Laboratory Report of Digital Signal Processing

Name: Junjie Fang

Student ID: 521260910018

Date: 2024/3/4

Score:

Contents

1. Signal operations

Given the parameters $A = 3, B = 4, D = 8$, the three gate functions are defined by:

$$g_0(t) := \begin{cases} 4 \text{ if } 0 \le t \le 3 \\ 0 \text{ otherwise} \end{cases}$$

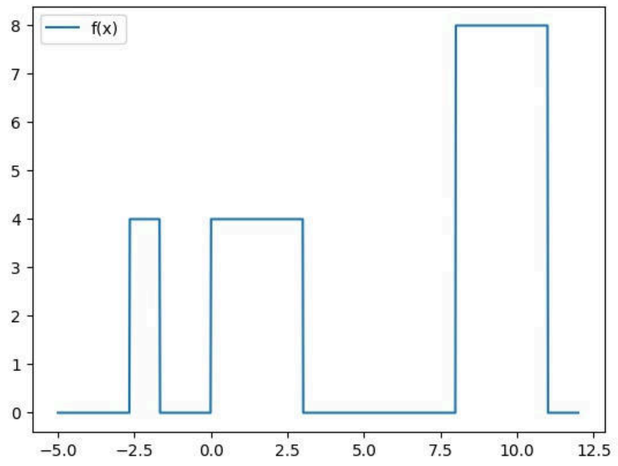
$$g_1(t) := \begin{cases} 4 \text{ if} -\frac{3}{8} \le t \le -\frac{3}{5} \\ 0 \text{ otherwise} \end{cases}$$

$$g_2(t) := \begin{cases} 8 \text{ if } 8 \le t \le 11 \\ 0 \text{ otherwise} \end{cases}$$

And we know:

$$x(t) := \sum_{i=0}^{2} g_i(t)$$

We plot the x function in the figure below:



It can be seen that the images of the three gate functions do not overlap.

In practice, we use python's `matplotlib` to draw function images. For scalability, we use the `gate_func()`, `func_transform()` and `add_func()` to generate, transform and add functions. See Section 7.a for the code.

2. Aliasing phenomenon in sampling process

Let the frequencies corresponding to the two peaks in the image be f_{a1}, f_{a2} and the function values be X_1, X_2. The sampling frequency is $f_s = 100\text{Hz}$. According to the sampling theorem, we have:

$$f_{a1} = \pm f_1 - k_1 f_s$$
$$f_{a2} = \pm f_2 - k_2 f_s$$

where:

$$k_1, k_2 \neq 0$$
$$800\text{Hz} \leq f_1, f_2 \leq 850\text{Hz}$$

Plug the data $f_{a1} = 14, f_{a2} = 3$ into the equation and we can determine that the only solution is:

$$k_1 = 8, f_1 = 814\text{Hz}$$
$$k_2 = 8, f_2 = 803\text{Hz}$$

Next, we can determine the amplitudes A_1 and A_2 by reviewing some of the properties of *Contiunous-Time Fourier Transform (CTFT)*. The Fourier Transform used in this question is in the form:

$$X(jf) = \int_{-\infty}^{+\infty} x(t) e^{-j2\pi f t} \, \mathrm{d}t$$
$$x(t) = \int_{-\infty}^{+\infty} X(jf) e^{j2\pi f t} \, \mathrm{d}f$$

In this form, the cosine wave with amplitude 1 and the following sum of two impulse function form a Fourier Transform pair:

$$\cos(2\pi f_0 t) \overset{\text{F.T.}}{\iff} \frac{1}{2}(\delta(f - f_0) + \delta(f + f_0))$$

Due to the linearity of Fourier Transform, we know that the amplitudes should be twice the height of peaks in the frequency domain. Therefore, we have:

$$A_1 = 2X_1 = 4$$
$$A_2 = 2X_2 = 2$$

Parameters	$i = 1$	$i = 2$
f_i	814Hz	803Hz
A_i	4	2

3. Continuous-Time Fourier Transform properties

3.a. Creation of Continuous-Time Fourier Transform (*CTFT*) function

The definition of *CTFT* is:

$$X_\omega = \int_{-\infty}^{+\infty} x(t) \cdot e^{-j\omega t} \,\mathrm{d}t$$

To integrate arbitrary functions in code, we use discrete sampling summation for approximate integration. The code is in Appendix 3.a. In this code, the `CTFT(x, t, w)` function take `x` and `t` as lists of sampled data in time domain, which should be calculated outside the function. For each element in `w`, which represents a frequency, the function calculates *CTFT* at this frequency, and finally return a list of complex numbers. The code is in Section 7.b.1.

In order to improve the approximation accuracy, we can increase the number of samples (i.e. `SAMPLE_N` parameter).

3.b. Comparison of g and shifted g

We can get $g_2 = g\left(t - \frac{D}{2}\right)$ by applying time shifting on g. To get g_2 in the code, we use a function called `func_tranform` to get the shifted function of g. The figure showing both functions is:

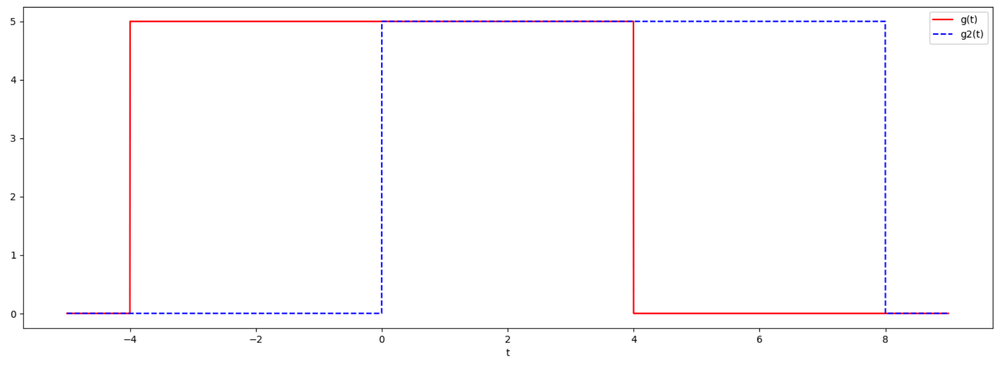


Figure 2: g and g_2 in the same plot

3.c. Plot of *CTFT* of g and g_2

Using the function `CTFT()` function realized in *1.a*, we can calculate the *CTFT* of g_2 and g respectively.

By observing the images of module, phase, real part and imaginary part of the two functions, **we can verify the followling properties of time shifting under *CTFT*:**

1. The module remains unchanged.
2. THe phase changes linearly with ω, and the distribution of real and imaginary parts changes.

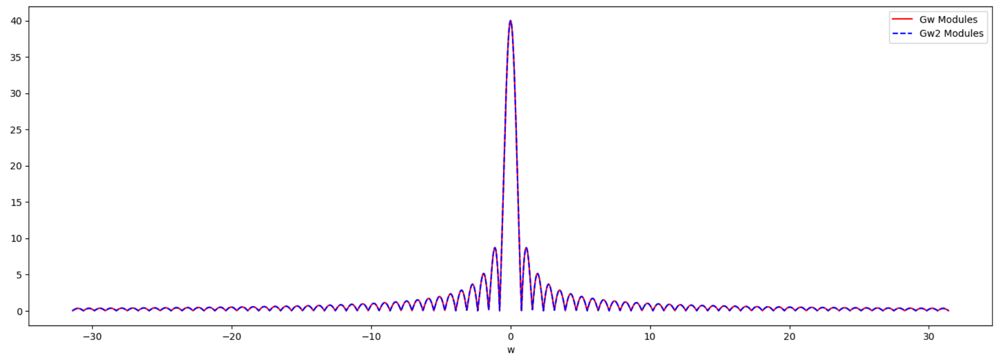


Figure 3: Module of g and g_2

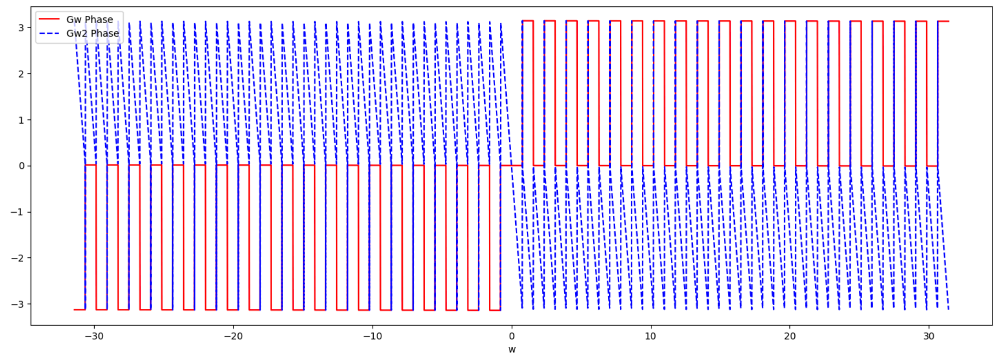


Figure 4: Phase of g and g_2

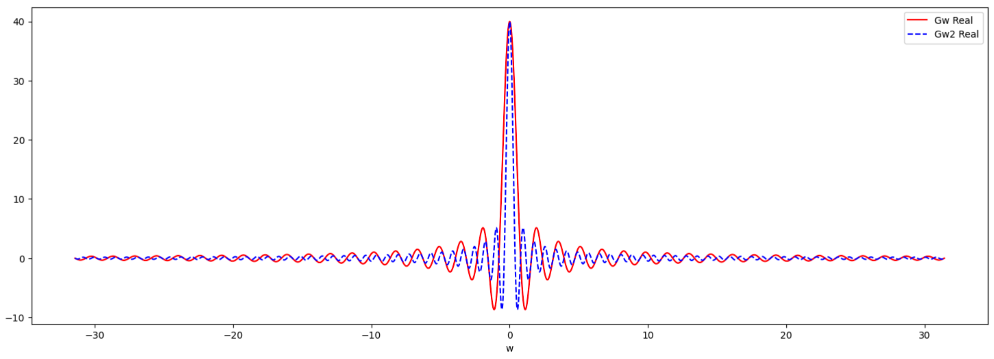


Figure 5: Real part of g and g_2

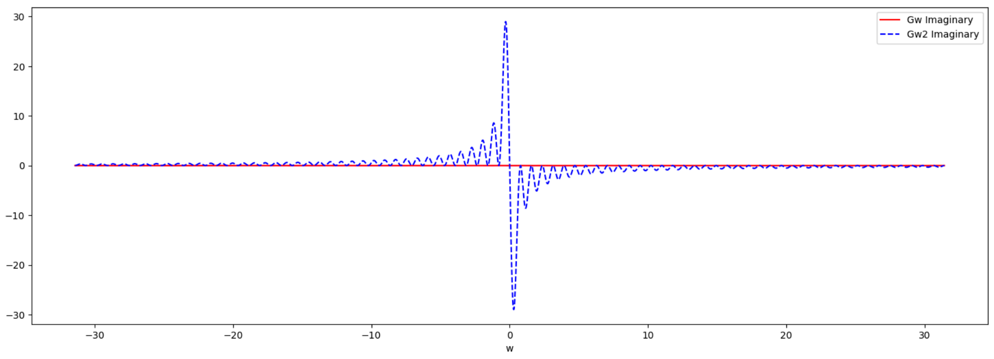


Figure 6: Imaginary part of g and g_2

3.d. Modulation

In the code, we can generate $y(t) = g(t) \times \cos(4\pi t)$ from $g(t)$. The figure below shows the comparison of $g(t)$ and $y(t)$ over $t = [-15, 15]$:

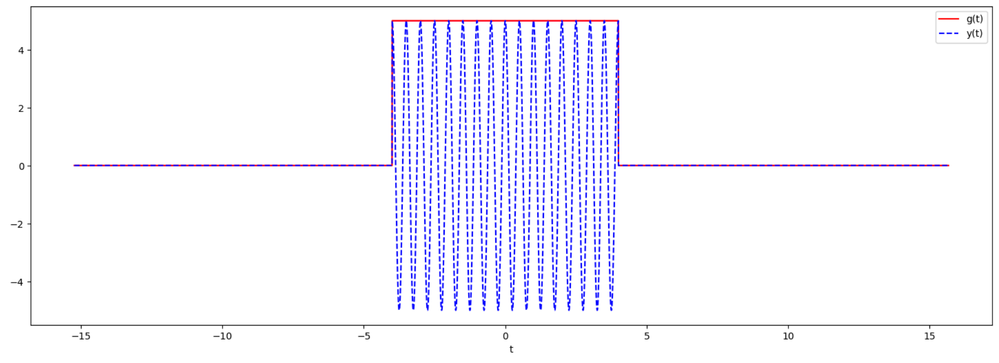


Figure 7: y and g in the same plot

3.e. Modulation properties of Fourier Tranform

To get the module and phase of $y(t)$ and $g(t)$, we can calculte their *CTFT* like in Section 3.c.

The modulation property of *CTFT* gives:

$$G_{T_1}(t)\cos(\omega_0 t) \overset{F.T.}{\Longleftrightarrow} \frac{1}{2}X[j(\omega-\omega_0)] + \frac{1}{2}X[j(\omega+\omega_0)]$$

This property can be verified from the figures below, as the module and phase of g is shifted to $\omega_0 = 4\pi$ and $-\omega_0 = -4\pi$ in the frequency domain. Note the peak value of module of y is half this value of g.

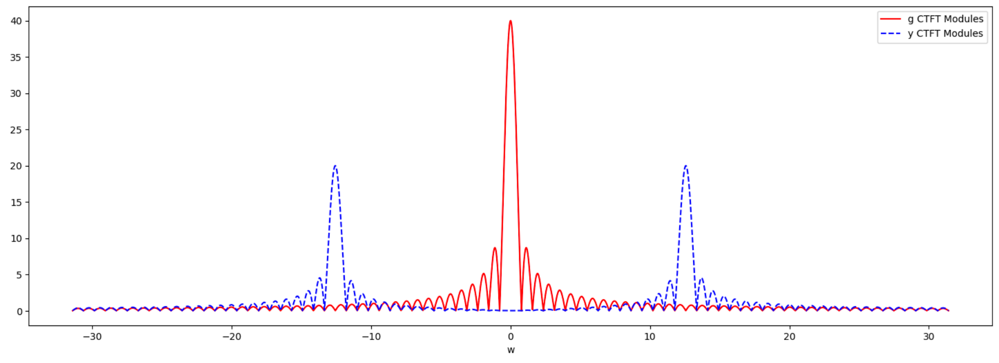


Figure 8: Module of y and g

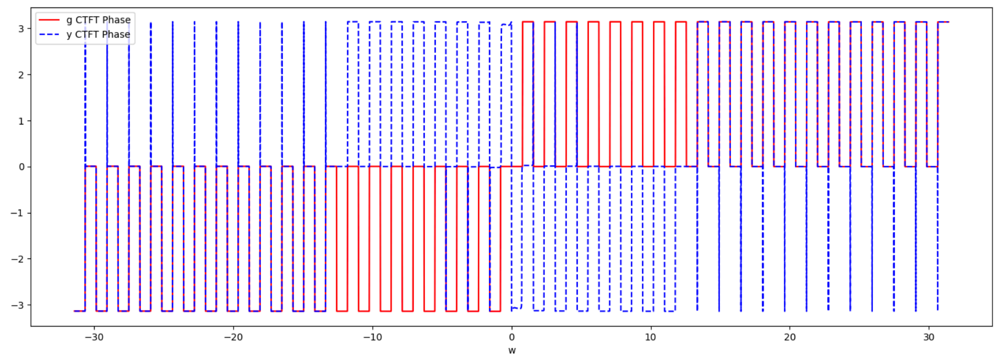


Figure 9: Phase of y and g

3.f. Verification of Parseval's formula

In the code we can get the energy in both time and frequency domain, which are 99.95 and 99.38. There is a slight difference, and we can consider the two energies to be the same. The difference comes from the error in the integral calculation and is very small (the error can be reduced by increasing the number of integral samples).

The reason behind that is Parseval's formula, which gives:

$$\int_{-\infty}^{+\infty} |x(t)|^2 \,\mathrm{d}t = \frac{1}{2\pi} \int_{-\infty}^{+\infty} |X(j\omega)|^2 \,\mathrm{d}\omega$$

This denotes that the energy in time domain is equal to the energy in frequency domain. The proof of this formula is as follows:

$$\begin{aligned}
&\int_{-\infty}^{+\infty} x^2(t) \,\mathrm{d}t \\
&= \int_{-\infty}^{+\infty} x(t) \left(\frac{1}{2\pi} \int_{-\infty}^{+\infty} X(j\omega) e^{j\omega t} \,\mathrm{d}\omega \right) \mathrm{d}t \\
&= \frac{1}{2\pi} \int_{-\infty}^{+\infty} X(j\omega) \left(\int_{-\infty}^{+\infty} x(t) e^{j\omega t} \,\mathrm{d}t \right) \mathrm{d}\omega \\
&= \frac{1}{2\pi} \int_{-\infty}^{+\infty} X(j\omega) X(-j\omega) \,\mathrm{d}t \\
&= \frac{1}{2\pi} \int_{-\infty}^{+\infty} |X(j\omega)|^2 \,\mathrm{d}t
\end{aligned}$$

4. Discrete-Time Fourier Transform properties

4.a. Creation of the Discrete-Time Fourier Transform (*DTFT*) function

The code in appendix implements `DTFT(nT, xn, w)` function, using the following formula:

$$X(\omega) = \sum_{n=-\infty}^{+\infty} x(nT) e^{-j\omega nT}$$

To avoid infinite calculation, we can set a start time and end time for sampling function, as long as it covers the whole signal. The code is in Section 7.c.1.

4.b. Plots of *DTFT* when $T = \frac{D}{80}$ and $T = \frac{D}{40}$

Using the function implemented in *3.a*, we rendered the images of $G_{w,1}$ and $G_{w,2}$ in a Nyquist interval,

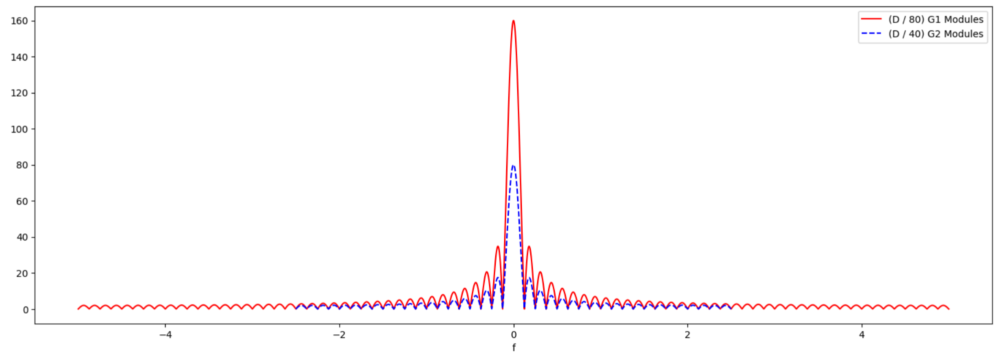


Figure 10: Module of $G_{w,1}$ and $G_{w,2}$ in f

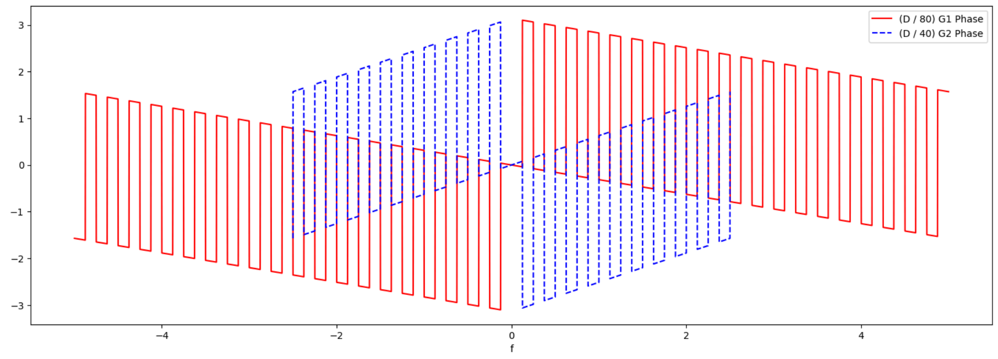


Figure 11: Phase of $G_{w,1}$ and $G_{w,2}$ in f

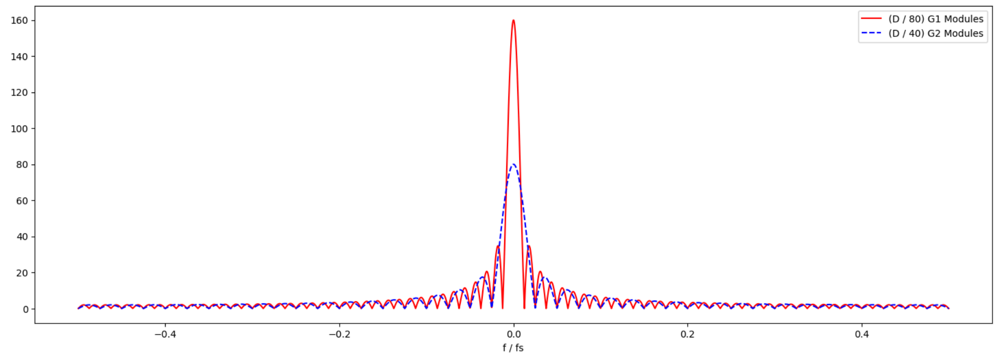


Figure 12: Module of $G_{w,1}$ and $G_{w,2}$ in $\frac{f}{f_s}$

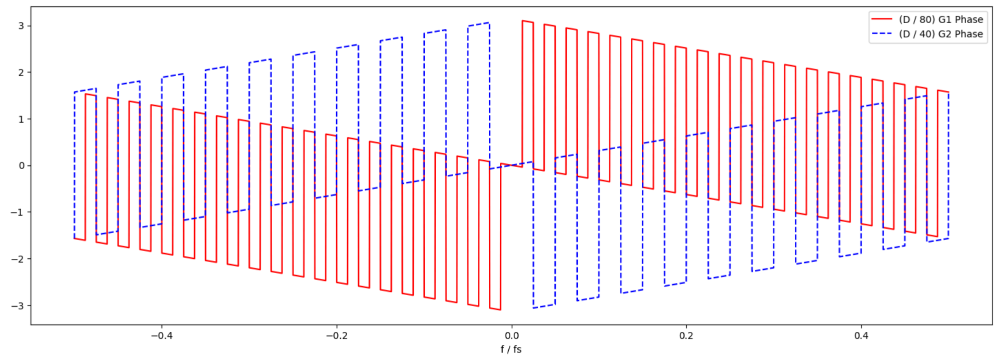


Figure 13: Phase of $G_{w,1}$ and $G_{w,2}$ in $\frac{f}{f_s}$

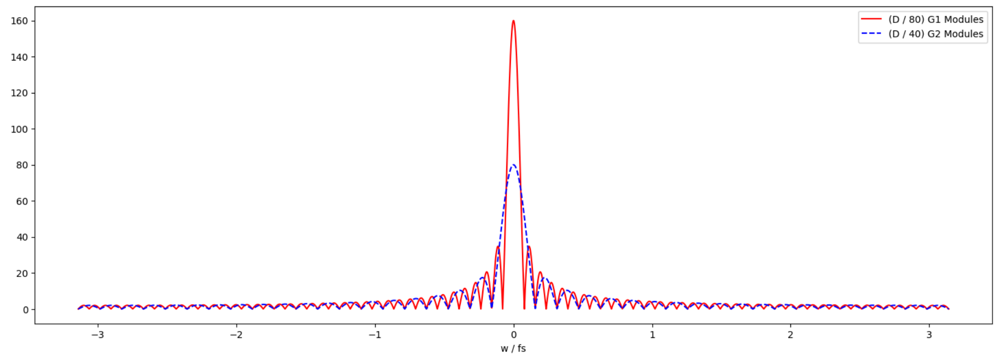


Figure 14: Module of $G_{w,1}$ and $G_{w,2}$ in $\frac{w}{f_s}$

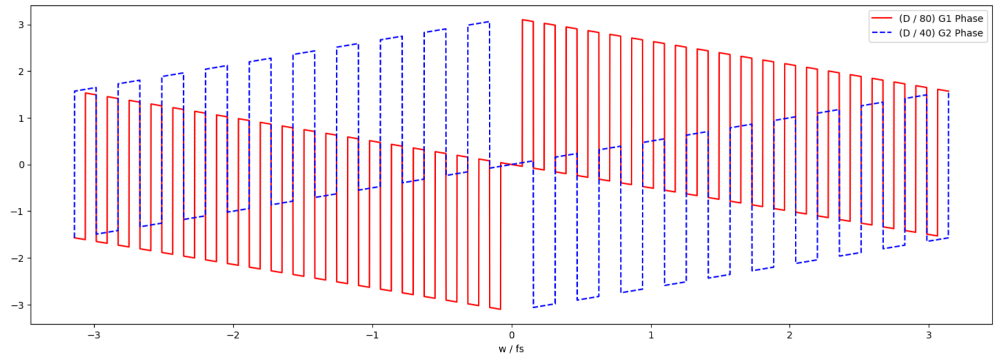


Figure 15: Phase of $G_{w,1}$ and $G_{w,2}$ in $\frac{w}{f_s}$

4.c. Deduciton of the theoretical *CTFT* function of g

The theoretical *CTFT* function of g is:

$$\begin{aligned} X(w) &= \int_{-\infty}^{+\infty} g(t)e^{-j\omega t}\,\mathrm{d}t \\ &= \int_{-4}^{4} 2e^{-j\omega t}\,\mathrm{d}t \\ &= \frac{2}{-j\omega}(e^{-4j\omega} - e^{4j\omega}) \\ &= 16\,\mathrm{sinc}(4\omega) \end{aligned}$$

We can plot them in the same figure:

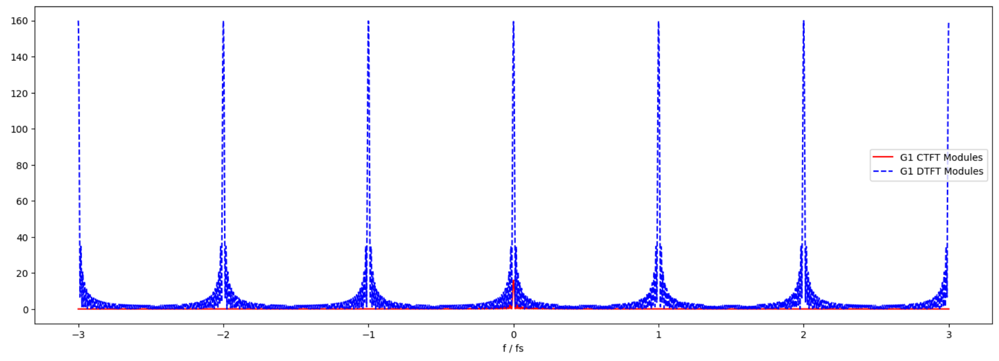


Figure 16: Module of $G_{w,1}$ and *CTFT* of g

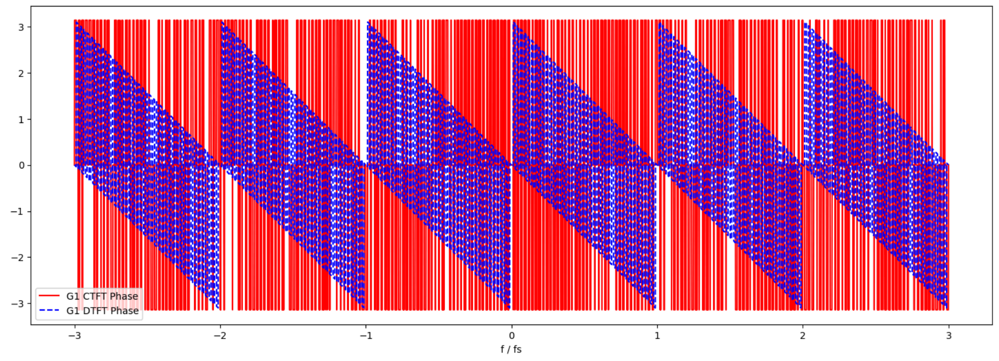


Figure 17: Phase of $G_{w,1}$ and *CTFT* of g

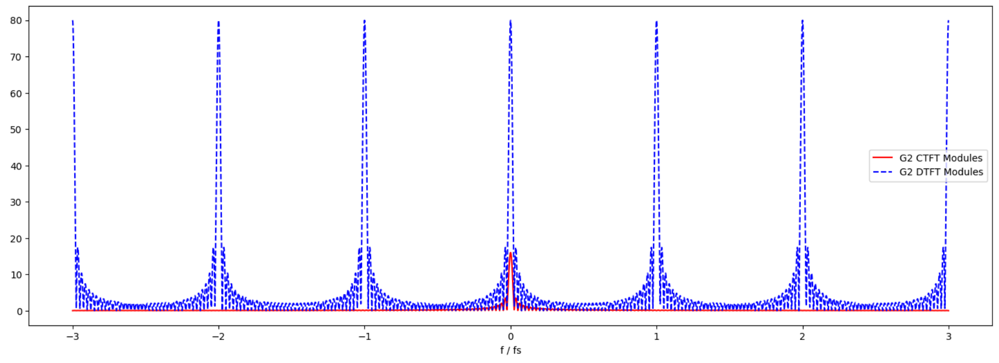


Figure 18: Module of $G_{w,2}$ and *CTFT* of g

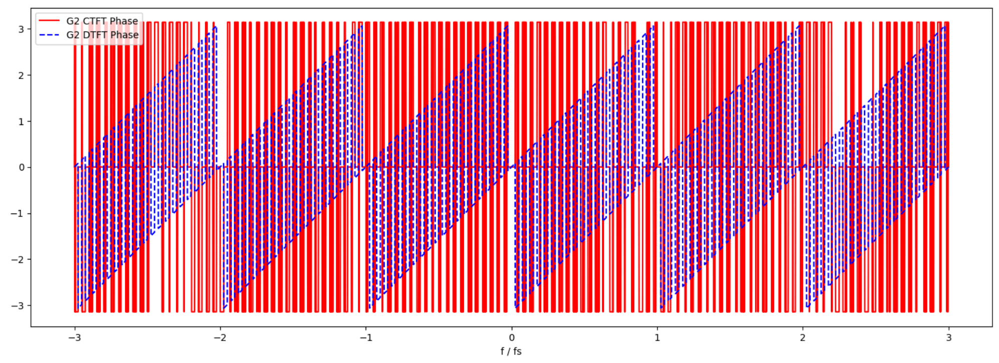


Figure 19: Phase of $G_{w,2}$ and *CTFT* of g

For $G_{w,1}$, the peak value at $\omega = 0$ is ten times the *CTFT* of g. That's because the sampling frequency is $f_s = 10$. And for $G_{w,2}$ it is five times, as the sampling frequency is $f_s = 5$.

4.d. Inverse *DTFT*

We can inverse *DTFT* using the formula:

$$x[nT] = \frac{1}{w_s} \int_{-\frac{w_s}{2}}^{+\frac{w_s}{2}} X[e^{j\omega}] e^{j\omega n} \, \mathrm{d}\omega$$

We get:

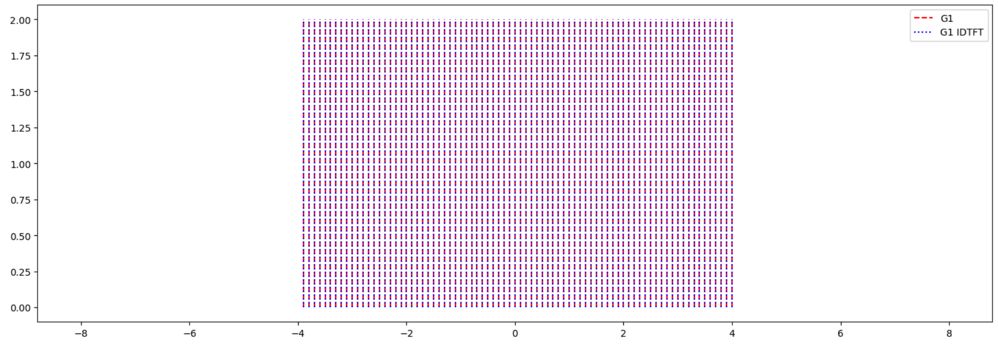


Figure 20: Figure of the discrete g_1 and the inverse of $G_{w,1}$

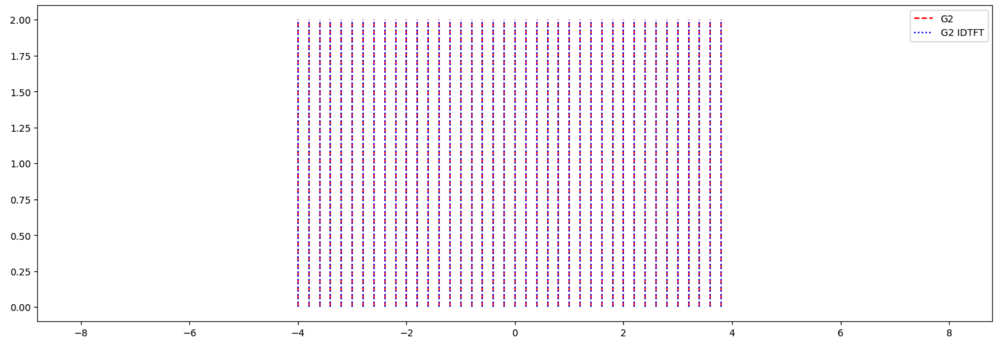


Figure 21: Figure of the discrete g_2 and the inverse of $G_{w,2}$

This two images shows that the inverse *DTFT* perfectly matches the discret sampling function.

4.e. Adjusted Parseval's formula

The former Parseval's formula is no longer validated for *DTFT*. If we calculate the energy of the the original function and the *DTFT* function (in one Nyquist interval to avoid infinite energy), we can get 31.99 and 3199.99, the latter one is 100 times the former one. That is due to the sampling frequency of the discrete function.

We can adjust this result by adding a factor of $\left(\frac{1}{f_s}\right)^2$ in the formula of *DTFT* function, which means the Parseval's formula would be:

$$\int_{-\infty}^{+\infty} |x(t)|^2 \, \mathrm{d}t = \frac{1}{2\pi f_s^2} \int_{-\frac{w_s}{2}}^{+\frac{w_s}{2}} |X(j\omega)|^2 \, \mathrm{d}\omega$$

.

5. Windowing effects of DTFT

5.a. *DTFT* of g with gate sampling function

We can adopt $\frac{2}{N}$ as factor to scale magnitudes of *DTFT* function. The figure is:

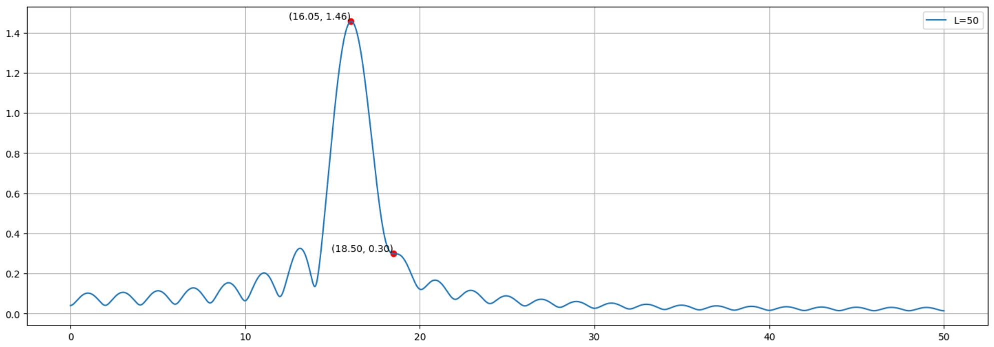


Figure 22: Figure and peak values when $L = 50$

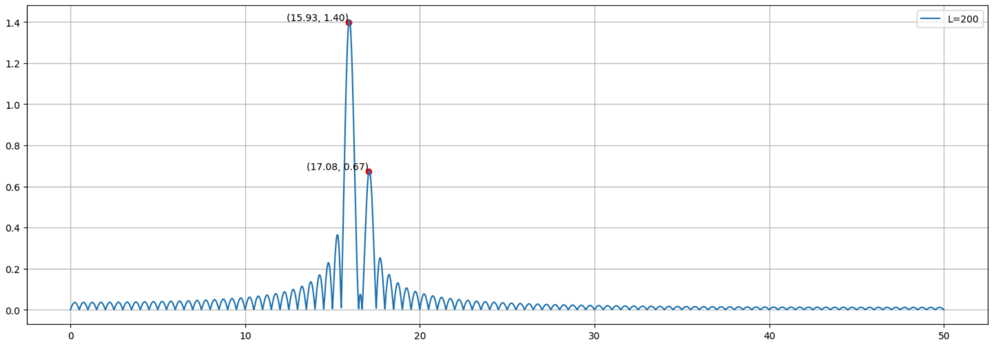


Figure 23: Figure and peak values when $L = 200$

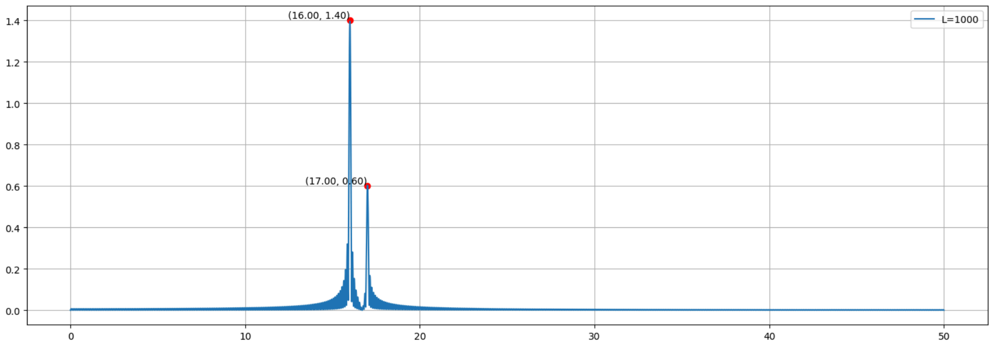


Figure 24: Figure and peak values when $L = 1000$

Larger the L, the more accurate we can find the right amplitude and frequency.

L	factor	A_1	A_2	f_1	f_2
50	$\frac{2}{50}$	1.46	0.30	16.05	18.50
200	$\frac{2}{200}$	1.40	0.67	15.93	17.08
1000	$\frac{2}{1000}$	1.40	0.60	16.00	17.00

5.b. *DTFT* of g with Hamming function

Using Hamming function, the factor should be $\frac{2}{Na_0}$, becuase the area of Hamming function is $\int_0^T a_0 - (1 - a_0)\cos\left(\frac{2\pi t}{T}\right) = a_0 T$, where $a_0 = 0.53836$. The figure is:

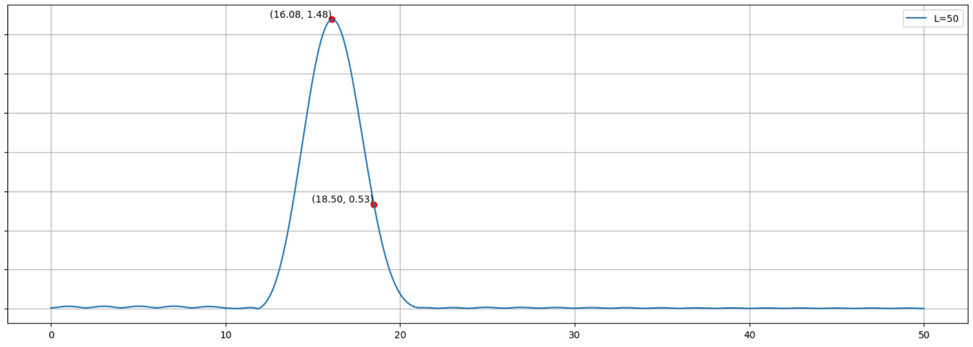


Figure 25: Figure and peak values when $L = 50$

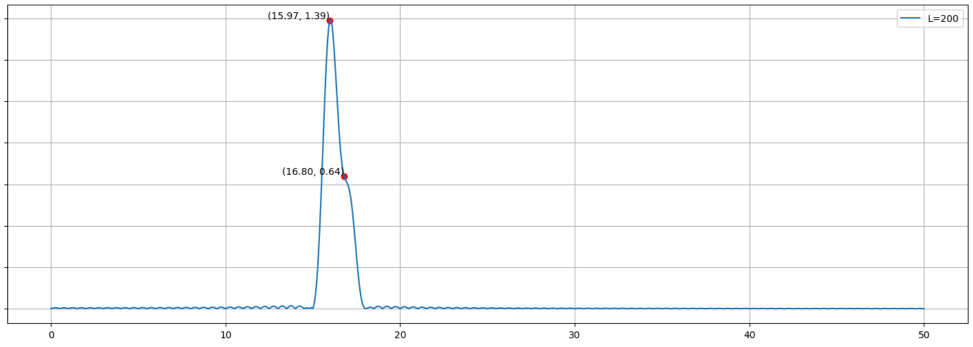


Figure 26: Figure and peak values when $L = 200$

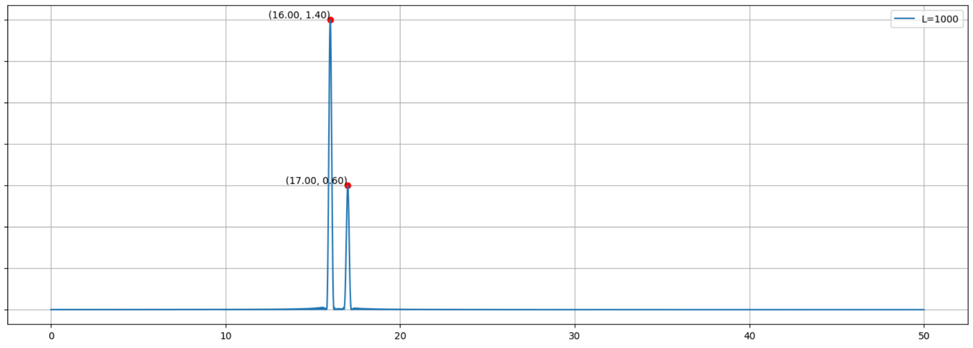


Figure 27: Figure and peak values when $L = 1000$

L	factor	A_1	A_2	f_1	f_2
50	$\frac{2}{50a_0}$	1.48	0.53	16.08	18.50
200	$\frac{2}{200a_0}$	1.39	0.64	15.97	16.80
1000	$\frac{2}{1000a_0}$	1.40	0.60	16.00	17.00

The sidelobes after applying Hamming function are much lower than the original ones, which means the frequency leakage is reduced. But the width of the main lobe is increased, leading to a reduction of frequency resolution.

6. DFT and FFT

6.a. Figure of the samples

The figure of y is:

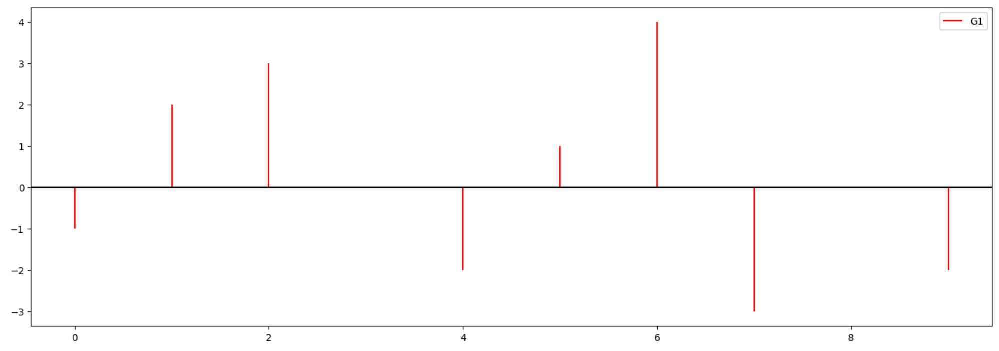


Figure 28: Figure of y

6.b. Module and phase of y's *DTFT*

We can use the `DTFT()` function defined in the previous questions. The modulus and phase of *DTFT* of y in a Nyquist interval are:

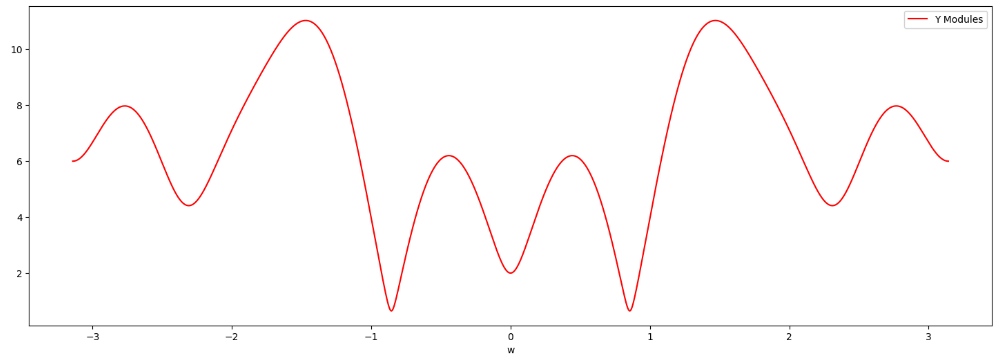


Figure 29: Module of *DTFT* y

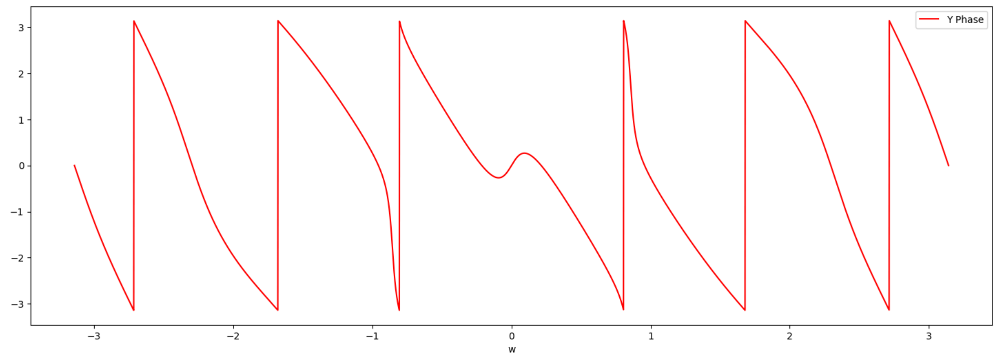


Figure 30: Phase of *DTFT* of y

The function is continuous in the frequency domain.

6.c. N-point *DFT* of y

The *DFT* algorithm discretizes *DTFT* samples in the frequency domain. The standard form is, for $k = 0, 1, ...N-1, w_k = \frac{2\pi k}{N}$,

$$X(\omega_k) = \sum_{n=0}^{N-1} x(n)e^{-j\omega_k n}$$

Using the new written `dft()` function, we can plot the two functions the same plot:

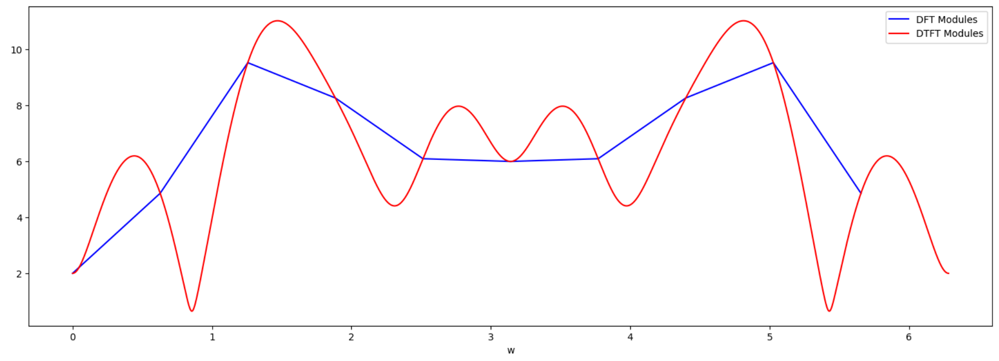


Figure 31: Module of y's *DFT* (blue) and *DTFT* (red)

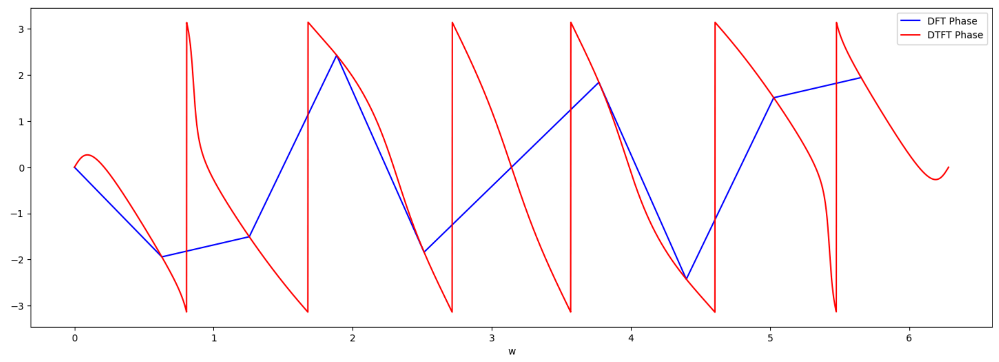


Figure 32: Phase of y's *DFT* (blue) and *DTFT* (red)

At the sampling points of *DFT*, the function values of the two remain consistant.

6.d. Inverse *DFT*

The formula of inverse *DTFT* is:

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{j \frac{2\pi k}{N} n}$$

The following figure shows that the inverse *DTFT* completely matches the original function:

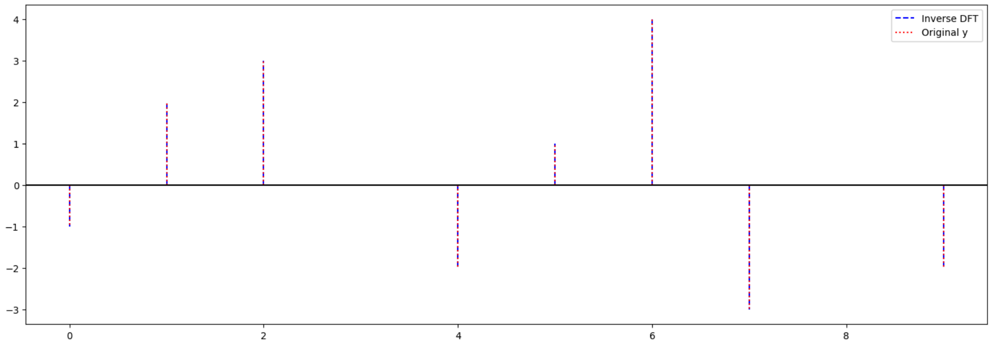


Figure 33: Original y and its inverse *DTFT*

6.e. Zero-padding

Using `numpy.pad()` function, we can apply zero-padding to $y[n]$. To get the *FFT* of y, we can use `numpy.fft.fft()` function. The modulus and phase of *FFT* of y are:

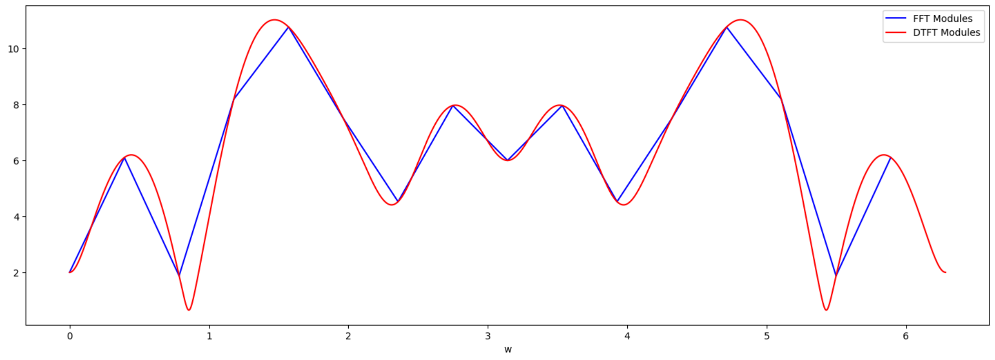


Figure 34: Module of *FFT* (N = 16) and *DTFT* of y

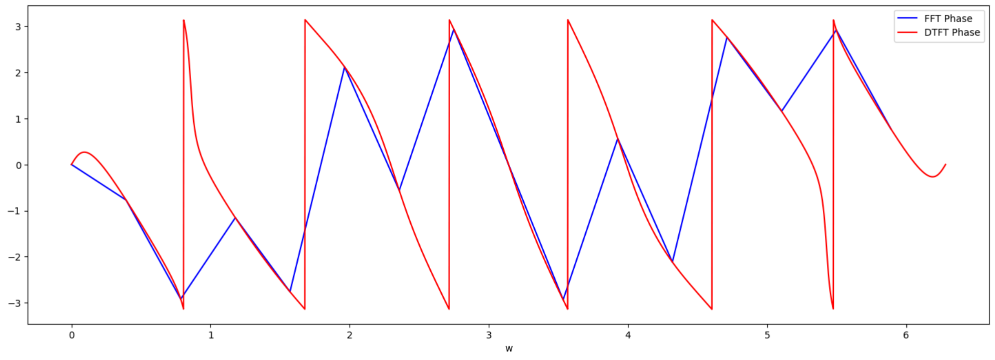


Figure 35: Phase of *FFT* (N = 16) and *DTFT* of y

It can be seen that the *FFT* of y is consistent with the *DTFT* of y on the sampling points.

6.f. Computational time of *DFT* and *FFT*

The time complexity of *DFT* for a sequence of length N is $O(N^2)$, while the time complexity of *FFT* is $O(N \log N)$. There is also a constant difference because `numpy.fft.fft()` is a built-in function and is implemented in C. On the contrary, the `dft()` function is implemented in Python and is slower.

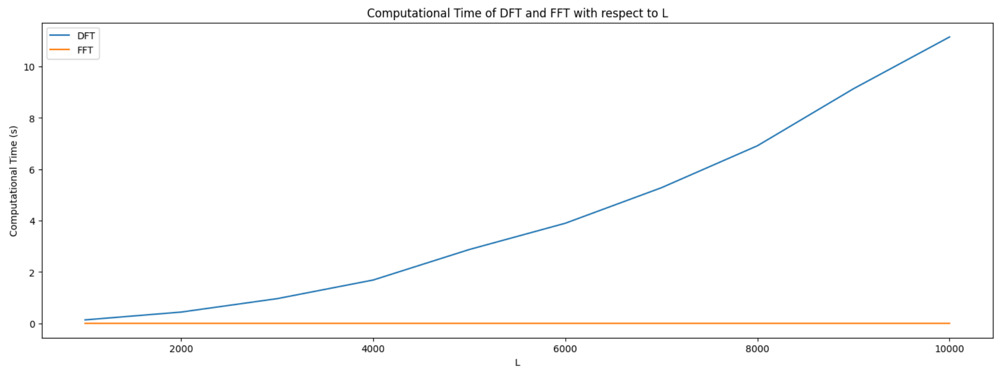


Figure 36: Computational time of *DFT* and *FFT*

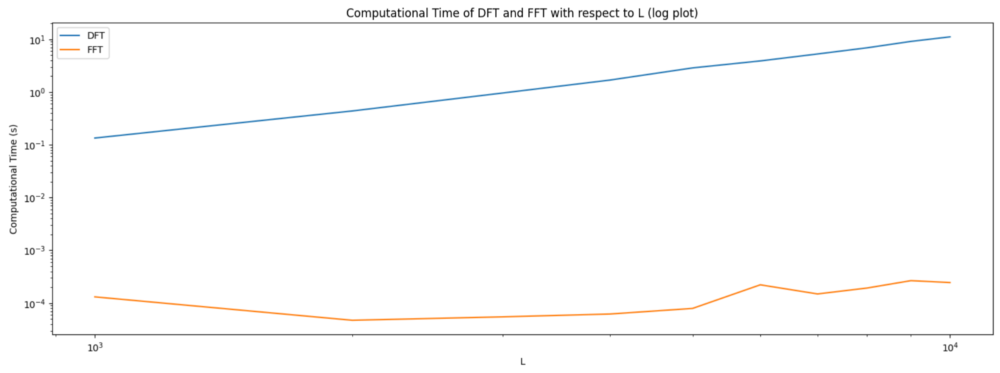


Figure 37: Computational time (log) of *DFT* and *FFT*

For $N = 10000$, numpy's *FFT* function still costs less than $0.001s$, while our *DFT* has cost more than $10s$. The difference is even more significant when N is larger.

7. Appendix Code (Python)

7.a. Signal operations

```python
import numpy as np
import matplotlib.pyplot as plt

# Generate a gate function with the given parameter
def gate_func(A, B):
    def output_func(t):
        return np.where((t >= 0) & (t <= A), B, 0)
    return output_func

# Transform a function. Parameter shifting is given by param_func(), and the value is multiplied by
`times`
def func_transform(func, param_func, times):
    def output_func(x):
        return func(param_func(x)) * times
    return output_func
```

```python
# Returns with a function whose output is the sum of the outputs of f and g
def add_func(f, g):
    def output_func(x):
        return f(x) + g(x)
    return output_func


A = 3
B = 4
D = 8

g0 = gate_func(A, B)
g1 = func_transform(g0, lambda t: 3 * t + D, 1)
g2 = func_transform(g0, lambda t: t - D, 2)

x_func = add_func(add_func(g0, g1), g2)

x_values = np.linspace(-5, 12, 1000)
y_values = x_func(x_values)
plt.plot(x_values, y_values, label=f'x(t)')
plt.xlabel('t')
plt.ylabel('x(t)')
plt.legend()
plt.show()
```

7.b. Continuous-Time Fourier Transform properties

7.b.1. Code for 3.a and 3.b

```python
import numpy as np
import matplotlib.pyplot as plt

def gen_g(d, h):
    def g(t):
        return np.where((t >= -d / 2) & (t <= d / 2), h, 0)
    return g

D = 8
H = 5
SAMPLE_N = 5000

g = gen_g(D, H)

def CTFT(x, t, w):
    """
    x[i] and t[i] is the i-th sample of the signal and time,
    for each w[i], calculate the CTFT of x(t) at w[i]
    """
    Xw = np.zeros_like(w, dtype=complex)
    dt = t[1] - t[0]
    for i, wi in enumerate(w):
        # Two iterators here, x and t
        Xw[i] = np.sum(x * np.exp(-1j * wi * t) * dt)
    return Xw

def func_transform(ori_func, param_func, times):
    def output_func(t):
        return ori_func(param_func(t)) * times
    return output_func

g2 = func_transform(g, lambda t: t - D / 2, 1)

t_values = np.linspace(-5, 9, SAMPLE_N)
g_values = g(t_values)
g2_values = g2(t_values)
fig = plt.figure(figsize=(18, 6))
plt.plot(t_values, g_values, 'r-', label=f'g(t)')
plt.plot(t_values, g2_values, 'b--', label=f'g2(t)')
plt.xlabel('t')
plt.legend()
fig.show()
```

7.b.2. Code for 3.c

```
maxw = 10 * np.pi
w_values = np.linspace(-maxw, maxw, SAMPLE_N)
Gw = CTFT(g_values, t_values, w_values)
Gw2 = CTFT(g2_values, t_values, w_values)
def get_mod_pha_real_imag(c):
    return np.abs(c), np.angle(c), c.real, c.imag
g_4plots = get_mod_pha_real_imag(Gw)
g2_4plots = get_mod_pha_real_imag(Gw2)
names = ['Modules', 'Phase', 'Real', 'Imaginary']
for i in range(4):
    print(f'Gw {names[i]}')
    fig = plt.figure(figsize=(18, 6))
    plt.plot(w_values, g_4plots[i], 'r-', label=f'Gw {names[i]}')
    plt.plot(w_values, g2_4plots[i], 'b--', label=f'Gw2 {names[i]}')
    plt.xlabel('w')
    plt.legend() # 图例...
    fig.show()
```

7.b.3. Code for 3.d

```
def y_func(t):
    return g(t) * np.cos(4 * np.pi * t)

t_values = np.linspace(-15.233, 15.666, SAMPLE_N)
y_values = y_func(t_values)
g_values = g(t_values)
fig = plt.figure(figsize=(18, 6))
plt.plot(t_values, g_values, 'r-', label=f'g(t)')
plt.plot(t_values, y_values, 'b--', label=f'y(t)')
plt.xlabel('t')
plt.legend() # 图例...
fig.show()
```

7.b.4. Code for 3.e

```
ctft_of_g = CTFT(g_values, t_values, w_values)
ctft_of_y = CTFT(y_values, t_values, w_values)
g_4plots = get_mod_pha_real_imag(ctft_of_g)
y_4plots = get_mod_pha_real_imag(ctft_of_y)

for prop in range(2):
    fig = plt.figure(figsize=(18, 6))
    plt.plot(w_values, g_4plots[prop], 'r-', label=f'g CTFT {names[prop]}')
    print(w_values.shape, g_4plots[prop].shape)
    plt.plot(w_values, y_4plots[prop], 'b--', label=f'y CTFT {names[prop]}')
    plt.xlabel('w')
    plt.legend() # 图例...
    fig.show()
```

7.b.5. Code for 3.f

```
def calculate_energy(ys, xs):
    dx = xs[1] - xs[0]
    return sum(ys * ys.conjugate() * dx)

print(calculate_energy(y_values, t_values))
print(calculate_energy(ctft_of_y, w_values) / 2 / np.pi)
```

7.c. Discrete-Time Fourier Transform properties

7.c.1. Code for 4.a and 4.b

```
import numpy as np
import matplotlib.pyplot as plt

def gen_g(d, h):
    def g(t):
        return np.where((t >= -d / 2) & (t <= d / 2), h, 0)
    return g

D = 8
H = 2
NUM_W = 5000
CTFT_NUM_T = 5000
g = gen_g(D, H)

def DTFT(nT, xn, w):
    Xw = np.zeros(len(w), dtype=complex)
    for i, wi in enumerate(w):
        # Only at t = nT[i], there is xn[i] * delta
        Xw[i] = np.sum(xn * np.exp(-1j * wi * nT))
    return Xw

def discret_samples(f, s, t, time_interval):
    t_values = np.arange(s, t, time_interval)
    return t_values, f(t_values)

def dtft_of_func_nyquist(f, s, t, time_interval):
    # The period (Nyquist interval) is ws (the sampling frequency)
    # s and t in the time domain
    sampling_angular_frequency = 2 * np.pi / time_interval
    w_vec = np.linspace(-sampling_angular_frequency / 2, +sampling_angular_frequency / 2, NUM_W)
    t_values, f_values = discret_samples(f, s, t, time_interval)
    return w_vec, DTFT(t_values, f_values, w_vec)

def get_mod_pha_real_imag(c):
    return np.abs(c), np.angle(c), c.real, c.imag

prop_desc = ['Modules', 'Phase']
x_axis_desc = ['f', 'f / fs', 'w / fs']

def compress_x_axis(opt, w_vec, omega_sampling):
    if opt == 0: # [w] -> [f]
        return w_vec / (2 * np.pi)
    f_sampling = omega_sampling / (2 * np.pi)
    if opt == 1: # [f / fs]
        return w_vec / (2 * np.pi) / f_sampling
    if opt == 2: # [w / fs]
        return w_vec / f_sampling
"""
f, f/f2, w/ws
    module, phase
        g1, g2
"""
SAMPLING_T1 = D / 80
SAMPLING_T2 = D / 40
w_vec_d1, dtft_d1 = dtft_of_func_nyquist(g, -D, D, SAMPLING_T1)
w_vec_d2, dtft_d2 = dtft_of_func_nyquist(g, -D, D, SAMPLING_T2)
plots_d1 = get_mod_pha_real_imag(dtft_d1)
plots_d2 = get_mod_pha_real_imag(dtft_d2)
for opt in range(3):
    for part in range(2):
        fig = plt.figure(figsize=(18, 6))
        x_vec1 = compress_x_axis(opt, w_vec_d1, 2 * np.pi / SAMPLING_T1)
        x_vec2 = compress_x_axis(opt, w_vec_d2, 2 * np.pi / SAMPLING_T2)
        plt.plot(x_vec1, plots_d1[part], 'r-', label=f'(D / 80) G1 {prop_desc[part]}')
        plt.plot(x_vec2, plots_d2[part], 'b--', label=f'(D / 40) G2 {prop_desc[part]}')
        plt.xlabel(x_axis_desc[opt])
        plt.legend()
        fig.show()
```

7.c.2. Code for 4.c

```
    def CTFT(x, t, w):
        """
        x[i] and t[i] is the i-th sample of the signal and time,
        for each w[i], calculate the CTFT of x(t) at w[i]
        """
        Xw = np.zeros_like(w, dtype=complex)
        dt = t[1] - t[0]
        for i, wi in enumerate(w):
```

```
        # Two iterators here, x and t
        Xw[i] = np.sum(x * np.exp(-1j * wi * t) * dt)
    return Xw

def ctft_of_func(f, s, t, w_max):
    # The period (Nyquist interval) is ws (the sampling frequency)
    # s and t in the time domain
    w_vec = np.linspace(-w_max, w_max, NUM_W)
    t_values = np.linspace(s, t, CTFT_NUM_T)
    f_values = f(t_values)
    return w_vec, CTFT(f_values, t_values, w_vec)

def dtft_of_func(f, s, t, time_interval, w_max):
    # The period (Nyquist interval) is ws (the sampling frequency)
    # s and t in the time domain
    w_vec = np.linspace(-w_max, +w_max, NUM_W)
    t_values = np.arange(s, t, time_interval)
    f_values = f(t_values)
    return w_vec, DTFT(t_values, f_values, w_vec)

w_s1 = 2 * np.pi / SAMPLING_T1
w_s2 = 2 * np.pi / SAMPLING_T2
g1_ctft_w_vec, g1_ctft = ctft_of_func(g, -D, D, 3 * w_s1)
g1_dtft_w_vec, g1_dtft = dtft_of_func(g, -D, D, SAMPLING_T1, 3 * w_s1)
g2_ctft_w_vec, g2_ctft = ctft_of_func(g, -D, D, 3 * w_s2)
g2_dtft_w_vec, g2_dtft = dtft_of_func(g, -D, D, SAMPLING_T2, 3 * w_s2)
g1_ctft_plots = get_mod_pha_real_imag(g1_ctft)
g1_dtft_plots = get_mod_pha_real_imag(g1_dtft)
g2_ctft_plots = get_mod_pha_real_imag(g2_ctft)
g2_dtft_plots = get_mod_pha_real_imag(g2_dtft)

# ct g vs dt g1
for i in range(2):
    fig = plt.figure(figsize=(18, 6))
    x_vec1 = compress_x_axis(1, g1_ctft_w_vec, w_s1)
    x_vec2 = compress_x_axis(1, g1_dtft_w_vec, w_s1)
    plt.plot(x_vec1, g1_ctft_plots[i], 'r-', label=f'G1 CTFT {prop_desc[i]}')
    plt.plot(x_vec2, g1_dtft_plots[i], 'b--', label=f'G1 DTFT {prop_desc[i]}')
    plt.xlabel('f / fs')
    plt.legend()
    fig.show()

# ct g vs dt g2
for i in range(2):
    fig = plt.figure(figsize=(18, 6))
    x_vec1 = compress_x_axis(1, g2_ctft_w_vec, w_s2)
    x_vec2 = compress_x_axis(1, g2_dtft_w_vec, w_s2)
    plt.plot(x_vec1, g2_ctft_plots[i], 'r-', label=f'G2 CTFT {prop_desc[i]}')
    plt.plot(x_vec2, g2_dtft_plots[i], 'b--', label=f'G2 DTFT {prop_desc[i]}')
    plt.xlabel('f / fs')
    plt.legend()
    fig.show()
```

7.c.3. Code for 4.d

```
def inverse_dtft(maxn, t_sample, dtft_w_vec, dtft_x_vec):
    # w_vec and x_vec should be in one Nyquist interval, from -ws / 2 to +ws / 2
    ns = np.arange(-maxn, maxn + 1)
    ts = ns * t_sample
    xs = np.zeros_like(ts, dtype=complex)
    dw = dtft_w_vec[1] - dtft_w_vec[0]
    w_sample = 2 * np.pi / t_sample
    for i in range(len(ts)):
        nT = ts[i]
        xs[i] = sum(dtft_x_vec * np.exp(1j * nT * dtft_w_vec) * dw) / w_sample
    return ts, xs

g1_t, g1_values = discret_samples(g, -D, D, SAMPLING_T1)
g1_dtft_w_vec, g1_dtft = dtft_of_func(g, -D, D, SAMPLING_T1, w_s1/ 2)
g1_idtft_t, g1_idtft = inverse_dtft(80, SAMPLING_T1, g1_dtft_w_vec, g1_dtft)
g1_idtft_plots = get_mod_pha_real_imag(g1_idtft) # complex

fig = plt.figure(figsize=(18, 6))
plt.vlines(g1_t, ymin = 0, ymax=g1_values, colors='r', linestyles='dashed', label='G1')
plt.vlines(g1_idtft_t, ymin = 0, ymax=g1_idtft_plots[0], colors='b', linestyles='dotted', label='G1
IDTFT')
plt.legend()
fig.show()

g2_t, g2_values = discret_samples(g, -D, D, SAMPLING_T2)
g2_dtft_w_vec, g2_dtft = dtft_of_func(g, -D, D, SAMPLING_T2, w_s2 / 2)
g2_idtft_t, g2_idtft = inverse_dtft(40, SAMPLING_T2, g2_dtft_w_vec, g2_dtft)
g2_idtft_plots = get_mod_pha_real_imag(g2_idtft) # complex

fig = plt.figure(figsize=(18, 6))
plt.vlines(g2_t, ymin = 0, ymax=g2_values, colors='r', linestyles='dashed', label='G2')
plt.vlines(g2_idtft_t, ymin = 0, ymax=g2_idtft_plots[0], colors='b', linestyles='dotted', label='G2
IDTFT')
plt.legend()
fig.show()
```

7.c.4. Code for 4.e

```
def calculate_energy(ys, xs):
    dx = xs[1] - xs[0]
    return sum(ys * ys.conjugate() * dx)

t_values, g_values = discret_samples(g, -D, D, SAMPLING_T1)
g_energy =  calculate_energy(g_values, t_values)
w_values, dtft_of_g = dtft_of_func(g, -D, D, SAMPLING_T1, w_s1 / 2)
g1_dtft_energy = calculate_energy(dtft_of_g, w_values) / 2 / np.pi
print(g_energy, g1_dtft_energy)
```

7.d. Windowing effects of DTFT

7.d.1. Code for 5.a

```
import numpy as np
import matplotlib.pyplot as plt

SAMPLING_T = 0.01
F_S = 100
F1 = 16
A1 = 1.4
DELTA_F = 1
F2 = F1 + DELTA_F
A2 = 0.6

NUM_W = 5000
W_S = 2 * np.pi * F_S

def func_x(t):
    # t = n * SAMPLING_T
    return A1 * np.sin(2 * np.pi * F1 * t) + A2 * np.sin(2 * np.pi * F2 * t)

def DTFT(nT, xn, w):
    Xw = np.zeros(len(w), dtype=complex)
    for i, wi in enumerate(w):
        # Only at t = nT[i], there is xn[i] * delta
        Xw[i] = np.sum(xn * np.exp(-1j * wi * nT))
    return Xw

def dtft_single_point(f, w, length):
    w_vec = np.array([w])
    ns = np.arange(length)
    ts = ns * SAMPLING_T
    fs = f(ts)
    return DTFT(ts, fs, w_vec)[0]

def dtft_of_func_half_nyquist(f, length):
    # The period (Nyquist interval) is ws (the sampling frequency)
    # s and t in the time domain
    sampling_angular_frequency = W_S
    w_vec = np.linspace(0, +sampling_angular_frequency / 2, NUM_W)
    ns = np.arange(length)
    ts = ns * SAMPLING_T
    fs = f(ts)
    return w_vec, DTFT(ts, fs, w_vec)

def compress_x_axis(opt, w_vec, omega_sampling):
    if opt == 0: # [w] -> [f]
        return w_vec / (2 * np.pi)
    f_sampling = omega_sampling / (2 * np.pi)
    if opt == 1: # [f / fs]
        return w_vec / (2 * np.pi) / f_sampling
    if opt == 2: # [w / fs]
        return w_vec / f_sampling

ls = [50, 200, 1000]
draw_fs = [
    [16.05, 18.5],
    [15.93, 17.08],
    [16.00, 17],
]
for i, length in enumerate(ls):
    w_vec, dtft = dtft_of_func_half_nyquist(func_x, length)
    fs = compress_x_axis(0, w_vec, W_S)
    fig = plt.figure(figsize=(18, 6))
    print(f'--- N={length}')
    for j in range(2):
        f1 = draw_fs[i][j]
        w1 = f1 * 2 * np.pi
        y1 = np.abs(dtft_single_point(func_x, w1, length)) * 2 / length
        print(f'A{j + 1} = {y1:.2f}, f{j + 1} = {f1:.2f}')
        plt.plot(f1, y1, 'ro')  # 'ro'表示红色圆点
        plt.text(f1, y1, f'({f1:.2f}, {y1:.2f})', ha='right', va='bottom')  # 标注坐标
    plt.plot(fs, np.abs(dtft) * 2 / length, label=f'L={length}')
    plt.grid(True)
    plt.legend()
    fig.show()
```

7.d.2. Code for 5.b

```
A0 = 0.53836
def hamming(n, N):
    return A0 - (1 - A0) * np.cos(2 * np.pi * n / (N - 1))

def dtft_of_func_half_nyquist_hamming(f, length):
    # The period (Nyquist interval) is ws (the sampling frequency)
    # s and t in the time domain
    sampling_angular_frequency = W_S
    w_vec = np.linspace(0, +sampling_angular_frequency / 2, NUM_W)
    ns = np.arange(length)
    ts = ns * SAMPLING_T
    fs = f(ts)
    for i in range(length):
        fs[i] *= hamming(i, length)
    return w_vec, DTFT(ts, fs, w_vec)

def dtft_single_point_hamming(f, w, length):
    w_vec = np.array([w])
    ns = np.arange(length)
    ts = ns * SAMPLING_T
    fs = f(ts)
    for i in range(length):
        fs[i] *= hamming(i, length)
    return DTFT(ts, fs, w_vec)[0]

ls = [50, 200, 1000]
draw_fs = [
    [16.08, 18.5],
    [15.97, 16.8],
    [16.00, 17],
]
for i, length in enumerate(ls):
    w_vec, dtft = dtft_of_func_half_nyquist_hamming(func_x, length)
    fs = compress_x_axis(0, w_vec, W_S)
    fig = plt.figure(figsize=(18, 6))
    print(f'--- N={length}')
    for j in range(2):
        f1 = draw_fs[i][j]
        w1 = f1 * 2 * np.pi
        y1 = np.abs(dtft_single_point_hamming(func_x, w1, length)) * 2 / length / A0
        print(f'A{j + 1} = {y1:.2f}, f{j + 1} = {f1:.2f}')
        plt.plot(f1, y1, 'ro')  # 'ro'表示红色圆点
        plt.text(f1, y1, f'({f1:.2f}, {y1:.2f})', ha='right', va='bottom')  # 标注坐标
    plt.plot(fs, np.abs(dtft) * 2 / length / A0, label=f'L={length}')
    plt.grid(True)
    plt.legend()
    fig.show()
```

7.e. DFT and FFT

7.e.1. Code for 6.a

```
import numpy as np
import matplotlib.pyplot as plt
L = 10
y = np.array([-1, 2, 3, 0, -2, 1, 4, -3, 0, -2])
ns = np.arange(L)

fig = plt.figure(figsize=(18, 6))
plt.vlines(ns, ymin = 0, ymax=y, colors='r', linestyles='solid', label='G1')
plt.axhline(y=0, color='k')
plt.legend()
fig.show()
```

7.e.2. Code for 6.b

```
NUM_W = 5000

def DTFT(nT, xn, w):
    Xw = np.zeros(len(w), dtype=complex)
    for i, wi in enumerate(w):
        # Only at t = nT[i], there is xn[i] * delta
        Xw[i] = np.sum(xn * np.exp(-1j * wi * nT))
    return Xw

def dtft_of_func_nyquist(x_values, y_values, time_interval):
    # The period (Nyquist interval) is ws (the sampling frequency)
    # s and t in the time domain
    sampling_angular_frequency = 2 * np.pi / time_interval
    w_vec = np.linspace(-sampling_angular_frequency / 2, +sampling_angular_frequency / 2, NUM_W)
    return w_vec, DTFT(x_values, y_values, w_vec)

def dtft_of_func_positive_nyquist(x_values, y_values, time_interval):
    # The period (Nyquist interval) is ws (the sampling frequency)
    # s and t in the time domain
    sampling_angular_frequency = 2 * np.pi / time_interval
    w_vec = np.linspace(0, +sampling_angular_frequency, NUM_W)
    return w_vec, DTFT(x_values, y_values, w_vec)

def get_mod_pha_real_imag(c):
    return np.abs(c), np.angle(c), c.real, c.imag

w_vec, dtft = dtft_of_func_nyquist(ns, y, 1)
dtft_plots = get_mod_pha_real_imag(dtft)
prop_desc = ['Modules', 'Phase']

def plot_mod_phase(x_vec, y_vec, x_name, y_name):
    plots = get_mod_pha_real_imag(y_vec)
    for part in range(2):
        fig = plt.figure(figsize=(18, 6))
        plt.plot(x_vec, plots[part], 'r-', label=f'{y_name} {prop_desc[part]}')
        plt.legend()
        plt.xlabel(x_name)
        fig.show()

plot_mod_phase(w_vec, dtft, 'w', 'Y')
```

7.e.3. Code for 6.c

```python
def dft(ys):
    n = len(ys)
    ns = np.arange(n)
    def omega_k(k):
        return 2 * np.pi * k / n
    w_vec = np.array([omega_k(k) for k in range(n)])
    dft_vec = np.array([sum(ys * np.exp(-1j * w * ns)) for w in w_vec])
    return w_vec, dft_vec

dft_w_vec, dft_vec = dft(y)
dtft_w_vec, dtft_vec = dtft_of_func_positive_nyquist(ns, y, 1)

for part in range(2):
    fig = plt.figure(figsize=(18, 6))
    dft_plots = get_mod_pha_real_imag(dft_vec)
    dtft_plots = get_mod_pha_real_imag(dtft_vec)
    plt.plot(dft_w_vec, dft_plots[part], 'b-', label=f'DFT {prop_desc[part]}')
    plt.plot(dtft_w_vec, dtft_plots[part], 'r-', label=f'DTFT {prop_desc[part]}')
    plt.legend()
    plt.xlabel('w')
    fig.show()
```

7.e.4. Code for 6.d

```python
def inverse_dtft(maxn, t_sample, dtft_w_vec, dtft_x_vec):
    # w_vec and x_vec should be in one Nyquist interval, from -ws / 2 to +ws / 2
    ns = np.arange(-maxn, maxn + 1)
    ts = ns * t_sample
    xs = np.zeros_like(ts, dtype=complex)
    dw = dtft_w_vec[1] - dtft_w_vec[0]
    w_sample = 2 * np.pi / t_sample
    for i in range(len(ts)):
        nT = ts[i]
        xs[i] = sum(dtft_x_vec * np.exp(1j * nT * dtft_w_vec) * dw) / w_sample
    return ts, xs

def inverse_dft(dft_vec):
    n = len(dft_vec)
    ns = np.arange(n)
    w_vec = np.array([2 * np.pi * k / n for k in range(n)])
    y_vec = np.array([sum(dft_vec * np.exp(1j * w * ns)) / n for w in w_vec])
    return ns, y_vec

# plot y and its inverse DFT in one figure
fig = plt.figure(figsize=(18, 6))
_, idft_y = inverse_dft(dft_vec)
plt.vlines(ns, ymin = 0, ymax=idft_y, colors='b', linestyles='dashed', label='Inverse DFT')
plt.vlines(ns, ymin = 0, ymax=y, colors='r', linestyles='dotted', label='Original y')
plt.legend()
plt.axhline(y=0, color='k')
fig.show()
```

7.e.5. Code for 6.e

```python
pad_x = np.arange(16)
pad_w = np.array([2 * np.pi * k / 16 for k in range(16)])
pad_y = np.pad(y, (0, 16 - len(y)), 'constant', constant_values=(0,))
fft = np.fft.fft(pad_y)

for part in range(2):
    fig = plt.figure(figsize=(18, 6))
    fft_plots = get_mod_pha_real_imag(fft)
    dtft_plots = get_mod_pha_real_imag(dtft_vec)
    plt.plot(pad_w, fft_plots[part], 'b-', label=f'FFT {prop_desc[part]}')
    plt.plot(dtft_w_vec, dtft_plots[part], 'r-', label=f'DTFT {prop_desc[part]}')
    plt.legend()
    plt.xlabel('w')
    fig.show()
```

7.e.6. Code for 6.f - Time statistics

```python
import numpy as np
import time
L_values = np.arange(1000, 10001, 1000)
log_l = np.log10(L_values)
dft_times = []
fft_times = []
for L in L_values:
    y_padded = np.pad(y, (0, L - len(y)), 'constant', constant_values=(0,))

    # Measure the time for DFT
    start_time = time.time()
    dft(y_padded)
    dft_time = time.time() - start_time
    dft_times.append(dft_time)

    # Measure the time for FFT
    start_time = time.time()
    np.fft.fft(y_padded)
    fft_time = time.time() - start_time
    fft_times.append(fft_time)
print(fft_times, dft_times)
```

7.e.7. Code for 6.f - Plot of time statistics

```python
# Plot the computational time curve
plt.figure(figsize=(18, 6))
plt.plot(L_values, dft_times, label='DFT')
plt.plot(L_values, fft_times, label='FFT')
plt.xlabel('L')
plt.ylabel('Computational Time (s)')
plt.title('Computational Time of DFT and FFT with respect to L')
plt.legend()
plt.show()
plt.figure(figsize=(18, 6))
plt.loglog(L_values, dft_times, label='DFT')
plt.loglog(L_values, fft_times, label='FFT')
plt.xlabel('L')
plt.ylabel('Computational Time (s)')
plt.title('Computational Time of DFT and FFT with respect to L (log plot)')
plt.legend()
plt.show()
```