Signal Processing in Practice
Assignment 17
Power Systems
Report

Saptarshi Mandal
SR. No.: 22925
MTech(Signal Processing)
Electrical Engineering

8 April 2024

Question 1

A benchmark system, IEEE 33bus - 12.66KV, is used to generate the fault data for detection and classification. Persistent Faults or uncleared fault is used as test signals for protective relaying to verify the operation of the IED. Files are present in Signalfiles.Persistent Fault

Part (a)

Select one file from High Impedance and Low Impedance folder each, and note down the signal count, sampling frequency, samples per cycle, number of windows and number of outputs per signal.

The Figure 1 shows the current and voltage waveforms for a particular high impedance.

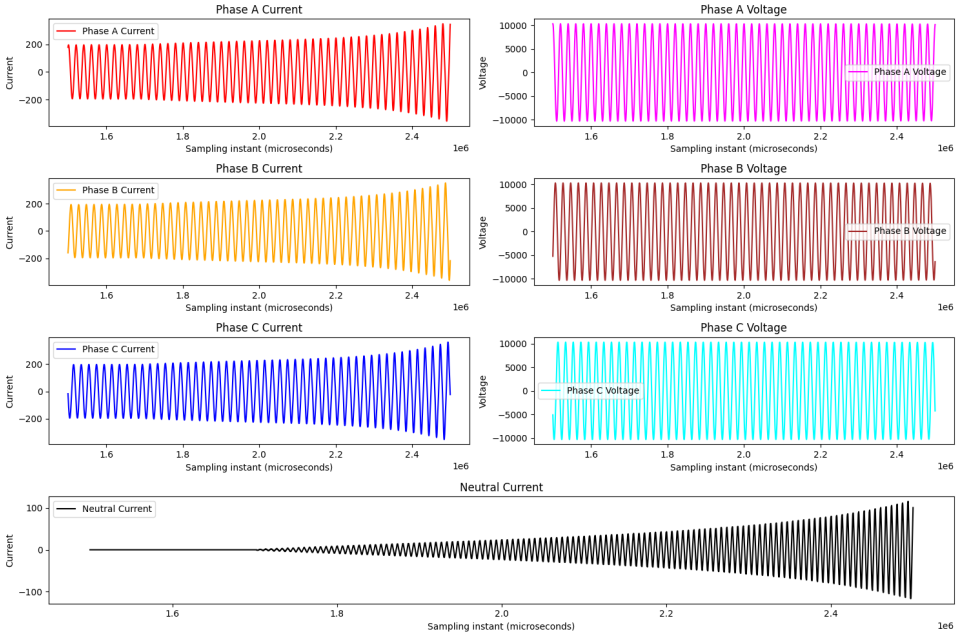

Figure 1: Plot of current and voltage waveform for High Impedance

```
 Analysis of the current and voltage signals for High
Impedance:

    Signal Count:  7
    Sampling Frequency:  4000 Hz
    Samples per cycle:  80
    Number of windows:  3921
    Number of outputs per signal:  3921
```

The Figure 2 shows the current and voltage waveforms for a particular low impedance.

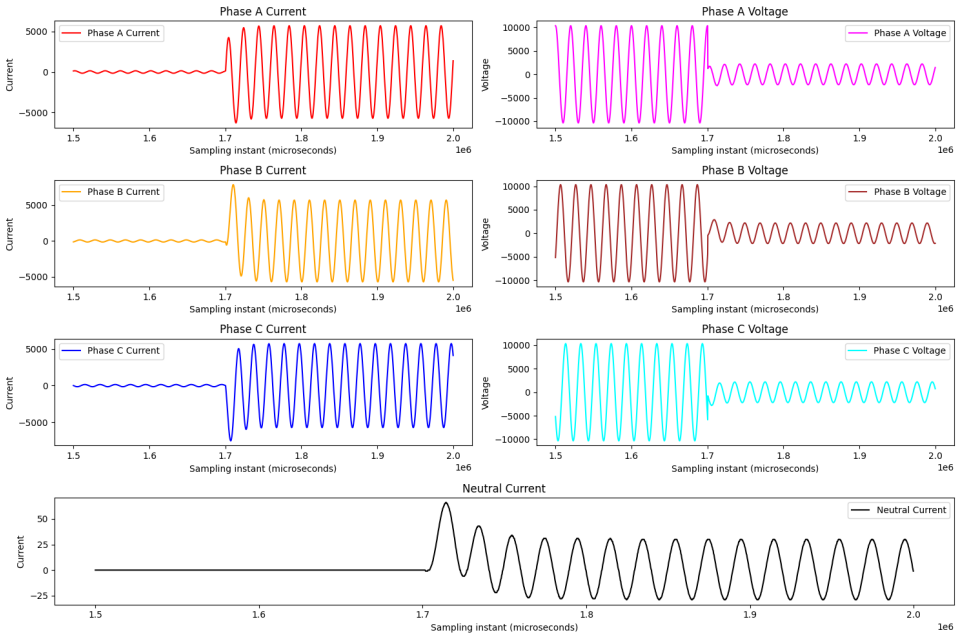

Figure 2: Plot of current and voltage waveform for Low Impedance

```
 Analysis of the current and voltage signals for Low
Impedance:
    Signal Count:  7
    Sampling Frequency:  4000 Hz
    Samples per cycle:  80
    Number of windows:  1921
    Number of outputs per signal:  1921
```

The code for generating the plots and signal analysis is given below. `plot_currents_voltages(df)` plots the currents and voltages from a given DataFrame `df`. The function extracts relevant data such as sampling frequency, samples per cycle, number of signals, number of windows, and number of outputs per signal, and then plots the currents and voltages for each phase.

```
def plot_currents_voltages(df):

    # Extract data
    # Sampling frequency in Hz
    sampling_frequency = 1 / (df.iloc[1, 1] - df.iloc[0, 1]) * 1e6
```

```
    # Samples per cycle
    samples_per_cycle = int(sampling_frequency / 50)
    # Number of signals (currents and voltages excluding first two columns)
    signal_count = df.shape[1] - 2
    window_size = samples_per_cycle
    # Number of windows
    number_of_windows = df.shape[0] - window_size + 1
    number_of_outputs_per_signal = number_of_windows

    # Plotting
    plt.figure(figsize=(15, 10))

    # Plot Line A
    plt.subplot(3, 2, 1)
    plt.plot(df.iloc[:, 1], df.iloc[:, 2], label='Phase A Current', color='Red')
    plt.xlabel('Sampling instant (microseconds)')
    plt.ylabel('Current')
    plt.title('Phase A Current')
    plt.legend()

    # Plot Line B
    plt.subplot(3, 2, 2)
    plt.plot(df.iloc[:, 1], df.iloc[:, 5], label='Phase A Voltage', color='magenta')
    plt.xlabel('Sampling instant (microseconds)')
    plt.ylabel('Voltage')
    plt.title('Phase A Voltage')
    plt.legend()

    # Plot Line C
    plt.subplot(3, 2, 3)
    plt.plot(df.iloc[:, 1], df.iloc[:, 3], label='Phase B Current', color='orange')
    plt.xlabel('Sampling instant (microseconds)')
    plt.ylabel('Current')
    plt.title('Phase B Current')
    plt.legend()

    plt.subplot(3, 2, 4)
    plt.plot(df.iloc[:, 1], df.iloc[:, 6], label='Phase B Voltage', color='brown')
    plt.xlabel('Sampling instant (microseconds)')
    plt.ylabel('Voltage')
    plt.title('Phase B Voltage')
    plt.legend()

    # Plot Line C
    plt.subplot(3, 2, 5)
    plt.plot(df.iloc[:, 1], df.iloc[:, 4], label='Phase C Current', color='blue')
    plt.xlabel('Sampling instant (microseconds)')
    plt.ylabel('Current')
    plt.title('Phase C Current')
    plt.legend()
```

```
    plt.subplot(3, 2, 6)
    plt.plot(df.iloc[:, 1], df.iloc[:, 7], label='Phase C Voltage',
     color='cyan')
    plt.xlabel('Sampling instant (microseconds)')
    plt.ylabel('Voltage')
    plt.title('Phase C Voltage')
    plt.legend()

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

Part (b)

Use an N-point DFT to find the amplitude of the fundamental and dc component; group and plot all voltages and currents with appropriate axis details. Provide amplitude and waveform combined plots for one phase current and respective phase voltage, and comment on the traced envelope.

Figure 3 gives the waveform of the fundamental current and voltage of phase A for high impedance.

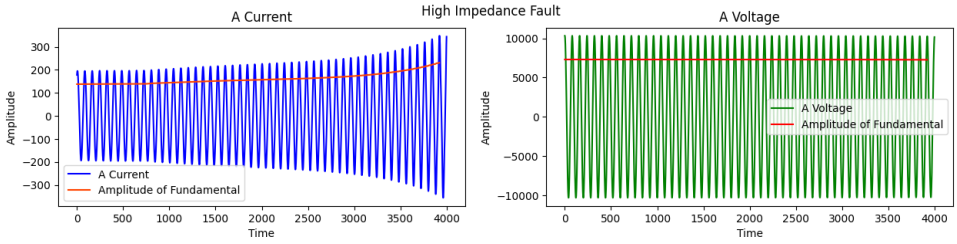

Figure 3: Fundamental current and voltage waveform for High Impedance

Figure 4 shows the waveform of the DC current and voltage for Phase A for high impedance.

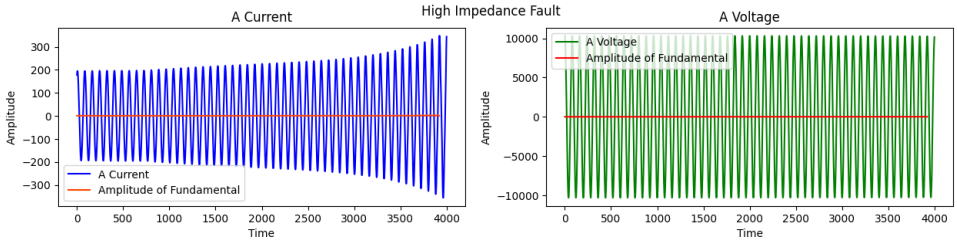

Figure 4: DC current and voltage waveform for High Impedance

Figure 5 gives the waveform of the fundamental current and voltage of phase A for low impedance.

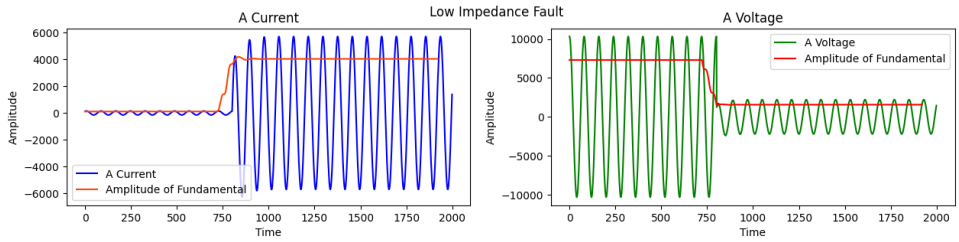

Figure 5: Fundamental current and voltage waveform for Low Impedance

Figure 6 shows the waveform of the DC current and voltage for Phase A for low impedance.

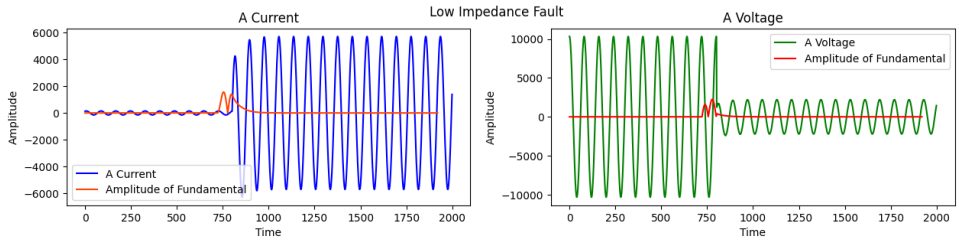

Figure 6: DC current and voltage waveform for Low Impedance

Observations The traced envelop in both the cases (High Impedance and Low Impedance) are seen to be accurate as it gives the RMS value of the fundamental component and the DC component in respective cases. The DC Value is almost zero for the pre-fault current and the post fault current, it's only non-zero at the time duration where the fault occurs.

The following Python functions `dft` and `compute_dft_for_all_signals` are used for computing the Discrete Fourier Transform (DFT) of signals.

```python
def dft(x, window_size, component = 1, all_components = False):

    start_time = time.time()

    N = len(x)
    num_windows = N - window_size + 1  # Calculate number of windows with a stride of 1

    if all_components:
        X = np.zeros((num_windows,window_size), dtype=np.complex128)
    else:
        X = np.zeros(num_windows, dtype=np.complex128)

    # Create matrices for cosine and sine terms
    n = np.arange(window_size)
    k = n.reshape((window_size, 1))
    Wc = np.cos(2 * np.pi * k * n / window_size)
    Ws = np.sin(2 * np.pi * k * n / window_size)

    for i in range(num_windows):

        window = x[i:i + window_size]  # Extract window with stride 1

        Xc = np.dot(Wc, window)
        Xs = np.dot(Ws, window)

        DFT = np.sqrt(2)/window_size * (Xc - 1j * Xs)

        if all_components == False:
            # Take only the required component
            X[i] = DFT[component]
        else:
            X[i] = DFT

    print(f'Running_time : {time.time() - start_time}s')

    return X
```

```python
def compute_dft_for_all_signals(df, window_size, component = 1):

    start_time = time.time()
    dft_results = {}
    original_signals = {}

    for col in df.columns[2:-1]:  # Exclude first two columns (serial number, sampling instant)
        signal = df[col]
        X = dft(signal, window_size, component)
        running_time = time.time() - start_time
        dft_results[col] = (X, running_time)
        original_signals[col] = signal.values  # Store original signal values

```

```
    print(f'Total Running_time : {time.time() - start_time}s')
    return dft_results, original_signals
```

Part (c)

Use a recursive DFT and compare with N-Point DFT, consider amplitude, phase and real and imaginary components along with FLOPS needed.

Figure 7 gives the waveform of the fundamental current and voltage of phase A for high impedance.

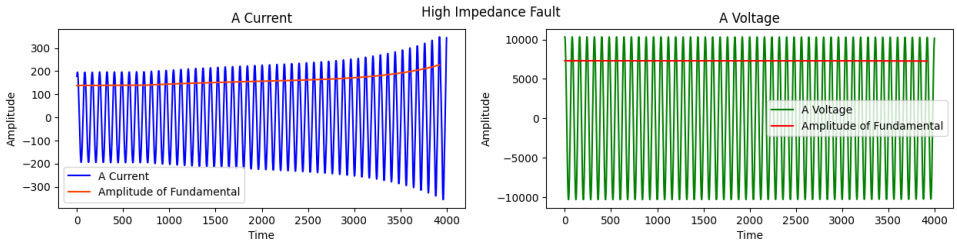

Figure 7: Fundamental current and voltage waveform for High Impedance (Recursive DFT)

Figure 8 shows the waveform of the DC current and voltage for Phase A for high impedance.

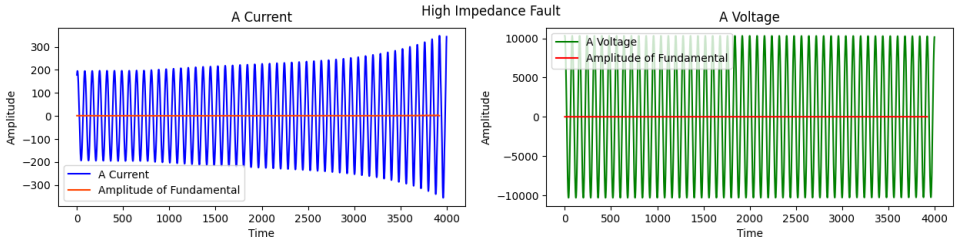

Figure 8: DC current and voltage waveform for High Impedance (Recursive DFT)

Figure 9 gives the waveform of the fundamental current and voltage of phase A for low impedance.

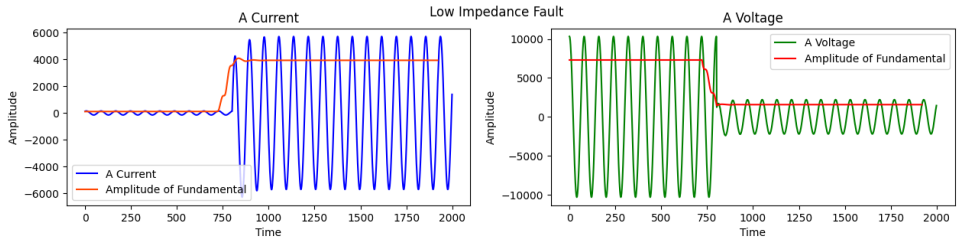

Figure 9: Fundamental current and voltage waveform for Low Impedance (Recursive DFT)

Figure 10 shows the waveform of the DC current and voltage for Phase A for low impedance.

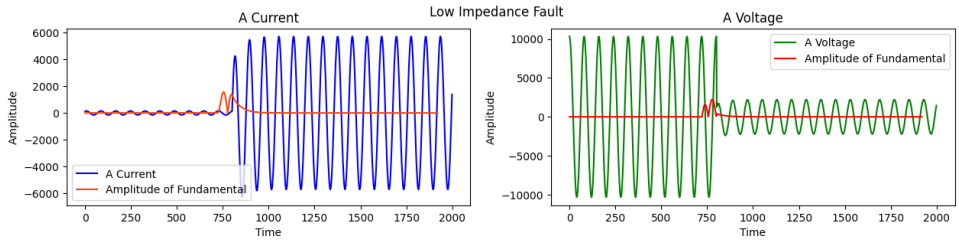

Figure 10: DC current and voltage waveform for Low Impedance (Recursive DFT)

The output of the recursive DFT, given in the above Figures is observed to be the same as the non-recursive ones.

The following Table. **??** is a comparison between the time taken (in msec) for DFT computation in The Basic Approach and Recursive Approach for High Impedance Fault.

High Impedance				
	Fundamental Component		DC component	
	Basic DFT	Recursive DFT	Basic DFT	Recursive DFT
Phase A current	613.6	0.988	382	1.09
Phase A voltage	613.08	0.592	355.57	0.97

The following Table **??** is a comparison between the time taken (in msec) for DFT computation in The Basic Approach and Recursive Approach for Low Impedance Fault.

Low Impedance				
	Fundamental Component		DC component	
	Basic DFT	Recursive DFT	Basic DFT	Recursive DFT
Phase A current	310.13	0.724	185.83	0.89
Phase A voltage	306.83	0.53	203.09	0.94

The code for generating the DFT in a recursive fashion is as follows:

```
def recursive_dft_update(X_prev, x_prev, x_new, k, N):

    # Compute cosine term
    cos_term = np.cos(2 * np.pi * k / N).item()

    # Compute sine term
    sine_term = np.sin(2 * np.pi * k / N).item()

    # Update real part of X_k1

    Xc_k1_real = X_prev.real * cos_term + X_prev.imag * sine_term +
     np.sqrt(2)/N * (x_new - x_prev) * cos_term

    Xs_k1_imag = X_prev.real * sine_term - X_prev.imag * cos_term +
     np.sqrt(2)/N * (x_new - x_prev) * sine_term

    # Compute updated DFT coefficient X_k1
    X_k1 = Xc_k1_real - 1j * Xs_k1_imag

    return X_k1

def recursive_dft(signal, window_size, component = 1):

    num_samples = signal.shape[0]
    n_windows = num_samples - window_size + 1

    X = np.zeros(n_windows, dtype=np.complex128)  # Initialize DFT
    coefficients

    # For all windows
    for i in range(n_windows):

        if i == 0:
            # First DFT
            X[i] = dft(signal[i:i + window_size], window_size,
    component).item()

        else:
            # Compute x_new for the current frequency bin k
            x_new = signal[i + window_size - 1]
            x_prev = signal[i - 1]
            X[i] = recursive_dft_update(X[i-1], x_prev, x_new, k=
    component, N=window_size)

    return X
```

```
def compute_recursive_dft_for_all_signals(df, window_size,
    component = 1):

    start_time = time.time()
    dft_results = {}
    original_signals = {}

    for col in df.columns[2:-1]:  # Exclude first two columns (
    serial number, sampling instant)
        signal = df[col]
```

```
        X = recursive_dft(signal, window_size, component)
        running_time = time.time() - start_time
        dft_results[col] = (X, running_time)
        original_signals[col] = signal.values  # Store original
    signal values

    print(f'Total Running_time : {time.time() - start_time}s')
    return dft_results, original_signals
```

Part (d)

Assume that the fault current threshold is twice that of the pre-fault current. When will the IED detect a fault? What is the delay from fault inception? Compare both files.

For high impedance:

```
Fault Inception for High Impedance:

    High Impedance Fault :  Delay from Fault Inception :
    No Fault Detected!!
```

In the implementation, we have assumed that the fault in detected when the current is twice the base current but in case of High Impedance Fault, the fault current never reaches that value. Hence, the IED doesn't detect a fault.

The following is the code that generates the above output:

```
fault_detection_time_ied = 200.25
base_current = np.abs(dft_results_high[2][0][0])
indices = np.where((dft_results_high[2][0] > (2 * base_current)) ==
     1)[0]
if len(indices)>0:
    fault_crossing_threshold = np.where((dft_results_high[2][0] >
    (2 * base_current)) == 1)[0][0] * 1 / sampling_frequency * 1e3
    if fault_crossing_threshold:
        if fault_crossing_threshold >=0:
            fault_crossing_threshold = fault_crossing_threshold
            print(f'High Impedance Fault : Delay from Fault
    Inception : {fault_detection_time_ied -
    fault_crossing_threshold}ms')
else:
    print(f'High Impedance Fault : Delay from Fault Inception : No
    Fault Detected!!')
```

For low impedance:

```
Fault Inception Time for Low Impedance:

    Low Impedance Fault :  Delay from Fault Inception :
    18.5ms
```

The following code generates the above output:

```
    fault_detection_time_ied = 200.25
base_current = np.abs(dft_results_low[2][0][0])
fault_crossing_threshold = np.where((dft_results_low[2][0] > (2 *
    base_current)) == 1)[0][0] * 1 / sampling_frequency * 1e3
print(f'Low Impedance Fault : Delay from Fault Inception : {
    fault_detection_time_ied - fault_crossing_threshold}ms')
```

Part (e)

Apply a short-time DFT - half cycle and quarter cycle, which enable quick detection of the high fundamental current. What is the difference between full cycle DFT?

Half Cycle DFT

The following Figure 11 gives the plot of the DFT considering half a cycle as the window size for High Impedance.

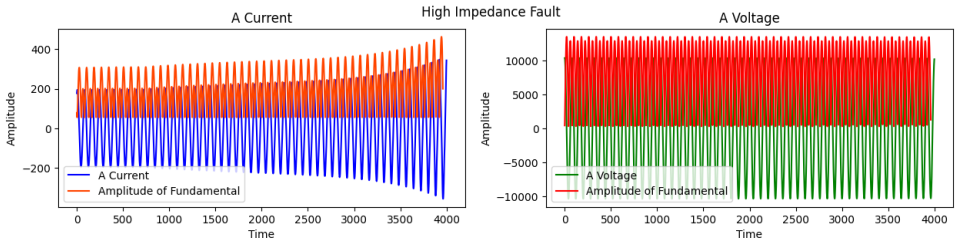

Figure 11: Half Cycle DFT for High Impedance

The following Figure 12 gives the plot of the DFT considering half a cycle as the window size for Low Impedance.

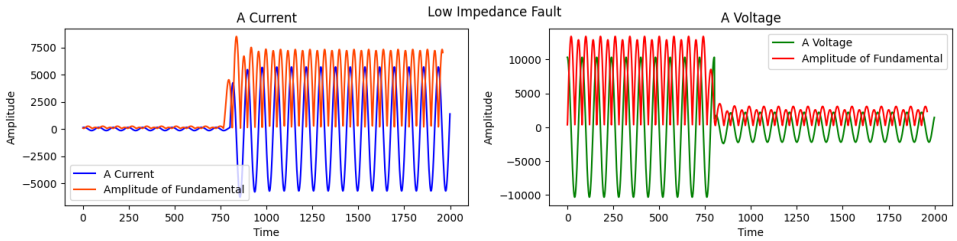

Figure 12: Half Cycle DFT for Low Impedance

The following code generates the data for half cycle windows:

```
window_size = 40  # Define the window size

sampling_frequency = 4000

half_cycle_dft_high_impedance_fault_A_current = recursive_dft(
    df_high_impedance_fault_A_current,window_size,component = 0)
half_cycle_dft_high_impedance_fault_A_voltage = recursive_dft(
    df_high_impedance_fault_A_voltage,window_size,component = 0)

half_cycle_dft_low_impedance_fault_A_current = recursive_dft(
    df_low_impedance_fault_A_current,window_size,component = 0)
half_cycle_dft_low_impedance_fault_A_voltage = recursive_dft(
    df_low_impedance_fault_A_voltage,window_size,component = 0)
```

Quarter Cycle DFT

The following Figure 13 gives the plot of the DFT considering quarter of a cycle as the window size for High Impedance.

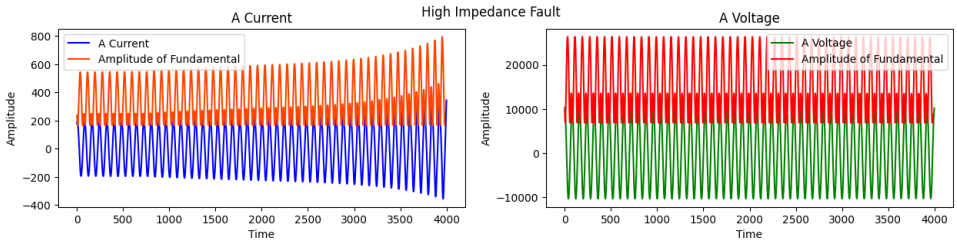

Figure 13: Quarter Cycle DFT for High Impedance

The following Figure 14 gives the plot of the DFT considering quarter of a cycle as the window size for Low Impedance.

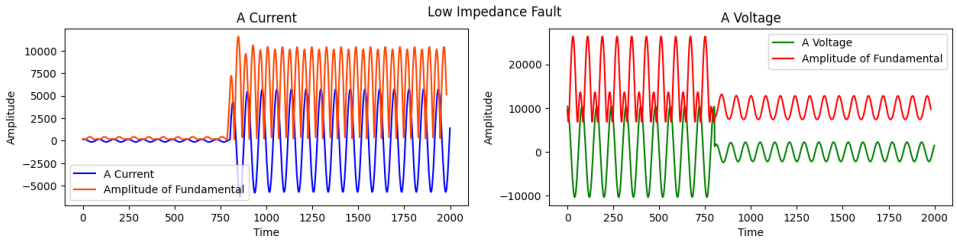

Figure 14: Quarter Cycle DFT for Low Impedance

The following code is used to generate the data:

```
window_size = 20  # Define the window size

sampling_frequency = 4000

quarter_cycle_dft_high_impedance_fault_A_current = recursive_dft(
    df_high_impedance_fault_A_current,window_size,component = 0)
quarter_cycle_dft_high_impedance_fault_A_voltage = recursive_dft(
    df_high_impedance_fault_A_voltage,window_size,component = 0)

quarter_cycle_dft_low_impedance_fault_A_current = recursive_dft(
    df_low_impedance_fault_A_current,window_size,component = 0)
quarter_cycle_dft_low_impedance_fault_A_voltage = recursive_dft(
    df_low_impedance_fault_A_voltage,window_size,component = 0)
```

The following Table **??** is a comparison of the time taken to detect high fundamental current in high and low impedances when considering full, half and quarter cycle.

Reasons for Observations in Delay from Fault Inception

Delay from Fault Inception (in msec)		
	High Impedance	Low Impedance
Full Cycle	No fault detected	18.25
Half Cycle	No fault detected	8.50
Quarter Cycle	No fault detected	4.00

The observations of delay from fault inception in milliseconds for different DFT analysis cycles (full cycle, half cycle, and quarter cycle) may vary due to the following reasons:

1. **Full Cycle DFT:**
   - In full cycle DFT analysis, the entire cycle of the waveform is considered for frequency analysis.
   - Due to the longer duration of the signal analyzed, the detection of faults may take more time, leading to a potentially longer delay from fault inception.
   - Consequently, the delay in fault detection for both high and low impedance faults is higher compared to shorter cycle analysis methods.
2. **Half Cycle DFT:**
   - Half cycle DFT analyzes only half of the cycle of the waveform.
   - By considering a shorter duration of the signal, the analysis can detect faults more quickly than full cycle analysis.
   - Therefore, the delay in fault detection for both high and low impedance faults is reduced compared to full cycle analysis.
3. **Quarter Cycle DFT:**
   - Quarter cycle DFT further reduces the duration of the signal analyzed to only a quarter of the cycle.
   - This shorter duration allows for even quicker fault detection compared to half cycle and full cycle analysis.
   - As a result, the delay in fault detection for both high and low impedance faults is the lowest among the three analysis methods.

In summary, the delay in fault detection decreases as the duration of the analyzed signal decreases. Full cycle DFT has the longest delay, followed by half cycle DFT, and quarter cycle DFT has the shortest delay due to the varying lengths of the analyzed signal and their impact on the detection sensitivity and speed.

Part (f)

Use an estimation algorithm and find the frequency trend of the power supply (use on voltage waveform, check the frequency estimation chapter of *A.G. Phadke, J.S. Thorp, "Synchronized Phasor Measurements and Their Applications", Springer)*

The code to estimate the frequency is:

```
    def estimate_frequency_trend(voltage_data, sampling_rate,
    stride = 1, window_size=None, window_func=None):

    # Define window function (if provided)
    if window_func is not None:
        window = window_func(window_size)

    else:
        window = np.ones(window_size)

    # Sliding window and frequency estimation
    frequencies = []
    time_stamps = []

    if window_size is None:
        # Single DFT for entire data
        fft = np.fft.fft(voltage_data * window)
        dominant_freq = np.abs(fft).argmax() / len(voltage_data) *
    sampling_rate
        frequencies.append(dominant_freq)
        time_stamps.append(len(voltage_data) / sampling_rate)

    else:
        # Sliding window approach
        for i in range(0, len(voltage_data) - window_size + 1,
    stride):
            windowed_data = voltage_data[i:i+window_size] * window
            fft = np.fft.fft(windowed_data)
            dominant_freq = np.abs(fft).argmax() / window_size *
    sampling_rate
            dominant_freq = min(dominant_freq, sampling_rate -
    dominant_freq)
            frequencies.append(dominant_freq)
            time_stamps.append(i / sampling_rate)

    return np.array(frequencies), np.array(time_stamps)
```

The Figure 15 and Figure 16 show the estimated frequency and the amplitude of the voltage for High Impedance and Low Impedance respectively.

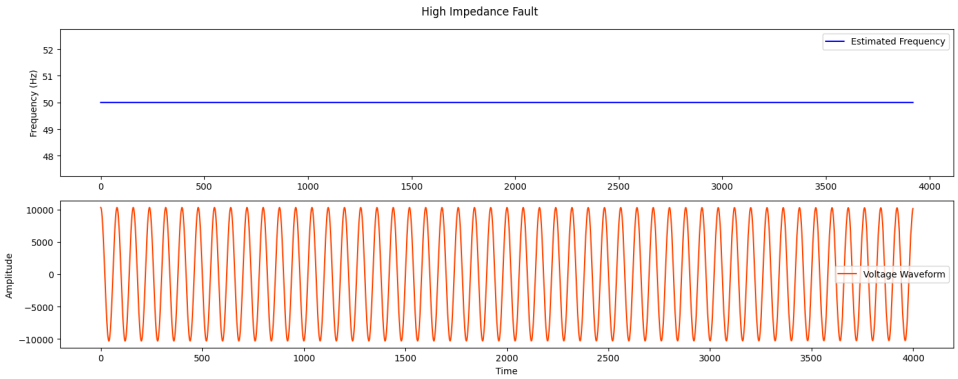

Figure 15: Frequency Estimation for High Impedance Fault

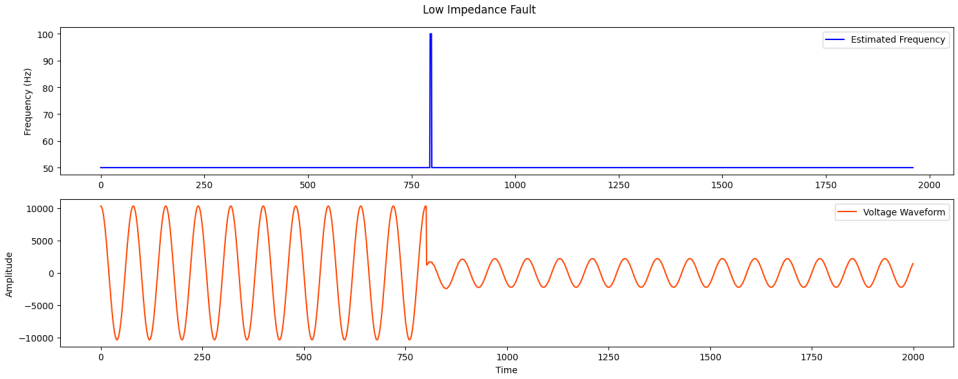

Figure 16: Frequency Estimation for Low Impedance Fault

Question 2

Switching On of large loads such as transformers causes inrush phenomena in the network. This will cause the non-fundamental frequency to present in the voltages and currents (fundamental is 50Hz). Power electronic devices introduce non-sinusoidal currents (mainly when not compensated). These are due to the non-linearities involved in the equipment and might cause the relays to pick up even though the events are Power Quality events, not faults.

The following Figure 17 is the visualization of the current and voltage for a rectifier without a capacitor.

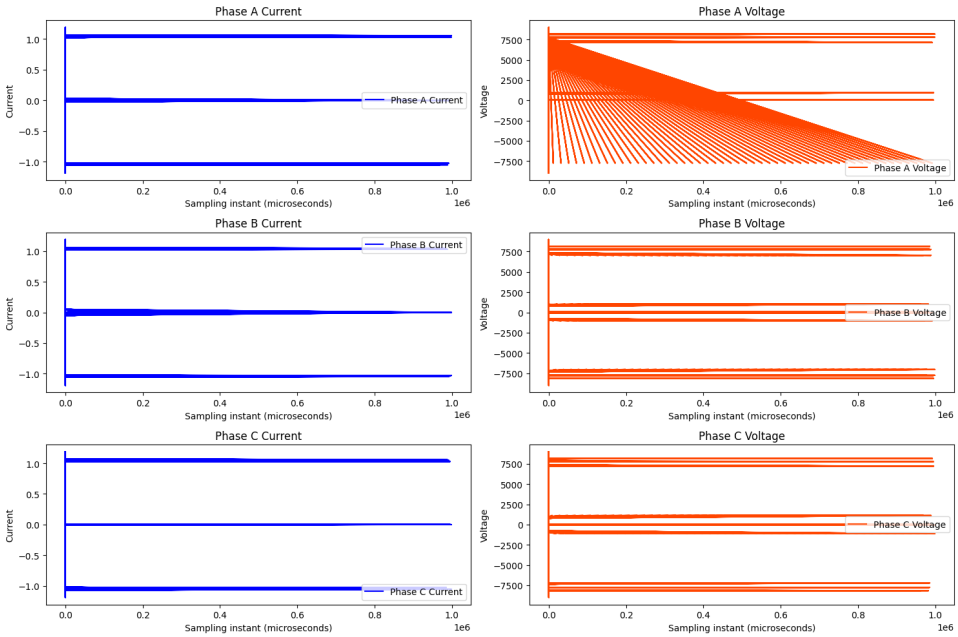

Figure 17: Plot of Current and Voltage Curves for rectifier without capacitor

The following Figure 18 is the visualization of the current and voltage curves for a transformer with load 10.

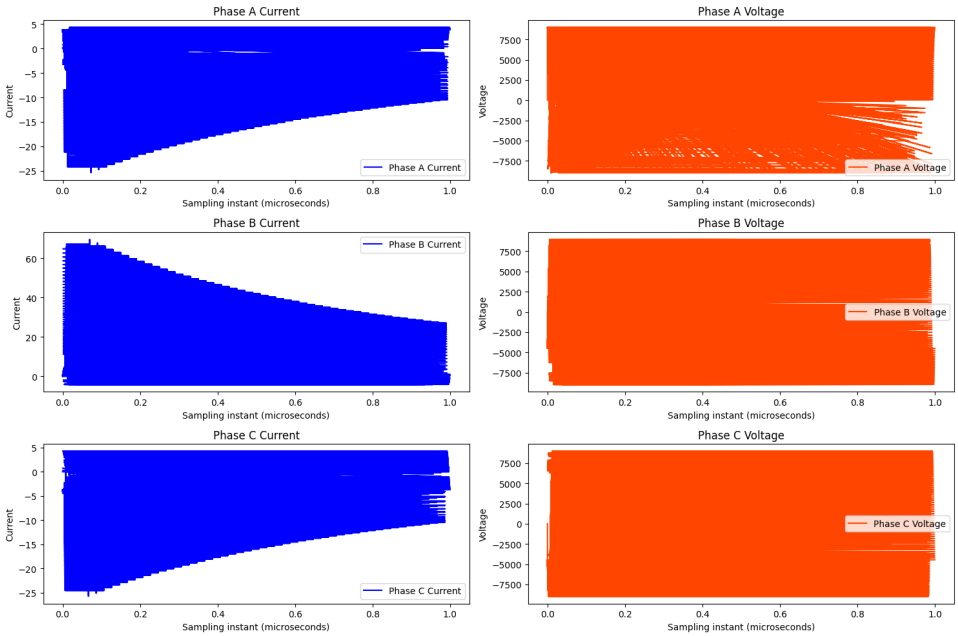

Figure 18: Plot of Current and Voltage Curves for transformer with load 10

Part (a)

Use "rectifier with cap.txt" and "trafo with load 10.txt" each, and note down the signal count, sampling frequency, samples per cycle, number of windows and number of outputs per signal.

```
Analysis of signals of Rectifier without capacitor:
    Signal Count:  9
    Sampling Frequency:  200000 Hz
    Samples Per Cycle:  4000
    Number of Windows:  297685
    Number of Outputs Per Signal:  297685
```

```
Analysis of signals of Transformer with load 10:
    Signal Count:  12
    Sampling Frequency:  200000 Hz
    Samples Per Cycle:  4000
    Number of Windows:  96287
    Number of Outputs Per Signal:  96287
```

The code that generates the above output is given below:

```python
def plot_currents_voltages(df):

    # Extract data
    sampling_frequency = round(1 / (df.iloc[2, 0] - df.iloc[1, 0]))  # Sampling frequency in Hz
    samples_per_cycle = int(sampling_frequency / 50)  # Samples per cycle
    signal_count = df.shape[1] - 1  # Number of signals (currents and voltages excluding first column)
    window_size = samples_per_cycle
    number_of_windows = df.shape[0] - window_size # Number of windows
    number_of_outputs_per_signal = number_of_windows

    print("Signal Count:", signal_count)
    print("Sampling Frequency:", sampling_frequency, "Hz")
    print("Samples Per Cycle:", samples_per_cycle)
    print("Number of Windows:", number_of_windows)
    print("Number of Outputs Per Signal:", number_of_outputs_per_signal)

    # Plotting
    plt.figure(figsize=(15, 10))

    # Plot currents and voltages for each phase
    for i in range(3):
        plt.subplot(3, 2, i * 2 + 1)  # Current subplot
```

```
        plt.plot(df.iloc[1:, 0], df.iloc[1:, i + 1], label=f'Phase {chr(65 + i)} Current', color='blue')
        plt.xlabel('Sampling instant (microseconds)')
        plt.ylabel('Current')
        plt.title(f'Phase {chr(65 + i)} Current')
        plt.legend()

        plt.subplot(3, 2, i * 2 + 2)  # Voltage subplot
        plt.plot(df.iloc[1:, 0], df.iloc[1:, i + 4], label=f'Phase {chr(65 + i)} Voltage', color='orangered')
        plt.xlabel('Sampling instant (microseconds)')
        plt.ylabel('Voltage')
        plt.title(f'Phase {chr(65 + i)} Voltage')
        plt.legend()

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

Part (b)

Use a full-cycle DFT to find the amplitude of the fundamental. Group and plot all voltages and currents with appropriate axis details. Provide amplitude and waveform combined plots for one phase current and respective phase voltage, and comment on the traced envelope.

The DFT of Phase A current and voltage in the rectifier without capacitor if given in the following Figure 19

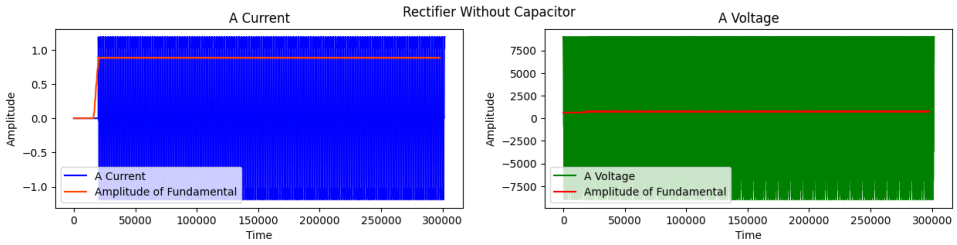

Figure 19: DFT of Phase A current and voltage of rectifier without capacitor

The DFT of Phase A current and voltage in the transformer with load10 if given in the following Figure 20

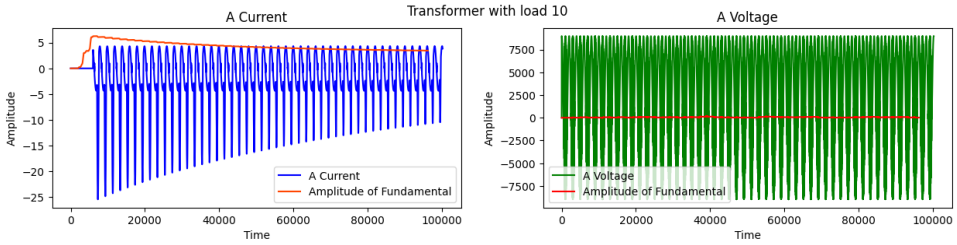

Figure 20: DFT of Phase A current and voltage of transformer with load10

Part (c)

The current signals pose two main components of interest: the fundamental and the harmonics; what are the predominant harmonics present? Plot the Total Harmonic Distortion in voltage and currents. If an IED must be made immune during the switch-on, how can it be done (based on the harmonic content)?

The following is the code for computing the Total Harmonic Distortion:

```
def compute_thd(fft_freq, fft_mag, fundamental_freq):

    # Find the index corresponding to the fundamental frequency
    fundamental_index = np.argmin(np.abs(fft_freq -
    fundamental_freq))

    # Exclude the fundamental frequency from the FFT
    harmonics_mag = np.delete(fft_mag, 0)
    harmonics_mag = np.delete(fft_mag, fundamental_index) * 10

    # Compute THD
    thd = np.sqrt(np.sum(harmonics_mag**2)) / fft_mag[
    fundamental_index]

    return thd
```

```
 Dominant Frequencies present in Rectifier without
Capacitor
     Top Frequencies present:  [ 0.  50.  100.  150.]Hz
```

The Figure 21 gives a plot of the predominant frequencies present in the current of rectifier without capacitor.

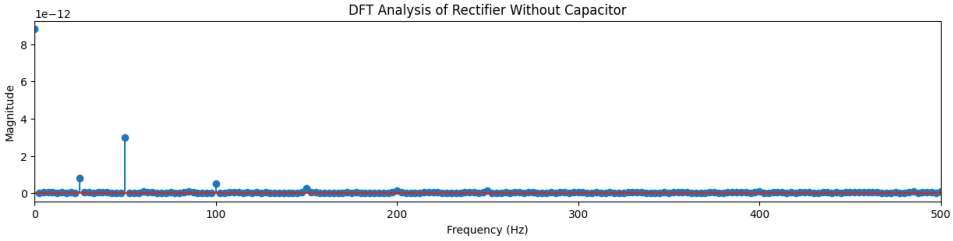

Figure 21: Harmonics in the current of rectifier without capacitor

The code for displaying the above output is:

```
# Rectifier Without Capacitor

plt.figure(figsize=(15, 3))

timestep = 1 / sampling_frequency
```

```
freq = np.fft.fftfreq(rectifier_without_cap_current[:
    sampling_frequency // 10].shape[0], d=timestep) / 4
DFT = np.abs(np.fft.fft(rectifier_without_cap_current[:
    sampling_frequency // 10]))

plt.stem(freq, DFT, linefmt='m', markerfmt='mp', basefmt='k')
plt.xlim(0, 500)
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.title('DFT Analysis of Rectifier Without Capacitor')
print(f'Top Frequencies present: {freq[np.argsort(DFT[:len(DFT) //
    2])[::-1][:4]}Hz')
plt.show()

fundamental_freq = 50  # Fundamental frequency in Hz
thd = compute_thd(freq, DFT, fundamental_freq)
print(f"THD: {thd:.2f}%")
```

```
THD present in Rectifier Current without Capacitor
    THD: 25.4%
```

```
Dominant Frequencies present in Transformer with Load 10
    Top Frequencies present:  [ 50.  0.  100.  150.]Hz
```

The Figure 22 gives a plot of the predominant frequencies present in the current of Transformer with load 10.

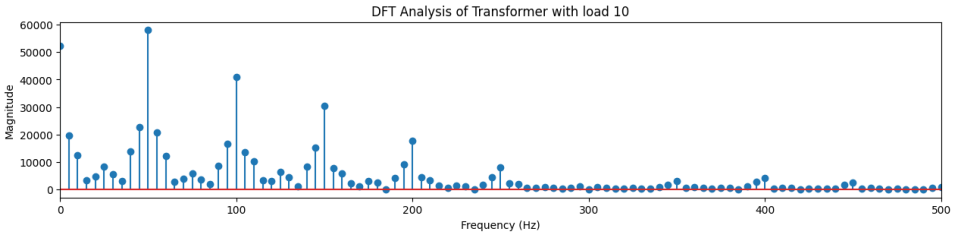

Figure 22: Harmonics in the current of transformer with load 10

```
THD present in Transformer Current with load 10
    THD: 33.7%
```

The following code gives the above output:

```
# Transformer With Load 10
```

```
2
3 plt.figure(figsize=(15, 3))
4
5 timestep = 1 / sampling_frequency
6 freq = np.fft.fftfreq(transformer_with_load_10_current[:
      sampling_frequency // 10].shape[0], d=timestep)/2
7 DFT = np.abs(np.fft.fft(transformer_with_load_10_current[:
      sampling_frequency // 10]))
8
9 plt.stem(freq, DFT, linefmt='m', markerfmt='mp', basefmt='k')
10 plt.xlim(0, 500)
11 plt.xlabel('Frequency (Hz)')
12 plt.ylabel('Magnitude')
13 plt.title('DFT Analysis of Transformer with Load 10')
14 print(f'Top Frequencies present: {freq[np.argsort(DFT[:len(DFT) //
      2])[::-1]][:4]}Hz')
15 plt.show()
16
17 fundamental_freq = 50  # Fundamental frequency in Hz
18 thd = compute_thd(freq, DFT, fundamental_freq)
19 print(f"THD: {thd:.2f}%")
```

Part (d)

The voltage signals are used for frequency measurements; plot the estimated frequency before and after turn-on.

The plot of estimated frequency for the rectifier without capacitor is given in Figure 23.

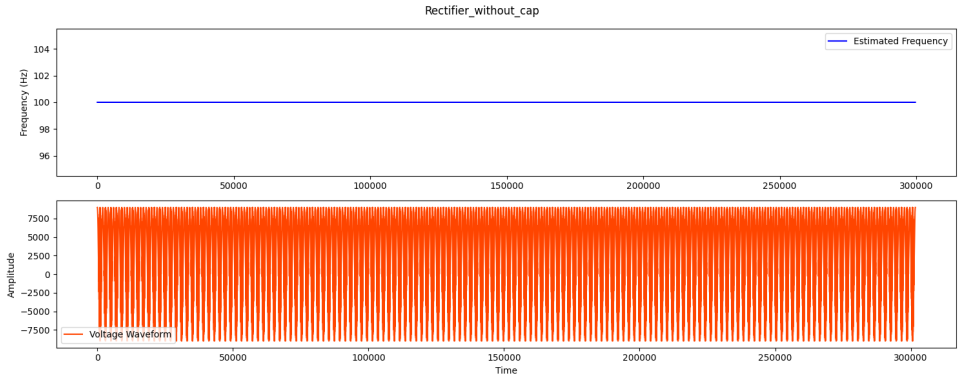

Figure 23: Frequency estimation of rectifier voltage without capacitor

The plot of the estimated frequency for the transformer voltage with load 10 is given in Figure 24.

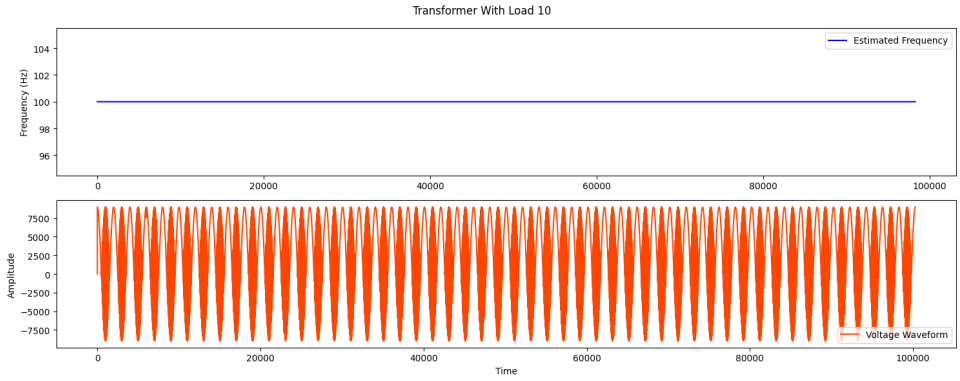

Figure 24: Frequency estimation of transformer voltage with load 10