Time Series Data Analysis (FFT) Ex:

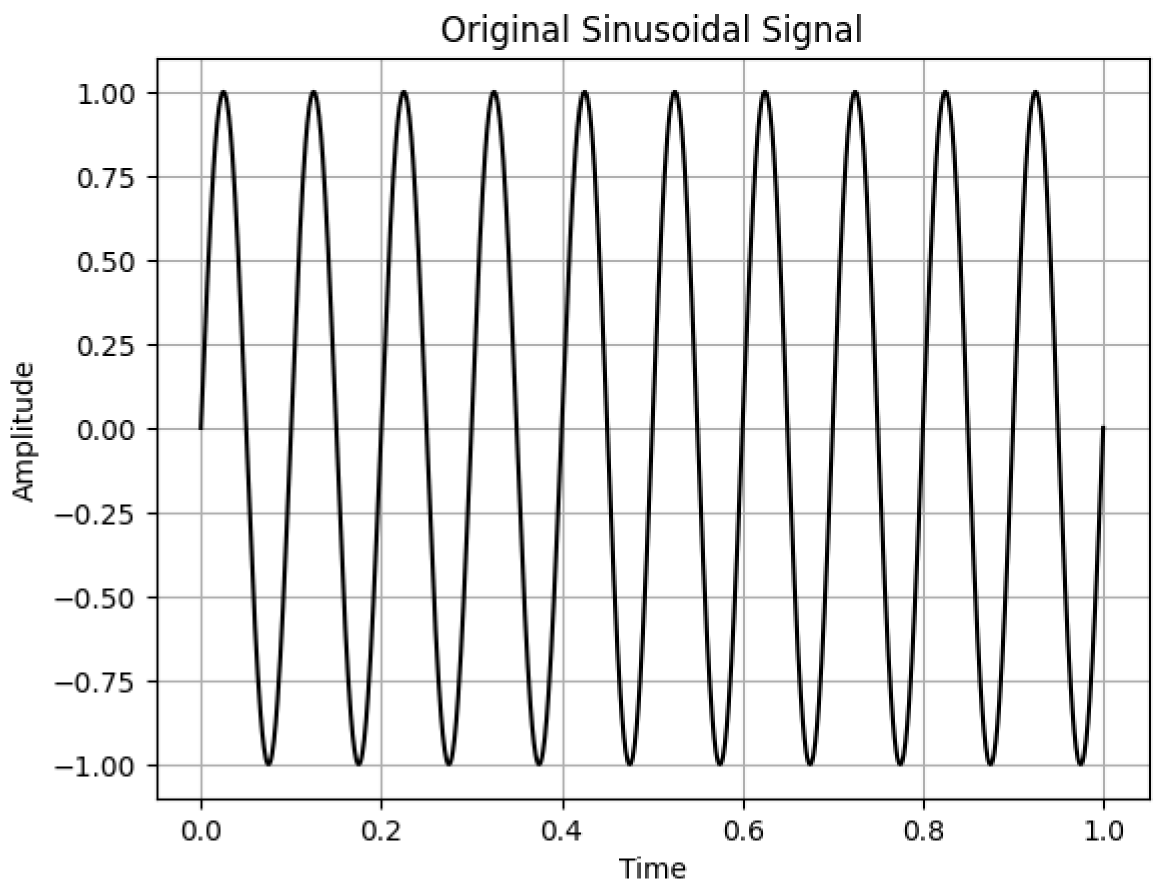
In [15]:
```
# dependencies
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import fft, ifft, fftfreq
```

Original signal

This is the original signal, that we will need to recover.

We are going to assume it a sine wave with a particular period.

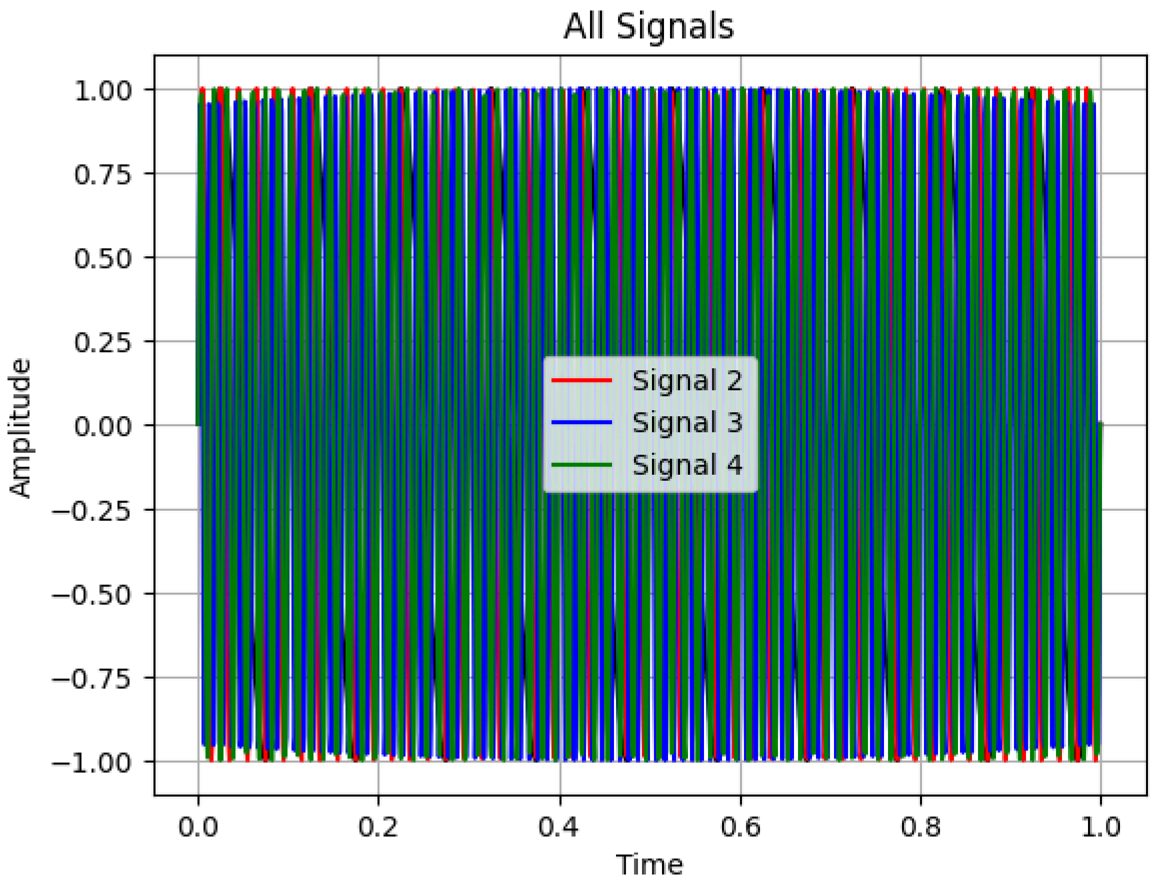
In [16]:
```
# Generate time values ranging from 0 to 2*Pi
# signal (y) = A* sin(wt)
# f = frequency
# A = amplitude
# t = time
# and w = 2*pi*f
# Generate sinusoidal signal at frequency 2 Hz

f_original = 10  # Hz
A_original = 1  # Amplitude of the sine wave

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()
plt.show()
```



The original signal is combined with other signals at varying frequencies and amplitudes.

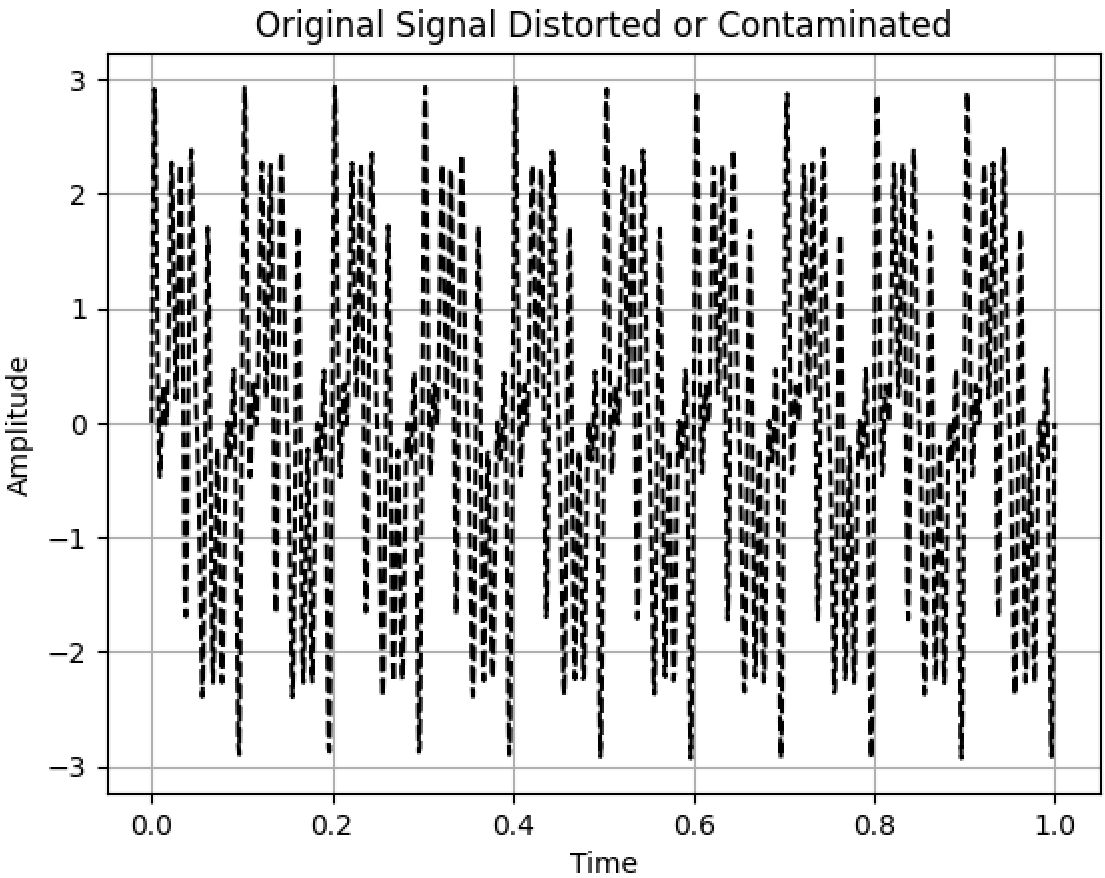
In [17]:
```
# let original signal get mixed with three other signals at frequency
#  5*f_original, 10*f_original, 7*f_original
# For simplicity, assume the amplitudes to be the same as original signal.
# In practice, we may not know the source of these signals

# 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

# 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)

plt.plot(time, original_signal, '-k')
plt.plot(time,signal_2, '-r', label= 'Signal 2')
plt.plot(time,signal_3, '-b', label= 'Signal 3')
plt.plot(time,signal_4, '-g', label= 'Signal 4')
```

```
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.title("All Signals")
# just for clarity
#plt.xlim([0,0.2])
plt.legend()
plt.grid()
plt.show()
```



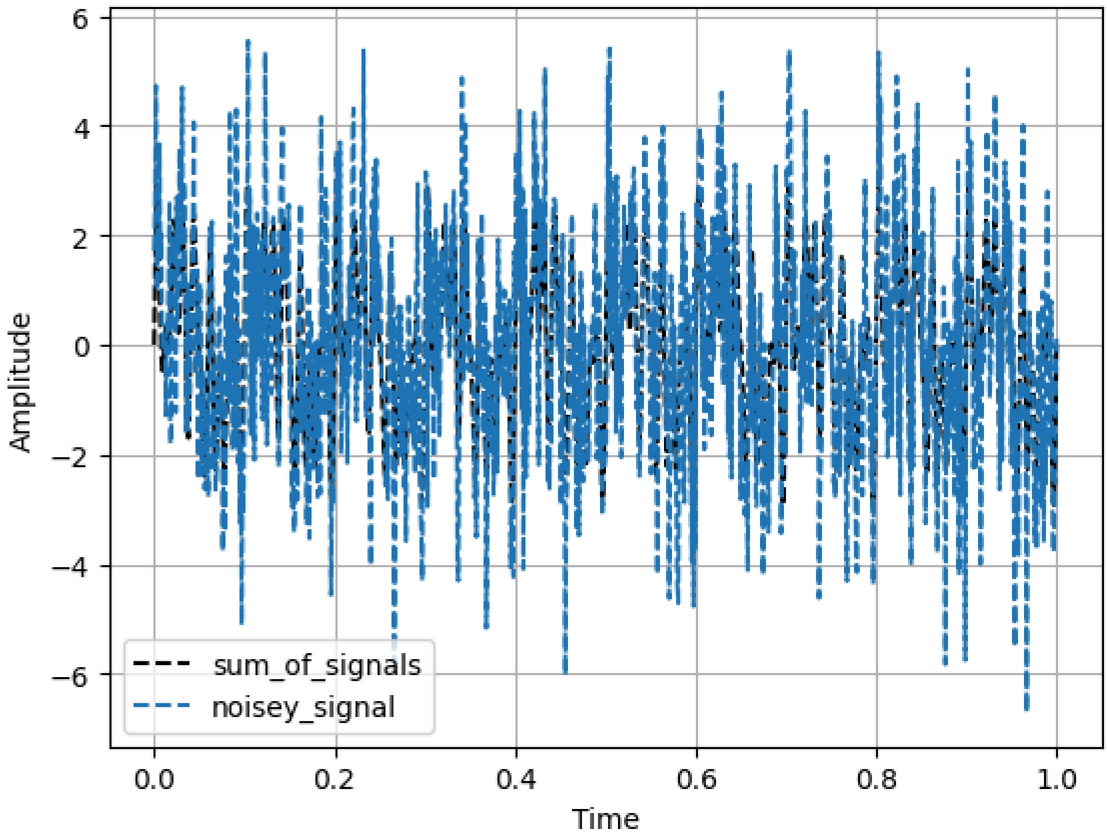
Original Signal gets distorted by other signal sources

In [18]:
```
# In practice we might not know the source of these Signals
#
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()
```



Add Random Noise:

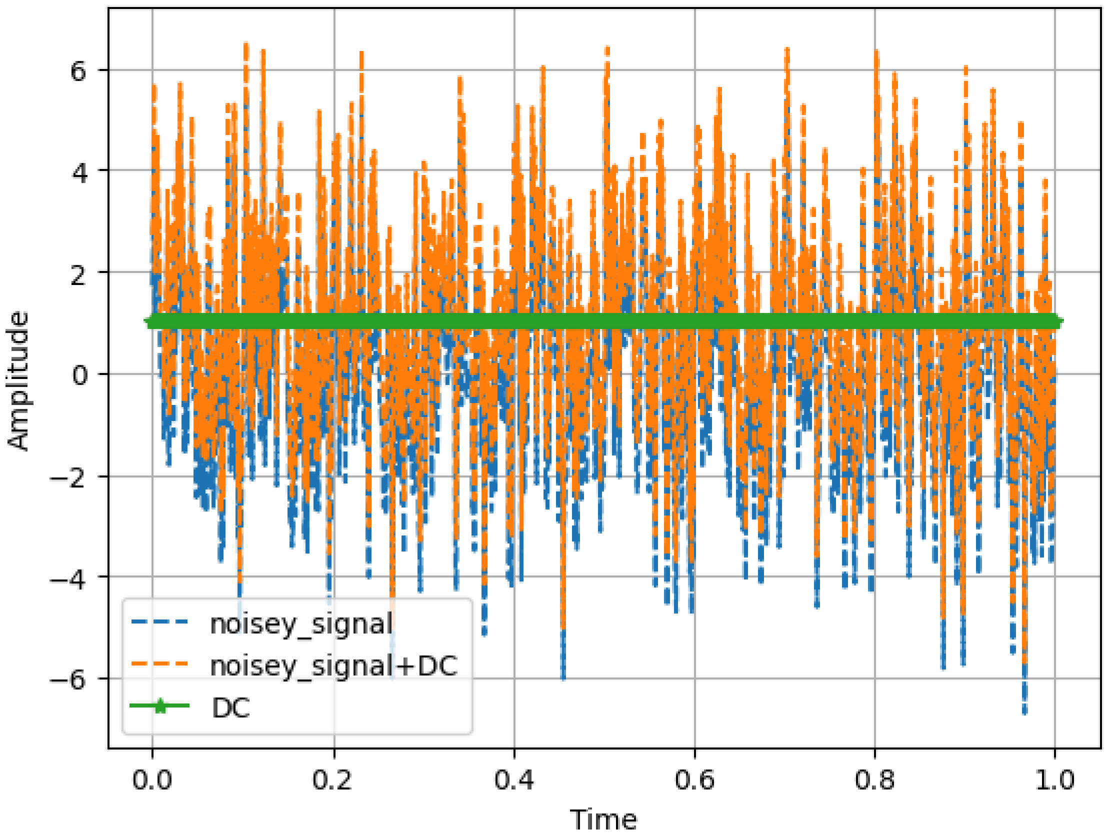
In [19]:
```
# Introduce random noise that affects the signal
# We assume the noise follows a white noise model, meaning it has a zero mean.
# The spread of the noise around its mean is determined by its variance or stand
# where std = sqrt(variance).

# Create Noise
noise_mean = 0
noise_std = 1.5
noise = np.random.normal(noise_mean,noise_std, len(time))  # Noise with mean 0 a

# Add  noise to the Signal
noisey_siginal = siginal_sum + noise

plt.plot(time,siginal_sum , '--k', label='sum_of_signals')
plt.plot(time,noisey_siginal, '--', label='noisey_signal')

plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.legend()
plt.grid()
plt.show()
```



Add a DC offset to the signal

In [20]:

```
# 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
plt.plot(time,noisey_siginal, '--', label='noisey_signal')
plt.plot(time,final_siginal, '--', label='noisey_signal+DC')

plt.plot(time,DC, '-*', label='DC')

plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.legend()
plt.grid()
plt.show()
```



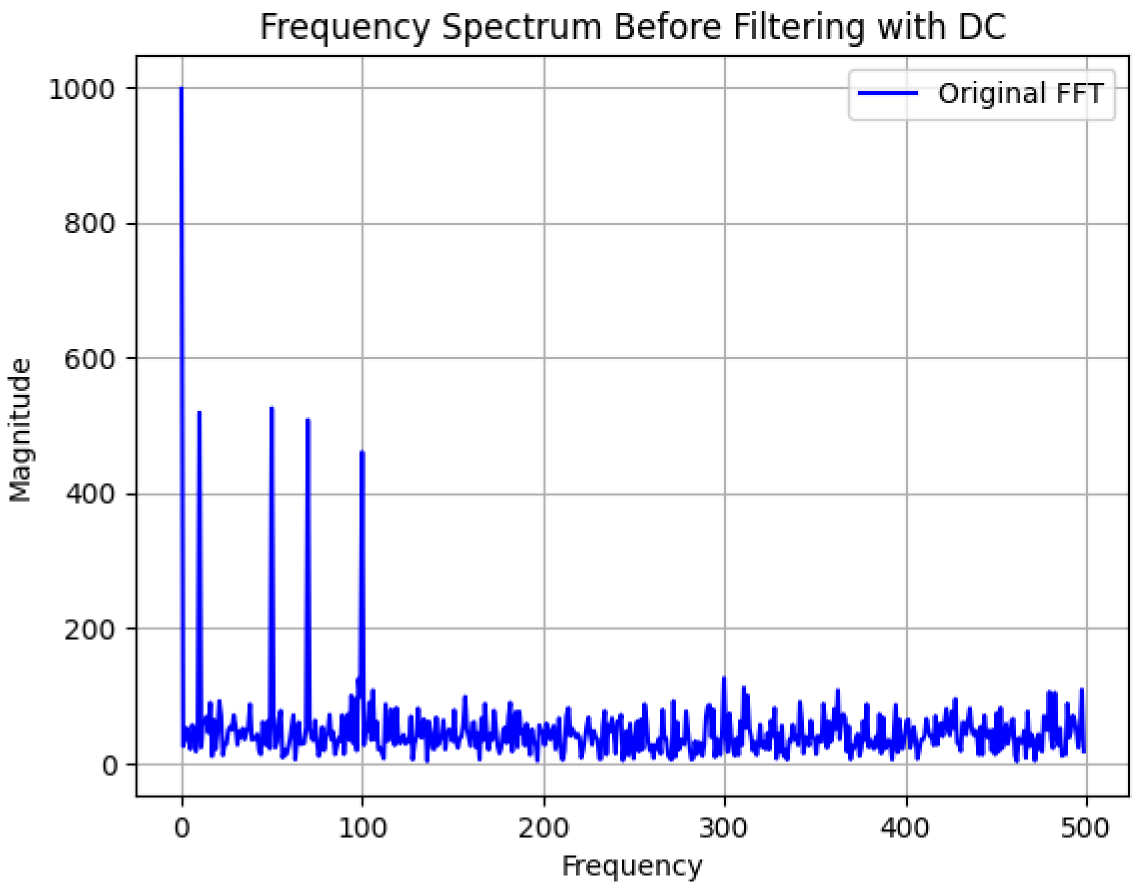
FFT

User-End Processing:

Question: Can We Successfully Recover the Original Signal?

In [21]:
```
# We will use FFT and tempt to recover the original signal

# 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 frequen

# remember FFT is symetric so we only take half of the spectrum representing the
on_real = len(frequencies)//2 # Floor Division (Integer Division)

plt.plot(frequencies[:on_real], np.abs(fft_signal[:on_real]), label="Original FF

plt.xlabel("Frequency")
plt.ylabel("Magnitude")
plt.title("Frequency Spectrum Before Filtering with DC")
```

```
plt.legend()
plt.grid()
```



In [22]:
```
# This is without any modification:
# and we see the value at zero frequence has avery high Magnitude:
# this corresponds to the DC value
# so before analysis, we might need to remove the background values or DC values
```

Remove the DC value

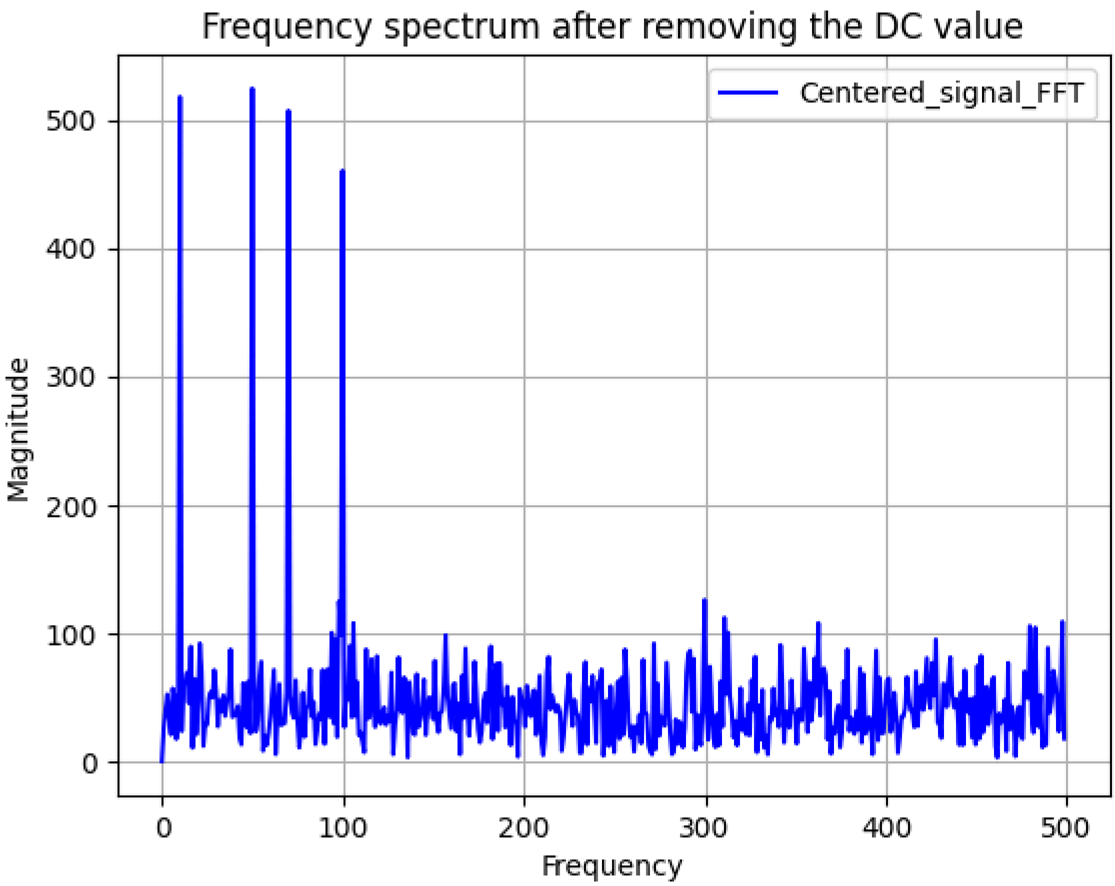
$$X[k] = \sum_{n=0} x[n]e^{-j2\pi kn/N}$$

When $k = 0$, the exponential term $e^{-j2\pi(0)n/N} = 1$, so:

$$X[0] = \sum_{n=0}^{N-1} x[n]$$

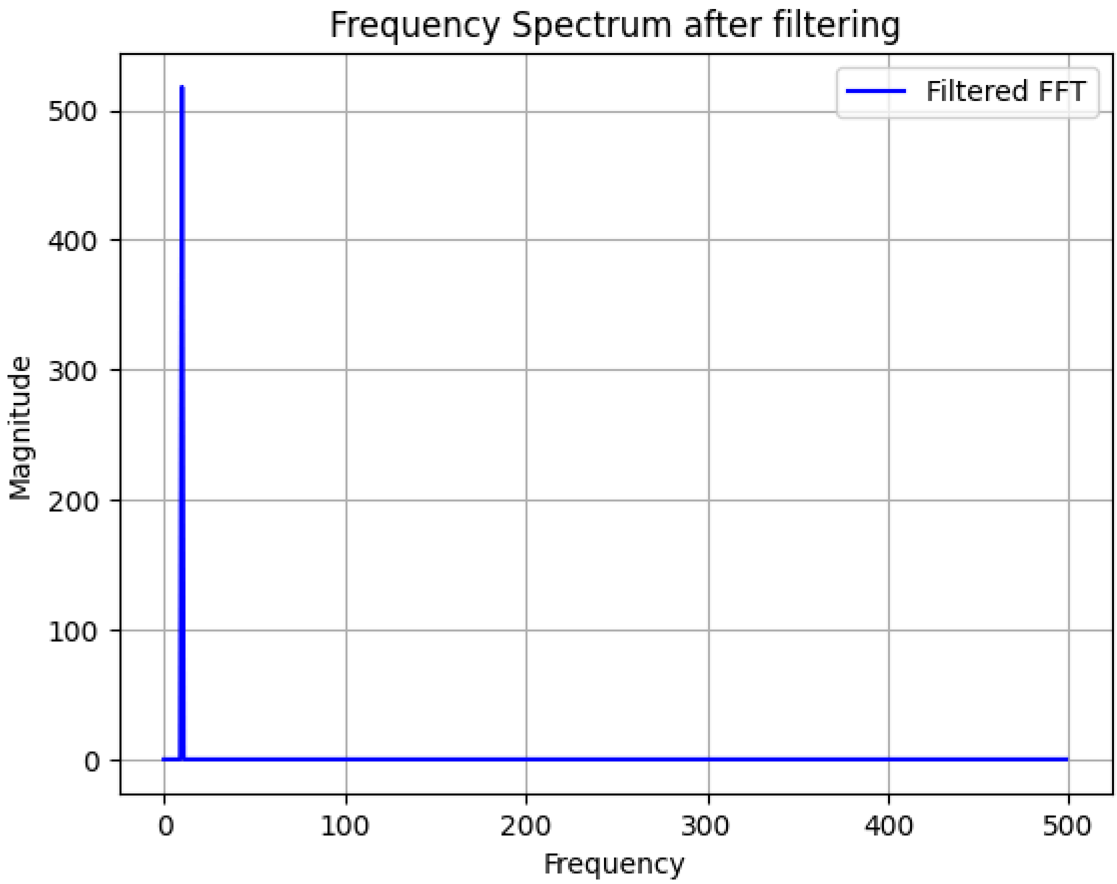
In [23]:
```
# Remove DC component before FFT
centered_signal = final_siginal - np.mean(final_siginal)
#plt.figure()
```

```
#plt.plot(time,centered_signal, label="Original FFT", color='b')
fft_centered_signal = fft(centered_signal)  # Compute the FFT

plt.plot(frequencies[:on_real ], np.abs(fft_centered_signal[:on_real]), label="C
plt.xlabel("Frequency")
plt.ylabel("Magnitude")
plt.title("Frequency spectrum after removing the DC value")
plt.legend()
plt.grid()
```



Extract the desired Signal

In [24]:
```
# We know that our target signal is confined to a specific frequency range.
# Apply a filter to remove higher frequencies and isolate the original signal.

fft_filtered = fft_centered_signal.copy() # lets get a copy before we mess thing
filter_freq = f_original

# creat a filter: there are better ways!
filter_array = (np.abs(frequencies) > (filter_freq)) |  (np.abs(frequencies) < (
fft_filtered[filter_array] = 0  # Keep only low frequencies desired and set the

# plot the outcome
plt.plot(frequencies[:on_real], np.abs(fft_filtered[:on_real]), label="Filtered
plt.xlabel("Frequency")
plt.ylabel("Magnitude")
plt.title("Frequency Spectrum after filtering")
plt.legend()
plt.grid()
```



Recover the signal (use ifft, iverse transeform)

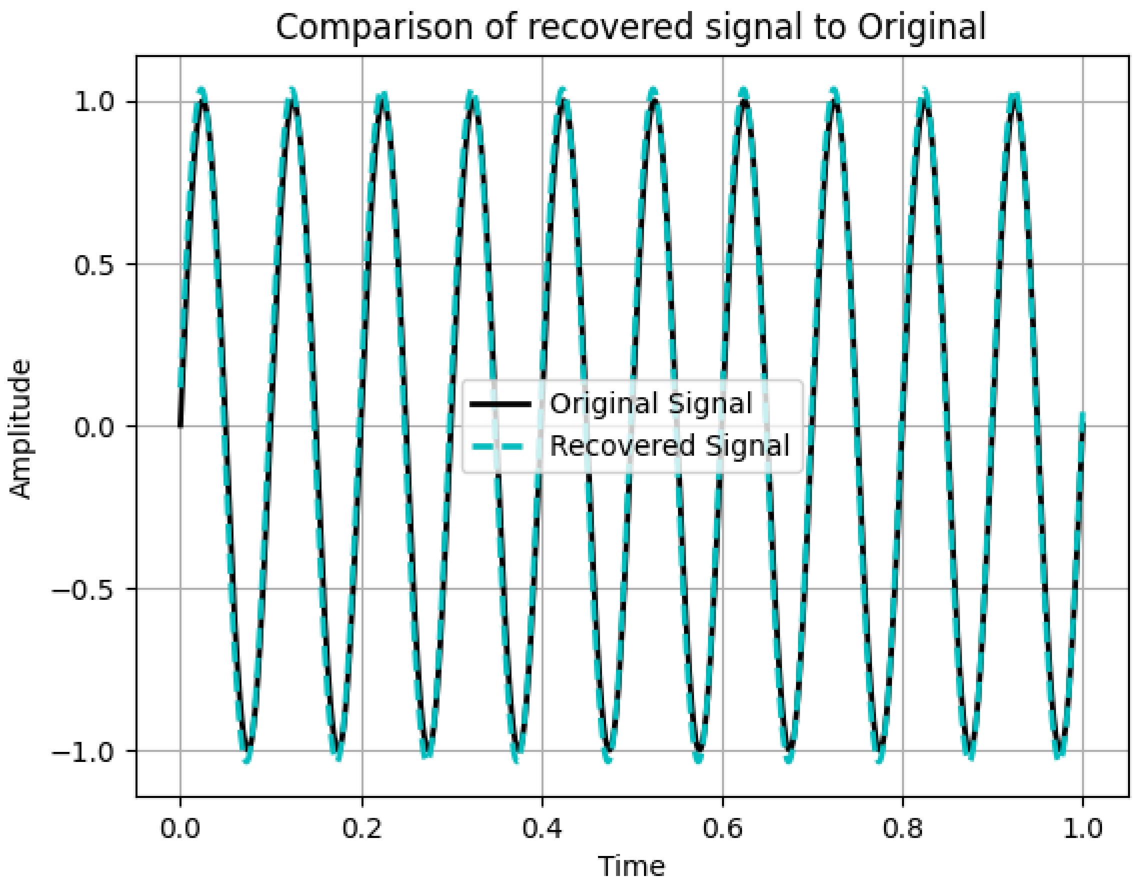
The IFFT is defined as:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{j2\pi kn/N}$$

where:

- x_n = time-domain signal at index n,
- X_k = frequency-domain component at index k,
- N = total number of points,
- j = imaginary unit ($j^2 = -1$),
- $e^{j2\pi kn/N}$ = inverse complex exponential (basis function).

In [25]:
```
# use ifft or iverse fft to get back the time domain
recovered_real_img = ifft(fft_filtered) # use ifft or iverse transeform

# once again only take the real part of the inverse transform
# Compute the inverse FFT but we are only intrested in the real part of the sign
recovered_signal = np.real(ifft(fft_filtered))

plt.plot(time, original_signal,'-', label="Original Signal", color='k', linewidt
plt.plot(time, recovered_signal, '--', label="Recovered Signal", color='c', line

# Add labels, title, and legend
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.title("Comparison of recovered signal to Original")
plt.legend()
plt.grid()
```



Using a Hanning window before fft and filtering

The Hanning window function is defined as:

$$w(n) = 0.5\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right)$$

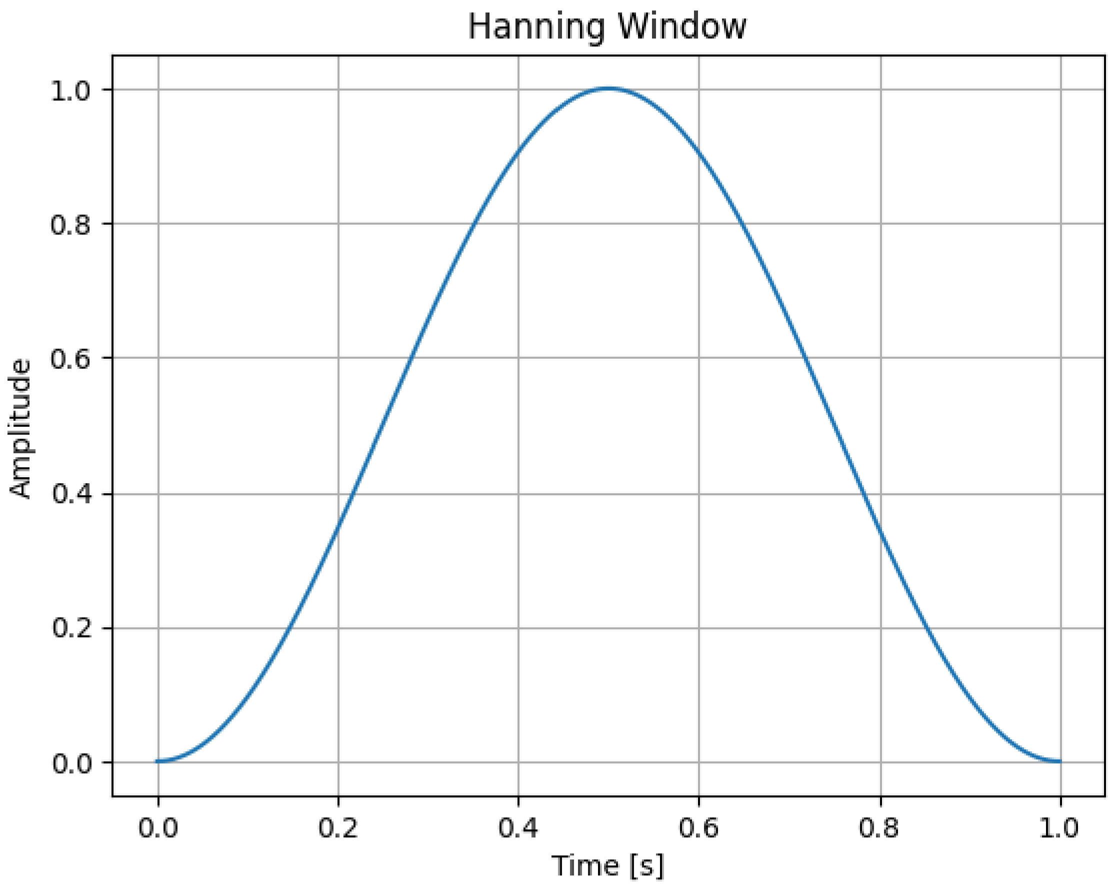
where:

- N is the window length.
- n is the sample index.

This function **gradually tapers the signal at the edges**, preventing sharp transitions.

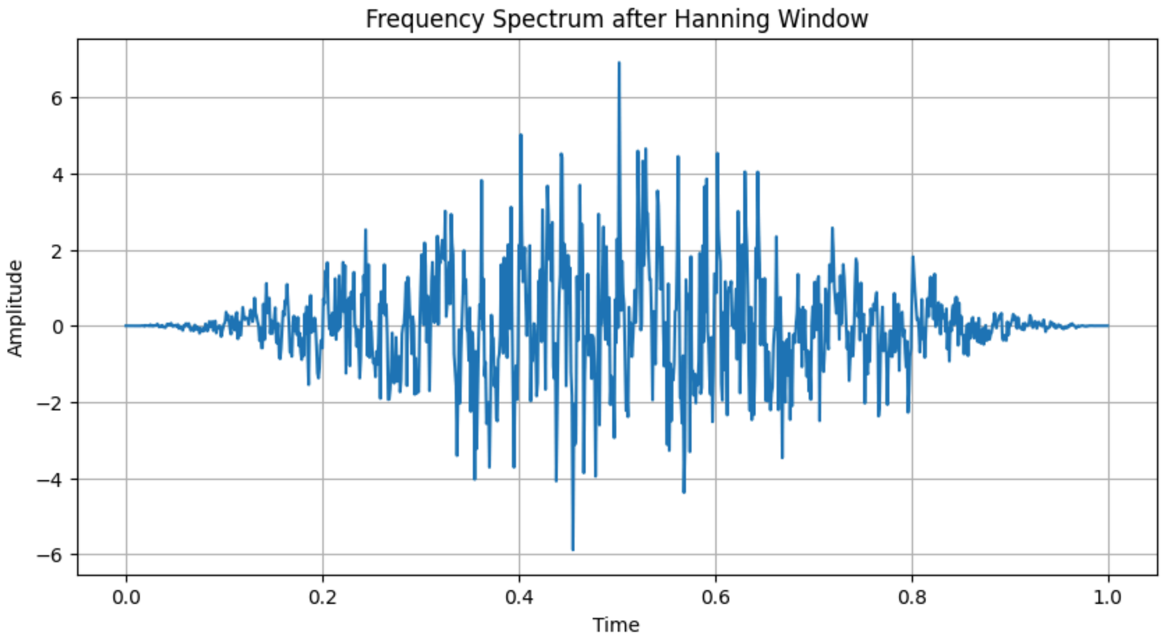
In [23]:
```
from scipy import signal
hann_window = signal.windows.hann(length_signal)  # Hanning window
length_signal = len(time)
fig, ax = plt.subplots()
```

```
plt.plot(time, hann_window)
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")
plt.grid()
plt.title("Hanning Window")
plt.show()
```



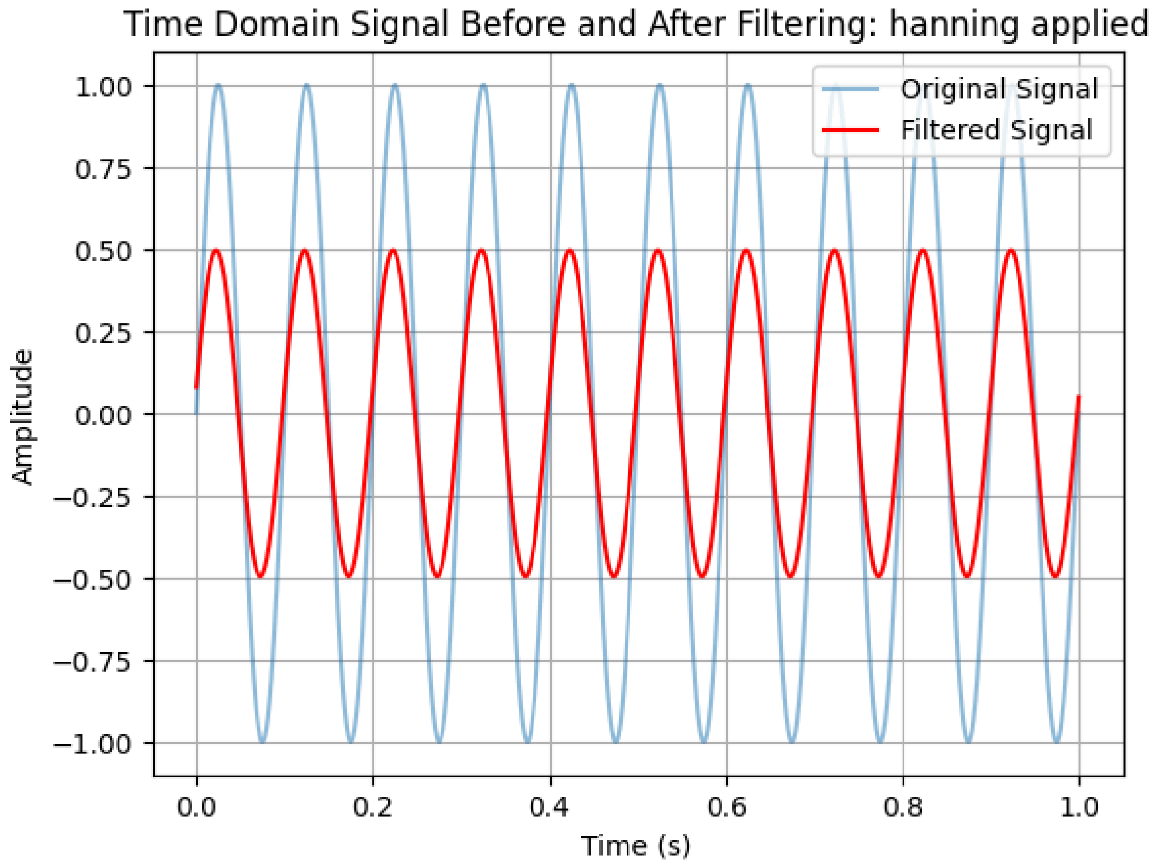
In [24]:
```
# Apply the Hanning window to smooth the FFT signal
centered_signal = final_siginal - np.mean(final_siginal)
centered_signal_hann = centered_signal * hann_window
```

In [25]:
```
# Compute inverse FFT to get filtered time-domain signal
# Plot the frequency spectrum after filtering
plt.figure(figsize=(10, 5))
plt.plot(time, centered_signal_hann)
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.title("Frequency Spectrum after Hanning Window")
plt.grid()
plt.show()
```



In [26]:
```
fft_signal_hann = fft(centered_signal_hann)  # get FFT
filter_freq = f_original
filter_array = (np.abs(frequencies) > (filter_freq)) | (np.abs(frequencies) < (f
fft_signal_hann[filter_array] = 0  # Zero out unwanted frequencies
```

In [27]:
```
# Compute inverse FFT to get filtered time-domain signal
filtered_signal = np.fft.ifft(fft_signal_hann).real

# Plot original and filtered signal in time domain
plt.plot(time, original_signal, label="Original Signal", alpha=0.5)
plt.plot(time, filtered_signal, label="Filtered Signal", color='r')
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.title("Time Domain Signal Before and After Filtering: hanning applied")
plt.legend()
plt.grid()
plt.show()
```


Time Domain Signal Before and After Filtering: hanning applied
Original Signal
Filtered Signal
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Amplitude
0.0
0.2
0.4
0.6
0.8
1.0
Time (s)

In [ ]: