In [41]:
```
import warnings
warnings.filterwarnings("ignore")
import pandas as pd
import yfinance as yf

from datetime import datetime
from itertools import product
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import statsmodels as ss
#import seaborn as sns
import warnings
from tqdm import tqdm
from scipy import stats, signal
from scipy.fft import fft
from statsmodels.graphics.tsaplots import month_plot, plot_acf, plot_pacf
from statsmodels.graphics.gofplots import qqplot
from statsmodels.tsa.stattools import adfuller, acf, pacf
from statsmodels.tsa.seasonal import STL, seasonal_decompose
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tools.eval_measures import mse
from statsmodels.tsa.statespace.tools import diff

np.random.seed(42)                # for reproducibility
warnings.filterwarnings("ignore")  # ignore warnings (usually a bad idea)
plt.rcParams['figure.figsize'] = (10, 4)
```

Exercises lecture 5

Exercise

- Look at sensor data that tracks atmospheric CO2 from continuous air samples at Mauna Loa Observatory in Hawaii. This data includes CO2 samples from MAR 1958 to DEC 1980.

In [2]:
```
# read the CSV file 'co2-ppm-mauna-loa-19651980.csv' into a dataframe
co2 = pd.read_csv('co2-ppm-mauna-loa-19651980.csv',
                  header = 0,
                  names = ['idx', 'co2'],
                  skipfooter = 2)

# convert the column idx into a datetime object and set it as the index
co2['idx'] = pd.to_datetime(co2['idx'])
co2.set_index('idx', inplace=True)

# Rmove the name "idx" from the index column
co2.index.name = None
co2
```

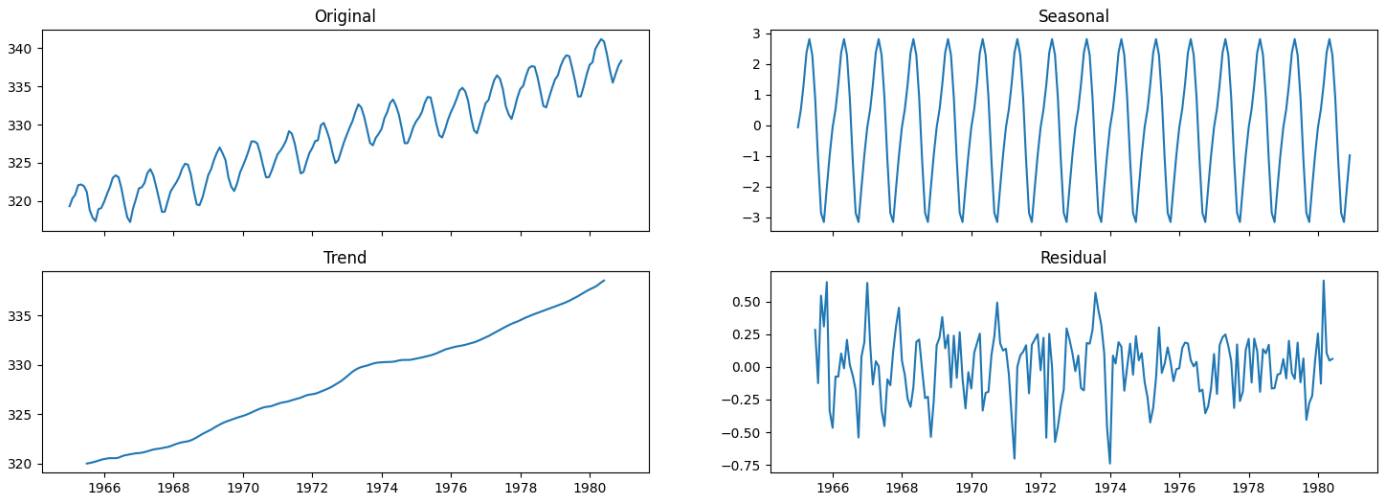
Out[2]:

	co2
1965-01-01	319.32
1965-02-01	320.36
1965-03-01	320.82
1965-04-01	322.06
1965-05-01	322.17
...	...
1980-08-01	337.19
1980-09-01	335.49
1980-10-01	336.63
1980-11-01	337.74
1980-12-01	338.36

192 rows × 1 columns

1. Determine the presence of main trend and seasonality in the data.

In [3]:
```
decomposition = seasonal_decompose(x=co2['co2'], model='additive', period=12)
seasonal, trend, resid = decomposition.seasonal, decomposition.trend, decomposition.resi

fig, axs = plt.subplots(2,2, sharex=True, figsize=(18,6))
axs[0,0].plot(co2['co2'])
axs[0,0].set_title('Original')
axs[0,1].plot(seasonal)
axs[0,1].set_title('Seasonal')
axs[1,0].plot(trend)
axs[1,0].set_title('Trend')
axs[1,1].plot(resid)
axs[1,1].set_title('Residual');
```



2. Determine if the data are stationary.

In [4]:
```
#the presence of trend and seaosanality suggest that data is not stationary
```

3. Split the data in train (90%) and test (10%)

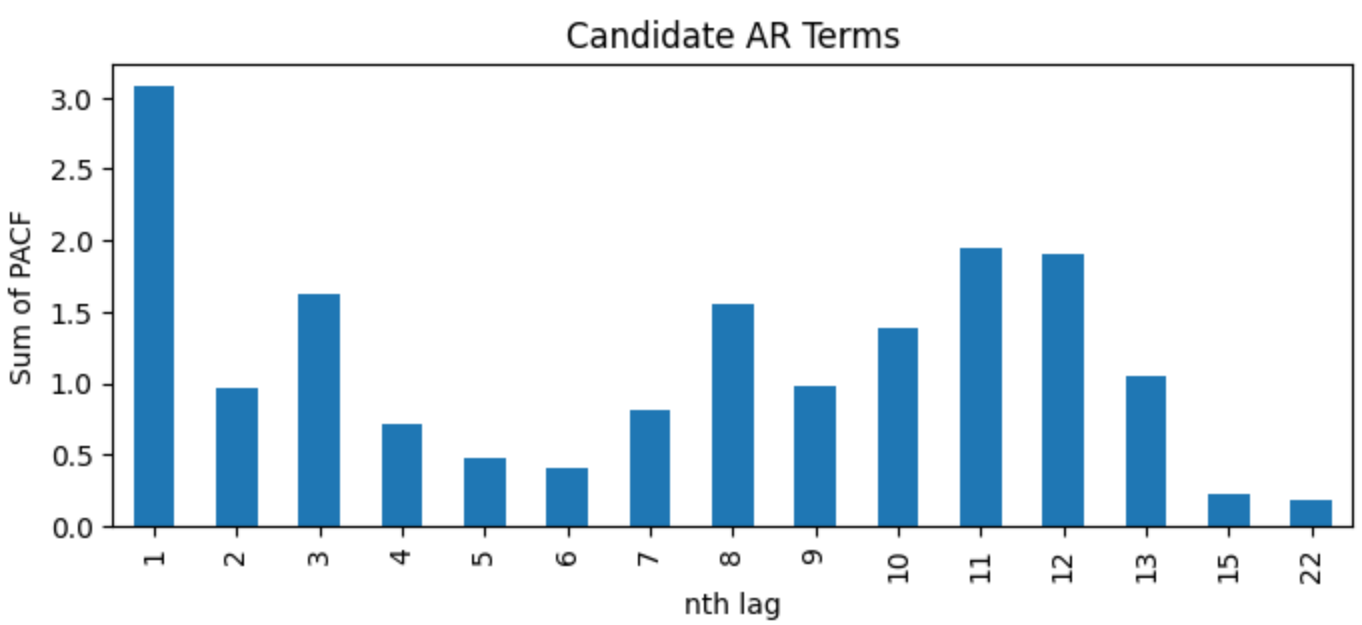
In [5]:
```
# Determine the number of rows for training and testing
total_rows = len(co2)
train_size = int(total_rows * 0.9)
test_size = total_rows - train_size

# Split the data into train (90%) and test (10%)
train_data = co2.iloc[:train_size]
test_data = co2.iloc[train_size:]

# Display the shapes of the train and test sets
print("Train set shape:", train_data.shape)
print("Test set shape:", test_data.shape)
```

```
Train set shape: (172, 1)
Test set shape: (20, 1)
```

4. Find a set of SARIMAX candidate models by looking at the ACF and PACF

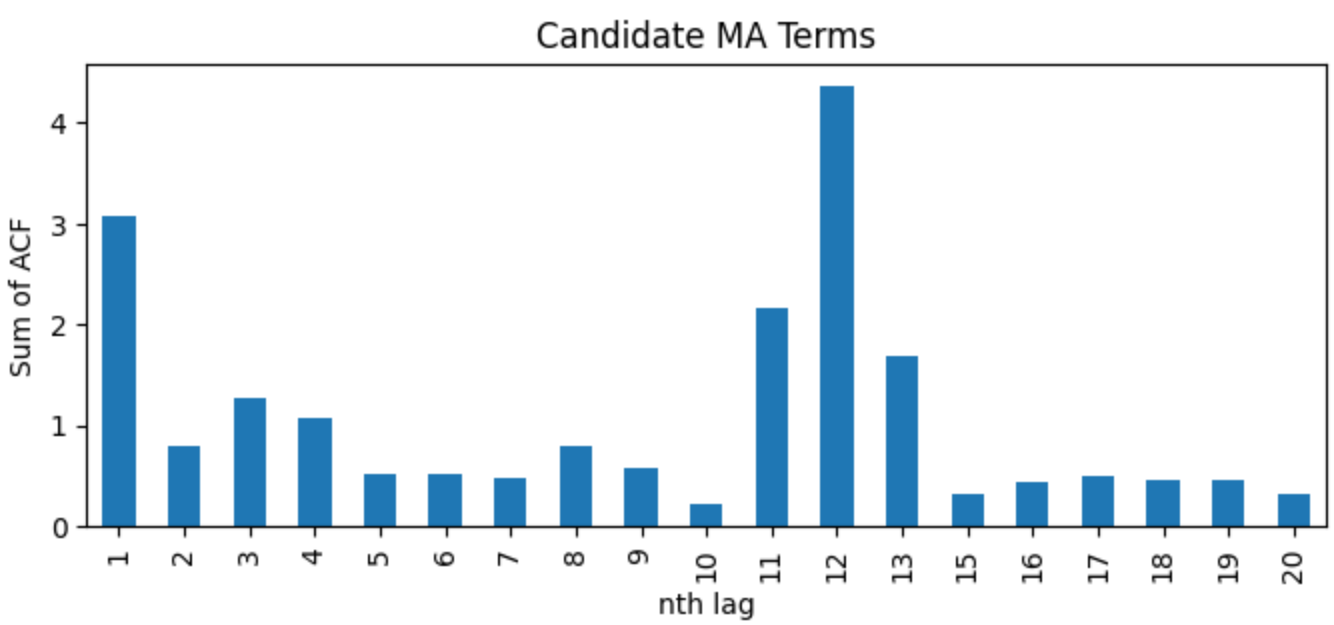
In [6]:
```
def differencing(timeseries, s, D_max=2, d_max=2):

    # Seasonal differencing from 0 to D_max
    seas_differenced = []
    for i in range(D_max+1):
        timeseries.name = f"d0_D{i}_s{s}"
        seas_differenced.append(timeseries)
        timeseries = timeseries.diff(periods=s)
    seas_df = pd.DataFrame(seas_differenced).T

    # General differencing from 0 to d_max
    general_differenced = []
    for j, ts in enumerate(seas_differenced):
        for i in range(1,d_max+1):
            ts = ts.diff()
            ts.name = f"d{i}_D{j}_s{s}"
            general_differenced.append(ts)
    gen_df = pd.DataFrame(general_differenced).T

    # concatenate seasonal and general differencing dataframes
    return pd.concat([seas_df, gen_df], axis=1)

# create the differenced series
diff_series = differencing(co2['co2'], s=12, D_max=2, d_max=2)



# create a summary of test results of all the series
def adf_summary(diff_series):
    summary = []

    for i in diff_series:
        # unpack the results
        a, b, c, d, e, f = adfuller(diff_series[i].dropna())
        g, h, i = e.values()
        results = [a, b, c, d, g, h, i]
        summary.append(results)

    columns = ["Test Statistic", "p-value", "#Lags Used", "No. of Obs. Used",
               "Critical Value (1%)", "Critical Value (5%)", "Critical Value (10%)"]
    index = diff_series.columns
    summary = pd.DataFrame(summary, index=index, columns=columns)
```

```
    return summary
# create the summary
summary = adf_summary(diff_series)

# filter away results that are not stationary
summary_passed = summary[summary["p-value"] < 0.05]

# output indices as a list
index_list = pd.Index.tolist(summary_passed.index)

# use the list as a condition to keep stationary time-series
passed_series = diff_series[index_list].sort_index(axis=1)


PACF, PACF_ci = pacf(passed_series.iloc[:,0].dropna(), alpha=0.05)

df_sp_p = pd.DataFrame() # create an empty dataframe to store values of significant spik
for i in passed_series:
    # unpack the results into PACF and their CI
    PACF, PACF_ci = pacf(passed_series[i].dropna(), alpha=0.05, method='ywm')

    # subtract the upper and lower limits of CI by PACF to centre CI at zero
    PACF_ci_ll = PACF_ci[:,0] - PACF
    PACF_ci_ul = PACF_ci[:,1] - PACF

    # find positions of significant spikes representing possible value of p & P
    sp1 = np.where(PACF < PACF_ci_ll)[0]
    sp2 = np.where(PACF > PACF_ci_ul)[0]

    # PACF values of the significant spikes
    sp1_value = abs(PACF[PACF < PACF_ci_ll])
    sp2_value = PACF[PACF > PACF_ci_ul]

    # store values to dataframe
    sp1_series = pd.Series(sp1_value, index=sp1)
    sp2_series = pd.Series(sp2_value, index=sp2)
    df_sp_p = pd.concat((df_sp_p, sp1_series, sp2_series), axis=1)

# Sort the dataframe by index
df_sp_p = df_sp_p.sort_index()

# visualize sums of values of significant spikes in PACF plots ordered by lag
df_sp_p.iloc[1:].T.sum().plot(kind='bar', title='Candidate AR Terms', xlabel='nth lag',
```



In [7]:

```
df_sp_q = pd.DataFrame()
for i in passed_series:
    # unpack the results into ACF and their CI
    ACF, ACF_ci = acf(passed_series[i].dropna(), alpha=0.05)

    # subtract the upper and lower limits of CI by ACF to centre CI at zero
    ACF_ci_ll = ACF_ci[:,0] - ACF
    ACF_ci_ul = ACF_ci[:,1] - ACF

    # find positions of significant spikes representing possible value of q & Q
    sp1 = np.where(ACF < ACF_ci_ll)[0]
    sp2 = np.where(ACF > ACF_ci_ul)[0]

    # ACF values of the significant spikes
    sp1_value = abs(ACF[ACF < ACF_ci_ll])
    sp2_value = ACF[ACF > ACF_ci_ul]

    # store values to dataframe
    sp1_series = pd.Series(sp1_value, index=sp1)
    sp2_series = pd.Series(sp2_value, index=sp2)
    df_sp_q = pd.concat((df_sp_q, sp1_series, sp2_series), axis=1)

# Sort the dataframe by index
df_sp_q = df_sp_q.sort_index()

# visualize sums of values of significant spikes in ACF plots ordered by lags
df_sp_q.iloc[1:].T.sum().plot(kind='bar', title='Candidate MA Terms', xlabel='nth lag',
```



In [8]:
```
# possible values of the parameters
p = [1, 2, 3]
d = [0, 1]
q = [1, 2]
P = [0, 1]
D = [0, 1, 2]
Q = [0, 1]
s = [12]

# create all combinations of possible values
pdq = list(product(p, d, q))
PDQm = list(product(P, D, Q, s))

print(f"Number of total combinations: {len(pdq)*len(PDQm)}")

warnings.simplefilter("ignore")
def SARIMA_grid(endog, order, seasonal_order):

    # create an empty list to store values
    model_info = []

    #fit the model
    for i in tqdm(order):
        for j in seasonal_order:
            try:
                model_fit = SARIMAX(endog=endog, order=i, seasonal_order=j).fit(disp=Fal
                predict = model_fit.predict()

                # calculate evaluation metrics: MAPE, RMSE, AIC & BIC
                MAPE = (abs((endog-predict)[1:])/(endog[1:])).mean()
                MSE = mse(endog[1:], predict[1:])
                AIC = model_fit.aic
                BIC = model_fit.bic

                # save order, seasonal order & evaluation metrics
                model_info.append([i, j, MAPE, MSE, AIC, BIC])
            except:
                continue

    # create a dataframe to store info of all models
    columns = ["order", "seasonal_order", "MAPE", "MSE", "AIC", "BIC"]
    model_info = pd.DataFrame(data=model_info, columns=columns)
```

```
    return model_info


# create train-test-split
train = co2['co2'].iloc[:int(len(co2)*0.9)]
test = co2['co2'].iloc[int(len(co2)*0.9):]
```

```
Number of total combinations: 144
```

5. Perform a grid search on the model candidates

In [9]:
```
# fit all combinations into the model
model_info = SARIMA_grid(endog=train, order=pdq, seasonal_order=PDQm)
```

```
100%|██████████| 12/12 [05:06<00:00, 25.54s/it]
```

6. Select the best models, based on performance metrics, model complexity, and normality of the residuals.

In [10]:
```
# the best model by each metric
L1 = model_info[model_info.MAPE == model_info.MAPE.min()]
L2 = model_info[model_info.MSE == model_info.MSE.min()]
L3 = model_info[model_info.AIC == model_info.AIC.min()]
L4 = model_info[model_info.BIC == model_info.BIC.min()]

best_models = pd.concat((L1, L2, L3, L4))
best_models
```

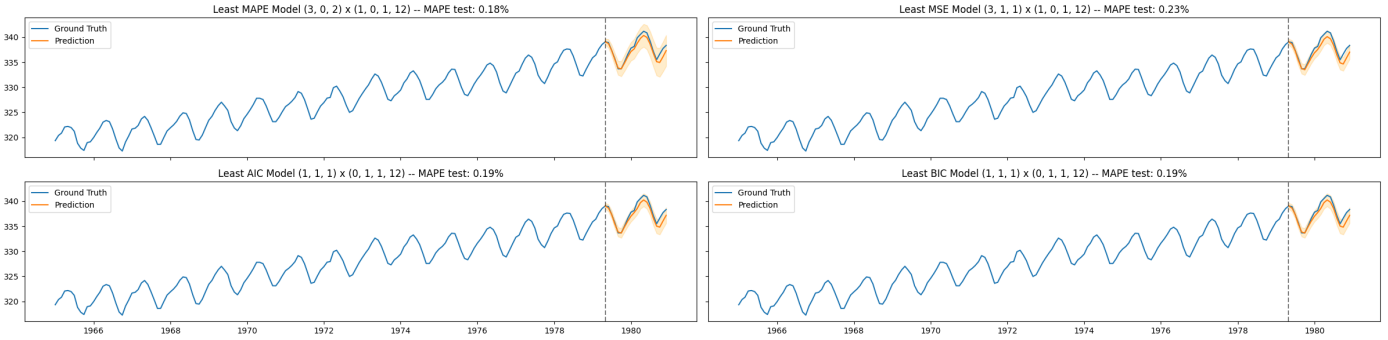
Out[10]:

	order	seasonal_order	MAPE	MSE	AIC	BIC
115	(3, 0, 2)	(1, 0, 1, 12)	0.000927	0.182750	160.460925	185.640881
127	(3, 1, 1)	(1, 0, 1, 12)	0.000936	0.173955	136.187635	158.179280
27	(1, 1, 1)	(0, 1, 1, 12)	0.003808	147.660730	91.094577	103.370193
27	(1, 1, 1)	(0, 1, 1, 12)	0.003808	147.660730	91.094577	103.370193

7. Compare the best model you found with the one from autoarima

In [11]:
```
# Take the parameters of the best models
ord_list = [tuple(best_models.iloc[i,0]) for i in range(best_models.shape[0])]
s_ord_list = [tuple(best_models.iloc[i,1]) for i in range(best_models.shape[0])]
preds, ci_low, ci_up, MAPE_test = [], [], [], []

# Fit the models and compute the forecasts
for i in range(4):
    model_fit = SARIMAX(endog=train, order=ord_list[i],
                        seasonal_order=s_ord_list[i]).fit(disp=False) # Fit the model
    pred_summary = model_fit.get_prediction(test.index[0],
                                            test.index[-1]).summary_frame() # Compute pr
    # Store results
    preds.append(pred_summary['mean'])
    ci_low.append(pred_summary['mean_ci_lower'][test.index])
    ci_up.append(pred_summary['mean_ci_upper'][test.index])
    MAPE_test.append((abs((test-pred_summary['mean'])/(test)).mean()))
```

```
# visualize the results of the fitted models
fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(24,6),
                        sharex=True, sharey=True)

titles = [f'Least MAPE Model {ord_list[0]} x {s_ord_list[0]}',
          f'Least MSE Model {ord_list[1]} x {s_ord_list[1]}',
          f'Least AIC Model {ord_list[2]} x {s_ord_list[2]}',
          f'Least BIC Model {ord_list[3]} x {s_ord_list[3]}']
k = 0
for i in range(2):
    for j in range(2):
        axs[i,j].plot(co2['co2'], label='Ground Truth')
        axs[i,j].plot(preds[k], label='Prediction')
        axs[i,j].set_title(titles[k] + f' -- MAPE test: {MAPE_test[k]:.2%}')
        axs[i,j].legend()
        axs[i,j].axvline(test.index[0], color='black', alpha=0.5, linestyle='--')
        axs[i,j].fill_between(x=test.index, y1=ci_low[k], y2=ci_up[k], color='orange', a

        k += 1
plt.tight_layout()
plt.show()
```



Exercises lecture 6

Exercise

- Download and plot the historical closing prices of Tesla (`TSLA`) and Equinor (`EQNR`) for the years `2019-12-31` - `2022-12-31` .

In [14]:
```
def get_data(tickerSymbol, period, start, end):
    # Get data on the ticker
    tickerData = yf.Ticker(tickerSymbol)
    # Get the historical prices for this ticker
    tickerDf = tickerData.history(period=period, start=start, end=end)
    return tickerDf

# Define the ticker symbols and period
ticker_symbols = ['TSLA', 'EQNR']
start_date = '2019-12-31'
end_date = '2022-12-31'

# Get historical data for each ticker
tesla_data = get_data('TSLA', period='1d', start=start_date, end=end_date)
equinor_data = get_data('EQNR', period='1d', start=start_date, end=end_date)
```

```
# Plotting the Closing Prices
plt.figure(figsize=(14, 5))
plt.plot(tesla_data['Close'], label='Tesla (TSLA) Closing Price')
plt.plot(equinor_data['Close'], label='Equinor (EQNR) Closing Price')
plt.title('Historical Closing Prices (2019-2022)')
plt.xlabel('Date')
plt.ylabel('Price (USD)')
plt.legend()
plt.grid(True)
plt.show()
```

- Test if the time series look stationary

In [16]:
```
# Function to perform ADF test
def perform_adf_test(series, title, regression_type):
    out = adfuller(series, regression=regression_type)
    print(f"Results for {title}:")
    print(f'ADF Statistic: {out[0]:.2f}')
    print(f'p-value: {out[1]:.3f}')
    print(f"Critical Values: {[f'{k}: {r:.2f}' for r,k in zip(out[4].values(), out[4].ke
```

In [17]:
```
# Perform Augmented Dickey-Fuller test
perform_adf_test(tesla_data['Close'], "Tesla (TSLA) Closing Prices", 'ct')
perform_adf_test(equinor_data['Close'], "Equinor (EQNR) Closing Prices", 'ct')
```

```
Results for Tesla (TSLA) Closing Prices:
ADF Statistic: -0.93
p-value: 0.953
Critical Values: ['1%: -3.97', '5%: -3.42', '10%: -3.13']

Results for Equinor (EQNR) Closing Prices:
ADF Statistic: -3.60
p-value: 0.030
Critical Values: ['1%: -3.97', '5%: -3.42', '10%: -3.13']
```

Tesla closing prices are clearly not stationary as the test statistic is greater than the critical values , but for specific thresholds the adf statistic is less than the critical values hence equior is deemed stationary.

- Compute the Hurst coefficient for both time series.

In [18]:
```python
def hurst(ts):

    # Create the range of lag values
    lags = range(2, 100)

    # Calculate the array of the variances of the lagged differences
    tau = [np.sqrt(np.std(np.subtract(ts[lag:], ts[:-lag]))) for lag in lags]

    # Use a linear fit to estimate the Hurst Exponent
    poly = np.polyfit(np.log(lags), np.log(tau), 1)

    # Return the Hurst exponent from the polyfit output
    return poly[0]*2.0
```

In [19]:
```python
# Compute Hurst coefficient for Tesla (TSLA) closing prices
hurst_tesla = hurst(tesla_data['Close'].values)
print(f"Hurst coefficient for Tesla (TSLA) closing prices: {hurst_tesla:.2f}")

# Compute Hurst coefficient for Equinor (EQNR) closing prices
hurst_equinor = hurst(equinor_data['Close'].values)
print(f"Hurst coefficient for Equinor (EQNR) closing prices: {hurst_equinor:.2f}")
```

```
Hurst coefficient for Tesla (TSLA) closing prices: 0.46
Hurst coefficient for Equinor (EQNR) closing prices: 0.33
```

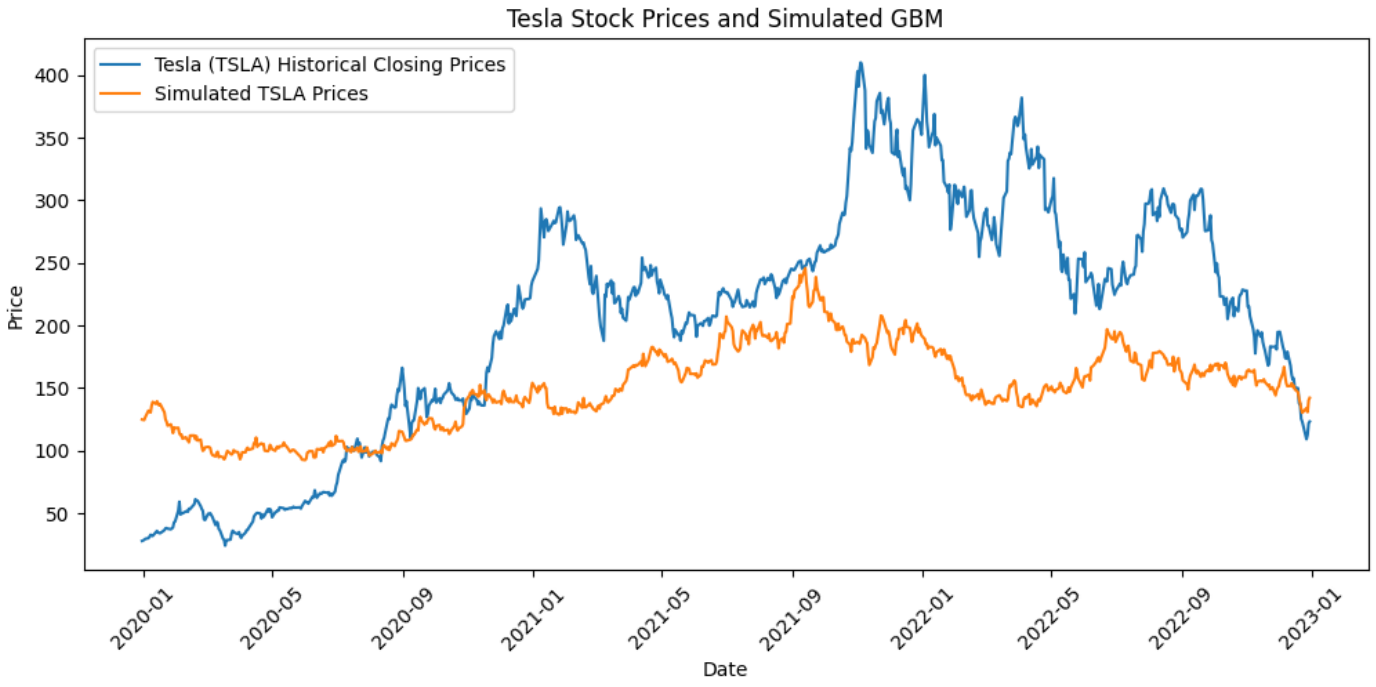
- Which stock would you like to invest into? Motivate your answer based on the tests and the value of H

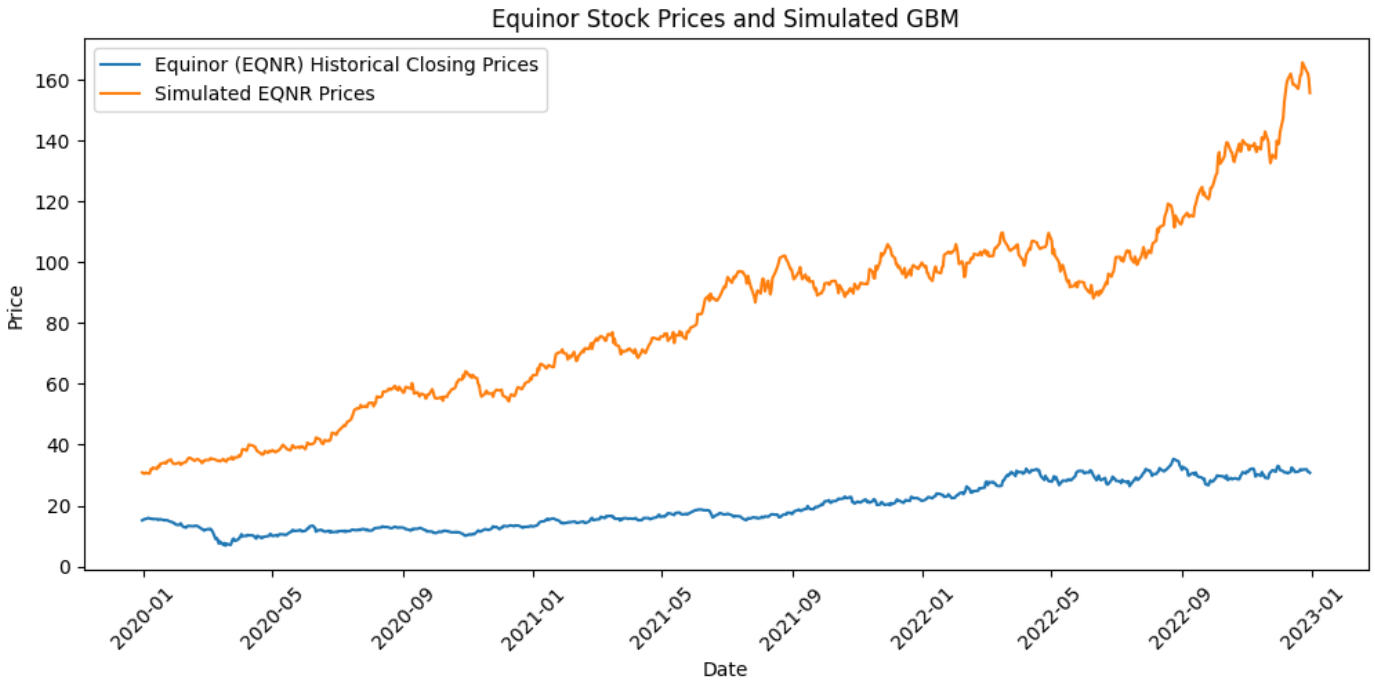
Lower H values suggest mean-reverting behavior, which might be desirable for trading strategies. So it is safer to invest in Equinor than tesla. A Hurst coefficient value close to 0.5 suggests a random walk or no autocorrelation . So a broader market analysis is recommended before investing in Tesla stock.

- Simulate the stock prices using GBM

In [20]:
```python
# Step 1: Get the historical data for Tesla (TSLA) and Equinor (EQNR) done already


# Step 2: Calculate Daily Returns
tesla_returns = tesla_data['Close'].pct_change()
equinor_returns = equinor_data['Close'].pct_change()

# Step 3: Estimate Parameters for GBM
tesla_mu = tesla_returns.mean() * 252  # Annualize the mean
tesla_sigma = tesla_returns.std() * np.sqrt(252)  # Annualize the std deviation
equinor_mu = equinor_returns.mean() * 252  # Annualize the mean
equinor_sigma = equinor_returns.std() * np.sqrt(252)  # Annualize the std deviation

# Step 4: Set GBM Parameters
T = 1  # Time horizon in years
dt = 1 / len(tesla_returns)  # Time step in years
N = len(tesla_returns)  # Number of time steps
tesla_S0 = tesla_data['Close'].iloc[-1]  # Starting stock price (latest close price)
equinor_S0 = equinor_data['Close'].iloc[-1]  # Starting stock price (latest close price)

# Step 5: Compute Simulation for Tesla (TSLA)
tesla_W = np.random.standard_normal(size=N)
tesla_W = np.cumsum(tesla_W) * np.sqrt(dt)  # Cumulative sum for the Wiener process
tesla_X = (tesla_mu - 0.5 * tesla_sigma ** 2) * np.linspace(0, T, N) + tesla_sigma * tes
tesla_S = tesla_S0 * np.exp(tesla_X)  # GBM formula
```

```python
# Step 6: Compute Simulation for Equinor (EQNR)
equinor_W = np.random.standard_normal(size=N)
equinor_W = np.cumsum(equinor_W) * np.sqrt(dt)  # Cumulative sum for the Wiener process
equinor_X = (equinor_mu - 0.5 * equinor_sigma ** 2) * np.linspace(0, T, N) + equinor_sig
equinor_S = equinor_S0 * np.exp(equinor_X)  # GBM formula

# Plot the results
plt.figure(figsize=(12, 5))
plt.plot(tesla_data['Close'], label='Tesla (TSLA) Historical Closing Prices')
plt.plot(tesla_data.index, tesla_S, label='Simulated TSLA Prices')
plt.legend()
plt.title('Tesla Stock Prices and Simulated GBM')
plt.xlabel('Date')
plt.ylabel('Price')
plt.xticks(rotation=45)
plt.show()

plt.figure(figsize=(12, 5))
plt.plot(equinor_data['Close'], label='Equinor (EQNR) Historical Closing Prices')
plt.plot(equinor_data.index, equinor_S, label='Simulated EQNR Prices')
plt.legend()
plt.title('Equinor Stock Prices and Simulated GBM')
plt.xlabel('Date')
plt.ylabel('Price')
plt.xticks(rotation=45)
plt.show()
```





- Which simulation seems to be more reliable? The one for Tesla or Equinor?
- To motivate your answer:
  1. compute the simulation at least 100 times.
  2. Compute the MSE between the true stock prices and the simulated ones.
  3. Compare the expected value of the MAPE for the two stocks.

In [21]:

```
def compute_mse(true_prices, simulated_prices):
    return np.mean((true_prices - simulated_prices) ** 2)

def compute_mape(true_prices, simulated_prices):
    return np.mean(np.abs((true_prices - simulated_prices) / true_prices)) * 100


# Define the number of simulations
num_simulations = 100

# Initialize arrays to store MSE and MAPE values for each simulation
tesla_mse_values = []
tesla_mape_values = []
equinor_mse_values = []
equinor_mape_values = []

# Perform simulations and compute MSE and MAPE for each simulation
for _ in range(num_simulations):
    # Simulate Tesla stock prices
    tesla_W = np.random.standard_normal(size=N)
    tesla_W = np.cumsum(tesla_W) * np.sqrt(dt)
    tesla_X = (tesla_mu - 0.5 * tesla_sigma ** 2) * np.linspace(0, T, N) + tesla_sigma *
    tesla_simulated_prices = tesla_S0 * np.exp(tesla_X)

    # Simulate Equinor stock prices
    equinor_W = np.random.standard_normal(size=N)
    equinor_W = np.cumsum(equinor_W) * np.sqrt(dt)
    equinor_X = (equinor_mu - 0.5 * equinor_sigma ** 2) * np.linspace(0, T, N) + equinor
    equinor_simulated_prices = equinor_S0 * np.exp(equinor_X)

    # Compute MSE and MAPE for Tesla
    tesla_mse = compute_mse(tesla_data['Close'], tesla_simulated_prices)
```

```
    tesla_mse_values.append(tesla_mse)
    tesla_mape = compute_mape(tesla_data['Close'], tesla_simulated_prices)
    tesla_mape_values.append(tesla_mape)

    # Compute MSE and MAPE for Equinor
    equinor_mse = compute_mse(equinor_data['Close'], equinor_simulated_prices)
    equinor_mse_values.append(equinor_mse)
    equinor_mape = compute_mape(equinor_data['Close'], equinor_simulated_prices)
    equinor_mape_values.append(equinor_mape)

# Compute the mean MSE and MAPE values for Tesla and Equinor
mean_tesla_mse = np.mean(tesla_mse_values)
mean_tesla_mape = np.mean(tesla_mape_values)
mean_equinor_mse = np.mean(equinor_mse_values)
mean_equinor_mape = np.mean(equinor_mape_values)
print("Mean MSE for Tesla:", mean_tesla_mse)
print("Mean MAPE for Tesla:", mean_tesla_mape)
print("Mean MSE for Equinor:", mean_equinor_mse)
print("Mean MAPE for Equinor:", mean_equinor_mape)
```

```
Mean MSE for Tesla: 19027.62410096756
Mean MAPE for Tesla: 69.32554205267061
Mean MSE for Equinor: 498.33090396676306
Mean MAPE for Equinor: 117.13637540869213
```

It appears that Equinor has a significantly lower mean MSE compared to Tesla, indicating that the simulated prices for Equinor are closer to the true historical prices on average, suggesting better performance of the simulation model for Equinor stock in terms of MSE with not having a significant difference in mean MAPE values.

Mean MAPE for Tesla: 69.29319650738705 , Mean MAPE for Equinor: 107.35020408335113

Exercises lecture 7

Exercise

This exercise refers to Example 1 in lecture 7: static one-dimensional data.

A. Choose a value for the estimated model variance `Q_est` that is larger than estimated measurement variance `R_est` .

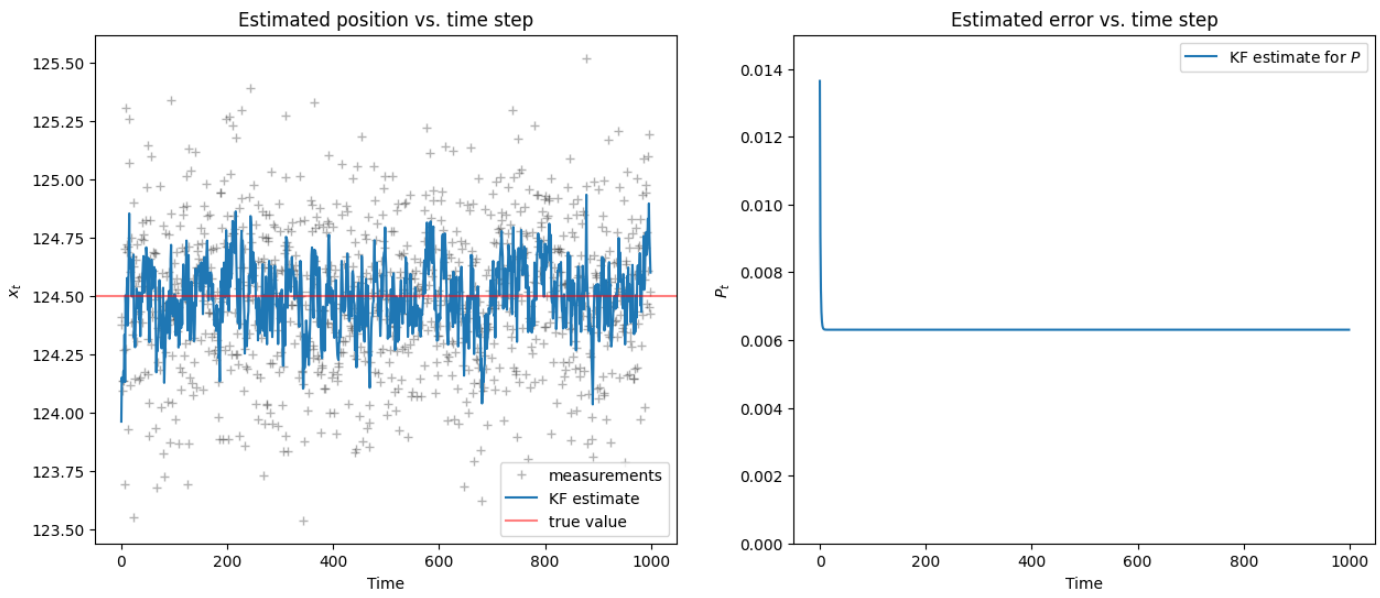
B. Repeat the analysis of Example 1 for this new value.

C. Does the KF estimate converge?

D. Why the estimates changed? Do they look more noisy than before? Why?

In [23]:
```
# Generate measurements
n_measurements = 1000
mu = 124.5 # Actual position
R = 0.1    # Actual standard deviation of actual measurements (R)

Z = np.random.normal(mu, np.sqrt(R), size=n_measurements)

# Estimated covariances
```

```
Q_est = 29e-4
R_est = 2e-2

def kalman_1d(x, P, measurement, R_est, Q_est):

    # Prediction
    x_pred = x
    P_pred = P + Q_est

    # Update
    K = P_pred / (P_pred + R_est)
    x_est = x_pred + K * (measurement - x_pred)
    P_est = (1 - K) * P_pred

    return x_est, P_est


# initial guesses
x = 123 # Use an integer (imagine the initial guess is determined with a meter stick)
P = 0.04 # error covariance P

KF_estimate=[] # To store the position estimate at each time point
KF_error=[] # To store estimated error at each time point
for z in Z:
    x, P = kalman_1d(x, P, z, R_est, Q_est)
    KF_estimate.append(x)
    KF_error.append(P)




def plot_1d_comparison(measurements_made, estimate, true_value, axis):
    axis.plot(measurements_made,'k+',label='measurements',alpha=0.3)
    axis.plot(estimate,'-',label='KF estimate')
    if not isinstance(true_value, (list, tuple, np.ndarray)):
        # plot line for a constant value
        axis.axhline(true_value,color='r',label='true value', alpha=0.5)
    else:
        # for a list, tuple or array, plot the points
        axis.plot(true_value,color='r',label='true value', alpha=0.5)
    axis.legend(loc = 'lower right')
    axis.set_title('Estimated position vs. time step')
    axis.set_xlabel('Time')
    axis.set_ylabel('$x_t$')

def plot_1d_error(estimated_error, lower_limit, upper_limit, axis):
    # lower_limit and upper_limit are the lower and upper limits of the vertical axis
    axis.plot(estimated_error, label='KF estimate for $P$')
    axis.legend(loc = 'upper right')
    axis.set_title('Estimated error vs. time step')
    axis.set_xlabel('Time')
    axis.set_ylabel('$P_t$')
    plt.setp(axis,'ylim',[lower_limit, upper_limit])

fig, axes = plt.subplots(1,2, figsize=(15, 6))
plot_1d_comparison(Z, KF_estimate, mu, axes[0])
plot_1d_error(KF_error, 0, 0.015, axes[1])
```



If Q is significantly larger , it causes the Kalman Filter estimate to converge more slowly or even diverge as shown in the figure. This is because the filter places more weight on the model predictions, which may not align well with the actual measurements if the model is not accurate. The estimates may also exhibit more oscillations or instability due to the increased reliance on the model predictions. The estimates may appear more noisy than before because the larger Q introduces more variability from the model predictions. This increased variability can lead to less stable and more erratic estimates over time.

Exercise

This exercise refers to Example 2: Dynamic one-dimensional data. In the case we examined above, the KF estimate was close to the measurements and both were different from the true value. Change the parameters of the algorithm until you find some combinations that achieve the following:

A. Measurements, KF estimate and true value are all close.

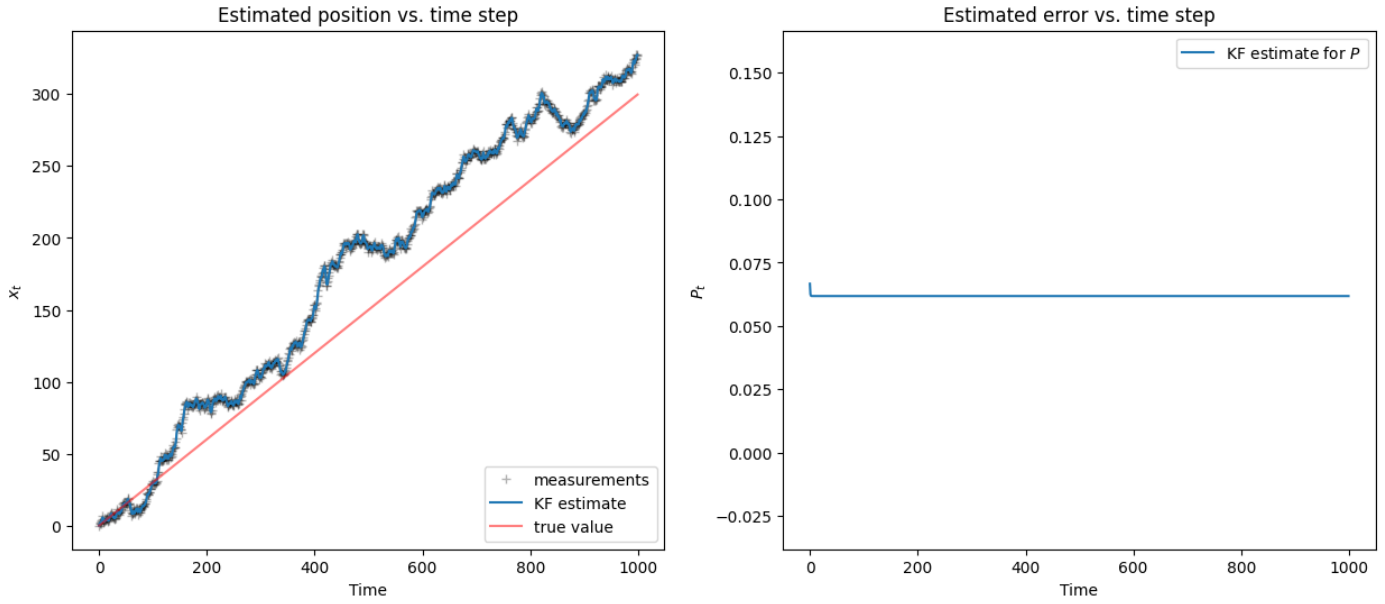
B. Measurements, KF estimate and true value are all noticeably different.

C. The measurements are close to the true value, but the KF estimate is different.

Discuss your findings.

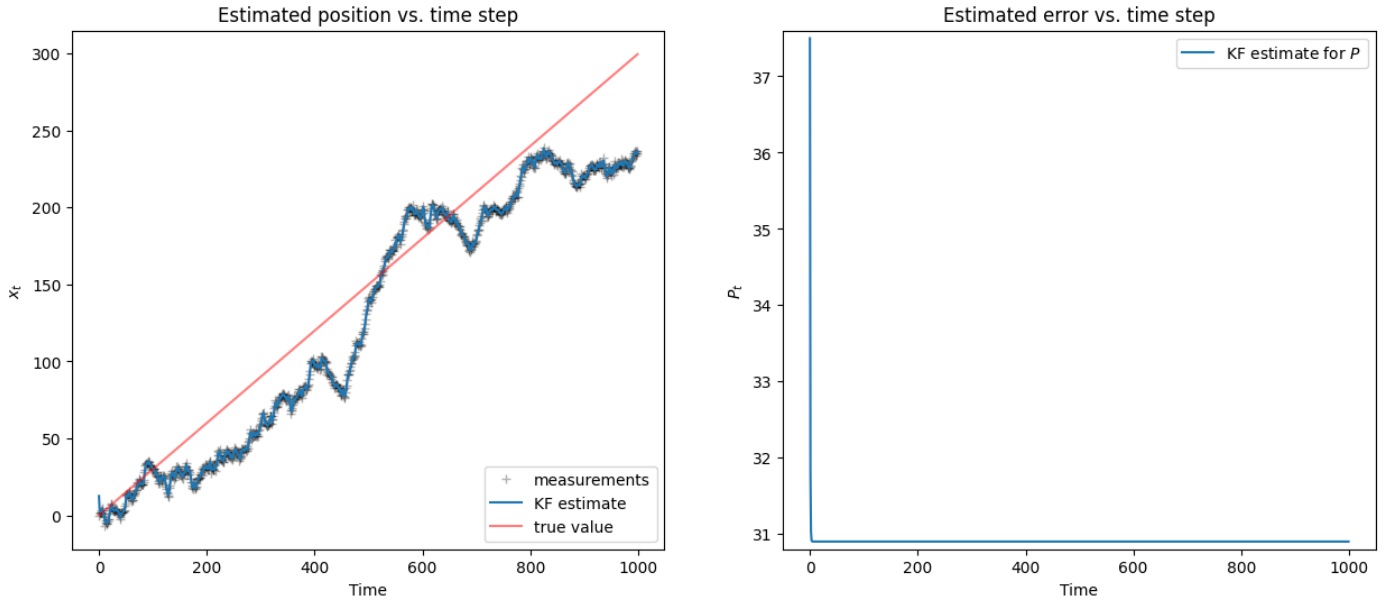
In [24]:
```
# initial parameters
v0 = 0.3
x0 = 0.0
R = 4.0

# generate noisy measurements
n_measurements = 1000
Zv = np.zeros(n_measurements) # velocity measurements
Zx = np.zeros(n_measurements) # position measurements
for t in range(0, n_measurements-1):
    Zv[t] = np.random.normal(v0, np.sqrt(R))
    Zx[t+1] = Zx[t] + Zv[t] * 1 # delta_t = 1

# generate true positions
Xt = np.zeros(n_measurements)
for t in range(0, n_measurements):
    Xt[t]= x0 + v0*t
```

```
# For Scenario A: Measurements, KF estimate, and true value are all close
# Set initial guess close to true value
x = 0
P = 0.1 # Small uncertainty
Q_est = 0.1   #   Small model noise
R_est = 0.1    #  Small measurement noise


KF_estimate = [] # To store the position estimate at each time point
KF_error = [] # To store estimated error at each time point

# Kalman filter
for z in Zx:
    x, P = kalman_1d(x, P, z, R_est, Q_est)
    KF_estimate.append(x)
    KF_error.append(P)

fig, axes = plt.subplots(1,2, figsize=(15, 6))
plot_1d_comparison(Zx, KF_estimate, Xt, axes[0])
plot_1d_error(KF_error, min(KF_error)-0.1, max(KF_error)+0.1, axes[1])
```



In [27]:
```
# initial parameters
v0 = 0.3
x0 = 0.0
R = 4.0

# generate noisy measurements
n_measurements = 1000
Zv = np.zeros(n_measurements) # velocity measurements
Zx = np.zeros(n_measurements) # position measurements
for t in range(0, n_measurements-1):
    Zv[t] = np.random.normal(v0, np.sqrt(R))
    Zx[t+1] = Zx[t] + Zv[t] * 1 # delta_t = 1

# generate true positions
Xt = np.zeros(n_measurements)
for t in range(0, n_measurements):
    Xt[t]= x0 + v0*t


# scenario B initial guesses and estimates
```

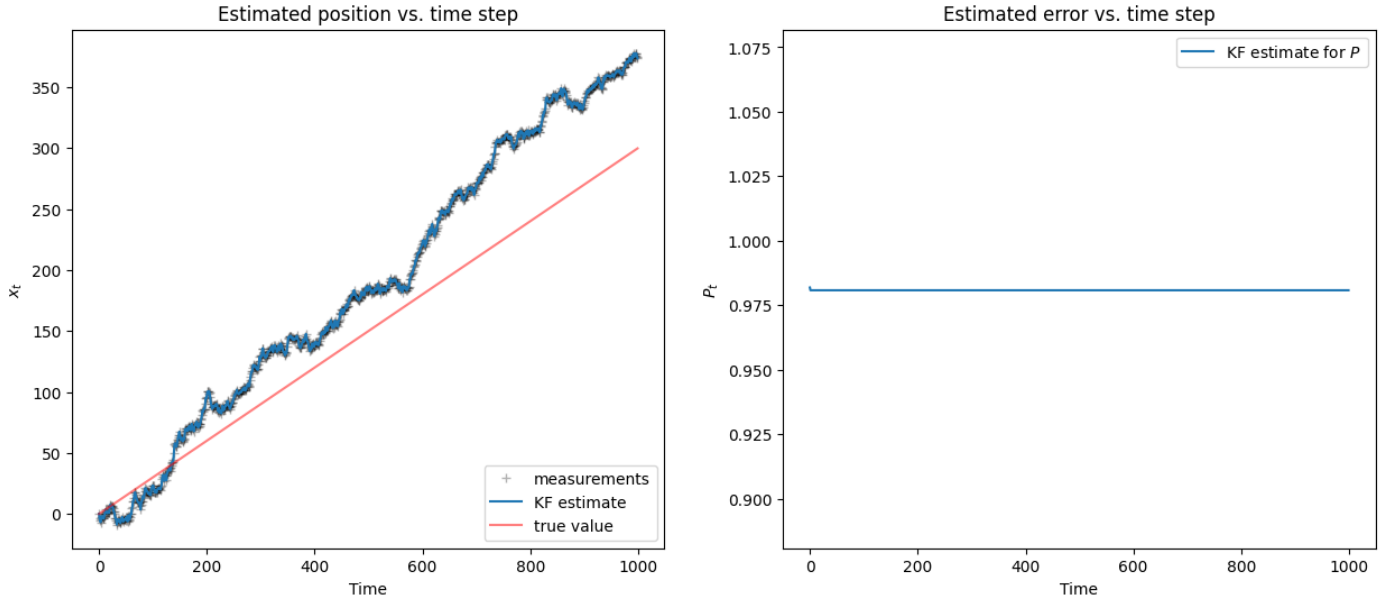
```
x = 50  # Set initial guess and uncertainty far from true value
P = 100
Q_est = 50   # large model noise
R_est = 50     # Large measurement noise


KF_estimate = [] # To store the position estimate at each time point
KF_error = [] # To store estimated error at each time point

# Kalman filter
for z in Zx:
    x, P = kalman_1d(x, P, z, R_est, Q_est)
    KF_estimate.append(x)
    KF_error.append(P)

fig, axes = plt.subplots(1,2, figsize=(15, 6))
plot_1d_comparison(Zx, KF_estimate, Xt, axes[0])
plot_1d_error(KF_error, min(KF_error)-0.1, max(KF_error)+0.1, axes[1])
```



In [28]:

```
# initial parameters
v0 = 0.3
x0 = 0.0
R = 4.0

# generate noisy measurements
n_measurements = 1000
Zv = np.zeros(n_measurements) # velocity measurements
Zx = np.zeros(n_measurements) # position measurements
for t in range(0, n_measurements-1):
    Zv[t] = np.random.normal(v0, np.sqrt(R))
    Zx[t+1] = Zx[t] + Zv[t] * 1 # delta_t = 1

# generate true positions
Xt = np.zeros(n_measurements)
for t in range(0, n_measurements):
    Xt[t]= x0 + v0*t


# For Scenario C: Measurements are close to true value, but KF estimate is different
# Set initial guess and uncertainty not close to true value
x = 10
P = 4
Q_est = 50    # Increase model noise to make KF estimate different from true value
```

```
R_est = 1     # Keep measurement noise small


KF_estimate = [] # To store the position estimate at each time point
KF_error = [] # To store estimated error at each time point

# Kalman filter
for z in Zx:
    x, P = kalman_1d(x, P, z, R_est, Q_est)
    KF_estimate.append(x)
    KF_error.append(P)

fig, axes = plt.subplots(1,2, figsize=(15, 6))
plot_1d_comparison(Zx, KF_estimate, Xt, axes[0])
plot_1d_error(KF_error, min(KF_error)-0.1, max(KF_error)+0.1, axes[1])
```



To achieve the specified scenarios, we need to adjust the parameters of the Kalman Filter algorithm. Here's how we can approach each scenario:

A. Set the initial guess x to be close to the true value. small values for P, Q_est, and R_est that result in a stable and accurate estimation (without introducing too much noise).

B. Introduce significant noise in the measurements by increasing R_est. S the initial guess x and P to values that are far from the true value and have a large uncertainty along with large model noise Q.

C. Q_est to a relatively large value to increase the influence of the model predictions. R_est small to maintain accuracy in measurements. x and P to values that are not close to the true value and have a small uncertainty.

Exercise #3

This exercise refers to Example 3: Dynamic two-dimensional data.

In the original example, we used both Position and Velocity (PV model) in the state vector, i.e., $\boldsymbol{x} = [\, x \quad y \quad \dot{x} \quad \dot{y} \,]^T$.

What happens if we use only the Position (P model) to describe the state?

After all, our measurements only provide position.

Do we really need to include the velocity?

Rewrite the algorithm above for the P model.

> Hint: what is the size of the state vector in this case? What are the dimensions of the matrices that characterize the system?

When using only the position (P model) to describe the state instead of both position and velocity (PV model), the dynamics of the system become simplified. The Kalman Filter estimates the position based solely on the measurements and the transition model. This is one of the advantages of kalman filter as it is possible to operate it under partial measurements. By using only the position, we lose information about the velocity of the system. In some cases, velocity information can be useful for predicting future states, especially if the system undergoes changes in speed or acceleration. Without velocity information, the filter relies solely on the transition model to predict the next position.

!!Reason for commenting the algorithm below is because of unavailability of open-CV pacakge (installation problem) in the system.

In [31]:
```
""" Initialize Kalman Filter
 kalman = cv2.KalmanFilter(2, 2, 0)  # 2 states (x and y), 2 measurements, 0 control vec


q = 1  # variance in the model
r = 20  # variance in the measurement
dtime = 1  # size of time step

# Measurement matrix (H)
kalman.measurementMatrix = np.array([[1, 0],
                                     [0, 1]], np.float32)

# Transition matrix (A)
kalman.transitionMatrix = np.array([[1, 0],
                                    [0, 1]], np.float32)

# Process noise covariance matrix (Q)
kalman.processNoiseCov = np.array([[1, 0],
                                   [0, 1]], np.float32) * q

# Measurement noise covariance matrix (R)
kalman.measurementNoiseCov = np.array([[1, 0],
                                       [0, 1]], np.float32) * r

KF_estimate_xy = []  # To store the position estimate at each time point

# Load precomputed data
xy_motion = np.genfromtxt('xy_motion_kalman_filter_example.csv', dtype='float32', delimi

for i in xy_motion:
    pred = kalman.predict()  # Predict new state using the model
    kalman.correct((i))  # Update estimated state with the measurement
    KF_estimate_xy.append((pred[0], pred[1]))  # Store the estimated position

x_est, y_est = zip(*KF_estimate_xy)
x_true, y_true = zip(*xy_motion)

plt.scatter(x_est, y_est, marker='.', label='KF estimate', alpha=0.5)
plt.scatter(x_true, y_true, marker='.', label='true value', alpha=0.5)
```

```
plt.legend(loc='lower center')
plt.title('2D position (P model)')
plt.xlabel('x coordinate')
plt.ylabel('y coordinate')
plt.show() """
```

Out[31]:
```
" Initialize Kalman Filter\n kalman = cv2.KalmanFilter(2, 2, 0)  # 2 states (x and y), 2
measurements, 0 control vector\n\n\nq = 1  # variance in the model\nr = 20  # variance i
n the measurement\ndtime = 1  # size of time step\n\n# Measurement matrix (H)\nkalman.me
asurementMatrix = np.array([[1, 0],\n                                        [0, 1]], np.fl
oat32)\n\n# Transition matrix (A)\nkalman.transitionMatrix = np.array([[1, 0],\n
                          [0, 1]], np.float32)\n\n# Process noise covariance matrix
(Q)\nkalman.processNoiseCov = np.array([[1, 0],\n                                    [0,
1]], np.float32) * q\n\n# Measurement noise covariance matrix (R)\nkalman.measurementNoi
seCov = np.array([[1, 0],\n                                      [0, 1]], np.float32) *
r\n\nKF_estimate_xy = []  # To store the position estimate at each time point\n\n# Load
precomputed data\nxy_motion = np.genfromtxt('xy_motion_kalman_filter_example.csv', dtype
='float32', delimiter=',')\n\nfor i in xy_motion:\n    pred = kalman.predict()  # Predic
t new state using the model\n    kalman.correct((i))  # Update estimated state with the
measurement\n    KF_estimate_xy.append((pred[0], pred[1]))  # Store the estimated positi
on\n\nx_est, y_est = zip(*KF_estimate_xy)\nx_true, y_true = zip(*xy_motion)\n\nplt.scatt
er(x_est, y_est, marker='.', label='KF estimate', alpha=0.5)\nplt.scatter(x_true, y_tru
e, marker='.', label='true value', alpha=0.5)\nplt.legend(loc='lower center')\nplt.title
('2D position (P model)')\nplt.xlabel('x coordinate')\nplt.ylabel('y coordinate')\nplt.s
how() "
```

Exercises lecture 8

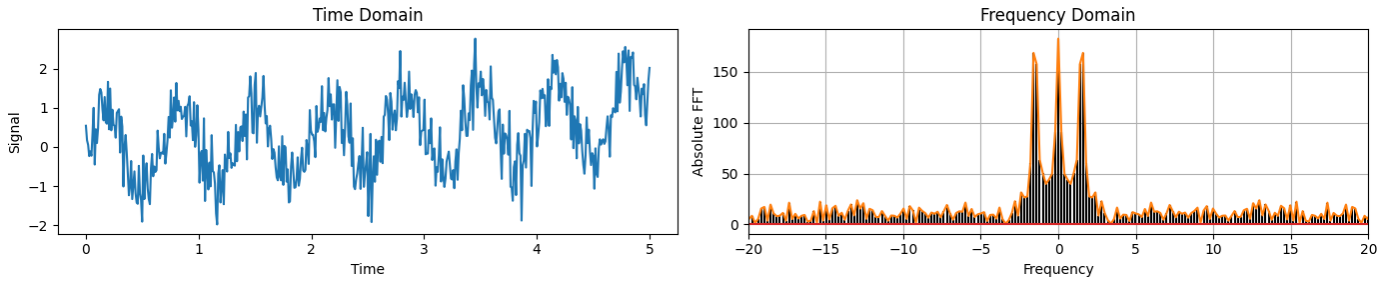
Exercise #1

- Add the three signals together. Observe the time and frequency domain components.
- Modify the amplitude of the sine, trend, and noise components so that each component, in turn, dominates over the others. Comment on how the FT of the total signal changes.
- Modify the sine into a signal that is the sum of 2 sine waves of different amplitudes at 1 and 10 Hz as well as a constant term. Make a plot in both time and frequency domains and comment the results.

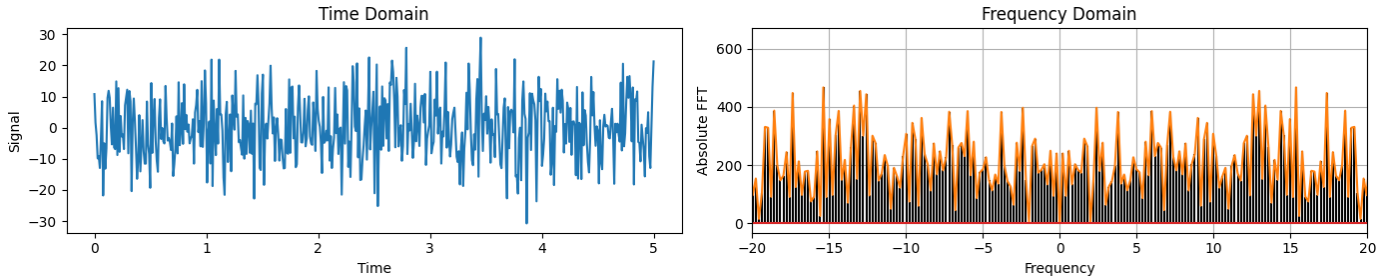
In [32]:
```
# Convenience function that creates both a time domain and frequency domain plot.
def plot_time_freq(t, y):
    # Converts Data into Frequncy Domain
    freq = np.fft.fftfreq(t.size, d=t[1]-t[0])
    Y = abs(np.fft.fft(y))

    # Time domain plot
    plt.figure(figsize = [14,3])
    plt.subplot(1,2,1)
    plt.plot(t,y)
    plt.title('Time Domain')
    plt.xlabel('Time')
    plt.ylabel('Signal')

    # Frequency domain plot
    plt.subplot(1,2,2)
    markerline, stemline, baseline = plt.stem(np.fft.fftshift(freq),np.fft.fftshift(Y),
                                              'k', markerfmt='tab:orange')
    plt.setp(stemline, linewidth = 1.5)
    plt.setp(markerline, markersize = 4)
    plt.title('Frequency Domain')
```

```
    plt.xlabel('Frequency')
    plt.xlim(-20, 20)
    plt.ylabel('Absolute FFT')
    plt.grid()

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

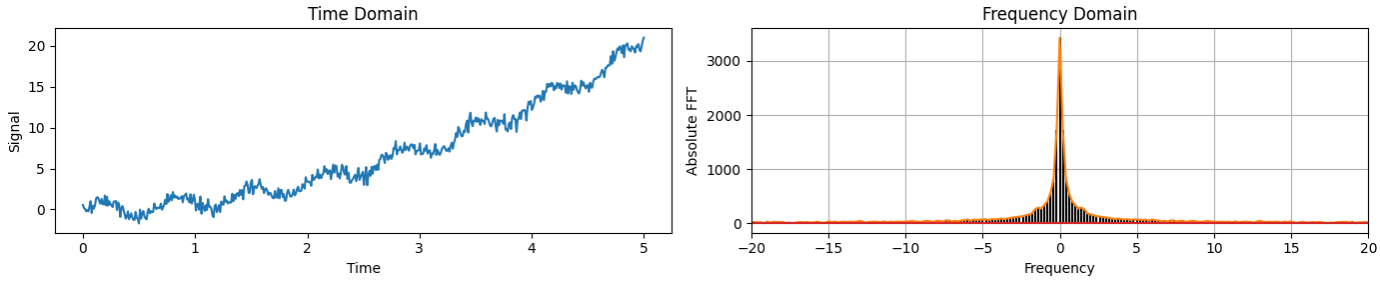
In [33]:
```
time = np.linspace(0, 5, 512)
freq = 1.5

y_sine = np.sin(2 * np.pi * freq * time)
y_noise = 0.5 * np.random.randn(len(time))
y_trend = (0.2 * time)**2

# Add all components together
y_combined = y_sine + y_trend + y_noise

# Plot time and frequency representations for the combined signal
plot_time_freq(time, y_combined)
```



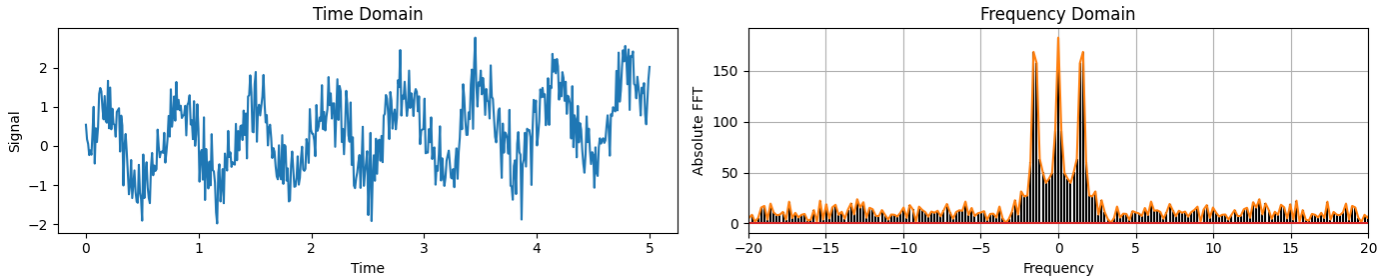
In [34]:
```
# Generate random noise
y_noise_n = 20 * y_noise  # Increase noise amplitude

# Add all components together
y_combined_n = y_sine + y_trend + y_noise_n


# Plot time and frequency representations for the combined signal
plot_time_freq(time, y_combined_n)
```



In [35]:
```
# Calculate trend component
y_trend_n = 20 * y_trend  # Increase trend amplitude



# Add all components together
y_combined_ne = y_sine + y_trend_n + y_noise


# Plot time and frequency representations for the combined signal
plot_time_freq(time, y_combined_ne)
```

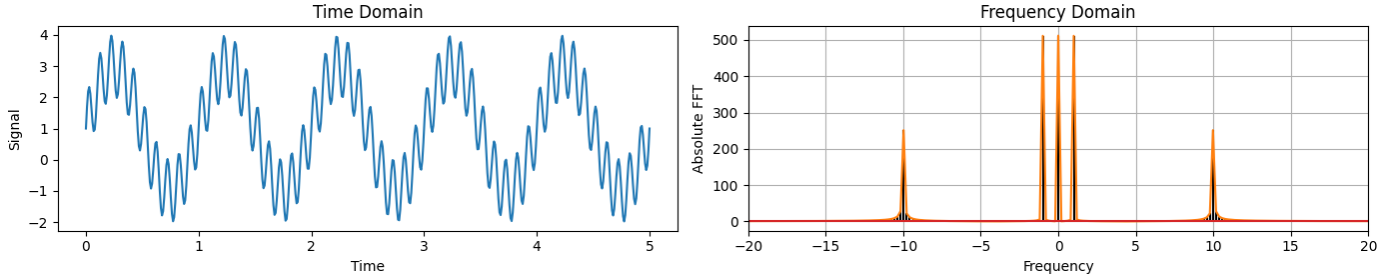


In [36]:
```
y_sine_n = 25 * y_sine  # Increase sine wave amplitude


# Add all components together
y_combined_new = y_sine + y_trend + y_noise


# Plot time and frequency representations for the combined signal
plot_time_freq(time, y_combined_new)
```



Now, each component dominates in turn, and you can observe the changes in the Fourier Transform of the total signal accordingly. You'll see how each dominant component influences the frequency domain representation of the total signal.

In [37]:
```
# Define frequencies and amplitudes
freq1 = 1  # 1 Hz
freq2 = 10  # 10 Hz
amp1 = 2
amp2 = 1
constant = 1

# Calculate signals
y_sine1 = amp1 * np.sin(2 * np.pi * freq1 * time)
y_sine2 = amp2 * np.sin(2 * np.pi * freq2 * time)
y_constant = constant * np.ones_like(time)

# Sum of signals
y_combined = y_sine1 + y_sine2 + y_constant

# Plot time and frequency representations
plot_time_freq(time, y_combined)
```



Exercise #2

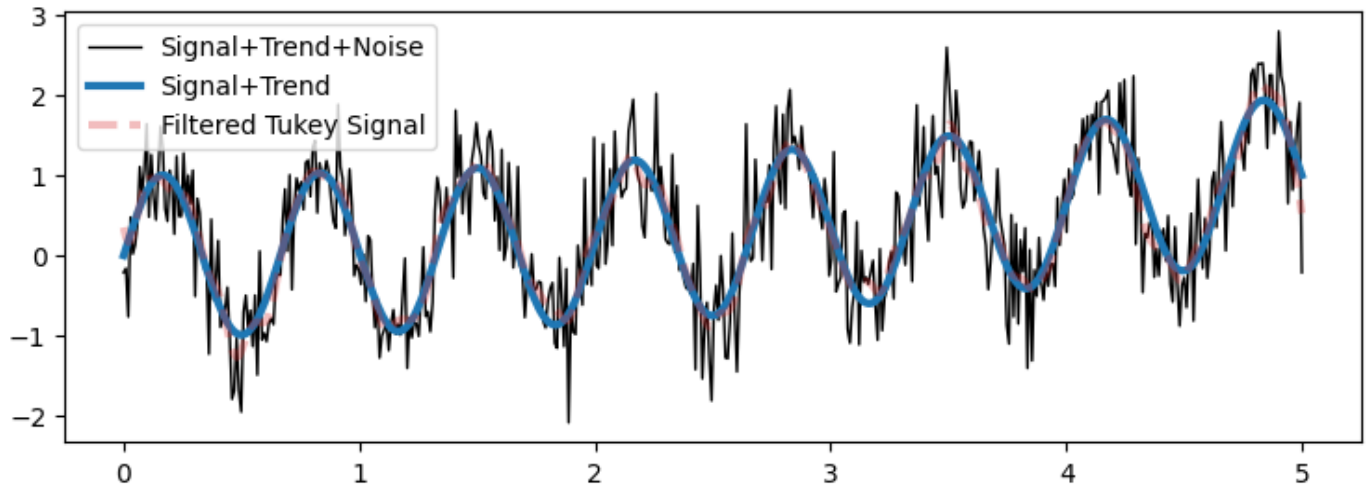
Modify the values of `alpha` and `div_factor` to optimize the Tukey filter.

In [38]:
```
# Let's create a function to show the results.
def filter_plot(time, y_noisy, y_clean, y_filtered, legend_names, alpha=1):
    plt.figure(figsize=[9,3])
    plt.plot(time, y_noisy, 'k', lw=1)
    plt.plot(time, y_clean, 'tab:blue', lw=3)
    plt.plot(time, np.real(y_filtered), 'tab:red', linestyle='--', lw=3, alpha=alpha)
    plt.legend(legend_names);
```

In [39]:
```
time = np.linspace(0, 5, 512)
freq = 1.5
y_sine = np.sin(2 * np.pi * freq * time)
y_trend = (0.2 * time)**2
y_noise = 0.5 * np.random.randn(len(time))
noisy_signal = y_sine + y_trend + y_noise
```

In [42]:
```
# Filter's parameters
alpha = 2  # Adjusted alpha
div_factor = 32  # Adjusted div_factor
win_len = int(len(time) / div_factor)
print(f"Window length: {win_len}")

# Compute window
window = signal.windows.tukey(win_len, alpha=alpha)

# Compute frequency response
response = np.fft.fft(window, len(time))
response = np.abs(response / abs(response).max())

# Apply filter
Y = np.fft.fft(noisy_signal)
y_tukey = np.fft.ifft(Y * response)

filter_plot(time, noisy_signal, y_sine + y_trend, y_tukey, ['Signal+Trend+Noise', 'Signa
plt.show()
```

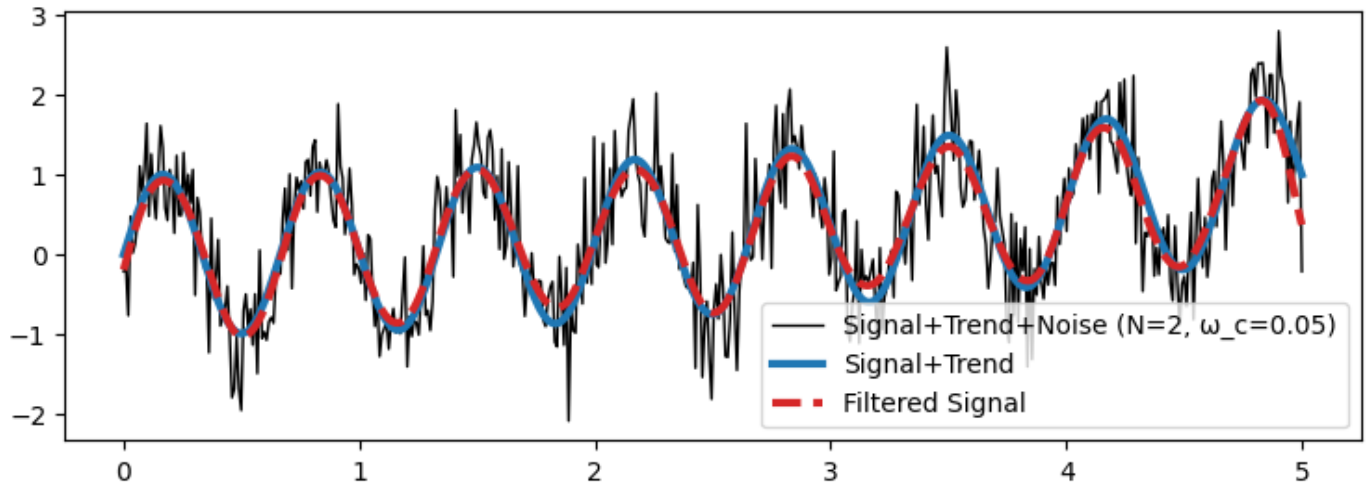
```
Window length: 16
```

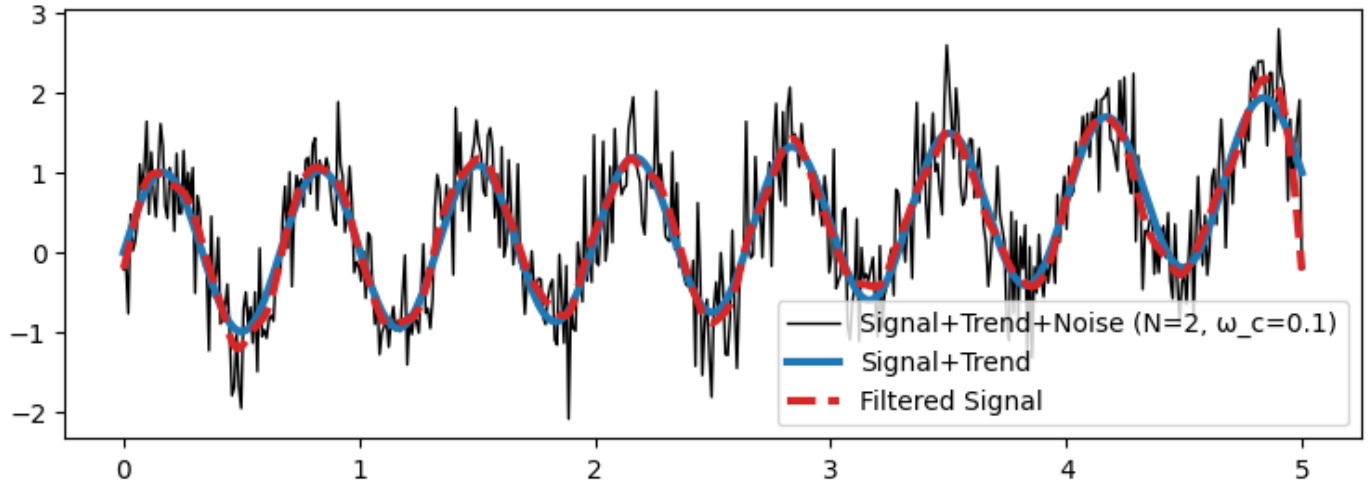


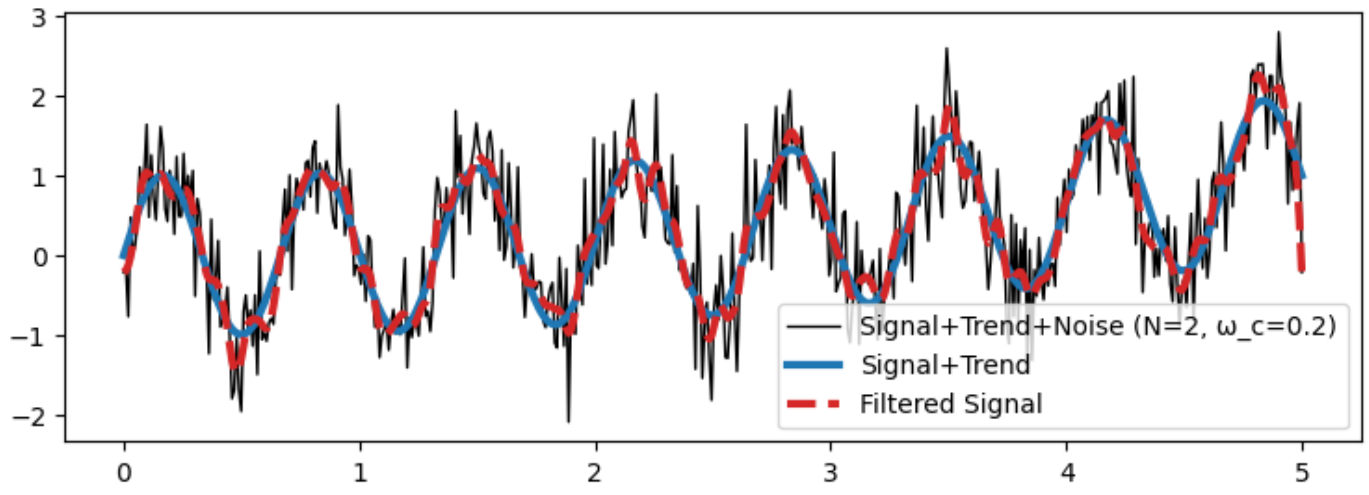
Exercise #3

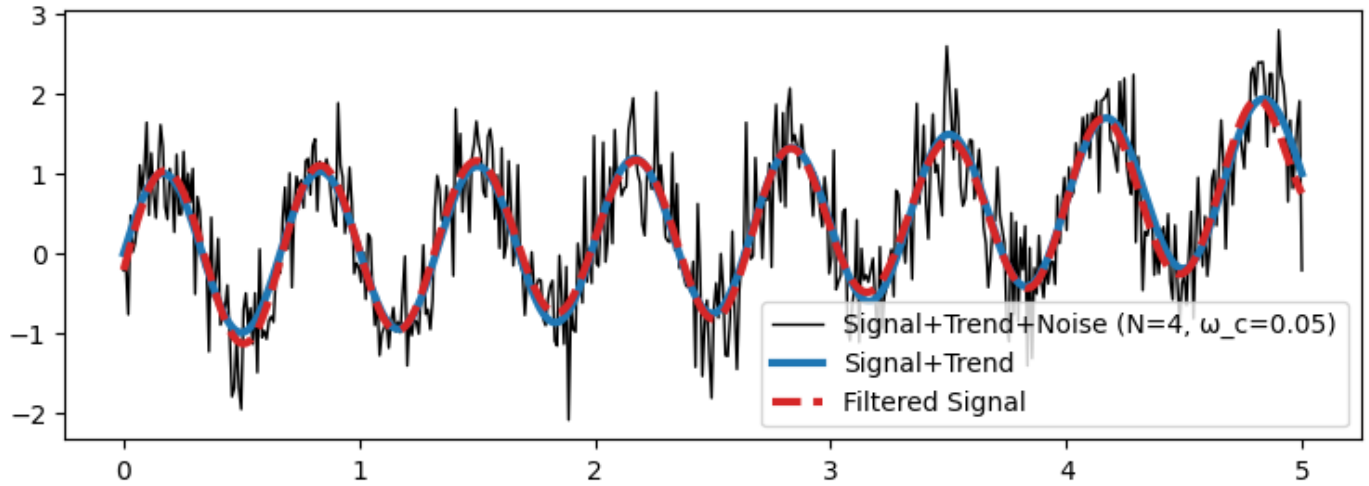
- Modify the filter order N and cutoff frequency ω_c of the Butterworth filter.
- Which values seem to be the best in getting rid of the noise?

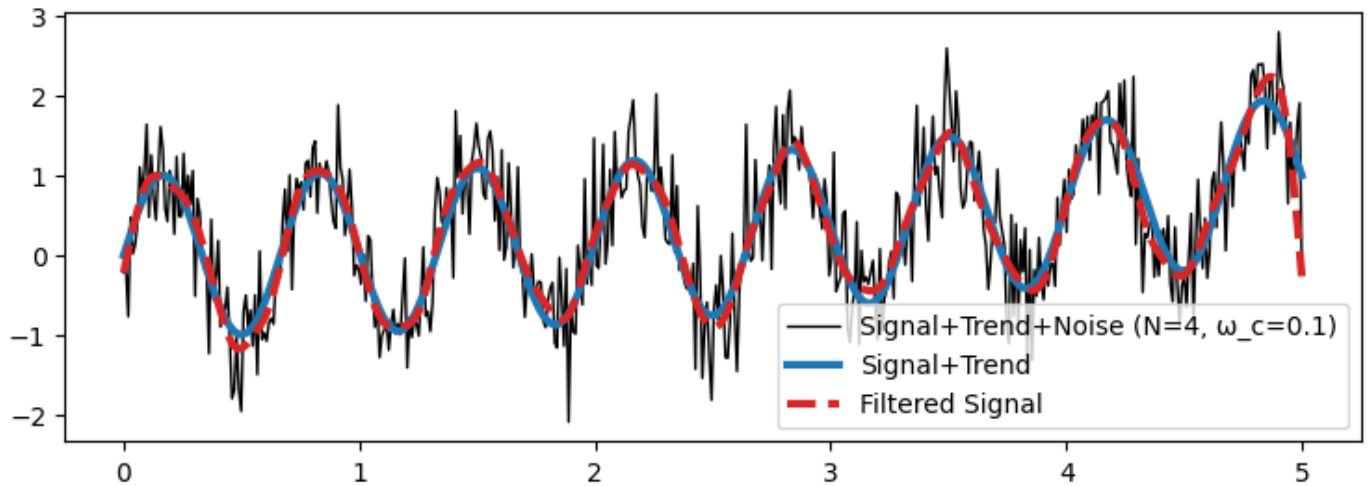
In [43]:
```
# Define different values for N and omega_c to test
orders = [2, 4, 6]  # Filter orders
cutoff_frequencies = [0.05, 0.1, 0.2]  # Cutoff frequencies

for N in orders:
    for omega_c in cutoff_frequencies:
        # Design the Butterworth filter
        B, A = signal.butter(N=N, Wn=omega_c, btype='lowpass')

        # Apply the filter
        y_low_butter = signal.filtfilt(B, A, noisy_signal)

        # Plot the filtered signal
        filter_plot(time, noisy_signal, y_sine + y_trend, y_low_butter,
                    [f"Signal+Trend+Noise (N={N}, ω_c={omega_c})", 'Signal+Trend', 'Filt

        plt.show()
```

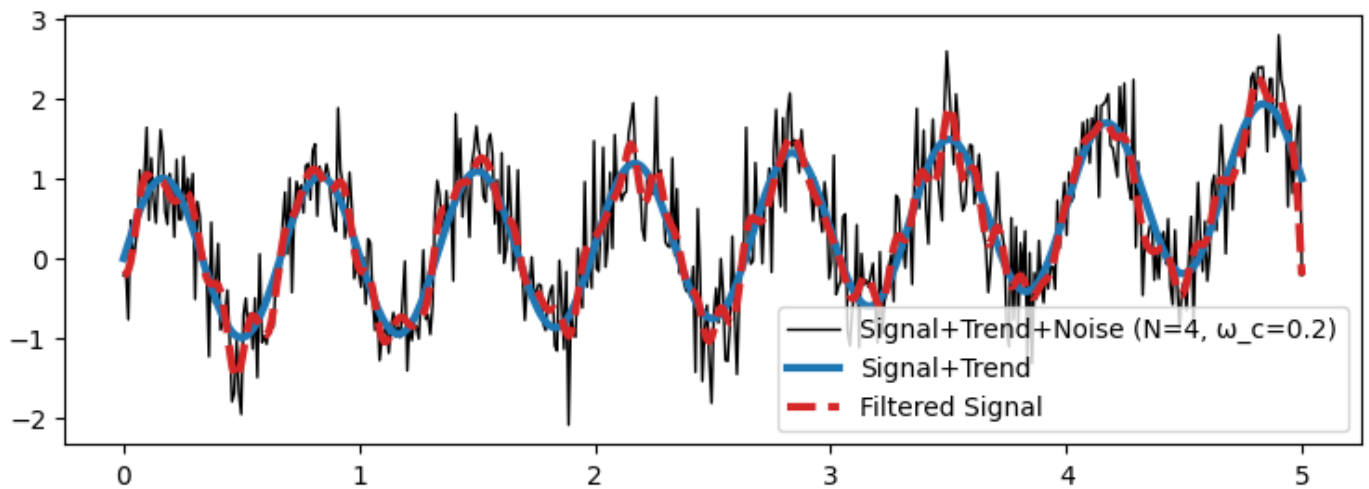


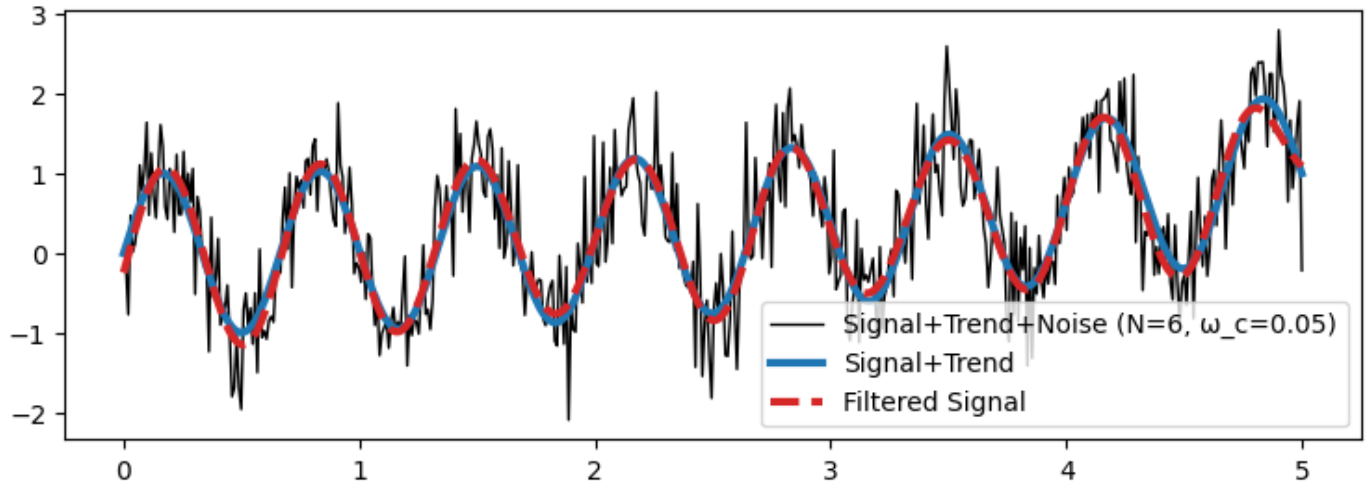


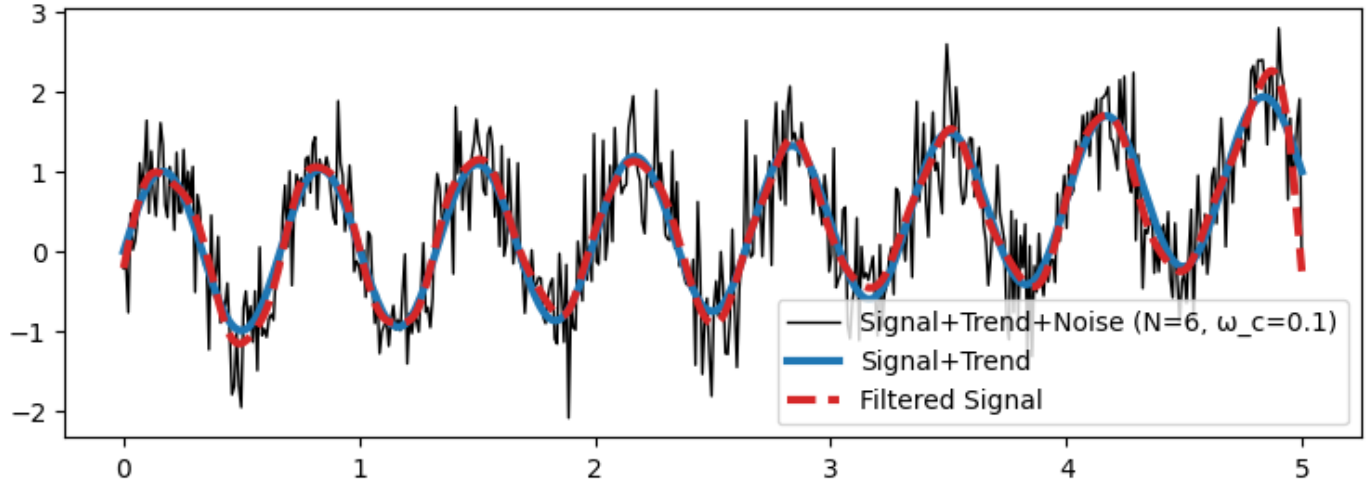

Signal+Trend+Noise (N=2, ω_c=0.2)
Signal+Trend
Filtered Signal

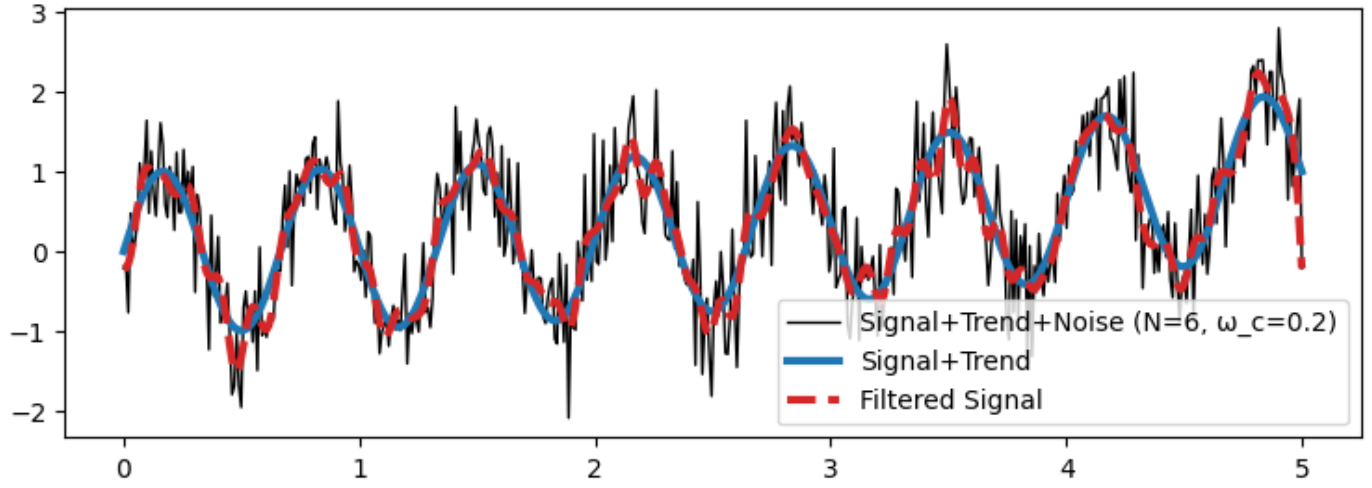

Signal+Trend+Noise (N=4, ω_c=0.05)
Signal+Trend
Filtered Signal


Signal+Trend+Noise (N=4, ω_c=0.1)
Signal+Trend
Filtered Signal


Signal+Trend+Noise (N=4, ω_c=0.2)
Signal+Trend
Filtered Signal







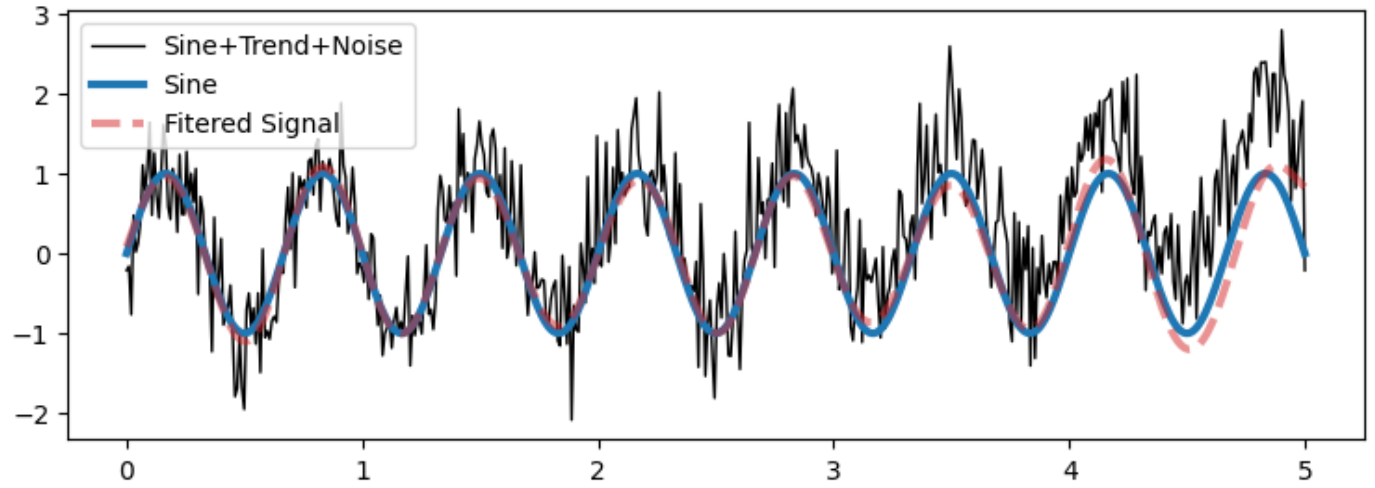
Typically, higher values of N and lower values of omega_c tend to offer better noise suppression, but they may also introduce more distortion. so N = 4 or 6 and omegac = 0.05 is a good choice

Exercise #4

- Optimize, by hand, the filter order N and cutoff frequency ω_c of the high-pass Butterworth filter.
- Apply the LPF in cascade to the HPF. The result should contain neither trend nor noise.

In [44]:
```
# High-pass Butterworth filter parameters
N_high = 4  # Filter order
omega_c_high = 0.02  # Cutoff frequency
```

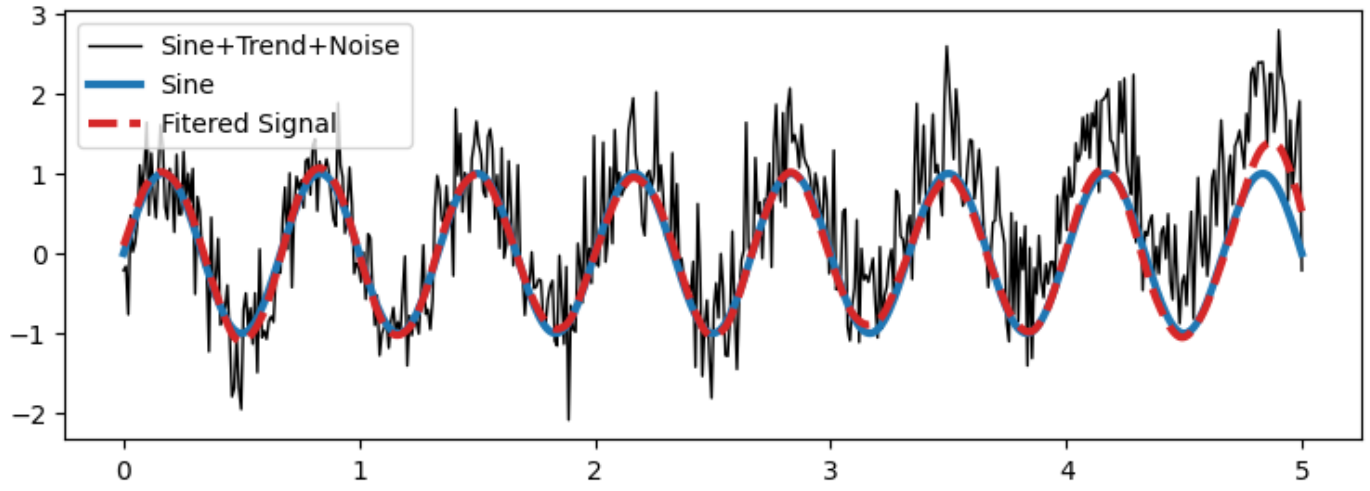
```
# Low-pass Butterworth filter parameters
N_low = 4  # Filter order
omega_c_low = 0.05  # Cutoff frequency

# Design high-pass Butterworth filter
B_high, A_high = signal.butter(N_high, omega_c_high, btype='highpass')

# Apply high-pass filter
y_high_butter = signal.filtfilt(B_high, A_high, noisy_signal)

# Design low-pass Butterworth filter
B_low, A_low = signal.butter(N_low, omega_c_low, btype='lowpass')

# Apply low-pass filter to the high-pass filtered signal
y_filtered = signal.filtfilt(B_low, A_low, y_high_butter)

# plot
filter_plot(time, noisy_signal, y_sine, y_filtered,
            ['Sine+Trend+Noise','Sine','Fitered Signal'], alpha=0.5)
```



Exercise #5

- Optimize, by hand, the filter order N and the cutoff frequencies ω_c^{LOW} and ω_c^{HIGH} of the band-pass Butterworth filter.
- Compare the result of the BPF with what you got in the previous ecercise when you applied a HPF and LPF in cascade.

In [45]:
```
# Band Pass Butterworth filter
N  = 1    # Filter order  2
omega_c = [0.02, 0.05] # Cutoff frequencies  0.02, 0.05
B, A = signal.butter(N, omega_c, btype = 'bandpass', output='ba', analog=True)

B, A = signal.butter(N, omega_c, btype = 'bandpass', output='ba')

# Apply the filter
y_band_butter = signal.filtfilt(B, A, noisy_signal)

# plot
filter_plot(time, noisy_signal, y_sine, y_band_butter,
            ['Sine+Trend+Noise','Sine','Fitered Signal'])
```



the results are similar to the cascading from the previous example

Exercise #6

- Download and preprocess the CO2 data `co2 = sm.datasets.get_rdataset("co2", "datasets").data`.
- Divide the data in training and test.
- Identify the optimal number of harmonics that gives the best MSE on the test set.

In [47]:
```
import statsmodels.api as sm

co2 = sm.datasets.get_rdataset("co2", "datasets").data
print(co2.head())

# Convert decimal year to pandas datetime
def convert_decimal_year_to_datetime(decimal_years):
    dates = [(pd.to_datetime(f'{int(year)}-01-01') + pd.to_timedelta((year - int(year))
             for year in decimal_years]
    return dates
co2['time'] = convert_decimal_year_to_datetime(co2['time'])

# Convert the column ds to datetime
co2['time'] = pd.to_datetime(co2['time'])
print("\nConverted:\n------------------\n", co2.head())

# Resample to monthly frequency based on the ds column
co2 = co2.resample('MS', on='time').mean().reset_index()

# Replace NaN with the mean of the previous and next value
co2['value'] = co2['value'].interpolate()
print("\nResampled:\n------------------\n", co2.head())
```

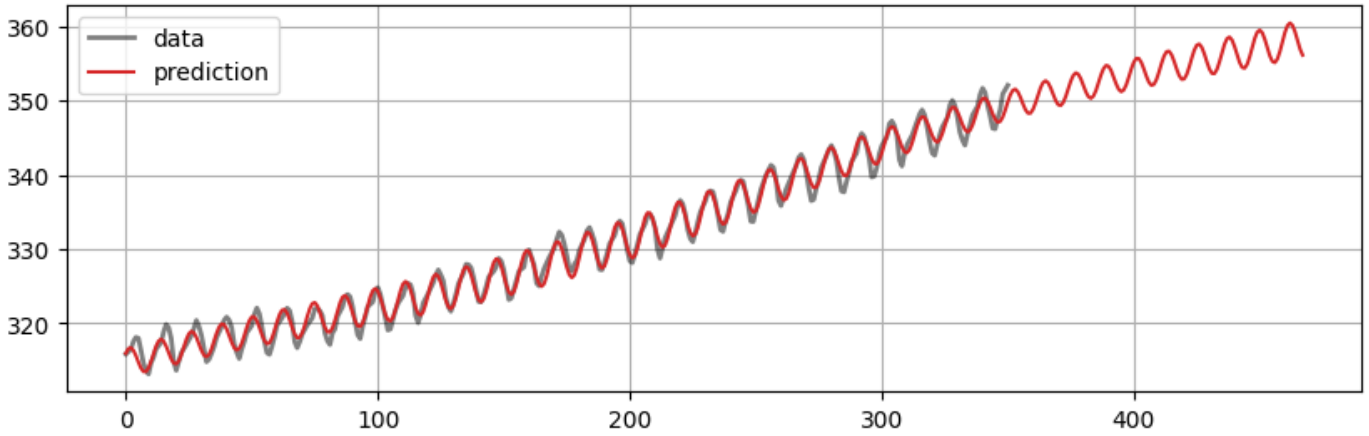
```
          time   value
0  1959.000000  315.42
1  1959.083333  316.31
2  1959.166667  316.50
3  1959.250000  317.56
4  1959.333333  318.13

Converted:
------------------
         time   value
0 1959-01-01  315.42
1 1959-01-31  316.31
2 1959-03-02  316.50
3 1959-04-02  317.56
4 1959-05-02  318.13

Resampled:
------------------
         time     value
0 1959-01-01  315.8650
1 1959-02-01  316.1825
2 1959-03-01  316.5000
3 1959-04-01  317.5600
4 1959-05-01  318.1300
```

In [48]:
```
# create train-test-split
train = co2['value'].iloc[:int(len(co2)*0.75)]
test = co2['value'].iloc[int(len(co2)*0.75):]
```

In [49]:
```
def fourierPrediction(y, n_predict, n_harm = 5):
    n = y.size                          # length of the time series
    t = np.arange(0, n)                 # time vector
    p = np.polyfit(t, y, 1)             # find linear trend in x
    y_notrend = y - p[0] * t - p[1]     # detrended x
    y_freqdom = np.fft.fft(y_notrend)  # detrended x in frequency domain
    f = np.fft.fftfreq(n)               # frequencies

    # Sort indexes by largest frequency components
    indexes = np.argsort(np.absolute(y_freqdom))[::-1]

    t = np.arange(0, n + n_predict)
    restored_sig = np.zeros(t.size)
    for i in indexes[:1 + n_harm * 2]:
        amp = np.absolute(y_freqdom[i]) / n   # amplitude
        phase = np.angle(y_freqdom[i])        # phase
        restored_sig += amp * np.cos(2 * np.pi * f[i] * t + phase)
    return restored_sig + p[0] * t + p[1] # add back the trend


def fourierPredictionPlot(y, prediction):
    plt.figure(figsize=(10, 3))
    plt.plot(np.arange(0, y.size), y, 'k', label = 'data', linewidth = 2, alpha=0.5)
    plt.plot(np.arange(0, prediction.size), prediction, 'tab:red', label = 'prediction')
    plt.grid()
    plt.legend()
    plt.show()
```

In [50]:
```
prediction = fourierPrediction(train, n_predict=117, n_harm=2)

fourierPredictionPlot(train, prediction)
```

```
# Extract the last 48 elements and store them in a temporary array
temp_array = prediction[-117:]

# Calculate the Mean Squared Error
mse = np.mean((temp_array - test) ** 2)
print("Mean Squared Error:", mse)
```




```
Mean Squared Error: 18.626119066305773
```

high values of nharmonics overfits the data but does not reduce the MSE of the test data. So a value of 2 is ideal for nharmonics parameter