VIBRATION ANALYSIS WITH FFT

OVERVIEW:

Any waveform is just the sum of a series of simple sinusoids of different frequencies, amplitudes, and phases. A Fourier series is that series of sine waves; and we use Fourier analysis or spectrum analysis to deconstruct a signal into its individual sine wave components. The result is acceleration/vibration amplitude as a function of frequency, which lets us perform analysis in the frequency domain (or spectrum) to gain a deeper understanding of our vibration profile.

SAMPLING THEOREM:

Nyquist sampling theorem states that, to sample an analog signal, we must take samples at a frequency which is at least twice the frequency of the message signal. Otherwise if the sampling frequency (Fs) is lesser than message frequency(Fm) the sampled values will contain less data, and reconstruction of the original signal is impossible.

So, to sample a signal of frequency $Fm = 50Hz$ we must sample the signal at $Fs \geq 2 \times Fm = 100Hz$

Conversely, if the sampling frequency is $50Hz$, then we cannot analyze the signals which has a frequency higher than $25Hz$.

FAST FOURIER TRANSFORM:

A fast Fourier transform (FFT) is just a Discrete Fourier Transform (DFT) using a more efficient algorithm that takes advantage of the symmetry in sinusoidal waves. The FFT requires the signal length to be in some power of 2, so the signal can be divided into halves and the function can be implemented recursively. This dramatically improves processing speed; if N is the length of the signal, a DFT needs N^2 operations while an FFT needs $N \log_2(N)$ operations.

PREREQUISITES:

The following are the requirements for implementing the FFT in Python3:

The required python libraries and modules can be installed by using $\{pip\ install\ "module\ name"\}$

- Pandas for data manipulation
  - Importing and exporting excel sheets
  - Converting them to python list
- Cmath module for complex number analysis:
  - Importing constants like e, i, π
- Math module for:
  - $\log_2 x$, ceil functions
- Bokeh module:
  - For plotting the output result and saving it as HTML file
- 'Xlsxwriter' module must be installed to export the data into excel sheet

```
import pandas as pd
from cmath import pi, exp
from math import log2, ceil
from bokeh.plotting import figure, show, output_file
from bokeh.models import Range1d
```

INPUT DATA FORMAT:

The data is read from the excel sheet using pandas library from a predefined location. The input_data_path and the output_data_path contains the location of input and output files respectively.

```
input_data_path = "Data/Vibration Data.xlsx"
output_data_path = "Data/Fourier transformed Vibration Data.xlsx"
vibration_data = pd.read_excel(input_data_path)
```

The data is read and stored as pandas data frame. We need the individual axis data and convert them into a list structure so that we can use the FFT function on it.

```
# Separating the values of X and Y axis data
vibraX = pd.DataFrame(vibration_data, columns=['VibraX'])
vibraY = pd.DataFrame(vibration_data, columns=['VibraY'])
vibraY = vibraY.values.tolist()
vibraX = vibraX.values.tolist()
```

The above vibraY is actually converted to a nested list, i.e. instead of having [1, 2, 3, 4, 5, . . .] we are having [[1], [2], [3], [4], [5], ……]. The nesting of lists is removed by iterating through each of the item and then appending to a separate list.

```
x_list, y_list = [], []
for i in range(len(vibraY)):
    x_list.append(vibraX[i][0])
    y_list.append(vibraY[i][0])
```

ZERO PADDING:

For implementing Cooley-Tukey's FFT algorithm, the input signal must of length of some power of 2(e.g. 1024 values, 4096 values etc.)

If the signal is not of the required form, it can be padded with some extra zeros at the end so that the signal is now of the required 2^n length.

To add extra zeros, we must first find which is the next nearest power of 2 to the given signal length. For that we can use $[log_2(x)]$. The log function returns the nearest power of 2 and the ceil function $[x]$ returns the next higher power of 2.

After finding the nearest power, we can subtract the signal length to find the number of zeros we need to add.

Ceil and $\log_2$ are imported from the math module

```
def nxt_power_2(x):
    return 2**ceil(log2(x))

def zero_padding(array):
    nextpwr = nxt_power_2(len(array))
    lengthOfArray = len(array)
    if nextpwr != lengthOfArray:
        for j in range(nextpwr-lengthOfArray):
            array.append(0)
```

The function is applied on the x_list and y_list

```
zero_padding(x_list)
zero_padding(y_list)
```

LIMITED NUMBER OF SAMPLES:

Sometimes, instead of padding the samples with zeros at end, we can take the first 2^n samples for calculation. E.g. If there is a continuous data which is being read, we don't have to zero pad, we can take sets of data for FFT. But the number of terms must be decided, it is always better to have a greater number of samples for a better resolution but that shouldn't increase the delay to receive the data.

```
length_fixed = 300    # To limit the number or samples
```

The different outputs for varied sample number is attached at the end.

REMOVING DC OFFSET:

Usually there is a peak at zero Hz after applying FFT, this is not of any value but just represents the offset in the given data, this peak can be removed by subtracting the average of the sample data with each of the sample.

```
# To calculate the sum of the data and then average
sum_x, sum_y = 0, 0
for i in range(length_fixed):
    sum_y += vibraY[i][0]
    sum_x += vibraX[i][0]

mean_x = sum_x / length_fixed
mean_y = sum_y / length_fixed


# Mean is subtracted from the data to remove the DC offset (Peak
```

```
appearing at 0Hz, though there is no DC component)
x_list, y_list = [], []
for i in range(length_fixed):
    x_list.append(vibraX[i][0] - mean_x)
    y_list.append(vibraY[i][0] - mean_y)
```

FFT FUNCTION:

The FFT function is implemented using the Cooley-Tukey's algorithm (Reference: Click here). This algorithm exploits the symmetry in the sine waves and uses recursion to reduce the time complexity.

The algorithm divides the signal into 2 halves of even and odd terms and computes the FFT of these two separate terms. This can be recursively done again and again to break down the list into smaller segments. After computing individually, the terms can be added to get the FFT of the whole signal.

The complexity is reduced from the original N^2 to $N \log(n)$ due to recursive implementation.

Here is a python implementation of the algorithm (For other languages: Click Here)

```
def FFT(x):
    N = len(x)
    if N <= 1:
        return x
    even_terms = FFT(x[0::2])
    odd_terms = FFT(x[1::2])
    T = [exp(-2j * pi * k / N) * odd_terms[k] for k in range(N // 2)]
    return [even_terms[k] + T[k] for k in range(N // 2)] + \
           [even_terms[k] - T[k] for k in range(N // 2)]
```

This returns unnormalized values of the signal which has very large amplitude, so each of the value is divided by the length of the signal to normalize the amplitude.

Also, the function gives complex numbers as the output, so we must take the absolute value of the signal to plot and analyze.

```
fourier_x_list = FFT(x_list)
abs_fourier_x = [abs(x)/len(fourier_x_list) for x in fourier_x_list]

fourier_y_list = FFT(y_list)
abs_fourier_y = [abs(y)/len(fourier_y_list) for y in fourier_y_list]
```

SIGNAL PARAMETERS:

For the given data, the sampling frequency $Fs = 1Hz$ which makes it impossible to detect the frequencies higher than $0.5Hz$. To detect higher frequencies, we must sample the signal according to the $Sampling\ Theorem$.

To plot the signal, we need the frequency range which corresponds to the peaks. The range can be constructed from the Sampling frequency and the number of samples.

Constructing an array of same length of signal and then dividing by the Total time gives a range till sampling frequency Fs, but we can exclude the second half since the sampling theorem allows till $\frac{Fs}{2}$.

```
Fs = 1                          # Sampling Frequency of the signal
n = len(x_list)                 # Number of samples
k = [i for i in range(n)]
T = n/Fs                        # Total time = No of sample/Sample frequency
frq = [x / T for x in k]        # Frequency range
frq = frq[:len(frq)//2]         # Only first half is taken
```

SYMMETRY IN OUTPUT:

The output gives a signal which is symmetrical, each of the peak is repeated twice after the half of the sampling frequency. i.e. If the sampling frequency is 50Hz, each peak less than 25Hz will repeat again after 25Hz. If there is a peak at 10Hz, then the same peak will arise at 40Hz (50-10).
Thus, we must exclude the later half of the signal before exporting and plotting.

```
final_fourier_x = abs_fourier_x[:len(abs_fourier_x)//2]
final_fourier_y = abs_fourier_y[:len(abs_fourier_y)//2]
```

OUTPUT AS EXCEL SHEET:

The normalized Fourier values can be exported to a excel sheet for other references.
Python cannot export natively to excel sheets, so we use pandas. The required lists are converted into pandas data frame and then using xlsxwriter we can export the data into excel.

```
outX = pd.DataFrame({"Fourier_X": final_fourier_x})
outY = pd.DataFrame({"Fourier_Y": final_fourier_y})

writer = pd.ExcelWriter(output_data_path, engine='xlsxwriter')

outX.to_excel(writer, sheet_name="sheet1")
outY.to_excel(writer, startcol=2, index=False, sheet_name='sheet1')
writer.save()
```

GRAPHICAL OUTPUT:

From the $bokeh.plotting$ module functions like $figure, show, output\_file$ are imported at the start.

The bokeh module allows us to export the graphical output into an HTML file, the name of the output file is mentioned. For plotting the data, a figure needs to be created first with a specified title, height

and width. Then the plot is added to the figure, the X and Y axis data is given as an array. The show() function displays the plot in a browser window.

```
output_file("Graph/FFT_own_x.html")
plot = figure(title="Vibration X fft - {}
samples".format(length_fixed),
              x_axis_label='Frequency (Hz)',
              y_axis_label='Amplitude (g)',
              y_range=Range1d(-0.005, 0.1),
              plot_width=1500,
              plot_height=700)

plot.line(frq, final_fourier_x)
show(plot)

output_file("Graph/FFT_own_y.html")
plot = figure(title="Vibration Y fft - {}
samples".format(length_fixed),
              x_axis_label='Frequency (Hz)',
              y_axis_label='Amplitude (g)',
              y_range=Range1d(-0.005, 0.1),
              plot_width=1500,
              plot_height=700)
```

WHOLE CODE:

```
import pandas as pd
from cmath import pi, exp
from math import log2, ceil
from bokeh.plotting import figure, show, output_file
from bokeh.models import Range1d

def nxt_power_2(x):
    # Returns the nearest power of 2 larger than given number
    return 2**ceil(log2(x))

def zero_padding(array):
    # Adds a series of 0s to the end of signal
    nextpwr = nxt_power_2(len(array))
    length_of_array = len(array)
    if nextpwr != length_of_array:
        for j in range(nextpwr-length_of_array):
            array.append(0)

def fft(x):
    # Calculates and returns the Discrete Fourier Transform using Cooley-
Tukey's algorithm

    length = len(x)
    if length <= 1:
        return x
```

```
    even_terms = fft(x[0::2])
    odd_terms = fft(x[1::2])
    fourier = [exp(-2j * pi * p / length) * odd_terms[p] for p in
range(length // 2)]
    return [even_terms[p] + fourier[p] for p in range(length // 2)] + \
           [even_terms[p] - fourier[p] for p in range(length // 2)]


input_data_path = "Data/Vibration Data.xlsx"
output_data_path = "Data/Fourier transformed Vibration Data.xlsx"
vibration_data = pd.read_excel(input_data_path)

# Separating the values of X and Y axis data
vibraX = pd.DataFrame(vibration_data, columns=['VibraX'])
vibraY = pd.DataFrame(vibration_data, columns=['VibraY'])
vibraY = vibraY.values.tolist()
vibraX = vibraX.values.tolist()

length_fixed = 300    # To limit the number or samples

# To calculate the sum of the data and then average
sum_x, sum_y = 0, 0
for i in range(length_fixed):
    sum_y += vibraY[i][0]
    sum_x += vibraX[i][0]

mean_x = sum_x / length_fixed
mean_y = sum_y / length_fixed


# Mean is subtracted from the data to remove the DC offset (Peak
appearing at 0Hz, though there is no DC component)
x_list, y_list = [], []
for i in range(length_fixed):
    x_list.append(vibraX[i][0] - mean_x)
    y_list.append(vibraY[i][0] - mean_y)

# Zeroes are added to the end of the signal
zero_padding(x_list)
zero_padding(y_list)

Fs = 1                          # Sampling Frequency of the signal
n = len(x_list)                 # Number of samples
print(n)
k = [i for i in range(n)]       # List from 0 to n [0, 1, 2, .... 4093,
4094, 4095]
T = n/Fs                        # Total time = No of sample/Sample frequency
frq = [x / T for x in k]        # Frequency range up to Fs/2 frq =
frq[:len(frq)//2]       # Only first half is taken

# FFT is applied on the X-Axis data, and then normalised
fourier_x_list = fft(x_list)
abs_fourier_x = [abs(x)/len(fourier_x_list) for x in fourier_x_list]
```

```
# FFT is applied on the Y-Axis data, and then normalised
fourier_y_list = fft(y_list)
abs_fourier_y = [abs(y)/len(fourier_y_list) for y in fourier_y_list]

# Only the left is taken as the output of FT will be symmetrical
final_fourier_x = abs_fourier_x[:len(abs_fourier_x)//2]
final_fourier_y = abs_fourier_y[:len(abs_fourier_y)//2]

# FT data is converted to data frame and then exported to a excelsheet
outX = pd.DataFrame({"Fourier_X": final_fourier_x})
outY = pd.DataFrame({"Fourier_Y": final_fourier_y})

writer = pd.ExcelWriter(output_data_path, engine='xlsxwriter')

outX.to_excel(writer, sheet_name="sheet1")
outY.to_excel(writer, startcol=2, index=False, sheet_name='sheet1')
writer.save()

output_file("Graph/FFT_own_x.html")
plot = figure(title="Vibration X fft - {} samples".format(length_fixed),
              x_axis_label='Frequency (Hz)',
              y_axis_label='Amplitude (g)',
              y_range=Range1d(-0.005, 0.1),
              plot_width=1500,
              plot_height=700)

plot.line(frq, final_fourier_x)
show(plot)

output_file("Graph/FFT_own_y.html")
plot = figure(title="Vibration Y fft - {} samples".format(length_fixed),
              x_axis_label='Frequency (Hz)',
              y_axis_label='Amplitude (g)',
              y_range=Range1d(-0.005, 0.1),
              plot_width=1500,
              plot_height=700)

plot.line(frq, final_fourier_y)
show(plot)
```

EXCEL INPUT:

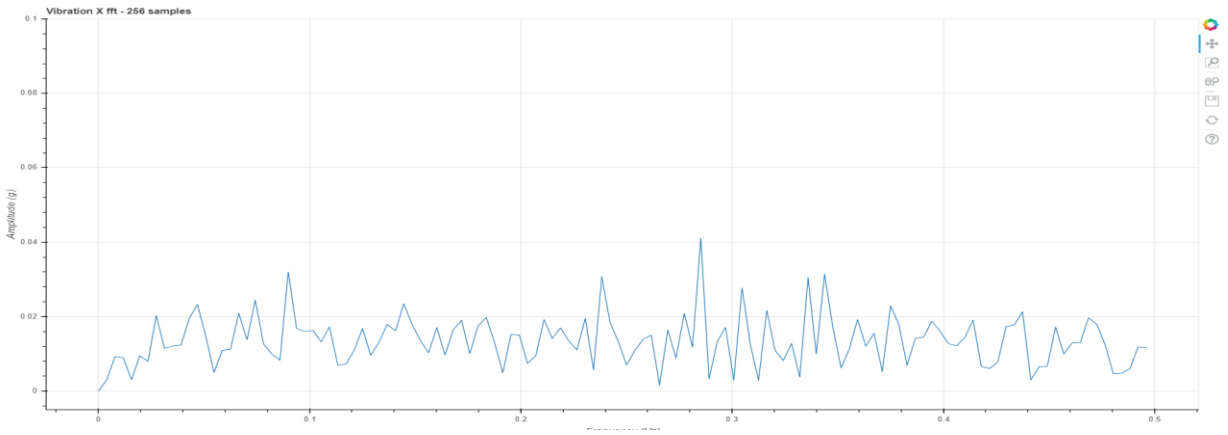
	A	B	C	D
1	Time	VibraX	VibraY	
2	26-03-2019 16:04	0.846055212	0.568021352	
3	26-03-2019 16:04	0.177611868	0.312074024	
4	26-03-2019 16:04	0.195563038	0.592176379	
5	26-03-2019 16:04	0.889125736	0.313843271	
6	26-03-2019 16:04	0.354602211	0.722310979	
7	26-03-2019 16:04	0.280934206	0.641682625	
8	26-03-2019 16:04	0.863441471	0.349909965	
9	26-03-2019 16:04	0.308955587	0.751970155	
10	26-03-2019 16:04	0.260010334	0.594757028	
11	26-03-2019 16:04	0.200056724	0.58940611	
12	26-03-2019 16:04	0.5552143	0.394447318	
13	26-03-2019 16:04	0.275721015	0.325203866	
14	26-03-2019 16:04	0.624254372	0.511772784	
15	26-03-2019 16:04	0.610752338	0.647962851	
16	26-03-2019 16:04	0.197415321	0.457503275	
17	26-03-2019 16:04	0.178838869	0.777335215	
18	26-03-2019 16:04	0.691179633	0.458369767	
19	26-03-2019 16:04	0.267590924	0.731742307	
20	26-03-2019 16:04	0.699294589	0.65264327	

EXCEL OUTPUT:

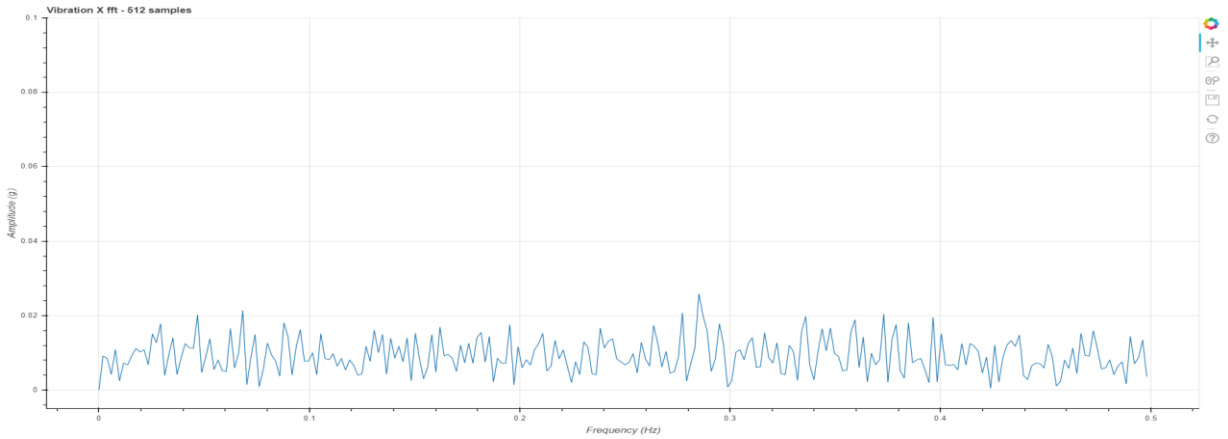
	A	B	C	D
1		Fourier_X	Fourier_Y	
2	0	0.30432	0.350131	
3	1	0.150186	0.169237	
4	2	0.048679	0.067729	
5	3	0.025796	0.039277	
6	4	0.037387	0.053251	
7	5	0.007809	0.010154	
8	6	0.024185	0.033568	
9	7	0.016847	0.003871	
10	8	0.006701	0.011217	
11	9	0.017807	0.012304	
12	10	0.005259	0.012443	
13	11	0.00937	0.014849	
14	12	0.012695	0.010104	
15	13	0.001761	0.001545	
16	14	0.011412	0.00765	
17	15	0.007598	0.013864	
18	16	0.007825	0.010591	
19	17	0.009458	0.019068	
20	18	0.000779	0.012303	
21	19	0.00989	0.015485	
22	20	0.004868	0.004029	

OUPUTS FOR VARIED NUMBER OF SAMPLES:

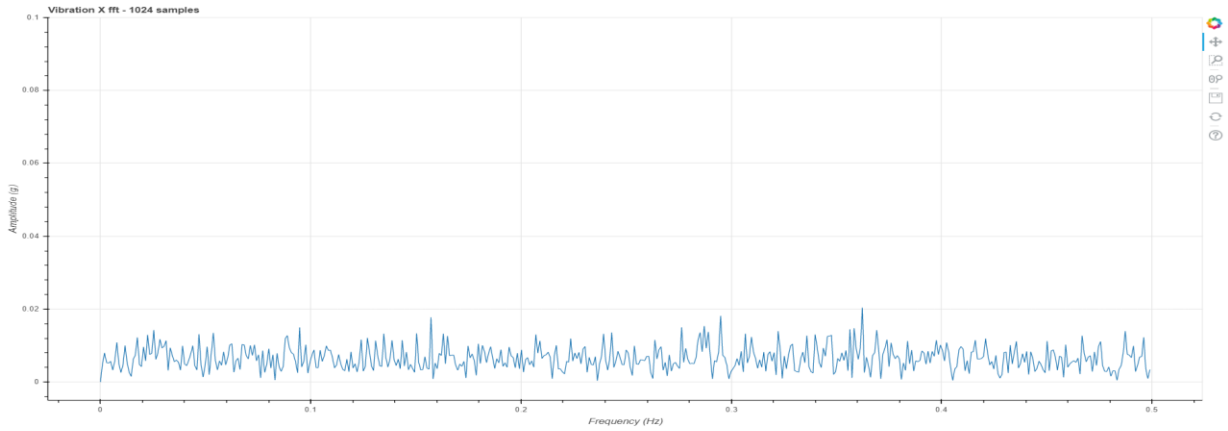
For 256 samples



For 512 samples:



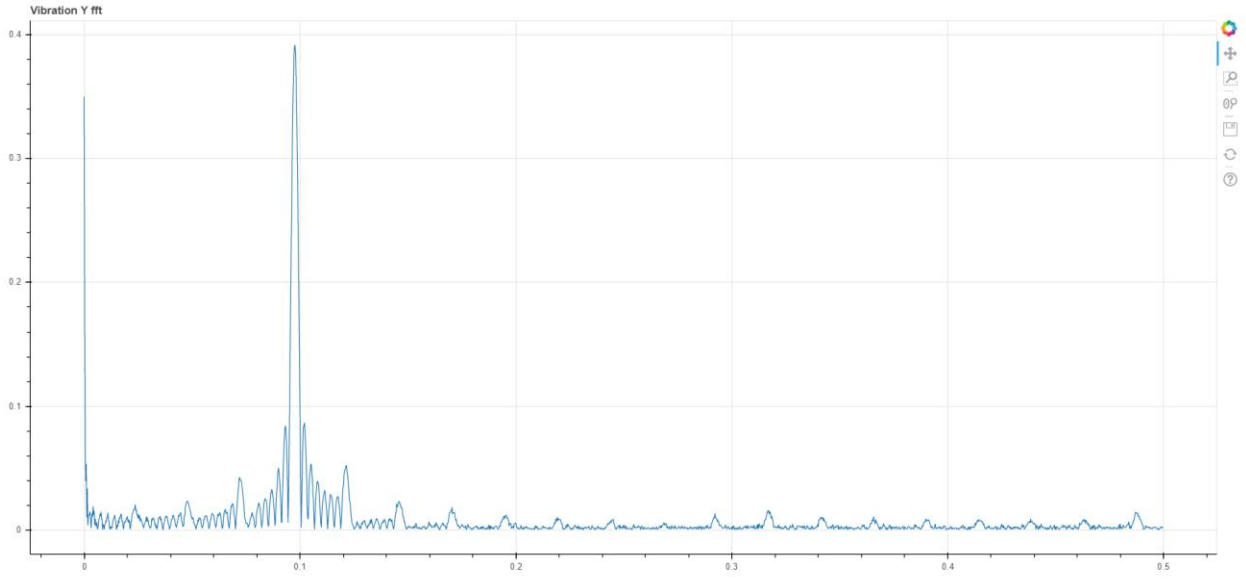
For 1024 samples:



From the above different graphical outputs, we can see that the resolution increases with the number of samples taken, so it is advisable to take more samples as possible.

REASONS FOR NO PEAK:

We cannot find any peak in the graph because the sample frequency is just $1Hz$. Which means we cannot detect any signal with frequency greater than $0.5Hz$. So, we must increase the sampling frequency to detect higher frequency signals. To check if we can detect signals smaller than $0.5Hz$, a sinusoidal signal of $0.1Hz$ is added manually, and the output is as following:



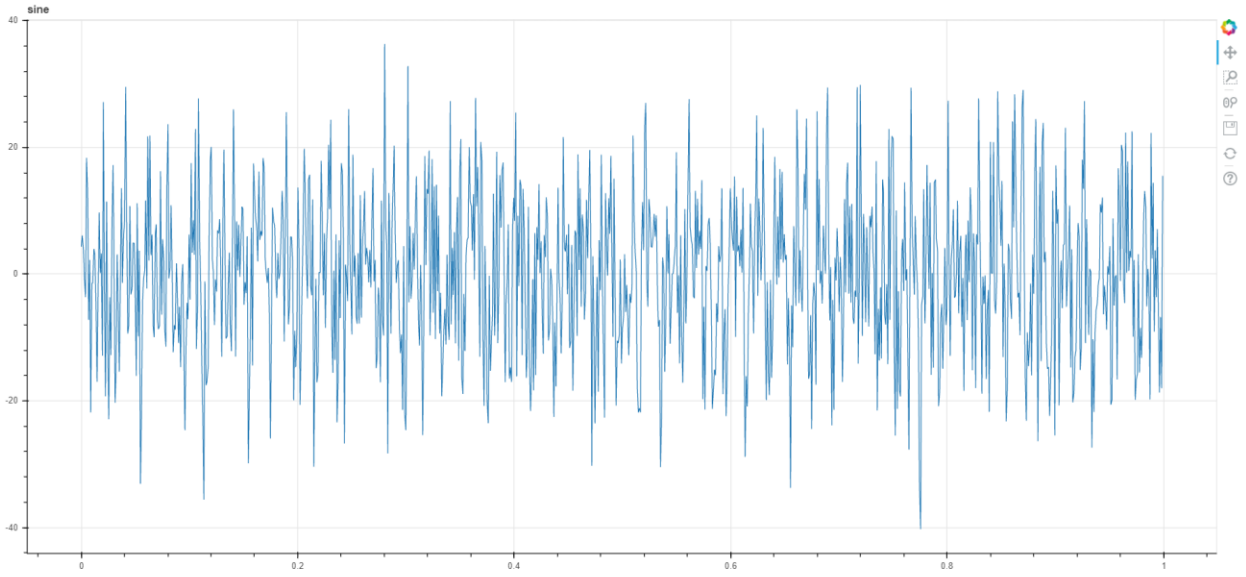
From this we can see that there is a peak at $0.1Hz$, which is as expected.

ANOTHER EXAMPLE:

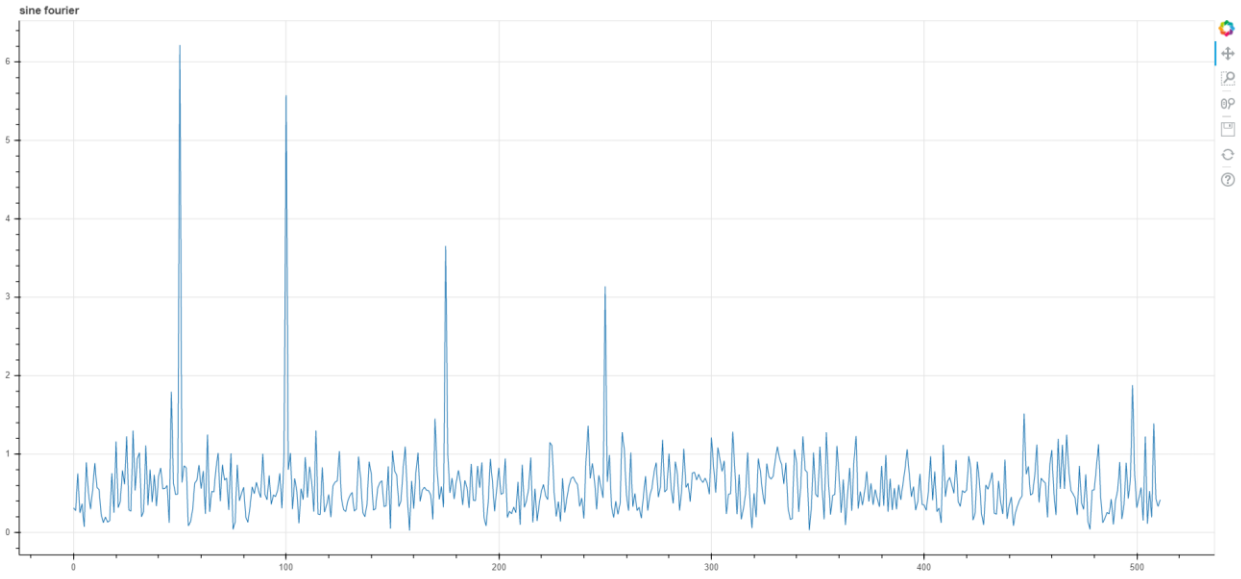
In this, four different sinusoidal signals of different frequencies and amplitude are added together with a noise.

$$y = 6\sin(2\pi 50t) + 5\sin(2\pi 100t) + 3\sin(2\pi 250t) + 4\sin(2\pi 175t) + gaussian\ noise$$

The sine wave after adding together looks like following:



In this we cannot see any sine wave and looks completely like noise. But even in this, our FFT algorithm can detect the individual signal frequency peak and their amplitude.



We can see the peaks at $50Hz, 100Hz, 175Hz\ and\ 250Hz$ and even their corresponding amplitudes.