HW4: Discrete Fourier Tranform

Austin Gardiner

September 2025

1 DFT Code

As this assignment follows the same process as the previous assignment, I was able to leverage the code written for that assignment. To improve efficiency, I moved the *plot_fft_magnitutes*() and *compute_fft*() functions to a Class and renamed them to be *plot_ft*() and *ft_routine*(). Having these functions in a class allows all functions to access the velocity data, which is now saved to the object, and a class variable called *ft.type*, which can hold a value of FFT or DFT. Now, in the *ft_routine*() function, instead of using *np.fft.fft*() to calculate the Fourier Transform, it instead calls a new function *compute_ft*(). *compute_ft*() reads the value of *ft.type* and then returns the FT magnitudes using either *np.fft.fft*() or a self built DFT.

```
def compute_ft(self, data=None):
    if data is None:
        data = self.data
    if self.ft_type == 'FFT':
        return np.fft.fft(data)
    elif self.ft_type == 'DFT':
        return self.dft(data)
```

For the DFT function, I originally wrote it using two for loops. The outer for loop determines which cell of the transformed values array, a, is being calculated, and the inner for loop calculates the portion of eq. 1 inside the sum and adds it to a sum variable created in the outer for loop. However, running this code was extremely inefficient. It involves multiple operations in each level of the for loop and completes all operations in serial. To complete the DFT calculations for the 16,384 record size, the code ran for over 18 minutes. The whole data set was even worse and required over 2 hours.

$$a(n) = \sum_{m=1}^{N-1} A(m) e^{-2\pi i n m/N} \tag{1}$$

To improve the efficiency of the code, I looked at ways to use linear algebra, which allows for parallel computations. Looking at eq. 1, the right side of the equation can be calculated using a dot product between two vectors. This dot product is written in eq. 2, where each value of the E vector is calculated using eq. 3.

$$\sum_{m=1}^{N-1} A(m) e^{-2\pi i n m/N} = A \cdot E \tag{2}$$

$$E(i) = e^{-2\pi i n m/N} \tag{3}$$

To improve processing speed, numpy arrays are prepopulated in each step of the code. Additionally, the arrays are specified to use complex numbers. In my original implementation, I did not specify the dtype=complex and this resulted in the complex portions calculated in 1 to be discarded. While the resulting curves still followed the same shape, the magnitude was off by nearly one order of magnitude. The DFT code is shown below.

```
def dft(self, x):
    # Compute the Discrete Fourier Transform (DFT)
    N = len(x)
    n = np.arange(N)
    A = np.arange(N, dtype=complex)
    for k in range(N):
        X = np.array(x, dtype=complex)
        E = np.arange(N, dtype=complex)
        E = E.reshape((N,1))
        E = np.exp(-2j * np.pi * k * n / N)
        A[k] = np.dot(X, E)
    return A
```

2 Timing

To calculate FT runtime, I calculate the start time before entering the *ft_routine*() function and subtract it from the end time which is determined after the *ft_routine*() completes. For windowed FT calculations with record sizes smaller than the set size, this runtime includes FT calculations for all windows, rather than

Record Size:	256	2,048	16,384	100,000
FFT:	0.0031	0.0017	0.0020	0.0099
DFT:	0.7083	3.4618	25.7963	181.6797

Table 1: Average FT runtimes

a single window. After running the code on all datasets for all record sizes, I averaged the runtime for each method and record size and recorded the average in table 1, with runtime in seconds.

As expected, FFT is significantly faster than DFT. However, with the more efficient DFT code, I was able to complete the calculations on all datasets. With the previous iteration of the code, this would have taken a full day to run. With DFT runtimes expected to scale with N^2, the runtime for larger record sizes should increase dramatically for the DFT method and this is seen in the results. Interestingly, the fastest runtime came from FFT with a 2048 record size. I believe this is due to the overhead required from running multiple windows being larger than the efficiency gains of smaller FFT windows. These averaged runtimes are plotted in Fig. 1 on a log-log plot.

To test the complexity of the FFT and DFT functions, I divided the averaged runtimes by the expected complexity. For FFT I divided each runtime by $NlogN$, where N is the record size. For DFT I divided each runtime by N^2. These ratios are put in table 2. If the runtime scales proportional to complexity, which is proportional to the number of operations, then these ratios should be the same for FFT and DFT results.

Record Size:	256	2,048	16,384	100,000
FFT:	5.08729E-06	2.547E-07	2.84383E-08	1.98909E-08
DFT:	1.08077E-05	8.25347E-07	9.60986E-08	1.8168E-08

Table 2: Runtimes divided by expected complexity

While there is a bit of a difference in these ratios, the results are very close. In particular, for the full data set, the complexity ratios are nearly the same. This suggests that these computation methods are achieving their expected complexities.

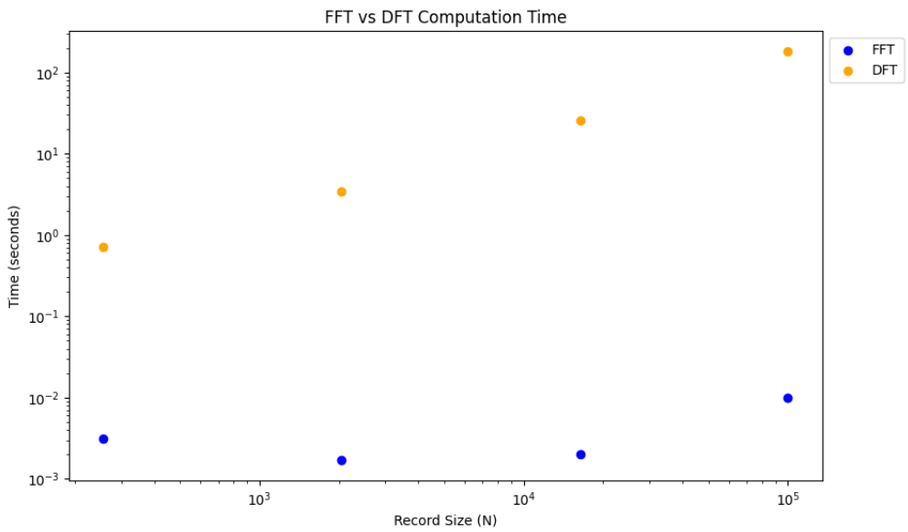

Figure 1: FFT and DFT averaged runtimes

3 Plots

The following plots are included to show that the results obtained by the FFT function and the DFT function are identical. In all plots, the FFT results are plotted with solid lines and DFT results are plotted with dashed lines.

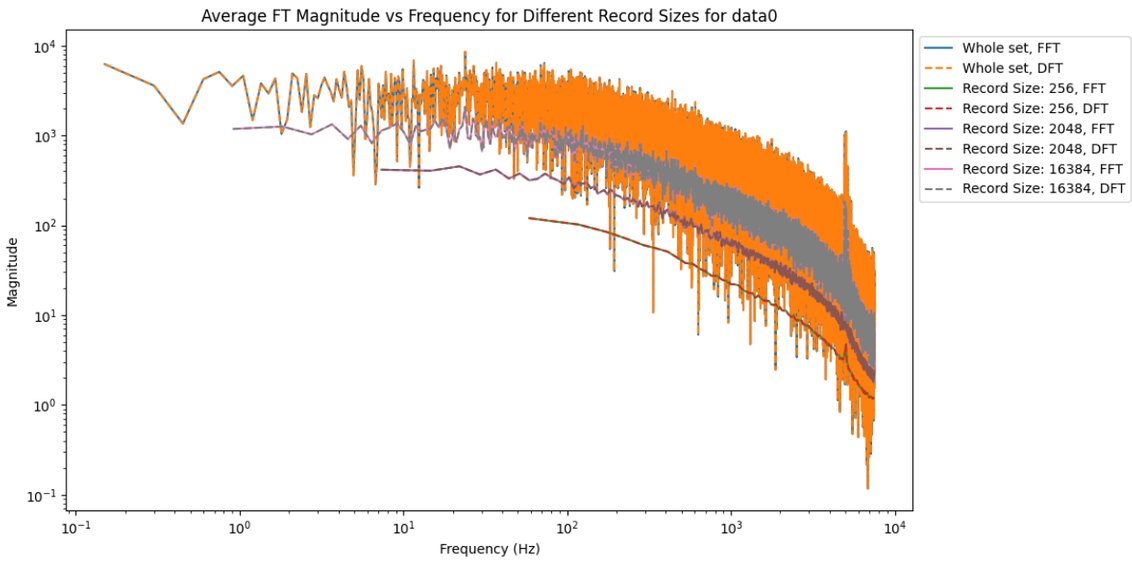

Figure 2: FT Curves computed using FFT and DFT for data0

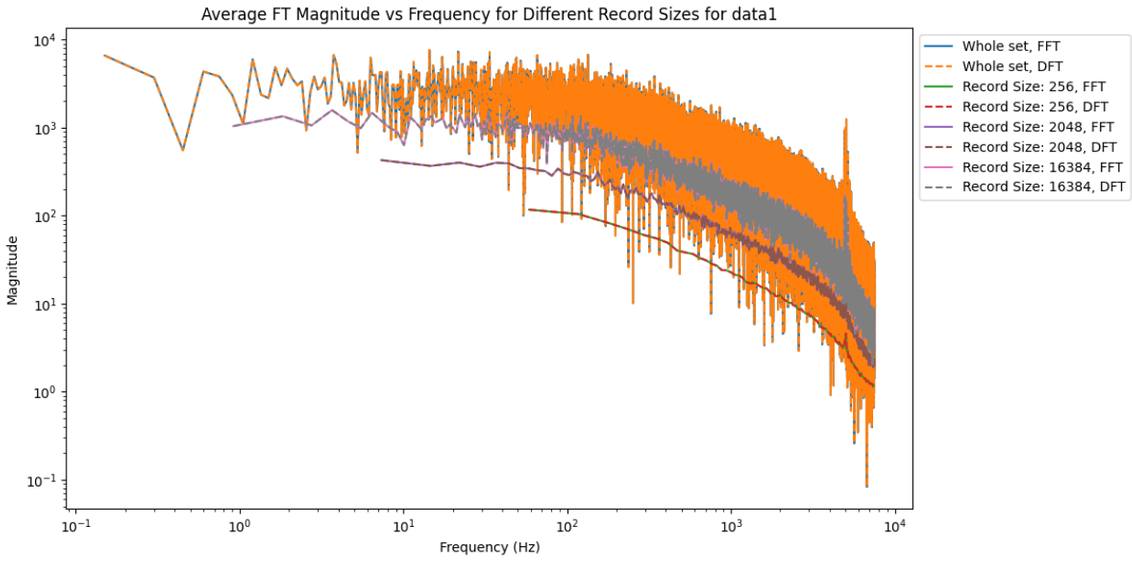

Figure 3: FT Curves computed using FFT and DFT for data1

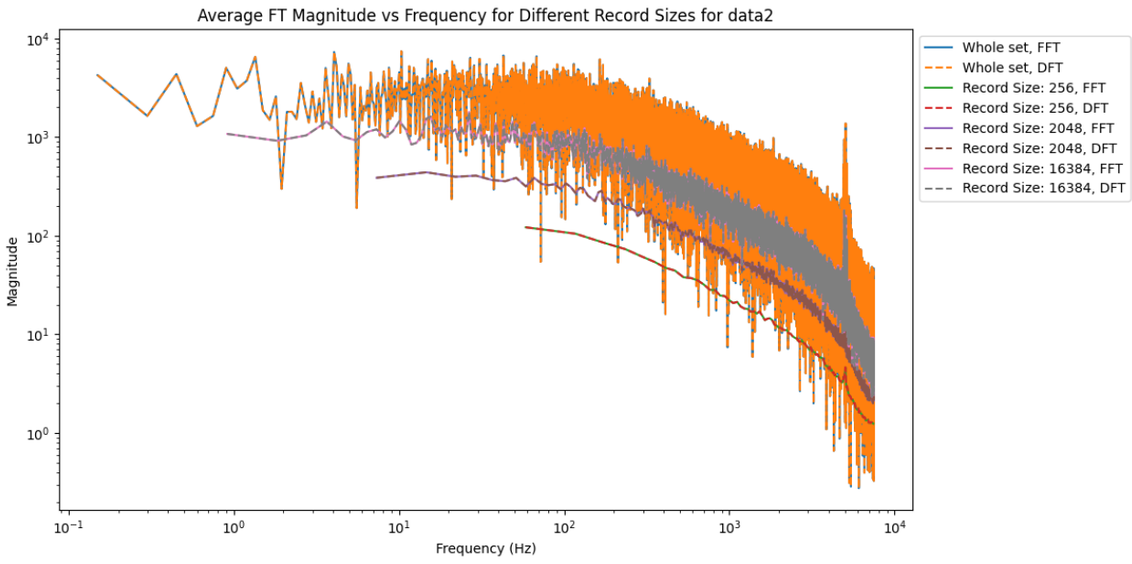

Figure 4: FT Curves computed using FFT and DFT for data2

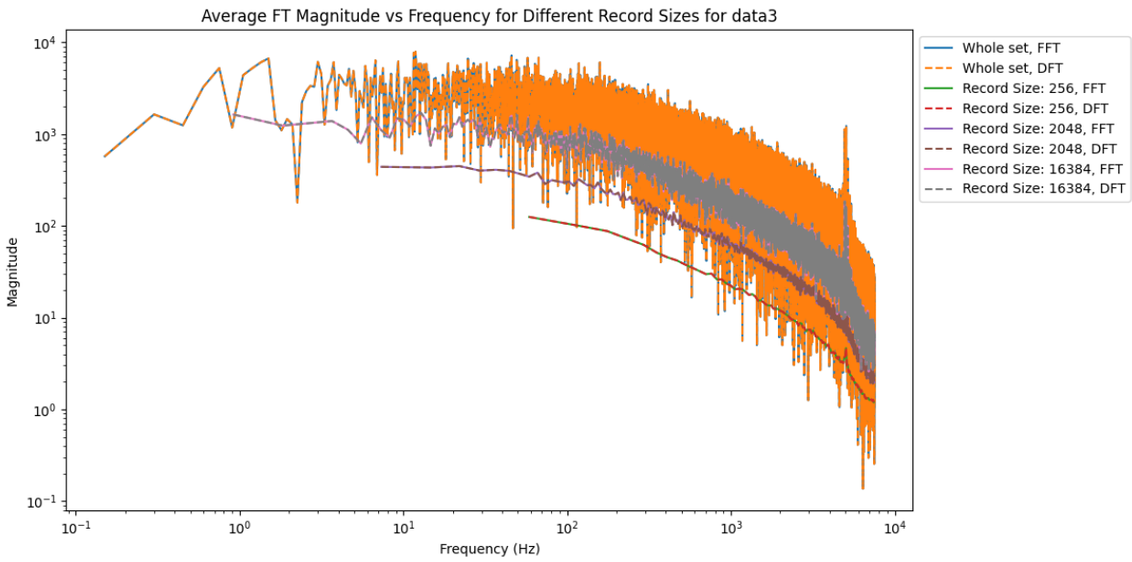

Figure 5: FT Curves computed using FFT and DFT for data3

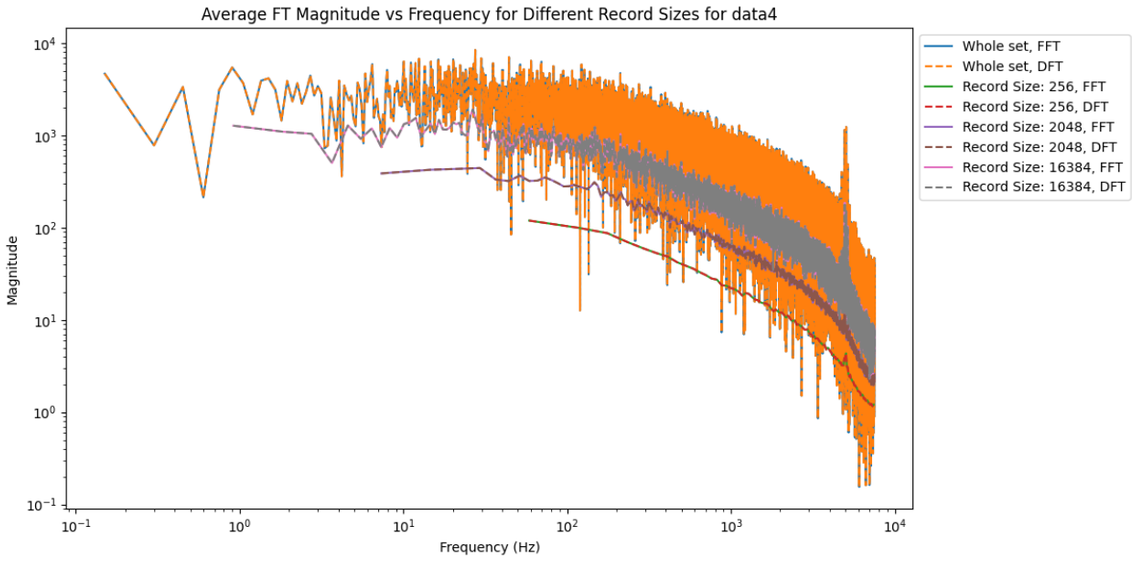

Figure 6: FT Curves computed using FFT and DFT for data4

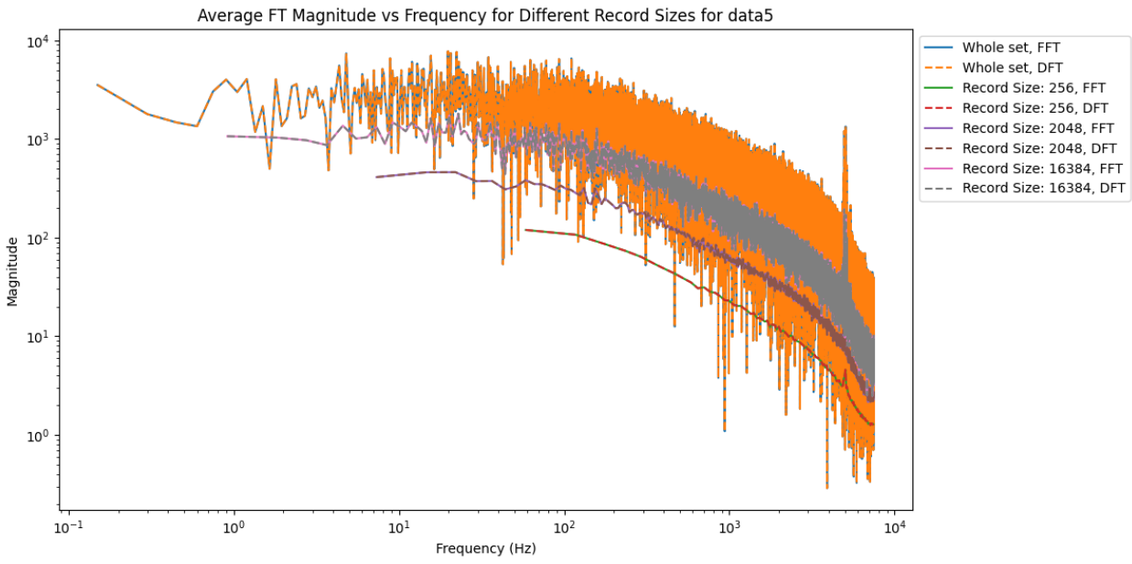

Figure 7: FT Curves computed using FFT and DFT for data5

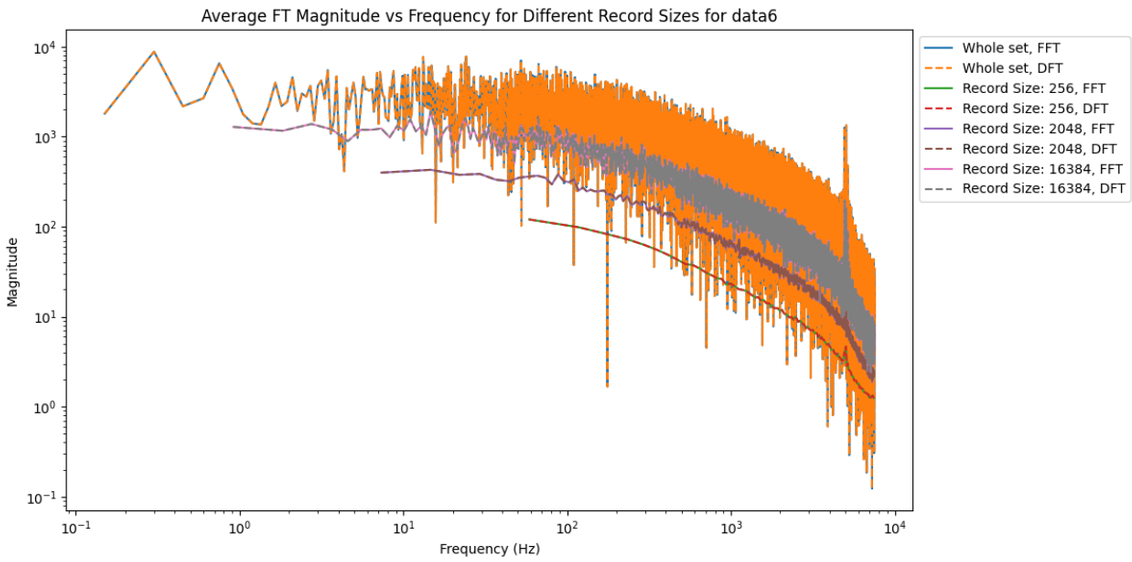

Figure 8: FT Curves computed using FFT and DFT for data6

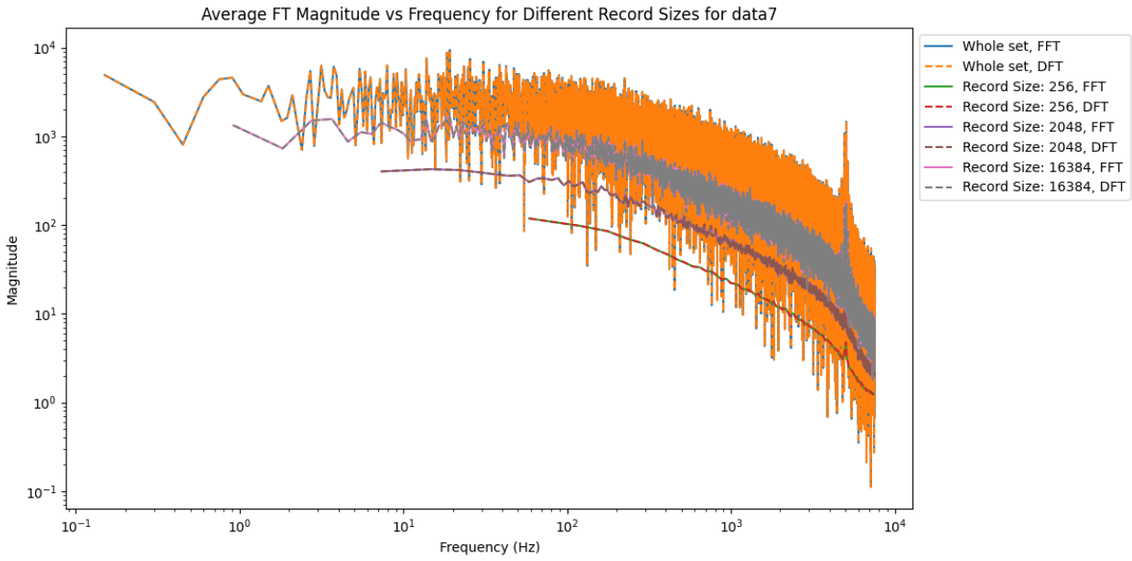

Figure 9: FT Curves computed using FFT and DFT for data7

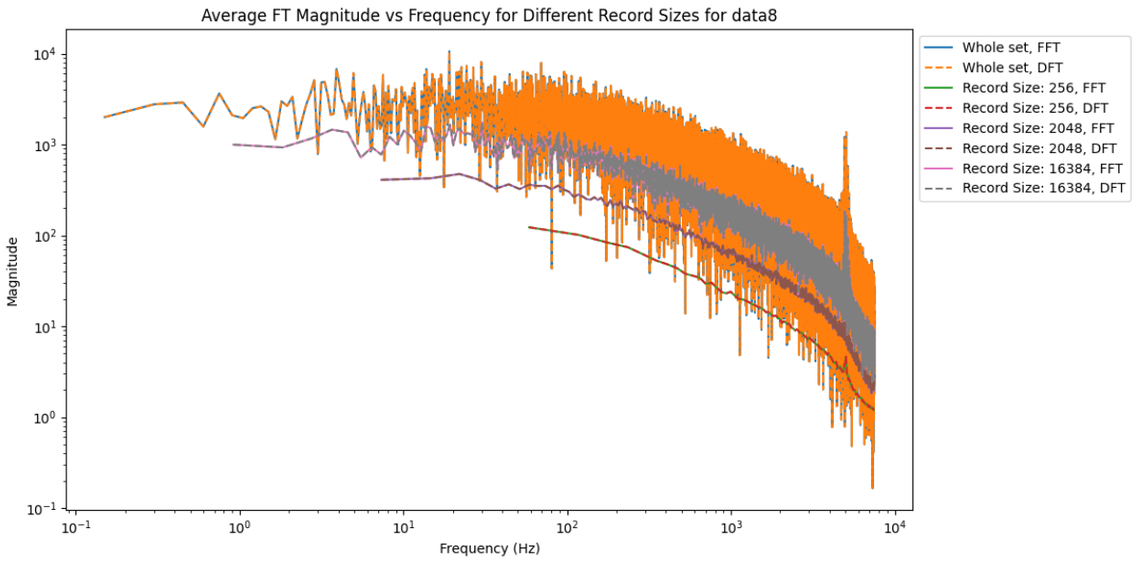

Figure 10: FT Curves computed using FFT and DFT for data8

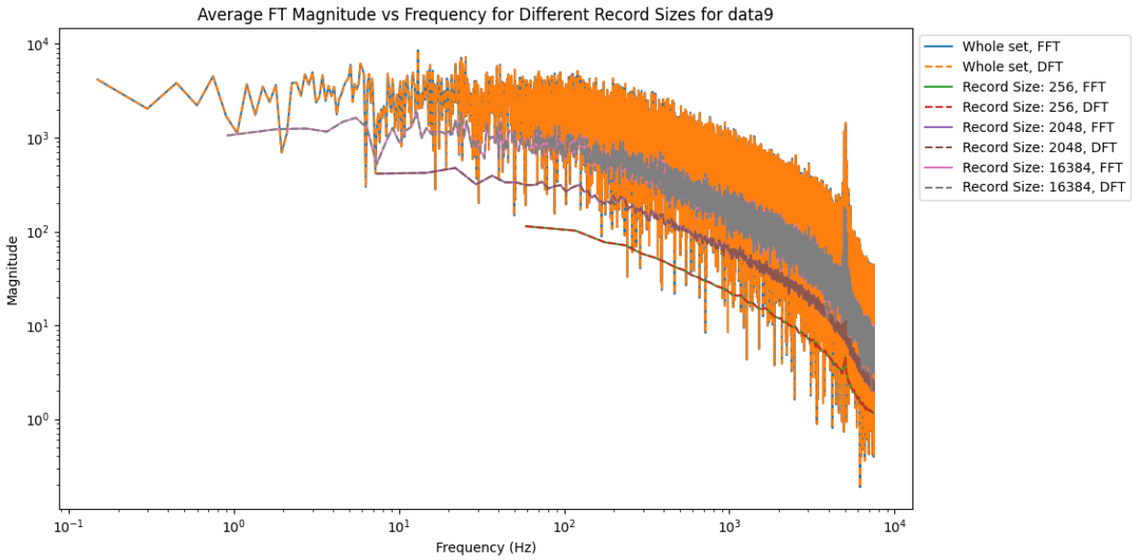

Figure 11: FT Curves computed using FFT and DFT for data9

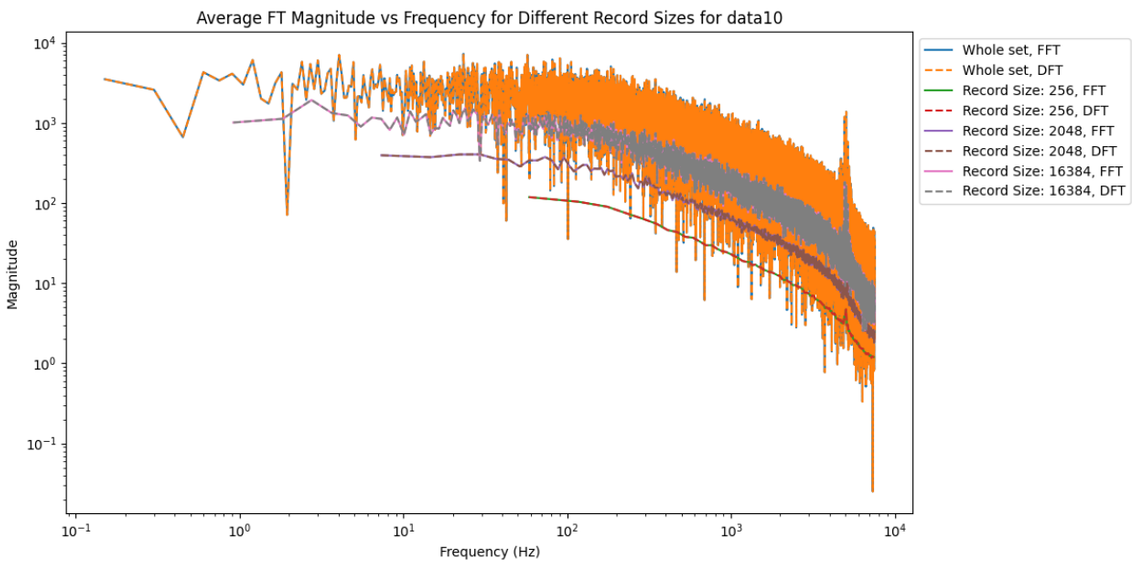

Figure 12: FT Curves computed using FFT and DFT for data10