Names

1- Mohamed Elsheshtawy

2- Mehta, Saanvi

3- Parkansky, Brandon

In [3]:
```
!pip install -q numpy matplotlib
```

In [4]:
```
import numpy as np
import matplotlib.pyplot as plt
import math

plt.rcParams["figure.dpi"] = 100
```

0 Useful functions

We provide several efficient implementations of functions that you can use throughout the lab.

In [6]:
```
def cexp(k, N):
    """
    Creates complex exponential with frequency k and length N.

    Args:
        k: discrete frequency
        N: length

    Returns:
        a numpy array of length N.
    """

    n = np.arange(N)
    return np.exp(2j*np.pi*k*n/N) / np.sqrt(N)

def inner_product(x, y):
    """
    Computes the inner product between two numpy arrays x and y.

    Args:
        x: numpy array.
        y: numpy array.
    """
    return x @ np.conj(y.T)

def plot_signal_and_dft(x, t, X, f, title=None):
    """
    : X.
```

```
    Args:
        x: (N,) Signal of length N.
        t: (N,) times
        X: (N,) DFT of x.
        f: (N,) frequencies
    """
    fig, ax = plt.subplots(2, 1, figsize=(6,4))
    fig.suptitle(title)

    ax[0].plot(t, x)
    ax[0].set_xlabel("n (s)")
    ax[0].set_ylabel("x(n)")
    ax[0].grid(True)

    ax[1].plot(f, np.abs(X))
    ax[1].set_xlabel("f (Hz)")
    ax[1].set_ylabel("$|(\mathcal{F}x)(f)|$")
    ax[1].grid(True)
```

```
<>:46: SyntaxWarning: invalid escape sequence '\m'
<>:46: SyntaxWarning: invalid escape sequence '\m'
C:\Users\flash\AppData\Local\Temp\ipykernel_24524\3496616944.py:46: SyntaxWarnin
g: invalid escape sequence '\m'
  ax[1].set_ylabel("$|(\mathcal{F}x)(f)|$")
```

1 Spectrum of pulses

1.1 Computation of the DFT

In [64]:
```
def compute_dft(x, fs: int):
    """
    Compute the DFT of x.

    Args:
        x: Signal of length N.
        fs: Sampling frequency.
    Returns:
        X: DFT of x.
        f: Frequencies
    """
    # TODO: Implement
    N = int(len(x))
    X = np.zeros(N+1)
    i = 0
    for k in range(int(-N/2), int(N/2)+1):
        X[i] = inner_product(x, cexp(k, N))
        i+=1
    f = np.zeros(N+1)
    i = 0
    for j in range(int(-N/2), int(N/2)+1):
        f[i] = (fs*j)/N
        i+=1
    return X, f
```

Response here.

Frequencies of range [-N/2, -1] is equivalent to range [N/2, N-1]. So we can just chop and shift those frequencies forward.

1.2 DFTs of square pulses

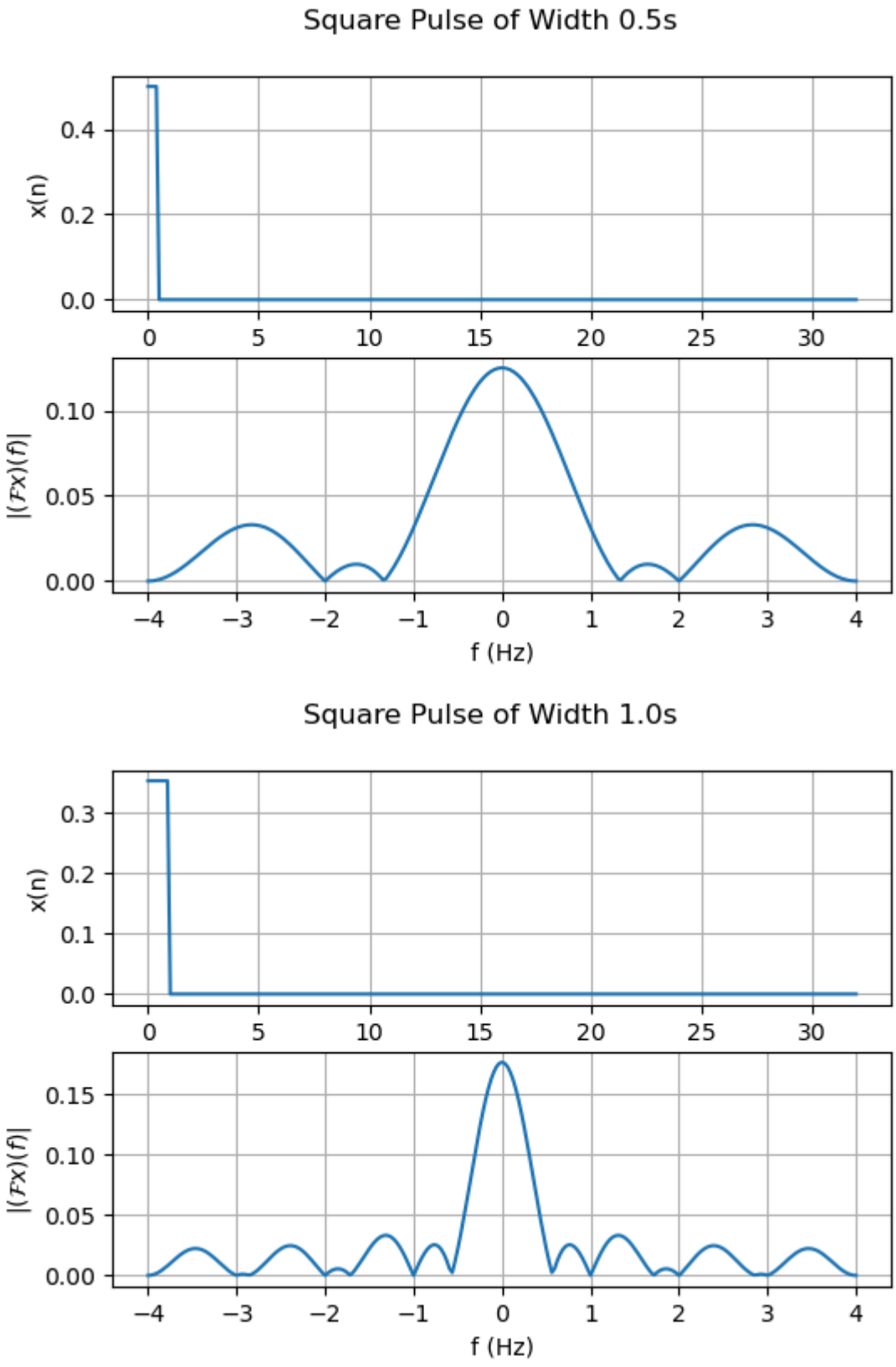
In [68]:
```
def make_square_pulse(T0: float, T: float, fs: int):
    """
    Args:
        T0: The width of the pulse in seconds.
        T: The duration of the pulse.
        fs: The sampling frequency
    """
    # TODO: Implement
    Ts = 1/fs
    N = int(T/Ts)
    M = int(T0/Ts)
    x = np.zeros(N)
    for i in range(N):
        if i < M:
            x[i] = 1/math.sqrt(M)
        else:
            x[i] = 0
    return x
```

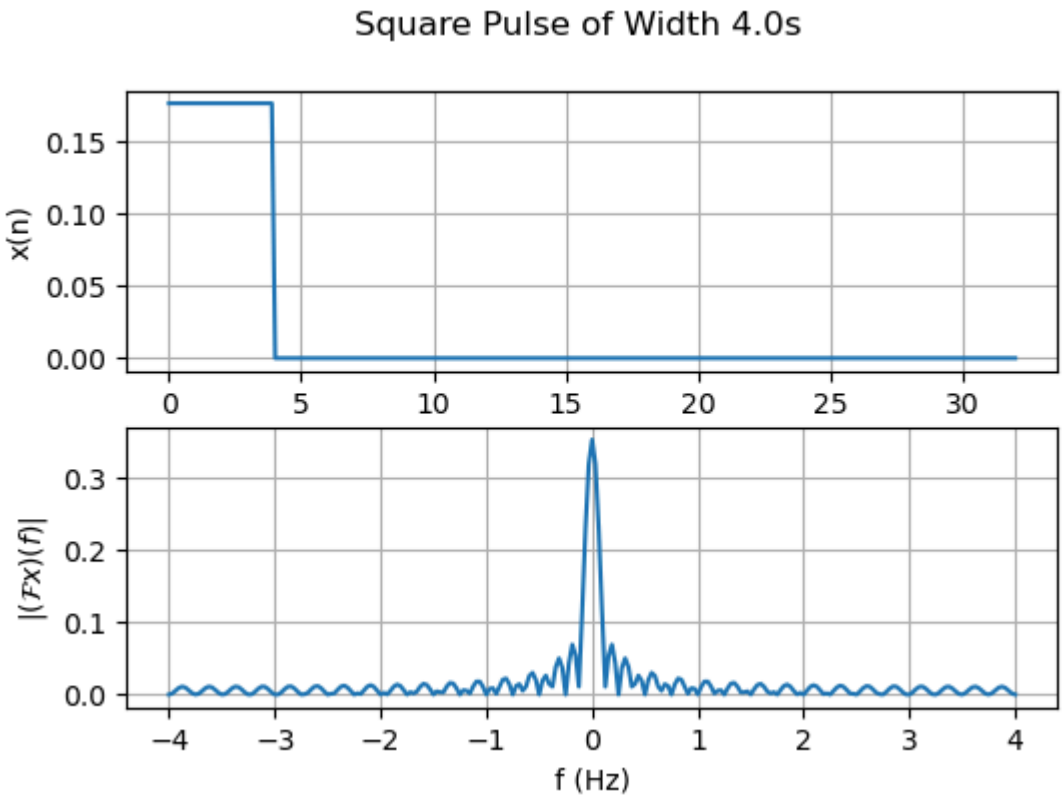
In [70]:
```
def compute_dft_energy_fraction(X, f, T0):
    """
    Compute the fraction of energy of X within [-1/T0,1/T0].

    Return:
        fraction: a floating number between 0 and 1
    """
    # TODO: Implement
    energy_total = inner_product(X,X)
    N = len(X)
    mask = (f >= -1/T0) & (f <= 1/T0)
    energy_center = np.sum(np.abs(X[mask]) ** 2)
    return energy_center / energy_total
```

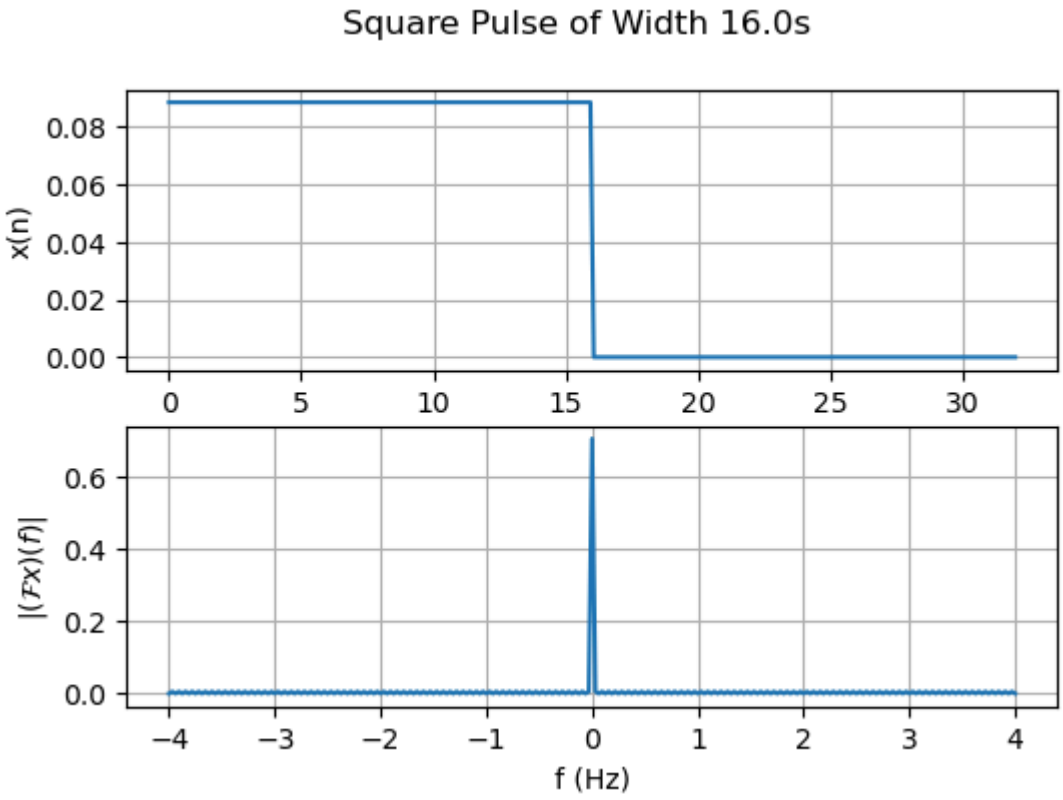
In [72]:
```
T = 32.0
fs = 8
print(f"{'T0': <4} | {'Fraction': <8}")
for T0 in [0.5, 1.0, 4.0, 16.0]:
    t = np.linspace(0, T, int(T*fs))
    x = make_square_pulse(T0, T, fs)
    X, f = compute_dft(x, fs)

    energy_fraction = compute_dft_energy_fraction(X, f, T0)
    print(f"{T0: <4} | {energy_fraction: <8}")

    title = f"Square Pulse of Width {T0}s"
    plot_signal_and_dft(x, t, X, f, title=title)
```

```
T0   | Fraction
0.5  | 0.9074366512328453
1.0  | 0.9085943925524305
4.0  | 0.9352428553572522
```

```
C:\Users\flash\AppData\Local\Temp\ipykernel_24524\68971169.py:17: ComplexWarning:
Casting complex values to real discards the imaginary part
  X[i] = inner_product(x, cexp(k, N))
```







1.3 DFTs of triangular pulses

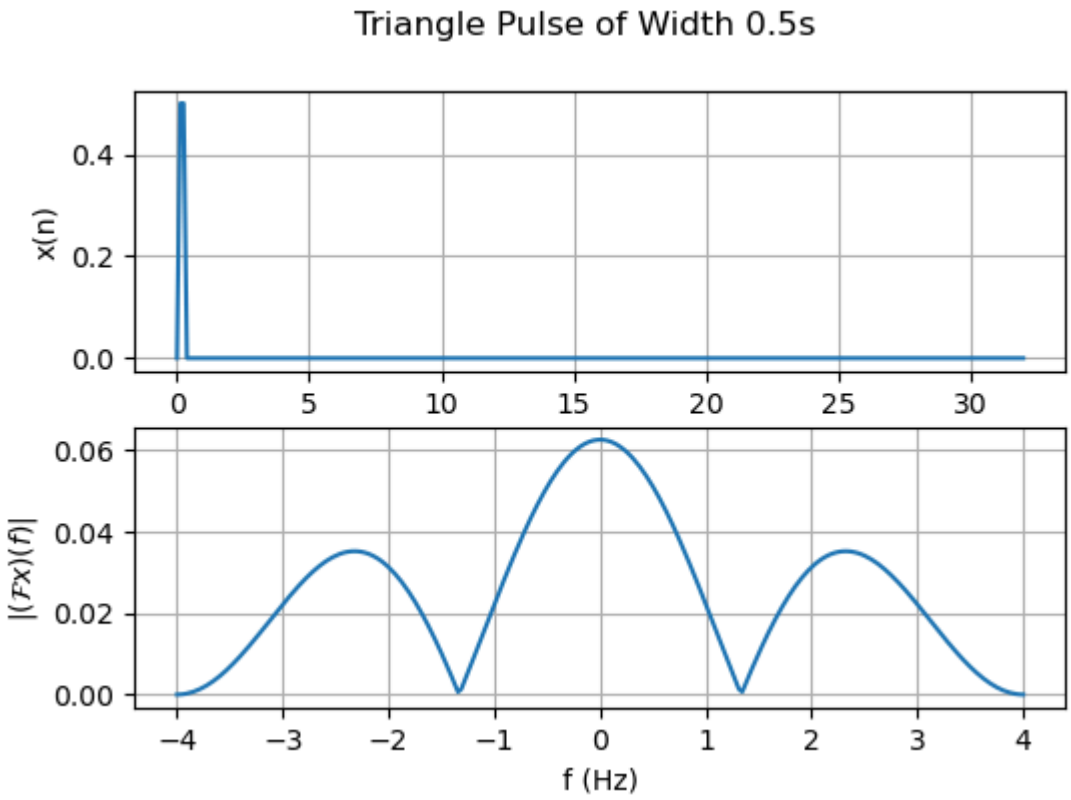
In [74]:
```
def make_triangle_pulse(T0: float, T: float, fs: int):
    """
    Args:
        T0: The width of the pulse in seconds.
        T: The duration of the pulse.
        fs: The sampling frequency
    """
```

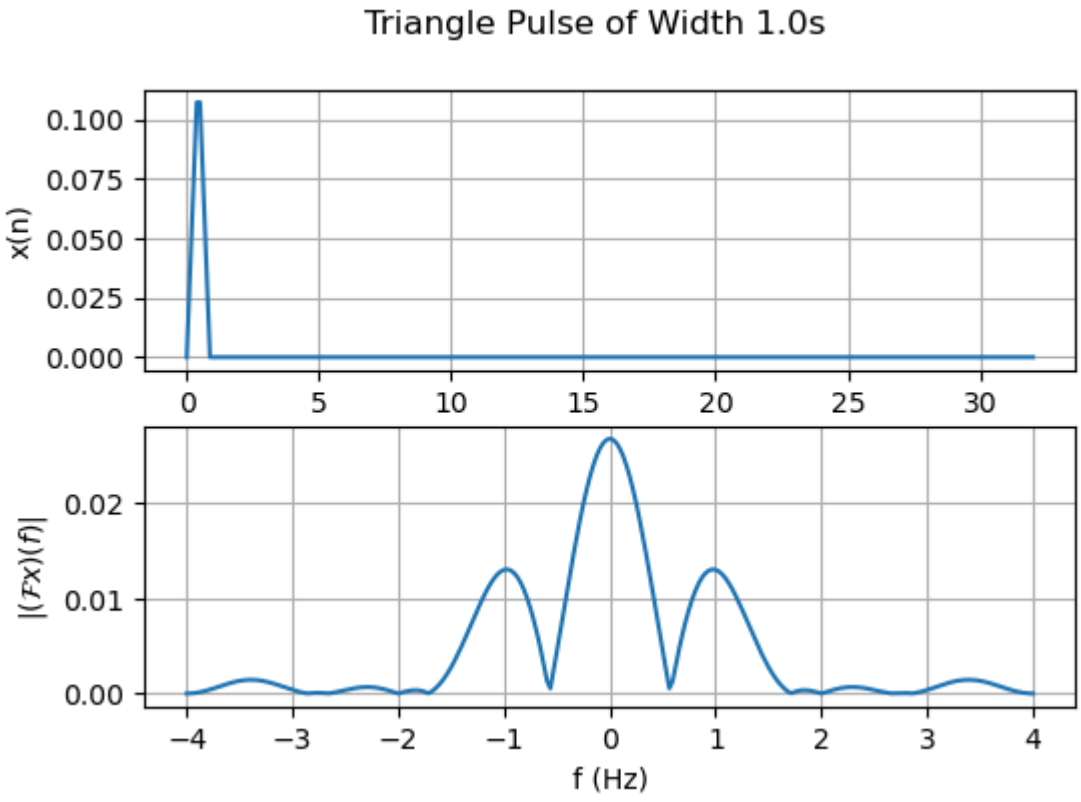
```
    # TODO: Implement
    Ts = 1/fs
    M = int(T0/Ts)
    N = int(T/Ts)
    x = np.zeros(N)
    for i in range(N):
        if i <M/2:
            x[i] = i
        elif i>=M/2 and i<M:
            x[i] = (M-1)-i
        else:
            x[i] = 0
    energy_total = abs(inner_product(x,x))
    x/=energy_total
    return x
```

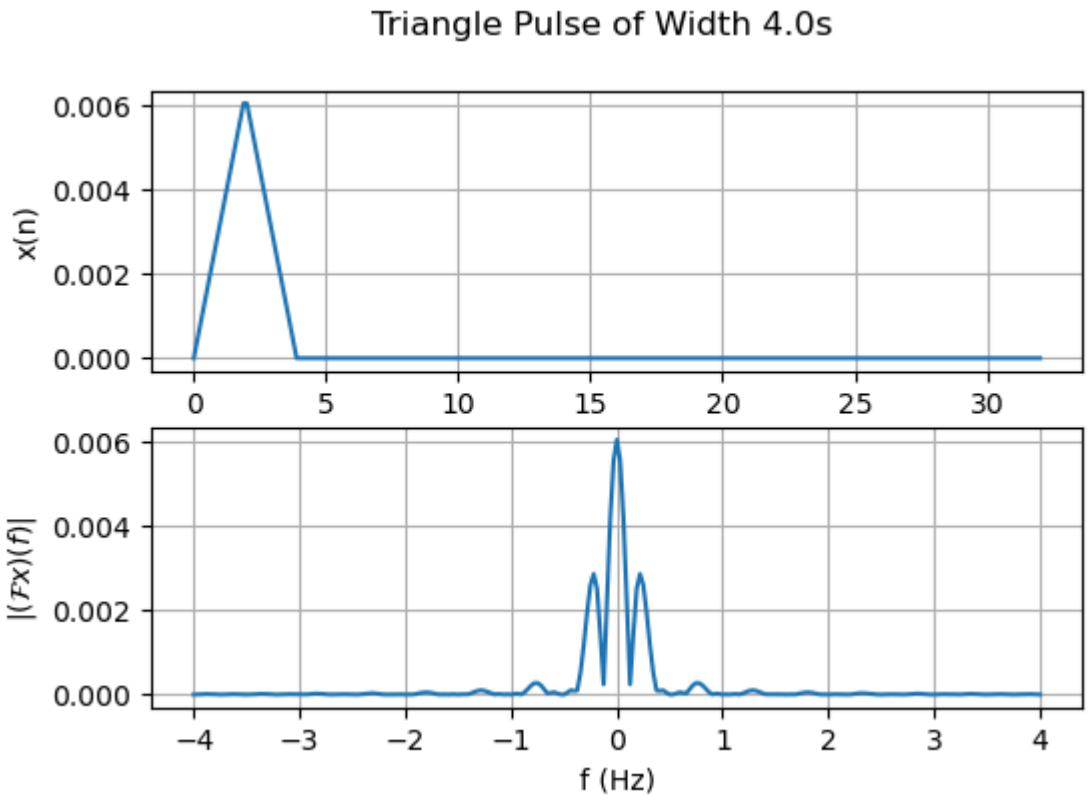
In [76]:
```
T = 32.0
fs = 8
print(f"{'T0': <4} | {'Fraction': <8}")
for T0 in [0.5, 1.0, 4.0, 16.0]:
    t = np.linspace(0, T, int(T*fs))
    x = make_triangle_pulse(T0, T, fs)
    X, f = compute_dft(x, fs)

    energy_fraction = compute_dft_energy_fraction(X, f, T0)
    print(f"{T0: <4} | {energy_fraction: <8}")

    title = f"Triangle Pulse of Width {T0}s"
    plot_signal_and_dft(x, t, X, f, title=title)
```

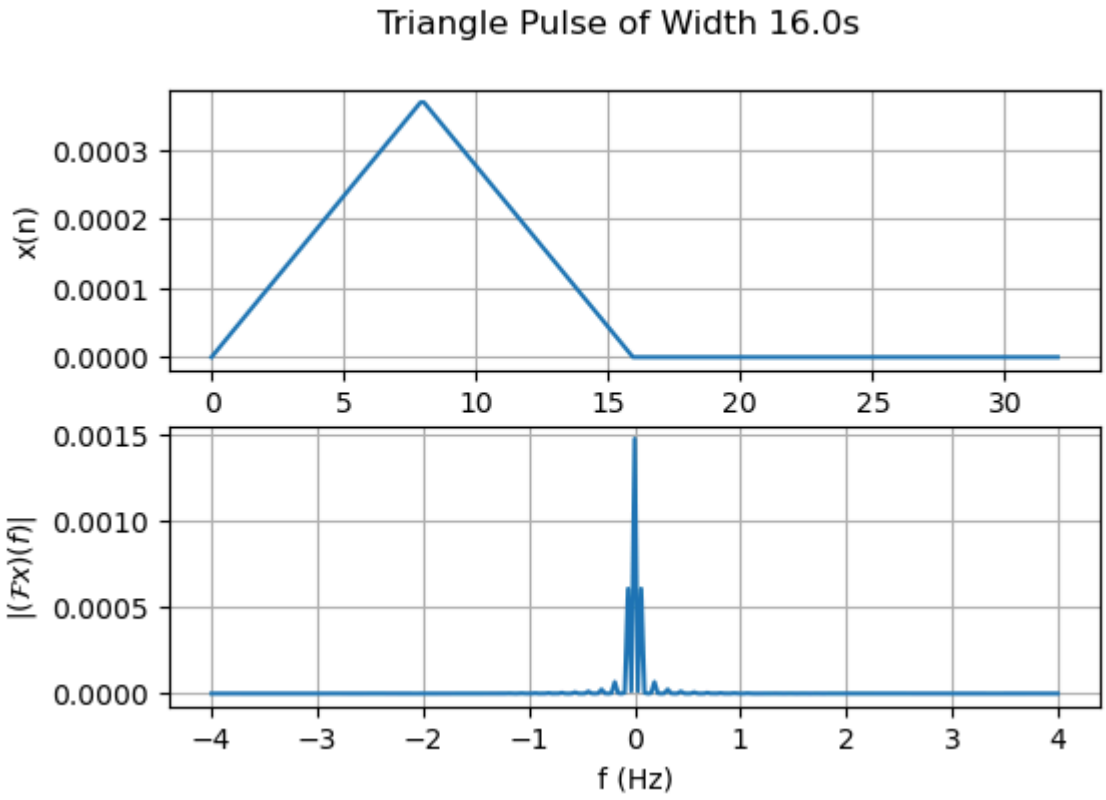
```
T0   | Fraction
0.5  | 0.7161448028939315
1.0  | 0.8546913932756969
4.0  | 0.9422405832131603
16.0 | 0.9963054116391472
```

```
C:\Users\flash\AppData\Local\Temp\ipykernel_24524\68971169.py:17: ComplexWarning:
Casting complex values to real discards the imaginary part
  X[i] = inner_product(x, cexp(k, N))
```









In [78]:
```
#We will calculate the variance for both graphs at each T0 and divide them by ea
T = 32.0
fs = 8
for T0 in [0.5, 1.0, 4.0, 16.0]:
    x1 = make_square_pulse(T0, T, fs)
    x2 = make_triangle_pulse(T0, T, fs)
    X1, f1 = compute_dft(x1, fs)
    X2, f2 = compute_dft(x2, fs)
    var1 = np.var(X1)
```

```
    print("the variance of square pulses to traingular one is = " + str(var1/var
```

```
the variance of square pulses to traingular one is = 1.5038910505836576
the variance of square pulses to traingular one is = 24.527237354085596
the variance of square pulses to traingular one is = 2403.1031128404666
the variance of square pulses to traingular one is = 169364.8774319066
```

```
C:\Users\flash\AppData\Local\Temp\ipykernel_24524\68971169.py:17: ComplexWarning:
Casting complex values to real discards the imaginary part
  X[i] = inner_product(x, cexp(k, N))
```

In [80]:
```
"""
As we can see the variance of the square pulse is higher in the first, second, t
"""
```

Out[80]:
```
'\nAs we can see the variance of the square pulse is higher in the first, secon
d, third and fourth. This is because of the abrupt change in the square pulse
\n'
```

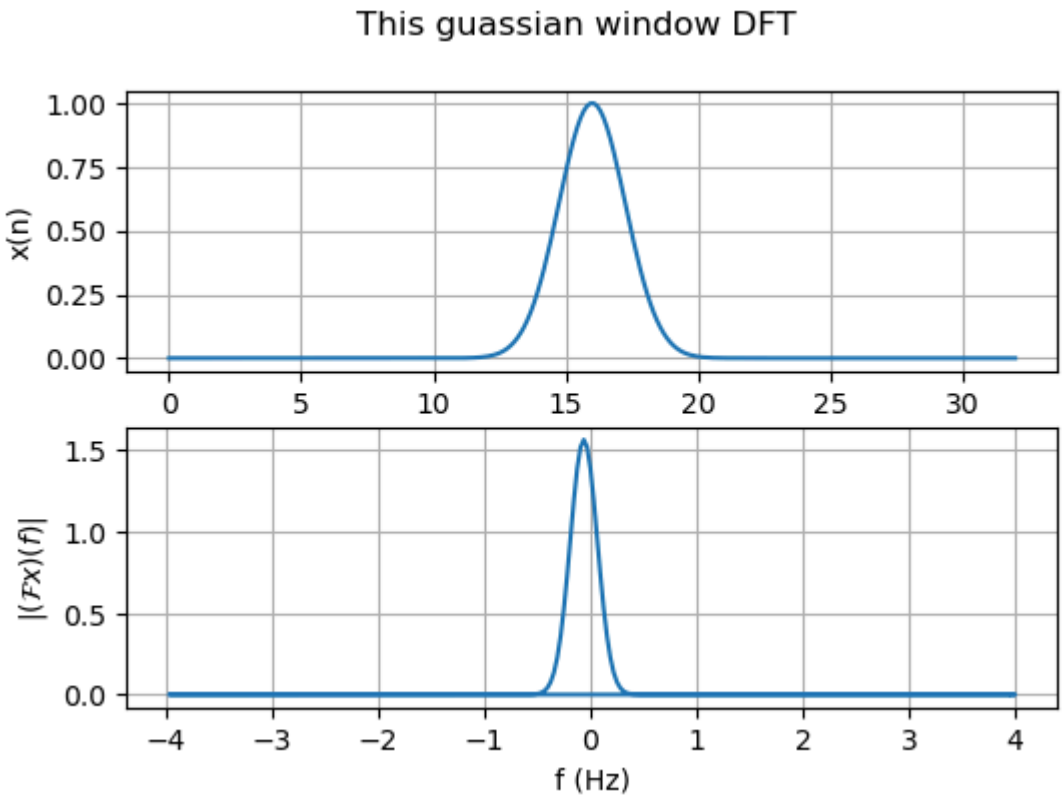
1.4 Other pulses

We will leave this more open ended then the previous examples. Here plot at least one other window, different from the square pulse or triangle pulse. Choose at least one from: Kaiser windows, raised cosine, Gaussian, and Hamming windows.

Feel free to try more than one!

In [39]:
```
# TODO:
def gaussian_window(sigma,T: float, fs: int):
    N = int(T*fs)
    n = np.arange(N)
    return np.exp(-0.5 * ((n - (N-1)/2) / sigma) ** 2)
sigma = 10
T =32.0
fs = 8
x = gaussian_window(sigma,T,fs)
t = np.linspace(0, T, int(T*fs))

# TODO: use `compute_dft` to compute DFT of x
X, f = compute_dft(x, fs)

title = f"This guassian window DFT"
plot_signal_and_dft(x, t, X, f, title=title)
```

```
C:\Users\flash\AppData\Local\Temp\ipykernel_8988\2799019226.py:25: ComplexWarnin
g: Casting complex values to real discards the imaginary part
  X[i] = inner_product(x, cexp(k, N))
```



In [41]:
```
"""
The spectra of gaussian window DFT is nearly equivalent to triangular pulse, but
We can find that its spectra is the same as the signal but a little bit narrow.
the amount of change in it isn't as abrupt as the square or traingular signal as
"""
```

Out[41]:
```
"\nThe spectra of gaussian window DFT is nearly equivalent to triangular pulse,
but it is quite different from the rectangular pulse.\nWe can find that its spe
ctra is the same as the signal but a little bit narrow.\nthe amount of change i
n it isn't as abrupt as the square or traingular signal as a result its spectra
doesn't contian signals with high frequency\n"
```

3 Spectra of musical tones

3.1 DFT of an A Note

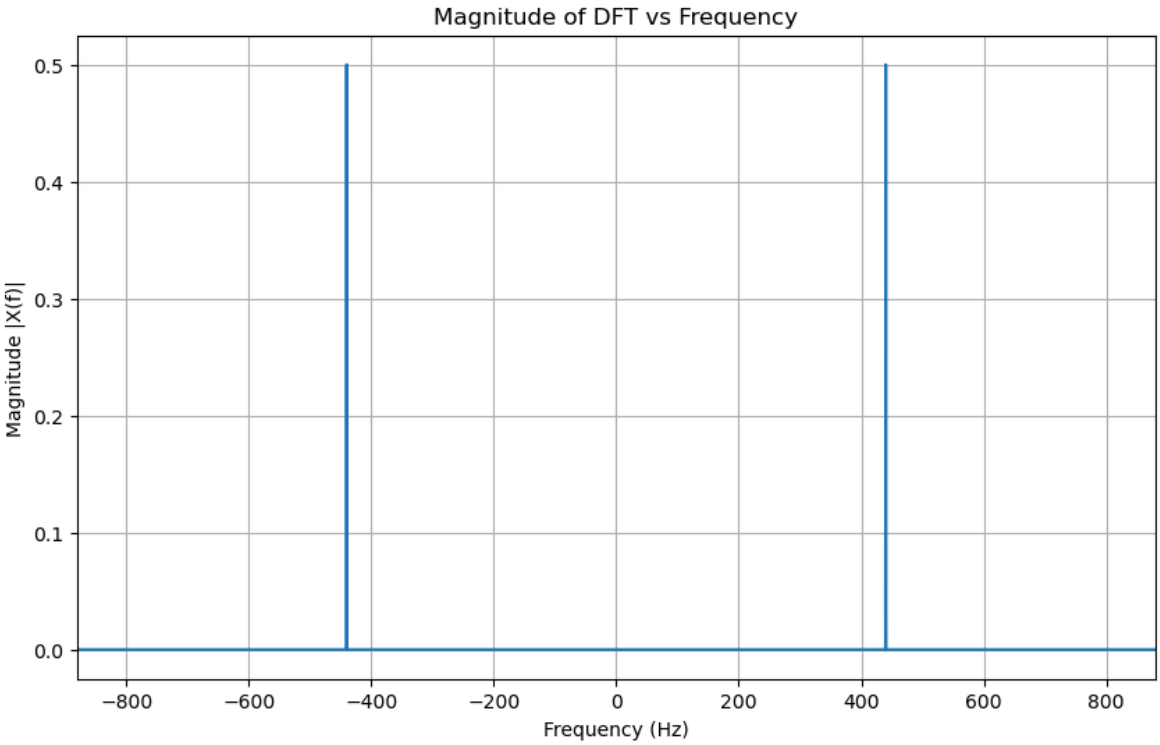
In [47]:
```
def plot_dft(X, f, title=None):
    """
    Given a DFT X and its frequencies f, make a plot of |X| vs f.
    """
    # TODO: Implement
    plt.figure(figsize=(10, 6))
    plt.plot(f, abs(X))  # Plot magnitude of X against f
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Magnitude |X(f)|')

    if title:
        plt.title(title)
    else:
        plt.title('Magnitude of DFT vs Frequency')
```

```
    plt.show()

def key_to_frequency(key):
    f0 = (2**((key-49)/12))*440
    return f0

def make_note(key, T, fs):
    """
    Produce a note of desired key.
    Return:
        x: the signal corresponding to the note.
        t: the times at which the signal is sampled.
    """
    f0 = key_to_frequency(key)
    # TODO: implement
    t = np.linspace(0, T, int(T*fs))
    Ts = 1/fs
    N = math.ceil(T/Ts)
    k = (f0/fs)*N
    x = cexp(k,N)
    x = x.real #since it is a real function
    return x, t
```

In [49]:

```
# Visualize the DFT of an A note
key = 49 # A note
T = 2.0 # duration of the note in seconds
fs = 44100 # sampling freq in Hz

# TODO: make an A note
x, t = make_note(key,T,fs)

def compute_fft(x, fs: int):
    """
    Compute the DFT of x using the FFT algorithm.

    Args:
        x: Signal of length N.
        fs: Sampling frequency.
    Returns:
        X: DFT of x.
        fs: Frequencies
    """
    N = len(x)
    X = np.fft.fft(x, norm="ortho")
    X = np.fft.fftshift(X)
    f = np.fft.fftfreq(N, 1/fs)
    f = np.fft.fftshift(f)
    return X, f

# TODO: use compute_dft to find dft of x
X, f = compute_fft(x,fs)

# TODO: use plot_dft to plot the DFT
plt.figure(figsize=(10, 6))
plt.plot(f, abs(X))  # Plot magnitude of X against f
f0 = key_to_frequency(key)
plt.xlim((-2*f0, 2*f0))
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.title('Magnitude of DFT vs Frequency')
```

```
plt.grid(True)
plt.show()
# Constrain the x axis within [-2f0, 2f0]
```



In [51]:
```
# TODO: confirm conjugate symmetry numerically
def check_conjugate_symmetry(X):
    """
    Checks the conjugate symmetry of the DFT array X.
    Returns True if conjugate symmetry is confirmed, False otherwise.
    """
    N = len(X)
    j = 0
    i= int(N/2)
    while i+j<N:
        if not np.isclose(X[i-j], np.conj(X[i+j])):
            return False
        j+=1
    return True

# Check conjugate symmetry of the DFT

if check_conjugate_symmetry(X):
    print("Conjugate symmetry confirmed.")
else:
    print("Conjugate symmetry not confirmed.")
```

```
Conjugate symmetry confirmed.
```

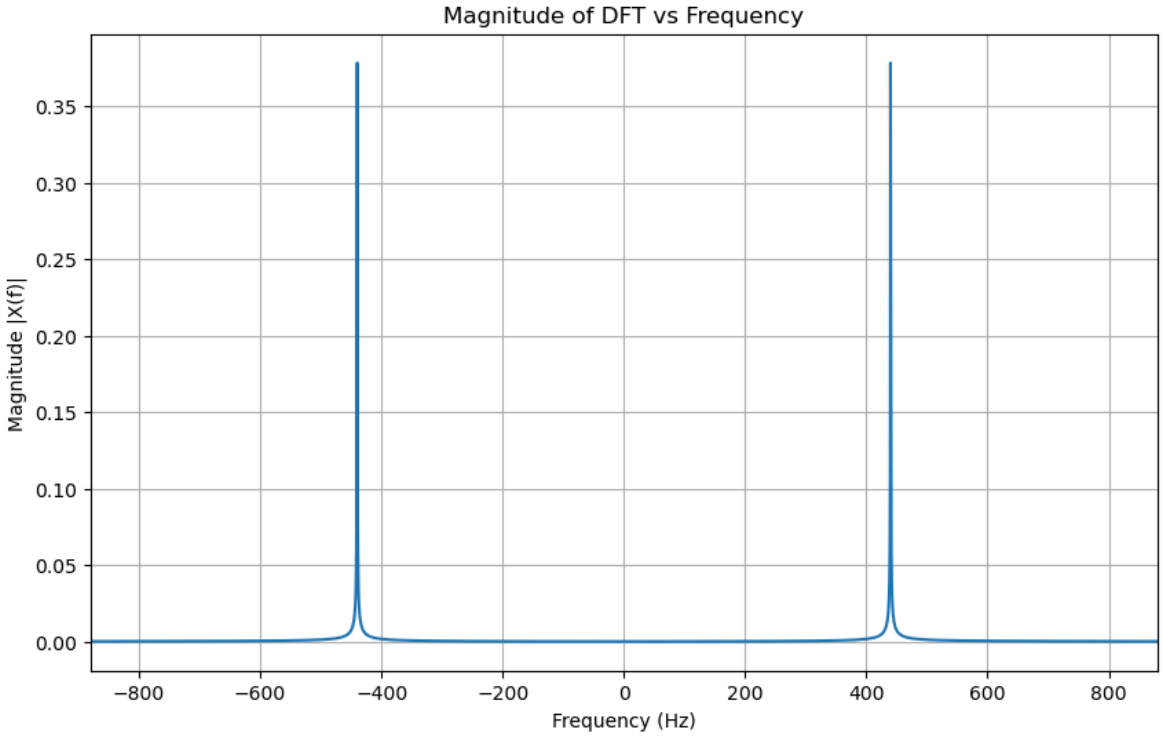
In [53]:
```
# TODO: confirm parseval's theorem numerically
#to confirm it we have to calculate the energy of both
Energy_x = inner_product(x,x)
Energy_X = inner_product(X,X)
print(Energy_X/Energy_x)
#if it is one then theorm is confirmed
```

```
(1+0j)
```

In [57]:
```
# TODO: find a frequency or frequency range that contains at least 90% of the DF
"""
The range is from -f0 to f0 since this is the only parts that has a value
But usually the DFT is symmetric because it is a real signal for the rest of the
if we take the corresponding part on the left
"""
def get_energy_pls(X,f):
    energy = np.abs(X) ** 2
    total_energy = np.sum(energy)
    center_index = len(f) // 2
    right_frequencies = f[center_index:]
    right_energy = energy[center_index:]
    cumulative_right_energy = np.cumsum(right_energy)
    threshold_energy = 0.45 * total_energy
    right_threshold_index = np.argmax(cumulative_right_energy >= threshold_energ
    right_range = (right_frequencies[0], right_frequencies[right_threshold_index
    left_range = (-right_frequencies[right_threshold_index],right_frequencies[0]
    RANGE = (-right_frequencies[right_threshold_index],right_frequencies[right_t
    print("Frequency range containing 45% of the energy to the left:", left_rang
    print("Frequency range containing 45% of the energy to the right:", right_ra
    print("Frequency range containing 90% of the energy:", RANGE)
get_energy_pls(X,f)
```

```
Frequency range containing 45% of the energy to the left: (-440.0, 0.0)
Frequency range containing 45% of the energy to the right: (0.0, 440.0)
Frequency range containing 90% of the energy: (-440.0, 440.0)
```

In [59]:
```
#Lets try the Anote but with T = 2.06
# Visualize the DFT of an A note
key = 49 # A note
T = 2.06 # duration of the note in seconds
fs = 44100 # sampling freq in Hz

# TODO: make an A note
x, t = make_note(key,T,fs)

# TODO: use compute_dft to find dft of x
X, f = compute_fft(x,fs)

# TODO: use plot_dft to plot the DFT
plt.figure(figsize=(10, 6))
plt.plot(f, abs(X))  # Plot magnitude of X against f
f0 = key_to_frequency(key)
plt.xlim((-2*f0, 2*f0))
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude |X(f)|')
plt.title('Magnitude of DFT vs Frequency')
plt.grid(True)
plt.show()
# Constrain the x axis within [-2f0, 2f0]
```



In [61]:
```
"""
We can see that the range of frequencies became wider and it isn't a dirac-delta
The problem is TF0 isn't an integer and as a result the cosine function doesn't
alignment resulting in a spectral leakage.
For example when we apply the dft since it is discrete we won't have the specifi
multiple functions sharing a part with the signal.
"""
```

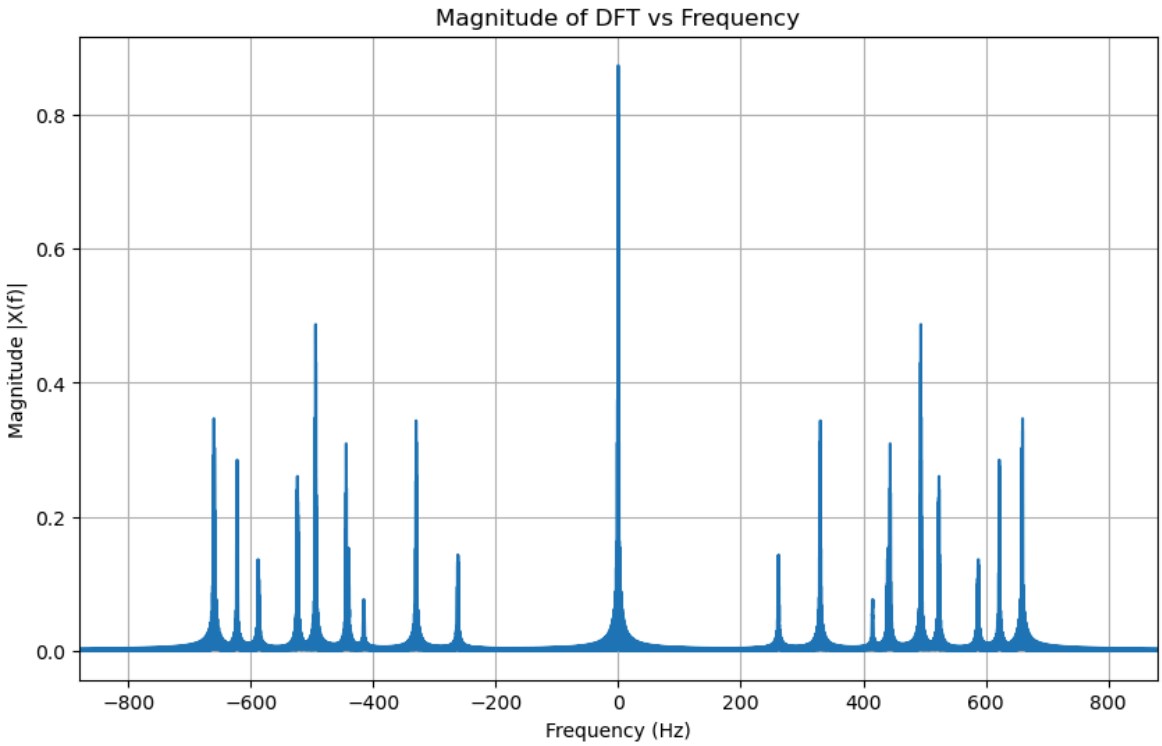
Out[61]:
```
"\nWe can see that the range of frequencies became wider and it isn't a dirac-d
elta function anymore\nThe problem is TF0 isn't an integer and as a result the
cosine function doesn't complete an integer number of cycles, which results in
unperfect\nalignment resulting in a spectral leakage.\nFor example when we appl
y the dft since it is discrete we won't have the specific function that is orth
ogonal with the signal and we will have\nmultiple functions sharing a part with
the signal.\n"
```

3.2 DFT of a Musical Piece

We leave this part to you.

In [64]:
```
def cexpt(f, T, fs):
    Ts = 1/fs
    N = math.ceil(T/Ts)
    k = (f/fs)*N
    x = cexp(k,N)
    return x, N

def q_33(f0, T, fs):
    time = 0.5
    cpxexp,num_samples = cexpt(0,0.5,fs)
    audio_sequence = cpxexp.real
    for i in range(len(f0)):
        cpxexp, num_samples = cexpt(f0[i],T[i],fs)
        temp = cpxexp.real
        audio_sequence = np.concatenate((audio_sequence, temp))
```

```
        time += T[i]
    t = np.linspace(0, time, int(time*fs))
    return audio_sequence,t

x,t = q_33([659.25,622.25,659.25,622.25,659.25,493.88,587.33,523.25,444,0,261.63
     [0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,1.0,0.5,0.5,0.5,0.5,1,0.5,0.5,0.5,0.5,1,0.
     ,44100)

#comput DFT and show it
X, f = compute_fft(x,fs)

# TODO: use plot_dft to plot the DFT
f0 = key_to_frequency(key)
def plot_it_pls(X,f,f0):
    plt.figure(figsize=(10, 6))
    plt.plot(f, abs(X))  # Plot magnitude of X against f
    plt.xlim((-2*f0, 2*f0))
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Magnitude |X(f)|')
    plt.title('Magnitude of DFT vs Frequency')
    plt.grid(True)
    plt.show()
plot_it_pls(X,f,f0)
```



In [66]:
```
# As we can see since it is a real function it is symmetric around frequency 0
# Also as we can see there is a spike at 0 since we had a lot of stops in the so
# However there is some spectrum leakage due to TF0 not equal to an integer
# Each frequency is a musical tone
```

3.3 Energy of Different Tones of a Musical Piece

We leave this part to you.

```
                                                    ltdict
tones_played = [659.25,622.25,659.25,622.25,659.25,493.88,587.33,523.25,444,0,26
```

```
Time_played  =  [0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,1.0,0.5,0.5,0.5,0.5,1,0.5,0.5,0
tone_duration = defaultdict(float)

for tone, duration in zip(Tones_played, Time_played):
    tone_duration[tone] += duration

def compute_energy_for_tone(frequency, X, f):
    idx = np.argmin(np.abs(f - frequency))
    energy = (np.abs(X[idx])**2)
    i = 1
    while f[idx+i] < frequency+5 and f[idx+i] > frequency-5:
        energy += (np.abs(X[idx+i])**2)
        energy += (np.abs(X[idx-i])**2)
        i+=1
    #multiply by two because it is negative and positive
    if tone!= 0:
        energy*=2
    return energy

total_energy = {}

for tone, duration in zip(Tones_played, Time_played):
    energy_per_tone = compute_energy_for_tone(tone, X, f)
    if tone in total_energy:
        continue
    else:
        total_energy[tone] = energy_per_tone

print("Total energy for each tone:")
for tone, energy in total_energy.items():
    print(f"Frequency {tone} Hz and time of the tone {tone_duration[tone]}s: Ene
```

```
Total energy for each tone:
Frequency 659.25 Hz and time of the tone 3.0s: Energy = 2.8831428497878413
Frequency 622.25 Hz and time of the tone 2.0s: Energy = 1.9218772698152116
Frequency 493.88 Hz and time of the tone 4.0s: Energy = 2.895712790535222
Frequency 587.33 Hz and time of the tone 1.0s: Energy = 0.968549820725596
Frequency 523.25 Hz and time of the tone 2.5s: Energy = 1.9357347933225375
Frequency 444 Hz and time of the tone 3.0s: Energy = 2.348866090480859
Frequency 0 Hz and time of the tone 2.5s: Energy = 5.755047450821449
Frequency 261.63 Hz and time of the tone 1.0s: Energy = 0.955946084733207
Frequency 329.63 Hz and time of the tone 2.5s: Energy = 2.397211260423204
Frequency 440 Hz and time of the tone 1.0s: Energy = 2.3799628557531434
Frequency 415.3 Hz and time of the tone 0.5s: Energy = 0.48347761827324814
```

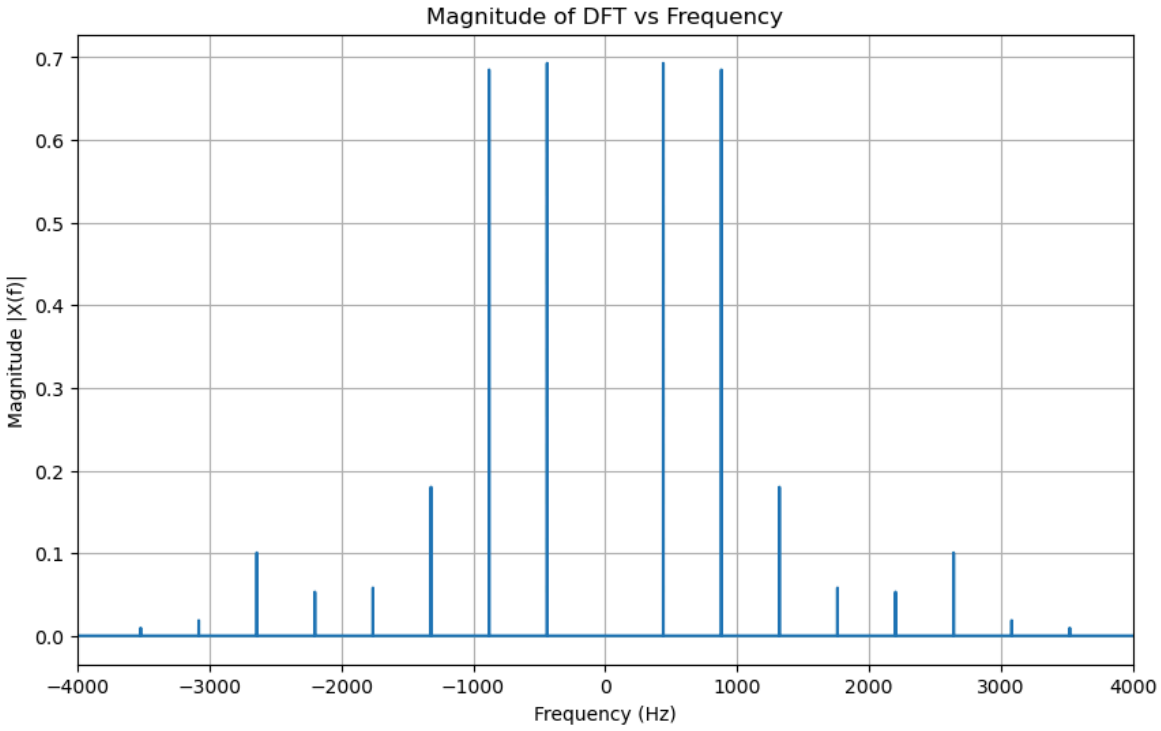
3.4 DFT of an A Note of Different Musical Instruments

We leave this part to you.

In [96]:
```
# First insturment
def oboe(f, T, fs):
    Ts = 1/fs
    N = math.ceil(T/Ts)
    k = (f/fs)*N
    x = np.zeros(N)
    alpha =  [1.386,1.370,0.360,0.116,0.106,0.201,0.037,0.019]
```

```
        y = alpha[i-1]*cexp(i*k,N)
        for i in range(len(x)):
            x[i]+= y[i]
    return x, N
def q_oboe(f0, T, fs):
    cpxexp, num_samples = oboe(f0, T, fs)
    Anote = cpxexp.real
    return Anote
x = q_oboe(440,2,44100)
X,f = compute_fft(x,fs)
plot_it_pls(X,f,2000)
get_energy_pls(X,f)
```

```
C:\Users\flash\AppData\Local\Temp\ipykernel_8988\3752261609.py:11: ComplexWarnin
g: Casting complex values to real discards the imaginary part
  x[i]+= y[i]
```

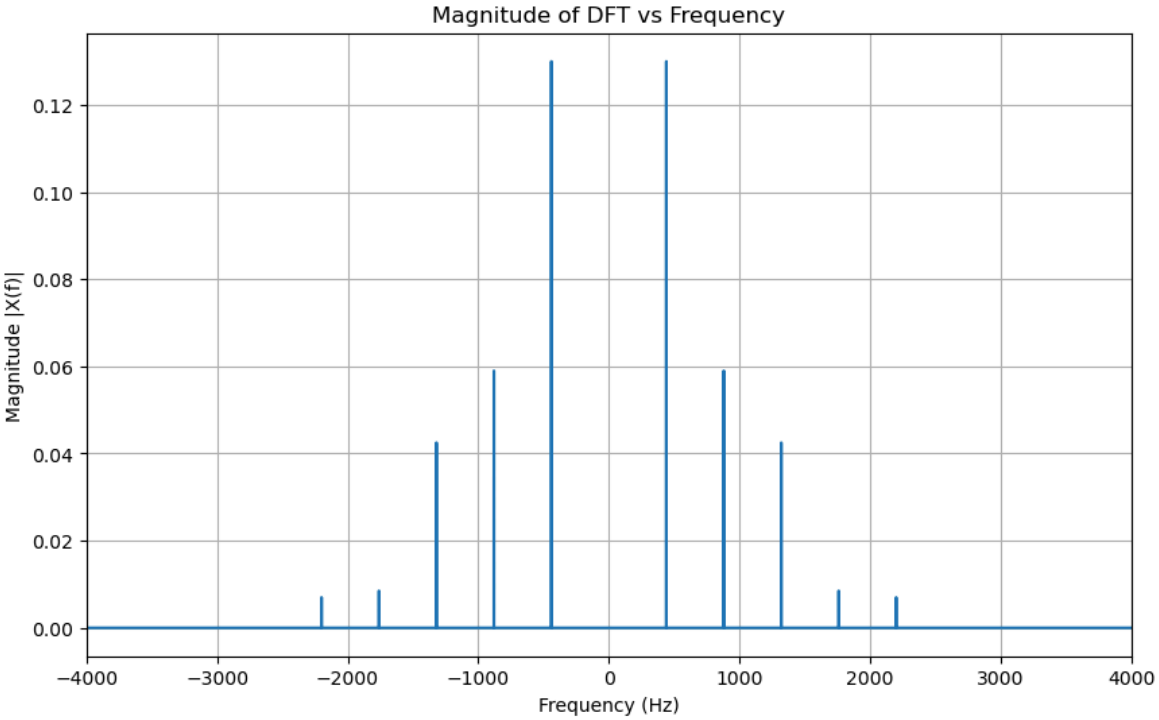



```
Frequency range containing 45% of the energy to the left: (-880.0, 0.0)
Frequency range containing 45% of the energy to the right: (0.0, 880.0)
Frequency range containing 90% of the energy: (-880.0, 880.0)
```

In [98]:
```
# second insturment
def flute(f, T, fs):
    Ts = 1/fs
    N = math.ceil(T/Ts)
    k = (f/fs)*N
    x = np.zeros(N)
    alpha =  [0.260,0.118,0.085,0.017,0.014]
    for i in range(1,6):
        y = alpha[i-1]*cexp(i*k,N)
        for i in range(len(x)):
            x[i]+= y[i]
    return x, N
def q_flute(f0, T, fs):
    cpxexp, num_samples = flute(f0, T, fs)
    Anote = cpxexp.real
    return Anote
x = q_flute(440,2,44100)
X,f = compute_fft(x,fs)
```

```
plot_it_pls(X,f,2000)
get_energy_pls(X,f)
```

```
C:\Users\flash\AppData\Local\Temp\ipykernel_8988\988452739.py:11: ComplexWarning:
Casting complex values to real discards the imaginary part
  x[i]+= y[i]
```

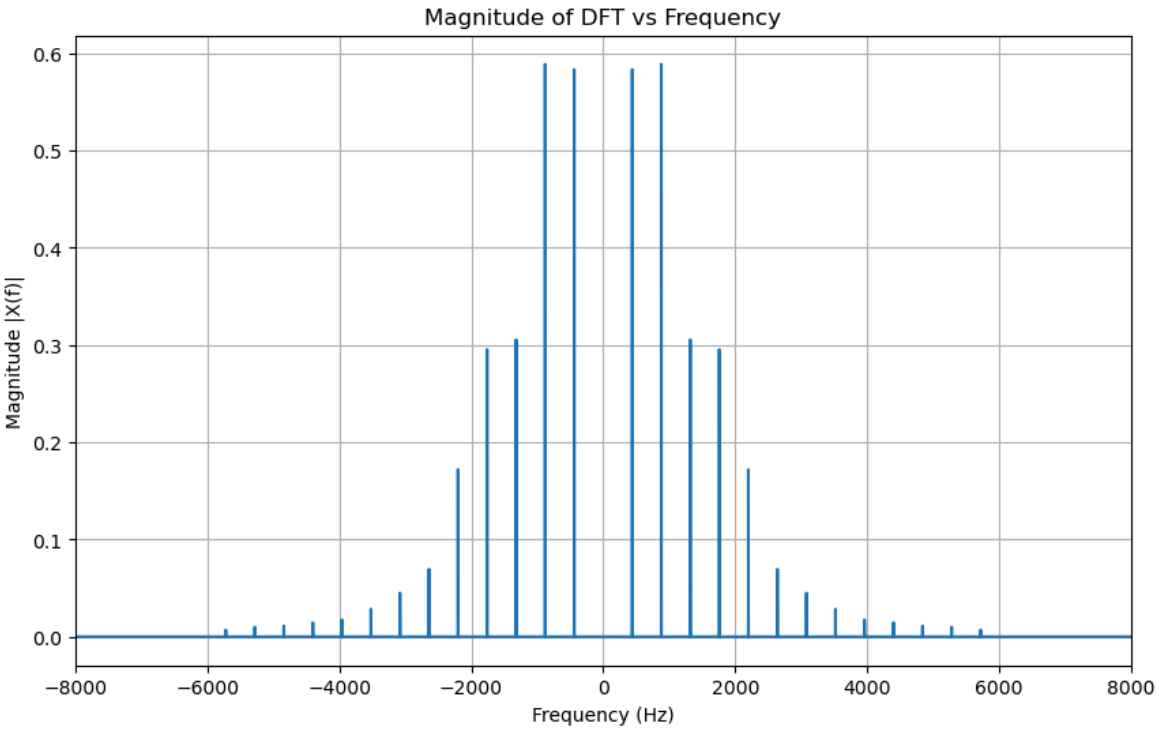



```
Frequency range containing 45% of the energy to the left: (-880.0, 0.0)
Frequency range containing 45% of the energy to the right: (0.0, 880.0)
Frequency range containing 90% of the energy: (-880.0, 880.0)
```

In [100…

```
# Third insturment
def quacky_trumpet(f, T, fs):
    Ts = 1/fs
    N = math.ceil(T/Ts)
    k = (f/fs)*N
    x = np.zeros(N)
    alpha =  [1.167,1.178,0.611,0.591,0.344,0.139, 0.090, 0.057,0.035,0.029,0.02
    for i in range(1,14):
        y = alpha[i-1]*cexp(i*k,N)
        for i in range(len(x)):
            x[i]+= y[i]
    return x, N
def q_quacky_trumpet(f0, T, fs):
    cpxexp, num_samples = quacky_trumpet(f0, T, fs)
    Anote = cpxexp.real
    return Anote
x = q_quacky_trumpet(440,2,44100)
X,f = compute_fft(x,fs)
plot_it_pls(X,f,4000)
get_energy_pls(X,f)
```

```
C:\Users\flash\AppData\Local\Temp\ipykernel_8988\732413434.py:11: ComplexWarning:
Casting complex values to real discards the imaginary part
  x[i]+= y[i]
```

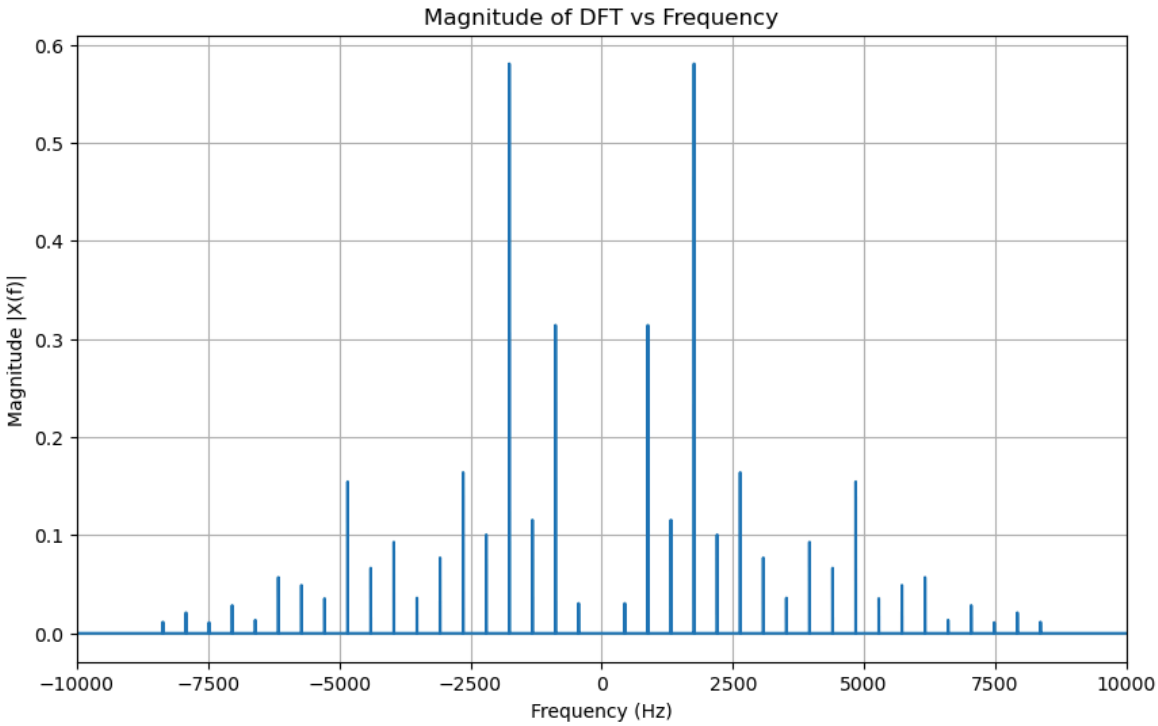



```
Frequency range containing 45% of the energy to the left: (-1760.0, 0.0)
Frequency range containing 45% of the energy to the right: (0.0, 1760.0)
Frequency range containing 90% of the energy: (-1760.0, 1760.0)
```

In [101…

```python
# Four insturment
def quacky_clarinet(f, T, fs):
    Ts = 1/fs
    N = math.ceil(T/Ts)
    k = (f/fs)*N
    x = np.zeros(N)
    alpha = [0.061,0.628,0.231,1.161,0.201,0.328,0.154,0.072,0.186,0.133,0.309,
    for i in range(1,20):
        y = alpha[i-1]*cexp(i*k,N)
        for i in range(len(x)):
            x[i]+= y[i]
    return x, N
def q_quacky_clarinet(f0, T, fs):
    cpxexp, num_samples = quacky_clarinet(f0, T, fs)
    Anote = cpxexp.real
    return Anote
x = q_quacky_clarinet(440,2,44100)
X,f = compute_fft(x,fs)
plot_it_pls(X,f,5000)
get_energy_pls(X,f)
```

```
C:\Users\flash\AppData\Local\Temp\ipykernel_8988\3931593035.py:11: ComplexWarnin
g: Casting complex values to real discards the imaginary part
  x[i]+= y[i]
```




```
Frequency range containing 45% of the energy to the left: (-2640.0, 0.0)
Frequency range containing 45% of the energy to the right: (0.0, 2640.0)
Frequency range containing 90% of the energy: (-2640.0, 2640.0)
```

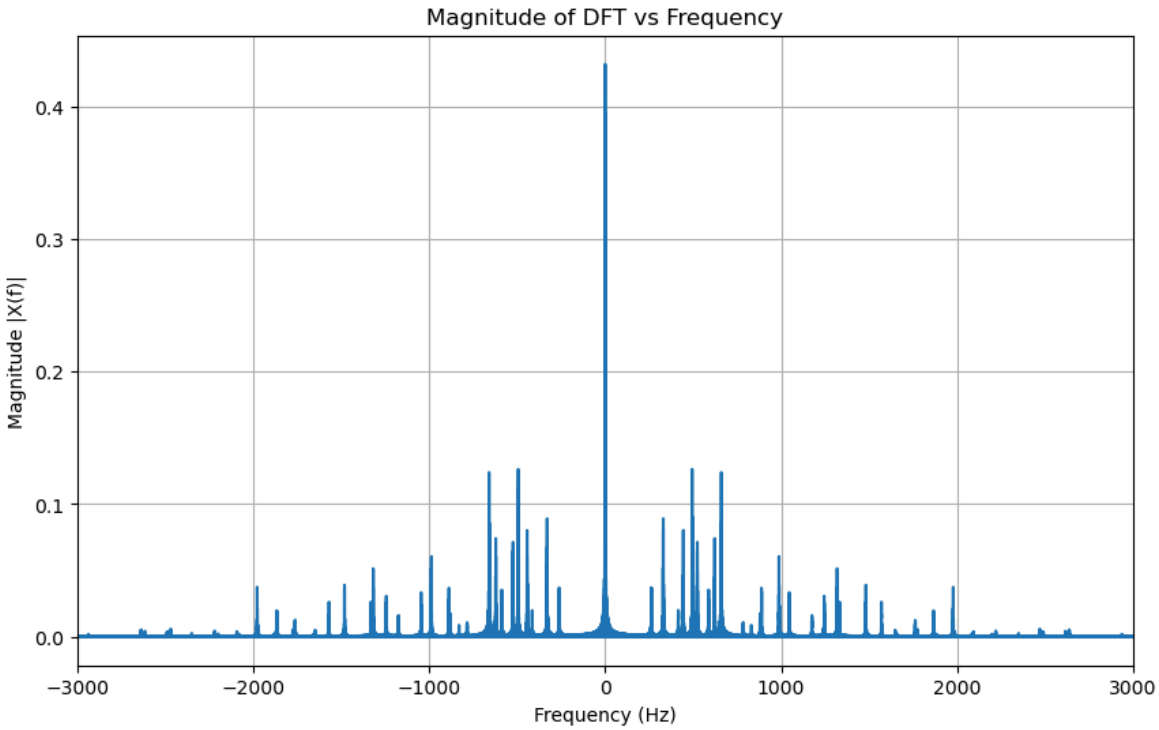
3.5 DFT of Your Song on a Musical Instrument

We leave this part to you.

In [105…

```
from scipy.io.wavfile import write
# I will choose the flute
def q_33(f0, T, fs):
    time = 0.5
    cpxexp,num_samples = flute(0,0.5,fs)
    audio_sequence = cpxexp.real
    for i in range(len(f0)):
        cpxexp, num_samples = flute(f0[i],T[i],fs)
        temp = cpxexp.real
        audio_sequence = np.concatenate((audio_sequence, temp))
        time += T[i]
    t = np.linspace(0, time, int(time*fs))
    # I want the sound to be high so I will increase and then decrease it again
    audio_sequence*=5
    write("Anote.wav", fs, audio_sequence.astype(np.float32))
    audio_sequence/=5
    return audio_sequence,t
x,t = q_33([659.25,622.25,659.25,622.25,659.25,493.88,587.33,523.25,444,0,261.63
     [0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,1.0,0.5,0.5,0.5,0.5,1,0.5,0.5,0.5,0.5,1,0.
     ,44100)
#comput DFT and show it
X, f = compute_fft(x,fs)
plot_it_pls(X,f,1500)
```

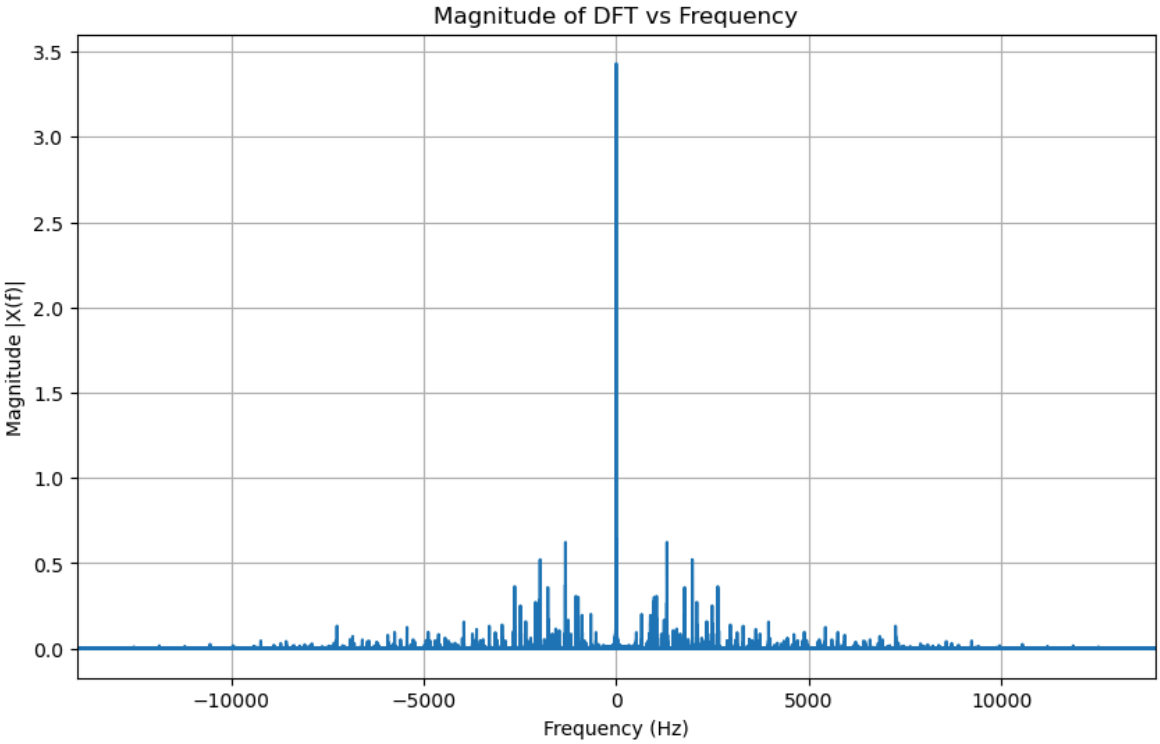
```
C:\Users\flash\AppData\Local\Temp\ipykernel_8988\988452739.py:11: ComplexWarning:
Casting complex values to real discards the imaginary part
  x[i]+= y[i]
```



In [106…
```
# As we can see the major musical noteshas the highes magnitude and the complime
# complimentery ones extending after 1000hz and after playing the music it sound
# Since I made general functions I will try another insturment Because I want to
```

In [107…
```
def q_33(f0, T, fs):
    time = 0.5
    cpxexp,num_samples = quacky_clarinet(0,0.5,fs)
    audio_sequence = cpxexp.real
    for i in range(len(f0)):
        cpxexp, num_samples = quacky_clarinet(f0[i],T[i],fs)
        temp = cpxexp.real
        audio_sequence = np.concatenate((audio_sequence, temp))
        time += T[i]
    t = np.linspace(0, time, int(time*fs))
    # I want the sound to be high so I will increase and then decrease it again
    audio_sequence*=5
    write("Anote.wav", fs, audio_sequence.astype(np.float32))
    audio_sequence/=5
    return audio_sequence,t
x,t = q_33([659.25,622.25,659.25,622.25,659.25,493.88,587.33,523.25,444,0,261.63
     [0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,1.0,0.5,0.5,0.5,0.5,1,0.5,0.5,0.5,0.5,1,0.
     ,44100)
#comput DFT and show it
X, f = compute_fft(x,fs)
plot_it_pls(X,f,7000)
```

```
C:\Users\flash\AppData\Local\Temp\ipykernel_8988\3931593035.py:11: ComplexWarnin
g: Casting complex values to real discards the imaginary part
  x[i]+= y[i]
```



(2.1) $\boxed{X(-k)} = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x(n)\, e^{-j2\pi(-k)n/N} = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x^*(n) \left(e^{-j2\pi kn/N} \right)^* =$

$$= \left[\frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x(n)\, e^{-j2\pi kn/N} \right]^* = \boxed{X^*(k)}$$

(2.2) $\boxed{\|X\|^2} = \sum_{k=0}^{N-1} X(k) X^*(k) = \sum_{k=0}^{N-1} \left[\frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x(n)\, e^{-j2\pi kn/N} \right] \left[\frac{1}{\sqrt{N}} \sum_{\tilde{n}=0}^{N-1} x^*(\tilde{n})\, e^{j2\pi k\tilde{n}/N} \right]$

$X(k)$ $\qquad$ $X^*(k)$

$$\left(X(k) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x(n)\, e^{-j2\pi kn/N} \right)$$

$$= \sum_{n=0}^{N-1} \sum_{\tilde{n}=0}^{N-1} x(n)\, x^*(\tilde{n}) \left[\sum_{k=0}^{N-1} \frac{1}{\sqrt{N}} e^{-j2\pi kn/N} \frac{1}{\sqrt{N}} e^{j2\pi k\tilde{n}/N} \right] =$$

$$\langle e_{n,N}, e_{\tilde{n},N} \rangle = \delta(\tilde{n} - n)$$

$$= \sum_{n=0}^{N-1} \sum_{\tilde{n}=0}^{N-1} x(n)\, x^*(\tilde{n})\, \delta(\tilde{n} - n) = \sum_{n=0}^{N-1} x(n) x^*(n) = \boxed{\|x\|^2} = \boxed{\sum_{k=N_0}^{N_0+N-1} [X(k)]^2}$$

(2.3) $z = \boxed{\mathcal{F}(ax + by)}$

$$z(k) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} \left[a x(n) + b y(n) \right] e^{-j2\pi kn/N} =$$

$$= \frac{a}{\sqrt{N}} \sum_{n=0}^{N-1} x(n)\, e^{-j2\pi kn/N} + \frac{b}{\sqrt{N}} \sum_{n=0}^{N-1} y(n)\, e^{-j2\pi kn/N}$$

$X = \mathcal{F}(x)$ $\qquad$ $Y = \mathcal{F}(y)$

$$z(k) = aX(k) + bY(k) = \boxed{a\mathcal{F}(x) + b\mathcal{F}(y)}$$

(2.4) Same as 2.2 but with $X(k)\,Y^*(k)$: ($X(k)$ ↓, $Y^*(k)$ ↓)

$$\boxed{\langle x, y\rangle = \sum_{n=0}^{N-1} x(n)\,y^*(n)} = \sum_{k=0}^{N-1}\left[\frac{1}{\sqrt{N}}\sum_{n=0}^{N-1} x(n)\,e^{-j2\pi kn/N}\right]\left[\frac{1}{\sqrt{N}}\sum_{\tilde{n}=0}^{N-1} y^*(\tilde{n})\,e^{j2\pi k\tilde{n}/N}\right] =$$

$$= \sum_{n=0}^{N-1}\sum_{\tilde{n}=0}^{N-1} x(n)\,y^*(\tilde{n})\left[\sum_{k=0}^{N-1}\frac{1}{\sqrt{N}}e^{-j2\pi kn/N}\,\frac{1}{\sqrt{N}}e^{j2\pi k\tilde{n}/N}\right] =$$

$$= \sum_{n=0}^{N-1}\sum_{\tilde{n}=0}^{N-1} x(n)\,y^*(\tilde{n})\,\delta(\tilde{n}-n) = \sum_{n=0}^{N-1} x(n)\,y^*(n) = \boxed{\langle x, y\rangle} =$$

$$= \boxed{\sum_{k=N_0}^{N_0+N-1} X(k)\,Y^*(k)}$$