

# VIBRATION ANALYSIS USING FFT

## OVERVIEW:

Any waveform is a sum of series of simple sinusoids of different frequency, amplitude or phase. A Fourier series is a series of sine waves; and we use Fourier analysis to deconstruct a signal into its individual sinusoidal 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, when sampling 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 ( $F_s$ ) is lesser than message frequency ( $F_m$ ) the sampled values will contain less data, and reconstruction of the original signal is impossible.

So, to sample a signal of frequency  $F_m = 50\text{Hz}$  we must sample the signal at  $F_s \geq 2 \times F_m = 100\text{Hz}$

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

Reference: [https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon\\_sampling\\_theorem](https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem)

## 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" }` in the command prompt.

- Pandas for data manipulation
  - Importing and exporting excel sheets
  - Converting them to python list
- Cmath module for complex number analysis:
  - Importing constants like  $e$ ,  $i$ ,  $\pi$
- 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 - Modified.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'])

# Converting dataframes in to a python list
vibraY = vibraY.values.tolist()
vibraX = vibraX.values.tolist()
```

## 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  $\lceil \log_2(x) \rceil$ . The log function returns the nearest power of 2 and the ceil function  $\lceil x \rceil$  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):
    # 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 such that signal length becomes a power of 2
    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)
```

The function is applied on the x\_list and y\_list

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

### LIMITING THE 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 specific data size 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.

```
# To limit the number or samples
length_fixed = 1024
```

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 use 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 Programming languages: [Click Here](#))

```
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)]
    # Returns the Fourier transformed data as the output
```

This returns unnormalized values of the signal.

## NORMALISING FFT AND POWER:

The unnormalized data 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.

We can also use the power of the FFT frequencies to analyze as the power series has the peaks distinguished easily and the lower amplitude frequencies are also suppressed which are not of much use.

The power is taken by squaring the amplitude of the individual data on the FFT series.

```
# 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]
power_fourier_x = [(abs(x) / len(fourier_x_list))**2 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]
power_fourier_y = [(abs(y) / len(fourier_y_list))**2 for y in fourier_y_list]
```

### SIGNAL PARAMETERS:

For the given data, the sampling frequency  $F_s = 1\text{Hz}$  which makes it impossible to detect the frequencies higher than  $0.5\text{Hz}$ . 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  $F_s$ , but we can exclude the second half since the sampling theorem allows till  $\frac{F_s}{2}$ .

```
Fs = 1          # Sampling Frequency of the signal
n = len(x_list) # Number of samples
k = [i for i in range(n)] # List of values 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 (from 0 Hz to 0.5 Hz)
frq = frq[:len(frq) // 2] # Only first half is taken, because FFT output will be symmetrical
```

### 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  $50\text{Hz}$ , each peak less than  $25\text{Hz}$  will repeat again after  $25\text{Hz}$ . If there is a peak at  $10\text{Hz}$ , then the same peak will arise at  $40\text{Hz}$  ( $50-10$ ).

Thus, we must exclude the later half of the signal before exporting and plotting.

```
# 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]

final_fourier_pwr_x = power_fourier_x[:len(power_fourier_x) // 2]
final_fourier_pwr_y = power_fourier_y[:len(power_fourier_y) // 2]
```

## PEAK DETECTION:

Instead of visualising the graph, we can use a peak detection algorithm to find the position of peaks in the given data after performing the Fourier transform. A point is considered as a peak when the points before and after are lesser than it.

```
def peak_pos(y_axis, x_axis):
    # Identifies the peaks from the data and returns the position of the peak(Freq) and also the Amplitude
    # of the peak
    peaks = []
    length = len(y_axis)
    mean = sum(y_axis)/length
    # stdev = sqrt(sum(pow(x-mean, 2)/length for x in y_axis))
    for j in range(1, len(y_axis)-1):
        if y_axis[j] > mean and y_axis[j-1] < y_axis[j] > y_axis[j+1]:
            x_value = round(x_axis[j], 2)
            y_value = round(y_axis[j], 2)
            if y_value != 0:
                peaks.append((x_value, y_value))
    return peaks
    # Returns a list of tuples containing an ordered pair of amplitude and frequency

print("Peaks in Y axis \n (Amp, Frq)\n", peak_pos(final_fourier_pwr_y, frq))
print("Peaks in X axis \n (Amp, Frq)\n", peak_pos(final_fourier_pwr_x, frq))
```

## 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 xlswriter we can export the data into excel.

```
# 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})
outPowerX = pd.DataFrame({"FourierPower_X": final_fourier_pwr_x})
outPowerY = pd.DataFrame({"FourierPower_Y": final_fourier_pwr_y})

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

outX.to_excel(writer, sheet_name="sheet1")
outY.to_excel(writer, startcol=2, index=False, sheet_name='sheet1')
outPowerX.to_excel(writer, startcol=3, index=False, sheet_name='sheet1')
outPowerY.to_excel(writer, startcol=4, 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_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, 1),
              plot_width=1500,
              plot_height=700)

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

output_file("Graph/FFT_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, 1),
              plot_width=1500,
              plot_height=700)

plot.line(freq, final_fourier_y)
show(plot)

output_file("Graph/Power_x.html")
plot = figure(title="Vibration X fft Power - {} samples".format(length_fixed),
              x_axis_label='Frequency (Hz)',
              y_axis_label='Amplitude (g)',
              y_range=Range1d(-0.005, 1),
              plot_width=1500,
              plot_height=700)

plot.line(freq, final_fourier_pwr_x)
show(plot)

output_file("Graph/Power_y.html")
plot = figure(title="Vibration Y fft Power - {} samples".format(length_fixed),
              x_axis_label='Frequency (Hz)',
              y_axis_label='Amplitude (g)',
              y_range=Range1d(-0.005, 1),
              plot_width=1500,
              plot_height=700)

plot.line(freq, final_fourier_pwr_y)
show(plot)
```

## WHOLE CODE:

```
import pandas as pd
from cmath import pi, exp
from math import log2, ceil
# from math import sqrt
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 such that signal length becomes a power of 2
    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)]
    # Returns the Fourier transformed data as the output

def peak_pos(y_axis, x_axis):
    # Identifies the peaks from the data and returns the position of the peak(Freq) and also the Amplitude
    of the peak
    peaks = []
    length = len(y_axis)
    mean = sum(y_axis)/length
    # stdev = sqrt(sum(pow(x-mean, 2)/length for x in y_axis))
    for j in range(1, len(y_axis)-1):
        if y_axis[j] > mean and y_axis[j-1] < y_axis[j] > y_axis[j+1]:
            x_value = round(x_axis[j], 2)
            y_value = round(y_axis[j], 2)
```



```

        if y_value != 0:
            peaks.append((x_value, y_value))
    return peaks
    # Returns a list of tuples containing an ordered pair of amplitude and frequency

input_data_path = "Data/Vibration Data - Modified.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'])

# Converting dataframes in to a python list
vibraY = vibraY.values.tolist()
vibraX = vibraX.values.tolist()

# To limit the number of samples
length_fixed = 1024

# 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
k = [i for i in range(n)] # List of values 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 (from 0 Hz to 0.5 Hz)
frq = frq[:len(frq) // 2] # Only first half is taken, because FFT output will be symmetrical

```

```

# 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]
power_fourier_x = [(abs(x) / len(fourier_x_list))**2 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]
power_fourier_y = [(abs(y) / len(fourier_y_list))**2 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]

final_fourier_pwr_x = power_fourier_x[:len(power_fourier_x) // 2]
final_fourier_pwr_y = power_fourier_y[:len(power_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})
outPowerX = pd.DataFrame({"FourierPower_X": final_fourier_pwr_x})
outPowerY = pd.DataFrame({"FourierPower_Y": final_fourier_pwr_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')
outPowerX.to_excel(writer, startcol=3, index=False, sheet_name='sheet1')
outPowerY.to_excel(writer, startcol=4, index=False, sheet_name='sheet1')

writer.save()

output_file("Graph/FFT_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, 1),
              plot_width=1500,
              plot_height=700)

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

```

```

output_file("Graph/FFT_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, 1),
              plot_width=1500,
              plot_height=700)

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

output_file("Graph/Power_x.html")
plot = figure(title="Vibration X fft Power - {} samples".format(length_fixed),
              x_axis_label='Frequency (Hz)',
              y_axis_label='Amplitude (g)',
              y_range=Range1d(-0.005, 1),
              plot_width=1500,
              plot_height=700)

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

output_file("Graph/Power_y.html")
plot = figure(title="Vibration Y fft Power - {} samples".format(length_fixed),
              x_axis_label='Frequency (Hz)',
              y_axis_label='Amplitude (g)',
              y_range=Range1d(-0.005, 1),
              plot_width=1500,
              plot_height=700)

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

print("Peaks in Y axis \n (Amp, Frq)\n", peak_pos(final_fourier_pwr_y, frq))
print("Peaks in X axis \n (Amp, Frq)\n", peak_pos(final_fourier_pwr_x, frq))

```

# EXCEL INPUT:

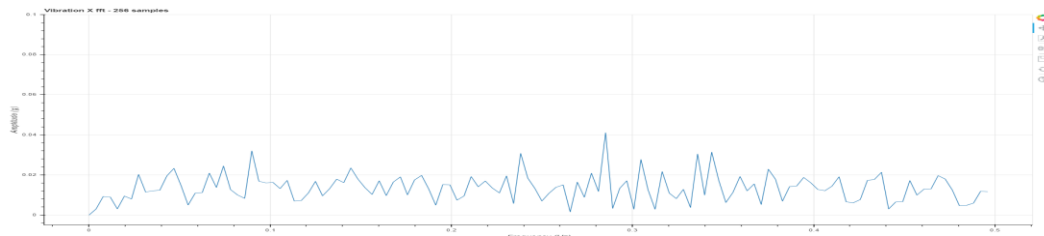
	A	B	C	D
1	Time	VibraX	VibraY	
2	26-03-2019 16:04	0.846055212	0.568021352	
3	26-03-2019 16:04	0.177611868	0.312074024	
4	26-03-2019 16:04	0.195563038	0.592176379	
5	26-03-2019 16:04	0.889125736	0.313843271	
6	26-03-2019 16:04	0.354602211	0.722310979	
7	26-03-2019 16:04	0.280934206	0.641682625	
8	26-03-2019 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:

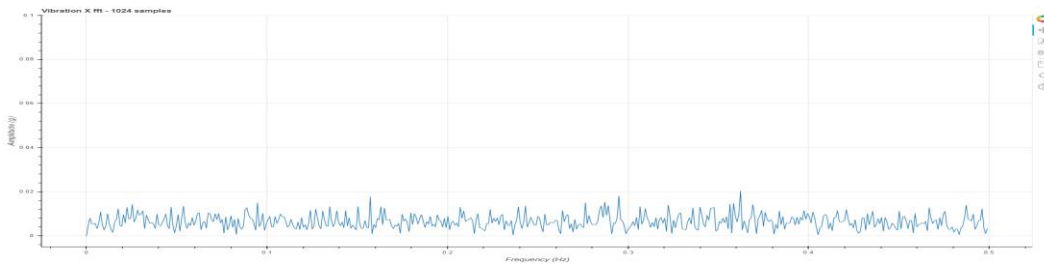
	A	B	C	D	E
1		Fourier_X	Fourier_Y	FourierPower_X	FourierPower_Y
2	0	1.661E-16	4.16334E-17	2.75891E-32	1.73334E-33
3	1	0.004920069	0.053808125	2.42071E-05	0.002895314
4	2	0.007988118	0.045862398	6.381E-05	0.00210336
5	3	0.005344511	0.005365434	2.85638E-05	2.87879E-05
6	4	0.005222866	0.038690781	2.72783E-05	0.001496977
7	5	0.005681915	0.029978959	3.22842E-05	0.000898738
8	6	0.003288056	0.005451733	1.08113E-05	2.97214E-05
9	7	0.005765194	0.023648815	3.32375E-05	0.000559266
10	8	0.010877271	0.047066055	0.000118315	0.002215214
11	9	0.005060442	0.014044295	2.56081E-05	0.000197242
12	10	0.002598881	0.021710588	6.75418E-06	0.00047135
13	11	0.004681149	0.039009073	2.19132E-05	0.001521708
14	12	0.009997637	0.019327786	9.99527E-05	0.000373563
15	13	0.005379568	0.0186011	2.89397E-05	0.000346001
16	14	0.002675443	0.030783137	7.15799E-06	0.000947602
17	15	0.00153809	0.024688337	2.36572E-06	0.000609514
18	16	0.006265127	0.013314058	3.92518E-05	0.000177264
19	17	0.007395081	0.045451134	5.46872E-05	0.002065806
20	18	0.012247254	0.029455862	0.000149995	0.000867648

## OUTPUTS FOR VARIED NUMBER OF SAMPLES:

For 256 samples



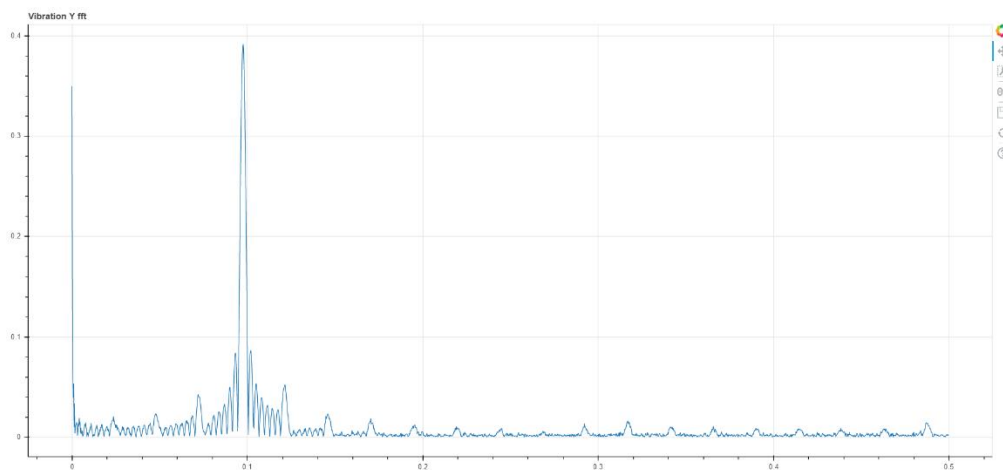
For 1024 samples:



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:

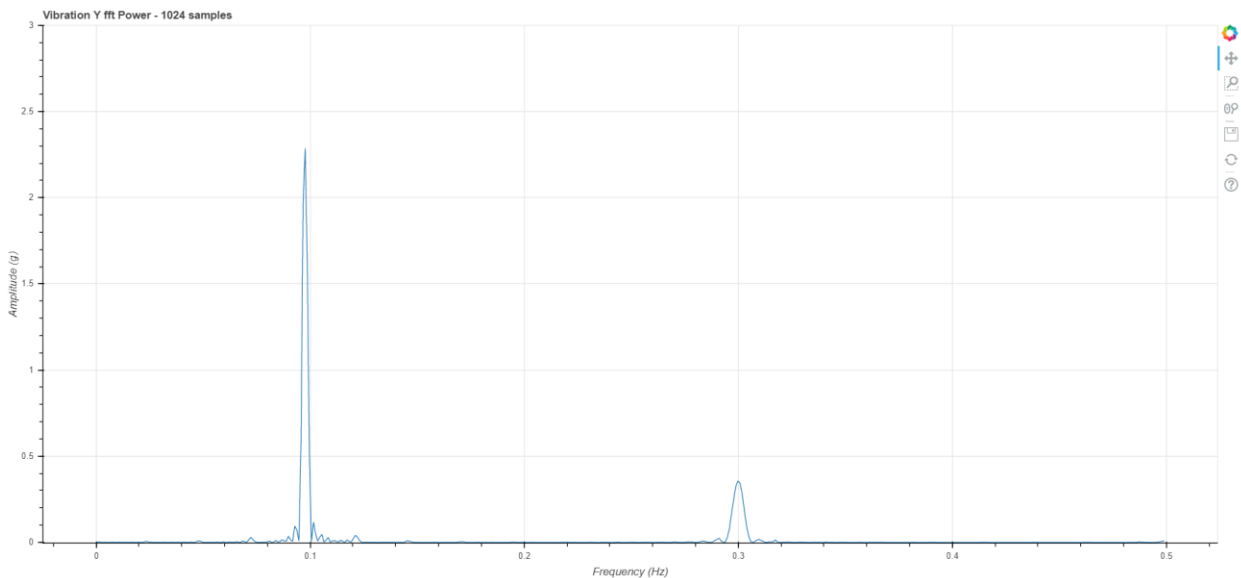
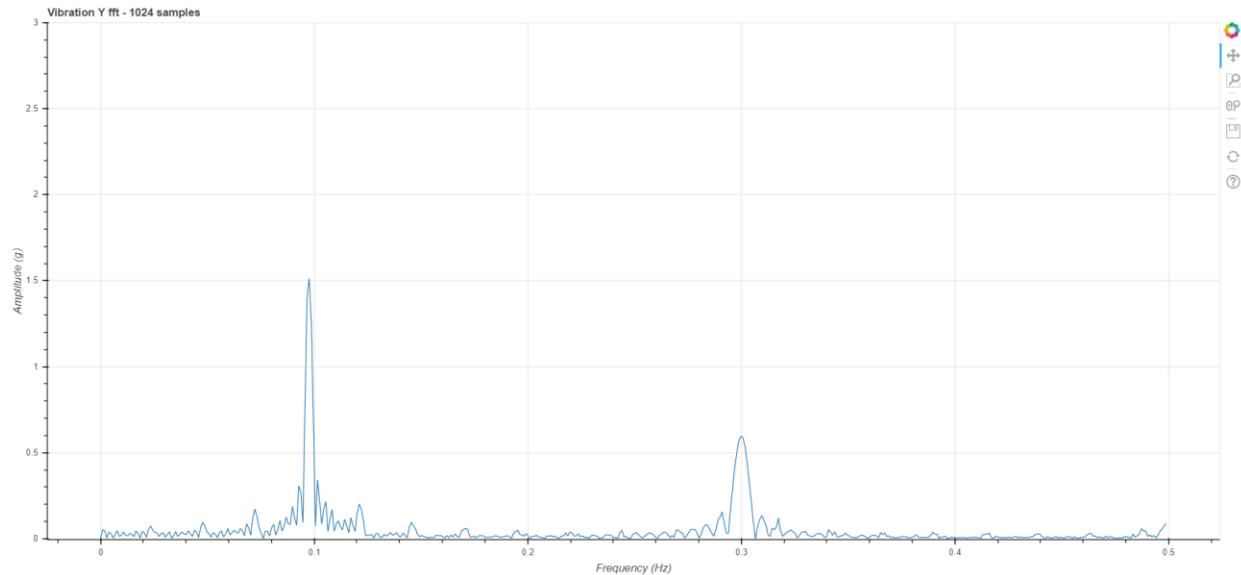
We cannot find any peak in the graph because the sample frequency is just  $1\text{Hz}$ . Which means we cannot detect any signal with frequency greater than  $0.5\text{Hz}$ . So, we must increase the sampling frequency to detect higher frequency signals. To check if we can detect signals smaller than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



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

## POWER:

The difference between the normalized FFT and the power is seen in this graph. The power reduces the effect of the lower amplitude frequencies on the graph. The larger amplitudes get larger and the lower amplitudes ones get even lower.

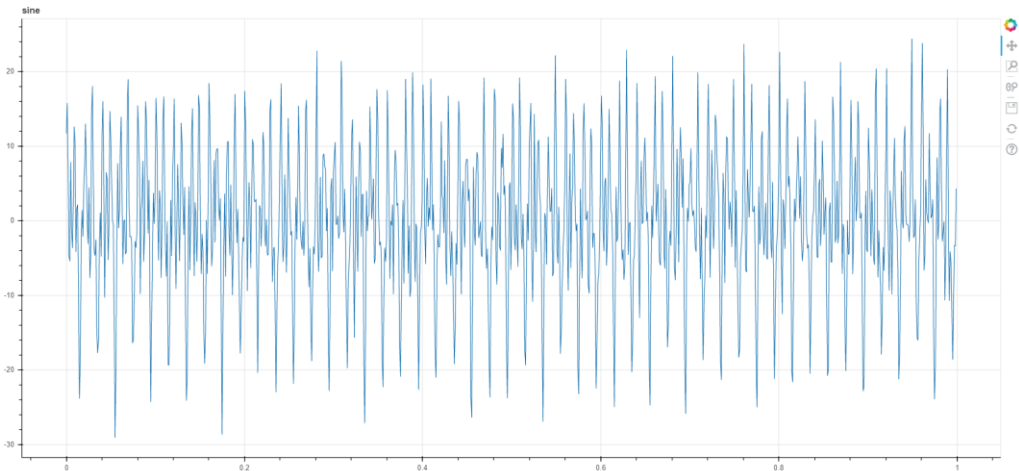


## ANOTHER EXAMPLE:

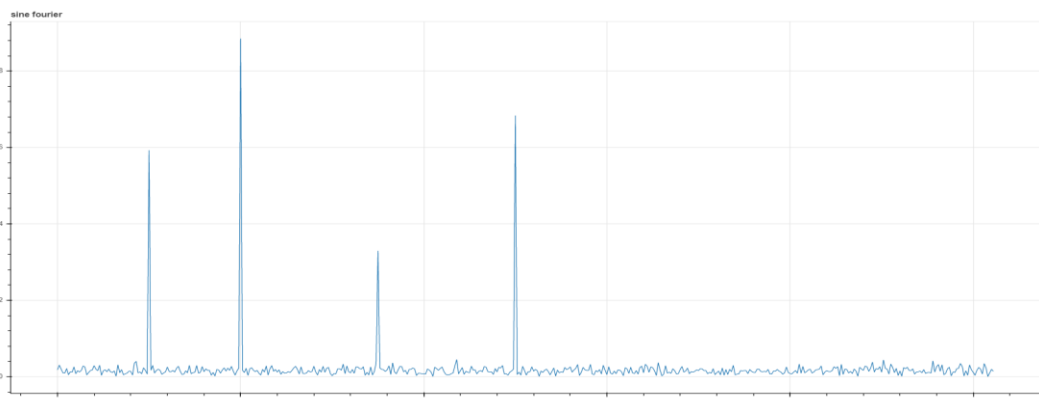
In this, four different sinusoidal signals of different frequencies and amplitude are added together with a noise.

$$y = 6 \sin(2\pi 50t) + 9 \sin(2\pi 100t) + 7 \sin(2\pi 250t) + 3 \sin(2\pi 175t) + \text{gaussian noise}$$

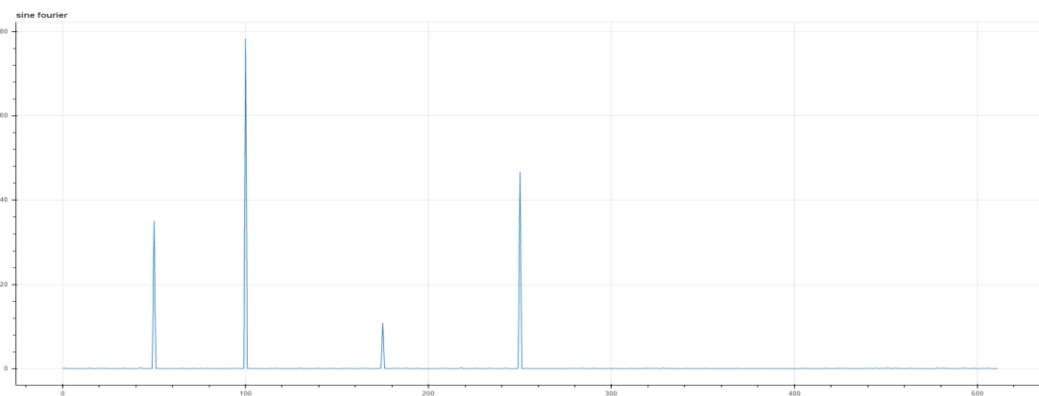
The sine wave after adding together looks like following:



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. But there are so much of noise at the lower amplitude level. In order to remove/suppress that, we can use the power to analyze it.



One thing we can notice is that while the peaks are clearer, it also increases the difference between them. In the first graph, four peaks are somewhat closer in amplitude but in the power, the 3<sup>rd</sup> peak is small.