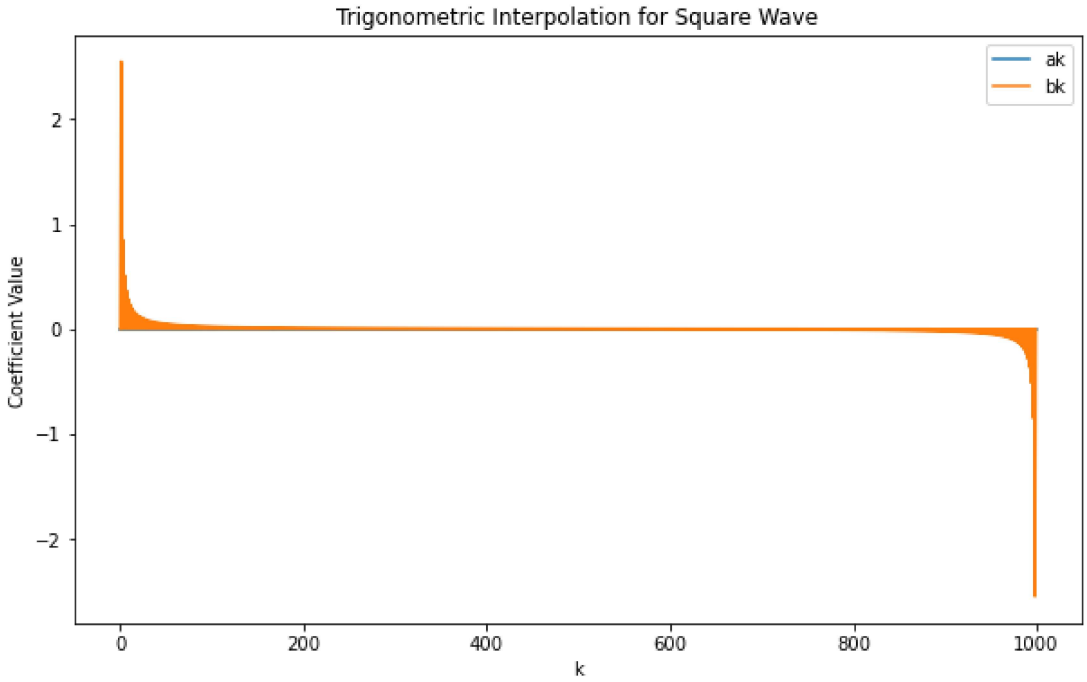
Computational Physics HW8: Fourier Transforms

In [1]:
```
import numpy as np
import random
import math
import matplotlib.pyplot as plt
import time
from scipy.fft import fft, dct
from scipy.fftpack import fft, ifft
from scipy.signal import savgol_filter
```

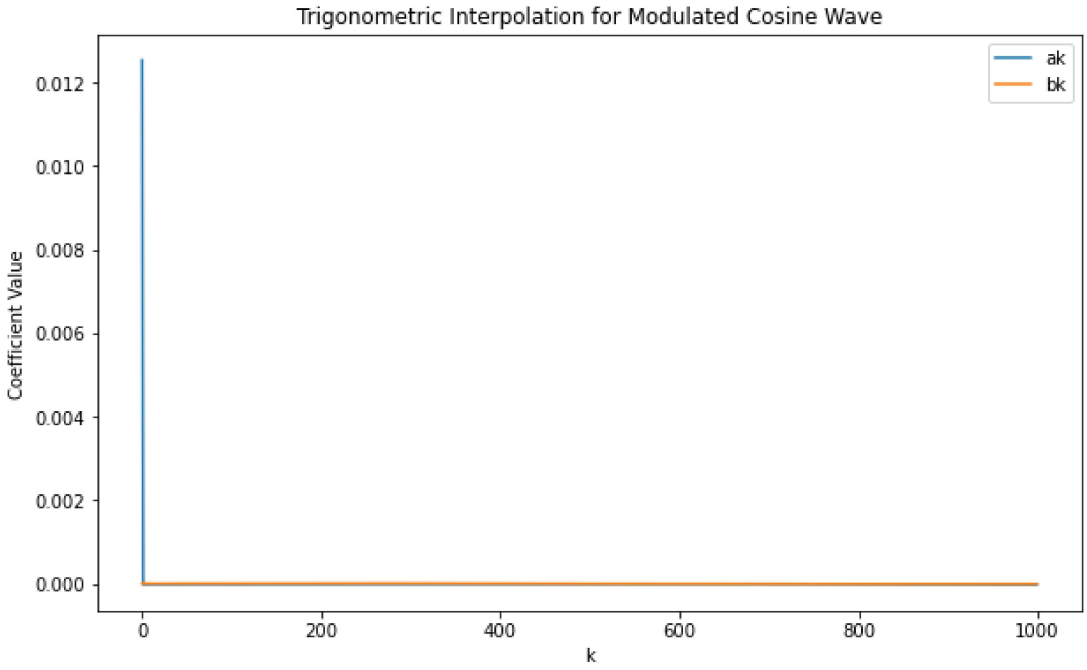
Problem 1:

In [3]:

```
# part a
pi = math.pi
def square_wave(x):
    return 1 if 0 <= x < np.pi else -1

# generate points
N= 1000
x_values = np.linspace(0, 2 * np.pi, N)

# ak bk coeffs
ak = np.zeros(N)
bk = np.zeros(N)

for i in range(N):
    ak[i] = 2 / np.pi * np.trapz([square_wave(x) * np.cos(i * x) for x i
    bk[i] = 2 / np.pi * np.trapz([square_wave(x) * np.sin(i * x) for x i

# Plot ak and bk coefficients
plt.figure(figsize=(10, 6))
plt.plot(ak, label='ak')
plt.plot(bk, label='bk')
plt.title('Trigonometric Interpolation for Square Wave')
plt.xlabel('k')
plt.ylabel('Coefficient Value')
plt.legend()
plt.show()
```



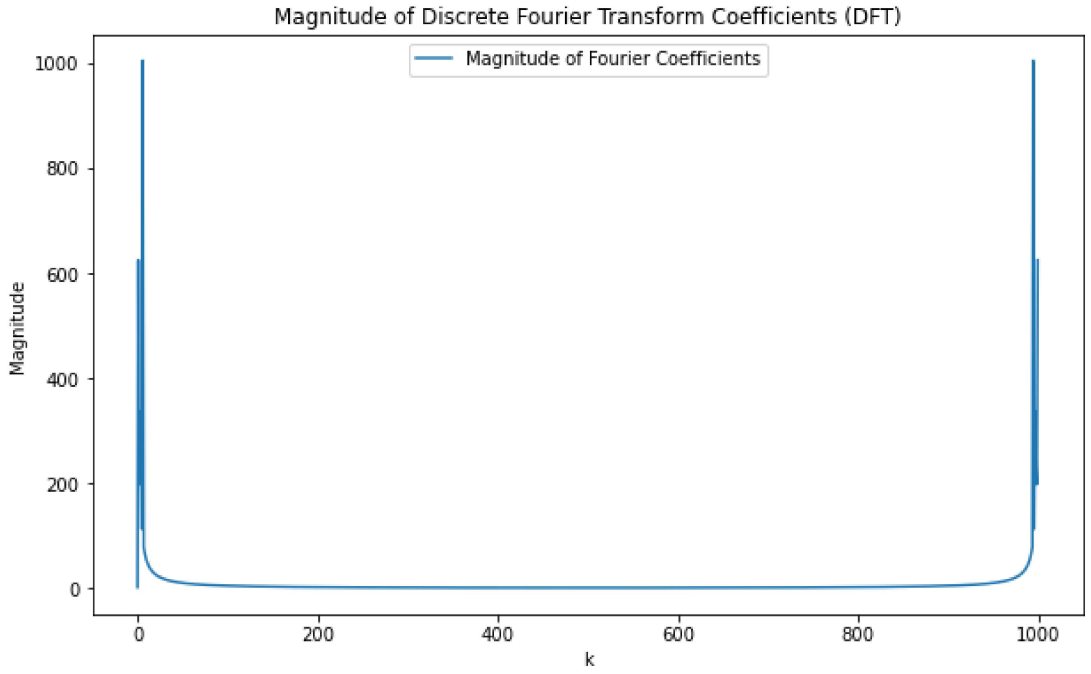
In [4]:
```
# part b
def modulated_cosine_wave(x, N):
    return np.cos(np.pi * x / N) * np.cos(5 * np.pi * x / N)


N = 1000
x_values = np.linspace(0, 2 * np.pi, N, endpoint=False)


ak = np.zeros(N)
bk = np.zeros(N)

for i in range(N):
    ak[i] = 2 / N * np.trapz([modulated_cosine_wave(x, N) * np.cos(i * x
    bk[i] = 2 / N * np.trapz([modulated_cosine_wave(x, N) * np.sin(i * x

# Plot ak and bk coefficients
plt.figure(figsize=(10, 6))
plt.plot(ak, label='ak')
plt.plot(bk, label='bk')
plt.title('Trigonometric Interpolation for Modulated Cosine Wave')
plt.xlabel('k')
plt.ylabel('Coefficient Value')
plt.legend()
plt.show()
```



Problem 2:

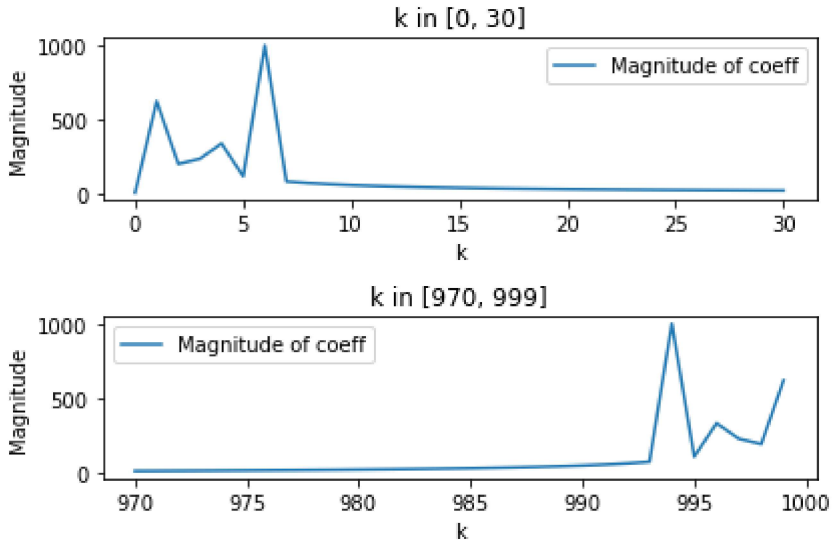
In [5]:

```
# part a
def f(x):
    return 1.5 * np.cos(x) + 0.2 * np.cos(5 * x) + 0.6 * np.cos(8 * x) +

N = 1000
x_values = np.linspace(0, np.pi, N)
y_values = f(x_values)

# part b
def discrete_fourier_transform(y_values):
    N = len(y_values)
    y_fft = np.zeros(N, dtype=np.complex128)
    for k in range(N):
        for n in range(N):
            y_fft[k] += y_values[n] * np.exp(-2j * np.pi * k * n / N)
    return y_fft

y_fft = discrete_fourier_transform(y_values)

# gets magnitude of coeff
magnitude = np.abs(y_fft)

# Plot the magnitude of the coefficients vs. their k value
plt.figure(figsize=(10, 6))
plt.plot(magnitude, label='Magnitude of Fourier Coefficients')
plt.title('Magnitude of Discrete Fourier Transform Coefficients (DFT)')
plt.xlabel('k')
plt.ylabel('Magnitude')
plt.legend()
plt.show()
```



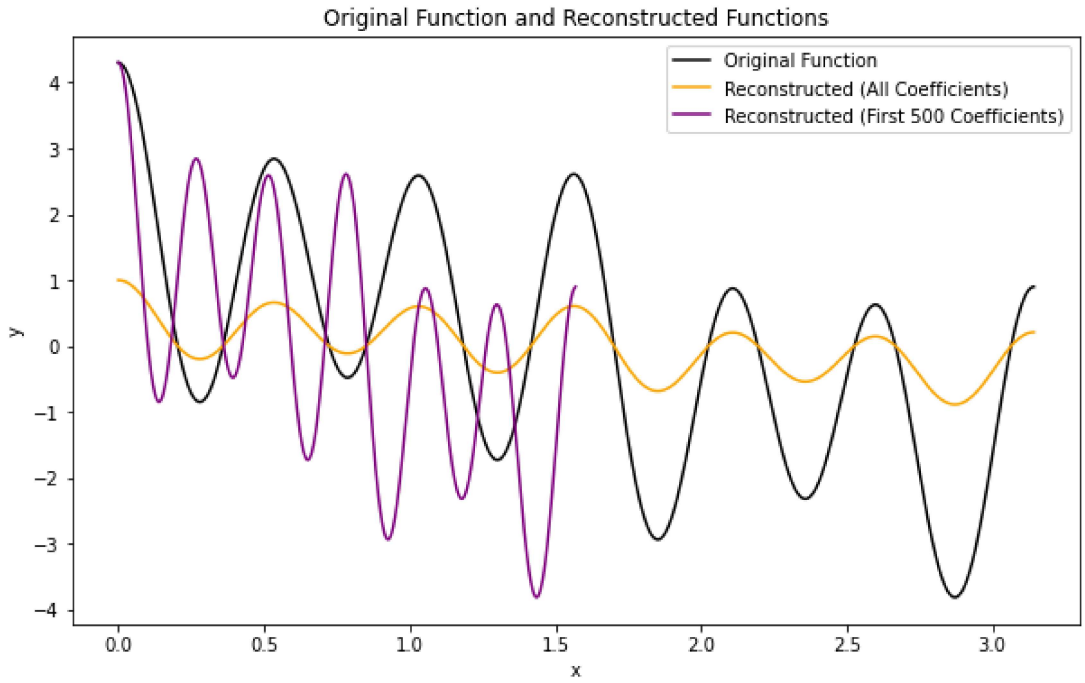
In [6]:

```
# part c
plt.subplot(2, 1, 1)
plt.plot(range(31), magnitude[:31], label='Magnitude of coeff')
plt.title('k in [0, 30]')
plt.xlabel('k')
plt.ylabel('Magnitude')
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(range(970, 1000), magnitude[970:], label='Magnitude of coeff')
plt.title('k in [970, 999]')
plt.xlabel('k')
plt.ylabel('Magnitude')
plt.legend()

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



In [7]:

```
# part d
import numpy as np
import matplotlib.pyplot as plt

def discrete_fourier_transform(y_values):
    N = len(y_values)
    y_fft = np.zeros(N, dtype=np.complex128)
    for k in range(N):
        for n in range(N):
            y_fft[k] += y_values[n] * np.exp(-2j * np.pi * k * n / N)
    return y_fft

def inverse_discrete_fourier_transform(y_fft):
    N = len(y_fft)
    y_values_reconstructed = np.zeros(N, dtype=np.complex128)
    for n in range(N):
        for k in range(N):
            y_values_reconstructed[n] += y_fft[k] * np.exp(2j * np.pi *
        y_values_reconstructed[n] /= N
    return y_values_reconstructed.real  # Take the real part

# Compute the discrete Fourier transform
y_fft = discrete_fourier_transform(y_values)

# Compute the inverse discrete Fourier transform for all coefficients
y_reconstructed_all = inverse_discrete_fourier_transform(y_fft)
y_reconstructed_all_normalized = y_reconstructed_all / np.max(np.abs(y_r

# Compute the inverse discrete Fourier transform for the first 500 coeff
y_reconstructed_500 = inverse_discrete_fourier_transform(y_fft[:500])

# Plot the original function and the reconstructed functions
plt.figure(figsize=(10, 6))
plt.plot(x_values, y_values, label='Original Function', color = 'black')
plt.plot(x_values, y_reconstructed_all_normalized, label='Reconstructed
plt.plot(x_values[:500], y_reconstructed_500, label='Reconstructed (Firs
plt.title('Original Function and Reconstructed Functions')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

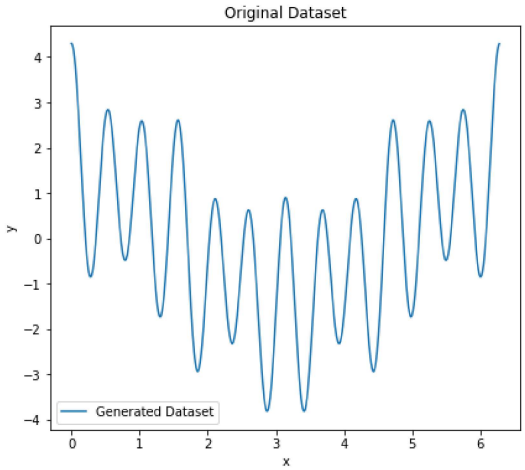

Original Function and Reconstructed Functions
Original Function
Reconstructed (All Coefficients)
Reconstructed (First 500 Coefficients)
y
x
4
3
2
1
0
−1
−2
−3
−4
0.0
0.5
1.0
1.5
2.0
2.5
3.0

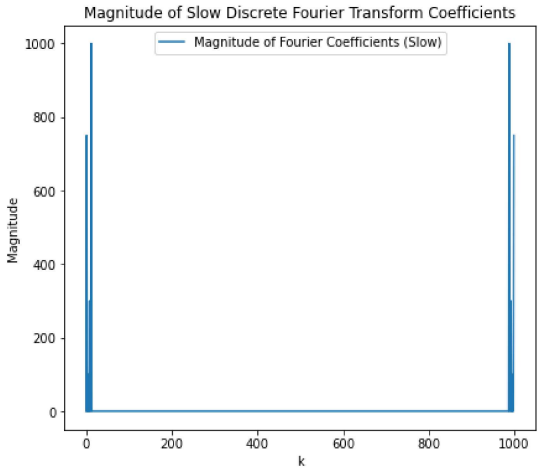
Problem 3:

In [8]:

```
# part a
def f(x):
    return 1.5 * np.cos(x) + 0.2 * np.cos(5 * x) + 0.6 * np.cos(8 * x) +

N = 1000
x_values = np.linspace(0, 2 * np.pi, N, endpoint=False)
y_values = f(x_values)

# part b
def slow_discrete_fourier_transform(y_values):
    N = len(y_values)
    y_fft = np.zeros(N, dtype=np.complex128)
    for k in range(N):
        for n in range(N):
            y_fft[k] += y_values[n] * np.exp(-2j * np.pi * k * n / N)
    return y_fft

# Measure the completion time of the slow Fourier transform
start_time = time.time()
y_slow_fft = slow_discrete_fourier_transform(y_values)
end_time = time.time()
completion_time = end_time - start_time

print(f"Slow Fourier Transform Completion Time: {completion_time:.6f} se

# Plot the magnitude of the coefficients
magnitude_slow = np.abs(y_slow_fft)

# Plot the original function and the slow Fourier transform
plt.figure(figsize=(15, 6))

plt.subplot(1, 2, 1)
plt.plot(x_values, y_values, label='Generated Dataset')
plt.title('Original Dataset')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(magnitude_slow, label='Magnitude of Fourier Coefficients (Slow)
plt.title('Magnitude of Slow Discrete Fourier Transform Coefficients')
plt.xlabel('k')
plt.ylabel('Magnitude')
plt.legend()

plt.show()
```

```
Slow Fourier Transform Completion Time: 5.699917 seconds
```

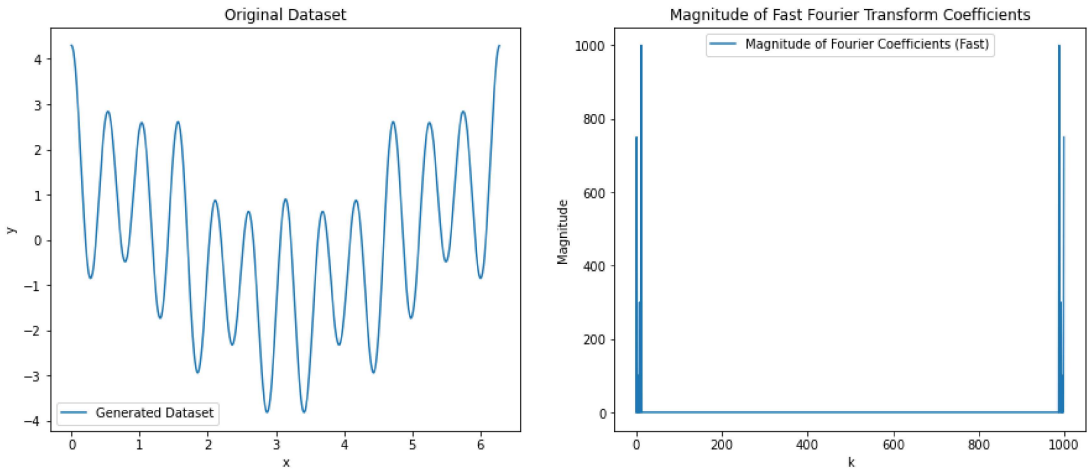

Original Dataset
Generated Dataset
x
y


Magnitude of Slow Discrete Fourier Transform Coefficients
Magnitude of Fourier Coefficients (Slow)
k
Magnitude

In [9]:

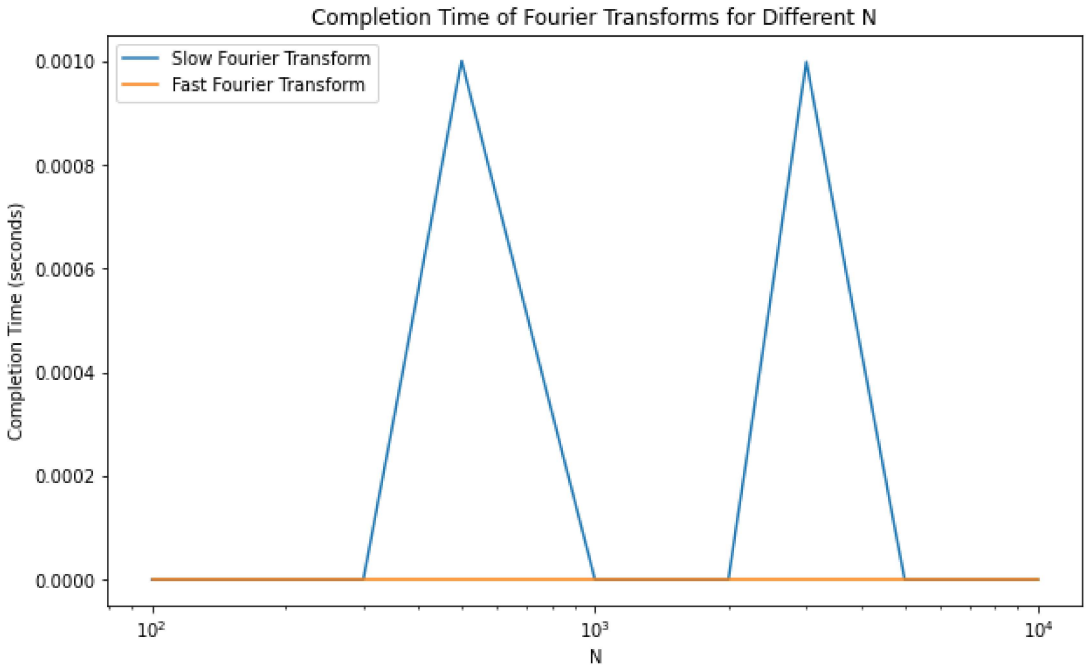
```
# part c
start_time = time.time()
y_fft = fft(y_values)
end_time = time.time()
completion_time = end_time - start_time

print(f"Fast Fourier Transform Completion Time: {completion_time:.6f} se

# Plot the magnitude of the coefficients
magnitude_fast = np.abs(y_fft)

# Plot the original function and the fast Fourier transform
plt.figure(figsize=(15, 6))

plt.subplot(1, 2, 1)
plt.plot(x_values, y_values, label='Generated Dataset')
plt.title('Original Dataset')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(magnitude_fast, label='Magnitude of Fourier Coefficients (Fast)
plt.title('Magnitude of Fast Fourier Transform Coefficients')
plt.xlabel('k')
plt.ylabel('Magnitude')
plt.legend()

plt.show()
```

```
Fast Fourier Transform Completion Time: 0.000000 seconds
```



In [10]:

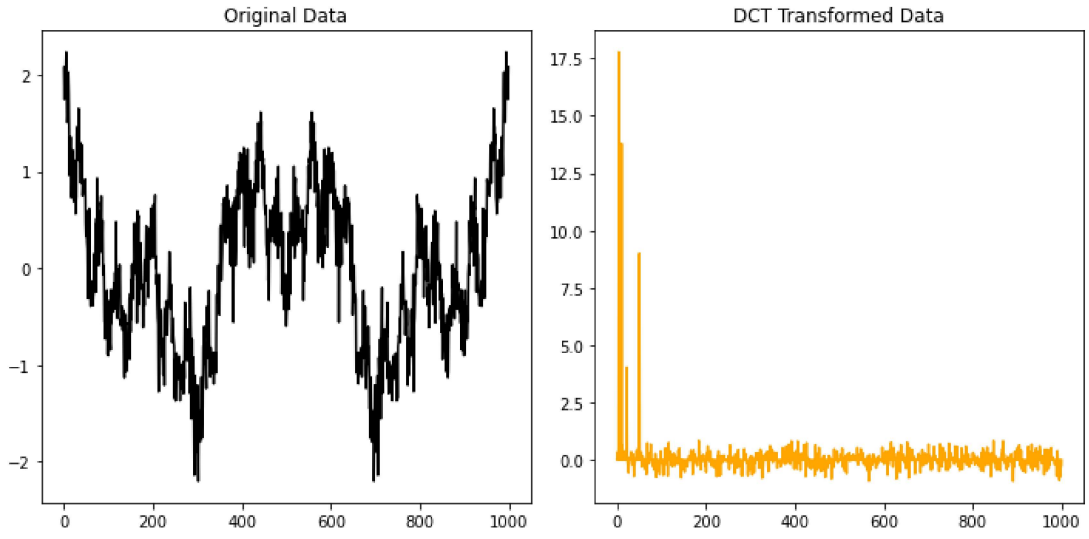
```
import numpy as np
import matplotlib.pyplot as plt
import time
from scipy.fft import fft

# Define the function f(x)
def f(x):
    return 1.5 * np.cos(x) + 0.2 * np.cos(5 * x) + 0.6 * np.cos(8 * x) +

# Values of N to test
N_values = [100, 200, 300, 500, 1000, 2000, 3000, 5000, 10000]

# Lists to store completion times
slow_completion_times = []
fast_completion_times = []

# Loop through different values of N
for N in N_values:
    # Generate equidistant points between 0 and (almost) 2π
    x_values = np.linspace(0, 2 * np.pi, N, endpoint=False)

    # Evaluate the function at each x value
    y_values = f(x_values)

    # Measure the completion time of the slow Fourier transform
    start_time_slow = time.time()
    y_slow_fft = np.fft.fft(y_values)
    end_time_slow = time.time()
    slow_completion_times.append(end_time_slow - start_time_slow)

    # Measure the completion time of the fast Fourier transform using sc
    start_time_fast = time.time()
    y_fast_fft = fft(y_values)
    end_time_fast = time.time()
    fast_completion_times.append(end_time_fast - start_time_fast)

# Plot the completion times
plt.figure(figsize=(10, 6))
plt.plot(N_values, slow_completion_times, label='Slow Fourier Transform'
plt.plot(N_values, fast_completion_times, label='Fast Fourier Transform'
plt.title('Completion Time of Fourier Transforms for Different N')
plt.xlabel('N')
plt.ylabel('Completion Time (seconds)')
plt.xscale('log')  # Use a logarithmic scale for better visualization
plt.legend()
plt.show()
```


Completion Time of Fourier Transforms for Different N
Slow Fourier Transform
Fast Fourier Transform
Completion Time (seconds)
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
10^2
10^3
10^4
N

Problem 4:

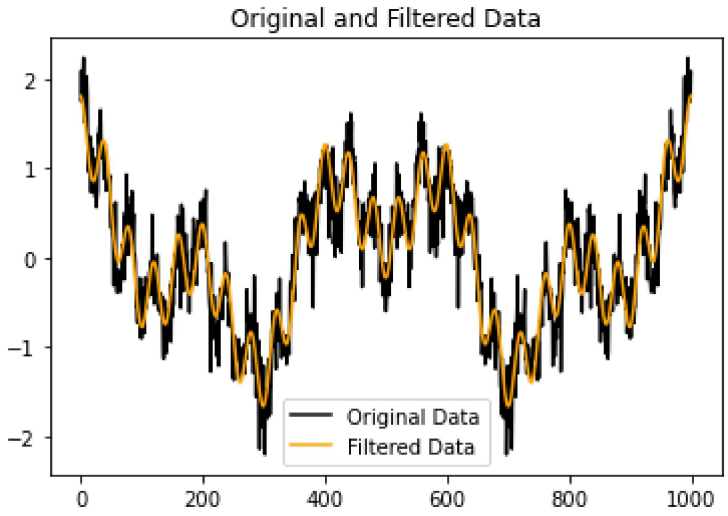
In [11]:

```
# part a and b
#data = np.loadtxt("C:\Users\vince\Downloads\hw8_prob4_data.txt", float)
#cs = scipy.fft.dct(ys)

file_path = r'C:\Users\vince\Downloads\hw8_prob4_data.txt'
with open(file_path, 'r') as file:
    lines = file.readlines()
    data = [float(line.strip()) for line in lines]

dct_result = dct(data, type=2, norm='ortho')

# Plot the original and transformed data
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(data, color = 'black')
plt.title('Original Data')

plt.subplot(1, 2, 2)
plt.plot(dct_result, color = 'orange')
plt.title('DCT Transformed Data')

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

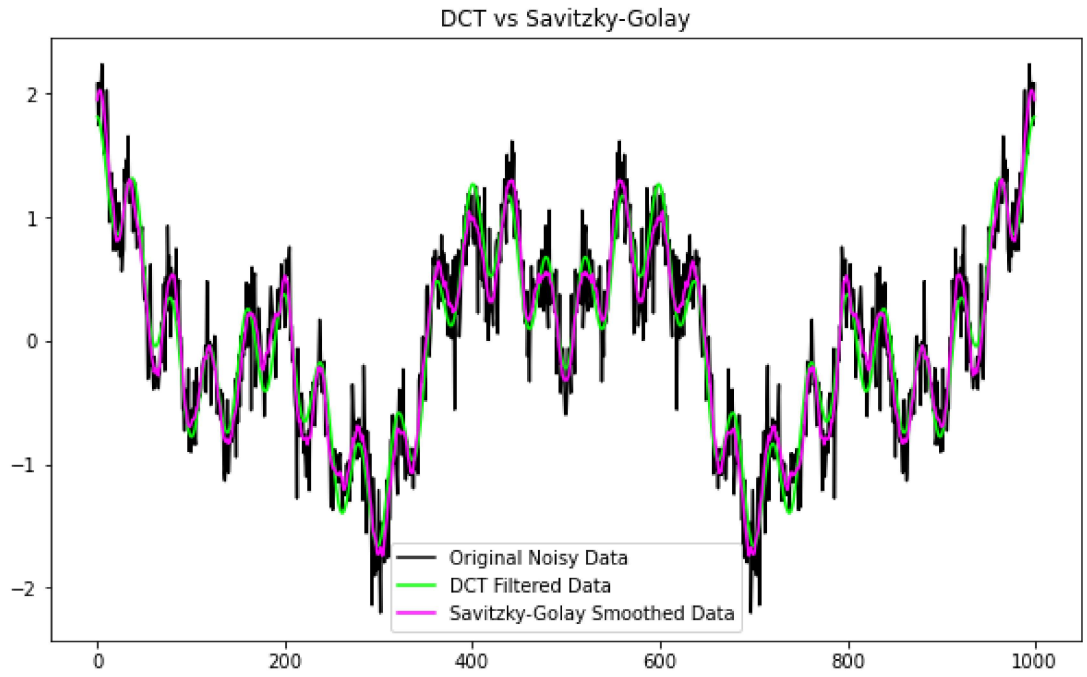


In [12]:

```
# part c
coefficients = fft(data)

# Set small coefficients to zero
coefficients[np.abs(coefficients) < 100] = 0

# Perform Inverse Fourier Transform
filtered_data = ifft(coefficients)

# Plot the original and filtered data
plt.plot(data, label='Original Data', color='black')

# Filtered Data
plt.plot(filtered_data.real, label='Filtered Data', color='orange')

plt.title('Original and Filtered Data')
plt.legend()
plt.show()
```



In [13]:
```
coefficients = fft(data)

# Set small coefficients to zero using DCT filtering
coefficients[np.abs(coefficients) < 100] = 0

# Perform Inverse Fourier Transform for DCT filtering
dct_filtered_data = ifft(coefficients).real

# Smooth the noisy data using Savitzky-Golay filter
window_size = 21 # Adjust as needed
poly_order = 3   # Adjust as needed
smoothed_data = savgol_filter(data, window_size, poly_order)

# Plot the data
plt.figure(figsize=(10, 6))

# Original Noisy Data
plt.plot(data, label='Original Noisy Data', color='black')

# DCT Filtered Data
plt.plot(dct_filtered_data, label='DCT Filtered Data', color='lime')

# Savitzky-Golay Smoothed Data
plt.plot(smoothed_data, label='Savitzky-Golay Smoothed Data', color='mag

plt.title('DCT vs Savitzky-Golay')
plt.legend()
plt.show()
```



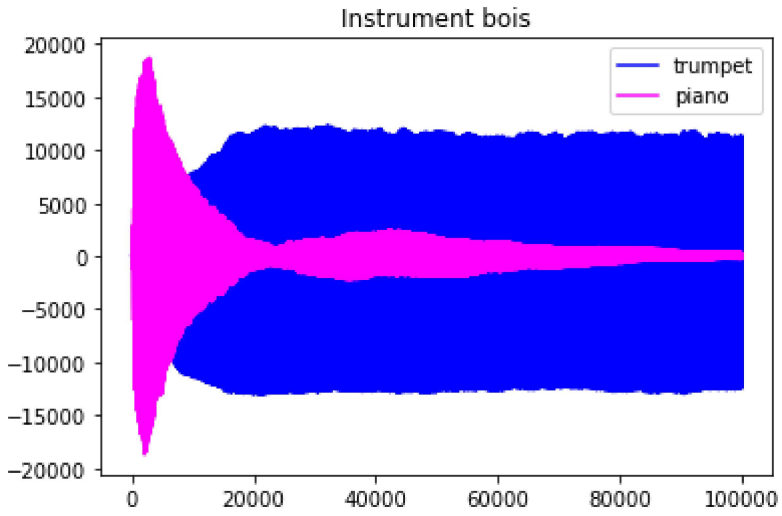
In [14]:
```
# problem results summary
print("DCT is applied in the frequency domain and is suitable for filter
```

```
DCT is applied in the frequency domain and is suitable for filtering ou
t high-frequency noise. It might be more effective in scenarios where t
he noise is represented by specific frequency components, such as spect
roscopy. Whereas the Savitsky-Golay method can be applied to situations
dealing with time or spatial domain. In this example, it appears that t
he Savitsky-Golay filter performs better than the DCT method, since Sav
itsky-Golay can be used for smoothing functions; however, there are add
itional bumps on each peak since a polynomial method is used to fit the
data.
```

Problem 5:

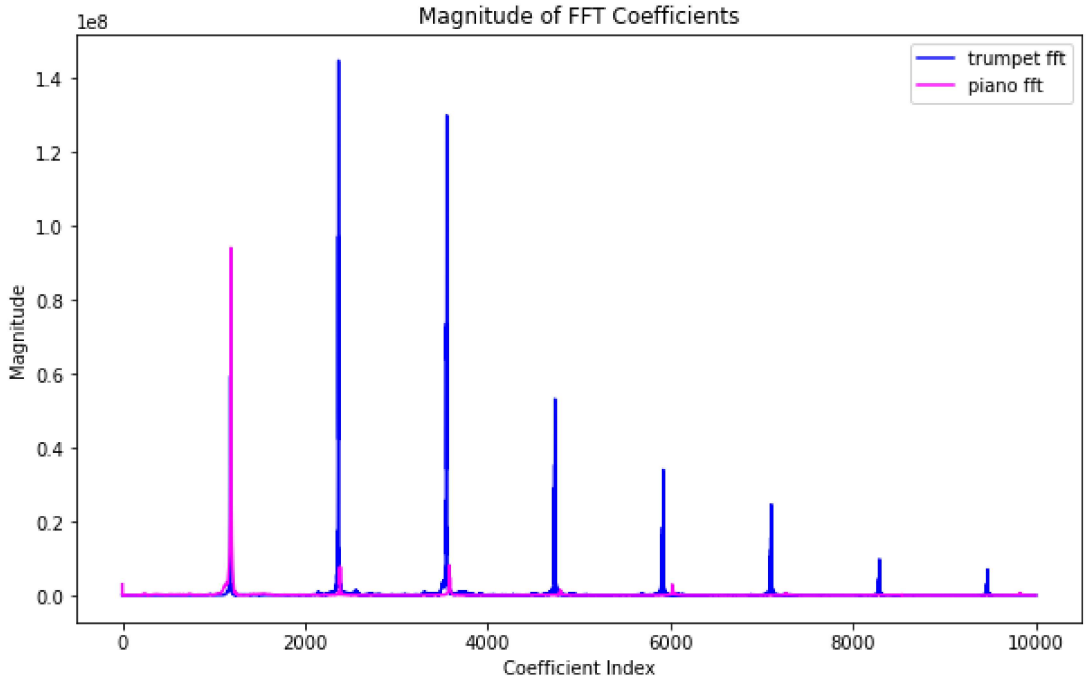
In [15]:
```
# part a
file_path = r'C:\Users\vince\Downloads\piano_waveform.txt'
with open(file_path, 'r') as file:
    lines = file.readlines()
    piano = [float(line.strip()) for line in lines]

file_path = r'C:\Users\vince\Downloads\trumpet_waveform.txt'
with open(file_path, 'r') as file:
    lines = file.readlines()
    trumpet = [float(line.strip()) for line in lines]

plt.plot(trumpet, color ='blue', label = 'trumpet')
plt.plot(piano, color = 'magenta', label = 'piano')
plt.title("Instrument bois")
plt.legend()
```

Out[15]:
```
<matplotlib.legend.Legend at 0x22189fdf1f0>
```



In [16]:

```
# part b
# Compute FFT for both waveforms
fft_trumpet = fft(trumpet)
fft_piano = fft(piano)

# Extract magnitude of the first 10,000 coefficients
magnitude1 = np.abs(fft_trumpet[:10000])
magnitude2 = np.abs(fft_piano[:10000])

# Plot the magnitudes
plt.figure(figsize=(10, 6))
plt.plot(magnitude1, label='trumpet fft', color = "blue")
plt.plot(magnitude2, label='piano fft', color = "magenta")
plt.title('Magnitude of FFT Coefficients')
plt.xlabel('Coefficient Index')
plt.ylabel('Magnitude')
plt.legend()
plt.show()
```



In [17]:

```
# part c
sampling_rate = 44100
num_samples = 10000

# Calculate length of interval
length_of_interval = num_samples / sampling_rate

# Calculate frequency of the note
frequency = 1 / length_of_interval

print(f" Length of Interval: {length_of_interval:.5f} seconds")
print(f" Frequency of the Note: {frequency:.2f} Hertz")

# 440 is a common reference point
reference_frequency = 440.0
semitone_ratio = 2 ** (1/12.0)
semitones_from_reference = 12 * np.log2(frequency / reference_frequency)


# Assuming A440 as reference
note_names = ['A', 'A#', 'B', 'C', 'C#', 'D', 'D#', 'E', 'F', 'F#', 'G',
note_index = round(semitones_from_reference) % 12
note_name = note_names[note_index]

print(f" Note: {note_name}")
```

```
 Length of Interval: 0.22676 seconds
 Frequency of the Note: 4.41 Hertz
 Note: C#
```