# Time Series Data Analysis (FFT) Ex:

In [5]:
```
# dependencies
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import fft, ifft, fftfreq
```

# TAsk 1: Original signal (Target)

This is the original signal, that we will need to recover. We are going to assume it a sine wave with a particular period. Task: Generate a sine wave at frequency: f_original = 10 Hz, Amplitude: A_original = 1, time range = 0, 1, at rate 1/1000 time = np.linspace(0, 1, 1000) # Time from 0 to 1, sample rate is 1/1000 # sine wave signal as our target that we will need to recover original_signal = A_original * np.sin(2 * np.pi * f_original * time) # plot the signal plt.plot(time, original_signal, '-k') plt.xlabel("Time") plt.ylabel("Amplitude") plt.title("Original Sinusoidal Signal") plt.grid()

# Task 2: Let original signal get mixed with three other signals at frequency

# let original signal get mixed with three other signals at frequency # f1: 5*f_original, f2: 10*f_original, f3: 7*f_original # For simplicity, assume the amplitudes to be the same as original signal. # In practice, we may not know the source of these signals # Amplitudes A_2 = A_original A_3 = A_original A_4 = A_original # frequecies f_2 = 5*f_original f_3 = 10*f_original f_4 = 7*f_original # Other Signals signal_2 = A_2 * np.sin(2 * np.pi * f_2 * time) signal_3 = A_3 * np.sin(2 * np.pi * f_3 * time) signal_4 = A_4 * np.sin(2 * np.pi * f_4 * time) Task Plot these Signals on the same graph and for clarity limit the range to between 0 and 2

# Task 3: Original Signal gets distorted by other signal sources

# In practice we might not know the source of these Signals # add the other signals to the Original signal and plot the final signal siginal_sum = original_signal + signal_2 + signal_3 + signal_4 plt.plot(time,siginal_sum , '--k') plt.xlabel("Time") plt.ylabel("Amplitude") plt.title("Original Signal Distorted or Contaminated") plt.grid() plt.show()

# Task: Add random noise

# We assume the noise follows a white noise model, meaning it has a zero mean. # The spread of the noise around its mean is determined by its variance or standard deviation (std), # where std = sqrt(variance). # Create Noise noise_mean = 0 noise_std = 1.5 noise = np.random.normal(noise_mean,noise_std, len(time)) # Noise with mean 0 and standard deviation 0.2 Add this Noise to the signal and plot the result:

# Task: Add a DC offset to the signal

# Add a DC(Or simply think of it as a background) offset to the signal dc_value = 5 # Adjust this value as needed DC = np.ones(time.size) # replicate just to one signle value final_siginal = DC + noisey_siginal # plot the results together with the noisy signal withouth the DC

# FFT

## User-End Processing:

# Question: Can We Successfully Recover the Original Signal?

# Task: # Transfer the signal into the frequency domain using FFT: fft_signal = fft(final_siginal) # Compute the FFT # get the length of the signal length_signal = len(time) # Compute the frequency beans based n frequencies = fftfreq(len(time), (time[1] - time[0])) # f = 1/T Compute frequency bins # remember FFT is symetric so we only take half of the spectrum representing the real part on_real = len(frequencies)//2 # Floor Division (Integer Division) # Plot the frequency vs the absolute value of your fft output plt.plot(frequencies[:on_real], np.abs(fft_signal[:on_real]), label="Original FFT", color='b') plt.xlabel("Frequency") plt.ylabel("Magnitude") plt.title("Frequency Spectrum Before Filtering with DC") plt.legend() plt.grid() # what do you observe and what can you observe at the frequece = 0 Hz ?

# Task: Remove the DC value

# Remove DC component before FFT and replot the fft spectrum of the centered data centered_signal = final_siginal - np.mean(final_siginal) fft_centered_signal = fft(centered_signal) # Compute the FFT plt.plot(frequencies[:on_real ], np.abs(fft_centered_signal[:on_real]), label="Centered_signal_FFT", color='b') plt.xlabel("Frequency") plt.ylabel("Magnitude") plt.title("Frequency spectrum after removing the DC value") plt.legend() plt.grid()

# Task: Extract the desired Signal

fft_filtered = fft_centered_signal.copy() # lets get a copy before we mess things up by filtering filter_freq = f_original # creat a filter: there are better ways! filter_array = (np.abs(frequencies) > (filter_freq)) | (np.abs(frequencies) < (filter_freq-.1)) fft_filtered[filter_array] = 0 # Keep only low frequencies desired and set the rest to zero # plot the outcome plt.plot(frequencies[:on_real], np.abs(fft_filtered[:on_real]), label="Filtered FFT", color='b') plt.xlabel("Frequency") plt.ylabel("Magnitude") plt.title("Frequency Spectrum after filtering") plt.legend() plt.grid()