# 9 Nov 23 - Activity: Applying The Fast Fourier Transform

We were able to find the complex expansion coefficients for a signal using a Fourier series. However, this method is only applicable to periodic signals. Here we will introduce the Discrete Fourier Transform (DFT) and the Fast Fourier Transform (FFT) which can be used to find the complex expansion coefficients for any signal. The DFT is the discrete analog to our expansion approach, the FFT improves the algorithmic efficiency of the DFT, which we won’t describe.

Let’s remind ourselves of the Fourier series expansion for a periodic signal with some period .

import numpy as np  
import matplotlib.pyplot as plt  
import random

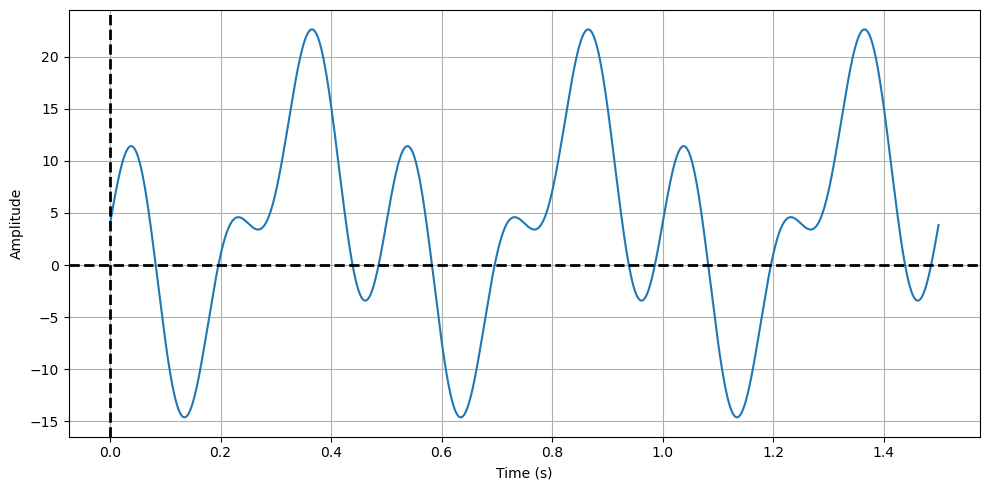
## A sinusoidal signal with a DC offset

In this example we will use a sinusoidal signal with a DC offset. The signal is defined as:

We create the signal a graph it.

def summed\_signal(t, A, f, C=0):  
 """Creates a signal with multiple frequencies."""  
 return A[0] \* np.sin(2 \* np.pi \* f[0]\*t) + A[1] \* np.sin(2 \* np.pi \* f[1]\*t) + A[2]\*np.sin(2 \* np.pi \* f[2]\*t) + C  
  
T0 = 0.5 # Signal period  
N = 1000 # Number of samples  
dt = T0/N # Time step  
t = np.arange(0, 3\*T0, dt) # Time points  
  
f = np.array([1/T0, 2/T0, 3/T0]) # Frequencies  
A = np.array([-10, 5, 8]) # Amplitudes  
C = 4 # DC Offset  
  
summed = summed\_signal(t, A, f, C)

plt.figure(figsize=(10, 5))  
  
plt.plot(t, summed)  
  
plt.axvline(x=0, color='k', linestyle='--', lw=2)  
plt.axhline(y=0, color='k', linestyle='--', lw=2)  
  
plt.xlabel('Time (s)')  
plt.ylabel('Amplitude')  
  
plt.grid()  
plt.tight\_layout()



png

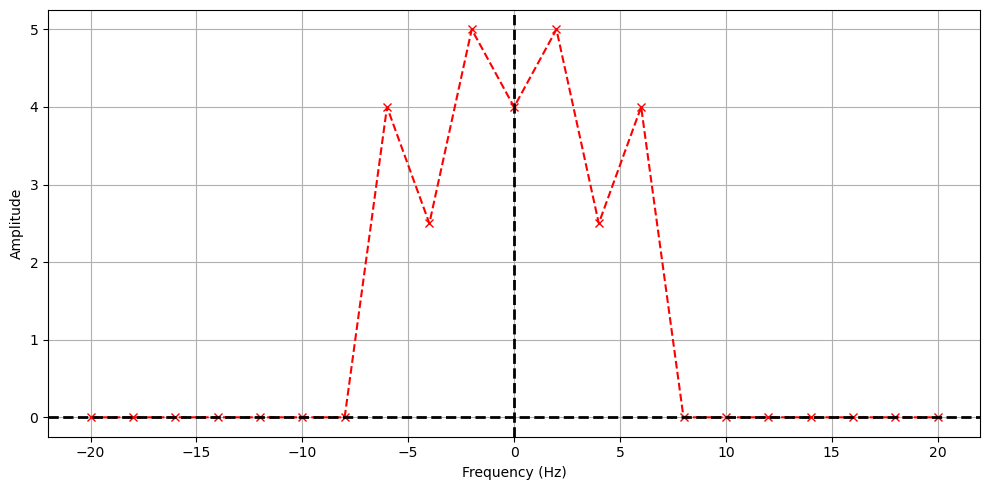
## Extracting the Fourier coefficients

Let’s extract the Fourier coefficients. We defined the base peridicity in the code above. If it was not changed it was 0.5 seconds. We will use this to define the base frequency of the signal Hz. We have two more parts of the signal from frequencies Hz and Hz. We will use the Fourier series expansion to find the complex expansion coefficients .

def compute\_cn(t\_, f, n):  
 """Computes the Fourier coefficient c\_n for a given time array.  
 Assumes that the time array is evenly spaced. f is the signal   
 and must have the same length as t."""  
 dt = t\_[1] - t\_[0]  
   
 return 1/T0 \* np.sum(f \* np.exp(-1j \* 2 \* np.pi \* n \* t\_/T0)) \* dt  
  
def compute\_power\_spectrum(t\_, f, N):  
 """Computes the power spectrum for a given time array and signal.  
 Returns the coefficients from -N,N."""  
 cn = np.zeros(2 \* N + 1, dtype=complex)  
 for n in range(-N , N + 1):  
 cn[n + N] = compute\_cn(t\_, f, n)  
 return cn

t\_ = np.arange(0, T0, dt) # Time points for a single period  
N = 10 # Number of coefficients (2N+1)  
freqs = np.arange(-N, N+1) / T0 # Frequencies  
  
cn = compute\_power\_spectrum(t\_, summed[:len(t\_)], N) # Compute the coefficients

plt.figure(figsize=(10, 5))  
  
plt.plot(freqs, np.abs(cn), '--rx')  
plt.xlabel('Frequency (Hz)')  
plt.ylabel('Amplitude')  
  
plt.axhline(y=0, color='k', linestyle='--', lw=2)  
plt.axvline(x=0, color='k', linestyle='--', lw=2)  
  
plt.tight\_layout()  
plt.grid()

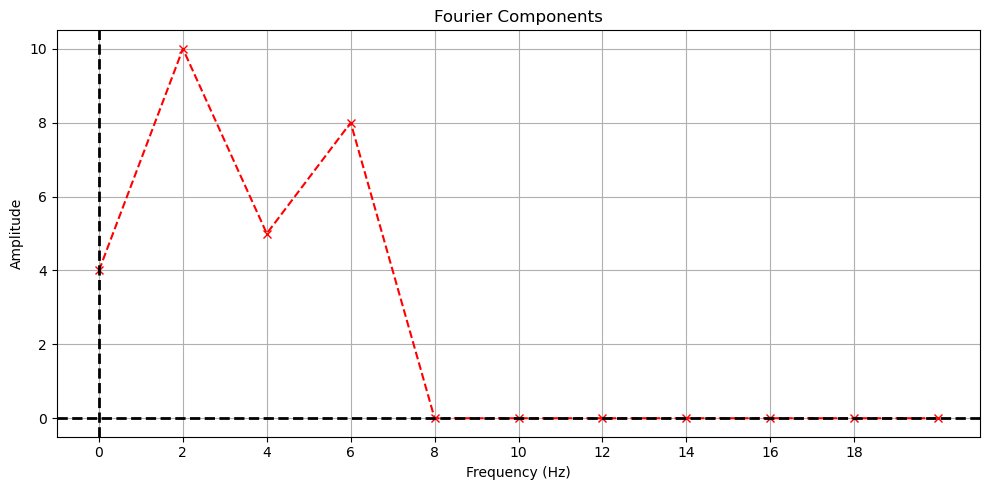


png

### Where are the coefficients?

We get both positive and negative frequencies. But notice that if we add the amplitudes of the matched positive and negative amplitudes, we get the value of the coefficients in our constructed signal. This is because the signal is real, so the coefficients are conjugate symmetric. We can use this to plot the coefficients in a more intuitive way. Notice the DC part of the signal is NOT conjugate symmetric, so we don’t add it to the negative frequency part.

plt.figure(figsize=(10, 5))  
  
x = np.insert(freqs[N+1:], 0, freqs[N])  
y = np.insert(2\*np.abs(cn[N+1:]), 0, np.abs(cn[N]))  
  
plt.plot(x, y, '--rx')  
  
plt.title('Fourier Components')  
plt.xlabel('Frequency (Hz)')  
plt.ylabel('Amplitude')  
  
plt.axhline(y=0, color='k', linestyle='--', lw=2)  
plt.axvline(x=0, color='k', linestyle='--', lw=2)  
plt.xticks(np.arange(0,20,2))  
  
plt.tight\_layout()  
plt.grid()

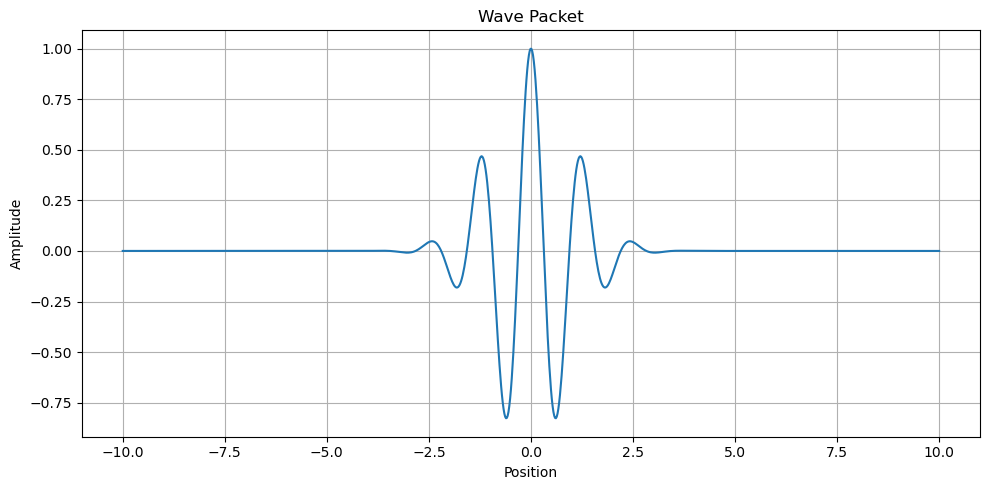


png

## The Discrete Fourier Transform

One of the issues about the decomposition we have done is that it assumes the signal is periodic. In practice, we often have signals that are not periodic. For example, consider the signal below, which is a wave packet that is not periodic. This signal is still of interest to us and can often contain useful physics. But the techniques we have used so far will not work for this signal.

# Define the wave packet parameters  
k0 = 5 # central wavenumber  
x0 = 0 # initial position  
spread = 1 # spread of the packet  
  
# Create a space vector  
x = np.linspace(-10, 10, 1000)  
  
# Define the wave packet function  
def wave\_packet(x, k0, x0, spread):  
 return np.exp(-0.5 \* ((x - x0) / spread) \*\* 2) \* np.cos(k0 \* x)  
  
# Calculate the wave packet  
psi = wave\_packet(x, k0, x0, spread)  
  
# Plot the wave packet  
plt.figure(figsize=(10, 5))  
plt.plot(x, psi)  
plt.title('Wave Packet')  
plt.xlabel('Position')  
plt.ylabel('Amplitude')  
plt.grid(True)  
plt.tight\_layout()



png

### Enter the Discrete Fourier Transform

The [Discrete Fourier Transform (DFT)](https://en.wikipedia.org/wiki/Discrete_Fourier_transform) is a way to decompose a signal into a sum of sinusoids, but it does not require the signal to be periodic. You can think of it as a discretization of the processes we developed earlier. The expansion of the signal is still as it was before, but we understand it represents a finite number of samples of the signal:

However, we now understand that the signal is only defined at discrete times where is the time between samples. We can then write the Fourier coefficients as:

where is the value of the signal at time and is the total time of the signal.

### Assumptions

In doing this, we are not creating a tool that will always work. We note a few important assumptions and caveats:

1. **Periodicity**: The DFT assumes that the input signal is periodic and that the period is exactly the length of the sample window. This means that the finite sequence of data points provided to the DFT is one period of a periodic signal that repeats indefinitely.
2. **Discrete Samples**: The signal being transformed is assumed to be sampled at discrete intervals. The DFT operates on these discrete samples and does not account for any data between the samples.
3. **Finite Duration**: The DFT is designed to handle signals of finite duration. The signal is assumed to have a finite number of samples, which is the number of points in the DFT.
4. **Equally-Spaced Samples**: The samples are assumed to be evenly spaced in time (or space, depending on the context of the signal). This uniform spacing is critical because non-uniformly spaced samples would require a different transform, such as the Non-uniform Discrete Fourier Transform (NDFT).

## The Fast Fourier Transform

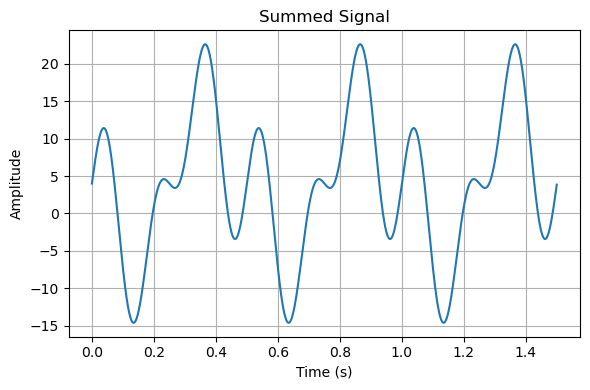
While we can use the DFT directly, the FFT is more efficient and has become the standard way to perform these analyses. The code to have numpy.fft perform the FFT is pretty simple. But there’s many things to understand about the FFT and a variety of ways to produce spurious and incoherent results. Let’s use the simple signal we have above to make go through an analysis.

### The FFT in numpy.fft

We start by importing the libraries. Then we need to:

1. Define the time array
2. Define the signal
3. Perform the FFT using numpy.fft.fft (*We imported numpy.fft below, so you can call fft directly.*)
4. Plot the coefficients
5. Plot the power spectrum

from numpy.fft import fft, fftfreq  
  
plt.figure(figsize=(6, 4))  
plt.plot(t, summed)  
  
plt.title('Summed Signal')  
plt.xlabel('Time (s)')  
plt.ylabel('Amplitude')  
  
plt.tight\_layout()  
plt.grid()



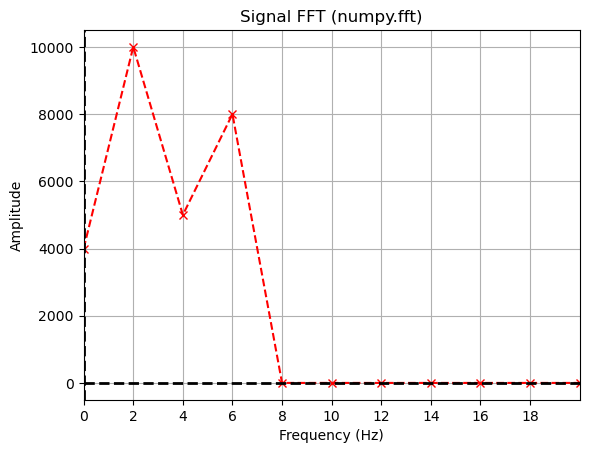
png

### Computing and plotting Fourier Coefficients

We now use fft to calculate the Fourier transform of the signal and fftfreq to calculate the frequencies associated with the transform. This is a standard way to do this analysis, but the most common mistakes involve the frequencies. Because it is a real signal, we don’t worry about the negative frequencies, they are duplicate information. The amplitudes of the non DC frequencies are doubled like before to illustrate the Fourier coefficients.

t\_ = np.arange(0, T0, dt) # Time points for a single period  
one\_cycle = summed[:len(t\_)] # One cycle of the summed signal  
N = int(T0/dt/2) # Maximum number of coefficients  
  
fft\_summed = fft(one\_cycle)  
freqs\_array = fftfreq(len(one\_cycle), dt)  
  
x = freqs\_array[:N]  
y = np.insert(2\*np.abs(fft\_summed[1:N]), 0, np.abs(fft\_summed[0]))

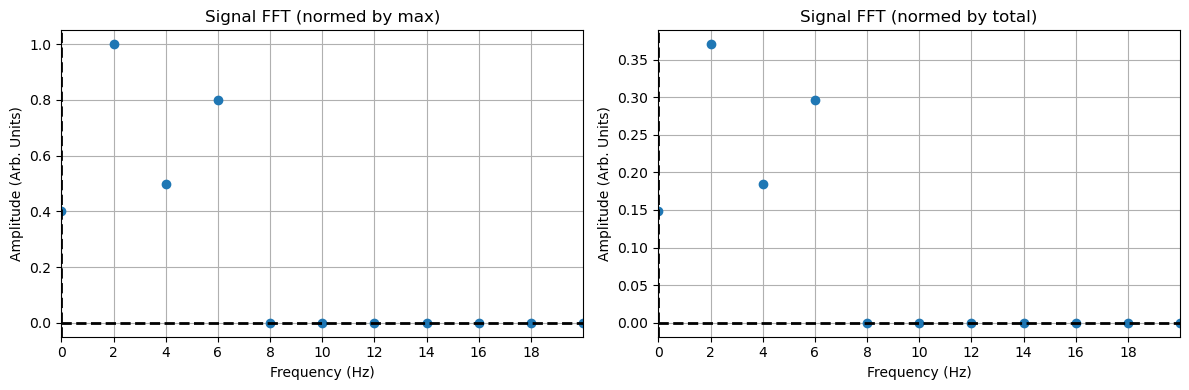
plt.plot(x, y, '--rx')  
  
plt.title('Signal FFT (numpy.fft)')  
plt.xlabel('Frequency (Hz)')  
plt.ylabel('Amplitude')  
  
plt.axhline(y=0, color='k', linestyle='--', lw=2)  
plt.axvline(x=0, color='k', linestyle='--', lw=2)  
  
plt.xlim(0, 20)  
plt.xticks(np.arange(0,20,2))  
  
plt.grid()



png

Notice the FFT looks precisely the same as before. But the Amplitudes are scaled differently. The total amplitude is not the same, but the relative values are still meaningful. If the maximum amplitude is scaled to the amplitude we had before, then the signal would look the same. However, it’s typical to instead plot these scaled in some way. The most common way is to either scale them by the maximum amplitude or the sum of all the amplitudes. Each have their own purpose and meaning. Below we plot them both for comparison.

plt.figure(figsize=(12, 4))  
  
plt.subplot(121)  
plt.plot(x, y/np.max(y), 'o')  
plt.xlim(0,20)  
plt.xticks(np.arange(0,20,2))  
  
plt.title('Signal FFT (normed by max)')  
plt.xlabel('Frequency (Hz)')  
plt.ylabel('Amplitude (Arb. Units)')  
  
plt.axhline(y=0, color='k', linestyle='--', lw=2)  
plt.axvline(x=0, color='k', linestyle='--', lw=2)  
  
plt.grid()  
  
plt.subplot(122)  
plt.plot(x, np.abs(y)/np.sum(y), 'o')  
plt.xlim(0,20)  
plt.xticks(np.arange(0,20,2))  
  
plt.title('Signal FFT (normed by total)')  
plt.xlabel('Frequency (Hz)')  
plt.ylabel('Amplitude (Arb. Units)')  
  
plt.axhline(y=0, color='k', linestyle='--', lw=2)  
plt.axvline(x=0, color='k', linestyle='--', lw=2)  
  
plt.tight\_layout()  
plt.grid()



png

### Investigating the FFT

Now that you have developed some code and some conceptual understanding of the FFT process, let’s investigate some of the more nuanced aspects.

**✅ Do this**

1. Complete the power spectrum analysis for the following signals that you construct:
   * Your signals have two frequencies that are close together (competing frequencies)
   * Your signal has frequencies that are not integer multiples of the sampling rate (aliasing)
2. Use fft to do these analyses and plot the spectra.
3. For the first case, what seems to be needed to resolve the two frequencies?
4. For the second case, what happens when the frequency is not an integer multiple of the sampling rate? What does this mean when signals have two frequencies that are not integer multiples of each other or the sampling rate?

## your code here