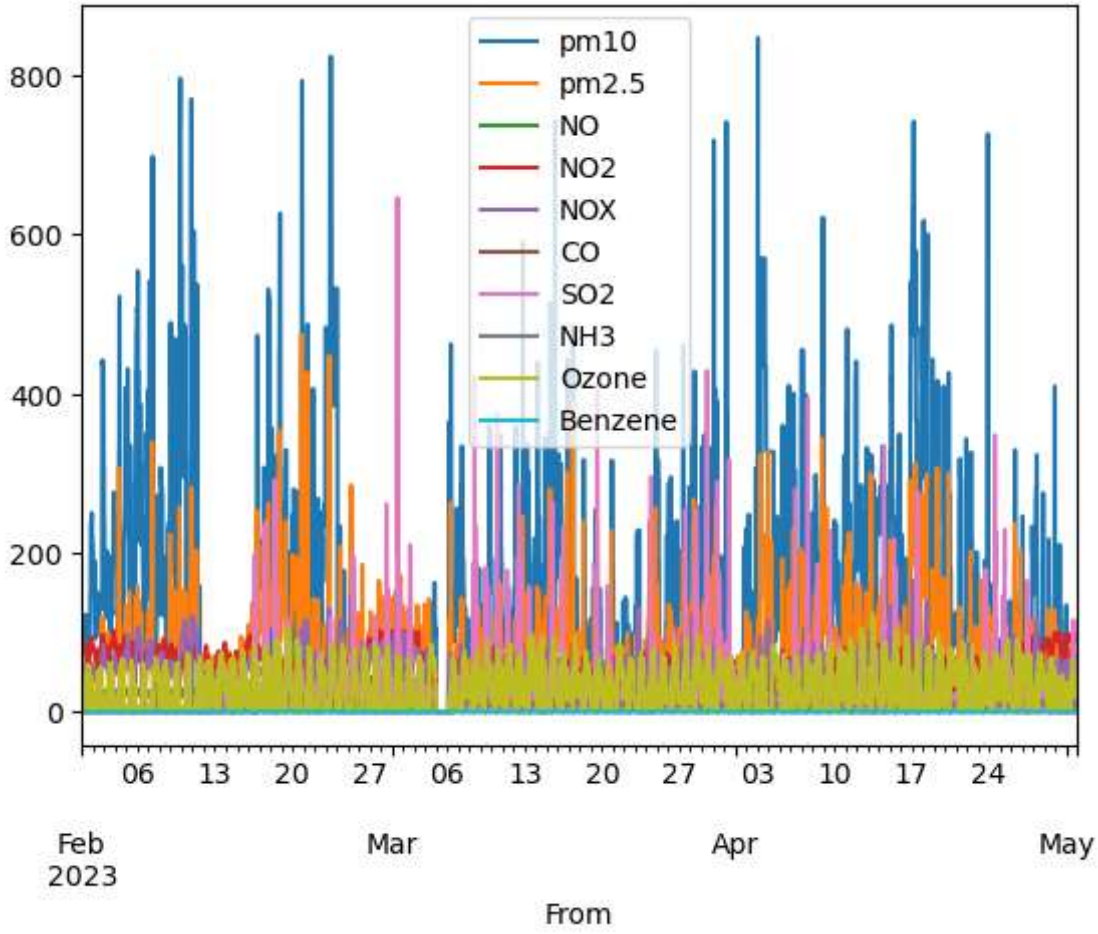
In [9]:
```
data['From']
```

Out[9]:
```
Sn
1      2023-02-01 00:00:00
2      2023-02-01 00:15:00
3      2023-02-01 00:30:00
4      2023-02-01 00:45:00
5      2023-02-01 01:00:00
               ...
8636   2023-05-01 22:45:00
8637   2023-05-01 23:00:00
8638   2023-05-01 23:15:00
8639   2023-05-01 23:30:00
8640   2023-05-01 23:45:00
Name: From, Length: 8640, dtype: datetime64[ns]
```

In [10]:
```
data.set_index(['From'], inplace=True)
```

In [11]:
```
data.plot()
```

Out[11]:
```
<Axes: xlabel='From'>
```



In [12]:
```
NO_C = data['NO']
NO2_C = data['NO2']
NOX_C = data['NOX']
CO_C = data['CO']
SO2_C = data['SO2']
NH3_C = data['NH3']
Ozone_C= data['Ozone']
Benzene_c = data['Benzene']
pm10_C = data['pm10']
pm25_C = data['pm2.5']
```

In [13]:
```
data
```

Out[13]:

From	pm10	pm2.5	NO	NO2	NOX	CO	SO2	NH3	Ozone	Benzene
2023-02-01 00:00:00	95.0	35.0	0.0	90.1	56.2	0.31	0.0	17.7	28.1	0.4
2023-02-01 00:15:00	95.0	35.0	0.0	88.0	55.1	0.33	0.0	18.3	27.1	0.4
2023-02-01 00:30:00	95.0	35.0	0.0	87.7	55.2	0.38	0.0	19.7	24.9	0.4
2023-02-01 00:45:00	122.0	34.0	0.0	88.9	55.7	0.38	0.0	21.3	21.9	0.4
2023-02-01 01:00:00	122.0	34.0	0.0	90.0	55.8	0.38	0.0	22.3	16.7	0.4
...	...	...	...	...	...	...	...	...	...	...
2023-05-01 22:45:00	19.0	11.0	17.9	100.0	67.8	0.63	10.0	10.7	26.1	0.1
2023-05-01 23:00:00	19.0	11.0	17.9	100.0	67.7	0.57	10.0	10.4	30.9	0.1
2023-05-01 23:15:00	19.0	11.0	19.6	100.2	69.2	0.58	9.9	10.5	29.6	0.1
2023-05-01 23:30:00	19.0	11.0	20.8	100.2	70.2	0.58	9.5	10.8	30.0	0.1
2023-05-01 23:45:00	32.0	6.0	21.8	98.8	70.3	0.00	0.0	11.0	33.5	0.1

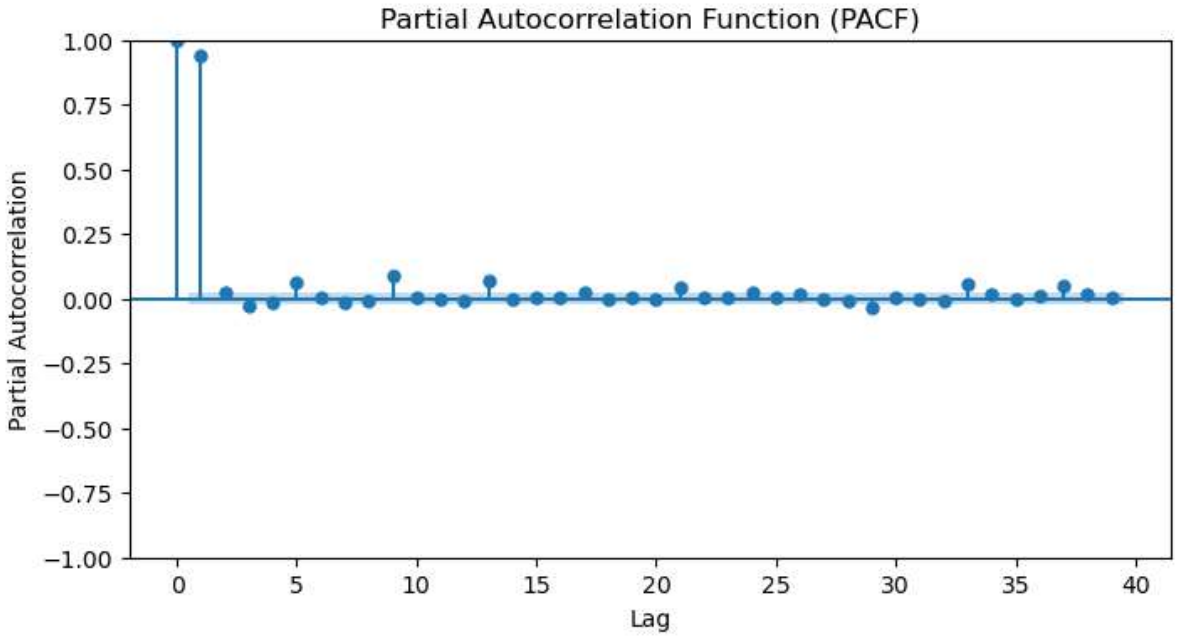
8640 rows × 10 columns

In [14]:
```
training = data.index[7540]
msk= (data.index <= training)
data_train = data[msk].copy()
data_test = data[~msk].copy()
```

In [15]:
```
fig, ax = plt.subplots(figsize=(8, 4))
plot_pacf(data_train['pm10'], ax=ax)
ax.set_xlabel('Lag')
ax.set_ylabel('Partial Autocorrelation')
ax.set_title('Partial Autocorrelation Function (PACF)')
```

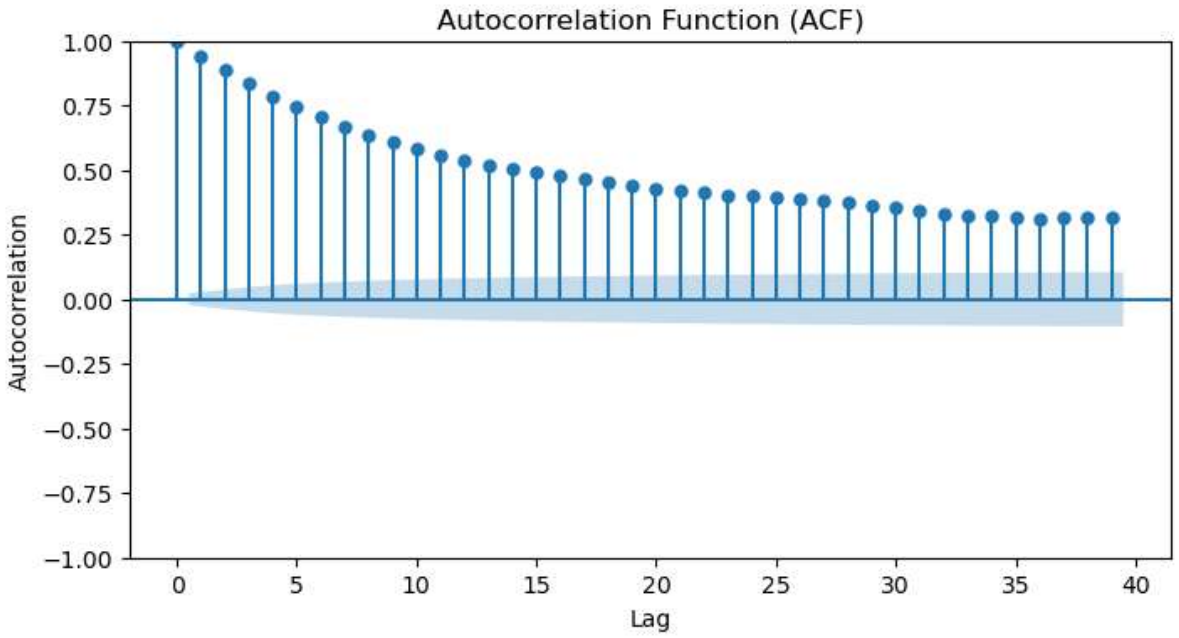
```
C:\ProgramData\anaconda3\lib\site-packages\statsmodels\graphics\tsaplots.py:348: F
utureWarning: The default method 'yw' can produce PACF values outside of the [-1,
1] interval. After 0.13, the default will change tounadjusted Yule-Walker ('ywm').
You can use this method now by setting method='ywm'.
  warnings.warn(
```

Out[15]:
```
Text(0.5, 1.0, 'Partial Autocorrelation Function (PACF)')
```



In [16]:
```
fig, ax = plt.subplots(figsize=(8, 4))
plot_acf(data_train['pm10'], ax=ax)
ax.set_xlabel('Lag')
ax.set_ylabel('Autocorrelation')
ax.set_title('Autocorrelation Function (ACF)')
```

Out[16]:
```
Text(0.5, 1.0, 'Autocorrelation Function (ACF)')
```



In [17]:
```
plt.show()
```

In [18]:
```
pm10_CC=pm10_C[1:7540]

# Perform ADF test
result = adfuller(pm10_CC)

# Extract test statistics and p-value
adf_statistic = result[0]
p_value = result[1]

# Print the test statistics and p-value
```

```
print("ADF Statistic:", adf_statistic)
print("p-value:", p_value)
```

```
ADF Statistic: -7.585551073161325
p-value: 2.612567982827315e-11
```

In [19]:
```
from pmdarima import auto_arima
```

In [20]:
```
from pmdarima.arima import auto_arima
def arimamodel(timeseriesarray) :
 autoarima_model = auto_arima(timeseriesarray, start_p=0, start_q=0, d=0, max_p=5,
 return autoarima_model
model = arimamodel(data_train['pm10'])
model.summary()
```

Out[20]:

SARIMAX Results

Dep. Variable:	y	**No. Observations:**	7541
Model:	SARIMAX(3, 0, 2)	**Log Likelihood**	-40051.179
Date:	Wed, 28 Jun 2023	**AIC**	80116.358
Time:	20:25:27	**BIC**	80164.855
Sample:	02-01-2023	**HQIC**	80133.007
	- 04-20-2023		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
intercept	16.0662	2.840	5.657	0.000	10.500	21.633
ar.L1	0.8973	0.026	34.614	0.000	0.846	0.948
ar.L2	-0.7990	0.033	-24.048	0.000	-0.864	-0.734
ar.L3	0.7902	0.022	36.559	0.000	0.748	0.833
ma.L1	0.0353	0.022	1.621	0.105	-0.007	0.078
ma.L2	0.8947	0.021	43.391	0.000	0.854	0.935
sigma2	2486.1247	12.544	198.194	0.000	2461.539	2510.710

Ljung-Box (L1) (Q):	1.79	**Jarque-Bera (JB):**	924634.76
Prob(Q):	0.18	**Prob(JB):**	0.00
Heteroskedasticity (H):	1.12	**Skew:**	0.45
Prob(H) (two-sided):	0.01	**Kurtosis:**	57.24

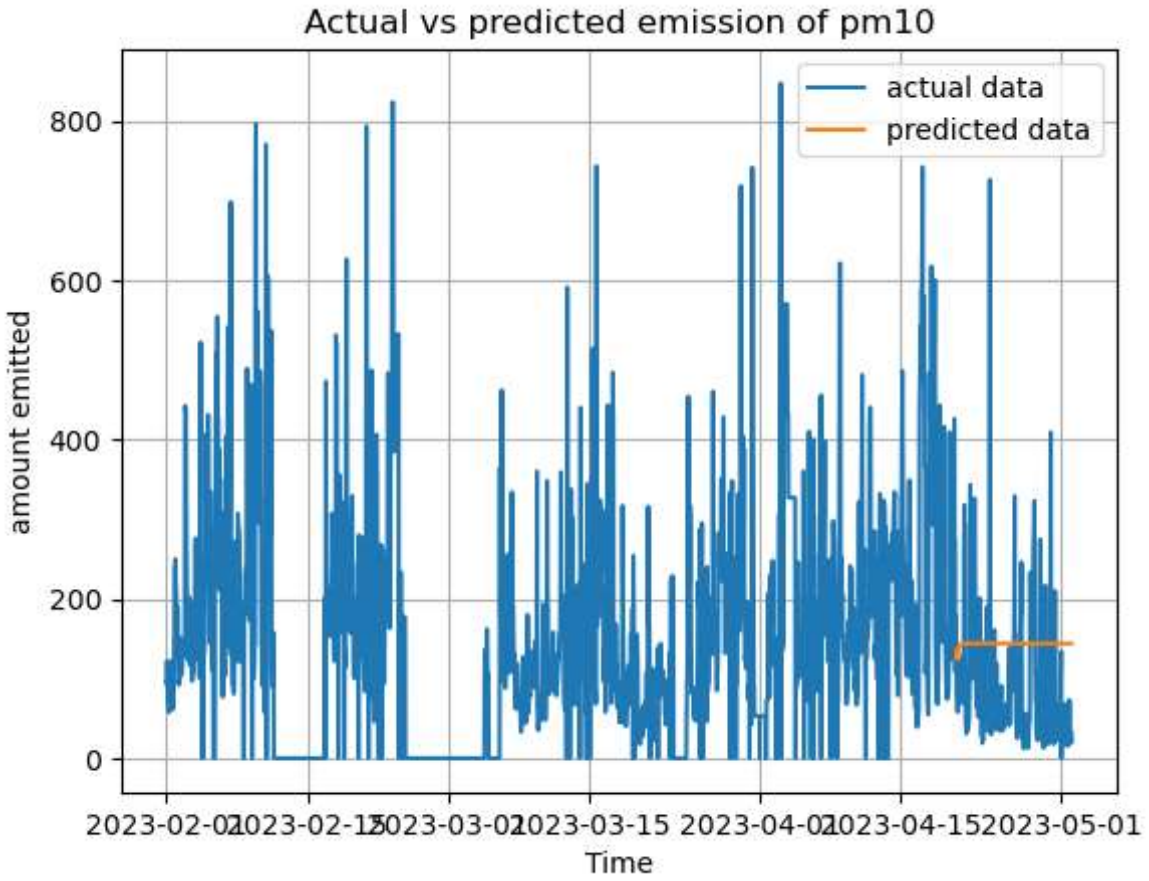
Warnings:

[1] Covariance matrix calculated using the outer product of gradients (complex-step).

In [21]:
```
auto = model.predict(n_periods = len(data_test['pm10']))
data['pm10 forecast'] = [None]*len(data_train['pm10']) + list(auto)
```

In [22]:
```
 %matplotlib inline

#matplotlib notebook
```

```
plt.plot(pm10_C,label='actual data')
plt.plot(data['pm10 forecast'], label='predicted data')
plt.xlabel('Time')
plt.ylabel('amount emitted')
plt.title('Actual vs predicted emission of pm10')
plt.grid()
plt.legend()
plt.show()
```



In [23]:
```
ape = np.abs((data_test['pm10'] - data['pm10 forecast']) / data_test['pm10'])
mape = np.mean(ape) * 100

print("Mean Absolute Percentage Error:", mape)
```

```
Mean Absolute Percentage Error:     40.5648123
```

In [ ]:
```

```

In [1]:
```
import pandas as pd
import pandas.plotting
import matplotlib.pyplot as plt
import numpy as np
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import adfuller
%matplotlib inline
```

ploting multivariate by replacing na values with mean

In [2]:
```
file = r'C:\Users\Sidharth\Desktop\798Q\.ipynb_checkpoints\data.csv.csv'
df=pd.read_csv(file,index_col=0)
```

In [3]:
```
data=df.copy()
```

In [4]:
```
data=data[:-3]
data
```

Out[4]:

Sn	From	to	pm10	pm2.5	NO	NO2	NOX	CO	SO2	NH3	Ozone	Benzene
1	01-02-2023 00:00	01-02-2023 00:15	95.0	35.0	NaN	90.1	56.2	0.31	NaN	17.7	28.1	0.4
2	01-02-2023 00:15	01-02-2023 00:30	95.0	35.0	NaN	88.0	55.1	0.33	NaN	18.3	27.1	0.4
3	01-02-2023 00:30	01-02-2023 00:45	95.0	35.0	NaN	87.7	55.2	0.38	NaN	19.7	24.9	0.4
4	01-02-2023 00:45	01-02-2023 01:00	122.0	34.0	NaN	88.9	55.7	0.38	NaN	21.3	21.9	0.4
5	01-02-2023 01:00	01-02-2023 01:15	122.0	34.0	NaN	90.0	55.8	0.38	NaN	22.3	16.7	0.4
...	...	...	...	...	...	...	...	...	...	...	...	...
8636	01-05-2023 22:45	01-05-2023 23:00	19.0	11.0	17.9	100.0	67.8	0.63	10.0	10.7	26.1	0.1
8637	01-05-2023 23:00	01-05-2023 23:15	19.0	11.0	17.9	100.0	67.7	0.57	10.0	10.4	30.9	0.1
8638	01-05-2023 23:15	01-05-2023 23:30	19.0	11.0	19.6	100.2	69.2	0.58	9.9	10.5	29.6	0.1
8639	01-05-2023 23:30	01-05-2023 23:45	19.0	11.0	20.8	100.2	70.2	0.58	9.5	10.8	30.0	0.1
8640	01-05-2023 23:45	02-05-2023 00:00	32.0	6.0	21.8	98.8	70.3	NaN	NaN	11.0	33.5	0.1

8640 rows × 12 columns

In [5]:

```
NO='NO'
avg_NO = data[NO].mean()
data[NO].fillna(avg_NO, inplace=True)
NO2='NO2'
avg_NO2 = data[NO2].mean()
data[NO2].fillna(avg_NO2, inplace=True)
NOX='NOX'
avg_NOX = data[NOX].mean()
data[NOX].fillna(avg_NOX, inplace=True)
CO='CO'
avg_CO = data[CO].mean()
data[CO].fillna(avg_CO, inplace=True)
SO2='SO2'
avg_SO2 = data[SO2].mean()
data[SO2].fillna(avg_SO2, inplace=True)
NH3='NH3'
avg_NH3 = data[NH3].mean()
data[NH3].fillna(avg_NH3, inplace=True)
```

```
Ozone='Ozone'
avg_Ozone = data[Ozone].mean()
data[Ozone].fillna(avg_Ozone, inplace=True)
Benzene='Benzene'
avg_Benzene = data[Benzene].mean()
data[Benzene].fillna(avg_Benzene, inplace=True)
pm10='pm10'
avg_pm10 = data[pm10].mean()
data[pm10].fillna(avg_pm10, inplace=True)
pm25='pm2.5'
avg_pm25 = data[pm25].mean()
data[pm25].fillna(avg_pm25, inplace=True)
```

In [6]:
```
data.drop(['to'], axis=1, inplace=True)
```

In [7]:
```
data.columns
```

Out[7]:
```
Index(['From', 'pm10', 'pm2.5', 'NO', 'NO2', 'NOX', 'CO', 'SO2', 'NH3',
       'Ozone', 'Benzene'],
      dtype='object')
```

In [8]:
```
data['From'] = pd.to_datetime(data['From'], format = '%d-%m-%Y %H:%M')
```

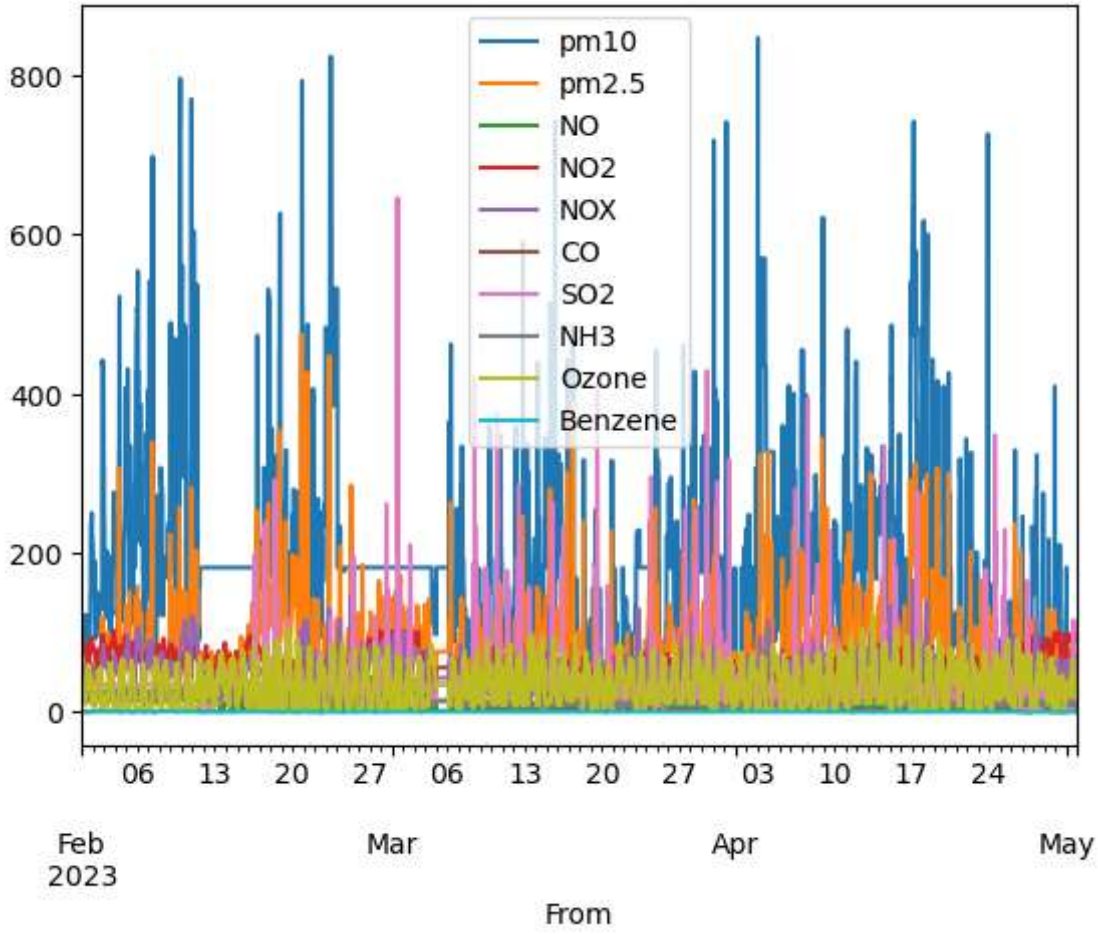
In [9]:
```
data['From']
```

Out[9]:
```
Sn
1      2023-02-01 00:00:00
2      2023-02-01 00:15:00
3      2023-02-01 00:30:00
4      2023-02-01 00:45:00
5      2023-02-01 01:00:00
               ...
8636   2023-05-01 22:45:00
8637   2023-05-01 23:00:00
8638   2023-05-01 23:15:00
8639   2023-05-01 23:30:00
8640   2023-05-01 23:45:00
Name: From, Length: 8640, dtype: datetime64[ns]
```

In [10]:
```
data.set_index(['From'], inplace=True)
```

In [11]:
```
data.plot()
```

Out[11]:
```
<Axes: xlabel='From'>
```



When replacing NA values with 0, you assign a fixed value of 0 to all the missing values in the dataset. This approach assumes that the missing values have no influence on the analysis and treats them as zero values. While it is a simple and quick method, it may not accurately represent the underlying data distribution and can introduce bias in the analysis.

On the other hand, replacing NA values with the mean takes into account the average value of the available data. It considers the distribution of the existing values and provides a more realistic estimate for the missing values. This approach helps to preserve the overall characteristics and patterns of the data, which can be important for accurate analysis and modeling.

In summary, replacing NA values with the mean tends to be a more accurate approach compared to replacing with 0, as it considers the data distribution and minimizes potential bias in the analysis.

In [ ]:

In [1]:
```
import pandas as pd
import pandas.plotting
import matplotlib.pyplot as plt
import numpy as np
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import adfuller
%matplotlib inline
```

In [2]:
```
file = r'C:\Users\Sidharth\Desktop\798Q\.ipynb_checkpoints\data.csv.csv'
data=pd.read_csv(file,index_col=0)
```

In [3]:
```
data=data[:-3]
data
```

Out[3]:

Sn	From	to	pm10	pm2.5	NO	NO2	NOX	CO	SO2	NH3	Ozone	Benzene
1	01-02-2023 00:00	01-02-2023 00:15	95.0	35.0	NaN	90.1	56.2	0.31	NaN	17.7	28.1	0.4
2	01-02-2023 00:15	01-02-2023 00:30	95.0	35.0	NaN	88.0	55.1	0.33	NaN	18.3	27.1	0.4
3	01-02-2023 00:30	01-02-2023 00:45	95.0	35.0	NaN	87.7	55.2	0.38	NaN	19.7	24.9	0.4
4	01-02-2023 00:45	01-02-2023 01:00	122.0	34.0	NaN	88.9	55.7	0.38	NaN	21.3	21.9	0.4
5	01-02-2023 01:00	01-02-2023 01:15	122.0	34.0	NaN	90.0	55.8	0.38	NaN	22.3	16.7	0.4
...	...	...	...	...	...	...	...	...	...	...	...	...
8636	01-05-2023 22:45	01-05-2023 23:00	19.0	11.0	17.9	100.0	67.8	0.63	10.0	10.7	26.1	0.1
8637	01-05-2023 23:00	01-05-2023 23:15	19.0	11.0	17.9	100.0	67.7	0.57	10.0	10.4	30.9	0.1
8638	01-05-2023 23:15	01-05-2023 23:30	19.0	11.0	19.6	100.2	69.2	0.58	9.9	10.5	29.6	0.1
8639	01-05-2023 23:30	01-05-2023 23:45	19.0	11.0	20.8	100.2	70.2	0.58	9.5	10.8	30.0	0.1
8640	01-05-2023 23:45	02-05-2023 00:00	32.0	6.0	21.8	98.8	70.3	NaN	NaN	11.0	33.5	0.1

8640 rows × 12 columns

In [4]:
```
NO='NO'
avg_NO = data[NO].mean()
```

```
data[NO].fillna(avg_NO, inplace=True)
NO2='NO2'
avg_NO2 = data[NO2].mean()
data[NO2].fillna(avg_NO2, inplace=True)
NOX='NOX'
avg_NOX = data[NOX].mean()
data[NOX].fillna(avg_NOX, inplace=True)
CO='CO'
avg_CO = data[CO].mean()
data[CO].fillna(avg_CO, inplace=True)
SO2='SO2'
avg_SO2 = data[SO2].mean()
data[SO2].fillna(avg_SO2, inplace=True)
NH3='NH3'
avg_NH3 = data[NH3].mean()
data[NH3].fillna(avg_NH3, inplace=True)
Ozone='Ozone'
avg_Ozone = data[Ozone].mean()
data[Ozone].fillna(avg_Ozone, inplace=True)
Benzene='Benzene'
avg_Benzene = data[Benzene].mean()
data[Benzene].fillna(avg_Benzene, inplace=True)
pm10='pm10'
avg_pm10 = data[pm10].mean()
data[pm10].fillna(avg_pm10, inplace=True)
pm25='pm2.5'
avg_pm25 = data[pm25].mean()
data[pm25].fillna(avg_pm25, inplace=True)
```

In [5]:
```
data.drop(['to'], axis=1, inplace=True)
```

In [6]:
```
data.columns
```

Out[6]:
```
Index(['From', 'pm10', 'pm2.5', 'NO', 'NO2', 'NOX', 'CO', 'SO2', 'NH3',
       'Ozone', 'Benzene'],
      dtype='object')
```

In [7]:
```
data['From'] = pd.to_datetime(data['From'], format = '%d-%m-%Y %H:%M')
```

In [8]:
```
data['From']
```

Out[8]:
```
Sn
1       2023-02-01 00:00:00
2       2023-02-01 00:15:00
3       2023-02-01 00:30:00
4       2023-02-01 00:45:00
5       2023-02-01 01:00:00
                  ...
8636    2023-05-01 22:45:00
8637    2023-05-01 23:00:00
8638    2023-05-01 23:15:00
8639    2023-05-01 23:30:00
8640    2023-05-01 23:45:00
Name: From, Length: 8640, dtype: datetime64[ns]
```

In [9]:
```
data.set_index(['From'], inplace=True)
```

When dealing with a dataset containing missing sensory data, it is important to establish an appropriate approach for modeling the data and making forecasts. In the case of a dataset with a sufficient number of entries, such as the one with around 8000 entries mentioned here, it is feasible to establish ARMA/ARIMA processes.

To address missing data in a per-column approach, each column should be treated independently. This is because each column represents a specific sensor or measurement, and the missing data in each column may have its own distinct pattern or behavior. By analyzing and modeling each column separately, you can capture the individual characteristics and dynamics of the sensors, leading to more accurate analysis and forecasting results.

ARMA (Autoregressive Moving Average) and ARIMA (Autoregressive Integrated Moving Average) models are commonly used for time series analysis and forecasting. These models take into account the autoregressive and moving average components of the data to capture patterns and predict future values.

Before applying ARMA/ARIMA modeling, it is necessary to handle the missing data appropriately. Various techniques can be employed, such as interpolation or replacing missing values with mean or zero. These techniques estimate the missing values in each column, ensuring that the data is complete for analysis.

Once the missing values are filled, the ARMA/ARIMA modeling can be applied independently to each column. This allows for the specific patterns and dynamics of each sensor or measurement to be captured. By modeling each column separately, you can analyze the variations and dependencies within each sensor's data, taking into account their unique characteristics.

In summary, when dealing with missing sensory data, it is advisable to establish ARMA/ARIMA processes on a per-column basis. This approach acknowledges the distinct characteristics of each sensor or measurement, leading to more accurate analysis and forecasting outcomes.

In [10]:
```
NO_C = data['NO']
NO2_C = data['NO2']
NOX_C = data['NOX']
CO_C = data['CO']
SO2_C = data['SO2']
NH3_C = data['NH3']
Ozone_C= data['Ozone']
Benzene_c = data['Benzene']
pm10_C = data['pm10']
pm25_C = data['pm2.5']
```

In [11]:
```
data
```

Out[11]:

From	pm10	pm2.5	NO	NO2	NOX	CO	SO2	NH3	Ozone	Benzene
2023-02-01 00:00:00	95.0	35.0	14.649636	90.1	56.2	0.310000	34.232731	17.7	28.1	0.4
2023-02-01 00:15:00	95.0	35.0	14.649636	88.0	55.1	0.330000	34.232731	18.3	27.1	0.4
2023-02-01 00:30:00	95.0	35.0	14.649636	87.7	55.2	0.380000	34.232731	19.7	24.9	0.4
2023-02-01 00:45:00	122.0	34.0	14.649636	88.9	55.7	0.380000	34.232731	21.3	21.9	0.4
2023-02-01 01:00:00	122.0	34.0	14.649636	90.0	55.8	0.380000	34.232731	22.3	16.7	0.4
...	...	...	...	...	...	...	...	...	...	...
2023-05-01 22:45:00	19.0	11.0	17.900000	100.0	67.8	0.630000	10.000000	10.7	26.1	0.1
2023-05-01 23:00:00	19.0	11.0	17.900000	100.0	67.7	0.570000	10.000000	10.4	30.9	0.1
2023-05-01 23:15:00	19.0	11.0	19.600000	100.2	69.2	0.580000	9.900000	10.5	29.6	0.1
2023-05-01 23:30:00	19.0	11.0	20.800000	100.2	70.2	0.580000	9.500000	10.8	30.0	0.1
2023-05-01 23:45:00	32.0	6.0	21.800000	98.8	70.3	1.408538	34.232731	11.0	33.5	0.1

8640 rows × 10 columns

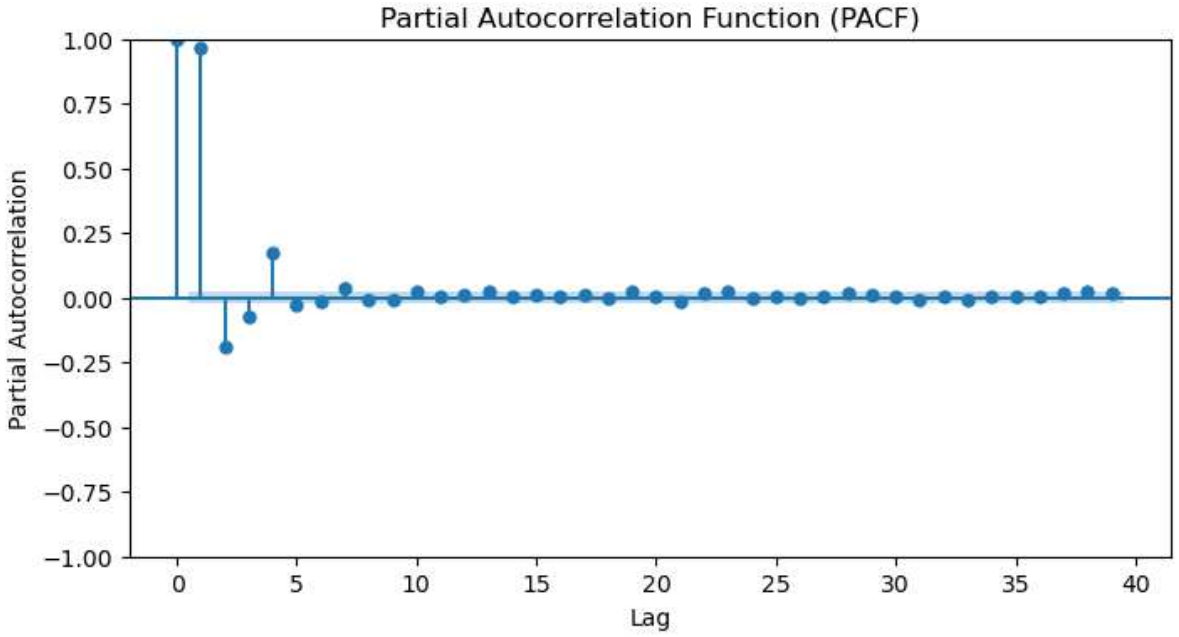
taking training data 7540 and testing data 1100 for better arima model

In [12]:
```
training = data.index[7540]
msk= (data.index <= training)
data_train = data[msk].copy()
data_test = data[~msk].copy()
```

In [13]:
```
fig, ax = plt.subplots(figsize=(8, 4))
plot_pacf(data_train['NO2'], ax=ax)
ax.set_xlabel('Lag')
ax.set_ylabel('Partial Autocorrelation')
ax.set_title('Partial Autocorrelation Function (PACF)')
```

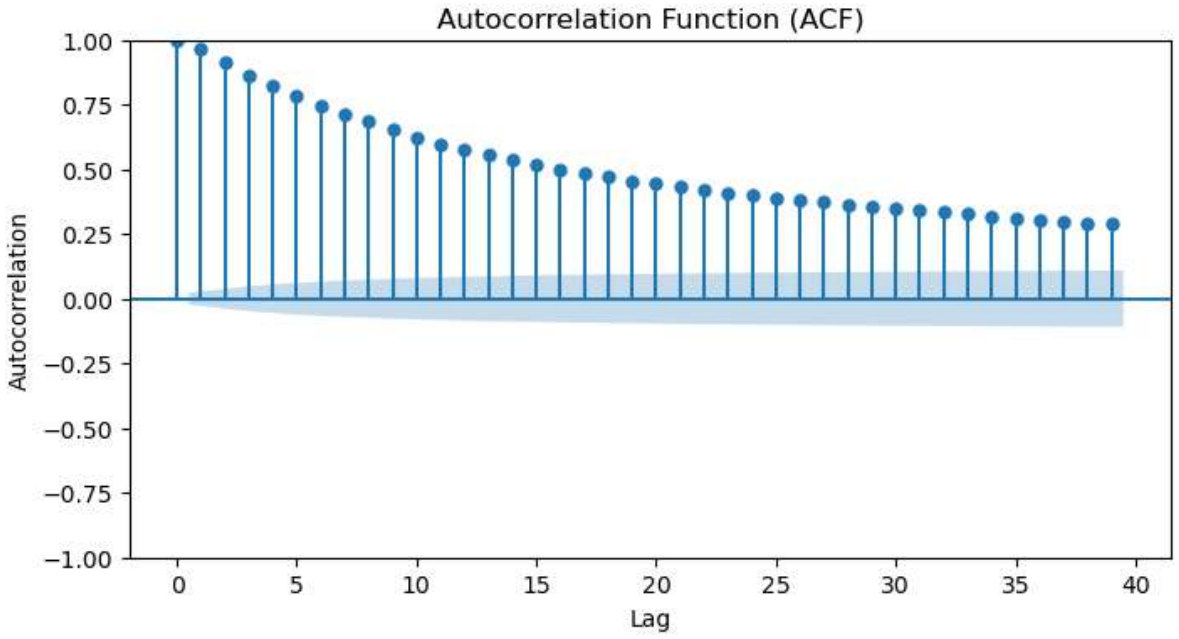
```
C:\ProgramData\anaconda3\lib\site-packages\statsmodels\graphics\tsaplots.py:348: F
utureWarning: The default method 'yw' can produce PACF values outside of the [-1,
1] interval. After 0.13, the default will change tounadjusted Yule-Walker ('ywm').
You can use this method now by setting method='ywm'.
  warnings.warn(
```

Out[13]: Text(0.5, 1.0, 'Partial Autocorrelation Function (PACF)')



In [14]:
```
fig, ax = plt.subplots(figsize=(8, 4))
plot_acf(data_train['NO2'], ax=ax)
ax.set_xlabel('Lag')
ax.set_ylabel('Autocorrelation')
ax.set_title('Autocorrelation Function (ACF)')
```

Out[14]: Text(0.5, 1.0, 'Autocorrelation Function (ACF)')



In [15]:
```
plt.show()
```

To implement the ARMA/ARIMA model in Jupyter Notebook, you can follow these steps:

ACF and PACF Analysis: Start by plotting the AutoCorrelation Function (ACF) and Partial AutoCorrelation Function (PACF) plots for the time series data. You can use the plot_acf and

plot_pacf functions from the statsmodels.graphics.tsaplots module to generate these plots. The ACF plot shows the correlation at different lags, while the PACF plot shows the correlation after removing the correlation already explained by shorter lags. Analyzing these plots can help determine the appropriate values for the AR and MA components of the ARIMA model.

Stationarity Check: Use statistical tests like the Augmented Dickey-Fuller (ADF) test or the Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test to check the stationarity of the time series data. Stationarity is an important assumption for ARMA/ARIMA modeling. If the data is found to be non-stationary, you can apply differencing (integration) to make it stationary. The adfuller function from the statsmodels.tsa.stattools module can be used to perform the ADF test.

Model Fitting: Once you have determined the appropriate values for the AR, MA, and differencing components (p, q, d), you can fit the ARIMA model to the data. Use the ARIMA class from the statsmodels.tsa.arima.model module to create an ARIMA model object. Fit the model to the data using the fit method, specifying the values of p, d, and q.

Forecasting: After fitting the ARIMA model, you can generate forecasts for future time points. Use the get_forecast method to generate the forecasts, specifying the number of steps (future time points) you want to forecast. The resulting forecasted values can be accessed using the predicted_mean attribute of the forecast object.

By following these steps in Jupyter Notebook, you can successfully implement the ARMA/ARIMA model for analyzing and forecasting the given time series data.

In [16]:
```
NO2_CC=NO2_C[1:7540]
NO2_CC
# Perform ADF test
result = adfuller(NO2_CC)

# Extract test statistics and p-value
adf_statistic = result[0]
p_value = result[1]

# Print the test statistics and p-value
print("ADF Statistic:", adf_statistic)
print("p-value:", p_value)
```

```
ADF Statistic: -12.172018633972712
p-value: 1.4136821762874538e-22
```

In [17]:
```
from pmdarima import auto_arima
```

In [18]:
```
from pmdarima.arima import auto_arima
def arimamodel(timeseriesarray) :
 autoarima_model = auto_arima(timeseriesarray, start_p=0, start_q=0, d=0, max_p=5,
 return autoarima_model
model = arimamodel(data_train['NO2'])
model.summary()
```

Out[18]:

SARIMAX Results

Dep. Variable:	y	**No. Observations:**	7541
Model:	SARIMAX(5, 0, 0)	**Log Likelihood**	-22609.413
Date:	Mon, 26 Jun 2023	**AIC**	45232.825
Time:	17:12:58	**BIC**	45281.322
Sample:	02-01-2023	**HQIC**	45249.474
	- 04-20-2023		
Covariance Type:	opg		

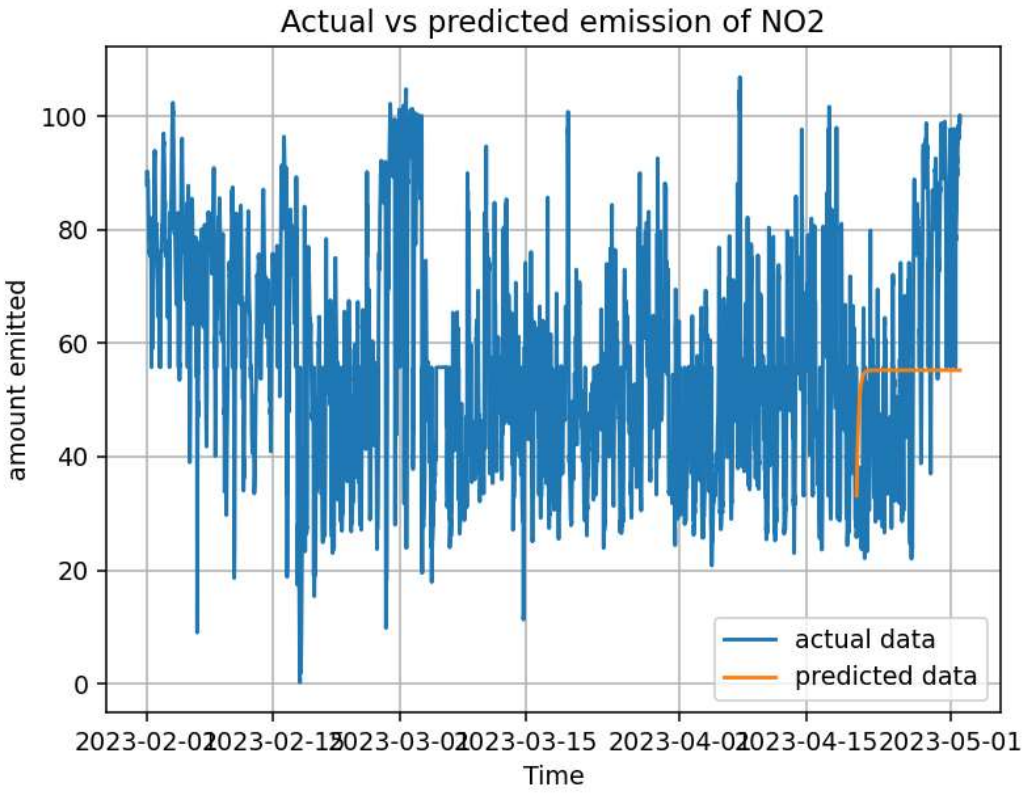
	coef	std err	z	P>|z|	[0.025	0.975]
intercept	2.1671	0.201	10.789	0.000	1.773	2.561
ar.L1	1.1494	0.006	180.277	0.000	1.137	1.162
ar.L2	-0.0951	0.012	-7.987	0.000	-0.118	-0.072
ar.L3	-0.2720	0.011	-23.741	0.000	-0.294	-0.250
ar.L4	0.2082	0.007	27.953	0.000	0.194	0.223
ar.L5	-0.0298	0.006	-4.770	0.000	-0.042	-0.018
sigma2	23.5262	0.148	158.570	0.000	23.235	23.817

Ljung-Box (L1) (Q):	0.00	**Jarque-Bera (JB):**	268178.01
Prob(Q):	0.95	**Prob(JB):**	0.00
Heteroskedasticity (H):	1.20	**Skew:**	-0.78
Prob(H) (two-sided):	0.00	**Kurtosis:**	32.17

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

In [19]:
```
auto = model.predict(n_periods = len(data_test['NO2']))
data['NO2 forecast'] = [None]*len(data_train['NO2']) + list(auto)
```

In [20]:
```
%matplotlib inline

%matplotlib notebook
plt.plot(NO2_C,label='actual data')
plt.plot(data['NO2 forecast'], label='predicted data')
plt.xlabel('Time')
plt.ylabel('amount emitted')
plt.title('Actual vs predicted emission of NO2')
plt.grid()
plt.legend()
plt.show()
```



In [21]:
```
ape = np.abs((data_test['NO2'] - data['NO2 forecast']) / data_test['NO2'])
mape = np.mean(ape) * 100

print("Mean Absolute Percentage Error:", mape)
```

```
Mean Absolute Percentage Error: 0.6354621862
```

The ARIMA model is a useful tool for predicting future trends, providing a rough idea of the expected pattern. While it may not always be accurate, it can still offer valuable insights.

When analyzing the ARIMA model's performance, one common measure is the forecast error. In your case, the ARIMA model applied to the testing data of 1100 observations has an error of 0.63. This error represents the average difference between the forecasted values and the actual values in the testing data.

A forecast error of 0.63 indicates that, on average, the forecasted values deviate from the actual values by 0.63 units. It's important to interpret this error in the context of your specific dataset and its scale. If the magnitude of the variable being forecasted is small, a forecast error of 0.63 might be considered acceptable. However, if the variable has a larger scale, such as thousands or millions, a forecast error of 0.63 might be relatively large.

It's worth noting that the forecast error should be interpreted alongside other evaluation metrics, such as Mean Absolute Error (MAE), Root Mean Square Error (RMSE), or percentage errors like Mean Absolute Percentage Error (MAPE). These metrics can provide a more comprehensive assessment of the model's accuracy and help in understanding the magnitude of forecast errors.

Overall, while the ARIMA model can give a rough idea of the following trend, it's important to consider the forecast error and other evaluation metrics to assess its performance accurately.

In [1]:
```
import pandas as pd
import pandas.plotting
import matplotlib.pyplot as plt
import numpy as np
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import adfuller
%matplotlib inline
```

using linear interpolation for forecasting

In [2]:
```
file = r'C:\Users\Sidharth\Desktop\798Q\.ipynb_checkpoints\data.csv.csv'
df=pd.read_csv(file,index_col=0)
```

In [3]:
```
data=df.copy()
```

In [4]:
```
data=data[:-3]
data
```

Out[4]:

Sn	From	to	pm10	pm2.5	NO	NO2	NOX	CO	SO2	NH3	Ozone	Benzene
1	01-02-2023 00:00	01-02-2023 00:15	95.0	35.0	NaN	90.1	56.2	0.31	NaN	17.7	28.1	0.4
2	01-02-2023 00:15	01-02-2023 00:30	95.0	35.0	NaN	88.0	55.1	0.33	NaN	18.3	27.1	0.4
3	01-02-2023 00:30	01-02-2023 00:45	95.0	35.0	NaN	87.7	55.2	0.38	NaN	19.7	24.9	0.4
4	01-02-2023 00:45	01-02-2023 01:00	122.0	34.0	NaN	88.9	55.7	0.38	NaN	21.3	21.9	0.4
5	01-02-2023 01:00	01-02-2023 01:15	122.0	34.0	NaN	90.0	55.8	0.38	NaN	22.3	16.7	0.4
...	...	...	...	...	...	...	...	...	...	...	...	...
8636	01-05-2023 22:45	01-05-2023 23:00	19.0	11.0	17.9	100.0	67.8	0.63	10.0	10.7	26.1	0.1
8637	01-05-2023 23:00	01-05-2023 23:15	19.0	11.0	17.9	100.0	67.7	0.57	10.0	10.4	30.9	0.1
8638	01-05-2023 23:15	01-05-2023 23:30	19.0	11.0	19.6	100.2	69.2	0.58	9.9	10.5	29.6	0.1
8639	01-05-2023 23:30	01-05-2023 23:45	19.0	11.0	20.8	100.2	70.2	0.58	9.5	10.8	30.0	0.1
8640	01-05-2023 23:45	02-05-2023 00:00	32.0	6.0	21.8	98.8	70.3	NaN	NaN	11.0	33.5	0.1

8640 rows × 12 columns

In [5]:
```
blasting_time = data['From']
```

In [6]:
```
 data.drop(['to'], axis=1, inplace=True)
```

In [7]:
```
data.columns
```

Out[7]:
```
Index(['From', 'pm10', 'pm2.5', 'NO', 'NO2', 'NOX', 'CO', 'SO2', 'NH3',
       'Ozone', 'Benzene'],
      dtype='object')
```

In [ ]:
```

```

In [8]:
```
data['From'] = pd.to_datetime(data['From'], format = '%d-%m-%Y %H:%M')
```

In [9]:
```
data['From']
```

Out[9]:
```
Sn
1       2023-02-01 00:00:00
2       2023-02-01 00:15:00
3       2023-02-01 00:30:00
4       2023-02-01 00:45:00
5       2023-02-01 01:00:00
                ...
8636    2023-05-01 22:45:00
8637    2023-05-01 23:00:00
8638    2023-05-01 23:15:00
8639    2023-05-01 23:30:00
8640    2023-05-01 23:45:00
Name: From, Length: 8640, dtype: datetime64[ns]
```

In [10]:
```
data.set_index(['From'], inplace=True)
```

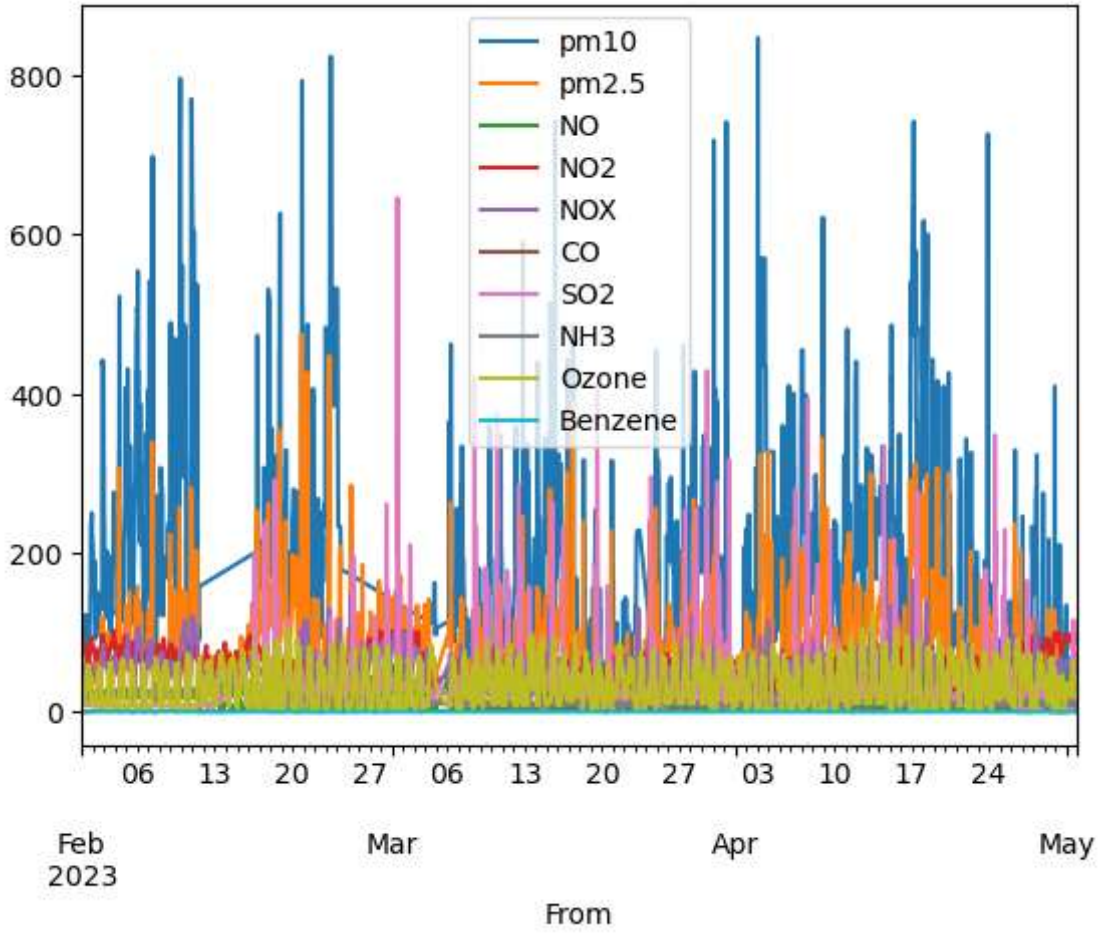
In [11]:
```
data_interp_linear=data.copy()
data_interp_cubic=data.copy()
data_interp_spline=data.copy()
```

In [12]:
```
# Perform linear interpolation on the DataFrame
data_interp_linear = data_interp_linear.interpolate(limit_direction='both',method=

# Print the interpolated DataFrame
print(data_interp_linear)
```

```
                     pm10  pm2.5    NO    NO2   NOX    CO   SO2   NH3  Ozone  \
From
2023-02-01 00:00:00   95.0   35.0  18.1   90.1  56.2  0.31   8.2  17.7   28.1
2023-02-01 00:15:00   95.0   35.0  18.1   88.0  55.1  0.33   8.2  18.3   27.1
2023-02-01 00:30:00   95.0   35.0  18.1   87.7  55.2  0.38   8.2  19.7   24.9
2023-02-01 00:45:00  122.0   34.0  18.1   88.9  55.7  0.38   8.2  21.3   21.9
2023-02-01 01:00:00  122.0   34.0  18.1   90.0  55.8  0.38   8.2  22.3   16.7
...                    ...    ...   ...    ...   ...   ...   ...   ...    ...
2023-05-01 22:45:00   19.0   11.0  17.9  100.0  67.8  0.63  10.0  10.7   26.1
2023-05-01 23:00:00   19.0   11.0  17.9  100.0  67.7  0.57  10.0  10.4   30.9
2023-05-01 23:15:00   19.0   11.0  19.6  100.2  69.2  0.58   9.9  10.5   29.6
2023-05-01 23:30:00   19.0   11.0  20.8  100.2  70.2  0.58   9.5  10.8   30.0
2023-05-01 23:45:00   32.0    6.0  21.8   98.8  70.3  0.58   9.5  11.0   33.5

                     Benzene
From
2023-02-01 00:00:00      0.4
2023-02-01 00:15:00      0.4
2023-02-01 00:30:00      0.4
2023-02-01 00:45:00      0.4
2023-02-01 01:00:00      0.4
...                      ...
2023-05-01 22:45:00      0.1
2023-05-01 23:00:00      0.1
2023-05-01 23:15:00      0.1
2023-05-01 23:30:00      0.1
2023-05-01 23:45:00      0.1

[8640 rows x 10 columns]
```

In [13]:
```
data_interp_linear.plot()
```

Out[13]:
```
<Axes: xlabel='From'>
```



When comparing the methods of replacing NA values with 0 and using linear interpolation, there are some key differences to consider.

Replacing NA values with 0 assigns a fixed value of 0 to all the missing values. This approach assumes that the missing values have no specific pattern or trend and treats them as equal to zero. It is a simple and straightforward method, but it may not accurately capture the underlying trends or patterns in the data. This can lead to inaccurate analysis or modeling results, especially if the missing values are not randomly distributed.

On the other hand, linear interpolation estimates the missing values based on the surrounding data points. It assumes a linear relationship between the available data points and fills in the missing values accordingly. Linear interpolation takes into account the trend or pattern in the data and provides a smoother estimation compared to simply replacing with 0. It can be a more accurate method, especially when the missing values follow a linear trend.

In summary, linear interpolation tends to provide a more accurate estimation of missing values compared to replacing with 0. It takes into account the trend or pattern in the data and provides a smoother estimate. However, the choice between the two methods ultimately depends on the nature of the data and the specific analysis or modeling objectives.

In [14]:
```
data_interp_linear=data_interp_linear.resample('D').mean()

NO_C = data_interp_linear['NO']
NO2_C = data_interp_linear['NO2']
NOX_C = data_interp_linear['NOX']
```

```
CO_C = data_interp_linear['CO']
SO2_C = data_interp_linear['SO2']
NH3_C = data_interp_linear['NH3']
Ozone_C= data_interp_linear['Ozone']
Benzene_c = data_interp_linear['Benzene']
pm10_C = data_interp_linear['pm10']
pm25_C = data_interp_linear['pm2.5']
```

In [15]:
```
data_interp_linear=data_interp_linear.resample('D').mean()
data_interp_linear
```

Out[15]:

From	pm10	pm2.5	NO	NO2	NOX	CO	SO2	NH3	
2023-02-01	114.739583	35.145833	18.100000	78.850000	48.514583	0.453490	8.200000	22.130208	32
2023-02-02	177.458333	52.020833	18.100000	79.049479	53.559896	1.312344	8.200000	22.416667	25
2023-02-03	171.270833	52.916667	18.100000	82.247396	56.590104	1.219896	8.200000	23.208333	27
2023-02-04	214.239583	70.703125	18.100000	76.579687	53.282812	1.135365	8.200000	25.262500	27
2023-02-05	262.171875	81.437500	18.100000	73.644271	63.118750	0.833542	8.200000	26.077083	24
...	...	...	...	...	...	...	...	...	
2023-04-27	65.645833	29.031250	7.333333	72.735937	44.648438	0.397083	25.185937	10.814583	48
2023-04-28	123.281250	48.572917	22.408854	78.778125	60.115625	1.330521	21.163542	11.039583	40
2023-04-29	72.458333	34.479167	6.615104	81.954167	48.966146	0.712917	23.288542	10.872917	41
2023-04-30	57.135417	16.281250	6.853125	88.653125	52.733854	0.349792	7.272917	10.100521	32
2023-05-01	44.020833	11.375000	11.435417	92.310417	58.400521	0.710104	11.853125	9.797917	35

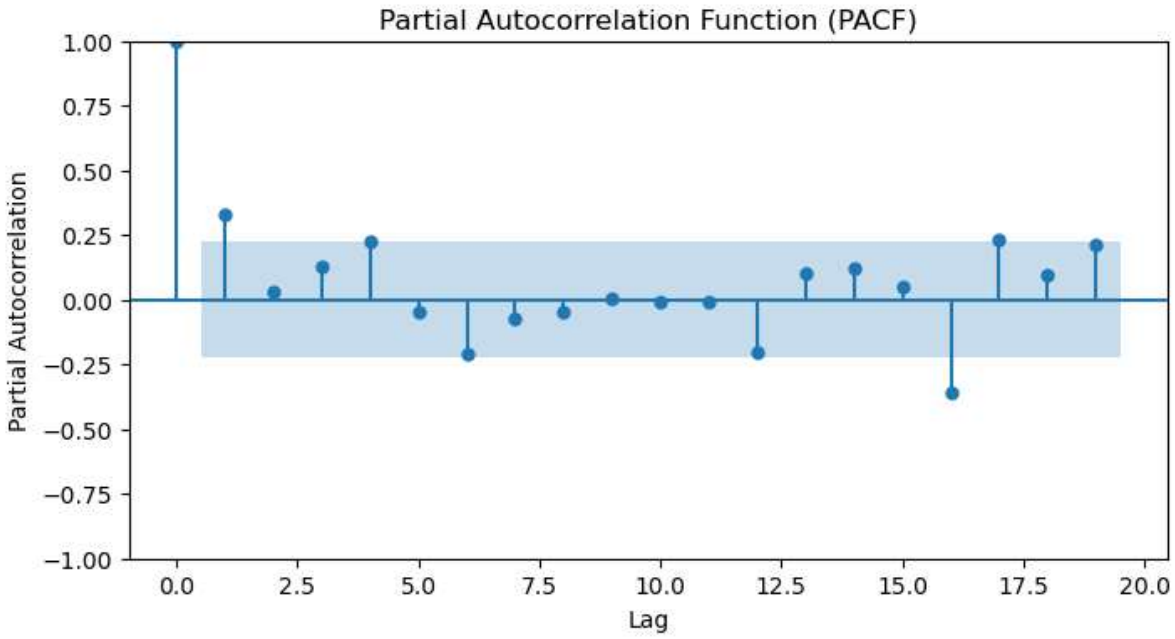
90 rows × 10 columns

In [16]:
```
training = data_interp_linear.index[75]
msk= (data_interp_linear.index <= training)
data_train = data_interp_linear[msk].copy()
data_test = data_interp_linear[~msk].copy()
```

In [17]:
```
fig, ax = plt.subplots(figsize=(8, 4))
plot_pacf(data_train['Ozone'], ax=ax)
ax.set_xlabel('Lag')
ax.set_ylabel('Partial Autocorrelation')
ax.set_title('Partial Autocorrelation Function (PACF)')
```

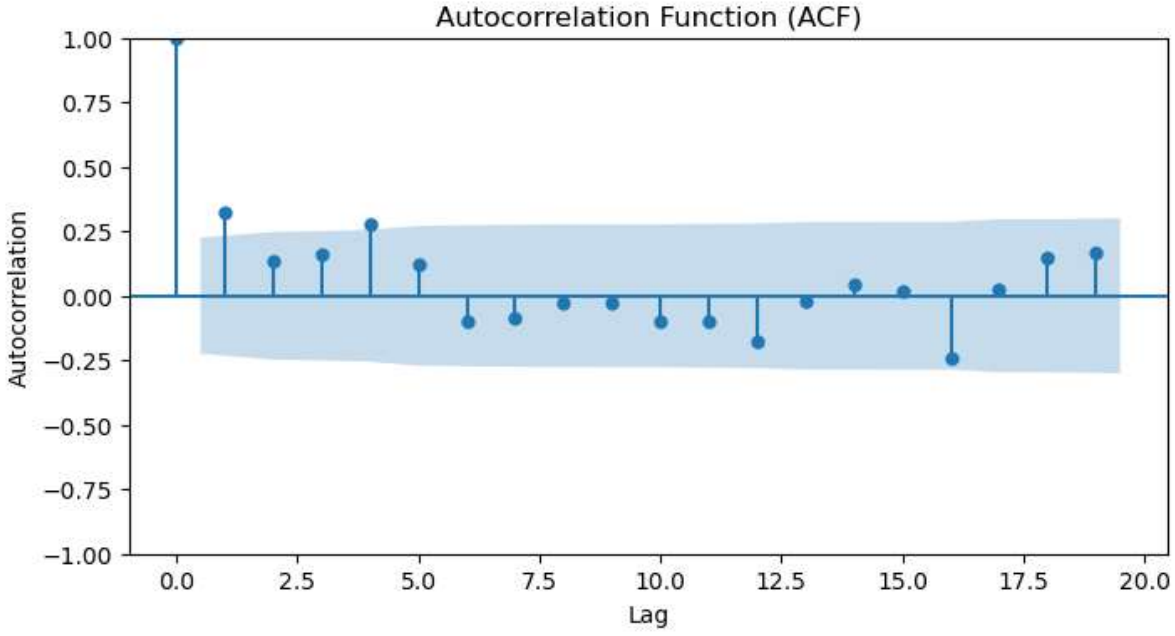
```
C:\ProgramData\anaconda3\lib\site-packages\statsmodels\graphics\tsaplots.py:348: F
utureWarning: The default method 'yw' can produce PACF values outside of the [-1,
1] interval. After 0.13, the default will change tounadjusted Yule-Walker ('ywm').
You can use this method now by setting method='ywm'.
  warnings.warn(
```

Out[17]:
```
Text(0.5, 1.0, 'Partial Autocorrelation Function (PACF)')
```



In [18]:
```
fig, ax = plt.subplots(figsize=(8, 4))
plot_acf(data_train['Ozone'], ax=ax)
ax.set_xlabel('Lag')
ax.set_ylabel('Autocorrelation')
ax.set_title('Autocorrelation Function (ACF)')
```

Out[18]:
```
Text(0.5, 1.0, 'Autocorrelation Function (ACF)')
```



In [19]:
```
plt.show()
```

In [20]:
```
#Ozone_CC=Ozone_C[1:7540]

# Perform ADF test
result = adfuller(data_train['Ozone'])
```

```
# Extract test statistics and p-value
adf_statistic = result[0]
p_value = result[1]

# Print the test statistics and p-value
print("ADF Statistic:", adf_statistic)
print("p-value:", p_value)
```

```
ADF Statistic: -6.07024657774616
p-value: 1.1554855941496654e-07
```

In [21]:
```
from pmdarima import auto_arima
```

In [22]:
```
from pmdarima.arima import auto_arima
def arimamodel(timeseries) :
 autoarima_model = auto_arima(timeseries, start_p=0, start_q=0, d=0, max_p=5, max_
 return autoarima_model
model = arimamodel(data_train['Ozone'])
model.summary()
```

Out[22]:

SARIMAX Results

Dep. Variable:	y	**No. Observations:**	76
Model:	SARIMAX(1, 0, 0)	**Log Likelihood**	-257.522
Date:	Wed, 28 Jun 2023	**AIC**	521.043
Time:	23:29:55	**BIC**	528.036
Sample:	02-01-2023	**HQIC**	523.838
	- 04-17-2023		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
intercept	22.9378	4.038	5.681	0.000	15.024	30.852
ar.L1	0.3258	0.116	2.818	0.005	0.099	0.552
sigma2	51.2887	9.055	5.664	0.000	33.541	69.036

Ljung-Box (L1) (Q):	0.00	**Jarque-Bera (JB):**	1.42
Prob(Q):	0.95	**Prob(JB):**	0.49
Heteroskedasticity (H):	1.11	**Skew:**	0.30
Prob(H) (two-sided):	0.79	**Kurtosis:**	3.31

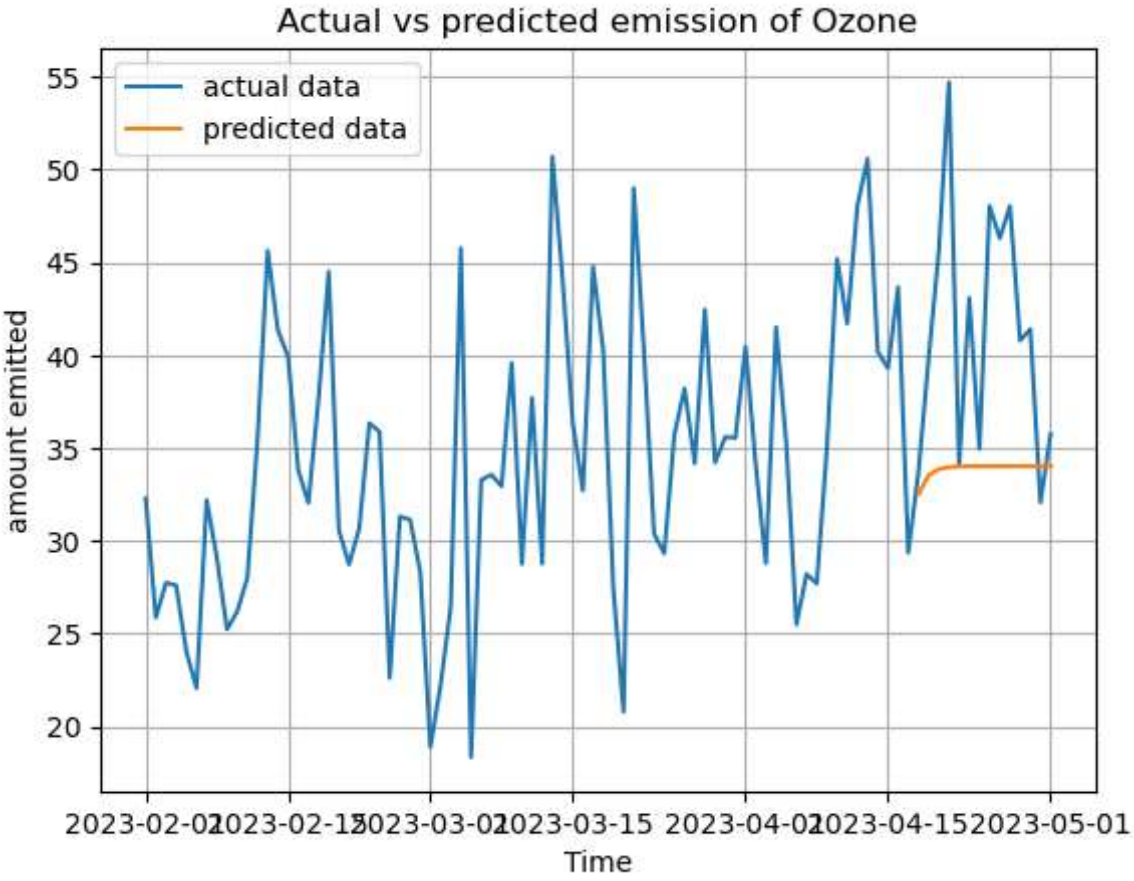
Warnings:

[1] Covariance matrix calculated using the outer product of gradients (complex-step).

In [23]:
```
auto = model.predict(n_periods = len(data_test['Ozone']))
data_interp_linear['Ozone forecast'] = [None]*len(data_train['Ozone']) + list(auto
```

In [24]:
```
 %matplotlib inline

#matplotlib notebook
plt.plot(data_interp_linear['Ozone'],label='actual data')
```

```
plt.plot(data_interp_linear['Ozone forecast'], label='predicted data')
plt.xlabel('Time')
plt.ylabel('amount emitted')
plt.title('Actual vs predicted emission of Ozone')
plt.grid()
plt.legend()
plt.show()
```



In [25]:
```
ape = np.abs((data_test['Ozone'] - data_interp_linear['Ozone forecast']) / data_te
mape = np.mean(ape) * 100

print("Mean Absolute Percentage Error:", mape)
```

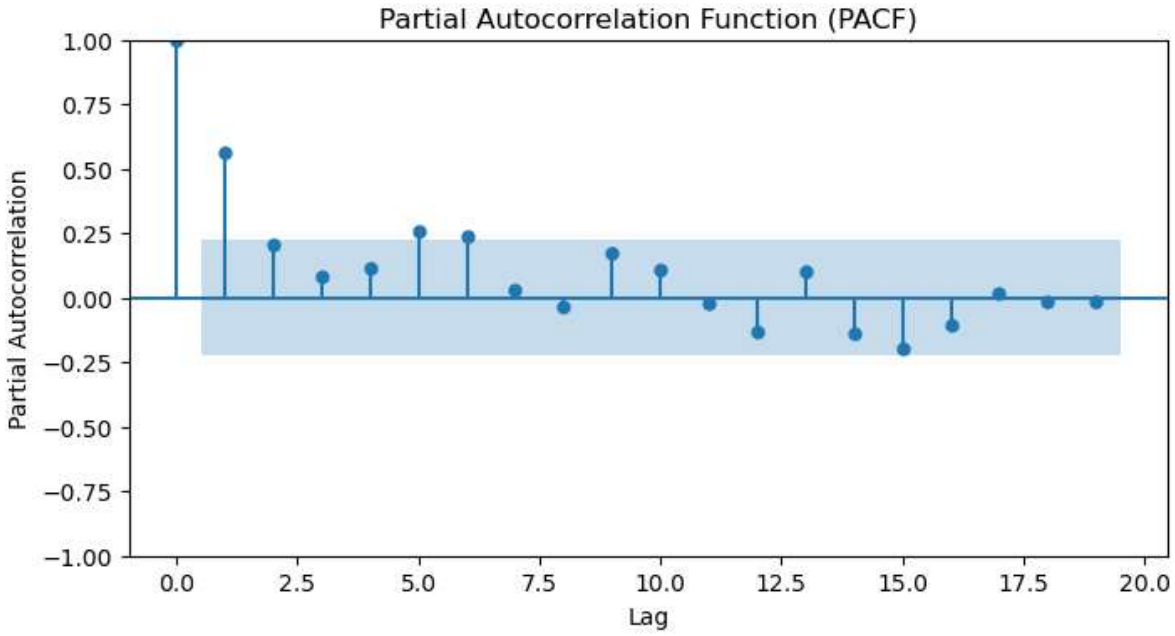
```
Mean Absolute Percentage Error: 16.9347932207782
```

In [ ]:
```

```

In [26]:
```
fig, ax = plt.subplots(figsize=(8, 4))
plot_pacf(data_train['CO'], ax=ax)
ax.set_xlabel('Lag')
ax.set_ylabel('Partial Autocorrelation')
ax.set_title('Partial Autocorrelation Function (PACF)')
```

```
C:\ProgramData\anaconda3\lib\site-packages\statsmodels\graphics\tsaplots.py:348: F
utureWarning: The default method 'yw' can produce PACF values outside of the [-1,
1] interval. After 0.13, the default will change tounadjusted Yule-Walker ('ywm').
You can use this method now by setting method='ywm'.
  warnings.warn(
```

Out[26]:
```
Text(0.5, 1.0, 'Partial Autocorrelation Function (PACF)')
```



In [27]:
```
# Perform ADF test
result = adfuller(data_train['CO'])

# Extract test statistics and p-value
adf_statistic = result[0]
p_value = result[1]

# Print the test statistics and p-value
print("ADF Statistic:", adf_statistic)
print("p-value:", p_value)
```

```
ADF Statistic: -2.858367800614889
p-value: 0.050407868184930885
```

In [28]:
```
from pmdarima.arima import auto_arima
def arimamodel(timeseries) :
 autoarima_model = auto_arima(timeseries, start_p=0, start_q=0, d=0, max_p=5, max_
 return autoarima_model
model = arimamodel(data_train['CO'])
model.summary()
```

Out[28]:

SARIMAX Results

Dep. Variable:	y	**No. Observations:**	76
Model:	SARIMAX(2, 0, 1)	**Log Likelihood**	-26.149
Date:	Wed, 28 Jun 2023	**AIC**	62.299
Time:	23:29:57	**BIC**	73.952
Sample:	02-01-2023	**HQIC**	66.956
	- 04-17-2023		
Covariance Type:	opg		

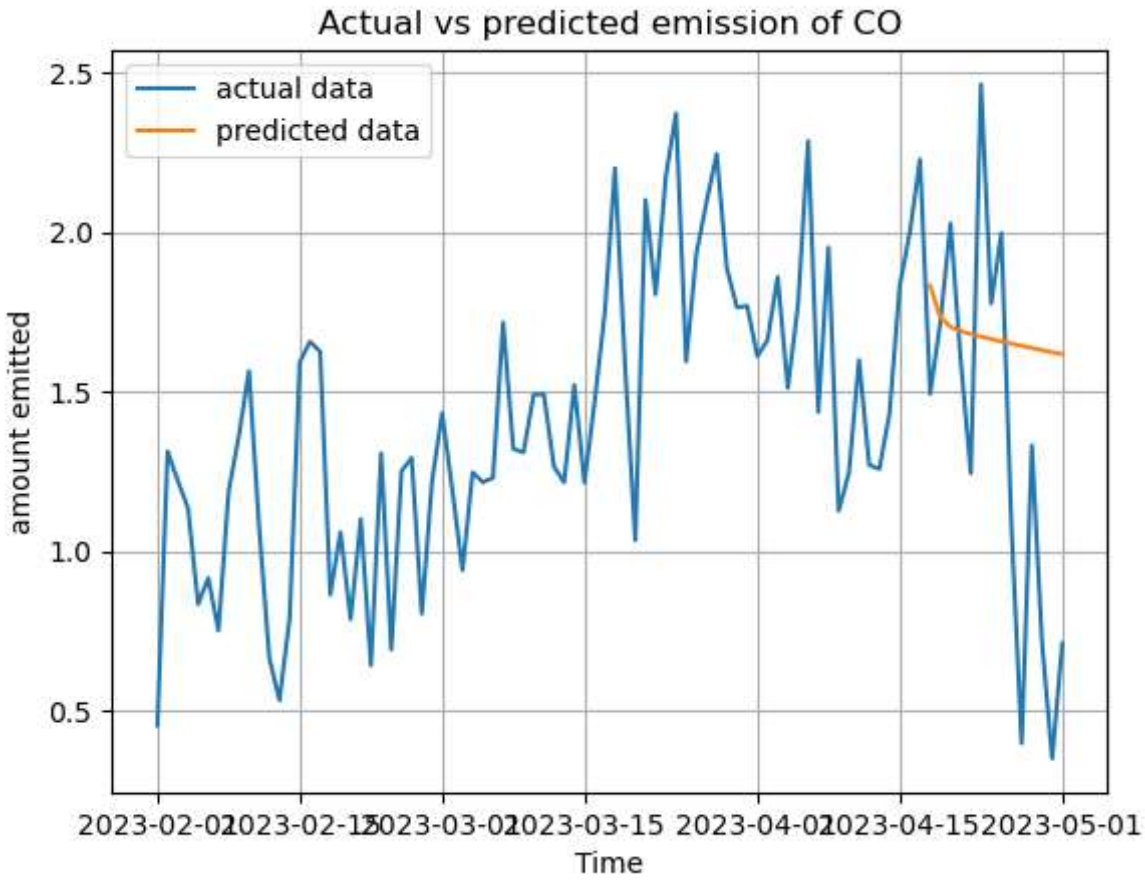
	coef	std err	z	P>|z|	[0.025	0.975]
intercept	0.0285	0.042	0.674	0.500	-0.054	0.111
ar.L1	1.2071	0.166	7.284	0.000	0.882	1.532
ar.L2	-0.2276	0.153	-1.485	0.138	-0.528	0.073
ma.L1	-0.8209	0.126	-6.490	0.000	-1.069	-0.573
sigma2	0.1150	0.024	4.879	0.000	0.069	0.161

Ljung-Box (L1) (Q):	0.12	**Jarque-Bera (JB):**	1.48
Prob(Q):	0.73	**Prob(JB):**	0.48
Heteroskedasticity (H):	0.81	**Skew:**	-0.05
Prob(H) (two-sided):	0.61	**Kurtosis:**	2.32

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

In [29]:
```
auto = model.predict(n_periods = len(data_test['CO']))
data_interp_linear['CO forecast'] = [None]*len(data_train['CO']) + list(auto)
```

In [30]:
```
 %matplotlib inline

#matplotlib notebook
plt.plot(CO_C,label='actual data')
plt.plot(data_interp_linear['CO forecast'], label='predicted data')
plt.xlabel('Time')
plt.ylabel('amount emitted')
plt.title('Actual vs predicted emission of CO')
plt.grid()
plt.legend()
plt.show()
```



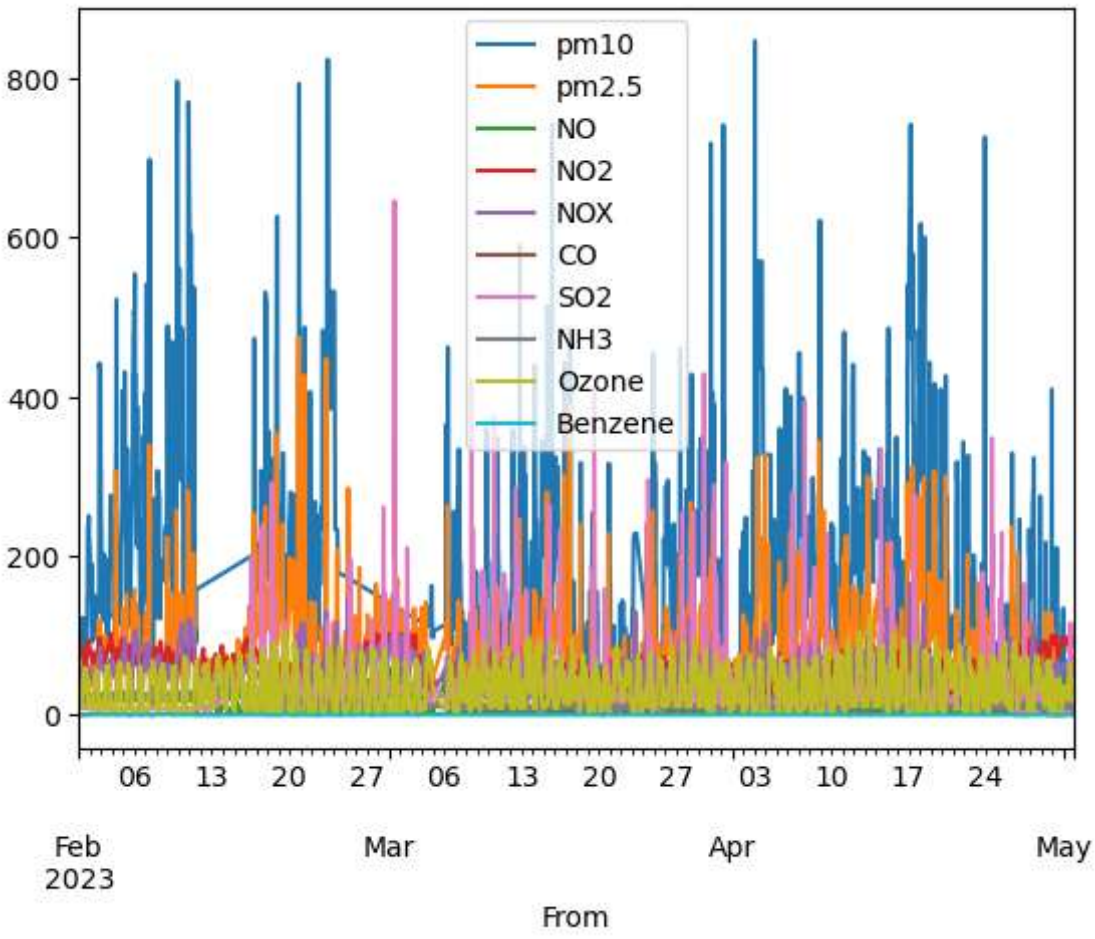
cubic interpolation

In [31]:
```
# Forward fill missing values from start
data_interp_cubic= data_interp_cubic.interpolate(limit_direction='both')


# Perform cubic interpolation for remaining NaN values
data_interp_cubic= data_interp_cubic.interpolate(method='cubic')

# Print the interpolated DataFrame
print(data_interp_cubic)
```

```
                     pm10  pm2.5    NO    NO2   NOX    CO   SO2   NH3  Ozone  \
From
2023-02-01 00:00:00   95.0   35.0  18.1   90.1  56.2  0.31   8.2  17.7   28.1
2023-02-01 00:15:00   95.0   35.0  18.1   88.0  55.1  0.33   8.2  18.3   27.1
2023-02-01 00:30:00   95.0   35.0  18.1   87.7  55.2  0.38   8.2  19.7   24.9
2023-02-01 00:45:00  122.0   34.0  18.1   88.9  55.7  0.38   8.2  21.3   21.9
2023-02-01 01:00:00  122.0   34.0  18.1   90.0  55.8  0.38   8.2  22.3   16.7
...                    ...    ...   ...    ...   ...   ...   ...   ...    ...
2023-05-01 22:45:00   19.0   11.0  17.9  100.0  67.8  0.63  10.0  10.7   26.1
2023-05-01 23:00:00   19.0   11.0  17.9  100.0  67.7  0.57  10.0  10.4   30.9
2023-05-01 23:15:00   19.0   11.0  19.6  100.2  69.2  0.58   9.9  10.5   29.6
2023-05-01 23:30:00   19.0   11.0  20.8  100.2  70.2  0.58   9.5  10.8   30.0
2023-05-01 23:45:00   32.0    6.0  21.8   98.8  70.3  0.58   9.5  11.0   33.5

                     Benzene
From
2023-02-01 00:00:00      0.4
2023-02-01 00:15:00      0.4
2023-02-01 00:30:00      0.4
2023-02-01 00:45:00      0.4
2023-02-01 01:00:00      0.4
...                      ...
2023-05-01 22:45:00      0.1
2023-05-01 23:00:00      0.1
2023-05-01 23:15:00      0.1
2023-05-01 23:30:00      0.1
2023-05-01 23:45:00      0.1

[8640 rows x 10 columns]
```

In [32]:
```
data_interp_cubic.plot()
```

Out[32]:
```
<Axes: xlabel='From'>
```



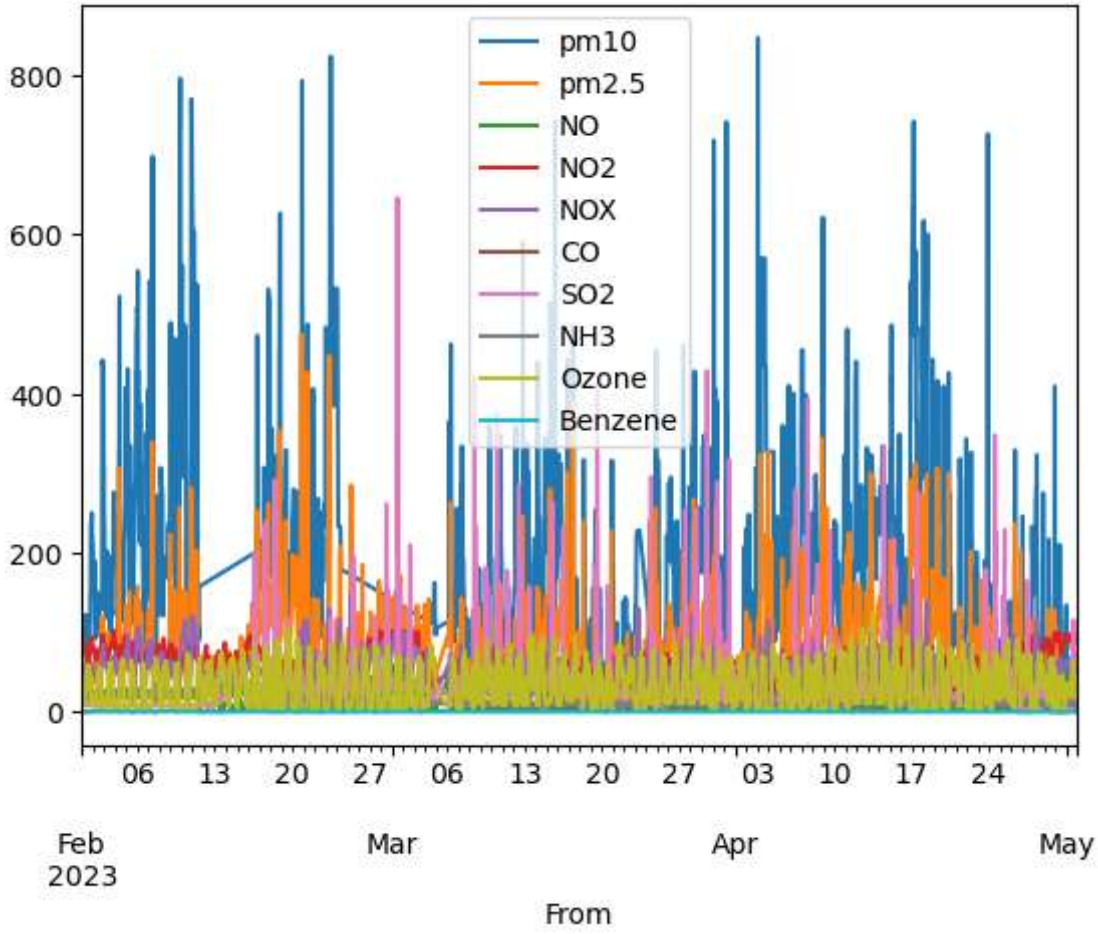
spline interpolation

In [33]:
```
data_interp_spline= data_interp_spline.interpolate(limit_direction='both')
data_interp_spline = data_interp_spline.interpolate(method='spline', order=3)

# Print the interpolated DataFrame
print(data_interp_spline)
```

```
                     pm10  pm2.5    NO    NO2   NOX    CO   SO2   NH3  Ozone  \
From
2023-02-01 00:00:00   95.0   35.0  18.1   90.1  56.2  0.31   8.2  17.7   28.1
2023-02-01 00:15:00   95.0   35.0  18.1   88.0  55.1  0.33   8.2  18.3   27.1
2023-02-01 00:30:00   95.0   35.0  18.1   87.7  55.2  0.38   8.2  19.7   24.9
2023-02-01 00:45:00  122.0   34.0  18.1   88.9  55.7  0.38   8.2  21.3   21.9
2023-02-01 01:00:00  122.0   34.0  18.1   90.0  55.8  0.38   8.2  22.3   16.7
...                    ...    ...   ...    ...   ...   ...   ...   ...    ...
2023-05-01 22:45:00   19.0   11.0  17.9  100.0  67.8  0.63  10.0  10.7   26.1
2023-05-01 23:00:00   19.0   11.0  17.9  100.0  67.7  0.57  10.0  10.4   30.9
2023-05-01 23:15:00   19.0   11.0  19.6  100.2  69.2  0.58   9.9  10.5   29.6
2023-05-01 23:30:00   19.0   11.0  20.8  100.2  70.2  0.58   9.5  10.8   30.0
2023-05-01 23:45:00   32.0    6.0  21.8   98.8  70.3  0.58   9.5  11.0   33.5

                     Benzene
From
2023-02-01 00:00:00      0.4
2023-02-01 00:15:00      0.4
2023-02-01 00:30:00      0.4
2023-02-01 00:45:00      0.4
2023-02-01 01:00:00      0.4
...                      ...
2023-05-01 22:45:00      0.1
2023-05-01 23:00:00      0.1
2023-05-01 23:15:00      0.1
2023-05-01 23:30:00      0.1
2023-05-01 23:45:00      0.1

[8640 rows x 10 columns]
```

In [34]:
```
data_interp_spline.plot()
```

Out[34]:
```
<Axes: xlabel='From'>
```



comparision of all three interpolation

The plot displays the results of three interpolation methods: Linear, Cubic, and Spline interpolation. Each method offers a different approach to estimating values between data points.

Linear Interpolation: Linear interpolation provides a simple and straightforward estimation by connecting two adjacent data points with a straight line. It assumes a constant rate of change between points.

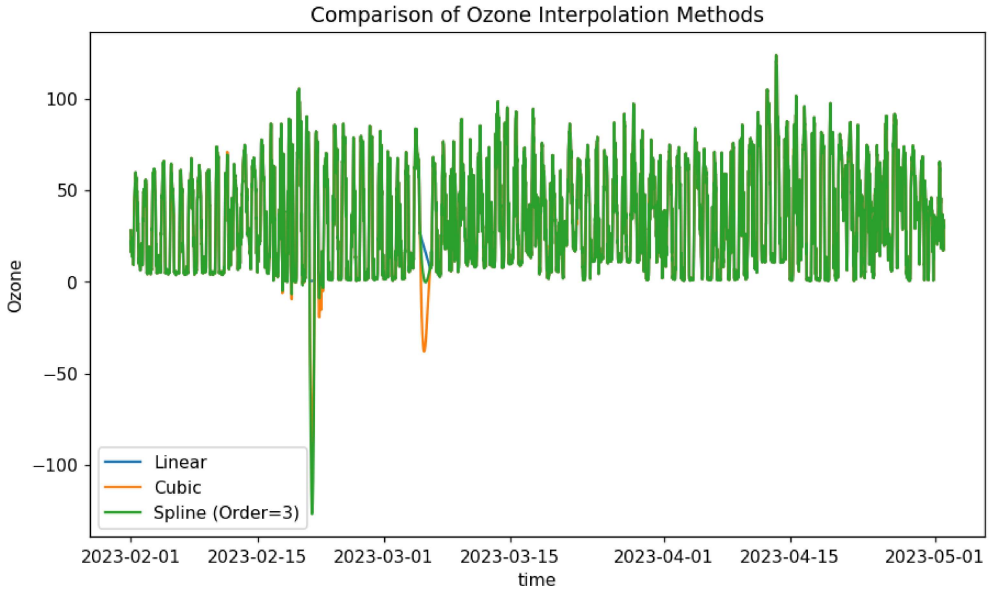
Cubic Interpolation: Cubic interpolation offers a more accurate estimation by using cubic polynomials to approximate the shape of the curve between data points. It provides a smoother interpolation compared to linear interpolation and captures more intricate patterns.

Spline Interpolation: Spline interpolation is a flexible method that utilizes piecewise-defined polynomials to create a smooth curve passing through each data point. It can capture complex and nonlinear patterns more effectively.

Analyzing the plot, we can observe that the cubic and linear interpolation curves are relatively similar, indicating that the cubic interpolation does not significantly deviate from the linear approximation in this case. However, the spline interpolation curve stands out as it differs significantly from the other two curves. The spline interpolation captures more intricate details and exhibits a smoother overall trend.

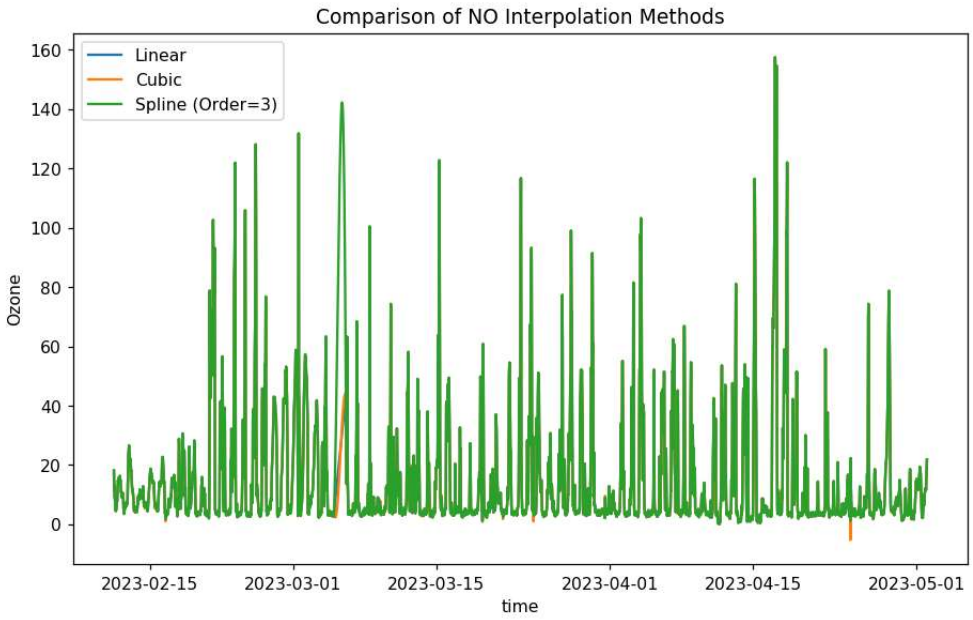
Based on this analysis, we can conclude that spline interpolation appears to be a better choice for this specific dataset and analysis. It provides a more accurate representation of the underlying pattern, capturing more nuanced variations between data points. However, the choice of interpolation method ultimately depends on the specific characteristics of the data and the desired level of accuracy or smoothness required for the analysis.

In [35]:
```
%matplotlib notebook

# Define the column to compare
column_to_compare = 'Ozone'

# Perform linear interpolation
linear_interpolated = data[column_to_compare].interpolate(method='linear')

# Perform cubic interpolation
cubic_interpolated = data[column_to_compare].interpolate(method='cubic')

# Perform spline interpolation
spline_interpolated = data[column_to_compare].interpolate(method='spline', order=3

# Initialize a figure and axis
fig, ax = plt.subplots()

# Plot the original data
#ax.plot(data.index, data[column_to_compare], label='Original')

# Plot the linear interpolated values
ax.plot(data.index, linear_interpolated, label='Linear')

# Plot the cubic interpolated values
ax.plot(data.index, cubic_interpolated, label='Cubic')

# Plot the spline interpolated values
ax.plot(data.index, spline_interpolated, label='Spline (Order=3)')

# Set labels and title
ax.set_xlabel('time')
ax.set_ylabel('Ozone')
ax.set_title(f'Comparison of {column_to_compare} Interpolation Methods')

# Add legend
ax.legend()

# Show the plot
plt.show()
```



NO

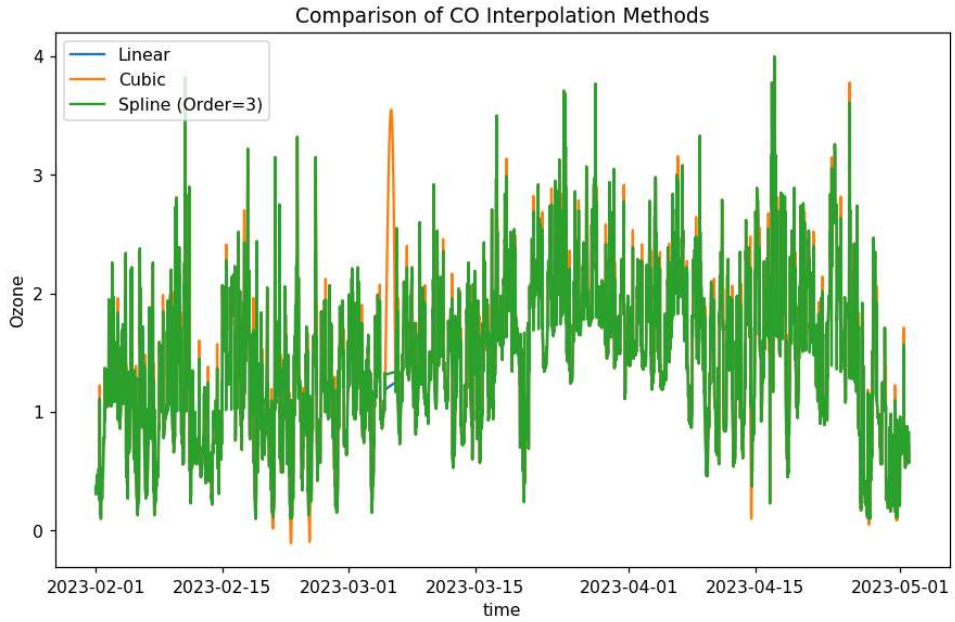
In [36]:
```
%matplotlib notebook

# Define the column to compare
column_to_compare = 'NO'

# Perform linear interpolation
linear_interpolated = data[column_to_compare].interpolate(method='linear')

# Perform cubic interpolation
cubic_interpolated = data[column_to_compare].interpolate(method='cubic')

# Perform spline interpolation
spline_interpolated = data[column_to_compare].interpolate(method='spline', order=3

# Initialize a figure and axis
fig, ax = plt.subplots()

# Plot the original data
#ax.plot(data.index, data[column_to_compare], label='Original')

# Plot the linear interpolated values
ax.plot(data.index, linear_interpolated, label='Linear')

# Plot the cubic interpolated values
ax.plot(data.index, cubic_interpolated, label='Cubic')

# Plot the spline interpolated values
ax.plot(data.index, spline_interpolated, label='Spline (Order=3)')

# Set labels and title
ax.set_xlabel('time')
ax.set_ylabel('Ozone')
ax.set_title(f'Comparison of {column_to_compare} Interpolation Methods')

# Add legend
ax.legend()

# Show the plot
plt.show()
```



CO

In [37]:
```
%matplotlib notebook

# Define the column to compare
column_to_compare = 'CO'

# Perform linear interpolation
linear_interpolated = data[column_to_compare].interpolate(method='linear')

# Perform cubic interpolation
cubic_interpolated = data[column_to_compare].interpolate(method='cubic')

# Perform spline interpolation
spline_interpolated = data[column_to_compare].interpolate(method='spline', order=3

# Initialize a figure and axis
fig, ax = plt.subplots()

# Plot the original data
#ax.plot(data.index, data[column_to_compare], label='Original')

# Plot the linear interpolated values
ax.plot(data.index, linear_interpolated, label='Linear')

# Plot the cubic interpolated values
ax.plot(data.index, cubic_interpolated, label='Cubic')

# Plot the spline interpolated values
ax.plot(data.index, spline_interpolated, label='Spline (Order=3)')

# Set labels and title
ax.set_xlabel('time')
ax.set_ylabel('Ozone')
ax.set_title(f'Comparison of {column_to_compare} Interpolation Methods')

# Add legend
ax.legend()
```

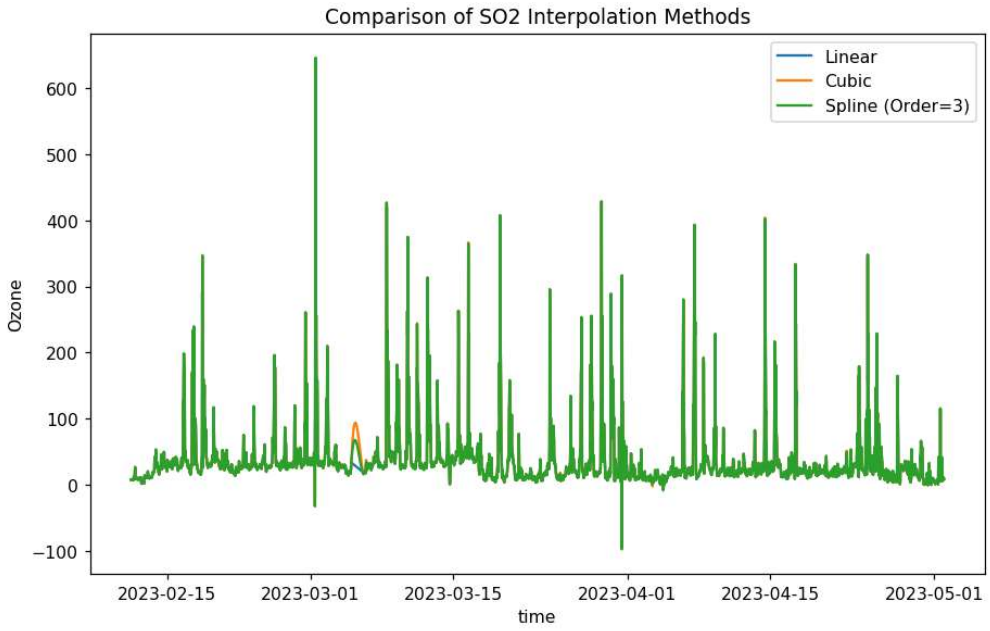
```
# Show the plot
plt.show()
```



SO2

In [38]:
```
%matplotlib notebook

# Define the column to compare
column_to_compare = 'SO2'

# Perform linear interpolation
linear_interpolated = data[column_to_compare].interpolate(method='linear')

# Perform cubic interpolation
cubic_interpolated = data[column_to_compare].interpolate(method='cubic')

# Perform spline interpolation
spline_interpolated = data[column_to_compare].interpolate(method='spline', order=3

# Initialize a figure and axis
fig, ax = plt.subplots()

# Plot the original data
#ax.plot(data.index, data[column_to_compare], label='Original')

# Plot the linear interpolated values
ax.plot(data.index, linear_interpolated, label='Linear')

# Plot the cubic interpolated values
ax.plot(data.index, cubic_interpolated, label='Cubic')

# Plot the spline interpolated values
ax.plot(data.index, spline_interpolated, label='Spline (Order=3)')

# Set labels and title
ax.set_xlabel('time')
ax.set_ylabel('Ozone')
ax.set_title(f'Comparison of {column_to_compare} Interpolation Methods')

# Add legend
```

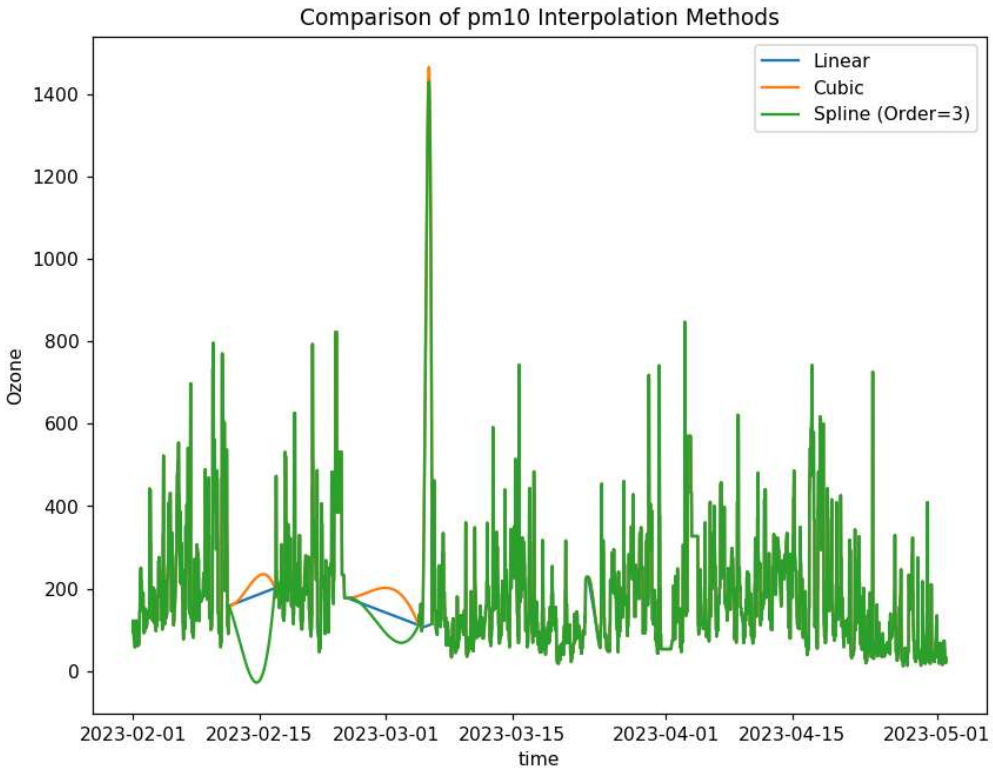
```
ax.legend()

# Show the plot
plt.show()
```



pm10

In [39]:
```
%matplotlib notebook

# Define the column to compare
column_to_compare = 'pm10'

# Perform linear interpolation
linear_interpolated = data[column_to_compare].interpolate(method='linear')

# Perform cubic interpolation
cubic_interpolated = data[column_to_compare].interpolate(method='cubic')

# Perform spline interpolation
spline_interpolated = data[column_to_compare].interpolate(method='spline', order=3

# Initialize a figure and axis
fig, ax = plt.subplots()

# Plot the original data
#ax.plot(data.index, data[column_to_compare], label='Original')

# Plot the linear interpolated values
ax.plot(data.index, linear_interpolated, label='Linear')

# Plot the cubic interpolated values
ax.plot(data.index, cubic_interpolated, label='Cubic')

# Plot the spline interpolated values
ax.plot(data.index, spline_interpolated, label='Spline (Order=3)')

# Set labels and title
ax.set_xlabel('time')
ax.set_ylabel('Ozone')
```

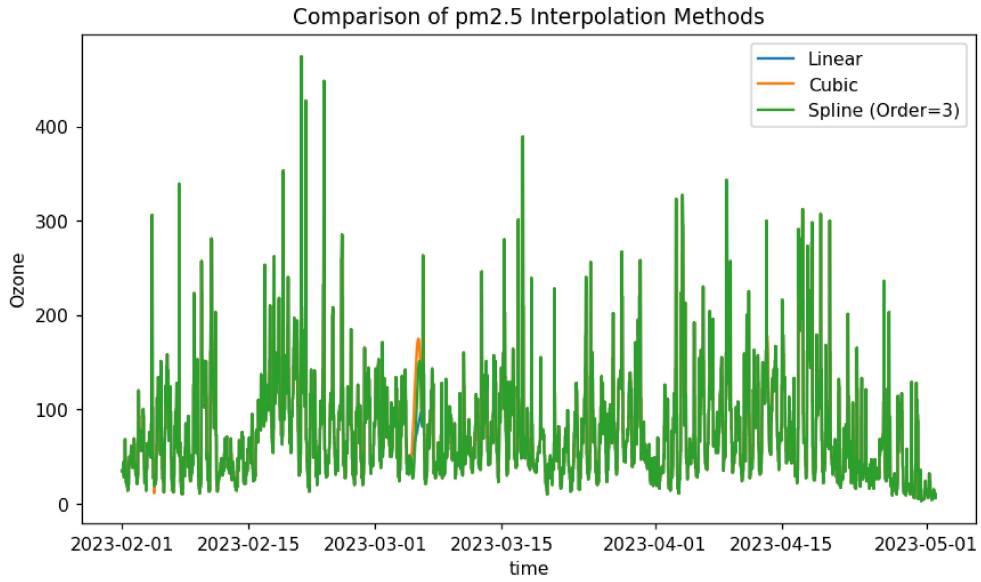
```
ax.set_title(f'Comparison of {column_to_compare} Interpolation Methods')

# Add legend
ax.legend()

# Show the plot
plt.show()
```



pm2.5

In [40]:
```
%matplotlib notebook

# Define the column to compare
column_to_compare = 'pm2.5'

# Perform linear interpolation
linear_interpolated = data[column_to_compare].interpolate(method='linear')

# Perform cubic interpolation
cubic_interpolated = data[column_to_compare].interpolate(method='cubic')

# Perform spline interpolation
spline_interpolated = data[column_to_compare].interpolate(method='spline', order=3

# Initialize a figure and axis
fig, ax = plt.subplots()

# Plot the original data
#ax.plot(data.index, data[column_to_compare], label='Original')

# Plot the linear interpolated values
ax.plot(data.index, linear_interpolated, label='Linear')

# Plot the cubic interpolated values
```

```
ax.plot(data.index, cubic_interpolated, label='Cubic')

# Plot the spline interpolated values
ax.plot(data.index, spline_interpolated, label='Spline (Order=3)')

# Set labels and title
ax.set_xlabel('time')
ax.set_ylabel('Ozone')
ax.set_title(f'Comparison of {column_to_compare} Interpolation Methods')

# Add legend
ax.legend()

# Show the plot
plt.show()
```

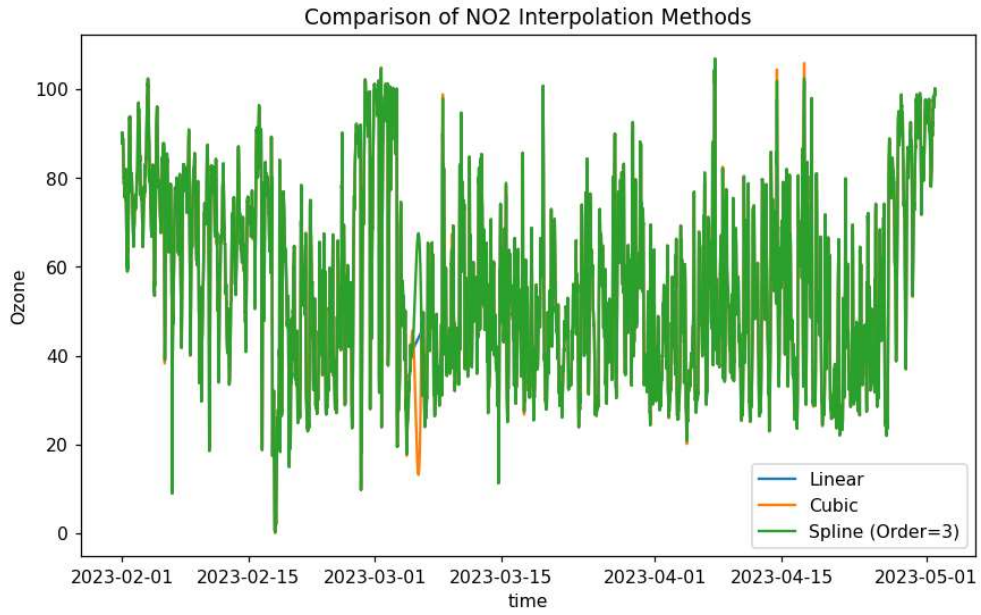


NO2

In [41]:

```
%matplotlib notebook

# Define the column to compare
column_to_compare = 'NO2'

# Perform linear interpolation
linear_interpolated = data[column_to_compare].interpolate(method='linear')

# Perform cubic interpolation
cubic_interpolated = data[column_to_compare].interpolate(method='cubic')

# Perform spline interpolation
spline_interpolated = data[column_to_compare].interpolate(method='spline', order=3

# Initialize a figure and axis
fig, ax = plt.subplots()

# Plot the original data
#ax.plot(data.index, data[column_to_compare], label='Original')

# Plot the linear interpolated values
ax.plot(data.index, linear_interpolated, label='Linear')

# Plot the cubic interpolated values
```

```
ax.plot(data.index, cubic_interpolated, label='Cubic')

# Plot the spline interpolated values
ax.plot(data.index, spline_interpolated, label='Spline (Order=3)')

# Set labels and title
ax.set_xlabel('time')
ax.set_ylabel('Ozone')
ax.set_title(f'Comparison of {column_to_compare} Interpolation Methods')

# Add legend
ax.legend()

# Show the plot
plt.show()
```

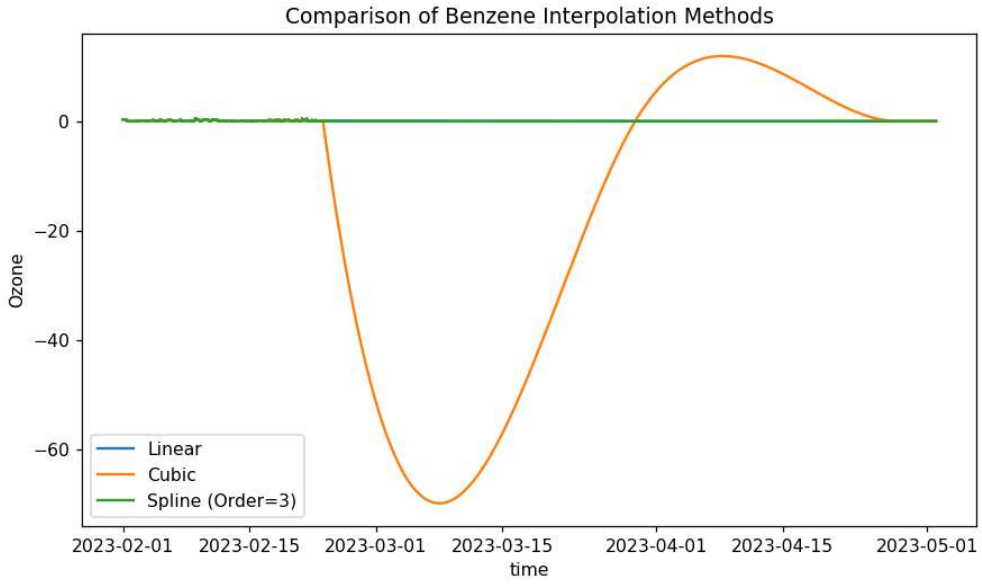


benzene

In [42]:

```
%matplotlib notebook

# Define the column to compare
column_to_compare = 'Benzene'

# Perform linear interpolation
linear_interpolated = data[column_to_compare].interpolate(method='linear')

# Perform cubic interpolation
cubic_interpolated = data[column_to_compare].interpolate(method='cubic')

# Perform spline interpolation
spline_interpolated = data[column_to_compare].interpolate(method='spline', order=3

# Initialize a figure and axis
fig, ax = plt.subplots()

# Plot the original data
#ax.plot(data.index, data[column_to_compare], label='Original')

# Plot the linear interpolated values
ax.plot(data.index, linear_interpolated, label='Linear')
```

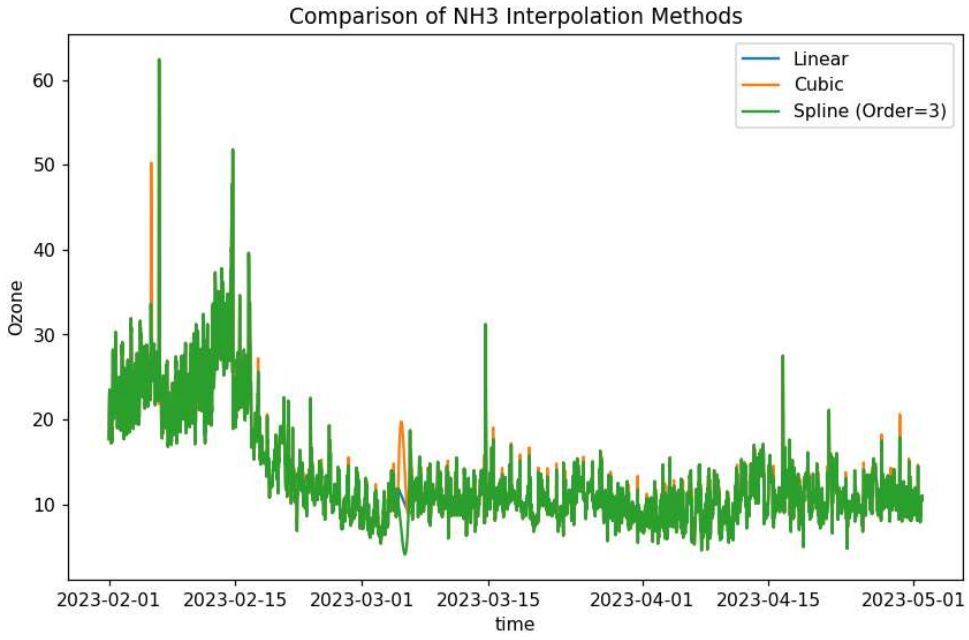
```
# Plot the cubic interpolated values
ax.plot(data.index, cubic_interpolated, label='Cubic')

# Plot the spline interpolated values
ax.plot(data.index, spline_interpolated, label='Spline (Order=3)')

# Set labels and title
ax.set_xlabel('time')
ax.set_ylabel('Ozone')
ax.set_title(f'Comparison of {column_to_compare} Interpolation Methods')

# Add legend
ax.legend()

# Show the plot
plt.show()
```



NH3

In [43]:
```
%matplotlib notebook

# Define the column to compare
column_to_compare = 'NH3'

# Perform linear interpolation
linear_interpolated = data[column_to_compare].interpolate(method='linear')

# Perform cubic interpolation
cubic_interpolated = data[column_to_compare].interpolate(method='cubic')

# Perform spline interpolation
spline_interpolated = data[column_to_compare].interpolate(method='spline', order=3

# Initialize a figure and axis
fig, ax = plt.subplots()

# Plot the original data
#ax.plot(data.index, data[column_to_compare], label='Original')

# Plot the linear interpolated values
ax.plot(data.index, linear_interpolated, label='Linear')
```

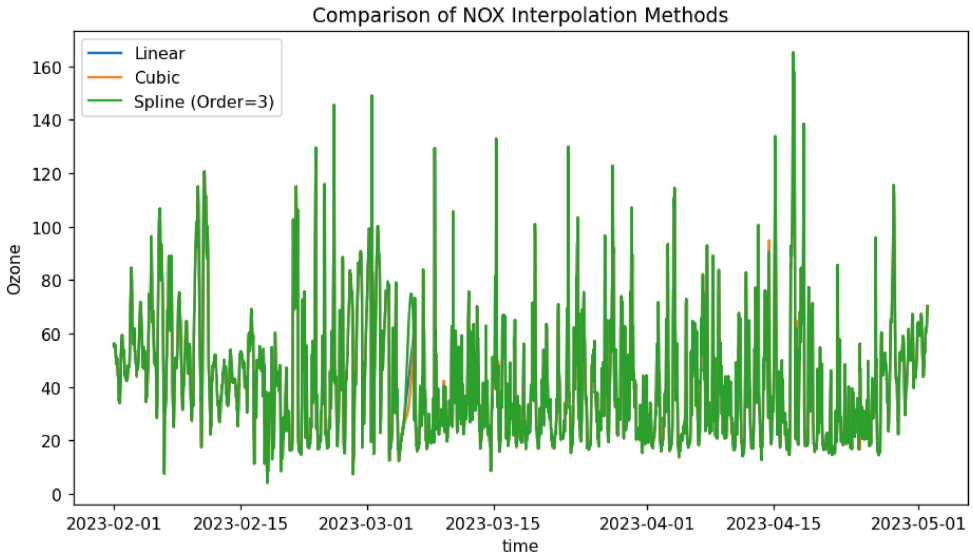
```
# Plot the cubic interpolated values
ax.plot(data.index, cubic_interpolated, label='Cubic')

# Plot the spline interpolated values
ax.plot(data.index, spline_interpolated, label='Spline (Order=3)')

# Set labels and title
ax.set_xlabel('time')
ax.set_ylabel('Ozone')
ax.set_title(f'Comparison of {column_to_compare} Interpolation Methods')

# Add legend
ax.legend()

# Show the plot
plt.show()
```



NOX

In [44]:
```
%matplotlib notebook

# Define the column to compare
column_to_compare = 'NOX'

# Perform linear interpolation
linear_interpolated = data[column_to_compare].interpolate(method='linear')

# Perform cubic interpolation
cubic_interpolated = data[column_to_compare].interpolate(method='cubic')

# Perform spline interpolation
spline_interpolated = data[column_to_compare].interpolate(method='spline', order=3

# Initialize a figure and axis
fig, ax = plt.subplots()

# Plot the original data
#ax.plot(data.index, data[column_to_compare], label='Original')

# Plot the linear interpolated values
```

```
ax.plot(data.index, linear_interpolated, label='Linear')

# Plot the cubic interpolated values
ax.plot(data.index, cubic_interpolated, label='Cubic')

# Plot the spline interpolated values
ax.plot(data.index, spline_interpolated, label='Spline (Order=3)')

# Set labels and title
ax.set_xlabel('time')
ax.set_ylabel('Ozone')
ax.set_title(f'Comparison of {column_to_compare} Interpolation Methods')

# Add legend
ax.legend()

# Show the plot
plt.show()
```

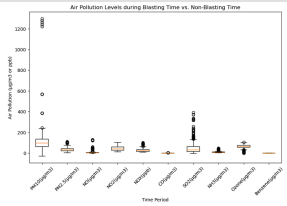


blasting

In [ ]:

```
blasting_time = df['From']
blasting_time=blasting_time(:-3)
air_pollution_columns = ['pm10', 'pm2.5', 'NO', 'NO2', 'NOX', 'CO', 'SO2', 'NH3',
air_pollution = data_intr[air_pollution_columns]

# Convert blasting time to datetime format
blasting_time = pd.to_datetime(blasting_time, format='%d-%m-%Y %H.%M')

# Set the blasting time range
blasting_start = pd.to_datetime('13:45', format='%H:%M').time()
blasting_end = pd.to_datetime('14:45', format='%H:%M').time()

# Create a boolean mask for blasting time
blasting_mask = (blasting_time.dt.time >= blasting_start) & (blasting_time.dt.time

# Create separate DataFrame for air pollution levels during blasting time and other
blasting_pollution = air_pollution[blasting_mask]
non_blasting_pollution = air_pollution[~blasting_mask]

# Perform a t-test for each column to determine if the difference is significant
```

```
p_values = np.zeros(len(air_pollution_columns))
for i, col in enumerate(air_pollution_columns):
    _, p_values[i] = ttest_ind(blasting_pollution[col], non_blasting_pollution[col

# Analyze the significance level (p-value) to validate the information for each col
significant_columns = []
for i, col in enumerate(air_pollution_columns):
    if p_values[i] < 0.05:
        significant_columns.append(col)

if len(significant_columns) > 0:
    print("The air pollution levels during blasting time are significantly differer
else:
    print("The air pollution levels during blasting time are not significantly dif

# Visualize the air pollution levels during blasting time and other periods using b
plt.figure(figsize=(10, 6))
plt.boxplot([blasting_pollution[col] for col in air_pollution_columns], labels=air_
plt.xlabel('Time Period')
plt.ylabel('Air Pollution (µg/m3 or ppb)')
plt.title('Air Pollution Levels during Blasting Time vs. Non-Blasting Time')
plt.xticks(rotation=45)
plt.show()
```



In [ ]:

```
blasting_time = df['From']
blasting_time=blasting_time(:-3)
air_pollution_columns = ['pm10', 'pm2.5', 'NO', 'NO2', 'NOX', 'CO', 'SO2', 'NH3',
air_pollution = data_intr[air_pollution_columns]

# Convert blasting time to datetime format
blasting_time = pd.to_datetime(blasting_time,  format='%d-%m-%Y %H.%M')

# Set the blasting time range
blasting_start = pd.to_datetime('13:45', format='%H:%M').time()
blasting_end = pd.to_datetime('14:45', format='%H:%M').time()

# Create a boolean mask for blasting time
blasting_mask = (blasting_time.dt.time >= blasting_start) & (blasting_time.dt.time

# Calculate the average pollution levels for blasting and non-blasting time periods
blasting_avg_pollution = air_pollution[blasting_mask].mean()
non_blasting_avg_pollution = air_pollution[~blasting_mask].mean()

# Plot the average pollution levels for blasting and non-blasting time periods
plt.figure(figsize=(10, 6))
plt.plot(blasting_avg_pollution, marker='o', label='Blasting Time')
plt.plot(non_blasting_avg_pollution, marker='o', label='Non-Blasting Time')
plt.xlabel('Pollution Factors')
plt.ylabel('Average Pollution Levels (µg/m3 or ppb)')
plt.title('Average Pollution Levels during Blasting Time vs. Non-Blasting Time')
plt.xticks(rotation=45)
plt.legend()
plt.show()
```

Identify the relevant air polluting factors: Determine the specific air polluting factors that are directly associated with the blasting activity. These factors could include particulate matter (PM), nitrogen dioxide (NO2), sulfur dioxide (SO2), carbon monoxide (CO), etc.

Collect historical data: Gather historical data for each of the identified air polluting factors. This data should cover the period during which the blasting activities occurred.

Normalize the data: Normalize the data for each air polluting factor to a common scale. This step is important to ensure that each factor contributes proportionately to the combined time-series data.

Determine weights: Assign weights to each air polluting factor based on their relative importance in capturing the pollution effect of blasting. The weights can be determined based on expert knowledge, domain expertise, or statistical analysis.

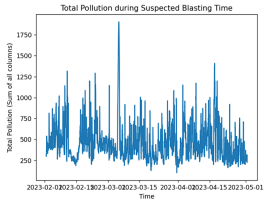
Calculate the weighted combination: Multiply each normalized air polluting factor by its corresponding weight. Sum up the weighted values for each factor to obtain a single combined time-series data.

Validate and refine: Validate the combined time-series data by comparing it with actual measurements or observed pollution levels during blasting activities. If necessary, refine the weights or adjust the normalization process to improve the accuracy of the combined data.

By deriving a combined weighted time-series data using this approach, we can capture the overall pollution effect of blasting activities by considering multiple air polluting factors. The weights assigned to each factor allow you to emphasize the factors that have a greater impact on pollution during blasting, providing a comprehensive representation of the pollution levels.

In [ ]:
```
%matplotlib notebook
blasting_time = df['From']
blasting_time=blasting_time(:-3)
air_pollution_columns = ['pm10', 'pm2.5', 'NO', 'NO2', 'NOX', 'CO', 'SO2', 'NH3',
air_pollution = data_intr[air_pollution_columns]

# Convert blasting time to datetime format
blasting_time = pd.to_datetime(blasting_time, format='%d-%m-%Y %H.%M')

# Set the suspected blasting time window
blasting_window_start = pd.to_datetime('13:45', format='%H:%M').time()
blasting_window_end = pd.to_datetime('14:45', format='%H:%M').time()

# Find the index range of suspected blasting time
blasting_start_index = blasting_time.index[blasting_time.dt.time == blasting_windo
blasting_end_index = blasting_time.index[blasting_time.dt.time == blasting_window_

# Calculate the sum of pollution across all columns within the suspected blasting
blasting_total_pollution = air_pollution.iloc[blasting_start_index:blasting_end_in

# Plot the total pollution during the suspected blasting time window
plt.plot(blasting_time[blasting_start_index:blasting_end_index+1], blasting_total_
plt.xlabel('Time')
plt.ylabel('Total Pollution (Sum of all columns)')
```

```
plt.title('Total Pollution during Suspected Blasting Time')
plt.show()
```

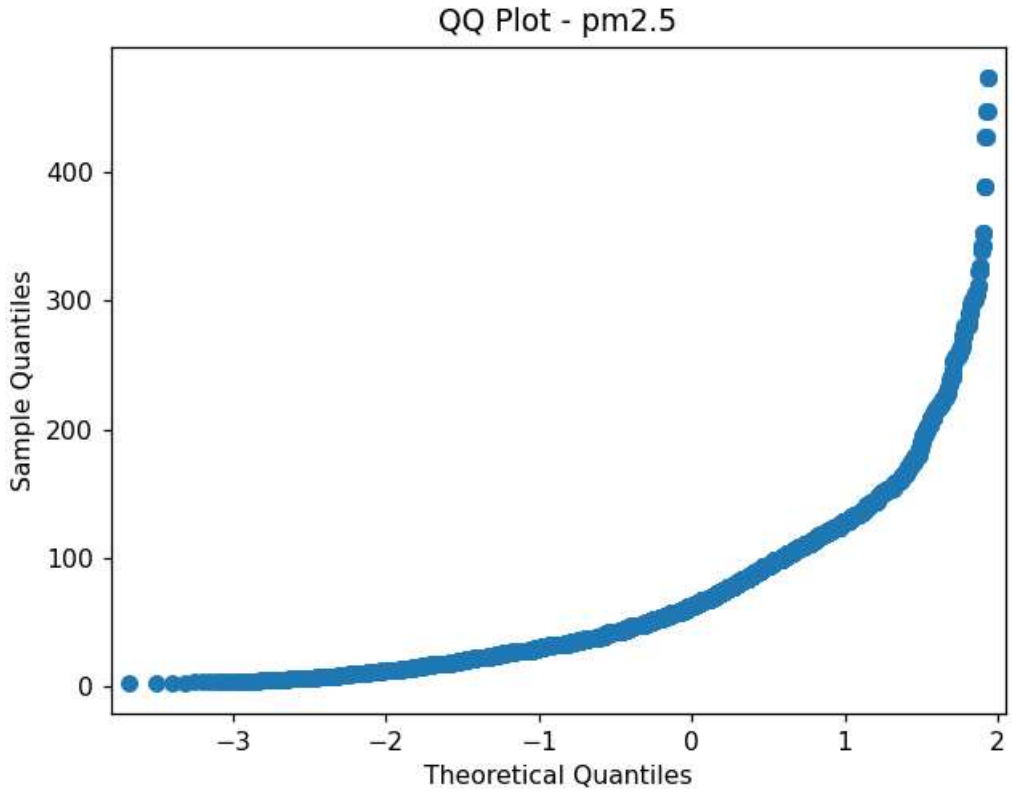


Yes, a QQ plot (Quantile-Quantile plot) can be used to infer whether a dataset follows a normal distribution or not. In a QQ plot, the quantiles of the dataset are plotted against the quantiles of a theoretical normal distribution. If the data points in the QQ plot approximately fall on or near a straight line, it suggests that the dataset follows a normal distribution. However, if the points deviate significantly from the straight line, it indicates a departure from normality.

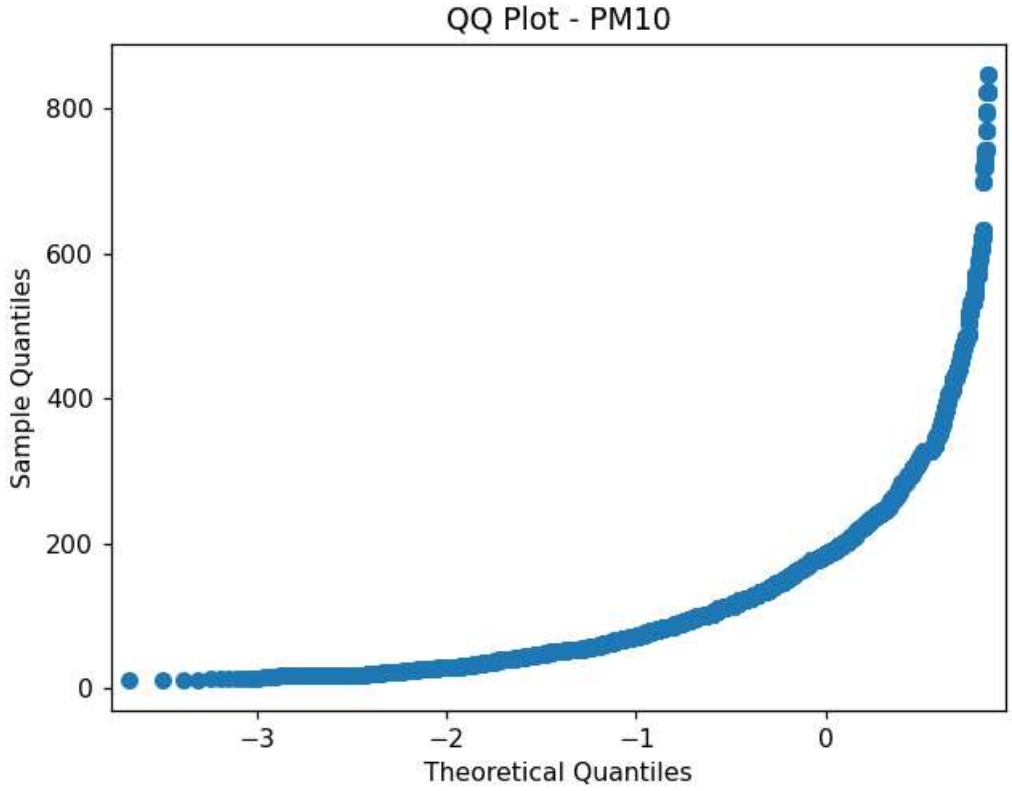
Specifically, if the points in the QQ plot:

Fall approximately along a straight line: It suggests that the dataset follows a normal distribution. Deviate from a straight line in an upward or downward curvature: It suggests a departure from normality, indicating non-normal distribution. Deviate from a straight line in the tails of the plot: It suggests heavy-tailed or light-tailed distributions compared to the normal distribution. Therefore, by analyzing the pattern and deviation of points in the QQ plot, you can make an inference about whether the dataset follows a normal distribution or not.

In [45]:
```
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm

# Select the column you want to create a QQ plot for
column_name = 'pm2.5'


# Extract the column data
dataqq = data[column_name]

# Generate theoretical quantiles
theoretical_quantiles = sm.ProbPlot(dataqq).theoretical_quantiles

# Create QQ plot
sm.qqplot(dataqq, line='s')

# Customize the plot
plt.title(f"QQ Plot - {column_name}")
plt.xlabel("Theoretical Quantiles")
plt.ylabel("Sample Quantiles")

# Show the plot
plt.show()
```



In [46]:
```
 #Extract the column data
dataqq = data['pm10']

# Generate theoretical quantiles
theoretical_quantiles = sm.ProbPlot(dataqq).theoretical_quantiles

# Create QQ plot
sm.qqplot(dataqq, line='s')

# Customize the plot
plt.title(f"QQ Plot - PM10")
plt.xlabel("Theoretical Quantiles")
plt.ylabel("Sample Quantiles")

# Show the plot
plt.show()
```


QQ Plot - PM10
Sample Quantiles
Theoretical Quantiles
800
600
400
200
0
−3
−2
−1
0

In [ ]: